# Hibernate i Relacje

# Spis treści



# Relacje

Podczas pracy z bazami danych tworzonych jest wiele tabel. W przykładach edukacyjnych, które tutaj poruszamy, pracujemy / będziemy pracować z bazami danych, które mają maksymalnie kilkanaście tabel. W praktyce spotkasz się z bazami danych, które będą miały setki tabel.

Wyjadę teraz z pytaniem, o kwestie, które już wcześniej były widoczne, ale niech będzie 😄. Czy te wszystkie tabele żyją niezależnie od siebie? **NIE!** Tabele są ze sobą związane i takie powiązania pomiędzy tabelami określa się słowem: **relacje**.

We wcześniejszych przykładach (chociażby w projekcie **zajavka store**) widzieliśmy, że tabele są ze sobą łączone przy wykorzystaniu kluczy obcych. Nazywaliśmy to również relacjami. Na etapie omawiania diagramów UML powiedzieliśmy sobie również o rodzajach relacji, czyli: **one-to-one** (*jeden-do-jednego*), **one-to-many** (*jeden-do-wielu*) oraz **many-to-many** (*wiele-do-wielu*). Mówiliśmy o tych relacjach w

kontekście obiektów. Teraz dodamy do tego, że relacje takie mogą być **jednokierunkowe** lub **dwukierunkowe**. Przejdziemy do omówienia tych relacji w kontekście tabel w bazach danych oraz mapowania tych relacji w encjach. W pierwszej kolejności natomiast przypomnijmy sobie, na czym polegała każda z tych relacji.

# Najpierw teoretycznie

## one-to-one (*jeden-do-jednego*)

Ten rodzaj relacji mówi nam, że jedna encja A może być powiązana z co najwyżej jedną encją B i odwrotnie. W odniesieniu do baz danych będzie to oznaczało, że rekord w jednej encji (tabeli) jest powiązany z dokładnie jednym rekordem w innej encji (tabeli).

Przykłady relacji **one-to-one**:

- jeden człowiek może mieć tylko jedno serce i jedno serce może znajdować się w tym samym człowieku,
- każdy kraj ma dokładnie jedną stolicę. Każda stolica jest stolicą dokładnie jednego kraju,
- w przypadku wielu stron internetowych jeden adres e-mail jest powiązany z dokładnie jednym kontem użytkownika, a każde konto użytkownika jest identyfikowane za pomocą jednego adresu e-mail,

Dla lepszego zrozumienia przykładu, które relacje nie są **one-to-one**?

- każde miasto znajduje się dokładnie w jednym kraju, ale większość krajów ma wiele miast,
- każdy pracownik ma dokładnie jednego bezpośredniego przełożonego lub kierownika, ale każdy kierownik zazwyczaj nadzoruje wielu pracowników,
- każdy nauczyciel w szkole uczy wielu uczniów, do tego każdy uczeń może być uczony przez wielu nauczycieli.

## one-to-many (*jeden-do-wielu*)

Relacja **one-to-many** odnosi się do relacji między dwiema encjami A i B, w której instancja A może być połączona z wieloma instancjami B, ale instancja B jest połączona tylko z jedną instancją A. W odniesieniu do baz danych będzie to oznaczało, że rekord w jednej encji (tabeli) jest powiązany z wieloma rekordami w innej encji (tabeli).

Przykłady relacji **one-to-many**:

- jedno mieszkanie może mieć wiele łazienek, ale łazienka może znajdować się tylko w jednym mieszkaniu,
- kraj może mieć wiele miast, ale jedno miasto znajduje się dokładnie w jednym kraju,
- każdy właściciel może mieć wiele zwierząt, ale jedno zwierze może mieć tylko jednego właściciela.

Dla lepszego zrozumienia przykładu, które relacje nie są **one-to-many**?

- klient może kupić w sklepie wiele produktów i produkty w sklepie mogą być kupione przez wielu klientów,

- książka może mieć wielu autorów i autor może napisać wiele książek,
- komputer może mieć jeden procesor, ale ten procesor może być tylko w jednym komputerze.

## many-to-many (*wiele-do-wielu*)

Relacja **many-to-many** odnosi się do relacji między dwiema encjami A i B, w której A może być powiązane z wieloma wystąpieniami B i na odwrót.

Jeżeli chodzi o stworzenie tej relacji w bazie danych, to sytuacja taka jest specyficzna. Podczas omawiania diagramów UML i relacji, padło takie stwierdzenie:

> W momencie, gdy będziemy rozmawiać o sytuacjach wiele do wielu, w praktyce wprowadza się klasę, która sama w sobie oznacza taką relację. Czyli możemy mieć np. `Samochód` i `Właściciela`. Samochód może mieć wielu właścicieli, a właściciel może mieć wiele samochodów. W takim przypadku możemy wprowadzić klasę `Ownership`.

W przypadku tworzenia tabel w bazach danych, rozwiązanie relacji **many-to-many** może wyglądać analogicznie. Możemy wprowadzić wtedy trzecią tabelę, która oznacza taką relację sama w sobie.

Jeżeli przykładowo tworzylibyśmy sklep internetowy i mielibyśmy tam dwie tabele: `product` i `customer`, to chcąc stworzyć relację **many-to-many** (produkt może być zakupiony przez wielu klientów i klient może kupić wiele produktów), moglibyśmy to rozwiązać wprowadzając trzecią tabelę `purchase`. Tabela `purchase` "spinałaby" wtedy informację o tym, który klient dokonał zakupu jakiego produktu i w jakiej ilości.

Przykłady relacji **many-to-many**:

- na jednej płycie DVD może być nagranych wiele filmów i film może być nagrany na wielu płytach DVD,
- mechanik może naprawiać wiele samochodów i samochód może być naprawiany przez wielu mechaników,
- każdy nauczyciel w szkole uczy wielu uczniów, do tego każdy uczeń może być uczony przez wielu nauczycieli.

Dla lepszego zrozumienia przykładu, które relacje nie są **many-to-many**?

- w autobusie może być wiele osób, ale osoba może być w jednym autobusie,
- na suszarce może wisieć wiele koszulek, ale koszulka może wisieć tylko na jednej suszarce,
- na palcu jest tylko jeden paznokieć i paznokieć może być tylko na jednym palcu.

## Unidirectional vs Bidirectional relation

Jaka jest różnica pomiędzy relacją jednokierunkową i dwukierunkową? Główną różnicą jest to, że relacja dwukierunkowa zapewnia nawigację w obu kierunkach relacji, czyli w prostych słowach, rodzic wie o swoim dziecku, a dziecko wie o swoim rodzicu. Jeżeli relacja będzie jednokierunkowa, będzie to oznaczało, że rodzic wie o swoim dziecku, ale dziecko o swoim rodzicu już nie.

Wiem, że może brzmieć to strasznie, ale nie mówimy tu o dziecku i rodzicu w

> kontekście rodzinnym, tylko o dziecku i rodzicu w kontekście relacji pomiędzy encjami.

Cytując link:

> Prefer bidirectional associations:
> Unidirectional associations are more difficult to query. In a large application, almost all associations must be navigable in both directions in queries.

Oczywiście mogą wystąpić sytuacje, gdzie domena biznesowa będzie od nas wymagała, żeby relacja była jednokierunkowa, ale w dużych aplikacjach, wygodniej jest stosować relacje dwukierunkowe.

To ile i jakich relacji będziemy mieli w bazie danych wynika z tego, jaką domenę biznesową potrzebujemy odwzorować w kodzie. Jeżeli chodzi o przykłady tych relacji w postaci tabel i encji, już do tego przechodzimy.

# Relacje w praktyce

Chcę wyraźnie zaznaczyć jedną kwestię. Na razie nie skupiamy się na tym, czy wykonywane zapytania na bazie są optymalne. Nie skupiamy się na wydajności tych zapytań, ani na tym, czy jest ich za dużo, czy mogłoby być ich wykonywanych mniej. Na tym etapie skupiamy się tylko i wyłącznie na samej istocie relacji i mapowania encji. Do poprawy wydajności przejdziemy później.

## one-to-one

Zaczniemy od praktycznego omówienia relacji **one-to-one**. Przypomnijmy, że:

- relacja **one-to-one** oznaczała, że rekord w jednej encji (tabeli) jest powiązany z dokładnie jednym rekordem w innej encji (tabeli),
- powyższa relacja między tabelami zostanie zaimplementowana przy wykorzystaniu klucza obcego (*foreign key*), który odwołuje się do klucza głównego (*primary key*) w innej tabeli,

W pierwszej kolejności skupimy się na relacji jednokierunkowej, później przejdziemy do relacji dwukierunkowej.

### Jednokierunkowe one-to-one

Relacja jednokierunkowa oznacza, że tylko jedna strona (strona posiadająca) jest w stanie nawigować do relacji. W poniższym przykładzie będzie to oznaczało, że tylko klient będzie mógł odnieść się do adresu, ale adres do klienta już nie.

Istnieje kilka sposobów na przedstawienie relacji **one-to-one** w bazie danych, natomiast na potrzeby tego materiału relacja zostanie przedstawiona przy wykorzystaniu kluczy obcych. W relacji **one-to-one** klucz obcy tworzony jest w encji właściciela. Dlatego kolumna `address_id` jest tworzona w tabeli `customer`.

*Tabela* address*:*

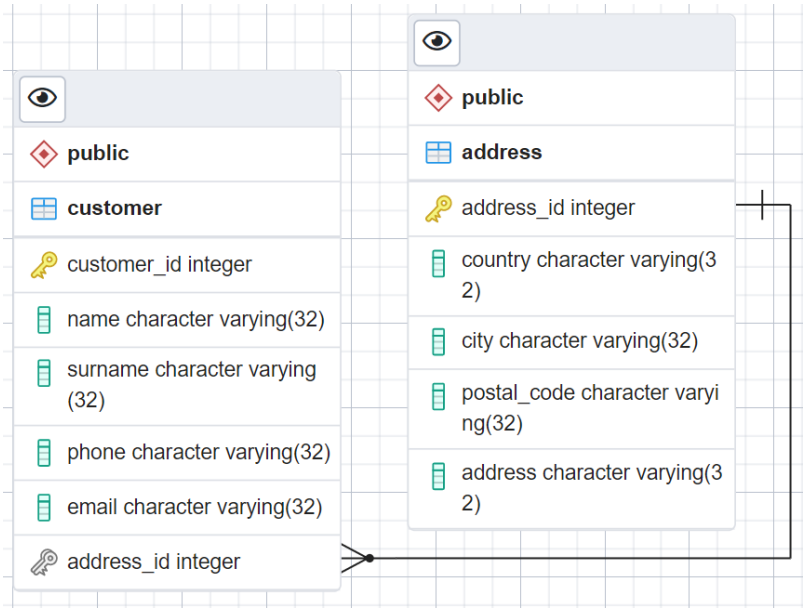
```
CREATE TABLE address
(
    address_id  SERIAL NOT NULL,
    country     VARCHAR(32) NOT NULL,
    city        VARCHAR(32) NOT NULL,
    postal_code VARCHAR(32) NOT NULL,
    address     VARCHAR(32) NOT NULL,
    PRIMARY KEY (address_id)
);
```

*Tabela* customer*:*

```
CREATE TABLE customer
(
    customer_id SERIAL NOT NULL,
    name        VARCHAR(32) NOT NULL,
    surname     VARCHAR(32) NOT NULL,
    phone       VARCHAR(32) NOT NULL,
    email       VARCHAR(32) NOT NULL,
    address_id  INT         NOT NULL,
    PRIMARY KEY (customer_id),
    UNIQUE (email),
    UNIQUE (address_id),
    CONSTRAINT fk_customer_address
        FOREIGN KEY (address_id)
            REFERENCES address (address_id)
);
```

Jeżeli po stworzeniu tych tabel, przejdziesz do pgAdmin, klikniesz prawym przyciskiem myszy na nazwie Twojej bazy danych (np. *zajavka*) i wybierzesz opcję **Generate ERD**, to zobaczysz wtedy na ekranie diagram Twoich tabel w tej bazie danych.

