**Spring Data JPA Hands-on Answers**

**1. HQL and Native Query, HQL vs JPQL, @Query Annotation, HQL Fetch, Aggregate Functions, Native Query Attribute**

* **HQL (Hibernate Query Language)** and **JPQL (Java Persistence Query Language)** are object-oriented query languages similar to SQL, but operate on Java entities and attributes.
* **JPQL** is a subset of HQL; all JPQL queries are valid HQL, but not vice versa.
* Both support SELECT, UPDATE, and DELETE. Only HQL supports INSERT.
* **@Query annotation**: Used in Spring Data JPA repositories to define custom queries using HQL/JPQL or native SQL.
  + Example:  
    @Query("SELECT e FROM Employee e WHERE e.permanent = 1")
* **HQL fetch keyword**: Used with JOIN to fetch associated entities in a single query, avoiding the N+1 problem.
  + Example:  
    @Query("SELECT e FROM Employee e LEFT JOIN FETCH e.department d LEFT JOIN FETCH e.skillList WHERE e.permanent = 1")
* **Aggregate functions in HQL**: Use AVG, COUNT, MIN, MAX, SUM.
  + Example:  
    @Query("SELECT AVG(e.salary) FROM Employee e")
* **Native Query**: Direct SQL query executed on the database, bypassing HQL/JPQL.
  + Example:  
    @Query(value="SELECT \* FROM employee", nativeQuery = true)
* **nativeQuery attribute**: Set to true in @Query to indicate a native SQL query.

**2. Need and Benefit of Criteria Query**

* **Criteria Query** allows dynamic, programmatic query construction in Java.
* Useful when query conditions depend on user input or multiple optional filters.
* Avoids hardcoded queries, improving flexibility and maintainability.
* Supports dynamic addition of predicates (conditions) at runtime.
* Example: E-commerce filtering, where users select filters (RAM, CPU, price) and queries are built dynamically.

**3. HQL to Get All Permanent Employees with Department and Skill List**

* Use HQL with fetch joins:

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

* This ensures all required data is fetched in a single query.

**4. Fetch Quiz Attempt Details Using HQL**

* Create entity classes for each table: user, question, options, attempt, attempt\_question, attempt\_option.
* Define O/R mapping based on schema.
* Repository method:

public Attempt getAttempt(int userId, int attemptId);

* Service method:

public Attempt getAttempt(int userId, int attemptId);

* HQL should join: user → attempt → attempt\_question → question → attempt\_option → options.
* Use fetch in HQL for one-to-many or many-to-many relationships.
* Example output format for displaying attempt details:

What is the extension of the hyper text markup language file?  
 1) .xhtm 0.0 false  
 2) .ht 0.0 false  
 3) .html 1.0 true  
 4) .htmx 0.0 false

**5. Get Average Salary Using HQL**

* Repository method:

@Query(value="SELECT AVG(e.salary) FROM Employee e")  
double getAverageSalary();

* To filter by department:

@Query(value="SELECT AVG(e.salary) FROM Employee e WHERE e.department.id = :id")  
double getAverageSalary(@Param("id") int id);

* Use @Param to bind the department id.

**6. Get All Employees Using Native Query**

* Repository method:

@Query(value="SELECT \* FROM employee", nativeQuery = true)  
List<Employee> getAllEmployeesNative();

* Native queries are direct SQL and should be minimized for better portability.

**7. Criteria Query Scenario**

* In scenarios like e-commerce filtering (e.g., Amazon), the WHERE clause must be dynamically built based on selected filters.
* **Criteria Query** enables programmatic addition of filters, making the query flexible and adaptable to user-selected criteria.

⁂