



**WYŻSZA SZKOŁA
INFORMATYKI i ZARZĄDZANIA**
z siedzibą w Rzeszowie

KOLEGIUM INFORMATYKI STOSOWANEJ

Kierunek: INFORMATYKA
Specjalność: Programowanie

Mykhailo Husiev
Nr albumu studenta 57063

Projekt i implementacja programu dla hodowli „Złota koza”

Promotor: dr inż. Leszek Gajecki

PRACA DYPLOMOWA INŻYNIERSKA

Rzeszów 2021

Spis treści

Wstęp.....	4
1 Wprowadzenie do problemu.....	5
2 Wybór narzędzi i technologii do implementacji.....	6
2.1 Pycharm CE.....	6
2.2 Python 3.8.....	6
2.3 SQLite3.....	6
2.4 Telegram(Telegram API).....	7
2.5 Telegram-bot-API.....	7
2.6 Aiogram.....	8
3 Projektowanie systemu.....	9
3.1 Analiza wymagań.....	9
3.1 Specyfikacja wymagań.....	9
3.2 Projekt bazy danych.....	10
3.3 Diagram przypadków użycia.....	12
3.4 Projekt interfejsu użytkownika.....	13
4 Implementacja.....	15
4.1 Implementacja programu.....	15
4.2 Opis działania aplikacji.....	26
5 Testowanie.....	29
Zakończenie.....	32
Literatura.....	33
Streszczenie.....	34

Wstęp

W ostatnim czasie programy typu messenger stały się niezwykle popularne, wraz z rosnącą popularnością smartfonów z których korzystają dużo ludzi. Okazało się, że bardzo wygodnie jest używać komunikatorów zamiast stosunkowo drogiej i niewygodnej wiadomości SMS, można wysyłać dowolne pliki, wiadomości są wysyłane szybko i za darmo. Na początku 2021 roku mamy takie trzy główne messengery jako -What'sUp ta Snapchat ta Facebook lite, w zasadzie oni są w jednej funkcjonalności. Jednak na rynku aplikacji mobilnych pojawił się nowy uczestnik -Telegram, opracowany przez zespół założyciela sieci społecznej "VKontakte" Pavel Durov. Od razu zaczął zdobywać popularność i skutecznie konkurować z trzema wspomnianymi wcześniej gigantami tego segmentu, Telegra ma dwie podstawowe zalety:

- Szybkość działania, która była znacznie szybsza niż w przypadku aplikacji What's App i Viber. Zarówno szybkość działania samej aplikacji, jak i szybkość dostarczania wiadomości.
- Bezpieczeństwo -Telegram był pierwszym, który zastosował pełne szyfrowanie wiadomości, jak również nie przechowywał tych wiadomości na zewnętrznych serwerach. Istnieje również możliwość tworzenia tajnych części z autodestrukcją wiadomości po określonym czasie.

Telegram, w przeciwieństwie do WhatsApp, jest komunikatorem opartym na chmurze z płynną synchronizacją. Dzięki temu możesz uzyskać dostęp do wiadomości z kilku urządzeń jednocześnie, w tym tabletów i komputerów, a także udostępniać nieograniczoną liczbę zdjęć, filmów i plików (doc, zip, mp3 itd.) o wielkości do 2 GB każdy. Telegram potrzebuje mniej niż 100 MB na twoim urządzeniu — możesz przechowywać wszystkie swoje multimedia w chmurze bez konieczności usuwania rzeczy — po prostu wyczyść pamięć podręczną, aby zwolnić miejsce.

Dzięki infrastrukturze centrów danych i szyfrowaniu Telegram jest szybszy i znacznie bezpieczniejszy. Co więcej, prywatna komunikacja Telegramem jest bezpłatna i pozostanie bezpłatna na zawsze – bez reklam, bez opłat abonamentowych.

API i kod Telegrama są otwarte, a programiści mogą tworzyć własne aplikacje w Telegramie. Posiadamy również Bot API, platformę dla programistów, która pozwala każdemu z łatwością zbudować specjalistyczne narzędzia dla Telegrama, zintegrować dowolne usługi, a nawet przyjmować płatności od użytkowników z całego świata.

A to tylko wierzchołek góry lodowej[WWW-1, 2020].

Z czasem Telegram jeszcze bardziej rozszerzył swoją funkcjonalność, dodając możliwość tworzenia botów i kanałów przez różne języki programowania. Te ostatnie pozwalają programistom dzielić się informacjami z nieograniczoną liczbą osób. Za pomocą API - Telegram-bot-API, "boty" kontrolowane przez program napisany na przykład w języku Python.

Możemy pracować z botem bezpośrednio, dodawać go do czatów grupowych i kanałów, aby korzystać z określonych funkcji itp. Zazwyczaj praca z botem polega na wysłaniu określonej komendy, po której bot wykonuje algorytm: pisze odpowiedź, wysyła dokument, podaje link do strony internetowej i inne. Ta funkcjonalność comiesięcznie się rozszerza, a boty mogą być teraz używane do płacenia za zakupy, pobierania plików, prowadzenia rzeczowych dialogów, a nawet grania w gry, które są w pełni zaimplementowane w komunikatorze.

Telegram bardzo szybko się rozwija, już wprowadzi monetyzację kontentu w 2021 r., aby zapłacić za infrastrukturę i pensje deweloperów,

Celem niniejszej pracy jest stworzenie bota do Telegrama, który umożliwi dostęp do bazy danych dla hodowców kóz hodowli „Złota koza”(w oryginale „Золотая коза”), a także umieszczenie bota na zewnętrznym serwerze zapewniającym stałą pracę i dostęp do bota.

1 Wprowadzenie do problemu

W niniejszym rozdziale zostanie przedstawiony problem programu oraz czynności związane z tym procesem. Praca z botem powinna być niezwykle prostą ze względu na różny poziom wykształcenia technicznego hodowców, jednocześnie pracując z botem tylko pracownicy z hodowli powinni mieć możliwości zmiany / usunięcia / uzupełnienia bazy danych, także program powinien zajmować jak można mniej pamięci na komórce. Aby bot był łatwy w użyciu, cała praca ogranicza się do dodania go do listy kontaktów Telegrama. Następnie, gdy czat jest otwierany pojawia się dostęp do bota.

Głównym plusem tworzenia w “Telegram” jest to, że użytkownikowi nie trzeba pobierać aplikacji z „PlayMarket”, i bot nie będzie zajmować pamięć na urządzeniu. Właśnie poprzez interfejs czatu użytkownicy mogą korzystać ze wszystkich dostępnych funkcjonalności.

Telegram został wybrany jako platforma dla projektu ze względu na jego uniwersalny charakter: opracowanie aplikacji mobilnej zajęłoby znacznie więcej czasu, ponieważ aplikacja musi być opracowana zarówno dla systemu iOS, jak i Android, a ewentualnie i Windows, podczas gdy Telegram jest już dostępny na wszystkich urządzeniach mobilnych, Linux, Windows, nawet na smart zegarkach. Jednym z najwygodniejszych języków programowania do pisania bota jest Python ze względu na jego prostotę i szeroki zestaw narzędzi do współpracy z TelegramAPI wykorzystano moduł aiogram. Do przechowywania informacji wybrano bazę danych SQLite3. Do hostingu i ciągłej obsługi aplikacji wybrano użyć maszynę wirtualną.

2 Wybór narzędzi i technologii do implementacji

W tym rozdziale zostaną opisane wybrane technologie oraz narzędzia do implementacji projektowanej aplikacji, zostaną również przeanalizowane ich wady oraz zalety.

W wyniku analizy wymagań zarówno funkcjonalnych, jak i нефункциональных (opisane w kolejnym rozdziale) zostały wybrane technologie, które umożliwiają wytwarzanie realizowanej aplikacji.

2.1 Pycharm CE

Pycharm CE – jest darmowym i wygodnym narzędziem do pracy z wieloma różnymi językami programowania. Ma wiele zalet:

1. Inteligentne uzupełnianie na podstawie wcześniej napisanego kodu.
2. Zintegrowana możliwość diagnozowania błędów.
3. Integracja z Git: łatwa kontrola wersji, w tym szybkie rozwiązywanie konfliktów.
4. Dzielenie okien: pomaga uzyskać lepszy obraz kodu podczas programowania.

2.2 Python 3.8

Python jest językiem programowania ogólnego przeznaczenia, skoncentrowanym na zwiększeniu produktywności programistów i czytelności kodu.

Składnia jądra Python jest minimalistyczna. Jednocześnie biblioteka standardowa zawiera dużą liczbę przydatnych integrowanych funkcji. Python wspiera kilka paradygmatów programowania. Należą do nich programowanie strukturalne, obiektowe, funkcyjne i imperatywne.

Główne cechy architektury to dynamiczne typowanie, automatyczne zarządzanie pamięcią, pełna introspekcja, mechanizm obsługi wyjątków, obsługa wielowątkowości oraz wygodne wysokopoziomowe struktury danych.

Kod w python jest zorganizowany w funkcje i klasy, które mogą być łączone w moduły (które z kolei mogą być łączone w pakiety). Python jest aktywnie rozwijającym się językiem programowania, interpreter Pythona ma interaktywny tryb pracy, w którym operatory wprowadzane z klawiatury są natychmiast wykonywane, a wynik jest wyświetlany na ekranie. Tryb ten pozwala na interaktywne testowanie dowolnego fragmentu kodu przed użyciem go w głównym programie, lub po prostu użycie go jako kalkulatora z dużym zestawem funkcji.

Konstrukcja języka Python opiera się na obiektowym modelu programowania. Implementacja OOP w Pythonie jest elegancka, potężna i dobrze zaprojektowana, ale także dość specyficzna w porównaniu z innymi językami zorientowanymi obiektowo.

Siła Pythona leży w jego popularności. W rezultacie pojawienie się ogromnej liczby bibliotek stron trzecich dla prawie wszystkich potrzeb. To właśnie umożliwiło napisanie bota.

