



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ

KATEDRA INFORMATYKI STOSOWANEJ I FIZYKI KOMPUTEROWEJ

Projekt dyplomowy

Bot do planowania wydarzeń w aplikacji Discord

Discord event scheduling bot

Autor: Alan Sławomir Guzek
Kierunek studiów: Informatyka Stosowana
Opiekun pracy: dr hab. inż. Małgorzata Krawczyk

Kraków, 2024

Spis treści

1	Wstęp	3
1.1	Opis problemu	3
1.2	Cel Projektu	3
2	Użyte technologie	5
2.1	Discord.py	5
2.2	MongoDB	5
2.3	Git	6
2.4	Docker	6
3	Implementacja	6
3.1	Rejestracja bota	6
3.2	Bot i komendy	8
3.3	Interfejs Użytkownika	10
3.4	Modelowanie danych	15
3.5	Algorytm wyboru	18
3.6	Testy	20
4	Prezentacja	21
4.1	Sposób użycia	21
4.1.1	Dodawanie aplikacji do gildii	21
4.1.2	Ustalanie terminu spotkania	22
4.2	Hosting	27
5	Podsumowanie	29

1 Wstęp

1.1 Opis problemu

Kalendarze osobiste wielu osób są bardzo zapełnione przez co bardzo często znalezienie dogodnego terminu, aby umówić się w gronie kilku osób, np. na wspólną naukę czy grę graniczy z niemożliwym. Zazwyczaj jeden z uczestników podaje losowy terminy i czeka na informację zwrotną, czy wszystkim to odpowiada. Takie podejście jest nieefektywne. Bardziej odpowiednim rozwiązaniem tego problemu wydaje się zebranie informacji od każdego z potencjalnych uczestników spotkania, a następnie używając odpowiedniego algorytmu, wybranie najlepszego terminu.

Jeśli dodatkowo chcemy zorganizować takie spotkanie z osobami nie mieszkającymi w pobliżu, a więc czy to z innego miasta, czy innego kontynentu, potrzebne jest użycie np. komunikatora internetowego takiego jak Zoom, Microsoft Teams, Discord czy Google Meet.

1.2 Cel Projektu

Aby rozwiązać opisany problem zdecydowano się napisać bota discordowego [1], który umożliwi wybranie idealnego terminu grupie osób.

Discord jest platformą internetową wybieraną przez miliony [2], w której możliwe jest nie tylko połączenie się z innymi przy użyciu mikrofonu i kamery, ale także oferuje ona wygodne do personalizacji gildie, na których dzięki kanałom tekstowym łatwo można zarządzać wiadomościami. Często kanały takie dedykowane są specyficznym zagadnieniom, jak np. „pomoc”, gdzie można zadawać pytania bardziej doświadczonym osobom w gildii, „informacje”, gdzie jedynie uprawnione osoby mogą pisać wiadomości z najważniejszymi informacjami czy też „bot” dla botów discordowych.

Boty są to aplikacje, które mając uprawnienia w gildii, potrafią monitorować różnego rodzaju wydarzenia takie jak dościsie nowego członka, wysłanie wiadomości czy pojawienie się nowej osoby na kanale głosowym. Boty często służą także do zapisywania notatek, przydzielania rang dla danych osób, czasami służą do pomagania w nauce. Mogą one również usprawnić wybieranie dogodnego terminu spotkania dla osób z gildii.

Aby stworzyć system do ustalania dat spotkań przy użyciu bota należy wyszczególnić kilka ważnych warunków, które musi on spełniać:

1. każdy członek gildii powinien mieć możliwość utworzenia wydarzenia, dla którego datę chcemy wybrać,
2. każdy z zaproszonych musi odpowiedzieć jakie terminy mu odpowiadają,

3. algorytm powinien wybrać najlepszą dla grupy godzinę, uwzględniając czas trwania wydarzenia.

Oprócz komendy do tworzenia wydarzeń, ważne będą także elementy widoków, takie jak przyciski, pola wyboru, modale, a także pola tekstowe. Informacje, które należy zebrać od użytkownika wybierającego termin wydarzenia to:

- zakres dat, w jakich wydarzenie powinno się odbyć,
- długość trwania wydarzenia,
- lista osób, które biorą w nim udział,
- dodatkowe informacje, takie jak nazwa i opis wydarzenia.

Po zebraniu informacji bot powinien umożliwić każdemu z zaproszonych uczestników wybranie godzin preferowanych (godziny "ok") oraz dopuszczalnych (godziny "maybe"). Pozwala to pokazać, jaki termin jest preferowany przez każdego z użytkowników ("ok") oraz zapewnia większe prawdopodobieństwo znalezienia odpowiedniej daty dzięki zaznaczeniu wszystkich godzin, w których jest on dostępny ("maybe").

Po zebraniu danych od każdego z uczestników wydarzenia, algorytm powinien wybrać możliwy termin wydarzenia lub zakomunikować, że nie ma daty odpowiadającej wszystkim zainteresowanym.

Oprócz podstawowej funkcjonalności osoba tworząca spotkanie powinna móc przeglądać stworzone wydarzenia, a także mieć możliwość ponownego zebrania informacji i wyszukania terminu.

Aby stworzyć tak opisaną aplikację należy napisać program komunikujący się z serwerami Discorda. Będzie on reagował na przychodzące zdarzenia, a także wywołania komend. Dodatkowo przetwarzał on będzie informacje od użytkowników i przechowywał je w bazie danych. Zdecydowano się do tego użyć języka Python. Posiada on dedykowaną do tego bibliotekę *discord.py* [3], która oferuje wiele funkcjonalności związanych z tworzeniem aplikacji discordowych, takich jak:

- rozsądne ograniczenie ilości wysyłanych zapytań zmniejszające szansę na pojawienie się błędu 429,
- podejście obiektowe ułatwiające tworzenie i ponowne używanie elementów,
- optymalizację pod względem czasu przetwarzania oraz używanej pamięci.

2 Użyte technologie

Do napisania aplikacji został użyty język Python w standardzie 3.10 [4]. Wybrano go ze względu na jego szerokie zastosowanie w wielu dziedzinach, jak również łatwą obsługę asynchroniczności, która przy częstym reagowaniu na wiadomości i komendy użytkowników staje się niezbędna. Posłużył on zarówno do stworzenia bota, jak i obsługi bazy danych, a także implementacji algorytmu pozwalającego wybrać odpowiedni dzień i godzinę spotkania. Dodatkowo, moduł służący do komunikowania się z Discordem (`discord.py`) jest często używany przez deweloperów aplikacji o czym może świadczyć duża ilość gwiazdek, którą posiada repozytorium tej biblioteki [5].

2.1 Discord.py

Głównym modułem wykorzystanym podczas tworzenia aplikacji był moduł `discord.py`, który jest implementacją udostępnionego przez Discroda interfejsu do obsługi botów. Dzięki podejściu obiektowemu pozwala na dostosowywanie elementów UI poprzez dziedziczenie po klasach podstawowych, takich jak `discord.ui.Button` oraz `discord.ui.TextInput`. Dodatkowo, moduł ten korzysta z dekoratorów dla funkcji w celu tworzenia zarówno interfejsu użytkownika, jak i rejestrowania nowych komend. Funkcje te są asynchroniczne, przez co możliwe jest wykonywanie kilku poleceń w tym samym momencie (zwłaszcza, jeżeli wykonanie polecenia zajmuje aplikacji dużo czasu). Posiada on również dobrze napisaną dokumentację [6].

2.2 MongoDB

Dokumentowa baza danych, jaką jest MongoDB [7], została wybrana ze względu na łatwość w rozbudowie oraz prostotę w użyciu. Zamiast wielu encji połączonych ze sobą, jak ma to miejsce w przypadku podejścia relacyjnego, postanowiono składować wszystkie informacje o wydarzeniu w jednym dokumencie. Brak zdefiniowanego schematu w bazie danych umożliwia dynamiczne dodawanie elementów do obiektu w przystępny sposób. Dodatkowo, obiekty w bazie danych zapisane w formacie JSON w łatwy sposób są konwertowane na pythonowe słowniki.

Do komunikacji z bazą danych użyto modułu w języku Python o nazwie *pymongo* [8], ze względu na jego popularność [9]. Pozwala on na stworzenie klienta komunikującego się z bazą przy użyciu linka *url*, a następnie wykonywanie wszystkich operacji zapisu, modyfikacji oraz odczytu z bazy danych.

2.3 Git

W celu kontrolowania wersji tworzonego oprogramowania użyto popularnego narzędzia git [10]. W celu zapisywania projektu w zdalnym repozytorium zdecydowano się na skorzystanie z platformy GitHub [11].

