Урок Боты Discord

Чат-боты 3

Аннотация

На заключительном уроке по созданию чат-ботов мы создадим бота для Discord, а также немного поговорим о таком понятии, как параллельное программирование.

1. Введение

Сегодня мы создадим бота для Discord. Discord — бесплатный голосовой и текстовый чат, ориентированный в первую очередь на тех, кто играет в компьютерные игры. Однако его удобство, надежность и поддержка разных платформ позволяет использовать его не только в целях организации совместной игры. Есть много небольших команд разработчиков, которые используют Discord в рабочем процессе, в том числе и для автоматизации различных задач.

Для начала работы необходимо скачать клиент Discord с **официального сайта**, установить его и зарегистрировать учетную запись, если ее еще нет.

Для работы с API Discord мы будем использовать библиотеку discord.py, которую сначала надо установить:

```
pip install discord.py
```

discord.py полностью реализует API Discord, и предоставляет возможность создавать собственных ботов как

с использованием декораторов (как при создании своих обработчиков во flask), так и с использованием объектно-ориентированного подхода.

Документацию на библиотеку можно найти **тут**, а описание самого API — вот **тут**.

Но прежде чем мы продолжим, надо упомянуть важный момент: discord.py — асинхронная библиотека. Это будет накладывать некоторый отпечаток на процесс разработки программ с ее использованием.

2. Асинхронное программирование

Все время до этого момента мы писали **синхронные** программы, то есть такие, которые выполняются поэтапно. Если в коде написан последовательный вызов двух функций, то в синхронной программе интерпретатор вызовет первую, дождется завершения ее выполнения, а только затем вызовет вторую.

Такой подход достаточно понятен в написании программы: у нас есть гарантия того, какой кусок кода выполнится после какого, гарантия, что какие-то ресурсы (например, стандартный поток ввода `stdin`) будут использованы именно теми функциями, от которых мы этого ожидаем в каждый конкретный момент времени. Однако такой подход далеко не всегда удобен.

Давайте представим, что операционная система нашего компьютера работает полностью синхронно. Мы выбрали файл и начали копировать его из одной папки в другую. При синхронном подходе на время копирования файлов наша работа с операционной системой заблокирована до завершения текущей операции. Согласитесь, звучит не очень.

Когда мы говорим об асинхронном программировании, то подразумеваем создание такого кода, который позволяет программе не ждать завершения некоторого процесса, а продолжать работу не зависимо от него. Например, мы начали скачивать в браузере какой-то файл и после этого переключились на другую вкладку и продолжили заниматься своими делами.

Изначально в Python для решения задач асинхронного программирования использовались **корутины**, основанные на генераторах. Но, начиная с Python 3.4, появился модуль asyncio, в котором реализованы механизмы асинхронного программирования. А в Python 3.5 появилась конструкция `async/await`.

С помощью ключевого слова `async` при создании функции можно указать интерпретатору, что она является корутиной и может выполняться асинхронно. С помощью ключевого слова `await` мы можем вызвать такую функцию. Давайте рассмотрим очень простой пример:

```
import asyncio


async def work(name):
    print(f'{name} задача началась')
    await asyncio.sleep(1)
    print(f'{name} задача завершилась')


async def main():
    await asyncio.gather(
        asyncio.create_task(work('Первая')),
        asyncio.create_task(work('Вторая')),
    )


if __name__ == '__main__':
```

```
    asyncio.run(main())
```

Если мы запустим этот код, увидим такой вывод:

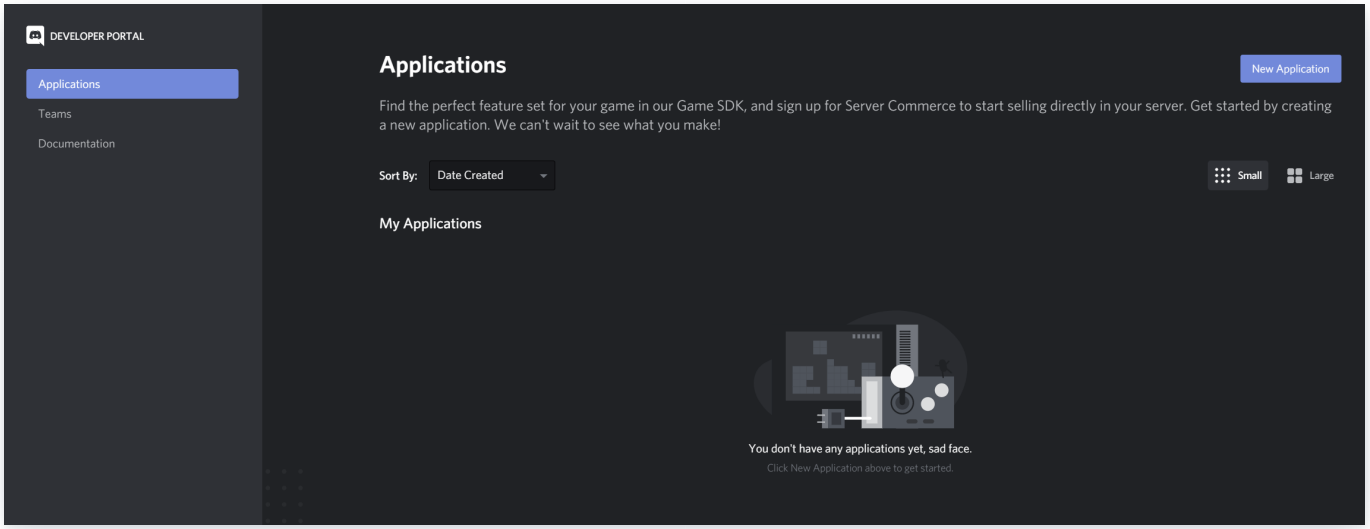
```
Первая задача началась
Вторая задача началась
Первая задача завершилась
Вторая задача завершилась
```

Внутри модуля asyncio много всего интересного, о чем можно почитать в **документации**, но для написания ботов для Discord нам будет достаточно этого небольшого введения.

