Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

Отчет по Курсовой работе

”Телеграм бот для ведения расходов”

по дисциплине ”Архитектура программных систем”

Худяков Г.А.
Выполнили студенты Никифорова Е.А.
гр. 5130904/10101 Абраамян А.М.

Преподаватель Коликова Т.В.

Содержание

1 Описание задачи

Хотелось бы видеть бота, который позволил бы эффективно фиксировать расходы и доходы, видеть текущий баланс, планировать траты на категории и смотреть статистику за определенный период времени.

Дополнительно желательно иметь возможность контроллировать поток выполнения программы так как будет нужно, без применения сторонних бибилотек таких как asyncio, aiogram и прочих.

2 Техническая реализация

Для выполнения будет использоваться

- Python 3.10
- PostgreSQL
- Docker Compose

2.1 Функциональные и нефункциональные требования

1. База данных должна быть поднята в PostgreSQL
2. Название для базы данных должно быть ”maindb”
3. Имя для суперпользователя по умолчанию ”main_user”
4. Пароль базы данных ”admin”
5. База данных должна автоматически перезапускаться при каждом фатальном сбое
6. Телеграм бот должен иметь возможность обработки нескольких пользователей одновременно
7. Бот должен работать в режиме non-stop
8. Пользователи в базе данных должны храниться с id из телеграма
9. Каждый пользователь должен иметь собственный баланс
10. У каждого пользователя должны быть свои категории покупок и доходов
11. У пользователя должна быть возможность фиксировать трату или прибыль по категории
12. У пользователя должна быть возможность смотреть историю своих покупок
13. Бот должен вести логирование и выводить их в файл general.log
14. Бот должен адекватно обрабатывать внезапно возникающие исключения
15. Токен для бота должен храниться в config.yaml по пути bot.token
16. Данные для базы данных также должны храниться в config.yaml
17. Имя базы данных database.dbname
18. Хост базы данных database.host
19. Пользователь базы данных database.user
20. Пароль базы данных database.password
21. Порт базы данных database.port

3 Архитектура

3.1 Код на питоне

Для реализации задуманной идеи мы с командой придумали следующую архитектуру. Первым делом настраивается логирование. Дальше считывается токен с файла и создается бот с использованием библиотеки telebot.

```
from logging_setup import logging_setup
from configreader import ConfigReader
import logging
import telebot
from messageprocessing.router.message_router import MessageRouter
from messageprocessing.botstate import BotState

def main():
    logging_setup()
    logging.info("Hello world!")
    bot = telebot.TeleBot(ConfigReader().bot_token)
    BotState(bot)
    bot.register_message_handler(MessageRouter().process_message, func =
    lambda x: True)
    bot.polling()


if __name__ == "__main__":
    main()
```

Listing 1: main.py

Для работы с конфигом используется класс ConfigReader.py из собственного пакета configreader. Класс содержит в себе свойства которые предоставляют доступ к полям конфиг файла. При отсутствии какого либо поля в конфиг файле будет выброшено исключение KeyError.

```
    from singleton_decorator import singleton
    import yaml

    @singleton
    class ConfigReader:
        def __init__(self, path_to_file) -> None:
            with open(path_to_file, 'r') as config_file:
                self.yaml_config = yaml.safe_load(config_file)

        @property
        def db_host(self):
            return self.yaml_config["database"]["host"]

        @property
        def db_name(self):
            return self.yaml_config["database"]["dbname"]

        @property
        def db_user(self):
            return self.yaml_config["database"]["user"]

        @property
        def db_password(self):
```

```
            return self.yaml_config["database"]["password"]

        @property
        def db_port(self):
            return self.yaml_config["database"]["port"]

        @property
        def bot_token(self):
            return self.yaml_config["bot"]["token"]
```

Listing 2: ConfigReader.py

За обработку всех сообщений будет отвечать класс MessageRouter. Он будет хранить в себе состояния для каждого пользователя. Или если точнее - хранить в себе хендлеры для каждого пользователя.

```
from singleton_decorator import singleton
from telebot.types import Message
from ..handlers.start_handler import StartHandler
from ..handlers.base_handler import BaseHandler
from typing import Dict


@singleton
class MessageRouter:

    def __init__(self) -> None:
        self.id_handler: Dict[int, BaseHandler] = {}

    def process_message(self, message: Message):
        # FIXME: add try except
        if not message.from_user:
            return

        if not message.from_user.id in self.id_handler:
            self.id_handler[message.from_user.id] = StartHandler.
    switch_to_this_handler(message)
            return

        handler = self.id_handler[message.from_user.id]
        self.id_handler[message.from_user.id] = handler.handle_message(
    message)
```

Listing 3: MessageRouter.py

Прежде чем идти дальше - необходимо выяснить что мы называем словом хендлер(handler). Можно провести аналогию с библиотекой aiogram, с классом State. Хендлер хранит в себе текущий этап общения с пользователем. Например мы хотим считать от пользователя число и текст. Для этого нам понадобится два раза поменять хендлер - на считывание числа и на считывание текста. С технической точки зрения хендлер это класс который наследуется от одного из BaseHandler, ReusableHandler, BaseInnerHandler и ReturningResultHandler, или от нескольких сразу.

