Górka Mateusz

Interfejsy Człowiek-Komputer
Projekt

„Bot pozwalający na głosowe zarządzanie zadaniami oraz integrację komunikatora Discord z aplikacją Todoist”

Spis treści

1 Temat i cel projektu

Tematem projektu był: „Bot pozwalający na głosowe zarządzanie zadaniami oraz integrację komunikatora Discord z aplikacją Todoist"

Uszczegóławiając, celem, jaki zaplanowano w ramach wykonania projektu, było stworzenie dwóch botów, chat bota oraz bota głosowego, pozwalających na integrację aplikacji ToDoist.

ToDoist jest aplikacją do zarządzania listami zadań i projektami. Umożliwia tworzenie list zadań, przypinanie ważnych zadań, przydzielanie priorytetów i ustawianie terminów końcowych. Aplikacja jest dostępna na różne platformy, takie jak komputery, telefony komórkowe i tablety, dzięki czemu można korzystać z niej w różnych miejscach i na różnych urządzeniach. ToDoist pozwala również na współpracę z innymi osobami przy projektach, a także synchronizację danych między urządzeniami.

Chat bot (pierwszy z zaimplementowanych botów) jest przeznaczony do wykorzystania na kanałach tekstowych platformy Discord. Discord jest aplikacją do komunikacji głosowej i tekstowej, przeznaczoną głównie dla graczy komputerowych, ale również używaną przez różne grupy społecznościowe oraz bardzo mocno zaadaptowaną w branży gier dla komunikacji zespołów w małych i średnich firmach. Pozwala na tworzenie prywatnych serwerów, na których użytkownicy mogą rozmawiać w grupach i kanałach tekstowych oraz przeprowadzać rozmowy głosowe.

Bot głosowy, przeznaczony jest do uruchomienia lokalnie na urządzeniu użytkownika. Docelowym urządzeniem są komputery z systemem Windows lub Linux posiadające mikrofon. Powinien pozwalać na głosowe wykonywanie poleceń w aplikacji i być możliwie jak najwygodniejszy dla użytkownika.

1.1 Opis aplikacji ToDoist

Aplikacja Todoist jest narzędziem do zarządzania zadaniami i projektami, które umożliwia tworzenie listy zadań, organizowanie ich w projekty oraz ustawianie terminów ich realizacji. Użytkownik może tworzyć nowe zadania, wprowadzając ich tytuł oraz opcjonalnie dodając opis, datę ważności, priorytet oraz projekt, do którego zadanie ma być przypisane. Aplikacja pozwala na współdzielenie zadań z innymi użytkownikami, co ułatwia pracę w zespole.

Zadania można przypisywać do różnych projektów, oznaczać je jako ważne, ustawiać terminy ich realizacji, a także oznaczać jako zakończone. Zadania można przeglądać według projektu, daty ważności, priorytetu lub według statusu (aktywne, zakończone).

Aplikacja pozwala na ustawianie przypomnień o terminach realizacji zadań, które mogą być wysyłane na e-mail lub push-notification. Todoist pozwala na integrację z innymi aplikacjami takimi jak Google Calendar, Slack, Alexa, itp., a projekt ma na celu stworzyć kolejne drogi do integracji i obsługi aplikacji.

2 Wykorzystane technologie oraz ich wykorzystanie

Językiem programowania, na jaki zdecydowano się w trakcie tworzenia projektu, jest język python. Decyzja kierowana była głównie szeroką gamą narzędzi dostępną w tym środowisku. Dla języka python istnieją narzędzia pozwalające między innymi na tworzenie botów dla platformy Discord, rozpoznawania mowy oraz dostępu do bazy danych.

2.1 Rozpoznawanie mowy

Do rozpoznawania mowy wykorzystano **speech_recognition** to biblioteka w Pythonie [6], która pozwala na rozpoznawanie mowy za pomocą komputera. Umożliwia ona nagrywanie dźwięku z mikrofonu lub pliku audio, a następnie przetwarzanie go na tekst. Biblioteka ta wykorzystuje różne usługi rozpoznawania mowy, takie jak Google Speech Recognition, Microsoft Bing Voice Recognition, Sphinx, i wiele innych. **speech_recognition** jest bardzo elastycznym narzędziem, pozwala na wykorzystanie różnych usług rozpoznawania mowy, a także przetwarzanie pliku audio, ustawienie języka, poziomu głośności.

Listing 1: Plik VoiceInput.py

```python
import speech_recognition as sr

class VoiceInput:
    def __init__(self) -> None:
        self.r: sr.Recognizer = sr.Recognizer()
        self.language: str = 'pl-PL'

    def get_text(self, display_thread) -> str | None:
        with sr.Microphone() as source:
            try:
                display_thread.msg = "Słucham"
                audio = self.r.listen(source, phrase_time_limit=6)
                display_thread.msg = "Przetwarzam"
                result = self.r.recognize_google(
                    audio, language=self.language, show_all=True)
                return result
            except Exception as e:
                print(f"Błąd: {e}")
                return None
```

Na listingu 1 przedstawiono kod wykorzystania biblioteki **speech_recognition** mający w celu rejestrację dźwięku z mikrofonu oraz przetworzenie go na tekst. Dzięki fladze **show_all=True** otrzymujemy z API pełną odpowiedź zawierającą alternatywne możliwości tekstu wraz z wagami pewności operacji rozpoznawania mowi.

