Hibernate Querying

Spis treści

Hibernate i zapytania

Naturalne jest, że w pracy z bazami danych będziemy musieli przeprowadzać na tych danych pewne operacje. Domyślasz się już zapewne, że będą to operacje odczytu, zapisu, aktualizacji i usuwania, czyli **CRUD**. Dotychczas poznaliśmy już bardzo podstawowe sposoby manipulacji danymi. Potrafisz już na tym etapie w bardzo podstawowy sposób zapisać dane, odczytać dane, zaktualizować dane oraz je usunąć. Co natomiast można zrobić podczas pracy z JPA, żeby móc wykonywać bardziej złożone zapytania? JPA pozwala nam na pracę z zapytaniami na 4 sposoby:

- **JPQL** - specyfikacja JPA definiuje język zapytań, który jest mieszanką SQL i obiektów. Język ten nazywa się **J**akarta **P**ersistence **Q**uery **L**anguage,
- **Criteria API** - specyfikacja JPA daje możliwość wykorzystania API zwanego Criteria API, które w pewien sposób odzwierciedla składnię zapytań JPQL. Jest to API pozwalające na definiowanie zapytań poprzez metody,
- **Native query** - native queries są rzeczywistymi zapytaniami SQL i wykorzystują wariant SQL, który jest dedykowany do bazy danych, na której pracujemy. Czyli są to "podstawowe" SQL, które możemy wykonywać na bazie danych,
- **Stored procedure** - procedura składowana jest pewnego rodzaju programem, który może zostać napisany na bazie danych. Procedura może realizować logikę, która inaczej mogłaby być zaimplementowana po stronie aplikacji.

Ze wspomnianych sposobów, skupimy się na pierwszych trzech. Nie będziemy wchodzić w tematykę procedur składowanych.

Warto tutaj zaznaczyć, że wspomniane sposoby nie są wzajemnie wykluczające. To developer podejmuje decyzję, w jaki sposób korzysta z dostarczonych mu/jej narzędzi. Dlatego warto jest mieć w swoim arsenale możliwości różne alternatywy. Dzięki temu masz świadomość, że jeden problem można rozwiązać na wiele sposobów.

JPQL oraz HQL

JPQL

Jeżeli zaczniesz czytać o JPQL, to znajdziesz również jego rozwinięcie znane jako *Java Persistence Query Language*. Wynika to z tego, że w pewnym momencie zmianie uległa konwencja nazewnictwa (nie chcę tu wchodzić w powody i szczegóły) i tak samo jak JPA kiedyś było znane jako Java Persistence API, a teraz mówi się o tym Jakarta Persistence API, tak samo JPQL jest teraz rozwijane jako *Jakarta Persistence Query Language*. Cytując Wikipedię:

> (...) a platform-independent object-oriented query language defined as part of the Jakarta Persistence (JPA; formerly Java Persistence API) specification (...)

Powtórzmy jeszcze raz to samo, JPQL jest językiem podobnym do SQL, ale będziemy tutaj operowali na obiektach.

HQL

W praktyce bardzo często spotkasz się też ze skrótem **HQL** (***H**ibernate **Q**uery **L**anguage*). HQL jest podzbiorem JPQL. Zatem poprawne zapytanie JPQL jest również poprawnym zapytaniem HQL, natomiast nie wszystkie poprawne zapytania HQL będą poprawnymi zapytaniami JPQL. Znowu wracamy tutaj do tej zabawy, która była wyjaśniana poprzednio. Możemy korzystać z JPA, które jest implementowane przez Hibernate, albo bezpośrednio z Hibernate. Jeżeli korzystamy z JPA, to łatwiej będzie nam wymienić providera / implementację. Jeżeli korzystamy bezpośrednio z Hibernate, to wymiana taka może być ciężka.

HQL jest wersją **SQL** dostarczaną przez Hibernate, która łączy w sobie koncepcję relacyjnej bazy danych z programowaniem obiektowym. Dzięki HQL zapytania są niezależne od używanej bazy danych. Oznacza to, że zmiana bazy nie jest dla Hibernate straszna. HQL zamiast operować na tabelach i kolumnach, operuje na obiektach i ich właściwościach. Wadą HQL może być to, że w kodzie dalej znajduje się coś podobnego do SQL, ale może być to mylące dla ludzi niezaznajomionych z tymi konstrukcjami.

Główne różnice między HQL a SQL:

- HQL ma własną składnię i gramatykę, nadal zapisuje się go w postaci Stringa, np: *from Customer cust*,
- HQL nie służy do modyfikacji schematu bazy danych, tylko do wykonywania zapytań CRUD,
- Zapytania HQL są tłumaczone przez Hibernate do konwencjonalnych SQL,
- Hibernate pozwala na wykorzystanie SQL bezpośrednio,
- SQL manipuluje danymi w bazie danych pracując w oparciu o table i kolumny. HQL robi to samo, ale pracuje używając klas i ich własności, które są następnie mapowane do struktury tabel w bazie danych.

Wiedząc już czym jest JPQL, HQL, jaka jest różnica między nimi i czym różnią się od SQL, możemy przejść do przykładów. W poniższych przykładach będziemy korzystać z modelu `Owner` oraz `Pet`, które były pokazywane w poprzednich przykładach. Możemy również uznać, że wykorzystywany język w zapytaniach to będzie HQL. Piszę o tym z tego powodu, że odnośniki będą do dokumentacji Hibernate, a nie JPA.

HQL - INSERT

Zanim przejdziemy do krótkiego omówienia **INSERT**, chciałbym zacytować fragment dokumentacji:

> The first sort of insert statement is not as useful. It's usually better to just use persist(). On the other hand, you might consider using it to set up test data.

Zatem, jeżeli chodzi o zapisywanie danych przez Hibernate, rekomendowanym sposobem jest metoda `persist()`.

Do tego mamy tutaj bardzo ciekawą sytuację. Dokumentacja wspomina, że w celach testowych można wykonać polecenie *INSERT*, nawet jeżeli zalecają stosowanie `persist()`. W przykładzie z dokumentacji wykorzystana jest metoda `createQuery()`, tyle, że jest ona wywołana na interfejsie `EntityManager`. Ta sama metoda w interfejsie `Session` jest oznaczona jako `@Deprecated`. Przejdźmy do przykładu:

Metoda insert() w klasie OwnerRepository

```
int insert() {
    EntityManager entityManager = null;
    EntityTransaction transaction = null;
    int result;
    try {
        entityManager = HibernateUtil.getEntityManager();
        if (Objects.isNull(entityManager)) {
            throw new RuntimeException("EntityManager is null");
        }
        transaction = entityManager.getTransaction();
        transaction.begin();
        Query query = entityManager.createQuery("""
            INSERT Owner (name, surname, phone, email) ①
            VALUES ('Romek', 'Zabawniacha', '+48 658 745 322', 'romek@zajavka.pl')
        """);
        result = query.executeUpdate();
        transaction.commit();
        entityManager.close();
    } catch (RuntimeException exception) {
        if (Objects.nonNull(transaction) && transaction.isActive()) {
            transaction.rollback();
        }
        throw exception;
    } finally {
        if (Objects.nonNull(entityManager)) {
            entityManager.close();
        }
    }
    return result;
}
```

① Nie ma tutaj słówka *into* jak w *insert into*. Zamiast nazwy tabeli, używamy nazwy encji. Intellij w wersji Ultimate pozwala kliknięcie na nazwę klasy w HQL i przejście do jej definicji, my natomiast korzystamy z Intellij Community.

Wynik działania tej metody zwróci 1, czyli ilość zmodyfikowanych rekordów. Z racji, że dodajemy jeden rekord, to wynik wynosi 1.

HQL - UPDATE

Możemy przejść do wykonania **UPDATE**. Tutaj umieszczam link do dokumentacji. Najlepiej jest wykorzystać query, które wykorzysta parametry na podstawie ich nazwy, czyli **named params**. Przykład:

Metoda update() w klasie OwnerRepository

```
int update(final String oldEmail, final String newPhone, final String newEmail) {
    EntityManager entityManager = null;
    EntityTransaction transaction = null;
```

```
    int result;
    try {
        entityManager = HibernateUtil.getEntityManager();
        if (Objects.isNull(entityManager)) {
            throw new RuntimeException("EntityManager is null");
        }
        transaction = entityManager.getTransaction();
        transaction.begin();
        Query query = entityManager.createQuery("""
            UPDATE Owner ow ①
            SET ow.phone = :newPhone, ow.email = :newEmail
            WHERE ow.email = :oldEmail
            """);
        query
            .setParameter("oldEmail", oldEmail)
            .setParameter("newEmail", newEmail)
            .setParameter("newPhone", newPhone);
        result = query.executeUpdate();
        transaction.commit();
        entityManager.close();
    } catch (RuntimeException exception) {
        if (Objects.nonNull(transaction) && transaction.isActive()) {
            transaction.rollback();
        }
        throw exception;
    } finally {
        if (Objects.nonNull(entityManager)) {
            entityManager.close();
        }
    }
    return result;
}
```

① Ponownie używamy nazwy encji, a nie tabeli.

Wynik działania tej metody zwróci 1, czyli ilość zmodyfikowanych rekordów. No chyba, że mamy w bazie kilku użytkowników z tym samym mailem (sic!), to wtedy ta wartość będzie większa niż 1. Jeżeli żaden użytkownik nie zostanie zaktualizowany to ta wartość wyniesie 0.

HQL - DELETE

Możemy przejść do wykonania **DELETE**. Dokumentacja jest umieszczona pod linkiem. Tak jak w poprzednim przypadku, najlepiej jest wykorzystać query z **named params**. Przykład:

Metoda delete() w klasie OwnerRepository

```
int delete(final String email) {
    EntityManager entityManager = null;
    EntityTransaction transaction = null;
    int result;
    try {
        entityManager = HibernateUtil.getEntityManager();
        if (Objects.isNull(entityManager)) {
            throw new RuntimeException("EntityManager is null");
        }
        transaction = entityManager.getTransaction();
        transaction.begin();
        Query query = entityManager.createQuery("""
```

```
            DELETE FROM Owner ow
            WHERE ow.email = :email
            """);
        query.setParameter("email", email);
        result = query.executeUpdate();
        transaction.commit();
        entityManager.close();
    } catch (RuntimeException exception) {
        if (Objects.nonNull(transaction) && transaction.isActive()) {
            transaction.rollback();
        }
        throw exception;
    } finally {
        if (Objects.nonNull(entityManager)) {
            entityManager.close();
        }
    }
    return result;
}
```

Wynik działania tej metody zwróci 1, czyli ilość zmodyfikowanych rekordów. W tym przypadku modyfikacją jest usunięcie, czyli 1 oznacza ilość usuniętych rekordów z bazy danych.

HQL - SELECT

Z całego CRUD, odczytywanie danych zostawiliśmy na koniec, bo tutaj jest najwięcej ciekawostek. Przechodzimy do omówienia **SELECT**, a tutaj jest dokumentacja. Zacznijmy od najprostszego wariantu, jaki można wykonać:

Metoda findAll() w klasie OwnerRepository

```
List<Owner> findAll() {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        session.beginTransaction();
        List<Owner> owners = session.createQuery("FROM Owner", Owner.class).getResultList();
        session.getTransaction().commit();
        return owners;
    }
}
```

Tak jak wspomniałem, jest to najkrótszy możliwy zapis HQL. JPQL różni się w tym momencie od JPQL w ten sposób, że JPQL w tym przypadku wymagałoby klauzuli *SELECT*. HQL jej nie wymaga, ale w HQL również można napisać *SELECT*. To czy w danym przypadku będziesz pisać *SELECT*, czy też nie to już jest kwestia konwencji przyjętej w zespole. Poniżej możesz zapoznać się z kilkoma przykładowymi selectami, zacznij natomiast od przygotowania poniższej metody:

Metoda selectExamples() w klasie OwnerRepository

```
void selectExamples() {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
```

```
        }
        session.beginTransaction();
        // Kod podany w poniższych przykładach należy umieścić tutaj
        session.getTransaction().commit();
    }
}
```

Mając już przygotowaną taką metodę, zasil bazę danych jakimiś danymi i możemy zacząć wykonywać poniższe przykłady.

