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.

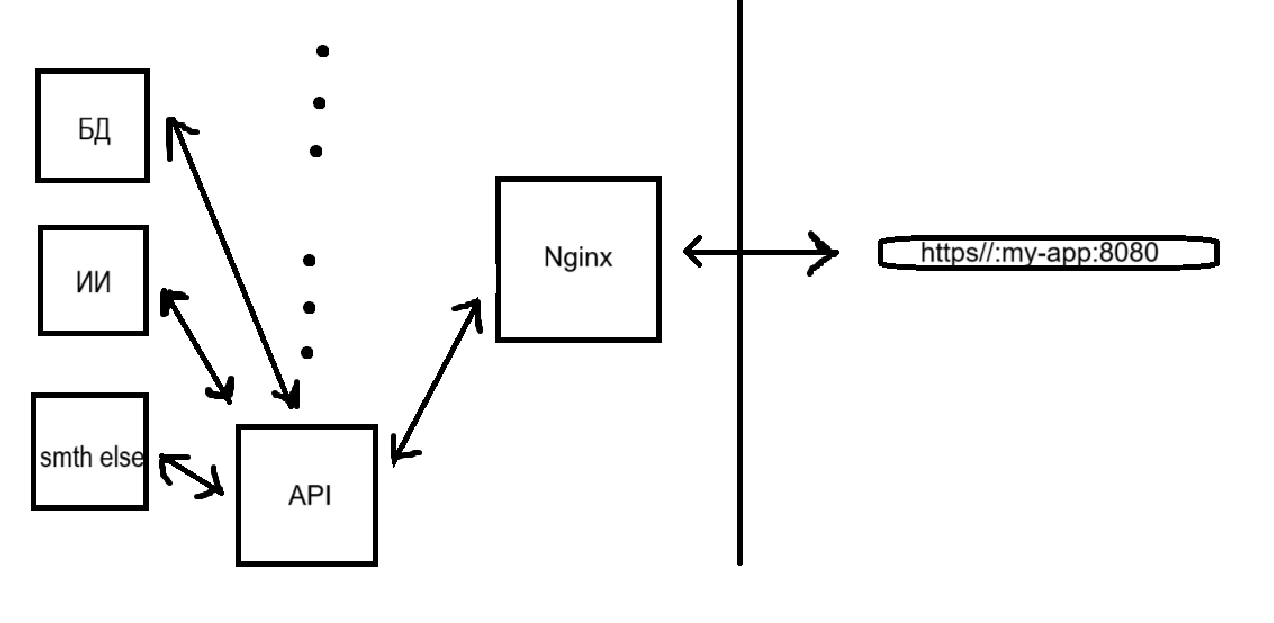
**Как контейнер может получать информацию от хоста**:

* **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



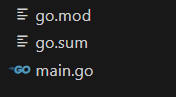
Проксирует http запросы от клиента nginx, однако

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

## 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)

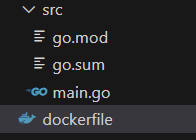
router.GET("/albums/:id", getAlbumByID)

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.

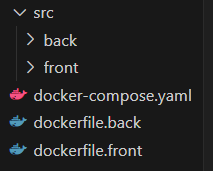
**редиректоривание**

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

ОсновАрхитВебПрил

Основные Архитектуры веб сервисов

# Статический сайт

### Классический статический сайт.

1. Статический сайт- это сайт который состоит и файлов *.html* заранее забилденных и размещенных на сервере. При запросе сервер просто отдаёт заранее приготовленные файлы пользователю. Каждый html отвечает за одну страницу.
2. механизм работы статического сайта:

* На сервере запускают базовый простой легковесный HTTP-сервер (например python http.server). В нём реализована логика какие файлы возвращать в ответ на какие запросы.

Но можно и реализовать самостоятельно через fastAPI тоже самое.

* Перемещают html файлы в директории в соответствии с логикой http сервера.

1. Создание и запуск.

* Создание папки содержащей проект
* помещение html документов и скриптов для них
* из директории этой папки запустить http сервер “python -m http.server 8000”

### SPA

#### Описание

1. Single Page Application (SPA) - тоже статический сайт, но который состоит только из одного html и множества JavaScript скриптов, которые изменяют html. Чтобы не загружать несколько html можно сделать скрипты, чтобы создать анимации или иллюзию множества страниц.

|  | **Статический сайт** | **SPA (Single Page Application)** |
| --- | --- | --- |
| **Количество HTML-страниц** | Много (index.html, about.html, contact.html, …) | Одна (index.html) |
| **Переходы между страницами** | Браузер перезагружает страницу | Всё происходит **без перезагрузки** (через JS) |
| **Навигация** | Ссылки ведут на отдельные файлы | Ссылки просто **меняют состояние React/Vue** |
| **Контент** | Статичный, заранее написанный | Генерируется динамически через JS |
| **Пример** | Сайт-визитка, документация | Gmail, Trello, Google Docs |

1. Только один html.