2.2 Synteza mowy

Wykorzystaną biblioteką, umożliwiającą programistom wykorzystanie syntezy mowy w aplikacjach jest Pyttsx3 [5]. Biblioteka w języku Python oparta na silniku TTS (Text-to-Speech) platformy Microsoft i pozwala na przetwarzanie tekstu na mowę przy użyciu różnych głosów i języków. Pyttsx3 jest łatwa w użyciu i posiada prosty interfejs API,

który pozwala na kontrolowanie prędkości, tonu i głośności mowy oraz na odtwarzanie plików dźwiękowych. Biblioteka ta jest darmowa i open-source.

Użycie biblioteki **pyttsx3** przedstawiono na listingu 2.

Listing 2: Plik VoiceOutput.py

```python
import pyttsx3 as tts

class VoiceOutput:
    def __init__(self) -> None:
        self.engine: tts.Engine = tts.init()
        self.engine.setProperty('rate', 150)
        self.voices = self.engine.getProperty('voices')
        self.engine.setProperty('voice', self.voices[0].id)

    def speak(self, text: str) -> None:
        print(text)
        self.engine.say(text)
        self.engine.runAndWait()
```

2.3 Bot na platformie Discord

W celu implementacji chat bota wykorzystano bibliotekę **discord.py** [1], który jest biblioteką napisaną w języku Python, która umożliwia tworzenie botów na platformie Discord. Pozwala ona na automatyzację różnych czynności, takich jak reagowanie na komendy użytkowników, wysyłanie wiadomości, zarządzanie serwerami itp. Biblioteka jest bardzo rozbudowana i posiada wiele funkcji, dzięki czemu umożliwia tworzenie bardzo zaawansowanych botów.

2.4 Integracja z aplikacją ToDoist

Przed rozpoczęciem implementacji projektu rozważano komunikację z aplikacją ToDoist bezpośrednio poprzez web api, jednak zdecydowano się na skorzystanie ze specjalnie przygotowanego przez twórców aplikacji ToDoist biblioteki ToDoist API SDK [7].

Jest to biblioteka software development kit (SDK) umożliwiająca programistom dostęp do API Todoist. Biblioteka pozwala na automatyzację różnych czynności, takich jak tworzenie, aktualizowanie i usuwanie zadań, zarządzanie projektami i etykietami, a także uzyskiwanie dostępu do danych użytkownika.

2.5 Baza danych

Do realizacji bazy danych wybrałem MongoDB [4]. MongoDB to bazodanowy system zarządzania danymi, który umożliwia przechowywanie i zarządzanie danymi w formacie dokumentów. Jest to tzw. system bazodanowy NoSQL, co oznacza, że nie jest on oparty o relacyjne modele danych. W MongoDB dane są przechowywane w postaci dokumentów JSON-podobnych, zwanych BSON.

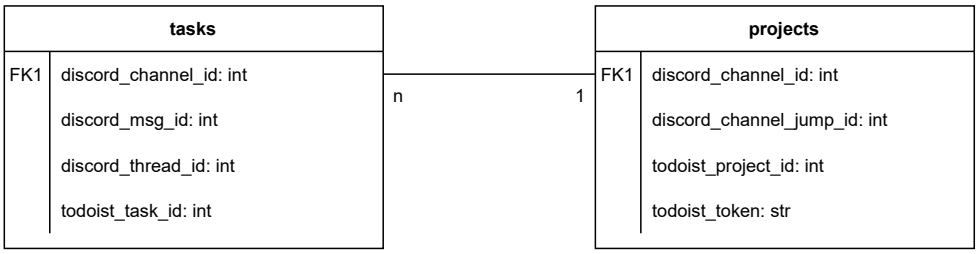
Jednym z głównych zalet MongoDB jest skalowalność. System ten pozwala na rozszerzanie klastra bez konieczności przestojów w działaniu bazy danych. Dzięki czemu MongoDB jest często wykorzystywany w aplikacjach o dużym obciążeniu, takich jak

aplikacje mobilne, aplikacje internetowe itp. MongoDB posiada także wiele narzędzi do zarządzania i analizowania danych, co ułatwia pracę administratorom baz danych.

MongoDB jest dostępny jako open-source oraz jako wersja Enterprise, która oferuje dodatkowe funkcjonalności i wsparcie techniczne.

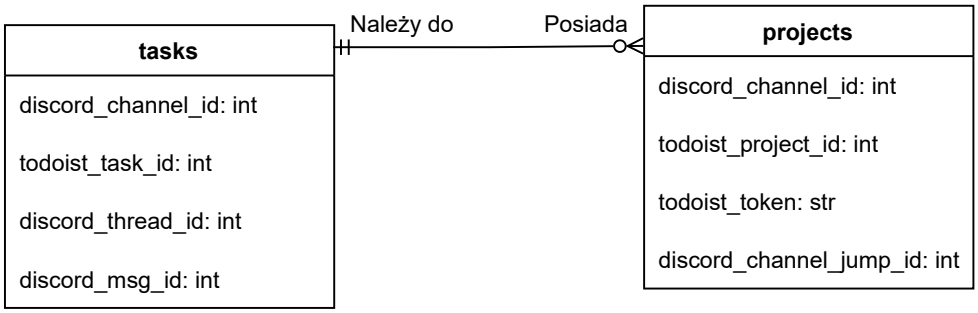
Do wykonywania operacji na bazie danych wykorzystano bibliotekę **`pymongo`** rozwijaną przez społeczność i jest aktywnie wspierana przez MongoDB Inc. Biblioteka udostępnia interfejs API, który pozwala programistom na łatwe połączenie się z bazą danych MongoDB, wykonywanie operacji CRUD (tworzenie, odczytywanie, aktualizowanie, usuwanie) oraz zarządzanie indeksami i agregacjami.

