Organizacja oraz prowadzenie gier fabularnych zdalnie z pomocą autorskiego bota na serwerze Discord

(Organizing and management of role-playing games remotly using custom made bot on Discord server)

Mateusz Zając

Praca inżynierska

Promotor: dr Marcin Młotkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

3 czerwca 2021

Streszczenie

Streszczenie po polsku - zostanie zamieszczone po napisaniu całości

---

Abstract in english - It's going to be written after the whole paper is finished.

Spis treści

Wstęp

Społeczności nie zawsze są w stanie spotkać się w świecie rzeczywistym, co stawia je przed wyborem sposobu zdalnej konwersacji oraz wymiany informacji. Podczas gdy proste spotkania można szybko i łatwo zorganizować na dowolnym komunikatorze, tak rozgrywanie `Sesji RPG`[12] na odległość nie jest już do końca trywialne. Szybko okazuje się, że prosty komunikator głosowy łączący uczestników bardzo ogranicza możliwości oraz jest w stanie istotnie obniżyć immersję świata przedstawionego względem rzeczywistych rozgrywek. Brakuje także programowych mechanizmów, które wspomagałyby podobne wydarzenia i wyręczały uczestników w powtarzalnych i żmudnych operacjach.

Istotą niniejszej pracy jest wytworzenie mechanizmów, który uproszczą zarządzanie użytkownikami podczas wydarzeń organizowanych zdalnie (w szczególności `Sesji RPG`). Dzięki użyciu biblioteki `discord.py` kod używa bardzo czytelnych abstracji i ma przejrzystą strukturę. Biblioteka `SQLAlchemy` służy odpowiedniemu składowaniu danych aplikacji i umożliwia działanie projektu na wielu serwerach jednocześnie. Interfejs programu jest realizowany przy pomocy komend na kanałach tekstowych serwera Discord. Celem wytworzonego rozwiązania jest znaczne ułatwienie przeprowadzania rozgrywek zdalnie, może być również użyty z powodzeniem w różnego rodzaju spotkaniach. W toku trwania prac nad projektem wytworzono sprawnie działający system, którego użycie nie powinno sprawiać problemów po krótkim zapoznaniu. W celu udostępnienia rozwiązania dużo szerszej grupie odbiorców wymagane są prace przy optymalizacji działania części funkcji.

[Tutaj powinien być dalszy opis pracy (ile jest rozdziałów, jaka jest ogólna struktura itd., być może w zmienionej kolejności względem powyższego tekstu)]

Rozdział 1.

Rys teoretyczny

1.1. Sesja RPG

Podstawą niniejszej pracy dyplomowej jest koncepcja `Sesji RPG`. Z języka angielskiego `Role-Playing Game`, czyli gra z odgrywaniem roli[1]. To wydarzenie, w którym uczestniczą gracze odgrywający postaci w świecie gry, Mistrz Gry (nazywany potocznie DM, GM lub MG[2]) będący narratorem historii i w szczególnych przypadkach obserwatorzy. Historia najczęściej jest prowadzona według wcześniej wybranego przez Mistrza Gry scenariusza, w określonym systemie RPG[3]. Odgrywanie postaci można porównać z powodzeniem do odgrywania roli przez aktora na scenie. To od uczestników w dużej mierze zależy jakość i sposób prowadzenia historii, dlatego bardzo istotnym jest, aby gracze „wczuli się" w swoją postać. Głównym ograniczeniem jest tylko wyobraźnia i kreatywność zarówno uczestników, jak i Mistrza gry. Zapisany skrypt, system RPG oraz realia mają tylko naprowadzać i stanowić wskazówkę w prowadzonej przygodzie. Sposób gry jest dowolny i nie ma narzuconych z góry scen do odegrania , co często powoduje wychodzenie poza przewidziane ramy scenariusza i wymusza na prowadzącym improwizację dalszych wydarzeń. Rozgrywka dąży do zrealizowania zaplanowanego na początku przygody celu, jednak to od uczestników zależy droga, którą podążą i sposób realizacji tego celu.

Przykładem systemu RPG może być ten oparty o twórczość pisarza H.P. Lovecrafta - `Call of Cthulhu`. Czas i miejsce akcji to zwykle lata 20. XX wieku w Ameryce, niedługo po wprowadzeniu prohibicji. Gracze wcielają się w role badaczy, zgłębiających tajemnice świata Wielkich Przedwiecznych. W trakcie swoich przygód rozwiązują sprawy paranormalne oraz nadnaturalne i często tylko od ich działań zależą dalsze wydarzenia oraz losy postaci[4].