Simple SELECT

Klasa Owner

```
@OneToMany(fetch = FetchType.EAGER, mappedBy = "owner", cascade = CascadeType.ALL)
private Set<Pet> pets;
```

Fragment metody selectExamples() w klasie OwnerRepository

```
String select1 = "SELECT ow FROM Owner ow";
session.createQuery(select1, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Po uruchomieniu kodu, na ekranie zostanie wydrukowane (dane są przykładowe):

```
Hibernate: select (...) from owner o1_0
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=46, name=Robert, surname=Nowacki, phone=+48 589 245 114, email=robert@zajavka.pl,
    pets=[Pet(id=63, name=Fafik, breed=DOG), Pet(id=62, name=Kiciek, breed=CAT)])
###Entity: Owner(id=47, name=Adrian, surname=Paczkomat, phone=+48 894 256 331, email=adrian@zajavka.pl,
    pets=[Pet(id=65, name=Szymek, breed=MONKEY), Pet(id=64, name=Gucio, breed=DOG)])
###Entity: Owner(id=48, name=Romek, surname=Zabawniacha, phone=+48 658 745 322, email=romek@zajavka.pl,
    pets=[])
###Entity: Owner(id=49, name=Stefan, surname=Zajavkowicz, phone=+48 845 114 894, email=stefan@zajavka.pl,
    pets=[])
```

Nawet jeżeli w zapytaniu nie określiliśmy, że chcemy dociągnąć zwierzątka, ale są one określone jako *FetchType.EAGER*, to Hibernate pod spodem wykonuje dodatkowe 4 zapytania, w stosunku do tego czego oczekujemy. Na ekranie mamy wydrukowanych dwóch właścicieli ze zwierzątkami, oraz dwóch właścicieli, którzy tych zwierzątek nie mają. Co się stanie, gdy przestawimy `fetch` na *FetchType.LAZY*? Na ekranie będzie wtedy drukowane:

```
Hibernate: select (...) from owner o1_0
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=46, name=Robert, surname=Nowacki, phone=+48 589 245 114, email=robert@zajavka.pl,
    pets=[Pet(id=62, name=Kiciek, breed=CAT), Pet(id=63, name=Fafik, breed=DOG)])
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=47, name=Adrian, surname=Paczkomat, phone=+48 894 256 331, email=adrian@zajavka.pl,
```

```
    pets=[Pet(id=64, name=Gucio, breed=DOG), Pet(id=65, name=Szymek, breed=MONKEY)])
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=48, name=Romek, surname=Zabawniacha, phone=+48 658 745 322, email=romek@zajavka.pl,
    pets=[])
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=49, name=Stefan, surname=Zajavkowicz, phone=+48 845 114 894, email=stefan@zajavka.pl,
    pets=[])
```

Hibernate nadal wykonuje dodatkowe zapytania, tyle, że robi to w innym momencie. Zapytania takie są wykonywane dopiero gdy występuje takie zapotrzebowanie. A zapotrzebowanie to wynika z implementacji metody `toString()`. Jeżeli zmienimy implementację metody `Owner#toString()` pozbywając się wywołania `getPets()`, to dodatkowe zapytania do tabeli `pet` nie zostaną wykonane.

Mam nadzieję, że po zapoznaniu się z powyższym przykładem masz już czucie jak ważne jest ustawienie parametru `fetch` zgodnie z oczekiwaniami. Ma on znaczenie nie tylko, gdy pobieramy dane przy wykorzystaniu metody `find()`, ale również gdy wykonujemy zapytania HQL. Ustawienie *FetchType.EAGER* może spowodować nadmierne obciążenie naszej bazy danych zapytaniami i w konsekwencji spadek wydajności aplikacji. Z drugiej strony bardzo skomplikowane zapytania również obciążają bazę danych, dlatego niezależnie od tego, czy stosujemy metodę `find()`, czy HQL, powinniśmy rozsądnie dobierać wartość tego parametru i sprawdzić co w naszym przypadku ma więcej sensu.

Możesz również zwrócić uwagę na używany **ALIAS**. Możemy go również zdefiniować w ten sposób:

```
SELECT ow FROM Owner as ow
```

Natomiast słówko **as** jest opcjonalne. Może wystąpić również taka sytuacja, że przy nazwie klasy trzeba będzie podać pełną nazwę paczki.

Pobieranie poszczególnych pól

Fragment metody selectExamples() w klasie OwnerRepository

```
String select2 = "SELECT ow.id, ow.name FROM Owner ow";
session.createQuery(select2, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Po uruchomieniu kodu, na ekranie zostanie wydrukowane:

```
org.hibernate.query.IllegalQueryOperationException:
Query defined multiple selections, return cannot be typed (other that Object[] or Tuple)
```

Taki przypadek jest specyficzny, Hibernate nie potrafi stworzyć wtedy instancji klasy `Owner` i dostajemy powyższy wyjątek. Jeden ze sposobów na poradzenie sobie z taką sytuacją wygląda tak:

Fragment metody selectExamples() w klasie OwnerRepository

```
String select2 = "SELECT ow.id, ow.name FROM Owner ow";
session.createQuery(select2, Object[].class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + Arrays.asList(entity)));
```

Czyli wczytaliśmy dane do tablicy obiektów, a zwrócona lista byłaby typu `List<Object[]>`. Moglibyśmy wtedy próbować pobierać dane na podstawie indeksu w tabeli. Inaczej można do tego podejść w ten sposób:

Rekord OwnerTemp

```
package pl.zajavka;

public record OwnerTemp(Integer id, String name) {}
```

Fragment metody selectExamples() w klasie OwnerRepository

```
String select2 = "SELECT new pl.zajavka.OwnerTemp(ow.id, ow.name) FROM Owner ow";
session.createQuery(select2, OwnerTemp.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

W takim przypadku rekord (klasa) `OwnerTemp`, może być traktowane jak zwykłe POJO, do którego zapisujemy rezultat wywołania zapytania. Następnie w kodzie możemy się do obiektu typu `OwnerTemp` odwoływać. Po wywołaniu powyższego przykładu, na ekranie zostanie wydrukowane:

```
Hibernate: select o1_0.owner_id,o1_0.name from owner o1_0
###Entity: OwnerTemp[id=46, name=Robert]
###Entity: OwnerTemp[id=47, name=Adrian]
###Entity: OwnerTemp[id=48, name=Romek]
###Entity: OwnerTemp[id=49, name=Stefan]
```

Widać, że Hibernate pyta tylko o te dane, które są nam potrzebne i nie pojawiają się dodatkowe zapytania do tabeli `pet`.

DTO

W praktyce spotkasz się ze stwierdzeniem **DTO** (***D**ata **T**ransfer **O**bject*). DTO są zwykłymi obiektami tak samo jak POJO. Są strukturami, które nie zawierają żadnej logiki biznesowej. Zawierają metody dostępowe służące do pobrania danych i służą do transportu danych. W kontekście baz danych, klasy takie mogą być tworzone i wykorzystywane do transportu danych w przypadku, gdy chcemy pobrać dane w pewien charakterystyczny sposób. Taki charakterystyczny sposób można ująć słowem "widok". Jest to pewna reprezentacja danych, które pobieramy z bazy danych, ale ta reprezentacja jest charakterystyczna z jakiegoś powodu (np. zawiera tylko określone pola, albo jakieś wyliczone wartości) i może być wtedy nazywana widokiem.

Wykorzystana wcześniej klasa (rekord) `OwnerTemp` jest przykładem DTO, który przedstawiał nam szczególny widok na dane. Widok ten zawierał tylko klucz główny i imię właściciela.

Określenie DTO jest częściej używane, gdy wystawiamy własne API, żeby móc komunikować się z naszą aplikacją przez Internet, natomiast można też użyć tego stwierdzenia w odniesieniu do konkretnego spojrzenia na dane z bazy danych.

Jak było to pokazane wcześniej, stosując HQL, możemy stworzyć taki DTO przy wykorzystaniu konstruktora w zapytaniu HQL. Byłoby to również potrzebne, gdybyśmy np. zastosowali klauzulę *GROUP BY*. Wtedy musimy stworzyć DTO, który będzie w stanie przetransportować otrzymany wynik.

W przypadku wystawiania swoich danych "na zewnątrz", DTO są przydatne, bo przecież nie ma potrzeby, żebyśmy wystawiali wszystkie swoje dane "na zewnątrz" - zawsze dobrze jest takie dane ograniczać tylko do absolutnie niezbędnych. Natomiast kontekst DTO został tutaj przywołany, bo można powiedzieć, że `OwnerTemp` jest DTO, bo służy do transportu danych w dalsze części aplikacji.

Named params

Poniższy przykład zostanie napisany w oparciu o **named parameters**. Przypomnę, że jest to również sposób na zabezpieczenie przed atakami **SQL Injection**, który był opisywany przy pierwszym kontakcie z JDBC. Mechanizm ten pozwala nam na podstawianie parametrów w zapytaniu na podstawie ich nazwy.

Fragment metody selectExamples() w klasie OwnerRepository

```
String select3_1 = "SELECT ow FROM Owner ow WHERE ow.email = :email";
session.createQuery(select3_1, Owner.class)
    .setParameter("email", "romek@zajavka.pl")
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Po uruchomieniu kodu, na ekranie zostanie wydrukowane (dane są przykładowe):

```
Hibernate: select (...) from owner o1_0 where o1_0.email=?
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=48, name=Romek, surname=Zabawniacha, phone=+48 658 745 322, email=romek@zajavka.pl,
pets=[])
```

W przykładzie, klasa `Owner` ma zdefiniowaną metodę `toString()`, która odwołuje się do metody `getPets()`, dlatego Hibernate ponownie wykonuje zapytanie do tabeli `pet`. Istotą tego przykładu jest pokazanie sposobu zapisu warunku `WHERE`.

W kolejnym podejściu napiszemy `WHERE`, ale z `LIKE`.

Fragment metody selectExamples() w klasie OwnerRepository

```
String select3_2 = "SELECT ow FROM Owner ow WHERE ow.email LIKE :email";
session.createQuery(select3_2, Owner.class)
    .setParameter("email", "romek%")
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Po uruchomieniu kodu, na ekranie zostanie wydrukowane (dane są przykładowe):

```
Hibernate: select (...) from owner o1_0 where o1_0.email like ? escape ''
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=48, name=Romek, surname=Zabawniacha, phone=+48 658 745 322, email=romek@zajavka.pl,
pets=[])
```

W przykładzie, klasa `Owner` ma zdefiniowaną metodę `toString()`, która odwołuje się do metody `getPets()`, dlatego Hibernate ponownie wykonuje zapytanie do tabeli `pet`. Istotą tego przykładu jest pokazanie sposobu zapisu warunku `WHERE` z `LIKE`.

Sortowanie

Tym razem napiszemy query z sortowaniem.

Fragment metody selectExamples() w klasie OwnerRepository

```
String select4 = "SELECT ow FROM Owner ow ORDER BY ow.email ASC, ow.name DESC";
session.createQuery(select4, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Wynikiem powyższego zapytania będą wyniki, które są posortowane najpierw rosnąco po `email`, a później malejąco po `name`. Domyślnym sposobem sortowania jest `ASC`, zatem możemy tego nie dodawać.

Ograniczanie wyników

W tym przykładzie skupimy się na ograniczaniu pobieranych wyników. W praktyce nie pisze się aplikacji, które wyciągają z bazy danych miliony rekordów i wyświetlają je wszystkie na ekranie na raz. Byłoby to zabójcze dla wydajności. Nie wspominając już o zbędnej zajętości pamięci. Często jest tak, że ograniczamy ilość wyświetlanych rekordów i przejdziemy teraz do pokazania sposobów na to jak można to zrobić.

Interfejs `Query` ma dostępne dwie dostępne metody:

- `setFirstResult()` - metoda ta przyjmuje `int`, który odnosi się do pierwszego wiersza w zwracanym `ResultSet`,
- `setMaxResults()` - ta metoda określa maksymalną ilość wierszy, które chcemy zwrócić.

Omawiane metody mogą służyć do zaimplementowania paginacji w naszej aplikacji.

Paginacja polega na stronicowaniu rekordów. Czyli nie wczytujemy wszystkiego na raz, a dzielimy rezultat na strony i przechodzimy po nich jak po kartkach w książce. Z paginacją masz do czynienia na stronach internetowych, gdzie np. szukasz produktów do kupienia i nie są pokazywane wszystkie na raz, tylko np. 10 na stronę. Po przejściu na kolejną stronę wyświetla się kolejne 10 i tak dalej. Ograniczając wyniki przy wykorzystaniu wspomnianych metod, można zaimplementować mechanizm paginacji.

