Методические указания

Урок 34.1 Настройка Webhook для чат-бота

Задачи урока:

- Настройка Webhook для чат-бота

0. Подготовка к уроку

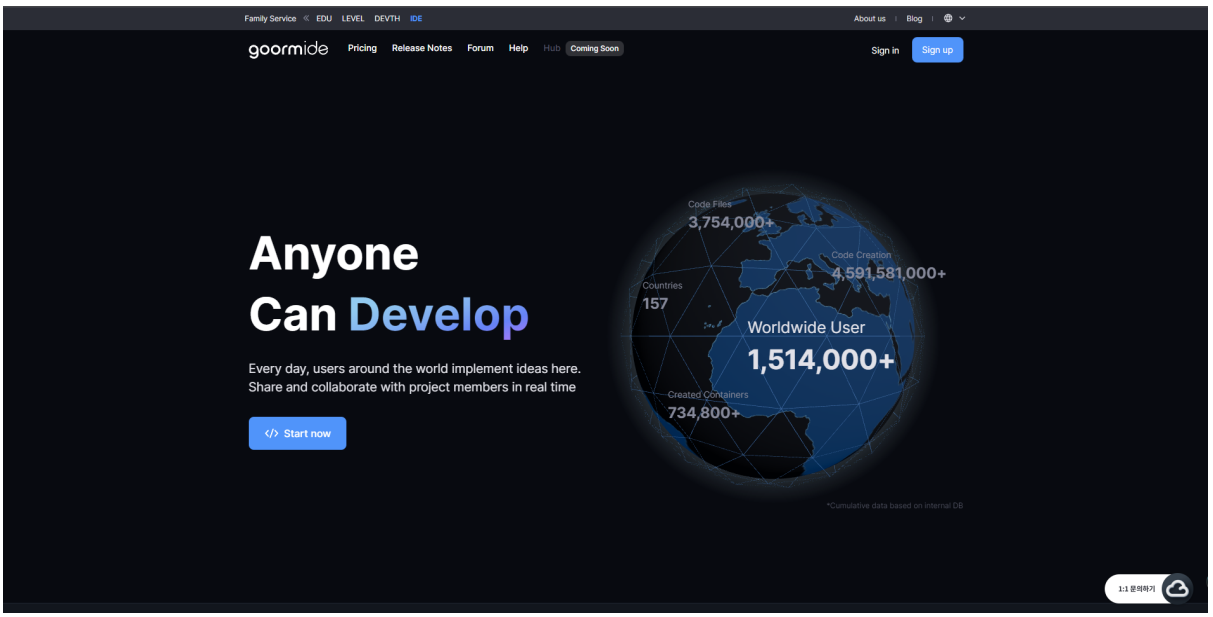
До начала урока преподавателю необходимо:

1) Просмотреть, как ученики справились с домашним заданием
2) Прочитать методичку

1. Настройка Webhook для чат-бота

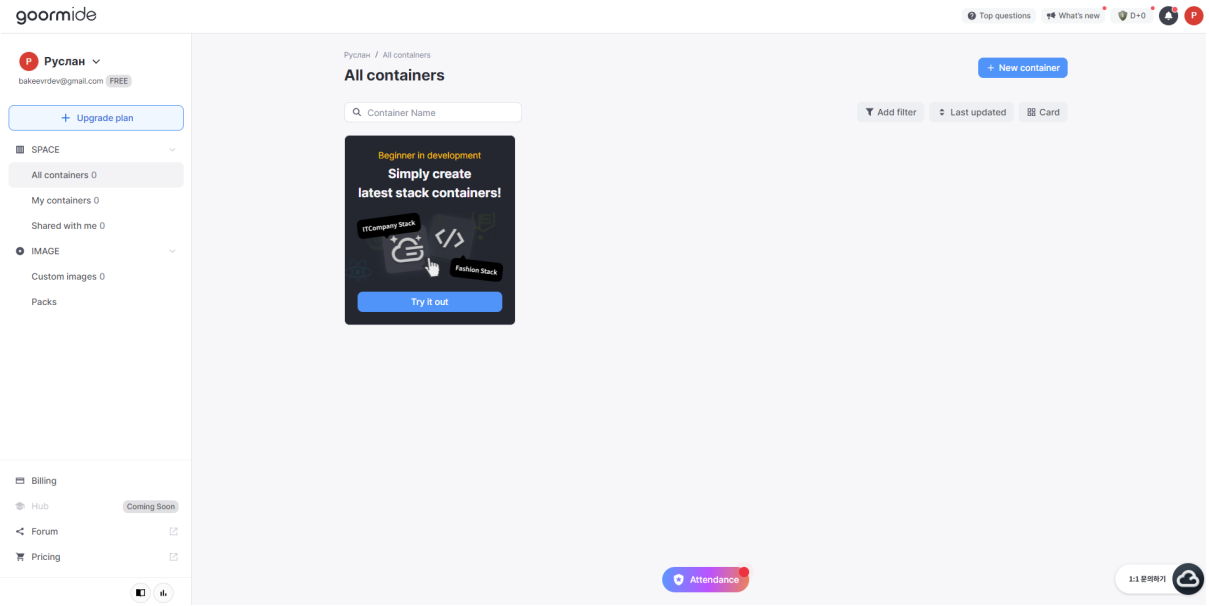
Учитель: На сегодняшнем занятии мы познакомимся еще с одним сервисом и поднимем на нем бота. А также немного познакомимся с работой с базой данных.
Сервис не русскоязычный и даже не англоязычный, но используйте возможность перевода встроенную в браузере.

