# Вкладка 6

## Виды соединений

### ssh

## Http

# бэкенд

### Терминал

#### Запуск контейнеров

docker build -t go-gin-app .

docker run -p 9090:8080 go-gin-app

docker system prune - удалить все контейнеры

docker container ls -a - показать ID всех контейнеров

docker container rm cc3f2ff51cab - Удалить конкретный контейнер по ID

#### Http запросы

curl -X GET "http://localhost:9200/"

#### Зайти внутреннюю консоль контейнера

docker ps - найти имя контейнера

docker exec -it strange\_dirac bash - зайти в терминал контейнера

docker exec -it strange\_dirac sh - если bash не установлен

???

docker exec -it elasticsearch curl -X GET "localhost:9200/\_search?q=rocky+mountain"

exit - выход из терминала контейнера

### Докер

Перед тем как работать с функциями докера, вроде надо запустить приложение

#### Ресурсы

[Docker: Accelerated Container Application Development](https://www.docker.com/)

документация

<https://docs.docker.com/>

#### Знания

##### Определения

Докер позволяет создавать и работать с образами и контейнерами

образ - файл хранящий саму функциональную часть нашей программы, А ТАКЖЕ хранящий функции всех библиотек, которые мы использовали у себя и многие другие настройки, чтобы наша программа работала.

контейнер - запущенная мини операционная система изолированная от нашей ОС и содержащую некоторую программу

##### Создание образа

На основе dockerfile можно создать docker образ: (docker build)

Пример создания образа для программы:

git clone <https://github.com/docker/welcome-to-docker>

cd welcome-to-docker

docker build -t welcome-to-docker .

\*\* После этого на докере появится образ, который можно запустить как отдельный контейнер\*\*

Эта программа welcome-to-docker уже содержит dockerfile, но для своих программ нужно создавать свой докерфайл

##### Volumes (тома)

##### Docker compose

Makefile - файл,содержащий как бы hotkeys для прописанных внутри него команд

#### Контейнеры и порты

**Основы IP и портов**:

* IP-адрес (например, 192.168.1.100) идентифицирует устройство в сети, а порт (например, 80, 9090) указывает конкретное приложение или сервис на этом устройстве.
* Адрес в сети записывается как IP:порт (например, 192.168.1.100:80), где порт определяет, к какому процессу направлен запрос.
* Хост (компьютер или сервер) слушает определённые порты (например, 9090) для приёма внешних запросов от устройств.

**Роль Docker и контейнеров**:

* Docker-контейнеры — это изолированные среды, которые запускаются на хосте и имеют собственное сетевое, файловое и процессное пространство.
* Каждый контейнер получает уникальный внутренний IP-адрес в сети Docker (например, 172.17.0.2), который изолирован от IP хоста и других контейнеров.

**Изоляция контейнеров от хоста**:

* Контейнеры изолированы от хоста с помощью Linux namespaces и cgroups, что означает, что они не имеют прямого доступа к сетевым интерфейсам, файлам или процессам хоста, если это не настроено explicitly.
* Сетевой localhost (127.0.0.1) внутри контейнера относится только к самому контейнеру, а не к хосту. Например, curl localhost:8080 внутри контейнера обращается к сервису внутри того же контейнера, а не к хосту.
* Файловая система, процессы и ресурсы контейнера отделены от хоста, если только не используются общие тома (-v) или другие механизмы.

**Как контейнеры общаются с хостом и внешним миром**:

* **Портовый маппинг (-p)**: Хост может "публиковать" порты контейнера, связывая их с своими портами. Например, -p 9090:8080 перенаправляет запросы с порта 9090 хоста на порт 8080 внутри контейнера через NAT (Network Address Translation).
* **Мостовые сети (Bridge Networks)**: По умолчанию Docker использует мост (bridge network), который создаёт виртуальную сеть для контейнеров. Хост и контейнеры общаются через этот мост, но без прямого доступа. Мост позволяет контейнерам иметь собственные IP-адреса и порты, а хост может перенаправлять трафик через него.
* **Host Network Mode**: Если использовать --network host, контейнер "обходит" изоляцию и использует сетевые интерфейсы хоста напрямую, что позволяет ему видеть localhost хоста и все его порты, но снижает безопасность.