2.4 Docker

Aby usprawnić proces tworzenia oraz wdrażania aplikacji w nowym środowisku zdecydowano się na konteneryzację przy użyciu narzędzia Docker [12]. Po wpisaniu komendy:

```
1 $ docker-compose up -d
```

Listing 1: Uruchamianie aplikacji w kontenerze Docker

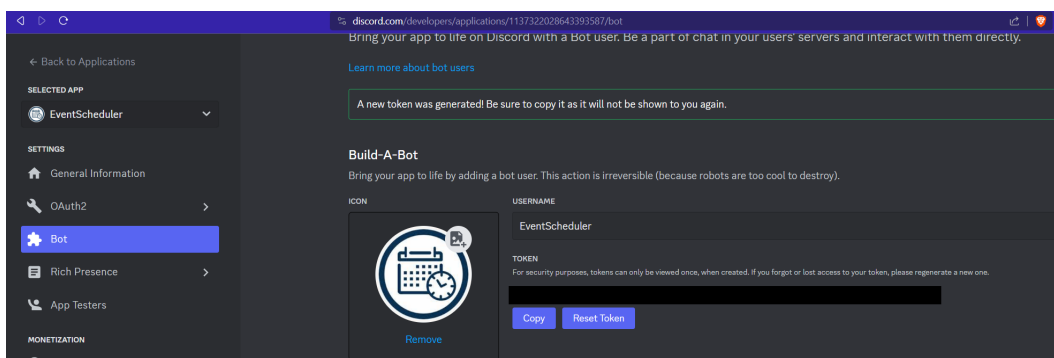
powstaną dwa kontenery, jeden dla bazy danych MongoDB i drugi zawierający aplikację napisaną w języku Python. Obraz dockerowy używa języka Python w wersji 3.10.0.

3 Implementacja

Proces tworzenia bota discordowego można podzielić na kilka etapów. Na początku rejestruje się go na platformie Discord i uzyskuje niezbędny token dostępu. Następnie przy jego użyciu, tworzy się aplikację w wybranym języku programowania, która będzie zawierała klienta komunikującego się z serwerami Discorda. Zdecydowano się użyć języka Python. Później używając stworzonego klienta rejestruje się komendy oraz to, co powinno się zdarzyć po ich wywołaniu (najczęściej odpowiedź na wiadomość użytkownika, w której załączony jest specjalny widok). Następnie przygotowuje się interfejs tak, aby użytkownik mógł osiągnąć zamierzony cel, jakim w tym przypadku było wybranie odpowiedniego terminu dla grupy osób.

3.1 Rejestracja bota

Na początku stworzono aplikację w platformie Discord. W tym celu zalogowano się i przekierowano pod adres <https://discord.com/developers/applications>. Tam, po naciśnięciu przycisku *New Application* oraz wpisaniu wymaganych informacji, stworzono nową aplikację. W celu rozpoczęcia pracy z nowo utworzonym botem wygenerowano token (Rys. 1) umożliwiający połączenie się z nim z poziomu programu napisanego w języku Python.



Rysunek 1: Generowanie tokena dla aplikacji

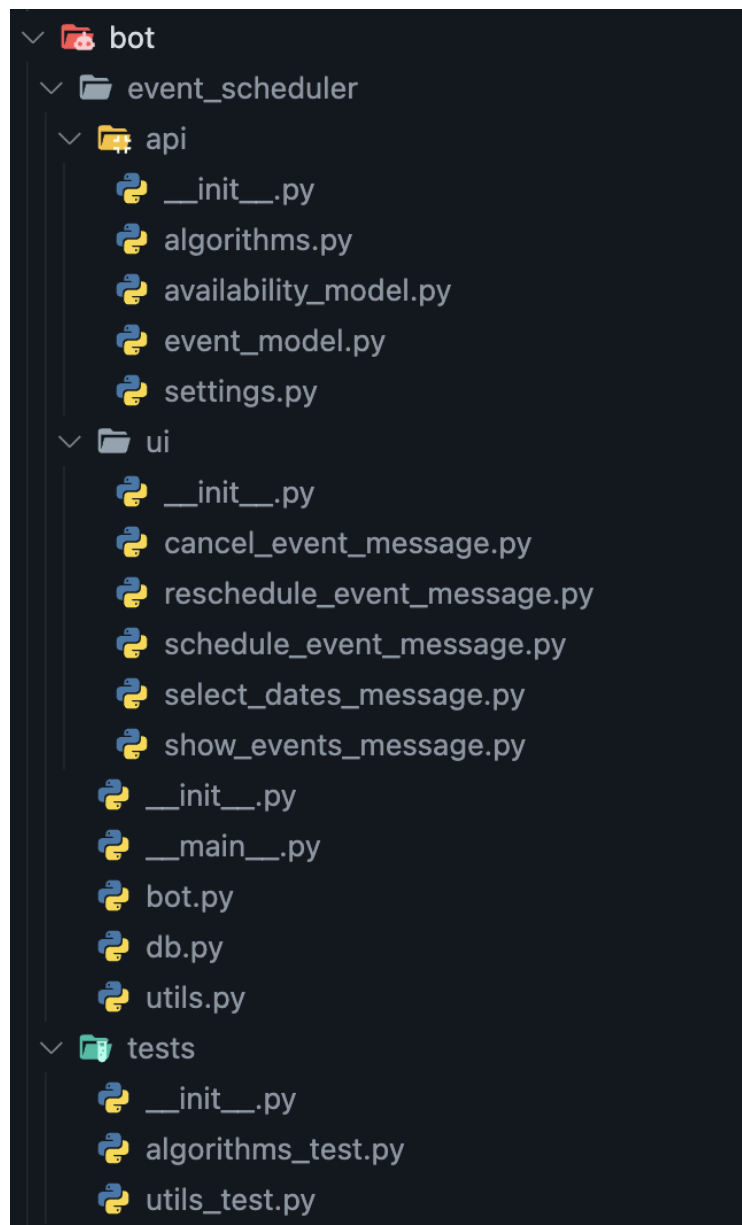
Do przechowywania danych wrażliwych, takich jak link do bazy danych czy token dla aplikacji discordowej, użyto pliku `.env` oraz dostarczono `.env.sample`, który pokazuje jak powinien wyglądać prawidłowy plik ze zmiennymi środowiskowymi.

Folder aplikacji bot (Rys. 2) składał się z pakietów aplikacji `event_scheduler` oraz testów jednostkowych `tests`. Pakiety pythonowe to tak naprawdę foldery zawierające w sobie plik `__init__.py`, natomiast moduły to pliki z rozszerzeniem `.py`. Folder aplikacji zawierał pakiety `api` oraz `ui`, a także moduły `bot.py`, `db.py`, `utils.py`, które odpowiadały odpowiednio za stworzenie klienta bota oraz jego komend, stworzenie klienta do bazy danych, dodatkowe przydatne funkcjonalności.

Pakiet `api` zawierał w sobie pliki modeli (`availability_model.py`, `event_model.py`), których zadaniem było przetrzymywanie danych dotyczących wydarzeń oraz ich zapis i odczyt z bazy, plik `settings.py` posiadający funkcjonalność do zapisywania ustawień bota dla danej gildii oraz `algorithms.py` zawierający algorytm wyboru odpowiedniej daty.

Pakiet `ui` gromadził wszystkie elementy interfejsu użytkownika podzielone na komendy, dla których są one przeznaczone (`[nazwa_komendy]_message.py`).

Pakiet testów `tests` zawierał moduły `algorithms_test.py` i `utils_test.py` sprawdzające odpowiednio funkcje używane w algorytmie wyboru odpowiedniego terminu, jak i sam algorytm oraz przydatne funkcje służące na przykład do konwersji dat i godzin.



Rysunek 2: Struktura plików aplikacji

3.2 Bot i komendy

Istnieją dwa sposoby na komunikowanie się z aplikacją jako użytkownik Discorda: komendy z prefixem oraz komendy poprzedzone ukośnikiem, czyli *"slash commands"* [13]. Drugi sposób jest nowszy i oferuje bardziej przyjemny dla użytkownika interfejs, dlatego zdecydowano się właśnie na niego. Warto zaznaczyć, że bot może wspierać oba sposoby tworzenia komend.

Tworzenie aplikacji zaczęto od utworzenia pliku `bot.py`. Odpowiada on za stworzenie

obiektu komunikującego się z serwerem przy użyciu tokena wygenerowanego wcześniej. W tym celu użyto klasy `discord.ext.commands.Bot`, która dziedziczy po bardziej ogólnej klasie `discord.Client`. Praca z klasą bazową `Client` wymagałaby implementowania wielu elementów, które są już dostępne w klasie `Bot` [14].

W celu zapisywania informacji oraz błędów aplikacji użyto modułu "logging", dzięki któremu wiadomości aplikacji były dostarczane do pliku `bot.log` zamiast do konsoli.

```
1 bot = commands.Bot(
2     command_prefix='!',
3     description="Set of commands to pick perfect date for your event",
4     intents=discord.Intents.all()
5 )
6 handler = logging.FileHandler(
7     filename='bot.log',
8     encoding='utf-8',
9     mode='a'
10 )
11
12 def run_bot() -> None:
13     """Runs the bot"""
14     bot.run(
15         os.getenv('TOKEN'),
16         log_handler=handler,
17         log_level=logging.INFO
18     )
```

Listing 2: Utworzenie obiektu klasy Bot

Po utworzeniu obiektu (List. 2) i przypisaniu go do zmiennej `bot` tworzenie komend odbywało się poprzez stworzenie funkcji asynchronicznych z użyciem dekoratora `bot.tree.command` (List. 3) dla tych poprzedzonych ukośnikiem oraz `bot.command` dla komend z prefixem.

```
1 @bot.tree.command(name='schedule-event')
2 async def add_event(interaction: discord.Interaction) -> None:
3     model = EventModel(creator_id=interaction.user.id,
4                         guild_id=interaction.guild.id)
5     view = ScheduleEventView(bot=bot, model=model)
6     await interaction.response.send_message(view=view, embed=view.embed)
```

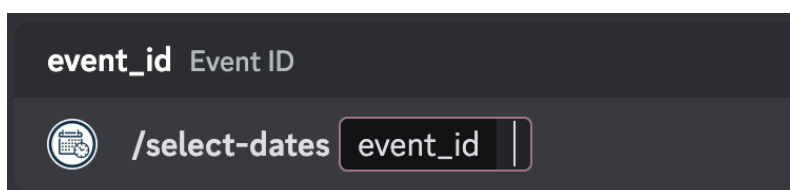
Listing 3: Komenda `/schedule-event`

Aplikacja posiada 5 "slash commands", odpowiedzialnych za tworzenie i zarządzanie wydarzeniami:

- `/add-event` - tworzy widok odpowiedzialny za wypełnienie informacji o wydarzeniu

oraz jego stworzenie,

- `/show-events` - pokazuje listę 5 najnowszych wydarzeń,
- `/reschedule-event` - inicjuje proces wyboru daty i godziny wybranego wydarzenia,
- `/cancel-event` - unieważnia wybrane wydarzenie,
- `/select-dates` - wysyła prywatną wiadomość z widokiem odpowiadającym za wybór dogodnych terminów. (Rys. 3)



Rysunek 3: Przykładowa komenda poprzedzona ukośnikiem

Dodatkowo bot obsługuje jedną komendę z prefixem, która po napisaniu `!set-channel` ustawia kanał, w którym jej użyto jako miejsce do komunikacji bota z użytkownikami w celu przekazania informacji takich jak wybranie terminu spotkania.