Итак начнем с регистрации. Переходим по ссылке https://ide.goorm.io/

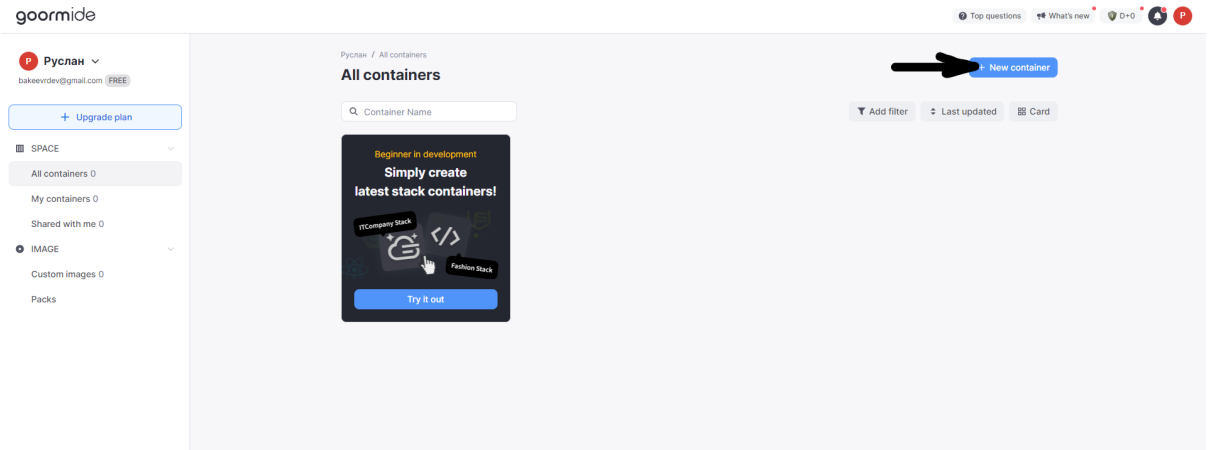


Нажимаем на Start now

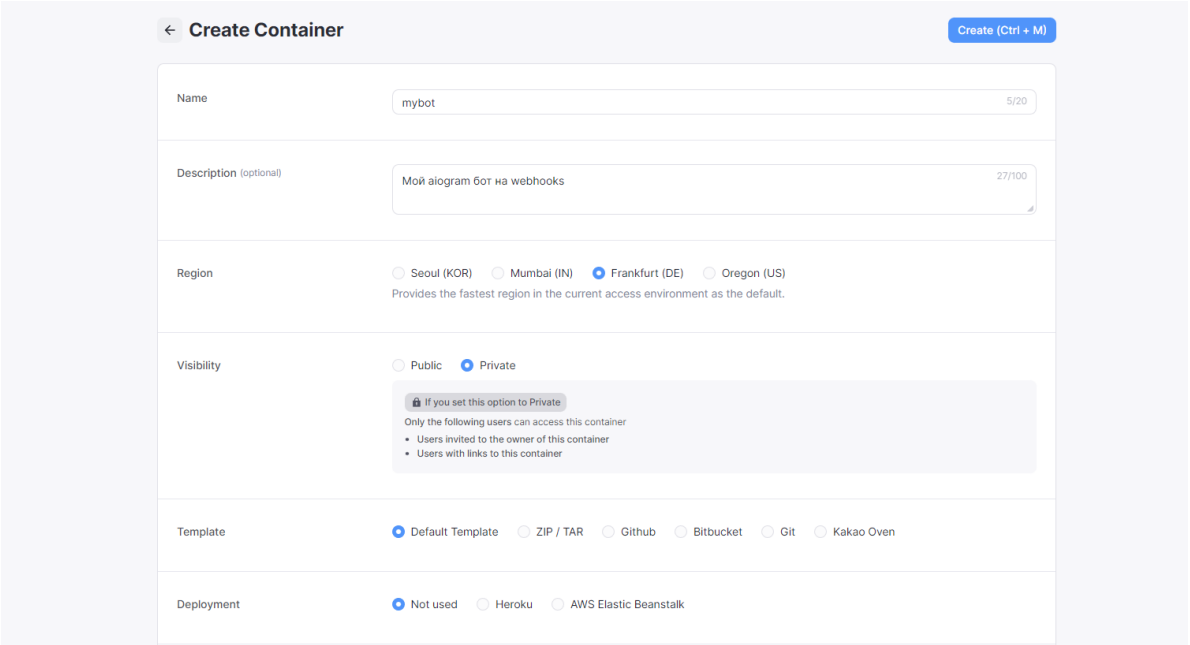
Проходим регистрацию через google аккаунт. Нас встречает интерфейс сервиса