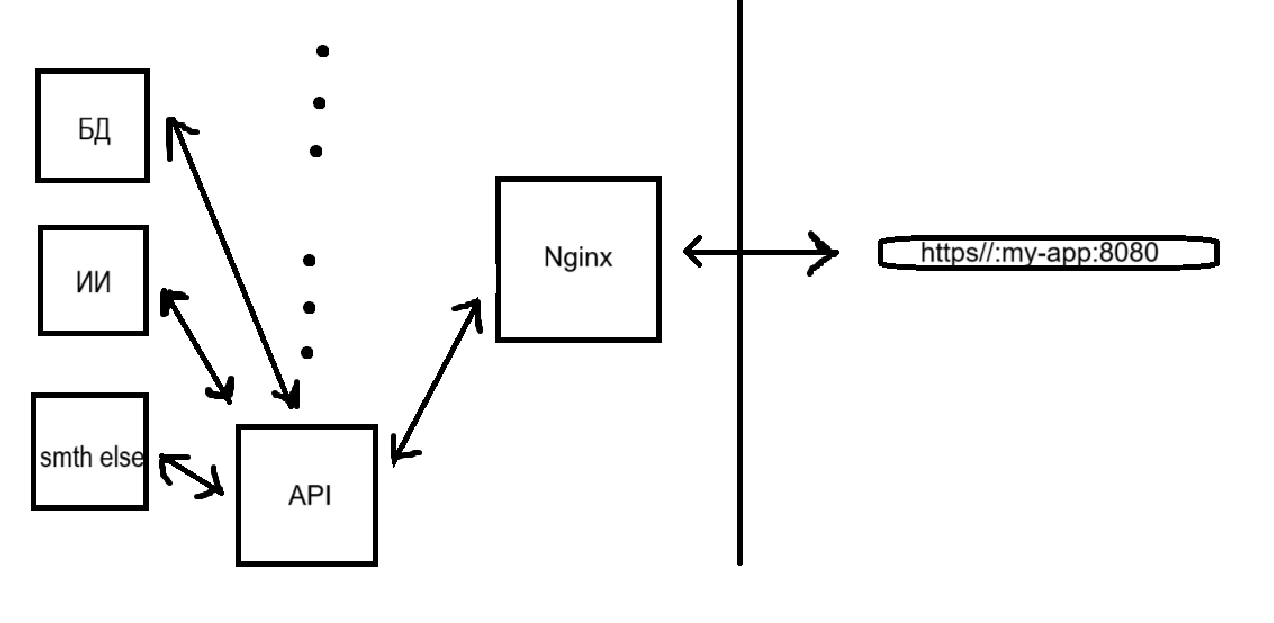
**Как контейнер может получать информацию от хоста**:

* **Через мосты (Bridge)**: В стандартной конфигурации (bridge network) контейнеры общаются с хостом через маппинг портов. Например, запрос от хоста на ХХХ.ХХ.ХХ:9090 перенаправляется в контейнер на его порт 8080.
* **Статический маршрут или DNS**: Контейнер может быть настроен так, чтобы знать IP хоста (например, 192.168.1.100) и обращаться к нему напрямую, если хост слушает определённые порты.
* **Общие ресурсы**: Хост может делиться данными с контейнером через монтируемые тома (-v /host/path:/container/path) или переменные окружения.
* **API или запросы**: Контейнер может отправлять HTTP-запросы или другие типы соединений на хост, если знает его IP и порт (например, curl 192.168.1.100:80).

**Настройка интерфейсов внутри контейнера**:

* Приложение внутри контейнера (например, Gin с router.Run()) может выбирать, на каких интерфейсах слушать:
  + :8080 или 0.0.0.0:8080 — принимает запросы от всех (хост, другие контейнеры, сеть).
  + localhost:8080 — принимает только от процессов внутри самого контейнера.
  + Конкретный IP — принимает только от устройств с этим IP (если такой IP настроен).