Na schemacie 1 przedstawiono model fizyczny bazy danych, który odnosi się do struktury, w której dane są przechowywane w bazie danych. Natomiast schemat 2 przedstawia model konceptualny, który używany jest do określenia struktury logicznej i treści bazy danych.



Rysunek 1: Model fizyczny

Relacja między kolekcjami **`Projects`** i **`Tasks`** jest relacją jeden do wielu. Oznacza to, że jeden projekt może mieć wiele zadań, ale jedno zadanie jest przypisane do konkretnego projektu. Relacja jest jedynie elementem logiczny, pozwalającym za modelować problem, ponieważ sama baza danych MongoDB jest bazą nierelacyjną.



Rysunek 2: Model konceptualny

2.6 Interfejs web API

W celu obustronnej integracji z aplikacji ToDoist, poza wysyłaniem żądań i zapytań do backendu aplikacji wymagane jest również reagowanie na webhooki, poprzez które aplikacja wysyła powiadomienia o zdarzeniach takich jak dodanie nowych zadań, zmiana stanu zadania. W tym celu wymagane było zaimplementowanie podstawowego interfejsu

web API, do czego wykorzystano bibliotekę **aiohttp**. Biblioteka pozwala na implementacje bardzo lekkiej, z punktu widzenia wymaganych zasobów, obsługi endpointów żądań HTTP [8].

Serwer API stworzony za pomocą **aiohttp** można skonfigurować w kilku liniach kodu. Całość konfiguracji i uruchomienia obsługi serwera webowego została zamieszczona na listingu 3. Web serwer uruchamiany jest jako proces asynchroniczny w ramach naszego chat bota.

Listing 3: Plik WebServer.py

```python
from asyncio import Queue
from aiohttp import web
from discord.ext import tasks
import logging

from src.WebhookServer.WebController import WebController

@tasks.loop()
async def web_server(queue: Queue):
    app = web.Application()
    controller = WebController(queue)

    app.add_routes([
        web.get("/", controller.welcome),
        web.get("/login", controller.login),
        web.get("/access_token", controller.access_token),
        web.post("/payload", controller.payload),
        web.post("/test", controller.test),
    ])

    runner = web.AppRunner(app)
    await runner.setup()

    site = web.TCPSite(runner, host='0.0.0.0', port=5100)
    await site.start()

    logging.info("WebServer started")
```

Implementację konkretnych dwóch endpointów, endpointu logowania umożliwiającego integrację oraz endpointu nasłuchiwania powiadomień (zmian w aplikacji ToDoist) przedstawiono na listingu 4.

Listing 4: Kontroler WebApi. Plik WebController.py

```python
from asyncio import Queue
from aiohttp import web
import requests

from src.config import client_id, scope, client_secret

class WebController:
```

```
8
9       (...)
10
11      async def login(self, request: web.Request):
12          url = f"https://todoist.com/oauth/authorize?\
13              client_id={client_id}&scope={scope}&state={state}"
14          return web.HTTPFound(url)
15
16      async def payload(self, request: web.Request):
17          if request.body_exists:
18              await request.read()
19              json = await request.json()
20              await self._eventQueue.put(json)
21          return web.Response(status=200)
22
23      (...)
```

2.7 System konteneryzacji

W celu uruchomienia aplikacji na środowisku produkcyjnym (w tym przypadku na wirtualnym serwerze) wykorzystałem narzędzie Docker [2] oraz jego rozszerzenie, jakim jest Docker Compose [3].

Docker pozwala na tworzenie i zarządzanie kontenerami. Kontenery to izolowane środowiska, które zawierają wszystko, czego potrzebuje aplikacja do działania – składniki systemowe, biblioteki, konfiguracje itp. Dzięki temu, że aplikacja jest izolowana od reszty systemu, jest ona odporna na zmiany środowiska, co ułatwia testowanie i wdrażanie.

Dzięki czemu za pomocą jednego polecenia ‘docker compose up’ i pliku konfiguracyjnego **`docker_compose.yaml`** (Listing 5) jestem w stanie uruchomić aplikację na nowym urządzeniu, wystarczy, że skopiuję kod aplikacji i zainstaluję środowisko Docker.

Listing 5: Plik docker_compose.yaml

```yaml
version: '3.4'

services:
  app:
    image: todoistintegration
    container_name: todoistintegration
    build:
      context: .
      dockerfile: ./Dockerfile
    ports:
      - 5100:5100
    volumes:
      - /var/log:/var/log
    environment:
      - connection_string=mongodb://${db_root_user}:${db_root_pass}@database0
          :40250/
      - todoist_client_id=${todoist_client_id}
      - todoist_client_secret=${todoist_client_secret}
      - todoist_scope=${todoist_scope}
      - discord_token=${discord_token}
      - logging_file=${logging_file}

  database0:
    image: mongo
    container_name: database0
    command: mongod --port 40250
    ports:
      - 40250:40250
    environment:
      MONGO_INITDB_ROOT_USERNAME: "${db_root_user}"
      MONGO_INITDB_ROOT_PASSWORD: "${db_root_pass}"
```

3 Analiza poleceń w języku naturalnym

Analiza poleceń w języku naturalnym to proces, w którym komputer jest w stanie rozpoznać i zrozumieć polecenia wyrażone w języku naturalnym przez człowieka. Jest to skomplikowana czynność, wymagająca rozpoznawania składni, semantyki, kontekstu i intencji polecenia. Celem analizy poleceń jest przekształcenie polecenia w języku naturalnym na język maszynowy, który jest zrozumiały dla komputera i pozwala na wykonanie określonej czynności. Analiza poleceń jest niezbędnym elementem w rozwoju chat botów i innych aplikacji, które wykorzystują język naturalny do komunikacji z użytkownikami.

