

ВСП 2.5

(Создание бота для Telegram)

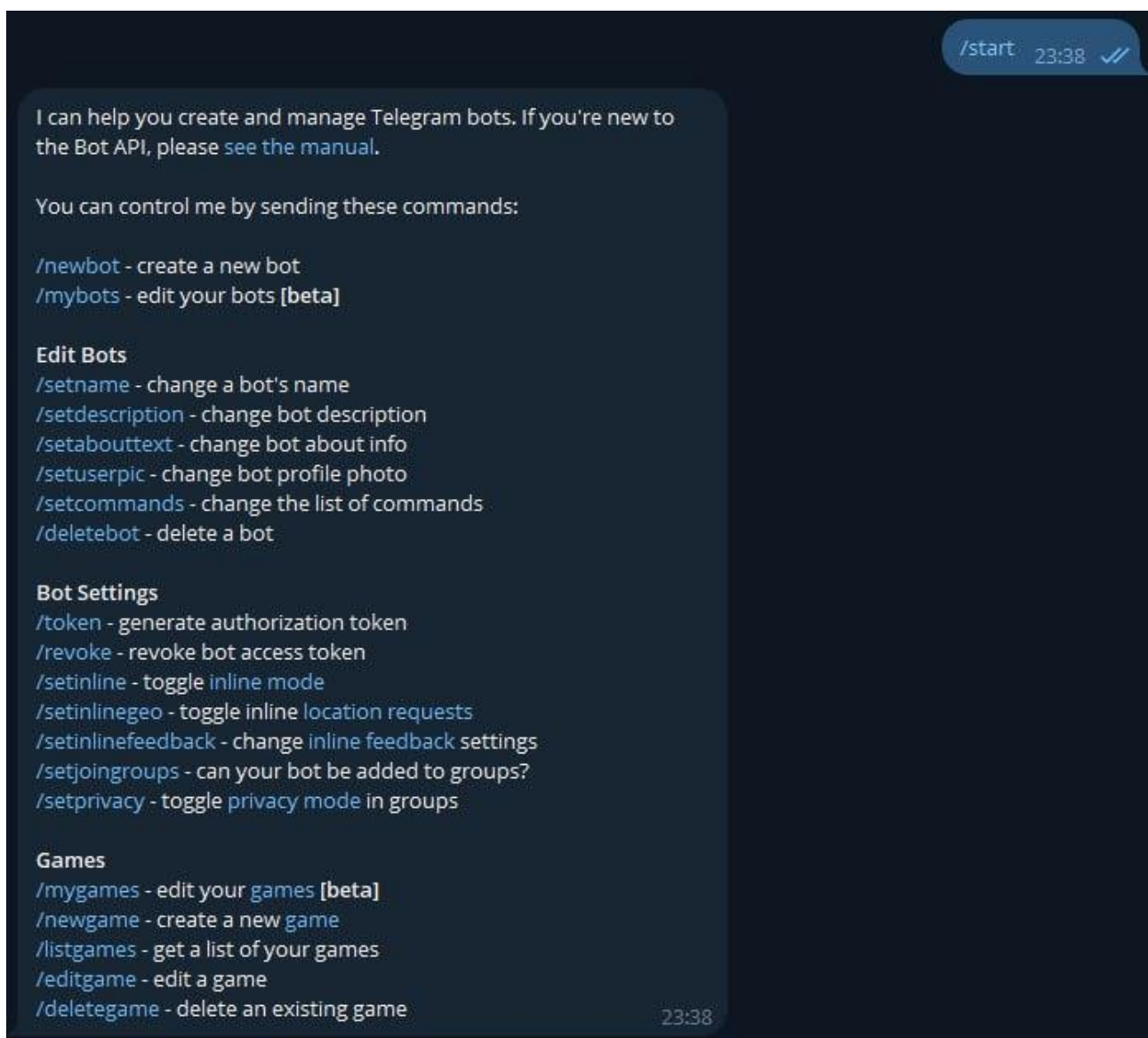
Введение

В рамках задания требовалось создать Telegram-бота с помощью онлайн-приложения. Однако в процессе работы было принято решение реализовать бота с использованием языка программирования Python, что потенциально позволяет добиться большей гибкости и расширенных возможностей.

