DronkaShop

Neo4J + Cypher + GraphQL / Kotlin

Danylo Knapp, Kacper Słoniec, Bartłomiej Szymański

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

13 czerwca 2023 r.

Spis treści

Wstęp

W tym projekcie inspirowaliśmy się takimi frameworkami jak EntityFramework i Hibernate, a więc stworzyliśmy swój własny półautomatyczny framework w języku Kotlin (JVM).

Postanowiliśmy korzystać z GraphQL – języka, za pomocą którego można w łatwy i naturalny sposób pobierać dane / je zapisywać.

Jako bazę danych zdecydowaliśmy użyć Neo4J – grafowej bazy danych napisanej w języku Java. W tym celu stworzyliśmy instancję AuraDB.

Warto również dodać, że zarówno z powodu pewnego buga jak i z powodu słabej implementacji po stronie biblioteki z której korzystaliśmy w celu generowania kodu Cypher z GraphQL, musieliśmy zrezygnować z automatycznej generacji kodu GraphQL, a więc byliśmy zmuszeni stworzyć własne mutacje korzystając z języka Cypher (kod w tym języku został wbudowany w mutacje GraphQL). Kod Cypher służący do pobierania danych jest generowany automatycznie przez w/w bibliotekę.

Ten projekt składa się z dwóch części:

1. Framework – główna część tego projektu, w której dzieje się “magia”;
2. Backend – DronkaShop, czyli przykład użycia naszego frameworka. Przykład ten składa się z dwóch części:
    a. Kod w języku Kotlin
    b. Schemat, zapytania, mutacje itd. – w językach GraphQL i Cypher

Zasada działania

Jak już zostało wspomniane wcześniej, framework, który stworzyliśmy jest półautomatyczny – czyli taki, który wymaga żeby pewna część kodu została napisana przez programistę (jak już zostało napisane wyżej, w powodu pewnych ograniczeń, z którymi się zetknęliśmy, zrezygnowaliśmy z automatycznej generacji kodu GraphQL).

A mianowicie trzeba napisać:

1. Schema – czyli schemat bazy danych (GraphQL)
2. Queries – czyli kod zapytań (GraphQL, właśnie ten kod mógłby zostać wygenerowany automatycznie, bo w tym przypadku translator działał dość dobrze, ale już zabrakło nam czasu)
3. Mutations – czyli kod mutacji, który trzeba napisać w dwóch językach: GraphQL (“frontend”) i Cypher (“backend”)

Przykładowa implementacja w/w elementów znajduje się tu: Schemat, zapytania, mutacje.

Zasada działania naszego frameworka głównie opiera się na refleksji, za pomocą której zarówno tworzymy obiekty (dalej zwane Entity lub Encjami), jak i aktualizujemy informacje / wykrywamy zmiany (a co za tym idzie, aktualizujemy tylko zmienione własności / relacje). Refleksja jest używana również m.in. dla ładnego i czytelnego wypisywania encji.

Możliwości

Żeby skorzystać z możliwości naszego frameworka, należy stworzyć klasę dziedziczącą z `Entity`, a następnie podać niezbędne elementy wspomniane w rozdziale Zasada działania.

Encja

Encją jest każda klasa dziedzicząca po `Entity`, która udostępnia metody służące do zarządzania encjami, takie jak np. `pull()`, `push()`, `persist()`. Również warto zwrócić uwagę na obiekt `EntityProvider`, która udostępnia dodatkowe funkcje do zarządzania encjami, takie jak np. pobieranie encji wg id.

System encji w frameworku jest zaimplementowany w taki sposób, że żeby uniknąć niepotrzebnego wielokrotnego pobierania danych, wszystkie już pobrane encje są przechowywane, tzn. przy poprawnym korzystaniu z frameworka, w tym samym czasie może istnieć dokładnie jedna instancja encji z tym samym id.