2.3 SQLite3

SQLite – system zarządzania relacyjną bazą danych, obsługująca język SQL.

SQLite posiada również API do różnych języków programowania, a mianowicie: ActionScript, Perl, PHP, Ruby, C++, Delphi, Python, Java, Tcl, Visual Basic, platformy .NET i wielu innych; a także interfejs powłokowy. Ma dużo łatwych sposobów do integracji.

Zawartość bazy danych przetrzymywana jest w jednym pliku. Baza SQLite jest utrzymywana na dysku przy użyciu B-drzew. Osobne drzewo jest używane dla każdej z tabel i każdego z indeksów.

Bazy danych zapisywane są jako pliki binarne lub przechowywane w pamięci o dostępie swobodnym. Ich bezpieczeństwo jest oparte na zabezpieczeniach oferowanych przez używany system plików. Istnieje też projekt który umożliwia szyfrowanie baz danych SQLite na bieżąco o nazwie SQLite Encryption Extension (SEE). Ma takie zalety:

1. Łatwość w wykorzystywaniu;
2. Przy tym, że cała baza chroniona w jednym pliku, ona ma: zapytania zagnieżdżone, widoki, klucze obce, transakcje, definiowanie własnych funkcji;
3. Przechowywanie baz danych w pamięci RAM komputera, co znacznie przyspiesza działanie.

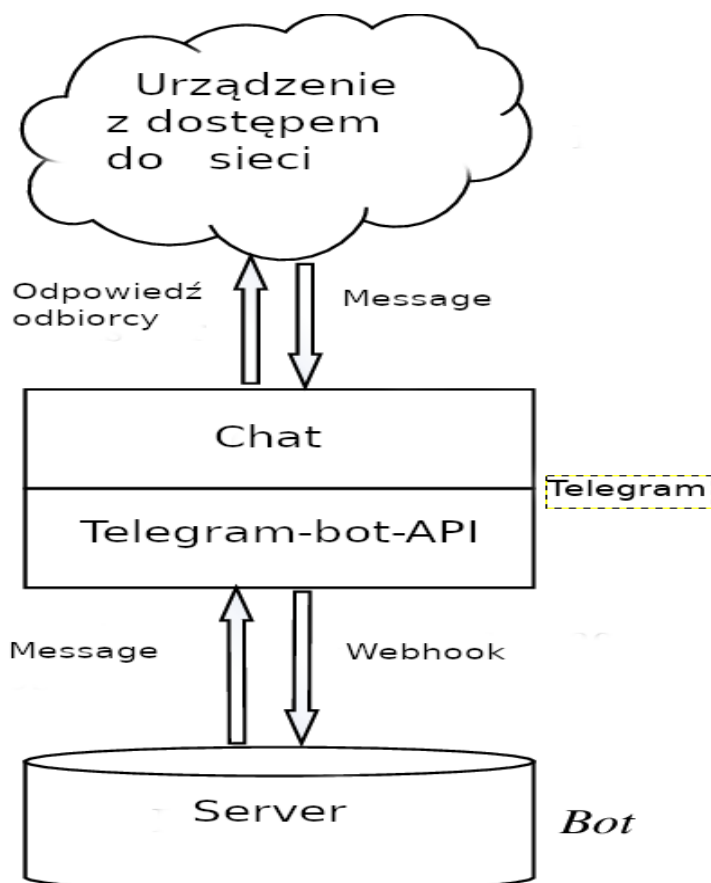
Sqlite ma problem w tym, że jego wykorzystanie w dużych projektach jest nie efektywne, ze względu na ostatni punkt, ale w takim projekcie dla jednej firmy, wykorzystanie sqlite jest dobrym pomysłem.

2.4 Telegram(Telegram API)

Telegram wykorzystuje własny protokół szyfrowania MTProto. MTProto API — inaczej Telegram API) to interfejs API, za którego pośrednictwem aplikacja Telegram komunikuje się z serwerem. Telegram API jest całkowicie otwarty, dzięki czemu każdy programista może napisać swojego klienta komunikatora. Z jakiegoś powodu niewiele osób wie, że boty mogą działać bezpośrednio za pośrednictwem interfejsu API Telegram. Co więcej, w ten sposób można nawet ominąć niektóre ograniczenia, które daje BotAPI.

2.5 Telegram-bot-API

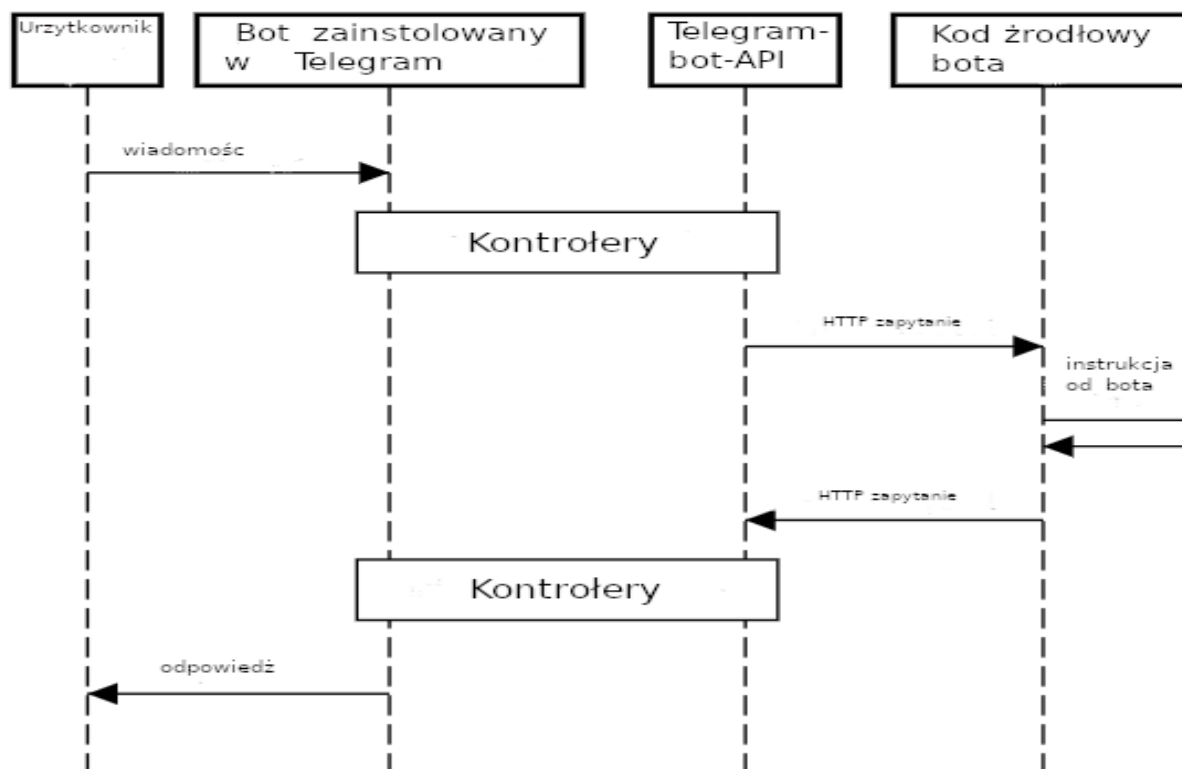
Telegram-bot-API(telebot) – API aplikacji Telegram dla stworzenia botów.



Rys. 1 Ogólna postać Telegram-bot-API
Źródło: opracowanie własne

Poniżej zostanie pokazany diagram sekwencji do takiego obiektu z webhookiem(Rys.2).

Telegram webhook - jest to technologia umożliwiająca śledzenie wydarzeń na czacie w czasie rzeczywistym i wysyłanie informacji o nich na konkretny adres.



Rys. 2 Diagram sekwencji webhook
Źródło: opracowanie własne

2.6 Aiogram

Aiogram – jest frameworkiem dla Telegram-Bot-API, zrealizowany w języku Python 3.7 z wykorzystaniem asyncio, wykorzystuje dekoratory i zawiera przydatne narzędzia programistyczne, na dzień dzisiejszy jest najlepszym frameworkiem dla tworzenia botów w Telegram. Framework został stworzony. Zalety aiogram:

- 1 Asynchroniczność - działania mogą być uruchamiane i kończone niezależnie.
- 2 Dokumentacja aiogram jest jedna najlepszych.
- 3 Automat skończony(FSM – finite state machine).
- 4 Umożliwia wygodnie wykorzystanie webhooków.

Osobno, wyróżnię, że aiogram ma bardzo zorganizowaną społeczność programistów, która zawsze pomoże. Komunikacja odbywa się przez chaty społecznościowe @aiogram oraz @aiogram_ru[WWW-2, 2021].

Automat skończony - jest to modelem obliczeń oparty na hipotetycznym automacie. W jednym momencie tylko jeden stan może być aktywny. Dlatego, aby wykonać jakiegokolwiek czynności, automat musi zmienić swój stan, służy się do logiki, żeby bot mógł zapamiętać informację wprowadzoną przez użytkownika

3 Projektowanie systemu

W tym rozdziale opisany zostanie projekt danego systemu, zostaną zdefiniowane wymagania funkcjonalne oraz нефункционалне, zostaną przeanalizowane najważniejsze przypadki użycia.

3.1 Analiza wymagań

Analiza wymagań jest, to proces co powinien oferować produkt końcowy. Od tego, czy dokładnie, została ona przeprowadzona zależy cały projektu.

Grupą docelową danego programu są osoby zatrudnione w dziedzinie hodowli, którzy dokonują obliczenia związane z tą dziedziną. System powinien być łatwy w użyciu ta oferować zarządzanie danych i zajmować jak można mniej pamięci urządzeniu.

Zmianę w bazie danych powinien mieć tylko admin. Z tego powodu jest zapotrzebowanie we wprowadzeniu dodatkowego poziomu zabezpieczenia takiej sytuacji.

Produkt końcowy powinien być zrealizowany w postaci bota z dostępem do bazy danych.

