Queries in JPA

Martin Ledvinka

KBSS

Winter Term 2023

Contents

JPQL

JPQL

- Java Persistence Query Language
- Query language with SQL-like syntax but based on the JPA object model
  - Work with entity classes and attributes instead of tables and columns
- Translated to SQL by the persistence provider
- Selection and data manipulation

SELECT – Projection

- Entity
  - Note that all such selected entities are *managed*

```
SELECT p FROM Product p
```

- Scalar value(s)

```
SELECT DISTINCT u.username FROM User u
```

- Constructor – fully-qualified name

```
SELECT new cz.cvut.kbss.ear.eshop.dto.Name(u.firstName, u.lastName)
    FROM User u
```

- `ResultSetMapping`

```
SELECT u.firstName, u.lastName, u.username FROM User u
```

FROM – Selection

- Entity

```
SELECT p FROM Product p
```

- Inner JOIN

```
SELECT p, c FROM Product p JOIN p.categories c
```

- LEFT (OUTER) JOIN
- Implicit JOIN

```
SELECT o FROM Order o WHERE o.customer.username LIKE :username
```

WHERE - Restriction

- =, <, >, <=, >=, <> (! =)
- `LIKE`
- `BETWEEN`
- `IS (NOT) NULL`
- `IN`
- `EMPTY`
- `MEMBER OF`

Parameter Binding

- Named parameters
  - `:param`
- Positional parameters
  - `?1`
- Use parameters, they protect against injection attacks!

```
em.createQuery("SELECT u FROM User u WHERE u.username = :username",
    User.class)
                .setParameter("username", username)
                .getSingleResult();
```

Criteria API

Criteria API

- Programatic query API for JPA
  - Build query using Java objects instead of strings
- Verbose, not easy to grasp
- Good for cases where query parameters are not known in advance
  - E.g., complex filtering APIs like an e-shop product filtering

Criteria API example

```
SELECT p FROM Product p WHERE p.name LIKE '%p%'
```

Static Metamodel

```
CriteriaBuilder cb =
  em.getCriteriaBuilder();
CriteriaQuery<Product> cq =
  cb.createQuery(Product.class);
Root<Product> r =
  cq.from(Product.class);
cq.where(
  cb.like(
    r.get(Product_.name)
  ,"%p%")
);
return
  em.createQuery(cq).getResultList();
```

Metamodel

```
Metamodel m = em.getMetamodel();
CriteriaBuilder cb =
  em.getCriteriaBuilder();
CriteriaQuery<Product> cq =
  cb.createQuery(Product.class);
Root<Product> r =
  cq.from(Product.class);
cq.where(
  cb.like(
    r.get(
      m.entity(Product.class)
      .getSingularAttribute("name",
      String.class))
  ,"%p%")
);
return
  em.createQuery(cq).getResultList();
```

Metamodel

Static Metamodel

- Generated before compilation (usually using a Maven plugin)
- Classes with static fields representing entities and their attributes
- Compile-time checking

Dynamic Metamodel

- Generated by persistence provider at runtime
- Use names of attributes
- More verbose, typos cause runtime errors

Criteria API - Main classes

1. `CriteriaBuilder` used to construct
   - Criteria queries, e.g., `cb.createQuery(User.class)`
   - Expressions, e.g., "test that a concrete student is member of a studygroup" – `cb.isMember(student, studygroup.get("students"))`
   - Predicates, e.g., `cb.and(expression1, expression2)`
2. `CriteriaQuery` defines top-level structure of a query such as
   - `Root<User> = cq.from(User.class)`
   - `cq.select(..).where(..)`
3. `Root` – root type in the from clause that references an entity. It can be used within
   - Expressions e.g., `dog.get(Dog_.color).in("brown", "black")`
   - Selections e.g., `q.select(dog.get("color"))`
   - ...

Native SQL

Native SQL

- JPQL is a limited subset of SQL, sometimes it is necessary to use a native SQL query
  - Subqueries outside WHERE, set operations (UNION, INTERSECT)
  - Highly optimized SQL queries
  - Complex analytical queries
  - Queries using database-specific features
- `EntityManager.createNativeQuery`
- `@NamedNativeQuery`

Native Query Result Mapping

- Untyped query returns a `List` of `Object`/`Object[]` – depending on how many variables are projected from the query
- Mapping to entity – select all mapped columns

```
SELECT u.id, u.firstName, u.lastName, u.username, u.password, u.role
    FROM User u
```

- `@SqlResultSetMapping`

```
@SqlResultSetMapping(
  name = "UserMapping",
  entities = @EntityResult(
    entityClass = User.class,
    fields = {
      @FieldResult(name = "id", column = "userId"),
      @FieldResult(name = "firstName", column = "firstName"),
      @FieldResult(name = "lastName", column = "lastName"),
      @FieldResult(name = "username", column = "email")}))
```

Conclusions

Conclusions

- Static queries with known parameters → JPQL
- Complex analytical queries, using vendor-specific optimizations → Native SQL
- Queries where parameters are not known beforehand → Criteria API

Demo Time

- Let's see this in practice in e-shop
  - `ProductDao`, `UserDao`, `CriteriaProductDao`

Resources

- https://thorben-janssen.com/jpql/
- https://thorben-janssen.com/jpa-native-queries/
- https://docs.oracle.com/javaee/6/tutorial/doc/gjrij.html