#### Архитектура RestAPI



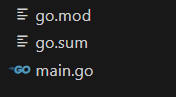
Проксирует http запросы от клиента nginx, однако

#### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### Начало

##### Запуск простого контейнера

###### 1) написать логику



main.go

package main

import (

"net/http"

"github.com/gin-gonic/gin"

)

type album struct {

ID string `json:"id"`

Title string `json:"title"`

Artist string `json:"artist"`

Price float64 `json:"price"`

}

var albums = []album{

{ID: "1", Title: "Blue Train", Artist: "John Coltrane", Price: 56.99},

{ID: "2", Title: "Jeru", Artist: "Gerry Mulligan", Price: 17.99},

{ID: "3", Title: "Sarah Vaughan and Clifford Brown", Artist: "Sarah Vaughan", Price: 39.99},

}

func getAlbums(c \*gin.Context) {

c.IndentedJSON(http.StatusOK, albums)

}

// postAlbums adds an album from JSON received in the request body.

func postAlbums(c \*gin.Context) {

var newAlbum album

// Call BindJSON to bind the received JSON to

// newAlbum.

if err := c.BindJSON(&newAlbum); err != nil {

return

}

// Add the new album to the slice.

albums = append(albums, newAlbum)

c.IndentedJSON(http.StatusCreated, newAlbum)

}

// getAlbumByID locates the album whose ID value matches the id

// parameter sent by the client, then returns that album as a response.

func getAlbumByID(c \*gin.Context) {

id := c.Param("id")

// Loop over the list of albums, looking for

// an album whose ID value matches the parameter.

for \_, a := range albums {

if a.ID == id {

c.IndentedJSON(http.StatusOK, a)

return

}

}

c.IndentedJSON(http.StatusNotFound, gin.H{"message": "album not found"})

}

func main() {

router := gin.Default()

router.GET("/albums", getAlbums)

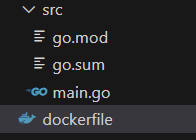
router.GET("/albums/:id", getAlbumByID)

router.POST("/albums", postAlbums)

router.Run("localhost:8080")

}

###### 2) написать dockerfile для логики



dockerfile:

FROM golang:1.24 AS builder

WORKDIR /app

# Копируем go.mod и go.sum из src

COPY src/go.mod src/go.sum ./

RUN go mod tidy

# Копируем весь src в рабочую директорию

COPY src/ .

# Сборка main.go

RUN GOOS=linux GOARCH=amd64 go build -o main main.go

# Финальный минимальный образ

FROM gcr.io/distroless/base

WORKDIR /root/

COPY --from=builder /app/main .

EXPOSE 8080

CMD ["./main"]

###### 3) Создание образа

в директории докерфайла прописать

docker build -t go-gin-app . - желтый это название образа

###### 4) Настройка портов

**Контейнер**

Внутри контейнера создаётся NAT, который слушает внутренний порт хоста (компьютера или сервера):

func main() {

router := gin.Default()

router.GET("/albums", getAlbums)

router.GET("/albums/:id", getAlbumByID)

router.POST("/albums", postAlbums)

router.Run("localhost:8080") // здесь мы заставляем слушать внутренний порт

}

Так контейнер слушает запросы приходящие на внутренний порт 8080 ТОЛЬКО локального хоста и больше НИКОГО  
Если сделать так 0.0.0.0:8080 - то тогда программа будет слушать ВСЕ запросы от всех устройств в сети.

**хост**

docker run -p 9090:8080 go-gin-app

Так мы говорим, чтобы хост слушал внешний порт 9090 и передавал его на внутренний 8080.

**dockerfile**

EXPOSE 8080

Это вообще не имеет отношение к портам, ни внутренним ни внешним

По сути ничего не делает, взуальная информация  
Но возможно в docker-compose может быть полезен

###### 5) запуск контейнера

docker run -p 9090:8080 go-gin-app

создание образа пользовательского контейнера

docker-compose.yaml автоматически запускает несколько таких контейнеров

##### Docker-compose up

###### 1) Создать файл инструкций по построению образов

На готовый dockerfile в туже директорию добавляем файл docker-compose.yaml

version: '3.8' # Указываем версию Compose (можно использовать 3.x)

services:

go-app: # Имя сервиса (можно изменить)

build:

context: . # Контекст сборки — текущая директория

dockerfile: Dockerfile # Указываем имя Dockerfile (если оно другое, измените)

ports:

- "9090:8080" # Маппинг портов: "внешний:внутренний" (хост:контейнер)

networks:

- app-network # Имя сети (будет создано автоматически, если не существует)

networks:

app-network: # Определение сети

driver: bridge # Используем мостовую сеть по умолчанию

###### 2) запустить через docker-compose

docker-compose up --build // со скачиванием или обновлением образов

docker-compose up // без обновления, а те которые сохранены

###### 3) Опустить все контейнеры

docker-compose down

#### Работа

##### Простейший API

package main

import (

"encoding/json"