Na początek zacznijmy do takiego przykładu:

Fragment metody selectExamples() w klasie OwnerRepository

```
String select5_1 = "SELECT ow FROM Owner ow ORDER BY ow.email DESC";
session.createQuery(select5_1, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Po uruchomieniu kodu, na ekranie zostanie wydrukowane (dane są przykładowe):

```
###Entity: Owner(id=49, name=Stefan, ...
###Entity: Owner(id=48, name=Romek, ...
###Entity: Owner(id=46, name=Robert, ...
###Entity: Owner(id=47, name=Adrian, ...
```

Po co to sortowanie? Bo chcemy mieć pewność, że elementy będą zawsze zwracane w tej samej kolejności.

Ważne jest tutaj to, żeby zwrócić uwagę, że zapytanie sortuje wyniki po mailu malejąco. Oznacza to, że za każdym razem sekwencja otrzymywanych id, wygląda tak: *49, 48, 46, 47*. Możemy teraz dołożyć metodę `setFirstResult()`:

Fragment metody selectExamples() w klasie OwnerRepository

```
String select5_2 = "SELECT ow FROM Owner ow ORDER BY ow.email DESC";
session.createQuery(select5_2, Owner.class)
    .setFirstResult(0)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Przetestuj poniższe przypadki:

- `setFirstResult(-1)`,
- `setFirstResult(0)`,
- `setFirstResult(1)`,
- `setFirstResult(2)`,
- `setFirstResult(3)`,

Gdy dodajemy wykonanie metody `setFirstResult()`, Hibernate dodaje do zapytania końcówkę *offset ? rows*:

```
select (...) from owner o1_0 order by o1_0.email desc offset ? rows
```

Poniżej te same przypadki z wypisanymi efektami:

- `setFirstResult(-1)` - tutaj dostajemy wyjątek, bo wartość musi być `>= 0`. *java.lang.IllegalArgumentException: first-result value cannot be negative : -1*,
- `setFirstResult(0)` - nadal dostajemy 4 wyniki w tej samej kolejności, czyli zwróciliśmy wyniki

zaczynając od indeksu `0`,

- `setFirstResult(1)` - pominięty został wiersz z *id = 49*, czyli zwróciliśmy wyniki zaczynając od indeksu `1`,
- `setFirstResult(2)` - pominięty został wiersz z *id = 49* oraz *id = 48*, czyli zwróciliśmy wyniki zaczynając od indeksu `2`,
- `setFirstResult(3)` - zwrócony został tylko wiersz z *id = 47*, czyli zwróciliśmy wyniki zaczynając od indeksu `3`,

Przejdźmy teraz do metody `setMaxResults()`.

Fragment metody selectExamples() w klasie OwnerRepository

```
String select5_3 = "SELECT ow FROM Owner ow ORDER BY ow.email DESC";
session.createQuery(select5_3, Owner.class)
    .setMaxResults(0)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Gdy dodajemy wykonanie metody `setMaxResults()`, Hibernate dodaje do zapytania końcówkę *fetch first ? rows only*:

```
select (...) from owner o1_0 order by o1_0.email desc fetch first ? rows only
```

Przetestuj poniższe przypadki:

- `setMaxResults(-1)`,
- `setMaxResults(0)`,
- `setMaxResults(1)`,
- `setMaxResults(2)`,
- `setMaxResults(3)`,

Poniżej te same przypadki z wypisanymi efektami:

- `setMaxResults(-1)` - tutaj dostajemy wyjątek, bo wartość musi być `>= 0`. *java.lang.IllegalArgumentException: max-results cannot be negative*,
- `setMaxResults(0)` - w tym przypadku nie zostały zwrócone żadne dane, co więcej Hibernate nawet nie wykonał zapytania na bazie danych - taki sprytny ☺,
- `setMaxResults(1)` - zwrócony został tylko właściciel z *id = 49*, czyli pierwszy z całej listy,
- `setMaxResults(2)` - zwrócony został właściciel z *id = 49* oraz *id = 48*, czyli dwóch pierwszych z całej listy - ograniczyliśmy wynik do 2 rekordów,
- `setMaxResults(3)` - zwróconych zostało pierwszych trzech właścicieli - ograniczyliśmy wynik do 3 rekordów,

Wspomniane metody można ze sobą łączyć:

Fragment metody selectExamples() w klasie OwnerRepository

```
String select5_4 = "SELECT ow FROM Owner ow ORDER BY ow.email DESC";
session.createQuery(select5_4, Owner.class)
    .setFirstResult(1)
    .setMaxResults(1)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

W powyższym przypadku zostanie zwrócony tylko właściciel z *id = 48*. Zaczęliśmy od *indeksu = 1*, czyli pomijamy klienta z *id = 49* i ograniczamy rezultaty do jednego, czyli zwracamy tylko klienta z *id = 48*.

Pamiętać należy, że jeżeli chodzi o zwracanie rezultatów z SQL, to każda baza danych może wymagać innej konstrukcji zapytania, w celu ograniczenia wyników. Tym jak takie zapytanie ma wyglądać, zajmuje się za nas Hibernate.

Unikalny rezultat

W tym przykładzie przejdziemy do zwrócenia unikalnego rezultatu z zapytania. W tym celu wykorzystywana jest metoda `uniqueResult()` lub `uniqueResultOptional()`. Jeżeli natomiast z zapytania SQL zostanie zwrócony więcej niż jeden rezultat, zostanie wyrzucony wyjątek `NonUniqueResultException`. Różnica między tymi metodami jest taka, że w przypadku braku znalezionej wartości, metoda `uniqueResult()` zwróci `null`, a `uniqueResultOptional()` zwróci `Optional.empty()`. Przykład:

Fragment metody selectExamples() w klasie OwnerRepository

```
String select6 = "SELECT ow FROM Owner ow WHERE ow.name = :name";
Optional<Owner> result = session.createQuery(select6, Owner.class)
    .setParameter("name", "Romek")
    .uniqueResultOptional();
System.out.println(result);
```

HQL Associations

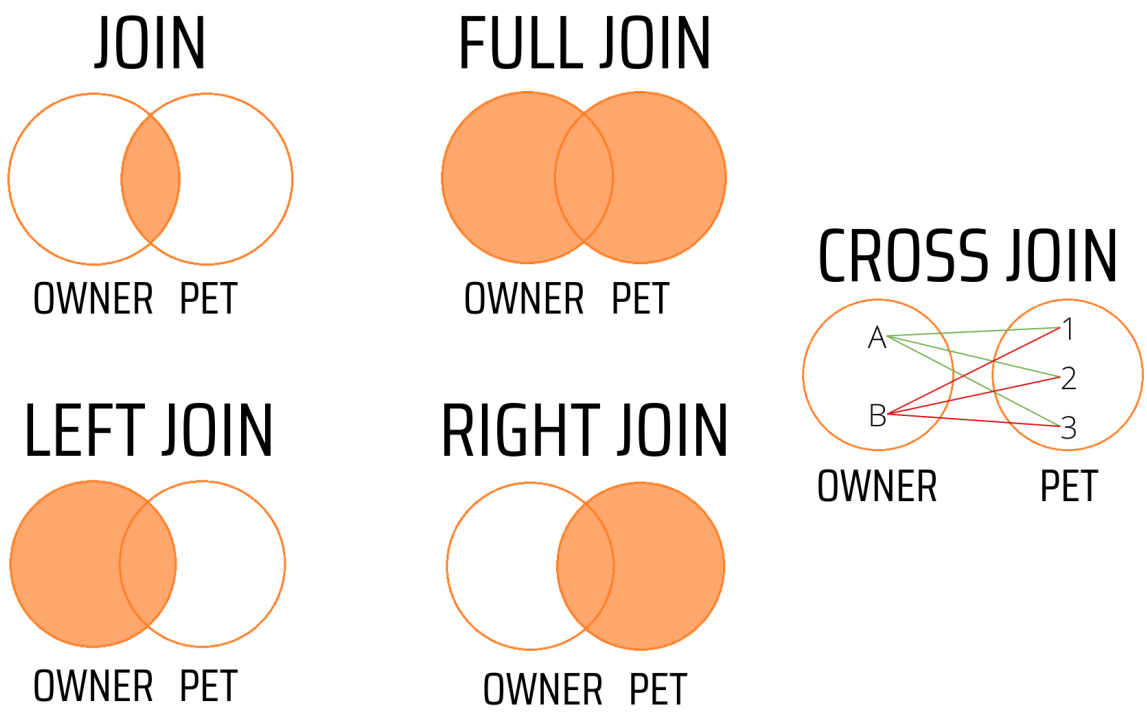
Czyli JOINy! HQL pozwala również na pisanie JOINów i do tego teraz przejdziemy. HQL wspiera poniższe typy JOINów:

- `INNER JOIN` - (domyślnie samo `JOIN` oznacza `INNER JOIN`) - szukamy przecięcia 2 zbiorów, czyli wyświetlimy tylko te rekordy z tabeli `owner`, dla których znajdziemy dopasowanie w tabeli `pet` i jednocześnie wyświetlimy tylko takie rekordy z tabeli `pet`, dla których znajdziemy dopasowanie w tabeli `owner`. Stąd analogia do przecięcia zbiorów,
- `LEFT OUTER JOIN` - (dłuższa nazwa na `LEFT JOIN`) - zwrócimy wszystkie rekordy z tabeli `owner`, nawet te, dla których nie znaleźliśmy dopasowania w tabeli `pet` i jednocześnie zwrócimy tylko te rekordy z tabeli `pet`, dla których znaleźliśmy dopasowanie w tabeli `owner`,
- `RIGHT OUTER JOIN` - (dłuższa nazwa na `RIGHT JOIN`) - zwrócimy wszystkie rekordy z tabeli `pet`, nawet te, dla których nie znaleźliśmy dopasowania w tabeli `owner` i jednocześnie zwrócimy tylko te rekordy z tabeli `owner`, dla których znaleźliśmy dopasowanie w tabeli `pet`
- `FULL OUTER JOIN` - (dłuższa nazwa na `FULL JOIN`) - zwrócimy wszystkie rekordy z tabeli `owner`, nawet te, dla których nie znajdziemy dopasowania w tabeli `pet` i jednocześnie zwrócimy wszystkie rekordy

z tabeli `pet`, nawet te, dla których nie znajdziemy dopasowania w tabeli `owner`,

- `CROSS JOIN` - ten rodzaj joina służy do generowania sparowanej kombinacji każdego wiersza z tabeli `owner` z każdym wierszem z tabeli `pet`. Inaczej jest to znane jako iloczyn kartezjański (*cartesian product*).

Żeby łatwiej było to sobie wyobrazić, spójrz na poniższą grafikę.



Obraz 1. Rodzaje JOINów

Przejdźmy teraz do omówienia każdego z nich.

INNER JOIN

Zacznijmy od zapytania *INNER JOIN*, spójrz na poniższy przykład:

Fragment metody selectExamples() w klasie OwnerRepository

```
String select7_1 = "SELECT ow FROM Owner ow INNER JOIN ow.pets pt"; ①
session.createQuery(select7_1, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

① Zamiast *INNER JOIN* możemy napisać samo *JOIN*.

Po wykonaniu powyższego przykładu, na ekranie zostanie wydrukowane:

```
Hibernate: select o1_0.owner_id,o1_0.email,o1_0.name,o1_0.phone,o1_0.surname
    from owner o1_0
    join pet p1_0 on o1_0.owner_id=p1_0.owner_id
Hibernate: select p1_0.owner_id,p1_0.pet_id,p1_0.breed,p1_0.name
    from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=46, name=Robert, surname=Nowacki, phone=+48 589 245 114, email=robert@zajavka.pl,
    pets=[Pet(id=62, name=Kiciek, breed=CAT), Pet(id=63, name=Fafik, breed=DOG)])
```

```
Hibernate: select p1_0.owner_id,p1_0.pet_id,p1_0.breed,p1_0.name
    from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=47, name=Adrian, surname=Paczkomat, phone=+48 894 256 331, email=adrian@zajavka.pl,
    pets=[Pet(id=64, name=Gucio, breed=DOG), Pet(id=65, name=Szymek, breed=MONKEY)])
```

Możesz zauważyć, że z jednej strony zostało wykonane zapytanie z JOIN, ale Hibernate i tak wykonał dwa dodatkowe zapytania do tabeli `pet`. W powyższym przykładzie, do bazy danych są również dodani właściciele, którzy nie mają zwierząt i w tym zapytaniu nie zostali oni zwróceni. Wynik jest prawidłowym przecięciem zbiorów `owner` i `pet`.

Żeby Hibernate nie wykonywał tych dodatkowych zapytań, moglibyśmy napisać zapytanie w ten sposób:

```
SELECT ow FROM Owner ow JOIN FETCH ow.pets pt
```

W takim przypadku, wykonywane zapytanie będzie wyglądało tak:

```
Hibernate: ①
    select
        o1_0.owner_id,
        o1_0.email,
        o1_0.name,
        p1_0.owner_id,
        p1_0.pet_id,
        p1_0.breed,
        p1_0.name,
        o1_0.phone,
        o1_0.surname
    from
        owner o1_0
    join
        pet p1_0
            on o1_0.owner_id=p1_0.owner_id
###Entity: Owner(id=46, name=Robert, surname=Nowacki, phone=+48 589 245 114, email=robert@zajavka.pl,
    pets=[Pet(id=62, name=Kiciek, breed=CAT), Pet(id=63, name=Fafik, breed=DOG)])
###Entity: Owner(id=47, name=Adrian, surname=Paczkomat, phone=+48 894 256 331, email=adrian@zajavka.pl,
    pets=[Pet(id=65, name=Szymek, breed=MONKEY), Pet(id=64, name=Gucio, breed=DOG)])
```

① Chwilowo na potrzeby tego zapytania zostało włączone formatowanie `hibernate.format_sql`. Dzięki temu prościej jest zauważyć, że Hibernate w klauzuli SELECT dodał również pola z tabeli `pet`.

Zjawisko wykonywania "zbyt dużej" ilości zapytań ma swoją nazwę i jeszcze do tego wrócimy.

LEFT OUTER JOIN

Przejdźmy do przykładu *LEFT JOIN*. W testowanej bazie danych są dodani właściciele w tabeli `owner`, którzy nie mają zwierząt. Spójrz na poniższy przykład:

Fragment metody selectExamples() w klasie OwnerRepository

```
String select7_2 = "SELECT ow FROM Owner ow LEFT JOIN FETCH ow.pets pt";
session.createQuery(select7_2, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Po wykonaniu powyższego przykładu, na ekranie zostanie wydrukowane:

```
Hibernate: select (...) from owner o1_0 left join pet p1_0 on o1_0.owner_id=p1_0.owner_id
###Entity: Owner(id=46, name=Robert, surname=Nowacki, phone=+48 589 245 114, email=robert@zajavka.pl,
    pets=[Pet(id=62, name=Kiciek, breed=CAT), Pet(id=63, name=Fafik, breed=DOG)])
###Entity: Owner(id=47, name=Adrian, surname=Paczkomat, phone=+48 894 256 331, email=adrian@zajavka.pl,
    pets=[Pet(id=64, name=Gucio, breed=DOG), Pet(id=65, name=Szymek, breed=MONKEY)])
###Entity: Owner(id=48, name=Romek, surname=Zabawniacha, phone=+48 658 745 322, email=romek@zajavka.pl,
    pets=[])
###Entity: Owner(id=49, name=Stefan, surname=Zajavkowicz, phone=+48 845 114 894, email=stefan@zajavka.pl,
    pets=[])
```

Zwrócone zostały rekordy z tabeli `owner` razem z tymi, które nie mają dowiązań w tabeli `pet` - czyli *LEFT JOIN* działa prawidłowo.

RIGHT OUTER JOIN

Przejdźmy do przykładu *RIGHT JOIN*. W testowanej bazie danych są dodani właściciele w tabeli `owner`, którzy nie mają zwierząt. Jednakże, w tabeli `pet` jest narzucony warunek, że klucz obcy do tabeli `owner` nie może być `null`. Oznacza to, że nie możemy mieć w bazie dodanego zwierzęcia, które nie ma skojarzonego właściciela. Spójrz na poniższy przykład:

Fragment metody selectExamples() w klasie OwnerRepository

```
String select7_3 = "SELECT ow FROM Owner ow RIGHT JOIN FETCH ow.pets pt";
session.createQuery(select7_3, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Po wykonaniu powyższego przykładu, na ekranie zostanie wydrukowane:

```
Hibernate: select (...)
    from owner o1_0
    right join pet p1_0 on o1_0.owner_id=p1_0.owner_id
###Entity: Owner(id=46, name=Robert, surname=Nowacki, phone=+48 589 245 114, email=robert@zajavka.pl,
    pets=[Pet(id=62, name=Kiciek, breed=CAT), Pet(id=63, name=Fafik, breed=DOG)])
###Entity: Owner(id=47, name=Adrian, surname=Paczkomat, phone=+48 894 256 331, email=adrian@zajavka.pl,
    pets=[Pet(id=64, name=Gucio, breed=DOG), Pet(id=65, name=Szymek, breed=MONKEY)])
```

Zwrócone zostały rekordy z tabeli `owner` razem z tymi, które mają dowiązania w tabeli `pet`, analogicznie jak w *INNER JOIN*. Sytuacja wynika z ograniczenia, które zostało wspomniane wcześniej.

Żeby pokazać prawidłowe działanie *RIGHT JOIN* musielibyśmy usunąć to ograniczenie, czyli rekord w `pet` może być dodany, jeżeli nie ma dowiązania do `owner`. Inaczej mówiąc, definicja tabeli musi wyglądać tak:

```
CREATE TABLE pet
(
    pet_id   SERIAL NOT NULL,
    name     VARCHAR(32) NOT NULL,
    breed    VARCHAR(32) NOT NULL,
    owner_id INT, ①
```

```
    PRIMARY KEY (pet_id),
    CONSTRAINT fk_pet_owner
        FOREIGN KEY (owner_id)
            REFERENCES owner (owner_id)
);
```

① W tym miejscu zdejmujemy ograniczenie `NOT NULL`.

Teraz będziemy mogli dodać testowe rekordy do tabeli `pet` i wykonać zapytanie z *RIGHT JOIN* ponownie.

Jeżeli wykonalibyśmy teraz takie zapytanie SQL:

```
SELECT * FROM Owner ow RIGHT JOIN Pet pet on ow.owner_id = pet.owner_id;
```

To na ekranie wydrukowałby się przykładowo taki wynik (dane są testowe):

ow.owner_id	ow.name	surname	phone	email	pet_id	pet.name	breed	pet.owner_id
null	null	null	null	null	1	Fafik	DOG	null
49	Stefan	Zajavkowicz	+48 845 114 894	stefan@zajavka.pl	2	Kiciek	CAT	49
null	null	null	null	null	3	Romek	MONKEY	null
47	Adrian	Paczkomat	+48 894 256 331	adrian@zajavka.pl	4	Stefek	MONKEY	47

Jeżeli wrócimy teraz do wykonania analogicznego zapytania HQL, to na ekranie zostanie wydrukowany taki wynik:

```
Hibernate: select (...)
    from owner o1_0
    right join pet p1_0 on o1_0.owner_id=p1_0.owner_id
###Entity: null
###Entity: Owner(id=49, name=Stefan, surname=Zajavkowicz, phone=+48 845 114 894, email=stefan@zajavka.pl,
    pets=[Pet(id=2, name=Kiciek, breed=CAT)])
###Entity: Owner(id=47, name=Adrian, surname=Paczkomat, phone=+48 894 256 331, email=adrian@zajavka.pl,
    pets=[Pet(id=4, name=Stefek, breed=MONKEY)])
```

Czyli w takim przypadku Hibernate zwraca jedną encję `null`, podczas gdy przy natywnym SQL mieliśmy 2 rekordy z wartościami `null`.

FULL OUTER JOIN

Przejdźmy do przykładu *FULL JOIN*. W testowanej bazie danych są dodani właściciele w tabeli `owner`, którzy nie mają zwierząt. W poprzednim przykładzie wprowadziliśmy możliwość istnienia zwierząt bez właścicieli i nadal ten przypadek podtrzymujemy. Oznacza to, że możemy mieć w bazie dane zwierzę, które nie ma skojarzonego właściciela. Zacznijmy przykład od takiego zapytania:

```
SELECT * FROM Owner ow FULL JOIN Pet pet on ow.owner_id = pet.owner_id;
```

Zwrócony zostałby wtedy przykładowo taki wynik:

ow.owner_id	ow.name	surname	phone	email	pet_id	pet.name	breed	pet.owner_id
null	null	null	null	null	1	Fafik	DOG	null
49	Stefan	Zajavkowicz	+48 845 114 894	stefan@zajavka.pl	2	Kiciek	CAT	49
null	null	null	null	null	3	Romek	MONKEY	null
47	Adrian	Paczkomat	+48 894 256 331	adrian@zajavka.pl	4	Stefek	MONKEY	47
46	Robert	Nowacki	+48 589 245 114	robert@zajavka.pl	null	null	null	null
48	Romek	Zabawniacha	+48 658 745 322	romek@zajavka.pl	null	null	null	null

Wykonajmy teraz analogiczny przykład HQL:

Fragment metody selectExamples() w klasie OwnerRepository

```
String select7_4 = "SELECT ow FROM Owner ow FULL JOIN FETCH ow.pets pt";
session.createQuery(select7_4, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Na ekranie zostanie wydrukowany taki wynik:

```
Hibernate: select (...)
    from owner o1_0
    full join pet p1_0 on o1_0.owner_id=p1_0.owner_id
###Entity: null
###Entity: Owner(id=49, name=Stefan, surname=Zajavkowicz, phone=+48 845 114 894, email=stefan@zajavka.pl,
    pets=[Pet(id=2, name=Kiciek, breed=CAT)])
###Entity: Owner(id=47, name=Adrian, surname=Paczkomat, phone=+48 894 256 331, email=adrian@zajavka.pl,
    pets=[Pet(id=4, name=Stefek, breed=MONKEY)])
###Entity: Owner(id=46, name=Robert, surname=Nowacki, phone=+48 589 245 114, email=robert@zajavka.pl,
    pets=[])
###Entity: Owner(id=48, name=Romek, surname=Zabawniacha, phone=+48 658 745 322, email=romek@zajavka.pl,
    pets=[])
```

Czyli w takim przypadku Hibernate zwraca jeden `null`, podczas gdy przy natywnym SQL mieliśmy 2 rekordy z wartościami `null`. Do tego można zrozumieć ten przykład jak nałożenie na siebie przykładów *LEFT JOIN* oraz *RIGHT JOIN*.

CROSS JOIN

O tym rodzaju JOINa nie rozmawialiśmy w poprzednich materiałach, ale przyszedł moment, gdzie warto jest żebyśmy go omówili. Zanim przejdziemy do omówienia *CROSS JOIN*, wyjaśnijmy w prostych słowach czym jest **iloczyn kartezjański**. Można powiedzieć, że **iloczyn kartezjański** to iloczyn dwóch zbiorów (zbioru A i zbioru B). Efekt takiego iloczynu jest zbiorem, który zawiera wszystkie pary (a, b),

gdzie a należy do zbioru A i b należy do zbioru B. Oznacza się go symbolem *A X B*. Jeżeli mielibyśmy dwa zbiory *A = {a, b}* oraz *B = {1,2,3}*, to iloczyn kartezjański wyglądałby tak:

```
A X B = {{a,1},{a,2},{a,3},{b,1},{b,2},{b,3}}
```

A jak się to ma do tabel w bazie danych?

Iloczyn kartezjański w przypadku baz danych może być przedstawiony przy wykorzystaniu joina *CROSS JOIN* (inne stwierdzenie jakie można spotkać to *cartesian join*). Wynikiem takiego joina będzie zbiór par, gdzie każdy wiersz z pierwszej tabeli będzie sparowany ze wszystkimi wierszami w drugiej tabeli. Jeżeli uprościmy dwie poprzednio używane tabele `owner` oraz `pet` do takiej postaci:

Tabela 1. Tabela Owner

owner_id	name
1	Robert
2	Stefan
3	Adam

Tabela 2. Tabela Pet

pet_id	name
1	Gucio
2	Fafik
3	Burek

Mając takie dane zastanówmy się teraz nad takim przypadkiem. Która kombinacja właściciel i zwierze będzie ważyła najmniej? Robert i Gucio? Czy może Stefan i Burek? *CROSS JOIN* ma za zadanie wygenerować wszystkie kombinacje par na podstawie danych w tabelach.

Jeżeli wykonamy teraz poniższe zapytanie na bazie danych:

```
SELECT ow.name, pet.name FROM Owner ow CROSS JOIN Pet pet;
```

To na ekranie zostanie wydrukowany taki rezutalt:

ow.name	pet.name
Robert	Gucio
Robert	Fafik
Robert	Burek
Stefan	Gucio
Stefan	Fafik
Stefan	Burek
Adam	Gucio
Adam	Fafik
Adam	Burek

Niektóre bazy danych wspierają jeszcze taki zapis:

```
SELECT ow.name, pet.name FROM Owner ow, Pet pet;
```

Jako wynik otrzymaliśmy zbiór par będących iloczynem dwóch zbiorów. Czyli otrzymaliśmy zbiór par każdego właściciela z każdym zwierzęciem. Łatwo policzyć, *3 x 3 = 9*.

A co ma do tego Hibernate?

