JPA - Criteria API

Spis treści

JPA - Criteria API

JPA pozwala na kilka sposobów na pracę z danymi. Omówiliśmy już **JPQL** razem z **HQL** oraz **native queries**. Teraz przyszła pora na omówienie **Criteria API**. Inna nazwa, z jaką możemy się spotkać to **Criteria Queries**.

Criteria API pozwala na budowanie zapytań przy wykorzystaniu obiektów. Dzięki temu będziemy mogli w prosty sposób te zapytania strukturyzować oraz ich poprawność będzie mogła być sprawdzana na etapie kompilacji. Dzięki Criteria API będziemy mogli budować obiekt zwany **Criteria Query**, na którym będziemy określali różnego rodzaju filtry, czy warunki.

Małe wyjaśnienie do Hibernate Criteria API, które jest `@Deprecated` od Hibernate w wersji `5.X`. Zespół Hibernate tworzył swoje Criteria API, które można rozumieć jak drugie Criteria API, obok standardowego JPA Criteria API. Od wersji Hibernate 5.X, twórcy Hibernate postanowili się skoncentrować na rozwoju funkcjonalności implementujących JPA Criteria API, czyli rozwijają teraz implementację JPA Criteria API. W Hibernate 6.0, Hibernate Criteria API zostało usunięte i powinniśmy się skupić na JPA Criteria API.

Pokręcone, co? W programowaniu dużo jest takich kwestii 😄.

Największą "zaletą" Criteria Queries w stosunku do HQL jest możliwość budowania zapytań, które są zorientowane obiektowo. Możemy dzięki temu pisać zapytania w sposób elastyczny i dynamiczny. Jednocześnie pomaga nam w tym wszystkim IDE, bo przecież pracujemy na obiektach. Z drugiej strony ma to również swoje wady. Bardzo skomplikowane przypadki może być łatwiej zrozumieć przy wykorzystaniu czystego SQL. Dlatego właśnie mamy tyle możliwości, to co wybierzemy powinno być

podyktowane kontekstem i potrzebą.

Prosty przykład

Budowanie zapytań przy wykorzystaniu Criteria API będzie się zaczynało od obiektu `Session`. Spójrz na poniższy przykład:

```
import jakarta.persistence.TypedQuery;
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Root;
import org.hibernate.Session;

// ...

void criteriaExample() {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        session.beginTransaction();
        CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder(); ①
        CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class); ②
        Root<Customer> root = criteriaQuery.from(Customer.class); ③
        criteriaQuery.select(root); ④

        TypedQuery<Customer> query = session.createQuery(criteriaQuery); ⑤
        List<Customer> result = query.getResultList(); ⑥
        result.forEach(customer -> System.out.println("###Customer: " + customer));
        session.getTransaction().commit();
    }
}
```

① Stworzenie instancji `CriteriaBuilder` przez wywołanie `session.getCriteriaBuilder()`. `CriteriaBuilder` może być rozumiane jako *factory*, które tworzy `CriteriaQuery`. Może być również pobrane z *EntityManagera*.

② Stworzenie instancji `CriteriaQuery` przez wywołanie `criteriaBuilder.createQuery()`.

③ Określenie jakiej encji będzie dotyczyło zapytanie. `Root` służy jednocześnie do określenia zakresu zapytania w klauzuli *FROM*.

④ Określenie, że query to ma być *SELECT*.

⑤ Stworzenie instancji `Query`.

⑥ Pobranie wyników.

Jeżeli uruchomisz teraz ten fragment kodu, to zobaczysz, że Hibernate wygenerował poniższe zapytania.

```
Hibernate: select (...) from customer c1_0
Hibernate: select (...) from address a1_0
    left join customer c1_0 on a1_0.address_id=c1_0.address_id where a1_0.address_id=?
Hibernate: select (...) from address a1_0
    left join customer c1_0 on a1_0.address_id=c1_0.address_id where a1_0.address_id=?
###Customer: Customer(id=15, name=Stefan, surname=Nowacki, phone=+48 589 245 114,
email=stefan@zajavka.pl,
```

```
    address=Address(id=15, country=Poland, city=Szczecin, postalCode=70-112, address=Witolda Starkiewicza
3))
###Customer: Customer(id=16, name=Adrian, surname=Paczkomat, phone=+48 894 256 331,
email=adrian@zajavka.pl,
    address=Address(id=16, country=Poland, city=Gdynia, postalCode=81-357, address=3 maja 16))
```

Przedstawiony przypadek pokazuje, że stosowanie `CriteriaQuery` do prostego przypadku może być przekombinowane, ale tak jak wszędzie zaznaczam, każda technika jest dedykowana do określonych sytuacji.