[1] Także `Gra fabularna`, `Gra wyobraźni` lub `Gra narracyjna`

[2] ang. `Dungeon Master/Game Master` lub w polskim przekładzie `Mistrz Gry`

[3] System RPG określa zasady rozgrywki, atrybuty postaci i generalny sposób rozgrywania historii.

[4] Takie sesje dobrze oddają nagrania w serwisie YouTube, np. na kanale `Baniak Baniaka`[1]

1.2. Komunikator Discord

W celu lepszego zrozumienia sposobu działania aplikacji oraz problemów przez nią rozwiązywanych należy przyjrzeć się lepiej komunikatorowi `Discord`. Jest to jedno z najbardziej popularnych rozwiązań komunikacji przez internet, łączące w sobie zalety zarówno `Skype`[9] jak i `TeamSpeak`[3], poprawiając wady tych rozwiązań. Umożliwia porozumiewanie się za pomocą tekstu, głosu, lecz także wideo. Chętnie wybierany przez użytkowników ze względu na dostępność użycia, nieduże użycie pamięci oraz oferowane możliwości. Dowolny użytkownik może stworzyć własny serwer o rozbudowanej strukturze oraz systemie rang użytkowników wraz z uprawnieniami. Komunikator oferuje szereg rozwiązań dostępnych także w `MS Teams`[8] (jak np. reakcje na wiadomości użytkowników czy wklejanie grafik ze schowka). Administrator może ponadto dodać do serwera boty, które znacząco rozszerzają możliwości oraz scenariusze użytkowania. Dzięki wykorzystaniu prostych funkcjonalności wbudowanych w komunikator możemy stworzyć wiele nowych narzędzi dla użytkowników.

1.3. Biblioteka discord.py

Biblioteka `discord.py` to w rzeczywistości `API Wrapper` dla Discorda napisany w języku Python. Udostępnia szereg interfejsów dla programisty, dzięki czemu główny nacisk projektu jest położony na tworzeniu funkcji serwerowych i komend, zamiast na niskopoziomowym zarządzaniu sprzętem oraz połączeniami z serwerem. Biblioteka jest łatwa w użyciu, napisana w sposób asynchroniczny oraz posiada bogatą dokumentację[11] wraz z przykładami użycia metod[10]. Używana jest do tworzenia tzw. `botów`, automatyzujących dzięki jej użyciu funkcje udostępnione przez `Discorda`.

Bot działając na serwerze obserwuje zdarzenia, które mają na nim miejsce oraz swoją własną skrzynkę odbiorczą wiadomości. Jeśli wiadomość zaczyna się od określonego prefiksu lub na serwerze wystąpiło określone zdarzenie, program wykonuje kod odpowiedzialny za przetworzenie takiego zdarzenia lub komendy. Każda instancja bota biblioteki discord.py posiada własną pętlę zdarzeń (event loop). Jest ona ściśle związana z pętlą zdarzeń biblioteki `asyncio` języka Python. To swojego rodzaju lista zadań do wykonania przez aplikację, po której program przełącza się, wykonując kod poszczególnych zadań. W momencie wywołania przez użytkownika komendy, tworzone jest nowe zadanie w pętli zdarzeń bota. W odpowiednim czasie zostanie ono przetworzone, a wynik zwrócony.

1.4. Programowanie asynchroniczne

Większość prostych aplikacji, które są napisane w celu wykonania konkretnego zadania lub zadań mogą być z powodzeniem napisane w sposób synchroniczny. Kod wykonuje się linijka po linijce, od początku do końca, krok po kroku. Problem może wystąpić, gdy jakaś część programu korzysta z czasochłonnych operacji, które znacznie opóźniają wykonanie innych części kodu. Przykładem mogą być tutaj operacje wejścia/wyjścia czy długie wyliczenia w algorytmach, z których korzysta program. W podejściu synchronicznym blokujemy działanie do czasu zakończenia obliczeń, potem możemy wznowić dalsze wykonanie. Pierwszym pomysłem na zaradzenie sobie z takimi problemami jest programowanie wielowątkowe. Operacje czasochłonne zlecamy oddzielnym wątkom, które system operacyjny budzi co jakiś czas, przerywając wykonanie aktualnego programu. Zapanowanie jednak nad takim kodem jest stosunkowo trudne, nie mamy także kontroli nad tym w którym miejscu kod zostanie przerwany.

