Projekt: grafowa baza danych

Tomasz Maczek

Przetwarzanie danych w chmurach obliczeniowych
WFiIS

3 Grudnia 2022

Spis treści

1. Opis tematu projektu

Projekt dotyczy serialu animowanego **Steven Universe**. W serialu tym ludzie w mieście Beach City żyją razem z rasą kosmitów zwanych Kryształami (Gems), które mają wiele ciekawych właściwości, na czele z faktem, że mogą dokonywać fuzji pomiędzy sobą. Tytułowy bohater jest pół człowiekiem - pół kryształem.

Serial miał 5 sezonów, w sumie 160 odcinków. Każdy odcinek miał przynajmniej 2 autorów (Writer).

W projekcie tym będą dostępne informacje (samplowe) o :

- Postaciach
- Odcinkach serialu
- Autorach odcinków
- Fuzjach postaci
- Grupach postaci w serialu

Link: `https://chmury-obliczeniowe-2022-tmaczek.onrender.com/`

2. Grafowa baza danych

Informacje przechowywane są w grafowej bazie danych **Neo4J AuraDB**. Wykorzystywany jest darmowy program. Stworzona jest tam instancja, z którą strona się łączy i operuje na danych.

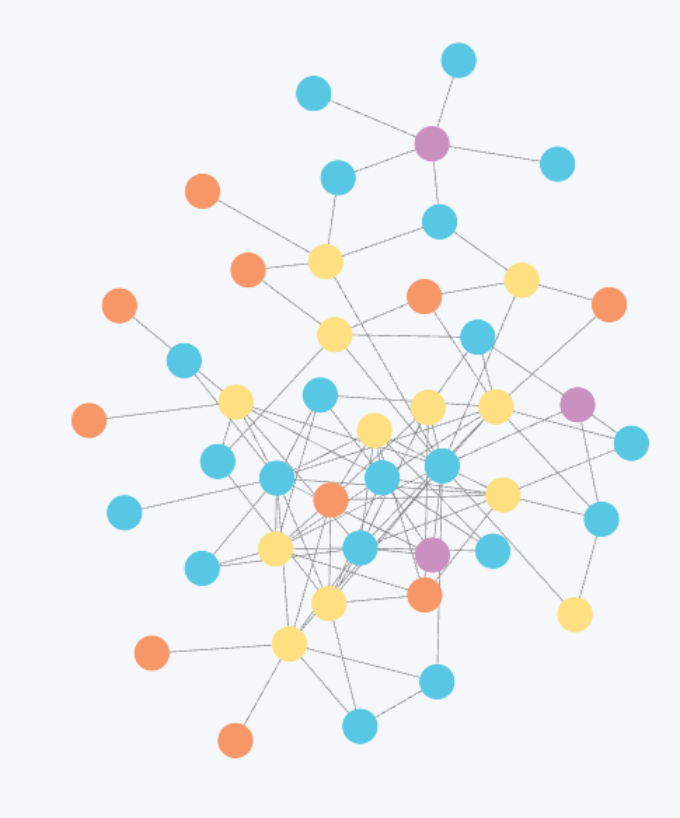
W grafowej bazie danych informacje znajdują się w węzłach (nodes), które połączone są krawędziami (vertices). Na krawędzi również mogą się znajdować dodatkowe informacje, jak również sama krawędź przekazuje informacje (jest nieco jak relacja).

W tym projekcie użyte zostały 4 typy węzłów. Na każdy z nich został narzucony constraint, by miał unikalną nazwę, jak również constraint dla odcinka by miał unikalny numer dla całości serii.

Węzły:

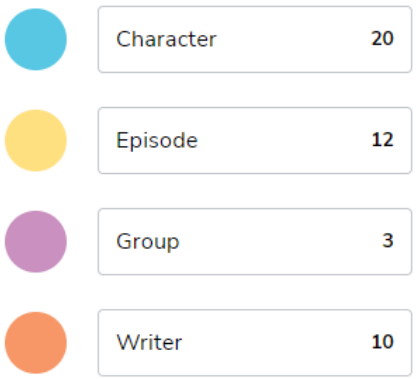
- Character - postać, atrybuty: name
- Episode - odcinek, atrybuty:
  - name,
  - number - number w sezonie (sezon nie ma więcej niż 52 odcinki),
  - season - sezon, jest 5 sezonów serialu,
  - overall - numer liczony dla całości serii, tj od 1 do 160
- Group - grupy w serialu, atrybuty: name
- Writer - autor piszący odcinki, atrybut: name

Pomiędzy różnymi typami węzłów mogą istnieć różne krawędzie (relacje).



