

**РОССИЙСКОЙ ФЕДЕРАЦИИ**  
**Федеральное государственное автономное**  
**образовательное учреждение высшего образования**  
**«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

**Кафедра инфокоммуникаций**  
**«Основы работы с Dockerfile»**

**Отчет по лабораторной работе**  
**по дисциплине «Анализ данных»**

Выполнил студент группы ИВТ-б-о-21-1

Богдашов Артём Владимирович

Подпись студента \_\_\_\_\_

Работа защищена « » \_\_\_\_\_ 20\_\_ г.

Проверил Воронкин Р.А. \_\_\_\_\_  
(подпись)

Ставрополь 2023

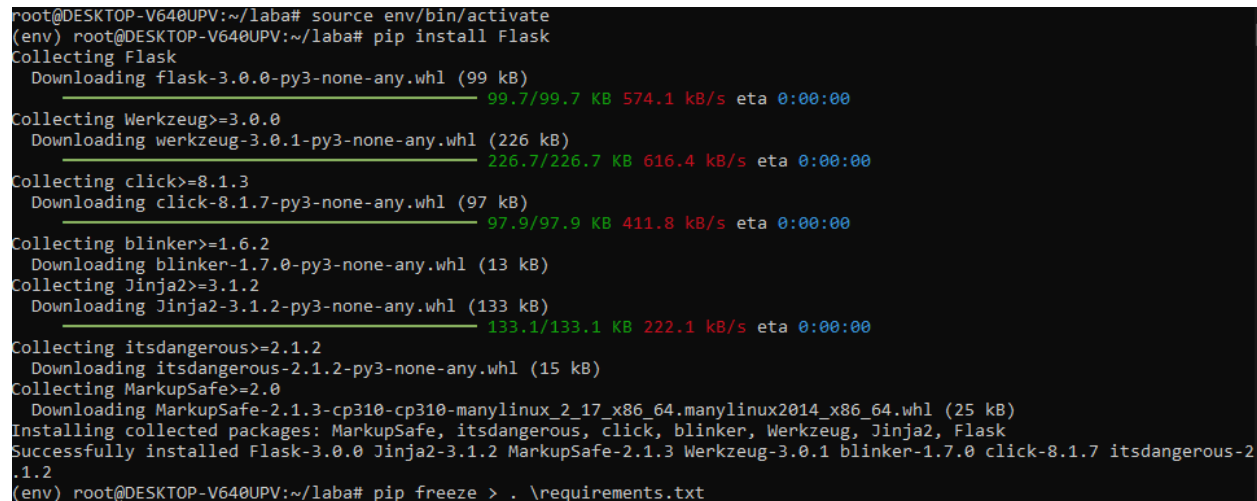
**Цель работы:** овладеть навыками создания и управления контейнерами Docker для разработки, доставки и запуска приложений. Понимание процесса создания Dockerfile, сборки и развертывания контейнеров Docker, а также оптимизации их производительности и безопасности.

### **Ход работы:**

## **Создание простого web-приложения на Python с использованием Dockerfile.**

Создайте проект веб-приложения на Python, включая код приложения и необходимые файлы.

Настраиваем виртуальное окружение:



```
root@DESKTOP-V640UPV:~/laba# source env/bin/activate
(env) root@DESKTOP-V640UPV:~/laba# pip install Flask
Collecting Flask
  Downloading flask-3.0.0-py3-none-any.whl (99 kB)
    99.7/99.7 KB 574.1 kB/s eta 0:00:00
Collecting Werkzeug>=3.0.0
  Downloading werkzeug-3.0.1-py3-none-any.whl (226 kB)
    226.7/226.7 KB 616.4 kB/s eta 0:00:00
Collecting click>=8.1.3
  Downloading click-8.1.7-py3-none-any.whl (97 kB)
    97.9/97.9 KB 411.8 kB/s eta 0:00:00
Collecting blinker>=1.6.2
  Downloading blinker-1.7.0-py3-none-any.whl (13 kB)
Collecting Jinja2>=3.1.2
  Downloading Jinja2-3.1.2-py3-none-any.whl (133 kB)
    133.1/133.1 KB 222.1 kB/s eta 0:00:00
Collecting itsdangerous>=2.1.2
  Downloading itsdangerous-2.1.2-py3-none-any.whl (15 kB)
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-2.1.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (25 kB)
Installing collected packages: MarkupSafe, itsdangerous, click, blinker, Werkzeug, Jinja2, Flask
Successfully installed Flask-3.0.0 Jinja2-3.1.2 MarkupSafe-2.1.3 Werkzeug-3.0.1 blinker-1.7.0 click-8.1.7 itsdangerous-2.1.2
(env) root@DESKTOP-V640UPV:~/laba# pip freeze > .\requirements.txt
```