Przykład z parametrami

Przejdźmy do kolejnego przykładu, gdzie będziemy chcieli przy wykorzystaniu `CriteriaQuery` napisać takie zapytanie:

```
SELECT (...) FROM customer WHERE email = 'adrian@zajavka.pl'
```

Analogiczne `CriteriaQuery` wyglądałoby w taki sposób:

```
import jakarta.persistence.TypedQuery;
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Root;
import org.hibernate.Session;

// ...

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
ParameterExpression<String> parameter1 = criteriaBuilder.parameter(String.class);
criteriaQuery.select(root).where(criteriaBuilder.equal(root.get("email"), parameter1)); ①

TypedQuery<Customer> query = session.createQuery(criteriaQuery);
query.setParameter(parameter1, "adrian@zajavka.pl"); ②
List<Customer> result = query.getResultList();
result.forEach(customer -> System.out.println("###Customer: " + customer));
```

① W tym miejscu nastąpiło rozbudowanie klauzuli *WHERE*.

② Tutaj natomiast wykorzystany został parametr `email` zdefiniowany w punkcie 1.

W powyższym przykładzie instancja `ParameterExpression` służyła do zdefiniowana parametru, który następnie miał zostać wykorzystany w zapytaniu. Jak również widać, metody z `CriteriaQuery` wspierają chainowanie, więc możemy w tym miejscu skonstruować zapytanie przy wykorzystaniu kolejnych metod. Następnie musimy tylko uzupełnić zdefiniowane parametry wartościami, co nastąpiło przy wykorzystaniu metody `setParameter()`.

Jeżeli uruchomisz teraz ten fragment kodu, to zobaczysz, że Hibernate wygenerował poniższe zapytania.

```
Hibernate: (...) from customer c1_0 where c1_0.email=?
Hibernate: select (...) from address a1_0
```

```
    left join customer c1_0 on a1_0.address_id=c1_0.address_id where a1_0.address_id=?
###Customer: Customer(id=22, name=Adrian, surname=Paczkomat, phone=+48 894 256 331,
email=adrian@zajavka.pl,
    address=Address(id=22, country=Poland, city=Gdynia, postalCode=81-357, address=3 maja 16))
```

Operatory

Tak samo jak SQL wspiera określone operatory, porównania, równości itp, tak samo jest w przypadku `CriteriaQuery`. Jeżeli spojrzysz w definicję interfejsu `CriteriaBuilder`, znajdziesz tam metody takie jak m.in. (obok metod jest umieszczony opis z dokumentacji):

- `isNotNull();` - *Create a predicate to test whether the expression is not null.*
- `isNull();` - *Create a predicate to test whether the expression is null.*
- `equal();` - *Create a predicate for testing the arguments for equality.*
- `notEqual();` - *Create a predicate for testing the arguments for inequality.*
- `gt();` - *Create a predicate for testing whether the first argument is greater than the second.*
- `ge();` - *Create a predicate for testing whether the first argument is greater than or equal to the second.*
- `lt();` - *Create a predicate for testing whether the first argument is less than the second.*
- `le();` - *Create a predicate for testing whether the first argument is less than or equal to the second.*
- `like();` - *Create a predicate for testing whether the expression satisfies the given pattern.*
- `between();` - *Create a predicate for testing whether the first argument is between the second and third arguments in.*
- `in();` - *Create predicate to test whether given expression is contained in a list of values.*
- `and();` - *Create a conjunction of the given boolean expressions.*
- `or();` - *Create a disjunction of the given boolean expressions.*

I wiele, wiele innych. Także jeżeli nie pamiętasz jakie operatory można stosować w zapytaniach SQL, to tutaj jest pewnego rodzaju ściąga ☺.

Przy wykorzystaniu wspomnianych operatorów, moglibyśmy zatem zapisać przykładowo poniższe zapytania:

```
criteriaQuery.select(root).where(criteriaBuilder.gt(root.get("salary"), 1000));
criteriaQuery.select(root).where(criteriaBuilder.like(root.get("productName"), "%basket%"));
criteriaQuery.select(root).where(criteriaBuilder.between(root.get("itemPrice"), 100, 120));
criteriaQuery.select(root).where(root.get("itemName").in("Ham", "Cheese", "Tuna"));
```

Spróbujmy jeszcze napisać bardziej skomplikowane zapytanie przy wykorzystaniu `CriteriaQuery`:

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
criteriaQuery
    .select(root)
    .where(
```

```
        criteriaBuilder.and(
            criteriaBuilder.like(root.get("email"), "%r%"),
            criteriaBuilder.isNotNull(root.get("phone"))
        ))
    .orderBy(criteriaBuilder.desc(root.get("email")));