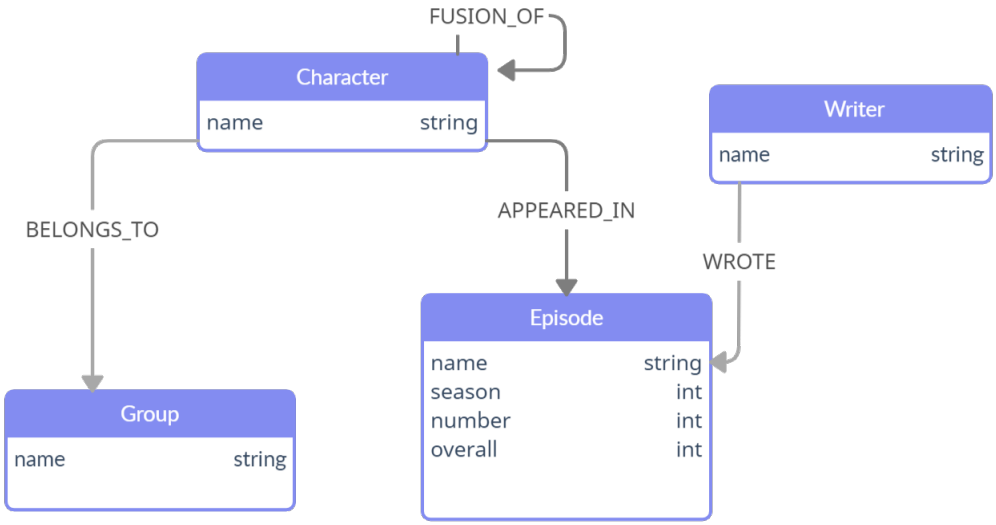
Rysunek 2.1: Widok całej stworzonej bazy

- (Character) - APPEARED_IN -> (Episode)
  Wskazuje na to w jakich odcinkach pojawia się dana postać.
- (Character) - BELONGS_TO -> (Group)
  Przypisuje postać do jednej z grup. Postać może należeć do wielu grup.
- (Writer) - WROTE -> (Episode)
  Przypisuje autora do odcinka. Jeden ocinek ma zwykle 2 autorów, ale może mieć więcej.
- (Character) -FUSION_OF -> (Character)
  Przypisuje do pierwszej postaci drugą, która jest częścią jej fuzji. W serialu fuzje mogą być z więcej niż 2 postaci, w tym projekcie dla uproszczenia zakładamy że są z dwóch.



Rysunek 2.2: Typy węzłów i ich ilość

Dla przykładowych danych w projekcie mamy 45 węzłów i 100 relacji.



Rysunek 2.3: Diagram UML

3. Przykład danych z serialu

Aby lepiej poczuć koncepcję projektu podam przykładowe dane i relacje, które można dodać do projektu.

- **Sunstone** - nowy Character

- **Change Your Mind** - nowy Episode - s5e29, overall 157

- **Rebecca Sugar** - twórczyni serialu i Writer

- Sunstone to fuzja (Fusion) 2 postaci w bazie: Steven + Garnet
  Sunstone - FUSION_OF -> Steven
  Sunstone - FUSION_OF -> Garnet

- Sunstone pojawia się w odcinku Change Your Mind
  Sunstone - APPEARED_IN - > Change Your Mind
- Rebecca Sugar jest jednym z autorów Change Your Mind
  Rebecca Sugar - WROTE -> Change Your Mind
- Sunstone można dodać do grupy Crystal Gems (jest fuzją dwojga z nich)
  Sunstone - BELONGS_TO -> Crystal Gems

4. Użyte technologie

- Baza danych
  - Neo4J AuraDB Free
  - Cypher Query Language
- Strona internetowa
  - Python
  - Flask i FlaskForms
  - neo4j - biblioteka Pythona
  - HTML
  - Bootstrap
- Hosting
  - Render
  - GitHub (źródło do deployów Rendera)
- IDE - PyCharm

5. Wdrożenie projektu

5.1 Wdrożenie lokalne

Aby uruchomić projekt na własnym komputerze należy:

1. Pobrać pliki projektu.
2. Zainstalować wymagane biblioteki z **requirements.txt**
3. Stworzyć w Neo4J AuraDB pustą instancję.
4. Storzyć w katalogu projektu plik **.env** i zapisać w nim jako zmienne środowiskowe dane logowania do instancji Neo4J.
5. Stworzyć tam również **SECRET_KEY** potrzebny dla FlaskForms - można go wymyślić, można wygenerować za pomocą Pythona:
   import secrets
   secrets.token_hex(16)
6. W konsoli Pythona uruchomić **flask –app app run**. Drugie **app** to nazwa naszego pliku. W razie potrzeby wspomóc się dokumentacją: `https://flask.palletsprojects.com/en/2.2.x/quickstart/`.
   Aplikacja dostępna jest wtedy pod `http://127.0.0.1:5000` .

