asyncVK – асинхронный фреймворк для создания ботов ВК. Преимущества: удобство, скорость выигрываемая за счёт асинхронности. Бот создаётся за счёт пяти структурных единиц:

1) Bot – это самая главная структурная единица. Это собственно сам бот, который подаёт ивенты обработчикам.  
2) Handler – эта структурная единица отвечает за обработку ивентов.   
3) Dispatcher – эта структурная единица отвечает за взаимодействие с ВК (ответы на сообщения, добавление комментариев). Она автоматически настраивается хандлерами.  
4) Keyboard – это второстепенная структурная единица. Она отвечает за создание кнопок в ВК.  
5) Condition (Condition, And, Or) – эта структурная единица отвечает за условия. С помощью неё можно строить сложные условия для хандлеров.

Как работать с библиотекой? Легко и интуитивно понятно! Для начала нужно импортировать саму библиотеку и создать бота.

from asyncVK import Handler, Dispatcher, Bot, run\_polling  
from asyncVK.condition import Condition, And, Or  
import asyncVK.keyboard as keyboard

TOKEN = "access\_token"  
GROUP\_ID = 182801600  
  
bot = Bot(TOKEN, GROUP\_ID)

Теперь мы можем запустить бота на LongPoll API:

if \_\_name\_\_ == "\_\_main\_\_":  
 run\_polling(bot)

Сейчас бот запущен, но ни на что не реагирует. Чтобы это исправить нам нужно создать обработчик и добавить его в бота. Как это сделать? Вот так:

@bot.handle  
@Handler.on.message\_new(Condition(command="привет!"), is\_lower=True)  
async def handler(dispatcher: Dispatcher):   
 await dispatcher.send\_message("Hi!")

В примере выше мы создали обработчик новых сообщений с помощью декоратора @Handler.on.message\_new и добавили его в бота с помощью декоратора @bot.handle  
 Вместо декоратора @bot.handle можно конечно просто прописать bot.handle(handler)

Как работают хандлеры (обработчики)? Когда мы засовываем экземпляр класса Bot в функцию run\_polling, мы как бы активируем метод bot.run\_polling. bot.run\_polling это бесконечный цикл. В экземпляре класса Bot есть список всех хандлеров, которые мы создали и добавили в него. И когда приходит какой-то ивент (событие), то этот цикл пересылает это событие всем хандлерам. И потом если условие истинно, то активируется функция, из которой мы сделали хандлер.

Что делает эта асинхронная функция? Она на сообщение «привет!» (в любом регистре) будет отвечать в тот же чат сообщением «Hi!». Как сделать чтобы она ответила не в тот же чат, а в ЛС? Легко! Заменить

await dispatcher.send\_message("Hi!")

на

await dispatcher.answer("Hi!")

Как строить условия? Какие условия можно построить? Строить условия легко, и можно построить абсолютно любые условия! Например мы хотим, чтобы бот отвечал на привет. В таком случае мы пишем:

Condition(command="привет")

Или мы хотим, чтобы бот отвечал на привет или если peer\_id равен 2000000001. В таком случае мы пишем:

Condition(command="привет", peer\_id=2000000001)

То есть прописывая дополнительные условия в Condition, мы как бы делаем ‘if command == “привет” or peer\_id == 2000000001‘. Также можно аналогично прописать вот так:

Or(Condition(command="привет"), Condition(peer\_id=2000000001))

Но лучше так не делать, лучше подобные условия прописывать без Or.

А что если мы хотим, чтобы бот отвечал либо если ему написали «привет», либо если в сообщении есть строка «а» и написал это пользователь с id 386746383. Тоже легко! Вот так:

Or(  
 Condition(command="привет"),   
 And(Condition(contains\_command="а"), Condition(user\_id=386746383))  
)

И так, давайте разбирать как же строить так любые запросы. Если мы пропишем несколько аргументов в Condition, то это будет ИЛИ (or). Если же мы засунем несколько условий в And, то тут условие будет истинным если все условия в And истинны, то есть это И (and). Если же мы засунем несколько условий в Or, то условие будет истинным если истинно хотя бы одно условие в нём, то есть это ИЛИ (or). В And и Or можно засовывать как и Condition, так и другие And и Or.

