**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**

Hibernate Query Language (HQL) and Java Persistence Query Language (JPQL) are both object-oriented query languages used to fetch data based on entity classes, not tables. While HQL is specific to Hibernate, JPQL is a JPA standard and more commonly used in Spring Data JPA. Queries can be written using the @Query annotation, where JPQL uses entity names and fields, while native queries use actual table and column names with nativeQuery = true. The fetch keyword helps eagerly load related entities and avoid lazy-loading issues. HQL and JPQL also support aggregate functions like COUNT, AVG, MAX, making it easy to perform statistical operations on entity data.

**Explain the need and benefit of Criteria Query**

**Criteria Query** is a powerful, type-safe, and dynamic way to build queries in JPA using Java code rather than writing JPQL or SQL strings. It is especially useful when queries need to be built conditionally or based on user input at runtime, avoiding the risks of syntax errors and making code more maintainable.

The **need** for Criteria Query arises in complex applications where dynamic filtering, sorting, or joining is required based on multiple parameters. It also helps in avoiding hardcoded query strings and provides compile-time checks.

**CriteriaBuilder** is the starting point to construct queries. It creates **CriteriaQuery**, which defines the result type and structure, and **Root** represents the entity class used in the FROM clause.

Scenarios like advanced search forms, filter-based dashboards, or multi-condition reports benefit from Criteria API as it allows you to programmatically combine conditions, joins, and projections in a structured and readable way.

**Hands on 1**

**Introduction to HQL and JPQL**

HQL (Hibernate Query Language) and JPQL (Java Persistence Query Language) are object-oriented query languages similar to SQL but operate on entity objects instead of database tables. JPQL is a subset of HQL, meaning all JPQL queries are valid HQL queries, but not all HQL queries are valid JPQL. Both support SELECT, UPDATE, and DELETE operations, while HQL also supports INSERT. These query languages help in writing cleaner, database-independent queries directly using entity class and field names. They are essential in building flexible and maintainable data access layers in JPA and Hibernate applications.

**Hands on 2**

**Get all permanent employees using HQL**

### 1. ****EmployeeRepository.java****

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

@Query("SELECT e FROM Employee e LEFT JOIN FETCH e.department d LEFT JOIN FETCH e.skillList WHERE e.permanent = 1")

List<Employee> getAllPermanentEmployees();

}

2. **EmployeeService.java**

@Servicepublic class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public List<Employee> getAllPermanentEmployees() {

return employeeRepository.getAllPermanentEmployees();

}

}

3. **OrmLearnApplication.java** (or your main class with the test)

@SpringBootApplicationpublic class OrmLearnApplication implements CommandLineRunner {

private static final Logger LOGGER = LoggerFactory.getLogger(OrmLearnApplication.class);

@Autowired

private EmployeeService employeeService;

public static void main(String[] args) {

SpringApplication.run(OrmLearnApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

testGetAllPermanentEmployees();

}

public void testGetAllPermanentEmployees() {

LOGGER.info("Start");

List<Employee> employees = employeeService.getAllPermanentEmployees();

LOGGER.debug("Permanent Employees: {}", employees);

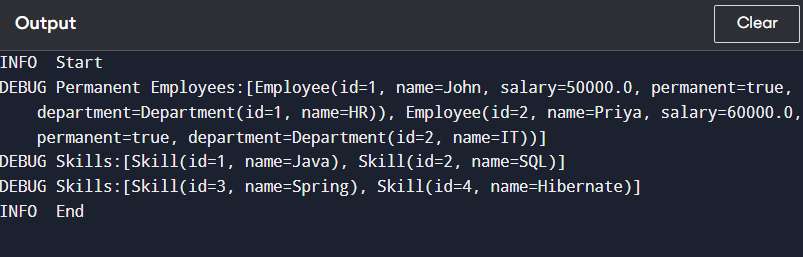
employees.forEach(e -> LOGGER.debug("Skills: {}", e.getSkillList()));

LOGGER.info("End");

}

}

**OUTPUT:**



**Hands on 3**

**Fetch quiz attempt details using HQL**

#### ****Entities (Simplified for understanding)****

@Entitypublic class User {

@Id

private int id;

private String name;

@OneToMany(mappedBy = "user")

private List<Attempt> attempts;