Pobieranie encji

Żeby mieć możliwość pobierania encji, należy w klasie dziedziczącej z `Entity` zaimplementować własność `updatePropertiesQuery`, czyli np.:

```
override val updatePropertiesQuery: GraphQLQuery =
    Resource.gets("/query/category/CategoryUpdateProperties.graphql")
```

Uwaga: żeby mapowanie się udało, nazwa mapowanej zmiennej danego typu w pliku schematu bazy (`schema.graphql`) musi się pokrywać z nazwą zmiennej w zaimplementowanej klasie.

Jeżeli nie chcemy, aby zmienna została zmapowana, należy użyć anotacji `@Ignore`.

Do pobierania encji służy metoda `pull()`.

Obsługiwane typy:

- `ID`
- `Int`
- `Float`
- `Boolean`
- `String`
- `LocalDateTime`
- `ToOneRelation`
- `ToManyRelation`

Modyfikowanie własności

Żeby mieć możliwość modyfikowania własności encji, należy w klasie dziedziczącej z `Entity` zaimplementować własność `mutatePropertiesQuery`, czyli np.:

```
override val mutatePropertiesQuery: GraphQLQuery =
    Resource.gets("/mutation/category/CategoryMutateProperties.graphql")
```

Zmodyfikowane własności zostaną wykryte automatycznie i wysłane na serwer po wywołaniu metody `push()`.

Modyfikowanie relacji

Żeby mieć możliwość modyfikowania relacji, należy w klasie dziedziczącej z `Entity` zaimplementować własność `mutateRelationsQuery`, czyli np.:

```
override val mutateRelationsQuery: GraphQLQuery =
   Resource.gets("/mutation/category/CategoryMutateRelations.graphql")
```

Zmodyfikowane relacje (to-one i to-many) zostaną wykryte automatycznie i wysłane na serwer po wywołaniu metody `push()`.

Utrwalanie Entity

Żeby utrwalić encję, należy w klasie dziedziczącej z `Entity` zaimplementować własność `createNodeQuery`, czyli np.:

```
override val createNodeQuery: GraphQLQuery =
   Resource.gets("/mutation/category/CategoryCreate.graphql")
```

Encja zostanie utrwalona po wywołaniu metody `persist()` albo `push()` (w tym przypadku `push()` zorientuje się, że ta encja jeszcze nie została utrwalona i wywoła `persist()`).

Relacje

Jak już zostało wspomniane wcześniej, nasz framework obsługuje relacje typu to-one i to-many, ale tylko *jednokierunkowe* (wsparcie dwukierunkowych relacji nie jest wcale skomplikowane, lecz wymaga zaimplementowania infrastruktury podobnej do tej z której korzysta `Entity`).

Relacje w frameworku są zaimplementowane w taki sposób, że węzeł, na który wskazuje relacja, jest automatycznie pobierany wtedy i tylko wtedy, gdy próbujemy go odczytać (wywołujemy metodę `dst()`).

ToOneRelation

Żeby zamodelować relację typu to-one, wystarczy że zmienna reprezentująca relację ma zmienny (mutable) typ `ToOneRelation<T>` (warto zwrócić uwagę, że jest to typ non-nullable), czyli np.:

```
var publishedBy = ToOneRelation.create<User>()
```

W celu pobrania wartości, na którą wskazuje relacja, należy użyć metody `dst()`, zwracającej tę wartość i pobierającej jej dane w razie konieczności (może to nie być konieczne jeżeli dane zostały pobrane w dowolny inny sposób, niekoniecznie przez wywołanie metody `dst()`).
Jest to alias do typu Relation.

ToManyRelation

Żeby zamodelować relację typu to-many, wystarczy że zmienna reprezentująca relację ma zmienny (mutable) typ `ToManyRelation<T>` (warto zwrócić uwagę, że jest to typ non-nullable), czyli np.:

```
var belongsTo = ToManyRelation<Category>()
```

Jest to lista obiektów typu Relation, a więc wszystko co zostało powiedziane w rozdziale ToOneRelation jest prawdziwe dla elementów tej listy.

Modyfikacja zachowania requestów do serwera

Framework pozwala na zaimplementowanie własnych strategii tworzenia requestów do serwera. Żeby stworzyć własną, wystarczy stworzyć klasę dziedziczącą po `QueryRunner` i zaimplementować abstrakcyjne metody.

