МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Костромской государственный университет»

(КГУ)

Институт физико-математических и естественных наук

Кафедра защиты информации

Направление 10.03.01 – Информационная безопасность

Профиль Организация и технология защиты информации

Дисциплина Базы данных

КУРСОВОЙ ПРОЕКТ

База данных для Шиномонтажного сервиса: Работа с PostgreSQL в Docker и  
 создание взаимодействующего микросервиса

Выполнил студент

Григорьев Данил Сергеевич

Группа 21-ИБбо-6

Проверил к. т. н., доцент кафедры защиты информации

Волков Антон Андреевич

Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись преподавателя\_\_\_\_\_\_\_\_\_\_\_\_

Кострома

2023

Оглавление

[ВВЕДЕНИЕ 2](#_Toc155459412)

[1.ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 3](#_Toc155459413)

[2.ПРАКТИЧЕСКАЯ ЧАСТЬ 5](#_Toc155459414)

[1. Создаём БД Шиномонтажного сервиса в pgAdmin 4 5](#_Toc155459415)

[2. Создаём файл с именем «Dockerfile» в директории нашей БД 8](#_Toc155459416)

[3. Сборка Docker-образа 10](#_Toc155459417)

[4. Запуск контейнера 11](#_Toc155459418)

[5. Создание CRUD Микросервиса с API, который с взаимодействует БД 13](#_Toc155459419)

[ЗАКЛЮЧЕНИЕ 18](#_Toc155459420)

[СПИСОК ЛИТЕРАТУРЫ 19](#_Toc155459421)

# ВВЕДЕНИЕ

В условиях современной информационной эры, где данные становятся ключевым ресурсом для успешной деятельности организаций, эффективное управление базами данных становится неотъемлемой частью разработки и обеспечения бизнес-процессов. Одним из важных аспектов этого процесса является использование передовых технологий контейнеризации для развертывания и управления базами данных.

Цель данной курсовой работы заключается в исследовании и применении технологии контейнеризации Docker для создания и эффективного управления базой данных PostgreSQL. Основной фокус исследования направлен на приобретение навыков развертывания и управления СУБД PostgreSQL в контейнеризированной среде.

В ходе работы мы углубимся в процесс создания базы данных, специально адаптированный к особенностям контейнерной среды, используя Docker. Особое внимание уделим практическим аспектам интеграции PostgreSQL в контейнеры, включая шаги по развертыванию, конфигурированию и обеспечению безопасности.

Однако, наряду с этим, значительную роль в рамках курсовой работы будет играть создание CRUD-микросервиса с API для взаимодействия с нашей базой данных. Этот компонент работы предоставит не только теоретический обзор контейнеризации PostgreSQL, но и конкретный пример практической реализации микросервиса, обеспечивающего основные операции с данными.

Изучение и практическое применение этих технологий позволит приобрести не только теоретические знания, но и конкретные практические навыки, необходимые для успешного использования PostgreSQL в контейнерах. Это создаст надежную основу для будущей работы с данными в условиях современного программного обеспечения и инфраструктуры.

# 1.ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

**Docker** - это открытая платформа для разработки, доставки и запуска приложений в контейнерах. Контейнеры представляют собой легковесные, автономные и исполняемые пакеты программного обеспечения, включающие в себя все необходимые для работы компоненты: код, библиотеки, зависимости, конфигурации и т. д. Docker обеспечивает стандартизацию окружения, что упрощает процесс разработки, тестирования и развертывания приложений.

Преимущества Docker включают в себя быструю развертывание, легкость масштабирования, изоляцию приложений и возможность использования контейнеров на разных средах, таких как локальная разработка, тестирование и продакшн. Docker также способствует повышению эффективности разработки и уменьшению различий между окружениями разработчиков и окружениями в продакшне.

Контейнер в технологии Docker представляет собой легковесное, автономное исполняемое окружение, которое включает в себя все необходимое для запуска приложения: код, среду выполнения, системные библиотеки и настройки. Контейнеры изолированы друг от друга и от окружающей системы, что делает их портативными и воспроизводимыми.