Главной особенностью SPA является, что у него только один html и множество js скриптов, работающих с ним. Это позволяет динамически изменять содержимое страницы без её перезагрузки, обеспечивая плавный и быстрый пользовательский опыт. Поэтому для высокой скорости работы сайта оборачивают несколько страниц в одну (один html).

1. Проблема с запросами на ресурсы SPA.

Именно из-за специфики работы SPA — когда адресная строка меняется (для удобства пользователей), но фактически используется один index.html — HTTP-сервер нужно настраивать особым образом.

Если пользователю скинули ссылку на конкретную часть сайта ( /about ), то браузер, переходя по ссылке SPA сайта с путём вроде /about или /profile, отправляет запрос к соответствующему ресурсу, которого физически не существует в папке с сайтом (там только index.html и статические файлы).

Чтобы всё работало корректно, необходимо:

* настроить клиентский JavaScript (роутер), чтобы он **обрабатывал маршруты на стороне клиента** и отрисовывал нужный фрагмент интерфейса;
* настроить сервер так, чтобы **на любой несуществующий маршрут он возвращал index.html**, позволяя SPA-приложению загрузиться и самостоятельно разобраться, что нужно показать.

1. Правильный http сервер для SPA.

Так как SPA билдится в обычный html файл, то по итогу это обычный *статический сайт* и для него также нужен сервер прослушивающий запросы и возвращающий правильные файлы, но

* Обычные статические серверы вроде python http.server не подойдут из-за особой механики работы SPA (нужно чтобы при любом запросе возвращался index.html).
* И используется вместо этого npm server.

#### Запуск

1. Механизм работы SPA сайта:

* Во время разработки происходят множественные модификации react файлов, которые в конечном итоге должны скомпилироваться в один html.

Для тестирования ещё не скомпилированных в html файловесть встроенный HTTP сервер, который умеет работать с такими сырыми файлами: *npm start*.

* Когда файлы готовы реакт проект билдится в одну папку dist, где находится один *html* и скрипты для него, которые срабатывают во время анимаций, изменения вида элементов, появления новых страниц и другого рода интерактивностей. Это уже готовая папка к работе и переносится на сервер.
* На сервере запускается особый, подходящий для SPA, http сервер слушающий запросы.

Обычно это npm serve.

Можно и fastAPI настроить и что угодно, но легче npm serve.

1. Создание и запуск:

* Создание react проекта (npm create vite@latest – –template react)
* билд проекта в папку dist:

npm run build

* Запуск сервера (serve -s) в папке dist

#### Доп

1. Роутинг.

Обеспечение корректной навигации в SPA, когда приложение использует разные адресные строки, но при этом работает только с одним HTML-файлом (*index.html*).

Роутинг - механизм навигации по SPA сайту: при нажатии на кнопку перехода на другую “страницу” сайта (/about) SPA не делает никаких запросов, а

* рендерит определенный фрагмент index.html.
* изменяет адресную строку.

С роутингом в SPA js скрипты изменяют в браузере адресную строку и отображаемый контент, но по факту являясь только одной страницей.

Это нужно, чтобы пользователь мог ориентироваться в приложении, делиться ссылками на конкретные "страницы" и браузер корректно отображал историю навигации (например, кнопки "назад" и "вперед").

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

Прямо в самом приложении реакт:

// App.jsx

import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Home from './pages/Home';

import About from './pages/About';

import NotFound from './pages/NotFound';

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

<Route path="\*" element={<NotFound />} />

</Routes>

</BrowserRouter>

);

}

export default App;

В React SPA нужно использовать <Link> из react-router-dom, чтобы избежать перезагрузки:

jsx

Копировать

Редактировать

import { Link } from 'react-router-dom';

<Link to="/about">О нас</Link>

Нельзя использовать обычный <a href="/about"> — он приведёт к полной перезагрузке страницы.

#### Стек

1. serve

<https://www.npmjs.com/package/serve/v/11.3.0>

npm i serve@11.3.0

npm install -g serve // установить глобально

npm утилита, которая запускает сервер и прослушивает порт.

