Berner
Fachhochschule

CAS Java Microservice Development

Java Persistence API - Queries

Simon Martinelli, simon.martinelli@bfh.ch, v2025.10

Content

Using Queries

Queries in JPA

- JPA has three options for querying data

1. Java Persistence QL
    - Independant of the underlying database
    - SQL subset
    - Queries are based on the class model, not on the data model
2. SQL
3. Criteria API since JPA 2.0

JPQL - Defining Queries

- Dynamic Queries

```
TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e", Employee.class);
```

- Named Queries

```
// Declaration
@Entity
@NamedQueries { @NamedQuery(name = "Employee.findAll",
                            query = "SELECT e FROM Employee e") }
public class Employee {...}

// Use
TypedQuery<Employee> q = em.createNamedQuery("Employee.findAll", Employee.class);
```

JPQL API

- `Query` and `TypedQuery<T>`
- Query methods
  - `List getResultList()`
    `List<T> getResultList()`
  - `Object getSingleResult()`
    `T getSingleResult()`
  - `int executeUpdate()`
    - Returns the number of affected databases rows

Executing Queries

```java
TypedQuery<Employee> q = em.createQuery("SELECT e FROM Employee e", Employee.class);

List<Employee> emps = q.getResultList();

for (Employee e: emps) {
  ...
}
```

Spring Data JPA

Spring Data JPA: Query Interface Methods

```java
public interface UserRepository extends Repository<User, Long> {

    Optional<User> findByEmailAddressAndLastname(String emailAddress, String lastname);
}
```

results in

```
select u from User u where u.emailAddress = ?1 and u.lastname = ?2
```

Spring Data JPA does a property check and traverses nested properties

Spring Data JPA: Query Return Types

- find* can have many return types
  - `void`, `T`, Primitives, Wrapper types
  - `Iterator<T>`, `Collection<T>`, `List<T>`, `Optional<T>`, `Stream<T>`
  - `Future<T>`, `CompletableFuture<T>`, `ListenableFuture`
  - `Slice`, `Page<T>`
  - `GeoResult<T>`, `GeoResults<T>`, `GeoPage<T>`
  - `Mono<T>`, `Flux<T>`, `Single<T>`, `Maybe<T>`, `Flowable<T>`
- **Repository CRUD methods that return a single instance should use `Optional<T>` to indicate the potential absence of a value**

Spring Data JPA: @Query

- You can define your own queries using `@Query` annotation

```
@Query("select e from Employee e order by e.name")
List<Employee> findAllEmployeesOrderByName();
```

JPQL Quick Start

Introduction

- Simplest Query

```
SELECT e FROM Employee e
```

- Path expressions, Navigation with.

```
SELECT e.name FROM Employee e
SELECT e.department FROM Employee e
```

- Filter results

```
SELECT e FROM Employee e WHERE e.department.name = 'NA42' AND e.address.state in ( 'NY', 'CA')
```

Introduction

- Projection

```
SELECT e.name, e.salary FROM Employee e
```

- Join

```
SELECT p.number FROM Employee e, Phone p WHERE e = p.employee AND e.department.name = 'NA42' AND p.type = 'Cell'
```

- Join with JOIN operator

```
SELECT p.number FROM Employee e JOIN e.phones p WHERE e.department.name = 'NA42' AND p.type = 'Cell'
```

- Aggregate functions

```
  SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary) FROM Department d JOIN d.employees e GROUP BY d HAVING COUNT(e) >= 5
```

Query Parameter

- Positional

```
SELECT e FROM Employee e WHERE e.department = ?1 AND e.salary > ?2
```

```
q.setParameter(1, "NA42");
```

- Named

```
SELECT e FROM Employee e WHERE e.department = :dept AND e.salary > :base
```

```
q.setParameter("dept", "NA42");
```

Spring Data JPA: Parameters

- Parameters can be positional

```
@Query("select u from User u where u.lastname = ?1 or u.firstname = ?2")
User findByLastnameOrFirstname(String lastname, String firstname);
```

- **But you should use named parameters**

```
@Query("select u from User u where u.lastname = :lastname or u.firstname = :firstname")
User findByLastnameOrFirstname(String lastname, String firstname);
```

Path Expressions

- A path expression enables direct navigation from an outer to an inner reference:

```
SELECT e.address FROM Employee e
SELECT e.address.name FROM Employee e
```

- A path expression can end in a collection:

```
SELECT e.projects FROM Employee e
```

- A path expression CANNOT navigate beyond a collection:

```
SELECT e.projects.name FROM Employee e -- Does not work!
```