Skrót ERD oznacza **E**ntity **R**elationship **D**iagram, czyli *Diagram Związków Encji*. W Internecie znajdziesz wiele konwencji rysowania tego Diagramu, podobnie jak w przypadku UML. W kolejnych materiałach będziemy generować taki diagram przy wykorzystaniu pgAdmin. Wrócimy do tego tematu podczas omawiania zagadnień praktycznych.




*Obraz 1. ERD Diagram dla tabel `address` i `customer`*

Jeżeli chodzi o **Diagramy ERD**, to na razie będą one pokazywane przy kolejnych przykładach i będziemy je omawiać wybiórczo. Do szerszego omówienia Diagramów ERD przejdziemy później.

Narzędzie `pgAdmin` wygenerowało nam powyższy diagram. Pozwala on zobrazować relację pomiędzy encjami w bazie danych oraz w tym przypadku przedstawia nam szczegółowe informacje o kolumnach w tabeli oraz o relacjach pomiędzy tabelami. Na powyższym diagramie widzimy nazwy tabeli, kolumny, klucze główne, klucze obce oraz relacje pomiędzy tabelami.

Ważne są tutaj oznaczenia przy strzałkach i należy tutaj zwrócić uwagę na pewną kwestię. Jeżeli spojrzysz w `pgAdmin` na górze ekranu na poniższe ikonki:


*Obraz 2. pgAdmin One-To-Many oraz Many-to-Many*

To zauważysz, że nie ma tam dostępnej relacji **one-to-one**. O powód tej decyzji należy zapytać twórców, a tutaj możesz zapoznać się z dokumentacją. Jednakże ma to wpływ na generowanie Diagramu ERD. Na naszym diagramie relacja pomiędzy `customer` i `address` jest przedstawiona za pomocą takiego oznaczenia:

*Obraz 3. pgAdmin oznaczenie relacji One-To-Many*

Zgodnie ze standardami Diagramów ERD, jest to oznaczenie relacji **one-to-many**. Pojedyncza pionowa kreska powinna znajdować się po stronie **one**, natomiast znaczek przypominający trójkąt po stronie **many**. Czyli w powyższym przykładzie można to zrozumieć w ten sposób, że *ten sam adres może być wykorzystany przez wielu klientów*. Patrząc na same tabele, jest to w sumie logiczne, bo wiele rekordów w tabeli `customer` może odnosić się do tego samego adresu w tabeli `address` (w przedstawionej wcześniej

*sql* nie za bardzo, bo został narzucony *unique key*). Dodam też w ramach ciekawostki, że nawet jeżeli narzucimy klucz unikalny na `address_id` w tabeli `customer`, żeby jeden adres mógł być wykorzystany tylko raz, to `pgAdmin` nadal generuje diagram ze strzałką z relacji `one-to-many`. Dlatego trzeba pamiętać, że narzędzia mogą różne kwestie implementować na różny sposób.

Zgodnie ze standardem Diagramów ERD, oznaczenie **one-to-one** powinno wyglądać w ten sposób:

*Obraz 4. Oznaczenie relacji One-To-One*

Czemu to takie pokręcone? Z perspektywy samych tabel w bazie danych, jeżeli łączymy tabele przy wykorzystaniu kluczy obcych, to analogicznie będą wyglądały relacje **one-to-one** oraz **one-to-many** (w kontekście kluczy obcych). Stanie się to bardziej jasne, gdy przejdziemy do przykładu **one-to-many**.

Wracamy do Hibernate. Zdefiniowaliśmy już tabele `customer` i `address`. W jaki sposób można teraz przygotować mapowanie encji:

*Klasa Customer*

```
package pl.zajavka;

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

@Data
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "customer")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="customer_id", unique = true, nullable = false) ①
    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;

    @OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL) ②
    @JoinColumn(name = "address_id", unique = true) ③
    private Address address; ④

}
```

① Zostały tutaj wykorzystane parametry mówiące o tym, że klucz główny jest unikalny: *unique* = *true* oraz, że wartość nie może być `null`: *nullable* = *false*. Wykorzystanie tych zapisów na poziomie encji daje nam dwie rzeczy:

- jeżeli generujemy DDL przy wykorzystaniu Hibernate (ustawienie *hibernate.hbm2ddl.auto*), to jest to informacja, w jaki sposób Hibernate ma generować definicje kolumn,
- w przypadku *nullable* = *false* uzyskujemy dodatkową walidację (sprawdzenie) na poziomie kodu przed wykonaniem faktycznego INSERT lub UPDATE w bazie danych. Hibernate wie wtedy już na poziomie kodu, że wartości są `null` i nie musi wykonywać wtedy zbędnych zapytań na bazie. Musimy tylko wtedy ustawić w Hibernate opcję `hibernate.check_nullability` na `true`, domyślnie jest `false`, źródło.

② W tym miejscu faktycznie stosujemy adnotację `@OneToOne`. Służy ona do oznaczenia relacji **one-to-one**. Do parametrów `fetch` i `cascade` wrócimy później.

③ Adnotacja ta określa nazwę klucza obcego, na podstawie którego robiony jest join w bazie danych.

④ Odwołujemy się tutaj do encji w postaci klasy.

*Klasa Address*

```
package pl.zajavka;

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

@Data
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "address")
public class Address { ①

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

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

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

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

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

}
```

① Zwróć uwagę, że encja `Address` nie ma pojęcia o encji `Customer`. Właśnie dlatego nazywamy tę relację jednokierunkową. Z poziomu encji `Address` nie odwołamy się do encji `Customer`.

Pominiemy kod tworzący konfigurację Hibernate, gdyż była ona powtarzana wielokrotnie. Skupmy się

natomiast na wywołaniu kilku zapytań na bazie danych.

*Klasa CustomerRepository*

```
package pl.zajavka;

import org.hibernate.Session;

import java.util.*;

public class CustomerRepository {

    Customer insertCustomer(final Customer customer) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            session.persist(customer);
            session.getTransaction().commit();
            return customer;
        }
    }

    List<Customer> listCustomers() {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            String query = "SELECT cust FROM Customer cust";
            List<Customer> customers = session.createQuery(query, Customer.class).list();
            session.getTransaction().commit();
            return customers;
        }
    }

    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));
        }
    }

    void updateCustomer(Integer customerId, Address newAddress) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            Customer customer = session.find(Customer.class, customerId);
            customer.setAddress(newAddress);
            session.getTransaction().commit();
        }
    }

    void deleteCustomer(Integer customerId) {
        try (Session session = HibernateUtil.getSession()) {
```

```
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            session.remove(session.find(Customer.class, customerId));
            session.getTransaction().commit();
        }
    }

    void deleteAll() {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            String query = "select cust from Customer cust";
            session.createQuery(query, Customer.class).list().forEach(session::remove);
            session.getTransaction().commit();
        }
    }
}
```

*Klasa ExampleData*

```
package pl.zajavka;

class ExampleData {

    static Address someAddress1() {
        return Address.builder().country("Poland").city("Szczecin")
            .postalCode("70-112").address("Witolda Starkiewicza 3").build();
    }

    static Address someAddress2() {
        return Address.builder().country("Poland").city("Gdynia")
            .postalCode("81-357").address("3 maja 16").build();
    }

    static Customer someCustomer1() {
        return Customer.builder().name("Stefan").surname("Nowacki").phone("+48 589 245 114")
            .email("stefan@zajavka.pl").address(someAddress1()).build();
    }

    static Customer someCustomer2() {
        return Customer.builder().name("Adrian").surname("Paczkomat").phone("+48 894 256 331")
            .email("adrian@zajavka.pl").address(someAddress2()).build();
    }
}
```

*Klasa ExampleRunner*

```
package pl.zajavka;

public class ExampleRunner {

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

```
        customerRepository.deleteAll();

        Customer customer1 = customerRepository.insertCustomer(ExampleData.someCustomer1());
        Customer customer2 = customerRepository.insertCustomer(ExampleData.someCustomer2());

        customerRepository.listCustomers()
            .forEach(customer -> System.out.println("###Customer listing: " + customer));

        System.out.println("###Customer1: " + customerRepository.getCustomer(customer1.getId()));
        System.out.println("###Customer2: " + customerRepository.getCustomer(customer2.getId()));

        Address newAddress = Address.builder().country("Poland").city("Sopot")
            .postalCode("81-713").address("Młyńska 2").build();
        customerRepository.updateCustomer(customer1.getId(), newAddress);
        System.out.println("###Customer update: " + customerRepository.getCustomer(customer1.getId()));

        customerRepository.listCustomers()
            .forEach(customer -> System.out.println("###Customer listing: " + customer));

        customerRepository.deleteCustomer(customer2.getId());

        customerRepository.listCustomers()
            .forEach(customer -> System.out.println("###Customer listing: " + customer));

        HibernateUtil.closeSessionFactory();
    }
}
```

Na podstawie przedstawionego przykładu możesz spróbować zakomentować fragmenty kodu i wykonywać oddzielnie INSERT, UPDATE, DELETE itp. Będziesz wtedy w stanie dokładniej obserwować wykonywane zapytania.

Jeżeli zastanawiasz się czemu Hibernate wykonuje tyle zapytań - wrócimy do tego.

### Dwukierunkowe one-to-one

Skoro został wcześniej zacytowany fragment z dokumentacji mówiący, że lepiej jest tworzyć mapowania dwukierunkowe, to przejdźmy teraz właśnie do takiego. Poniższy przykład będzie rozwinięciem przykładu jednokierunkowego one-to-one. Zakładamy, że pliki, które nie zostaną niżej przedstawione, nie ulegają zmianie. Zacznijmy od klasy `Customer`, kod mapowań w tej klasie się nie zmienia, ale wyjątkowo ta klasa jest powtórzona, żeby pomóc Ci utrwalić wiedzę.

*Klasa Customer*

```
package pl.zajavka;

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

@Data
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "customer")
public class Customer {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="customer_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;

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

}
```

Zmienia się natomiast zapis w klasie `Address`:

*Klasa Address*

```
package pl.zajavka;

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

@Data
@ToString(exclude = "customer")
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "address")
public class Address {

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

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

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

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

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

    @OneToOne(fetch = FetchType.EAGER, mappedBy = "address", cascade = CascadeType.ALL)
```

