

|  |  |
| --- | --- |
| https://lh7-us.googleusercontent.com/QuBaagabzZYLr7U3ZbYOtMusd5FbWWxIvMUTN8jrLHzSLMg534z9gXRTIG1Us4i_lOwmWlaBxKedNt-SQ26dm4WmyqwjGDmEO6z8GE3QrZosqvHM88J2EFeVf1u0GzyCZQlhWmp1Zeo85tKo4LJVXQ | МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)» |

Институт № 3 «Системы управления, информатика и электроэнергетика»

Кафедра 304 «Вычислительные машины, системы и сети»

Отчёт по лабораторной работе №24  
по учебной дисциплине «Web-технологии»

на тему «Развертывание приложения при помощи Docker»

Выполнил  
студент группы М3О-119БВ-24

Нарзиев А.Т.

Принял

Ассистент каф.304 Борисов А.И.

Москва  
2024

# Содержание

[Содержание 2](#_Toc199477405)

[Задание 4](#_Toc199477406)

[Задание №24. Развертывание приложения при помощи Docker 4](#_Toc199477407)

[Решение 5](#_Toc199477408)

[1. Создание Docker-образа веб-приложения 5](#_Toc199477409)

[1.1. Фронтенд: 5](#_Toc199477410)

[1.2. Бэкенд: 5](#_Toc199477411)

[2. Настройка Dockerfile 7](#_Toc199477412)

[2.1. Теория: 7](#_Toc199477414)

[2.2. Действия: 7](#_Toc199477415)

[3. Запуск контейнера и проверка 7](#_Toc199477416)

[4. Развертывание БД в контейнере 9](#_Toc199477417)

[5.1. Теория 9](#_Toc199477421)

[5.2. Действия: 9](#_Toc199477422)

[5. Docker Compose для управления сервисами 11](#_Toc199477423)

[5.3. Теория: 11](#_Toc199477424)

[5.4. Действия: 11](#_Toc199477425)

[6. Масштабирование веб-приложения 14](#_Toc199477426)

[6.1. Теория: 14](#_Toc199477428)

[6.2. Действия: 16](#_Toc199477432)

[7. Автоматическое обновление образов 18](#_Toc199477433)

[7.1. Теория: 18](#_Toc199477435)

[7.2. Интеграция DockerHub 20](#_Toc199477436)

[7.3. Интеграция Watchtower 22](#_Toc199477437)

[7.4. Проверка работы Watchtower: 23](#_Toc199477438)

[8. Развертывание в Docker Swarm 25](#_Toc199477439)

[8.1. Теория: 25](#_Toc199477441)

[8.2. Действия: 27](#_Toc199477442)

[9. Мониторинг с Prometheus и Grafana 30](#_Toc199477443)

[9.1. Создана следующая структура каталогов: 30](#_Toc199477445)

[9.2. Настройка Prometheus 30](#_Toc199477446)

[9.3. Настройка Loki 31](#_Toc199477447)

[9.4. Настройка Grafana 31](#_Toc199477448)

[9.5. Обновление docker-compose.yml 32](#_Toc199477449)

[9.6. Настройка экспорта метрик в приложении 34](#_Toc199477450)

[10. Финальное тестирование с одновременным подключением клиентов 35](#_Toc199477451)

[Заключение 36](#_Toc199477452)

[Вывод 37](#_Toc199477453)

# Задание

# Задание №24. Развертывание приложения при помощи Docker

1. Создать Docker образ веб-приложения. Образ должен содержать все необходимые зависимости и файлы приложения.
2. Настроить Dockerfile для сборки образа веб-приложения, включая установку зависимостей, копирование файлов и запуск приложения.
3. Запустить контейнер с созданным Docker образом веб-приложения и проверить его работоспособность через браузер.
4. Развернуть базу данных (например, MySQL) в Docker контейнере и настроить веб-приложение на подключение к этой базе данных.
5. Создать Docker-compose файл для запуска веб-приложения и базы данных в отдельных контейнерах, а также настройки их взаимодействия.
6. Добавить возможность масштабирования веб-приложения путем запуска нескольких инстансов контейнера с помощью Docker-compose.
7. Настроить автоматичесное обновление Docker образа веб-приложения при обновлении кода из репозитория (например, GitHub).
8. Использовать Docker Swarm для развертывания кластера из нескольких узлов и размещения веб-приложения на этом кластере.
9. Настроить мониторинг и логирование контейнеров с помощью инструментов, таких как Prometheus и Grafana.
10. Запустить в Docker приложении сервер из задания №2, подключиться к нему 3-мя клиентами.