Process Query Results

- The `List` of `getResultList()` or the object from `getSingleResult()` can contain:
    - Primitive types and `String`, Entity types, `Object[]`, and user types through constructor expression
- If the result is an entity, it will be managed if the query was performed within a transaction
- If a query is executed without a transaction, it is called a read-only query

Projection

- In case of a projection in the select clause a `List` of `Object[]` is returned

```
Query q = em.createQuery("""
    SELECT e.name, e.department.name
    FROM Project p JOIN p.employees e where p.name = 'ZLD'
    """);

List<Object[]> result = q.getResultList();

for (Object[] values : result) {
  System.out.println(values[0] + "," + values[1]);
}
```

Constructor Expression

```
public record EmployeeDTO(String employeeName, String deptName) {
}
```

```
// Important: the fully qualified class name must be used!
TypedQuery<EmployeeDTO> q = em.createQuery("""
    SELECT NEW ch.bfh.jmd.hr.EmployeeDTO(e.name, e.department.name)
    FROM Project p JOIN p.employees e where p.name = 'ZLD'
    """);

List<EmployeeDTO> result = q.getResultList();

for (EmployeeDTO emp : result) {
  System.out.println(emp.employeeName() + "," + emp.deptName());
}
```

Spring Data JPA: @Query with DTO-based Projection

- Using the constructor expression to create DTOs in the Repository

```
@Query("""
      SELECT NEW ch.bfh.jmd.EmployeeDTO(e.name, e.department.name)
      FROM Project p JOIN p.employees e where p.name = 'ZLD'
      """)
List<EmployeeDTO> findAllOnlyWithNameAndDepartmentName();
```

Spring Data JPA: Interface-based Projection

- Using Interfaces

```
public interface DepartmentNames {
  String getName();
}

@Query("select d.name as name from Department d join d.employees e group by d.name")
List<DepartmentNames> findAllDepartmentNames();
```

Paging

- Paging is used to reduce the result size
    - `Query.setFirstResult(int pos)`
    - `Query.setMaxResults(int max)`

Spring Data JPA: Paging and Sorting

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
        extends CrudRepository<T, ID> {

    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);
}
```

```
Page<User> users = repository.findAll(new PageRequest(1, 20, Sort.by("name")));
```

Synchronization

- `Query.setFlushModeType(FlushModeType type)`
  - `FlushModeType.AUTO`
    Pending changes are also included in the result
  - `FlushModeType.COMMIT`
    Only committed data is returned

Bulk Update and Delete

- Bulk Delete

```
Query q = em.createQuery("DELETE from Employee e");
int count = q.executeUpdate();
```

- Bulk Update

```
Query q = em.createQuery("""
  UPDATE Employee e SET e.name = 'Simon' WHERE e.name = 'Peter'""");
int count = q.executeUpdate();
```

- **CAUTION! The entity manager is bypassed!**

Spring Data JPA: Modifying Queries

- You can use Update and Delete, but you have to tell Spring Data that these are modifying queries

```
@Modifying
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);
```

Query Hints

- Directly from the Query