```
    private Customer customer; ①

}
```

① Dwukierunkowość relacji objawia się w tym miejscu. Podajemy tutaj nazwę pola z klasy `Customer`.

Wcześniej, gdy relacja była jednokierunkowa, oznaczało to, że klasa `Address` nie ma zielonego pojęcia o klasie `Customer`. Teraz z racji, że tworzymy relację dwukierunkową, to taką "świadomość" dodaliśmy.

Co oznacza `mappedBy`? Cytując dokumentację:

> (Optional) The field that owns the relationship. This element is only specified on the inverse (non-owning) side of the association.

Czyli ten parametr jest dodawany po stronie (*non-owning*). Możemy na tej podstawie zrozumieć, że "właścicielem" relacji *Customer - Address* jest *Customer* i *Customer* "posiada" relację przy wykorzystaniu pola `address`.

Jeżeli teraz posłużysz się debuggerem, to zobaczysz, że zarówno encja `Customer` wskazuje na powiązane encje `Address` i encje `Address` wskazują na powiązane encje `Customer`.

## one-to-many

Przechodzimy do praktycznego omówienia relacji **one-to-many**. Cytując link:

> A one-to-many relationship is the most common relationship found between tables in a relational database.

I w sumie coś w tym jest. W przypadku relacji **one-to-many**, przejdziemy od razu od omówienia relacji dwukierunkowej. Posłużymy się przykładem: *każdy właściciel może mieć wiele zwierząt, ale jedno zwierze może mieć tylko jednego właściciela.*

Istnieje kilka sposobów na przedstawienie relacji **one-to-many** w bazie danych, natomiast na potrzeby tego materiału relacja zostanie przedstawiona przy wykorzystaniu kluczy obcych. W relacji **one-to-many** klucz obcy z tabeli `pet` będzie odnosić się do klucza głównego w tabeli `owner`. W ten sposób można powiązać wiele zwierząt z jednym właścicielem, ale nie można powiązać wielu właścicieli z jednym zwierzęciem. Oczywiście, żeby działało to poprawnie to zwierze musi wskazywać na jakąś wartość z właściciela - stąd wskazujemy na klucz główny.

*Tabela `owner`:*

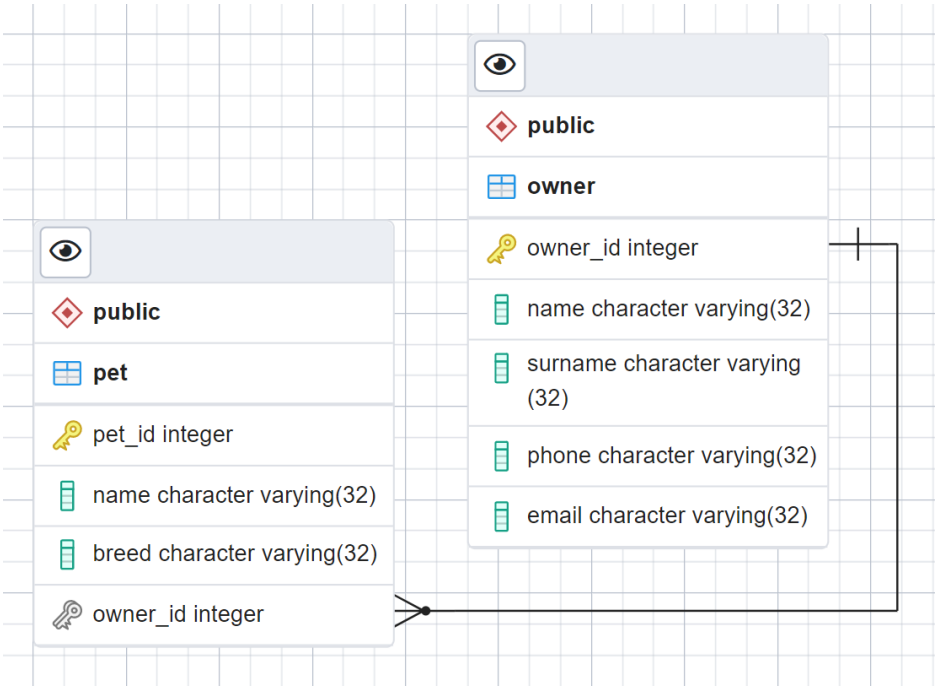
```
CREATE TABLE owner
(
    owner_id SERIAL NOT NULL,
    name     VARCHAR(32) NOT NULL,
    surname  VARCHAR(32) NOT NULL,
    phone    VARCHAR(32) NOT NULL,
    email    VARCHAR(32) NOT NULL,
    PRIMARY KEY (owner_id)
);
```

*Tabela* `pet`*:*

```
CREATE TABLE pet
(
    pet_id   SERIAL NOT NULL,
    name     VARCHAR(32) NOT NULL,
    breed    VARCHAR(32) NOT NULL,
    owner_id INT         NOT NULL,
    PRIMARY KEY (pet_id),
    CONSTRAINT fk_pet_owner
        FOREIGN KEY (owner_id)
            REFERENCES owner (owner_id)
);
```

Przypomnij sobie, że jeżeli po stworzeniu tych tabel, przejdziesz do `pgAdmin`, klikniesz prawym przyciskiem myszy na nazwie Twojej bazy danych (np. *zajavka*) i wybierzesz opcję **Generate ERD**, to zobaczysz wtedy na ekranie diagram Twoich tabel w tej bazie danych.

Spójrzmy teraz na Diagram ERD dla stworzonej relacji.




*Obraz 5. ERD Diagram dla tabel* `owner` *i* `pet`

Widzimy te same oznaczenia co wcześniej, tyle że tym razem wygenerowany Diagram przedstawia relację zgodnie z "oczekiwaniami". Pojedyncza pionowa kreska powinna znajdować się po stronie **one**, natomiast znaczek przypominający trójkąt po stronie **many**. Czyli w powyższym przykładzie można to zrozumieć w ten sposób, że *ten sam właściciel może posiadać wiele zwierząt*. Inaczej mówiąc, znaczek w kształcie trójkąta jest tam, gdzie klucz obcy.

Wracamy do Hibernate. Zdefiniowaliśmy już tabele `owner` i `pet`. W jaki sposób można teraz przygotować mapowanie encji:

*Klasa Owner*

```
package pl.zajavka;

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

import java.util.Set;

@Getter
@Setter ①
@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.EAGER, 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() + ")";
    }
}
```

① Zwróć uwagę, że nie używamy adnotacji `@Data`, tylko bezpośrednio `@Getter` i `@Setter`. Zastosowanie `@Data` prowadzi do powstania `java.lang.StackOverflowError`. Wynika to z cyklicznej zależności, jaką wprowadza w tym przypadku automatycznie generowana metoda `toString()`.

*Klasa Pet*

```
package pl.zajavka;

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

@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) ①
    @Column(name = "breed")
    private Breed breed;

    @ManyToOne(fetch = FetchType.LAZY) ②
    @JoinColumn(name = "owner_id")
    private Owner owner;

    @Override
    public String toString() {
        return "Pet(id=" + this.getId() + ", name=" + this.getName()
            + ", breed=" + this.getBreed() + ")"; ③
    }
}
```

① Jeżeli chcemy mapować enum na String to wystarczy dodać adnotację `@Enumerated(EnumType.STRING)`.

② `@ManyToOne` może być rozumiane jako relacja **one-to-many**, ale od drugiej strony. Zasada jest ta sama, wiele zwierząt może przynależeć do tego samego właściciela.

③ W metodzie `toString()` nie dodajemy wywołania `getOwner()`, gdyż wprowadziłoby to cykliczną zależność i doprowadziło do `StackOverflowError`.

*Enum Breed*

```
package pl.zajavka;

public enum Breed {
    CAT, DOG, MONKEY
}
```

Przypomnę, że parametry `fetch` oraz `cascade` omówimy później.

Tym razem konfiguracja Hibernate, dla przykładów edukacyjnych, będzie wyglądała lekko inaczej.

*Konfiguracja Hibernate w klasie `HibernateUtil`:*

```
package pl.zajavka;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
```

```
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Environment;
import org.hibernate.service.ServiceRegistry;

import java.util.Map;

public class HibernateUtil {

    private static final Map<String, Object> SETTINGS = Map.ofEntries(
        Map.entry(Environment.DRIVER, "org.postgresql.Driver"),
        Map.entry(Environment.URL, "jdbc:postgresql://localhost:5432/zajavka"),
        Map.entry(Environment.USER, "postgres"),
        Map.entry(Environment.PASS, "postgres"),
        Map.entry(Environment.DIALECT, "org.hibernate.dialect.PostgreSQLDialect"),
        Map.entry(Environment.HBM2DDL_AUTO, "none"),
        Map.entry(Environment.SHOW_SQL, true),
        Map.entry(Environment.FORMAT_SQL, false)
    );

    private static final SessionFactory sessionFactory = loadSessionFactory();

    private static SessionFactory loadSessionFactory() {
        try {
            ServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()
                .applySettings(SETTINGS)
                .build();

            Metadata metadata = new MetadataSources(standardRegistry)
                .addAnnotatedClass(Pet.class)
                .addAnnotatedClass(Owner.class)
                .getMetadataBuilder()
                .build();

            return metadata.getSessionFactoryBuilder().build();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    static void closeSessionFactory() {
        try {
            sessionFactory.close();
        } catch (Throwable ex) {
            System.err.println("Exception while closing SessionFactory: " + ex);
        }
    }

    static Session getSession() {
        try {
            return sessionFactory.openSession();
        } catch (Throwable ex) {
            System.err.println("Exception while getting Session: " + ex);
        }
        return null;
    }
}
```

Skupmy się następnie na wywołaniu kilku zapytań na bazie danych.

*Klasa OwnerRepository*

```
package pl.zajavka;

import org.hibernate.Session;

import java.util.*;

public class OwnerRepository {

    Owner insertData(final Owner owner, final Set<Pet> pets) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            owner.setPets(pets);
            pets.forEach(pet -> pet.setOwner(owner));
            session.persist(owner);
            session.getTransaction().commit();
            return owner;
        }
    }

    List<Owner> listOwners() {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            String query = "SELECT owner FROM Owner owner";
            List<Owner> owners = session.createQuery(query, Owner.class).list();
            session.getTransaction().commit();
            return owners;
        }
    }

    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));
        }
    }

    void updateOwner(Integer ownerId, Pet newPet) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            Owner owner = session.find(Owner.class, ownerId);
            owner.getPets().add(newPet);
            session.getTransaction().commit();
        }
    }

    void deleteOwner(Integer ownerId) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
```

```
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            session.remove(session.find(Owner.class, ownerId));
            session.getTransaction().commit();
        }
    }

    void deleteAll() {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            String query = "select owner from Owner owner";
            session.createQuery(query, Owner.class).list().forEach(session::remove);
            session.getTransaction().commit();
        }
    }
}
```

*Klasa ExampleData*

```
package pl.zajavka;

class ExampleData {

    static Owner someOwner1() {
        return Owner.builder().name("Stefan").surname("Nowacki")
            .phone("+48 589 245 114").email("stefan@zajavka.pl").build();
    }

    static Owner someOwner2() {
        return Owner.builder().name("Adrian").surname("Paczkomat")
            .phone("+48 894 256 331").email("adrian@zajavka.pl").build();
    }

    static Pet somePet1() {
        return Pet.builder().name("Fafik").breed(Breed.DOG).build();
    }

    static Pet somePet2() {
        return Pet.builder().name("Kiciek").breed(Breed.CAT).build();
    }

    static Pet somePet3() {
        return Pet.builder().name("Szymek").breed(Breed.MONKEY).build();
    }

    static Pet somePet4() {
        return Pet.builder().name("Gucio").breed(Breed.DOG).build();
    }
}
```

*Klasa ExampleRunner*

```
package pl.zajavka;
```

```
import java.util.Set;

public class ExampleRunner {

    public static void main(String[] args) {
        OwnerRepository ownerRepository = new OwnerRepository();

        ownerRepository.deleteAll();

        Owner owner1 = ownerRepository.insertData(
            ExampleData.someOwner1(),
            Set.of(ExampleData.somePet1(), ExampleData.somePet2()));
        Owner owner2 = ownerRepository.insertData(
            ExampleData.someOwner2(),
            Set.of(ExampleData.somePet3(), ExampleData.somePet4()));

        ownerRepository.listOwners()
            .forEach(owner -> System.out.println("###Owner listing: " + owner));