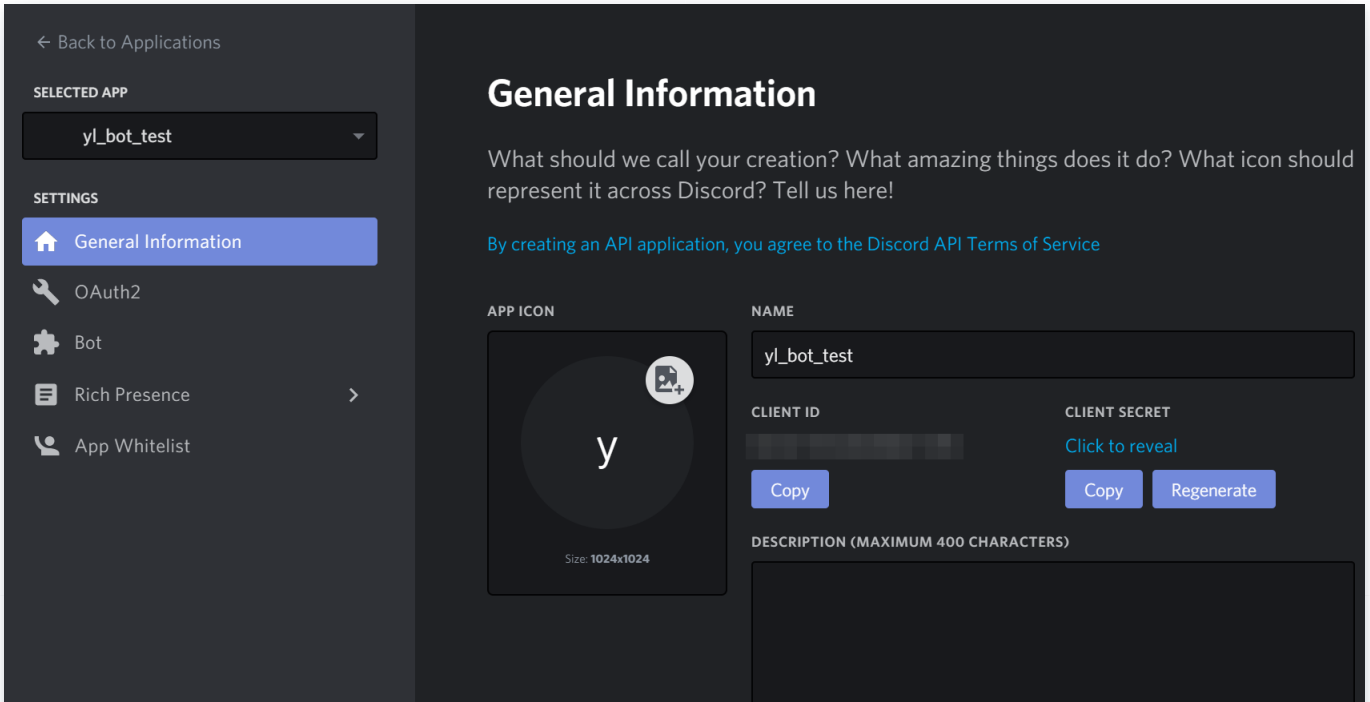
3. Подготовка

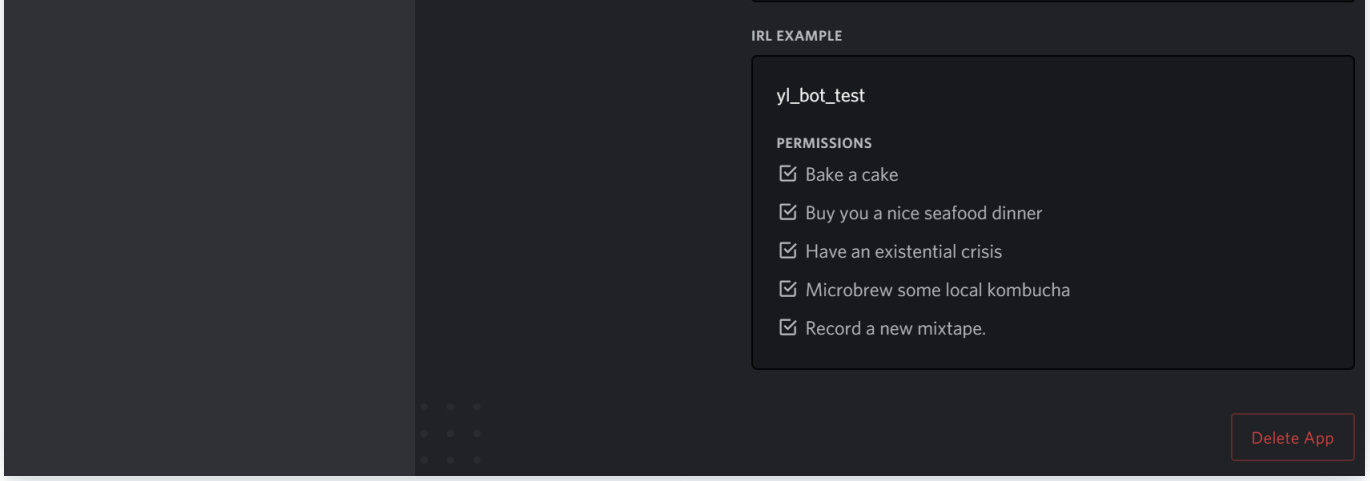
Чтобы получить доступ к API Discord, необходимо проделать несколько шагов и получить несколько токенов доступа. Во-первых, необходимо зарегистрироваться на самой платформе и подтвердить почту, которая была указана при регистрации.

После этого надо зайти на **портал для разработчиков**, авторизоваться там и нажать кнопку New application. Discord устроен так, что для создания бота необходимо сначала создать приложение для получения доступа к API, а потом к этому приложению можно будет добавить бота.



