

Департамент образования и науки города Москвы  
Государственное автономное образовательное учреждение высшего  
образования города Москвы  
«Московский городской педагогический университет»  
Институт цифрового образования  
Департамент информатики, управления и технологий

Макарова Екатерина Павловна

## **ЛАБОРАТОРНАЯ РАБОТА 2**

### **Создание Dockerfile и сборка образа**

Интеграция и развертывание программного обеспечения с помощью  
контейнеров

Направление подготовки

38.03.05 Бизнес-информатика

Профиль подготовки

Аналитика данных и эффективное управление

Курс обучения: 4

Форма обучения: очная

Преподаватель: кандидат технических наук,  
доцент Босенко Тимур Муртазович

Москва

2025

**Цель работы:** научиться создавать Dockerfile и собирать образы Docker для приложений.

**Задачи:**

- Создать Dockerfile для указанного приложения.
- Собрать образ Docker с использованием созданного Dockerfile.
- Запустить контейнер из собранного образа и проверить его работоспособность.
- Выполнить индивидуальное задание.

**Вариант 8.** Создайте Dockerfile для приложения на Rust, которое выводит "Hello, Rust!" при запуске.

## Ход работы

### Выполнение общего задания:

1. Создание Dockerfile для приложения на Flask
- 1.1. Создан новый каталог для проекта и осуществлён переход в него

(Рис. 1).

```
kate@beady:~$ mkdir flask-app
kate@beady:~$ ls
Projects flask-app
kate@beady:~$ cd flask-app
kate@beady:~/flask-app$ |
```

Рис. 1

- 1.2. Создание файла app.py (Рис. 2).

```
kate@beady:~/flask-app$ vim app.py
kate@beady:~/flask-app$ ls
app.py
```

Рис. 2

Содержимое созданного файла app.py (Рис. 3).

```
kate@beady: ~/flask-app
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, World!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

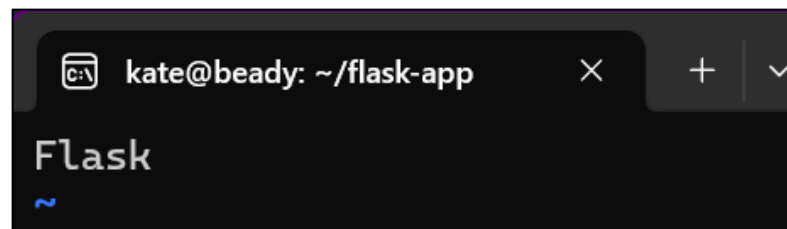
Рис. 3

- 1.3. Создание файла requirements.txt (Рис. 4).

```
kate@beady:~/flask-app$ vim requirements.txt
kate@beady:~/flask-app$ ls
app.py  requirements.txt
```

Рис. 4

Содержимое созданного файла (Рис. 5).

A screenshot of a terminal window with a dark background. The title bar shows 'kate@beady: ~/flask-app'. The terminal content shows the word 'Flask' followed by a tilde '~' on the next line, indicating the content of the requirements.txt file.