Jeszcze innym pomysłem jest programowanie asynchroniczne. To programista decyduje kiedy dane zadanie może zostać zatrzymane (w Pythonie służy do tego słowo kluczowe `await`), a na jego miejsce może wejść inne. Zyskujemy wtedy dużą responsywność aplikacji, prostą koncepcję działania oraz dużo mniej potencjalnych błędów. Należy pamiętać, aby w metodach asynchronicznych nie używać `blokujących` funkcji (zatrzymujących program do czasu wykonania zadania), ponieważ odbiera to kontrolę pętli zdarzeń (kod staje się synchroniczny).

1.5. Biblioteka SQLAlchemy

Biblioteka programistyczna `SQLAlchemy` służy do pracy z bazami danych typu SQL. Wspiera m.in. SQLite, MySQL, Microsoft SQL Server. Głównymi zaletami jest spamiętywanie wyników zapytań (wewnętrzny `cache`) oraz śledzenie stanu utworzonych (lub pobranych do pamięci podręcznej) obiektów zmapowanych na tabele bazy danych. Dzięki niej skupiamy się bardziej na obiektach określonych klas, niż wierszach tabel bazy danych. SQLAlchemy chroni także program przed atakami typu `SQLInjection`, ze względu na stosowanie mechanizmu `Escape Characters`. W większości przypadków korzystanie z takiej abstrakcji jest wystarczające do poprawnego i wydajnego działania. Możemy również pominąć warstwę ORM i pisać „surowe" zapytania bezpośrednio do bazy danych. Wtedy niestety nie korzystamy z niektórych zalet biblioteki (np. spamiętywanie obiektów), jednak czasami nie są one potrzebne. `SQL Alchemy` sam tworzy model bazy na podstawie wskazanych modeli oraz połączeń pomiędzy nimi. Nie musimy (a nawet nie powinniśmy) tworzyć modelu sami, ponieważ może to skutkować błędami w działaniu biblioteki. Do testów aplikacji możemy używać bazy danych zapisywanej lokalnie w pliku na dysku (`sqlite`[4]). Dokumentacja i opis kodu[2] bardzo pomagają w implementacji pożądanych funk-

cji. W dokumentach oprócz opisu logiki poszczególnych obiektów i funkcjonalności zostały zamieszczone również stosowne do omawianego zagadnienia przykłady. Podążając za tymi wskazówkami można nauczyć się korzystania z biblioteki od podstaw i zaimplementować ją bez większych przeszkód w danym projekcie.

Rozdział 2.

Instalacja projektu

2.1. Przygotowanie aplikacji

Tworzymy aplikację w bazie Discorda przy użyciu załączonego poradnika [5]. Ponadto, do poprawnego działania niektórych funkcji należy włączyć „Privileged Gateway Intents". W tym celu należy przejść do zakładki „Bot" i utworzyć nowego bota. Na nowo wygenerowanej stronie wystarczy włączyć „Presence Intent" oraz „Server Members Intent", a następnie zapisać zmiany. Tak przygotowaną aplikację możemy dodać do własnego serwera Discord[6][5]. Po wykonaniu tych działań bot powinien pojawić się na liście użytkowników, ze statusem `Offline`.

2.2. Środowisko uruchomieniowe

Wszystkie zależności niezbędne do uruchomienia aplikacji znajdują się w folderze venv (łącznie z interpreterem języka python). Otwierając projekt przy pomocy `JetBrains PyCharm`[7] możemy skorzystać z wcześniej skonfigurowanego środowiska wirtualnego, bez potrzeby instalacji dodatkowych bibliotek. Jeśli natomiast uruchamiamy z linii poleceń, za pomocą czystego interpretera języka Python, potrzebujemy zależności z Tabeli 2.1.

