dokumentacja-projektu

Mechanizm działania mapowania obiektowo-relacyjnego

Mapowanie obiektowo-relacyjne (ORM) polega na odzwierciedleniu obiektów utworzonych w obiektowym języku programowania (np. JAVA) na tabele w relacyjnej bazie danych (np. PostgreSQL, Oracle,...).

Do najważniejszych elementów narzędzia ORM należą:

- Tworzenie zapytań SQL w celu tworzenia tabel w bazie danych oraz relacji pomiędzy nimi
- Tworzenie zapytań SQL w celu pobierania, wstawiania, aktualizacji oraz kasowania danych w bazie danych
- Mapowanie danych z obiektów na dane do tabeli lub tabel w bazie danych
- Mapowanie danych z tabeli lub tabel w bazie danych na obiekty

ORM tworzy dodatkową warstwę abstrakcji pomiędzy kodem aplikacji a bazą danych, dzięki czemu programista może się skupić na tworzeniu aplikacji w języku obiektowym bez konieczności pisania zapytań w języku SQL. Wywołując metody z narzędzia ORM, programista jest w stanie operować na danych przechowywanych w bazie danych w taki sam sposób, w jaki operowałby na obiektach w kodzie aplikacji.

Gotowe narzędzia ORM, takie jak Hibernate, pozwalają na pracę z różnymi dostawcami baz danych, takimi jak Oracle, PostgreSQL, H2, itp. Jedyne co musi zrobić programista, to zmienić typ obsługiwanej przez Niego bazy danych w konfiguracji Hibernate, natomiast nie musi zmieniać żadnej logiki aplikacji w celu dostosowania jej do innej bazy danych. Każda baza danych może, lecz nie musi, w inny sposób obsługiwać niektóre operacje na danych, a niektórych może w ogóle nie obsługiwać. Za te wszystkie zmienne operacje odpowiada właśnie ORM, dzięki czemu zmiana bazy danych w przyszłości, lub zmiana bazy danych w środowiskach developerskich, testowych, produkcyjnych, może odbywać się zazwyczaj poprzez zmianę kilku wartości w konfiguracji narzędzia, takich jak adres do bazy, użytkownik, hasło oraz typ bazy danych.

Zalety narzędzia ORM

- uniezależnienie od bazy danych - można pracować z różnymi systemami bazodanowymi bez konieczności znajomości ich specyfikacji
- brak lub minimalna ilość zapytań SQL - wiele zapytań SQL jest już zaimplementowana w narzędziu ORM, dzięki czemu programista nie musi tracić czasu na ich pisanie, wystarczy wywołać odpowiednią metodę z narzędzia ORM
- ochrona przed SQL Injection - dobrze napisane narzędzie ORM wprowadza dodatkowe algorytmy, chroniące przed atakami typu SQL Injection

- Ilość gotowych narzędzi - na rynku istnieje wiele gotowych rozwiązań ORM, dzięki czemu każdy znajdzie coś, co najbardziej będzie odpowiadało wiedzy lub wymaganiom projektowym. Przykładowe narzędzia to Hibernate (JAVA), Sequelize (Node.js), EntityFramework (.NET). Praktycznie każdy z popularnych języków programowania ma jedną lub więcej implementacji narzędzia ORM

Wady narzędzia ORM

- mniejsza wydajność - narzędzia ORM wprowadzają dodatkowe operacje oraz dodatkową warstwę abstrakcji, przez co aplikacja staje się wolniejsza. Jest to jednak na tyle niezauważalne, że nie bierze się tego aż tak pod uwagę. Dodatkowo, w niektórych przypadkach, narzędzia ORM mogą wykonywać nadmiarowe zapytania SQL, przez co jest zwiększony ruch pomiędzy bazą danych a aplikacją
- poziom skomplikowania - często zdarza się, że narzędzia ORM są trudniejsze w użyciu niż zapytanie SQL. Jednak jeśli już pozna się dobrze możliwości danego narzędzia, praca z nim staje się o wiele łatwiejsza
- zmiana narzędzia ORM - często zdarza się, że zmiana jednego narzędzia ORM na inne niesie ze sobą wiele problemów oraz wiele dodatkowych zmian w kodzie aplikacji, dlatego trzeba dobrze się zastanowić i przemyśleć używanie danego narzędzia, gdyż będzie to prawdopodobnie używane już przez większą część życia aplikacji
- większe zużycie pamięci - narzędzia takie jak Hibernate, często posiadają własny rodzaj Cache, do którego zapisują dane z bazy danych w celu przyspieszenia operacji na bazie oraz zmniejszeniu ilości zapytań do bazy w celu przyspieszenia działania całej aplikacji. Owa optymalizacja wiąże się niestety z tym, że te dane trzeba przechowywać w pamięci, przez co wymagana jest jej większa ilość w systemie operacyjnym