```
kate@beady: ~/flask-app
Flask
~
```

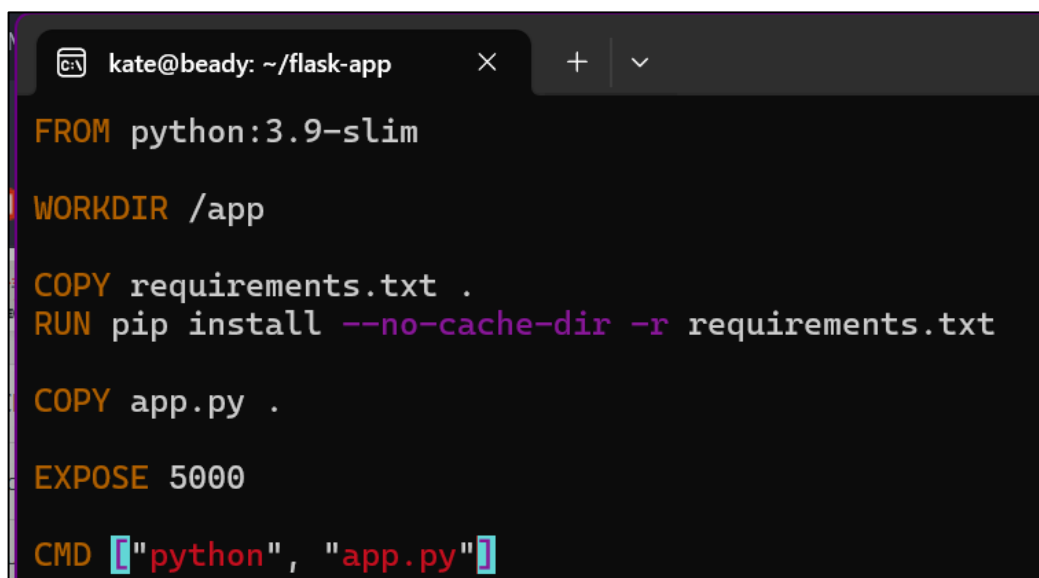
Рис. 5

1.4. Создание файла Dockerfile (Рис. 6).

```
kate@beady:~/flask-app$ vim Dockerfile
kate@beady:~/flask-app$ ls
Dockerfile  app.py  requirements.txt
```

Рис. 6

Содержимое файла (Рис. 7).

A screenshot of a terminal window with a dark background. The title bar shows 'kate@beady: ~/flask-app'. The terminal content shows the Dockerfile instructions: FROM, WORKDIR, COPY, RUN, COPY, EXPOSE, and CMD.

```
kate@beady: ~/flask-app
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py .
EXPOSE 5000
CMD ["python", "app.py"]
```

Рис. 7

2. Сборка образа Docker

2.1. Выполнение команды для сборки образа (Рис. 8).

```

kate@beady:~/flask-app$ docker buildx build -t flask-app .
[+] Building 13.0s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 204B
=> [internal] load metadata for docker.io/library/python:3.9-slim
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/python:3.9-slim@sha256:f9364cd6e0c146966f8f23fc4fd85d53f2e604bdde74e3c06565194dc
=> => resolve docker.io/library/python:3.9-slim@sha256:f9364cd6e0c146966f8f23fc4fd85d53f2e604bdde74e3c06565194dc
=> => sha256:b26995e9f45f14d5d4a45d1d0c50396482c1a6c1fcbbf5e6105cac49ef9afbba 248B / 248B
=> => sha256:7e8ac65e25aaa3b7a0cef09164fd2c6879c88b27bc9d50b5be34166a11479656 14.93MB / 14.93MB
=> => sha256:9315c4821e723a7fb42220024e56534ecd042bd9ebb91aa3487c0a90cbb9710 3.51MB / 3.51MB
=> => extracting sha256:9315c4821e723a7fb42220024e56534ecd042bd9ebb91aa3487c0a90cbb9710
=> => extracting sha256:7e8ac65e25aaa3b7a0cef09164fd2c6879c88b27bc9d50b5be34166a11479656
=> => extracting sha256:b26995e9f45f14d5d4a45d1d0c50396482c1a6c1fcbbf5e6105cac49ef9afbba
=> [internal] load build context
=> => transferring context: 258B
=> [2/5] WORKDIR /app
=> [3/5] COPY requirements.txt .
=> [4/5] RUN pip install --no-cache-dir -r requirements.txt
=> [5/5] COPY app.py .
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:c724245fa8acbe62b7819242ffffb4119fc0889077d5ba7630ed88c50a9dce75
=> => exporting config sha256:25198c0bebaeclacd69f54b2df2bcd0d7d3ba77da6fbd6b0e4afd8f68fce045
=> => exporting attestation manifest sha256:12f9c68732e95c9c648d12e057993b50e82e1defd3a20396e07b02e96ef9cd94
=> => exporting manifest list sha256:3732af5fd6677c566173a5b8e9d6d3b5426ea79b1c67ccca77bd91d3e1d088be
=> => naming to docker.io/library/flask-app:latest
=> => unpacking to docker.io/library/flask-app:latest

```

Рис. 8

### 3. Запуск контейнера

#### 3.1. Запуск контейнера из собранного образа (Рис. 9).