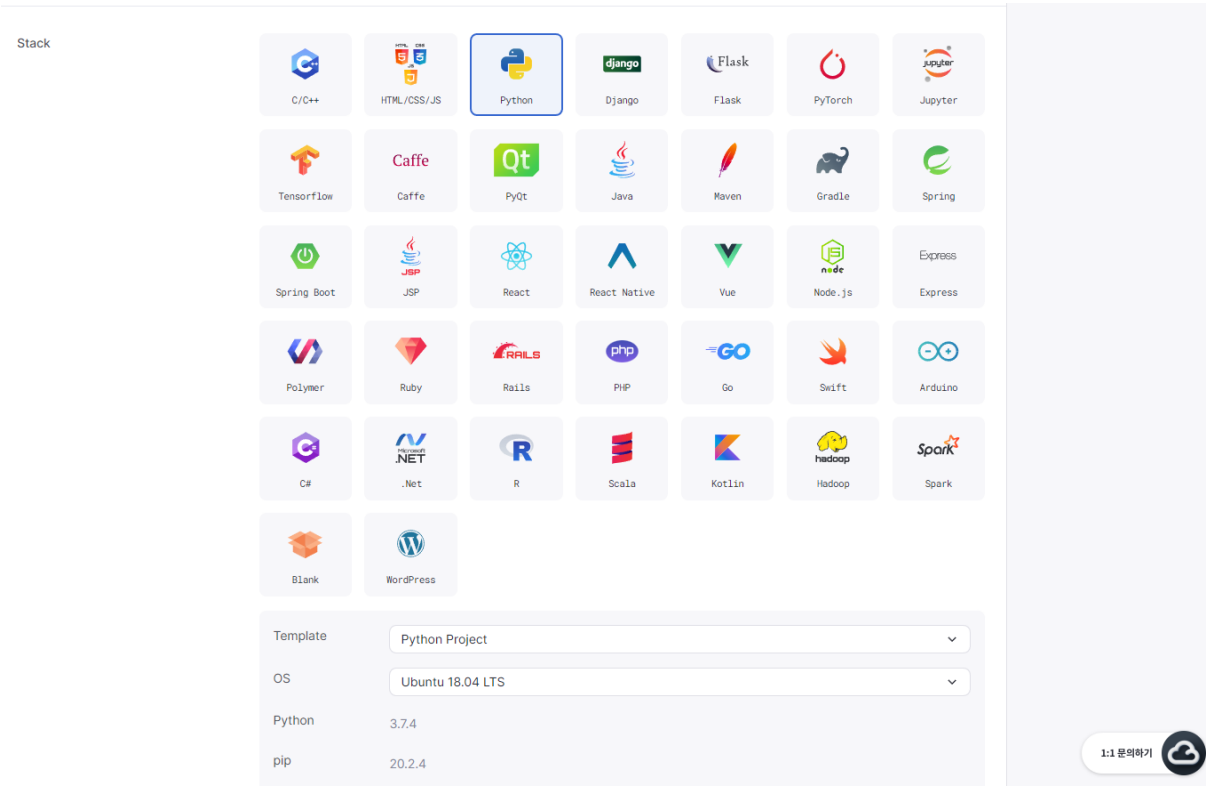
Далее мы создадим новый контейнер



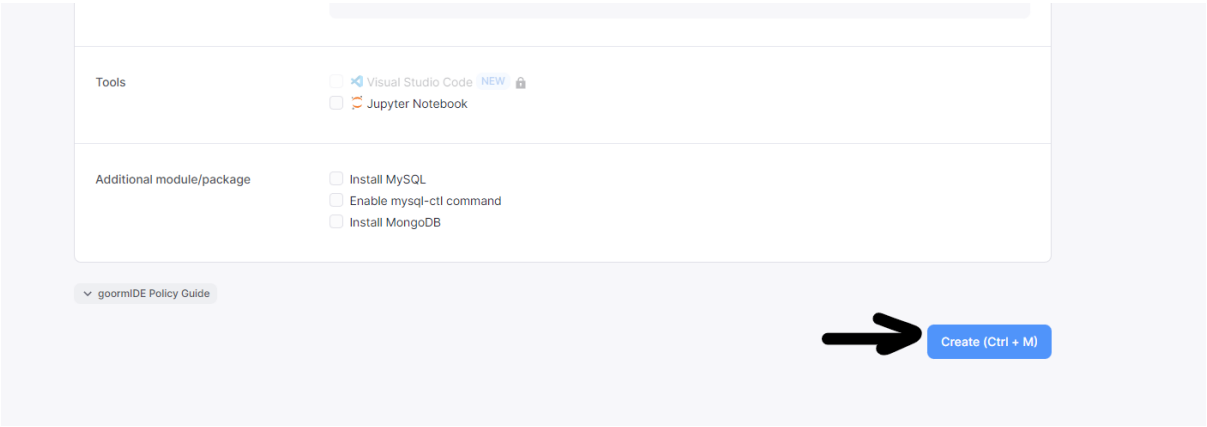
Заполняем имя и описание



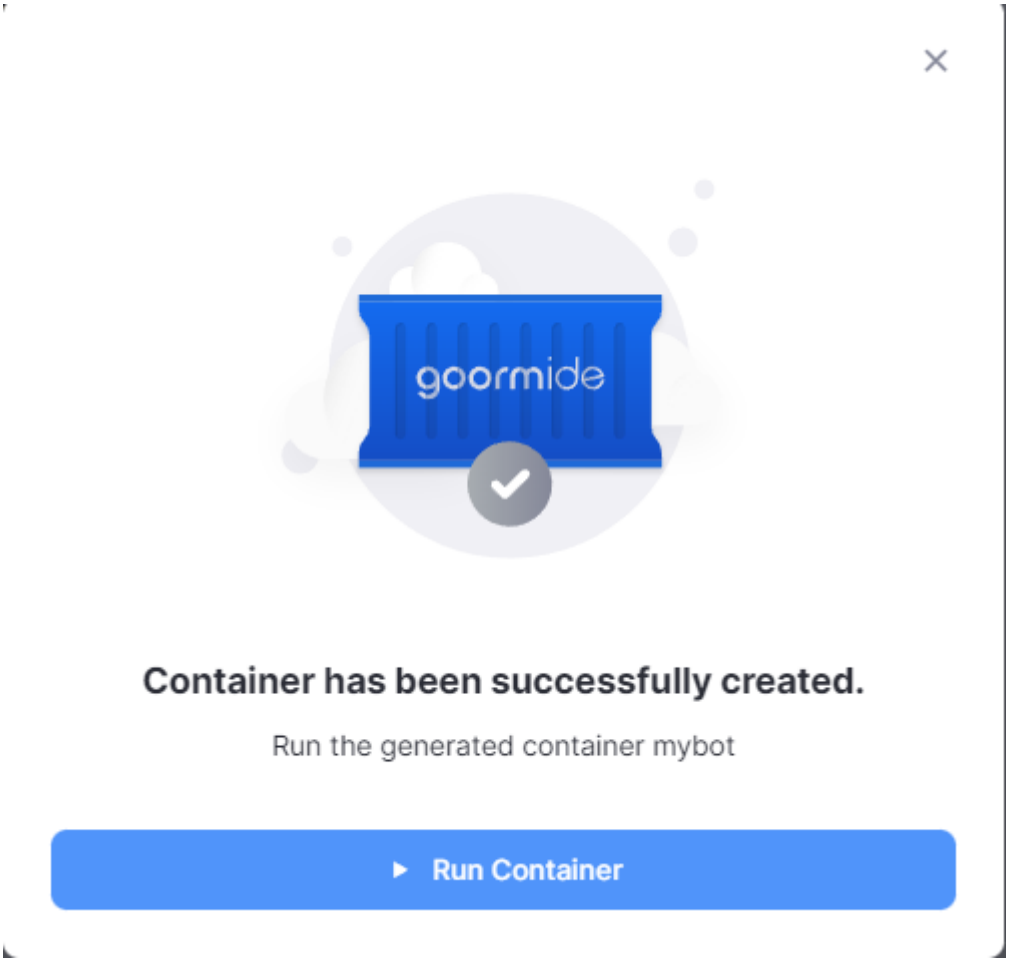
Выбираем необходимый язык



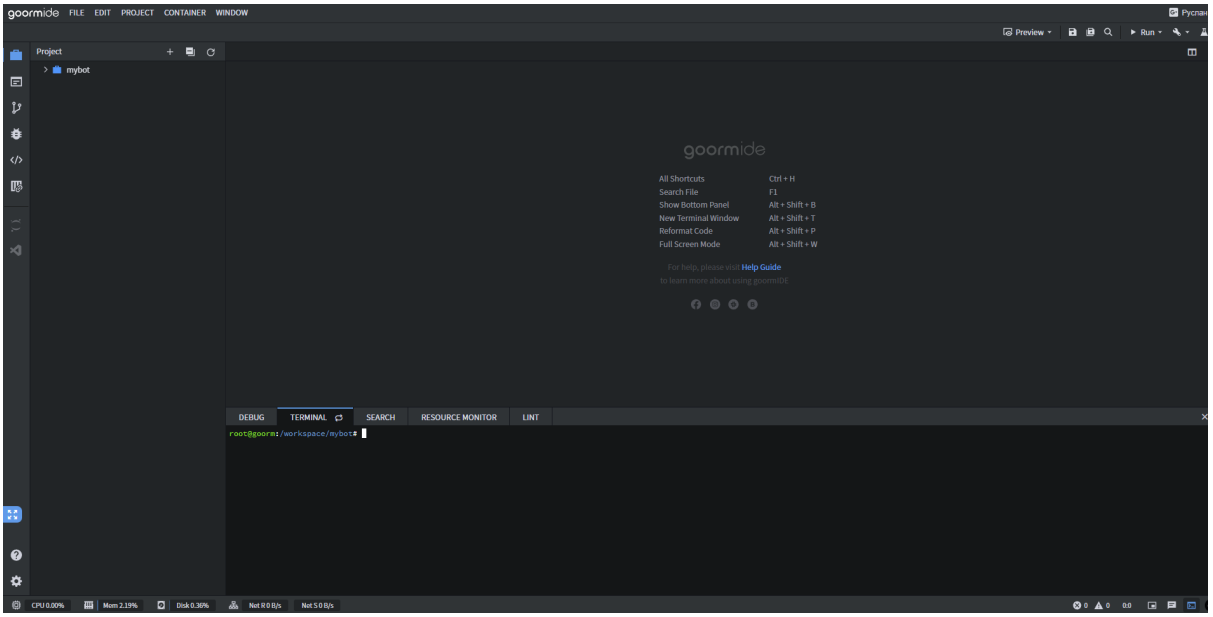
Нажимаем на создание контейнера