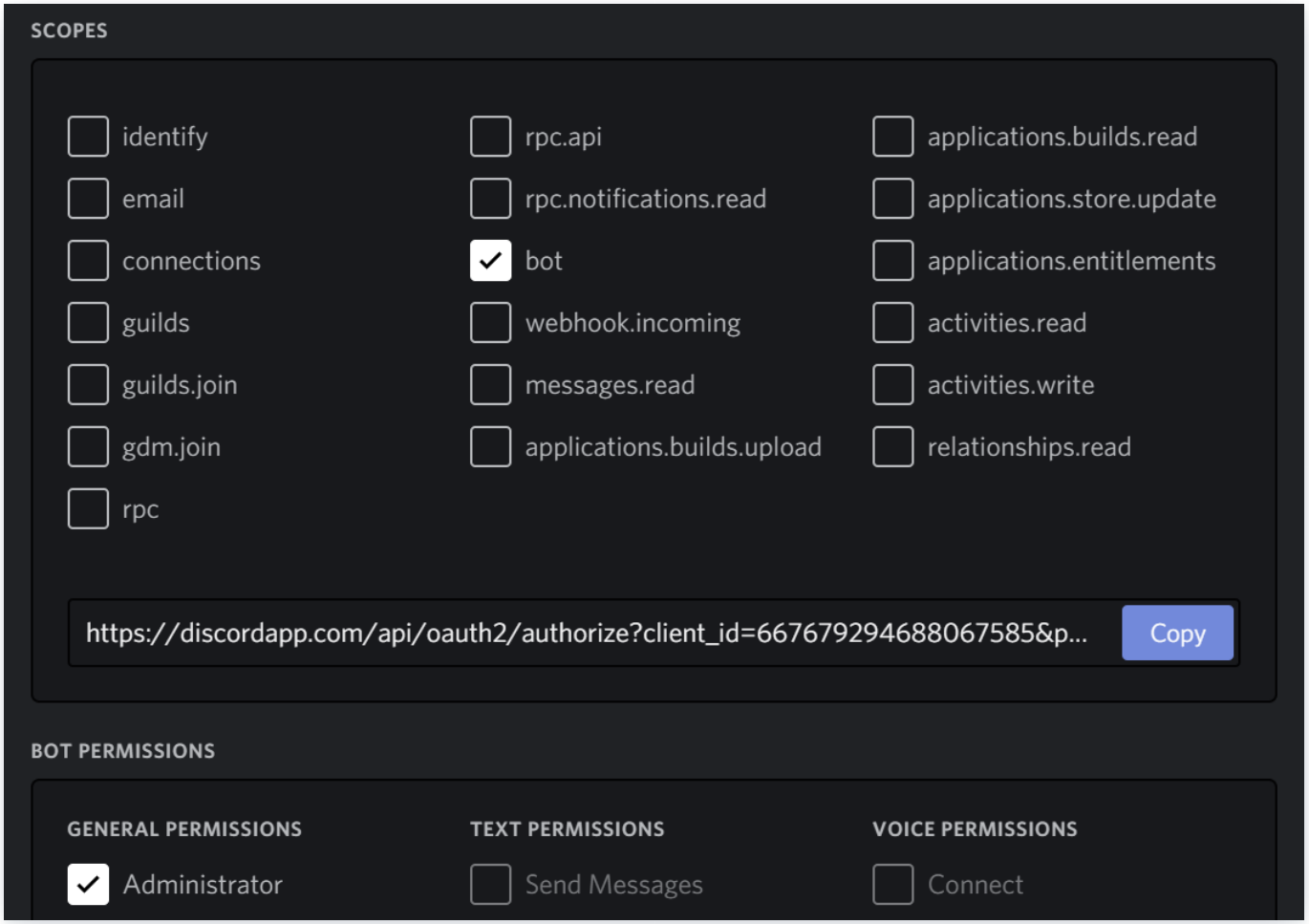
После ввода имени приложения мы перейдем в настройки приложения.





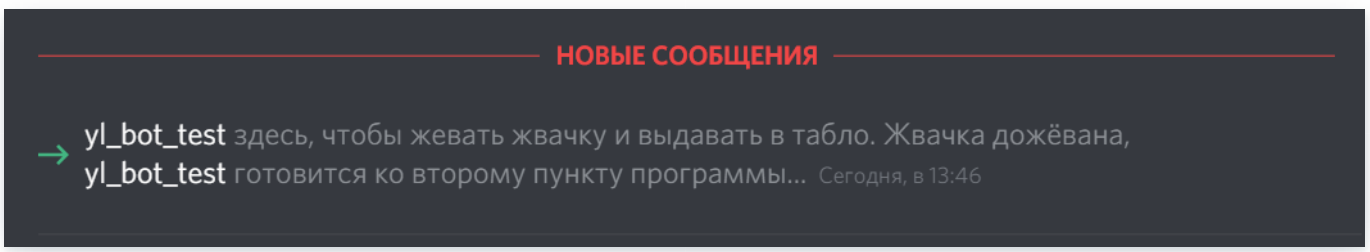
Затем для добавления к своему приложению бота перейдите на вкладку Bot и нажмите кнопку Add Bot. Можно сделать публичного бота, которого смогут добавлять все желающие, либо приватного, которого сможете добавлять только вы.

Остался последний штрих — добавить боту прав. Для этого воспользуемся вкладкой OAuth2. Выберем URL Generator — Scopes — Bot, а в Bot Permissions — Administrator.