Hibernate również wspiera wykonywanie *CROSS JOIN*, spójrz na poniższy przykład:

```
String select7_5 = "SELECT ow.name, pt.name FROM Owner ow CROSS JOIN Pet pt";
session.createQuery(select7_5, Object[].class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + Arrays.asList(entity)));
```

Na ekranie zostaną teraz wydrukowane pary, analogicznie do pokazanych wcześniej tabel:

```
###Entity: [Robert, Fafik]
###Entity: [Robert, Kiciek]
###Entity: [Robert, Romek]
###Entity: [Adrian, Fafik]
###Entity: [Adrian, Kiciek]
###Entity: [Adrian, Romek]
###Entity: [Stefan, Fafik]
###Entity: [Stefan, Kiciek]
###Entity: [Stefan, Romek]
```

Wiele joinów

Na potrzeby tego przykładu, do właściciela i zwierzątek dodajmy jeszcze zabawki. Zabawki mogą być wykorzystywane przez wiele zwierzątek, bo przecież jeden właściciel może mieć wiele zwierzątek i jeżeli wszyscy mieszkają pod jednym dachem, to jedno zwierzątko może bawić się wieloma zabawkami i jedna zabawka może być używana przez wiele zwierzątek. Dodajmy zatem poniższe tabele:

Tabela Toy

```
CREATE TABLE toy
(
    toy_id SERIAL NOT NULL,
    what   VARCHAR(32) NOT NULL,
    color  VARCHAR(32) NOT NULL,
    PRIMARY KEY (toy_id)
);
```

Tabela Pet_Toy

```
CREATE TABLE pet_toy
(
    pet_toy_id SERIAL NOT NULL,
    pet_id     INT NOT NULL,
    toy_id     INT NOT NULL,
    CONSTRAINT fk_pet_toy_pet
```

```
        FOREIGN KEY (pet_id)
            REFERENCES pet (pet_id),
    CONSTRAINT fk_pet_toy_toy
        FOREIGN KEY (toy_id)
            REFERENCES toy (toy_id)
);
```

Mapowanie Owner

```
package pl.zajavka;

import jakarta.persistence.*;
import lombok.*;

import java.util.Set;

@Getter
@Setter
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "owner")
public class Owner {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="owner_id", unique = true, nullable = false)
    private Integer id;

    @Column(name = "name")
    private String name;

    @Column(name = "surname")
    private String surname;

    @Column(name = "phone")
    private String phone;

    @Column(name = "email")
    private String email;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "owner", cascade = CascadeType.ALL)
    private Set<Pet> pets;

    @Override
    public String toString() {
        return "Owner(id=" + this.getId() + ", name=" + this.getName()
            + ", surname=" + this.getSurname() + ", phone=" + this.getPhone()
            + ", email=" + this.getEmail() + ", pets=" + this.getPets() + ")";
    }
}
```

Mapowanie Pet

```
package pl.zajavka;

import jakarta.persistence.*;
```

```
import lombok.*;

import java.util.Set;

@Getter
@Setter
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "pet")
public class Pet {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "pet_id", unique = true, nullable = false)
    private Long id;

    @Column(name = "name")
    private String name;

    @Enumerated(EnumType.STRING)
    @Column(name = "breed")
    private Breed breed;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "owner_id")
    private Owner owner;

    @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinTable(
        name = "pet_toy",
        joinColumns = {@JoinColumn(name = "pet_id")},
        inverseJoinColumns = {@JoinColumn(name = "toy_id")}
    )
    private Set<Toy> toys;

    @Override
    public String toString() {
        return "Pet(id=" + this.getId() + ", name=" + this.getName()
            + ", breed=" + this.getBreed() + ", toys=" + this.getToys() + ")";
    }
}
```

Mapowanie Toy

```
package pl.zajavka;

import jakarta.persistence.*;
import lombok.*;

import java.util.Set;

@Getter
@Setter
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "toy")
```

```
public class Toy {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "toy_id")
    private Integer toyId;

    @Column(name = "what")
    private String what;

    @Column(name = "color")
    private String color;

    @ManyToMany(mappedBy = "toys")
    private Set<Pet> pets;

    @Override
    public String toString() {
        return "Toy(toyId=" + this.getToyId()
            + ", what=" + this.getWhat() + ", color=" + this.getColor() + ")";
    }
}
```

Teraz musimy dodać dane do bazy danych:

Metoda saveData() w klasie OwnerRepository

```
void saveTestData() {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        session.beginTransaction();
        // Toy creation and saving
        Toy toy1 = ExampleData.someToy1();
        Toy toy2 = ExampleData.someToy2();
        Toy toy3 = ExampleData.someToy3();
        Toy toy4 = ExampleData.someToy4();
        session.persist(toy1);
        session.persist(toy2);
        session.persist(toy3);
        session.persist(toy4);

        // Tet creation
        Pet pet1 = ExampleData.somePet1();
        Pet pet2 = ExampleData.somePet2();
        Pet pet3 = ExampleData.somePet3();
        Pet pet4 = ExampleData.somePet4();
        pet1.setToys(Set.of(toy1, toy2));
        pet2.setToys(Set.of(toy2, toy3));
        pet3.setToys(Set.of(toy1, toy2, toy3));
        pet4.setToys(Set.of(toy2, toy3, toy4));

        // Owner creation and saving
        Owner owner1 = ExampleData.someOwner1();
        Owner owner2 = ExampleData.someOwner2();
        owner1.setPets(Set.of(pet1, pet2));
        owner2.setPets(Set.of(pet3, pet4));
        owner1.getPets().forEach(pet -> pet.setOwner(owner1));
        owner2.getPets().forEach(pet -> pet.setOwner(owner2));
```

```
        session.persist(owner1);
        session.persist(owner2);
        session.getTransaction().commit();
    }
}
```

Jeżeli teraz spróbujemy wywołać taki fragment kodu:

```
String select8 = "SELECT ow FROM Owner ow INNER JOIN FETCH ow.pets pt INNER JOIN FETCH pt.toys ts";
session.createQuery(select8, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

To Hibernate wygeneruje nam takie zapytanie:

```
select o1_0.owner_id,
       o1_0.email,
       o1_0.name,
       p1_0.owner_id,
       p1_0.pet_id,
       p1_0.breed,
       p1_0.name,
       t1_0.pet_id,
       t1_1.toy_id,
       t1_1.color,
       t1_1.what,
       o1_0.phone,
       o1_0.surname
from owner o1_0
         join pet p1_0 on o1_0.owner_id = p1_0.owner_id
         join (pet_toy t1_0 join toy t1_1 on t1_1.toy_id = t1_0.toy_id) on p1_0.pet_id = t1_0.pet_id
```

Na ekranie zostanie wtedy wydrukowanych dwóch właścicieli, każdy właściciel będzie miał dodanego swojego zwierzaka, do tego każdy zwierzak będzie miał przypisaną zabawkę, którą się bawi. Przykład ten pokazuje, że tworzenie skomplikowanych zapytań z wielokrotnymi zagnieżdżeniami może być łatwiejsze przy wykorzystaniu Hibernate, niż gdyby pisać to ręcznie.

Agregacje

HQL tak samo jak SQL obsługuje metody agregujące. Przypomnę, że metody agregujące to były m.in:

- `avg()` - wyliczanie wartości średniej,
- `count()` - zliczanie ilości wystąpień,
- `max()` - wyliczanie wartości maksymalnej,
- `min()` - wyliczanie wartości minimalnej,
- `sum()` - zliczanie sumy,

Działanie metod agregujących jest takie same w HQL jak w SQL, przy czym w HQL odnosimy się do pól w encjach, a nie do kolumn w tabelach. Poniżej przykład:

```
String select9_1 = "SELECT COUNT(t.toyId) FROM Toy t";
session.createQuery(select9_1, Long.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Hibernate wygeneruje wtedy zapytanie:

```
select count(t1_0.toy_id) from toy t1_0
```

A na ekranie zostanie wydrukowane np. 4.

Jeżeli stosowalibyśmy bardziej skomplikowane zapytanie z agregacjami, jak w przykładzie poniżej, to musielibyśmy np. wykorzystać swoje **DTO**:

```
public record ToyStat(String maxWhat, Long calculated) {}
```

```
String select9_2 = """
    SELECT new pl.zajavka.ToyStat(
            MAX(t.what),
            SUM(t.toyId) / COUNT(t.toyId)
        ) FROM Toy t
    """;
session.createQuery(select9_2, ToyStat.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Wyliczanie średniej z kluczy głównych nie ma za bardzo sensu, ale powyższy przykład miał cel edukacyjny 😊.

HQL - Named query

W pokazywanych przykładach stworzyliśmy bardzo dużo zapytań. Pojawia się zatem pytanie, czy da się to jakoś uporządkować. Oczywiście, że się da, możemy w tym celu wykorzystać **named queries**.

Named Queries pomagają w grupowaniu zapytań HQL w jednym miejscu i możemy się do nich odwoływać przy wykorzystaniu określonych nazw.

Zalety stosowania named queries:

- Named queries pozwalają na grupowanie zapytań HQL w jednym miejscu, a w efekcie mamy czystszy kod. Zwiększa to również możliwość ich wielokrotnego wykorzystania,
- Składnia takich zapytań jest sprawdzana na etapie tworzenia *SessionFactory*, co powoduje, że aplikacja wyrzuci nam błąd już na etapie jej uruchamiania. Ma to wielkie znaczenie, gdy aplikacja taka będzie uruchamiana na serwerze.

Named queries są definiowane przy wykorzystaniu adnotacji @NamedQuery. Adnotację `@NamedQuery` osadza się w klasach z adnotacjami `@Entity`. Adnotacje takie można grupować przy wykorzystaniu

adnotacji `@NamedQueries`.

Przykład definicji `@NamedQuery`:

```
import org.hibernate.annotations.NamedQueries;
import org.hibernate.annotations.NamedQuery;

@NamedQueries({
    @NamedQuery(
            name = "Person.findAll",
            query = "FROM Person"),
    @NamedQuery(
            name = "Person.findPersonByName",
            query = "FROM Person WHERE name = :name"
    )
})
@Entity
public class Person {
    // ...
}
```

Przykład użycia `@NamedQuery`:

```
import org.hibernate.query.Query;

// ...

Person person = session.createNamedQuery("Person.findPersonByName", Person.class)
    .setParameter("name", "Karol")
    .getSingleResult();
```

FetchMode

Oprócz parametru `FetchType`, Hibernate pozwala na ustawienie parametru `FetchMode`. Parametr `FetchType` można rozumieć jak: *Czy pobierać powiązane encje?*, natomiast `FetchMode` można rozumieć jak: *Jak pobierać powiązane encje?*. Bardzo przyjemnie opisuje to dokumentacja enuma `FetchMode`:

> How the association should be fetched. Defines the "how", compared to jakarta.persistence.FetchType which defines "when".

Ustawienie tego parametru ma w konsekwencji wpływ na wydajność aplikacji, a za wydajność rozwiązania odpowiedzialny jest programista ☺.

FetchMode.SELECT

W kilku przypadkach jest to ustawienie domyślne. Każda encja powiązana z encją główną była pobierana przy wykorzystaniu dodatkowego zapytania *SELECT*. Jak możesz się domyślić, im mniej zapytań zostanie wykonanych w celu pobrania takiej samej ilości danych tym lepiej. Przykład:

Metoda getOwner() z klasy OwnerRepository

```
Optional<Owner> getOwner(Integer ownerId) {
```

```
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        return Optional.ofNullable(session.find(Owner.class, ownerId));
    }
}
```

Zachowanie Hibernate będzie w tej sytuacji zależne od ustawionych parametrów. Na początku załóżmy, że mamy określoną taką konfigurację:

Klasa Owner

```
import org.hibernate.annotations.Fetch;
import org.hibernate.annotations.FetchMode; ①

// ...