Отлично. После того как контейнер создастся, мы сможем начать писать код, прямо в нашем сервисе. Нажимаем запустить контейнер

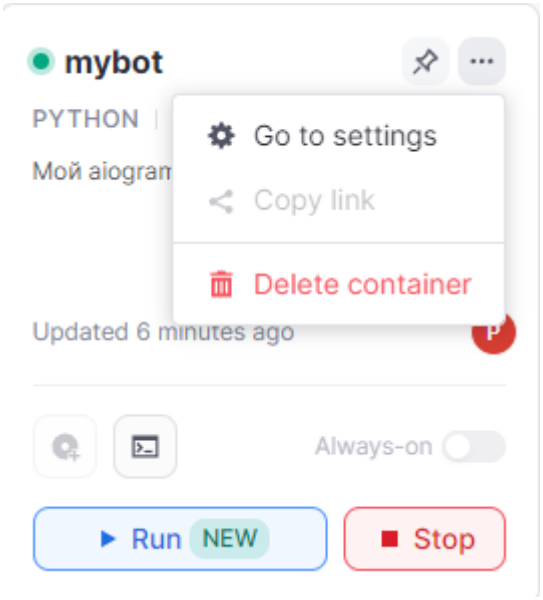


Далее мы переходим в интерфейс ide

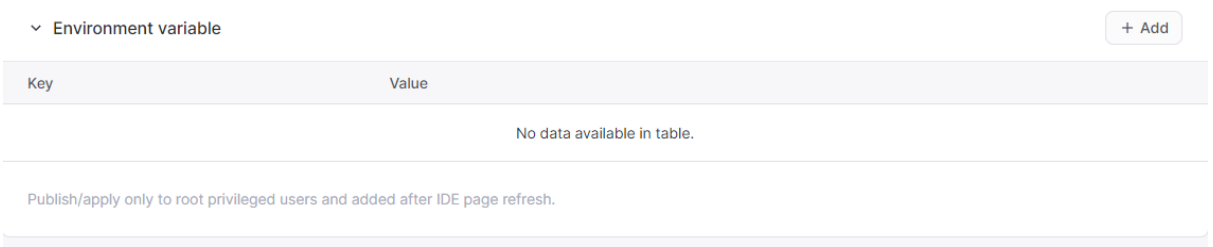


Изначально у нас уже есть файл index.py, в котором прописано hello world. Запускается файл по кнопке run. После любых изменений в файле. Необходимо сохранять изменения нажав на кнопку с дискеткой.

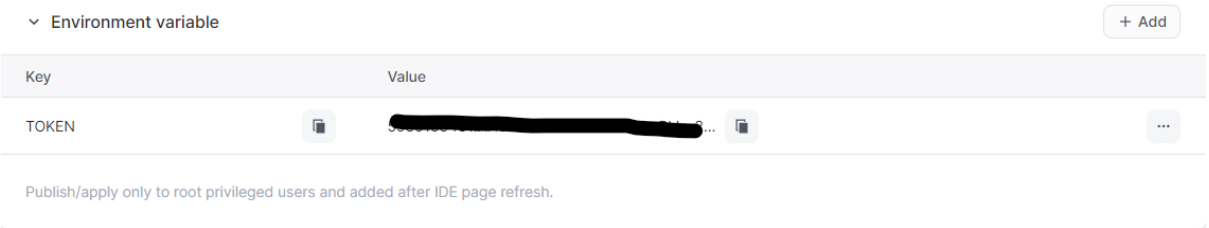
Теперь определим переменные окружения. Возвращаемся на главную страницу, где создавали контейнер. Рядом с контейнером нажимаем на ...



