POLITECHNIKA RZESZOWSKA

Bazy danych - Projekt

System zarządzania kolekcją gier planszowych

Części 1., 2. i 3.

Data wykonania: 8.06.2024 r.

Kamil Pomykała L03

Część 1.

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 posłuży 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.

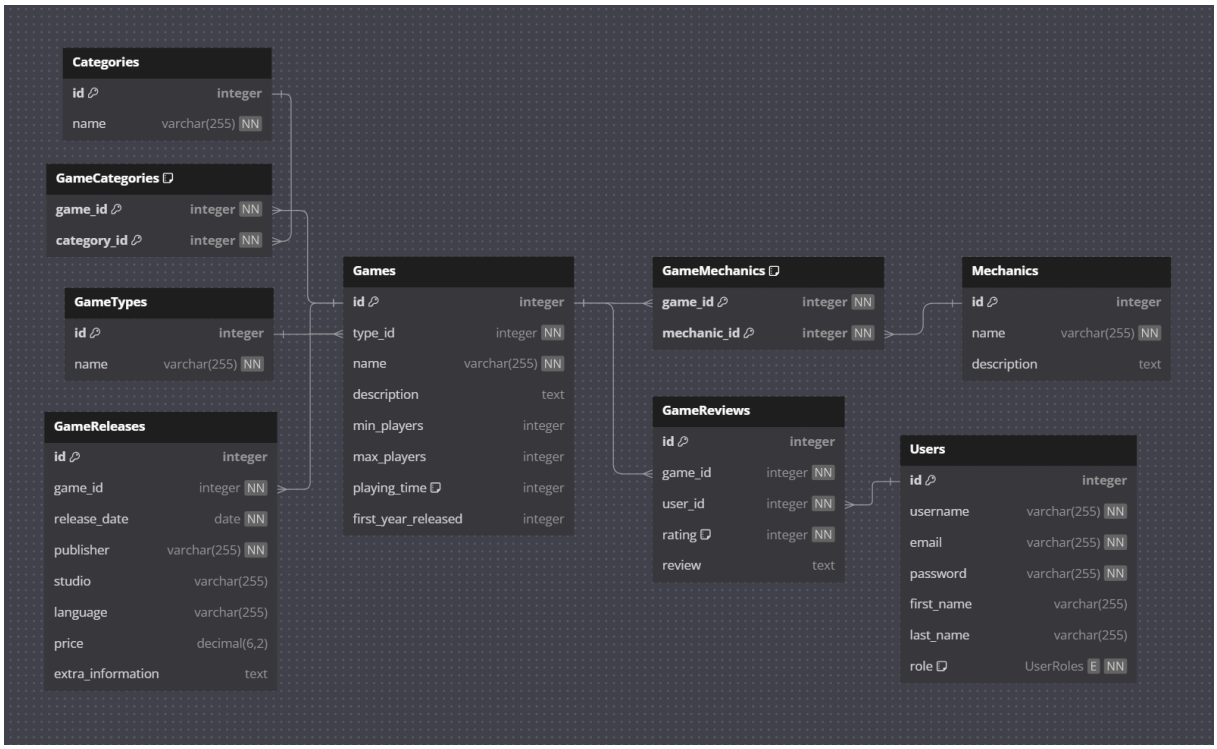
Kontrola wersji projektu zostanie zrealizowana za pomocą narzędzia **Git** i platformy **GitHub**. Git umożliwia zarządzanie kodem źródłowym projektu, a GitHub umożliwia przechowywanie kodu źródłowego w chmurze. Repozytorium projektu będzie dostępne publicznie pod tym linkiem.

Część 2.

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.