@OneToMany(fetch = FetchType.LAZY, mappedBy = "owner", cascade = CascadeType.ALL)
@Fetch(value = FetchMode.SELECT) ②
private Set<Pet> pets;
```

① Warto tutaj zaznaczyć jedną kwestię. Enum `FetchType` jest umieszczony w paczce `jakarta.persistence`, natomiast enum `FetchMode` jest umieszczony w paczce `org.hibernate.annotations`. Oznacza to, że `FetchType` jest mechanizmem JPA, natomiast `FetchMode` jest mechanizmem Hibernate. Warto pamiętać o takich smaczkach, bo każda implementacja JPA może mieć dostępne inne "swoje własne" mechanizmy.

② Nie musimy tego dopisywać, gdyż domyślna wartość to *FetchMode.SELECT*.

Jeżeli przy wykorzystaniu powyższych ustawień wywołamy teraz metodę `getOwner()` (zakładając, że mamy jakieś dane w bazie):

Klasa ExampleRunner

```
package pl.zajavka;

public class ExampleRunner {

    public static void main(String[] args) {
        OwnerRepository ownerRepository = new OwnerRepository();
        ownerRepository.getOwner(58);
        HibernateUtil.closeSessionFactory();
    }
}
```

To Hibernate wydrukuje taki rezultat:

```
Hibernate: select o1_0.owner_id,o1_0.email,o1_0.name,o1_0.phone,o1_0.surname from owner o1_0 where
o1_0.owner_id=?
```

Dwa ustawienia mają teraz tutaj znaczenie: `FetchType.LAZY` oraz `FetchMode.SELECT`. Nigdzie w kodzie nie odwołuję się do `getPets()`, zatem nie są one pobierane, bo nie są potrzebne. Jeżeli natomiast zmienię

`FetchType.LAZY` na `FetchType.EAGER`, to Hibernate "dociągnie" zwierzątka przy wykorzystaniu kolejnego selecta. Z drugiej strony, jeżeli zostawię parametr `FetchType.LAZY` i zmienię metodę `getOwner()` na taką:

Metoda getOwner() z klasy OwnerRepository

```
Optional<Owner> getOwner(Integer ownerId) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        Owner owner = session.find(Owner.class, ownerId);
        System.out.println("###Owner: " + owner);
        return Optional.ofNullable(owner);
    }
}
```

To jeżeli metoda `Owner#toString()` wywołuje pod spodem `getPets()`, to Hibernate wykona dodatkowe zapytanie do tabeli `pet`. Jak widać w załączonym przykładzie, każde dodatkowe pobranie danych jest realizowane przy wykorzystaniu dodatkowego zapytania *SELECT*. Natomiast parametr `FetchMode` przyjmuje również inne wartości.

FetchMode.JOIN

Korzystając z tego samego przykładu, napiszmy teraz taki kod:

Klasa Owner

```
import org.hibernate.annotations.Fetch;
import org.hibernate.annotations.FetchMode;

// ...

@OneToMany(fetch = FetchType.EAGER, mappedBy = "owner", cascade = CascadeType.ALL)
@Fetch(value = FetchMode.JOIN) ①
private Set<Pet> pets;
```

① Parametr jest zmieniony z *FetchMode.SELECT* na *FetchMode.JOIN*.

Jeżeli uruchomimy powyższy przykład ponownie, to Hibernate wykona teraz takie zapytanie:

```
select
    o1_0.owner_id,
    o1_0.email,
    o1_0.name,
    p1_0.owner_id,
    p1_0.pet_id,
    p1_0.breed,
    p1_0.name,
    o1_0.phone,
    o1_0.surname
from
    owner o1_0
left join
    pet p1_0
        on o1_0.owner_id=p1_0.owner_id
```

```
where
    o1_0.owner_id=?
```

Na tej podstawie widać, że tryb *FetchMode.JOIN* zmienia tryb "dociągania" encji zależnych z dodatkowego zapytania *SELECT* na *JOIN*. Można powiedzieć, że efekt jaki zobaczyliśmy na ekranie jest podobny do wykonania zapytania HQL:

```
SELECT ow FROM Owner ow LEFT JOIN FETCH ow.pets pt
```

FetchMode.SUBSELECT

Parametr `FetchMode` daje nam jeszcze możliwość trzeciego ustawienia. Do przykładu `Owner` oraz `Pet` dodajmy również `Toy` tak jak to było widoczne w poprzednich przykładach i narzućmy taką konfigurację:

Klasa Owner

```
@OneToMany(fetch = FetchType.EAGER, mappedBy = "owner", cascade = CascadeType.ALL)
@Fetch(value = FetchMode.SELECT)
private Set<Pet> pets;
```

Klasa Pet

```
@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
@JoinTable(
    name = "pet_toy",
    joinColumns = {@JoinColumn(name = "pet_id")},
    inverseJoinColumns = {@JoinColumn(name = "toy_id")}
)
@Fetch(FetchMode.SUBSELECT)
private Set<Toy> toys;
```

Klasa Toy

```
@ManyToMany(mappedBy = "toys")
private Set<Pet> pets;
```

Jeżeli przykładowo wykonasz teraz metodę `getOwner()` to Hibernate wygeneruje m.in. takie zapytanie:

```
select
    t1_0.pet_id,
    t1_1.toy_id,
    t1_1.color,
    t1_1.what
from
    pet_toy t1_0
join
    toy t1_1
        on t1_1.toy_id=t1_0.toy_id
where
    t1_0.pet_id in(select ①
```

```
        p1_0.pet_id
    from
        pet p1_0
    where
        p1_0.owner_id=?)
```

① Istotą parametru *FetchMode.SUBSELECT* jest właśnie to zapytanie. Czyli parametr ten mówi, że dodatkowe dane mają być "dociągane" przy wykorzystaniu zagnieżdżonego *SELECT*, inaczej mówiąc *SUBSELECT*. Będzie to oznaczało, że i tak wykonają się dwa zapytania, jedno do pobrania encji nadrzędnych i jedno do pobrania encji podrzędnych.

Może pojawić się zatem pytanie: *Czy da się w przypadku* `Owner`, `Pet`, `Toy` *wykonać metodę* `getOwner()` *tak żeby zostało wykonane tylko jedno zapytanie?*. Ustaw w klasie `Owner` oraz `Pet` parametr *FetchMode.JOIN*. Hibernate wykona wtedy takie zapytanie:

```
select
    o1_0.owner_id,
    o1_0.email,
    o1_0.name,
    p1_0.owner_id,
    p1_0.pet_id,
    p1_0.breed,
    p1_0.name,
    t1_0.pet_id,
    t1_1.toy_id,
    t1_1.color,
    t1_1.what,
    o1_0.phone,
    o1_0.surname
from
    owner o1_0
left join
    pet p1_0
        on o1_0.owner_id=p1_0.owner_id
left join
    (pet_toy t1_0
join
    toy t1_1
        on t1_1.toy_id=t1_0.toy_id)
            on p1_0.pet_id=t1_0.pet_id
    where
        o1_0.owner_id=?
```

W praktyce ustawienia *FetchMode* są wykorzystywane do optymalizacji wykonywanych zapytań.

Podsumowanie

Podsumowując, parametr `FetchMode` określa **jak** Hibernate ma dociągnąć encje podrzędne, a nie czy w ogóle ma to zrobić. To `FetchType` odpowiada na pytanie **czy** dociągać encje podrzędne. Może pojawić się zatem pytanie, jakie wartości domyślne są stosowane przez Hibernate, jeżeli nie ustawimy parametru `FetchMode` sami. Rozpiszmy te przypadki:

- Jeżeli ustawimy *FetchType.LAZY* to domyślną wartością będzie *FetchMode.SELECT*,
- Jeżeli ustawimy *FetchType.EAGER*

- Jeżeli pobieramy wartość przez `session.find()`, to domyślną wartością będzie *FetchMode.JOIN*,
- Jeżeli pobieramy wartość przez JPQL, to domyślną wartością będzie *FetchMode.SELECT*,

Odpowiednie ustawienie `FetchMode` może znacznie wpłynąć na wydajność naszej aplikacji, zwłaszcza gdy musimy łączyć ze sobą tabele w zapytaniach. Należy również pamiętać, że w bardziej skomplikowanych przypadkach, bardziej wydajne może okazać się napisane własnego zapytania JPQL. Dlatego zawsze warto jest sprawdzać i dopasowywać ustawienia do naszego indywidualnego przypadku. Na koniec, dokumentacja.

Problem *n + 1 zapytań*

Troszeczkę podrapiemy temat wydajności działania Hibernate w kontekście generowania nadmiernych zapytań i omówimy problem *n + 1 zapytań*.

W poprzednich materiałach powiedziałem, że wrócimy do tego dlaczego Hibernate generuje tak dużo zapytań i jak sobie z tym poradzić. Wspomniałem również, że problem taki nosi swoją nazwę. Właśnie doszliśmy do tej tematyki, a problem ten nazywa się `n + 1`.

Problem ten jest związany z pobieraniem kolekcji. Najczęściej objawia się w relacjach **one-to-many**, a z racji, że tak jak wspomniałem wcześniej, relacja **one-to-many** jest w praktyce najbardziej popularna, to ten problem również często występuje w praktyce.

Jak widzisz, Hibernate jest potężnym narzędziem i należy go używać z pełną świadomością stosowanych przez nas parametrów. To samo ma się do wszystkich stosowanych przez nas narzędzi. Nie możemy opierać się w 100% na narzędziach, które zrobią wszystko za nas. Powinniśmy również rozumieć jak wiele rzeczy działa pod spodem i znać konsekwencje naszych wyborów.

Tutaj chcę uspokoić, to wszystko przychodzi z czasem. Im więcej będziesz mieć doświadczenia w programowaniu, tym więcej mechanizmów/technik i narzędzi będziesz rozumieć. Nikt nie będzie oczekiwał od Ciebie znajomości całego narzędzia na wylot (przynajmniej ja nikogo takiego nie znam) razem ze wszystkimi możliwymi ustawieniami - do tego jest dokumentacja. Powinniśmy jednak rozumieć konsekwencje wybieranych ustawień.

Dlaczego ten problem jest istotny

Jeżeli oprzemy się o wartości domyślne i będziemy kodować bez zastanowienia, to możemy dojść do sytuacji, gdzie aplikacja w ciągu sekundy będzie wykonywała na bazie danych np. 500 zapytań, a odpowiednia konfiguracja parametrów mogłaby ograniczyć ilość takich zapytań do np. 10. Przykład ten pokazuje również, że nie dość, że programista jest odpowiedzialny za odpowiednie "obkodowanie" procesu biznesowego, to jeszcze w jego/jej kwestii leży odpowiednie dobranie parametrów, żeby aplikacja pracowała wydajnie.

Czyli najważniejszą kwestią, która dotyczy tego problemu jest tragiczny spadek wydajności aplikacji. Pamiętaj, że z aplikacji produkcyjnych korzystają często miliony klientów. Najzabawniejsze w tej całej sytuacji jest to, że jeden parametr może przesądzić o tym, że setkom tysięcy klientów będzie się z aplikacji korzystało dobrze, albo będą mieli jej serdecznie dość ☺.

Jak go zauważyć

Tutaj dochodzimy do ciekawej kwestii. Problem ten można najprościej wykryć przez obserwację i analizę wykonywanych przez Hibernate zapytań. Przypomnę tylko, że do włączenia logowania wykonywanych przez Hibernate zapytań służy ustawienie `hibernate.show_sql`. Jeżeli nasza aplikacja będzie działała wolno, warto sprawdzić na początku, czy Hibernate nie wykonuje zbyt dużo zapytań na bazie danych.

Przykład

Przechodząc już do konkretów, widzieliśmy już ten problem wcześniej, zwyczajnie go nie nazwaliśmy. Ponownie oprzemy się o przykład `Owner` oraz `Pet`. Jeżeli ustawisz w kodzie poniższą konfigurację:

Klasa Owner

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "owner", cascade = CascadeType.ALL)
private Set<Pet> pets;
```

Klasa Pet

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "owner_id")
private Owner owner;
```

I dodasz do swojej bazy danych dane w taki sposób, żeby mieć 5 właścicieli i każdy z tych właścicieli ma mieć po jednym zwierzątku, na następnie wykonasz poniższy kod:

```
String select1 = "SELECT ow FROM Owner ow";
session.createQuery(select1, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

To na ekranie zostanie wydrukowane 6 zapytań:

```
Hibernate: select (...) from owner o1_0
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=?
###Entity: Owner(id=63, name=Robert, ...
###Entity: Owner(id=64, name=Adrian, ...
###Entity: Owner(id=65, name=Agnieszka, ...
###Entity: Owner(id=66, name=Dominik, ...
###Entity: Owner(id=67, name=Dawid, ...
```

Widać jak na dłoni, że wykonanych zostało 6 zapytań, czyli w sumie bardziej `1 + n`, niż `n + 1`, ale problem nazywa się `n + 1` ☺. Hibernate domyślnie pobiera listę użytkowników (1 zapytanie). Do tego dla każdego użytkownika chcemy pobrać listę zwierzątek, dlatego dla każdego użytkownika, Hibernate wykonuje jeszcze jedno zapytanie, przy czym `n` oznacza tutaj liczbę użytkowników.

Jak można sobie z tym poradzić

Rozwiązanie tego problemu poznaliśmy już wcześniej, ale powtórzmy ten przykład dla utrwalenia. Musimy powiedzieć Hibernate'owi, że chcemy pobrać te rekordy w jednym zapytaniu, a nie w 6. Zrobimy to dodając słówko `fetch` w zapytaniu HQL.

```
SELECT ow FROM Owner ow JOIN FETCH ow.pets pt
```

Jest to zalecane rozwiązanie tego problemu. Hibernate w tym przypadku pobierze wszystkich właścicieli wraz ze zwierzątkami w jednym zapytaniu.