3.3 Interfejs Użytkownika

Podczas tworzenia interfejsu użytkownika aplikacji korzystano z modułu `discord.ui` [15] oraz klasy `discord.Embed`. Klasa `View` odpowiadała za utworzenie wiadomości zawierającej różne komponenty UI takie jak przyciski i pola wyboru. Klasa `Embed` reprezentowała obiekt discordowy w wiadomości, który pozwalał na zebranie informacji w elegancki sposób (co zostało pokazane na Rys. 14 w rozdziale 4 dotyczącym prezentacji działania aplikacji).

Szczegółową implementację można omówić na przykładzie komendy `/schedule-event`, gdyż posiada ona najwięcej elementów UI wykorzystywanych w aplikacji. Wszystkie pliki dotyczące tworzenia interfejsu użytkownika znajdowały się w folderze `ui`.

Komenda `/schedule-event` tworzy obiekt klasy `ScheduleEventView` (List. 4), a następnie reaguje na wiadomość użytkownika dostarczając widok oraz `embed`¹ (List. 3). Dodatkowo, tworzony jest obiekt klasy `EventManager` odpowiedzialny za składowanie informacji oraz komunikację z bazą danych, który zostanie omówiony w późniejszych rozdziałach.

¹ Jest to angielska nazwa elementu UI Discorda używana w tekście ze względu na brak polskiego odpowiednika

```

1 class ScheduleEventView(ui.View):
2     def __init__(self, model: EventModel, bot: discord.Client = None):
3         super().__init__()
4         self.bot=bot
5         self.embed=ScheduleEventEmbed(model=model)
6         self.embed=self.embed.reload_embed()
7         self.add_participant_button=AddParticipantButton(self.embed)
8         self.add_item(self.add_participant_button)
9         self.remove_participant_button=RemoveParticipantButton(self.embed)
10        self.add_item(self.remove_participant_button)
11        self.add_description_button=AddDescriptionButton(self.embed)
12        self.add_item(self.add_description_button)
13        self.save_button=SaveButton(self.embed)
14        self.add_item(self.save_button)
15        self.add_item(CancelButton(self.embed))

```

Listing 4: Klasa ScheduleEventView

Do stworzenia widoku należy utworzyć klasę pochodną do klasy `discord.ui.View`. Sam widok nie potrzebuje przeładowania żadnych metod oprócz funkcji `__init__`. W niej inicjalizujemy klasę nadrzędną a następnie tworzymy obiekty elementów widoku i rejestrujemy je przy użyciu metody `add_item`. Jeżeli elementy będą na siebie reagować, należy zapisać obiekty elementów do atrybutów a następnie zarejestrować je w widoku, co umożliwi odwoływanie się do nich oraz ich edycję.

W przypadku widoku `ScheduleEventView` (List. 4) ważne jest dodanie obiektu `ScheduleEventEmbed`, który będzie przechowywał model danych. Z uwagi na to, że to *embed* zarządza danymi przy użyciu modelu oraz wyświetla je użytkownikowi, należy przekazać go do każdego elementu korzystającego z danych, jak na przykład obiektu przycisku do dodawania uczestników klasy `AddParticipantButton`.

Ze względu na to, iż widok ten po zaakceptowaniu wysyła prywatne wiadomości do uczestników musiał on mieć dostęp do funkcjonalności bota discordowego, a więc i obiektu `bot`.

`Embed` (List. 5) oprócz podstawowej funkcjonalności posiadał również metodę *reload_embed*, dzięki której możliwa była zmiana wyświetlanych informacji po interakcjach użytkownika. Z uwagi na to, że klasa `Embed` nie posiada metod do aktualizacji informacji [16] należy usunąć wszystkie informacje, a następnie dodać je ponownie. Tutaj właśnie wykorzystywane są zapisane w modelu `EventModel` informacje.

```

1 class ScheduleEventEmbed(discord.Embed):
2     def __init__(self, model:EventModel, title:str="Schedule Event"):
3         super().__init__(title=title, color=discord.Color.pink())
4         self.model = model

```

```

5
6 def reload_embed(self):
7     self.clear_fields()
8     if self.model.picked_datetime:
9         self.add_field(
10             name="Date",
11             value=utils.datetime_to_str(self.model.picked_datetime),
12             inline=False
13         )
14     self.add_field(
15         name="Event Name",
16         value=self.model.name,
17         inline=False
18     )
19     if self.model.description:
20         self.add_field(
21             name="Description",
22             value=self.model.description,
23             inline=False
24         )
25     if self.model.duration:
26         self.add_field(
27             name="Duration",
28             value=f"{self.model.duration} minutes",
29             inline=False
30         )
31     self.add_field(
32         name="Participants",
33         value=self.model.get_trunc_participants(),
34         inline=False
35     )
36     return self

```

Listing 5: Klasa ScheduleEventEmbed

Elementy widoku takie jak przyciski, pola wyboru czy modal zdefiniowano jako obiekty osobnych klas, które dziedziczyły po podstawowych klasach modułu *discord.ui*. Większość z nich zawierała asynchroniczną metodę *callback* (List. 6). Dzięki niej możliwe było reagowanie na interakcje użytkownika z danym elementem. Z uwagi na ograniczenia czasu reakcji aplikacji spowodowane przez Discord API, kiedy wykonywano zapytania do bazy danych lub skomplikowane obliczenia odraczano dostarczenia odpowiedzi przy użyciu metody *defer*. Następnie, przy użyciu atrybutu *followup*, możliwe było dokończenie reakcji na interakcję użytkownika.

```

1 async def callback(self, interaction: Interaction):

```

```

2  await interaction.response.defer()
3  if event_id := self.embed.model.save_in_database():
4      for p in self.embed.model.participants:
5          self.view.bot.dispatch(
6              "start_schedule_event",
7              event_id,
8              p.id
9          )
10     await interaction.followup.edit_message(
11         interaction.message.id,
12         content="Event created!",
13         embed=None,
14         view=None
15     )
16 else:
17     await interaction.followup.send_message(
18         interaction.message.id,
19         content=utils.error_message("Ups, something went wrong!")
20     )

```

Listing 6: Asynchroniczna funkcja *callback* klasy `SaveButton`

Oprócz podstawowych elementów widoku widok ten zawierał również modal pozwalający na zapisanie wszystkich niezbędnych informacji potrzebnych do stworzenia wydarzenia. Modal podobnie jak widok pozwalał na dodawanie elementów, jednak mógł on zawierać w sobie jedynie pola tekstowe, czyli obiekty klasy `discord.ui.TextInput`. Tak jak w przykładzie poniżej (List. 7) używano zarówno wartości domyślnych, jak i symboli zastępczych, aby użytkownik wiedział w jakim formacie zapisać informacje.

```

1  self.start_time = ui.TextInput(
2      label="FROM",
3      placeholder="DD/MM/YYYY",
4      default=default_start_date,
5      required=True
6  )
7  self.add_item(self.start_time)

```

Listing 7: Dodawanie pola tekstowego do modalu

Modal posiadał 2 asynchroniczne funkcje, które można było nadpisać: *on_submit* (List. 8) oraz *on_error*. Z uwagi na to, że użytkownik mógł błędnie wpisać informacje (zwłaszcza przy wpisywaniu dat), sprawdzano poprawność danych przed ich zapisaniem. Użyto do tego funkcji *validate*, która w razie znalezienia błędu w jakimś polu zwracała jego nazwę.

```

1  async def on_submit(self, interaction: Interaction) -> None:

```

```

2     if field := self.validate():
3         await interaction.response.send_message(
4             utils.error_message(f"Invalid input: {field}"),
5             ephemeral=True
6         )
7         return
8
9     self.view.add_description_button.label="Edit Info"
10    self.view.add_description_button.style=discord.ButtonStyle.
secondary
11    self.embed.model.name=self.name.value
12    self.embed.model.description=self.description.value
13    self.embed.model.duration=self.duration.value
14    self.embed.model.set_start_date(self.start_time.value)
15    self.embed.model.set_end_date(self.end_time.value)
16
17    self.view.save_button.is_info_added = True
18    self.view.save_button.maybe_able()
19
20    await interaction.response.edit_message(
21        view=self.view,
22        embed=self.embed.reload_embed()
23    )

```

Listing 8: Metoda *on_submit* klasy *AddDescriptionModal*

Inne komendy zostały stworzone w sposób analogiczny wykorzystując taką samą strukturę i schemat działania.