3.1 Specyfikacja wymagań

Wymagania funkcjonalne są podstawą każdego projektu informatycznego. W celu dostarczenia produktu końcowego niezbędna jest realizacja funkcji opisanych w tym obszarze. Wymagania funkcjonalne opisują działania, które mogą być wykonywane przez użytkowników danego systemu. Wymagania funkcjonalne mogą obejmować obliczenia, szczegóły techniczne, manipulację i przetwarzanie danych oraz inne specyficzne funkcje, które określają, co system ma osiągnąć. Wymagania funkcjonalne kierują architekturą aplikacji systemu, podczas gdy wymagania нефункционалне kierują architekturą techniczną systemu. W wyniku analizy została wydedukowana następująca lista wymagań funkcjonalnych:

Tab. 1. Wymagania funkcjonalne

ID	Opis wymagania	Priorytet
1	Rejestracja administratora, oraz dostęp ograniczony	Wysoki
2	Wprowadzanie danych do bazy danych	Wysoki
3	Przeglądanie historii zmian danych	Wysoki
4	Przeglądanie statystyki danych	Średni

Źródło: opracowanie własne

Wymagania niefunkcjonalne opisują wymagania związane z częścią wizualną, z bezpieczeństwem aplikacji, ograniczeniami oraz wydajnością.

Tab. 2. Wymagania niefunkcjonalne

ID	Opis wymagania	Priorytet
1	Program powinien być minimalistyczny oraz nie zajmować dużo pamięci	Wysoki
2	Token autoryzacyjny powinien zawierać jednego administratora, przypadkowa osoba nie powinna korzystać z bota	Wysoki
3	System powinien być dostępny na Android oraz IOS	Średni

Źródło: opracowanie własne

3.2 Projekt bazy danych

Kolejnym etapem projektowania systemu jest to projektowanie struktury bazy danych. Po analizie wszystkich wymagań zarówno funkcjonalnych jak i niefunkcjonalnych, została zaprojektowana struktura bazy danych, która będzie niezbędna do osiągnięcia celu.

Podczas projektowania bazy danych zostały uwzględnione wymagania zarówno funkcjonalne jak i niefunkcjonalne. Zostało zaprojektowano 3 tablice bazy o następującej strukturze, oraz spis kategorii dochodów.

Główna tabela „**dochod**”, ta tabela reprezentując dochody hodowli, ma pola:

- id - primary key.
- amount – suma dochodów.
- created datetime – datę dodawania dochodu.
- category_codename – id kategorii, foreign_key do category(codename).
- raw_message – na sam wypadek pobieramy całą wiadomość wprowadzoną przez użytkownika.

```

create table dochod_dzieny(
    codename varchar(255) primary key,
    dochod_dzien integer
);

create table category(
    codename varchar(255) primary key,
    name varchar(255),
    is_dzieny_dochod boolean,
    aliases text
);

create table dochod(
    id integer primary key,
    amount integer,
    created datetime,
    category_codename integer,
    raw_text text,
    FOREIGN KEY(category_codename) REFERENCES category(codename)
);

```

Rys. 3 struktura bazy danych
Źródło: opracowanie własne

```

insert into category (codename, name, is_dzieny_dochod, aliases)
values
    ("products", "produkty", true, "toward, twr"),
    ("milk", "mleko", true, "molo, mlk"),
    ("meat", "mieso", true, "miaso, mns"),
    ("cheese", "ser", true, "syr, cziz"),
    ("animal", "bydlo", false, "kozy, deti"),
    ("other", "inne", true, "");

```

Rys. 4 kategorie bazy danych
Źródło: opracowanie własne

Kategorie jest wprowadzone w postaci słów, a nie cyfr, dlatego że z słowami pracować wygodniej niż z 1,2,3.

INSERT INTO używamy by wprowadzić/ usunąć rekord danych/ dane:

Codename – jest identyfikatorem.

Name – nazwa po polsku.

True/false – czy liczy wprowadzony dochód do wszystkich dochodów dzisiejszych.

Głównym plusem wykorzystania SQLite3 jest to, że on jest wbudowany w python oraz jest prosty w implementacji, tworzymy plik z bazy danych dbbot.db oraz robimy inicjalizację danych z .sql pliku

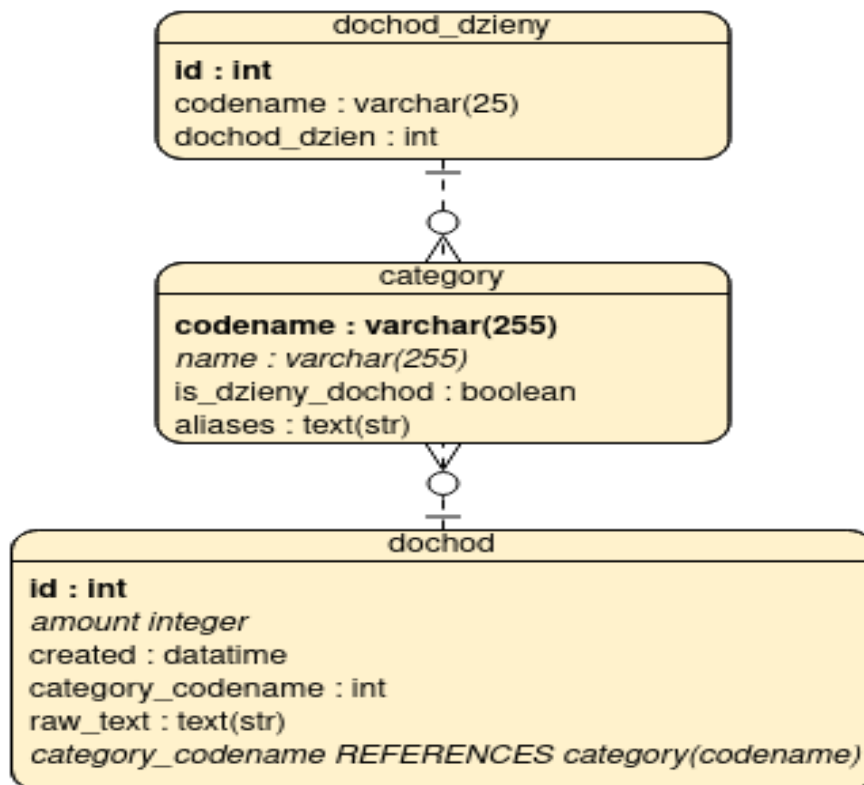
```

conn = sqlite3.connect('dbbot.db')
cursor = conn.cursor()

def _init_db():
    with open("createdb.sql", "r") as f:
        sql = f.read()
        cursor.executescript(sql)
        conn.commit()

```

Rys. 5 Połączenie z SQLite3
Źródło: opracowanie własne



Rys. 6 Diagram ERD
Źródło: opracowanie własne

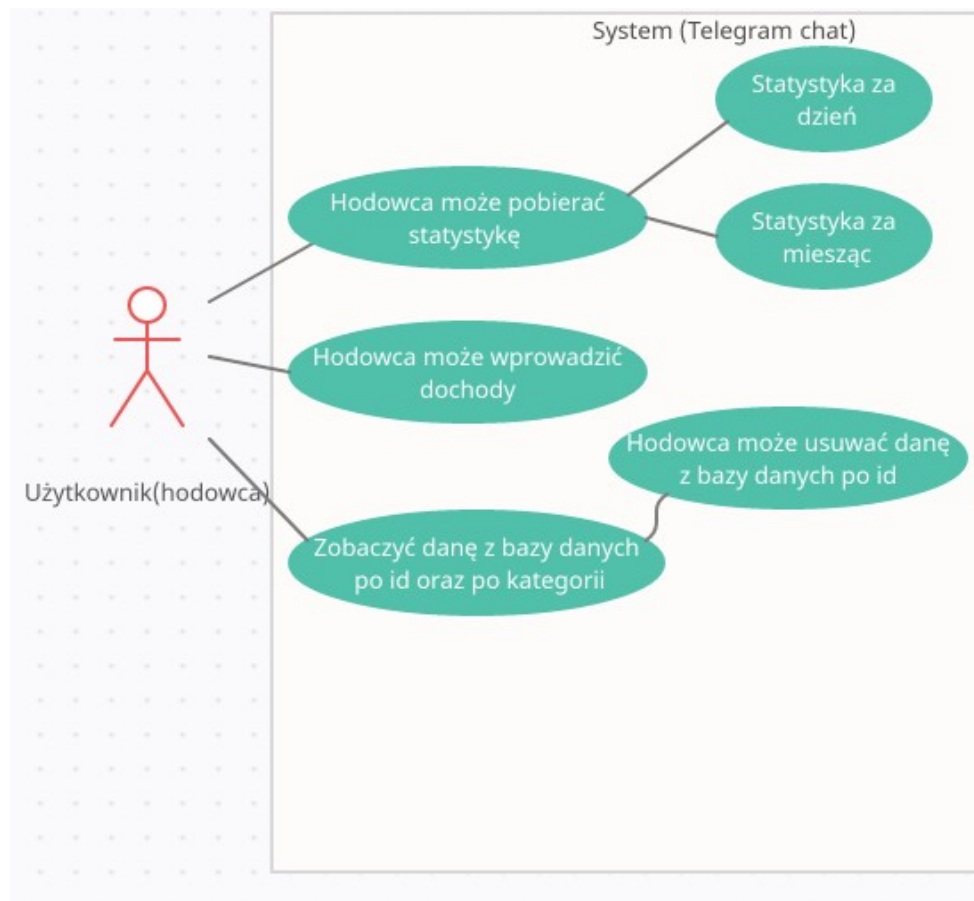
3.3 Diagram przypadków użycia

W następnej kolejności projektowania aplikacji został opracowany diagram przypadków użyciu, który jest bardzo ważnym elementem procesu projektowania aplikacji, ponieważ wizualizuje wszystkie przypadki, które muszą być uwzględnione podczas kolejnych etapów projektowania oraz podczas procesu implementacji rozwiązania

Użytkownik projektu(administrator) – jest to hodowca, który dokonywać żadnych modyfikacji w ramach danego projektu. Taki użytkownik ma uprawnienia:

- Dodawać dane o dochodach przez chat.
- Pobierać statystykę dzienną oraz miesięczną.
- Sprawdzać oraz usuwać dane z bazy danych.

