

Bazy danych - Projekt
System zarządzania kolekcją gier planszowych
Dokumentacja

Spis treści

Bazy danych - Projekt	1
System zarządzania kolekcją gier planszowych	1
Dokumentacja	1
SPOTKANIE 2.	4
Tematyka i zakres projektu	4
Zagadnienia związane z tematem	4
Funkcje bazy danych i ich priorytety	4
Technologie i narzędzia	5
Technologie i rodzaj bazy danych	5
Narzędzia	5
SPOTKANIE 3.	7
Prezentacja diagramu ERD	7
Opis tabel i ich funkcji	7
Zapytania SQL	9
Zapytania tworzące tabele	9
Zapytania wprowadzające dane	11
Przykładowe zapytania selekcyjne	12
SPOTKANIE 4.	14
Zaawansowane zapytania SQL	14
Dodawanie danych	14
Aktualizacja danych	14
Selekcja danych	16
Role i uprawnienia	18
Przykładowe zapytania z wykorzystaniem ról	19
Funkcje i procedury	20
Funkcja <code>get_game_details</code>	20
Procedura <code>add_game_review</code>	21
Funkcja <code>get_games_by_category_and_mechanic</code>	22
Procedury <code>remove_mechanic_from_game</code>	23
Procedura <code>add_mechanic_to_game</code>	23
Backup i restore	24
Backup	24
Restore	25
Wersja testowa bazy danych	27
Komunikacja z językiem programowania	28
Połączenie z bazą danych	28
Przykładowe zapytania	28
Rust ORM	31
Połączenie z bazą danych	31
Autoryzacja użytkownika	31

Menu aplikacji	32
Opcje aplikacji	34
Modele	38
PODSUMOWANIE	40
Wnioski	40
Konfrontacja z innym silnikami bazodanowymi	40
Główne różnice między PostgreSQL a MySQL	41
Wnioski z porównania	41

SPOTKANIE 2.

Tematyka i zakres projektu

Projekt skupia się na stworzenie systemu zarządzania kolekcją gier planszowych. Głównym celem projektu jest stworzenie bazy danych, która umożliwi katalogowanie gier planszowych, mechanik tych gier, recenzji oraz historii wydań.

System pozwala na zarządzanie kolekcją jednej osoby lub grupy domowników. W projekcie zostaną zaimplementowane dwie główne role użytkowników: administratora, który ma pełen dostęp do funkcji systemu, oraz zwykłego użytkownika, który może korzystać z podstawowych funkcji przeglądania bazy gier oraz ma możliwość dodania recenzji. Dzięki takiemu podejściu użytkownicy mogą bez obaw o stratę danych, zarządzać swoją kolekcją gier planszowych.

Projekt nazwałem „**GameVault**”. Nazwa nawiązuje do przechowywania gier w wirtualnym „skarbcu”.

Zagadnienia związane z tematem

- **Katalogowanie gier:** Należy opracować strukturę bazy danych, która pozwoli na przechowywanie informacji o grach planszowych, takich jak nazwa gry, gatunek, liczba graczy, czas trwania rozgrywki, opis (gdzie powinny znaleźć się dodatkowe informacje o grze).
- **Mechaniki gier:** System powinien umożliwiać przypisanie mechanik gry do danej gry planszowej. Mechaniki gier to zestaw reguł, które określają sposób rozgrywki. Każda gra może posiadać wiele mechanik.
- **Recenzje gier:** Użytkownicy systemu mogą dodawać recenzje gier planszowych. Recenzje powinny zawierać ocenę gry oraz opis. Każda gra może posiadać wiele recenzji.
- **Historia wydań:** System powinien przechowywać informacje o wydaniach danej gry planszowej. Historia wydań zawiera informacje o dacie wydania, wydawcy, numerze wydania oraz opisie gdzie można znaleźć dodatkowe informacje o wydaniu. Każda gra może posiadać wiele wydań.
- **Role użytkowników:** System powinien umożliwiać zarządzanie rolami użytkowników. Administrator ma pełen dostęp do funkcji systemu, natomiast zwykły użytkownik ma ograniczony dostęp do funkcji systemu.

Funkcje bazy danych i ich priorytety

1. Przechowywanie informacji o grach planszowych

- Priorytet: *Wysoki*
- Opis: Najważniejsza funkcjonalność systemu, która umożliwia przechowywanie informacji o grach planszowych. Bez tej funkcjonalności projekt nie ma za dużego sensu.

2. Przechowywanie informacji o mechanikach gier

- Priorytet: *Średni*
- Opis: Funkcjonalność umożliwiająca przypisanie mechanik gry do danej gry planszowej. Dzięki tej funkcjonalności opisy gier stają się bardziej kompleksowe. Każdy użytkownik z dostępem do bazy gier może dowiedzieć się, jakie mechaniki posiada dana gra i podejrzeć je podczas rozgrywki.

3. Przechowywanie informacji o recenzjach gier

- Priorytet: *Średni*
- Opis: Funkcjonalność umożliwiająca dodawanie recenzji gier planszowych. Recenzje gier są ważne dla użytkowników, którzy chcą znaleźć informacje zwrotne na temat danej gry. Dzięki tej funkcjonalności użytkownicy mogą dzielić się swoimi opiniami na temat gier planszowych.

4. Przechowywanie informacji o historii wydań gier

- Priorytet: *Niski*
- Opis: Funkcjonalność umożliwiająca przechowywanie informacji o wydaniach danej gry planszowej. Ma najniższy priorytet, ponieważ nie jest to funkcjonalność, która jest niezbędna do działania systemu. Jednakże, dzięki tej funkcjonalności użytkownicy mogą lepiej zarządzać swoją kolekcją gier planszowych.

Technologie i narzędzia

Technologie i rodzaj bazy danych

Projekt zostanie zrealizowany w oparciu o bazę danych **PostgreSQL**. Jest to jeden z najpopularniejszych silników bazodanowych, który oferuje zaawansowane funkcje i możliwości. PostgreSQL jest otwartoźródłowym systemem, co dla mnie - zwolennika takich rozwiązań, jest bardzo ważne. W projekcie wykorzystam wersję 16.2 PostgreSQL, która podczas pisania tego dokumentu jest najnowszą wersją stabilną. Użyję tego obrazu Dockerowego.

Do obsługi bazy danych posłużę terminalowy **system ORM**. Do jego stworzenia wykorzystam język programowania **Rust**. Wykorzystam też poniższe biblioteki:

- **Diesel**: ORM dla języka Rust, który umożliwia łatwe zarządzanie bazą danych PostgreSQL.
- **terminal-menu**: Biblioteka do tworzenia menu w terminalu. Umożliwia łatwe zarządzanie interfejsem użytkownika w terminalu.
- **Serde i Serde_json**: Biblioteka do serializacji i deserializacji danych w języku Rust. Umożliwia łatwe przekształcanie danych do formatu JSON.
- **Inne biblioteki**, które mogą okazać się przydatne podczas implementacji systemu.

W projekcie wykorzystam wersję 1.77.2 języka Rust. Nie jest to najnowsza wersja języka, ale jest to wersja, która jest stabilna i sprawdzona. Program będzie kompilowany do wersji wykonywalnych pod systemy Windows i Linux więc nie jest potrzebny Docker. Podczas implementacji systemu będę korzystał z najnowszych wersji bibliotek, które są dostępne w repozytorium Cargo.

Narzędzia

Przygotowanie dokumentacji projektu zostanie zrealizowane w języku **Typst**. Typst to alternatywne narzędzie do LaTeX, które umożliwia tworzenie dokumentacji w sposób tekstowy.

Do stworzenia diagramów ERD użyję narzędzia **dbdiagram.io**. Jest to narzędzie online, które wykorzystywałem już w przeszłości. W szczególności podoba mi się fakt, że diagramy tworzy się w sposób tekstowy. Dzięki temu można skupić się na projektowaniu bazy danych, a nie na rysowaniu diagramów.

Lokalne i produkcyjne środowisko projektu zostanie zrealizowane w oparciu o kontenery **Docker** i narzędzie **Docker Compose**. Docker umożliwia łatwe zarządzanie środowiskami deweloperskimi oraz produkcyjnymi. Repozytoria Dockerowe zawierają gotowe obrazy PostgreSQL i Rust, które można zmodyfikować według własnych potrzeb. Docker Compose umożliwia zarządzanie wieloma kontenerami jednocześnie za pomocą jednego pliku konfiguracyjnego `docker-compose.yml`.

Do zarządzania bazą danych użyję narzędzia **DataGrip** firmy **JetBrains**. Jest to zaawansowane narzędzie do zarządzania bazą danych, które oferuje wiele funkcji ułatwiających pracę z bazą danych. JetBrains oferuje darmową licencję dla studentów, co jest dodatkowym atutem tego narzędzia.

Do implementacji systemu ORM wykorzystam środowisko programistyczne **RustRover**. RustRover to nowe IDE firmy JetBrains, które oferuje wiele funkcji ułatwiających pracę z językiem Rust. RustRover oferuje integrację z Cargo, co umożliwia łatwe zarządzanie zależnościami w projekcie.

Wdrożenie projektu końcowego zostanie zrealizowane na platformie **Railway**. Railway umożliwia na łatwe wdrożenie kontenerów Docker na serwerach w chmurze. Ich darmowy plan wystarczy na potrzeby projektu.

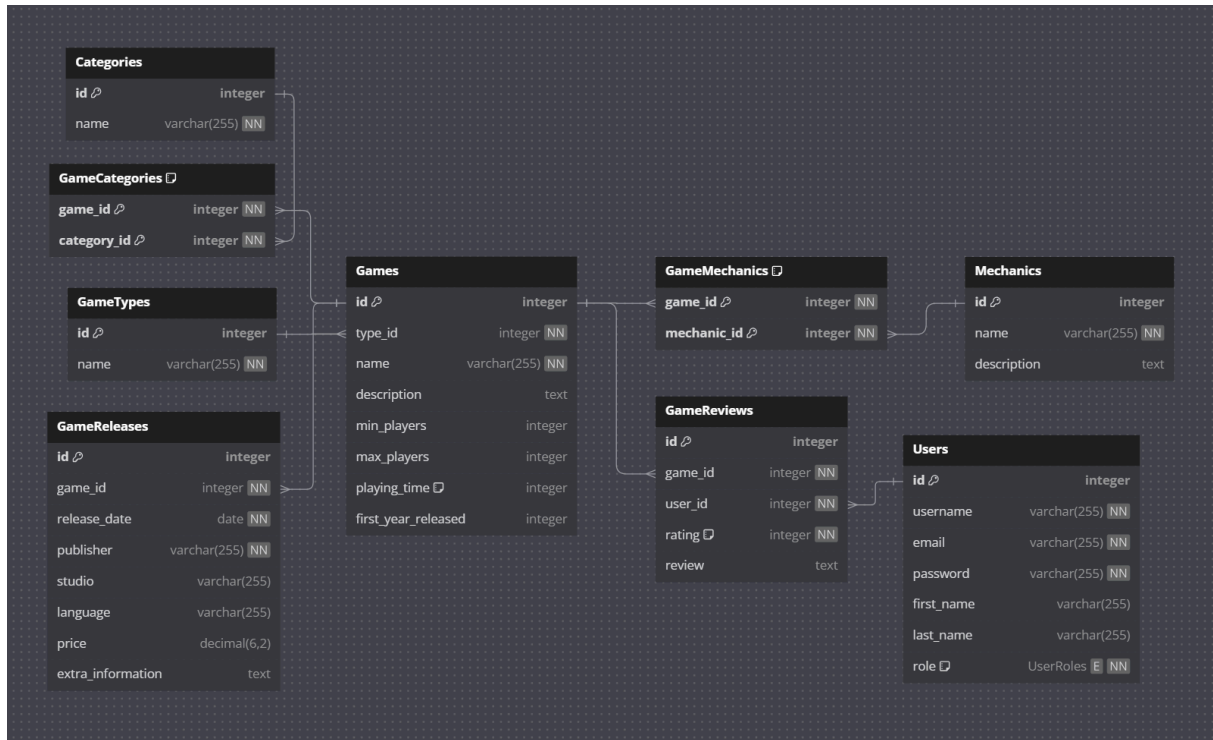
Kontrola wersji projektu zostanie zrealizowana za pomocą narzędzia **Git** i platformy **GitHub**. Git umożliwia zarządzanie kodem źródłowym projektu, a GitHub umożliwia przechowywanie kodu źródłowego w chmurze. Repozytorium projektu będzie dostępne publicznie pod [tym linkiem](#).

SPOTKANIE 3.

Prezentacja diagramu ERD

Diagram ERD przedstawia strukturę bazy danych, która będzie wykorzystywana w tworzonym projekcie. Zawiera on informacje o encjach, atrybutach oraz relacjach między nimi.