   Można też uruchomić poprzed opcję **Run** w PyCharm - rozpozna on, że applikacja jest we Flasku i również uruchomi stronę.
7. Aby wypełnić bazę, przy pierwszym uruchomieniu programu można w **app.py** użyć metody **def add_series_data** z klasy **DatabaseApp**. Można ją wkleić do bloku main lub 'globalnie' (ważne by wcześniej połączyć się bazą i stworzyć obiekt DatabaseApp).

5.2 Wdrożenie na Render

Niedawno straciliśmy możliwość wrzucania aplikacji na IBMCloud i Heroku, gdzie to drugie było wyjątkowo popularne ze względu na łatwość używania. Szukając innych opcji trafiłem na **Render** (`https://render.com/`) gdzie można stworzyć Web Service. Jest on wyjątkowo łatwy w użyciu: aby postawić serwis należy:

1. Połączyć konto na Renderze z kontem na GitHubie.
2. Mieć swój projekt wrzucony na repozytorium.
3. Przy tworzeniu Web Service wybrać repozytorium na którym jest nasza aplikacja.

Plusy Rendera to automatyczny deploy przy każdym commicie na GitHubie, łatwość cofnięcia do starszego commitu i ogólna przejrzystość użycia.
Minusami jest relatywnie wolne pierwsze ładowanie strony (co jest rozumiane przez darmowy plan) oraz czasami pojawiające się błędy w deployu z ich strony (w moim przypadku 2 razy - za 3 razem poprawnie).

6. Opis kodu programu

Struktura plików w katalogu projektu to:

- *static* - katalog z obrazami i css,
- *templates* - katalog z templetkami HTML'a,
- **DatabaseApp.py** - plik ze stworzoną klasą DatabaseApp, w której są metody dotyczące operowania na bazie grafowej,
- **app.py** - plik aplikacji Flaska, w którym zdefiniowane jest routowanie, generowanie formularzy i komunikacja przez DatabaseApp z bazą,
- **forms.py** - zawiera definicje formularzy jako klasy dziedziczące po FlaskForm,
- .env - zmienne środowiskowe,
- requirements.txt - wymagania bibliotek Pythona,
- .gitignore - głównie by nie wrzucić pliku .env na repozytorium i tym sposobem nie udostępnić światu danych połączenia z bazą (repozytorium jest obecnie prywatne, jednak jest to bezpieczna praktyka).

6.1 DatabaseApp.py

W pliku tym używając biblioteki **neo4j** z **GraphDatabase** stworzone są metody do:

- tworzenia i zamykania połączenia z bazą,
- tworzenia rekordów i relacji,

- sprawdzania czy rekord lub relacja isnieje,
- usuwania rekordu/relacji/całości bazy,
- dodawania przykładowych danych,
- tworzenia odpowiednich podstron i inne funkcje pomocnicze do nich

6.2 app.py

Jest to główny plik aplikacji bazujący na technologii Flask. Zadania w nim wykonywane to:

- zdefiniowane routingu do podstron,
- łączenie z bazą, przetwarzanie i wyświetlanie otrzymanych danych,
- tworzenie formularzy do dodawania i usuwania elementów bazy