Pomiędzy użytkownikiem oraz administratorem w konieczności nie ma różnicy, ponieważ bot będzie reagował tylko na wiadomości z konkretnego id użytkownika.



Rys. 7 Diagram przypadków użycia
Źródło: opracowanie własne

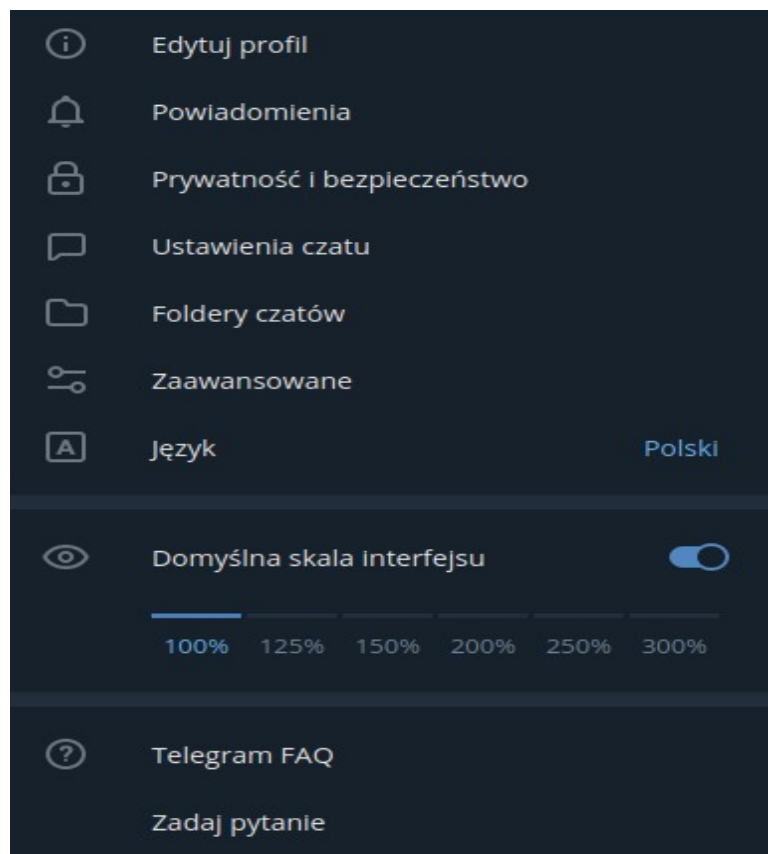
3.4 Projekt interfejsu użytkownika

Ze względu na to, że program jest integrowany do Telegram, interfejs jest prawie już realizowany, interfejs chatu może być zmieniony o dowolnych ustawieniach w dowolnym koloru przez ustawienia użytkownika.



Rys. 8 Ustawienia graficzne czatu
Źródło: opracowanie własne

Interfejs aplikacji Telegram jest minimalistyczny co spełnia jedną z wymagań do programu, praca c programem przez chat będzie łatwa, ustawie



Rys. 9 ustawienia graficzne czatu
Źródło: opracowanie własne

Praca z botem zostanie realizowana w postaci napisania komend do czatu.

Pomiędzy tym, że użytkownik może korzystać z unikalnej funkcjonalności bota, on może zarządzać dane oraz pliki, które on chce przez standardową funkcjonalnością Telegram chatu.

Wygląd bota można dostosować w BotFather: menu /mybots → Edit Bot. Tam można zmienić:

- Nazwę bota.
- Opis (Description) - To tekst, który użytkownicy zobaczą na początku okna dialogowego bota pod nagłówkiem.
- Informacje (About) - to tekst, który będzie widoczny w profilu bota.
- Avatar - Awatary botów, w przeciwieństwie do awatarów użytkowników i czatów, nie mogą być animowane, tylko w postaci zdjęcia.
- Polecenia - oznacza to podpowiedź poleceń w bocie. Więcej o poleceniach poniżej.
- Inline Placeholder - informacje o trybie inline można znaleźć poniżej.

4 Implementacja

W tym rozdziale zostanie opisany proces implementacji programu.

4.1 Implementacja programu

Aby osiągnąć te cele, należało implementować następujące zadania:

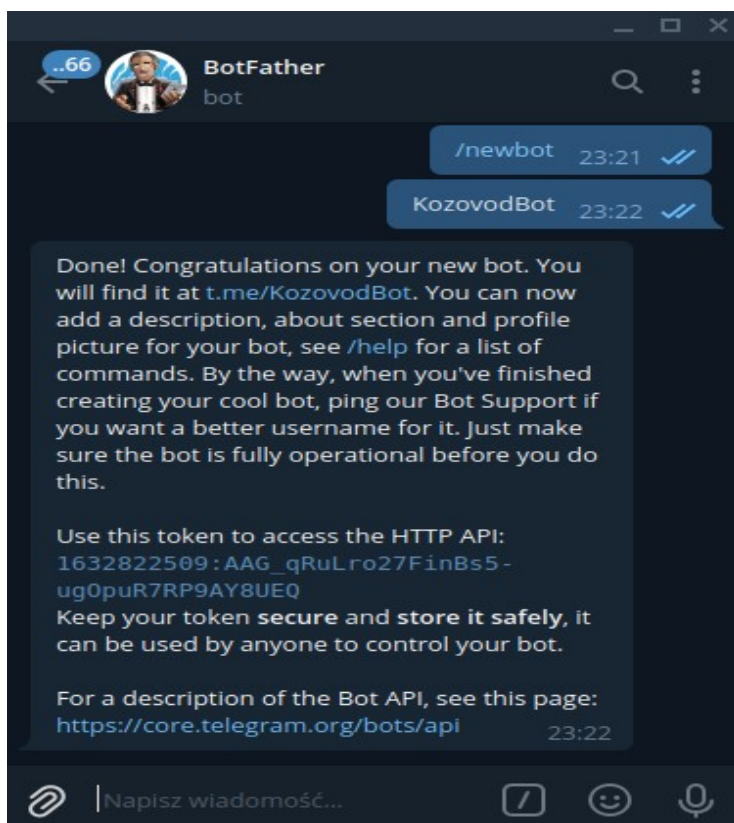
1. Należy wybrać język programowania do realizacji bota;
2. Stworzyć bazę danych do zarządzania dochodów;
3. Podłączyć bazę do bota;
4. Umieszczać bota na maszynie wirtualnej jaka będzie znajdować w hodowli kóz;

Implementacja programu zostanie wykonana zgodnie z wymagami z punktu 3.1 pracy dyplomowej.

Pierwszym krokiem w implementacji programu będzie otrzymanie specjalnego tokena dla bota, żeby otrzymać ten token, potrzebne napisać do „ojca” wszystkich botów w Telegram – BotFather. Przy pomocy komendy /newbot stworzymy bota ta wprowadzamy nazwę bota.

Każdy użytkownik, Bot, Grupa, kanał w Telegramie ma swój własny identyfikator. Czaty w kodzie bota powinny być rozróżniane według identyfikatora, ponieważ nigdy się nie zmienia. W tokenie bota pierwszą częścią jest jego identyfikator.

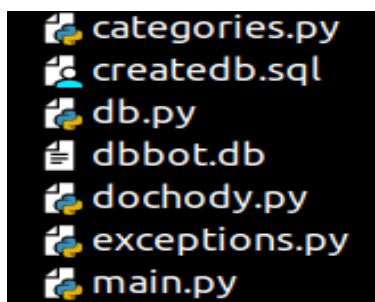
Na przykład my mamy token "1632822509:AAG_qRuLro27FinBs5-ugOpuR7RP9AY8UEQ" — który należy do bota o identyfikatorze 1632822509[WWW-9, 2021].



Rys. 10 Generacja tokenu
Źródło: opracowanie własne

Następnym krokiem będzie inicjalizacja bota w Pycharm przy pomocy tokena bota, oraz wprowadzenie id administratora lub kilku administratorów, żeby nikt inny nie mógł napisać coś

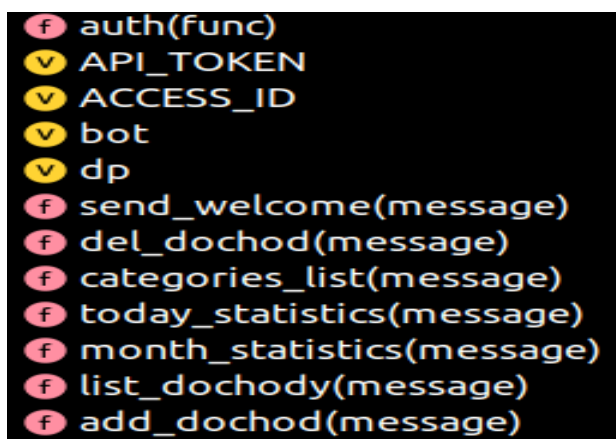
swoje do bota. Struktura programu zawiera cztery główne moduły, oraz kilka wspomagających o następującej strukturze.