Już zaimplementowane strategie tworzenia requestów:

1. `SessionQueryRunner` – domyślna strategia
2. `TransactionQueryRunner` – strategia umożliwiająca tworzenie transakcji. Udostępnia dodatkowo metody `commit()` i `rollback()`. Uwaga: używając tę strategię należy pamiętać o jej skomitowaniu.

W celu skorzystania z tej możliwości, należy przekazać instancję wybranego runnera do funkcji pobierających / zapisujących dane jako argument `runner` (domyślnie jest ustawiony na defaultowy runner).

Architektura frameworku

Implementacja frameworku znajduje się w pakiecie `pl.edu.agh.db2.dronkashop.framework`.

Poniżej znajduje się opis najbardziej istotnych pakietów.

`entity`

Entity

- Klasa abstrakcyjna
- Klasa bazowa dla wszystkich encji
- Posiada metody umożliwiające zarządzanie encjami
- Jest odpowiedzialna za tworzenie, pobieranie i aktualizowanie encji

Do czego ta klasa służy zostało również omówione w rozdziale Możliwości: Encja.

Relation

- Klasa szablonowa reprezentująca pojedynczą relację
- Przechowuje docelową encję
- Docelowa encja może zostać pobrana za pomocą metody `dst()`

- Id docelowej encji (bez konieczności pobierania jej danych) może zostać pobrane za pomocą metody `dstId()`

Ta klasa została również omówiona w rozdziale Możliwości: Relacje.

ToOneRelation

Alias do `Relation`.

ToManyRelation

Klasa szablonowa. Lista `Relation`.

`provider`

DBProvider

Obiekt odpowiadający za połączenie z bazą danych Neo4J.

GraphQLProvider

Obiekt odpowiadający za przekształcenie GraphQL kodu na Cypher – język z którego korzysta Neo4J.

EntityProvider

Główna klasa służąca do zarządzania wszystkimi Entity. Przechowuje, pobiera, aktualizuje encje. Do czego ta klasa służy zostało również omówione w rozdziale Możliwości: Encja.

`runner`

Więcej informacji na temat tego pakietu można znaleźć w rozdziale Możliwości: Modyfikacja zachowania requestów do serwera.

QueryRunner

- Klasa abstrakcyjna
- Klasa bazowa dla wszystkich runnerów

SessionQueryRunner

- Domyślny runner
- Korzysta z `org.neo4j.driver.Session`

TransactionQueryRunner

- Umożliwia tworzenie transakcji
- Udostępnia dodatkowo metody `commit()` i `rollback()`
- Należy pamiętać o skomitowaniu

Architektura backendu

Implementacja frameworku znajduje się w pakiecie `pl.edu.agh.db2.dronkashop.backend`.

Backend jest zaprojektowany w taki sposób, że każdemu węzłowi w bazie danych Neo4J odpowiada pewna klasa w Kotlinie; niektórym relacjom (tym, które oprócz nazwy mają dodatkowe właściwości) odpowiada również klasa w Kotlinie.

`entity`

Ten pakiet zawiera wszystkie zaimplementowane encje.

Item

- Klasa reprezentująca przedmiot

Category

- Klasa reprezentująca kategorię

User

- Klasa reprezentująca użytkownika

Role

- Klasa reprezentująca rolę

Order

- Klasa reprezentująca zamówienie

OrderedItem

- Klasa reprezentująca własności relacji `(:Order)-[:Includes]->(:Item)`

Payment

- Klasa reprezentująca płatność

`provider`

W tym pakiecie znajdują się klasy służące do pobierania encji o kastomowych (tzn. np. pobranie encji wg id) własnościach z bazy.

