ЧАТ БОТ В ПОМОЩЬ СТУДЕНТУ

@Melanchenko @Gorbulin @Kugaevskiy @Sinkevich @Belikova @Alekseeva

ОПИСАНИЕ ПРОЕКТА/ЦЕЛИ

Цель - Разработка программного продукта “Персональный помощник для студентов”, который позволит повысить усвояемость учебного материала и увеличить доступное учащимся время для наработки практических навыков.

Описание работы – Чат бот для генерации расписания и генерации практических советов для учащихся

КОМАНДА ПРОЕКТА И РОЛИ

- Александр М. — Developer
- Сергей Г. — Исследования, тестирование
- Виталий К. — Developer
- Денис С. — Developer, Орг. вопросы, оформление
- Юлия Б. — Developer
- Алёна А. — Исследования, тестирование, технический писатель

1 ЭТАП СОЗДАНИЕ ФАЙЛОВОЙ СТРУКТУРЫ ПРОЕКТА

Файловая структура нашего бота:

- **main.py** — точка входа, код запуска бота и инициализации всех остальных модулей
- **config.py** — файл со всеми конфигурационными параметрами, такими как токен бота и данные подключения к БД.
- **text.py** — все тексты, используемые ботом. В этом файле будут лежать все приветствия, сообщения об ошибках и другие текстовые данные для бота.
- **kb.py** — все клавиатуры, используемые ботов. В этом файле будут находиться абсолютно все клавиатуры, как статические, так и динамически генерируемые через функции
- **states.py** — будет хранить вспомогательные классы для FSM (машины состояний), а также фабрики Callback Data для кнопок Inline клавиатур
- **utils.py** — различные функции. В этом файле будут лежать функции для рассылки, генерации текста и изображений через API и другие
- **handlers.py** — основной файл, в котором будет содержать почти весь код бота. Будет состоять из функций-обработчиков с декораторами (фильтрами)
- **admin.py** — обработчики событий, клавиатуры, классы и весь остальной код админки бота. (в разработке)
- **pars.py** – на основе введённых студентом данных формирует датасет с расписанием занятий
- **rest.py** – на основе введённых студентом данных формирует датасет с временным промежутком, для свободного времени

2 ЭТАП MAIN, HANDLERS, KB, TEXT

```python
async def main():

    bot = Bot(token=config.BOT_TOKEN, parse_mode=ParseMode.HTML)
    dp = Dispatcher(storage=MemoryStorage())
    dp.message.middleware(ChatActionMiddleware())
    dp.include_router(router)
    await bot.delete_webhook(drop_pending_updates=True)
    await dp.start_polling(bot, allowed_updates=dp.resolve_used_update_types())

if __name__ == "__main__" :
    logging.basicConfig(level=logging.INFO)
    asyncio.run(main())

```

Мы объявляем функцию main(), в которой будет запускаться бот.

Далее мы создаём объект бота с нашим токеном.

parse_mode, отвечает за используемую по умолчанию разметку сообщений. Мы используем HTML, чтобы избежать проблем с экранированием символов.

Затем мы создаём объект диспетчера, параметр storage=MemoryStorage() говорит о том, что все данные бота, которые мы не сохраняем в БД (к примеру состояния), будут стёрты при перезапуске.

Строка dp.include_router(router) подключает к нашему диспетчеру все обработчики, которые используют router.

Строка await bot.delete_webhook(drop_pending_updates=True) удаляет все обновления, которые произошли после последнего завершения работы бота, чтобы бот обрабатывал только те сообщения, которые пришли ему непосредственно во время его работы, а не за всё время.

Следующая строка запускает бота.