Diagram stworzyłem przy pomocy programu online „dbdiagram.io”. Wybór tego narzędzia wynika z mojego wcześniejszego doświadczenia z jego użyciem oraz z użycia języka DBML do tworzenia diagramów. [Link do diagramu na dbdiagram.io](https://dbdiagram.io).



Rysunek 1: Diagram ERD bazy danych projektu.

Opis tabel i ich funkcji

Tabela Games

Tabela Games jest główną tabelą w bazie danych. Zawiera informacje o grach planszowych, takie jak nazwa, opis, minimalna i maksymalna liczba graczy, średni czas gry oraz kategorie. Kluczem głównym tej tabeli jest id, podobnie jak w pozostałych tabelach.

Tabela posiada relacje z tabelami GameTypes, Categories, Mechanics, GameReleases oraz GameReviews:

- relacja z tabelą GameTypes jest typu jeden do wielu, ponieważ jeden typ może być przypisany do wielu gier, ale jedna gra może mieć przypisaną tylko jednego typu. Relacja jest zrealizowana za pomocą klucza obcego type_id w tabeli Games.
- relacja z tabelą GameMechanics jest typu wiele do wielu, ponieważ jedna gra może mieć wiele mechanik, a jedna mechanika może być przypisana do wielu gier. Relacja ta jest zrealizowana za pomocą tabeli pośredniczącej GameMechanics.

- relacja z tabelą `Categories` jest typu wiele do wielu, ponieważ jedna gra może być przypisana do wielu kategorii, a jedna kategoria może być przypisana do wielu gier. Relacja ta jest zrealizowana za pomocą tabeli pośredniczącej `GameCategories`.
- relacja z tabelą `GameReleases` jest typu jeden do wielu, ponieważ jedna gra może mieć wiele wydań, ale jedno wydanie może dotyczyć tylko jednej gry. Relacja jest zrealizowana za pomocą klucza obcego `game_id` w tabeli `GameReleases`.
- relacja z tabelą `GameReviews` jest typu jeden do wielu, ponieważ jedna gra może mieć wiele recenzji, ale jedna recenzja dotyczy tylko jednej gry. Relacja jest zrealizowana za pomocą klucza obcego `game_id` w tabeli `GameReviews`.

Table `GameTypes`

Tabela `GameTypes` przechowuje informacje o typach gier planszowych. Zawiera jedynie pole `name`, które jest unikalne i służy do identyfikacji typu.

Tabela ta posiada relację z tabelą `Games` opisaną wcześniej.

Tabela `Categories`

Tabela `Categories` przechowuje informacje o kategoriach gier planszowych. Zawiera jedynie pole `name`, które jest unikalne i służy do identyfikacji kategorii.

Tabela ta posiada relację z tabelą `Games` opisaną wcześniej.

Tabela `Mechanics`

Tabela `Mechanics` przechowuje informacje o mechanikach gier planszowych. Zawiera pola `name` oraz `description`, z których pierwsze jest unikalne i służy do identyfikacji mechaniki.

Tabela ta posiada relację z tabelą `Games` opisaną wcześniej.

Tabela asocjacyjna `GameMechanics`

Tabela `GameMechanics` jest tabelą pośredniczącą między tabelami `Games` i `Mechanics`. Zawiera pola `game_id` oraz `mechanic_id`, które są kluczami obcymi do odpowiednich tabel. Te pola tworzą jeden klucz główny tej tabeli, który zachowuje unikalność relacji między grami a mechanikami.

Tabela `GameReleases`

Tabela `GameReleases` przechowuje informacje o wydaniach gier planszowych. Zawiera pola `game_id`, `release_date`, `publisher`, `studio`, `language`, `price` oraz `extra_information`.

Tabela ta posiada relację z tabelą `Games` opisaną wcześniej.

Tabela `GameReviews`

Tabela `GameReviews` przechowuje informacje o recenzjach gier planszowych, pisanych przez użytkowników. Zawiera pola `game_id`, `user_id`, `rating` oraz `review`.

Tabela ta posiada relację z tabelą `Games` opisaną wcześniej oraz relację z tabelą `Users`. Relacja z tabelą `Users` jest typu jeden do wielu, ponieważ jedna recenzja może być napisana przez jednego użytkownika, ale jeden użytkownik może napisać wiele recenzji. Relacja ta jest zrealizowana za pomocą klucza obcego `user_id` w tabeli `GameReviews`.

Tabela Users

Tabela Users przechowuje informacje o użytkownikach kolekcji. Zawiera pola username, email, password, first_name, last_name oraz role, która określa rolę użytkownika w systemie.

Rola może być jedną z dwóch wartości: USER lub ADMIN. Ten wybór jest zrealizowany za pomocą typu ENUM.

Tabela ta posiada relację z tabelą GameReviews opisaną wcześniej.

Zapytania SQL

Zapytania tworzące tabele

Poniżej znajdują się zapytania SQL tworzące tabele w bazie danych projektu. Zapytania te zawierają definicje tabel, kluczy głównych, kluczy obcych oraz ograniczeń.

```
1 CREATE TYPE "UserRoles" AS ENUM ( 'ADMIN', 'USER' );
2
3 CREATE TABLE "Users"
4 (
5     "id"          serial PRIMARY KEY,
6     "username"    varchar(255) NOT NULL,
7     "email"       varchar(255) NOT NULL,
8     "password"    varchar(255) NOT NULL,
9     "first_name"  varchar(255),
10    "last_name"   varchar(255),
11    "role"        "UserRoles" NOT NULL
12 );
13
14 CREATE TABLE "GameTypes"
15 (
16     "id"          serial PRIMARY KEY,
17     "name"        varchar(255) UNIQUE NOT NULL
18 );
19
20 CREATE TABLE "Games"
21 (
22     "id"          serial PRIMARY KEY,
23     "type_id"     integer      NOT NULL REFERENCES "GameTypes" ("id"),
24     "name"        varchar(255) NOT NULL,
25     "description" text,
26     "min_players" integer,
27     "max_players" integer,
28     "playing_time" integer,
29     "first_year_released" integer
30 );
31
32 CREATE TABLE "Categories"
33 (
```

```

34     "id"      serial PRIMARY KEY,
35     "name"    varchar(255) UNIQUE NOT NULL
36 );
37
38 CREATE TABLE "GameCategories"
39 (
40     "game_id"      integer NOT NULL REFERENCES "Games" ("id"),
41     "category_id" integer NOT NULL REFERENCES "Categories" ("id"),
42     CONSTRAINT "pk_game_categories" PRIMARY KEY ("game_id", "category_id")
43 );
44
45 COMMENT ON COLUMN "Games"."playing_time" IS 'Approximate playing time in minutes';
46
47 CREATE TABLE "Mechanics"
48 (
49     "id"          serial PRIMARY KEY,
50     "name"        varchar(255) UNIQUE NOT NULL,
51     "description" text
52 );
53
54 CREATE TABLE "GameMechanics"
55 (
56     "game_id"      integer NOT NULL REFERENCES "Games" ("id"),
57     "mechanic_id" integer NOT NULL REFERENCES "Mechanics" ("id"),
58     CONSTRAINT "pk_game_mechanics" PRIMARY KEY ("game_id", "mechanic_id")
59 );
60
61 CREATE TABLE "GameReviews"
62 (
63     "id"          serial PRIMARY KEY,
64     "game_id"     integer NOT NULL REFERENCES "Games" ("id"),
65     "user_id"     integer NOT NULL REFERENCES "Users" ("id"),
66     "rating"      integer NOT NULL,
67     "review"      text
68 );
69
70 COMMENT ON COLUMN "GameReviews"."rating" IS 'Rating from 1 to 10';
71
72 CREATE TABLE "GameReleases"
73 (
74     "id"          serial PRIMARY KEY,
75     "game_id"     integer NOT NULL REFERENCES "Games" ("id"),
76     "release_date" date NOT NULL,
77     "publisher"   varchar(255) NOT NULL,
78     "studio"      varchar(255),
79     "language"    varchar(255),
80     "price"       decimal(6, 2),

```

```
81     "extra_information" text
82 );
```

Zapytania wprowadzające dane

Poniżej znajdują się zapytania SQL wprowadzające przykładowe dane do tabel.

Tabela GameCategories

```
1 INSERT INTO "GameTypes" (name) VALUES
  ('Abstract'), ('Area Control'), ('Cooperative'), ('Deck Building'), ('Economic'),
2  ('Family'), ('Party'), ('Thematic'), ('War Games'), ('Strategy');
```

Tabela Games

```
1 INSERT INTO "Games" (type_id, name, description, min_players, max_players,
  playing_time, first_year_released) VALUES
2  ((SELECT id FROM "GameTypes" WHERE name = 'Abstract'), 'Chess', 'A classic two-
  player strategy game of capturing the opponent's king.', 2, 2, 30, 1475),
  ((SELECT id FROM "GameTypes" WHERE name = 'Abstract'), 'Go', 'An abstract
3  strategy game for two players involving surrounding and capturing your opponent's
  stones.', 2, 2, 60, -2200),
4  ...
```

Tabela Categories i GameCategories

```
1 INSERT INTO "Categories" (name) VALUES
  ('Word Games'), ('Spies/Secret Agents'), ('Humor'), ('Fantasy'), ('Science
2  Fiction'), ('Adventure'), ('Novel-based'), ('Horror'), ('Territory Building'),
  ('Medieval');
3
4 INSERT INTO "GameCategories" (game_id, category_id) VALUES
5  ((SELECT id FROM "Games" WHERE name = 'Codenames'), (SELECT id FROM "Categories"
  WHERE name = 'Word Games')),
6  ((SELECT id FROM "Games" WHERE name = 'Codenames'), (SELECT id FROM "Categories"
  WHERE name = 'Spies/Secret Agents')),
7  ...
```

Tabela Mechanics i GameMechanics

```
1 INSERT INTO "Mechanics" (name) VALUES
  ('Tile Placement'), ('Hand Management'), ('Dice Rolling'), ('Team-Based
2  Game'), ('Trading'), ('Memory'), ('Auction/Bidding'), ('Map Addition'), ('Player
  Elimination'), ('Deck Building'), ('Income'), ('End Game Bonuses');
3
4 INSERT INTO "GameMechanics" (game_id, mechanic_id) VALUES
5  ((SELECT id FROM "Games" WHERE name = 'Carcassonne'), (SELECT id FROM "Mechanics"
  WHERE name = 'Tile Placement')),
6  ((SELECT id FROM "Games" WHERE name = 'Carcassonne'), (SELECT id FROM "Mechanics"
  WHERE name = 'Map Addition')),
7  ...
```

Tabele GameReviews i Users

```
1 INSERT INTO "Users" (username, email, password, role) VALUES
2     ('admin', '174725@stud.prz.edu.pl', crypt('kamil123', gen_salt('bf', 10)),
3     'ADMIN'),
4     ('testuser', 'kpomykała2002@gmail.com', crypt('test123', gen_salt('bf', 10)),
5     'USER');
6
7 INSERT INTO "GameReviews" (game_id, user_id, rating, review) VALUES
8     ((SELECT id FROM "Games" WHERE name = 'Carcassonne'), (SELECT id FROM "Users"
9     WHERE username = 'testuser'), 4, 'Lorem ipsum dolor sit amet, consectetur adipiscing
10    elit. Etiam id consequat lacus. Cras ultricies, nunc molestie placerat tincidunt,
11    enim sapien imperdiet nulla, sit amet tincidunt felis lorem pretium erat.'),
12    ...
```

Funkcje `crypt()` i `gen_salt()` służą do zabezpieczenia haseł użytkowników przed przechowywaniem ich w postaci jawnej w bazie danych. Pochodzą one z rozszerzenia `pgcrypto`, które jest dostępne w PostgreSQL.