ItemProvider

Obiekt udostępniający następujące metody:

- `getById(): Item`
- `getByName(): Item`

CategoryProvider

Obiekt udostępniający następujące metody:

- `getById(): Category`
- `getByName(): Category`

UserProvider

Obiekt udostępniający następujące metody:

- `getById(): User`
- `getByName(): User`

RoleProvider

Obiekt udostępniający następujące metody:

- `getById(): Role`
- `getByName(): Role`

OrderProvider

Obiekt udostępniający następujące metody:

- `getById(): Order`

PaymentProvider

Obiekt udostępniający następujące metody:

- `getById(): Payment`

Architektura bazy

Baza zawiera następujące węzły:

1. Item
2. Category
3. User
4. Role
5. Order
6. Payment

Item

Reprezentuje sprzedawany przedmiot.

Pola:

- `name: String!`
- `description: String!`
- `quantity: Int!`
- `price: Float!`
- `date: DateTime!`

- `isDiscontinued: Boolean!`

Związki:

→ `BelongsTo: [Category!]!`
→ `PublishedBy: User!`

Category

Reprezentuje kategorię.

Pola:

- `name: String!`
- `description: String!`

Związki:

→ `Includes: [Item!]!`

User

Reprezentuje użytkownika. Użytkownikiem jest zarówno osoba fizyczna jak i prawna. Typ osoby definiuje jej rola.

Pola:

- `displayName: String!`
- `login: String!`
- `passwordHash: String!`
- `email: String!`
- `registrationDate: Date!`

Związki:

→ `BelongsTo: Role!`
→ `Has: [Order!]!`
→ `Sells: [Item!]!`

Role

Reprezentuje rolę.

Pola:

- `name: String!`
- `description: String!`

Związki:

→ `Includes: [User!]!`

Przykładowe role:

- Client
- VerifiedClient (może sprzedawać)
- Moderator

- Administrator

Order

Reprezentuje zamówienie.

Pola:

- `isPayed: Boolean!`
- `isCancelled: Boolean!`
- `date: DateTime!`

Związki:

→ `Includes: [Item!]!`
  - `quantity: Int!`

→ `By: User!`

→ `PayedWith: Payment!`

Payment

Reprezentuje płatność.

Pola:

- `date: DateTime!`
- `amount: Float!`
- `type: String!`

Związki:

→ `belongsTo: Order!`

Przykłady

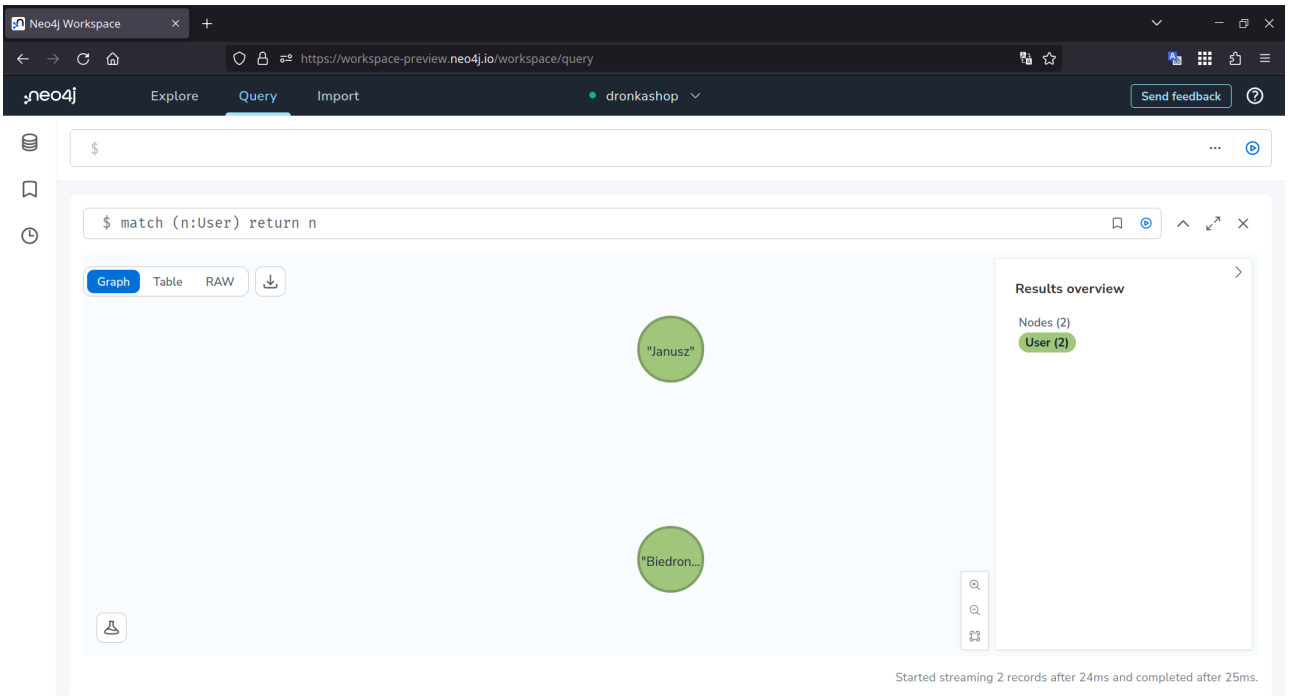
Przykładem korzystania z zaimplementowanego frameworka jest to, co nazywamy backendem – czyli DronkaShop. Poniżej są przykłady oparte o bazę DronkaShop.