2 ЭТАП MAIN, HANDLERS, KB, TEXT

```python
router = Router()

@router.message(Command("start"))
async def start_handler(msg: Message):
    await msg.answer(text.greet.format(name=msg.from_user.full_name), reply_markup=kb.menu)


@router.message(F.text == "Меню")
@router.message(F.text == "Выйти в меню")
@router.message(F.text == "◀ Выйти в меню")
async def menu(msg: Message):
    await msg.answer(text.menu, reply_markup=kb.menu)

@router.callback_query(F.data == "generate_text")
async def input_text_prompt(clbck: CallbackQuery, state: FSMContext):
    await state.set_state(Gen.text_prompt)
    await clbck.message.edit_text(text.gen_text)
    await clbck.message.answer(text.gen_exit, reply_markup=kb.exit_kb)

@router.message(Gen.text_prompt)
@flags.chat_action("typing")
async def generate_text(msg: Message, state: FSMContext):
    prompt = msg.text
    mesg = await msg.answer(text.gen_wait)
    res = await utils.generate_text(prompt)
    if not res:
        return await mesg.edit_text(text.gen_error, reply_markup=kb.iexit_kb)
    await mesg.edit_text(res[0] + text.text_watermark, disable_web_page_preview=True)
```

В функции start мы отправляем текст из переменной greet модуля text, форматируем его, подставляя имя пользователя (msg.from_user.full_name), а также прикрепляем к сообщению inline-клавиатуру.

Далее добавился обработчик menu. Как вы могли заметить, перед объявлением функции стоят целых три декоратора. Это означает, что функция запустится, если сработает любой из трёх фильтров.

Функция отправит текст text.menu с клавиатурой kb.menu.

В нашем коде встречается такой декоратор как callback_query. Он означает, что функция будет реагировать на нажатия inline-кнопок с определённым фильтром. Если при обработке сообщений надо было использовать выражение F.text, то теперь мы используем F.data.

Добавили функцию установки состояний через set_state. Данная функция устанавливает для пользователя переданное ей состояние (предварительно созданное как класс). Потом мы обрабатываем сообщения с фильтром состояния @router.message(Gen.text_prompt). Это означает что функция будет реагировать только на те входящие сообщения, которые были отправлены после установки состояния в предыдущей функции.

Декоратор @flags.chat_action(«typing»). Именно для его использования мы подключали ChatActionMiddleware. Его функция очень проста — пока выполняется функция, к которой он прикреплён, у пользователя будет отображаться, что бот «печатает…», также можно задать любой другой статус.

Далее код с функциями answer и edit_text интуитивно понятен — мы либо отвечаем текстом и клавиатурой на сообщение, либо редактируем то сообщение, от которого нам пришёл Callback Query.

В функциях где работа идёт с функциями из utils выполняется проверка на ошибки — если API вернёт ошибку или она произойдёт в самой функции, то она вернёт нам None. Поэтому перед отправкой пользователю сообщения с ответом мы проверяем, успешно ли функция отработала. К каждому ответу от бота будет прикреплён специальный текст, у меня помещённый в переменную text.text_watermark. Это обеспечит нам некоторую рекламу, если пользователь решит переслать ответ нейросети другому пользователю.

Параметр disable_web_page_preview отвечает за отображение превью ссылок.

2 ЭТАП MAIN, HANDLERS, KB, TEXT

```
menu = [
    [InlineKeyboardButton(text="Спросить ИИ", callback_data="generate_text"),
    InlineKeyboardButton(text="Узнать Расписание", callback_data="generate_rasp")],
    [InlineKeyboardButton(text="Краткое содержание", callback_data="generate_recenz"),
    InlineKeyboardButton(text="Краткая информация по предмету", callback_data="generate_shpora")],
    [InlineKeyboardButton(text="🔎 Помощь", callback_data="help")]
]
menu = InlineKeyboardMarkup(inline_keyboard=menu)
exit_kb = ReplyKeyboardMarkup(keyboard=[[KeyboardButton(text="◀ Выйти в меню")]], resize_keyboard=
iexit_kb = InlineKeyboardMarkup(inline_keyboard=[[InlineKeyboardButton(text="◀ Выйти в меню", call
```

Меню нашего бота реализовано с помощью inline кнопок. Все клавиатуры будут храниться в файле kb.py.

