**Документация по проекту**:

Реализация ETL процесса и создание REST API для витрины данных.

**Цель проекта**:

Разработать ETL процесс для интеграции данных из различных

источников (веб-страница, CSV файл, SQL база данных) и создать REST API в виде витрины данных для дальнейшего построения дашборда в системе бизнес-аналитики (BI).

Проектная команда:

Рызыванова, Сатдаров, Гренадеров, Измайлов, Зайцев, Новикова.

***1.*** Разработка скрипта для парсинга данных с веб-страницы с использованием библиотек для веб-скрапинга.

**Выбор веб-страницы**:

В качестве страницы для парсинга необходимых данных был выбран сайт 'https://horosho-tam.ru/rossiya/coronavirus', так как изначально предложенный сайт прекратил свою работу.

Был изучен HTML-документ сайта, и выбраны необходимые теги, с которых будет спарсена необходимая информация, а именно:

Country – Регион, а именно Россия.

Date – время обновления данных.

Cases – количество заразившихся.

Deaths – количество смертей.





**Выбор необходимых библиотек**:

1)

Для парсинга данных с веб-страницы была использована популярная библиотека Python **Beautiful Soup**, так как она позволяет легко извлекать данные из HTML, навигироваться по структуре веб-страницы, находить нужные элементы и извлекать информацию из них.

Несколько причин, почему Beautiful Soup часто выбирают для веб-скрапинга:

1. **Простота использования:** Beautiful Soup предоставляет простой и понятный API для парсинга веб-страниц. Это делает ее привлекательным выбором для новичков и опытных разработчиков.
2. **Гибкость:** Она может работать с различными типами разметки HTML и XML и способна обрабатывать "грязные" данные, которые могут вызывать ошибки при использовании других парсеров.
3. **Мощный поиск и навигация:** Благодаря функциям поиска, фильтрации и навигации, Beautiful Soup упрощает поиск и извлечение данных из HTML-документов.
4. **Поддержка других библиотек:** Beautiful Soup хорошо интегрируется с другими библиотеками Python, такими как requests (для отправки HTTP-запросов) и pandas (для анализа данных).

2)

Также была использована библиотека **requests**, которая позволяет отправлять и обрабатывать HTTP-запросы.

3)Также была необходима библиотека для работы с базами данных и в качестве таковой была выбрана **pyodbc**. Она использует технологию ODBC – стандартный интерфейс для взаимодействия приложений с системами управления базами данных.

Можно подключиться к различным базам данных, что облегчает переносимость кода.

4) Библиотека **logging** в Python предоставляет гибкий механизм для записи сообщений о событиях в приложении. Она используется для создания и настройки системы ведения журнала (логирования). Логирование полезно для записи информации о том, что происходит в приложении в процессе его выполнения, и может быть полезным для отладки, мониторинга и аудита.

**База данных**:

В качестве базы данных была выбрана Microsoft Sql Server.



Переходим непосредственно к парсеру и пройдёмся по его основным моментам и функциям:

# Конфигурация логгера для HTML-парсера

html\_logger = logging.getLogger('html\_parser')

html\_logger.setLevel(logging.INFO)

html\_handler = logging.FileHandler('html\_parser.log')

html\_formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

html\_handler.setFormatter(html\_formatter)

html\_logger.addHandler(html\_handler)

1. **html\_logger = logging.getLogger('html\_parser')**: Создается объект логгера с именем **'html\_parser'**. Имя логгера может быть любым и обычно отражает область или компонент приложения, для которого используется этот логгер.
2. **html\_logger.setLevel(logging.INFO)**: Устанавливается уровень логирования для логгера. Здесь используется уровень **INFO**, что означает, что будут логгироваться сообщения с уровнем **INFO** и более высоким.
3. **html\_handler = logging.FileHandler('html\_parser.log')**: Создается обработчик (handler) для логгера. Обработчик определяет, куда будут направлены сообщения логгера. В данном случае, создается файловый обработчик, который записывает логи в файл с именем 'html\_parser.log'.
4. **html\_formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')**: Создается форматтер, который определяет, как будет отформатирован каждый лог-запись. В данном случае, используется формат, включающий время, уровень логирования и текст сообщения.
5. **html\_handler.setFormatter(html\_formatter)**: Устанавливается форматтер для файлового обработчика.
6. **html\_logger.addHandler(html\_handler)**: Добавляется файловый обработчик к логгеру. Теперь лог-сообщения, отправленные логгером **'html\_parser'**, будут записываться в указанный файл с использованием указанного форматирования.

def get\_covid\_data():

    try:

        url = 'https://horosho-tam.ru/rossiya/coronavirus'

        response = requests.get(url)

        if response.status\_code == 200:

            soup = BeautifulSoup(response.text, 'html.parser')

            date\_cells = soup.find('tr').find\_all('th')

            current\_date = None

            if len(date\_cells) > 1:

                # Используем текущий год и текущий месяц

                date\_cell = date\_cells[1]

                date\_text = date\_cell.get\_text(strip=True)

                # Извлекаем числовую часть из текста

                day\_str = ''.join(filter(str.isdigit, date\_text))

                if day\_str:

                 current\_date = datetime(datetime.now().year, datetime.now().month, int(day\_str))

            rows = soup.find\_all('tr', {'class': ['tb\_counter\_odd', 'tb\_counter\_even']})

            country\_name = ''

            active\_cases = 0

            deaths = 0

            for row in rows:

                title\_element = row.find('strong')

                value\_element = row.find('b', {'class': 'tb\_counter\_sum'})

                if title\_element and value\_element:

                    title\_text = title\_element.text.strip()

                    value\_text = value\_element.text.strip()

                    if 'title\_active' in title\_element.get('class', []):

                        country\_name = title\_text

                        active\_cases = int(value\_text.replace('+', '').replace(' ', ''))

                    elif 'title\_deaths' in title\_element.get('class', []):

                        deaths = int(value\_text.replace(' ', ''))

            if country\_name and active\_cases is not None and deaths is not None:

                html\_logger.info("Данные успешно получены.")

                return {

                    'country': country\_name,

                    'date': current\_date,

                    'active\_cases': active\_cases,

                    'deaths': deaths

                }

            else:

                html\_logger.warning("Данные для России не найдены на странице.")

                return None

        else:

            html\_logger.error(f"Ошибка при получении данных. Код статуса: {response.status\_code}")

            return None

    except Exception as e:

        html\_logger.error(f"Ошибка при получении данных: {e}")

        return None

**1)Отправка запроса на сайт:**

* Формируется GET-запрос по указанному URL: **'https://horosho-tam.ru/rossiya/coronavirus'**.
* Полученный ответ проверяется на статусный код 200, что означает успешный запрос.

**2)Парсинг HTML-контента:**

* Если запрос успешен, используется библиотека BeautifulSoup для парсинга HTML-контента страницы.
* Находятся все ячейки **<th>** в первой строке таблицы (**<tr>**), содержащей дату.
* Извлекается текст из второй ячейки, она содержит информацию о дате.

**3)Извлечение даты:**

* Извлекается числовая часть из текста ячейки, фильтруя только цифры.
* Создается объект **datetime** с использованием текущего года и месяца, а также извлеченного числа дня.