Przed uruchomieniem kodu należy również dostosować ustawienia w pliku `config.json`, w folderze projektu. W miejscu `botToken` należy wpisać swój token z poradnika[5]. Pole `ownerId` powinno zostać uzupełnione o ID użytkownika, który ma być administratorem bota. W celu włączenia wyświetlania ID użytkownika należy:

1. Wejść w ustawienia konta w aplikacji Discord
2. Przejść do sekcji `Zaawansowane`
3. Włączyć `Tryb developera` oraz zapisać zmiany

Nazwa	**Wersja**	**Nazwa**	**Wersja**
PyNaCl	1.4.0	idna	3.1
SQLAlchemy	1.4.15	multidict	5.1.0
aiohttp	3.7.4.post0	mutagen	1.45.1
async-timeout	3.0.1	pip	21.1.2
attrs	21.2.0	pycparser	2.20
cffi	1.14.5	setuptools	57.0.0
chardet	4.0.0	six	1.16.0
discord	1.0.1	typing-extensions	3.10.0.0
discord.py[voice]	1.7.2	yarl	1.6.3
ffmpeg	1.4	youtube-dl	2021.4.7
greenlet	1.1.0		

Tablica 2.1: Zależności niezbędne do uruchomienia projektu

Teraz wystarczy nacisnąć prawym przyciskiem na swój profil na liście użytkowników serwera i wybrać `Kopiuj ID`. Tak uzupełniony plik konfiguracyjny można zapisać i zamknąć, a następnie włączyć aplikację. Jeśli projekt załaduje się poprawnie, na konsoli wyświetli się ciąg znaków: `Application has started properly`. W celu wyświetlenia panelu pomocy, wystarczy wpisać na czacie komendę `‘help`.

Rozdział 3.

Implementacja

Aplikacja jest podzielona na kilka różnych modułów, które zawierają w sobie nawiązujące do nich funkcjonalności. Dzięki użyciu mechanizmu `cogs` biblioteki `discord.py` możemy tymi modułami dowolnie zarządzać w trakcie wykonania programu (podłączać oraz odłączać je od bota podczas działania). Wszystkie moduły są składowane w folderze `cogs`. Każdą z dostępnych komend poprzedzić należy znakiem „`” (klawisz tyldy) jeśli prefix nie został zmieniony w pliku `config.cfg`.

3.1. Struktura modułu

Każdy oddzielny moduł to nowa klasa, dziedzicząca po klasie `discord.ext.commands.Cog`. Aby zdefiniować nowy moduł (np. o nazwie `Example`) wystarczy utworzyć klasę taką jak na Listingu 3..1.

```
#exampleCog.py
from discord.ext import commands

class Example(commands.Cog):
    def __init__(self, client):
        self.client = client

def setup(client):
    client.add_cog(Example(client))
```

Listing 3..1: Przykładowy kod modułu `Example`

Tak utworzony moduł wystarczy załadować za pomocą metody `load_extension` obiektu `Bot`[1] lub komendą `‘load <nameOfModule>` (użytkownik musi być ownerem bota).

[1] Przykład można znaleźć w pliku `main.py`, gdzie w pętli są ładowane wszystkie moduły projektu z katalogu `cogs`

3.2. Struktura komendy

Definiowanie komend bota nie różni się znacznie od definiowania zwykłych metod asynchronicznych. Należy jedynie dodać parametr klasy `discord.ext.commands.context` zaraz po parametrze `self` oraz dodać dekorator `@commands.command()` nad nagłówkiem funkcji. Przykładową komendę wypisującą na czacie wiadomość „Hello world!" zamieszczono w Listingu 3..2

```
# exampleCommand.py
@commands.command()
async def helloworld(self, ctx: commands.context):
    await ctx.send("Hello, world!")
```

Listing 3..2: Przykładowa komenda wyświetlająca Hello World!

Możemy także użyć innych dekoratorów, np. aby umożliwić wykonanie komendy tylko określonym użytkownikom[2] czy zablokować możliwość zbyt częstego wykonania danej akcji przez pojedynczego użytkownika[3].

3.3. Baza danych

Dzięki `SQL Alchemy` aplikacja korzysta z bazy danych działającej na silniku `SQLite` przy użyciu abstrakcji ORM (object-relational mapping). W celach implementacyjnych powstały klasy, modelujące poszczególne tabele oraz relacje, które pomiędzy nimi zachodzą. Każdy z tych modeli jest zawarty w katalogu `Models` i ma prefiks `Model`.