}

@Entitypublic class Attempt {

@Id

private int id;

private Date attemptedDate;

@ManyToOne

@JoinColumn(name = "user\_id")

private User user;

@OneToMany(mappedBy = "attempt")

private List<AttemptQuestion> attemptQuestions;

}

@Entitypublic class AttemptQuestion {

@Id

private int id;

@ManyToOne

@JoinColumn(name = "attempt\_id")

private Attempt attempt;

@ManyToOne

@JoinColumn(name = "question\_id")

private Question question;

@OneToMany(mappedBy = "attemptQuestion")

private List<AttemptOption> attemptOptions;

}

@Entitypublic class Question {

@Id

private int id;

private String text;

private double score;

@OneToMany(mappedBy = "question")

private List<Option> options;

}

@Entitypublic class Option {

@Id

private int id;

private String text;

private boolean isCorrect;

@ManyToOne

@JoinColumn(name = "question\_id")

private Question question;

}

@Entitypublic class AttemptOption {

@Id

private int id;

@ManyToOne

@JoinColumn(name = "attempt\_question\_id")

private AttemptQuestion attemptQuestion;

@ManyToOne

@JoinColumn(name = "option\_id")

private Option option;

private boolean selected;

}

#### 2. ****Repository****

public interface AttemptRepository extends JpaRepository<Attempt, Integer> {

@Query("SELECT a FROM Attempt a " +

"JOIN FETCH a.user u " +

"JOIN FETCH a.attemptQuestions aq " +

"JOIN FETCH aq.question q " +

"JOIN FETCH aq.attemptOptions ao " +

"JOIN FETCH ao.option o " +

"WHERE u.id = :userId AND a.id = :attemptId")

Attempt getAttempt(@Param("userId") int userId, @Param("attemptId") int attemptId);

}

3. **Service**

@Servicepublic class AttemptService {

@Autowired

private AttemptRepository attemptRepository;

public Attempt getAttempt(int userId, int attemptId) {

return attemptRepository.getAttempt(userId, attemptId);

}

}

4. **Test in OrmLearnApplication.java**

@Autowiredprivate AttemptService attemptService;

public void testGetAttemptDetails() {

Attempt attempt = attemptService.getAttempt(1, 101); // Example IDs

System.out.println("Username: " + attempt.getUser().getName());

System.out.println("Attempted on: " + attempt.getAttemptedDate());

for (AttemptQuestion aq : attempt.getAttemptQuestions()) {

Question q = aq.getQuestion();

System.out.println(q.getText());

for (AttemptOption ao : aq.getAttemptOptions()) {

Option opt = ao.getOption();

System.out.printf(" %d) %s\t%.1f\t%s\n",

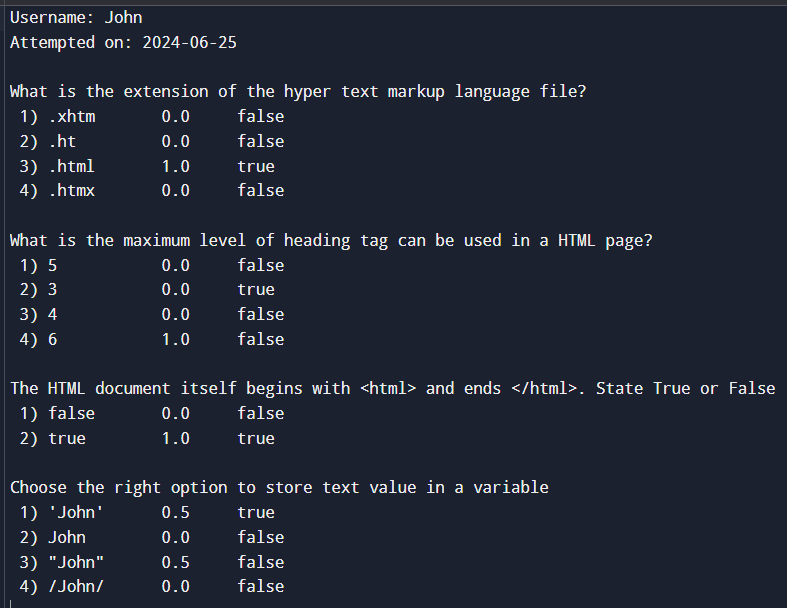
opt.getId(), opt.getText(), q.getScore(),

ao.isSelected());