3.1 Wybrana metoda przetwarzania języka naturalnego

Wykorzystaną w projekcie metodą analizy języka naturalnego jest skorzystanie z reguł sformowanych za pomocą wyrażeń regularnych. Na potrzeby projektu stworzono specjalną składnię, która za pomocą składni JSON pozwala na zapisywanie uproszczonych wyrażeń regularnych. Przykład takiej konfiguracji został zamieszczony na przykładzie 6.

Listing 6: Przykład reguł do rozpoznawania poleceń

```
{
    "select_&_done": [
        "(zakończ|odznacz|skończono|skończyłem|skończyłem) zadanie [title]",
        "zaznacz zadanie jako skończone [title]"
    ],
    "done": [
        "(zakończ|odznacz|skończono|skończone|gotowe|zamknij)",
        "(zaznacz|odznacz) (zadanie)? za skończone",
        "(zaznacz|odznacz) (zadanie|to)? jako wykonane",
        "zadanie (zakończono|odznaczono|skończono|skończone|gotowe|zamknij)",
        "(skończyłem|skończyłam) (to)? zadanie"
    ]
}
```

Stworzona składnia oferuje kilka operatorów, które dla uproszczenia zostały zamieszczone w tabeli 1. Wykorzystane operatory są następnie przetwarzane przez stworzony przeze mnie program i analizowane pod względem znanych i obsługiwanych operacji w aplikacji.

Tablica 1: Operatory składni reguł

Operator	Funckja
(...)	Grupa
?	0 lub 1 powtórzenie
|	alternatywa
[...]	argument polecenia

Zarówno dla bota głosowego, jak i tekstowego polecenia w języku naturalnym analizowane są w ten sam sposób. Dla bota głosowego przed analizą zachodzi przetworzenie głosu na tekst za pomocą narzędzia **speech_recognition** (opisanego w następnym rozdziale).

Reguły poleceń zostały stworzone tak, aby objąć, jak największy zakres możliwości, w jaki użytkownik może sformować polecenie dla bota.

Listing 7: Przykład reguł do rozpoznawania poleceń

```
...
"select_&_add_comment": [
      "(dodaj|stwórz) komentarz [content] do zadania [title]"
],
"add_comment": [
      "(dodaj|stwórz) komentarz [content]",
      "skomentuj zadanie [content]"
],
"select": [
      "(wybierz|znajdź|zaznacz) (zadanie)? [title]"
],
...
```

Szczególnie ciekawym przykładem zestawu reguł jest operacja dodawania komentarza do zadania. Aby wykonać tę operację, program musi znać treść komentarza oraz znać zadanie, do którego się odnosi. Zatem bot powinien albo przyjmować te dwa argumenty „na wejście" lub potrafić zrozumieć je z kontekstu. W projekcie zrealizowano obie te możliwości.

W przypadku chat bota problem jest dużo prostszy, ponieważ kontekst, do jakiego zadania odnosi się polecenie, wynika wprost z tego, na jakim kanale wydajemy polecenie. Natomiast dla bota głosowego wykorzystano jednakowo metodę z „poznaniem kontekstu" oraz z dwoma argumentami na wejściu.

select_&_add_comment jest identyfikatorem polecenia, w którym wybieramy zadanie, którego nazwa (lub jej część) zawarta jest jako argument nazwany **title** a treść komentarza zawarta jest w argumencie **content**. W ten sposób użytkownik może wykonać całą operację, wydając polecenie: „Dodaj komentarz kup cebulę do zadania zrób zakupy". Pole „kup cebulę" będzie treścią komentarza (**content**), „zrób zakupy" - nazwą zadania (**title**)

Operacje **select** i **add_comment** są identyfikatorami poleceń, dzięki którym możemy przeprowadzć z chat botem następujący dialog:

- Użytkownik: Wybierz zadanie „zrób zakupy"
- Bot: Wybrano zadanie „zrób zakupy"
- Użytkownik: Dodaj komentarz „kup cebulę"
- Bot: Dodano komentarz

Po poleceniu „wybierz" możemy wykonać wiele operacji odnoszących się do wybranego zadania bez powtarzania jego nazwy, w jednej takiej sekwencji możemy na przykład dodać kilka komentarzy, zmodyfikować nazwę zadania oraz zamknąć zadanie.

Sam sposób implementacji zawarty został w klasie **LanguageProcessor** (listing 8) oraz pobocznej klasie **Rule** (listing 9), których zadaniem jest przetworzenie pliku konfiguracji z listingu 6 na obiekty zrozumiałe dla programu i możliwe do wykorzystania w trakcie rozpoznawania poleceń. Na listingu przedstawiającym klasę **Rule** widzimy, że w kilku linijkach przy wykorzystaniu rozbudowanego wyrażenia regularnego można zrealizować bardzo ciekawe operacje.