В контексте Docker:

* **Образ контейнера** (Container Image): Это шаблон, который включает в себя все необходимые компоненты для запуска приложения. Образ контейнера является статичным и неизменным. Он содержит файловую систему, среду выполнения и другие зависимости.
* **Контейнер** (Container): Это экземпляр образа контейнера, который запущен и работает в изолированном окружении. Контейнер использует ядро хост-системы, но имеет свою собственную файловую систему, процессы и сеть.

**PostgreSQL** - это мощная открытая объектно-реляционная система управления базами данных (ORDBMS). Она использует реляционную модель данных, предоставляет полную совместимость с SQL, обеспечивает масштабируемость, поддерживает транзакции для обеспечения целостности данных, легко расширяется и предоставляет возможности для работы с объектами. PostgreSQL широко применяется в различных областях, таких как веб-разработка, геоинформационные системы и анализ данных.

**CRUD** — это акроним, обозначающий четыре основные операции, выполняемые в системах управления базами данных (СУБД) или веб-приложениях для управления данными. Вот расшифровка каждого компонента:

1. **Create (Создание):**
   * Операция, позволяющая создавать новые записи, объекты или данные в базе данных.
2. **Read (Чтение):**
   * Операция, позволяющая извлекать информацию из базы данных, то есть читать данные.
3. **Update (Обновление):**
   * Операция, которая позволяет изменять существующие записи или объекты в базе данных.
4. **Delete (Удаление):**
   * Операция, позволяющая удалять записи или объекты из базы данных.

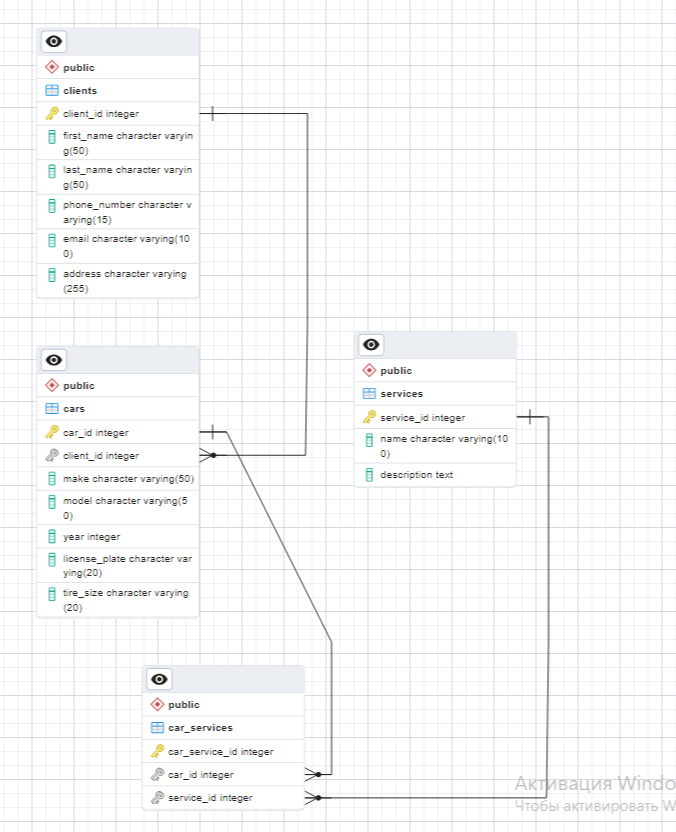
Эти операции представляют собой основные действия, которые можно выполнить с данными в системе. Концепция CRUD широко используется в разработке программного обеспечения и баз данных для организации и управления информацией. API (Интерфейс программирования приложений) и веб-приложения часто реализуют функциональность CRUD для взаимодействия с данными.