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.

```sql
CREATE TYPE "UserRoles" AS ENUM ( 'ADMIN', 'USER' );

CREATE TABLE "Users"
(
    "id"         serial PRIMARY KEY,
    "username"   varchar(255) NOT NULL,
    "email"      varchar(255) NOT NULL,
    "password"   varchar(255) NOT NULL,
    "first_name" varchar(255),
    "last_name"  varchar(255),
    "role"       "UserRoles"  NOT NULL
);

CREATE TABLE "GameTypes"
(
    "id"   serial PRIMARY KEY,
    "name" varchar(255) UNIQUE NOT NULL
);

CREATE TABLE "Games"
(
    "id"                 serial PRIMARY KEY,
    "type_id"            integer      NOT NULL REFERENCES "GameTypes" ("id"),
    "name"               varchar(255) NOT NULL,
    "description"        text,
    "min_players"        integer,
    "max_players"        integer,
    "playing_time"       integer,
    "first_year_released" integer
);

CREATE TABLE "Categories"
(
```

```
    "id"   serial PRIMARY KEY,
    "name" varchar(255) UNIQUE NOT NULL
);

CREATE TABLE "GameCategories"
(
    "game_id"     integer NOT NULL REFERENCES "Games" ("id"),
    "category_id" integer NOT NULL REFERENCES "Categories" ("id"),
    CONSTRAINT "pk_game_categories" PRIMARY KEY ("game_id", "category_id")
);

COMMENT ON COLUMN "Games"."playing_time" IS 'Approximate playing time in minutes';

CREATE TABLE "Mechanics"
(
    "id"          serial PRIMARY KEY,
    "name"        varchar(255) UNIQUE NOT NULL,
    "description" text
);

CREATE TABLE "GameMechanics"
(
    "game_id"     integer NOT NULL REFERENCES "Games" ("id"),
    "mechanic_id" integer NOT NULL REFERENCES "Mechanics" ("id"),
    CONSTRAINT "pk_game_mechanics" PRIMARY KEY ("game_id", "mechanic_id")
);

CREATE TABLE "GameReviews"
(
    "id"      serial PRIMARY KEY,
    "game_id" integer NOT NULL REFERENCES "Games" ("id"),
    "user_id" integer NOT NULL REFERENCES "Users" ("id"),
    "rating"  integer NOT NULL,
    "review"  text
);

COMMENT ON COLUMN "GameReviews"."rating" IS 'Rating from 1 to 10';

CREATE TABLE "GameReleases"
(
    "id"               serial PRIMARY KEY,
    "game_id"          integer      NOT NULL REFERENCES "Games" ("id"),
    "release_date"     date         NOT NULL,
    "publisher"        varchar(255) NOT NULL,
    "studio"           varchar(255),
    "language"         varchar(255),
    "price"            decimal(6, 2),
```

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

Zapytania wprowadzające dane

Poniżej znajdują się zapytania SQL wprowadzające przykładowe dane do tabel.

Tabela `GameCategories`

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

Tabela `Games`

```SQL
INSERT INTO "Games" (type_id, name, description, min_players, max_players, playing_time, first_year_released) VALUES
    ((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 strategy game for two players involving surrounding and capturing your opponent''s stones.', 2, 2, 60, -2200),
    ...
```

Tabele `Categories` i `GameCategories`

```SQL
INSERT INTO "Categories" (name) VALUES
    ('Word Games'), ('Spies/Secret Agents'), ('Humor'), ('Fantasy'), ('Science Fiction'), ('Adventure'), ('Novel-based'), ('Horror'), ('Territory Building'), ('Medieval');

INSERT INTO "GameCategories" (game_id, category_id) VALUES
    ((SELECT id FROM "Games" WHERE name = 'Codenames'), (SELECT id FROM "Categories" WHERE name = 'Word Games')),
    ((SELECT id FROM "Games" WHERE name = 'Codenames'), (SELECT id FROM "Categories" WHERE name = 'Spies/Secret Agents')),
    ...
```

Tabele `Mechanics` i `GameMechanics`

```SQL
INSERT INTO "Mechanics" (name) VALUES
    ('Tile Placement'), ('Hand Management'), ('Dice Rolling'), ('Team-Based Game'), ('Trading'), ('Memory'), ('Auction/Bidding'), ('Map Addition'), ('Player Elimination'), ('Deck Building'), ('Income'), ('End Game Bonuses');

INSERT INTO "GameMechanics" (game_id, mechanic_id) VALUES
    ((SELECT id FROM "Games" WHERE name = 'Carcassonne'), (SELECT id FROM "Mechanics" WHERE name = 'Tile Placement')),
    ((SELECT id FROM "Games" WHERE name = 'Carcassonne'), (SELECT id FROM "Mechanics" WHERE name = 'Map Addition')),
    ...
```

Tabele `GameReviews` **i** `Users`

```sql
INSERT INTO "Users" (username, email, password, role) VALUES
    ('admin', '174725@stud.prz.edu.pl', crypt('kamil123', gen_salt('bf', 10)), 'ADMIN'),
    ('testuser', 'kpomykala2002@gmail.com', crypt('test123', gen_salt('bf', 10)), 'USER');

INSERT INTO "GameReviews" (game_id, user_id, rating, review) VALUES
    ((SELECT id FROM "Games" WHERE name = 'Carcassonne'), (SELECT id FROM "Users" WHERE username = 'testuser'), 4, 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam id consequat lacus. Cras ultricies, nunc molestie placerat tincidunt, enim sapien imperdiet nulla, sit amet tincidunt felis lorem pretium erat.'),
    ...
```

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`

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

Przykładowe zapytania selekcyjne

Zapytanie o gry z kategorii „Fantasy”

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

Zapytanie o gry z mechaniką „Hand Management”

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

Zapytanie o gry, które posiadają średnią recenzji powyżej 4

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

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

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

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

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

Zapytanie o gry wydane po 2010 roku

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

Część 3.