Listing 8: Plik LanguageProcessor.py

```
class LanguageProcessor:
    def __init__(self) -> None:
        self.rules: List[Rule] = []

    def __init__(self, file_name: str = None) -> None:
        self.rules: List[Rule] = []
        if file_name:
            file = open(file_name, encoding="utf-8")
            dict = json.load(file)
            self.__import_rules(dict)

    def __add_rule(self, command: str, rule: str):
        self.rules.append(Rule(command, rule))

    def __import_rules(self, json: Dict[str, List[str]]):
        for command, rules in json.items():
            for rule in rules:
                self.__add_rule(command, rule)

    def process(self, text: str) -> List[Result]:
        if text is None:
            return list()

        results: Dict[str, List[Result]] = {}
        for rule in self.rules:
            for r in rule.process(text):
                if r.text in results:
                    results[r.text].append(r)
                else:
                    results[r.text] = [r]

        matches = []
        for _, match in results.items():
            match = sorted(match)
            matches.append(match[0])
        return matches
```

Listing 9: Plik Rule.py

```
class Rule:
    def __init__(self, command: str, rule: str) -> None:
        self.command: str = command
        regex = re.sub(r"\[([a-zA-ZĄęŚćŻżŹźŁó0-9]*)\]",
                       r"(?P<\g<1>>.*)",
                       rule)
        self.reg = re.compile(regex)

    def process(self, text: str):
        text = text.lower()
        s = self.reg.finditer(text)
        return [Result(self.command, i.group(), i.groupdict()) for i in s]
```

4 Wspólne funkcje botów

Oba boty pozwalają na następujący zestaw operacji na skonfigurowanej sekcji zadań w aplikacji ToDoist.

- Dodawanie zadania;
- Dodawanie komentarza;
- Modyfikację treści zadania;
- Odznaczanie zadania jako wykonanego, zakończonego;
- Usuwanie zadania;

Zależnie od interfejsu realizacja operacji będzie się różnić zarówno w implementacji, jak i w konstrukcji interfejsu użytkownika.

5 Chat bot

Aplikacja ToDoist nie oferuje zbyt rozbudowanych możliwości, jak chodzi o komunikację osób w zespole, celem chat bota jest możliwość połączenia naturalnego dla pracy zdalnej synchronicznej i asynchronicznej miejsca komunikacji, jakim jest chat tekstowy z aplikacją do zarządzania listą zadań. Bot ma na celu umożliwić zarządzanie listą zadań bez przełączania się pomiędzy oknami w systemie operacyjnym i nie odrywając jednocześnie skupienia od komunikatora i zespołu, z którym użytkownik pracuje nad daną listą zadań.

5.1 Zestaw poleceń

Polecenia dla chat bota możemy formować jednakowo korzystając z konkretnej składni przedstawionej w tabeli 2 oraz za pomocą analizy tekstu naturalnego poprzez wydanie polecenia po znaku zachęty składającego się z dwóch wykrzykników `!!`. Proces analizy tekstu naturalnego dokładnie omówiony został w osobnym rozdziale.

5.2 Tworzenie wątków

Dodatkowo chat bot tworzy wątek poświęcony na komunikację w zespole lub notatki do każdego z synchronizowanych zadań. Dzięki czemu zyskujemy wiele zalet, takich jak:

- Poprawa efektywności komunikacji:

  Chat bot umożliwia szybkie i łatwe przesyłanie informacji między członkami zespołu, co pozytywnie wpływa na efektywność pracy zespołu.

- Uproszczenie zarządzania zadaniami:

  Chat bot umożliwia automatyczne tworzenie notatek do każdego z synchronizowanych zadań, co ułatwia zarządzanie projektami i zadaniami.

- Dostępność informacji:

  Chat bot udostępnia informacje o zadaniach i projektach w czasie rzeczywistym, co pozwala na szybką reakcję na zmiany.

Tablica 2: Składnia poleceń chat bot

Polecenie	Opis
!token ...	Dodaje token dostępu do projektu w aplikacji ToDoist
!project ...	Wybiera synchronizowany projekt (sekcję) z zadaniami
!rmtoken	Usuwa token dostępu
!test	Wyświetla status bota, czy działa, czy został poprawnie uruchomiony
!add ... !dodaj ...	Dodaje zadanie
!delete !usun	Usuwa zadanie
!done !gotowe	Oznacza zadanie za wykonane
!add_comment ... !dodaj_komentarz ...	Dodaje komentarz
!update !zmien	Modyfikuje zadanie
!help !pomoz	Wyświetla listę dostępnych komend

- Oszczędność czasu:

  Chat bot automatyzuje proces tworzenia notatek i komunikacji, co pozwala na oszczędność czasu i skupienie się na innych ważnych zadaniach.

- Łatwość obsługi:

  Chat bot posiada prosty interfejs, który jest łatwy do zrozumienia i obsługi, co pozwala na szybkie rozpoczęcie pracy z nim.

5.3 Realizacja bota tekstowego

Jak wspomniano wcześniej bota zrealizowano za pomocą biblioteki **`discord.py`**. Na listingu zamieściłem fragment programu odpowiedzialny za wykonanie ustalonej akcji na każde z zamieszczonych w tabeli 2 poleceń. W kodzie widzimy metodę **`on_message`**, która wywoływana jest za każdym razem, kiedy na nasłuchiwanym kanale pojawi się nowa wiadomość. Metoda ta sprawdza, czy dana wiadomość rozpoczyna się od odpowiedniego prefiksu zawierającego nazwę oprogramowanej komendy.

Na przykład wiadomość „!dodaj oddaj projekt zaliczeniowy” program zrozumie jako nazwa polecenie (**`prefix`**) „!dodaj” oraz treść (**`content`**) „oddaj projekt zaliczeniowy” i na podstawie nazwy polecenia wykona asynchronicznie odpowiednią akcję (linia 18).