Query + Mutation

Załóżmy, że chcemy pobrać encję reprezentującą pewnego użytkownika. Żeby zademonstrować jak działa framework, najpierw musimy wiedzieć ID tego użytkownika. W tym celu otwieramy stronę https://workspace-preview.neo4j.io/workspace/query, łączymy się z bazą i wykonujemy:

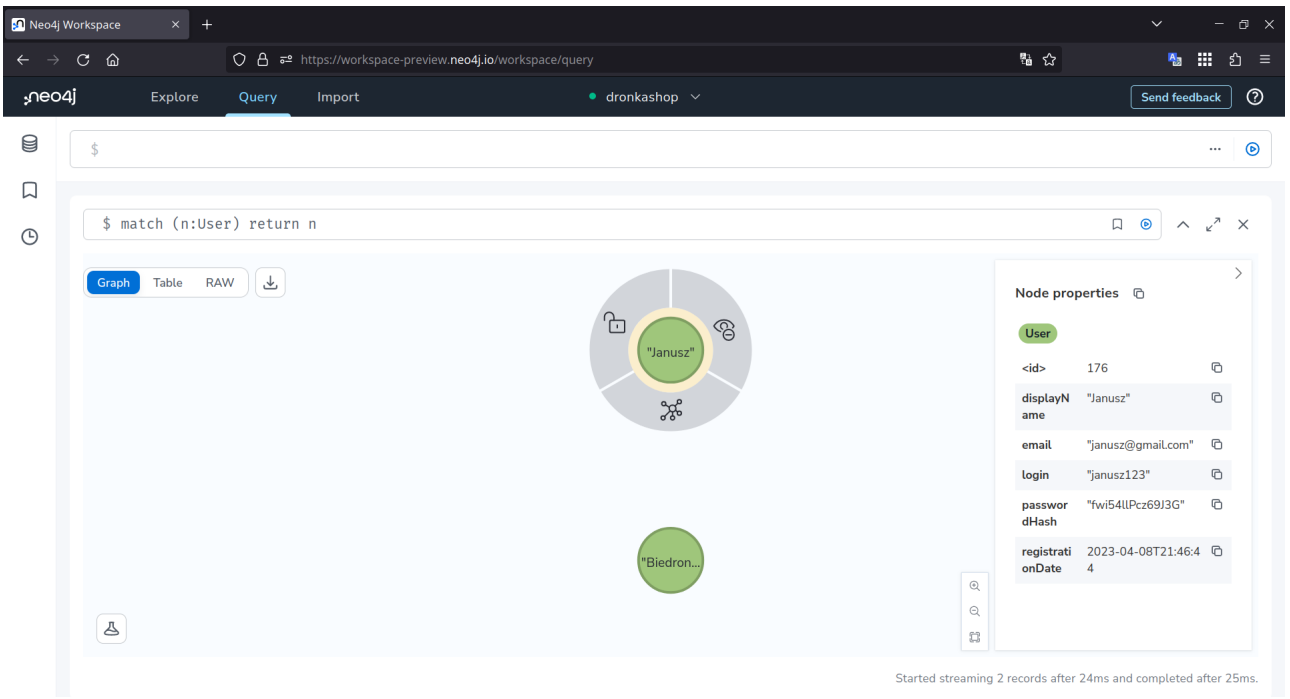
```
match (n:User) return n
```

Przykładowy wynik:



Rys. 1. Użytkownicy

Klikając na użytkownika *Janusz* możemy zobaczyć jego ID:



Rys. 2. Własności węzła reprezentującego użytkownika *Janusz*

A więc już wiemy ID.

Wiedząc ID, możemy korzystając z `UserProvider` w łatwy sposób pobrać dane użytkownika:

```
val user176 = UserProvider.getById(ID(176))
println(user176)
```

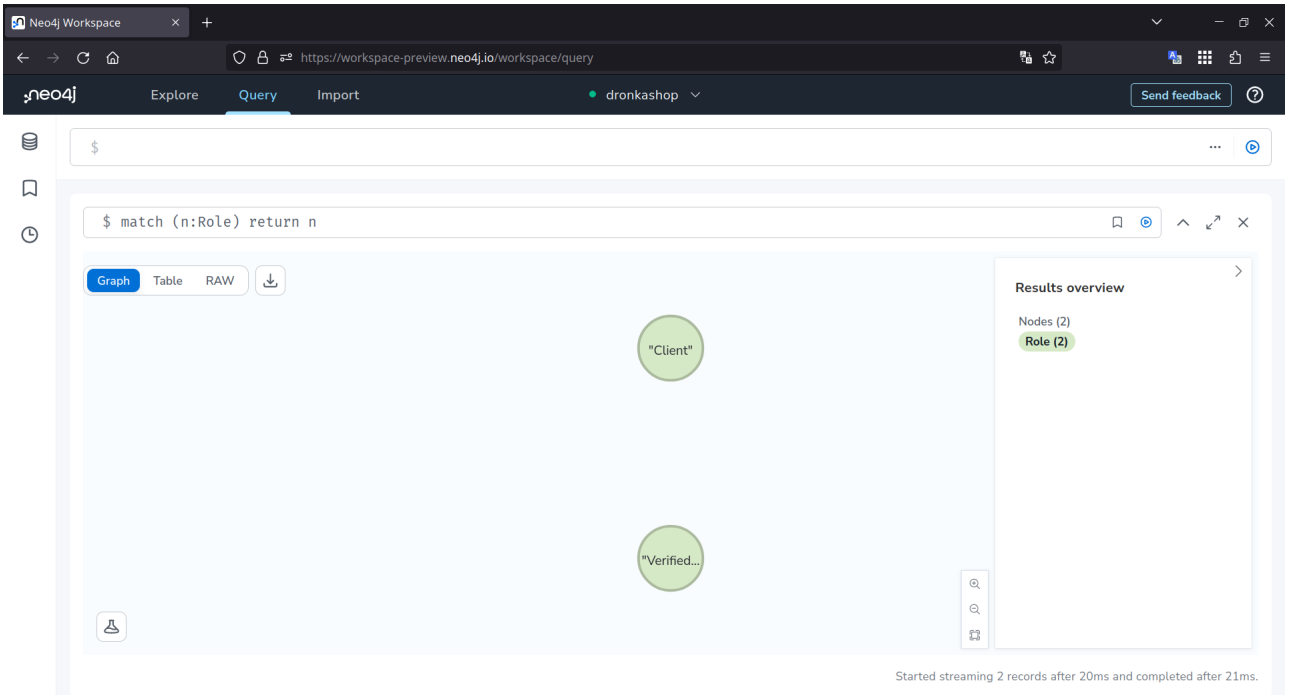
Wynik:

```
User {
    belongsTo: Relation(dst=null)
    displayName: Janusz
    email: janusz@gmail.com
    has: []
    login: janusz123
    registrationDate: 2023-04-08T21:46:44
    sells: []
    id: ID(value=176)
}
```