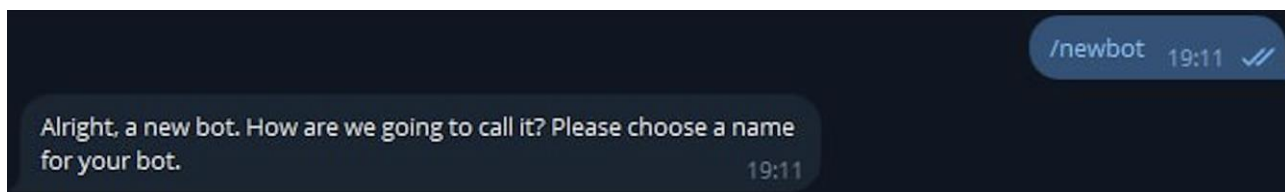
В данном отчёте показан пошаговый аннотированный алгоритм создания Telegram-бота и написания для него кода на языке Python. Несмотря на отличие от изначально предложенного метода (онлайн-конструктор), принципы работы Telegram-ботов остаются неизменными:

Регистрация бота

После того, как была придумана идея и цель бота, в Telegram необходимо найти аккаунт BotFather, это бот, созданный командой разработчиков Telegram, который позволяет создавать и управлять ботами пользователей. После ввода команды «start», BotFather пришлёт список всех команд.



Сразу же можно нажать на команду `/newbot`, тем самым создав нового бота. После, BotFather вышлет сообщение, в котором попросит указать название этого бота.

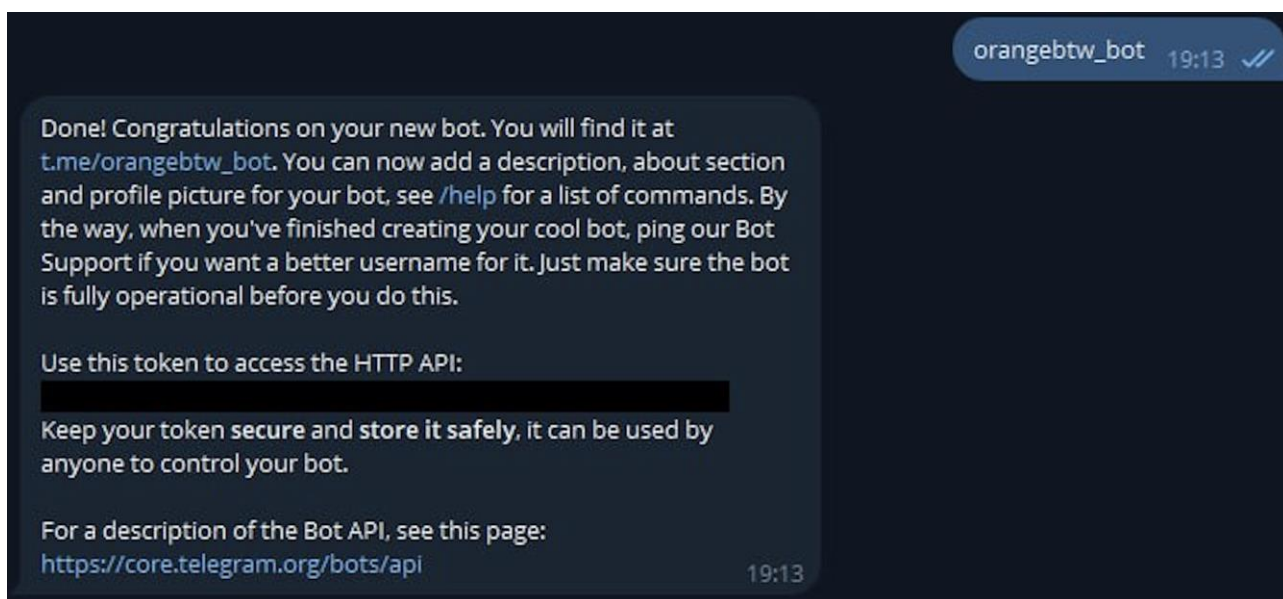


Указав название, BotFather попросит придумать username.



Имя бота должно быть уникальным. Если оно используется каким-либо другим ботом, то BotFather скажет об этом и попросит придумать другое.

Когда будет введено имя, которое подходит под все критерии, BotFather выдаст специальный API-токен: это своего рода серийный номер, который понадобится в будущем для настройки и управления созданным ботом.