Tabela GameReleases

```
1 INSERT INTO "GameReleases" (game_id, release_date, publisher, studio,
2 language, price, extra_information) VALUES
3     ((SELECT id FROM "Games" WHERE name = 'Carcassonne'), '2020-01-01', 'Bard Centrum
4     Gier', 'Bard Centrum Gier', 'Polish', 101.00, null),
5     ((SELECT id FROM "Games" WHERE name = 'Catan'), '2023-06-01', 'NeoTroy Games',
6     null, 'Turkish', 150.00, 'Limited edition'),
7     ...
```

Przykładowe zapytania selekcyjne

Zapytanie o gry z kategorii „Fantasy”

```
1 SELECT "Games"."name" FROM "Games"
2 INNER JOIN "GameCategories" ON "Games"."id" = "GameCategories"."game_id"
3 INNER JOIN "Categories" ON "GameCategories"."category_id" = "Categories"."id"
4 WHERE "Categories"."name" = 'Fantasy';
```

Zapytanie o gry z mechaniką „Hand Management”

```
1 SELECT "Games"."name" FROM "Games"
2 INNER JOIN "GameMechanics" ON "Games"."id" = "GameMechanics"."game_id"
3 INNER JOIN "Mechanics" ON "GameMechanics"."mechanic_id" = "Mechanics"."id"
4 WHERE "Mechanics"."name" = 'Hand Management';
```

Zapytanie o gry, które posiadają średnią recenzji powyżej 4

```
1 SELECT "Games"."name", AVG("GameReviews"."rating") AS "average_rating",  
   COUNT("GameReviews"."rating") AS "number_of_ratings" FROM "Games"  
2 RIGHT JOIN "GameReviews" ON "Games"."id" = "GameReviews"."game_id"  
3 GROUP BY "Games"."id"  
4 HAVING AVG("GameReviews"."rating") ≥ 4;
```

Zapytanie o kategorie, które posiadają więcej niż 10, ale mniej niż 20 gier

```
1 SELECT "Categories"."name", COUNT("Games"."id") AS "game_count" FROM  
   "Categories"  
2 INNER JOIN "GameCategories" ON "Categories"."id" = "GameCategories"."category_id"  
3 INNER JOIN "Games" ON "GameCategories"."game_id" = "Games"."id"  
4 GROUP BY "Categories"."id"  
5 HAVING COUNT("Games"."id") > 10 AND COUNT("Games"."id") < 20;
```

Zapytanie o użytkowników z rolą „USER” i ilością napisanych recenzji

```
1 SELECT "Users"."username", COUNT("GameReviews"."id") AS "review_count" FROM  
   "Users"  
2 LEFT JOIN "GameReviews" ON "Users"."id" = "GameReviews"."user_id"  
3 WHERE "Users"."role" = 'USER'  
4 GROUP BY "Users"."id";
```

Zapytanie o gry wydane po 2010 roku

```
1 SELECT "Games"."name", "Games"."first_year_released" FROM "Games"  
2 WHERE "Games"."first_year_released" > 2010;
```

SPOTKANIE 4.

Zaawansowane zapytania SQL

Dodawanie danych

Zaawansowane dodawanie danych potraktowałem jako dodanie większej ilości danych do tabel w bazie danych. W tym celu przygotowałem kolejny skrypt SQL, który dodaje przykładowe dane. Tym razem postarałem się o zwiększenie ilości danych, aby pokazać jak system zachowuje się przy większej ilości rekordów.

Wcześniejsza iteracja pliku SQL zawierała prawie 100 linijek kodu. Nowy plik zawiera prawie 700 linijek. Widać, że ilość danych znacznie wzrosła.

Po wykonaniu nowego skryptu, w bazie danych znajduję się 130 gier, 35 kategorii gier, 10 typów gier, 33 mechaniki, 54 użytkowników, 53 recenzje oraz 56 wydań gier.

Aktualizacja danych

Tutaj potraktowałem frazę „zaawansowany” poważniej i przewidziałem parę różnych przypadków aktualizacji danych, które mogą wystąpić podczas pracy na bazie danych.

Powiększenie cen wydań gier w języku niemieckim o 25%

```
1 UPDATE "GameReleases"
2 SET "price" = "price" * 1.25
3 WHERE "language" = 'German';
```



Zmniejszenie cen wydań gier z mechaniką „Hand Management” o 10%

```
1 UPDATE "GameReleases"
2 SET "price" = "price" * 0.9
3 FROM "Games"
4     INNER JOIN "GameMechanics" ON "Games"."id" = "GameMechanics"."game_id"
5     INNER JOIN "Mechanics" ON "GameMechanics"."mechanic_id" = "Mechanics"."id"
6 WHERE "Games"."id" = "GameReleases"."game_id"
7 AND "Mechanics"."name" = 'Hand Management';
```



Zwiększenie cen wydań gier z recenzją o ocenie 4 lub więcej o 15%

```
1 UPDATE "GameReleases"
2 SET "price" = "price" * 1.15
3 WHERE "game_id" IN (
4     SELECT "game_id"
5     FROM "GameReviews"
6     WHERE "GameReviews"."rating" ≥ 4
7 );
```



Zmiana roli użytkowników na podstawie ilości recenzji gier.

Jeśli użytkownik napisał więcej niż 5 recenzji, to zmień jego rolę na „ADMIN”, w przeciwnym wypadku na „USER”.

```
1 UPDATE "Users"
2 SET "role" = CASE
3     WHEN (SELECT COUNT(*) FROM "GameReviews" WHERE "Users"."id" =
4           "GameReviews"."user_id") > 5 THEN 'ADMIN'
5     ELSE 'USER'
6 END
7 FROM "GameReviews"
8 WHERE "Users"."id" = "GameReviews"."user_id";
```

Aktualizacja opisu mechanik.

Jeśli mechanika nie jest używana w żadnej grze, to ustaw opis na „*This mechanic is not used in any game*”. W przeciwnym wypadku ustaw opis na „*This mechanic is used in X games and Y game types*”. Gdzie X to liczba gier, a Y to liczba typów gier, w których jest używana mechanika.

```
1 UPDATE "Mechanics"
2 SET "description" =
3     CASE
4         WHEN (SELECT COUNT(*) FROM "GameMechanics" WHERE "Mechanics"."id" =
5               "GameMechanics"."mechanic_id") = 0 THEN 'This mechanic is not used in any game'
6         ELSE
7             'This mechanic is used in '
8             || (SELECT COUNT(DISTINCT "game_id") FROM "GameMechanics" WHERE
9                  "Mechanics"."id" = "GameMechanics"."mechanic_id")
10             || ' games and '
11             || (SELECT COUNT(DISTINCT "type_id")
12                  FROM "Games"
13                  WHERE "Games"."id" IN (SELECT "game_id"
14                                           FROM "GameMechanics"
15                                           WHERE "Mechanics"."id" = "GameMechanics"."mechanic_id"))
16             || ' game types'
17     END
18 WHERE "Mechanics"."description" IS NULL;
```

Selekcja danych

W przypadku selekcji danych również postanowiłem pokazać kilka bardziej zaawansowanych zapytań, które mogą być przydatne podczas pracy z bazą danych.

Zapytanie o gry, których suma cen wydań wynosi ponad 200 zł

```
1 SELECT  "Games"."name",    SUM("GameReleases"."price") AS  "price_sum",
2 COUNT("GameReleases"."price") AS "number_of_releases"
3 FROM  "Games"
4 INNER JOIN "GameReleases" ON "Games"."id" = "GameReleases"."game_id"
5 GROUP BY "Games"."id"
6 HAVING SUM("GameReleases"."price") ≥ 200;
```

Zapytanie o ranking użytkowników na podstawie ilości napisanych recenzji

Z pominięciem użytkowników, którzy nie napisali żadnej recenzji.

```
1 SELECT
2     "Users"."username",
3     ROUND(AVG("GameReviews"."rating"), 3) AS "average_rating",
4     COUNT("GameReviews"."rating") AS "number_of_ratings",
5     (SELECT "GameReviews"."review"
6 FROM "GameReviews"
7 WHERE "GameReviews"."user_id" = "Users"."id"
8 ORDER BY "GameReviews"."id" DESC
9 LIMIT 1) AS "latest_review"
10 FROM "Users"
11 LEFT JOIN "GameReviews" ON "Users"."id" = "GameReviews"."user_id"
12 GROUP BY "Users"."id"
13 HAVING COUNT("GameReviews"."rating") > 0
14 ORDER BY COUNT("GameReviews"."rating") DESC;
```


Zapytanie o gry i ich pierwsze, najnowsze wydanie oraz oryginalny rok wydania

```
1  SELECT
2      "Games"."name",
3      (SELECT "GameReleases"."release_date"
4      FROM "GameReleases"
5      WHERE "GameReleases"."game_id" = "Games"."id"
6      ORDER BY "GameReleases"."release_date" DESC
7      LIMIT 1) AS "latest_release",
8      (SELECT "GameReleases"."release_date"
9      FROM "GameReleases"
10     WHERE "GameReleases"."game_id" = "Games"."id"
11     ORDER BY "GameReleases"."release_date" ASC
12     LIMIT 1) AS "first_release",
13     "Games"."first_year_released"
14 FROM "Games"
15 RIGHT JOIN "GameReleases" ON "Games"."id" = "GameReleases"."game_id"
16 GROUP BY "Games"."id";
```

Zapytanie o typy gier oraz ich średnią ilość minimalną i maksymalną graczy

```
1  SELECT
2      "GameTypes"."name",
3      ROUND(AVG("Games"."min_players"), 2) AS "average_min_players",
4      ROUND(AVG("Games"."max_players"), 2) AS "average_max_players"
5 FROM "GameTypes"
6 INNER JOIN "Games" ON "GameTypes"."id" = "Games"."type_id"
7 GROUP BY "GameTypes"."id";
```

Zapytanie o kategorie gier, ilość gier, lista gier oraz średnią ocenę gier w danej kategorii

Lista gier to nazwy gier w danej kategorii, oddzielone przecinkami.

```
1  SELECT
2      "Categories"."name",
3      COUNT("Games"."id") AS "game_count",
4      STRING_AGG("Games"."name", ', ') AS "game_list",
5      (SELECT ROUND(AVG("GameReviews"."rating"), 2)
6      FROM "GameReviews"
7      INNER JOIN "Games" ON "GameReviews"."game_id" = "Games"."id"
8      INNER JOIN "GameCategories" ON "Games"."id" = "GameCategories"."game_id"
9      WHERE "GameCategories"."category_id" = "Categories"."id") AS "average_rating"
10 FROM "Categories"
11 INNER JOIN "GameCategories" ON "Categories"."id" = "GameCategories"."category_id"
12 INNER JOIN "Games" ON "GameCategories"."game_id" = "Games"."id"
13 GROUP BY "Categories"."id";
```

Role i uprawnienia

PostgreSQL bazuje na modelu ról i uprawnień, które pozwalają na kontrolę dostępu do danych w bazie. We wcześniejszych wersjach PostgreSQL, był podział na użytkowników i grupy, teraz role zastępują oba te pojęcia i pozwalają na bardziej elastyczne zarządzanie dostępem do danych.

W projekcie stworzyłem dwie role bez uprawnień do logowania (wcześniej takie role nazywały się grupami): user i admin. Rola user ma ograniczony dostęp do danych, może przeglądać tabele, dodawać i aktualizować w tabeli GameReviews. Rola admin ma pełny dostęp do danych, może przeglądać, dodawać, aktualizować i usuwać dane z wszystkich tabel.

```
1 CREATE ROLE admins WITH NOLOGIN;
2 CREATE ROLE users WITH NOLOGIN;
3
4 SET ROLE NONE; -- Reset role to NONE
5 GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO admins;
6 GRANT CREATE ON SCHEMA public TO admins;
7 GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO admins;
8
9 SET ROLE NONE;
10 GRANT SELECT ON ALL TABLES IN SCHEMA public TO users;
11 GRANT INSERT, UPDATE ON "GameReviews" TO users;
12 GRANT USAGE, SELECT ON "GameReviews_id_seq" TO users;
```

Musiałem dodać uprawnienia do sekwencji, ponieważ kolumny serial w PostgreSQL są zaimplementowane za pomocą sekwencji, które są automatycznie tworzone przez bazę danych. Bez tych uprawnień, role nie mogą korzystać z kolumn serial.

W tym samym pliku SQL, stworzyłem również role z możliwością logowania (użytkowników), które odpowiadają rekordom z tabeli Users.

```
1 -- Create administrators
2 SET ROLE NONE;
3 CREATE ROLE admin WITH LOGIN PASSWORD 'kamil123' IN ROLE admins; -- Passwords are
  always encrypted using sha256
4 CREATE ROLE brownleopard167 WITH LOGIN PASSWORD 'adriana' IN ROLE admins;
5 CREATE ROLE brownelephant395 WITH LOGIN PASSWORD 'marianne' IN ROLE admins;
6 ...
7
8 -- Create users
9 SET ROLE NONE;
10 CREATE ROLE testuser WITH LOGIN PASSWORD 'test123' IN ROLE users;
11 CREATE ROLE bigdog363 WITH LOGIN PASSWORD 'lespaul' IN ROLE users;
12 ...
```

Hasła można zapisać tutaj bez funkcji crypt(), ponieważ PostgreSQL automatycznie je zaszyfruje.