Mimo, iż korzystamy z abstrakcji obiektów i klas, nadal musimy pamiętać o relacjach, które zachodzą między obiektami. Przyjrzyjmy się relacji wiele-do-wielu tabel `UserNameBackup` oraz `ServerSession`, której tabelą pośrednią jest `UserName`. Taką relację możemy modelować łącząc dwie relacje (`one-to-many` oraz `many-to-one`) (przykład na Listingu 3..3).

```
#serverSession.py
from sqlalchemy import Column, Integer, String, Boolean
from sqlalchemy.orm import relationship
from DBManager import dbmanager

class ServerSessionModel(dbmanager.Base):
    __tablename__ = "ServerSession"

    ID_ServerSession = Column(Integer,
                              primary_key=True,
```

[2]Taki dekorator jest użyty np. nad funkcją `load` w pliku main.py

[3]Przykład użycia takich dekoratorów zamieszczony w module `Session`

```
                                    autoincrement=True)
    ID_Server = Column(Integer)
    SessionShort = Column(String)
    ID_GM = Column(Integer)
    SoundBoardSwitch = Column(Boolean)

    UsersNames = relationship("UserNameModel",
                                    cascade="all,delete,delete-orphan")
```

Listing 3..3: Wycinek implementacji serverSession.py (tabela po stronie `one`)

Jak widzimy powyżej, modele dziedziczą po `dbmanager.Base` (musi to być ten sam obiekt, który „budujemy" w pliku `main.py`[4]). Większość pól klasy to obiekty `Column`, z odpowiednim typem podanym w argumencie. Po stronie relacji `one` należy utworzyć atrybut (obiekt klasy `relationship`), będący listą obiektów w relacji. Po stronie `many` dodajemy natomiast pole, będące obiektem klasy `Column`, jednak posiadające w argumencie (oprócz typu) obiekt klasy `ForeignKey`. Biblioteka zbuduje cały model bazy, jeśli ten nie istnieje.

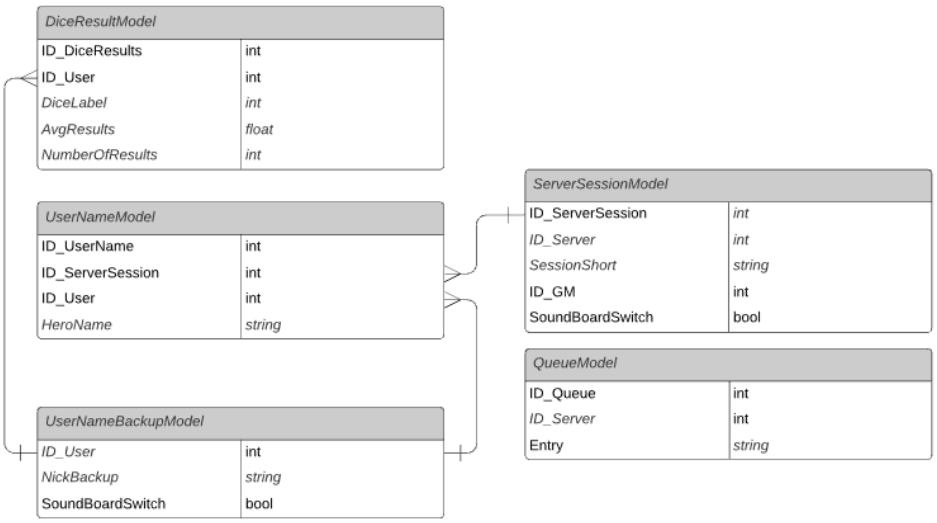
```
#username.py
from sqlalchemy import Column, Integer, String, ForeignKey
from DBManager import dbmanager

class UserNameModel(dbmanager.Base):
    __tablename__ = "UserName"

    ID_UserName =
            Column(Integer, primary_key=True)
    ID_ServerSession =
            Column(Integer,
                    ForeignKey('ServerSession.ID_ServerSession'))
    HeroName = Column(String)
    ID_User =
            Column(Integer,
                    ForeignKey('UserNameBackup.ID_User'))
```

Listing 3..4: Wycinek implementacji username.py (tabela po stronie `many`)

[4] Budowanie odbywa się w linijce `DBManager.dbmanager.Base.metadata.create_all(bind=DBManager.dbmanager.engine)`