и выбираем Go to settings. Находим необходимый пункт и нажимаем Add

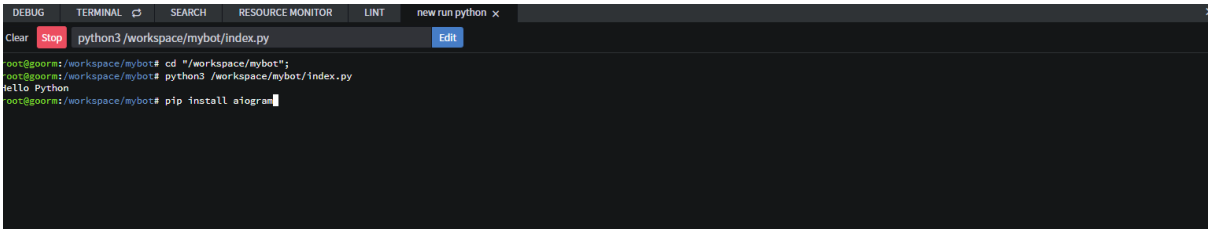


Вводим имя переменной и значение нашего токена



Готово. Вы также можете дополнительно определить любые переменные, значения которых, вы не хотите указывать в проекте напрямую.

Теперь вернемся в редактор и изменим файл index.py указав там обычного эхо бота на пуллинге для проверки.
Не забываем в консоли установить aiogram



После этого прописываем стандартный код

```
import logging
import os

from aiogram import Bot, Dispatcher, executor, types

TOKEN = os.environ.get('TOKEN')
API_TOKEN = TOKEN

logging.basicConfig(level=logging.INFO)

bot = Bot(token=API_TOKEN)
dp = Dispatcher(bot)


@dp.message_handler(commands=['start', 'help'])
async def send_welcome(message: types.Message):
    await message.reply("Привет! Вы ввели одну из команд.")



@dp.message_handler()
async def echo(message: types.Message):
    await message.answer(message.text)
```

```
if __name__ == '__main__':
    executor.start_polling(dp, skip_updates=True)
```

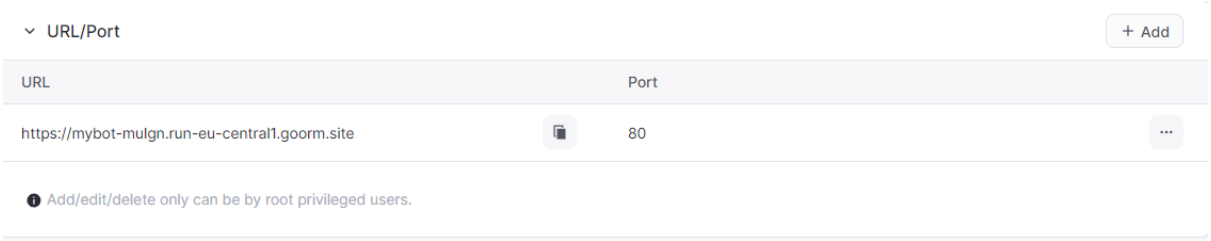
Запускаем и проверяем. Пулинг работает отлично

Теперь давайте переделаем все на вебхуки

```
import logging
import os

from aiogram import Bot, types
from aiogram.contrib.middlewares.logging import LoggingMiddleware
from aiogram.dispatcher import Dispatcher
from aiogram.dispatcher.webhook import SendMessage
from aiogram.utils.executor import start_webhook


API_TOKEN = os.environ.get('TOKEN')


WEBHOOK_HOST = 'https://your.domain'
WEBHOOK_PATH = '/path/to/api'
WEBHOOK_URL = f"{WEBHOOK_HOST}{WEBHOOK_PATH}"


WEBAPP_HOST = 'localhost'  # or ip
WEBAPP_PORT = 3001

logging.basicConfig(level=logging.INFO)

bot = Bot(token=API_TOKEN)
dp = Dispatcher(bot)
dp.middleware.setup(LoggingMiddleware())


@dp.message_handler()
async def echo(message: types.Message):
    return SendMessage(message.chat.id, message.text)


async def on_startup(dp):
    await bot.set_webhook(WEBHOOK_URL)


async def on_shutdown(dp):
    logging.warning('Shutting down..')
    await bot.delete_webhook()
    logging.warning('Bye!')


if __name__ == '__main__':
    start_webhook(
```

```
    dispatcher=dp,
    webhook_path=WEBHOOK_PATH,
    on_startup=on_startup,
    on_shutdown=on_shutdown,
    skip_updates=True,
    host=WEBAPP_HOST,
    port=WEBAPP_PORT,
)
```

Теперь прежде, чем изменить код шаблона, настроим кое что в сервисе. Идем опять на главную страницу. Находим URL/Port



Копируем url. (Если нет url и порта, то создаем)

```
WEBHOOK_HOST = 'https://mybot-mulgn.run-eu-central1.goorm.site'
WEBHOOK_PATH = ''
WEBHOOK_URL = f"{WEBHOOK_HOST}{WEBHOOK_PATH}"
```

Меняем строчки

```
WEBAPP_HOST = '0.0.0.0'
WEBAPP_PORT = 80
```

Запускаем бота. Все работает! Отлично!