**API (Интерфейс программирования приложений)** - это набор определенных правил и инструментов, который позволяет программам взаимодействовать между собой. API определяет способы, как различные компоненты программного обеспечения должны взаимодействовать друг с другом.

API может быть представлен различными способами, включая библиотеки, протоколы, классы, функции, или даже веб-сервисы. Он определяет, какие запросы можно делать к программе или сервису, какие данные и в каком формате можно получить в ответ, и какие действия доступны для выполнения.

# 2.ПРАКТИЧЕСКАЯ ЧАСТЬ

## 1. Создаём БД Шиномонтажного сервиса в pgAdmin 4

  
Рис. 1

Создаём 4 таблицы (клиенты, автомобили, услуги и история применения услуг для авто) и заполняем их

CREATE TABLE services (

service\_id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

description TEXT

);

INSERT INTO services (service\_id, name, description)

VALUES

('1', 'Сезонная замена шин', 'Замена шин в соответствии с текущим сезоном'),

('2', 'Балансировка колес', 'Балансировка передних и задних колес'),

('3', 'Ремонт проколов', 'Восстановление проколотых шин');

CREATE TABLE clients (

client\_id SERIAL PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

last\_name VARCHAR(50) NOT NULL,

phone\_number VARCHAR(15) NOT NULL,

email VARCHAR(100),

address VARCHAR(255)

);

INSERT INTO clients (client\_id, first\_name, last\_name, phone\_number, email, address)

VALUES

('1', 'Иван', 'Иванов', '123-456-7890', 'ivan@example.com', 'ул. Пушкина, д.1'),

('2', 'Мария', 'Петрова', '987-654-3210', 'maria@example.com', 'ул. Лермонтова, д.2');

CREATE TABLE cars (

car\_id SERIAL PRIMARY KEY,

client\_id INTEGER REFERENCES clients(client\_id),

make VARCHAR(50) NOT NULL,

model VARCHAR(50) NOT NULL,

year INTEGER,

license\_plate VARCHAR(20) NOT NULL,

tire\_size VARCHAR(20)

);

INSERT INTO cars (car\_id, client\_id, make, model, year, license\_plate, tire\_size)

VALUES

('1', '1', 'Toyota', 'Camry', 2018, 'ABC123', '225/60R16'),

('2', '2', 'Honda', 'Accord', 2020, 'XYZ789', '205/55R17');

CREATE TABLE car\_services (

car\_service\_id SERIAL PRIMARY KEY,

car\_id INTEGER REFERENCES cars(car\_id),

service\_id INTEGER REFERENCES services(service\_id)

);

INSERT INTO car\_services (car\_service\_id, car\_id, service\_id)

VALUES

('1', '1', '1'), -- Toyota Camry выбрал услугу "Сезонная замена шин"

('2', '2', '2'), -- Honda Accord выбрал услугу "Балансировка колес"

('3', '2', '3'); -- Honda Accord выбрал услугу "Ремонт проколов"

**Сохраняем нашу БД (shinomontage.sql) в отдельную директорию:**

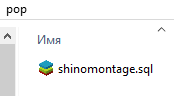


Рис. 2

## 2. Создаём файл с именем «Dockerfile» в директории нашей БД

Записываем внутри файла данный код:

**FROM postgres:latest**

**ENV POSTGRES\_DB=shinomontazh**

**ENV POSTGRES\_USER=lessed**

**ENV POSTGRES\_PASSWORD=12345666**

**COPY shinomontage.sql /docker-entrypoint-initdb.d/**

**CMD ["postgres"]**

И сохраняем его в ту же директорию, где и наша БД.