Классы которые наследуются от **BaseHandler** получают в подарок статический абстрактный метод `switch_to_this_handler(message: Message) -> BaseHandler`. Данный метод инициирует переключение на выбранный хендлер и возвращает объект хендлер которым нужно заменить старый. Сказано, что возвращает он

BaseHandler, но в питоне есть выражение

"If it walks like a duck, and it quacks like a duck, then it must be a duck"

На самом деле достаточно чтобы абстрактный метод возвращал класс обладающий всеми способностями BaseHandler. Также наследующийся класс получает метод handle_message, который MessageRouter с радостью будет вызывать, если у пользователя сейчас именно этот хендлер. После вызова этот метод возвращает либо свой объект-хендлер, либо возвращает новый объект, тем самым переключая пользователя на другое состояние.

```
class BaseHandler(ABC):

    def __init__(self) -> None:
        pass

    @abstractmethod
    def handle_message(self, message) -> BaseHandler:
        pass

    @staticmethod
    @abstractmethod
    def switch_to_this_handler(message: Message) -> BaseHandler:
        pass
```

Listing 4: class BaseHandler

Классы наследующиеся от ReusableHandler получают абстрактный метод `switch_to_existin` Этот метод позволяет переключиться обратно к объекту хендлера без необходимости заново его создавать, данный подход позволяет оптимизировать выполнение кода. Возвращает он свой объект.

```
class ReusableHandler(BaseHandler):
@abstractmethod
def switch_to_existing_handler(self, message: Message) -> ReusableHandler:
    """
    This will make possible to reuse handler.
    E.g. if you have handler object and want to activate it - call this
   method.
    """
    pass
```

Listing 5: class ReusableHandler

Есть также абстрактный класс **BaseInnerHandler**, который аналогичным образом описывает хендлер, но по завершению он подразумевает переключение на outter_handler. Хендлеры могут сколько угодно раз вкладываться друг в друга.

```
class BaseInnerHandler(BaseHandler):
    def __init__(self, outter_handler: ReusableHandler) -> None:
        BaseHandler.__init__(self)
        self.outter_handler = outter_handler

    @staticmethod
    @abstractmethod
    def switch_to_this_handler(message: Message, outter_handler:
   ReusableHandler) -> BaseInnerHandler:
        pass
```

Listing 6: class BaseInnerHandler

Вишенкой на торте является **ReturningResultHandler**. Благодаря этому классу хендлеры можно использовать почти как функции. Можно переключиться на данный хендлер и в outter_handler.return_result он положит результат своей работы с пользователем. Это открывает огромные возможности для переиспользования кода.

```
class ReturningResultHandler(BaseInnerHandler):
    """
        When switching back to the outter_handler - return result will be
    in
        outter_handler.return_result
    """
    def __init__(self, outter_handler: ReusableHandler) -> None:
        ReusableHandler.__init__(self)
        self.outter_handler = outter_handler
```

Listing 7: class ReturningResultHandler

Возвращаясь к MessageRouter, этот класс хранит состояния для каждого пользователя. По умолчанию, если у пользователя еще нет состояния - оно создается классом StartHandler. Дальше уже оно раскручивается в другие хендлеры в зависимости от хода выполнения.

Именно этот класс отвечает за вызов handle_message метода хендлера с передачей туда сообщения пользователя.

3.2 Работа с базой данных

Работа с базой данных происходит с использованием библиотеки psycopg2. Для создания соединения с базой данных используется класс DatabaseConnection.

```
import psycopg2
from singleton_decorator import singleton
from configreader import ConfigReader
import logging

class DatabaseConnection:
    """
    Usage
    -----
    DatabaseConnection.connection()
    """
    @staticmethod
    def connection():
        connection = psycopg2.connect(
            dbname = ConfigReader().db_name,
            user = ConfigReader().db_user,
            password = ConfigReader().db_password,
            host = ConfigReader().db_host,
            port = ConfigReader().db_port
        )
        return connection
```