Теперь немного поговорим о базе данных

Существует множество различных типов баз данных. Выбор наилучшей базы данных для конкретной компании зависит от того, как она намеревается использовать данные.

- Реляционные базы данных Реляционные базы данных стали преобладать в 1980-х годах. Данные в реляционной базе организованы в виде таблиц, состоящих из столбцов и строк. Реляционная СУБД обеспечивает быстрый и эффективный доступ к структурированной информации.
- Объектно-ориентированные базы данных Информация в объектно-ориентированной базе данных представлена в форме объекта, как в объектно-ориентированном программировании.

- Распределенные базы данных Распределенная база данных состоит из двух или более частей, расположенных на разных серверах. Такая база данных может храниться на нескольких компьютерах.
- Хранилища данных Будучи централизованным репозиторием для данных, хранилище данных представляет собой тип базы данных, специально предназначенной для быстрого выполнения запросов и анализа.
- Базы данных NoSQL База данных NoSQL, или нереляционная база данных, дает возможность хранить и обрабатывать неструктурированные или слабоструктурированные данные (в отличие от реляционной базы данных, задающей структуру содержащихся в ней данных). Популярность баз данных NoSQL растет по мере распространения и усложнения веб-приложений.
- Графовые базы данных Графовая база данных хранит данные в контексте сущностей и связей между сущностями.
- Базы данных OLTP. База данных OLTP — это база данных предназначенная для выполнения бизнес-транзакций, выполняемых множеством пользователей.

Это лишь некоторые из десятков типов баз данных, используемых в настоящее время. Другие, менее распространенные базы данных, предназначены для очень специфических научных, финансовых и иных задач.

В данном случае мы будем знакомиться с SQL базами данных, а непосредственно SQLite. Данная база данных не требует установки на компьютер и взаимодействует с Python с помощью модуля sqlite3.

Модуль sqlite3 реализует интерфейс доступа к внутрипроцессной базе данных SQLite. База данных SQLite спроектирована таким образом, чтобы ее можно было встраивать в приложения, а не использовать отдельную серверную программу, такую как MySQL, PostgreSQL или Oracle.
SQLite — быстрая, тщательно протестированная и гибкая база данных, что делает ее весьма удобной для прототипирования и производственного развертывания в случае некоторых приложений.

База данных SQLite хранится в файловой системе в виде одиночного файла. Библиотека управляет доступом к этому файлу, включая его блокирование с целью предотвращения повреждения данных, когда одновременно несколько программ пытаются записать данные в файл. База данных создается при первой попытке доступа к файлу, но ответственность за создание таблицы определений, или схемы базы данных, возлагается на приложение.
В этом примере программа сначала осуществляет поиск файла базы данных, прежде чем открыть его с помощью функции connect (), поэтому ей известно, в каких случаях следует создавать схему для новых баз данных.

```
import os
import sqlite3
```

```
db_filename = 'todo.db'
db_is_new = not os.path.exists(db_filename)
conn = sqlite3.connect(db_filename)
if db_is_new:
   print('Создана новая база')
else:
   print('База данных уже существует.')
conn.close()
```

Выполнение этого кода два раза подряд показывает, что в том случае, когда файл не существует, создается пустой файл.

После создания нового файла базы данных следующим шагом является создание схемы для определения таблиц в базе данных.

```
import os
import sqlite3
db_filename = 'todo.db'
schema_filename = 'todo_schema.sql'
db_is_new = not os.path.exists(db_filename)
with sqlite3.connect(db_filename) as conn:
   if db_is_new:
       print('Creating schema')
       with open(schema_filename, 'rt') as f:
           schema = f.read()
       conn.executescript(schema)
       print('Inserting initial data')
       conn.executescript("""
       insert into project (name, description, deadline)
       values ('pymotw', 'Python Module of the Week',
       '2022-11-01');
       insert into task (details, status, deadline, project)
       values ('write about select', 'done', '2022-04-25',
       'pymotw');
       insert into task (details, status, deadline, project)
       values ('write about random', 'waiting', '2022—08—22',
       'pymotw');
       insert into task (details, status, deadline, project)
       values ('write about sqlite3', 'active', '2021—07—31',
       'pymotw');
       """)
   else:
       print('Database exists, assume schema does, too.')
```

Вслед за созданием таблиц выполняются инструкции вставки, с помощью которых создается пробный проект и относящиеся к нему задачи.