После этого перейдите по ссылке из поля GENERATED URL и добавьте бота на сервер. Это может быть личный сервер или любой, который вы создали в приложении Discord.

Бот добавился и написал автоматическое приветственное сообщение:



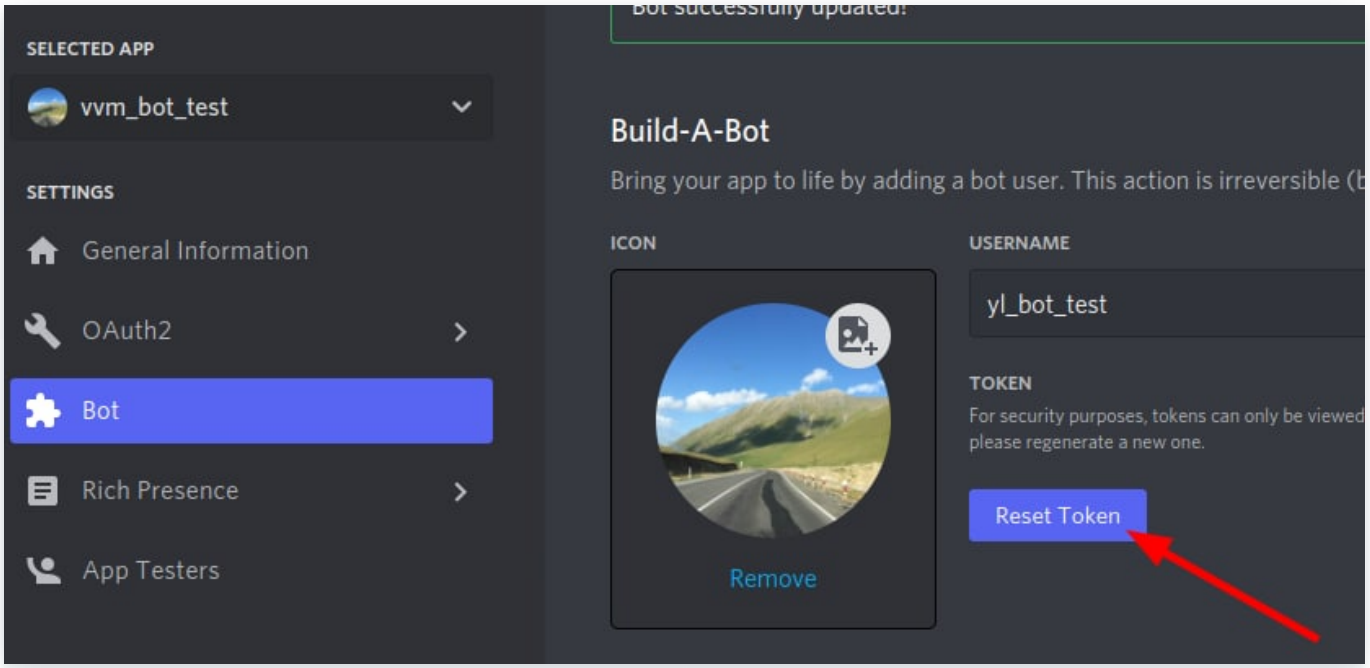
4. Простой бот

Самый простой бот для Discord выглядит следующим образом:

```
import discord

TOKEN = "BOT_TOKEN"
intents = discord.Intents.default()
client = discord.Client(intents=intents)
client.run(TOKEN)
```

Чтобы эта программа заработала, надо получить для своего бота token.



Этот бот, конечно, ничего не делает, но можно явно посмотреть на ту структуру, которую предлагает нам модуль. Согласитесь, она очень похожа на flask. Так же, как и во flask, обработчики событий можно создавать с использованием декоратора. Тут это декоратор `client.event`.

Давайте добавим между созданием клиента и его запуском обработчик события `on_ready`, который выполнится после того, как бот будет запущен и подключится к Discord, то есть появится «В сети».

```
@client.event
async def on_ready():
    print(f'{client.user} подключен к Discord!')
    for guild in client.guilds:
        print(
            f'{client.user} подключились к чату:\n'
            f'{guild.name}(id: {guild.id})'
        )
```

Этот обработчик пройдет по всем гильдиям (чатам), в которые добавлен бот, и выведет информацию об этом. Обратите внимание: наша функция асинхронная и определена с использованием ключевого слова `async`. Собственно, на этом все особенности в данном случае и заканчиваются.