Porównanie implementowanego rozwiązania z Hibernate

Funkcjonalność

Implementowane rozwiązanie

Zaimplementowane rozwiązanie posiada podstawowe funkcjonalności narzędzia ORM, którymi są:

- skanowanie klas w poszukiwaniu encji bazodanowych
- tworzenie tabel w bazie danych na podstawie znalezionych encji
- tworzenie kluczy głównych
- tworzenie tabel przejściowych przy relacjach wiele do wielu
- tworzenie relacji jeden do jednego, jeden do wielu oraz wiele do wielu
- zapis danych w bazie
- odczyt danych z bazy

- aktualizacja danych w bazie
- kasowanie danych z bazy

Hibernate

Hibernate jest rozbudowanym narzędziem ORM, które jest na rynku już od około 24 lat. Jest szeroko używany w środowisku języka JAVA, przez co doczekał się wielu wersji i jest nadal rozwijany i utrzymywany. Hibernate posiada wszystkie funkcjonalności, które oferuje moje rozwiązanie, jednak posiada również szereg innych funkcjonalności, takich jak:

- obsługa dziedziczenia w JAVIE
- obsługa obiektów JAVY jako dodatkowe kolumny w tabeli (`@Embeddable` oraz `@Embedded`)
- przechowywanie danych w lokalnej pamięci podręcznej aby przyspieszyć komunikację z bazą danych oraz zmniejszyć ilość połączeń
- leniwe ładowanie danych
- obsługa transakcji
- niestandardowe zapytania do bazy danych (możliwość pisania własnych zapytań w języku SQL, lub HQL
- opiera się na implementacji JPA
- tworzy sesje bazodanowe
- obsługuje wiele połączeń jednocześnie

Implementowane rozwiązanie

Implementacja rozwiązania została napisana w języku JAVA z wykorzystaniem systemu refleksji. W celu poprawnego wykorzystania tej techniki, zaprojektowane zostały odpowiednie adnotacje oraz ich procesory, czyli klasy, które operują na klasach z adnotacjami i wykonują operacje, które są powiązane z konkretną adnotacją. Na przykład adnotacja `@Entity` służy do określenia klasy jako encję bazodanową, na jej podstawie, klasa procesora określa nazwę tabeli w bazie danych oraz, bazując na jej polach, generuje kolumny w tej tabeli.

Wykorzystywaną bazą danych w projekcie jest *PostgreSQL*, jednak kod został przygotowany w taki sposób, aby w łatwy sposób można było podmienić bazę danych na inną. Wystarczy zaimplementować interfejs `SqlDialect` i dodać implementację tego interfejsu jako parametr funkcji inicjującej `initialize` klasy `DatabaseInitializer`.

Zaimplementowane adnotacje

- `@Entity` - ustawiana na poziomie klasy, określa, że dana klasa reprezentuje encję i odpowiadającą jej tabelę w bazie danych. Adnotacja posiada parametr `name`, który określa docelową nazwę tabeli w bazie danych, kiedy jej nie ma, lub jest pusta, jest brana wartość domyślna, czyli nazwa klasy, która jest przekształcona z typu *CammelCase* na nazwę z podkreśleniami - np. UserEntity zostanie zamienione na USER_ENTITY.