Рисунок 1. Настройка виртуального окружения

Код программы на Python (файл app.py):

```
# app.py
from flask import Flask, request

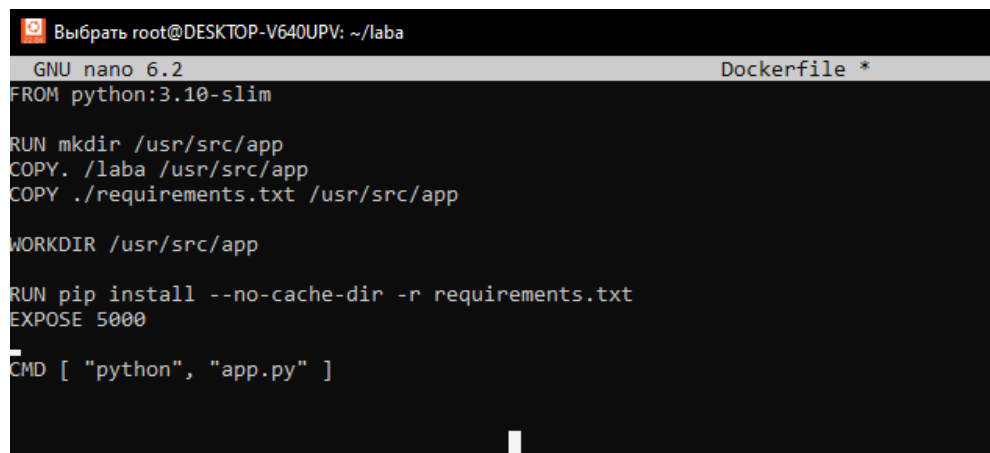
app = Flask(__name__)

@app.route('/')
def hello():
    user_name = request.args.get('name', 'Guest')
    return f'Hello, {user_name}!'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
Код html:
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>FlaskApp</title>
</head>
<body>
  <h1>{{ message }}</h1>
</body>
</html>
```

Создайте Dockerfile для сборки образа Docker вашего приложения. Определите инструкции для сборки образа, включая копирование файлов, установку зависимостей и настройку команд запуска.

A terminal window titled 'Выбрать root@DESKTOP-V640UPV: ~/laba' shows the GNU nano 6.2 editor editing a file named 'Dockerfile \*'. The content of the Dockerfile is as follows:

```
FROM python:3.10-slim

RUN mkdir /usr/src/app
COPY . /usr/src/app
COPY ./requirements.txt /usr/src/app

WORKDIR /usr/src/app

RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000

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

Результат работы программы:

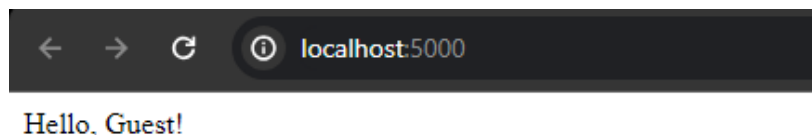


Рисунок 2. Результат работы программы в браузере

### Установка дополнительных пакетов в образ Docker.

Создайте многоэтапной Dockerfile, состоящий из двух этапов: этап сборки и этап выполнения. На этапе сборки установите дополнительный пакет, такой как библиотеку NumPy, используя команду RUN. На этапе выполнения скопируйте созданное приложение из этапа сборки и укажите команду запуска.

Код Dockerfile:

FROM python:3.10-slim

WORKDIR /usr/src/app

COPY myapp /usr/src/app

COPY ./requirements.txt /usr/src/app

RUN pip install --no-cache-dir -r requirements.txt

EXPOSE 5000

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

Сборка и запуск контейнера:

```
[+] Building 31.2s (10/10) FINISHED                                docker:default
=> [internal] load .dockerignore                                  0.0s
=> => transferring context: 2B                                    0.0s
=> [internal] load build definition from Dockerfile              0.0s
=> => transferring dockerfile: 237B                               0.0s
=> [internal] load metadata for docker.io/library/python:3.10-slim 0.8s
=> [1/5] FROM docker.io/library/python:3.10-slim@sha256:25f03d17398b3f001e040fc951b4ee9404862f1b65c5eea1aa31c04 24.6s
=> => resolve docker.io/library/python:3.10-slim@sha256:25f03d17398b3f001e040fc951b4ee9404862f1b65c5eea1aa31c042 0.0s
=> => sha256:25f03d17398b3f001e040fc951b4ee9404862f1b65c5eea1aa31c042dfdb527 1.65kB / 1.65kB 0.0s
=> => sha256:9956522e7eafd57e3e7bb4b102d56f02882924019867cd2036c1a7c3ee56b174 1.37kB / 1.37kB 0.0s
=> => sha256:3e27f11955d637ad643f1ebf37168854043fbaaafa20176af0339c01d51f1c1c 6.94kB / 6.94kB 0.0s
=> => sha256:af107e978371b6cd6339127a05502c5eacd1e6b0e9eb7b2f4aa7b6fc87e2dd81 29.13MB / 29.13MB 20.1s
=> => sha256:8ce3f2b601ccac03ff1858022363c325355bafba224123a4563dade58bc8e70f 3.51MB / 3.51MB 7.7s
=> => sha256:64119eae2e5157ca18d0ec9b9c7865e454b09bca702c3e2f579703deb0eaaeaf 12.38MB / 12.38MB 15.7s
=> => sha256:a28cd3d033268c30cb986f4a0d36c4aeda4c5aeaa5c764562f323c73d1f4f771 244B / 244B 8.8s
=> => sha256:2c1fe8bcf114038da9146a2528b2c3b606a09873aee7122f8529d45418fcd9e9 3.36MB / 3.36MB 13.4s
=> => extracting sha256:af107e978371b6cd6339127a05502c5eacd1e6b0e9eb7b2f4aa7b6fc87e2dd81 2.3s
=> => extracting sha256:8ce3f2b601ccac03ff1858022363c325355bafba224123a4563dade58bc8e70f 0.3s
=> => extracting sha256:64119eae2e5157ca18d0ec9b9c7865e454b09bca702c3e2f579703deb0eaaeaf 0.8s
=> => extracting sha256:a28cd3d033268c30cb986f4a0d36c4aeda4c5aeaa5c764562f323c73d1f4f771 0.0s
=> => extracting sha256:2c1fe8bcf114038da9146a2528b2c3b606a09873aee7122f8529d45418fcd9e9 0.5s
=> [internal] load build context                                0.0s
=> => transferring context: 249B                                    0.0s
=> [2/5] WORKDIR /usr/src/app                                  0.2s
=> [3/5] COPY myapp /usr/src/app                                0.1s
=> [4/5] COPY ./requirements.txt /usr/src/app                    0.1s
=> [5/5] RUN pip install --no-cache-dir -r requirements.txt      5.2s
=> exporting to image                                           0.2s
=> => exporting layers                                             0.2s
=> => writing image sha256:f468b426d4e8940530db981a14521f4031d424768eab49d1a442d9c0fb18b959 0.0s
=> => naming to docker.io/library/python-laba                    0.0s
```

Рисунок 3. Сборка контейнера

## Настройка переменных окружения среды в образе Docker.

Изменим Dockerfile:

# Stage 1: Build Stage

FROM python:3.10-slim AS builder

WORKDIR /usr/src/app

COPY myapp /usr/src/app

COPY requirements.txt /usr/src/app

```
# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt
```

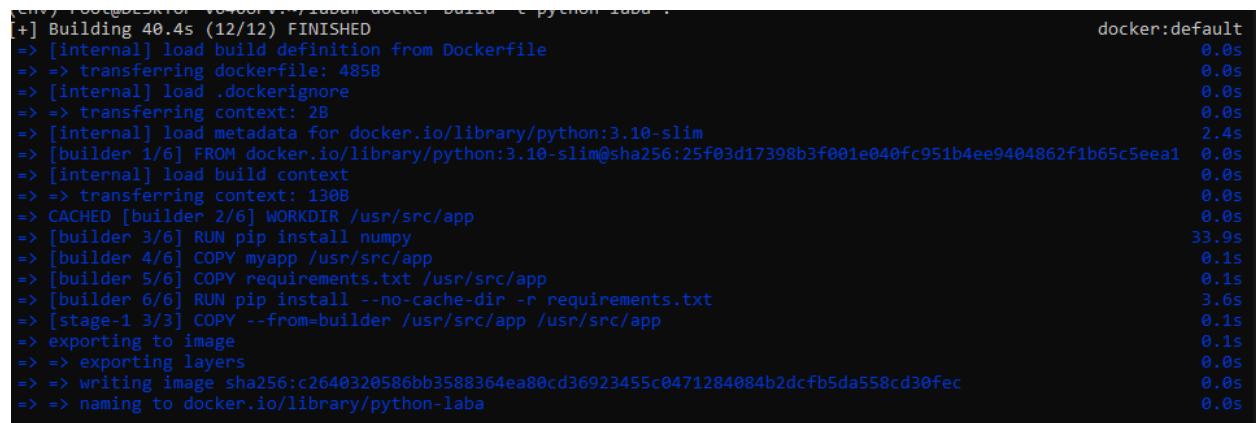
```
# Stage 2: Production Stage
FROM python:3.10-slim
```