"log"

"net/http"

)

type Message struct {

Text string `json:"text"`

}

func helloHandler(w http.ResponseWriter, r \*http.Request) {

msg := Message{Text: "Привет из бэкенда на Go!"}

w.Header().Set("Content-Type", "application/json")

json.NewEncoder(w).Encode(msg)

}

func main() {

http.HandleFunc("/api/hello", helloHandler)

log.Fatal(http.ListenAndServe(":8080", nil))

}

##### HTTP запросы

###### Принять запрос

package main

import (

"fmt"

"io"

"log"

"net/http"

)

func queryHandler(w http.ResponseWriter, r \*http.Request) {

fmt.Println("Request Method:", r.Method)

fmt.Println("Request URL:", r.URL.String())

body, \_ := io.ReadAll(r.Body)

fmt.Println("Request Body:", string(body))

}

func main() {

http.HandleFunc("/api/process\_query", queryHandler)

log.Fatal(http.ListenAndServe(":8080", nil))

}

###### Отправка ответа

<https://labex.io/tutorials/go-how-to-write-http-response-body-450894>

Для ответа на запрос используют w http.ResponseWriter

Так например, можно использовать функцию вывода текста, для того, чтобы отправить ответ на запрос. Эта функция вернёт ответ

fmt.Println(w , “hello”)

Когда ты вызываешь:

json.NewEncoder(w).Encode(response)

происходит следующее:

Создаётся JSON-энкодер, который пишет не в файл, не в память, а прямо в HTTP-ответ (в поток, который ассоциирован с клиентским соединением).

В Go:

Ты не создаёшь ответ целиком заранее, как в некоторых фреймворках.

Ты записываешь в поток w по ходу работы обработчика.

Всё, что ты запишешь в w — окажется в body HTTP-ответа, когда обработчик завершится.

Метод Encode превращает структуру response в JSON-строку (с автоматическим добавлением \n на конце) и пишет этот JSON прямо в w — в тело HTTP-ответа.

Когда функция queryHandler завершает работу, сервер автоматически отправляет всё, что было записано в w, в качестве тела ответа клиенту.

А заголовки (вроде Content-Type) тоже уже готовы к этому моменту

###### Совершение своего запроса

#### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Примеры

#### WebServer\_101

##### Gin

Запуск сервера на Go (gin framework):

mkdir web-service-gin

go mod init example/web-service-gin

//Создайте файл main.go

Код

package main

import (

"net/http"

"github.com/gin-gonic/gin"

)

type album struct {

ID string `json:"id"`

Title string `json:"title"`

Artist string `json:"artist"`

Price float64 `json:"price"`

}

var albums = []album{

{ID: "1", Title: "Blue Train", Artist: "John Coltrane", Price: 56.99},

{ID: "2", Title: "Jeru", Artist: "Gerry Mulligan", Price: 17.99},

{ID: "3", Title: "Sarah Vaughan and Clifford Brown", Artist: "Sarah Vaughan", Price: 39.99},

}

func getAlbums(c \*gin.Context) {

c.IndentedJSON(http.StatusOK, albums)

}

// postAlbums adds an album from JSON received in the request body.

func postAlbums(c \*gin.Context) {

var newAlbum album

// Call BindJSON to bind the received JSON to

// newAlbum.

if err := c.BindJSON(&newAlbum); err != nil {

return

}

// Add the new album to the slice.

albums = append(albums, newAlbum)

c.IndentedJSON(http.StatusCreated, newAlbum)

}

// getAlbumByID locates the album whose ID value matches the id

// parameter sent by the client, then returns that album as a response.

func getAlbumByID(c \*gin.Context) {

id := c.Param("id")

// Loop over the list of albums, looking for

// an album whose ID value matches the parameter.

for \_, a := range albums {

if a.ID == id {

c.IndentedJSON(http.StatusOK, a)

return

}

}

c.IndentedJSON(http.StatusNotFound, gin.H{"message": "album not found"})

}

func main() {

router := gin.Default()

router.GET("/albums", getAlbums)

router.GET("/albums/:id", getAlbumByID)

router.POST("/albums", postAlbums)

router.Run("localhost:8080")

}

##### Standard library

package main

import (

"fmt"

"net/http"

)

func hello(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintf(w, "Hello!")