Widzimy, że użytkownik nie jest przepisany do żadnej roli. Możemy to w łatwy sposób naprawić, tworząc odpowiednią relację. Jak i poprzednio, najpierw wyświetlimy istniejące w bazie role:

```
match (n:Role) return n
```

Wynik:



Rys. 3. Dostępne role

Tym razem pobierzemy encję wg nazwy, czyli skorzystamy z funkcji `getByName()` obiektu `RoleProvider`. Uwaga: ta funkcja zwraca listę encji, ale w naszym przypadku potrzebujemy tylko pierwszy element z tej listy, a więc w tym celu korzystamy z tzw. destructing declaration:

```
val (roleClient) = RoleProvider.getByName("Client")
println(roleClient)
```

Wynik:

```
Role {
    description: Moze kupować produkty
    includes: []
```

```
    name: Client
    id: ID(value=177)
}
```

Już mamy zarówno klienta jak i rolę. A więc już możemy stworzyć relację typu to-one:

```
// tworzymy relację user --> role
user176.apply {
   assert(!isModified()) // Dane użytkownika jeszcze nie zostały zmienione
   belongsTo = Relation.create(roleClient)
   assert(isModified()) // System wykrywania zmian wykrył zmianę relacji to-one
   push() // wysyłamy zmienione dane do serwera
}

// tworzymy relację role --> user
roleClient.apply {
   assert(!isModified()) // Dane roli jeszcze nie zostały zmienione
   includes.add(Relation.create(user176))
   assert(isModified()) // System wykrywania zmian wykrył zmianę relacji to-many
   push() // wysyłamy zmienione dane do serwera
}
```

W powyższym fragmencie w celu zwiększenia czytelności użyto scope functions.

Uruchamiamy kod z poprzednich punktów raz jeszcze żeby sprawdzić czy zmiany zostały utrwalone:

```
val user176 = UserProvider.getById(ID(176))
println(user176)

val (roleClient) = RoleProvider.getByName("Client")
println(roleClient)
```

Wynik:

```
User {
    belongsTo: Relation(dst=ID(value=177))
    displayName: Janusz
    email: janusz@gmail.com
    has: []
    login: janusz123
    registrationDate: 2023-04-08T21:46:44
    sells: []
    id: ID(value=176)
}
Role {
    description: Moze kupować produkty
    includes: [Relation(dst=ID(value=176))]
    name: Client
    id: ID(value=177)
}
```

Czyli jak widać, wszystko jest OK.

W analogiczny sposób możemy zmieniać pola encji.

Lazy loading

Ostatnio dla użytkownika otrzymaliśmy następujący wynik:

```
User {
    belongsTo: Relation(dst=ID(value=177))
    displayName: Janusz
    email: janusz@gmail.com
    has: []
    login: janusz123
    registrationDate: 2023-04-08T21:46:44
    sells: []
    id: ID(value=176)
}
```

Zwróćmy uwagę na relację `belongsTo`. Chcemy pokazać, że dane tej relacji są pobierane tylko wtedy, gdy próbujemy ją odczytać. W tym celu skorzystamy z "niskopoziomowego" obiektu `EntityProvider` (uwaga: robimy to tylko w celach demonstracyjnych, bo wiemy, co robimy), a dokładniej mówiąc poprosimy go aby zwrócił encję bez automatycznego pobrania jej danych w przypadku gdy jest ona *incomplete*:

```
val user176 = UserProvider.getById(ID(176))
println(user176)

// tworzymy nowe scope, żeby nie zaśmiecać globalne
run {
   val roleId = user176.belongsTo.dstId() // pobieramy tylko same id
   val role = EntityProvider.entityById<Role>(roleId, incomplete = true)

   assert(role.isIncomplete())
   println("Encja jest niepełna: ${role.isIncomplete()}")

   user176.belongsTo.dst() // odczytujemy relacje

   assert(!role.isIncomplete())
   println("Encja jest niepełna: ${role.isIncomplete()}")
}
```