**4)Поиск статистики COVID-19:**

* Находятся все строки таблицы с классами **tb\_counter\_odd** и **tb\_counter\_even**.
* Извлекаются информация о стране, количестве активных случаев и числе смертей.

**5)Логирование и возврат данных:**

* Если все необходимые данные успешно извлечены, происходит логирование этого события.
* Возвращается словарь с информацией о стране, дате, активных случаях и смертях.
* Если какие-то данные отсутствуют, генерируется предупреждение и возвращается **None**.
* Если возникает какая-либо ошибка в процессе выполнения кода, также генерируется ошибка и возвращается **None**.

Так как теперь данные обновляются раз в неделю, было принято решение создать доп.функцию, проверяющую, что уже прошла неделя с момента последнего парсинга данных.

def get\_last\_update\_date(cursor):

    cursor.execute('SELECT MAX(date) FROM Covid\_stats')

    last\_update\_date = cursor.fetchone()[0]

    if last\_update\_date:

        if isinstance(last\_update\_date, datetime):

            last\_update\_date = last\_update\_date.date()

        elif isinstance(last\_update\_date, str):

            last\_update\_date = datetime.strptime(last\_update\_date, '%Y-%m-%d').date()

    return last\_update\_date

1. **cursor.execute('SELECT MAX(date) FROM Covid\_stats')**: Этот вызов выполняет SQL-запрос к базе данных. Запрос **SELECT MAX(date) FROM Covid\_stats** выбирает максимальное значение из столбца **date** в таблице **Covid\_stats**.
2. **cursor.fetchone()**: Этот метод получает одну строку результата запроса. В данном случае, запрос возвращает одну ячейку со значением максимальной даты.
3. **last\_update\_date = cursor.fetchone()[0]**: Полученное значение присваивается переменной **last\_update\_date**.
4. **if last\_update\_date:**: Проверка, что значение **last\_update\_date** не является пустым или **None**.
5. **if isinstance(last\_update\_date, datetime):**: Проверка типа переменной **last\_update\_date**. Если это объект **datetime**, то ничего не делается, так как он уже в нужном формате.
6. **elif isinstance(last\_update\_date, str):**: Если переменная **last\_update\_date** имеет тип **str**, то выполняется преобразование строки в объект **datetime** с использованием метода **datetime.strptime**.
7. **last\_update\_date = datetime.strptime(last\_update\_date, '%Y-%m-%d').date()**: Преобразование объекта **datetime** в объект **date**, если это был объект **datetime**. В итоге функция возвращает максимальную дату последнего обновления в виде объекта **date** или **None**, если такой даты в базе данных нет.

def save\_to\_database(data, cursor):

    try:

        cursor.execute('''

            IF NOT EXISTS (SELECT 1 FROM INFORMATION\_SCHEMA.TABLES WHERE TABLE\_NAME = 'Covid\_stats')

            CREATE TABLE Covid\_stats (

                id INT PRIMARY KEY IDENTITY(1,1),

                date DATE,

                country NVARCHAR(255),

                cases INT,

                deaths INT,

                source NVARCHAR(255)

            )

        ''')

        last\_update\_date = get\_last\_update\_date(cursor)

        if not last\_update\_date or (datetime.now().date() - last\_update\_date).days >= 7:

            #data['country'] = 'Russia'

            cursor.execute('SELECT 1 FROM Covid\_stats WHERE date = ?', (data['date'],))

            duplicate\_exists = cursor.fetchone()

            if not duplicate\_exists:

                cursor.execute('''

                    INSERT INTO Covid\_stats (date, country, cases, deaths, source)

                    VALUES (?, ?, ?, ?, 'from\_html')

                ''', (data['date'], data['country'], data['active\_cases'], data['deaths']))

                cursor.commit()

                html\_logger.info("Данные успешно добавлены.")

            else:

                 html\_logger.warning(f"Дубликат данных для даты {data['date']} обнаружен. Пропускаем.")

        else:

            html\_logger.info("Недостаточно времени прошло с последнего обновления.")

    except Exception as e:

        html\_logger.error(f"Ошибка при сохранении данных в базу: {e}")

**1)Создание таблицы:**

SQL-запрос проверяет существование таблицы **Covid\_stats**. Если таблица не существует, она создается с полями **id** (первичный ключ), **date**, **country**, **cases**, **deaths** и **source**.

2) **Проверка последнего обновления:**

Вызывается функция **get\_last\_update\_date**, которая возвращает последнюю дату обновления из таблицы. Если эта дата не существует или прошло больше 7 дней с последнего обновления, данные могут быть обновлены.

**3)Проверка наличия дубликатов:**

Выполняется запрос к базе данных для проверки наличия записи с такой же датой. Если запись уже существует, это может быть дубликат, и добавление данных пропускается.

**4)Добавление данных в базу данных:**

Если данные не считаются дубликатом и прошло достаточно времени с последнего обновления, они добавляются в таблицу **Covid\_stats**.

5) **Логирование результатов:**

Используется **html\_logger** для записи информации о процессе сохранения данных в базу. Логирование помогает отслеживать статус операций и обнаруживать возможные проблемы.

def main():

    try:

        # Параметры подключения к SQL Server

        server = r'MY-NOTEBOOK-00\SQLEXPRESS'

        database = 'CovidData'

        username = 'MY-NOTEBOOK-00\Я'

        # Строка подключения

        connection\_string = f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={server};DATABASE={database};UID={username};Trusted\_Connection=yes;'

        # Подключение к базе данных

        conn = pyodbc.connect(connection\_string)

        cursor = conn.cursor()

        covid\_data = get\_covid\_data()

        if covid\_data:

            save\_to\_database(covid\_data, cursor)

        else:

           html\_logger.warning("Не удалось получить данные.")

        # Закрытие соединения

        conn.close()

    except Exception as e:

        html\_logger.error(f"Ошибка при выполнении программы: {e}")

**1)Параметры подключения к SQL Server:**

Задаются параметры подключения к SQL Server, включая имя сервера (**server**), имя базы данных (**database**), и имя пользователя (**username**).

2) **Строка подключения:**

Формируется строка подключения (**connection\_string**) с использованием f-строки. В строке указывается драйвер (**ODBC Driver 17 for SQL Server**), сервер, база данных, имя пользователя и флаг **Trusted\_Connection=yes**, который указывает на использование Windows-аутентификации.

3) **Подключение к базе данных:**

Создается соединение (**conn**) и курсор (**cursor**) для выполнения SQL-запросов.

**4)Получение данных о COVID-19:**

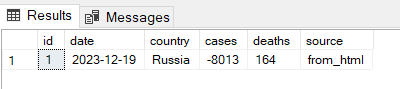
Вызывается функция **get\_covid\_data** для получения данных о COVID-19.

5) **Сохранение данных в базу данных:**

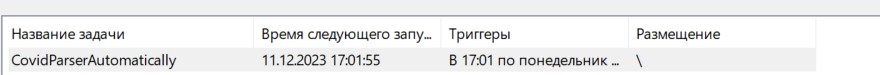
Проверяется, успешно ли были получены данные о COVID-19. Если данные успешно получены, вызывается функция **save\_to\_database** для сохранения данных в базу с использованием курсора.