Przykładowe zapytania z wykorzystaniem ról

Na samym końcu pliku SQL, dodałem kilka przykładowych zapytań, które potwierdzają poprawne działanie ról i uprawnień.

```
1  -- Example of using the admins group
2  SET ROLE admin; -- Switch to the admin role
3
4  SELECT * FROM "Games";
5
6  INSERT INTO "Games" (type_id, name, description, min_players, max_players,
7  playing_time, first_year_released)
8  VALUES (1, 'Test Game', 'This is a test game', 2, 4, 60, 2019);
9
10 UPDATE "Games" SET description = 'This is a test game that has been updated' WHERE
11 name = 'Test Game';
12
13 DELETE FROM "Games" WHERE name = 'Test Game';
```

Tutaj wszystkie zapytania powinny zostać wykonane bez problemów, ponieważ rola admin ma pełne uprawnienia do wszystkich tabel.

```
1  -- Example of using the users group
2  SET ROLE testuser; -- Switch to the testuser role
3
4  SELECT * FROM "Games";
5
6  SELECT * FROM "GameReviews";
7
8  INSERT INTO "GameReviews" (game_id, user_id, rating, review)
9  VALUES ((SELECT id FROM "Games" WHERE name = 'Chess'), (SELECT id FROM "Users"
10 WHERE username = 'testuser'), 5, 'This is a great game!');
11
12 UPDATE "GameReviews" SET review = 'This is a great game! I love it!' WHERE game_id
13 = (SELECT id FROM "Games" WHERE name = 'Chess') AND user_id = (SELECT id FROM
14 "Users" WHERE username = 'testuser');
15
16 -- This will fail because the user group does not have delete permissions on the
17 GameReviews table
18 DELETE FROM "GameReviews" WHERE game_id = (SELECT id FROM "Games" WHERE name =
19 'Chess') AND user_id = (SELECT id FROM "Users" WHERE username = 'testuser');
20
21 -- This will fail because the user group does not have insert permissions on the
22 Games table
23 INSERT INTO "Games" (type_id, name, description, min_players, max_players,
24 playing_time, first_year_released)
25 VALUES (1, 'Test Game', 'This is a test game', 2, 4, 60, 2019);
26
```

```

20 -- This will fail because the user group does not have update permissions on the
    Games table
21 UPDATE "Games" SET description = 'This is a test game that has been updated' WHERE
    name = 'Test Game';

```

Natomiast tutaj niektóre zapytania powinny zakończyć się błędem, ponieważ rola user ma ograniczone uprawnienia. Komentarze w kodzie SQL wyjaśniają, które i dlaczego zapytania nie powiodą się.

Funkcje i procedury

W PostgreSQL można tworzyć funkcje i procedury, które pomagają w automatyzacji zadań i wykonywaniu bardziej skomplikowanych operacji na bazie danych. Funkcje mogą przyjmować argumenty i zwracać wartości, podczas gdy procedury są bardziej podobne do skryptów, które wykonują operacje na bazie danych.

W projekcie stworzyłem plik SQL zawierający kilka przykładowych funkcji i procedur, które mogą być przydatne w pracy z moją bazą danych.

Funkcja get_game_details

W tej funkcji, zwracam zestaw informacji o grze na podstawie jej nazwy. Funkcja przyjmuje nazwę gry jako argument i zwraca ID gry, nazwę, typ, kategorie, mechaniki oraz wydania w formacie JSON.

```

1  CREATE OR REPLACE FUNCTION get_game_details(game_name_param VARCHAR(255)) SQL
2      RETURNS TABLE
3      (
4          game_id    INTEGER,
5          game_name  VARCHAR(255),
6          game_type  VARCHAR(255),
7          categories VARCHAR[],
8          mechanics  VARCHAR[],
9          releases   JSON
10     )
11 AS
12 $$
13 BEGIN
14     RETURN QUERY
15     SELECT g.id          AS ID,
16            g.name        AS Name,
17            gt.name       AS Type,
18            ARRAY_AGG(DISTINCT c.name) AS Categories,
19            ARRAY_AGG(DISTINCT m.name) AS Mechanics,
20            JSON_AGG(
21                JSON_BUILD_OBJECT(
22                    'id', gr.id,
23                    'release_date', gr.release_date,
24                    'publisher', gr.publisher,
25                    'studio', gr.studio,
26                    'language', gr.language,

```

```

27         'price', gr.price,
28         'extra_information', gr.extra_information
29     )
30 ) AS Releases
31 FROM "Games" g
32     JOIN "GameTypes" gt ON g.type_id = gt.id
33     LEFT JOIN "GameCategories" gc ON g.id = gc.game_id
34     LEFT JOIN "Categories" c ON gc.category_id = c.id
35     LEFT JOIN "GameMechanics" gm ON g.id = gm.game_id
36     LEFT JOIN "Mechanics" m ON gm.mechanic_id = m.id
37     LEFT JOIN "GameReleases" gr ON g.id = gr.game_id
38 WHERE g.name = game_name_param
39 GROUP BY g.id, gt.name;
40 END
41 $$ LANGUAGE plpgsql;
42
43 -- Example of calling the function
44 SELECT * FROM get_game_details('Champions of Midgard');
```

get_game_details()

Output Example of calling the function

	game_id	game_name	game_type	categories	mechanics	releases
1	124	Champions of Midgard	Strategy	{null}	{Dice Rolling, Set Co...	[{"id": 118, "release_date": "2015-01-01", "publ

Rysunek 2: Wynik przykładowego zapytania z wykorzystaniem funkcji `get_game_details`

Procedura `add_game_review`

Ta procedura za zadanie dodanie recenzji do gry na podstawie nazwy gry, ID użytkownika, oceny i recenzji. Procedura przyjmuje te parametry i dodaje nowy rekord do tabeli `GameReviews`.

```

1 CREATE OR REPLACE PROCEDURE add_game_review(
2     game_name_param VARCHAR(255),
3     user_id_param INTEGER,
4     rating_param INTEGER,
5     review_param TEXT
6 )
7 AS
8 $$
9 BEGIN
10     INSERT INTO "GameReviews" (game_id, user_id, rating, review)
11     VALUES ((SELECT id FROM "Games" WHERE name = game_name_param), user_id_param,
12     rating_param, review_param);
13 END;
14 $$ LANGUAGE plpgsql;
```

```

14
15 -- Example of calling the procedure
16 CALL add_game_review('Champions of Midgard', 1, 5, 'Great game, highly
    recommended!');

```

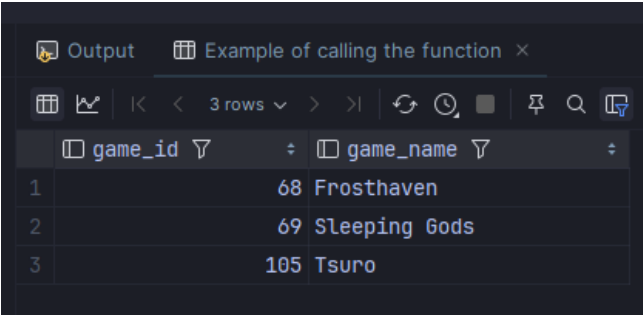
Funkcja get_games_by_category_and_mechanic

Kolejna funkcja zwraca listę gier, które należą do danej kategorii i posiadają daną mechanikę. Funkcja przyjmuje nazwę kategorii i mechaniki jako argumenty i zwraca ID gry i nazwę gry.

```

1  CREATE OR REPLACE FUNCTION get_games_by_category_and_mechanic(
2      category_name_param VARCHAR(255),
3      mechanic_name_param VARCHAR(255)
4  )
5      RETURNS TABLE
6      (
7          game_id  INTEGER,
8          game_name VARCHAR(255)
9      )
10 AS
11 $$
12 BEGIN
13     RETURN QUERY
14         SELECT g.id AS game_id,
15                g.name AS game_name
16         FROM "Games" g
17         JOIN "GameCategories" gc ON g.id = gc.game_id
18         JOIN "Categories" c ON gc.category_id = c.id
19         JOIN "GameMechanics" gm ON g.id = gm.game_id
20         JOIN "Mechanics" m ON gm.mechanic_id = m.id
21         WHERE c.name = category_name_param
22                AND m.name = mechanic_name_param;
23 END;
24 $$ LANGUAGE plpgsql;
25
26 -- Example of calling the function
27 SELECT * FROM get_games_by_category_and_mechanic('Fantasy', 'Hand Management');

```



	game_id	game_name
1	68	Frosthaven
2	69	Sleeping Gods
3	105	Tsuro

Rysunek 3: Wynik przykładowego zapytania z wykorzystaniem funkcji `get_games_by_category_and_mechanic`

Procedury remove_mechanic_from_game

Procedura remove_mechanic_from_game usuwa mechanikę z gry na podstawie nazwy gry i nazwy mechaniki. Procedura znajduje ID gry i ID mechaniki na podstawie nazw i usuwa rekord z tabeli GameMechanics.

```
1 CREATE OR REPLACE PROCEDURE remove_mechanic_from_game(  
2     game_name_param VARCHAR(255),  
3     mechanic_name_param VARCHAR(255)  
4 )  
5 AS  
6 $$  
7 BEGIN  
8     DELETE  
9     FROM "GameMechanics"  
10    WHERE game_id = (SELECT id FROM "Games" WHERE name = game_name_param)  
11    AND mechanic_id = (SELECT id FROM "Mechanics" WHERE name = mechanic_name_param);  
12 END;  
13 $$ LANGUAGE plpgsql;  
14  
15 -- Example of calling the procedure  
16 CALL remove_mechanic_from_game('Champions of Midgard', 'Worker Placement');
```

Procedura add_mechanic_to_game

Procedura add_mechanic_to_game dodaje mechanikę do gry na podstawie nazwy gry i nazwy mechaniki. Procedura znajduje ID gry i ID mechaniki na podstawie nazw i dodaje nowy rekord do tabeli GameMechanics.

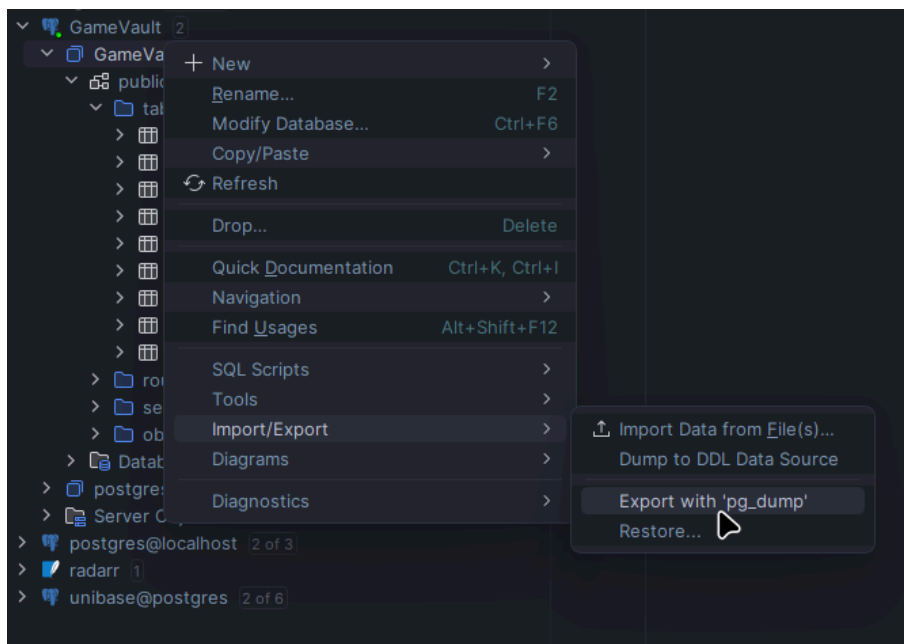
```
1 CREATE OR REPLACE PROCEDURE add_mechanic_to_game(  
2     game_name_param VARCHAR(255),  
3     mechanic_name_param VARCHAR(255)  
4 )  
5 AS  
6 $$  
7 BEGIN  
8     INSERT INTO "GameMechanics" (game_id, mechanic_id)  
9     VALUES ((SELECT id FROM "Games" WHERE name = game_name_param), (SELECT id FROM  
10    "Mechanics" WHERE name = mechanic_name_param));  
11 END;  
12 $$ LANGUAGE plpgsql;  
13  
14 -- Example of calling the procedure  
15 CALL add_mechanic_to_game('Champions of Midgard', 'Worker Placement');
```

Backup i restore

W tym punkcie posłużyłem się głównie narzędziami dostępnymi w aplikacji do zarządzania bazą danych DataGrip. DataGrip automatycznie wykonuje komendy `pg_dump` i `psql`, które są odpowiedzialne za tworzenie kopii zapasowych i przywracanie danych z kopii zapasowej.