Rysunek 3.1: Model bazy danych całej aplikacji

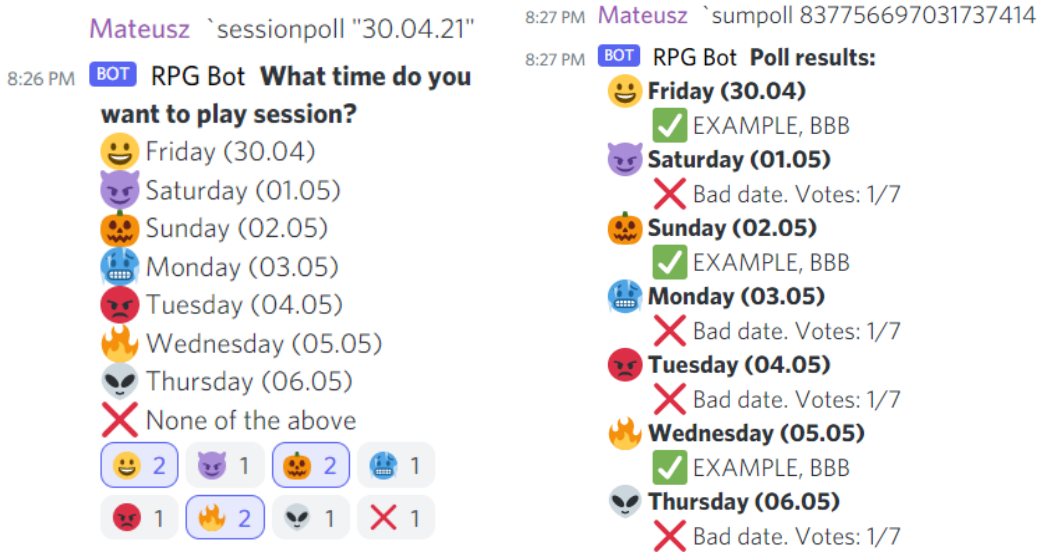
3.4. Przykłady mechanizmów aplikacji

3.4.1. Głosowanie na termin

Jednym z większych problemów w organizacji zarówno sesji RPG jak i innych wydarzeń w większej grupie osób jest znalezienie terminu. Jeśli organizujemy spotkanie w małej liczbie osób lub nie spotykamy się za często, ręczne sprawdzenie jest rozsądnym podejściem. Sytuacja staje się tym bardziej skomplikowana, im więcej mamy typów wydarzeń i osób do sprawdzenia. Pomysłem, który jest prosty w swoich założeniach jest utworzenie ankiety, w której uczestnicy będą głosować przy pomocy emotikon Discorda. Poźniej aplikacja użyje wyników, aby podsumować ankietę i sprawdzić kiedy można zorganizować dane wydarzenie. Przykład użycia mechanizmu został zamieszczony na Rysunku 3.2.

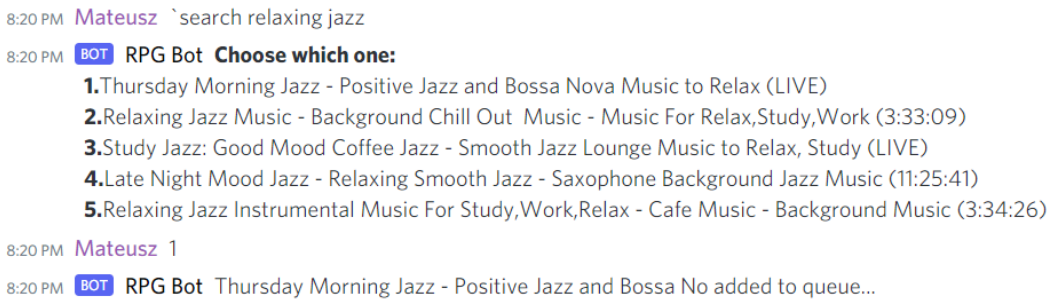
3.4.2. Strumieniowanie muzyki z serwisu YouTube

Niekiedy podczas spotkania organizator chciałby odtworzyć dźwięk lub utwór muzyczny, który będzie słyszany przez wszystkich uczestników na kanale. Istnieją rozwiązania synchronizujące odtwarzacze użytkowników, jednak wymagają użycia dodatkowych mechanizmów poza Discordem. Opisywana aplikacja jest w stanie odtwarzać dźwięk z serwisu YouTube na bieżąco, ale także umożliwia wygodne wyszukiwanie utworów poprzez czat. Przykład zastosowania mechanizmu zamieszczono na Rysunku 3.3