6) **Закрытие соединения.**

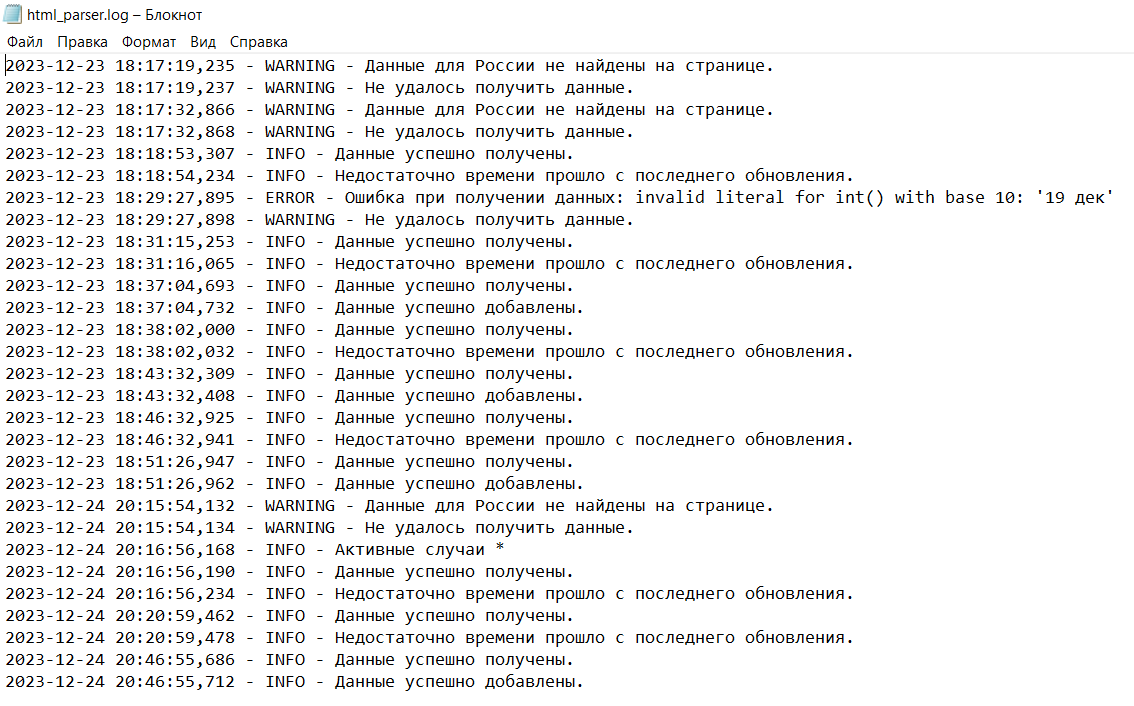
**Запись в базе данных**:



Для организации регулярного сбора данных по расписанию (еженедельно, так как информация обновляется на сайте раз в неделю) был использован планировщик задач Windows, в котором была создана задача, запускающая еженедельно в понедельник скрипт.



Пример системы логирования для html-парсера:



***2.*** Создание модуля для чтения данных из CSV файла.

Пройдёмся по основным функциям парсера:

# Конфигурация логгера для CSV-парсера

csv\_logger = logging.getLogger('csv\_parser')

csv\_logger.setLevel(logging.INFO)

csv\_handler = logging.FileHandler('csv\_parser.log')

csv\_formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

csv\_handler.setFormatter(csv\_formatter)

csv\_logger.addHandler(csv\_handler)

Аналогично как с html-парсером, тут присутствует система логирования.

def parse\_dated\_csvfile(filepath, date\_column\_name, country\_column\_name, \*other\_columns):

    plist = []

    try:

        with open(filepath, newline='') as csvfile:

            read = csv.DictReader(csvfile, delimiter=',', quotechar='|')

            for row in read:

                for k, v in row.items():

                    try:

                        v.isnumeric()

                    except:

                        row[k] = v[0]

                good\_keys = [date\_column\_name, country\_column\_name, \*other\_columns]

                bad\_keys = [key for key in row.keys() if key not in good\_keys]

                for key in bad\_keys:

                    del row[key]

                row = {k: float(v) if v.replace('.', '').isnumeric() else v for k, v in row.items()}

                row[date\_column\_name] = parse(row[date\_column\_name], dayfirst=True, yearfirst=True).date()

                if row[country\_column\_name] == 'Russia':

                    plist.append(row)

            plist.sort(key=lambda x: x[date\_column\_name])

        csv\_logger.info("Данные успешно получены.")

    except Exception as e:

        csv\_logger.error(f"Произошла ошибка при чтении CSV-файла: {e}")

        return None

    return plist

1) Создается пустой список **plist**, который будет заполнен распарсенными данными.

2) Функция использует контекстный менеджер **with**, чтобы гарантировать правильное закрытие файла после чтения. CSV-файл открывается, и его содержимое читается с использованием **csv.DictReader**, что позволяет обращаться к значениям по ключам (названиям столбцов).

3) Происходит итерация по каждой строке CSV-файла, и для каждой ячейки (**k, v**) проверяется, является ли значение **v** числовым.

4) Отбираются только нужные столбцы, указанные в аргументах функции. Все остальные столбцы удаляются из словаря **row**.

5) Для каждой строки выполняется преобразование числовых значений в тип **float**, а затем преобразование значения в столбце с датой с использованием функции **parse** из библиотеки **dateutil.parser**. Если страна в строке - 'Russia', то эта строка добавляется в список **plist**.

6) Список **plist** сортируется по значению даты.

7) В конце функции возвращается список **plist**, содержащий распарсенные данные.

8)Также присутствует логирование и обработка исключений.

def rename\_keys(record, \*cols):

    newcols = ['date', 'cases', 'deaths', 'country']

    for i in range(len(newcols)):

        record[newcols[i]] = record.pop(cols[i])

    return record

Эта функция **rename\_keys** принимает словарь **record** и переменное количество аргументов **\*cols**, представляющих собой имена столбцов, которые нужно переименовать. Внутри функции заданы новые имена столбцов в списке **newcols** (date, cases, deaths, country). Функция затем итерирует по этим новым именам, извлекает соответствующие значения из словаря **record** по старым именам (переданным в **\*cols**), и затем добавляет их обратно в словарь с новыми именами. Новые имена заданы в фиксированном порядке.

def save\_csv\_to\_database(data\_list, cursor):

    try:

        # Проверяем существование таблицы при открытии соединения

        cursor.execute('''

            IF NOT EXISTS (SELECT 1 FROM INFORMATION\_SCHEMA.TABLES WHERE TABLE\_NAME = 'Covid\_stats')

            CREATE TABLE Covid\_stats (

                id INT PRIMARY KEY IDENTITY(1,1),

                date DATE,

                country NVARCHAR(255),

                cases INT,

                deaths INT,

                source NVARCHAR(255)

            )

        ''')

        # Итерируемся по данным

        for data in data\_list:

            # Проверяем наличие дубликатов по дате

            cursor.execute('SELECT 1 FROM Covid\_stats WHERE date = ?', (data['date'],))

            duplicate\_exists = cursor.fetchone()

            if not duplicate\_exists:

                cases = int(float(data['cases']))

                deaths = int(float(data['deaths']))

                # Добавляем данные, если нет дубликатов

                cursor.execute('''

                    INSERT INTO Covid\_stats (date, country, cases, deaths, source)

                    VALUES (?, ?, ?, ?, 'from\_csv')

                ''', (data['date'], data['country'], cases, deaths))

            else:

                csv\_logger.warning(f"Дубликат данных для даты {data['date']} обнаружен. Пропускаем.")

        cursor.commit()

        csv\_logger.info("Данные успешно добавлены.")

    except Exception as e:

        csv\_logger.error(f"Произошла ошибка при сохранении данных в базу данных: {e}")

Практически аналогичное добавление данных из csv-парсера в базу данных.