Rys. 11 Struktura programu w postaci plików
Źródło: opracowanie własne

1. Moduł main.py

To jest moduł, który zawiera kontrolery programu oraz inicjalizację bota, kontrolery – to jest klej pomiędzy biznes logiką oraz interfejsem użytkownika, przez ten moduł użytkownik może pobierać statystykę, dodawać dane do bazy, usuwać dane. Kontroler powinien być krótko zrealizowany oraz nie powinien się zawierać biznes logiki. Zawiera główny kontroler **add_dochod** który otrzymuje tekst wprowadzony przez chat użytkownika oraz dodaje dane do biznes logiki w module dochod.py, oraz kilka innych kontrolerów do wyświetlania statystyki. Zawiera następującą strukturę:



Rys. 12 Struktura main.py
Źródło: opracowanie własne

Najpierw trzeba zrobić inicjalizację bota, na to służy API_TOKEN, ACCESS_ID, bot, dp
bot – przyjmuje API_TOKEN, dp – inicjalizacja Dispatcher(bot). Zrobione zgodnie z dokumentacją

```
API_TOKEN = '1632822509:AAG_qRuLro27FInBs5-ug0puR7RP9AY8UEQ'  
ACCESS_ID = 573332887  
  
bot = Bot(token=API_TOKEN)  
dp = Dispatcher(bot)
```

Rys. 13 Inicjalizacja bota
Źródło: opracowanie własne

Kontroler **send_welcome** przyjmuje komendę /start wprowadzoną przez administratora oraz dokonuje inicjalizację bota. Kontroler jest realizowany zgodnie z dokumentacją aiogram[WWW-4, 2021].

```
@dp.message_handler(commands=['start', 'help'])
async def send_welcome(message: types.Message):
    await message.answer(
        "Bot Złota Koza\n\n"
        "Dodajemy dochód: napisz dochód oraz kategorię\n\n"
        "Statystyka za dzisiaj: /today\n\n"
        "Statystyka za miesiąc: /month\n\n"
        "Ostatnie dochody: /dochody\n\n"
        "categories: /categories")
```

Rys. 14 Kontroler send_welcome
Źródło: opracowanie własne

Kontroler **del_dochod** przyjmuje komendę /del, która umożliwia usunięcie danych[WWW-5, 2021]. **Await** – jest słowem kluczowym do tworzenia funkcji asynchronicznej. Taka konstrukcja oznacza, że program będzie działał, dopóki nie znajdzie await-wyrażenia, a następnie wywoła funkcję i zawiesi wykonanie, dopóki działanie wywołanej funkcji nie zostanie zakończone

```
@dp.message_handler(lambda message: message.text.startswith('/del'))
async def del_dochod(message: types.Message):
    row_id = int(message.text[4:])
    dochody.delete_dochod(row_id)
    answer_message = "Usunięto"
    await message.answer(answer_message)
```

Rys. 15 Kontroler add_dochod
Źródło: opracowanie własne

Kontroler **categories_list** przyjmuje komendę /categories która wyświetli spis dostępnych kategorii[WWW-6, 2021].

```
@dp.message_handler(commands=['categories'])
async def categories_list(message: types.Message):
    categories = Categories().get_all_categories()
    answer_message = "Kategorie dochodów:\n\n* " + \
        (" \n* ".join([c.name + ' (' + ", ".join(c.aliases) + ')'] for c in categories)))
    await message.answer(answer_message)
```

Rys. 16 Kontroler categories_list
Źródło: opracowanie własne

Kontrolery **today_statistics/month_statistics** przyjmują komendę /today oraz /month, która wyświetli dochody dzisiejsze.

```

@dp.message_handler(commands=['today'])
async def today_statistics(message: types.Message):
    answer_message = dochody.get_today_statistics()
    await message.answer(answer_message)

@dp.message_handler(commands=['month'])
async def month_statistics(message: types.Message):
    answer_message = dochody.get_month_statistics()
    await message.answer(answer_message)

```

Rys. 17 Kontrolery today_statistics/month_statistics
Źródło: opracowanie własne

Kontroler **list_dochody** przyjmuje komendę /dochody do wyświetlania danych z bazy danych.

```

@dp.message_handler(commands=['dochody'])
async def list_dochody(message: types.Message):
    last_dochody = dochody.last()
    if not last_dochody:
        await message.answer("Nie masz jeszcze dochodów")
        return

    last_dochody_rows = [
        f"{dochod.amount} zł. na {dochod.category_name} - naćisni ",
        f"/del{dochod.id} żeby usunąć"
        for dochod in last_dochody]
    answer_message = "Ostatnie dochody:\n\n* " + "\n\n* " \
        .join(last_dochody_rows)
    await message.answer(answer_message)

```

Rys. 18 Kontroler list_dochody
Źródło: opracowanie własne

Kontroler **add_dochod** przyjmuje tekst wprowadzony do chatu przez użytkownika, służy do komunikacji z biznes logiką modułu dochody.py[WWW-8, 2021].

```

@dp.message_handler()
async def add_dochod(message: types.Message):
    try:
        dochod = dochody.add_dochod(message.text)
    except exceptions.NotCorrectMessage as e:
        await message.answer(str(e))
        return

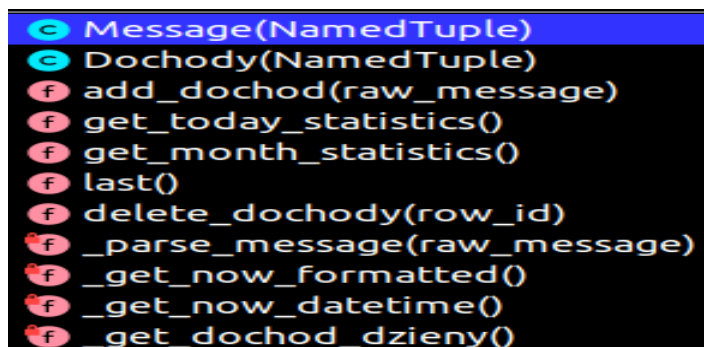
    answer_message = (
        f"Dodajemy dochod {dochod.amount} zł na {dochod.category_name}.\n\n"
        f"{dochody.get_today_statistics()}")
    await message.answer(answer_message)

```

Rys. 19 Kontroler add_dochod
Źródło: opracowanie własne

2. Moduł dochody.py

Ten moduł bezpośrednio dodaje nowe dochody do bazy, zawiera funkcjonalność do pracy z biznes logiką programu, ma dwie klasy do definicji typu wiadomości oraz typu dochodu, zawiera następującą strukturę:



```
class Message(NamedTuple)
class Dochod(NamedTuple)
def add_dochod(raw_message)
def get_today_statistics()
def get_month_statistics()
def last()
def delete_dochody(row_id)
def _parse_message(raw_message)
def _get_now_formatted()
def _get_now_datetime()
def _get_dochod_dzieny()
```

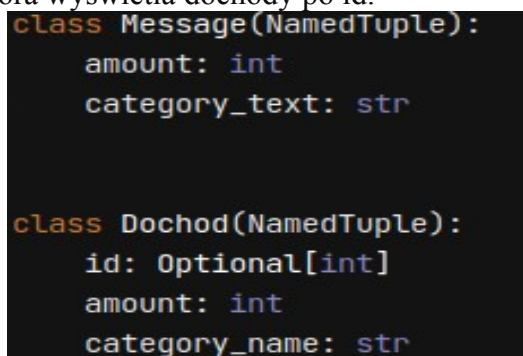
Rys. 20 Struktura dochod.py
Źródło: opracowanie własne

Zawiera 2 klasy oraz 9 funkcję, 4 z których są niepubliczne funkcję z biznes logiką:

Klasa **Message** – jest prostą strukturą NamedTuple która przyjmuje ilość dochodu oraz tekst wiadomości;

NamedTuple jest wykorzystywany ponieważ jest efektywny ze względu na rozmiar takiego oraz on jest dobrze czytelny.

Klasa **Dochod** – jest klasą która wyświetla dochody po id.



```
class Message(NamedTuple):
    amount: int
    category_text: str

class Dochod(NamedTuple):
    id: Optional[int]
    amount: int
    category_name: str
```

Rys. 21 Klasa Message oraz Dochod
Źródło: opracowanie własne

Funkcja **add_dochod** – biznes logika bota, analizuje wprowadzony tekst przez użytkownika, dzieli się dochodem na kategorie, komponuje wiadomość do bazy danych „dochod”.

```

def add_dochod(raw_message: str) -> Dochod:
    parsed_message = _parse_message(raw_message)
    category = Categories().get_category(
        parsed_message.category_text)
    db.insert("dochod", {
        "amount": parsed_message.amount,
        "created": _get_now_formatted(),
        "category_codename": category.codename,
        "raw_text": raw_message
    })
    return Dochod(id=None,
                  amount=parsed_message.amount,
                  category_name=category.name)

```

Rys. 22 Struktura dochod.py
Źródło: opracowanie własne

Funkcji `get_today/month_statistics` – wyświetla dochody za dzień oraz za miesiąc.

```

def get_today_statistics() -> str:
    cursor = db.get_cursor()
    cursor.execute("select sum(amount) "
                  "from dochod where date(created)=date('now', 'localtime')")
    result = cursor.fetchone()
    if not result[0]:
        return "Dzisiaj nie masz dochodów"
    all_today_dochody = result[0]
    cursor.execute("select sum(amount) "
                  "from dochod where date(created)=date('now', 'localtime') "
                  "and category_codename in (select codename "
                  "from category where is_dzieny_dochod=true)")
    result = cursor.fetchone()
    base_today_dochody = result[0] if result[0] else 0
    return (f"Dochód za dzisiaj:\n"
            f"razem - {all_today_dochody} zł.\n"
            f"minimalne - {base_today_dochody} zł. z {_get_dochod_dzieny()} zł.\n"
            f"za miesiąc: /month")

```

Rys. 23 Funkcja `get_today_statistics`
Źródło: opracowanie własne

```

def get_month_statistics() -> str:
    now = _get_now_datetime()
    first_day_of_month = f'{now.year:04d}-{now.month:02d}-01'
    cursor = db.get_cursor()
    cursor.execute(f"select sum(amount) "
                  f"from dochod where date(created) >= '{first_day_of_month}'")
    result = cursor.fetchone()
    if not result[0]:
        return "Wprawie nie masz dochodów"
    all_today_dochody = result[0]
    cursor.execute(f"select sum(amount) "
                  f"from dochod where date(created) >= '{first_day_of_month}' "
                  f"and category_codename in (select codename "
                  f"from category where is_dzieny_dochod=true)")
    result = cursor.fetchone()
    base_today_dochody = result[0] if result[0] else 0
    return (f"Dochód za miesiąc:\n"
            f"Razem - {all_today_dochody} zł.\n"
            f"Minimalne - {base_today_dochody} zł. z "
            f"{now.day * _get_dochod_dzieny()} zł.")