```
q.setHint("toplink.cache-usage", "DoNotCheckCache");
```

- As Named Query

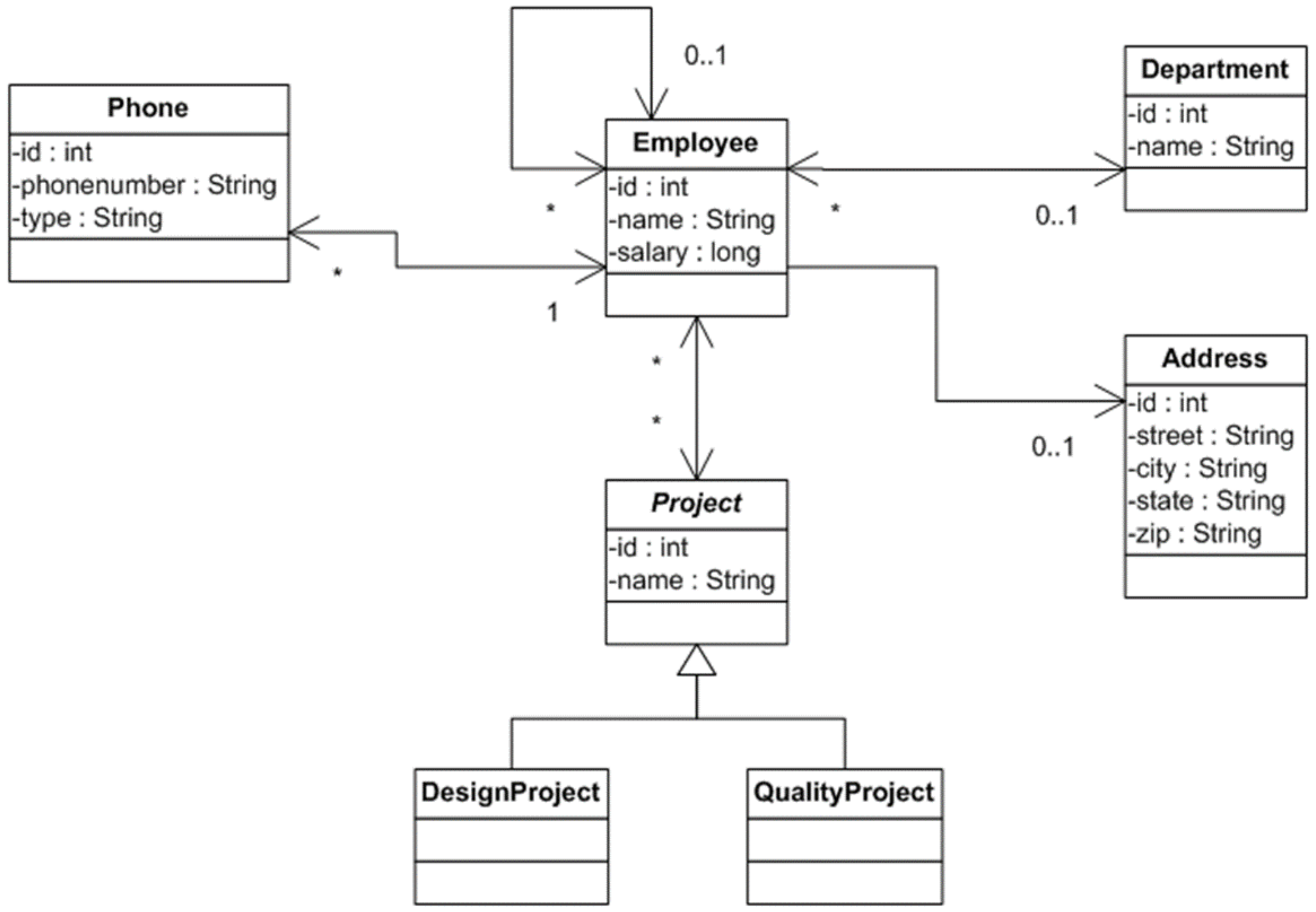
```
@NamedQuery(name = "findAll",
          query = "SELECT e FROM Employee e",
          hints = {@QueryHint(name = "toplink-cache-usage",
                              value = "DoNotCheckCache")})
public class Employee {
  ...
}
```

Recommendations

- Use Parameterized Queries
- Use Projection and the Constructor expression
  If the data is not modified, you don't need entities
- Use Query Hints for optimization
- Use Bulk Update and Delete in an isolated transaction and then clean up the Persistence Context
- Be aware of differences of the used JPA provider
  Check the generated SQLs to get a feel for your JPA implementation

Query Language

Entity Model



SELECT

```
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[ORDER BY <order_by_clause>]
```

Example

```
SELECT e FROM Employee e
WHERE e.name = 'John Doe'
ORDER BY e.salary
```

FROM

- Identification variable

  ```
  FROM Employee e
  ```

- SQL joins are created automatically if
  - two or more range variables are used

    ```
    FROM Employee e, Department d
    ```

  - the JOIN operator is used

    ```
    FROM Employee e JOIN e.department d
    ```

  - a path navigates expression about a relationship

    ```
    SELECT e.department
    ```

JOINS

- INNER JOINS

  JPQL: `SELECT p FROM Employee e JOIN e.phones p`

  SQL: `SELECT p.* FROM emp e, phone p WHERE e.id = p.emp_id`

- OUTER JOINS

  JPQL: `SELECT e, d FROM Employee e LEFT JOIN e.department d`

  SQL: `SELECT * FROM emp e, d dept WHERE e.dept_id = d.id (+)`

- SPECIAL CASE: FETCH JOINS

  `SELECT e FROM Employee e JOIN FETCH e.address`

WHERE

- PARAMETERS
  - Named `:name` or positional `?1`
- OPERATORS PRECEDENCE
  - Navigation Operator .
  - Unary operators +/-
  - Multiplication (*) and division (/), Addition (+) and subtraction (-)
  - Comparison Operators
    =, >, >=, <, <=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
  - Logical operators (AND, OR, NOT)

BETWEEN