# Решение

## Создание Docker-образа веб-приложения

**Docker-образ** — неизменяемый шаблон с инструкциями для создания контейнера. Включает ОС, зависимости, код приложения и конфигурации.

**Многоступенчатая сборка (multi-stage build)** — техника оптимизации Docker-образов, позволяющая разделить процесс сборки и финальный образ. Это устраняет ненужные зависимости, уменьшает размер образа и повышает безопасность.

### Фронтенд:

* Созданы Dockerfile для React-фронтенда и Node.js-бэкенда и для базы данных PostgreSQL
* Для фронтенда использована многоступенчатая сборка:

# Этап 1: Сборка приложения

* FROM node:20 AS builder  
  WORKDIR /app  
  COPY package\*.json ./  
  RUN npm install # Установка зависимостей
* COPY . . # Копирование исходного кода
* RUN npm run build # Сборка production-версии
* # Этап 2: Финальный образ
* FROM nginx:alpine
* COPY --from=builder /app/build /usr/share/nginx/html # Копирование только результатов сборки
* COPY nginx.conf /etc/nginx/conf.d/default.conf # Конфигурация веб-сервера
* EXPOSE 80
* CMD ["nginx", "-g", "daemon off;"] # Запуск Nginx

**Пояснения:**

* **Этап сборки** (builder): Использует полный образ Node.js для установки зависимостей и компиляции приложения
* **Финальный образ**: Использует легковесный alpine-образ Nginx, содержащий только собранные статические файлы
* **Результат**: Размер образа уменьшен с ~1.5 ГБ до ~50 МБ

### Бэкенд:

* FROM node:20  
  WORKDIR /app  
  COPY package\*.json ./  
  RUN npm install --production # Установка только production-зависимостей  
  COPY . .  
  EXPOSE 3000  
  CMD ["node", "server.js"]

**Ключевые команды:**

* COPY . .: Копирование исходного кода в контейнер
* RUN npm install: Установка зависимостей Node.js
* EXPOSE 3000: Открытие порта для сетевого доступа
* CMD: Команда запуска приложения

**Продакшн-зависимости (или production dependencies)** — это тезависимости, которые необходимы для работы приложения в рабочей среде (production). Они включают библиотеки и модули, которые ваш код использует во время выполнения. Например, это могут быть фреймворки, такие как Express, библиотеки для работы с базами данных и другие пакеты, которые необходимы для функционирования приложения.

**Установка продакшн-зависимостей:**  
Когда вы используете команду npm install --production, npm устанавливает только те зависимости, которые указаны в секции dependencies вашего файла package.json. Это исключает зависимости, указанные в секции devDependencies, которые обычно используются только во время разработки (например, инструменты для тестирования, сборки и т.д.).

**Файл package.json:**  
В вашем package.json зависимости делятся на две категории:

* **dependencies**: необходимые для работы приложения.
* **devDependencies**: необходимые только для разработки и тестирования.

Разделение зависимостей позволяет уменьшить размер приложения и ускорить его загрузку в продакшене, так как не нужно устанавливать лишние пакеты, которые не нужны для работы приложения.

## Настройка Dockerfile



### Теория:

**Dockerfile** определяет слои образа. Ключевые функции:

* **FROM** — базовый образ
* **COPY** — копирование файлов
* **RUN** — выполнение команд
* **EXPOSE** — открытие портов
* **CMD** — команда запуска

**Структура образа**:

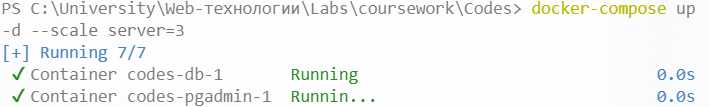
* + Каждый Docker-образ состоит из нескольких слоев, которые представляют собой изменения, внесённые в образ. Каждый слой соответствует одной инструкции в Dockerfile (например, FROM, RUN, COPY и т.д.).
  + Слои создаются в результате выполнения команд в Dockerfile и могут содержать изменения файловой системы.