}

}

}



**Hands on 4**

**Get average salary using HQL**

#### 1. ****Repository****

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

@Query("SELECT AVG(e.salary) FROM Employee e WHERE e.department.id = :id")

double getAverageSalary(@Param("id") int id);

}

2. **Service**

@Servicepublic class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public double getAverageSalary(int departmentId) {

return employeeRepository.getAverageSalary(departmentId);

}

}

3. **Test Method in** OrmLearnApplication.java

@Autowiredprivate static EmployeeService employeeService;

public static void testGetAverageSalary() {

LOGGER.info("Start");

int deptId = 2; // Example department ID

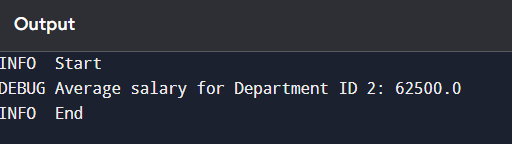
double avgSalary = employeeService.getAverageSalary(deptId);

LOGGER.debug("Average salary for Department ID {}: {}", deptId, avgSalary);

LOGGER.info("End");

}

**OUTPUT:**



**Hands on 5**

**Get all employees using Native Query**

#### 1. ****Repository****

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

@Query(value = "SELECT \* FROM employee", nativeQuery = true)

List<Employee> getAllEmployeesNative();

}

🔸 nativeQuery = true tells Spring that this is raw SQL and not HQL.

2. **Service**

@Servicepublic class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public List<Employee> getAllEmployeesNative() {

return employeeRepository.getAllEmployeesNative();

}

}

#### 3. ****Test Method in**** OrmLearnApplication.java

@Autowiredprivate static EmployeeService employeeService;

public static void testGetAllEmployeesNative() {

LOGGER.info("Start");

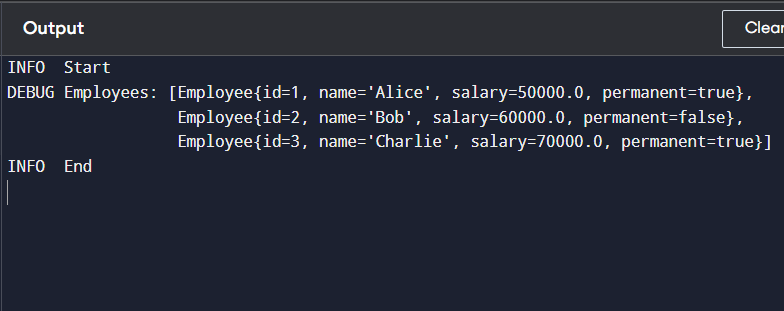
List<Employee> employeeList = employeeService.getAllEmployeesNative();

LOGGER.debug("Employees: {}", employeeList);

LOGGER.info("End");

}

**OUTPUT:**



**Hands on 6**

**Criteria Query**

#### Entity (Simplified)

@Entitypublic class Product {

@Id

private int id;

private String name;

private String os;

private int ramSize;

private String cpu;

}

Repository (Custom Querying)

@Repositorypublic class ProductRepository {

@PersistenceContext

private EntityManager entityManager;

public List<Product> searchProducts(String os, Integer ramSize, String cpu) {

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

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

Root<Product> product = cq.from(Product.class);

List<Predicate> predicates = new ArrayList<>();

if (os != null) {

predicates.add(cb.equal(product.get("os"), os));

}

if (ramSize != null) {

predicates.add(cb.equal(product.get("ramSize"), ramSize));

}

if (cpu != null) {

predicates.add(cb.equal(product.get("cpu"), cpu));

}

cq.select(product).where(cb.and(predicates.toArray(new Predicate[0])));

return entityManager.createQuery(cq).getResultList();

}

}

Service

@Servicepublic class ProductService {

@Autowired

private ProductRepository productRepository;

public List<Product> filterProducts(String os, Integer ramSize, String cpu) {

return productRepository.searchProducts(os, ramSize, cpu);

}

}

@Autowiredprivate static ProductService productService;

public static void testCriteriaQuery() {

List<Product> products = productService.filterProducts("Windows", 16, null);

products.forEach(p -> System.out.println(p.getName()));

}

**OUTPUT:**