Drugi sposób na rozwiązanie tego problemu to ustawienie parametru `BatchSize`. Spójrz na poniższy przykład:

Klasa Owner

```
import org.hibernate.annotations.BatchSize;

// ...

@OneToMany(fetch = FetchType.EAGER, mappedBy = "owner", cascade = CascadeType.ALL)
@BatchSize(size = 3)
private List<Pet> pets;
```

Zastosowanie tej adnotacji z parametrem spowoduje, że encje podrzędne będą pobierane "W porcjach". Czyli jeżeli mamy w naszej bazie danych 100 właścicieli i każdy z nich ma po 5 zwierząt, to jeżeli ustawimy parametr *size = 25* to zamiast 100 dodatkowych zapytań, będziemy musieli wykonać tylko 4. Jeżeli poprzednie rozwiązanie będzie powodowało problemy, możemy spróbować podejść do problemu również w ten sposób. Jeżeli uruchomimy teraz przykład z powyższym ustawieniem, to Hibernate wygeneruje takie zapytania:

```
Hibernate: select (...) from owner o1_0
Hibernate: select (...) from pet where p1_0.owner_id in(?,?,?) ①
Hibernate: select (...) from pet where p1_0.owner_id in(?,?)
```

① Hibernate dodał teraz klauzulę *IN* żeby pobierać zwierzątka w porcjach, po kilku właścicieli. Zatem zamiast 5 dodatkowych zapytań, mamy teraz 2.

Trzecim sposobem jest wykorzystanie parametru *FetchMode*, którego zastosowanie widzieliśmy już wcześniej.

Set vs List w mapowaniach

Pewnie zwróciłeś/zwróciłaś uwagę, że w mapowaniach encji stosowany był `Set`, a nie `List`. To jest dosyć niejednoznaczna kwestia, której kolekcji używać w mapowaniach Hibernate i nie ma to jednej odpowiedzi (chyba jak w większości problemów w programowaniu 😆). To, która z nich powinna być używana zależy od sytuacji i ponownie, żeby dokonać wyboru, trzeba znać potencjalne konsekwencje. Oczywiście mówimy tutaj o używaniu `Set` vs `List` w kontekście relacji one-to-many oraz many-to-many.

List

Jeżeli zdecydujemy się na stosowanie w mapowaniach interfejsu `List`, to należy pamiętać, że Hibernate do implementacji listy wykorzystuje **PersistentBag**. Jest to implementacja analogiczna do `LinkedList` czy `ArrayList` i posiada pewne swoje charakterystyczne cechy. **PersistentBag** może zawierać duplikaty i jest nieuporządkowana. Cytując dokumentację **PersistentBag**:

> An unordered, unkeyed collection that can contain the same element multiple times. The Java collections API, curiously, has no Bag. Most developers seem to use Lists to represent bag semantics, so Hibernate follows this practice.

Możliwość dodawania do listy duplikatów w przypadku Hibernate oznacza, że gdy dodajemy i usuwamy z tej listy elementy, to może to spowodować generowanie dodatkowych zapytań do baz danych. Dlatego dla bezpieczeństwa można powiedzieć, że dobrze jest w relacjach one-to-many oraz many-to-many stosować `Set`, bo eliminujemy wspomniany problem. Spójrz na poniższy przykład:

Klasa Pet

```
@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
@JoinTable(
    name = "pet_toy",
    joinColumns = {@JoinColumn(name = "pet_id")},
    inverseJoinColumns = {@JoinColumn(name = "toy_id")}
)
@Fetch(FetchMode.JOIN)
private Set<Toy> toys;
```

Klasa Toy

```
@ManyToMany(mappedBy = "toys")
private Set<Pet> pets;
```

Fragment metody manipulateData() w klasie OwnerRepository

```
List<Pet> pets = session.find(Owner.class, ownerId).getPets();
pets.forEach(pet -> pet.getToys().remove(pet.getToys().stream().findAny().get())); ①
```

① Usuwamy zwierzątkom jakąkolwiek zabawkę

Hibernate wygeneruje w takim przypadku poniższe zapytania:

```
Hibernate: select (...) from owner o1_0
    left join pet p1_0 on o1_0.owner_id=p1_0.owner_id
    left join (pet_toy t1_0 join toy t1_1 on t1_1.toy_id=t1_0.toy_id) on p1_0.pet_id=t1_0.pet_id
    where o1_0.owner_id=?
Hibernate: delete from pet_toy where pet_id=?
Hibernate: delete from pet_toy where pet_id=?
```

Jeżeli natomiast mapowania wyglądałyby w taki sposób:

Klasa Pet

```
@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
@JoinTable(
    name = "pet_toy",
    joinColumns = {@JoinColumn(name = "pet_id")},
    inverseJoinColumns = {@JoinColumn(name = "toy_id")}
)
@Fetch(FetchMode.JOIN)
private List<Toy> toys;
```

Klasa Toy

```
@ManyToMany(mappedBy = "toys")
private List<Pet> pets;
```

Fragment metody manipulateData() w klasie OwnerRepository

```
List<Pet> pets = session.find(Owner.class, ownerId).getPets();
pets.forEach(pet -> pet.getToys().remove(0)); ①
```

① Usuwamy zwierzątkom jakąkolwiek zabawkę z pozycji 0

Hibernate wygeneruje w takim przypadku poniższe zapytania:

```
Hibernate: select (...) from owner o1_0 left join pet p1_0 on o1_0.owner_id=p1_0.owner_id where
o1_0.owner_id=?
Hibernate: select (...) from pet_toy t1_0 join toy t1_1 on t1_1.toy_id=t1_0.toy_id where t1_0.pet_id=?
Hibernate: select (...) from pet_toy t1_0 join toy t1_1 on t1_1.toy_id=t1_0.toy_id where t1_0.pet_id=?
Hibernate: delete from pet_toy where pet_id=?
Hibernate: insert into pet_toy (pet_id, toy_id) values (?, ?)
Hibernate: insert into pet_toy (pet_id, toy_id) values (?, ?)
Hibernate: delete from pet_toy where pet_id=?
Hibernate: insert into pet_toy (pet_id, toy_id) values (?, ?)
Hibernate: insert into pet_toy (pet_id, toy_id) values (?, ?)
```

Czyli jak widać w załączonym przykładzie, już samo zastosowanie `Set` zamiast `List` ograniczyło ilość generowanych zapytań. Hibernate w przypadku stosowania mapowania z `List` usuwa wpisy z tabeli `pet_toy`, a później dodaje je ponownie.

A co oznacza to nieuporządkowanie? Implementacja **PersistentBag** nie zachowuje kolejności wstawiania elementów w liście, co może wydawać się dziwne, bo przecież robi to `ArrayList` oraz `LinkedList`. W przypadku `List`, można z niej zrobić listę uporządkowaną, należy w tym celu np. wykorzystać adnotację `@OrderBy`. Przykład:

Klasa Owner

```
import jakarta.persistence.OrderBy;

// ...

@OneToMany(fetch = FetchType.EAGER, mappedBy = "owner", cascade = CascadeType.ALL)
@OrderBy("breed ASC")
```

```
private List<Pet> pets;
```

Czyli podsumowując, Listę można używać, gdy zależy nam na posiadaniu duplikatów w zwróconej kolekcji oraz gdy chcemy zwrócony wynik posortować.

Set

`Set` ze względu na swoją naturę nie pozwala na przechowywanie duplikatów. Widzieliśmy również wcześniej, że stosując go w przypadku relacji np. **many-to-many**, ograniczamy ilość generowanych przez Hibernate zapytań. Trzeba tutaj również dodać, że `Set` w swojej implementacji zajmuje więcej pamięci niż `List`, ale tym będzie się za nas zajmował *Garbage Collector*. Hibernate'owa implementacja `Set` to **PersistentSet**. Dokumentacja tej klasy przedstawia taki opis:

> A persistent wrapper for a java.util.Set. The underlying collection is a HashSet.

Czyli stosując `Set` należy poprawnie zaimplementować metody `equals()` oraz `hashCode()`, natomiast co to znaczy poprawnie w kontekście Hibernate - do tego przejdziemy za moment.

Podsumowanie

Tak jak widzisz, nie ma jednoznacznej odpowiedzi, że `Set` albo `List` jest lepsze. Musimy wybierać z dostępnych możliwości jednocześnie rozumiejąc konsekwencję naszego wyboru. Pamiętaj, że przy lokalnej pracy z Hibernate zawsze warto jest mieć włączone logowanie wykonywanych zapytań, żeby na bieżąco sprawdzać, czy nie popełniliśmy gdzieś jakiegoś błędu i Hibernate zamiast 2 zapytań wykonuje 10. Później przy dołożeniu skali setek tysięcy klientów, takie rzeczy potrafią być zabójcze dla wydajności aplikacji.

Encje a equals() i hashCode()

Wiemy już, że w przypadku pracy z POJO, do porównywania ze sobą obiektów potrzebne jest nadpisanie metod `equals()` oraz `hashCode()` i wiemy również, jaki jest kontrakt pomiędzy tymi metodami. W przypadku pracy z encjami również potrzebne jest nadpisanie tych metod, żeby Hibernate mógł wydajnie pracować z encjami.

Do czego jest to potrzebne? Przy pracy z Hibernate wykorzystujemy instancję `Session`, która jest odpowiedzialna za wczytywanie, przechowywanie i zarządzanie encjami. Jeżeli `Session` będzie wczytywało z bazy danych cały czas ten sam wiersz, to w tej samej sesji nie powinna być tworzona nowa instancja encji. `Session` powinno "wiedzieć", że wczytana została encja, która już istnieje w pamięci programu i encja ta powinna zostać odświeżona o nowe informacje.

Dlatego właśnie nadpisane `equals()` oraz `hashCode()` w encjach jest istotne - `Session` dzięki temu ma być w stanie odróżnić, czy dana encja już jest wczytana do pamięci, czy też nie.

Przejdźmy teraz do przykładu. Tym razem przykład będzie oparty o klasy `Customer` oraz `Address`, które były omawiane już wcześniej.

Klasa Customer

```
package pl.zajavka;
```

```
import jakarta.persistence.*;
import lombok.*;

@Getter
@Setter
@ToString
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "customer")
public class Customer { ①

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "customer_id", unique = true, nullable = false)
    private Integer id;

    // ...

    @OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", unique = true)
    private Address address;

}
```

① Specjalnie nie używam adnotacji `@Data`, żeby nie nadpisywać domyślnych implementacji `equals()` oraz `hashCode()`.

Jeżeli wykonasz teraz poniższy kod:

Metoda testSession() w klasie CustomerRepository

```
public void testSession() {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        session.beginTransaction();

        Integer customerId = 1;
        Customer c1 = session.find(Customer.class, customerId);
        Customer c2 = session.find(Customer.class, customerId);

        System.out.println("c1 == c2: " + (c1 == c2));
        System.out.println("c1.equals(c2): " + (c1.equals(c2)));

        session.getTransaction().commit();
    }
}
```

To na ekranie zostanie wydrukowane:

```
c1 == c2: true
c1.equals(c2): true
```

Czyli w obrębie tej samej sesji, Hibernate jest w stanie wywnioskować, że mamy do czynienia z tym samym obiektem. Spróbujmy teraz zrobić to samo, ale w dwóch różnych sesjach:

Metoda getCustomer() w klasie CustomerRepository

```
Optional<Customer> getCustomer(Integer customerId) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        return Optional.ofNullable(session.find(Customer.class, customerId));
    }
}
```

Klasa ExampleRunner

```
package pl.zajavka;

public class ExampleRunner {

    public static void main(String[] args) {
        CustomerRepository customerRepository = new CustomerRepository();

        Customer c1 = customerRepository.getCustomer(1).orElseThrow();
        Customer c2 = customerRepository.getCustomer(1).orElseThrow();

        System.out.println("c1 == c2: " + (c1 == c2));
        System.out.println("c1.equals(c2): " + (c1.equals(c2)));

        HibernateUtil.closeSessionFactory();
    }
}
```

Jeżeli uruchomisz teraz powyższy przykład, to na ekranie zostanie wydrukowane:

```
c1 == c2: false
c1.equals(c2): false
```

Oznacza to, że Hibernate w dwóch różnych sesjach stworzył dwie instancje tej samej klasy do reprezentacji tego samego wiersza w bazie danych. Jak w takim razie można zaimplementować metody `equals()` oraz `hashCode()`?