# cols must be in order like 'date', 'country', 'cases', 'deaths'

def add\_csv\_data\_to\_database(cursor, csv\_filepath, \*cols):

    try:

        # Загрузка данных из CSV

        date\_column\_name = cols[0]

        country\_column\_name = cols[1]

        other\_columns = [cols[2], cols[3]]

        covid\_data\_csv = parse\_dated\_csvfile(csv\_filepath, date\_column\_name, country\_column\_name, \*other\_columns)

        if covid\_data\_csv is not None:

            covid\_data\_csv = [rename\_keys(rec, date\_column\_name, other\_columns[0], other\_columns[1], country\_column\_name) for rec in covid\_data\_csv]

            # print(covid\_data\_csv)

            # Добавление данных в базу данных

            save\_csv\_to\_database(covid\_data\_csv, cursor)

    except Exception as e:

        csv\_logger.error(f"Произошла ошибка при обработке данных из CSV: {e}")

1) Из переданных аргументов **\*cols** извлекаются имена столбцов для даты, страны и других столбцов.

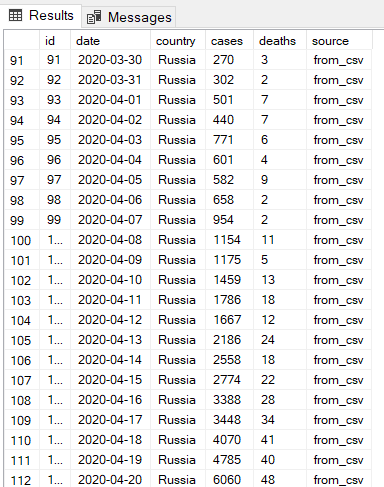
2) Вызывается функция **parse\_dated\_csvfile**, которая парсит данные из CSV-файла, используя переданные имена столбцов для даты, страны и других столбцов.

3) Для каждой записи данных из CSV вызывается функция **rename\_keys**, которая переименовывает столбцы в соответствии с заданным порядком.

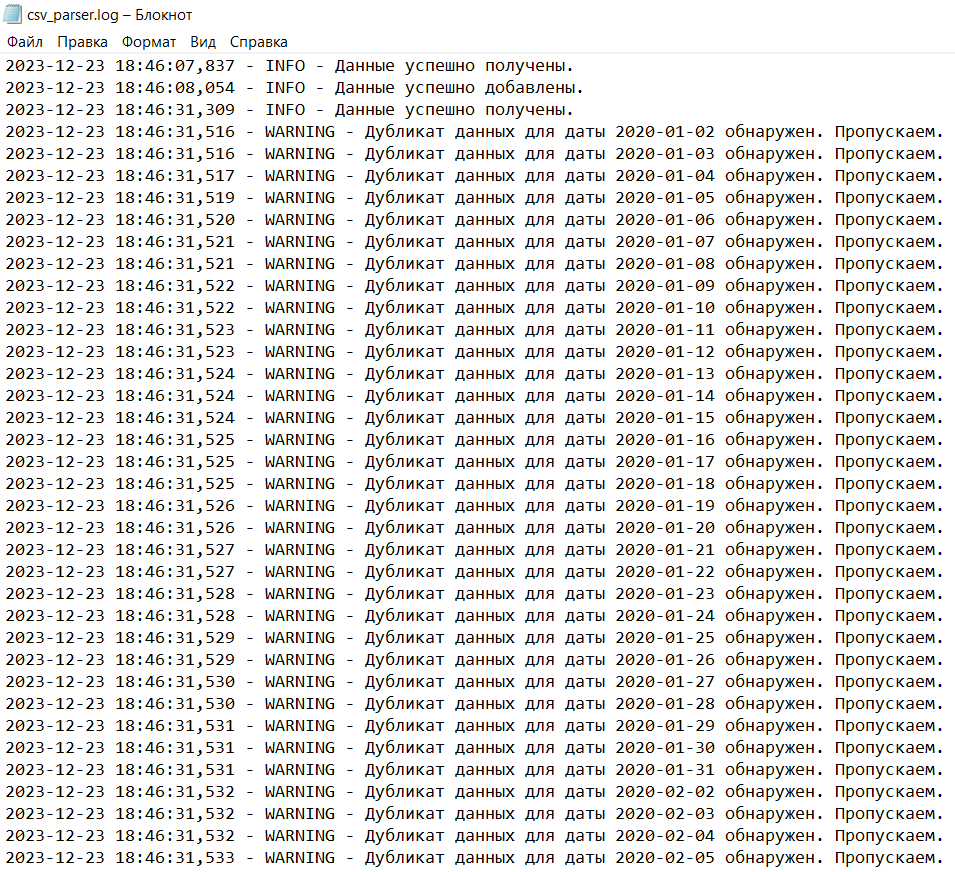
4) Вызывается функция **save\_csv\_to\_database**, которая сохраняет данные в базу данных.

5)Обработка исключений и логирование.

Записи в базе данных:

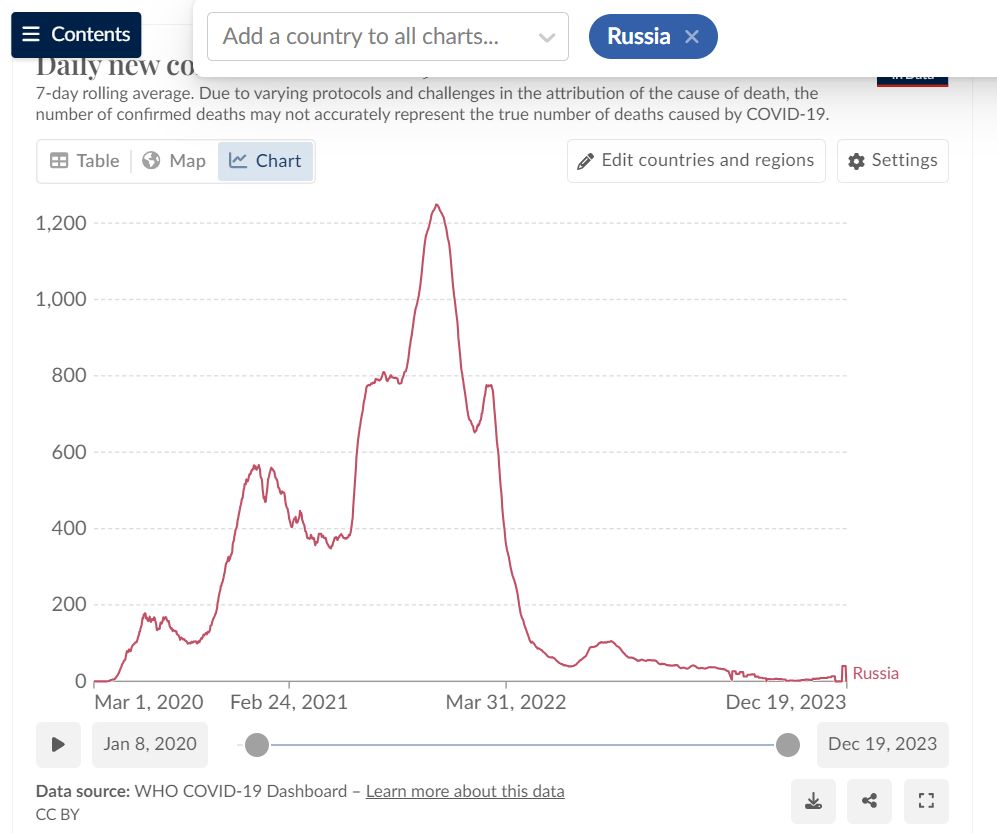


Пример системы логирования:



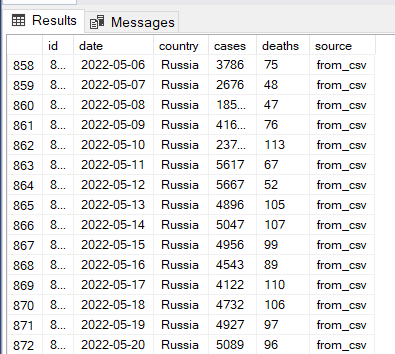
3. Найти актуальные данные после декабря 2020 года и до сентября 2023 и положить их в SQL таблицу.

Актуальные данные были найдены на сайте <https://ourworldindata.org>.



Был скачан датасет, используемый на данном сайте в формате csv и успешно спарсен благодаря нашему csv-парсеру.

Записи в базе данных:



4. Витрина данных.

REST API - это стандарт архитектурного стиля для построения веб-служб, который использует принципы и протоколы HTTP. Он предоставляет стандартизированный способ взаимодействия между клиентами и серверами, основываясь на принципе передачи состояния представления.

**Почему Flask?**

1. **Простота и гибкость:** Flask является микрофреймворком, что означает его легковесность и простоту использования. Если проект не требует больших структурных компонентов, Flask может быть отличным выбором. Он предоставляет базовый функционал, и вы можете добавлять необходимые расширения по мере необходимости.
2. **REST API:** Flask является отличным инструментом для создания RESTful API. Он предоставляет простой и понятный способ создания конечных точек API, обработки HTTP-запросов и возврата данных в формате JSON.
3. **Расширяемость:** Flask легко расширяется с использованием различных сторонних библиотек и расширений. Вы можете выбирать только те компоненты, которые вам действительно нужны, что делает приложение более легким и управляемым.
4. **Доступность обучающих материалов:** Flask обладает большим сообществом разработчиков, и для него доступно множество обучающих ресурсов. Это облегчает изучение и решение возможных проблем.

Выбор использования JSON (JavaScript Object Notation) в REST API обоснован несколькими преимуществами:

1. **Легкость чтения и записи:** JSON предоставляет простой и легко читаемый формат данных. Это важно для взаимодействия между клиентами и серверами, где человек может легко интерпретировать данные.
2. **Поддержка множества языков программирования:** JSON поддерживается множеством языков программирования, что делает его универсальным и обеспечивает простую интеграцию с различными технологиями.
3. **Легкая передача данных по сети:** JSON представляет собой легкий формат данных, что делает его эффективным для передачи по сети. Это особенно важно в случае REST API, где эффективность передачи данных по HTTP играет важную роль.
4. **Поддержка различных структур данных:** JSON поддерживает различные типы данных, включая строки, числа, логические значения, массивы и объекты. Это обеспечивает гибкость в представлении различных видов информации.
5. **Широкое распространение и стандартизация:** JSON является широко распространенным форматом данных в веб-разработке и имеет стандартизированный синтаксис. Это делает его предпочтительным выбором для обмена данными в веб-приложениях.

Теперь перейдём непосредственно к Rest-api и разберём его основные эндпоинты:

Для начала по классике организуем систему логирования и подключимся к нашей базе данных:

# Конфигурация логгера для Flask-приложения

logging.basicConfig(filename='app.log', level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

 # Параметры подключения к SQL Server

server = r'MY-NOTEBOOK-00\SQLEXPRESS'

database = 'CovidData'

username = 'MY-NOTEBOOK-00\Я'

# Строка подключения

connection\_string = f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={server};DATABASE={database};UID={username};Trusted\_Connection=yes;'

conn = pyodbc.connect(connection\_string)

Механизм проверки наличия и валидности API-ключа в заголовке HTTP запроса.

def require\_api\_key(view\_function):

    @wraps(view\_function)

    def decorated\_function(\*args, \*\*kwargs):

        api\_key = request.headers.get('Api-Key')

        if api\_key and api\_key in API\_KEYS.values():

            # Логирование успешного доступа

            logging.info(f'Successful access with API key: {api\_key}')

            return view\_function(\*args, \*\*kwargs)

        else:

            error\_message = 'Unauthorized access'

            logging.warning(error\_message)

            return jsonify({'error': error\_message}), 401

    return decorated\_function

1. **@wraps(view\_function)**: Это декоратор **@wraps** из модуля **functools**. Он используется для копирования метаданных (например, названия и документации) из оригинальной функции **view\_function** в обернутую функцию **decorated\_function**.
2. **def decorated\_function(\*args, \*\*kwargs):**: Объявление обернутой функции, которая заменяет оригинальную функцию **view\_function**.
3. **api\_key = request.headers.get('Api-Key')**: Извлечение значения API-ключа из заголовка запроса. Заголовок с именем 'Api-Key' используется для передачи API-ключа.
4. **if api\_key and api\_key in API\_KEYS.values():**: Проверка наличия API-ключа и его соответствия одному из значений в словаре **API\_KEYS**.
5. **return view\_function(\*args, \*\*kwargs)**: Если API-ключ валиден, вызывается оригинальная функция **view\_function** с переданными аргументами.
6. **return jsonify({'error': error\_message}), 401**: Возврат JSON-ответа с информацией об ошибке и статусом HTTP 401 (Unauthorized) в случае невалидного доступа.

Дополнительная функция для получения данных из базы, используя фильтрацию, с дальнейшим возвращение результатов в формате JSON:

def get\_covid\_data\_util():

    try:

        with pyodbc.connect(connection\_string) as conn:

            cursor = conn.cursor()

            # Get request parameters for filtering

            start\_date = request.args.get('start\_date')

            end\_date = request.args.get('end\_date')

            min\_cases = request.args.get('min\_cases')

            max\_cases = request.args.get('max\_cases')

            min\_deaths = request.args.get('min\_deaths')

            max\_deaths = request.args.get('max\_deaths')

            # Construct query based on filters

            query = "SELECT \* FROM Covid\_stats"

            filters = []

            if start\_date:

                filters.append(f"DATE >= '{start\_date}'")

            if end\_date:

                filters.append(f"DATE <= '{end\_date}'")

            if min\_cases:

                filters.append(f"CASES >= '{min\_cases}'")

            if max\_cases:

                filters.append(f"CASES <= '{max\_cases}'")

            if min\_deaths:

                filters.append(f"DEATHS >= '{min\_deaths}'")

            if max\_deaths:

                filters.append(f"DEATHS <= '{max\_deaths}'")

            if filters:

                query += " WHERE " + " AND ".join(filters)

            # Execute query

            cursor.execute(query)

            data = cursor.fetchall()