- `@Column` - ustawiana na poziomie pola w klasie, określa dodatkowe dane do pola, które zostaną odwzorowane w bazie danych. Adnotacja posiada 3 parametry:
  - `name` - określa docelową nazwę kolumny w taki sam sposób jak nazwa w tabeli z adnotacji `@Entity`
  - `unique` - jeśli wartość jest ustawiona na `true`, kolumna w bazie danych dostaje ograniczenie o wartości unikalnej
  - `nullable` - jeśli wartość jest ustawiona na `true`, pozwala na umieszczanie wartości `null` w bazie danych, w przeciwnym razie kolumna dostaje ograniczenie `not null`
- `@Id` - ustawiana na poziomie pola w klasie, które musi być typu `Long` określając w ten sposób, że dane pole jest identyfikatorem wiersza w tabeli w bazie danych
- `@OneToOne` - ustawiana na poziomie pola w klasie, określa relację jeden do jednego pomiędzy dwiema encjami w kodzie JAVA oraz pomiędzy dwiema tabelami w bazie danych. Adnotacja posiada 2 parametry:
  - `name` - określa docelową nazwę kolumny (podobnie jak przy adnotacji `@Column`
  - `entity` - określa klasę docelową, która jest używana przy mapowaniu danych z bazy do obiektów w kodzie JAVA
- `@OneToMany` - ustawiana na poziomie pola w klasie, określa relację jeden do wielu pomiędzy dwiema encjami w kodzie JAVA. W tabeli bazy danych nie jest generowana kolumna z wartościami określającymi relację (brak kluczy obcych do drugiej tabeli). Adnotacja posiada 3 argumenty:
  - `name` - określa docelową nazwę kolumny (podobnie jak przy adnotacji `@Column`
  - `entity` - określa klasę docelową, która jest używana przy mapowaniu danych z bazy do obiektów w kodzie JAVA
  - `mappedBy` - określa nazwę pola w drugiej encji, które służy do przechowywania referencji do obiektu tej encji
- `@ManyToOne` - ustawiana na poziomie pola w klasie, określa relację wiele do jednego pomiędzy dwiema encjami w kodzie JAVA oraz pomiędzy dwiema tabelami w bazie danych. W tej tabeli jest ustawiany klucz obcy do drugiej tabeli, dzięki czemu wyeliminowane zostało tworzenie dodatkowej tabeli łączącej pomiędzy tymi dwiema tabelami w bazie danych. Adnotacja posiada 3 argumenty:
  - `name` - określa docelową nazwę kolumny (podobnie jak przy adnotacji `@Column`
  - `entity` - określa klasę docelową, która jest używana przy mapowaniu danych z bazy do obiektów w kodzie JAVA
  - `mappedBy` - określa nazwę pola w drugiej encji, które służy do przechowywania referencji do obiektu tej encji
- `@ManyToMany` - ustawiana na poziomie pola w klasie, określa relację wiele do wielu pomiędzy dwiema encjami w kodzie JAVA oraz pomiędzy dwiema tabelami w bazie danych. W celu utworzenia takiej relacji, tworzona jest dodatkowa tabela łącząca w bazie danych, której nazwa jest połączeniem nazw dwóch tabel oddzielonych znakiem podkreślenia, posiadająca dwie kolumny, której wartościami są klucze obce tych dwóch tabel. Adnotacja posiada 3 argumenty:

    - `name` - określa docelową nazwę kolumny (podobnie jak przy adnotacji `@Column`
    - `entity` - określa klasę docelową, która jest używana przy mapowaniu danych z bazy do obiektów w kodzie JAVA
    - `mappedBy` - określa nazwę pola w drugiej encji, które służy do przechowywania referencji do obiektu tej encji

Testowanie rozwiązania

W celu przetestowania zaimplementowanego rozwiązania, zostały przeprowadzone 3 rodzaje testów:

- testy jednostkowe - w celu przetestowania, czy generowane polecenia w języku `SQL` są poprawne. Testy te zostały zaimplementowane zgodnie z metodyką TDD
- testy integracyjne (manualne, poprzez uruchomienie programu) - w celu przetestowania, czy tabele są tworzone w sposób poprawny oraz czy operacje na encjach są wykonywane w sposób poprawny bezpośrednio w bazie danych
- testy manualne - w celu przetestowania rozwiązania w sposób manualny

Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %
all classes	42,9% (12/28)	29,5% (66/224)	20,1% (82/408)	20% (249/1248)

Coverage Breakdown

Package	Class, %	Method, %	Branch, %	Line, %
pl.kielce.tu.orm	0% (0/2)	0% (0/21)	0% (0/24)	0% (0/191)
pl.kielce.tu.orm.annotations.processors	100% (4/4)	96,4% (27/28)	79,5% (35/44)	93,9% (153/163)
pl.kielce.tu.orm.cache	100% (2/2)	100% (15/15)	87,5% (7/8)	100% (21/21)
pl.kielce.tu.orm.classloader	100% (1/1)	100% (6/6)	75% (9/12)	90% (27/30)
pl.kielce.tu.orm.config	0% (0/1)	0% (0/6)		0% (0/7)
pl.kielce.tu.orm.connector	0% (0/1)	0% (0/6)	0% (0/8)	0% (0/26)
pl.kielce.tu.orm.definitions	100% (1/1)	100% (1/1)		100% (1/1)
pl.kielce.tu.orm.dialects	100% (1/1)	100% (8/8)	68,8% (11/16)	81,8% (18/22)
pl.kielce.tu.orm.entities	0% (0/5)	0% (0/49)		0% (0/67)
pl.kielce.tu.orm.entities.manytomany	0% (0/2)	0% (0/14)		0% (0/14)
pl.kielce.tu.orm.exceptions	100% (1/1)	50% (1/2)		50% (1/2)
pl.kielce.tu.orm.initializer	0% (0/1)	0% (0/9)	0% (0/2)	0% (0/43)
pl.kielce.tu.orm.repository	0% (0/1)	0% (0/5)	0% (0/10)	0% (0/36)
pl.kielce.tu.orm.repository.impl	0% (0/2)	0% (0/33)	0% (0/224)	0% (0/537)
pl.kielce.tu.orm.sql	66,7% (2/3)	38,1% (8/21)	33,3% (20/60)	31,8% (28/88)