API-токен следует хранить в секрете, как и любой пароль, поэтому на картинке он скрыт.

Программирование бота

Введение

Python был выбран языком программирования на котором будет написан бот, потому что это удобный и мощный язык для которого существует огромное количество удобных библиотек, а также библиотек для работы с API Телеграма.

Для работы с API Телеграма была выбрана библиотека «**aiogram**», потому что она предоставляет удобный интерфейс для работы с функционалом Телеграм ботов. Также эта библиотека внутренне использует библиотеку «**asyncio**», которая позволяет писать асинхронный код. Благодаря этому, бота сможет использовать несколько человек одновременно.

1. Инициализация бота

Для начала необходимо инициализировать объект «**Bot**» из библиотеки «**aiogram**»:

```
bot = Bot(token=TELEGRAM_TOKEN, default=DefaultBotProperties(parse_mode="html"))
```

Значением параметра «**token**» является API токен, который был получен от BotFather ранее.

Затем нужно инициализировать объект «**Dispatcher**», он является главным «маршрутизатором» всех событий, которые будет получать бот (например получение сообщения от пользователя):

```
dp = Dispatcher()
```

Далее в асинхронной функции «**main**» запускаем бота вызовом метода «**start_polling**» у объекта «**Dispatcher**», передав в неё экземпляр объекта «**Bot**».

```
async def main() → None:  
    await dp.start_polling(bot)
```

Затем необходимо инициализировать библиотеку логирования «**logging**» и вызвать асинхронную функцию «**main**» с помощью функции «**run**» из библиотеки «**asyncio**»:

```
if __name__ == "__main__":  
    logging.basicConfig(level=logging.INFO, stream=sys.stdout)  
    asyncio.run(main())
```

В консоль выведется информация об успешном запуске бота:

```
INFO:aiogram.dispatcher:Start polling  
INFO:aiogram.dispatcher:Run polling for bot @orangebtw_bot id=7858480757 - 'botik'
```

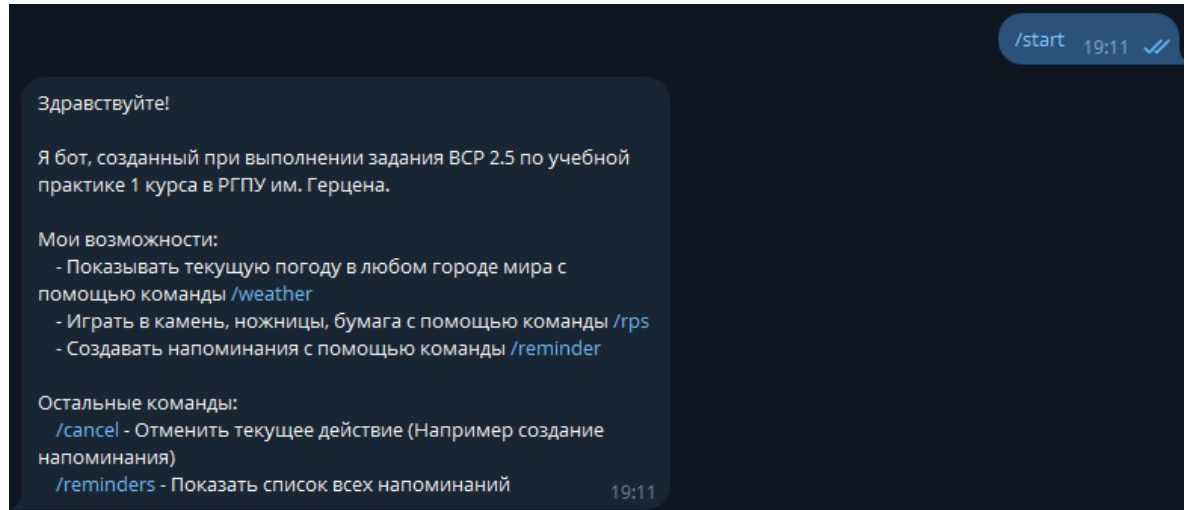
2. Получение сообщений от пользователя

Чтобы начать получать сообщения от пользователя, нужно «подписаться» на получение события «**message**». Для этого необходимо создать асинхронную функцию, которая будет вызываться при получении события. Затем необходимо связать эту функцию с событием «**message**» с помощью экземпляра объекта «**Dispatcher**»:

```
@dp.message(CommandStart())  
async def command_start_handler(message: Message) → None:  
    await message.answer(START_MESSAGE)
```

На картинке выше, мы создали асинхронную функцию «**command_start_handler**» с аргументом типа «**Message**» и подписали её на событие «**message**» с определённым фильтром «**CommandStart**». С фильтром «**CommandStart**» эта функция вызовется только тогда, когда пользователь отправит сообщение с командой «**start**».