        System.out.println("###Owner1: " + ownerRepository.getOwner(owner1.getId()));
        System.out.println("###Owner2: " + ownerRepository.getOwner(owner2.getId()));

        Pet newPet = Pet.builder().name("Drapek").breed(Breed.MONKEY).owner(owner1).build();
        ownerRepository.updateOwner(owner1.getId(), newPet);
        System.out.println("###Owner updated: " + ownerRepository.getOwner(owner1.getId()));

        ownerRepository.listOwners()
            .forEach(owner -> System.out.println("###Owner listing: " + owner));

        ownerRepository.deleteOwner(owner2.getId());

        ownerRepository.listOwners()
            .forEach(owner -> System.out.println("###Owner listing: " + owner));

        HibernateUtil.closeSessionFactory();
    }
}
```

Na podstawie przedstawionego przykładu możesz spróbować zakomentować fragmenty kodu i wykonywać oddzielnie INSERT, UPDATE, DELETE itp. Będziesz wtedy w stanie dokładniej obserwować wykonywane zapytania.

Podsumowując przedstawiony przykład, możemy zaobserwować na konkretnym przykładzie relację **one-to-many**, czyli *jedno zwierze może mieć tylko jednego właściciela, ale jeden właściciel może mieć wiele zwierząt*. Tak jak zaznaczyliśmy wcześniej, nie będziemy poświęcać czasu na omówienie przypadku jednokierunkowego **one-to-many**.

### Aktualizacja listy encji podrzędnych

Skupmy się jeszcze na chwilę nad przypadkiem metody `updateOwner()`, która była przedstawiona wcześniej w klasie `OwnerRepository`. Metoda ta wyglądała w ten sposób:

*Metoda updateOwner() z klasy OwnerRepository*

```
void updateOwner(Integer ownerId, Pet newPet) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
```

```
            throw new RuntimeException("Session is null");
        }
        session.beginTransaction();
        Owner owner = session.find(Owner.class, ownerId);
        owner.getPets().add(newPet); ①
        session.getTransaction().commit();
    }
}
```

① Zwróć uwagę, że podczas aktualizacji listy encji podrzędnych (dodaliśmy nowe zwierzątko) nie wykonujemy operacji `session.persist()` dla zwierzątek. Hibernate sam wychwyci, że do listy encji podrzędnych (zwierzątka) zostało dodane nowe zwierzątko i wykona stosowny `insert` w bazie danych.

Tak samo zadziała to w momencie, gdybyśmy np. chcieli zmienić imię właściciela. Wystarczy, że wykonamy odpowiedni `setName()` i nie musimy wołać `session.persist()` ponownie. Hibernate sam wychwyci zmianę stanu obiektu i wykona odpowiednie zapytanie na bazie danych.

## many-to-many

Przechodzimy do praktycznego omówienia relacji **many-to-many**. Przypomnijmy, że relacja **many-to-many** odnosi się do relacji między dwiema encjami A i B, w której A może być powiązane z wieloma wystąpieniami B i na odwrót. Za przykład weźmy sytuację, gdzie pracownik może pracować nad projektami w firmie. Wielu pracowników może pracować nad jednym projektem i jednocześnie jeden pracownik może pracować nad wieloma projektami.

Gdy chcemy utworzyć relację **many-to-many** między dwiema lub większą liczbą tabel, najprostszym sposobem jest użycie tzw. *junction table*. Tabela taka inaczej może być nazywana tabelą pomostową lub tabelą asocjacyjną i łączy ona ze sobą tabele, odwołując się do kluczy głównych każdej tabeli w relacji. Tabela zawiera wpisy oznaczające relację samą w sobie. Nie jest to jedyny możliwy sposób na zdefiniowanie relacji many-to-many w bazie danych, ale na potrzeby tych materiałów skorzystamy z tego. Spójrzmy na przykład wspomnianych pracowników i projektów.

*Tabela* `employee`*:*

```
CREATE TABLE employee
(
    employee_id SERIAL                      NOT NULL,
    name        VARCHAR(20)                 NOT NULL,
    surname     VARCHAR(20)                 NOT NULL,
    salary      NUMERIC(7, 2)               NOT NULL,
    since       TIMESTAMP WITH TIME ZONE NOT NULL,
    PRIMARY KEY (employee_id)
);
```

*Tabela* `project`*:*

```
CREATE TABLE project
(
    project_id  SERIAL                      NOT NULL,
    name        VARCHAR(64)                 NOT NULL,
    description TEXT                        NOT NULL,
    deadline    TIMESTAMP WITH TIME ZONE NOT NULL,
    PRIMARY KEY (project_id),
```

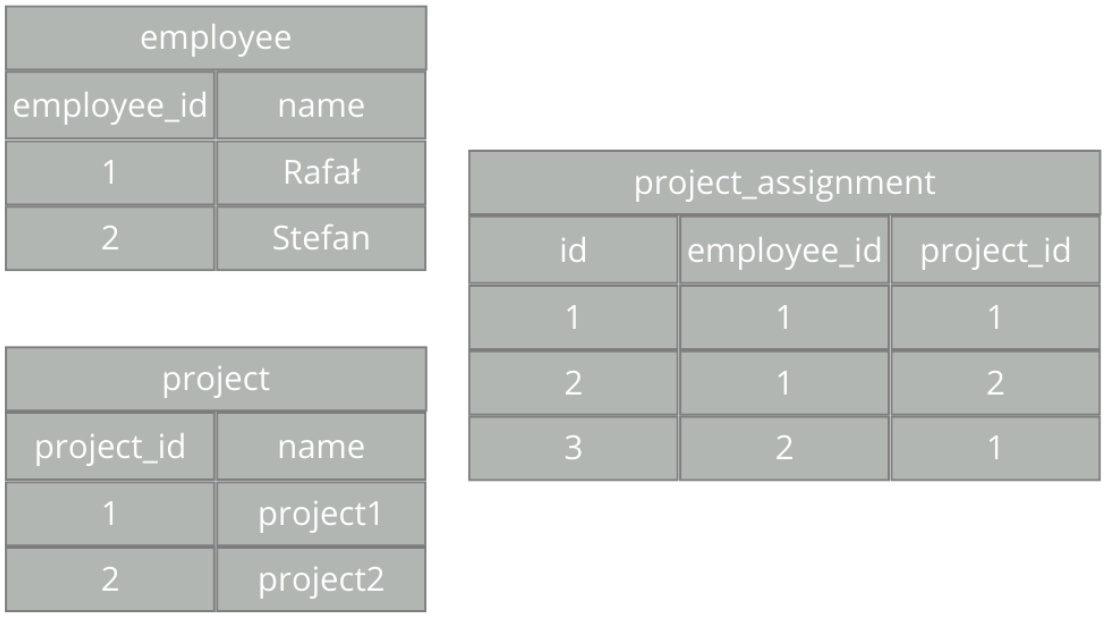
```
    UNIQUE (name)
);
```

I tabela oznaczająca relację samą w sobie:

*Tabela* `project_assignment`*:*

```
CREATE TABLE project_assignment
(
    project_assignment_id SERIAL NOT NULL,
    employee_id           INT    NOT NULL,
    project_id            INT    NOT NULL,
    PRIMARY KEY (project_assignment_id),
    CONSTRAINT fk_project_assignment_employee
        FOREIGN KEY (employee_id)
            REFERENCES employee (employee_id),
    CONSTRAINT fk_project_assignment_project
        FOREIGN KEY (project_id)
            REFERENCES project (project_id)
);
```

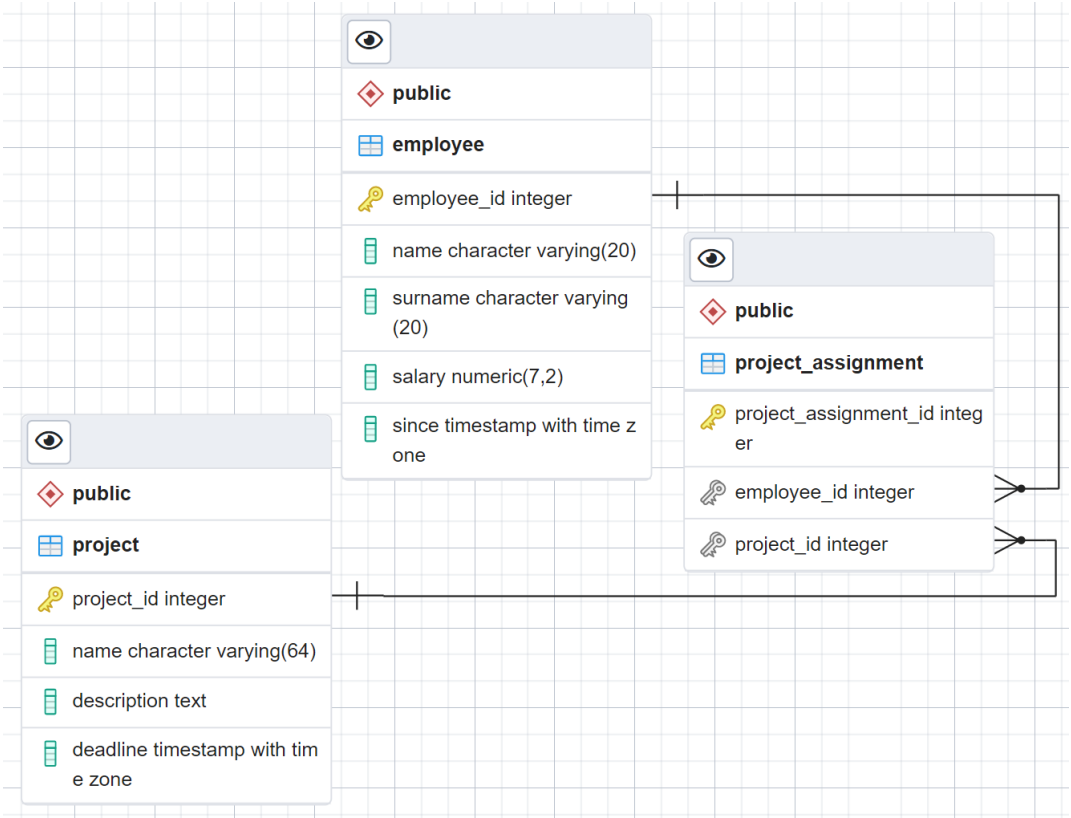
Na podstawie powyższej definicji możemy zauważyć, że tabela `project_assignment` będzie "złączała" ze sobą wpisy z tabeli `employee` z wpisami z tabeli `project`. W ten sposób dowiemy się z jakimi projektami powiązani są konkretni pracownicy. Żeby łatwiej to sobie zwizualizować, spójrz na poniższą grafikę:




*Obraz 6. Schemat danych dla tabel* `employee`, `project` *i* `project_assignment`

Na powyższej grafice możesz zauważyć, że tabela `project_assignment` złącza ze sobą wpisy z tabel `employee` oraz `project`. Tutaj mamy zapisane informacje, że pracownik z *employee_id=1* pracuje nad projektami z *project_id=1* oraz *project_id=2*. Natomiast pracownik z *employee_id=2* pracuje nad projektem z *project_id=1*.

Spójrzmy teraz na Diagram ERD dla stworzonej relacji.




*Obraz 7. ERD Diagram dla tabel* `employee`, `project` *i* `project_assignment`

Widzimy te same oznaczenia co wcześniej. Z racji relacji **many-to-many**, widzimy teraz trzy tabele. Wcześniej wspomnieliśmy, że pojedyncza pionowa kreska powinna znajdować się po stronie **one**, natomiast znaczek przypominający trójkąt po stronie **many**. Czyli w powyższym przykładzie można to zrozumieć w ten sposób, że *pracownik może mieć wiele przydziałów do projektów i projekt może mieć przydzielonych wielu pracowników*. Ponownie, znaczek przypominający trójkąt, jest po stronie tej tabeli, która ma zdefiniowane klucze obce.

Wracamy do Hibernate. Zdefiniowaliśmy już tabele `employee`, `project` i `project_assignment`. W jaki sposób można teraz przygotować mapowanie encji:

*Klasa Owner*

