

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”

Выполнили студенты
гр. 5130904/10101



Худяков Г.А.
Никифорова Е.А.
Абраамян А.М.

Преподаватель

Коликова Т.В.

Содержание

1	Описание задачи	3
2	Техническая реализация	4
2.1	Функциональные и нефункциональные требования	5
3	Архитектура	6
3.1	Код на питоне	6
3.2	Работа с базой данных	9
3.2.1	Итоги	10
3.3	База данных	10
4	Заключение	13

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 контролировать поток выполнения программы так как будет нужно, без применения сторонних библиотек 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.

```
1 from logging_setup import logging_setup
2 from configreader import ConfigReader
3 import logging
4 import telebot
5 from messageprocessing.router.message_router import MessageRouter
6 from messageprocessing.botstate import BotState
7
8 def main():
9     logging_setup()
10    logging.info("Hello world!")
11    bot = telebot.TeleBot(ConfigReader().bot_token)
12    BotState(bot)
13    bot.register_message_handler(MessageRouter().process_message, func =
14    lambda x: True)
15    bot.polling()
16
17 if __name__ == "__main__":
18     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.

```
1 from singleton_decorator import singleton
2 import yaml
3
4 @singleton
5 class ConfigReader:
6     def __init__(self, path_to_file) -> None:
7         with open(path_to_file, 'r') as config_file:
8             self.yaml_config = yaml.safe_load(config_file)
9
10    @property
11    def db_host(self):
12        return self.yaml_config["database"]["host"]
13
14    @property
15    def db_name(self):
16        return self.yaml_config["database"]["dbname"]
17
18    @property
19    def db_user(self):
20        return self.yaml_config["database"]["user"]
21
22    @property
23    def db_password(self):
```

```

24         return self.yaml_config["database"]["password"]
25
26     @property
27     def db_port(self):
28         return self.yaml_config["database"]["port"]
29
30     @property
31     def bot_token(self):
32         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.

```

1 from singleton_decorator import singleton
2 from telebot.types import Message
3 from ..handlers.start_handler import StartHandler
4 from ..handlers.base_handler import BaseHandler
5 from typing import Dict
6
7
8 @singleton
9 class MessageRouter:
10
11     def __init__(self) -> None:
12         self.id_handler: Dict[int, BaseHandler] = {}
13
14     def process_message(self, message: Message):
15         # FIXME: add try except
16         if not message.from_user:
17             return
18
19         if not message.from_user.id in self.id_handler:
20             self.id_handler[message.from_user.id] = StartHandler.
21             switch_to_this_handler(message)
22             return
23
24         handler = self.id_handler[message.from_user.id]
25         self.id_handler[message.from_user.id] = handler.handle_message(
26             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 иницирует 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.

```
1 class BaseHandler(ABC):
2
3     def __init__(self) -> None:
4         pass
5
6     @abstractmethod
7     def handle_message(self, message) -> BaseHandler:
8         pass
9
10    @staticmethod
11    @abstractmethod
12    def switch_to_this_handler(message: Message) -> BaseHandler:
13        pass
```

Listing 4: class BaseHandler

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

```
1 class ReusableHandler(BaseHandler):
2     @abstractmethod
3     def switch_to_existing_handler(self, message: Message) -> ReusableHandler:
4         """
5         This will make possible to reuse handler.
6         E.g. if you have handler object and want to activate it - call this
7         method.
8         """
9         pass
```

Listing 5: class ReusableHandler

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

```
1 class BaseInnerHandler(BaseHandler):
2     def __init__(self, outter_handler: ReusableHandler) -> None:
3         BaseHandler.__init__(self)
4         self.outter_handler = outter_handler
5
6     @staticmethod
7     @abstractmethod
8     def switch_to_this_handler(message: Message, outter_handler:
9         ReusableHandler) -> BaseInnerHandler:
10        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.

```
1 class ReturningResultHandler(BaseInnerHandler):
2     """
3     When switching back to the outter_handler - return result will be
4     in
5     outter_handler.return_result
6     """
7     def __init__(self, outter_handler: ReusableHandler) -> None:
8         ReusableHandler.__init__(self)
9         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`.

```
1 import psycopg2
2 from singleton_decorator import singleton
3 from configreader import ConfigReader
4 import logging
5
6 class DatabaseConnection:
7     """
8     Usage
9     -----
10    DatabaseConnection.connection()
11    """
12    @staticmethod
13    def connection():
14        connection = psycopg2.connect(
15            dbname = ConfigReader().db_name,
16            user = ConfigReader().db_user,
17            password = ConfigReader().db_password,
18            host = ConfigReader().db_host,
19            port = ConfigReader().db_port
20        )
21        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`.

```

1 from singleton_decorator import singleton
2 from .connection import DatabaseConnection
3 from .types.person import Person
4 from .types.category import Category
5 from .types.operation import Operation
6 from functools import lru_cache
7 import psycopg2
8 import logging
9
10
11 @singleton
12 class DatabaseApi:
13     def get_person_by_id(self, person_id, conn = None):
14         """
15         Raise
16         -----
17         - ProgrammingError if no person found
18         - OperationalError if connection establishing failed
19         """
20         is_connection_local = False
21         if conn is None:
22             conn = DatabaseConnection.connection()
23             is_connection_local = True
24         try:
25             with conn.cursor() as cursor:
26                 cursor.execute("SELECT * FROM person WHERE id = %s", (
27                     person_id,))
28                 result = cursor.fetchone()
29                 if result is None:
30                     raise psycopg2.ProgrammingError(
31                         "Person with id %s was not found" % (person_id,))
32         finally:
33             if is_connection_local:
34                 conn.commit()
35                 conn.close()
36         return Person.fromTuple(result)
37 ...

```

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.

```

1 services:
2
3 postgres:
4     build: docker_database
5     environment:

```

```

6     POSTGRES_DB: "maindb"
7     POSTGRES_USER: "main_user"
8     POSTGRES_PASSWORD: "admin"
9
10    ports:
11      - "30042:5432"
12
13    volumes:
14      - postgres-data:/var/lib/postgresql/data
15
16    networks:
17      - postgres
18
19    restart: unless-stopped
20
21    pgadmin:
22      image: dpage/pgadmin4
23      container_name: pgadmin4_container
24
25      ports:
26        - "30043:80"
27
28      environment:
29        PGADMIN_DEFAULT_EMAIL: pgadmin@jeeeee.com
30        PGADMIN_DEFAULT_PASSWORD: admin
31
32      networks:
33        - postgres
34
35      volumes:
36        - ./pgadmin-data:/var/lib/pgadmin
37
38      restart: unless-stopped
39
40    volumes:
41      postgres-data:
42
43    networks:
44      postgres:
45        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.

```

1 CREATE TABLE person (
2   id BIGINT PRIMARY KEY,
3   name VARCHAR(30) NOT NULL,
4   cathegory_ids BIGINT[],
5   balance BIGINT NOT NULL
6 );
7
8 CREATE TABLE cathegory_type (
9   id BIGSERIAL PRIMARY KEY,
10  type_name TEXT NOT NULL
11 );

```

```

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.