В теле функции мы вызываем асинхронный метод «**answer**» у аргумента «**message**» типа «**Message**», передав в него текст приветственного сообщения. Вот как это будет выглядеть от лица пользователя:



3. Реализация предоставления текущей погоды

Наш бот должен уметь предоставлять пользователю текущую погоду в любом городе мира. Для этого мы будем использовать сервис «**OpenWeatherMap**», который позволяет бесплатно в небольших количествах использовать их API.

Для удобства, мы будем использовать библиотеку «**pyowm**», которая облегчит взаимодействие с API сервиса «**OpenWeatherMap**».

Сперва инициализируем объект «**OWM**» из библиотеки «**pyowm**»:

```
owm = OWM(OWM_TOKEN, config=OWM_CONFIG)
```

Первым аргументом конструктора объекта «**OWM**» является API токен сервиса «**OpenWeatherMap**», который можно получить, зарегистрировавшись на их официальном сайте.

Затем инициализируем объект «**WeatherManager**», вызвав метод «**weather_manager**» у созданного выше экземпляра объекта «**OWM**».

Далее создадим функцию для обработки события команды «**/weather**»:

```
@dp.message(Command("weather"))
async def command_weather_handler(message: Message, command: CommandObject) → None:
```

В начале тела функции «`command_weather_handler`» сделаем обработку случая, когда пользователь ввёл команду без указания города.

```
if command.args is None or command.args.isspace():
    await message.answer("Вы не указали город.\nПример использования: /weather Санкт-Петербург")
    return
```

В случае, когда пользователь указал город сделаем запрос к сервису «`OpenWeatherMap`» для получения текущей погоды в этом городе:

```
observation = mgr.weather_at_place(command.args)
```

В случае, если указанный пользователем город не будет найден (например, если введённое название города не существует), необходимо уведомить его об этом:

```
try:
    observation = mgr.weather_at_place(command.args)
except OwmNotFoundError:
    await message.answer("Город не найден :(")
    return
```