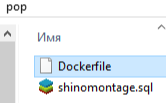


Рис. 3

В предоставленных строках Dockerfile происходит создание Docker-образа для PostgreSQL с определенными параметрами и настройками. Давайте разберем каждую строку:

* **FROM postgres:latest –** указывает, что для создания нового образа мы будем использовать официальный образ PostgreSQL с тегом latest
* **POSTGRES\_DB –** устанавливает название базы данных, которая будет создана при первом запуске контейнера
* **POSTGRES\_USER –** устанавливает имя пользователя для доступа к базе данных
* **POSTGRES\_PASSWORD –** устанавливает пароль для пользователя
* **COPY shinomontage.sql /docker-entrypoint-initdb.d/ –** копирует файл *shinomontage.sql* из текущей директории (где находится Dockerfile) внутрь образа в папку /docker-entrypoint-initdb.d/. В этой папке PostgreSQL автоматически выполняет SQL-скрипты при инициализации базы данных.
* **FROM postgres:latest** – определяет команду по умолчанию для запуска контейнера. В данном случае, при запуске контейнера, будет выполнена команда postgres, что означает запуск сервера PostgreSQL**.**

**Dockerfile —** это текстовый файл, который содержит инструкции для сборки Docker-образа. Dockerfile определяет, как должен выглядеть и работать контейнер внутри Docker-окружения.

## 3. Сборка Docker-образа

В командной строке мы сначала заходим в директорию, где находится наш *Dockerfile,* а затем прописываем команду:  
docker build –t postgresql\_montage .

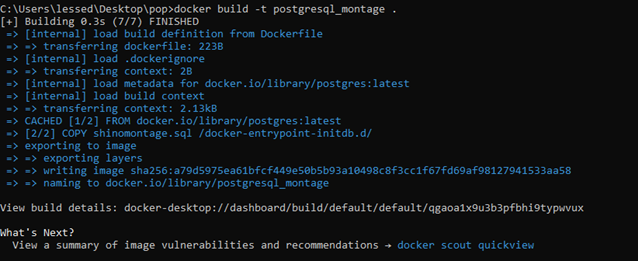


Рис. 4

Сборка Docker-образа — это процесс создания "шаблона" для контейнеров. Он описывает все необходимые зависимости, настройки и команды, которые должны быть выполнены при создании контейнера из этого образа. Это необходимый этап, который превращает приложение и его зависимости в контейнер, который может быть запущен и использован в любой среде, где установлен Docker.

Сборка образа:

* **docker build:** Эта команда запускает процесс сборки образа Docker. Она принимает несколько параметров, но основными являются:
* **-t postgresql\_montage:** Опция -t используется для указания тега (имени) образа. В данном случае, мы называем образ postgresql\_montage. Теги позволяют идентифицировать и версионировать образы.
* **.** : Последний аргумент в команде docker build представляет собой контекст сборки, то есть директорию, в которой находится Dockerfile и другие файлы, необходимые для сборки образа. В данном случае, **.** указывает на текущую директорию.

## 4. Запуск контейнера

Также прописываем в командной строке:

**docker run -d -p 5432:5432 --name my-shinomontazhka-container postgresql\_montage**

* **docker run:** Команда docker run используется для создания и запуска контейнера из определенного Docker-образа. Она принимает несколько параметров, и в данном случае, мы используем следующие:
  + -**d**: Опция -d означает запуск контейнера в фоновом режиме (detached mode). Таким образом, после запуска контейнера, вы получите обратную связь в виде контейнерного идентификатора.
  + **-p 5432:5432**: Опция -p используется для проброса портов между хостовой машиной и контейнером. В данном случае, мы пробрасываем порт 5432, на котором работает PostgreSQL, с хостовой машины на порт 5432 внутри контейнера.
  + **--name my-shinomontazhka-container**: Опция --name позволяет назначить имя контейнеру. В данном случае, мы дали ему имя my-shinomontazhka-container.
  + **postgresql\_montage**: Это имя образа, из которого будет создан контейнер.