Вот все аргументы Condition:  
 command – проверяет на равенство текста (если сообщение, то текста сообщения, если комментарий, то текста комментария) с этим аргументом.  
 contains\_command – проверяет на то, есть ли строка contains\_command в тексте.  
 user\_id – проверяет на равенство id пользователя, инициировавшего событие, и этим аргументом.  
 peer\_id – проверяет на равенство id чата с этим аргументом.  
 post\_id – проверяет на равенство id записи на стене/id записи в обсуждениях с этим аргументом  
 owner\_id – проверяет на равенство id сообщества, где произошло событие (если событие было в сообществе)

Вот весь список хандлеров:  
 Handler.on.message\_new – новое сообщение.  
 Handler.on.message\_edit – редактирование сообщения.  
 Handler.on.wall\_reply\_new – новые комментарий на стене.  
 Handler.on.wall\_reply\_edit – редактирование комментария на стене.  
 Handler.on.wall\_post\_new – новый пост на стене.  
 Handler.on.board\_post\_new – новый комментарий в обсуждениях.  
 Handler.on.board\_post\_edit – редактирование комментария в обсуждениях.  
  
 Списки их аргументов абсолютно идентичны.

Возможности диспетчера:

dispatcher.answer – ответить в ЛС. Можно активировать при любом событии (отправит сообщение инициатору события (пользователю, запустившего событие)). Список аргументов:   
 text – текст сообщения.  
 attachment – вложение сообщения (в виде части ссылки такого рода: от ссылки <https://vk.com/id386746383?z=photo386746383_457256628%2Falbum386746383_0> берём только photo386746383\_457256628 и передаём это в качестве аргумента).   
 keyboard – кнопки ВК.

dispatcher.send\_message – ответить в том же чате. Список аргументов идентичен с answer.

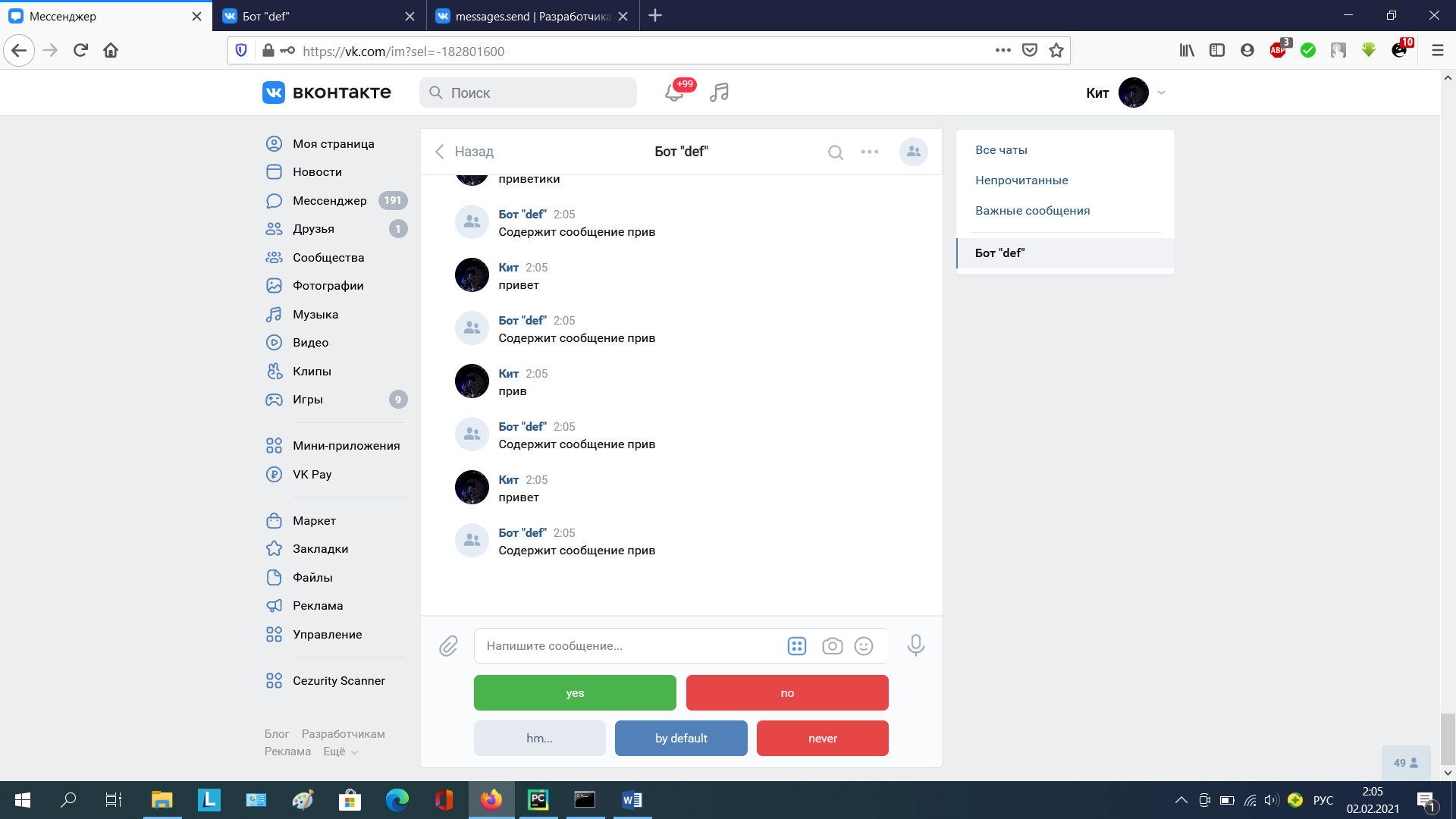
dispatcher.send\_comment – ответить в комментариях. Список аргументов идентичен с answer, но аргумент keyboard отсутствует.