Здесь мы создаём основную клавиатуру menu, сразу же добавляя все кнопки в два столбца. И последнюю помощь замыкающей.

В нашем проекте для создания меню используется передача двумерного списка кнопок как аргумент при создании клавиатуры. Он удобен когда клавиатура статичная и все данные для неё заранее известны.

2 ЭТАП MAIN, HANDLERS, KB, TEXT

```
greet = "Привет, {name}, я робот, использующий нейросеть от OpenAI, чем Вам помочь?"
menu = "Главное меню"
gen_text = "Отправьте текст запроса к нейросети для генерации текста"
gen_exit = "Чтобы выйти из диалога с нейросетью нажмите на кнопку ниже"
gen_error = f'🚫 Ошибка генерации. Возможные причины:\n1. Перегружены сервера OpenAI\n2.
text_watermark = '*Создано при помощи OpenAI*'
gen_wait = "⏳Пожалуйста, подождите немного, пока нейросеть обрабатывает ваш запрос..."

err = "🚫 К сожалению произошла ошибка, попробуйте позже"
```

Все сообщения с текстами которые отправляются пользователю

```
    await clbck.message.edit_text(text.gen_text)
    await clbck.message.answer(text.gen_exit, reply_markup=kb.exit_kb)
```

Хранятся в модуле text.py

3 ЭТАП ПОДКЛЮЧЕНИЕ НЕЙРОСЕТИ

```python
openai.api_key = config.OPENAI_TOKEN

async def generate_text(prompt) -> dict:
    try:
        response = await openai.ChatCompletion.acreate(
            model="gpt-3.5-turbo",
            messages=[
                {"role":"user","content": prompt}
            ]

        )
        return response ['choices'][0]['message']['content'
    except Exception as e:
        logging.error(e)
```

После импорта мы настраиваем библиотеку openai, давая ей наш ключ от API и обьявляем функцию generate_text.

В функции используется конструкция try-catch для обработки исключений, в этом случае мы ничего не делаем, а лишь выводим ошибку в логи и возвращаем пустое значение.

Функция openai.ChatCompletion.acreate генерирует текст с помощью моделей завершения текста. В качестве параметров передаём используемую модель, в нашем случае gpt-3.5-turbo — самая дешёвая и быстрая на данный момент, и сообщения — список словарей с ключами system, user, assistant.

Мы передаём только сообщение от пользователя и используем поведение модели по умолчанию, но можно также передавать системные сообщения (role: system) для реализации режимов работы, например отдельные функции для написания кода и ответов на теоретические вопросы. Также мы планируем усовершенствовать функцию, чтобы сохранять контекст общения — нейросеть будет «помнить» предыдущие сообщения от пользователя и учитывать их при создании ответа.

3 ЭТАП ПОДКЛЮЧЕНИЕ НЕЙРОСЕТИ

```python
from aiogram.fsm.state import StatesGroup, State

class Gen(StatesGroup):
    text_prompt = State()
```

Здесь нам нужно познакомиться с одной из самых удобных и мощных на мой взгляд функций aiogram, которой нет во многих других библиотеках — машиной состояний (её также называют машиной конечных автоматов или просто FSM).

FSM мы будем использовать чтобы бот принимал промпты для генерации текста и изображений только после нажатия соответствующей кнопки в меню, а также для того, чтобы различать сообщения из разных пунктов меню, так как бот не знает, в каком разделе меню находится пользователь.

Мы создали класс Gen и создали в нём состояние text_prompt — бот будет воспринимать сообщения как промпты для ChatGPT.