```

kate@beady:~/flask-app$ docker run -d --name my-flask-app -p 5000:5000 flask-app
1af8afc017a696d2874b56dfba977eec6c7fb811e089c351564bca596bc36b0a
kate@beady:~/flask-app$ |

```

Рис. 9

Выполнение команды на проверку запущенных контейнеров (Рис. 10).

```

kate@beady:~/flask-app$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                               NAMES
1af8afc017a6   flask-app "python app.py"         About a minute ago Up About a minute   0.0.0.0:5000->5000/tcp   my-flask-app

```

Рис. 10

Проверка работоспособности в веб-браузере (Рис. 11).

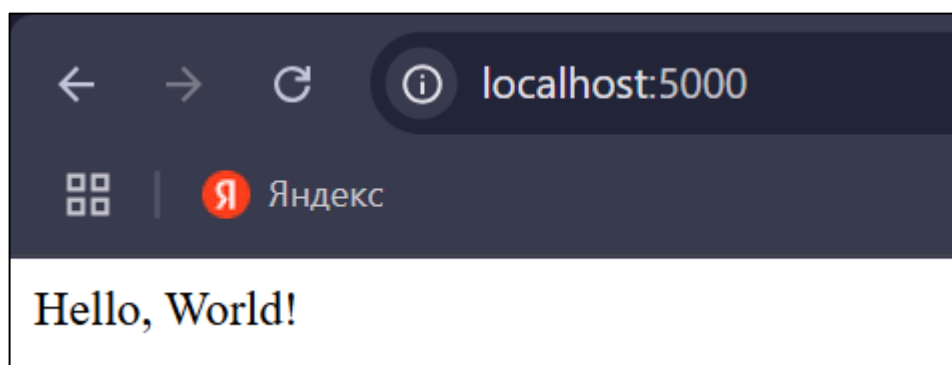


Рис. 11

### 4. Остановка и удаление контейнера

#### 4.1. Выполнение команды на остановку контейнера (Рис. 12).

```
kate@beady:~/flask-app$ docker stop my-flask-app  
my-flask-app
```

Рис. 12

4.2. Выполнение команды для удаления контейнера (Рис. 13).

```
kate@beady:~/flask-app$ docker rm my-flask-app  
my-flask-app
```

Рис. 13

### Выполнение индивидуального задания:

Вариант 8. Создайте Dockerfile для приложения на Rust, которое выводит "Hello, Rust!" при запуске.

Структура проекта для развёртывания Rust-приложения в Docker (Рис. 14).

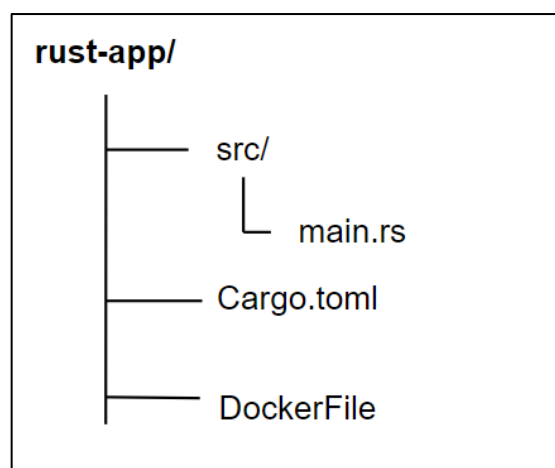


Рис. 14

Директория `src` (сокращение от "source") в Rust-проектах — это стандартная папка для хранения исходного кода.

В директории `src` необходимо создать файл `main.rs`, который будет содержать исходный код.

В файле `Cargo.toml` будут содержаться зависимости.

Файл `DockerFile` содержит конфигурацию.

1. Создание Dockerfile для приложения на Rust

1.1. Создание директории проекта и переход в неё (Рис. 15).

