asyncVK is an asynchronous library for creating VK bots. Advantages: convenience, speed gained due to asynchrony. The bot is created at the expense of five structural units:

1) Bot – it is the most important structural unit. This is the bot itself, which serves events to the handlers.  
2) Handler – this structural unit is responsible for processing events.  
3) Dispatcher – this structural unit is responsible for interaction with VK (replies to messages, adding comments). It is automatically configured by handlers.  
4) Keyboard – it is a minor structural unit. She is responsible for creating buttons in VK.  
5) Condition (Condition, And, Or) – this structural unit is responsible for conditions. With it, you can build complex conditions for handlers.

How to work with the library? Easy and intuitive! First you need to import the library itself and create a bot.

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)

Now we can launch the bot using the LongPoll API:

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

Now the bot is running, but it doesn't react to anything. To fix this, you need to create a handler and add it to the bot. How to do it? Like this:

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

In the example above, we created a new message handler using the @Handler.on.message\_new decorator and added it to the bot using the @bot.handle decorator.  
 Instead of the @bot.handle decorator, you can of course just add bot.handle(handler)

How do handlers work? When we push an instance of the Bot class into the run\_polling function, we kind of activate the bot.run\_polling method. bot.run\_polling is an endless loop. An instance of the Bot class has a list of all the handlers that we created and added to it. And when an event comes, this cycle sends this event to all handlers. And then if the condition is true, then the function from which we made the handler is activated.

What does this asynchronous function do? She to the message "привет!" (in any case) will reply to the same chat with the message "Hi!" How to make her answer not in the same chat, but in PM? Easy! Replace:

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

to:

await dispatcher.answer("Hi!")

How to build conditions? What conditions can you build? It is easy to build conditions, and you can build absolutely any conditions! For example, we want the bot to reply hi. In this case, we write:

Condition(command="привет")

Or we want the bot to respond to hello or if the peer\_id is 2000000001. In this case, we write:

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

That is, by writing additional conditions in Condition, we kind of do ‘if command == “hello” or peer\_id == 2000000001’. You can also similarly write like this:

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

But it is better not to do this, it is better to prescribe such conditions without Or.

And what if we want the bot to answer either if they wrote “привет” to it, or if the message contains the line “a” and the user with id 386746383 wrote it. It's also easy! Like this:

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

And so, let's analyze how to build any queries like that. If we write several arguments in Condition, then it will be OR. If we put several conditions in And, then the condition will be true if all conditions in And are true, that is, this is AND. If we put several conditions in Or, then the condition will be true if at least one condition in it is true, that is, it is OR (or). In And and Or, you can stick both Condition and other And and Or.

Here are all the Condition parameters:  
 command – checks for equality of the text (if a message, then the text of the message, if a comment, then the text of the comment) with this argument.  
 contains\_command – checks if the contains\_command string is in the text.  
 user\_id – checks for equality between the id of the user who raised the event and this argument.  
 peer\_id – checks for equality of chat id with this argument.  
 post\_id – checks for equality of wall post id / post id in discussions with this argument. owner\_id – checks for equality of community id where the event occurred (if the event was in the community).

Here is the entire list of handlers:  
 Handler.on.message\_new – new message handler.  
 Handler.on.message\_edit – edit message handler.  
 Handler.on.wall\_reply\_new – new comment on the wall handler.  
 Handler.on.wall\_reply\_edit – edit comment on the wall handler.  
 Handler.on.wall\_post\_new – new post on the wall handler.  
 Handler.on.board\_post\_new – new post on the board handler.  
 Handler.on.board\_post\_edit – edit post on the board handler.  
  
 Their lists of arguments are exactly the same.

Dispatcher capabilities:

dispatcher.answer – answer in PM. Can be activated on any event (will send a message to the event initiator (the user who triggered the event)). Argument list:  
 text – message text.  
 attachment – message attachment (as part of a link of this kind: from the link https://vk.com/id386746383?z=photo386746383\_457256628%2Falbum386746383\_0 we take only photo386746383\_457256628 and pass it as an argument).   
 keyboard – VK buttons.

dispatcher.send\_message – reply in the same chat. The argument list is identical with answer.