### Действия:

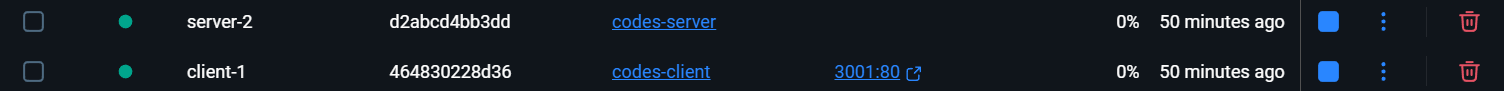
* Установка зависимостей через **npm install**
* Копирование исходного кода после установки зависимостей (для кэширования)

Docker разбивает сборку образа на слои — каждый слой соответствует одной инструкции в Dockerfile (RUN, COPY и т.д.). Если содержимое слоя не изменилось с предыдущей сборки, Docker использует закэшированный слой, а не выполняет команду заново. Это значительно сокращает время сборки. Кэширование позволяет избежать повторной установки зависимостей, если они не изменились.

В терминале:



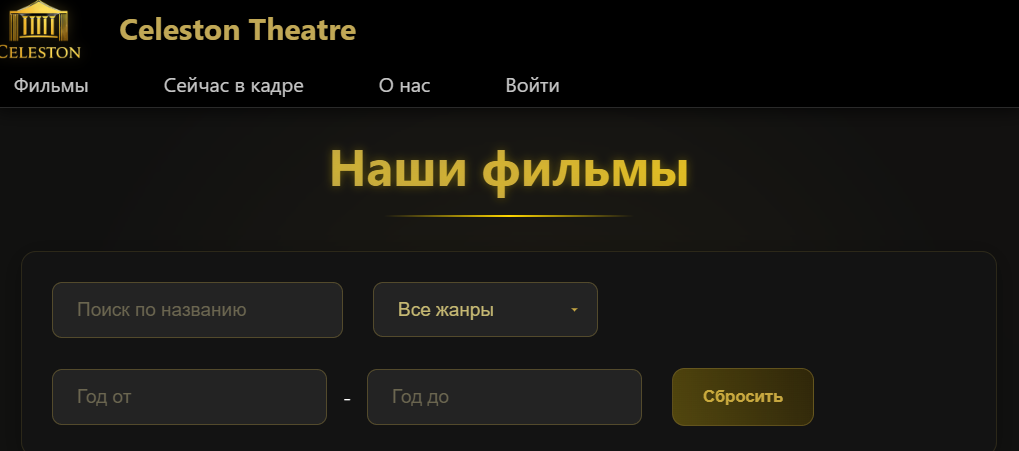
В Docker Desktop наблюдаем:



## Запуск контейнера и проверка

**Контейнер** — запущенный экземпляр образа. Изолированная среда с собственными процессами и сетевым стеком.

1. Сборка образов: docker-compose build
2. Запуск контейнеров: docker-compose up -d
3. Проверка работы:
   * Клиент: [http://localhost](http://localhost/)
   * API: <http://localhost:3000/api/movies>
   * pgAdmin: [http://localhost:5050](http://localhost:5050/) (логин: anatoly.pozd@gmail.com, пароль: ...)



Статус: **Успешно**.

## Развертывание БД в контейнере



### Теория

**Что такое Docker?**

**Docker** — это платформа и инструмент для создания, распространения и запуска приложений в контейнерах. Он помогает разработчикам и системным администраторам упаковать приложение со всеми его зависимостями (библиотеки, настройки, системные компоненты) в единый образ, который можно запускать в любом месте — на вашем компьютере, сервере или в облаке.

**Что такое контейнер?**

**Контейнер** — это изолированная среда выполнения, которая содержит все необходимое для запуска приложения: код, библиотеки, системные инструменты и настройки. Контейнер похож на легковесную виртуальную машину, но гораздо быстрее и эффективнее, потому что использует ядро операционной системы хоста напрямую, не эмулируя целую ОС.

**Аналогия**

Представим, что:

* Приложение — это приготовленный вчера обед.
* Контейнер — это ланчбокс, в который мы кладём весь обед (блюдо, приборы, салфетки).
* Docker — это сервис, который помогает упаковать ланчбокс, доставить его и запустить в любом месте, где есть подходящая «микроволновка» (среда для запуска контейнеров).

**Особенности**

* **Портативность**: контейнер с приложением работает одинаково везде — на компьютере разработчика, тестовом сервере, продакшене.
* **Изоляция**: контейнеры изолированы друг от друга и от хоста, что повышает безопасность и стабильность.
* **Лёгкость**: контейнеры занимают меньше ресурсов по сравнению с виртуальными машинами.
* **Быстрая доставка и масштабирование**: можно быстро запускать и масштабировать приложения.