```
kate@beady:~$ mkdir rust-app
kate@beady:~$ ls
Projects flask-app rust-app
kate@beady:~$ cd rust-app
kate@beady:~/rust-app$ |
```

Рис. 15

1.2. Создание директории src (Рис. 16).

```
kate@beady:~/rust-app$ mkdir src
kate@beady:~/rust-app$ ls
src
```

Рис. 16

1.3. Создание файла main.rs (Рис. 17).

```
kate@beady:~/rust-app$ cd src
kate@beady:~/rust-app/src$ vim main.rs
```

Рис. 17

Внесение содержимого в файл main.rs (Рис. 18).

```
use actix_web::{get, App, HttpResponse, HttpServer, Responder};

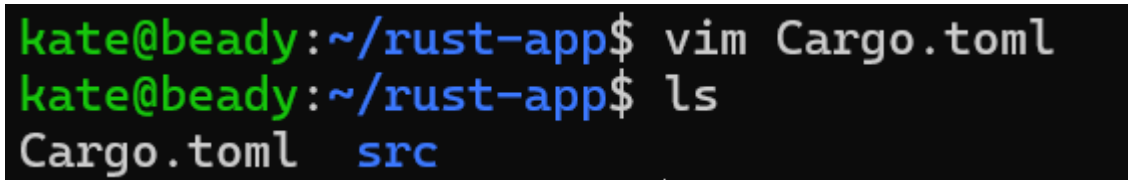
#[get("/")]
async fn hello() -> impl Responder {
    HttpResponse::Ok().body("Hello, Rust!")
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new().service(hello)
    })
    .bind("0.0.0.0:5000")?
    .run()
    .await
}
```

Рис. 18

Код создает простое веб-приложение на Rust с использованием фреймворка Actix-web. Сервер получает запросы по порту 5000. При получении HTTP GET запроса по корневому маршруту (/) он отвечает текстом "Hello, Rust!" с кодом состояния 200 (OK).

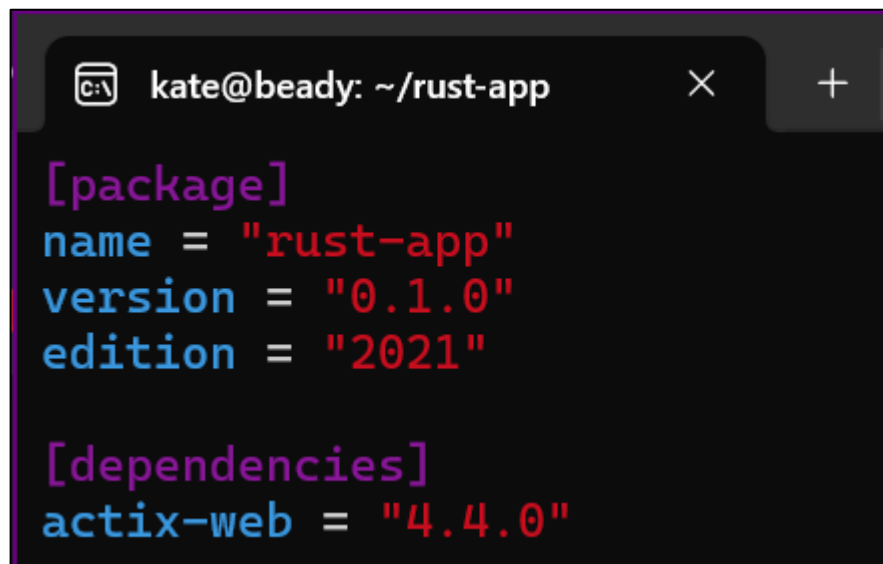
1.4. Создание файла Cargo.toml (Рис. 19).

A terminal window with a dark background. The prompt is 'kate@beady:~/rust-app\$'. The user enters 'vim Cargo.toml'. The prompt changes to 'kate@beady:~/rust-app\$'. The user enters 'ls'. The output is 'Cargo.toml src'.