TypedQuery<Customer> query = session.createQuery(criteriaQuery);
List<Customer> result = query.getResultList();
result.forEach(customer -> System.out.println("###Customer: " + customer));
```

Hibernate generuje wtedy takie zapytanie:

```
Hibernate: select (...)
    from customer c1_0
    where c1_0.email
        like ? escape ''
        and c1_0.phone is not null
    order by c1_0.email desc
```

Zawężanie wyników

Na etapie omawiania HQL wprowadzone zostały dwie metody `setFirstResult()` oraz `setMaxResults()`. Jeżeli chcielibyśmy ograniczyć zwracane wyniki przy wykorzystaniu `CriteriaQuery`, moglibyśmy to zrobić w podobny sposób:

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
criteriaQuery.select(root);

TypedQuery<Customer> query = session.createQuery(criteriaQuery);
query.setFirstResult(1);
query.setMaxResults(2);
List<Customer> result = query.getResultList();
result.forEach(customer -> System.out.println("###Customer: " + customer));
```

Metody te zadziałają tak samo, jak w omawianych wcześniej przypadkach.

Pobranie pojedynczego rekordu

Poprzednie przykłady pokazywały pobieranie listy rekordów. Pojawia się zatem pytanie, w jaki sposób można pobrać pojedynczy rekord przy wykorzystaniu `CriteriaQuery`. Trzeba natomiast pamiętać, że wykonanie poniższego kodu na danych, które zwrócą więcej niż jeden rekord, skończy się wyjątkiem `NonUniqueResultException`. Jeżeli natomiast spróbujemy pobrać tylko jeden rekord, a baza danych nie zwróci żadnych rekordów, to również zostanie wyrzucony wyjątek: `NoResultException`. Przykład:

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
criteriaQuery.select(root);
```

```
TypedQuery<Customer> query = session.createQuery(criteriaQuery);
Customer customer = query.getSingleResult();
System.out.println("###Customer: " + customer);
```

Distinct

Tak samo jak SQL pozwala na eliminację wszystkich duplikatów, tak samo pozwala na to `CriteriaQuery`. Jednocześnie też wprowadzimy mechanizm, który pozwala na pobieranie pojedynczych pól. W poniższym przykładzie chcemy pobrać unikalne imiona klientów. W tym celu zostanie wykorzystany interfejs `Tuple`. Można to rozumieć jak taki generyczny obiekt, z którego możemy wyciągać wartości na konkretnych pozycjach. Inaczej można na to spojrzeć jak na tablicę `Object[]`, do której odwołujemy się na podstawie indeksu.

```
import jakarta.persistence.Tuple;

// ...

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Tuple> criteriaQuery = criteriaBuilder.createQuery(Tuple.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
criteriaQuery.multiselect(root.get("name")); ①
criteriaQuery.distinct(true);

TypedQuery<Tuple> query = session.createQuery(criteriaQuery);
List<Tuple> result = query.getResultList();
result.forEach(tuple -> System.out.println("###Tuple: " + tuple.get(0)));
```

① `mutliselect()` pozwala na wybranie wartości, które mają być pobrane. Można to rozumieć jak: *SELECT val1, val2, val3 ....*

Hibernate wykona wtedy zapytanie:

```
select distinct c1_0.name from customer c1_0
```

Sortowanie

Sortowanie wyników wygląda analogicznie jak w przypadku HQL. Mamy w tym celu przygotowaną dedykowaną metodę:

```
import jakarta.persistence.criteria.Order;

// ...

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);

criteriaQuery.select(root);
criteriaQuery.orderBy(
    criteriaBuilder.asc(root.get("name")),
    criteriaBuilder.desc(root.get("phone"))
```

```
);

TypedQuery<Customer> query = session.createQuery(criteriaQuery);
List<Customer> result = query.getResultList();
result.forEach(customer -> System.out.println("###Customer: " + customer));
```

Na podstawie tak skonstruowanego zapytania, Hibernate wykona poniższy SQL:

```
select c1_0.customer_id,c1_0.address_id,c1_0.email,c1_0.name,c1_0.phone,c1_0.surname
from customer c1_0
order by c1_0.name asc,c1_0.phone desc
```

JOIN

W tym zestawieniu przykładów nie mogło oczywiście zabraknąć informacji o tym jak przy wykorzystaniu `CriteriaQuery` można zbudować zapytanie z JOIN. Przejdźmy do konkretów:

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class);
root.join("address");

TypedQuery<Customer> query = session.createQuery(criteriaQuery);
List<Customer> result = query.getResultList();
result.forEach(customer -> System.out.println("###Customer: " + customer));
```