Backup

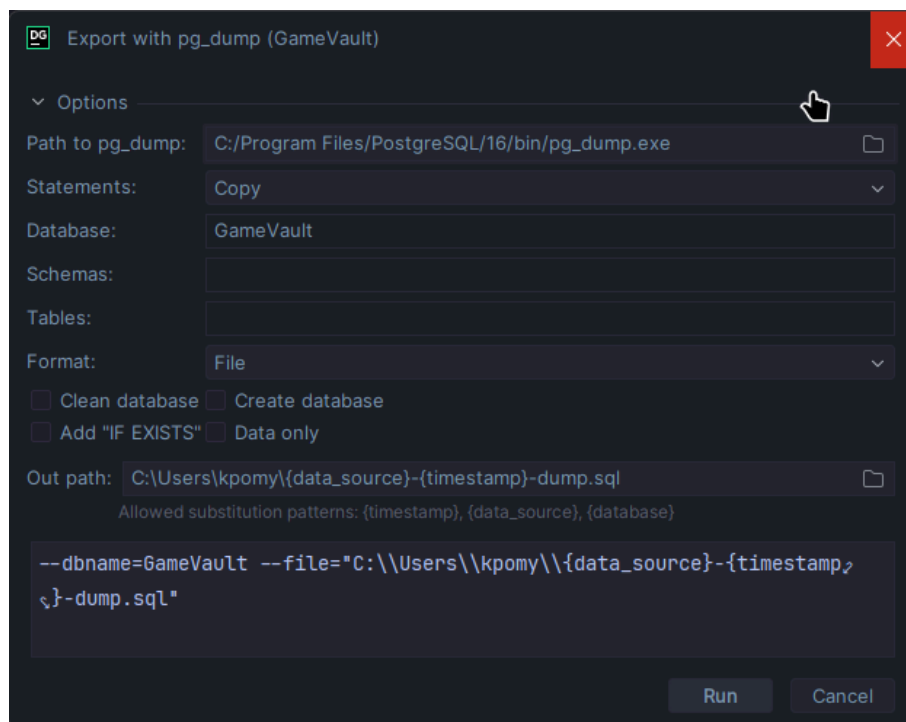
Aby wykonać kopię zapasową bazy danych, wystarczy wybrać odpowiednią opcję w menu kontekstowym bazy danych w DataGrip.



Rysunek 4: Wybór opcji backup z menu kontekstowego bazy danych w DataGrip

Po wybraniu opcji pojawia się okno dialogowe. W przypadku systemu Windows, DataGrip pytał mnie o lokalizację skryptu `pg_dump`, który jest częścią instalacji PostgreSQL (możliwe, że brakowało mi skryptu w zmiennych środowiskowych). Natomiast w przypadku systemu Linux, DataGrip automatycznie wykrył lokalizację skryptu.

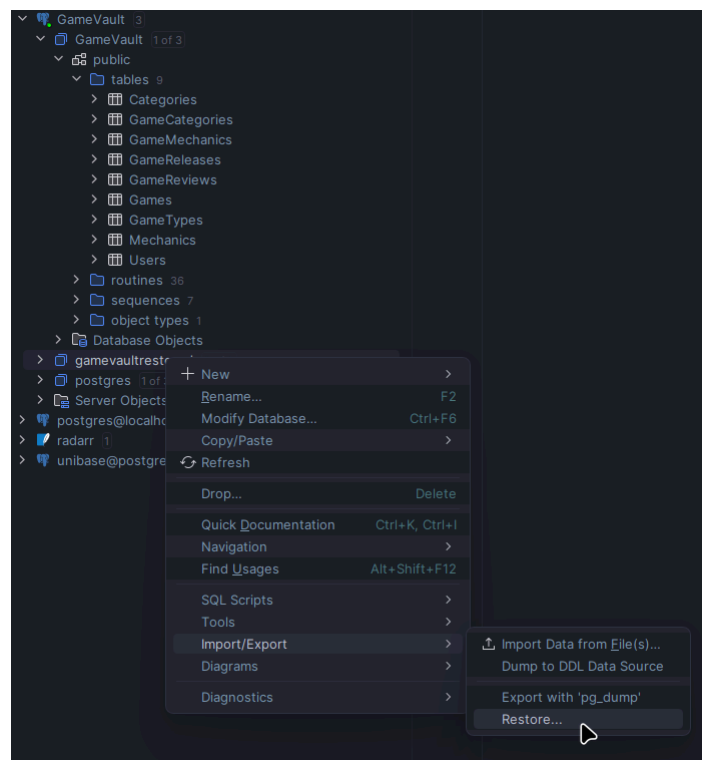
W oknie dialogowym można wybrać lokalizację pliku z kopią zapasową, format pliku, opcje tworzenia kopii zapasowej oraz dodatkowe opcje.



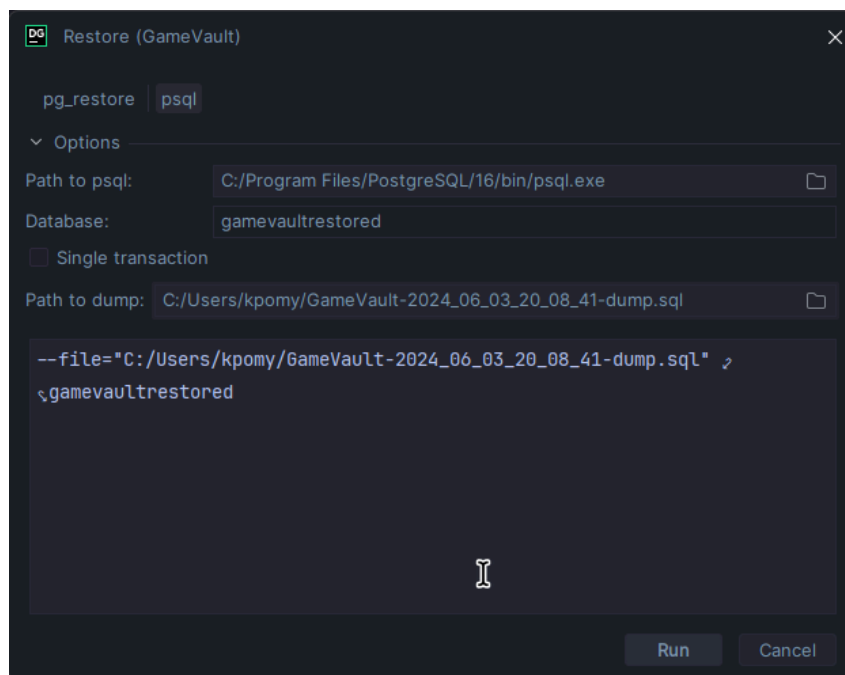
Rysunek 5: Okno dialogowe backupu bazy danych w DataGrip

Restore

Komenda Restore działa na podobnej zasadzie. W zależności od formatu kopii zapasowej, trzeba przygotować odpowiednio bazę danych. W przypadku plików .sql, trzeba utworzyć pustą bazę danych. Następnie wybieramy opcję „Restore...” z menu kontekstowego bazy danych i wybieramy plik z kopią zapasową.

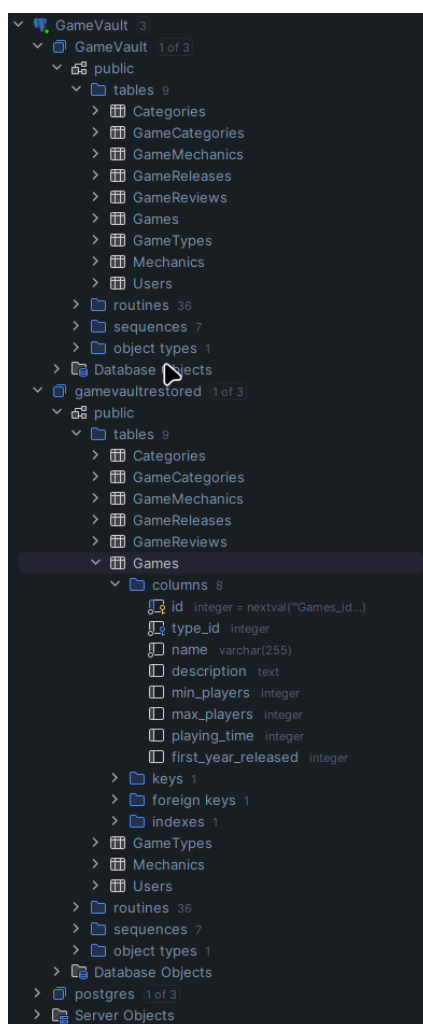


Rysunek 6: Wybór opcji restore z menu kontekstowego bazy danych w DataGrip



Rysunek 7: Okno dialogowe restore bazy danych w DataGrip

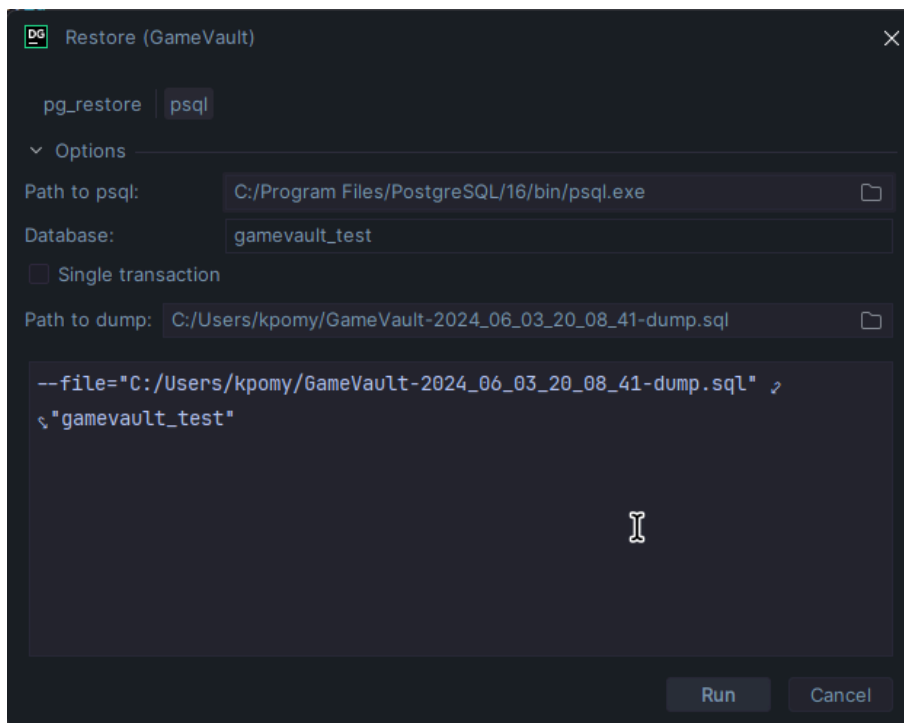
Po wybraniu pliku, DataGrip automatycznie przywraca bazę danych z kopii zapasowej.



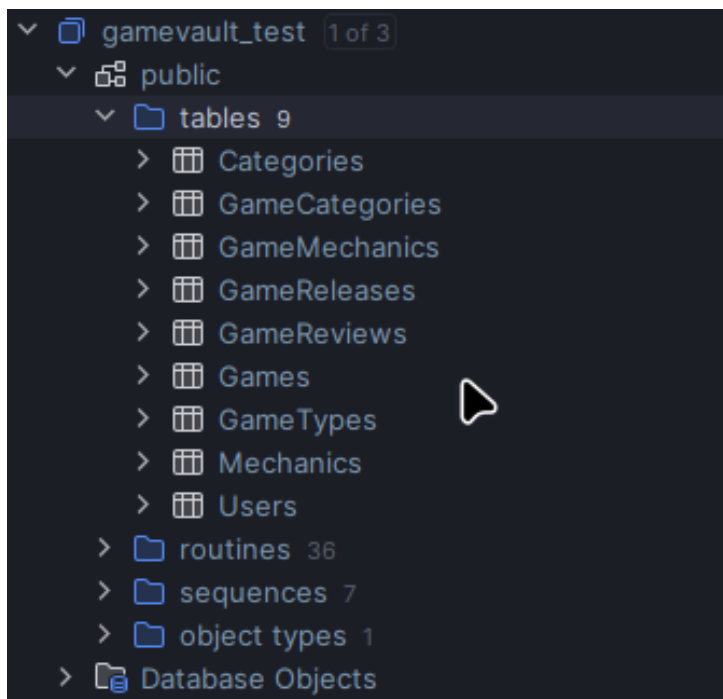
Rysunek 8: Dwie bazy danych obok siebie - oryginalna i przywrócona z kopii zapasowej

Wersja testowa bazy danych

Testową bazę danych stworzyłem na serwerze lokalnym, korzystając z opcji backupu i restore opisanych powyżej.