            # Convert data to JSON format

            result = []

            for row in data:

                result.append({

                    'id': row.id,

                    'date': row.date.strftime('%Y-%m-%d'),

                    'location': row.country,

                    'cases': row.cases,

                    'deaths': row.deaths

                })

    except Exception as e:

        print(f"Произошла ошибка: {e}")

    return result

1. **with pyodbc.connect(connection\_string) as conn:**: Используется контекстный менеджер **with**, чтобы автоматически закрыть соединение с базой данных после завершения блока кода. **pyodbc.connect(connection\_string)** создает соединение с базой данных с использованием строки подключения **connection\_string**.
2. **cursor = conn.cursor()**: Создается объект курсора для выполнения SQL-запросов.
3. **start\_date = request.args.get('start\_date')**: Извлечение значения параметра 'start\_date' из строки запроса. То же самое выполняется для других параметров фильтрации, таких как 'end\_date', 'min\_cases', 'max\_cases', 'min\_deaths', 'max\_deaths'.
4. **Construct query based on filters (Построение запроса на основе фильтров)**: Строится SQL-запрос, учитывая заданные фильтры. Если, например, указан параметр 'start\_date', то добавляется условие **DATE >= '{start\_date}'**, и так далее для других фильтров.
5. **if filters:**: Если были применены какие-либо фильтры, они добавляются к базовому запросу с использованием оператора WHERE и объединяются с помощью оператора AND.
6. **cursor.execute(query)**: Выполнение SQL-запроса с учетом всех фильтров.
7. **data = cursor.fetchall()**: Извлечение всех строк, соответствующих запросу, из результата выполнения запроса.
8. **Convert data to JSON format (Преобразование данных в формат JSON)**: Происходит преобразование полученных данных в формат JSON. Создается список словарей, где каждый словарь представляет одну строку данных.
9. **return result**: Возвращение результата в формате JSON.

@app.route('/covid-data', methods=['GET'])

@require\_api\_key

def get\_covid\_data():

    try:

        data\_records = get\_covid\_data\_util()

        if data\_records:

            # Логирование успешного получения данных

            logging.info('Successfully retrieved COVID data.')

            return jsonify(data\_records)

        else:

            # Логирование отсутствия данных

            logging.warning('No COVID data available.')

            return jsonify({'error': 'No data'})

    except Exception as e:

        # Логирование ошибки

        error\_message = f'Error : {e}'

        logging.error(error\_message)

        return jsonify({'error': error\_message}), 500

Этот фрагмент кода представляет собой маршрут Flask-приложения для обработки HTTP GET-запросов по пути '/covid-data'. Давайте разберем основные элементы:

1. **@app.route('/covid-data', methods=['GET'])**: Декоратор маршрута определяет, какой URL будет обрабатываться этой функцией и какие HTTP-методы принимаются. В данном случае, функция будет обрабатывать только GET-запросы по пути '/covid-data'.
2. **@require\_api\_key**: Декоратор **require\_api\_key** применяется к функции. Этот декоратор представляет собой механизм проверки наличия API-ключа в заголовке запроса. Если ключ отсутствует или не совпадает с ожидаемыми ключами, возвращается HTTP-ответ с кодом 401 (Unauthorized).
3. **def get\_covid\_data():**: Сама функция, которая будет вызываться при GET-запросе по пути '/covid-data'. Внутри функции происходит вызов функции **get\_covid\_data\_util()**, которая возвращает данные о COVID-19 из базы данных с учетом заданных фильтров.
4. **if data\_records:**: Проверка наличия данных после вызова **get\_covid\_data\_util()**. Если данные есть, они возвращаются в формате JSON с использованием **jsonify**. В случае отсутствия данных возвращается JSON-объект с сообщением об ошибке.
5. **Логирование**: Различные события (успешное получение данных, отсутствие данных, ошибка) логируются с использованием модуля **logging**. Логи помогают отслеживать и анализировать процессы работы приложения.
6. **Обработка исключений**: Если происходит ошибка во время выполнения функции, она логируется, и возвращается HTTP-ответ с кодом 500 (Internal Server Error) и сообщением об ошибке в формате JSON.

@app.route('/covid-data-page', methods=['GET'])

def get\_covid\_data\_html():

    try:

        data\_records = get\_covid\_data\_util()

        if data\_records:

            logging.info('HTML page successfully rendered with data.')

            return render\_template('records.html', records=data\_records, colnames=data\_records[0].keys())

        else:

            logging.warning('No data available.')

            return jsonify({'error': 'No data'})

    except Exception as e:

        error\_message = f'Error rendering HTML page: {e}'

        logging.error(error\_message)

        return jsonify({'error': error\_message}), 500

1. **@app.route('/covid-data-page', methods=['GET'])**: Этот декоратор устанавливает маршрут для обработки HTTP GET-запросов по пути '/covid-data-page'. То есть, при обращении к этому пути будет вызываться функция, определенная ниже.
2. **def get\_covid\_data\_html():**: Эта функция обрабатывает GET-запросы по пути '/covid-data-page'. Внутри функции вызывается функция **get\_covid\_data\_util()**, которая возвращает данные о COVID-19 из базы данных.
3. **if data\_records:**: После получения данных происходит проверка на их наличие. Если данные присутствуют, происходит логирование успешного рендеринга HTML-страницы, и вызывается **render\_template**. Эта функция отвечает за рендеринг HTML-страницы с использованием шаблона 'records.html'. В этот шаблон передаются данные и имена столбцов (ключей) для отображения.
4. **else:**: Если данных нет, логируется предупреждение, и возвращается JSON-объект с сообщением об ошибке.
5. **Обработка исключений**: Если происходит ошибка во время выполнения функции, она логируется, и возвращается HTTP-ответ с кодом 500 (Internal Server Error) и сообщением об ошибке в формате JSON.

# Новый маршрут для получения данных о конкретном элементе по ID

@app.route('/covid-data/<int:item\_id>', methods=['GET'])

@require\_api\_key

def get\_covid\_data\_by\_id(item\_id):

    cursor = conn.cursor()

    try:

        # Подготовка и выполнение запроса

        query = f"SELECT \* FROM Covid\_stats WHERE id = {item\_id}"

        cursor.execute(query)

        data = cursor.fetchone()

        # Проверка наличия данных

        if data:

            result = {

                'id': data.id,

                'date': data.date.strftime('%Y-%m-%d'),

                'location': data.country,

                'cases': data.cases,

                'deaths': data.deaths

            }

            logging.info(f'Successfully retrieved data for item with ID {item\_id}.')

            return jsonify(result)

        else:

            error\_message = f'Item with ID {item\_id} not found.'

            logging.warning(error\_message)

            return jsonify({'error': error\_message}), 404

    except Exception as e:

        error\_message = f'Error retrieving data for item with ID {item\_id}: {e}'

        logging.error(error\_message)

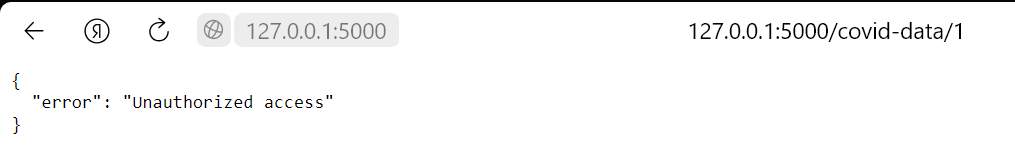
        return jsonify({'error': error\_message}), 500