Wybór dogodnych terminów, czyli komenda */select-dates*, umożliwiała użytkownikowi interakcję z widokiem *SelectDatesView* (Rys. 11).

Pokazywał on przy użyciu *embedu* dzień, dla którego wybierano dostępność, pola wielokrotnego wyboru oraz przyciski do zmiany dnia.

Pola wyboru dziedziczyły po klasie *discord.ui.Select*, która wymagała, aby wszystkie możliwe opcje (w tym ich domyślne wartości) były zdefiniowane podczas konstrukcji obiektu. Maksymalna liczba opcji, jakie można dodać to 25, dlatego zdecydowano się na umożliwienie wyboru o równych godzinach (24 opcji). W przypadku godzin, które były już zajęte przez dowolnego uczestnika wydarzenia, były one oznaczane jako "Unavailable" i ignorowane w przypadku ich wyboru.

Przyciski z pojedynczą strzałką pozwalały na zmianę daty o jeden dzień, natomiast te z podwójnymi strzałkami o cały tydzień. Zmiana dnia powodowała przeładowanie się nie tylko *embedu*, ale i stworzenie nowych obiektów pól wielokrotnego wyboru klasy *SelectHours*.

Kolejnym elementem interfejsu użytkownika był `ShowEventsEmbed` ukazujący się po wykonaniu komendy `/show-events`. Tutaj ze względu na to, iż miał on jedynie pokazywać informacje, a nie zbierać dane od użytkownika, użycie widoku było zbędne. Obiekt tej klasy przyjmował podczas swojego tworzenia typ (status) wydarzeń, które należało wyświetlić. Te, które zostały utworzone, natomiast ich data nie została jeszcze wybrana posiadały status `"created"`, natomiast te z wybranym terminem `"confirmed"` (Rys. 14). Odrzucone wydarzenia posiadały status `"canceled"`.

Ostatnie dwie komendy poprzedzone ukośnikiem `/reschedule-event` i `/cancel-event` wyglądały bardzo podobnie i polegały na wybraniu wydarzenia przy użyciu pola wyboru.

Informacje zwrotne oraz błędy przekazywane były jako formatowane wiadomości (czasami efemeryczne, to znaczy widoczne jedynie dla osoby, która dokonała interakcji z danym widokiem). Pierwsze kolorowane są na niebieski kolor, natomiast drugie na czerwono (Rys. 4).

3.4 Modelowanie danych

Dane przetrzymywano w dokumentowej bazie danych MongoDB. Każde wydarzenie posiadało jeden dokument (List. 9), w którym znajdowały się wszystkie informacje na jego temat. Zalety jakie się z tym wiązały znacznie przewyższały ograniczenia jakie mogły spowodować [17].

```
1 {  
2   "creator_id": "393456997359550474",  
3   "guild_id": "896504932604198983",  
4   "name": "Klub Książki 5",  
5   "description": "",  
6   "duration": "120",  
7   "participants": [  
8     "393456997359550474"  
9   ],  
10  "start_date": "2023-12-23T00:00:00.000Z",  
11  "end_date": "2023-12-30T00:00:00.000Z",  
12  "status": "confirmed",  
13  "availability": [...],  
14  "date": "2023-12-24T19:00:00.000Z"  
15 }
```

Listing 9: Przykładowy dokument w bazie danych

Przedstawiony dokument zawierał pola:

- `"creator_id"` - indeks użytkownika, który stworzył wydarzenie,
- `"guild_id"` - indeks serwera discordowego, w którym wydarzenie zostało stworzone,

- "name" - nazwa wydarzenia,
- "description" - krótki opis wydarzenia,
- "duration" - czas wydarzenia zapisany w minutach,
- "participants" - lista indeksów użytkowników biorących udział w wydarzeniu,
- "start_date" - początek ram czasowych, w których planowane jest wydarzenie,
- "end_date" - koniec ram czasowych, w których planowane jest wydarzenie,
- "status" - status wydarzenia, może być "confirmed", "created" lub "canceled",
- "availability" - jest to lista dokumentów przechowujących informacje na temat dostępności każdego z uczestników wydarzenia,
- "date" - ustalona data wydarzenia (w przypadku wydarzeń ze statusem "created" to pole było puste).

Wydarzenia, które zostały utworzone posiadają status "created". Jeżeli proces wybierania terminu powiedzie się to zostaje on zmieniony na "confirmed". W przypadku anulowania wydarzenia w dowolnym momencie status zmienia się na "canceled".

Do komunikowania się z bazą danych wykorzystano modele znajdujące się w folderze `api`. Były to `EventModel` przechowujący wszystkie informacje o wydarzeniu oraz `AvailabilityModel` służący do zarządzania jedynie dostępnością. Drugi był wykorzystywany przy komendzie `/select-dates`.

Klasa `EventModel` posiadała metody pozwalające przekształcać odpowiedzi użytkownika do odpowiednich typów danych, na przykład podczas ustawiania pola "start_date" (List. 10).

```

1 def set_start_date(self, date: str) -> bool:
2     try:
3         self.start_date = utils.str_to_date(date)
4         return True
5     except Exception:
6         return False

```

Listing 10: Funkcja `set_start_date` klasy `EventModel`

Dodatkowo klasa ta posiadała metody statyczne zwracające obiekty tych klas uzupełnione informacjami z bazy danych (List. 11). Funkcja `get_from_database` zwracała wydarzenie o konkretnym indeksie, natomiast `get_from_database_by_creator` filtrowała wydarzenia po indeksie twórcy, statusie oraz pozwalała na ograniczenie ilości wyników poprzez zmienną `limit`.


```

1 def get_from_database(event_id: str, bot: commands.Bot, status: str =
  None):
2     collection = get_database()["events"]
3     filter = {"_id": ObjectId(event_id), "status": status} if status
    else {
4         "_id": ObjectId(event_id)}
5     if event := collection.find_one(filter):
6         return model_from_database_data(event, bot)
7     return None
8
9 def get_from_database_by_creator(creator_id: int, bot: commands.Bot,
  limit: int = 0, status: str = None):
10    collection = get_database()["events"]
11    if status:
12        query = {"creator_id": creator_id, "status": status}
13    else:
14        query = {"creator_id": creator_id}
15    if events := collection.find(query).sort("date", DESCENDING).limit
    (limit):
16        return [model_from_database_data(event, bot) for event in
    events]
17    return None

```

Listing 11: Metody klasy EventModel służące do zwracania wydarzeń

Model AvailabilityModel pozwalał na przetrzymywanie szczegółowych informacji na temat dostępności danej osoby oraz zapisanie ich jako dokumentu, będącego elementem listy "availability" w dokumencie wydarzenia.

Klasa ta (List. 12) zawierała atrybuty zawierające informacje dotyczące danego dnia oraz ram czasowych wydarzenia (current_date, start_date, end_date), dane wydarzenia oraz użytkownika wypełniające informacje (event_id, user_id) oraz listę słowników availability odpowiadającą za dostępności danej osoby w kolejnych dniach. Tylko ostatni atrybut był zapisywany w bazie danych, jednak dzięki strukturze modelu pozwalającej kontrolować aktualnie wybrany dzień, zapisywanie odpowiedzi użytkownika do odpowiedniej struktury było o wiele łatwiejsze.

```

1 class AvailabilityModel:
2     times = [time(hour=hour) for hour in range(24)]
3     def __init__(self, event_id: int, user_id: int, start_date: datetime
    , end_date: datetime) -> None:
4         self.current_date = start_date
5         self.start_date = start_date
6         self.end_date = end_date
7         self.event_id = event_id

```

```

8     self.user_id = user_id
9     self.availability = {
10         utils.date_to_str(start_date + timedelta(days=i)): {"ok": [], "
maybe": [], "no": []} for i in range((end_date - start_date).days
+ 1)
11     }
12     self.not_available_datetimes = self.
get_user_not_available_datetimes()
13 def get_user_not_available_datetimes(self):
14     if user := get_database()["users"].find_one({"user_id": self.
user_id}):
15         return [e["date"] for e in user["events"]]
16     return None
17 ...

```

Listing 12: Fragment klasy AvailabilityModel

Dodatkowo, klasa ta posiadała metody odpowiedzialne za zmianę wybranego dnia (*change_day*), sprawdzenia czy dana godzina jest dostępna (*is_time_available*) oraz dodania listy godzin (*add_times*) danego dnia, z podaną dostępnością ("ok", "maybe" lub "no").