(a) Ankieta oraz osiem reakcji (b) Podsumowanie ankiety

Rysunek 3.2: Przykładowa ankieta



Rysunek 3.3: Wyszukiwanie utworów oraz wybór

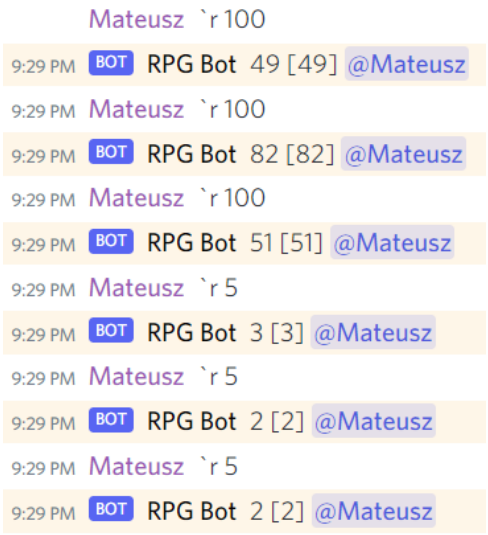
3.4.3. Podsumowanie rzutów kostką po wydarzeniu

Aplikacja zapisuje wyniki rzutów kostką użytkowników, którzy należą do co najmniej jednej sesji. Sesja to tylko zbiór użytkowników wraz z podstawowymi informacjami o nich. System nie śledzi tego w jakiej sesji padły jakie rzuty (nie ma to sensu, ponieważ dany użytkownik nie gra więcej niż jednej sesji w danej jednostce czasu). Podsumowując rzuty korzystamy z tej struktury, aby wiedzieć jakie rzuty uwzględnić w podsumowaniu. Aby usunąć dane o rzutach, należy albo wywołać odpowiednią komendę (`‘resetrolls <nameOfSession>`), lub skorzystać z komendy przywracania pseudonimów graczy po sesji (`‘changenicks <nameOfSession: str> True`), która także usuwa informacje o rzutach.

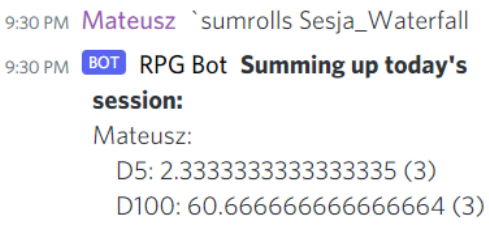
Zdefiniujmy sesję o nazwie `Sesja_Waterfall`, która zawiera w sobie gracza `Mateusz` (`‘makesession Sesja_Waterfall 0 @Mateusz`)[5]. Po rzuceniu kilka razy

[5]Konstrukcja komendy tworzenia sesji to: `‘makesession <nameOfSession: str> <soundboard: bool> <*members: discord.Member>`, gdzie `nameOfSession` to nazwa sesji, `soundboard` to przełącznik reakcji dźwiękowych bota na rzuty, `members` to lista uczestników sesji

kostką d100 oraz d5 (oznaczenie po literze `d` to liczba ścian kostki)[6] możemy zobaczyć średnią tych rzutów (Rysunek 3.4).



(a) Kilka rzutów kostką d100 oraz d5



(b) Średni wynik po rzutach kostką

Rysunek 3.4: Przykładowa średnia rzutów kostką

Aplikacja oblicza nową średnią na podstawie wzoru i zapisuje ją z powrotem do bazy danych. Dzięki temu zużywamy minimalną ilość pamięci do spamiętywania wyników, a ilość możliwych rzutów zanim licznik się „przekręci" to $2^{63}-1$ (maksymalna wartość pola `INTEGER` w `SQLite`).

$$\text{newAverage} = \frac{\text{oldAverage} * \text{oldNumberOfResults} + \text{sumOfCurrentRolls}}{\text{oldNumberOfResults} + \text{numberOfCurrentRolls}}$$

[6] W sesjach rozgrywanych na żywo 100-ścienna kostka d100 jest zastępowana przez dwie kostki. Jedna z cyfrą dziesiątek, druga z cyfrą jedności.

Rozdział 4.

Inne rozwiązania

Z opisanych tutaj rozwiązań korzysta wiele użytkowników na całym świecie. Są ogólnodostępne, a ich bezpłatna wersja umożliwia podstawową realizację określonych funkcji.