Если всё прошло успешно, отправим пользователю сообщение с информацией о погоде, текущей температуре, минимальной температуре, максимальной температуре, эффективной температуре (температура, которую ощущает человек при заданной скорости ветра и влажности воздуха), влажности и скорости ветра:

```
w: Weather = observation.weather
l: Location = observation.location

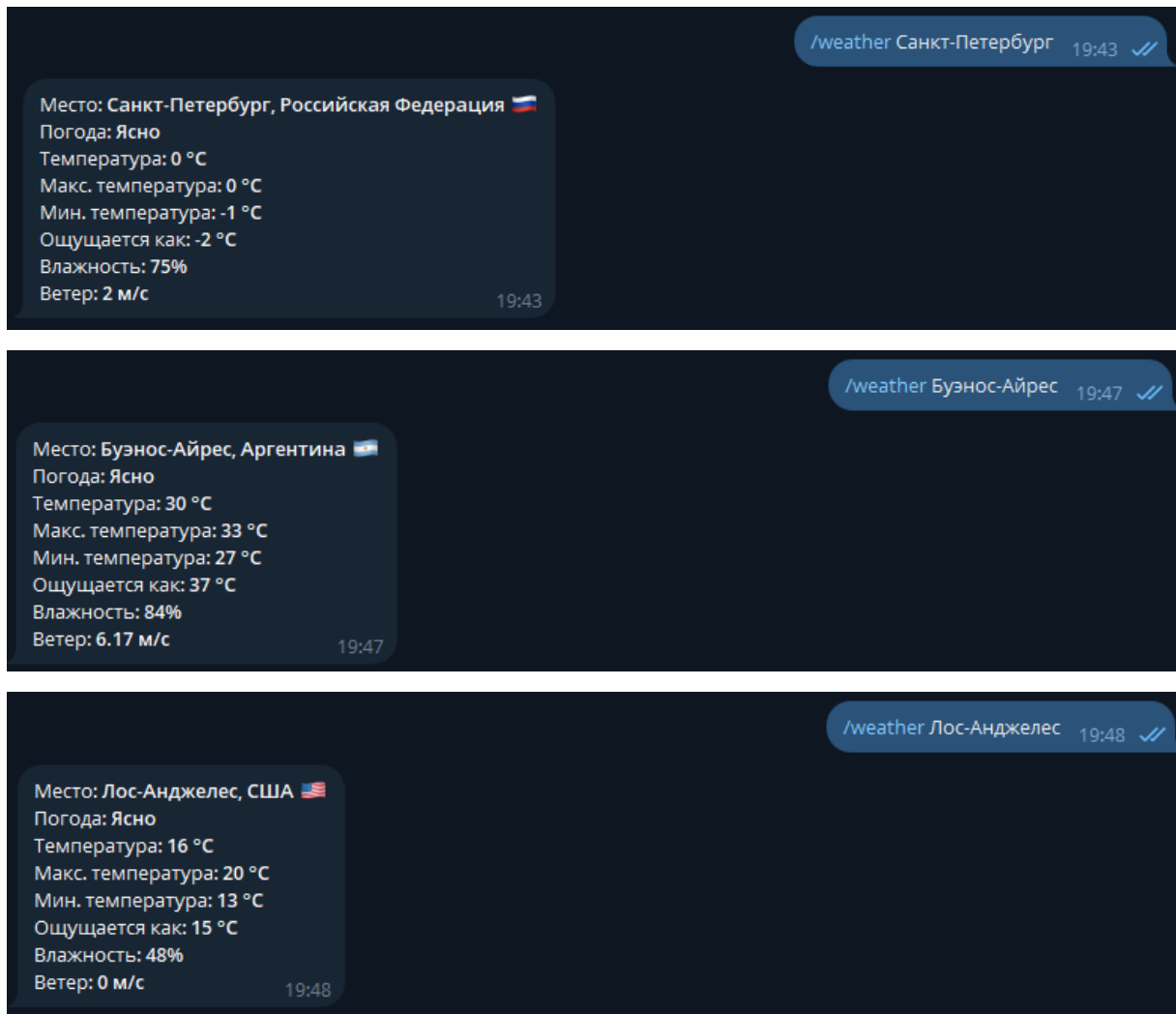
detailed_status: str = w.detailed_status
country_name = COUNTRY_CODE_TO_COUNTRY_NAME[l.country]
temperature = w.temperature(unit="celsius")
temp = int(round(temperature["temp"]))
temp_max = int(round(temperature["temp_max"]))
temp_min = int(round(temperature["temp_min"]))
feels_like = int(round(temperature["feels_like"]))
humidity = w.humidity

wind = w.wind()
wind_speed = wind["speed"]

flag = COUNTRY_CODE_TO_FLAG[l.country]

await message.answer((
    f"Место: <b>{l.name}, {country_name} {flag}</b>\n"
    f"Погода: <b>{detailed_status.capitalize()}</b>\n"
    f"Температура: <b>{temp} °C</b>\n"
    f"Макс. температура: <b>{temp_max} °C</b>\n"
    f"Мин. температура: <b>{temp_min} °C</b>\n"
    f"Ощущается как: <b>{feels_like} °C</b>\n"
    f"Влажность: <b>{humidity}%</b>\n"
    f"Ветер: <b>{wind_speed} м/с</b>"
))
```


Вот как это выглядит от лица пользователя:



4. Реализация игры «камень, ножницы, бумага»

Научим нашего бота играть в игру «камень, ножницы, бумага». Для этого создадим класс «**RpsVariant**», который будет означать вариант ответа в игре:

```
class RpsVariant(Enum):
    ROCK = 1
    PAPER = 2
    SCISSORS = 3

    def from_name(name: str) → Self | None:
        match name:
            case "камень": return RpsVariant.ROCK
            case "бумага": return RpsVariant.PAPER
            case "ножницы": return RpsVariant.SCISSORS
            case _: return None

    @property
    def name(self) → str:
        match self:
            case RpsVariant.ROCK: return "камень"
            case RpsVariant.PAPER: return "бумага"
            case RpsVariant.SCISSORS: return "ножницы"

    @property
    def name_acusative(self) → str:
        match self:
            case RpsVariant.ROCK: return "камень"
            case RpsVariant.PAPER: return "бумагу"
            case RpsVariant.SCISSORS: return "ножницы"
```