Oprócz dokumentów wydarzeń w bazie danych zapisywano również ustawienia dla każdego serwera (List. 13), a dokładnie indeks kanału, na którym bot powinien się komunikować z użytkownikami.

```

1 {
2     "guild_id": "900077363398852620",
3     "bot_channel_id": "1117531070011801762"
4 }

```

Listing 13: Dokument kolekcji "config"

3.5 Algorytm wyboru

Funkcja *pick_date* (List. 14) była odpowiedzialna za wybór odpowiedniego dnia oraz godziny i znajdowała się w pliku *algorithms.py*.

```

1 def pick_date(availabilities:list, duration:int) -> datetime or None:
2     ok_datetimes,maybe_datetimes,no_datetimes = parse_availabilities(
3         availabilities
4     )
5     full_additional_duration_hours = math.ceil(max(0,duration-60)/60)
6     no_datetimes = add_duration_to_no_times(
7         no_datetimes,
8         full_additional_duration_hours

```

```

9 )
10 if ok_datetime := select_datetime(ok_datetimes, no_datetimes):
11     return ok_datetime
12 if maybe_datetime := select_datetime(
13     ok_datetimes + maybe_datetimes,
14     no_datetimes
15 ):
16     return maybe_datetime
17 else:
18     return None

```

Listing 14: Funkcja *pick_date*

Korzystała ona z funkcji pomocniczych zawartych w tym samym pliku. Algorytm na początku parsował dane przy użyciu *parse_availabilities* zwracając trzy listy zawierające terminy, które najbardziej pasują (*ok_datetimes*), w których są dostępni (*maybe_datetimes*) oraz takie, kiedy uczestnicy są niedostępni (*no_datetimes*).

Później dodawał czas trwania wydarzenia zaokrąglony w górę do pełnych godzin do niedostępnych terminów wykorzystując funkcję *add_duration_to_no_datetimes*. Polegało to na dodaniu tylu dodatkowych godzin przed każdym niedostępnym terminem ile zaokrąglonych godzin uzyskano. Zapewniało to wystarczająco dużo czasu by wybrany termin nie kolidował z godzinami, w których użytkownik nie ma czasu na spotkanie.

Następnie przy użyciu list wybierano odpowiedni termin wykorzystując do tego funkcję *select_datetime* (List. 15). Sprawdzano najpierw godziny, które użytkownicy zaznaczyli jako preferowane, a następnie uwzględniane były wszystkie dopuszczalne terminy, czyli odpowiednio *ok_datetimes* i *maybe_datetimes*.

```

1 def select_datetime(datetimes:list, no_datetimes:list) -> datetime:
2     distinct_datetimes = remove_repetitions(datetimes)
3     filtered_datetimes = [
4         dt for dt in distinct_datetimes if dt not in no_datetimes
5     ]
6     if legit_datetimes := [
7         i for i in datetimes if i in filtered_datetimes
8     ]:
9         if len(legit_datetimes) > 0:
10             return Counter(legit_datetimes).most_common(1)[0][0]
11         else:
12             return None
13     else:
14         return None

```

Listing 15: Funkcja *select_datetime*

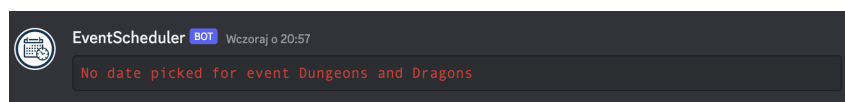
Wybór godziny i daty składa się z 5 etapów. W pierwszej kolejności tworzone listę zawierającą terminy, w której nie ma powtórzeń (`distinct_datetimes`) przy użyciu funkcji `remove_repetitions`.

Następnie zapisywano listę tylko tych godzin, które nie znajdowały się w liście `no_datetimes`. Tworzyła ona filtr wykorzystywany później do wygenerowania listy `legit_datetimes` zawierającej odpowiednie daty wraz z powtórzeniami.

Po otrzymaniu listy godzin sprawdzono, czy nie jest ona pusta, co by oznaczało, że nie istnieje godzina odpowiadająca wszystkim osobom. W takim przypadku funkcja zwracała `None`.

W przeciwnym przypadku wybierano najczęściej powtarzającą się wartość. Użyto do tego klasy `Counter` z wbudowanego modułu `collections`.

Jeśli zarówno przy wyborze dogodnych terminów, jak i wszystkich możliwych nie udało się znaleźć pasującej godziny algorytm zwrócił `None`, a aplikacja przekazywała użytkownikom informację (Rys. 4), że nie udało się znaleźć terminu, który odpowiadałby wszystkim.



Rysunek 4: Wiadomość informująca o braku terminu dla wydarzenia

W takim przypadku twórca wydarzenia mógł ponownie rozpocząć proces wybierania terminu przy użyciu komendy `reschedule_event`. Użytkownicy byli wtedy w stanie zmienić swoje decyzje, a tym samym umożliwić aplikacji wybór.

Algorytm nie sprawdzał, czy termin, który wybierze użytkownik nie jest już zajęty przez inne spotkanie, gdyż było to wcześniej weryfikowane podczas wybierania przez uczestników ich dostępności (dana odpowiedź nie była dla nich dostępna).

3.6 Testy

Z uwagi na dużą ilość graficznej funkcjonalności aplikacji w postaci rozbudowanego interfejsu użytkownika stworzonego przy użyciu gotowych komponentów udostępnianych przez Discord zdecydowano się na stworzenie testów jednostkowych dla algorytmu wyboru terminu, z uwagi na to iż jest to najważniejszy element aplikacji. Zastosowano najpopularniejszą bibliotekę do pisania testów w języku Python, czyli `pytest`. Aby uruchomić testy można skorzystać ze skryptu `test.sh` znajdującego się w folderze `bot`.

Kod testujący został zaprojektowany w taki sposób, aby możliwe było jego łatwe rozszerzenie. Pozwala to na szybkie dodawanie nowych przypadków testowych wraz z

możliwym rozwojem aplikacji.

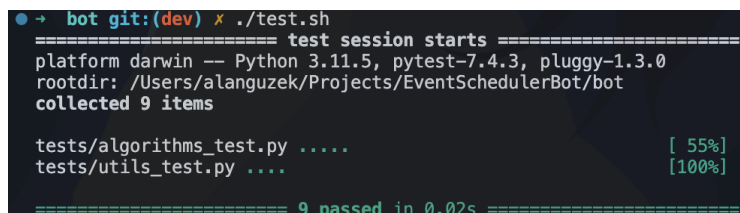
Oprócz głównej funkcji wybierającej termin testowane są również funkcje pomocnicze. Do symulacji danych wejściowych dla funkcji skorzystano z mechanizmu znanego jako „ustawienia pomocnicze” (ang. fixtures). Umożliwia on tworzenie i wykorzystywanie predefiniowanych obiektów wykorzystywanych w późniejszych testach.

Przykładowy test jednostkowy (List. 16) definiowany był w module jako funkcja zaczynająca się od `test_`. W tym przypadku przyjmowała ona jeden argument, którym były ustawienia pomocnicze zawierające przykładowe dostępności.

```
1 def test_select_datetime_ok_datetimes(example_availability):
2     ok_datetimes, _, no_datetimes = algorithms.parse_availabilities(
3         example_availability
4     )
5     assert algorithms.select_datetime(
6         ok_datetimes,
7         no_datetimes
8     ) == datetime(2023, 11, 13, 1)
```

Listing 16: Test sprawdzający funkcję `select_datetime`

Uruchomienie testów (Rys. 5) potwierdziło poprawność działania algorytmu oraz funkcji pomocniczych.



```
bot git:(dev) x ./test.sh
===== test session starts =====
platform darwin -- Python 3.11.5, pytest-7.4.3, pluggy-1.3.0
rootdir: /Users/alanguzek/Projects/EventSchedulerBot/bot
collected 9 items

tests/algorithms_test.py ..... [ 55%]
tests/utils_test.py .... [100%]

===== 9 passed in 0.02s =====
```

Rysunek 5: Wynik uruchomienia testów jednostkowych

4 Prezentacja

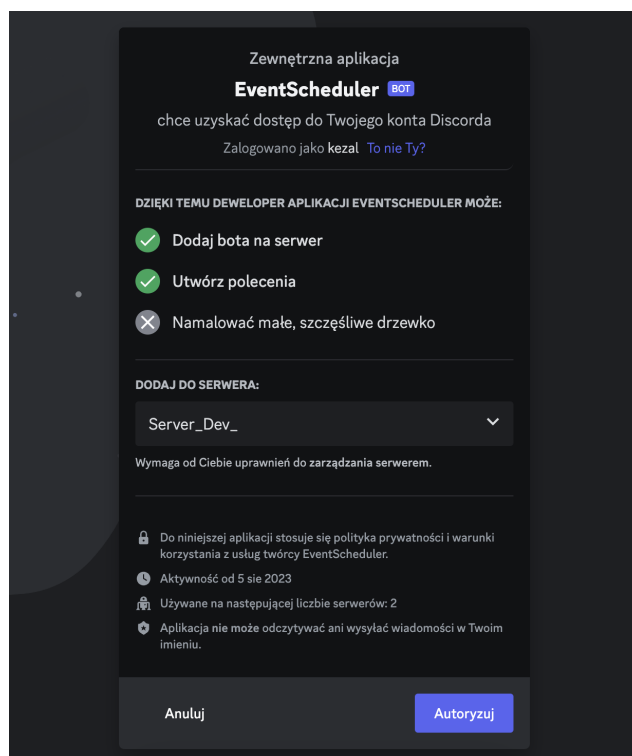
4.1 Sposób użycia

4.1.1 Dodawanie aplikacji do gildii

Korzystanie z gotowej aplikacji udostępnianej na serwerze Railway.app [18] należy rozpocząć od dodania bota do swojej gildii na platformie <https://discord.com/>. Wymagane do tego jest posiadanie konta oraz własnej gildii (należy być administratorem, gdyż będzie to potrzebne podczas ustawiania kanału odpowiedzialnego za komunikację z botem).

W celu dodania aplikacji należy skorzystać z linku [19]. Można go znaleźć w zakładce OAuth na stronie <https://discord.com/developers/applications/>.