Wynik:

```
User {
    belongsTo: Relation(dst=ID(value=177))
    displayName: Janusz
    email: janusz@gmail.com
    has: []
    login: janusz123
    registrationDate: 2023-04-08T21:46:44
    sells: []
    id: ID(value=176)
}
```

```
Encja jest niepełna: true
Encja jest niepełna: false
```

A więc widzimy, że po odczytaniu relacji dane automatycznie się pobrały. W tym przypadku również zostało pokazane, że w tym samym czasie istnieje dokładnie jedna encja o tym samym ID: przypominamy, że `user176.belongsTo.dst()` zwraca encję, na którą wskazuje dana relacja, ale w powyższym przykładzie zignorowaliśmy wynik tej funkcji, bo i tak już mamy referencję na tę encję, czyli `role`.

Warto dodać, że żeby niekompletna (incomplete) encja stała się kompletną wystarczy wywołać metodę `pull()`.

Tworzenie węzłów

Załóżmy, że tym razem chcemy dodać do bazy nowy węzeł – produkt. W tym celu wystarczy stworzyć instancję typu `Item`, wypełnić ją danymi i relacjami a następnie wywołać `push()` albo `persist()`, czyli:

```
Item().apply {
   name = "Kawa z mlekiem"
   description = "Smaczna kawa z dodatkiem mleka"
   quantity = 10
   price = 7.5f
   date = LocalDateTime.now()
   isDiscontinued = false

   // encja nie ma przypisanego id
   assert(id == ID.INVALID)

   // utrwalamy encję
   persist() // albo push()

   // po utrwaleniu encja już ma przypisane id
   assert(id != ID.INVALID)

   println(this)
}
```

Wynik:

```
Item {
    belongsTo: []
    date: 2023-06-13T11:33:29.629087435
    description: Smaczna kawa z dodatkiem mleka
    isDiscontinued: false
    name: Kawa z mlekiem
    price: 7.5
    publishedBy: Relation(dst=null)
    quantity: 10
    id: ID(value=107)
}
```

A więc, jak widać, nowo utworzona encja została zapisana na serwerze i otrzymała ID=107.

Wnioski

- Inspirując się takimi frameworkami jak EntityFramework i Hibernate stworzyliśmy własny w języku Kotlin (JVM) służący do ułatwienia zarządzania danymi w bazie danych;
- Do przetwarzania danych użyliśmy GraphQL, jako bazę danych – Neo4J, grafową bazę napisaną w języku Java;
- Zaimplementowany framework ma możliwość pobierania encji z serwera, aktualizowania zarówno własności jak i relacji, tworzenia nowych encji;
- Framework wspiera jednokierunkowe relacje to-one i to-many;
- System wykrywania zmian pozwala aktualizować tylko te dane, które zostały zmienione;
- Żeby nie pobierać zbędnych danych, korzystamy z Lazy loadingu – dane są pobierane tylko wtedy, gdy próbujemy je odczytać;
- Stworzony framework ma kilka wad, które przy obecnej architekturze nie są trudne do usunięcia i wynikają głównie z braku czasu;
- Framework może zostać ulepszony następująco:
    - Autogeneracja schematu bazy w oparciu o zaimplementowane w Kotlinie klasy;
    - Autogeneracja kodu GraphQL
    - Autogeneracja kodu Cypher (jest potrzebna z tego powodu, że biblioteka, z której korzystamy, ma dość słabe wsparcie mutacji, a więc to jest główna przyczyna czemu autogeneracja schematu i GraphQL zapytań/mutacji nie została zaimplementowana);
    - Wsparcie dwukierunkowych relacji (jak już zostało wspomniane w rozdziale Możliwości: Relacje, nie jest to trudna rzecz do zaimplementowania, lecz wymaga czasu na dobre projektowanie i implementację);
    - Dodatkowe anotacje, umożliwiające m.in. modyfikowanie nazw mapowanych własności / relacji / encji;
    - Różne sposoby mapowania dziedziczenia;
    - Wsparcie nullable properties i to-one relacji;