В принципе, можно продолжать писать ботов с использованием декораторов, но библиотека discord.py позволяет использовать и объектно-ориентированный подход.

5. Объектно-ориентированный подход

Давайте перепишем этот простой пример с использованием объектно-ориентированного подхода:

```
import discord

TOKEN = "BOT_TOKEN"


class YLBotClient(discord.Client):
    async def on_ready(self):
        print(f'{self.user} has connected to Discord!')
        for guild in self.guilds:
            print(
                f'{self.user} подключились к чату:\n'
                f'{guild.name}(id: {guild.id})')


intents = discord.Intents.default()
client = YLBotClient(intents=intents)
client.run(TOKEN)
```

Код получился еще проще и понятнее.

Добавим еще несколько штрихов к нашей программе, чтобы получилось как в настоящих промышленных системах. Надо добавить в нашу программу логирование. В саму библиотеку **discord.py** уже встроен стандартный логгер, надо только его настроить. Добавьте в начало программы следующий код.

```
import logging

logger = logging.getLogger('discord')
logger.setLevel(logging.DEBUG)
handler = logging.StreamHandler()
handler.setFormatter(logging.Formatter('%(asctime)s:%(levelname)s:%(name)s: %(message)s'))
logger.addHandler(handler)
```

Теперь в консоль будет выводиться вся информация о работе нашего бота, в т.ч. мы увидим какие события он обрабатывает.

```
2022-03-27 12:35:49,590:WARNING:discord.client: PyNaCl is not installed, voice will NOT be suppo
2022-03-27 12:35:49,590:INFO:discord.client: logging in using static token
2022-03-27 12:35:51,041:INFO:discord.gateway: Shard ID None has sent the IDENTIFY payload.
2022-03-27 12:35:51,446:INFO:discord.gateway: Shard ID None has connected to Gateway: ["gateway-
2022-03-27 12:35:53,460:INFO:discord: yl_bot_test#0766 has connected to Discord!
```

Если вам покажется, что информации слишком много, то можно изменить подробность логирования, указав другой уровень. Например, заменить `logger.setLevel(logging.DEBUG)` на `logger.setLevel(logging.INFO)`. Теперь заменим "принты" на правильное логирование:

```
    async def on_ready(self):
        logger.info(f'{self.user} has connected to Discord!')
        for guild in self.guilds:
            logger.info(
                f'{self.user} подключились к чату:\n'
```

```
            f'{guild.name}(id: {guild.id})')
```

Давайте добавим несколько обработчиков событий. Для начала напишем обработчик, который будет приветствовать присоединение к серверу нового пользователя (`on_member_join`) личным сообщением. Обработчик этого события будет принимать еще один аргумент — объект нового пользователя. Добавим новый метод к нашему классу.

```
async def on_member_join(self, member):
    await member.create_dm()
    await member.dm_channel.send(
        f'Привет, {member.name}!'
    )
```

Но этого еще недостаточно, чтобы сообщение о событии (`on_member_join`) отправлялось нашему боту. Надо еще сделать небольшую настройку.

На странице настройки приложения надо перейти на закладку **Bot** добавить для бота разрешения на получение событий об изменении статусов пользователей сервера. Надо включить **PRESENCE INTENT** и **SERVER MEMBERS INTENT**.

Строку инициализации клиента `client = YLBotClient()` заменить на следующий код:

```
intents = discord.Intents.default()
intents.members = True
client = YLBotClient(intents=intents)
```

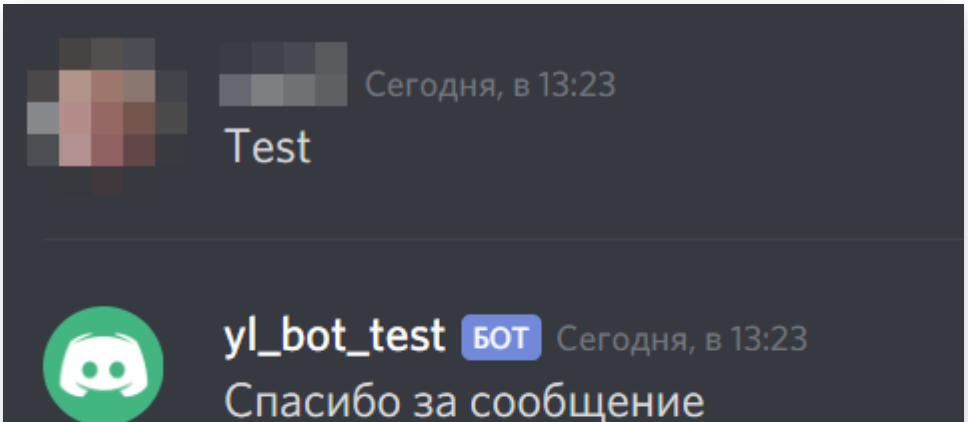
Здесь мы создаем объект класса `intents`, в котором указываем, что наш бот будет получать события, связанные с пользователями.