```

Rys. 24 Funkcja `get_month_statistics`
Źródło: opracowanie własne

Funkcja **last** – wraca kilka ostatnich dochodów wprowadzonych przez użytkownika.

```
def last() -> List[Dochod]:
    cursor = db.get_cursor()
    cursor.execute(
        "select e.id, e.amount, c.name "
        "from dochod e left join category c "
        "on c.codename=e.category_codename "
        "order by created desc limit 10")
    rows = cursor.fetchall()
    last_dochody = [Dochod(id=row[0], amount=row[1], category_name=row[2]) for row in rows]
    return last_dochody
```

Rys. 25 Funkcja get_month_statistics
Źródło: opracowanie własne

Funkcja **delete_dochody** – usuwa dochody po id.

Funkcja **_parse_message** - wyrażenie regularne do analizy tekstu wprowadzonego przez użytkownika.

Wyrażenie regularne (ang. regular expression, w skrócie regex lub regexp) – wzorec opisujący łańcuch symboli. Teoria wyrażeń regularnych jest związana z teorią języków regularnych. Wyrażenia regularne mogą określać zbiór pasujących łańcuchów, jak również wyszczególniać istotne części łańcucha[WWW-3, 2020].

```
def _parse_message(raw_message: str) -> Message:
    regexp_result = re.match(r"([\d ]+) (.*)", raw_message)
    if not regexp_result or not regexp_result.group(0) \
        or not regexp_result.group(1) or not regexp_result.group(2):
        raise exceptions.NotCorrectMessage(
            "Nie rozumiem, napisz"
            "na przykład:\n150 mleko")

    amount = regexp_result.group(1).replace(" ", "")
    category_text = regexp_result.group(2).strip().lower()
    return Message(amount=amount, category_text=category_text)
```

Rys. 26 Funkcja get_month_statistics
Źródło: opracowanie własne

Ze względu na rozmiar następnych funkcji, oni zostali umieszczone na jednym obrazku(Rys.)

Funkcja **_get_now_formatted** - wraca dzisiejszą datę.

Funkcja **_get_now_datetime** – wraca czas.

Funkcja **_get_dochod_dzieny** – wraca obowiązkową liczbę dochodu.


```

def _get_now_formatted() -> str:
    return _get_now_datetime().strftime("%Y-%m-%d %H:%M:%S")

def _get_now_datetime() -> datetime.datetime:
    tz = pytz.timezone("Poland")
    now = datetime.datetime.now(tz)
    return now

def _get_dochod_dzieny() -> int:
    return db.fetchall("dochod_dzieny", ["dochod_dzien"])[0]["dochod_dzien"]

```

Rys. 27 Funkcja get_month_statistics

Źródło: opracowanie własne

3. Moduł db.py

To jest moduł dla pracy z bazą danych. **Cursor** - Udostępnia podzbiór danych zdefiniowanych przez zapytanie, dane pobierane są do pamięci i cursor jest do niej wskaźnikiem. Otwarcie kursora to pobranie części bieżących danych, co oznacza, że zmiana, dodanie lub usunięcie danych po jego otwarciu nie będzie miało odzwierciedlenia w zbiorze który zwrócił cursor.

Funkcji modułu db.py zawierają dwie wcześniej zdefiniowane klasę o nazwie Message oraz Dochody.

Na podstawie danych wchodzących NamedTuple, zapisuję ilość dochodów hodowli do bazy;

Funkcja **get_cursor** tworzy kursora, za pomocą kursora możemy dokonać wybór w bazie danych;

Funkcja **_init_db** inicjalizacje bazy danych. Funkcja check_db_exists, sprawdza czy baza jest włączona, jeżeli nie – robię inicjalizację;

Funkcja **get_cursor** – umożliwia dostęp do kursora, który jest potrzebny manipulacji w bazie danych.

```

def get_cursor():
    return cursor

def _init_db():
    with open("createdb.sql", "r") as f:
        sql = f.read()
    cursor.executescript(sql)
    conn.commit()

def check_db_exists():
    cursor.execute("SELECT name FROM sqlite_master "
                  "WHERE type='table' AND name='dochod'")
    table_exists = cursor.fetchall()
    if table_exists:
        return
    _init_db()

check_db_exists()

```

Rys. 28 Funkcja get_month_statistics
Źródło: opracowanie własne

```

def insert(table: str, column_values: Dict):
    columns = ', '.join(column_values.keys())
    values = [tuple(column_values.values())]
    placeholders = ', '.join("?" * len(column_values.keys()))
    cursor.executemany(
        f"INSERT INTO {table} "
        f"({columns}) "
        f"VALUES ({placeholders})",
        values)
    conn.commit()

```

Rys. 29 Funkcja insert
Źródło: opracowanie własne

```
def fetchall(table: str, columns: List[str]) -> List[Dict[str, Any]]:
    columns_joined = ", ".join(columns)
    cursor.execute(f"SELECT {columns_joined} FROM {table}")
    rows = cursor.fetchall()
    result = []
    for row in rows:
        dict_row = {}
        for index, column in enumerate(columns):
            dict_row[column] = row[index]
        result.append(dict_row)
    return result
```

Rys. 30 Funkcja fetchall
Źródło: opracowanie własne

4. Moduł categories.py

Ten moduł zawiera funkcjonalność do wyróżnienia kategorii dochodów.

Używany jako oddzielny moduł ponieważ, tutaj wygodnie zarządzać same kategorie w egzemplarze tej klasy, dlatego żeby dalej nie pobierać ich z bazy danych, to skłęsa ilość zapytań do bazy. Zawiera dwie klasy:

1. Category – struktura kategorii, zawiera cztery zmienne: codename(id), name(nazwa), is_dzieny_dochod, aliases.
2. Categories - klasa zawiera cztery funkcje dla pracy z kategorii dochodów, oraz wprowadzi filtr do wyróżnienia kategorii dochodu.

W konstruktorze otrzymujemy kategorie, które pobieramy przez funkcję `_load_categories`, także jest wykorzystana wcześniej wspomniana funkcja `fetchall`.

```
def _load_categories(self) -> List[Category]:
    categories = db.fetchall(
        "category", "codename name is_dzieny_dochod aliases".split()
    )
    categories = self._fill_aliases(categories)
    return categories
```

Rys. 31 Funkcja load_categories
Źródło: opracowanie własne


```

@staticmethod
def _fill_aliases(categories: List[Dict]) -> List[Category]:

    categories_result = []
    for index, category in enumerate(categories):
        aliases = category["aliases"].split(",")
        aliases = list(filter(None, map(str.strip, aliases)))
        aliases.append(category["codename"])
        aliases.append(category["name"])
        categories_result.append(Category(
            codename=category['codename'],
            name=category['name'],
            is_dzieny_dochod=category['is_dzieny_dochod'],
            aliases=aliases
        ))
    return categories_result

```

Rys. 32 Funkcja fill_aliases

Źródło: opracowanie własne

Funkcja **fill_aliases** – to funkcja na uzupełnienie aliasów, my dodajemy dla każdej z kategorii, poła aliases, w której będzie codename/name.

Funkcja **get_all_categories** po prostu wraca kategorie polem prywatnym o nazwie self_categories, to znaczy, że my nie pracujemy na proste z polem prywatnym self_categories, a otrzymujemy go przez metodę get_all_categories.

Funkcja **get_category** – wraca jedną kategorię po nazwie (name), zawiera cykl, po cykle znajduję jedną z kategorii.

```

def get_all_categories(self) -> List[Category]:

    return self._categories

def get_category(self, category_name: str) -> Category:
    finded = None
    other_category = None
    for category in self._categories:
        if category.codename == "other":
            other_category = category
            for alias in category.aliases:
                if category_name in alias:
                    finded = category
    if not finded:
        finded = other_category
    return finded