Listing 10: Fragment pliku DiscordClient.py

```python
class DiscordClient(discord.Client):
    def __init__(self) -> None:
        self.on_messages: Dict[str, OnMessageComponent] = {}
        self.on_notification: Dict[str, OnNotificationComponent] = {}
        self.tasks = []
        intents = discord.Intents.all()
        super().__init__(intents=intents)

    async def on_message(self, message: discord.Message):
        if message.author == self.user:
            return

        prefix, _, content = message.content.partition(' ')
        logging.info(f"Processed message | {prefix} | {content}")

        try:
            if prefix in self.on_messages:
                await self.on_messages[prefix].process(message, content)

        except HTTPError as err:
            await message.reply(err)
            raise err

        except Exception as err:
            await message.reply("Błąd")
            await message.reply(err)
            raise err
```

Implementacja każdego z poleceń obsługiwanych przez bota jest osobną klasą dziedziczącą po klasie **OnMessageComponent**, dzięki czemu program ma strukturę komponentów, co sprawia, że dodawanie nowych poleceń jest niezmiernie proste, a oprogramowanie jest otwarte na zmiany i dalszy rozwój.

Listing 11: Fragment pliku DiscordComponent.py

```python
class OnMessageComponent(DiscordComponent):
    @abstractclassmethod
    async def process(self, msg: discord.Message, content: str) -> None:
        raise NotImplementedError()
```

Przykładowym komponentem jest klasa **DonTask** realizująca reakcję na polecenie zakończenia zadania. Metoda **process()** wyszukuje zadanie z bazy danych i składa odpowiednie żądanie do klasy **ApiClient** obsługującej ToDoist SDK, aby zamknąć zadanie. Jeśli zadanie zostanie pomyślnie zamknięte, wyświetlana jest odpowiednia reakcja oraz wysyłana wiadomość – w przeciwnym razie wyświetlana jest wiadomość o błędzie.

Listing 12: Plik DoneTask.py

```python
class DoneTask(OnMessageComponent):
    async def process(self, msg: discord.Message, content: str | None):
        entity: TaskEntity = self.db.find_one(
            TaskEntity, {"discord_thread_id": msg.channel.id})

        todoist: ApiClient = self.todoist.get_client(entity.
            discord_channel_id)

        if todoist.close_task(entity.todoist_task_id):
            await self.report(entity, "Zamknięto zadanie", done_reaction)
        else:
            communicate = "Coś poszło nie tak... Nie udało się zamknąć
                zadania"
            await self.report(entity, communicate)

    async def process_command(self, msg: discord.Message, command: Result):
        await self.process(msg, None)
```

5.4 Integracja z powiadomieniami z aplikacji ToDoist

Dotychczas głównym tematem przewijającym się w opisach było wykonywanie akcji użytkownika, poprzez bota tak, aby zmienić stan listy zadań. Natomiast, aby zrealizować element synchronizacji i integracji, nasz bot tekstowy powinien, także reagować w jakiś sposób na zmiany, które użytkownik wykonał na przykład bezpośrednio w aplikacji ToDoist.

Element ten udało się zrealizować dzięki webhooką zapewnionych przez aplikację ToDoist. Webhooks to technika, która pozwala aplikacji odbierać ogłoszenia o zmianach w określonym zasobie zdalnym bez konieczności ustalania pobierania rutynowych danych. To przydatne narzędzie, które można wykorzystać do natychmiastowego powiadamiania aplikacji o zmianach, które wprowadzono w zasobie. W panelu dostępnym dla deweloperów pracujących z aplikacją ToDoist konfigurujemy adres URL, na który aplikacja będzie wysyłać powiadomienia. W naszym przypadku jest to punkt dostępowy **`payload`** przedstawiony w kodzie na listingu 4.

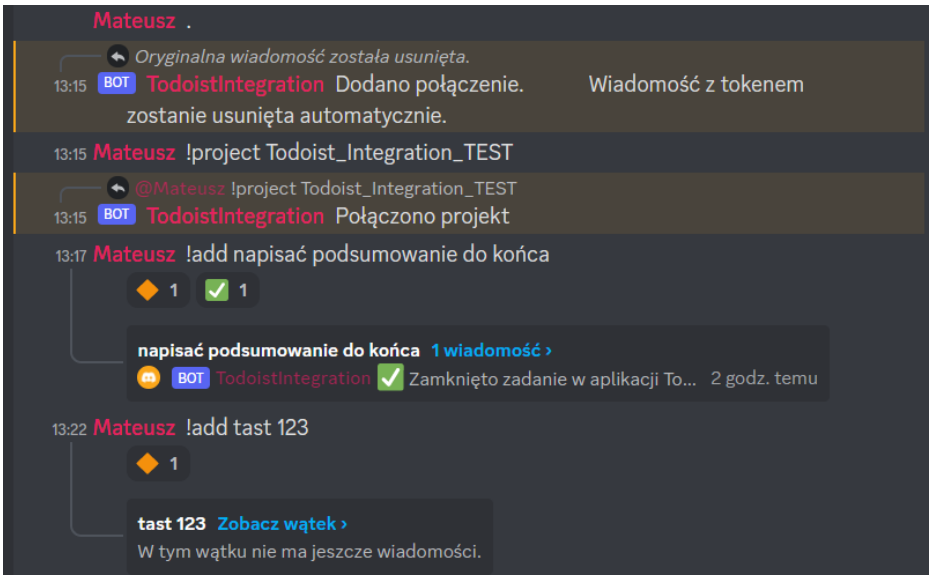
Tablica 3: Identyfikatory zdarzeń webhook