Przypomnij sobie, że wiersze w tabeli w bazie danych powinny mieć swoje unikalne klucze (*primary key*). Oprócz tego, wiersze mogą mieć również swoje identyfikatory biznesowe (jakieś wartości, które muszą być unikalne z punktu widzenia domeny biznesowej, np. email użytkownika). Skoro taka wartość świadczy o unikalności rekordu, to możemy z powodzeniem jej użyć do wygenerowania metod `equals()` oraz `hashCode()`. Poniżej przykład:

```
@Getter
@Setter
@ToString
@EqualsAndHashCode(of = "email") ①
@Entity
```

```
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "customer")
public class Customer {
    // ...
}
```

① Jak znamy Lomboka to wystarczy w sumie tyle ☺

Jeżeli uruchomimy teraz przykład z dwiema sesjami ponownie, to na ekranie zostanie wydrukowane:

```
c1 == c2: false
c1.equals(c2): true
```

Podsumowując:

- Jeżeli jesteśmy w obrębie tej samej sesji, pobieranie tej samej encji wielokrotnie, zwróci ten sam obiekt,
- Jeżeli jesteśmy w obrębie innych sesji, pobieranie tej samej encji wielokrotnie może zwrócić inne obiekty, jeżeli nie nadpiszemy poprawnie metod `equals()` oraz `hashCode()`,
- Od tego momentu możemy przyjąć za dobrą praktykę nadpisywanie metod `equals()` oraz `hashCode()` w encjach. Zatem od tego momentu robimy to zawsze. Wcześniej nie było to pokazane, żeby nie komplikować przykładów.

Sortowanie otrzymanych danych

Generalnie to temat sortowania był już ugryziony w sposób rozproszony, ale chciałbym to zebrać w jednym miejscu. Poznaliśmy dwa sposoby na posortowanie zwracanych wyników.

Sposób 1: wykorzystujemy *ORDER BY* w HQL

Fragment metody selectExamples() w klasie OwnerRepository

```
String select4 = "SELECT ow FROM Owner ow ORDER BY ow.email ASC, ow.name DESC";
session.createQuery(select4, Owner.class)
    .getResultList()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Sposób 2: wykorzystujemy adnotację `@OrderBy`

Klasa Owner

```
import jakarta.persistence.OrderBy;

// ...

@OneToMany(fetch = FetchType.EAGER, mappedBy = "owner", cascade = CascadeType.ALL)
@OrderBy("breed ASC")
private List<Pet> pets;
```

Należy tutaj dodać jedną kwestię. Jeżeli chcemy zwracać posortowane dane, to często lepiej jest je posortować na poziomie bazy danych, niż w kodzie w aplikacji. Takie podejście jest często bardziej wydajne.

Mapowanie wyjątków

Jak dotychczas nie skupialiśmy się jakoś wyjątkowo mocno na łapaniu wyjątków, bo przypadki nad którymi pracujemy to były tzw. **happy path**. W tej części notatki przejdziemy do kilku przykładów, które mają zobrazować jak Hibernate pomaga nam w zrozumieniu źródła problemu, który wystąpił podczas wykonywania zapytań na bazie danych.

Przypomnijmy sobie definicję tabeli `customer`. Na kolumnie `email` jest tam dodany klucz unikalny:

```
CREATE TABLE customer
(
    customer_id SERIAL,
    -- ...
    email       VARCHAR(32) NOT NULL,
    -- ..
    UNIQUE (email),
    -- ..
);
```

Oznacza to, że nie możemy mieć w bazie danych dwóch użytkowników z tym samym adresem `email`. Co się stanie, gdy spróbujemy taki zapis wykonać?

Na ekranie drukowany jest taki **stacktrace**:

```
Exception in thread "main" jakarta.persistence.PersistenceException:
    Converting `org.hibernate.exception.ConstraintViolationException` to JPA `PersistenceException` :
        could not execute statement
            at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:165)
            ...
            at pl.zajavka.ExampleRunner.main(ExampleRunner.java:16)
Caused by: org.hibernate.exception.ConstraintViolationException:
    could not execute statement
        at
org.hibernate.exception.internal.SQLStateConversionDelegate.convert(SQLStateConversionDelegate.java:95)
        at
org.hibernate.exception.internal.StandardSQLExceptionConverter.convert(StandardSQLExceptionConverter.java
:56)
        at org.hibernate.engine.jdbc.spi.SqlExceptionHelper.convert(SqlExceptionHelper.java:109)
        ...
        at
org.hibernate.event.internal.AbstractSaveEventListener.performSave(AbstractSaveEventListener.java:192)
        at
org.hibernate.event.internal.AbstractSaveEventListener.saveWithGeneratedId(AbstractSaveEventListener.java
:122)
Caused by: org.hibernate.exception.ConstraintViolationException: could not execute statement
        at
org.hibernate.event.internal.DefaultPersistEventListener.entityIsTransient(DefaultPersistEventListener.ja
va:184)
        ...
        at org.hibernate.internal.SessionImpl.firePersist(SessionImpl.java:733)
```

```
        ... 3 more
Caused by: org.postgresql.util.PSQLException: ERROR: duplicate key value violates unique constraint
"customer_email_key"
    Szczeg�y: Key (email)=(stefan@zajavka.pl) already exists.
        at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2675)
        at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:2365)
        ...
        at
org.hibernate.engine.jdbc.internal.ResultSetReturnImpl.executeUpdate(ResultSetReturnImpl.java:197)
        ... 21 more
Caused by: org.postgresql.util.PSQLException: ERROR: duplicate key value violates unique constraint
"customer_email_key"
```

Widzimy tutaj wypisanych kilka wyjątków:

- `jakarta.persistence.PersistenceException` - unchecked,
- `org.hibernate.exception.ConstraintViolationException` - unchecked, dziedziczy z PersistenceException,
- `org.postgresql.util.PSQLException` - checked.

Wynika to z tego, że Hibernate jest warstwą abstrakcji nad samym JDBC, więc wyjątki wyrzucone przez JDBC są opakowywane przez stosowne wyjątki Hibernate. Możemy to wykorzystać przy obsłudze takiego wyjątku.

Skoro `PSQLException` jest checked, a w dodatku jest on po drodze opakowywany przez wyjątek Hibernate, to nie możemy go bezpośrednio złapać w `catch`, dostaniemy wtedy błąd kompilacji. Możemy natomiast spróbować dostać się do źródła:

```
try {
    // ...
} catch (PersistenceException persistenceException) {
    JDBCException jdbcException = (JDBCException) persistenceException.getCause(); ①
    System.err.println(jdbcException.getSQL()); ②
    System.err.println(jdbcException.getSQLState()); ③
    SQLException sqlException = jdbcException.getSQLException(); ④
    System.err.println(sqlException.getErrorCode()); ⑤
    System.err.println(sqlException.getMessage()); ⑥
    return null;
}
```

① W tym miejscu "odpakowujemy" wyjątek i próbujemy go rzutować na `JDBCException`.

② Cytując dokumentację: *Get the actual SQL statement being executed when the exception occurred.*. W tym przypadku zostało zwrócone: *n/a*.

③ Cytując dokumentację: *Get the X/Open or ANSI SQL SQLState error code from the underlying SQLException.*. W tym przypadku zostało zwrócone: *23505*, a znaczenie tego kodu można sprawdzić tutaj.

④ Pobieramy kolejny wyjątek - `SQLException`.

⑤ Cytując dokumentację: *Retrieves the vendor-specific exception code for this SQLException object.* W tym przypadku zostało zwrócone: *0*.

⑥ To już jest metoda z `Throwable`, która zwraca wiadomość towarzyszącą wyjątkowi. W tym przypadku

zostało zwrócone:

```
ERROR: duplicate key value violates unique constraint "customer_email_key"
Szczeg�y: Key (email)=(stefan@zajavka.pl) already exists.
duplicate key value violates unique constraint "customer_email_key"
```

Po co to wszystko? Często jest tak, że błędy wyrzucane na poziomie JDBC mogą być trudne w odszyfrowaniu, więc Hibernate opakowuje te błędy swoimi odpowiednikami. Jeżeli wiadomość błędu za wiele nam nie mówi, możemy się wtedy "dokopać" do kodu błędu i szukać w dokumentacji jakie jest jego znaczenie.

Trzeba jednak pamiętać, że błędy, które dostajemy, są opakowane przez Hibernate, nie dostajemy błędów typowych dla JDBC, tylko te "opakowane". Możemy jednak dostać się do źródła błędu, żeby pobrać kody, które dadzą nam bardziej szczegółowe informacje o źródle problemu.

Drukowanie parametrów

W dotychczasowych SQL, które były logowane przez Hibernate, nie widzieliśmy parametrów zapytań. Zamiast tego wyświetlały się `?`. Jak można to zmienić? Wystarczy dodać w ustawieniach logowania poniższy wpis:

```
<logger name="org.hibernate.orm.jdbc.bind" additivity="false" level="TRACE">
    <appender-ref ref="CONSOLE"/>
</logger>
```

Pamiętaj jednak, że ta specyficzna nazwa paczki zadziała dla Hibernate w wersji `6.X`. W przypadku poprzednich wersji Hibernate, ustawienie wygląda analogicznie, ale musimy ustawić logowanie na innej paczce.

Podsumowanie

Można powiedzieć, że doszliśmy do końca omawiania JPQL i HQL. Na tym etapie powinieneś/powinnaś mieć już dobre zrozumienie tego czym różni się HQL od SQL, w jakich sytuacjach można go stosować oraz jak radzić sobie ze skomplikowanymi zapytaniami.

Mam też nadzieję, że udało mi się zobrazować, że Hibernate to potężne narzędzie i odkryliśmy właśnie kolejną jego część. Hibernate posiada wiele możliwości, które Ty jako programista/programistka będziesz w swojej pracy wykorzystywać i dlatego warto jest mieć zrozumienie tych mechanizmów z potencjalnymi konsekwencjami ich stosowania.

Native Query

Na etapie wprowadzenia do zapytań w Hibernate, oprócz HQL, zostało wspomniane również **Native Query**. Były to zapytania SQL wykorzystujące wariant SQL, który jest dedykowany do bazy danych, na której pracujemy. Czyli "podstawowe" SQL, które możemy wykonywać na bazie danych. Teraz omówimy ten mechanizm.

Jeżeli kiedykolwiek w pracy z Hibernate usłyszysz stwierdzenie **native query**, będzie się ono odnosiło do wykonania rzeczywistej SQL na bazie danych, ale z pomocą Hibernate. Czyli będą to takie zapytania jakie pisaliśmy przy wykorzystaniu JDBC, ale tym razem zrobimy to przez Hibernate. Zapoznaj się z bardzo podstawowym przykładem native query, który jest umieszczony poniżej.

```
String nativeQueryString = "SELECT * FROM Owner";
NativeQuery<Owner> nativeQuery = session.createNativeQuery(nativeQueryString, Owner.class);
nativeQuery.list()
    .forEach(entity -> System.out.println("###Entity: " + entity));
```

Jeżeli masz ustawione w encji `Owner` parametr *fetch = FetchType.LAZY* i nie wywołujesz nigdzie metody `getPets()`, to na ekranie powinien Ci się wydrukować rezultat podobny do:

```
Hibernate: SELECT * FROM Owner
###Entity: Owner(id=58, name=Robert, surname=Nowacki, phone=+48 589 245 114, email=robert@zajavka.pl)
###Entity: Owner(id=59, name=Adrian, surname=Paczkomat, phone=+48 894 256 331, email=adrian@zajavka.pl)
```

Jeżeli natomiast, wywołasz gdzieś metodę `getPets()`, to Hibernate zacznie wykonywać dodatkowe zapytania, żeby "dociągnąć" informacje o encji Pet.

Jak widać na przykładzie, udało nam się wykonać natywne SQL przy wykorzystaniu Hibernate i jest to kolejny sposób na pobieranie danych. Pojawia się zatem pytanie: *Kiedy tego używać?*. Raczej w mocno skomplikowanych przypadkach, które będą od nas wymagały dużo kombinacji na "surowych" SQL. Inaczej, jeżeli będziemy stosowali tylko native queries to można zadać sobie pytanie: *To po co ten ORM?*

NamedNativeQuery

Tak samo jak w przypadku zapytań HQL, tak samo **native query** mogą być uporządkowane w klasach oznaczanych adnotacją `@Entity`. W celu ustrukturyzowania **native queries**, można wykorzystać adnotację @NamedNativeQuery. Poniżej przykład:

Przykład definicji `@NamedNativeQuery`:

```
import org.hibernate.annotations.NamedNativeQueries;
import org.hibernate.annotations.NamedNativeQuery;

@NamedNativeQueries({
    @NamedNativeQuery(
            name = "Person.findPersonByName",
            query = "SELECT * FROM Person WHERE name = :name",
            resultClass = Person.class
    ),
    // pozostałe
})
@Entity
public class Person {
    // ...
}
```

Przykład użycia `@NamedNativeQuery`*:*

```
import org.hibernate.query.NativeQuery;

// ...

NativeQuery query = session.getNamedNativeQuery("Person.findPersonByName");
query.setParameter("name", "Karol");
Person result = (Person) query.getSingleResult();
```