1. **@app.route('/covid-data/<int:item\_id>', methods=['GET'])**: Этот декоратор устанавливает маршрут для обработки HTTP GET-запросов по пути '/covid-data/int:item\_id', где **<int:item\_id>** является переменной в пути и указывает на тип данных (целое число). Это означает, что при обращении к этому пути ожидается целочисленный параметр, который передается в функцию в качестве **item\_id**.
2. **@require\_api\_key**: Этот декоратор обеспечивает проверку наличия и валидности API ключа перед выполнением функции. Если ключа нет или он невалиден, возвращается HTTP-ответ с кодом 401 (Unauthorized).
3. **def get\_covid\_data\_by\_id(item\_id):**: Эта функция обрабатывает GET-запросы по пути '/covid-data/int:item\_id'. Внутри функции выполняется SQL-запрос для получения данных о COVID-19 для элемента с заданным ID.
4. **query = f"SELECT \* FROM Covid\_stats WHERE id = {item\_id}"**: Подготовка SQL-запроса для выборки данных из таблицы 'Covid\_stats' по заданному ID.
5. **cursor.execute(query)**: Выполнение SQL-запроса.
6. **data = cursor.fetchone()**: Получение результата запроса.
7. **if data:**: Проверка наличия данных. Если данные присутствуют, они преобразуются в формат JSON и возвращаются в HTTP-ответе с кодом 200 (OK).
8. **else:**: Если данных нет, возвращается HTTP-ответ с кодом 404 (Not Found) и сообщением об ошибке.
9. **Обработка исключений**: В случае возникновения ошибки во время выполнения функции, она логируется, и возвращается HTTP-ответ с кодом 500 (Internal Server Error) и сообщением об ошибке в формате JSON.

Пример системы логирования:

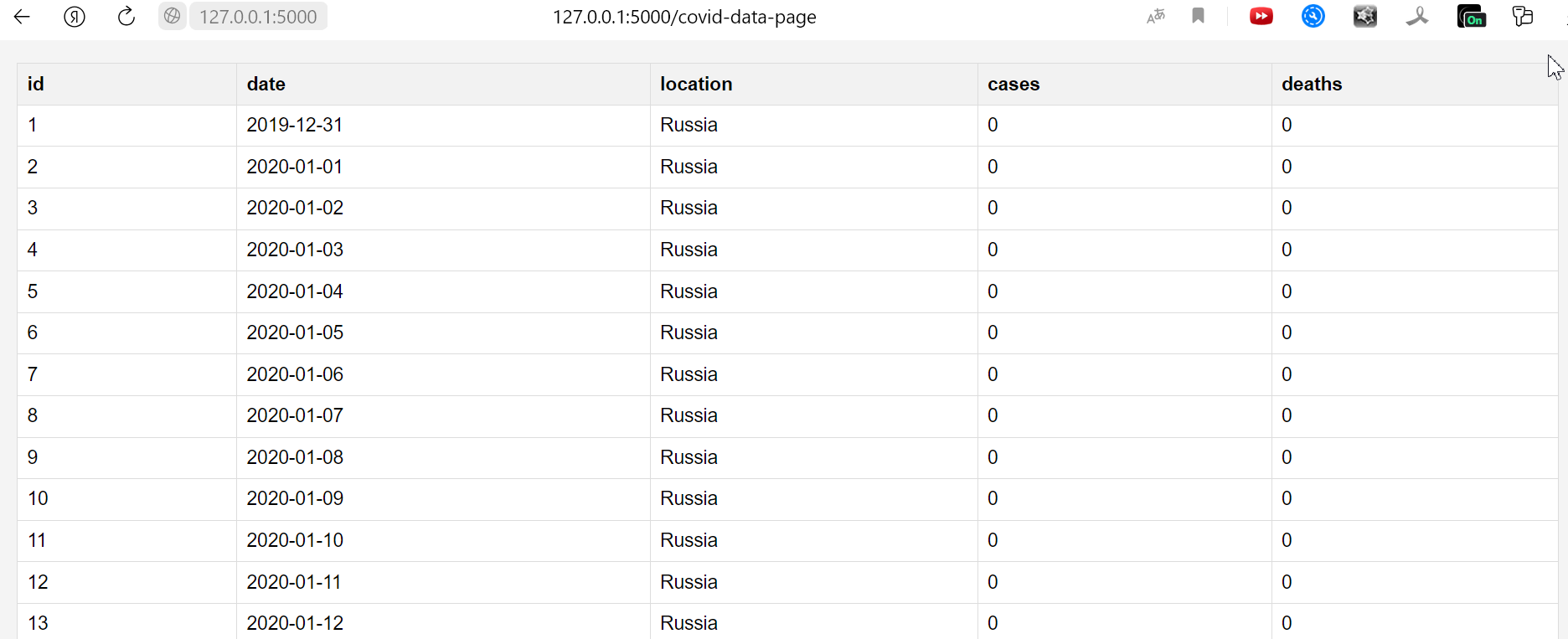


Пример отсутствия доступа без ключей:





Html-отображение данных:



Пример скриптов для фильтрации:

Конечная дата:

curl -X GET "http://127.0.0.1:5000/covid-data?end\_date=2022-12-31" -H "Api-Key:123user"

Начальная дата:

curl -X GET "http://127.0.0.1:5000/covid-data?start\_date=2022-01-01" -H "Api-Key:123user"

Начальная и конечная дата:

curl -X GET "http://127.0.0.1:5000/covid-data?start\_date=2022-01-01&end\_date=2022-12-31" -H "Api-Key:123user"

Минимальное количество смертей:

curl -X GET "http://127.0.0.1:5000/covid-data?min\_cases=5000" -H "Api-Key:123user"

Максимально количество смертей:

curl -X GET "http://127.0.0.1:5000/covid-data?max\_cases=1000" -H "Api-Key:123user"

Минимальное и максимальное количество смертей:

curl -X GET "http://127.0.0.1:5000/covid-data?min\_cases=1000&max\_cases=2000" -H "Api-Key:123user"

Минимальное количество смертей:

curl -X GET "http://127.0.0.1:5000/covid-data?min\_deaths=100" -H "Api-Key:123user"

Максимальное количество смертей:

curl -X GET "http://127.0.0.1:5000/covid-data?max\_deaths=100" -H "Api-Key:123user"

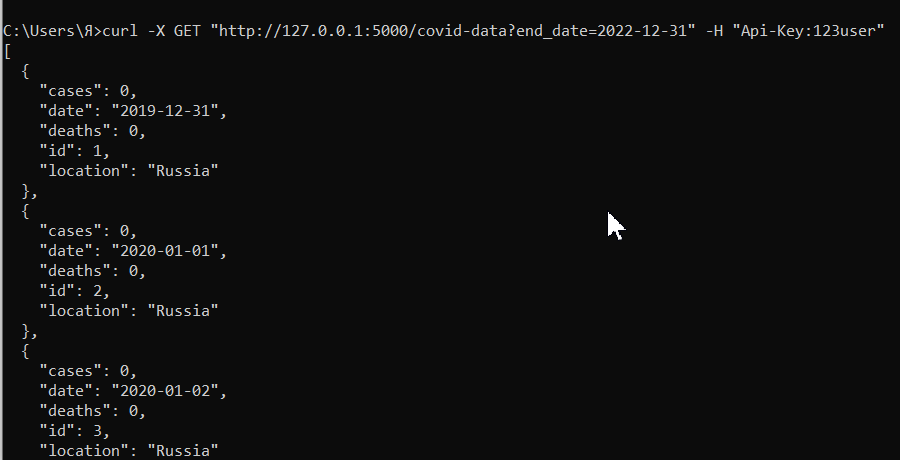
Минимальное и максимальное количество смертей:

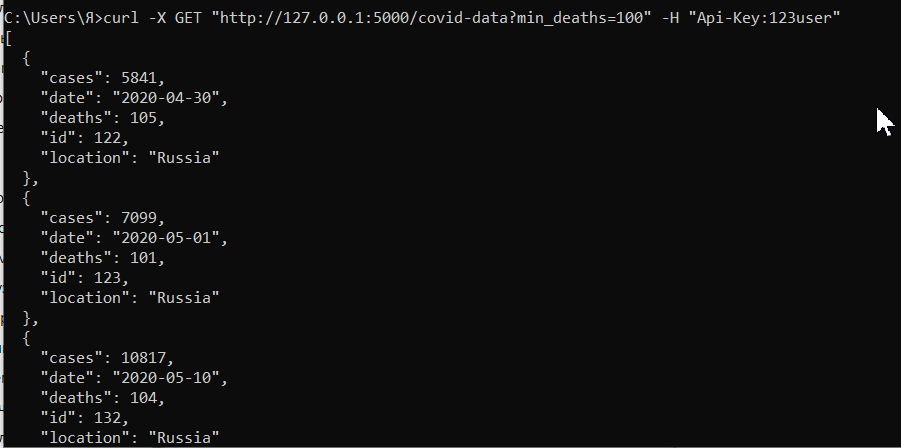
curl -X GET "http://127.0.0.1:5000/covid-data?min\_deaths=100&max\_deaths=200" -H "Api-Key:123user"

Также возможны комбинации этих фильтров

Как видно, в каждом запросе указывается Api-Key, без которого доступ к данным не будет получен.

Пример получения данных по запросу:





5. Docker

Docker - это открытая платформа для разработки, доставки и запуска приложений в контейнерах. Контейнеры представляют изолированные и легковесные среды, которые включают в себя все необходимое для запуска приложения, включая код, библиотеки, зависимости и настройки. Docker обеспечивает надежность и повторяемость при разработке и развертывании приложений.

Почему используют Docker:

* **Портативность:** Контейнеры обеспечивают портативность приложений между различными окружениями. Если приложение работает в контейнере на одной машине, оно с большой вероятностью будет работать и на другой, где установлен Docker.
* **Изоляция и безопасность:** Контейнеры обеспечивают изоляцию приложений, что означает, что они не влияют на другие контейнеры или системные ресурсы. Это также повышает безопасность, поскольку контейнеры могут быть запущены с минимальными привилегиями.
* **Масштабируемость:** Docker упрощает процесс масштабирования приложений. Вы можете легко запускать несколько экземпляров контейнеров для обработки трафика или задачи в масштабе.
* **Управление зависимостями:** Docker позволяет упаковывать зависимости и окружение приложения в контейнер, что облегчает управление зависимостями и предотвращает конфликты версий.
* **Удобство развертывания:** С Docker вы можете легко идентифицировать и развернуть приложение, включая все его зависимости, в любом окружении, поддерживающем Docker.

Dockerfile:

FROM python:3.13-rc-alpine3.18

WORKDIR /app

COPY . /app

RUN apk update

RUN apk add python3-dev libc-dev g++ libffi-dev libxml2 unixodbc-dev build-base gnupg

RUN apk --no-cache add curl sudo

RUN curl -O https://download.microsoft.com/download/e/4/e/e4e67866-dffd-428c-aac7-8d28ddafb39b/msodbcsql17\_17.10.5.1-1\_amd64.apk \

    && curl -O https://download.microsoft.com/download/e/4/e/e4e67866-dffd-428c-aac7-8d28ddafb39b/mssql-tools\_17.10.1.1-1\_amd64.apk \

    #(Optional) Verify signature, if 'gpg' is missing install it using 'apk add gnupg':

    && curl -O https://download.microsoft.com/download/e/4/e/e4e67866-dffd-428c-aac7-8d28ddafb39b/msodbcsql17\_17.10.5.1-1\_amd64.sig \

    && curl -O https://download.microsoft.com/download/e/4/e/e4e67866-dffd-428c-aac7-8d28ddafb39b/mssql-tools\_17.10.1.1-1\_amd64.sig \

    && curl https://packages.microsoft.com/keys/microsoft.asc  | gpg --import - \

    && gpg --verify msodbcsql17\_17.10.5.1-1\_amd64.sig msodbcsql17\_17.10.5.1-1\_amd64.apk \

    && gpg --verify mssql-tools\_17.10.1.1-1\_amd64.sig mssql-tools\_17.10.1.1-1\_amd64.apk \

    #Install the package(s)

    && sudo apk add --allow-untrusted msodbcsql17\_17.10.5.1-1\_amd64.apk \

    && sudo apk add --allow-untrusted mssql-tools\_17.10.1.1-1\_amd64.apk

RUN pip install -r requirements.txt

EXPOSE 5000

CMD ["python", "app.py"]

1. **FROM python:3.13-rc-alpine3.18**: Определяет базовый образ для вашего приложения. Используется официальный образ Python с тегом 3.13-rc-alpine3.18, который основан на Alpine Linux.
2. **WORKDIR /app**: Устанавливает рабочий каталог внутри контейнера как /app. Все последующие команды будут выполняться в этом каталоге.
3. **COPY . /app**: Копирует содержимое текущего каталога (все файлы и каталоги) внутрь контейнера в /app.
4. **RUN apk update**: Обновляет индекс пакетов Alpine.
5. **RUN apk add python3-dev libc-dev g++ libffi-dev libxml2 unixodbc-dev build-base gnupg**: Устанавливает несколько зависимостей и инструментов, необходимых для сборки некоторых библиотек в дальнейших шагах.
6. **RUN apk --no-cache add curl sudo**: Устанавливает curl и sudo. Curl используется для загрузки файлов, а sudo для выполнения команд с повышенными привилегиями.
7. **RUN curl -O ...**: Загружает и устанавливает Microsoft ODBC Driver и Microsoft SQL Server Command Line Tools. Эти инструменты используются для взаимодействия с SQL Server из Python.
8. **RUN pip install -r requirements.txt**: Устанавливает зависимости Python, перечисленные в файле requirements.txt.
9. **EXPOSE 5000**: Определяет порт 5000, который будет использоваться для внешнего доступа к вашему приложению.
10. **CMD ["python", "app.py"]**: Задает команду, которая будет выполнена при запуске контейнера. Запускает ваше приложение с помощью Python и файла app.py.

Requirments.txt:

blinker==1.7.0

click==8.1.7

colorama==0.4.6

Flask==3.0.0

itsdangerous==2.1.2

Jinja2==3.1.2

MarkupSafe==2.1.3

pyodbc==5.0.1

Werkzeug==3.0.1