```

Rys. 33 Funkcja get_all_categories

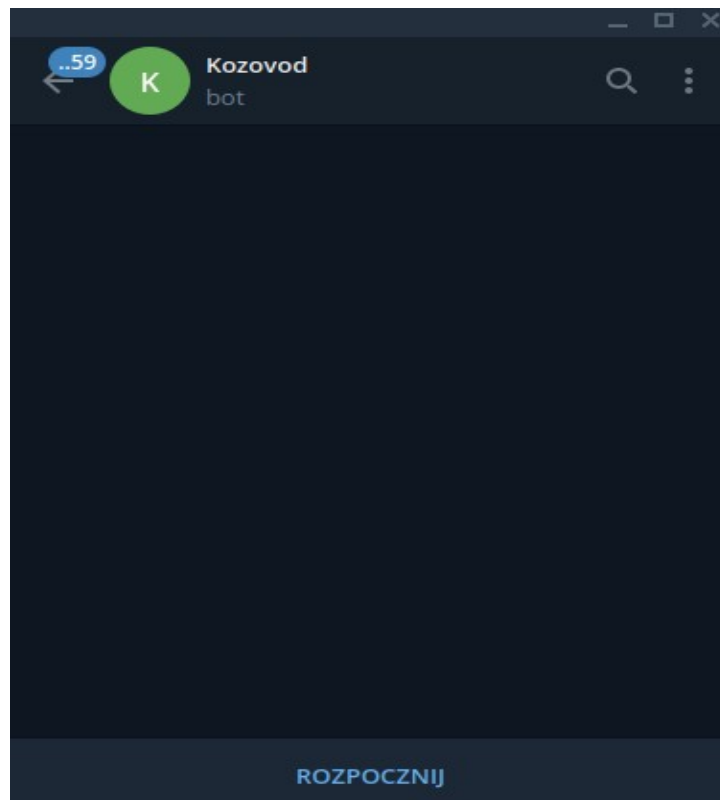
Źródło: opracowanie własne

4.2 Opis działania aplikacji

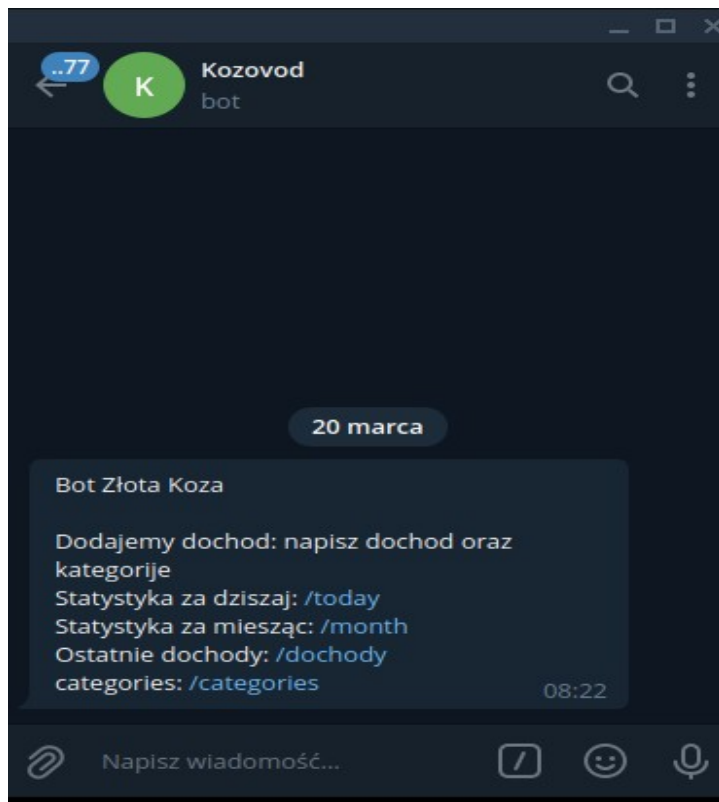
Główna funkcja `add_dochod` modułu `main.py` otrzymuje tekst wprowadzony przez chat użytkownika, oraz dodaje ten tekst do bazy danych.

Funkcja `add_dochod` importuje moduł `dochody.py` oraz przyjmuje tekst wprowadzony przez chat, dalej analizuje kategorie wprowadzonych dochodów.

Moduł `dochody.py` to jest moduł który bezpośrednio pracuje z dodawaniem dochodów, główna funkcja `add_dochod` zarządza wprowadzone dane w bazie danych, i wraca klasę `Dochody` – ta klasa to jest zwykły `named tuple` który ma dwa pola z sumą oraz nazwą kategorii dochodów.



Rys. 34 Start programu
Źródło: opracowanie własne



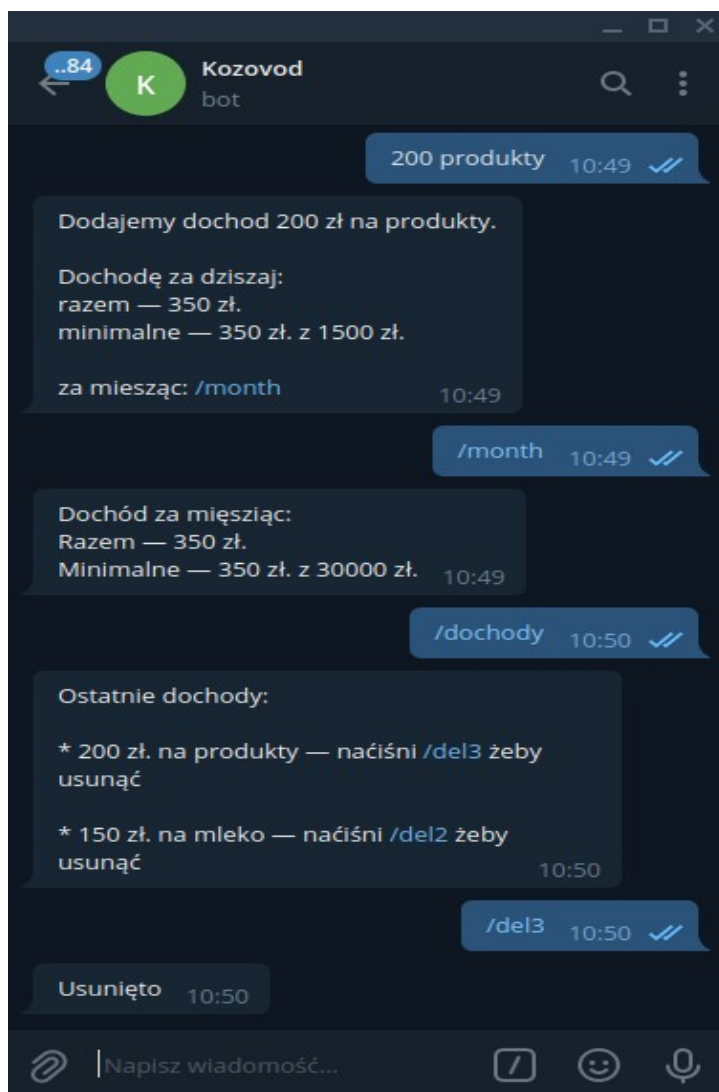
Rys. 35 Hello
Źródło: opracowanie własne



Rys. 36 Dodanie dochodów
Źródło: opracowanie własne

Najpierw zrobimy test, czy sprawdza bot wiadomość napisana zgodnie z wymogami programu, ta możemy spróbować dodać coś do bazy danych. Jak można zobaczyć na Rys.29, nam udało się dodać dochody o 150zł do kategorii mleko.

Teraz spróbujemy dodać dane do innej kategorii, a potem usunąć dane z bazy danych i wyświetlić statystykę za miesiąc.



Rys. 37 Usunięcie dochodów
Źródło: opracowanie własne

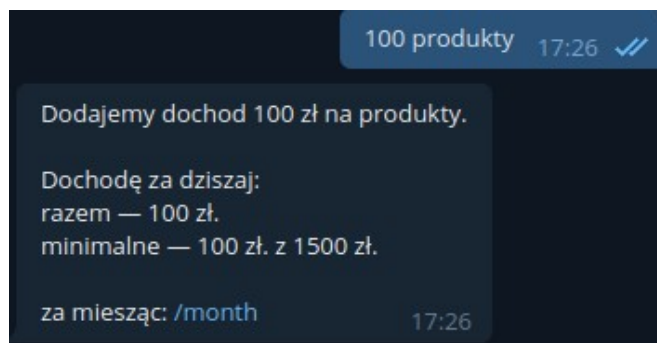
Jak można zobaczyć na Rys. 8, nam udało się dodać dane o innej kategorii, potem wyświetlić statystykę za miesiąc i usunąć konkretni dochód.

5 Testowanie

W trakcie implementacji oraz po zakończeniu implementacji poszczególnych modułów aplikacji, były przeprowadzone testy automatyczne oraz manualne.

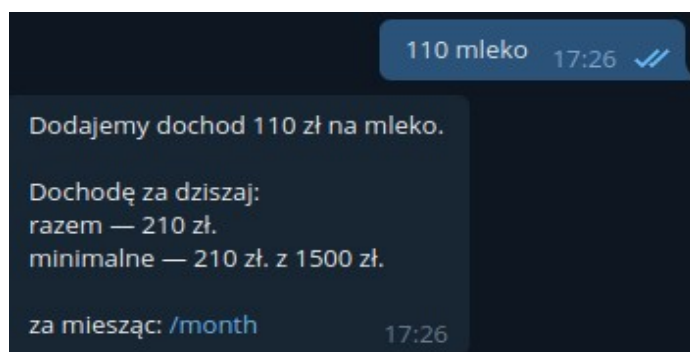
Na poniższych obrazkach, zostanie pokazany proces dodawanie dochodów dla każdej kategorii:

1. Dodawanie 100 zł do kategorii „produkty”, jak można zobaczyć na obrazku (Rys. 35).



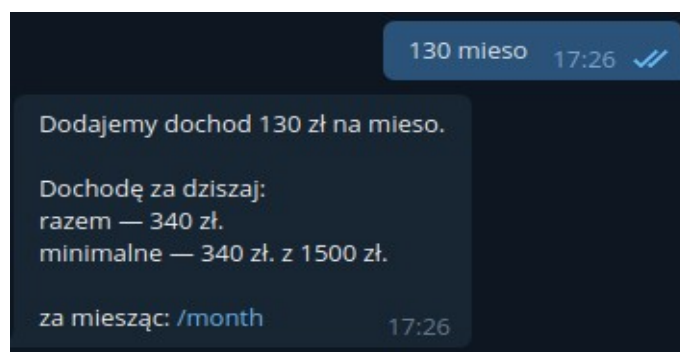
Rys. 38 Kategorie „produkty”
Źródło: opracowanie własne

2. Dodawanie 110 zł do kategorii „mleko”, jak można zobaczyć na obrazku (Rys. 36).



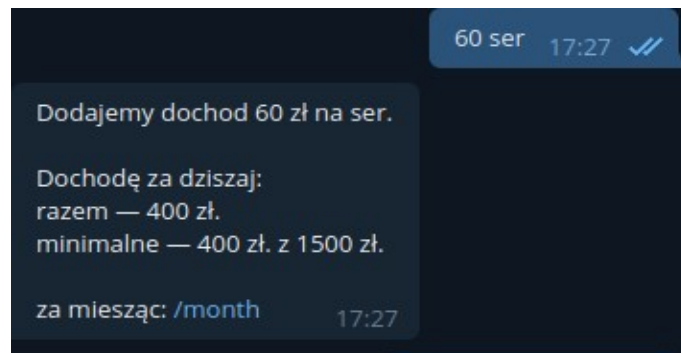
Rys. 39 Kategorie „mleko”
Źródło: opracowanie własne