```
WORKDIR /usr/src/app
```

```
# Copy only necessary files from the builder stage
COPY --from=builder /usr/src/app /usr/src/app
```

```
EXPOSE 5000
```

```
CMD ["python", "app.py"]
Сборка контейнера:
```



```
env: /usr/bin/docker: /usr/bin/docker build -t python-laba .
[+] Building 40.4s (12/12) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile                0.0s
=> => transferring dockerfile: 485B                                0.0s
=> [internal] load .dockerignore                                   0.0s
=> => transferring context: 2B                                       0.0s
=> [internal] load metadata for docker.io/library/python:3.10-slim 2.4s
=> [builder 1/6] FROM docker.io/library/python:3.10-slim@sha256:25f03d17398b3f001e040fc951b4ee9404862f1b65c5eea1 0.0s
=> [internal] load build context                                   0.0s
=> => transferring context: 130B                                       0.0s
=> CACHED [builder 2/6] WORKDIR /usr/src/app                       0.0s
=> [builder 3/6] RUN pip install numpy                             33.9s
=> [builder 4/6] COPY myapp /usr/src/app                           0.1s
=> [builder 5/6] COPY requirements.txt /usr/src/app                0.1s
=> [builder 6/6] RUN pip install --no-cache-dir -r requirements.txt 3.6s
=> [stage-1 3/3] COPY --from=builder /usr/src/app /usr/src/app     0.1s
=> exporting to image                                              0.1s
=> => exporting layers                                              0.0s
=> => writing image sha256:c2640320586bb3588364ea80cd36923455c0471284084b2dcfb5da558cd30fec 0.0s
=> => naming to docker.io/library/python-laba                      0.0s
```

Рисунок 4. Сборка контейнера

## Контрольные вопросы:

### 1. Что такое Dockerfile?

Dockerfile — это текстовый файл, который содержит инструкции для автоматизированного создания образа Docker. Dockerfile определяет, какие операции и конфигурации должны быть выполнены внутри контейнера при его создании.

### 2. Какие основные команды используются в Dockerfile?

FROM - Указывает базовый образ, на основе которого будет создан новый образ.

RUN - Выполняет команды в процессе создания образа.

CMD - Указывает команду, которая будет выполняться при запуске контейнера из образа.

**COPY** - Копирует файлы из хоста в образ.

**EXPOSE** - Указывает порты, которые будут открыты в контейнере.

### **3. Для чего используется команда FROM?**

**FROM:** Эта строка указывает базовый образ, который будет использоваться для сборки нового образа.

### **4. Для чего используется команда WORKDIR?**

**WORKDIR:** Эта строка устанавливает рабочую директорию для контейнера.

### **5. Для чего используется команда COPY?**

**COPY:** Эта строка копирует файлы из хоста в образ Docker.

### **6. Для чего используется команда RUN?**

**RUN:** Эта строка выполняет команды в процессе сборки образа.

### **7. Для чего используется команда CMD?**

**CMD:** Эта строка указывает команду, которая будет выполняться при запуске контейнера.

### **8. Для чего используется команда EXPOSE?**

**EXPOSE:** Эта строка указывает порты, которые должны быть открыты в контейнере.

### **9. Для чего используется команда ENV?**

Самый простой способ настроить переменную среды в образе Docker — это использовать команду ENV.

### **10. Для чего используется команда USER?**

Команда USER в Dockerfile указывает пользователя, от имени которого будет выполняться основная команда контейнера (CMD или ENTRYPOINT).

### **11. Для чего используется команда HEALTHCHECK?**

**HEALTHCHECK** — инструкции, которые Docker может использовать для проверки работоспособности запущенного контейнера.

### **12. Для чего используется команда LABEL?**

**LABEL** — описывает метаданные. Например — сведения о том, кто создал и поддерживает образ.

### **13. Для чего используется команда ARG?**

ARG — задаёт переменные для передачи Docker во время сборки образа.