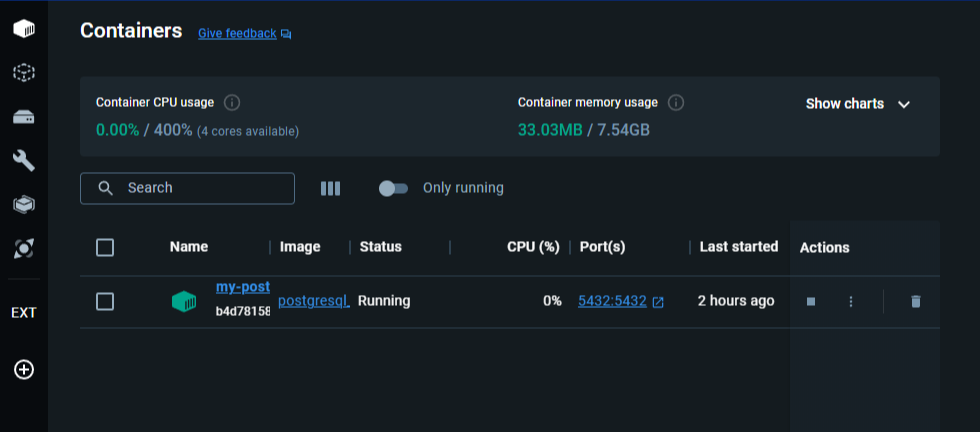
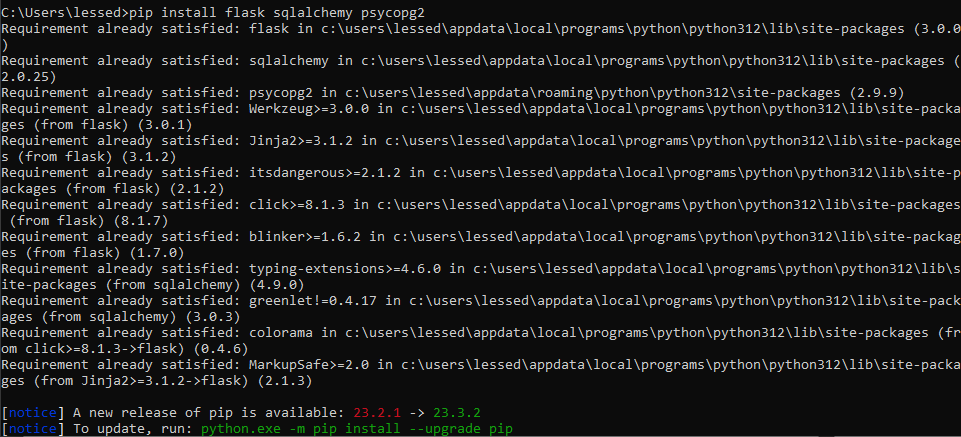


Рис. 5

Таким образом, мы создали и запустили контейнер с PostgreSQL, содержащий базу данных для шиномонтажного сервиса. Это практическое применение позволяет вам ознакомиться с основными шагами работы с Docker и PostgreSQL в контейнеризированной среде.

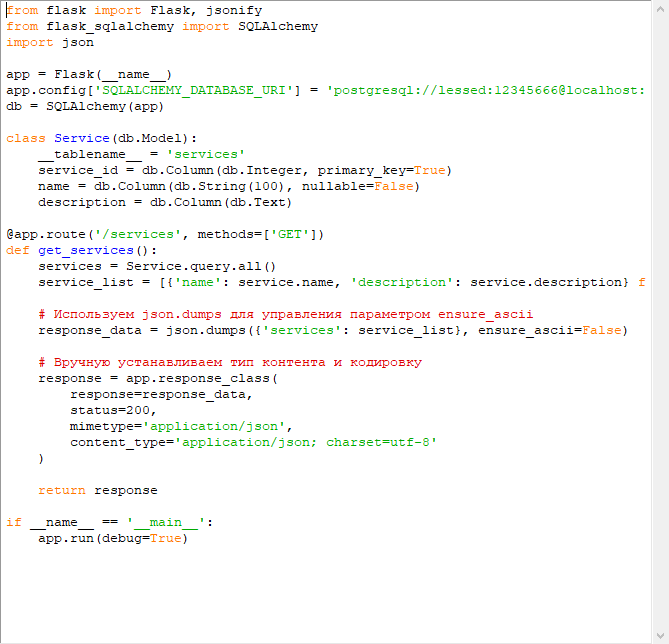
## Создание CRUD Микросервиса с API, который с взаимодействует БД