Затем объявим список со всеми возможными вариантами в игре (он понадобится нам позже, для случайного выбора ботом варианта):

```
RPS_VARIANTS = [RpsVariant.ROCK, RpsVariant.PAPER, RpsVariant.SCISSORS]
```

Далее объявим словарь, где в качестве ключа будет предмет, который выигрывает тот предмет, что является значением ключа:

```
RPS_MAP = {
    RpsVariant.ROCK: RpsVariant.SCISSORS,
    RpsVariant.PAPER: RpsVariant.ROCK,
    RpsVariant.SCISSORS: RpsVariant.PAPER
}
```

Теперь объявим функцию для обработки события ввода команды «rps» пользователем:

```
@dp.message(Command("rps"))
async def command_rps_handler(message: Message, command: CommandObject) → None:
```

Далее в начале тела функции добавим обработку случая, когда пользователь не указал никакой предмет:

```
if not command.args or command.args.isspace():
    await message.answer("Вы не выбрали предмет.\nПример использования: /rps камень")
    return
```

Затем преобразуем аргумент пользователя в экземпляр класса «RpsVariant»:

```
user_variant = RpsVariant.from_name(command.args.lower())
```

Добавим обработку случая, когда пользователь неправильно ввёл название предмета:

```
if user_variant is None:
    await message.answer((
        f"Такого варианта нет.\n"
        f"Возможные варианты: {RpsVariant.ROCK.name}, {RpsVariant.SCISSORS.name}, {RpsVariant.PAPER.name}."
    ))
    return
```

Случайно выберем предмет из списка вариантов в качестве выбора бота:

```
variant = random.choice(RPS_VARIANTS)
```

Затем с помощью словаря «RPS_MAP» сообщим пользователю кто победил:

```
if RPS_MAP.get(user_variant) == variant:
    await message.answer(f"Вы выиграли! Я выбрал <b>{variant.name_acusative}</b>.")
elif RPS_MAP.get(variant) == user_variant:
    await message.answer(f"Вы проиграли! Я выбрал <b>{variant.name_acusative}</b>.")
else:
    await message.answer(f"Ничья! Я выбрал <b>{variant.name_acusative}</b>.")
```

5. Реализация напоминаний

Наш бот должен уметь создавать напоминания и отправлять уведомления пользователям по истечении срока напоминания. Для этого нам понадобится библиотека для работы с базами данных «sqlite3».

Создадим функцию для создания базы данных напоминаний для каждого пользователя:

```
def create_reminders_db(chat_id: int) → None:
    database_file = f"./databases/{chat_id}.db"

    if os.path.exists(database_file):
        os.remove(database_file)

    with sqlite3.connect(database_file) as conn:
        conn.execute("CREATE TABLE Reminders (id INTEGER PRIMARY KEY AUTOINCREMENT, expires_in INTEGER, content TEXT)")
```

Добавим вызов функции создания базы данных напоминаний в функцию обработки команды «start». Вызов этой функции должен осуществляться на другом потоке, для этого воспользуемся функцией «to_thread» из библиотеки «asyncio»:

```
await asyncio.to_thread(create_reminders_db, message.chat.id)
```

Далее объявим класс «Reminder», который будет хранить информацию об напоминании:

```
@dataclass
class Reminder:
    id: int
    date: int
    text: str
    active: bool = True
```

Далее создадим словарь, в который будут записываться напоминания всех пользователей (это нужно, чтобы уменьшить количество запросов к базе данных, тем самым ускорив работу бота):

```
REMINDERS: dict[int, list[Reminder]] = {}
```

Создадим функцию для создания напоминания в базе данных:

```
def add_reminder(chat_id: int, text: str, date: int) → None:
    database_file = f"./databases/{chat_id}.db"

    with sqlite3.connect(database_file) as conn:
        cur = conn.cursor()
        cur.execute("INSERT INTO Reminders (expires_in, content) VALUES (?, ?)", (date, text))

    if not chat_id in REMINDERS:
        REMINDERS[chat_id] = []

    REMINDERS[chat_id].append(Reminder(cur.lastrowid, date, text))
```

Далее объявим класс «**NewReminderState**», который будет хранить состояние диалога создания нового напоминания с пользователем:

```
class NewReminderState(StatesGroup):  
    text = State()  
    date = State()
```