Hibernate wygeneruje wtedy takie zapytanie:

```
select c1_0.customer_id,c1_0.address_id,c1_0.email,c1_0.name,c1_0.phone,c1_0.surname
from customer c1_0
left join address a1_0 on a1_0.address_id=c1_0.address_id
```

Jeżeli interesuje nas konkretny typ JOIN, możemy wtedy zapisać wywołanie metody `join()` w taki sposób:

```
root.join("address", JoinType.LEFT);
```

Funkcje agregujące

`CriteriaQuery` pozwala na wykonywanie funkcji agregujących, spójrzmy na poniższy przykład:

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Tuple> criteriaQuery = criteriaBuilder.createQuery(Tuple.class);
Root<Customer> root = criteriaQuery.from(Customer.class);

criteriaQuery.multiselect(
    criteriaBuilder.count(root.get("name")),
    criteriaBuilder.max(root.get("surname")));
```

```
TypedQuery<Tuple> query = session.createQuery(criteriaQuery);
Tuple result = query.getSingleResult();
System.out.printf("COUNT: %s, MAX: %s%n", result.get(0), result.get(1));
```

Na ekranie zostanie wtedy wydrukowane:

```
Hibernate: select count(c1_0.name),max(c1_0.surname) from customer c1_0
COUNT: 2, MAX: Paczkomat
```

Czyli przy wykorzystaniu `CriteriaQuery` możemy wykonywać funkcje agregujące.

Criteria Update

Przedstawiane przykłady Criteria API dotyczyły odczytywania danych. Przy wykorzystaniu Criteria API może również dane modyfikować i usuwać. Spójrz na poniższy przykład:

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaUpdate<Customer> criteriaUpdate = criteriaBuilder.createCriteriaUpdate(Customer.class);
Root<Customer> root = criteriaUpdate.from(Customer.class);
criteriaUpdate.set("name", "Tomasz");
criteriaUpdate.where(criteriaBuilder.equal(root.get("email"), "adrian@zajavka.pl"));
session.createMutationQuery(criteriaUpdate).executeUpdate();
```

Tym razem zamiast interfejsu `CriteriaQuery` skorzystaliśmy z interfejsu `CriteriaUpdate`. Po uruchomieniu powyższego przykładu, Hibernate wykonuje następujące zapytanie i *Adrian* zmienia imię na *Tomasz*:

```
update customer set name=? where email=?
```

Criteria Delete

Jak możesz się domyślić, w przypadku, gdy będziemy rozmawiać o usuwaniu danych, zamiast `CriteriaUpdate` będziemy stosowali `CriteriaDelete`. Poniżej przykład:

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaDelete<Customer> criteriaDelete = criteriaBuilder.createCriteriaDelete(Customer.class);
Root<Customer> root = criteriaDelete.from(Customer.class);
criteriaDelete.where(criteriaBuilder.equal(root.get("email"), "adrian@zajavka.pl"));

session.createMutationQuery(criteriaDelete).executeUpdate();
```

Po wykonaniu powyższego kodu, Hibernate wykonuje poniższe zapytanie:

```
delete from customer where email=?
```

Należy tutaj zwrócić uwagę na pewną istotną kwestię. Encja `Customer` ma cały czas ustawione *CascadeType.ALL* i pomimo tego Hibernate nie usunął wierszy z tabeli `address`. Przykład ten można dać ku przestrodze, że narzędzia są fajną rzeczą, która ma ułatwić pracę, ale to programista finalnie ma się upewnić, czy narzędzia działają zgodnie z jego oczekiwaniami. Czasami łatwiej jest nad czymś zapanować, gdy jest to robione ręcznie i nie opieramy się o magię, którą dają nam narzędzia.

Podsumowanie

Podsumowując, przedstawiliśmy kolejny sposób na definiowanie zapytań przy wykorzystaniu JPA. Criteria JPA jest preferowanym sposobem, gdy potrzebujemy budować zapytania dynamicznie, w zależności od logiki w danym przypadku. Są one weryfikowane na etapie kompilacji, gdyż budujemy je przy wykorzystaniu metod. Nie wykluczają one stosowania JPQL, czy native queries, możemy wykorzystywać w projekcie całą trójkę jednocześnie. Są uzupełnieniem do poprzedników, który ma Ci pomóc zarządzać zapytaniami w określonych przypadkach. Na koniec jeszcze możesz zapoznać się z tym linkiem. Ciekawym źródłem, gdzie można poczytać o możliwościach `CriteriaQuery` jest również ta strona.