Можно запустить в разных режимах в зависимости от того какой сайт:

* для статических файлов можно запустить как

serve public/

* Для SPA:

serve -s build/

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

# Хостинги

1. Vercel

<https://vercel.com/home>

<https://test-app-six-rouge.vercel.app/>

1. Render

<https://render.com/>

tools для веб архитектур

# Elastic search

сайт

[Elasticsearch: The Official Distributed Search & Analytics Engine | Elastic](https://www.elastic.co/elasticsearch)

облачные услуги (моя личная среда)

<https://37786ab484b645808192ed124d462b99.us-central1.gcp.cloud.es.io/app/elasticsearch/overview>

## Инфо

Elasticsearch — это документо-ориентированная поисковая система с API-интерфейсом, работающая поверх движка Apache Lucene.

В отличие от реляционных баз (типа MySQL или PostgreSQL), здесь всё строится вокруг документов в формате JSON, а не таблиц.

### Как это работает

#### Добавление документов

## **1. Получение и распределение запроса**

### **1.1. REST API и Coordinating Node**

* **REST API:** Клиент отправляет JSON-документ через HTTP-запрос (например, с помощью curl или другого инструмента).
* **Coordinating Node:** Узел, принимающий запрос, выполняет роль маршрутизатора. Он:  
  + Принимает запрос.
  + Выясняет, на какой шард (часть индекса) нужно сохранить документ согласно стратегии распределения.
  + Проводит предварительную валидацию и, при необходимости, применяет фильтры безопасности или трансформации запроса.

*Ссылки:*

* [Elasticsearch REST API](https://www.elastic.co/guide/en/elasticsearch/reference/current/rest-apis.html)

## **2. Определение места хранения: индекс, шард и реплика**

### **2.1. Индекс и маппинг**

* **Индекс:** Логическая структура, похожая на таблицу в реляционной базе данных. Каждому индексу можно задать маппинг, который определяет типы и настройки полей (например, text для полнотекстового анализа, keyword для точных значений, date, float и т.д.).
* **Маппинг:** Определяет, как будут анализироваться и храниться поля документа. Например, поле name типа text будет обрабатываться анализатором для разбиения на токены.

*Ссылки:*

* [Mapping в Elasticsearch](https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html)

### **2.2. Шард и репликация**

* **Шард:** Каждый индекс разбивается на один или несколько шард, чтобы распределять нагрузку и обеспечить параллельную обработку запросов.
* **Реплика:** Копии шард, которые используются для отказоустойчивости и повышения производительности чтения.

## **3. Обработка документа: Анализ и индексирование**

### **3.1. Анализ документа**

* **Анализатор (Analyzer):** Для полей типа text Elasticsearch применяет анализатор, который состоит из:  
  + **Char Filters:** Предварительная обработка текста (удаление HTML-тегов, нормализация символов).
  + **Tokenizer:** Разбивает строку на токены (например, слово «Elasticsearch Tutorial» превращается в токены ["elasticsearch", "tutorial"]).
  + **Token Filters:** Применяются фильтры для нормализации (приведение к нижнему регистру, удаление стоп-слов, стемминг).

В результате исходный JSON-документ преобразуется:  
 Исходное поле:

json

КопироватьРедактировать

"name": "Elasticsearch Tutorial"

Преобразуется в список токенов:

json

КопироватьРедактировать

"tokens": ["elasticsearch", "tutorial"]

*Ссылки:*

* [Анализаторы в Elasticsearch](https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis.html)

### **3.2. Построение инвертированного индекса (Inverted Index)**

* **Inverted Index:** После анализа для каждого токена создается структура, связывающая термин с идентификаторами документов (и позициями токенов внутри документа).

Пример (псевдо-JSON представление):  
  
 json  
КопироватьРедактировать  
{

"inverted\_index": {

"elasticsearch": [{ "doc\_id": 1, "positions": [1] }],

"tutorial": [{ "doc\_id": 1, "positions": [2] }]

}

}

* Это позволяет при поиске быстро определить, в каких документах содержится запрошенный термин.

## **4. Физическое хранение: Lucene и сегменты**

### **4.1. Роль Apache Lucene**

* **Apache Lucene:** Ядро, отвечающее за построение и хранение полнотекстовых индексов. Elasticsearch использует Lucene для:  
  + Хранения инвертированного индекса.
  + Организации данных в сегменты.
  + Выполнения высокопроизводительного поиска и агрегаций.