```
package pl.zajavka;

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

import java.math.BigDecimal;
import java.time.OffsetDateTime;
import java.util.Set;

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

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "employee_id")
    private Integer employeeId;

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

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

    @Column(name = "salary")
    private BigDecimal salary;

    @Column(name = "since")
    private OffsetDateTime since;

    @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinTable(
        name = "project_assignment",
        joinColumns = {@JoinColumn(name = "employee_id")},
        inverseJoinColumns = {@JoinColumn(name = "project_id")}
    )
    private Set<Project> projects;
```

```
    @Override
    public String toString() {
        return "Employee(employeeId=" + this.getEmployeeId() + ", name=" + this.getName()
            + ", surname=" + this.getSurname() + ", salary=" + this.getSalary()
            + ", since=" + this.getSince() + ", projects=" + this.getProjects() + ")";
    }
}
```

*Klasa Project*

```
package pl.zajavka;

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

import java.time.OffsetDateTime;
import java.util.Set;

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

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

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

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

    @Column(name = "deadline")
    private OffsetDateTime deadline;

    @ManyToMany(mappedBy = "projects")
    private Set<Employee> employees;

    @Override
    public String toString() {
        return "Project(projectId=" + this.getProjectId() + ", name=" + this.getName()
            + ", description=" + this.getDescription() + ", deadline=" + this.getDeadline() + ")";
    }
}
```

Powyższy przypadek jest ciekawy, bo nie mamy tutaj klasy dedykowanej pod tabelę `project_assignment`. Nazwa tej tabeli jest wspomniana w mapowaniu w klasie `Employee` razem z nazwami kolumn z tabeli `project_assignment` w parametrach `joinColumns` oraz `inverseJoinColumns`.

Pominiemy kod tworzący konfigurację Hibernate, gdyż była ona powtarzana wielokrotnie. Skupmy się natomiast na wywołaniu kilku zapytań na bazie danych.

*Klasa EmployeeRepository*

```
package pl.zajavka;

import org.hibernate.Session;

import java.util.*;

public class EmployeeRepository {

    List<Employee> insertData(final List<Employee> employees) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            employees.forEach(session::persist);
            session.getTransaction().commit();
            return employees;
        }
    }

    List<Employee> listEmployees() {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            String query = "SELECT employee FROM Employee employee";
            List<Employee> employees = session.createQuery(query, Employee.class).list();
            session.getTransaction().commit();
            return employees;
        }
    }

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

    void updateEmployee(Integer employeeId, Project newProject) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            Employee employee = session.find(Employee.class, employeeId);
            employee.getProjects().add(newProject);
            session.getTransaction().commit();
        }
    }

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

```
            session.beginTransaction();
            session.remove(session.find(Employee.class, employeeId));
            session.getTransaction().commit();
        }
    }

    void deleteAll() {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            session.beginTransaction();
            String query = "select employee from Employee employee";
            session.createQuery(query, Employee.class).list().forEach(session::remove);
            session.getTransaction().commit();
        }
    }
}
```

*Klasa ExampleData*

```
package pl.zajavka;

import java.math.BigDecimal;
import java.time.*;

class ExampleData {

    static Employee someEmployee1() {
        return Employee.builder().name("Agnieszka").surname("Pracownik")
            .salary(new BigDecimal("5910.73")).since(OffsetDateTime.now()).build();
    }

    static Employee someEmployee2() {
        return Employee.builder().name("Stefan").surname("Nowacki")
            .salary(new BigDecimal("3455.12")).since(OffsetDateTime.now()).build();
    }

    static Employee someEmployee3() {
        return Employee.builder().name("Tomasz").surname("Adamski")
            .salary(new BigDecimal("6112.42")).since(OffsetDateTime.now()).build();
    }

    static Project someProject1() {
        return Project.builder()
            .name("Database migration")
            .description("Lorem ipsum dolor sit amet, consectetur adipiscing elit.")
            .deadline(OffsetDateTime.of(2027, 10, 3, 12, 0, 0, 0, ZoneOffset.UTC))
            .build();
    }

    static Project someProject2() {
        return Project.builder()
            .name("Some external system integration")
            .description("Nullam hendrerit tellus nisl, tempus eleifend dui posuere nec.")
            .deadline(OffsetDateTime.of(2025, 10, 2, 12, 0, 0, 0, ZoneOffset.UTC))
            .build();
    }

    static Project someProject3() {
```

```
        return Project.builder()
            .name("Email sending refactoring")
            .description("Nulla maximus ac tellus nec elementum.")
            .deadline(OffsetDateTime.of(2024, 4, 2, 12, 0, 0, 0, ZoneOffset.UTC))
            .build();
    }
}
```

*Klasa ExampleRunner*

```
package pl.zajavka;

import java.time.*;
import java.util.*;

public class ExampleRunner {

    public static void main(String[] args) {
        EmployeeRepository employeeRepository = new EmployeeRepository();

        employeeRepository.deleteAll();

        List<Employee> employeesCreated = createEmployees(employeeRepository);

        employeeRepository.listEmployees()
            .forEach(employee -> System.out.println("###Employee listing: " + employee));

        System.out.println("###Employee 1: " + employeeRepository
            .getEmployee(employeesCreated.get(employeesCreated.size() - 1).getEmployeeId()));
        System.out.println("###Employee 2: " + employeeRepository
            .getEmployee(employeesCreated.get(employeesCreated.size() - 2).getEmployeeId()));

        updateEmployees(employeeRepository, employeesCreated);

        employeeRepository.listEmployees()
            .forEach(employee -> System.out.println("###Employee listing: " + employee));

        employeeRepository.deleteEmployee(employeesCreated
            .get(employeesCreated.size() - 2).getEmployeeId());

        employeeRepository.listEmployees()
            .forEach(employee -> System.out.println("###Employee listing: " + employee));

        HibernateUtil.closeSessionFactory();
    }

    private static List<Employee> createEmployees(final EmployeeRepository employeeRepository) {
        Project project1 = ExampleData.someProject1();
        Project project2 = ExampleData.someProject2();
        Project project3 = ExampleData.someProject3();
        Employee employee1 = ExampleData.someEmployee1();
        Employee employee2 = ExampleData.someEmployee2();
        Employee employee3 = ExampleData.someEmployee3();
        employee1.setProjects(Set.of(project1, project2));
        employee2.setProjects(Set.of(project2));
        employee3.setProjects(Set.of(project2, project3));

        return employeeRepository.insertData(List.of(employee1, employee2, employee3));
    }
```

```
    private static void updateEmployees(
        final EmployeeRepository employeeRepository,
        final List<Employee> employeesCreated
    ) {
        Employee employeeToBeUpdated = employeesCreated.get(employeesCreated.size() - 1);
        Project newProject = Project.builder()
            .name("Performance optimization")
            .description("Sed efficitur, diam sit amet maximus malesuada, mauris.")
            .deadline(OffsetDateTime.of(2025, 2, 1, 12, 0, 0, 0, ZoneOffset.UTC))
            .build();
        employeeRepository.updateEmployee(employeeToBeUpdated.getEmployeeId(), newProject);
        System.out.println("###Employee update: " + employeeRepository
            .getEmployee(employeesCreated.get(employeesCreated.size() - 1).getEmployeeId()));
    }
}
```

Na podstawie przedstawionego przykładu możesz spróbować zakomentować fragmenty kodu i wykonywać oddzielnie INSERT, UPDATE, DELETE itp. Będziesz wtedy w stanie dokładniej obserwować wykonywane zapytania.

Podsumowując przedstawiony przykład, możemy zaobserwować na konkretnym przykładzie relację **many-to-many**, czyli *wielu pracowników może pracować nad jednym projektem i jednocześnie jeden pracownik może pracować nad wieloma projektami.*

## Podsumowanie

Poniżej umieszczamy tabelę, w której znajdziesz użyte adnotacje razem z kontekstem ich użycia w zależności od rodzaju relacji. Dodaliśmy tutaj również informacje o mapowaniach jednokierunkowych **one-to-many** oraz **many-to-many**, których nie omawialiśmy w przykładach.

*Tabela 1. Definiowanie relacji jednokierunkowych:*

| Rodzaj relacji | Encja 1 | Encja 2 | Miejsce klucza obcego |
|---|---|---|---|
| jeden-do-jednego 1:1 | @OneToOne | - | Tabela ENCJA_1 |
| jeden-do-wielu 1:N | @OneToMany<br>@JoinColumn(name="") | - | Tabela ENCJA_2 |
| wiele-do-wielu N:N | @ManyToMany | - | W tabeli reprezentującej relację |

*Tabela 2. Definiowanie relacji dwukierunkowych:*

| Rodzaj relacji | Encja1 | Encja2 | Miejsce klucza obcego |
|---|---|---|---|
| jeden-do-jednego 1:1 | @OneToOne | @OneToOne(mappedBy="") | Tabela ENCJA_1 |
| jeden-do-wielu 1:N | @OneToMany<br>@JoinColumn(name="") | @ManyToOne | Tabela ENCJA_2 |
| wiele-do-wielu N:N | @ManyToMany | @ManyToMany(mappedBy="") | W tabeli reprezentującej relację |

# Parametr fetch

Przejdziemy teraz do omówienia dwóch parametrów, które były wcześniej pominięte: `fetch` i `cascade`. Zaczniemy od `fetch`.

## Lazy Loading

Zjawisko, które zostanie poniżej przedstawione, nosi nazwę **lazy loading**. Wcześniej widzieliśmy już, że Java Streams są lazy. Stream nie zostanie wykonany do momentu, gdy nie zostanie uruchomiona operacja terminująca. Przykład, który zobaczymy za moment, też służy do implementacji zjawiska **lazy loading**.

A kiedy może się to przydać w praktyce? Tak naprawdę, to mamy z tym do czynienia cały czas. Gdy kupujesz produkty w sklepie internetowym, to aplikacja nie ładuje szczegółów wszystkich produktów, gdy jesteś na głównej stronie. Zaciągane są tylko niezbędne informacje, teksty i grafiki. Dopiero po wejściu w konkretny produkt, przeglądarka zaciąga jego szczegóły. Gdyby było odwrotnie, to miałoby to tragiczny wpływ na wydajność aplikacji. Analogicznie działa to w przypadku Hibernate.

## FetchType

Relacje w bazach danych są definiowane przy wykorzystaniu **JOIN**. Poruszyliśmy już wcześniej relacje takie jak **one-to-one** (*jeden-do-jednego*), **one-to-many** (*jeden-do-wielu*) oraz **many-to-many** (*wiele-do-wielu*). W momencie, gdy korzystamy z takich relacji, konfigurując mapowanie encji w Hibernate, możemy zdefiniować parametr `fetch`.