4.1. YAGPDB.xyz - bot

Bot YAGPDB.xyz posiada funkcję rzutu kostką N-ścienną K razy dzięki komendzie `-roll`. Można również zdefiniować ile rzutów ma zostać wykonanych przez bota (-roll KdN). Udostępnia również komendę do tworzenia prostych ankiet. Nie zapisuje niestety średnich wartości rzutów wykonanych przez każdego użytkownika oraz nie udostępnia dodatkowych mechanizmów typowo pod sesję RPG. Ankiety są bardzo prostym mechanizmem, nie ma możliwości sprawdzenia pasującego terminu dla określonej sesji. `https://yagpdb.xyz/`

4.2. Rhytm - bot

Rhytm to najbardziej popularny bot muzyczny, obsługujący zarówno muzykę z URL YouTube, jak i wyszukiwanie utworów poprzez czat Discorda. Prowadzi kolejkę odtwarzania oraz wszelkie mechanizmy niezbędne dla odtwarzacza muzycznego. Służy tylko odtwarzaniu muzyki, nie posiada innych mechanizmów związanych z sesjami RPG. Opcjonalnie może służyć jako drugi bot do odtwarzania dźwięków (boty mogą odtwarzać na raz tylko jedno źródło dźwięku). `https://rythm.fm/`

4.3. Roll20 - serwis internetowy

Serwis Roll20 jest stworzony z myślą o prowadzeniu oraz rozgrywaniu sesji RPG. W podstawowej, bezpłatnej wersji umożliwia rzuty kostką, prowadzenie kart po-

staci oraz dużo więcej. Jest jednak dodatkowym serwisem (jeśli użytkownik używa Discorda do porozumiewania się), który każdy z graczy musi regularnie sprawdzać i odpowiednio ustawiać. Nie ma również możliwości zagłosowania na termin organizowanej sesji. Wiele graczy zgłaszało, że lepiej im prowadzić kartę postaci w pliku pdf na komputerze lub na kartce papieru, niż na powyższym serwisie. `https://app.roll20.net`

Rozdział 5.

Zakończenie

Bibliografia

[1] Baniak Baniaka. *Baniak Baniaka - YouTube*. URL: https://www.youtube.com/channel/UCXnI7wpHJ-x8bafp1BK3DUQ. (dostęp: 20.05.2021).

[2] Michael Bayer. *SQLAlchemy - The Database Toolkit for Python*. URL: https://www.sqlalchemy.org/. (dostęp: 20.05.2021).

[3] TeamSpeak Systems GmbH. *Home — TeamSpeak*. URL: https://www.teamspeak.com/pl/. (dostęp: 20.05.2021).

[4] D. Richard Hipp. *SQLite Home Page*. URL: https://www.sqlite.org/index.html. (dostęp: 20.05.2021).

[5] Discord Inc. *Creating a Bot Account*. URL: https://discordpy.readthedocs.io/en/stable/discord.html. (dostęp: 20.05.2021).

[6] Discord Inc. *Jak stworzyć serwer? – Discord*. URL: https://support.discord.com/hc/pl/articles/204849977-Jak-stworzy%C4%87-serwer-. (dostęp: 20.05.2021).

[7] JetBrains. *PyCharm: the Python IDE for Professional Developers by JetBrains*. URL: https://www.jetbrains.com/pycharm/. (dostęp: 20.05.2021).

[8] Microsoft. *Konferencje wideo, spotkania, połączenia — Microsoft Teams*. URL: https://www.microsoft.com/pl-pl/microsoft-teams/group-chat-software. (dostęp: 20.05.2021).

[9] Microsoft. *Skype — Twój sposób na bezpłatne rozmowy i czat*. URL: https://www.skype.com/pl/. (dostęp: 20.05.2021).

[10] Rapptz. *discord.py/examples at master · Rapptz/discord.py · GitHub*. URL: https://github.com/Rapptz/discord.py/tree/master/examples. (dostęp: 20.05.2021).

[11] Rapptz. *Welcome to discord.py*. URL: https://discordpy.readthedocs.io/en/stable/. (dostęp: 20.05.2021).

[12] Wikipedia. *Gra fabularna – Wikipedia, wolna encyklopedia*. URL: https://pl.wikipedia.org/wiki/Gra_fabularna. (dostęp: 20.05.2021).