Po kliknięciu w link wyświetlane jest okno, w którym możliwe jest wybranie serwera, do którego chcemy zaprosić bota (Rys. 6). Wynika to z tego, że dana osoba może mieć wiele gildii, które do niej należą.



Rysunek 6: Autoryzacja bota na własnym serwerze

Po dodaniu bota sugerowane jest utworzenie oddzielnego kanału tekstowego, a następnie użycie w nim komendy `!set-channel`, która ustawi aktualny kanał tekstowy na miejsce do komunikacji aplikacji z użytkownikami. Kiedy aplikacja będzie potrzebowała nadać komunikat samoistnie, a nie tylko odpowiedzieć na wiadomość użytkownika, zostanie ona wysłana na tym właśnie kanale.

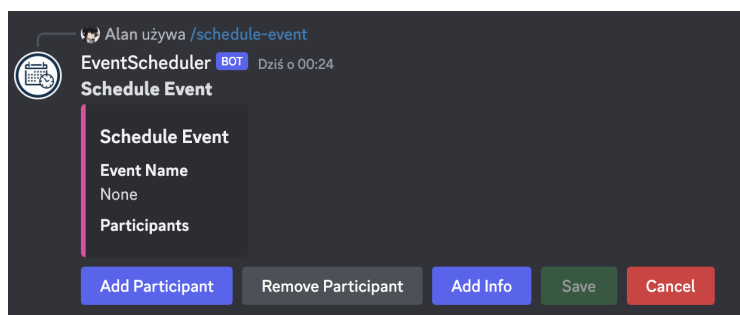
Po wykonaniu powyższych instrukcji korzystanie z aplikacji jest możliwe dla każdego członka gildii.

4.1.2 Ustalanie terminu spotkania

Proces ustalania terminu przy użyciu zaproszonego bota wymaga wykonania następujących kroków.

W pierwszej kolejności wywołujemy komendę `/schedule-event`, na którą aplikacja

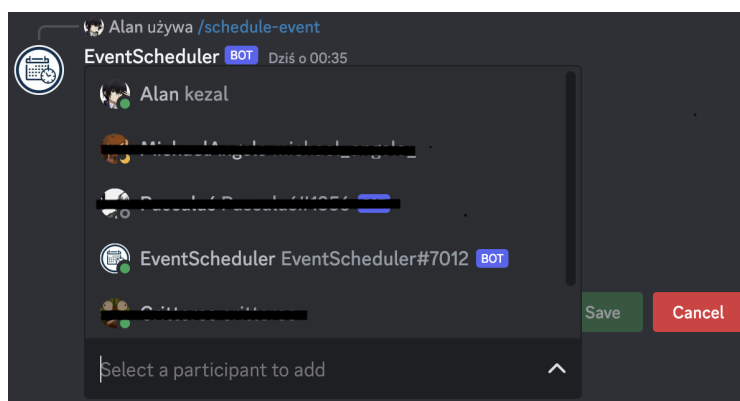
zareaguje wysłaniem wiadomości z odpowiednim widokiem (Rys. 7) na kanale, w którym użytkownik ją wywołał. Jeśli w trakcie rozmyślimy się, lub też było to przypadkowe użycie komendy, możemy skorzystać z przycisku *Cancel*.



Rysunek 7: Widok reakcji na komendę ”/schedule-event”

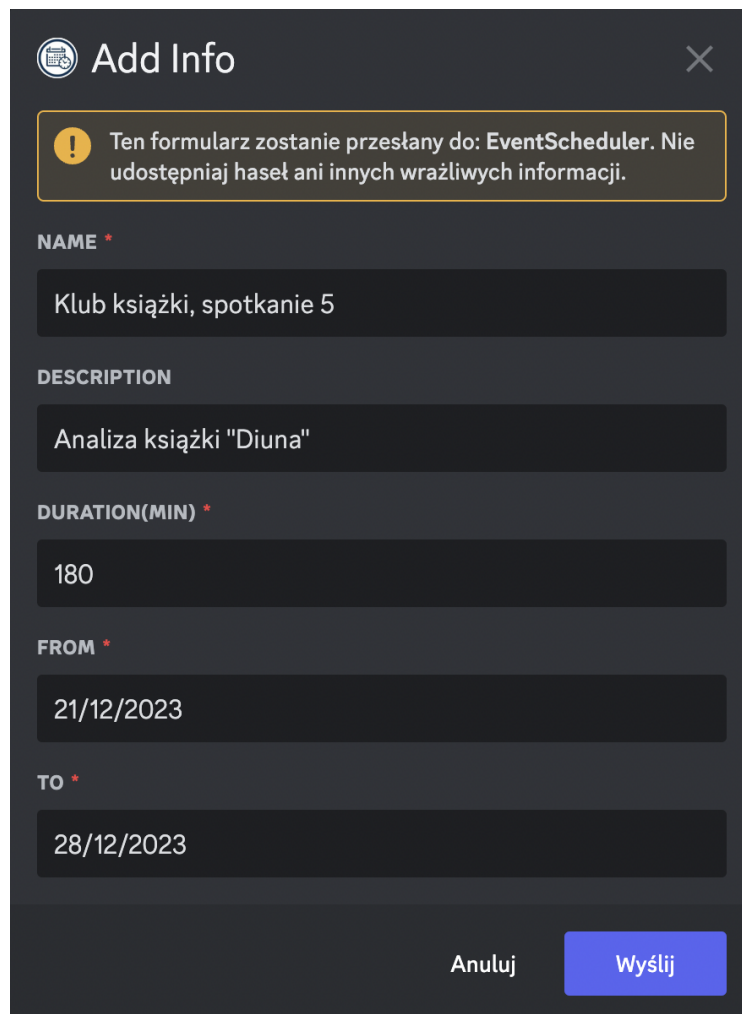
Aby utworzyć wydarzenie należy dodać użytkowników oraz uzupełnić istotne dla niego dane, w szczególności nazwę, czas trwania oraz zakres dat, w których wydarzenie jest planowane, przy czym kroki te można wykonać w dowolnej kolejności.

Dodawanie uczestników możliwe jest przy użyciu przycisku *Add Participant* (Rys. 8). W przypadku pomyłki lub zmiany zdania możliwe jest usunięcie użytkownika za pomocą przycisku *Remove Participant*. Jeżeli do wydarzenia dodany został przynajmniej jeden uczestnik, kolor przycisku zmienia się na szary sygnalizując, iż możliwe jest przejście do kolejnego etapu.



Rysunek 8: Widok pola wyboru ”Add Participant”

Zdefiniowanie kluczowych informacji o wydarzeniu możliwe jest natomiast przy użyciu przycisku *Add Info*, który otworzy modal przedstawiony na Rys. 9. Obowiązkowe do wypełnienia pola oznaczone są czerwoną gwiazdką.



Add Info

! Ten formularz zostanie przesłany do: EventScheduler. Nie udostępniaj haseł ani innych wrażliwych informacji.

NAME *

Klub książki, spotkanie 5

DESCRIPTION

Analiza książki "Diuna"

DURATION(MIN) *

180

FROM *

21/12/2023

TO *

28/12/2023

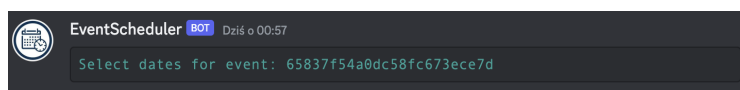
Anuluj Wyślij

Rysunek 9: Modal dodawania informacji "Add Info"

W przypadku podania błędnych danych, na przykład źle sformatowanej daty modal zostanie zamknięty, a aplikacja wyśle komunikat o błędzie. W takim przypadku należy ponownie podać wszystkie informacje. Po poprawnym wpisaniu wszystkich wymaganych danych nazwa przycisku zostanie zmieniona na *Edit Info*, a jego kolor na szary.

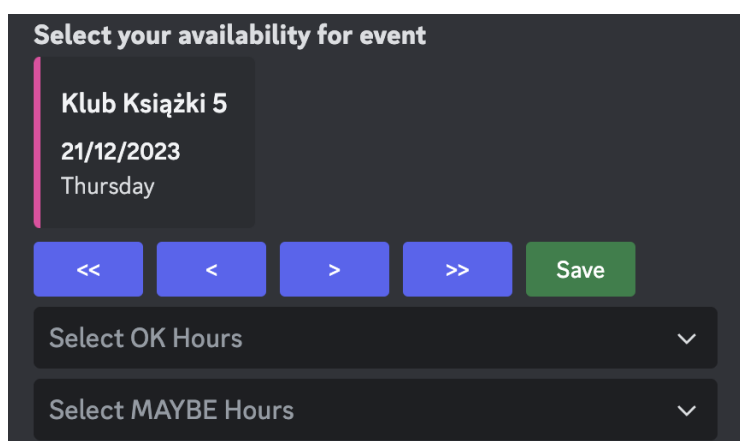
Po wypełnieniu informacji oraz dodaniu uczestników odblokowany zostanie przycisk *Save*, którym możemy zapisać informacje i rozpocząć proces wybierania terminu.

Do każdego z uczestników wydarzenia zostanie wysłana prywatna wiadomość z prośbą o zaznaczenie swojej dyspozycyjności dla danego nowego wydarzenia (Rys. 10). Należy skopiować ID wydarzenia a następnie w dowolnym kanale gildii (preferowany jest kanał do komunikacji z botem) użyć komendy `/select-dates` dodając jako parametr skopiowane ID.