Rysunek 9: Okno dialogowe restore bazy testowej w DataGrip



Rysunek 10: Nowo utworzona i przywrócona testowa baza danych

Komunikacja z językiem programowania

Komunikacja z bazą danych PostgreSQL jest możliwa z poziomu wielu języków programowania. W projekcie użyłem języka Rust i biblioteki `sqlx` do komunikacji z bazą danych.

Program składa się z jednego pliku `main.rs`. W pliku tym, łączę się z bazą danych, wykonuję kilka zapytań i wyświetlam wyniki.

Połączenie z bazą danych

Połączenie z bazą danych z użyciem `sqlx` jest bardzo proste. Wystarczy stworzyć strukturę `Pool` i przekazać adres URL bazy danych. Adres ten jest brany z pliku `.env` dla bezpieczeństwa.

```
1 async fn get_connection(database_url: &str) → Pool<Postgres> {  
2     PgPoolOptions::new()  
3         .max_connections(5)  
4         .connect(database_url)  
5         .await  
6         .expect("Failed to connect to Postgres.")  
7 }
```

Struktura `Pool` przechowuje połączenia do bazy danych i zarządza nimi automatycznie. Za jej pomocą można wykonywać zapytania do bazy danych.

Przykładowe zapytania

W pliku `main.rs` znajdują się 3 przykładowe zapytania.

Zapytanie o 10 najnowszych gier

Funkcja `print_ten_latest_games` wykonuje zapytanie do bazy danych o 10 najnowszych gier i wyświetla ich nazwy i opisy za pomocą makra `cprintln!`, które dodaje kolor do konsoli.

```
1 async fn print_ten_latest_games(pool: &Pool<Postgres>) {  
2     let result = sqlx::query!("SELECT name, description FROM \"Games\" ORDER BY  
3         id DESC LIMIT 10")  
4         .fetch_all(pool)  
5         .await  
6         .expect("Failed to fetch games.");  
7     cprintln!("\n<strong>Latest 10 games:</strong>\n");  
8     for record in result {  
9         let description = record.description.unwrap_or_else(|| String::from("No  
10             description available."));  
11         cprintln!("<strong><b>{}</b> - <i>{}</i>", record.name, description);  
12     }  
13  
14     println!("\n");  
15 }
```

```
Latest 10 games:
Jenga - Carefully remove blocks from the tower and stack them on top to build the tallest structure without causing it to collapse in this classic dexterity game.
Tichu - Form sets, play cards, and call Tichu to score points and win tricks in this classic partnership climbing card game.
The Lord of the Rings - Embark on a perilous journey to Mount Doom to destroy the One Ring and save Middle-earth in this cooperative adventure game based on J.R.R. Tolkien's epic fantasy.
Above and Below - Build a village above ground and explore the caverns below in this storytelling game that combines worker placement and narrative adventure.
Stratego - Deploy your army, outmaneuver your opponent, and capture the flag in this classic strategy game of hidden movement and deduction.
Castle Panic - Defend your castle from hordes of monsters by coordinating with your fellow players to protect the castle walls in this cooperative tower defense game.
Samurai - Compete to control feudal Japan by placing tiles, collecting resources, and influencing regions in this classic area control game.
Champions of Midgard - Lead a Viking clan to glory by recruiting warriors, battling monsters, and completing quests in this strategic game set in the world of Norse mythology.
```

Rysunek 11: Wynik zapytania o 10 najnowszych gier w języku Rust

Zapytanie o kategorie z 5-15 grami

Tym razem funkcja `print_categories_with_5_15_games` wykonuje zapytanie o kategorie, które posiadają od 5 do 15 gier. Wyniki są wyświetlane w konsoli.

```
1  async fn print_categories_with_5_15_games(pool: &Pool<Postgres>) { Rust
2      let result = sqlx::query!("SELECT c.name, COUNT(g.id) AS game_count FROM
3          \"Categories\" c
4          INNER JOIN \"GameCategories\" gc ON c.id = gc.category_id
5          INNER JOIN \"Games\" g ON gc.game_id = g.id
6          GROUP BY c.id
7          HAVING COUNT(g.id) > 5 AND COUNT(g.id) < 15")
8          .fetch_all(pool)
9          .await
10         .expect("Failed to fetch categories.");
11
12     cprintln!("\n<strong>Categories with more than 5 and less than 15 games:</strong>\n");
13     for record in result {
14         cprintln!("\n<strong><b>{}</b> - {}</strong>", record.name,
15             record.game_count.unwrap_or(0));
16     }
17     println!("\n");
18 }
```

```
Categories with more than 5 and less than 15 games:
Fantasy - 11
Adventure - 10
Card Game - 6
Fighting - 6
```

Rysunek 12: Wynik zapytania o kategorie z 5-15 grami w języku Rust

Zapytanie o użytkowników i ilość recenzji

Ostatnie zapytanie w funkcji `print_users_and_number_of_reviews` zwraca użytkowników z rolą „USER” i ilością napisanych recenzji. Ominięci są użytkownicy, którzy nie napisali żadnej recenzji. Wyniki zapytania są wyświetlane w konsoli.

```
1  async fn print_users_and_number_of_reviews(pool: &Pool<Postgres>) { Rust
2      let result = sqlx::query!("SELECT u.username, COUNT(gr.id) AS review_count
3      FROM \"Users\" u
4      LEFT JOIN \"GameReviews\" gr ON u.id = gr.user_id
5      WHERE u.role = 'USER'
6      AND gr.id IS NOT NULL
7      GROUP BY u.id")
8      .fetch_all(pool)
9      .await
10     .expect("Failed to fetch users.");
11
12     cprintln!("\n<strong>Users and the number of reviews they have made (exclude
13     users without reviews):</strong>");
14     for record in result {
15         cprintln!("<strong><b>{}</b> - <i>{:?}</i>", record.username,
16         record.review_count.unwrap_or(0));
17     }
18     println!("\n");
19 }
```

```
Users and the number of reviews they have made (exclude users without reviews):
organicdog182 - 2
crazycat330 - 1
smallladybug987 - 3
bluelion607 - 1
ticklishgoose127 - 2
silverwolf552 - 1
beautiful ladybug647 - 1
whitefrog396 - 1
brownleopard490 - 1
beautifulpeacock570 - 3
purplekoala865 - 1
blackbutterfly238 - 1
testuser - 4
bigdog363 - 1
organicmouse894 - 3
redelephant587 - 3
```

Rysunek 13: Wynik zapytania o użytkowników i ilość recenzji w języku Rust

Rust ORM

Oprócz krótkiego programu w Rust, który korzysta z biblioteki `sqlx`, można również korzystać z ORM (Object-Relational Mapping) w Rust. Jedną z popularniejszych bibliotek ORM w Rust jest `diesel` i z niej skorzystałem w projekcie. Za jej pomocą stworzyłem prostą aplikację konsolową do zarządzania bazą danych.

Oprócz tego skorzystałem z następujących bibliotek:

- `dotenvy` - do ładowania zmiennych środowiskowych z pliku `.env`
- `terminal-menu` - do stworzenia prostego menu w konsoli
- `inquire` - do interakcji z użytkownikiem

Aplikacja składa się z ponad 1000 linii kodu.

Połączenie z bazą danych

Podobnie jak w przypadku `sqlx`, połączenie z bazą danych w `diesel` jest proste. Tym razem projekt podzieliłem na wiele sekcji i plików, aby zachować porządek. W pliku `db.rs` znajduje się funkcja `establish_admin_connection`, która łączy się z bazą danych jako administrator i zwraca połączenie.

Połączenie administratora jest potrzebne do pierwszej weryfikacji logującego się użytkownika.

```
1 pub fn establish_admin_connection() → PgConnection {  
2     let database_url = env::var("DATABASE_URL").expect("DATABASE_URL must be set");  
3     PgConnection::establish(&database_url)  
4         .unwrap_or_else(|_| panic!("Error connecting to {}", database_url))  
5 }
```

Podobnie jak we wcześniejszym programie, URL bazy danych jest pobierany z pliku `.env`.

Po poprawnym połączeniu z bazą danych, aplikacja przechodzi do kroków autoryzacji użytkownika.

Autoryzacja użytkownika

Przez fakt, że w bazie danych każdy użytkownik z tabeli `Users` powinien posiadać rolę z możliwością logowania do bazy danych. W aplikacji sprawdzane jest czy istnieje i rekord w tabeli `Users`, i rola z możliwością logowania. Oprócz tego sprawdzana jest poprawność hasła i nazwy użytkownika.

```
1 pub fn check_user(login: &Login) → bool {  
2     check_db_user(login) && check_gamevault_user(login)  
3 }  
4  
5 // Rola z możliwością logowania do bazy danych  
6 fn check_db_user(login: &Login) → bool {  
7     login.check_if_can_connect()  
8 }  
9  
10 // Rekord z tabeli Users  
11 fn check_gamevault_user(login: &Login) → bool {  
12     User::check_with_credentials(login)  
13 }
```



Rysunek 14: Autoryzacja użytkownika w aplikacji konsolowej

Po zalogowaniu się, użytkownikowi wyświetla się menu aplikacji.

Menu aplikacji

Menu aplikacji jest stworzone za pomocą biblioteki `terminal-menu`. Użytkownik może wybrać jedną z opcji, które są wyświetlane w konsoli. W zależności od roli użytkownika, wyświetlane są różne opcje - administracja widzi wszystkie, a użytkownik tylko te, na które ma uprawnienia.

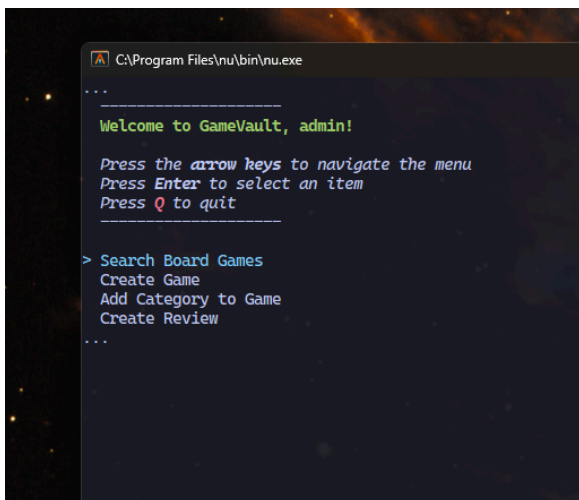
```
1 fn build_menu(user: &User) → TerminalMenu {  
2     let role = user.role.to_string();  
3  
4     return match role.as_str() {  
5         "ADMIN" ⇒ build_admin_menu(user),  
6         "USER" ⇒ build_user_menu(user),  
7         _ ⇒ panic!("Invalid role"),  
8     };  
9 }  
10  
11 // Menu dla administratora  
12 fn build_admin_menu(user: &User) → TerminalMenu {  
13     menu(vec![  
14         label("-----"),  
15         label(format!(cstr!("<bold,green>Welcome to GameVault, {}!</>"),  
16             user.username)),  
17         label(""),  
18         label(cstr!(  
19             "<italic>Press the <bold>arrow keys</> to navigate the menu</>"  
20         )),  
21         label(cstr!("<italic>Press <bold>Enter</> to select an item</>")),  
22         label(cstr!("<italic>Press <bold,red>Q</> to quit</>")),  
23         label("-----"),  
24         label(""),  
25         submenu(  
26             "Search Board Games",  
27             vec![
```



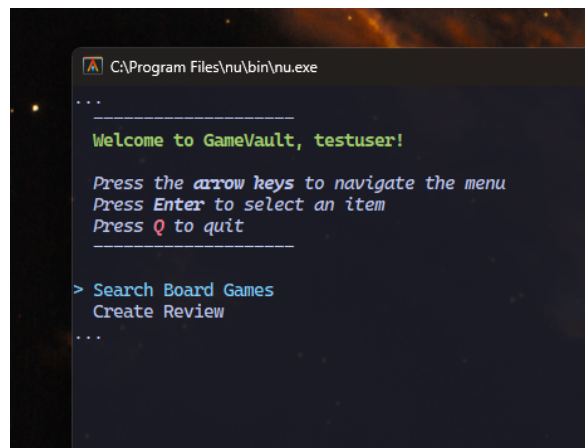
```

27         label("-----"),
28         label("Select a search criteria"),
29         label("-----"),
30         button("Search by Name"),
31         button("Search by Type"),
32         button("Search by Category"),
33         button("Search by Mechanic"),
34         back_button("Back"),
35     ],
36 ),
37 button("Create Game"),
38 button("Add Category to Game"),
39 button("Create Review"),
40 ])
41 }
42
43 // Menu dla użytkownika
44 fn build_user_menu(user: &User) → TerminalMenu {
45     ...
46 }