dispatcher.mark\_as\_read – пометить сообщение как «прочитанное». Никаких аргументов не принимает.

dispatcher.set\_typing\_status – установить статус на набор текста / запись голосового сообщения. Принимает один аргумент: typing\_status. Его значение по умолчанию “typing” (набор текста). Можно изменить на “audiomessage” – запись голосового сообщения.

Как создать кнопки? Тоже несложно! Вот пример:

@bot.handle  
@Handler.on.message\_new(Condition(contains\_command="прив"), is\_lower=True)  
async def handler(dispatcher: Dispatcher):  
 buttons = keyboard.get\_keyboard([  
 [  
 ("yes", "positive"),  
 ("no", "negative")  
 ],  
 [  
 ("hm...", "default"),  
 ("by default", "primary"),  
 ("never", "negative")  
 ]  
 ])  
  
 await dispatcher.send\_message("Содержит сообщение прив", keyboard=buttons)



По умолчанию кнопки создаются обычными и одноразовыми. Чтобы сделать их многоразовыми пропишите one\_time=False, а если хотите сделать их инлайновыми, то пропишите inline=True. К примеру:

buttons = keyboard.get\_keyboard([  
 [  
 ("yes", "positive"),  
 ("no", "negative")  
 ],  
 [  
 ("hm...", "default"),  
 ("by default", "primary"),  
 ("never", "negative")  
 ]  
], inline=True)

Тогда эти кнопки будут в сообщении (инлайновыми).

Можно строить любые запросы, даже если этого не предполагает отсутствие метода в диспетчере:

@bot.handle  
@Handler.on.message\_new(Condition(contains\_command="прив"), is\_lower=True)  
async def handler(dispatcher: Dispatcher):  
 result = await bot.execute("messages.send", peer\_id=dispatcher.peer\_id,   
 message="okay", random\_id=0)  
 print(result)

В этом примере мы на новое сообщение, содержащее «прив» отвечаем «okay» нашим построенным запросом. peer\_id же берём из диспетчера. Какие параметры можно высунуть из диспетчера? Можно высунуть: token, user\_id, peer\_id, post\_id (если событие это новая запись на стене, новый комментарий на стене или в обсуждении), owner\_id (если событие было внутри группы, то owner\_id это id группы), event (объект, в котором содержится вся информация о событии) и text (если к примеру событие это новое сообщение, то text это текст сообщения, если это к примеру новый комментарий, то text это текст комментария и т.д.)

Если вы хотите выполнить сразу несколько запросов асинхронно, то можно просто воспользовать библиотекой asyncio. К примеру:

@bot.handle  
@Handler.on.message\_new(Condition(contains\_command="прив"), is\_lower=True)  
async def handler(dispatcher: Dispatcher):  
 tasks = [asyncio.create\_task(dispatcher.mark\_as\_read()),  
 asyncio.create\_task(dispatcher.set\_typing\_status()),  
 asyncio.create\_task(asyncio.sleep(9))]  
  
 await asyncio.gather(\*tasks)  
 await dispatcher.send\_message("okay")  
  
  
@bot.handle  
@Handler.on.message\_new(contains\_command="а", is\_lower=True)  
async def handler(dispatcher: Dispatcher):  
 await dispatcher.send\_message("Б!")

Хандлер, обрабатывающий сообщение, где есть строка «прив» сперва пометит сообщение как прочитанное, потом установит статус «печатает…» и через 9 секунд отправит сообщение okay и всё это асинхронно. P.S. хандлеры друг друга не блокируют, так что во время работы первого хандлера вы можете написать «а» и бот ответит «Б!», несмотря на работу первого хандлера.

По-мимо этого можно делать хандлеры не для условий, а для всего события целиком. Например:

@bot.handle  
@Handler.on("message\_new")  
async def handler(dispatcher: Dispatcher):  
 if dispatcher.text.lower() == "abs":  
 await dispatcher.send\_message("peer")  
 elif dispatcher.text.lower() == "help me":  
 await dispatcher.send\_message("no")

Этот хандлер будет обрабатывать все события типа «message\_new». В данном случае он на «abs» будет отвечать «peer», а на «help me» будет отвечать «no». И также регистр сообщения не важен, ибо мы применили метод lower.