4. ЭТАП ДОБАВЛЕНИЕ СКРИПТОВ ДЛЯ АНАЛИЗА РАСПИСАНИЯ

```python
def handle_text(text):
    x = text.split(", ")
    if len(x) == 4:
        date, time, name, task = None, None, None, None
        for line in x:
            if isDateFormat(line):
                date = line
            elif isTimeFormat(line):
                time = line
            elif line == "да" or line == "нет":
                task = line
            else:
                name = line

        if all((date, time, name, task)):
            # Добавляем данные в DataFrame
            global df
            new_row = pd.DataFrame([[date, time, name, task]], columns=['date', 'time', 'name', 'task'])
            df = pd.concat([df, new_row], ignore_index=True)
            print("Данные добавлены в датасет")
        else:
            print("Некорректные данные")
```

Функция “handle_text” – из строки с данными, распознает дату, время, название предмета и наличие задания по предмету и добавляет данные в датасет.

Функции:“isDateFormat”,“isTimeFormat”, “isTaskFormat”, проверяют являются ли введенные данные датой, временем или задачей соответсвенно.

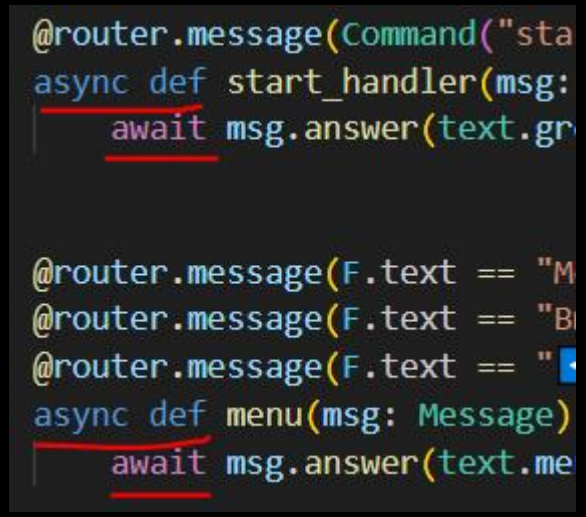
4. ЭТАП ДОБАВЛЕНИЕ СКРИПТОВ ДЛЯ АНАЛИЗА РАСПИСАНИЯ

```python
def input_data():
    data, firsttime, lasttime = None, None, None

    for i in range(3):
        try:
            data_str = input("Введите дату в формате дд/мм/гггг: ").strip()
            firsttime_str = input("Введите время начала отдыха в формате чч:мм: ").strip()
            lasttime_str = input("Введите время окончания отдыха в формате чч:мм: ").strip()

            data = DT.datetime.strptime(data_str, '%d/%m/%Y').date()
            firsttime = DT.datetime.strptime(firsttime_str, '%H:%M').time()
            lasttime = DT.datetime.strptime(lasttime_str, '%H:%M').time()

            break
        except ValueError:
            print(f'Вы ввели некорректные данные. Проверьте правильность ввода. Осталось попыток: {2 - i}')
            time.sleep(3)

    if data is not None and firsttime is not None and lasttime is not None:
        df = pd.DataFrame([[data, firsttime, lasttime]],
                          columns=['Дата', 'Время начала отдыха', 'Время окончания отдыха'])
        return df
    else:
        return None
```

- Функция “its_freetime” выполняет проверку на то какие собирается ввести пользователь, являются ли они отдыхом

- Функция “input_data” позволяет пользователю ввести данные по дате, времени началу отдыха и времени окончания отдыха, проверяет на корректность этих данных и возвращает датасет

ПОЯСНЕНИЯ



Как вы могли заметить в нашем проекте использованно асинхронное программирование, это позволяет нам выполнять несколько задач одновременно без блокирования основного потока программы. В Python для этого используются корутины, asyncio и async/await.

Корутины — это специальный тип функций, который может быть приостановлен и возобновлен в процессе выполнения. В Python корутины создаются с помощью ключевого слова async def.

Asyncio это библиотека для асинхронного программирования, встроенная в стандартную библиотеку Python начиная с версии 3.4. Она предоставляет событийный цикл, который управляет выполнением корутин и обеспечивает асинхронную работу с сетью, файлами и другими операциями.

Async/await — это синтаксический сахар для работы с корутинами. await используется для вызова асинхронных функций и ожидания их выполнения.