```
kate@beady:~/rust-app$ vim Cargo.toml
kate@beady:~/rust-app$ ls
Cargo.toml  src
```

Рис. 19

Содержимое файла Cargo.toml (Рис. 20).

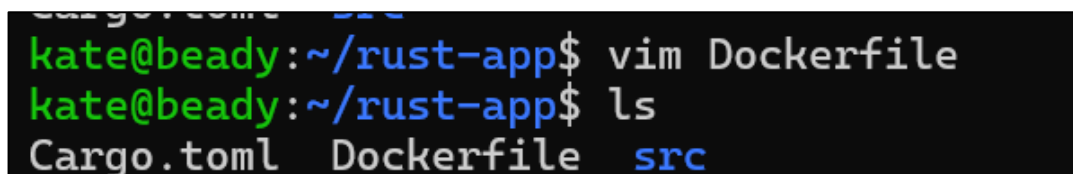
A screenshot of a code editor window titled 'kate@beady: ~/rust-app'. The file 'Cargo.toml' is open, showing the following content:

```
[package]
name = "rust-app"
version = "0.1.0"
edition = "2021"

[dependencies]
actix-web = "4.4.0"
```

Рис. 20

1.5. Создание файла Dockerfile (Рис. 21).

A terminal window with a dark background. The prompt is 'kate@beady:~/rust-app\$'. The user enters 'vim Dockerfile'. The prompt changes to 'kate@beady:~/rust-app\$'. The user enters 'ls'. The output is 'Cargo.toml Dockerfile src'.

```
kate@beady:~/rust-app$ vim Dockerfile
kate@beady:~/rust-app$ ls
Cargo.toml  Dockerfile  src
```

Рис. 21

Содержимое файла Dockerfile (Рис. 22).



```
kate@beady: ~/rust-app × + ▾  
FROM rust:1.75-slim-bookworm AS builder  
  
# Устанавливаем системные зависимости  
RUN apt-get update && \  
    apt-get install -y \  
    pkg-config \  
    libssl-dev \  
    && rm -rf /var/lib/apt/lists/*  
  
WORKDIR /app  
COPY . .  
  
# Кэшируем зависимости  
RUN cargo fetch  
  
# Собираем релиз  
RUN cargo build --release  
  
# Финальный образ  
FROM debian:bookworm-slim  
RUN apt-get update && \  
    apt-get install -y \  
    ca-certificates \  
    && rm -rf /var/lib/apt/lists/*  
  
WORKDIR /app  
COPY --from=builder /app/target/release/rust-app /app/rust-app  
  
EXPOSE 5000  
CMD ["/rust-app"]  
~  
-- INSERT --
```

Рис. 22

## 2. Сборка образа Docker.

### 2.1. Выполнение команды для сборки образа (Рис. 23).

```
kate@beady:~/rust-app$ docker build -t rust-app .  
[+] Building 166.8s (18/18) FINISHED  
=> [internal] load build definition from Dockerfile  
=> => transferring dockerfile: 682B  
=> [internal] load metadata for docker.io/library/debian:bookworm-slim  
=> [internal] load metadata for docker.io/library/rust:1.75-slim-bookworm  
=> [auth] library/rust:pull token for registry-1.docker.io  
=> [auth] library/debian:pull token for registry-1.docker.io  
=> [internal] load .dockerignore  
docker:default  
0.0s  
0.0s  
1.1s  
2.2s  
0.0s  
0.0s  
0.0s
```

Рис. 23

## 3. Запуск контейнера.

### 3.1. Выполнение команды для сборки контейнера (Рис. 24).