Wynik z pokrycia kodu testami jednostkowymi

Z powyższego raportu można wywnioskować, iż pokrycie kodu nie jest na zbyt wysokim poziomie. Dzieje się tak dlatego, iż w projekcie znajdują się takie klasy jak:

- encje - czyli obiekty, które są wykorzystywane do operacji na bazie danych
- adnotacje - czyli interfejsy, które są wykorzystywane do oznaczania, który obiekt jest encja bazy danych oraz pozwalające na dodatkową konfigurację odwzorowania w bazie danych
- konfiguracje - czyli obiekty przechowujące konfigurację biblioteki

Jeśli chodzi o główny cel projektu, czyli generowanie tabel, relacji oraz wykonywanie operacji na bazie danych, został przetestowany w granicach ~90% za pomocą testów jednostkowych.

W testach jest sprawdzana poprawność generowania poleceń w języku SQL dla określonych operacji.

Obsługa bazy danych

W projekcie można znaleźć 2 rodzaje obsługi bazy danych

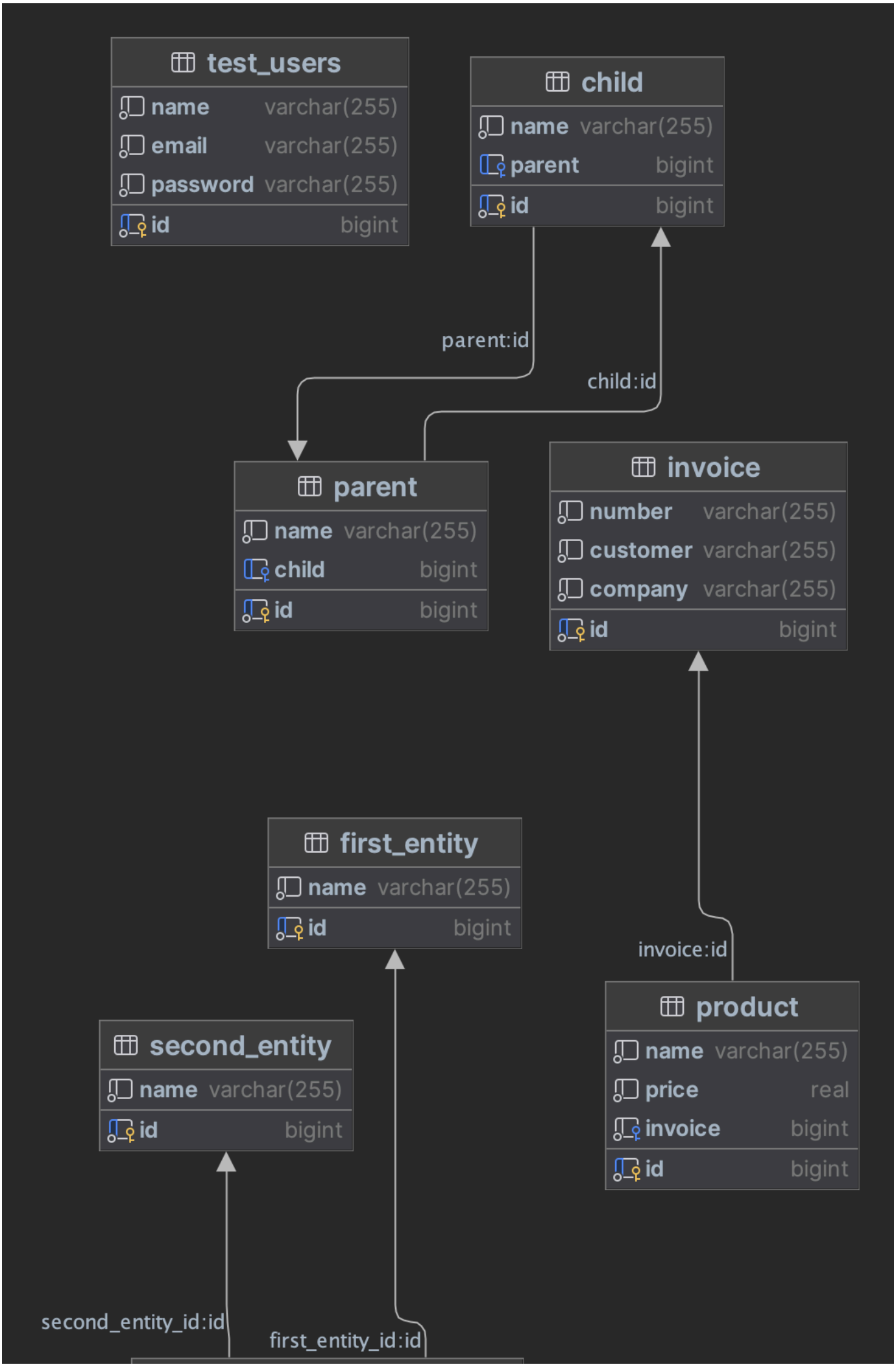
1. Generowanie tabel oraz relacji, osbywające się podczas uruchomienia aplikacji. Przy uruchomieniu, kod skanuje określony pakiet (przekazany w parametrze konstruktora) oraz jego pakiety wewnętrzne i szuka klas oznaczonych adnotacją `@Entity`. Jeśli znajdzie taką klasę, jest to sygnał, że na jej podstawie powinna zostać utworzona tabela w bazie danych. Aplikacja wtedy przystępuję do generowania tabeli oraz relacji pomiędzy tabelami, jeśli takie istnieją.
2. Operacje CRUD (Create, Read, Update, Delete) na bazie danych. Do tego celu jest wykorzystywany obiekt `RepositoryFactory`, który na podstawie przekazanej encji generuje obiekt typu `CrudRepository`, który odpowiada za obsługę encji z bazą danych. Dodatkowo istnieje możliwość napisania własnej implementacji obiektu `CrudRepository`, który można przekazać do fabryki. Przykład z utworzoną własną implementacją można zaobserwować w interfejsie `UserRepository` znajdującym się w pakiecie `pl.kielce.tu.orm.repository` oraz w klasie implementującej ten interfejs, którą jest `UserRepositoryimpl`, która znajduje się w pakiecie `pl.kielce.tu.orm.repository.impl`.