Rysunek 10: Wiadomość przypominająca o wysłaniu dostępności

Jeżeli ID jest poprawne zostanie do nas wysłana prywatna wiadomość z widokiem pozwalającym na wybranie wolnych terminów w danym okresie (Rys. 11). Dwuetapowy sposób wysyłania interfejsu pozwalającego na wybór dostępności jest spowodowany tym, że istnieje ograniczony czas, przez jaki aplikacja może reagować na zdarzenia takie jak kliknięcie przycisku, dlatego pozwalamy użytkownikowi uruchomić ten proces w momencie, w którym będzie miał na to czas.



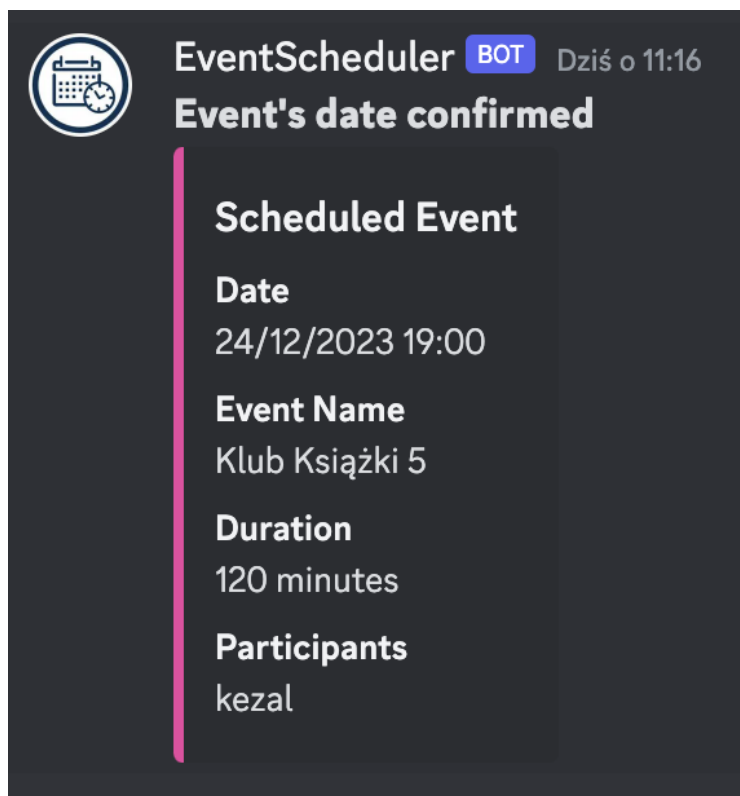
Rysunek 11: Widok wyboru dostępności

Pojedyncze strzałki umożliwiają zmianę aktualnie wybranego dnia, natomiast podwójne strzałki przeniosą nas o cały tydzień. Pod przyciskami strzałek znajdują się 2 pola wielokrotnego wyboru:

- *Select OK Hours* - pozwala wybrać terminy, w których chcemy, aby spotkanie się odbyło,
- *Select MAYBE Hours* - daje możliwość wyboru terminów, w których jesteśmy dostępni (natomiast mogą nie być dla nas idealne).

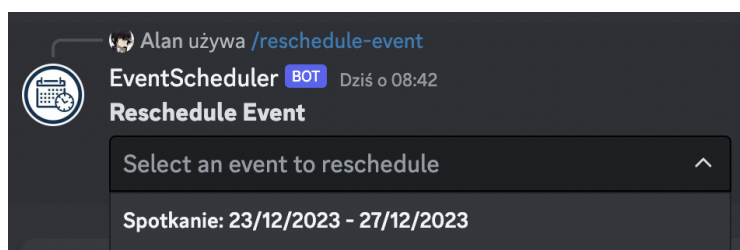
Podział na te dwie kategorie daje możliwość uwzględnienia preferencji uczestników, jak również powiększa pulę dostępnych opcji, co zwiększa szanse na znalezienie terminu odpowiadającego każdemu. Godziny, które nie są zaznaczone w żadnym z tych pól zostaną uznane za te, w których uczestnik nie ma czasu. Po zaznaczeniu swojej dyspozycyjności klikamy przycisk *Save*.

Kiedy wszyscy uczestnicy wydarzenia udzielili odpowiedzi, aplikacja uruchomi algorytm wyszukiwania terminu i wyśle wiadomość na głównym kanale pokazującą wydarzenie z wybranym terminem (Rys. 12) lub informację, że nie udało się wyznaczyć godziny pasującej każdemu (Rys. 4).



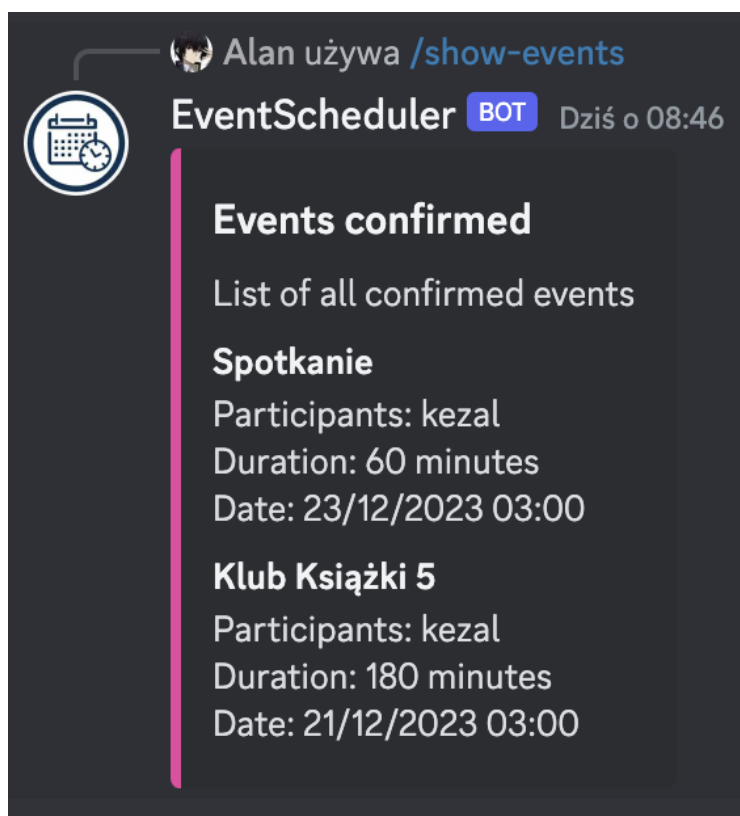
Rysunek 12: Wiadomość informująca o znalezionym terminie dla wydarzenia

Jeśli dojdzie do drugiej sytuacji możliwe jest ponowne uruchomienie procesu wybierania godziny przy użyciu komendy `/reschedule-event` z parametrem `"Created"` (Rys. 13). Z uwagi na brak wybranego dnia wydarzenie nie mogło zostać potwierdzone, czyli posiadać statusu `"Confirmed"`.



Rysunek 13: Użycie komendy `/reschedule-event`, z parametrem `"Created"`

Jeśli którykolwiek użytkownik chce zobaczyć wydarzenia, które organizuje oraz na jakim są one etapie może skorzystać z komendy `/show-events` (Rys. 14), która podobnie jak `/reschedule-event` posiada parametr odpowiadający za wybór statusu wydarzenia ("Created", "Confirmed", "Canceled").



Rysunek 14: Embed komendy `/show-events` dla statusu "Confirmed"

Jeżeli z dowolnego powodu należy odwołać wydarzenie czy to stworzone, czy też potwierdzone można skorzystać z komendy `/cancel-event`. Pozwoli ona wybrać wydarzenie, a następnie, jeśli było ono już potwierdzone, wyśle do każdego uczestnika wiadomość, iż zostało ono usunięte.

4.2 Hosting

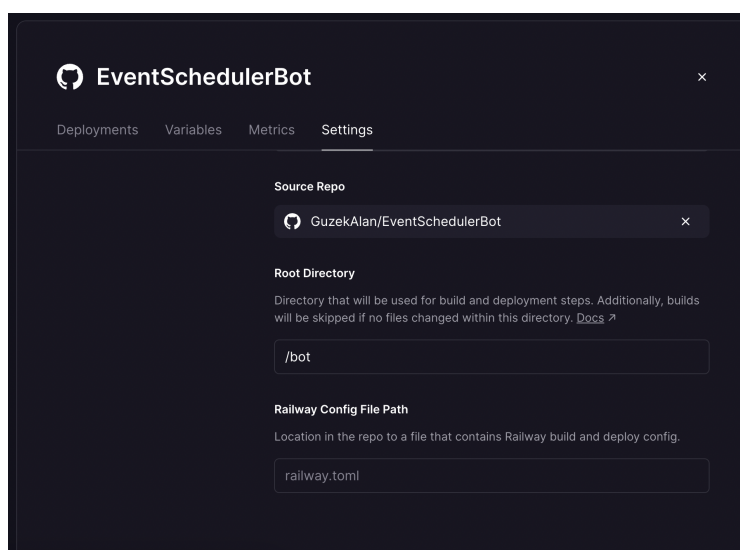
W przeciwieństwie do aplikacji internetowych, możliwość korzystania z bota w sieci była dostępna nawet uruchamiając go w lokalnym środowisku. Nie wymaga on udostępniania swojego adresu IP i portu w sieci, lecz przy użyciu tokenu tworzy stałe połączenie z serwerami Discord.