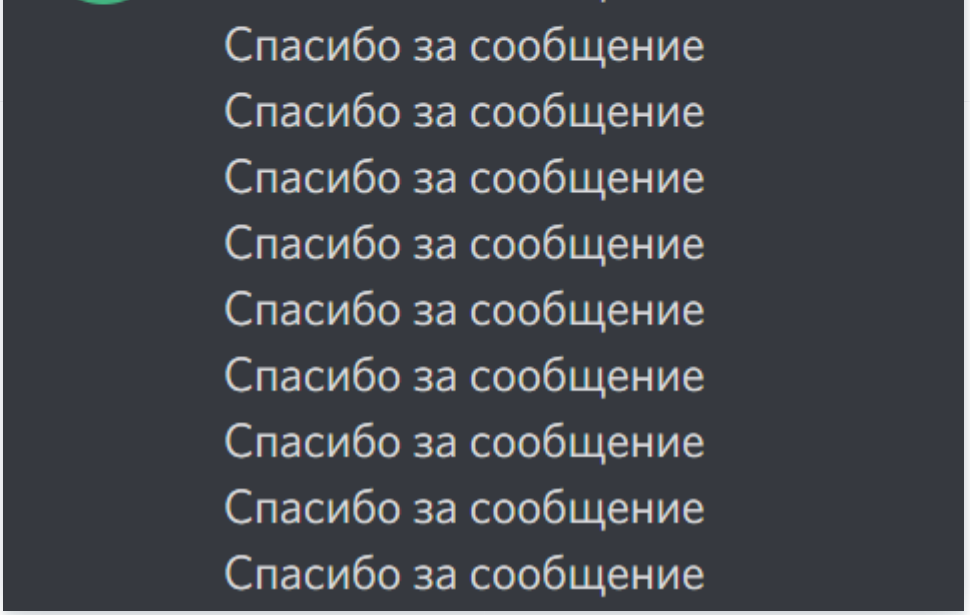
В тот момент, когда к нам приходит новый пользователь, бот создает чат для прямой отправки сообщений этому пользователю и посылает в него сообщение. Обратите внимание: все вызовы внутренних функций и методов объектов модуля discord.py должны происходить с указанием ключевого слова `await`. Проверьте как это работает. Для этого подключите бота к своему серверу, включите его и пригласите кого-нибудь на сервер. Как только новый пользователь зайдет на сервер, он тут же получит в личку привет от нашего бота.

Теперь давайте добавим реакцию бота на какое-либо сообщение. Для этого необходимо в класс добавить метод `on_message`, который в качестве параметра принимает объект сообщения.

```
async def on_message(self, message):
    await message.channel.send("Спасибо за сообщение")
```

Мы получаем сообщение и отправляем свое в чат, в котором это сообщение было получено. Давайте запустим и посмотрим, что произойдет.





Наш бот начнет отправлять сообщения бесконечно. Это связано с тем, что сообщения от самого бота также вызывают событие on_message. Чтобы этого не происходило, можно добавить проверку на автора сообщения:

```
if message.author == self.user:
    return
```

Разумеется внутри обработчика мы можем получить доступ к содержанию сообщения message. Сначала надо добавить для бота разрешения на получение текстов сообщений от пользователей сервера (**MESSAGE CONTENT INTENT**). Давайте проверим, что пользователь с нами поздоровался:

```
if "привет" in message.content.lower():
    await message.channel.send("И тебе привет")
```