```
SELECT e
FROM Employee e
WHERE e.salary BETWEEN 40000 AND 45000
```

==

```
SELECT e
FROM Employee e
WHERE e.salary >= 40000 AND e.salary <= 45000
```

IN AND SUBQUERIES

IN

```
SELECT e
FROM Employee e
WHERE e.address.state IN ('NY', 'CA’)
```

Subquery

```
SELECT e
FROM Employee e
WHERE e.department = (SELECT DISTINCT d FROM d Department
                                        JOIN d.employees de
                                        JOIN de.projects p
                                        WHERE p.name LIKE '% QA')
```

Collections: IS EMPTY

Regarding collections, IS EMPTY is equivalent to IS NULL in fields

```
SELECT e FROM Employee e
WHERE e.directs IS NOT EMPTY
```

==

```
SELECT m FROM Employee m
WHERE (SELECT COUNT (s) from Employee e WHERE e.manager = m) > 0
```

Collections : MEMBER OF

MEMBER OF checks if the object is a member of a collection

```
SELECT e FROM Employee e
WHERE :project MEMBER OF e.projects
```

==

```
SELECT e FROM Employee e
WHERE :project IN (SELECT p FROM e.projects p)
```

EXISTS

```
SELECT e
FROM Employee e
WHERE NOT EXISTS (SELECT p
                  FROM e.phones p
                  WHERE p.type = 'Cell')
```

ANY, ALL, and SOME

```
SELECT e
FROM   Employee e
WHERE e.directs IS NOT EMPTY
AND    e.salary > ALL (SELECT d.salary FROM e.directs d)
```

FUNCTIONS

JPQL provides these standard functions:

```
ABS(number), CONCAT(string1, string2), CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP,
LENGTH(string), LOCATE(string1, string2 [, start]), LOWER (string)
MOD(number1, number2), SIZE(collection), SQRT(number),
SUBSTRING(string, start, end), UPPER(STRING), TRIM
```

To call database function that is not in the list, use:

```
function('calculate', 1, 2)
```

ORDER BY

Single sort field (default sort order is ascending)

```
SELECT e FROM Employee e ORDER BY DESC e.name
```

Multiple sort fields

```
SELECT e FROM Employee e ORDER BY e.name, e.salary DESC
```

The field used in the ORDER BY clause must be contained in SELECT clause!

The following query is NOT allowed:

```
SELECT e.name FROM Employee e ORDER BY e.salary DESC
```

Aggregate Queries

```
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[GROUP BY <group_by_clause>]
[HAVING <conditional_expression>]
[ORDER BY <order_by_clause>]
```

Example

```
SELECT AVG (e.salary)
FROM Employee e
```

Aggregate Functions

- **AVG** Average of the group
- **COUNT** Number of values in the group
- **MAX** Maximum value in the group
- **MIN** Minimum value in the group
- **SUM** Sum of the group

GROUP BY

Defines a grouping for aggregation of the results

```
SELECT d.name, COUNT(e)
FROM Department d JOIN d.employee e
GROUP by d.name
```

If GROUP BY is not defined, the entire query is used as a group

```
SELECT COUNT(e) FROM Department d JOIN d.employee e
```

HAVING

Defines a filter based on the grouping of the results

```
SELECT e, COUNT (p)
FROM Employee e JOIN e.projects p
GROUP BY e
HAVING COUNT (p)> = 2
```

UPDATE

```
UPDATE <entity_name> [[AS] <identification_variable>]
SET <update_statement> {, <update_statement>}*
[WHERE <conditional_expression>]
```

Example

```
UPDATE Employee e
SET e.salary = 60000
WHERE e.salary = 55000
```

DELETE

```
DELETE FROM <entity_name> [[AS] <identification_variable>]
[WHERE <condition>]
```

Example

```
DELETE FROM Employee e
WHERE e.department IS NULL
```

JPA 2 Enhancements

Timestamp

```
SELECT t from Bank Transaction t
WHERE t.txTime > {ts ‘2008–06–01 10:00:01.0’}
```

Non-polymorphic queries

```
SELECT e FROM Employee e
WHERE TYPE (e) = FullTimeEmployee OR e.wage = "SALARY"
```

Collection Parameters in IN Expression

```
SELECT emp FROM Employee emp
WHERE emp.project.id IN :projectIds
```

JPA 2 Enhancements

Ordered List Index