Parametr `fetch` specyfikuje, czy mają zostać załadowane wszystkie dane dotyczące relacji od razu (*EAGER*), czy może dopiero gdy będą one potrzebne (*LAZY*). W kodzie źródłowym każdej adnotacji można zobaczyć, czy parametrem domyślnym jest `EAGER`, czy `LAZY` - zwracam na to uwagę, bo różne adnotacje mają ustawione to inaczej.

- **FetchType.EAGER** - pobierz encje podrzędne w relacji wraz z pobraniem danych rodzica,
- **FetchType.LAZY** - jak sama nazwa wskazuje, pobiera encje podrzędne w relacji leniwie, czyli dopiero wtedy, gdy są one faktycznie potrzebne.

Podejście *LAZY* zwiększa wydajność aplikacji, gdyż nie wykonuje ona wtedy niepotrzebnych zapytań. Do tego zmniejszamy w ten sposób zużycie pamięci, bo niepotrzebne obiekty nie są do niej ładowane. Podejście *EAGER* będzie powodowało wykonywanie niepotrzebnych zapytań i zwiększenie użycie pamięci. Powinno być ono jednak stosowane np. gdy chcemy zaznaczyć, że dane podrzędne są absolutnie niezbędne do realizacji logiki biznesowej.

## Przykład

Opierając się o przedstawiony wcześniej przykład tabel `owner` oraz `pet` oraz mapowań `Owner` i `Pet`, napiszmy teraz taki kod (fragmenty, które nie są pokazane, nie ulegają zmianie w stosunku do poprzednich przykładów).

Żeby lepiej było widać przedstawione przykłady, zmień ustawienie `hibernate.format_sql` na false.

*Klasa OwnerRepository*

```
package pl.zajavka;

import org.hibernate.Session;
import java.util.*;

public class OwnerRepository {

    Owner insertData(final Owner owner, final Set<Pet> pets) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            System.out.println("###BEFORE INSERT\n----------------------------");
            session.beginTransaction();
            owner.setPets(pets);
            pets.forEach(pet -> pet.setOwner(owner));
            session.persist(owner);
            session.getTransaction().commit();
            System.out.println("----------------------------\n###AFTER INSERT");
            return owner;
        }
    }

    Optional<Owner> getOwner(Integer ownerId) {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            System.out.println("###BEFORE GET OWNER\n----------------------------");
            Owner owner = session.find(Owner.class, ownerId);
            System.out.println("----------------------------\n###AFTER GET OWNER");
            System.out.println("###BEFORE GET PETS\n----------------------------");
            System.out.println(owner.getPets());
            System.out.println("----------------------------\n###AFTER GET PETS");
            return Optional.ofNullable(owner);
        }
    }

    void deleteAll() {
        try (Session session = HibernateUtil.getSession()) {
            if (Objects.isNull(session)) {
                throw new RuntimeException("Session is null");
            }
            System.out.println("###BEFORE DELETE ALL\n----------------------------");
            session.beginTransaction();
            String query = "select owner from Owner owner";
            session.createQuery(query, Owner.class).list().forEach(session::remove);
            session.getTransaction().commit();
            System.out.println("----------------------------\n###AFTER DELETE ALL");
        }
    }
}
```

*Klasa ExampleRunner*

```
package pl.zajavka;
```

```
import java.util.Set;

public class ExampleRunner {

    public static void main(String[] args) {
        OwnerRepository ownerRepository = new OwnerRepository();

        ownerRepository.deleteAll();

        Owner owner = ownerRepository.insertData(
            ExampleData.someOwner1(),
            Set.of(ExampleData.somePet1(), ExampleData.somePet2())
        );

        ownerRepository.getOwner(owner.getId()).orElseThrow();

        HibernateUtil.closeSessionFactory();
    }
}
```

Jeżeli uruchomisz teraz program, to na ekranie zostanie wydrukowane:

```
###BEFORE GET OWNER
----------------------------
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=? ①
----------------------------
###AFTER GET OWNER
###BEFORE GET PETS
----------------------------
[Pet(id=11, name=Kiciek, breed=CAT), Pet(id=10, name=Fafik, breed=DOG)]
----------------------------
###BEFORE GET PETS
```

① Zapamiętaj ten JOIN

Zmień teraz w kodzie jeden parametr (*EAGER* na *LAZY*) i uruchom ten sam program ponownie:

*Klasa Owner*

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

Na ekranie zostanie teraz wydrukowane:

```
###BEFORE GET OWNER
----------------------------
Hibernate: select (...) from owner o1_0 where o1_0.owner_id=? ①
----------------------------
###AFTER GET OWNER
###BEFORE GET PETS
----------------------------
Hibernate: select (...) from pet p1_0 where p1_0.owner_id=? ②
[Pet(id=13, name=Kiciek, breed=CAT), Pet(id=12, name=Fafik, breed=DOG)]
----------------------------
```

```
###BEFORE GET PETS
```

① Tutaj teraz nie ma JOIN

② Za to tutaj jest wykonywane dodatkowe zapytanie, dopiero jak wywołamy `.getPets()` w kodzie. Jeżeli nie wywołalibyśmy `.getPets()`, to to zapytanie nie zostałoby wykonane i nie załadowalibyśmy listy zwierząt do pamięci, bo nie byłyby nam potrzebne.

Spróbuj teraz wykonać wariant *EAGER* bez wywołania `.getPets()` oraz *LAZY* bez wywołania `.getPets()`. Zobacz, jaki otrzymasz rezultat.

Podsumowując, *EAGER* dociąga dane przez JOIN, nawet jeżeli nie są nam potrzebne. *LAZY* robi to dopiero w momencie, gdy faktycznie wystąpi zapotrzebowanie na pobranie pewnych danych.

**Należy tutaj jednak pamiętać o jednej rzeczy**. Jeżeli napiszę ten przykład tak:

*Klasa OwnerRepository*

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

*Klasa ExampleRunner*

```
Owner ownerCreated = ownerRepository.getOwner(owner.getId()).orElseThrow();
System.out.println(ownerCreated.getPets());
```

To w trakcie działania programu zostanie wyrzucony wyjątek:

```
org.hibernate.LazyInitializationException:
failed to lazily initialize a collection of role:
pl.zajavka.Owner.pets: could not initialize proxy - no Session
```

Czyli przy pobieraniu danych, które są *LAZY* musimy pamiętać, że konieczna jest aktywna sesja Hibernate.

# Kaskady

Możemy teraz przejść do omówienia drugiego parametru, czyli `cascade`. Wcześniej również go używaliśmy, ale teraz przejdziemy do jego omówienia.

Jeżeli zastanawiasz się czemu, gdy rozmawialiśmy o relacjach, to niektóre encje były "magicznie" zapisywane lub usuwane, to właśnie przez parametr `cascade`. Teraz

przejdziemy do jego omówienia.

W poniższych przykładach będziemy opierać się o przedstawiony wcześniej przykład `owner` oraz `pet` oraz mapowania `Owner` i `Pet`. Przypomnijmy sobie, że w klasie `Owner` widzieliśmy taki zapis:

*Klasa Owner*

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

Podczas gdy klasa `Pet` miała dodany taki zapis:

*Klasa Pet*

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

W powyższym przykładzie został zastosowany zapis *cascade = CascadeType.ALL*. Oznacza on, że każda zmiana, która nastąpiła w `Owner`, musi być również propagowana kaskadowo do `Pet`. Jeżeli dokonamy zapisu właściciela, to wszystkie powiązane z nim zwierzęta również zostaną zapisane w bazie danych. Jeżeli usuniemy właściciela, to ponownie, wszystkie zwierzęta powiązane w tym właścicielem również zostaną usunięte. Za to wszystko odpowiedzialne jest ustawienie `CascadeType.ALL`. Zabawa zaczyna się wtedy, gdy chcemy, żeby kaskadowo propagowane były tylko niektóre rodzaje operacji. Do tego mamy dedykowane inne ustawienia parametru `cascade`.

## CascadeType

W JPA mamy dostępne następujące rodzaje kaskad:

- **CascadeType.PERSIST** - to ustawienie oznacza, że operacje `persist()` będą kaskadowane do powiązanych encji,
- **CascadeType.MERGE** - to ustawienie oznacza, że operacje `merge()` będą kaskadowane do powiązanych encji,
- **CascadeType.REFRESH** - to ustawienie oznacza, że operacje `refresh()` będą kaskadowane do powiązanych encji,
- **CascadeType.REMOVE** - to ustawienie oznacza, że operacje `remove()` będą kaskadowane do powiązanych encji,
- **CascadeType.DETACH** - to ustawienie oznacza, że operacje `detach()` będą kaskadowane do powiązanych encji,
- **CascadeType.ALL** - Cytując dokumentację: *The value cascade=ALL is equivalent to cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}*.

W JPA nie ma domyślnego typu kaskadowego. Domyślnie żadna operacja nie jest kaskadowana.

Parametr `cascade` akceptuje tablicę wartości. Możemy przykładowo ustawić jednocześnie wartości `PERSIST` oraz `REMOVE`. Wyglądałoby to następująco:

```
@OneToMany(cascade={CascadeType.PERSIST, CascadeType.REMOVE}, fetch = FetchType.LAZY)
```

## Przykład one-to-one

Przejdźmy do omówienia kaskad przy wykorzystaniu relacji **one-to-one**.

### Dodawanie danych do bazy danych

Zaczniemy od przykładu `Customer` oraz `Address`. Była to relacja **one-to-one**, w której adres nie mógł istnieć bez klienta. Przygotuj sobie poniższy przykład:

*Klasa Customer*

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

*Klasa Address*

```
@OneToOne(fetch = FetchType.LAZY, mappedBy = "address")
private Customer customer;
```

*Metoda insertCustomer() w klasie CustomerRepository*

```
Customer insertCustomer(final Customer customer) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        System.out.println("###BEFORE INSERT\n-----------------------------");
        session.beginTransaction();
        session.persist(customer); ①
        session.getTransaction().commit();
        System.out.println("-----------------------------\n###AFTER INSERT");
        return customer;
    }
}
```

① Zwróć uwagę, że nigdzie nie zapisujemy ręcznie wartości do tabeli `address`.

Jeżeli wykonasz ten kod z ustawieniem *CascadeType.ALL*, to na ekranie zostanie wydrukowane:

```
###BEFORE INSERT
-----------------------------
Hibernate: insert into address (address, city, country, postal_code) values (?, ?, ?, ?)
Hibernate: insert into customer (address_id, email, name, phone, surname) values (?, ?, ?, ?, ?)
-----------------------------
###AFTER INSERT
```

Czyli Hibernate nie dość, że za nas dodał wpis do tabeli `address`, to jeszcze zrobił to w dobrej kolejności. Encja Address musi być zapisana przed encją Customer, ze względu na to, że to Customer jest zależne od

Address, a nie odwrotnie.