Далее создадим функцию для обработки ввода команды «/reminder» пользователем:

```
@reminder_router.message(Command("reminder"))  
async def command_new_reminder_handler(message: Message, state: FSMContext) → None:  
    current_state = await state.get_state()  
    if current_state is not None:  
        await state.clear()  
  
    await state.set_state(NewReminderState.text)  
    await message.answer("Введите текст напоминания", reply_markup=ReplyKeyboardRemove())
```

Затем создадим функцию для сохранения введенного пользователем текста напоминания:

```
@reminder_router.message(NewReminderState.text)  
async def new_reminder_text(message: Message, state: FSMContext) → None:  
    await state.update_data(text=message.text)  
    await state.set_state(NewReminderState.date)  
    await message.answer((  
        "Теперь введите время и дату напоминания в формате: <b>ЧЧ:ММ ДД/ММ/ГГГГ</b>\n\n"  
        "Примечание:\n"  
        "<i>Дата не обязательна, по умолчанию - текущий день</i>"  
    ))
```

Теперь создадим функции для сохранения введенной пользователем даты напоминания:

```
@reminder_router.message(NewReminderState.date)  
async def new_reminder_date(message: Message, state: FSMContext) → None:
```

Разобьём введенную пользователем время и дату на напоминания в часы, минуты, год, месяц и день, и преобразуем их в числа (код не содержит проверку даты на ошибки для экономии места):

```
date_time = message.text.split(" ")
t = date_time[0].split(":")

hours = int(t[0])
minutes = int(t[1])

today = datetime.today()

day = today.day
month = today.month
year = today.year

if len(date_time) > 1:
    d = date_time[1].split('/')

    day = int(d[0])
    month = int(d[1])
    year = int(d[2])
```

Далее получим текущую дату и преобразуем её и дату напоминания в формат «Unix-время»:

```
now = datetime.now()
now_timestamp = calendar.timegm(now.timetuple())

date = datetime(year, month, day, hours, minutes)
timestamp = calendar.timegm(date.timetuple())
```

Затем добавим напоминание в базу данных, вызвав функцию «add_reminder»:

```
data = await state.update_data(date=timestamp)
await asyncio.to_thread(add_reminder, message.chat.id, data['text'], data['date'])
```

Далее сообщим пользователю об успешном создании напоминания и очистим состояние диалога создания напоминания:

```
await message.answer(f"Напоминание успешно создано!")
await state.clear()
```

Создадим функцию для обработки ввода пользователем

команды «reminders»:

```
@dp.message(Command("reminders"))
async def command_reminders_handler(message: Message) → None:
    reminders = await asyncio.to_thread(get_reminders, message.chat.id)
    if len(reminders) == 0:
        await message.answer("У вас нет напоминаний.")
        return

    text = "Список напоминаний:\n"
    for i, reminder in enumerate(reminders, start=1):
        date = datetime.fromtimestamp(reminder.date, UTC).strftime("%H:%M, %d/%m/%Y")
        status = '\u231B' if reminder.active else '\u2705'
        text += f"{i}. {status} [{date}] {reminder.text}\n\n"

    await message.answer(text)
```

Далее создадим функцию, которая будет отправлять напоминания пользователям при истечении времени.

```
async def check_reminders_expiration() → None:
    now = datetime.now()
    timestamp = calendar.timegm(now.timetuple())

    for chat_id, reminders in REMINDERS.items():
        for reminder in reminders:
            if reminder.active and timestamp > reminder.date:
                reminder.active = False

                date = datetime.fromtimestamp(reminder.date, UTC).strftime("%H:%M, %d/%m/%Y")
                await bot.send_message(chat_id, f"Напоминание:\n\n{reminder.text}\n\n{date}")
```

Затем с помощью библиотеки «apscheduler», будем асинхронно вызывать эту функцию каждые 10 секунд:

```
scheduler = AsyncIOScheduler()
scheduler.add_job(check_reminders_expiration, IntervalTrigger(seconds=10))
scheduler.start()
```

Заключение

В данном отчёте были показаны наиболее важные этапы создания Telegram-бота, а также прокомментированы фрагменты его исходного кода. Благодаря коду на языке Python, в дальнейшем бота можно будет более гибко настраивать, добавлять новые функции, тем самым улучшая его функционал