Для начала нам нужно установить необходимые библиотеки:  
Прописываем в командной строке - **pip install flask sqlalchemy psycopg2**Рис. 6

1. **Flask**: Flask - это микрофреймворк для создания веб-приложений на языке Python. Он предоставляет простой способ создания веб-приложений и API. В данном коде Flask используется для создания веб-приложения, обработки HTTP-запросов и возврата JSON-ответов.
2. **SQLAlchemy**: SQLAlchemy - это библиотека для работы с базами данных в языке Python. Она предоставляет инструменты для объектно-реляционного отображения (ORM), что позволяет работать с базой данных, используя объекты Python, вместо SQL-запросов. В коде SQLAlchemy используется для определения модели данных (**Service**), которая соответствует таблице в базе данных, и для выполнения запросов к этой базе данных.
3. Flask-SQLAlchemy, которую мы используем, может использовать драйвер базы данных для взаимодействия с PostgreSQL, и по умолчанию, для PostgreSQL, Flask-SQLAlchemy использует **psycopg2** в качестве драйвера.

Объединив эти библиотеки, код реализует простой веб-сервер с использованием Flask, обращается к базе данных PostgreSQL с использованием SQLAlchemy для получения данных о сервисах, и отправляет эти данные клиенту в формате JSON.

Далее создаем файл для нашего Микросервиса (в моем случае это “f.py”)

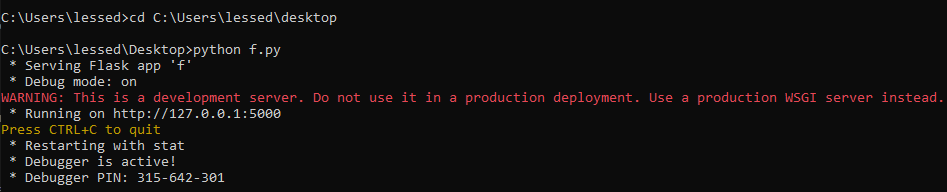
Где прописываем:   
Рис. 7

1. **from flask import Flask, jsonify**: Импортируются класс **Flask** и функция **jsonify** из библиотеки Flask. **Flask** используется для создания веб-приложений на языке Python, а **jsonify** - для удобного возврата JSON-ответов.
2. **from flask\_sqlalchemy import SQLAlchemy**: Импортируется расширение SQLAlchemy для работы с базой данных в Flask. SQLAlchemy обеспечивает объектно-реляционное отображение (ORM) и упрощает взаимодействие с базой данных.
3. **import json**: Импортируется стандартная библиотека Python для работы с JSON.
4. **app = Flask(\_\_name\_\_)**: Создается объект приложения Flask.
5. **app.config['SQLALCHEMY\_DATABASE\_URI'] = 'postgresql://lessed:12345666@localhost:5432/shinomontazh'**: Устанавливается конфигурация для подключения к базе данных PostgreSQL. Эта строка задает URI для подключения к базе данных **shinomontazh** с именем пользователя **lessed** и паролем **12345666**.
6. **db = SQLAlchemy(app)**: Создается объект SQLAlchemy, связанный с приложением Flask.
7. **class Service(db.Model):**: Определяется модель данных **Service** для работы с таблицей **services** в базе данных.
8. **\_\_tablename\_\_ = 'services'**: Устанавливается имя таблицы в базе данных для модели **Service**.
9. **service\_id = db.Column(db.Integer, primary\_key=True)**: Определяется поле **service\_id** типа Integer в качестве первичного ключа.
10. **name = db.Column(db.String(100), nullable=False)**: Определяется поле **name** типа String с максимальной длиной 100 символов и запретом на значения NULL.
11. **description = db.Column(db.Text)**: Определяется поле **description** типа Text (для более длинных текстов).
12. **@app.route('/services', methods=['GET'])**: Декоратор маршрута для обработки HTTP-запросов к **/services** методом GET.
13. **def get\_services():**: Объявление функции для обработки запросов к **/services**.
14. **services = Service.query.all()**: Получение всех записей из таблицы **services** с использованием SQLAlchemy.
15. **service\_list = [{'name': service.name, 'description': service.description} for service in services]**: Создание списка словарей с данными о сервисах из полученных записей.
16. **response\_data = json.dumps({'services': service\_list}, ensure\_ascii=False)**: Преобразование списка словарей в JSON-строку с использованием **json.dumps**. **ensure\_ascii=False** позволяет сохранить не-ASCII символы в JSON-строке.
17. Создание объекта ответа Flask:

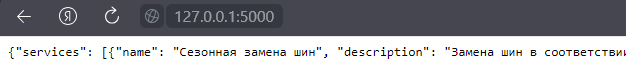
* **response=response\_data**: Устанавливается тело ответа.
* **status=200**: Устанавливается код статуса 200 (успех).
* **mimetype='application/json'**: Устанавливается тип контента.
* **content\_type='application/json; charset=utf-8'**: Устанавливается кодировка контента.

1. **return response**: Возвращается объект ответа Flask.
2. **if \_\_name\_\_ == '\_\_main\_\_':**: Проверка, запускается ли скрипт напрямую (не импортируется ли он в другом файле).
3. **app.run(debug=True)**: Запуск встроенного сервера Flask с включенным режимом отладки (debug mode).

После чего сохраняем файл и в командной строке запускаем наш микросервис:

Рис. 8

Проверяем:  
Прописываем в адресной строке браузера - **http://127.0.0.1:5000/services**  
Где нам выводятся данные про сервисы из нашей БД

Рис. 9

# **ЗАКЛЮЧЕНИЕ**

В заключение нашей курсовой работы подчеркиваем, что освоение технологии контейнеризации Docker в сочетании с управлением базой данных PostgreSQL является важным аспектом разработки в условиях современной информационной эры. В свете растущей значимости данных для бизнес-процессов, эффективное управление базами данных становится стратегически важным.

Целью нашей работы было исследование и практическое применение Docker для управления PostgreSQL. Мы углубились в создание базы данных, адаптированной к контейнерной среде, с акцентом на практических аспектах интеграции PostgreSQL в контейнеры. Особенное внимание было уделено созданию CRUD-микросервиса с API, что предоставило не только теоретический обзор, но и конкретный пример успешной реализации.

Полученные практические навыки и знания создают прочную основу для дальнейшей работы с данными в условиях современного программного обеспечения и инфраструктуры. Результаты нашей работы подчеркивают актуальность применения контейнеров в управлении базами данных и подчеркивают важность практического опыта для эффективного использования этих технологий в реальных проектах.

# СПИСОК ЛИТЕРАТУРЫ

* <https://habr.com/ru/companies/ruvds/articles/438796/> - Изучаем Docker, часть 1: основы
* <https://habr.com/ru/articles/310460/> - Полное практическое руководство по Docker: с нуля до кластера на AWS
* <https://dzen.ru/video/watch/63d29e64339130538939e47d?f=d2d> - Введение в Docker
* <https://habr.com/ru/articles/578744/> - Запускаем PostgreSQL в Docker: от простого к сложному
* <https://learn.microsoft.com/ru-ru/dotnet/architecture/microservices/multi-container-microservice-net-applications/data-driven-crud-microservice> - Создание простой микрослужбы CRUD на основе данных
* <https://flask.palletsprojects.com/en/latest/> - Flask Documentation
* <https://docs.sqlalchemy.org/en/20/> - SQLAlchemy Documentatio
* <https://medium.com/geekculture/crud-operations-explained-2a44096e9c88> - Что такое CRUD?