}

func main() {

http.HandleFunc("/hello", hello)

fmt.Println("Server is running on :8090...")

http.ListenAndServe(":8090", nil)

}

Для запуска сервера:

go run .

И из другого терминала можно посылать запросы:

Invoke-WebRequest -Uri "http://localhost:8090/hello"

#### RestAPI\_101

структура проекта

project-root/

├── backend/

│ ├── go.mod

│ ├── main.go

│ └── dockerfile

├── frontend/

│ ├── dockerfile

│ ├── index.html

│ └── nginx.conf

│

└── docker-compose.yml

BACK

dockerfile

FROM golang:1.24-alpine

WORKDIR /app

COPY go.mod ./

COPY main.go ./

RUN go build -o main .

EXPOSE 8080

CMD ["./main"]

main.go

package main

import (

"encoding/json"

"log"

"net/http"

)

type Message struct {

Text string `json:"text"`

}

func helloHandler(w http.ResponseWriter, r \*http.Request) {

msg := Message{Text: "Привет из бэкенда на Go!"}

w.Header().Set("Content-Type", "application/json")

json.NewEncoder(w).Encode(msg)

}

func main() {

http.HandleFunc("/api/hello", helloHandler)

log.Fatal(http.ListenAndServe(":8080", nil))

}

FRONT

docker

FROM nginx:alpine

COPY index.html /usr/share/nginx/html/index.html

COPY nginx.conf /etc/nginx/conf.d/default.conf

index

<!DOCTYPE html>

<html lang="ru">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Frontend</title>

</head>

<body>

<h1>Фронтенд</h1>

<button onclick="fetchMessage()">Получить сообщение</button>

<p id="output"></p>

<script>

async function fetchMessage() {

try {

const response = await fetch('/api/hello');

const data = await response.json();

document.getElementById('output').innerText = data.text;

} catch (error) {

document.getElementById('output').innerText = "Ошибка запроса: " + error;

}

}

</script>

</body>

</html>

nginx

server {

listen 80;

location / {

root /usr/share/nginx/html;

index index.html;

try\_files $uri $uri/ =404;

}

location /api/ {

proxy\_pass http://backend:8080;

}

}

\_\_\_\_\_\_\_\_\_\_\_\_

docker-compose

version: '3'

services:

backend:

build: ./backend

ports:

- "8080:8080"

networks:

- appnet

frontend:

build: ./frontend

ports:

- "80:80"

networks:

- appnet

depends\_on:

- backend

environment:

- NGINX\_HOST=frontend

- NGINX\_PORT=80

networks:

appnet:

\*Здесь depends on backend не влияет на работу сервиса, это нужно чтобы контейнеры запустились в правильном порядке

Архитектура выглядит так:

1) пользователь заходит на URL (в данном случае localhost:80)

2) отправляется запрос на nginx

3) nginx отправляет файлы которые у него есть в /usr/share/nginx/html/index.html пользователю

4) пользователь нажимает кнопку

5) по нажатию кнопки js инициализирует запрос на /api/....

6) запрос приходит обратно на nginx по 80 порту, nginx настроен чтобы перенаправлять запросы с /api/... на внутренний порт 8080

nginx.config:

location /api/ {

proxy\_pass http://backend:8080;

}

7) а внутренний порт как раз прослушивает API

8) апи ловит запрос и создает json msg := Message{Text: "Привет из бэкенда на Go!"}

9) после чего апи формирует http ответ по пришедшему запросу.

w.Header().Set("Content-Type", "application/json")

json.NewEncoder(w).Encode(msg)

10) и этот ответ направляется туда откуда пришел: сначала в nginx , а nginx обратно пользователю

11) js пользователя получает сообщение в response, расшифровывает json

12) расшифрованные данными он делает вот это:

document.getElementById('output').innerText = data.text;

я полагаю эта особоая структура в js, она пустая до тех пор пока не сделать .innerText

<p id="output"></p>

# Фронтенд

### JavaScript React

#### Инфо

Чтобы использовать react приложение в контейнере нужно собрать react приложение в продакшен билд: npm run build. Иначе плохо

#### Начало

Проверка библиотек для минимального фронта

1. node -v
2. npm -v # или, если используете Yarn: yarn -v

Создание проекта

1. зайти в нужную директорию
2. npx create-react-app front

npx create-react-app –template typescript

1. cd front

npm start

Для начала разработки приложения

#### Работа

# фронт-бэк взаимодействие

## Go - JS React

### Знание

#### Статические файлы react

React — это клиентское приложение. Когда ты запускаешь команду npm run build в React, создается статический билд твоего приложения (по пути /front/build и они копируются в исходный контейнер /app/build и в папку финального контейнера /usr/share/nginx/html) .