```
SELECT t FROM CreditCard c JOIN c.transactionHistory t
WHERE INDEX (t) BETWEEN 0 AND 9
```

CASE Statement

```
UPDATE Employee SET e e.salary =
CASE
  e.rating WHEN 1 THEN e.salary * 1.1
WHEN 2 THEN
  e.salary * 1.05
ELSE
  e.salary * 1.01
END
```

Exercise: Query Language: Constraints

- Use both JpaRepositories with and without @Query annotation and the EntityManager in the exercises
- As result types use entities, DTOs and interfaces
- Solve at least one task with SQL
- Use named queries

Exercise: Query Language: Queries to Implement

1. Find all employees who live in the canton of Zurich
2. Calculate the average salary of employees per department
3. Find the employee with the lowest salary (Hint: there are many ways to Rome…)
4. Create a query that returns the employee name and the complete address, ordered by the employee’s name
5. Find employees that are not assigned to a project
6. Find all business phone numbers ordered by number
7. Find employees who do not have a business phone number yet

Criteria API

Motivation

- Before JPQL, most O/R-mappers had an object-oriented query language
- With JPA 2.0, these APIs were standardized in the Criteria API
- Unlike JPQL, queries can be implemented using the Criteria API with Java, and a metamodel
- Query and metamodel are checked during compile time

References

- Hibernate Docs
  https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#criteria
- Java EE Tutorial
  https://docs.oracle.com/javaee/7/tutorial/persistence-criteria001.htm#GJRIJ
- Spring Data Specifications
  https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#specifications

Simple Query

JPQL

```
SELECT e FROM Employee e WHERE e.name = 'John Smith’;
```

Criteria API

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);

Root<Employee> employee = cq.from(Employee.class);
cq.select(employee).where(cb.equal(employee.get("name"), "John Smith"));
```

Projection

JPQL

```
SELECT e.id FROM Employee e
```

Criteria API

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object> cq = cb.createQuery(Object.class);
Root<Employee> employee = cq.from(Employee.class);
cq.multiselect(employee.get("id"));
```

Advantages and Disadvantages

Advantages

- Especially composing dynamic queries (e.g. in search forms) with JPQL query as a string is a tedious and error-prone. That's were Criteria API shines

Disadvantage

- Criteria API feels "over designed"
    - This makes getting started harder than with JPQL, and the queries are harder to read
- Criteria API is not sufficient enough alone to provide full type safety

Metamodel API

- Based on the mapping, JPA generates a metamodel
- This metamodel may be used in the Queries and increases type safety
- The name of the metamodel class is derrived from the entity name and the suffix
    - Employee -> Employee_
- The task of generation is done by the Persistence Provider with annotation processing

Generated Metamodel

```java
@Generated(value = "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Employee.class)
public abstract class Employee_ {

   public static volatile SingularAttribute<Employee, Address> address;
   public static volatile SingularAttribute<Employee, Employee> boss;
   public static volatile SetAttribute<Employee, Project> projects;
   public static volatile SingularAttribute<Employee, String> name;
   public static volatile ListAttribute<Employee, Phone> phones;
   public static volatile SingularAttribute<Employee, Integer> id;
   public static volatile SetAttribute<Employee, Employee> directs;
   public static volatile SingularAttribute<Employee, Long> salary;
   public static volatile SingularAttribute<Employee, Department> department;

   public static final String ADDRESS = "address";
   public static final String BOSS = "boss";
   public static final String PROJECTS = "projects";
   public static final String NAME = "name";
   public static final String PHONES = "phones";
   public static final String ID = "id";
   public static final String DIRECTS = "directs";
   public static final String SALARY = "salary";
   public static final String DEPARTMENT = "department";

}
```

Metamodel Usage

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);

Root<Employee> employee = cq.from(Employee.class);
cq.select(employee).where(cb.equal(employee.get(Employee_.name), "John Smith"));
```

Spring Data JPA Specification

- By extending `JpaSpecificationExecutor` we get `find*` methods that take a Specification as a parameter.
- The Specification method toPredicate provides all objects to create a Criteria API Predicate

```
List<Employee> list = employeeRepository.findAll(new Specification() {

    @Override
    public Predicate toPredicate(Root employee, CriteriaQuery query,
                                             CriteriaBuilder criteriaBuilder) {
      return criteriaBuilder.equal(employee.get(Employee_.name), name);
    }
});
```

Exercise: Criteria API

1. Transform some queries to Criteria API
2. Use Spring Data JPA Specification