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%

```SQL
UPDATE "GameReleases"
SET "price" = "price" * 1.25
WHERE "language" = 'German';
```

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

```SQL
UPDATE "GameReleases"
SET "price" = "price" * 0.9
FROM "Games"
        INNER JOIN "GameMechanics" ON "Games"."id" = "GameMechanics"."game_id"
        INNER JOIN "Mechanics" ON "GameMechanics"."mechanic_id" = "Mechanics"."id"
WHERE "Games"."id" = "GameReleases"."game_id"
  AND "Mechanics"."name" = 'Hand Management';
```

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

```SQL
UPDATE "GameReleases"
SET "price" = "price" * 1.15
WHERE "game_id" IN (
  SELECT "game_id"
  FROM "GameReviews"
  WHERE "GameReviews"."rating" ⩾ 4
);
```

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`”.

```sql
UPDATE "Users"
SET "role" = CASE
        WHEN (SELECT COUNT(*) FROM "GameReviews" WHERE "Users"."id" = "GameReviews"."user_id") > 5 THEN 'ADMIN'
    ELSE 'USER'
END
FROM "GameReviews"
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.

```sql
UPDATE "Mechanics"
SET "description" =
    CASE
        WHEN (SELECT COUNT(*) FROM "GameMechanics" WHERE "Mechanics"."id" = "GameMechanics"."mechanic_id") = 0 THEN 'This mechanic is not used in any game'
    ELSE
        'This mechanic is used in '
            || (SELECT COUNT(DISTINCT "game_id") FROM "GameMechanics" WHERE "Mechanics"."id" = "GameMechanics"."mechanic_id")
        || ' games and '
        || (SELECT COUNT(DISTINCT "type_id")
            FROM "Games"
            WHERE "Games"."id" IN (SELECT "game_id"
                                   FROM "GameMechanics"
                                   WHERE "Mechanics"."id" = "GameMechanics"."mechanic_id"))
        || ' game types'
    END
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

```SQL
SELECT "Games"."name", SUM("GameReleases"."price") AS "price_sum",
COUNT("GameReleases"."price") AS "number_of_releases"
FROM "Games"
INNER JOIN "GameReleases" ON "Games"."id" = "GameReleases"."game_id"
GROUP BY "Games"."id"
HAVING SUM("GameReleases"."price") ⩾ 200;
```

Zapytanie o ranking użytkowników na podstawie ilości napisanych recenzji

Z pominięciem użytkowników, którzy nie napisali żadnej recenzji.

```SQL
SELECT
    "Users"."username",
    ROUND(AVG("GameReviews"."rating"), 3) AS "average_rating",
    COUNT("GameReviews"."rating")         AS "number_of_ratings",
    (SELECT "GameReviews"."review"
    FROM "GameReviews"
    WHERE "GameReviews"."user_id" = "Users"."id"
    ORDER BY "GameReviews"."id" DESC
    LIMIT 1)                              AS "latest_review"
FROM "Users"
LEFT JOIN "GameReviews" ON "Users"."id" = "GameReviews"."user_id"
GROUP BY "Users"."id"
HAVING COUNT("GameReviews"."rating") > 0
ORDER BY COUNT("GameReviews"."rating") DESC;
```

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

```sql
SELECT
    "Games"."name",
    (SELECT "GameReleases"."release_date"
    FROM "GameReleases"
    WHERE "GameReleases"."game_id" = "Games"."id"
    ORDER BY "GameReleases"."release_date" DESC
    LIMIT 1) AS "latest_release",
    (SELECT "GameReleases"."release_date"
    FROM "GameReleases"
    WHERE "GameReleases"."game_id" = "Games"."id"
    ORDER BY "GameReleases"."release_date" ASC
    LIMIT 1) AS "first_release",
    "Games"."first_year_released"
FROM "Games"
RIGHT JOIN "GameReleases" ON "Games"."id" = "GameReleases"."game_id"
GROUP BY "Games"."id";
```

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

```sql
SELECT
    "GameTypes"."name",
    ROUND(AVG("Games"."min_players"), 2) AS "average_min_players",
    ROUND(AVG("Games"."max_players"), 2) AS "average_max_players"
FROM "GameTypes"
INNER JOIN "Games" ON "GameTypes"."id" = "Games"."type_id"
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.

```sql
SELECT
    "Categories"."name",
    COUNT("Games"."id")                                    AS "game_count",
    STRING_AGG("Games"."name", ', ')                       AS "game_list",
    (SELECT ROUND(AVG("GameReviews"."rating"), 2)
    FROM "GameReviews"
             INNER JOIN "Games" ON "GameReviews"."game_id" = "Games"."id"
         INNER JOIN "GameCategories" ON "Games"."id" = "GameCategories"."game_id"
    WHERE "GameCategories"."category_id" = "Categories"."id") AS "average_rating"
FROM "Categories"
INNER JOIN "GameCategories" ON "Categories"."id" = "GameCategories"."category_id"
INNER JOIN "Games" ON "GameCategories"."game_id" = "Games"."id"
GROUP BY "Categories"."id";
```