Все исходные файлы React (JS, HTML, CSS, изображения и т.д.) компилируются в набор статических файлов, которые браузер может загрузить.

Статические файлы: это такие файлы, которые не меняются при каждом запросе, например, HTML, CSS, JS. Они готовы к отдаче и не требуют обработки на сервере каждый раз, когда их запрашивают.

#### Коммуникация по интернету

В принципе любая коммуникация по внешней сети между разными хостами осуществляется с помощью http запросов

Http запрос устроен таким образом, чтобы однажды получив его ты мог ответить и ответ придёт на адрес с которого он был отправлен.

По-мимо самого механизма запроса в http есть поле для данных, которые ты хочешь передать. Обычно данные передают в виде JSON - сериализованных данных.

#### Взаимодействие react с внешним миром

Взаимодействие происходит через веб-сервер Nginx в который мы передаём статические данные нашего приложения react, а он сам отправляет на нужный порт.

Когда мы забилдили наше реакт приложение и собираем образ фронтенда в dockerfile.front мы создаём контейнеры

1. Первый контейнер с nodeJS для того, чтобы забилдить наш frontend
2. Второй контейнер с Nginx в который мы просто копируем статические данные из первого контейнера в папку /usr/share/nginx/html (это специальная папка nginx из которой nginx отправляет данные на выходной порт пользователю в браузер)

И первый контейнер удаляется и не используется, а второй работает

#### Веб сервер

Сервер — это устройство или программа, которая предоставляет ресурсы, данные или услуги другим устройствам или программам (клиентам) через сеть. Это может быть любой тип сервера, например, файловый сервер, почтовый сервер, баз данных и т. д.

Веб-сервер — это специфический тип сервера, который обрабатывает HTTP-запросы от клиентов (чаще всего от веб-браузеров) и отправляет им веб-страницы. Веб-серверы могут обрабатывать запросы на веб-страницы, изображения, скрипты, стили и другие ресурсы, которые находятся на веб-сайте.

Различия:

1. **Назначение:**
   * **Сервер** — общее понятие, которое может касаться различных типов серверов, например, файловых, почтовых, баз данных и других.
   * **Веб-сервер** — специализирован для обработки HTTP-запросов и предоставления данных через веб-протокол.
2. **Тип запросов:**
   * **Сервер** может обрабатывать запросы различных типов (например, файловые запросы, запросы к базе данных и т. д.).
   * **Веб-сервер** обрабатывает только HTTP-запросы.
3. **Протоколы:**
   * **Сервер** может использовать разные протоколы в зависимости от его назначения (например, FTP, SMTP, POP3, и т. д.).
   * **Веб-сервер** использует HTTP или HTTPS.

#### Как устроен Nginx

#### Использование Nginx

**отправка статических данных по внешнему порту (проксирование)**

Если у тебя есть программа на твоем локальном хосте (компьютере или сервере), то другие люди не будут иметь к ней доступ.

Но если ты установил и настроил у себя nginx и чтобы он слушал внешние порты, то nginx может принимать запросы пользователей из сети, проксировать их на твой компьютер, чтобы твоя программа выполнить что то для них.

И потом ты снова может результат программа отправить на твой локальный nginx, то он может обратно отправить пользователям результат.

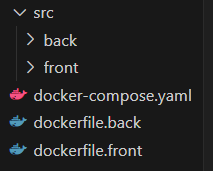
**редиректоривание**

если пользователь переходит по ссылке, то nginx автоматически перенаправляет пользователя на другой

### Начало

#### Создание цельного фронтенд бекенд докер проекта

Для поднятия докер проектa с фронтом и бекендом - используют архитектуру:



Создают 2 независимых образа для бека и фронта.

И поднимают вместе через docker-compose.