### **14. Для чего используется команда ONBUILD?**

Инструкция ONBUILD добавляет к образу инструкцию-триггер, которая будет выполнена позже, когда образ будет использоваться в качестве основы для другой сборки.

### **15. Что такое многоэтапная сборка?**

Многоэтапный Dockerfile состоит из двух основных этапов:

1. Этап сборки: Этот этап отвечает за компиляцию и сборку приложения. Он использует базовый образ с необходимыми инструментами для сборки, такими как компилятор Golang и соответствующие зависимости.
2. Этап выполнения: Этот этап отвечает за запуск и выполнение приложения. Он использует более минимальный базовый образ, например, Alpine Linux, содержащий только необходимые библиотеки для выполнения приложения.

### **16. Какие преимущества использования многоэтапной сборки?**

Уменьшение размера образа: при использовании многоэтапных сборок только необходимые файлы для выполнения приложения включаются в окончательный образ, что значительно уменьшает его размер. Повышение безопасности: Многоэтапные сборки уменьшают риск уязвимостей безопасности, поскольку они изолируют этапы сборки и выполнения, ограничивая доступ к ненужным инструментам и зависимостям.

### **17. Какие недостатки использования многоэтапной сборки?**

Сложность конфигурации и поддержки процесса сборки может возрасти. Необходимо следить за последовательностью этапов, управлять зависимостями и обеспечивать корректное выполнение каждого этапа. Это требует дополнительных знаний и времени на настройку, особенно для больших и сложных проектов. Кроме того, многоэтапная сборка Docker требует наличия основного образа системы, который может быть достаточно большим и содержать лишние компоненты. Это может увеличить размер

окончательного образа, что негативно отразится на скорости его развертывания и потреблении ресурсов.

#### **18. Как определить базовый образ в Dockerfile?**

FROM node:latest

#### **19. Как определить рабочую директорию в Dockerfile?**

WORKDIR /usr/src/app

#### **20. Как скопировать файлы в образ Docker?**

COPY. /my-app /usr/src/app/

#### **21. Как выполнить команды при сборке образа Docker?**

Команда RUN выполняет команды в процессе создания образа.

#### **22. Как указать команду запуска контейнера?**

Команда CMD в Dockerfile указывает команду, которая будет выполняться при запуске контейнера. Она может состоять из одной или нескольких команд, разделенных пробелами. Команда ENTRYPOINT в Dockerfile указывает исполняемый файл, который будет использоваться в качестве основной точки входа в контейнер.

#### **23. Как открыть порты в контейнере?**

Запуск образа с флагом -p перенаправляет общедоступный порт на частный порт внутри контейнера.

#### **24. Как задать переменные среды в образе Docker?**

Самый простой способ настроить переменную среды в образе Docker – это использовать команду ENV. Вы также можете настроить переменные среды в образе Docker с помощью файла .env. Чтобы использовать файл .env для настройки переменных среды в образе Docker, вы должны добавить команду ADD .env /app/.env в Dockerfile. Вы также можете настроить переменные среды при запуске контейнера. Для этого используйте флаг --env или -e.

#### **25. Как изменить пользователя, от имени которого будет выполняться контейнер?**

При помощи команды USER.



**26. Как добавить проверку работоспособности к контейнеру?**

При помощи команды HEALTHCHECK.

**27. Как добавить метку к контейнеру?**

При помощи команды LABEL.

**28. Как передать аргументы при сборке образа Docker?**

При помощи команды ARG.

**29. Как выполнить команду при первом запуске контейнера?**

При помощи команды ENTRYPOINT.

**30. Как определить зависимости между образами Docker?**

При помощи команды ONBUILD.

**Вывод:** в ходе данной лабораторной работы были освоены навыки создания и управления контейнерами Docker для разработки, доставки и запуска приложений. Понимание процесса создания Dockerfile, сборки и развёртывания контейнеров Docker, а также оптимизации их производительности и безопасности.