```



Rysunek 15: Menu aplikacji widziane przez administratora



Rysunek 16: Menu aplikacji widziane przez użytkownika

Wszystkie opcje w menu są interaktywne i użytkownik może wybrać jedną z nich za pomocą klawiszy strzałek i klawisza Enter. W zależności od wybranej opcji, aplikacja wykonuje odpowiednie akcje.

```

1 fn run_selected_action(selected: String, login: &Login) {
2     let mut conn = login.connect_or_panic();
3     let user = login.get_user().unwrap();
4
5     match selected.as_str() {
6         "Search by Name" => search_games::search_games_by_name(&mut conn),
7         "Search by Type" => search_games::search_games_by_type(&mut conn),
8         "Search by Category" => search_games::search_games_by_category(&mut conn),

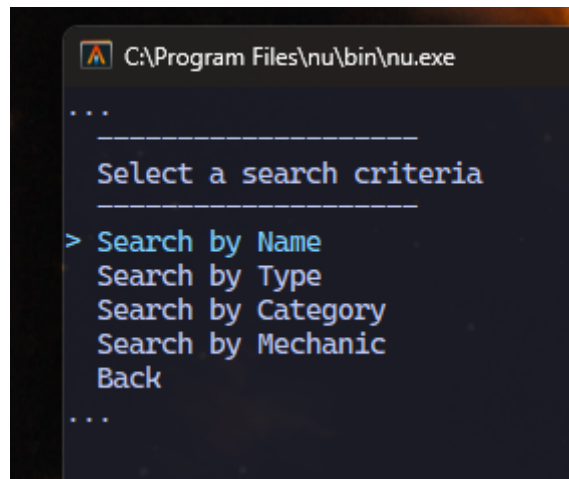
```

```

9      "Search by Mechanic" => search_games::search_games_by_mechanic(&mut conn),
10     "Create Game" => create_game::main(&mut conn),
11     "Add Category to Game" => add_category_to_game::main(&mut conn),
12     "Create Review" => create_review::main(&mut conn, &user),
13     _ => println!("Invalid selection"),
14 }
15 }

```

Opcja „Search Board Games” przenosi użytkownika do podmenu, gdzie może wybrać kryterium wyszukiwania.

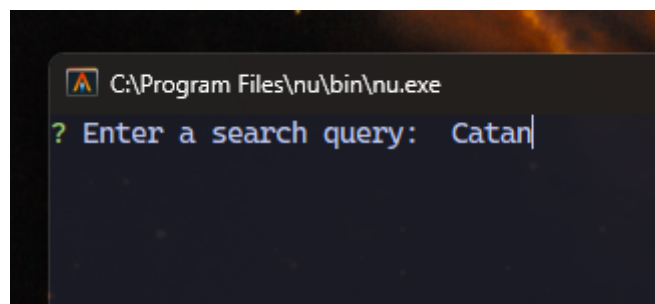


Rysunek 17: Podmenu wyszukiwania gier w aplikacji konsolowej

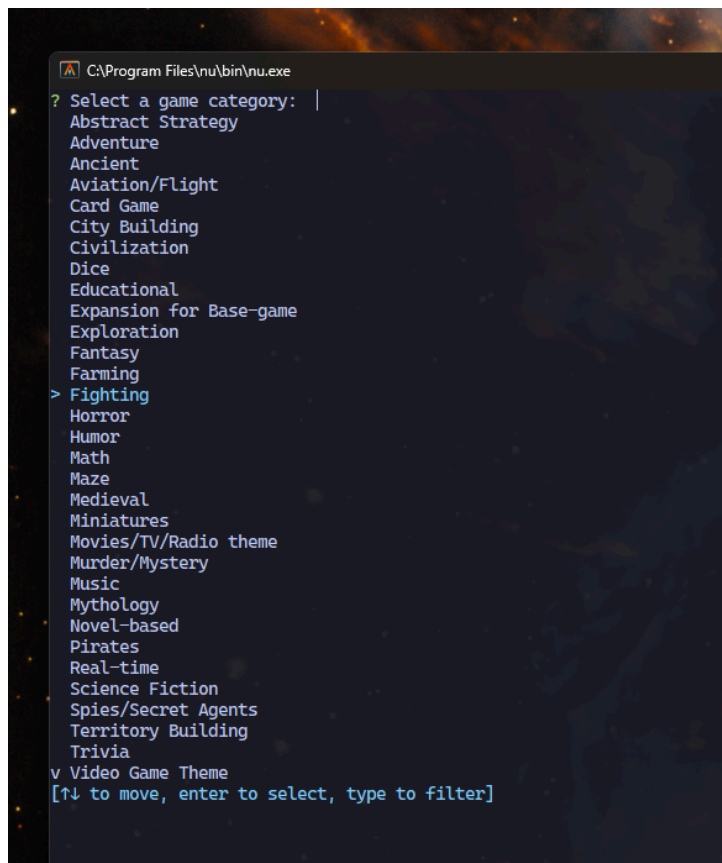
Opcje aplikacji

Wyszukiwanie gier

W aplikacji istnieje wiele sposobów na wyszukiwanie gier. Użytkownik może wyszukać grę po nazwie, typie, kategorii lub mechanice. Wyszukiwanie przez nazwę odbywa się standardowo - przez wpisanie części lub całości nazwy gry. Natomiast inne kryteria pozwalają na wybranie opcji z listy.

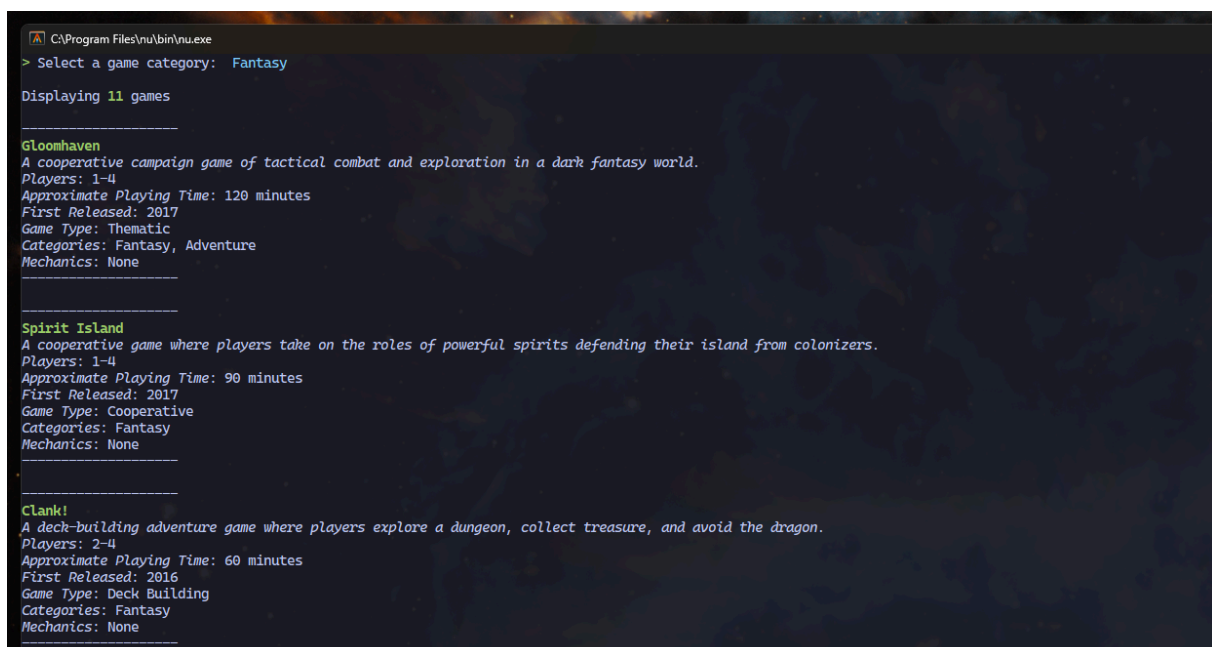


Rysunek 18: Wyszukiwanie gier po nazwie (wpisanie nazwy)



Rysunek 19: Wyszukiwanie gier po kategorii (wybranie z listy)

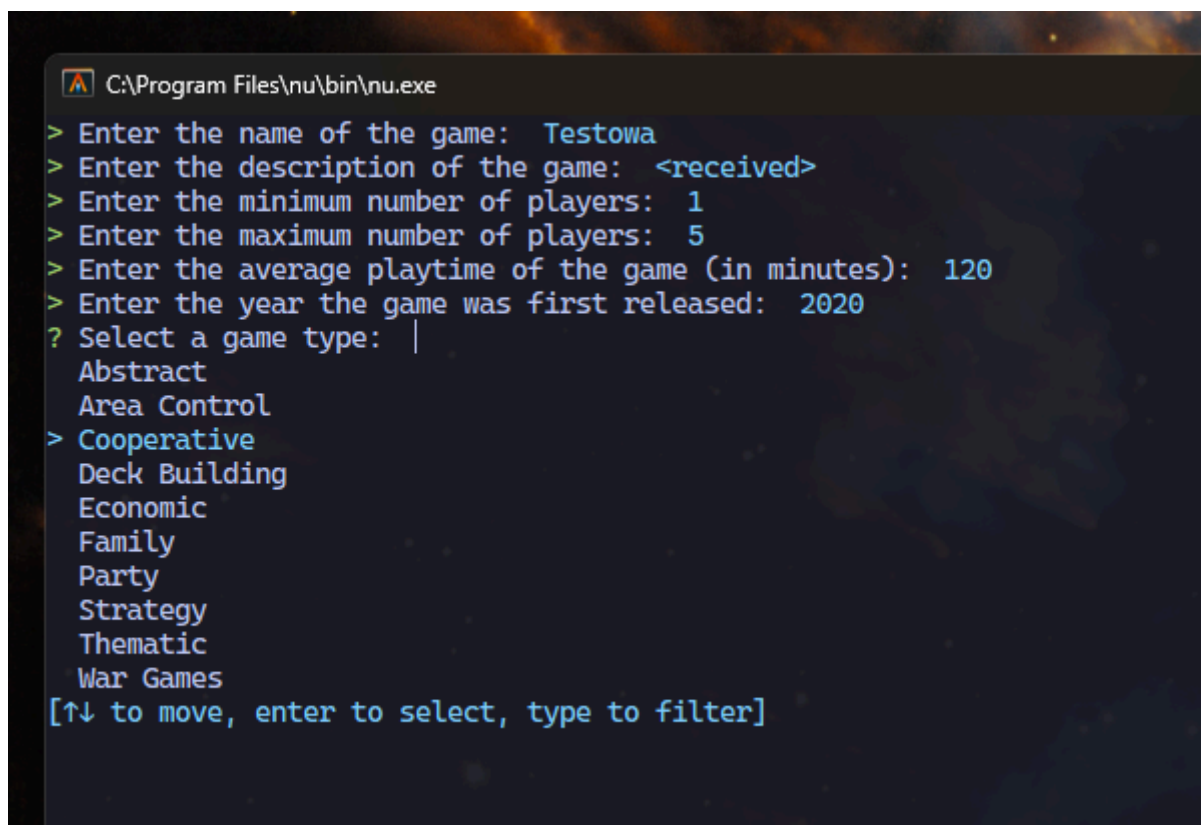
Po wpisaniu/wybraniu kryterium, aplikacja wyświetla wyniki w konsoli.



Rysunek 20: Przykładowe wyniki wyszukiwania gier wyświetlane w konsoli

Dodawanie gry

Opcja „Create Game” pozwala na dodanie nowej gry do bazy danych. Użytkownik wprowadza dane o grze, takie jak nazwa, opis, minimalna i maksymalna ilość graczy, czas gry, rok pierwszego wydania oraz typ gry.