#### Создание dockerfile для фронта

# Этап сборки

FROM node:22 AS builder

WORKDIR /app

COPY src/front/package\*.json ./

RUN npm install

COPY src/front/ ./

RUN npm run build

# Финальный образ

FROM nginx:alpine

COPY --from=builder /app/build /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

Nginx — это веб-сервер, который будет обслуживать статические файлы (например, скомпилированные файлы React-приложения).

Alpine — минималистичный Linux, что позволяет сделать твой образ легче и быстрее.

COPY --from=builder /app/build /usr/share/nginx/html

Эта строка копирует содержимое каталога build, созданного на предыдущем этапе сборки, в финальный контейнер.

--from=builder означает, что ты берёшь файлы из предыдущего шага сборки, где был использован образ с Node.js (этап, который ты указал как builder).

/app/build — это путь в контейнере сборки (там, где находятся статические файлы после выполнения команды npm run build).

/usr/share/nginx/html — это стандартное место, куда Nginx помещает свои HTML-файлы для раздачи. То есть, React-приложение будет доступно через этот путь.

MD ["nginx", "-g", "daemon off;"]

CMD задаёт команду, которая будет выполнена при запуске контейнера.

В данном случае ты запускаешь Nginx с флагом -g "daemon off;".

-g "daemon off;" — этот флаг заставляет Nginx работать в foreground, т.е. в режиме без демона. Это нужно, потому что в контейнере Docker процесс должен быть основным, иначе контейнер завершится.

Без этой команды Nginx бы пытался работать как фоновый процесс, и контейнер сразу завершился бы.

### Работа

#### RestAPI коммуникация

##### Front call func ← back

Здесь на клиентской части js доходит до момента, когда нужно получить что то от бекенда или обработать бекендом

async function fetchMessage(query) {

const response = await fetch('/api/process\_query', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ query: query })

});

const data = await response.json();

setValue(data);

}

fetch() — это функция из браузера, которая отправляет HTTP-запрос.

заранее известен внешний порт, тк мы на нём находимся http://localhost:3000 и таким образом формируется адрес URI для запроса:

http://localhost:3000/api/process\_query

Если порт не указан, то по умолчанию:

http:// → порт 80

https:// → порт 443

# Деплой веб приложений

### Push своих образов на docker hub

Для запуска своего приложения на удаленном сервере можно загрузить свои контейнеры в docker hub и docker-compose.yaml и таким образом их запускать

Обычно разные сервисы (бекенд, nginx) хранят в отдельных репозиториях для удобства.

#### Загрузка образов

1. бекенд

docker images

docker login

docker tag fuzzysearch\_alfa-backend heavymetal0/fuzzysearch\_alfa-backend:latest

docker push heavymetal0/fuzzysearch\_alfa-backend:latest

1. фронт

docker tag fuzzysearch\_alfa-nginx heavymetal0/fuzzysearch\_alfa-nginx:latest

docker push heavymetal0/fuzzysearch\_alfa-nginx:latest

#### Создание yaml для запуска контейнеров

Пример yaml

version: '3.8'

services:

backend:

image: heavymetal0/fuzzysearch\_alfa-backend:latest

restart: unless-stopped

ports:

- "8000:8000" # Пример: бекенд на порту 8000

environment:

- ENV\_VAR=value # Если нужны переменные окружения

networks:

- fuzzysearch\_net

nginx:

image: heavymetal0/fuzzysearch\_alfa-nginx:latest

restart: unless-stopped

ports:

- "80:80" # Nginx слушает 80 порт

- "443:443" # Если нужен HTTPS

depends\_on:

- backend

networks:

- fuzzysearch\_net

networks:

fuzzysearch\_net:

driver: bridge

#### Итого

Теперь когда мы загрузили свои образы в докерхаб  
любой человек, который создаст у себя файл docker-compose.yaml и запустит docker-compose up автоматически подгрузит с серверов нужные библиотеки и запустит контейнеры локально

### Попытка 1

1. Настройка сервера

Получил данные:

admin lainnetwork

ssh-key-1744656125577

сгенерировал ssh ключ через cmd ssh-keygen

и вставил его в яндекс

1. Запустил сервер

Подключение к терминалу удаленного сервера:

ssh -i C:\Users\Mikhail\.ssh\id\_ed25519 lainnetwork@84.252.132.166