Identyfikator	Opis
item:added	Informuje o dodaniu nowego zadania
item:completed	Informuje o zakończeniu zadaniu
item:deleted	Informuje o usunięciu zadania

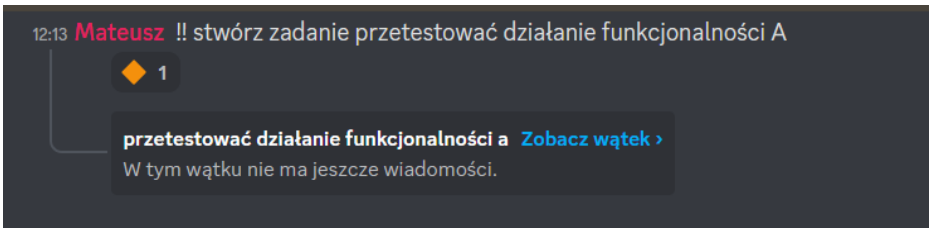
Na przykład po otrzymaniu powiadomienia **`item:completed`** chat bot wyśle nam wiadomość na wątku odpowiadającym dla danego zadania o jego zakończeniu oraz doda emotikon oznaczający zakończenie zadania do wiadomości.

5.5 Przykłady działania

Poniżej przedstawiono kilka zrzutów ekranów przedstawiających działanie bota na platformie Discord. Na zrzutach ekranu widać interakcje z jedynie jednym użytkownikiem, jednak z bota mogą korzystać wszyscy użytkownicy chatu.

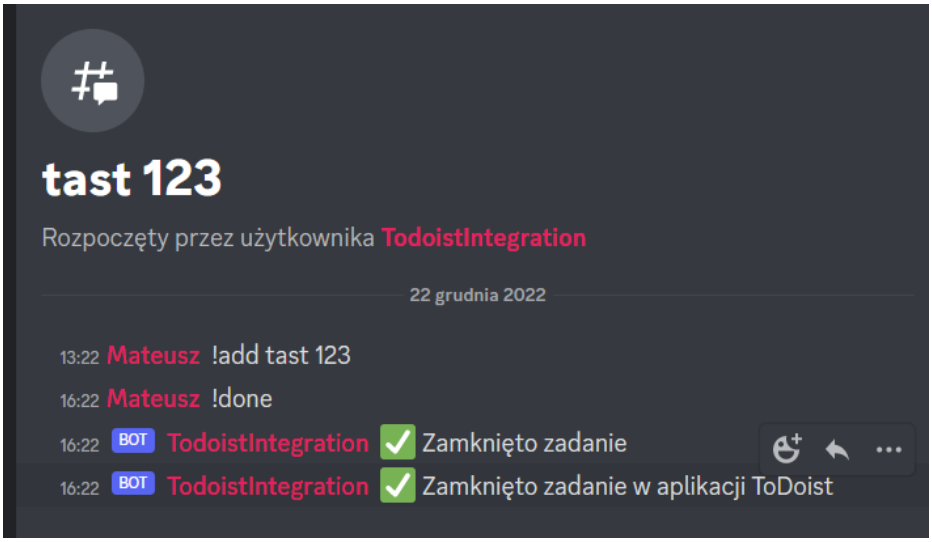


Rysunek 3: Konfiguracja kanału tekstowego oraz dodawanie zadania



Rysunek 4: Przykład wykorzystania NLP poprzez bota tekstowego

Rysunek 5: Dodawanie komentarza



Rysunek 6: Zamykanie zadania

6 Bot głosowy

Celem stworzenia bota głosowego było umożliwienie użytkownikom wykonywania podstawowych operacji bez potrzeby wykorzystywania myszki lub klawiatury. Specyfiką aplikacji zarządzania listą zadań jest wymóg szybkiego i łatwego dostępu do listy, tak aby planowanie zadań na liście zajmowało, jak najmniej czas i wymagało jak najmniej energii. Dzięki botowi głosowemu użytkownik może zarządzać listą zadań głosowo, nawet nie odrywając się od pracy, którą zwykle wykonuje za pomocą klasycznych interfejsów: myszki i klawiatury.

Komunikacja z botem głosowym opiera się na dwóch procesach. Pierwszy z nich rozpoznawanie mowy oznacza, że bot jest w stanie rozpoznawać mowę użytkownika i interpretować jego intencje. Do tego celu wykorzystywane narzędzie **speech_recognition**.

Drugim procesem jest synteza mowy, bot jest w stanie syntetyzować mowę, aby móc się z użytkownikami komunikować. Wszystkie komunikaty zwrotne bota wyświetlane są tekstowo na ekranie oraz wypowiadanie głosowo. Tak jak opisano w poprzednim rozdziale wykorzystano do tego narzędzie **pyttsx3**.

6.1 Niwelacja błędów

W celu niwelacji błędów skorzystano z alternatywnych transkrypcji. Metoda ta polega na przetworzeniu kilku najbardziej prawdopodobnych transkrypcji danego fragmentu mowy na tekst w celu rozpoznania polecenia. Metoda ta zapewnia lepszą jakość wymiany danych między użytkownikami a systemami głosowymi.