Rysunek 21: Widok dodawania nowej gry w aplikacji

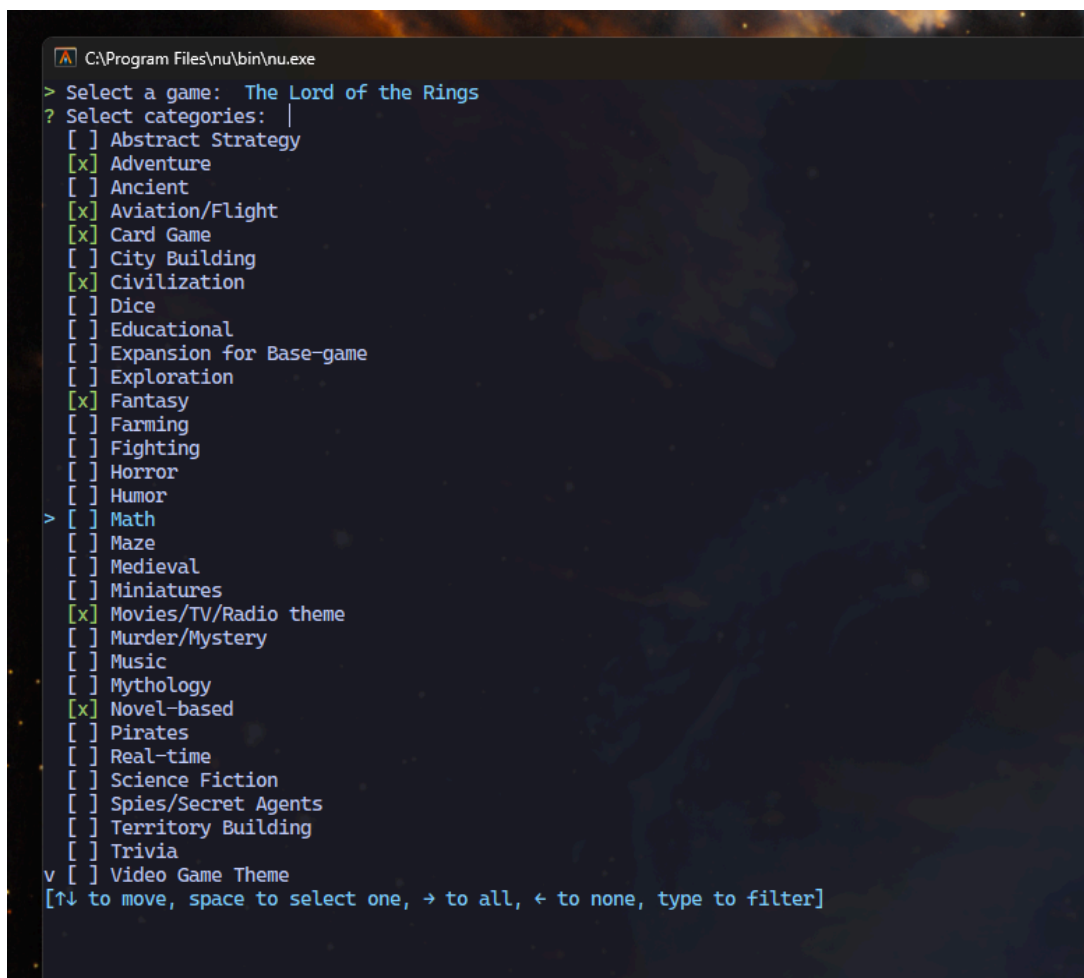
Po wprowadzeniu danych i zatwierdzeniu, aplikacja dodaje nową grę do bazy danych i informuje użytkownika o sukcesie.

Dodawanie kategorii do gry

Następna opcja - „Add Category to Game” pozwala na dodanie kategorii do istniejącej gry. Użytkownik wybiera grę z listy, a następnie wybiera kategorie z listy wielokrotnego wyboru.

Jako że kategorie posiadają relację wiele do wielu z grami, pierwsze sprawdzamy jakie kategorie są już przypisane do gry i umieszczamy je na liście wyboru jako domyślnie zaznaczone.

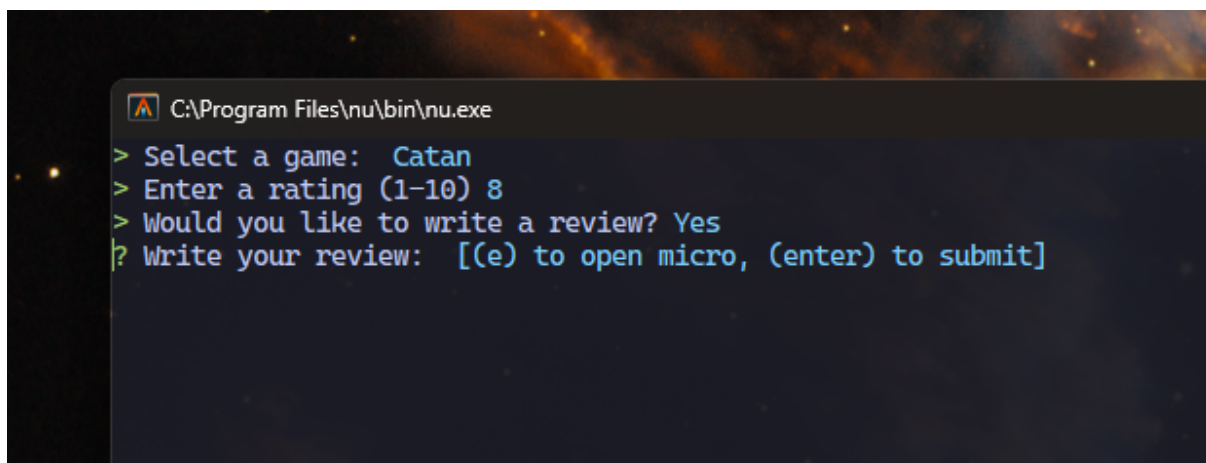
Aplikacja po zatwierdzeniu wyboru, pierwsze usuwa wszystkie kategorie przypisane do gry, a następnie dodaje nowe.



Rysunek 22: Widok dodawania kategorii do gry w aplikacji

Dodawanie recenzji

Ostatnią opcją jest możliwość dodania recenzji do gry. Użytkownik wybiera grę z listy, a następnie wprowadza ocenę i recenzję.



Rysunek 23: Widok dodawania recenzji do gry w aplikacji

Modele

W aplikacji korzystam z modeli, które reprezentują struktury danych w bazie danych. Modele są zdefiniowane w folderze `models` i zawierają strukturę i metody związane z danymi.

Przykładowy model `game_review`:

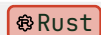
```
1 // Struktura modelu recenzji gry Rust
2 #[derive(Identifiable, Selectable, Queryable, Associations, Debug)]
3 #[diesel(belongs_to(Game))]
4 #[diesel(belongs_to(User))]
5 #[diesel(table_name = schema::game_reviews)]
6 #[diesel(primary_key(game_id, user_id))]
7 pub struct GameReview {
8     pub id: i32,
9     pub game_id: i32,
10    pub user_id: i32,
11    // Number from 1 to 10
12    pub rating: i32,
13    pub review: Option<String>,
14 }
15
16 // Struktura formularza do dodawania recenzji
17 #[derive(Insertable)]
18 #[diesel(table_name = schema::game_reviews)]
19 pub struct GameReviewForm {
20     pub game_id: i32,
21     pub user_id: i32,
22     pub rating: i32,
23     pub review: Option<String>,
24 }
25
26 // Implementacja metod dla struktury modelu recenzji gry
27 impl GameReview {
28     // Metoda do dodawania recenzji do bazy danych
29     pub fn insert(conn: &mut PgConnection, game_id: i32, user_id: i32, rating:
i32, review: Option<String>) → QueryResult<usize> {
30         let form = GameReviewForm {
31             game_id, user_id, rating, review,
32         };
33
34         diesel::insert_into(schema::game_reviews::table)
35             .values(&form)
36             .execute(conn)
37     }
38 }
```

Pomocnicze schematy tabel

Oprócz modeli, diesel potrzebuje również schematów tabel, które definiują strukturę bazy danych. Oprócz tego informują też o relacjach i kluczach obcych między tabelami. Schematy są zdefiniowane w pliku `schema.rs`.

Przykładowy schemat tabeli `game_reviews`:

```
1 diesel::table! {
2     #[sql_name = "GameReviews"]
3     game_reviews (id) {
4         id → Integer,
5         game_id → Integer,
6         user_id → Integer,
7         rating → Integer,
8         review → Text,
9     }
10 }
11
12 diesel::joinable!(game_reviews → games (game_id));
13 diesel::joinable!(game_reviews → users (user_id));
14 diesel::allow_tables_to_appear_in_same_query!(game_reviews, games, users);
```



PODSUMOWANIE

W projekcie stworzyłem bazę danych w PostgreSQL, która przechowuje informacje o grach planszowych, recenzjach, kategoriach, mechanikach, wydaniach oraz użytkownikach. Baza danych jest kompleksowa i zawiera wiele tabel, relacji wiele do wielu oraz wiele do jednego.

Stworzyłem również role i uprawnienia, które pozwalają na kontrolę dostępu do danych w bazie. Role są zdefiniowane w taki sposób, aby użytkownicy mieli ograniczony dostęp do danych, a administratorzy mieli pełne uprawnienia.

Wdrożenie bazy danych na serwerze lokalnym zrealizowałem za pomocą konteneryzacji. Użyłem Docker'a i Docker Compose do uruchomienia kontenera z bazą danych PostgreSQL. Jest to jeden z najłatwiejszych i najszybszych sposobów na uruchomienie bazy danych.

W projekcie wykorzystałem również funkcje i procedury, które pomagają w automatyzacji zadań i wykonywaniu bardziej skomplikowanych operacji na bazie danych. Funkcje pozwalają na zwracanie zestawów danych, podczas gdy procedury są bardziej podobne do skryptów, które wykonują operacje na bazie danych.

W ostatnim etapie stworzyłem dwie aplikacje w języku Rust, które komunikują się z bazą danych PostgreSQL. Pierwsza aplikacja korzysta z biblioteki `sqlx` do wykonywania prostych zapytań SQL. Druga - ORM, korzysta z biblioteki `diesel` do zarządzania bazą danych. Aplikacja ta pozwala na dodawanie gier i recenzji, przypisywanie kategorii do gier oraz wyszukiwanie gier w bazie danych. Obsługiwana jest przez konsolowy interfejs.

Wnioski

Tworzenie i zarządzanie bazą danych w PostgreSQL jest stosunkowo proste i intuicyjne. PostgreSQL oferuje wiele funkcji, które ułatwiają pracę z bazą danych, takie jak role, uprawnienia, funkcje, procedury, a także wsparcie dla transakcji i zapytań zagnieżdżonych.

Jedynym problemem, na jaki natrafiłem, było zrozumienie konceptu ról, który zastępuje tradycyjnych użytkowników i grupy. Na szczęście PostgreSQL jest bardzo popularnym silnikiem bazodanowym i istnieje wiele materiałów i dokumentacji, które pomagają w zrozumieniu tych koncepcji.

W projekcie wykorzystałem również język Rust, który jest językiem systemowym i coraz bardziej popularnym językiem programowania. Rust oferuje wiele zalet, takich jak bezpieczeństwo pamięci, wydajność i wygodne narzędzia do zarządzania zależnościami i budowania aplikacji. Biblioteki `sqlx` i `diesel` są doskonałymi narzędziami do komunikacji z bazą danych. Ich dokumentacje są bardzo dobre i zawierają wiele przykładów i porad.

Konfrontacja z innym silnikami bazodanowymi

W projekcie skupiłem się na bazie danych PostgreSQL, ale warto porównać ją z innym popularnym silnikiem bazodanowym - MySQL.

PostgreSQL i MySQL to dwa najpopularniejsze otwarte silniki bazodanowe. Oba są bardzo wydajne i oferują wiele funkcji, takich jak transakcje, indeksy, widoki, procedury składowane i wiele innych.

Główne różnice między PostgreSQL a MySQL

- **Zgodność z SQL** - PostgreSQL jest znany ze ścisłego przestrzegania standardów SQL i obsługuje szerszy zakres funkcji SQL w porównaniu do MySQL. Z drugiej strony, MySQL posiada kilka własnych rozszerzeń SQL niedostępnych w innych silnikach bazodanowych;
- **Typy danych** - PostgreSQL ma bardziej rozbudowany zestaw typów danych, w tym obsługę tablic, JSON i zaawansowanych typów danych, takich jak typy zakresów i geometryczne typy danych;
- **Kontrola współbieżności** - PostgreSQL wykorzystuje bardziej zaawansowany system Multi-Version Concurrency Control (MVCC), który zapewnia lepszą wydajność i izolację dla współbieżnych transakcji. MySQL używa blokowania na poziomie wiersza, co może prowadzić do wyższego narzutu blokowania;
- **Licencjonowanie** - PostgreSQL korzysta z licencji PostgreSQL, wolnej i otwartej licencji podobnej do MIT. MySQL jest dostępny na licencji GPL i licencjach własnościowych;
- **Wydajność** - PostgreSQL jest zoptymalizowany pod kątem złożonych zapytań i operacji zapisu, podczas gdy MySQL jest generalnie szybszy w przypadku prostych zapytań i operacji odczytu

Wnioski z porównania

Oba silniki bazodanowe są bardzo rozbudowane i wybór między nimi zależy od konkretnych wymagań projektu. W moim przypadku wybrałem PostgreSQL ze względu na moją wcześniejszą znajomość tego rozwiązania, jak i fakt że jest on w pełni otwartoźródłowy.