Вот полный код нашего бота:

```
import discord
import logging

logger = logging.getLogger('discord')
logger.setLevel(logging.DEBUG)
handler = logging.StreamHandler()
handler.setFormatter(logging.Formatter('%(asctime)s:%(levelname)s:%(name)s: %(message)s'))
logger.addHandler(handler)

TOKEN = "BOT_TOKEN" # вставь свой токен


class YLBotClient(discord.Client):
    async def on_ready(self):
        logger.info(f'{self.user} has connected to Discord!')
        for guild in self.guilds:
            logger.info(
                f'{self.user} подключились к чату:\n'
                f'{guild.name}(id: {guild.id})')

    async def on_member_join(self, member):
        await member.create_dm()
        await member.dm_channel.send(
```

```
            f'Привет, {member.name}!'
        )

    async def on_message(self, message):
        if message.author == self.user:
            return
        if "привет" in message.content.lower():
            await message.channel.send("И тебе привет")
        else:
            await message.channel.send("Спасибо за сообщение")


intents = discord.Intents.default()
intents.members = True
intents.message_content = True
client = YLBotClient(intents=intents)
client.run(TOKEN)
```

6. Команды

Помимо простого анализа текста в Discord, как и в Telegram, можно создавать свои команды. При этом немного изменится структура программы: надо создавать уже не объект класса `Client`, а объект класса `Bot`, и использовать декоратор `bot.command`. Давайте сделаем бота, который будет возвращать случайное число от минимального до максимального, указанного пользователем.

Импортируем из модуля discord.ext модуль commands.

Сначала сделаем это с помощью отдельной функции:

```
import discord
from discord.ext import commands
import random, logging

logger = logging.getLogger('discord')
logger.setLevel(logging.DEBUG)
handler = logging.StreamHandler()
handler.setFormatter(logging.Formatter('%(asctime)s:%(levelname)s:%(name)s: %(message)s'))
logger.addHandler(handler)

intents = discord.Intents.default()
intents.members = True
intents.message_content = True

bot = commands.Bot(command_prefix='#!', intents=intents)


@bot.command(name='randint')
async def my_randint(ctx, min_int, max_int):
    num = random.randint(int(min_int), int(max_int))
    await ctx.send(num)
```

```
TOKEN = "BOT_TOKEN"

bot.run(TOKEN)
```

При создании бота необходимо указать подстроку `command_prefix`, с которой будут начинаться все команды. Параметры команды должны идти в сообщении через пробел после названия команды, но первым аргументом передается **контекст**, из которого мы можем получить чат и другую полезную информацию.

А теперь сделаем имитацию броска шестигранных кубиков, но уже с использованием объектно-ориентированного подхода:

```
import asyncio
import discord
from discord.ext import commands
import random, logging

logger = logging.getLogger('discord')
logger.setLevel(logging.DEBUG)
handler = logging.StreamHandler()
handler.setFormatter(logging.Formatter('%(asctime)s:%(levelname)s:%(name)s: %(message)s'))
logger.addHandler(handler)

intents = discord.Intents.default()
intents.members = True
intents.message_content = True
dashes = ['\u2680', '\u2681', '\u2682', '\u2683', '\u2684', '\u2685']


class RandomThings(commands.Cog):

    def __init__(self, bot):
        self.bot = bot

    @commands.command(name='roll_dice')
    async def roll_dice(self, ctx, count):
        res = [random.choice(dashes) for _ in range(int(count))]
        await ctx.send(" ".join(res))

    @commands.command(name='randint')
    async def my_randint(self, ctx, min_int, max_int):
        num = random.randint(int(min_int), int(max_int))
        await ctx.send(num)


bot = commands.Bot(command_prefix='!#', intents=intents)

TOKEN = "BOTTOKEN"


async def main():
    await bot.add_cog(RandomThings(bot))
    await bot.start(TOKEN)
```

```
asyncio.run(main())
```

Обратите внимание: в таком случае нам надо наследоваться не от класса `Bot`, а от класса `Cog`, определять команды там, а потом добавлять их в бота методом `bot.add_cog`.

7. Заключение

Возможности API Discord достаточно широкие, есть даже возможность подключаться к прослушиванию аудио-сообщений от пользователей канала. Простор для экспериментов действительно огромен.

Помимо рассмотренных, есть еще ряд платформ с хорошим API и библиотекам к нему на Python. К части из них доступ получить просто и мгновенно, где-то (например, для доступа к API Twitter) вас попросят написать мини-пояснение на тему «как вы хотите использовать API», где-то — пройти дополнительную идентификацию и подтверждение личности.

Кроме того, рекомендуем посмотреть в сторону параллельного программирования. Такую парадигму поддерживает множество классных библиотек, например, веб-фреймворк **Tornado**.