Так можно делать обработчики для любых событий. К примеру обработчик для новых комментариев:

@bot.handle  
@Handler.on("wall\_reply\_new")  
async def handler(dispatcher: Dispatcher):  
 if dispatcher.text.lower() == "nice":  
 await dispatcher.send\_comment("ok")  
 elif dispatcher.text.lower() == "not bad":  
 await dispatcher.send\_comment("no, very bad!")

Какой обработчик использовать? Для условий или для всего события целиком? Если вам нужно сделать обработчик для простых команд (ответить на то этим и что-то в этом роде), то лучше все эти команды прописать в обработчике события, в данном случае это будет @Handler.on(“message\_new”) ведь нам нужно отвечать на сообщения. А если же команды сложные, а не простые ответы с какими-то дополнительными действиями, то лучше их прописать в обработчике условия. К примеру нам нужно, чтобы при сообщении «статистика» бот получил статистику откуда-то, рассортировал и отфильтровал её и потом отправил. Такое лучше прописывать в обработчике условия, в данном случае @Handler.on.message\_new(Condition(command=”статистика”))  
 Но нужно смотреть на код в целом, ибо иногда может пригодится сделать исключение и написать сложную команду в обработчик события, а простую в обработчик условия. То есть выбор должен зависеть от ситуации и структуры вашего кода.

В хандлерах и в самом боте не предусмотрена синхронизация. Поэтому если вы будете пользоваться асинхронной реализацией, к примеру какой-то базы-данных, будет состояние гонки. А пользоваться синхронными реализациями базы-данных плохая идея, это снизит скорость бота. Такая структура позволяет боту быть очень быстрым. Но в фреймворке есть реализация асинхронной базы-данных с синхронизацией, которой если вы будете правильно пользоваться, то состояния гонки не будет и бот будет оставаться таким же быстрым. Пример бота с этой реализацией бд:

from asyncVK.asyncDB import SQLite

db = SQLite("data.db")  
bot = Bot(TOKEN, GROUP\_ID)

async def create\_db():  
 async with db:  
 await db.execute("""  
 CREATE TABLE IF NOT EXISTS profile (  
 user\_id INTEGER,  
 money INTEGER  
 )  
 """)  
  
  
@bot.handle  
@Handler.on("message\_new")  
async def handler(dispatcher: Dispatcher):  
 if dispatcher.text.lower() == "create db" and dispatcher.user\_id == OWNER\_ID:  
 await create\_db()  
 await dispatcher.send\_message("db was created!")  
  
 elif dispatcher.text.lower() == "register":  
 async with db:  
 await db.execute(f"""  
 INSERT INTO profile   
 VALUES ({dispatcher.user\_id}, 0)  
 """)  
  
 await dispatcher.send\_message("you are was registered!")  
  
  
@bot.handle  
@Handler.on.message\_new(Condition(command="click"), is\_lower=True)  
async def handler(dispatcher: Dispatcher):  
 async with db:  
 await db.execute(f"""  
 UPDATE profile  
 SET money=money+1  
 WHERE user\_id={dispatcher.user\_id}  
 """)  
  
 state = await db.execute(f"""  
 SELECT money  
 FROM profile  
 WHERE user\_id={dispatcher.user\_id}  
 """)  
  
 money = state[0][0]  
 await dispatcher.send\_message(f"Money: {money}")

OWNER\_ID это константа, которая должна ваш ID в ВК, это условие запрещает создавать база-данных кому-либо кроме вас командой.   
 Что делает «async with db?». «async with db» ждёт пока база-данных откроется для запросов, потом закрывает базу-данных для запросов и как все ваши запросы прошли к базе, она опять открывает базу-данных для запросов.   
 Метод «db.execute» отправляет ваш запрос к базе-данных.

Весь код целиком для старта:

from asyncVK import Handler, Dispatcher, Bot, run\_polling  
import asyncVK.keyboard as keyboard  
  
  
TOKEN = "access\_token"  
GROUP\_ID = 182801600  
  
bot = Bot(TOKEN, GROUP\_ID)  
  
  
@bot.handle  
@Handler.on.message\_new(Condition(contains\_command="прив"), is\_lower=True)  
async def handler(dispatcher: Dispatcher):  
 buttons = keyboard.get\_keyboard([  
 [  
 ("yes", "positive"),  
 ("no", "negative")  
 ],  
 [  
 ("hm...", "default"),  
 ("by default", "primary"),  
 ("never", "negative")  
 ]  
 ], inline=True)  
  
 await dispatcher.send\_message("Содержит сообщение прив", keyboard=buttons)

result = await bot.execute("messages.send", peer\_id=dispatcher.peer\_id,   
 message="okay", random\_id=0)  
 print(result)

if \_\_name\_\_ == "\_\_main\_\_":  
 run\_polling(bot)