Natomiast, jeżeli ustawienie `cascade` nie będzie ustawione wcale lub będzie ustawione na *CascadeType.REMOVE*, to na ekranie wydrukowane zostanie:

```
org.postgresql.util.PSQLException: ERROR:
null value in column "address_id" of relation "customer" violates not-null constraint
```

Oznacza to, że tym razem, Hibernate nie wykonał za nas *insert into address*, zatem nie można było poprawnie dodać rekordu do tabeli `customer`. Jeżeli parametr `cascade` nie będzie ustawiony wcale, lub w tym przypadku będzie ustawiony na *CascadeType.REMOVE*, to dodanie encji Address do bazy danych musimy przeprowadzić ręcznie:

```
// ...
session.beginTransaction();
session.persist(customer.getAddress());
session.persist(customer);
session.getTransaction().commit();
// ...
```

### Usuwanie danych z bazy danych

Parametr `cascade` będzie również przydatny, jeżeli będziemy chcieli usuwać dane z bazy danych i zależy nam na tym, żeby dane były usuwane kaskadowo. Przygotuj sobie następujący przykład:

*Klasa Customer*

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

*Klasa Address*

```
@OneToOne(fetch = FetchType.LAZY, mappedBy = "address")
private Customer customer;
```

*Metoda deleteCustomer() w klasie CustomerRepository*

```
void deleteCustomer(Integer customerId) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        System.out.println("###BEFORE DELETE\n----------------------------");
        session.beginTransaction();
        session.remove(session.find(Customer.class, customerId)); ①
        session.getTransaction().commit();
        System.out.println("----------------------------\n###AFTER DELETE");
    }
}
```

① Zwróć uwagę, że nigdzie nie usuwamy ręcznie wpisów z tabeli `address`.

Jeżeli wykonasz ten kod z ustawieniem *CascadeType.REMOVE*, to na ekranie zostanie wydrukowane:

```
###BEFORE DELETE
----------------------------
Hibernate: select (...) from customer c1_0
    left join address a1_0 on a1_0.address_id=c1_0.address_id
    where c1_0.customer_id=?
Hibernate: delete from customer where customer_id=?
Hibernate: delete from address where address_id=?
----------------------------
###AFTER DELETE
```

Czyli Hibernate nie dość, że za nas usunął wpis z tabeli `address`, to jeszcze zrobił to w dobrej kolejności. Encja Address musi być usunięta po encji Customer, ze względu na to, że to Customer jest zależne od Address, a nie odwrotnie.

Natomiast, jeżeli ustawienie `cascade` nie będzie ustawione wcale lub będzie ustawione na *CascadeType.PERSIST*, to na ekranie wydrukowane zostanie:

```
###BEFORE DELETE
----------------------------
Hibernate: delete from customer where customer_id=?
----------------------------
###AFTER DELETE
```

Oznacza to, że tym razem, Hibernate nie wykonał za nas *delete from address*, zatem wpis dotyczący adresu nie zostaje usunięty. Jeżeli parametr `cascade` nie będzie ustawiony wcale, lub w tym przypadku będzie ustawiony na *CascadeType.PERSIST*, to usunięcie encji Address z bazy danych musimy przeprowadzić ręcznie:

```
// ...
session.beginTransaction();
Customer toRemove = session.find(Customer.class, customerId);
session.remove(toRemove.getAddress());
session.remove(toRemove);
session.getTransaction().commit();
// ...
```

## Przykład one-to-many

Przejdźmy do omówienia kaskad przy wykorzystaniu relacji **one-to-many**.

### Dodawanie danych do bazy danych

Spójrzmy teraz ponownie na przykład `Owner` oraz `Pet`. Dostosuj klasy do stanu poniżej:

*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;
```

*Metoda insertData() w klasie OwnerRepository*

```
Owner insertData(final Owner owner, final Set<Pet> pets) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        System.out.println("###BEFORE INSERT\n----------------------------");
        session.beginTransaction();
        owner.setPets(pets);
        pets.forEach(pet -> pet.setOwner(owner));
        session.persist(owner); ①
        session.getTransaction().commit();
        System.out.println("----------------------------\n###AFTER INSERT");
        return owner;
    }
}
```

① Zauważ, że nigdzie nie jest wołane `session.persist()` dla encji `Pet`.

Po wykonaniu powyższego kodu, na ekranie zostanie wydrukowane:

```
###BEFORE INSERT
----------------------------
Hibernate: insert into owner (email, name, phone, surname) values (?, ?, ?, ?)
Hibernate: insert into pet (breed, name, owner_id) values (?, ?, ?)
Hibernate: insert into pet (breed, name, owner_id) values (?, ?, ?)
----------------------------
###AFTER INSERT
```

Hibernate nie dość, że za nas dodał wpis do tabeli `pet`, to jeszcze zrobił to w dobrej kolejności. Encja Owner musi być zapisana przed encją Pet, ze względu na to, że to Pet ma zdefiniowany klucz obcy do Owner, a nie odwrotnie. Czyli najpierw musimy zapisać Owner, żeby Pet mógł się do tego wpisu odwołać.

Natomiast, jeżeli ustawienie `cascade` nie będzie ustawione wcale lub będzie ustawione na *CascadeType.REMOVE*, to na ekranie wydrukowane zostanie:

```
###BEFORE INSERT
----------------------------
Hibernate: insert into owner (email, name, phone, surname) values (?, ?, ?, ?)
----------------------------
###AFTER INSERT
```

Ustawienie *CascadeType.ALL*, które w sobie "zawiera" ustawienie *CascadeType.PERSIST* powodowało, że Hibernate oprócz zapisania w bazie encji Owner, dokonywał również zapisu encji powiązanych, czyli Pet. Jeżeli nie ustawimy wspomnianego *CascadeType*, to dodanie encji Pet musimy wtedy przeprowadzić ręcznie:

```
// ...
session.beginTransaction();
owner.setPets(pets);
pets.forEach(pet -> pet.setOwner(owner));
session.persist(owner);
pets.forEach(session::persist);
session.getTransaction().commit();
// ...
```

### Usuwanie danych z bazy danych

Spójrzmy na ten sam przykład, ale tym razem spróbujmy usunąć dane z bazy danych. Dostosuj klasy do stanu poniżej:

*Klasa Owner*

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

*Klasa Pet*

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

*Metoda deleteOwner() w klasie OwnerRepository*

```
void deleteOwner(Integer ownerId) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        System.out.println("###BEFORE DELETE\n-------------------------");
        session.beginTransaction();
        session.remove(session.find(Owner.class, ownerId));
        session.getTransaction().commit();
        System.out.println("-------------------------\n###AFTER DELETE");
    }
}
```

Jeżeli wykonasz ten kod z powyższym ustawieniem `cascade`, to na ekranie zostanie wydrukowane:

```
###BEFORE DELETE
```

```
----------------------------
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: delete from pet where pet_id=?
Hibernate: delete from pet where pet_id=?
Hibernate: delete from owner where owner_id=?
----------------------------
###AFTER DELETE
```

Czyli Hibernate nie dość, że za nas usunął wpis z tabeli `pet`, to jeszcze zrobił to w dobrej kolejności. Encja Pet musi być usunięta przed encją Owner, ze względu na to, że to Pet jest zależne od Owner, a nie odwrotnie. Skoro Pet "wskazuje na" Owner, to Pet musi być usunięte w pierwszej kolejności.

Natomiast, jeżeli ustawienie `cascade` nie będzie ustawione wcale lub będzie ustawione na *CascadeType.PERSIST*, to na ekranie wydrukowane zostanie:

```
###BEFORE DELETE
----------------------------
Hibernate: delete from owner where owner_id=?
WARN  - SQL Error: 0, SQLState: 23503
ERROR - ERROR: update or delete on table "owner" violates foreign key constraint "fk_pet_owner" on table
"pet"
  Szczegóły: Key (owner_id)=(49) is still referenced from table "pet".
----------------------------
###AFTER DELETE
```

Wynika to z tego, że Hibernate stara się usunąć wpis z tabeli `owner`, ale na ten wpis z tej tabeli wskazuje inny wpis z tabeli `pet`.

Czyli ustawienie *CascadeType.ALL*, które w sobie "zawiera" ustawienia *CascadeType.PERSIST* oraz *CascadeType.REMOVE* powoduje, że Hibernate oprócz usunięcia z bazy encji Owner, dokonuje również usuwania encji powiązanych, czyli Pet. Jeżeli nie ustawimy wspomnianego *CascadeType*, to usuwanie encji Pet musimy wtedy przeprowadzić ręcznie:

```
// ...
session.beginTransaction();
Owner owner = session.find(Owner.class, ownerId);
owner.getPets().forEach(session::remove);
session.remove(owner);
session.getTransaction().commit();
// ...
```

## Przykład many-to-many

Przejdźmy do omówienia kaskad przy wykorzystaniu relacji **many-to-many**.

### Dodawanie danych do bazy danych

Spójrzmy teraz ponownie na przykład `Employee` oraz `Project`. Dostosuj klasy do stanu poniżej:

*Klasa Employee*

```
@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
@JoinTable(
    name = "project_assignment",
    joinColumns = {@JoinColumn(name = "employee_id")},
    inverseJoinColumns = {@JoinColumn(name = "project_id")}
)
private Set<Project> projects;
```

*Klasa Project*

```
@ManyToMany(mappedBy = "projects")
private Set<Employee> employees;
```

*Metoda insertData() w klasie EmployeeRepository*

```
List<Employee> insertData(final List<Employee> employees) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        session.beginTransaction();
        for (Employee employee : employees) {
            System.out.println("###BEFORE INSERT\n----------------------------");
            session.persist(employee); ①
            System.out.println("----------------------------\n###AFTER INSERT");
        }
        session.getTransaction().commit();
        return employees;
    }
}
```

① Zauważ, że nigdzie nie jest wołane `session.persist()` dla encji `Project`.

Jeżeli wykonasz ten kod z powyższym ustawieniem `cascade`, to na ekranie zostanie wydrukowane przykładowo:

```
###BEFORE INSERT
----------------------------
Hibernate: insert into employee (name, salary, since, surname) values (?, ?, ?, ?)
Hibernate: insert into project (deadline, description, name) values (?, ?, ?)
Hibernate: insert into project (deadline, description, name) values (?, ?, ?)
----------------------------
###AFTER INSERT
###BEFORE INSERT
----------------------------
Hibernate: insert into employee (name, salary, since, surname) values (?, ?, ?, ?)
----------------------------
###AFTER INSERT
###BEFORE INSERT
----------------------------
Hibernate: insert into employee (name, salary, since, surname) values (?, ?, ?, ?)
Hibernate: insert into project (deadline, description, name) values (?, ?, ?)
----------------------------
###AFTER INSERT
```

```
Hibernate: insert into project_assignment (employee_id, project_id) values (?, ?)
Hibernate: insert into project_assignment (employee_id, project_id) values (?, ?)
Hibernate: insert into project_assignment (employee_id, project_id) values (?, ?)
Hibernate: insert into project_assignment (employee_id, project_id) values (?, ?)
Hibernate: insert into project_assignment (employee_id, project_id) values (?, ?)
```

Zauważ, że wartości na ekranie nie są drukowane w takiej kolejności w jakiej oczekujemy, chodzi tutaj o wpisy dotyczące `project_assignment`. To jest poprawne zachowanie. Hibernate wykonuje zapytania na bazie danych i moment wykonania insertów do tabeli `project_assignment` następuje na etapie commita, który wykonywany jest po wydrukach *BEFORE INSERT* oraz *AFTER INSERT*.

Natomiast, jeżeli ustawienie `cascade` nie będzie ustawione wcale lub będzie ustawione na *CascadeType.REMOVE*, to na ekranie wydrukowane zostanie:

```
java.lang.IllegalStateException:
    org.hibernate.TransientObjectException:
        object references an unsaved transient instance -
            save the transient instance before flushing: pl.zajavka.Project
```

Czyli ustawienie *CascadeType.ALL*, które w sobie "zawiera" ustawienie *CascadeType.PERSIST* powodowało, że Hibernate oprócz zapisania w bazie encji Employee, dokonywał również zapisu encji powiązanych, czyli Project oraz dodawał wpis do tabeli `project_assignment`. Jeżeli nie ustawimy wspomnianego *CascadeType*, to musimy zatroszczyć się o dodanie takich wpisów ręcznie:

```
session.beginTransaction();
for (Employee employee : employees) {
    System.out.println("###BEFORE INSERT\n----------------------------");
    employee.getProjects().forEach(session::persist); ①
    session.persist(employee);
    System.out.println("----------------------------\n###AFTER INSERT");
}
session.getTransaction().commit();
```

① W tym miejscu zapisujemy projekty ręcznie.

W tym przypadku możesz zwrócić uwagę, że Hibernate nadal za nas doda wpisy do tabeli `project_assignment`. Jest to o tyle ciekawy przypadek, że nie dodaliśmy klasy reprezentującej tabelę `project_assignment`, Hibernate zarządza tym za nas.

### Usuwanie danych z bazy danych

Spójrzmy na ten sam przykład, ale tym razem spróbujmy usunąć dane z bazy danych. Dostosuj klasy do stanu poniżej:

*Klasa Employee*

```
@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.REMOVE}, fetch = FetchType.EAGER)
@JoinTable(
    name = "project_assignment",
    joinColumns = {@JoinColumn(name = "employee_id")},
```

```
    inverseJoinColumns = {@JoinColumn(name = "project_id")}
)
private Set<Project> projects;
```

*Klasa Project*

```
@ManyToMany(mappedBy = "projects")
private Set<Employee> employees;
```

*Metoda deleteEmployee() w klasie EmployeeRepository*

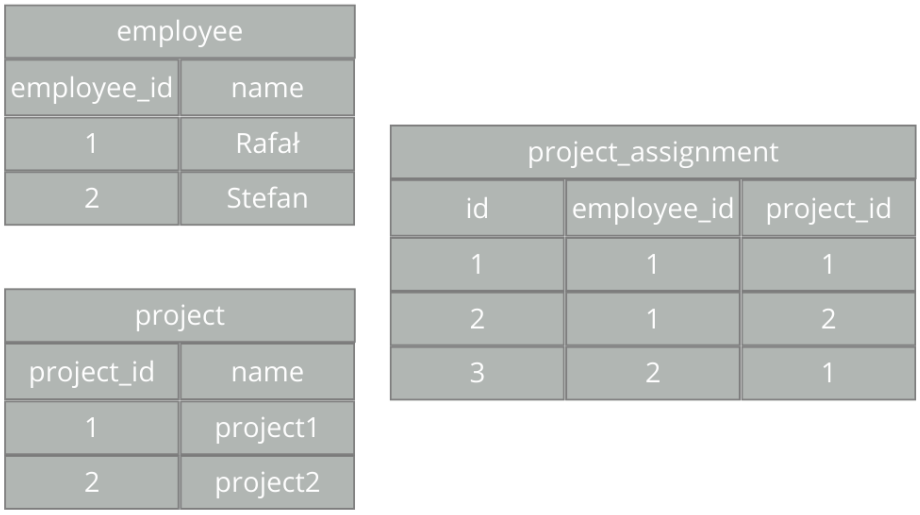
```
void deleteEmployee(Integer employeeId) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
        System.out.println("###BEFORE DELETE\n----------------------------");
        session.beginTransaction();
        session.remove(session.find(Employee.class, employeeId));
        session.getTransaction().commit();
        System.out.println("----------------------------\n###AFTER DELETE");
    }
}
```

Jeżeli wykonasz ten kod z powyższym ustawieniem `cascade`, to na ekranie zostanie wydrukowane:

```
###BEFORE DELETE
----------------------------
Hibernate: select (...) from employee e1_0
    left join (project_assignment p1_0
    join project p1_1 on p1_1.project_id=p1_0.project_id) on e1_0.employee_id=p1_0.employee_id
    where e1_0.employee_id=?
Hibernate: delete from project_assignment where employee_id=?
Hibernate: delete from project where project_id=?
Hibernate: delete from project where project_id=?
Hibernate: delete from employee where employee_id=?
----------------------------
###AFTER DELETE
```

Hibernate w powyższym przykładzie nie dość, że za nas usunął wpisy z tabel `project` oraz `project_assignment`, to jeszcze zrobił to w dobrej kolejności. Wpisy z tabeli `project_assignment` muszą być usunięte w pierwszej kolejności, ze względu na to, że to wpisy w tej tabeli "spinają" ze sobą pracowników i projekty. Dopiero po usunięciu wpisów z tabeli `project_assignment`, można usuwać wpisy z tabel `employee` oraz `project`. Hibernate robi to za nas i w dobrej kolejności.

W relacjach **many-to-many** trzeba uważać z ustawieniami `cascade`, szczególnie przy usuwaniu. Skoro jeden pracownik może pracować na wielu projektach i nad jednym projektem może pracować wielu pracowników, to normalną jest sytuacja, że wpisy w tabeli `project` są powiązane z wieloma pracownikami. Musimy natomiast uważać w sytuacji, która jest widoczna poniżej.




*Obraz 8. Schemat danych dla tabel* `employee`, `project` *i* `project_assignment`

Na powyższej grafice możesz zauważyć, że tabela `project_assignment` złącza ze sobą wpisy z tabel `employee` oraz `project`. Tutaj mamy zapisane informacje, że pracownik z *employee_id=1* pracuje nad projektami z *project_id=1* oraz *project_id=2*. Natomiast pracownik z *employee_id=2* pracuje nad projektem z *project_id=1*.

Jeżeli będziemy chcieli usunąć pracownika *employee_id=1*, to Hibernate w wyniku kaskady spróbuje też usunąć *project_id=1*. Nad tym projektem pracuje też pracownik *employee_id=2*, dostaniemy w takim przypadku błąd:

```
Caused by: org.postgresql.util.PSQLException: ERROR: update or delete on table
"project" violates foreign key constraint "fk_project_assignment_project" on table
"project_assignment"
```

Przykład ten pokazuje, że stosując kaskady musimy uważać co robimy z danymi i stosować je rozważnie. Stosując kaskady bez zastanowienia, możemy doprowadzić do sytuacji, gdzie Hibernate będzie próbował kasować dane, których kasować nie może, bo będą wtedy naruszane klucze obce. Zatem w uproszczeniu można stwierdzić, że bezpieczniej jest nie używać *CascadeType.REMOVE* z adnotacjami `@ManyToMany`.

Natomiast, jeżeli ustawienie `cascade` nie będzie ustawione wcale lub będzie ustawione na *CascadeType.PERSIST*, to na ekranie wydrukowane zostanie:

```
###BEFORE DELETE
----------------------------
Hibernate: select (...) from employee e1_0
    left join (project_assignment p1_0
    join project p1_1 on p1_1.project_id=p1_0.project_id) on e1_0.employee_id=p1_0.employee_id
    where e1_0.employee_id=?
Hibernate: delete from project_assignment where employee_id=?
Hibernate: delete from employee where employee_id=?
----------------------------
###AFTER DELETE
```

W tym przypadku ustawienie *CascadeType.PERSIST*, powoduje, że usuwanie danych nie jest kaskadowane. W takim przypadku Hibernate usuwa tylko wpisy z tabel `project_assignment` oraz

employee. Nie są usuwane wpisy z tabeli project. Trzeba o tym pamiętać, bo w takiej sytuacji możemy doprowadzić do tego, że w prawdzie usuniemy pracowników, ale w bazie zostaną nam projekty, nad którymi nikt nie pracuje. Trzeba też pamiętać o tym, że jeżeli nie korzystamy z kaskadowania, to w takim przypadku encje Project należy usunąć ręcznie:

```
// ...
System.out.println("###BEFORE DELETE\n-------------------------");
session.beginTransaction();
Employee employee = session.find(Employee.class, employeeId);
employee.getProjects().forEach(session::remove);
session.remove(employee);
session.getTransaction().commit();
System.out.println("-------------------------\n###AFTER DELETE");
// ...
```

# Orphan Removal

Oprócz parametru cascade, możliwe jest również ustawienie parametru orphanRemoval. Zapewnia to możliwość usunięcia "osieroconych" encji z bazy danych. Encja osierocona może być rozumiana jako taka, która nie jest połączona ze swoim rodzicem. Czyli jest to encja podrzędna, która nie ma zapewnionego powiązania ze swoją encją nadrzędną.

Jeżeli wrócimy teraz do przykładu owner i pet, skonfigurujmy encje w następujący sposób:

*Klasa Owner*

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

// ...

public void removePet(final Pet pet) {
    pets.remove(pet);
}
```

*Klasa Pet*

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

*Metoda orphanRemovalExample() w klasie CustomerRepository*

```
void orphanRemovalExample(final Integer ownerId) {
    try (Session session = HibernateUtil.getSession()) {
        if (Objects.isNull(session)) {
            throw new RuntimeException("Session is null");
        }
```

```
        System.out.println("###BEFORE ORPHAN\n----------------------------");
        session.beginTransaction();
        Owner owner = session.find(Owner.class, ownerId);
        Pet petToRemove = owner.getPets().stream().findAny().orElseThrow();
        owner.removePet(petToRemove);
        session.merge(owner); ①
        session.getTransaction().commit();
        System.out.println("----------------------------\n###AFTER ORPHAN");
    }
}
```

① Zwróć uwagę, że wykonujemy `merge()`, a nie `persist()` albo `remove()`.

Jeżeli wywołamy teraz powyższy kod, to na ekranie zostanie wydrukowane:

```
###BEFORE ORPHAN
----------------------------
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: delete from pet where pet_id=?
----------------------------
###AFTER ORPHAN
```

Czyli Hibernate zatroszczył się o usunięcie encji Pet, która nie miała powiązania z encją Owner. Jeżeli natomiast ustawimy *orphanRemoval* = *false*, które jest wartością domyślną, to nie zobaczymy na ekranie wpisu *delete from*. Hibernate zostawi wtedy wpis w Pet, który nie jest nigdzie użyty. Zwróć uwagę, że ustawiliśmy *cascade* = *CascadeType.PERSIST*, czyli nie korzystamy z *CascadeType.REMOVE*.

Parametr `orphanRemoval` może być stosowany z adnotacjami `@OneToOne` oraz `@OneToMany`, natomiast nie może być używany z adnotacjami `@ManyToOne` oraz `@ManyToMany`.

### CascadeType.REMOVE vs orphanRemoval

*CascadeType.REMOVE* jest sposobem na usunięcie encji podrzędnych, gdy nastąpi usunięcie jej encji nadrzędnej. Natomiast `orphanRemoval`, daje nam możliwość usunięcia osieroconych jednostek z bazy danych.

## Podsumowanie

Na tym etapie udało Ci się już zapoznać z bardzo dużą ilością mechanizmów, jakie daje nam JPA i które są implementowane przez Hibernate. Chcę tutaj wyraźnie zaznaczyć, że to czy dany mechanizm jest w kodzie stosowany czy nie, jest decyzją dewelopera i zależy od przypadku biznesowego. Czyli to czy narzucisz *FetchType.EAGER*, czy *FetchType.LAZY* jest Twoją dedycją w kodzie. To, czy wykorzystasz mechanizm kaskad, czy też nie i wtedy będziesz dbać o powiązania między encjami ręcznie, też jest Twoją decyzją.

Tak samo, jak w przypadku stosowania każdego innego mechanizmu, tak i tutaj należy pamiętać, że jakiej decyzji byśmy nie podjęli (stosować kaskady, czy nie), musisz wiedzieć jakie konsekwencje niesie ze sobą stosowanie danego parametru, bo dopiero wtedy będziesz w stanie zrozumieć: *Czemu ten kod się tak dziwnie zachowuje*. 😀