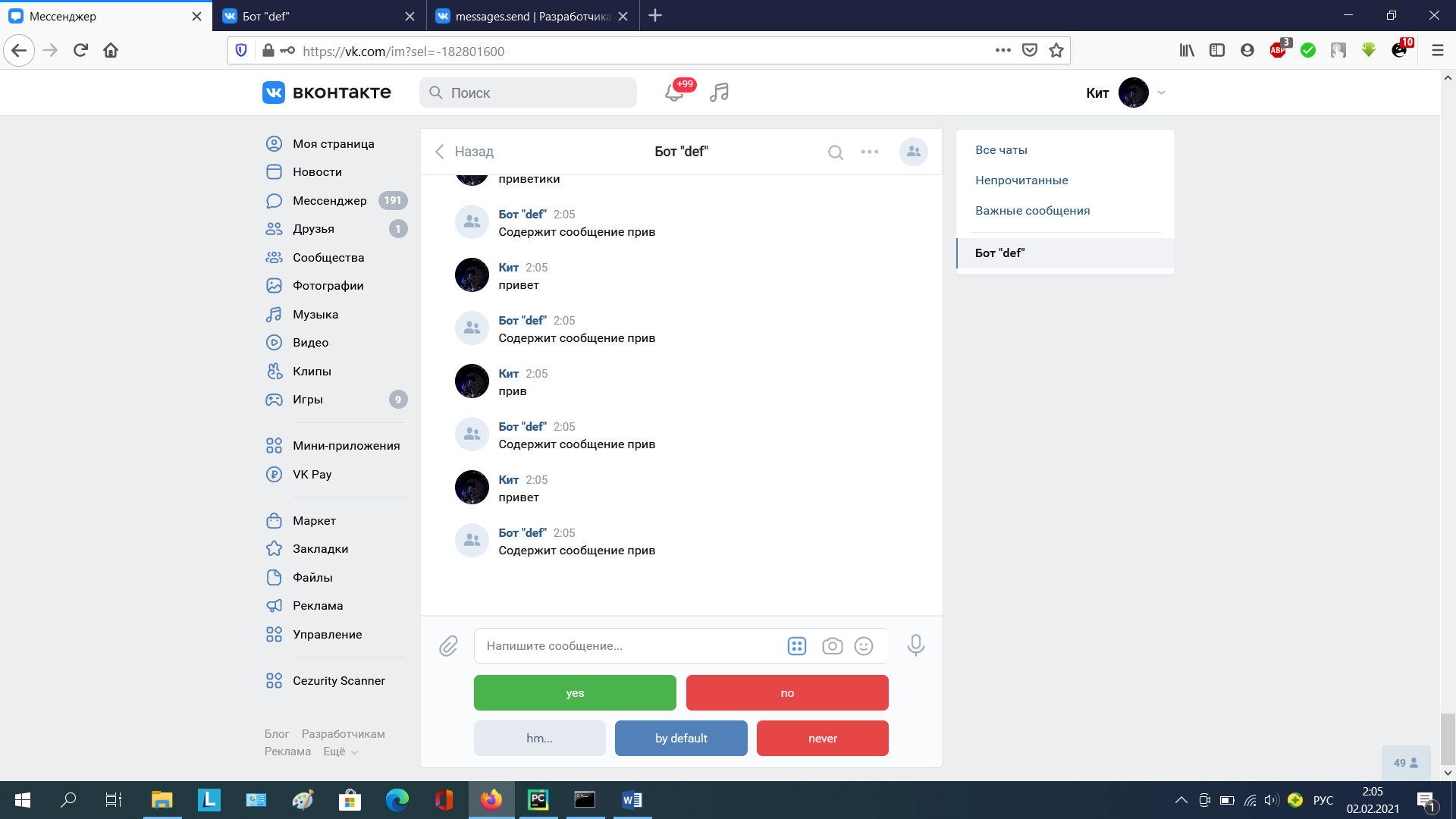
dispatcher.send\_comment – reply in the comments. The argument list is identical to answer, but the keyboard argument is missing.

dispatcher.mark\_as\_read – mark the message as read. Doesn't take any arguments.

dispatcher.set\_typing\_status – set the status to typing / recording a voice message. It takes one argument: typing\_status. Its default value is “typing”. Can be changed to “audiomessage” - recording a voice message.

How do I create buttons? It's not difficult too! Here's an example:

@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)



By default, buttons are created as regular and disposable. To make them reusable, write one\_time = False, and if you want to make them inline, write inline = True. For example:

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

Then these buttons will be in the message (inline).

You can build any queries, even if this method is not present in the dispatcher:

@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)

In this example, we reply with “okay” to a new message containing "прив" with our constructed query. peer\_id is taken from the dispatcher. What parameters can be poked out of the dispatcher? You can stick it out: token, user\_id, peer\_id, post\_id (if the event is a new post on the wall, a new comment on the wall or in a discussion), owner\_id (if the event was inside a group, then owner\_id is the group id), event (an object that contains all information about the event) and text (if, for example, the event is a new message, then text is the text of the message, if, for example, a new comment, then text is the text of the comment, etc.)

If you want to execute multiple requests asynchronously at once, you can simply use the asyncio library. For example:

@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!")

The handler that processes the message with the string "прив" will first mark the message as read, then set the status "printing ..." and after 9 seconds send the message okay and all this is asynchronous. P.S. handlers do not block each other, so while the first handler is running, you can write "a" and the bot will answer "Б!", despite the work of the first handler.

In addition, you can make handlers not for conditions, but for the entire event as a whole. For example:

@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")

This handler will handle all events of type message\_new. In this case, he will answer "abs" with the message "peer", and answer "help me" with the message "no." And also the case of the message is not important, because we applied the lower method.

This is how you can do handlers for any events. For example, a handler for new comments:

@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!")

Which handler should I use? For conditions or for the whole event? If you need to make a handler for simple commands (answer that with this and something like that), then it is better to write all these commands in the event handler, in this case it will be @Handler.on(“message\_new”) because we need to answer on messages. And if the commands are complex, and not simple answers with some additional actions, then it is better to register them in the condition handler. For example, we need the bot to receive statistics from somewhere when the "statistics" message is sent, sorted and filtered them, and then send them. It is better to write this in the condition handler, in this case @Handler.on.message\_new(Condition(command="statistics"))

But you need to look at the code as a whole, because sometimes it can be useful to make an exception and write a complex command into the event handler, and a simple one into the condition handler. That is, the choice should depend on the situation and structure of your code.

Synchronization is not provided in the handlers and in the bot itself. Therefore, if you use an asynchronous implementation, for example, a database, there will be a race condition. And using synchronous database implementations is a bad idea, it will slow down the bot's speed. This structure allows the bot to be very fast. But the framework has an implementation of an asynchronous database with synchronization, which if you use it correctly, there will be no race condition and the bot will remain as fast. An example of a bot with this database implementation:

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 is a constant that needs your ID in VK, this condition prohibits creating a database for someone other than you with a command.   
 What does "async with db" do? "Async with db" waits for the database to open for requests, then closes the database for requests and as all your requests have passed to the database, it opens the database for requests again.  
 The "db.execute" method sends your request to the database.

The entire code for start:

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)