3. Dodawanie 130 zł do kategorii „mięso”, jak można zobaczyć na obrazku (Rys. 37).



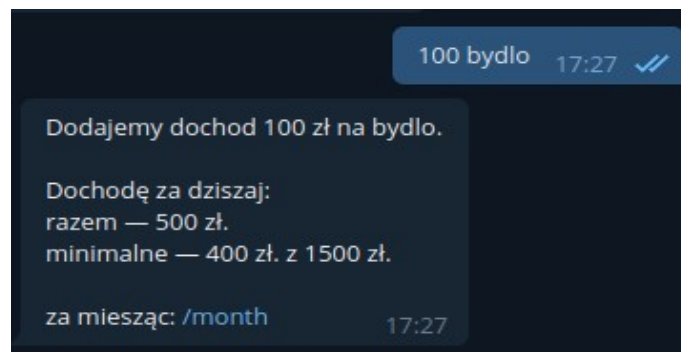
Rys. 40 Kategorie „mięso”
Źródło: opracowanie własne

4. Dodawanie 60 zł do kategorii „ser”, jak można zobaczyć na obrazku (Rys. 38).



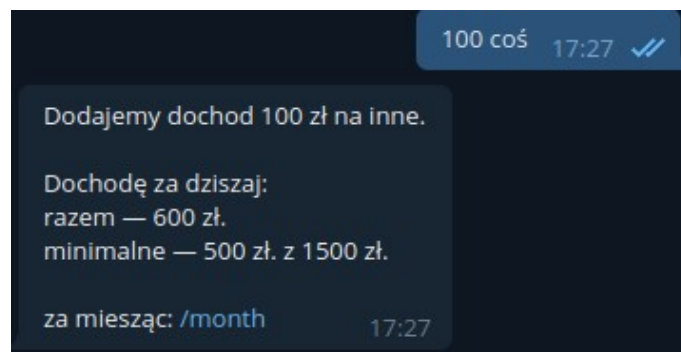
Rys. 41 Kategorie „ser”
Źródło: opracowanie własne

5. Dodawanie 100 zł do kategorii „bydło”, jak można zobaczyć na obrazku (Rys. 39).



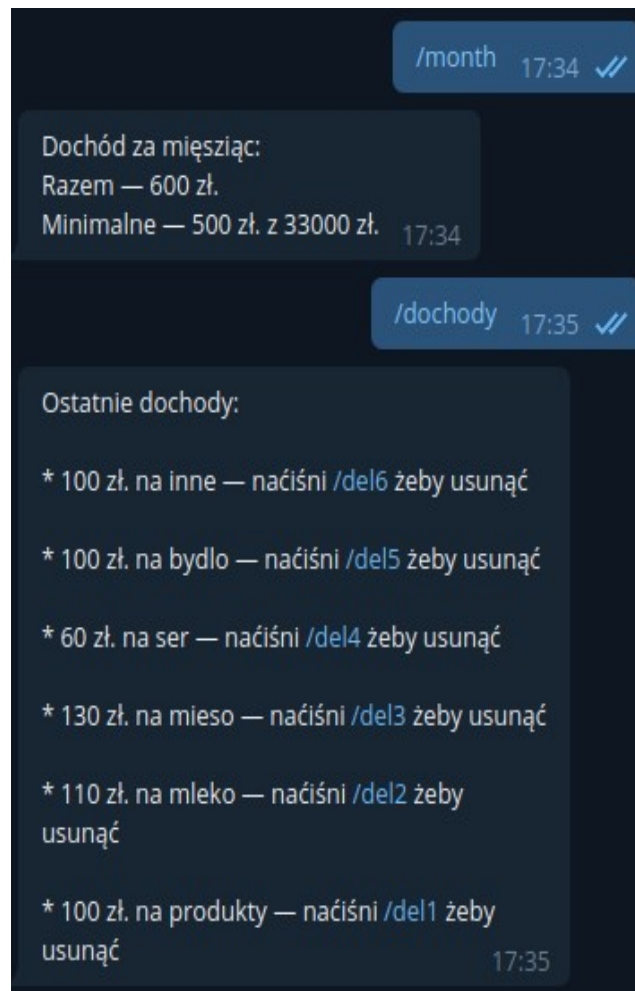
Rys. 42 Kategorie „bydło”
Źródło: opracowanie własne

6. Dodawanie 100 zł do kategorii „inne”, jak można zobaczyć na obrazku (Rys. 40).



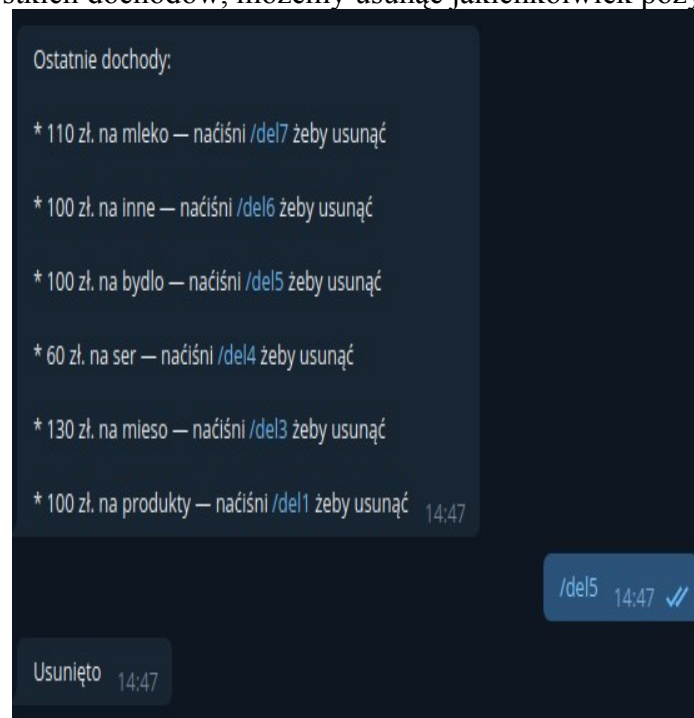
Rys. 43 Kategorie „inne”
Źródło: opracowanie własne

Na obrazku (Rys. 41) można zobaczyć, że wszystkie dochody są umieszczone do odpowiednich kategorii



Rys. 44 Wszystkie kategorie
Źródło: opracowanie własne

Po wyświetlaniu wszystkich dochodów, możemy usunąć jakichkolwiek pozycję z bazy.



Rys. 45 Usuwanie jednej kategorii
Źródło: opracowanie własne

Zakończenie

W ramach pracy dyplomowej został zaprojektowany i zaimplementowany bot do zarządzania danymi dochodów, który spełnił wymagania zarówno funkcjonalne jak i нефункционалне, а także spełnia główne wymaganie do projektu. Program rozwiązuje problem opisany w pierwszym rozdziale pracy. Takie rozwiązanie wykorzystujące boty w dniu dzisiejszym ma duży potencjał na rynku, а dzięki strukturze może być rozwijany i rozszerzany.

W trakcie projektowania oraz implementacji rozwiązania zostało przeanalizowanych kilka dostępnych na rynku narzędzi, oraz programy w postaci Telegram-bot-API. Ze względu na to oraz na nietypowy problem, który był rozwiązywany, została wzbogacona wiedza nie tylko w zakresie wytwarzania oprogramowania.

Oprócz nawyków technologicznych, które były wzbogacone podczas wykonywania pracy dyplomowej, ważnym zadaniem było również nauczenie się projektowania użytecznych oraz nowoczesnych interfejsów użytkownika.

Literatura

Źródła internetowe (WWW)

[WWW-1, 2021]

<https://telegram.org/faq/pl#p-czym-jest-telegram-co-moge-nim-robic>, z dnia 20.02.2021.

[WWW-2, 2021]

<https://pypi.org/project/aiogram>, z dnia 20.03.2021

[WWW-3, 2021]

https://pl.wikipedia.org/wiki/Wyrazenia_regularne, z dnia 20.02.2021.

[WWW-4, 2021]

https://docs.aiogram.dev/en/latest/quick_start.html, z dnia 19.02.2021.

[WWW-5, 2021]

<https://pypi.org/project/pyTelegramBotAPI/>, z dnia 17.03.2021.

[WWW-6, 2021]

<https://pythonru.com/primery/funkcionalnost-telegram-bota>, z dnia 17.03.2021.

[WWW-7, 2021]

<https://buildmedia.readthedocs.org/media/pdf/aiogram/dev-2.x/aiogram.pdf>, z dnia 12.03.2021.

[WWW-8, 2021]

<https://github.com/mahenzon/aiogram-lessons/blob/master/lesson-03/bot.py#L20>, z dnia 18.03.2021.

[WWW-9, 2021]

<https://habr.com/ru/post/262247/>, z dnia 20.03.2021.

[WWW-10, 2021]

<https://pythonworld.ru/moduli/modul-unittest.html/>, z dnia 19.03.2021.

Streszczenie

Wyższa Szkoła Informatyki i Zarządzania z siedzibą w Rzeszowie

Kolegium Informatyki Stosowanej

Streszczenie pracy dyplomowej

Projekt i implementacja programu typu bot dla hodowli „Złota koza”

Autor: Mykhailo Husiev

Promotor: dr inż. Leszek Gajecki

Słowa kluczowe: baza danych dla hodowli, zarządzanie dochodów, Telegram-bot-API, Aoigram

Celem danej pracy dyplomowej było zaprojektowanie i implementacja programu dla hodowli „Złota koza” który uprości proces zarządzania dochodów z hodowli. Program, który jest produktem końcowym jest to bot w chacie po stronie klienta oraz Telegram-bot-API w technologii Python 3.8 po stronie serwera. W ramach pracy dyplomowej zostały przeanalizowane dostępne narzędzia dla rozwiązania danego problemu, wady oraz zalety danych narzędzi i na podstawie tego został wybrany odpowiedni stos technologii. Wykonane główne wymaganie do programu – „żeby program zajmował jak najmniej pamięci”.

Załączniki

Załączniki zostaną zamieszczone na płycie CD, na której będą dodane:

- 1 Folder z kodem źródłowym.
- 2 Folder z kodem źródłowym do Telegram-bot-API.