Kod przedstawiony na listingu 13 kieruje alternatywne transkrypcje do obiektu klasy **LanguageProcessor** (klasy zajmującej się rozpoznawaniem wzorców w tekście), która dopasowuje do każdego z tekstów operację w aplikacji. Następnie na podstawie prawdopodobieństwa dopasowania oraz ilości rozpoznanych argumentów wybierana jest najbardziej prawdopodobna intencja użytkownika.

Listing 13: Przetworzenie alternatywnych transkrypcji

```
1 def process_alterantives(processor: LanguageProcessor, result: Any)
2     commands: List[Result] = list()
3     for alternative in result["alternative"]:
4         transcript = alternative["transcript"]
5         command: List[Result] = processor.process(transcript)
6         if command is not None:
7             commands.extend(command)
8     return commands
```

6.2 Potwierdzenie wrażliwych operacji

Bot głosowy z powodu na specyfikę interfejsu, która zakłada wysokie prawdopodobieństwo pomyłek w interpretacji intencji użytkownika, wymaga dodatkowego zatwierdzenia dla wrażliwych operacji. Bot realizowany w tym projekcie takim potwierdzeniem obejmuje operację „usuń” z powodu na jej nieodwracalne skutki. Tak, aby minimalizować

możliwe niedogodności z powodu błędnego rozpoznania komend. Przykładowy dialog z chat botem:

- Użytkownik: Wybierz zadania „ABC”
- Bot: Wybrano polecenie ABC
- Użytkownik: Usuń
- Bot: Wykonuję polecenie „usuń”. Czy jesteś pewny, że chcesz usunąć zadanie?
- Użytkownik: Tak jestem pewien
- Bot: Usunięto zadanie

Dodatkowo jak zostało przedstawione na powyższym dialogu, każda z operacji wykonanych przez bota jest sygnalizowana użytkownikowi, tak aby dać komunikat zwrotny zarówno przy przetwarzaniu zakończonym błędem, jak i sukcesem. Programowo zrealizowano to poprzez „odkładanie” ostatniej z intencji wymagających potwierdzenie w specjalnym polu klasy.

6.3 Komendy złożone (wieloparametrowe)

Rozbudowane komendy głosowe, wymagające więcej niż dwóch „argumentów” lub zależne od kontekstu, które nazywamy komendami złożonymi lub wieloparametrowymi, zostały zaimplementowane zarówno w wersji rozbitej na operacje składowe, jak i komendę złożoną pozwalającą odczytać jednocześnie oba składniki wypowiedzi. Przykładami reguł opisującymi takie komendy są reguły przedstawione na listingu 7.

Dla przykładu: operacja „dodaj komentarz” wymaga tekstu komentarza oraz kontekstu lub drugiego argumentu zawierającego informację, do jakiego zadania dodać komentarz, zatem zrealizowano to polecenie na dwa sposoby:

- **`Dodaj komentarz ... do zadania ...`**

  Polecenie to analizuje tekst, a następnie z pierwszego pola odczytuje treść komentarza, a z drugiego nazwę, lub fragment nazwy zadania, do którego ma dodać zadanie.

  Operacja interpretowania wielu argumentów zrealizowana jest dzięki oprogramowaniu przedstawionemu w rozdziale związanym z przetwarzaniem języka naturalnego.

- **`Wybierz zadanie ...`** | **`Dodaj komentarz ...`**

  Znakiem | rozdzieliłem dwie osobne polecenia, które użytkownik może wypowiedzieć w celu wykonania operacji dodawania zadania.

  Do implementacji rozdzielenia operacji, wystarczyło zapamiętywanie kontekstu ostatniego wybranego zadania w ramach serii kilku poleceń głosowych.

7 Zakończenie

Projekt, który polegał na stworzeniu bota, który umożliwia głosowe zarządzanie zadaniami oraz integrację komunikatora Discord z aplikacją Todoist. Bot został oparty o podstawowe elementy natural language processing (NLP) i pozwala użytkownikom na tworzenie, edytowanie i usuwanie zadań za pomocą głosu. Dzięki integracji z Discordem użytkownicy mogą korzystać z bota z poziomu komunikatora, co ułatwia zarządzanie zadaniami.

Bot został opracowany z wykorzystaniem technologii, jakimi są Speech-to-text, Text-to-Speech oraz API Todoist. Bot jest przykładem jak połączenie różnych technologii, pozwala na stworzenie innowacyjnych rozwiązań.

Projekt udało się zrealizować w całości. Z powodu napotkanych trudności niestety nie udało się skorzystać z metod uczenia głębokiego do rozpoznawania poleceń w tekście, ale zastosowana metoda oparta na regułach sprawdza się wyjątkowo dobrze.

Podsumowując, opisywany projekt był ciekawą propozycją dla osób, które chcą mieć dostęp do swojego listy zadań w sposób głosowy, a także dla tych, którzy chcą mieć dostęp do swoich zadań z poziomu komunikatora Discord. Bot oparty na analizie języka naturalnego pozwala na łatwe i szybkie zarządzanie swoimi zadaniami, a integracja z Todoist pozwala na synchronizację listy zadań na wielu urządzeniach i rozbudowane metody zarządzania nimi, obsługi powiadomień itp.

Spis rysunków

Spis tablic

Spis listingów

Literatura

[1] *Dokumentacja Discord.py*.

[2] *Dokumentacja Docker*.

[3] *Dokumentacja Docker Compose*.

[4] *Dokumentacja MongoDb*.

[5] *Dokumentacja Pyttsx3*.

[6] *Dokumentacja Speech Recognitions*.

[7] *Dokumentacja ToDoist API*.

[8] *Hypertext Transfer Protocol*.