*Ссылки:*

* [Apache Lucene](https://lucene.apache.org/core/)

### **4.2. Сегменты и их структура**

* **Сегменты:** Каждый шард в Elasticsearch представлен набором сегментов Lucene. Сегменты – это неизменяемые структуры, содержащие:  
  + **Инвертированный индекс:** Хранит токены и списки документов (postings lists).
  + **Stored Fields:** Хранят исходные значения полей, необходимые для возврата данных клиенту.
  + **Doc Values:** Для сортировки и агрегаций.
* Процесс записи:  
  + Документ сначала поступает в оперативную память и транзакционный журнал (transaction log).
  + На основе буферов формируются маленькие сегменты.
  + По мере накопления происходит **слияние сегментов**, когда несколько маленьких сегментов объединяются в один более крупный для оптимизации поиска.

*Ссылки:*

* [Lucene Index File Formats](https://lucene.apache.org/core/8_11_0/core/org/apache/lucene/codecs/lucene50/package-summary.html) – информация о форматах файлов
* [Segment Merging](https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules.html#_index_merging)

### **4.3. Запись на диск**

* Сегменты сохраняются как бинарные файлы на жестком диске.
* **Memory-Mapped Files (mmap):** Часто используются для ускорения доступа к сегментам, позволяя операционной системе эффективно кэшировать данные.

## **5. Итоговый процесс от начала до конца**

**Получение запроса:** Клиент отправляет JSON-документ через REST API.  
 Пример запроса:  
  
 bash  
КопироватьРедактировать  
curl -X POST "localhost:9200/products/\_doc/1" -H 'Content-Type: application/json' -d '{

"name": "Elasticsearch Tutorial",

"price": 999,

"category": "smartphones",

"created\_at": "2024-04-15"

}'

1. **Распределение запроса:** Coordinating Node определяет, к какому шард следует отправить документ согласно стратегиям распределения и маппингу индекса.
2. **Анализ и обработка:**
   * Поля типа text обрабатываются анализаторами.
   * Исходный текст превращается в набор токенов.
   * Формируется инвертированный индекс, где для каждого токена создаётся запись с идентификаторами и позициями документа.
3. **Формирование сегментов:**
   * Обработанные данные попадают в оперативную память и транзакционный лог.
   * Из буферов формируются небольшие сегменты Lucene.
4. **Сохранение данных на диск:**
   * Сегменты записываются как бинарные файлы на Data Node (дисковое хранилище).
   * Происходит слияние сегментов для повышения эффективности поиска.
5. **Дальнейшая работа:**
   * Данные доступны для поиска с помощью инвертированного индекса.
   * Запросы на полнотекстовый поиск обращаются к postings list, что позволяет быстро находить документы.

## **Полезные источники для углубления знаний**

* [Elasticsearch: The Definitive Guide](https://www.elastic.co/guide/en/elasticsearch/guide/current/index.html)
* [Understanding Lucene's Inverted Index](https://www.elastic.co/blog/understanding-lucene-inverted-index)
* [Elasticsearch Analysis and Analyzers](https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis.html)

### Индекс Lucene

Так называется структура данных которая создаётся при добавлении документов в elastic search Lucene

Сегмент

**Внутри сегментов Lucene содержатся:**

* **Инвертированный индекс:** Для каждого текстового поля (например, «name») анализатор разбивает исходный текст на токены (слова).  
   Для каждого токена создаётся запись (postings list), которая указывает, в каких документах и на каких позициях он встречается.  
   Это позволяет проводить быстрый полнотекстовый поиск.
* **Doc Values:** Это колонно-ориентированные структуры, предназначенные для быстрого доступа при сортировке и агрегациях.  
   Они сохраняют значения полей в оптимизированном для чтения формате.
* **Stored Fields:** Некоторые поля могут сохраняться отдельно (как есть) для возврата данных клиенту при поиске, без необходимости воссоздавать их из инвертированного индекса.

### Инвертированные индексы

### Сравнение с базой данных

Однако, Elasticsearch не является реляционной БД. Он оптимизирован для поиска и анализа данных с фокусом на скорость обработки больших объёмов информации, а не на строгую нормализацию и поддержание связей между таблицами.