SQLite для Python предлагает меньше типов данных, чем есть в других реализациях SQL. С одной стороны, это накладывает ограничения, но, с другой стороны, в SQLite многое сделано проще. Вот основные типы:

- NULL — значение NULL

- INTEGER — целое число
- REAL — число с плавающей точкой
- TEXT — текст
- BLOB — бинарное представление крупных объектов, хранящееся в точности с тем, как его ввели

Разберем создание базы данных и таблицы подробнее. Есть несколько способов создания базы данных в Python с помощью SQLite. Для этого используется объект Connection, который и представляет собой базу. Он создается с помощью функции connect().
Создадим файл .db, поскольку это стандартный способ управления базой SQLite. Файл будет называться test.db. За соединение будет отвечать переменная conn.

```
conn = sqlite3.connect('test.db')
```

Эта строка создает объект connection, а также новый файл orders.db в рабочей директории. Если нужно использовать другую, ее нужно обозначить явно:

```
conn = sqlite3.connect('ПУТЬ-К-ПАПКИ/test.db')
```

Если файл уже существует, то функция connect осуществит подключение к нему. Функция connect создает соединение с базой данных SQLite и возвращает объект, представляющий ее.

Еще один способ создания баз данных с помощью SQLite в Python — создание их в памяти. Это отличный вариант для тестирования, ведь такие базы существуют только в оперативной памяти.

```
conn = sqlite3.connect(:memory:)
```

После создания объекта соединения с базой данных нужно создать объект cursor. Он позволяет делать SQL-запросы к базе. Используем переменную cur для хранения объекта:

```
cur = conn.cursor()
```

Теперь выполнять запросы можно следующим образом:

```
cur.execute("ВАШ-SQL-ЗАПРОС-ЗДЕСЬ;")
```

Обратите внимание на то, что сами запросы должны быть помещены в кавычки — это важно. Это могут быть одинарные, двойные или тройные кавычки. Последние используются в случае особенно длинных запросов, которые часто пишутся на нескольких строках.

Давайте разберем как создать таблицу, в которой у нас есть имя фамилия пол, ну и порядковый номер. Для этого необходимо создать запрос

```
cur.execute("""CREATE TABLE IF NOT EXISTS users(
   userid INT PRIMARY KEY,
   fname TEXT,
   lname TEXT,
   gender TEXT);
""")
conn.commit()
```

В коде выполняются следующие операции:

1. Функция execute отвечает за SQL-запрос
2. SQL генерирует таблицу users
3. IF NOT EXISTS поможет при попытке повторного подключения к базе данных. Запрос проверит, существует ли таблица. Если да — проверит, ничего ли не поменялось.
4. Создаем первые четыре колонки: userid, fname, lname и gender. Userid — это основной ключ.
5. Сохраняем изменения с помощью функции commit для объекта соединения.

Вернемся к локальному эхо боту с пуллингом из прошлых уроков создадим в боте таблицу

```
def create_table():
  conn = sqlite3.connect('users.db')
  cur = conn.cursor()
  cur.execute("""CREATE TABLE IF NOT EXISTS users(
          userid INT PRIMARY KEY,
          username TEXT,
          message TEXT);
          """)
  conn.commit()
  cur.close()
  conn.close()
```

И вызовем данную функцию

```
if __name__ == '__main__':
  create_table()
  executor.start_polling(dp, skip_updates=True)
```

Наш бот будет проверять есть ли пользователь в базе данных. Если нет, то добавлять, если есть, то выводить данные по нему. Сделаем заготовку

```
@dp.message_handler(text='база')
async def echo(message: types.Message):
  conn = sqlite3.connect('users.db')
  cur = conn.cursor()
  user = cur.execute(f"""SELECT * FROM users WHERE userid = {message.chat.id}""").fetchone()
  if not user:
    print('NO')
  else:
    print("YES")
  await message.answer(message.text)
```

Запрос cur.execute(f"""SELECT * FROM users WHERE userid = {message.chat.id}""").fetchone() позволяет получить пользователя, который написал сообщение, из указанной таблицы. Пока в таблице у нас никого нет.
if not user: проверяет получен ли такой пользователь

Теперь если пользователя нет в базе данных добавим его с помощью запроса INSERT

```
if not user:
  data = (message.chat.id, message.chat.username)
  cur.execute("""INSERT INTO users(userid, username) VALUES (?,?)""", data)
  conn.commit()
  conn.close()
  await message.answer('Вы добавлены в базу данных')
```

Если пользователь уже есть, то выведем его данные

```
@dp.message_handler(text='база')
async def echo(message: types.Message):
  conn = sqlite3.connect('users.db')
  cur = conn.cursor()
  user = cur.execute(f"""SELECT * FROM users WHERE userid = {message.chat.id}""").fetchone()
  if not user:
    data = (message.chat.id, message.chat.username)
    cur.execute("""INSERT INTO users(userid, username) VALUES (?,?)""", data)
    conn.commit()
    conn.close()
    await message.answer('Вы добавлены в базу данных')
  else:
    id, username = user[0]
    await message.answer(f'Вы уже есть в базе данных. Ваш id = {id}, username = {username}')
```

Готово.

Дополнительно

Если на уроке остается время, то ученикам можно предложить начать прорешивать домашнее задание.

Домашняя работа

Задача 1

1. Добавить в бота реализацию работы с SQLite как на занятии