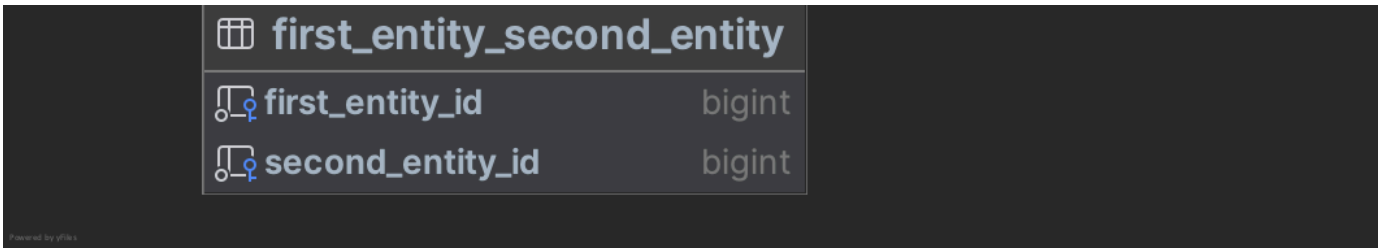
Ogólny schemat działania programu

Schemat działania programu

Schemat bazy danych

Poniżej znajduje się schemat bazy danych. Tabele (wraz z relacjami) przedstawione na schemacie zostały wygenerowane automatycznie przez zaprojektowane narzędzie.


test_users
name varchar(255)
email varchar(255)
password varchar(255)
id bigint
child
name varchar(255)
parent bigint
id bigint
parent:id
child:id
parent
name varchar(255)
child bigint
id bigint
invoice
number varchar(255)
customer varchar(255)
company varchar(255)
id bigint
first_entity
name varchar(255)
id bigint
invoice:id
product
name varchar(255)
price real
invoice bigint
id bigint
second_entity
name varchar(255)
id bigint
second_entity_id:id
first_entity_id:id


first_entity_second_entity
first_entity_id
bigint
second_entity_id
bigint