Listing 8: class ReturningResultHandler

Дальше уже это соединение используется в классе DatabaseApi, который является singleton. Класс содержит в себе огромное количество методов для абстрактной работы с базой данных, приведём get_person_by_id. Данный метод позволяет получить объект пользователя из базы данных по id.

```
from singleton_decorator import singleton
from .connection import DatabaseConnection
from .types.person import Person
from .types.cathegory import Cathegory
from .types.operation import Operation
from functools import lru_cache
import psycopg2
import logging


@singleton
class DatabaseApi:
    def get_person_by_id(self, person_id, conn = None):
        """
        Raise
        -----
            - ProgrammingError if no person found
            - OperationalError if connection establishing failed
        """
        is_connection_local = False
        if conn is None:
            conn = DatabaseConnection.connection()
            is_connection_local = True
        try:
            with conn.cursor() as cursor:
                cursor.execute("SELECT * FROM person WHERE id = %s", (
   person_id,))
                result = cursor.fetchone()
                if result is None:
                    raise psycopg2.ProgrammingError(
                        "Person with id %s was not found" % (person_id,)
                    )
        finally:
            if is_connection_local:
                conn.commit()
                conn.close()
        return Person.fromTuple(result)
...
```

Listing 9: class ReturningResultHandler

3.2.1 Итоги

Здесь была рассмотрена архитектурная модель написанного кода, рассмотрены ключевые классы обычные и абстрактные, рассмотрены прочие аспекты связанные с технической реализацией.

3.3 База данных

Сама база данных поднималась в Docker Compose с помощью вот такого ***docker-compose.yml*** файла.

```
    services:

    postgres:
      build: docker_database
      environment:
```

```
        POSTGRES_DB: "maindb"
        POSTGRES_USER: "main_user"
        POSTGRES_PASSWORD: "admin"

      ports:
        - "30042:5432"

      volumes:
        - postgres-data:/var/lib/postgresql/data

      networks:
        - postgres

      restart: unless-stopped

    pgadmin:
      image: dpage/pgadmin4
      container_name: pgadmin4_container

      ports:
        - "30043:80"

      environment:
        PGADMIN_DEFAULT_EMAIL: pgadmin@jeeeez.com
        PGADMIN_DEFAULT_PASSWORD: admin

      networks:
        - postgres

      volumes:
        - ./pgadmin-data:/var/lib/pgadmin

      restart: unless-stopped

  volumes:
    postgres-data:

  networks:
    postgres:
      driver: bridge
```

Listing 10: class ReturningResultHandler

Здесь мы можем увидеть два микросервиса - непосредственно база данных, а также PgAdmin4 чтобы можно было удобно смотреть текущее содержимое таблиц в базе данных.

Таблицы создавались с помощью следующего SQL скрипта.

```
CREATE TABLE person (
  id BIGINT PRIMARY KEY,
  name VARCHAR(30) NOT NULL,
  cathegory_ids BIGINT[],
  balance BIGINT NOT NULL
);

CREATE TABLE cathegory_type (
  id BIGSERIAL PRIMARY KEY,
  type_name TEXT NOT NULL
);
```

```
12
13 INSERT INTO cathegory_type(type_name) VALUES ('expense'), ('income');
14
15 CREATE TABLE operation_type (
16   id BIGSERIAL PRIMARY KEY,
17   type_name TEXT NOT NULL
18 );
19
20 INSERT INTO operation_type(type_name) VALUES ('change_balance');
21
22 CREATE TABLE cathegory (
23   id BIGSERIAL PRIMARY KEY,
24   person_id BIGINT REFERENCES person(id) ON DELETE CASCADE,
25   cathegory_type_id BIGINT REFERENCES cathegory_type(id) ON DELETE CASCADE,
26   name TEXT NOT NULL,
27   money_limit BIGINT NOT NULL DEFAULT 0,
28   current_money BIGINT NOT NULL DEFAULT 0
29 );
30
31 CREATE TABLE operation (
32   id BIGSERIAL PRIMARY KEY,
33   date TIMESTAMP NOT NULL DEFAULT now(),
34   operation_type_id BIGINT REFERENCES operation_type(id) ON DELETE CASCADE,
35   person_id BIGINT REFERENCES person(id) ON DELETE CASCADE,
36   cathegory_id BIGINT REFERENCES cathegory(id) ON DELETE CASCADE,
37   money_amount BIGINT NOT NULL,
38   commentary TEXT
39 );
```

Listing 11: class ReturningResultHandler

4 Заключение

Мы научились красиво работать в команде над совместным проектом. Освоили новые методы проектирования, разработку на пайтоне, освоили создание контейнеров с базами данных в Docker Compose. Освоили совместную работу в Github.