## Терминал

На последних версиях нужно:

1) делать запрос только через защищенное соединение https

2) И с аргументом -k (позволяет согласится на небезопасный запрос)

Самые основные команды

<https://www.elastic.co/guide/en/elasticsearch/guide/current/_talking_to_elasticsearch.html>

curl -X GET "https://localhost:9200/" -u elastic:password -k - Базовый запрос

### Работа с эластик сёрчем

Создание индекса

curl -X PUT "<https://localhost:9200/products>" -u elastic:password -k

Создание индекса с маппингом

curl -X PUT "[https://](https://localhost:9200/products)localhost:9200/products" -u elastic:password -k -H 'Content-Type: application/json' -d '

{

"mappings": {

"properties": {

"name": {

"type": "text"

},

"price": {

"type": "float"

},

"category": {

"type": "keyword"

},

"created\_at": {

"type": "date"

}

}

}

}'

Добавление маппинга к существующему индексу

curl -X PUT "https://localhost:9200/products/\_mapping" -u elastic:password -k -H 'Content-Type: application/json' -d '

{

"properties": {

"name": {

"type": "text"

},

"price": {

"type": "float"

},

"category": {

"type": "keyword"

},

"created\_at": {

"type": "date"

}

}

}'

Добавление документа в индекс

curl -X POST "https://localhost:9200/products/\_doc/1" -u elastic:password -k -H 'Content-Type: application/json' -d '

{

"name": "iPhone 14",

"price": 999,

"category": "smartphones",

"created\_at": "2024-04-15"

}'

Поиск по индексу документа

curl -X GET "<https://localhost:9200/products/_doc/1>" -u elastic:password -k

Поиск по значению поля документа

curl -X GET "https://localhost:9200/products/\_search" \

-u elastic:password \

-k \

-H 'Content-Type: application/json' \

-d '

{

"query": {

"match": {

"name": "iPhone 14"

}

}

}'

Поиск по значению поля:

curl -X GET "https://localhost:9200/products/\_search" \

-u elastic:password \

-k \

-H 'Content-Type: application/json' \

-d '

{

"query": {

"match": {

"name": "iPhone 14"

}

}

}'

Фильтр:

curl -X GET "[https://](https://localhost:9200/products/_doc/1)localhost:9200/products/\_search" -u elastic:password -k -H 'Content-Type: application/json' -d '

{

"query": {

"range": {

"price": {

"gte": 500,

"lte": 1000

}

}

}

}'

Запрос по значению поля + фильт:

curl -X GET "[https://](https://localhost:9200/products/_doc/1)localhost:9200/products/\_search" \

-u elastic:password \

-k \

-H 'Content-Type: application/json' \

-d '

{

"query": {

"bool": {

"must": [

{

"match": {

"name": "iPhone 14"

}

},

{

"range": {

"price": {

"gte": 500,

"lte": 1000

}

}

}

]

}

}

}'

удаление документа

curl -X DELETE "[https://](https://localhost:9200/products/_doc/1)localhost:9200/products/\_doc/1" -u elastic:password -k

удаление индекса

curl -X DELETE "[https:/](https://localhost:9200/products/_doc/1)/localhost:9200/products" -u elastic:password -k

### Поиск по тексту

Полнотекстовый поиск (можно искать по отдельным словам):

curl -X GET "https://localhost:9200/products/\_search" \

-u elastic:password \

-k \

-H 'Content-Type: application/json' \

-d '

{

"query": {

"match": {

"name": "iPhone"

}'} }

Нечёткий поиск (с ошибками):

curl -X GET "https://localhost:9200/products/\_search" \

-u elastic:password \

-k \

-H 'Content-Type: application/json' \

-d '

{

"query": {

"match": {

"name": {

"query": "iPhane",

"fuzziness": "AUTO"

}

}

}

}'

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

## Архитектуры с Elastic

Для обеспечения надежности транзакций обычно используют комбинированные архитектуры с Elastic search и реляционной БД

Примеры архитектур

1. **Система-источник (Source of Truth) плюс поисковый движок** В этой схеме база данных является основным хранилищем данных. Все операции записи, обновления и удаления происходят в БД, а затем данные реплицируются или индексируются в Elasticsearch.  
   * **Репликация данных:** Используется механизм синхронизации или обмен сообщений (например, через очереди, Change Data Capture), чтобы изменения в БД сразу отражались в индексе Elasticsearch.
   * **Отложенная синхронизация:** При некоторых сценариях допускается задержка между обновлением в БД и обновлением в Elasticsearch, если скорость поиска не критична.
2. **Дублирование функций для повышения масштабируемости** Некоторые системы используют Elasticsearch даже для операций аналитики или отчетности, оставляя транзакционные операции в БД. Это позволяет разгрузить основное хранилище данных и улучшить масштабируемость системы.
3. **Кеширование сложных поисковых запросов** Elasticsearch может выступать как высокопроизводительный слой для поиска, избавляя базу данных от избыточной нагрузки на выполнение сложных запросов, особенно в сценариях, когда требуется быстрый отклик для пользователей.