```
kate@beady:~/rust-app$ docker run -p 5000:5000 rust-app
```

Рис. 24

3.2. Проверка работоспособности в веб-браузере (Рис. 25).

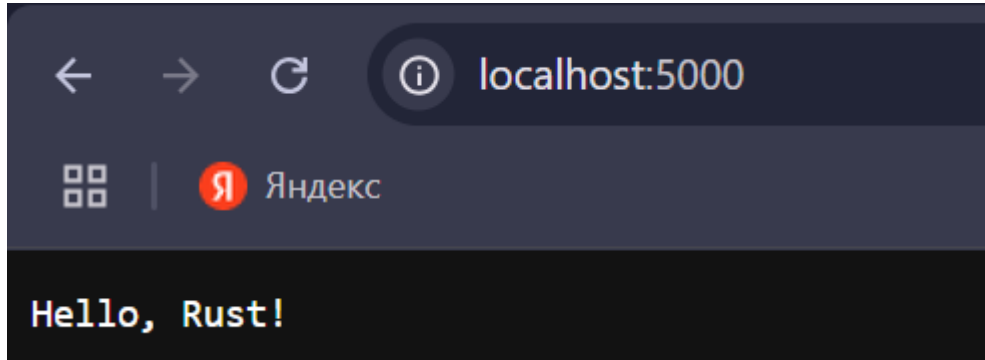


Рис. 25

4. Удаление контейнера

4.1. Просмотр всех контейнеров (Рис. 26).

```
kate@beady:~/rust-app$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c19cb7ceffd0	rust-app	"/rust-app"	2 minutes ago	Exited (0) About a minute ago		clever_
hoover						

Рис. 26

4.2. Удаление контейнера (Рис. 27).

```
kate@beady:~/rust-app$ docker rm clever_hoover  
clever_hoover
```

Рис. 27

## **Выводы по работе**

В ходе выполнения лабораторной работы успешно освоены ключевые этапы создания Docker-образов для приложений на языке Rust.

Приложение, выводящее сообщение «Hello, Rust!» через HTTP-эндпоинт с использованием фреймворка actix-web, было размещено в стандартной структуре проекта (директория src/), что соответствует требованиям экосистемы Rust. После сборки образа контейнер запущен командой `docker run -p 5000:5000 rust-app`, а его работоспособность подтверждена через обращение по адресу `http://localhost:5000`.

## Контрольные вопросы

1. Что такое Dockerfile и для чего он используется?

Dockerfile — это текстовый файл с набором инструкций, описывающих процесс сборки Docker-образа. Он используется для:

- Автоматизации создания образов контейнеров.
- Определения зависимостей, переменных окружения и команд запуска приложения.
- Обеспечения воспроизводимости окружения на разных системах.
- Версионирования конфигурации приложения и его инфраструктуры.

2. Какие основные инструкции используются в Dockerfile?

Ключевые инструкции Dockerfile:

- FROM — задаёт базовый образ (например, FROM rust:1.75).
- WORKDIR — устанавливает рабочую директорию внутри контейнера.
- COPY / ADD — копирует файлы из хоста в контейнер.
- RUN — выполняет команды во время сборки (например, установка пакетов).
- EXPOSE — указывает порты, которые контейнер будет слушать.
- CMD / ENTRYPOINT — задаёт команду для запуска при старте контейнера.
- ENV — определяет переменные окружения.
- ARG — задаёт переменные, используемые во время сборки.

3. Как выполняется сборка образа Docker с использованием Dockerfile?

Сборка образа выполняется командой: **docker build -t <имя\_образа> <путь\_к\_Dockerfile>**

Этапы сборки:

- 1) Docker считывает инструкции из Dockerfile.
- 2) Каждая инструкция создаёт новый слой образа.

- 3) Слои кэшируются для ускорения последующих сборок.
- 4) Итоговый образ сохраняется в локальном хранилище Docker.

4. Как запустить контейнер из собранного образа?

Для запуска контейнера используется команда: **docker run [опции]**  
**<имя\_образа>**

5. Каковы преимущества использования Dockerfile для создания образов Docker?

- Автоматизация — исключение ручных настроек.
- Воспроизводимость — идентичное окружение на всех этапах.
- Версионность — изменения в Dockerfile можно отслеживать через Git.
- Легковесность — образы основаны на слоях, что экономит ресурсы.
- Изоляция — приложение работает в независимом окружении.