Dodatkowo należy pamiętać, że uruchomienie kilku aplikacji połączonych tym samym tokenem do tego samego bota powoduje, że będzie on reagował wielokrotnie na

poszczególne komendy. W celu uzyskania wielu aplikacji, z których każda reagowałaby jedynie na swoje komendy, należałoby je osobno zarejestrować w portalu dewelopera.

Używanie lokalnego środowiska w celu udostępniania swojej aplikacji innym użytkownikom nie jest najlepszym pomysłem, gdyż usługa powinna być dostępna cały czas. Z tego względu zdecydowano się użyć do tego specjalnego serwera. Ilość platform oferujących usługi hostingu jest duża. Dodatkowo możliwe jest użycie własnego serwera, jeśli się taki posiada. Podczas tworzenia aplikacji zdecydowano się użyć platformy Railway [18]. Pozwala ona na proste wdrażanie aplikacji internetowych lub jak w tym przypadku botów discordowych. Dzięki połączeniu z GitHubem, na którym znajduje się zdalne repozytorium projektu Railway jest w stanie przebudowywać aplikację za każdym razem, kiedy zauważy zmiany.

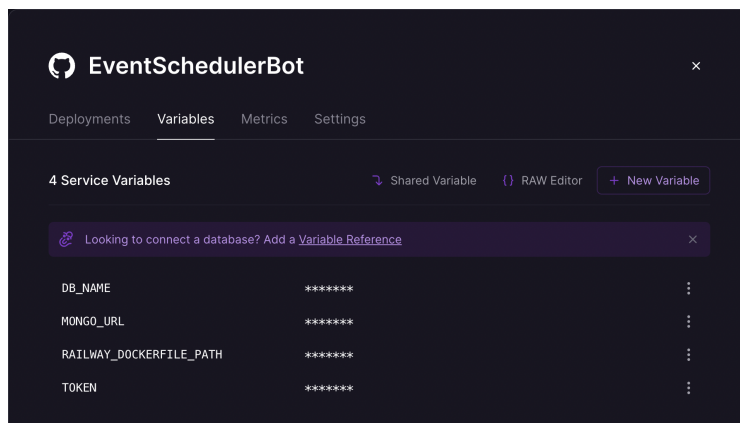
Aby skorzystać z serwisu należy podłączyć swoje konto na platformie GitHub, a następnie wybrać projekt. Z uwagi na to, iż projekt zawiera plik Dockerfile, Railway może zbudować oraz uruchomić naszą aplikację bez żadnego problemu. Nie obsługuje on jednak plików "docker-compose.yml", dlatego postanowiono zbudować samą aplikację bez bazy danych. W takiej sytuacji należało również zmienić główny folder na `/bot` (Rys. 15).



Rysunek 15: Ustawienia aplikacji w serwisie Railway

Bazę danych stworzono jako oddzielny serwis. Railway posiada gotowe szablony dla popularnych baz danych.

Po utworzeniu serwisu dla MongoDB uzupełniono zmienne środowiskowe aplikacji w zakładce *Variables* (Rys. 16).



Rysunek 16: Panel zmiennych środowiskowych serwisu w aplikacji Railway

Odpowiadały one za:

- DB.NAME - nazwę bazy danych (mogła być to dowolna nazwa)
- MONGO_URL - link do bazy danych zawierający login oraz hasło w celu uwierzytelnienia
- RAILWAY_DOCKERFILE_PATH - wskazywał ścieżkę do pliku Dockerfile, jeśli nie znajdował się on w głównym folderze
- TOKEN - pozwalał na uwierzytelnienie aplikacji w serwisie Discrod (Rys. 1)

W przypadku hostowania aplikacji na własnym serwerze należy uzupełnić plik ".env" a następnie wykonać komendę przedstawioną powyżej (List. 1). Plik Dockerfile dla aplikacji pythonowej został skonfigurowany dla systemu operacyjnego Linux i nie jest przystosowany do pracy na innych systemach.

Jeżeli wystąpiły problemy podczas uruchamiania kontenerów dockerowych mogło to być spowodowane brakiem demona odpowiedzialnego za konteneryzację. W takiej sytuacji należy użyć komendy poniższej komendy (List. 17).

```
1 $ sudo dockerd &
```

Listing 17: Uruchamianie demona dockerowego

5 Podsumowanie

Celem projektu było stworzenie aplikacji discordowej umożliwiającej wybór terminu spotkania dla grupy osób. Miała ona za zadanie zebrać informacje od wszystkich użytkowników odnośnie ich dostępności, a następnie wybrać najlepszy termin odpowiadający wszystkim osobom.

Utworzona aplikacja spełnia wszystkie założenia, jakie zostały jej postawione. Liczba elementów interfejsu użytkownika, które udostępnia Discord jest ograniczona, jednak mimo to aplikacja umożliwia wygodne w obsłudze widoki pozwalające zarówno zapisać wszystkie informacje o wydarzeniu używając przy tym walidacji, jak również zaznaczyć swoją dostępność na przestrzeni wielu dni.

Podjęcie obiektowe oraz kompozycja większych elementów takich jak widoki z mniejszych (na przykład przyciski czy pola wyboru) umożliwiła napisanie interfejsu w przejrzysty sposób. Stworzenie reaktywnych komponentów, które działałyby w zamierzony sposób wiązało się w niektórych momentach z użyciem zabiegów nieoczywistych takich jak usuwanie oraz dodawanie elementu (z uwagi na brak możliwości ich edycji).

Wydarzenia zapisywano w bazie danych w postaci jednego dokumentu, co ułatwiło ich przegląd oraz dostęp do wszystkich potrzebnych elementów. Sama baza danych służyła jedynie do przechowywania informacji. Nie używano skomplikowanych zapytań przez co nie trzeba było używać bazy relacyjnej.

Podczas implementacji algorytmu wyboru skupiono się na łatwym pozyskiwaniu i przetwarzaniu danych, a nie na optymalizacji czasowej, gdyż cały proces wyboru odpowiedniego terminu często przeprowadzany będzie na przestrzeni kilku dni.

Pomimo faktu, iż aplikacja jest w pełni działająca możliwe jest udoskonalenie jej oraz rozwinięcie poprzez dodanie nowych funkcjonalności takich jak:

- dodanie możliwości przypominania o wydarzeniach,
- synchronizacja z kalendarzami (iCalendar, Kalendarz Googale, itd.),
- dodanie etykiet i umożliwienie wyszukiwania wydarzeń używając ich.

Repozytorium z kodem projektu znajduje się pod linkiem <https://github.com/GuzekAlan/EventSchedulerBot>

Bibliografia

- [1] Guru Staff. *Jak działają boty discordowe*. URL: <https://www.guru.com/blog/how-does-a-discord-bot-work/> (term. wiz. 03.10.2023).
- [2] Werner Geyser. *The Latest Discord Statistics: Servers, Revenue, Data, and More*. URL: <https://influencemarketinghub.com/discord-stats/> (term. wiz. 12.12.2023).
- [3] *Discord.py*. URL: <https://pypi.org/project/discord.py/> (term. wiz. 05.10.2023).
- [4] *Wydanie Python 3.10*. URL: <https://www.python.org/downloads/release/python-3100/> (term. wiz. 01.10.2023).
- [5] *Repozytorium biblioteki discord.py*. URL: <https://github.com/Rapptz/discord.py> (term. wiz. 01.02.2024).
- [6] *Dokumentacja biblioteki Discord.py*. URL: <https://discordpy.readthedocs.io/en/stable/> (term. wiz. 01.10.2023).
- [7] *MongoDB*. URL: <https://www.mongodb.com/>.
- [8] *PyMongo*. URL: <https://pypi.org/project/pymongo/> (term. wiz. 10.10.2023).
- [9] Lakshay Arora. *Pymongo – Python Library to Query a MongoDB Database*. URL: <https://www.analyticsvidhya.com/blog/2020/08/query-a-mongodb-database-using-pymongo/> (term. wiz. 20.11.2023).
- [10] *Top 5 Version Control technologies in 2023*. URL: <https://6sense.com/tech/version-control> (term. wiz. 20.11.2023).
- [11] *GitHub*. URL: <https://github.com/>.
- [12] *Overview of Docker Desktop*. URL: <https://docs.docker.com/desktop/> (term. wiz. 10.12.2023).
- [13] Librarian. *Slash Commands FAQ*. URL: <https://support.discord.com/hc/en-us/articles/1500000368501-Slash-Commands-FAQ> (term. wiz. 10.12.2023).
- [14] Regina Hand. *How do Bot and Client Differ from Each Other?* URL: <https://copyprogramming.com/howto/what-are-the-differences-between-bot-and-client> (term. wiz. 10.12.2023).
- [15] *Dokumentacja Bot UI Kit*. URL: <https://discordpy.readthedocs.io/en/stable/interactions/api.html?highlight=ui%20kit#bot-ui-kit> (term. wiz. 05.12.2023).

- [16] *Dokumentacja klasy discord.Embed*. URL: <https://discordpy.readthedocs.io/en/stable/api.html?#discord.Embed> (term. wiz. 20.12.2023).
- [17] Joe Karlsson. *MongoDB Schema Design Best Practices*. URL: <https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/> (term. wiz. 20.11.2023).
- [18] *Serwis Railway*. URL: <https://docs.railway.app/overview/about-railway> (term. wiz. 22.12.2023).
- [19] *Link umożliwiający dodanie bota*. URL: https://discordapp.com/oauth2/authorize?&client_id=1137322028643393587&scope=bot (term. wiz. 20.12.2023).