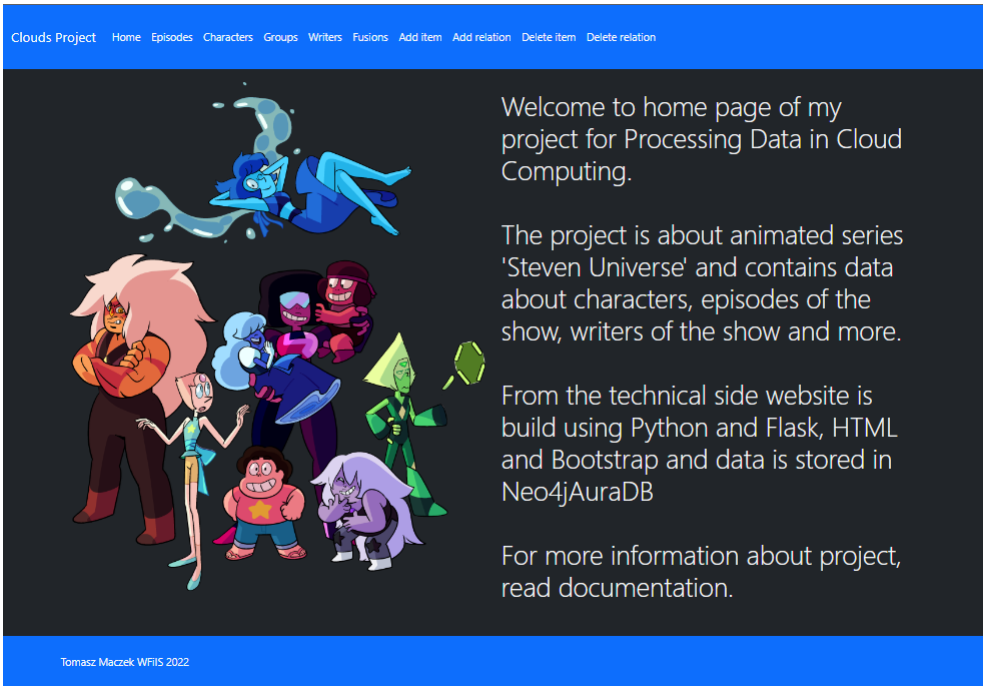
6.3 forms.py

W tym pliku tworzymy klasy odpowiadające każdemu formularzowi na stronie. Bazują one na FlaskForm. Formularze tam zdefiniowane można podzielić na grupy:

- Tworzenie węzłów: **CharacterForm**, **WriterForm**, **EpisodeFrom**
- Tworzenie relacji: **CharacterToEpisode**, **CharacterToGroup**, **WriterToEpisode**, **CharactersToFusion**
- Usuwanie węzłów: **DeleteCharacter**, **DeleteEpisode**, **DeleteWriter**

W każdej z tych klas definiujemy jakie pola ma formularz mieć i przez jakie walidacje ma przechodzić.

7. Funkcjonalność strony

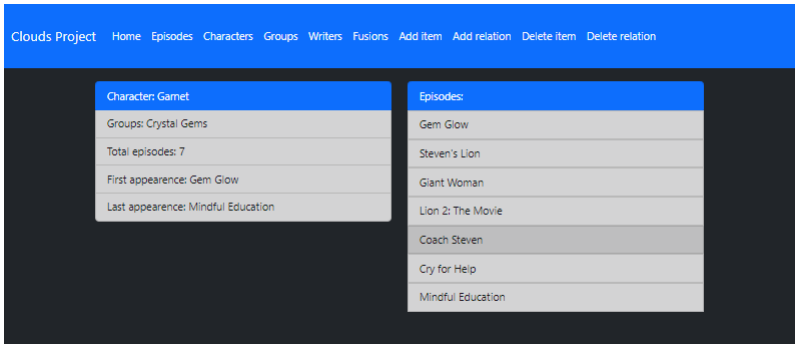


Rysunek 7.1: Strona główna

Strona jest stworzona w HTML używając stylów Bootstrapa. W menu głównym mamy link do podstron wyświetlających, dodających i usuwających dane. Istnieją też podstrony wyświetlające dane dla każdej postaci, odcinka i autora.

Podstrony wyświetlające wszystkie odcinki, postacie i autorów zawierają linki do podstron dla poszczególnych z nich, na których są również linki do np.

odcinków, które autor napisał, odcinków w których postać się pojawiła itp. Dzięki temu można sprawnie nawigować po stronie.



Rysunek 7.2: Podstrona postaci z linkami do każdego odcinka w którym się pojawiła

Mamy dostępne formularze:

- tworzące nowe postacie, odcinki, autorów
- tworzące nowe relacje między istniejącymi węzłami
- usuwające postacie, odcinki, autorów
- usuwające relacje między węzłami

8. Wnioski i przemyślenia

- Jest to moją pierwsza przygoda z bazą grafową, Flaskiem oraz Bootstrapem i było to bardzo pozytywne doświadczenie - jeśli będę miał możliwość użyję ich ponownie w przyszłości ze względu na łatwość i intuicyjność użycia.
- Dane wprowadzone w bazie są okrojone - w serialu było dużo więcej odcinków, postaci, autorów itd., lecz samplowana ilość danych powinna dobrze prezentować funckjonalność.
- Baza grafowa nie została użyta w pełni możliwości - elementy takie jak atrybuty relacji nie były użyte, większość węzłów na tylko atrybut name, co można rozbudować w przyszłości. Można by też użyć algorytmów typowych dla grafów (szukania sąsiadów z czymś wspólnym, przeszukiwania, ścieżki itp.).
- Niektóre funkcje w DatabaseApp używają kilku zapytań zwracających wyniki zamiast jednego zwracającego wynik taki jak wszystkie razem - zrobione zostało tak dla prostoty, jednak przy poznaniu lepiej działania Cyphera można to poprawić.
- Formularze tworzenia i usuwania relacji mają w opcjach selecta wszystkie dostępne wartości w tego typu węzłach. Można by to poprawić by dla dodawania nie wyświetlały się opcje, gdzie relacja już istnieje i dla usuwania wyświetlać tylko istniejące. Zrealizowane teraz rozwiązanie jest prostsze i działa poprawnie.
- Dla uproszczenia projektu nie każdy element z serialu jest odwzorowany 1:1 - np. fuzje więcej niż 2 postaci. Można je realizować za pomocą innych fuzji (nie jest to idealne). Grupy zostały ograniczone do 3 - można wymyślić ich dużo więcej.