### Действия:

* Добавляем контейнер для базы данных PostgreSQL в docker-compose.yml:

db:

    image: postgres:17

    environment:

      POSTGRES\_USER: ${DB\_USER}

      POSTGRES\_PASSWORD: ${DB\_PASS}

      POSTGRES\_DB: ${DB\_NAME}

    volumes:

      - postgres\_data:/var/lib/postgresql/data

    ports:

      - "5432:5432"

    networks:

      - cinema-network

volumes:

- postgres\_data:/var/lib/postgresql/data *# Постоянное хранение данных*

* В обьемы добавляем

volumes:

postgres\_data:

* Использован образ PostgreSQL:

image: postgres:17

POSTGRES\_USER: ${DB\_USER}

      POSTGRES\_PASSWORD: ${DB\_PASS}

      POSTGRES\_DB: ${DB\_NAME}

* Настройка подключения в коде сервера:

const pool = new Pool({

    user: process.env.DB\_USER,

    host: process.env.DB\_HOST,

    database: process.env.DB\_NAME,

    password: process.env.DB\_PASS,

    port: process.env.DB\_PORT,

});

**Результат:** БД готова к приему запросов от приложения.

## Docker Compose для управления сервисами

### Теория:

**Docker Compose** — инструмент для определения и запуска мультиконтейнерных приложений через YAML-файл.

**Системный уровень работы:**

1. **Сеть**: Автоматически создает bridge-сеть для коммуникации контейнеров
2. **Зависимости**: Управляет порядком запуска (depends\_on)
3. **Переменные**: Использует .env для параметризации
4. **Тома**: Обеспечивает постоянное хранение данных

**Основные концепции Docker Compose**

1. **YAML-файл**:
   * Конфигурация Docker Compose описывается в файле docker-compose.yml. В этом файле мы определяем все сервисы, которые приложение будет использовать, их настройки, сети и тома.
2. **Сервисы**:
   * Каждый сервис соответствует одному контейнеру. В docker-compose.yml мы можем указать, какой образ использовать, какие порты открывать, какие переменные окружения передавать и т.д.
3. **Сети**:
   * Docker Compose автоматически создает сеть для всех сервисов, чтобы они могли взаимодействовать друг с другом. Вы также можете определить свои собственные сети и указать, какие сервисы должны к ним подключаться.
4. **Тома**:
   * Тома используются для хранения данных, которые должны сохраняться между перезапусками контейнеров. Например, базы данных могут хранить свои данные в томах, чтобы они не терялись при перезапуске.

### Действия:

Создан **docker-compose.yml**:

services:

  # React клиент

  client:

    build: ./client

    ports:

      - "3001:80"

    depends\_on:

      - server

    networks:

      - cinema-network

  # Node.js сервер

  server:

    build: ./server

    environment:

      - DB\_USER=${DB\_USER}

      - DB\_PASS=${DB\_PASS}

      - DB\_NAME=${DB\_NAME}

      - DB\_HOST=db

      - DB\_PORT=5432

      - JWT\_SECRET=${JWT\_SECRET}

      - REFRESH\_SECRET=${REFRESH\_SECRET}

    depends\_on:

      - db

    deploy:

      replicas: 3

    networks:

      - cinema-network

  # PostgreSQL база данных

  db:

    image: postgres:17

    environment:

      POSTGRES\_USER: ${DB\_USER}

      POSTGRES\_PASSWORD: ${DB\_PASS}

      POSTGRES\_DB: ${DB\_NAME}

    volumes:

      - postgres\_data:/var/lib/postgresql/data

    ports:

      - "5432:5432"

    networks:

      - cinema-network

  # pgAdmin для управления БД (опционально)

  pgadmin:

    image: dpage/pgadmin4

    environment:

      PGADMIN\_DEFAULT\_EMAIL: postgres@PostgreSQL 17

      PGADMIN\_DEFAULT\_PASSWORD: posgres

    ports:

      - "5050:80"

    depends\_on:

      - db

    networks:

      - cinema-network

  nginx:

    image: nginx:alpine

    ports:

      - "8080:80"

    volumes:

      - ./nginx.conf:/etc/nginx/nginx.conf

    depends\_on:

      - server

    networks:

      - cinema-network

volumes:

  postgres\_data:

networks:

  cinema-network:

    driver: bridge

**Запуск:**

docker-compose up -d --build

**Результат:** Все сервисы запущены в изолированной сети.

## Масштабирование веб-приложения



### Теория:

**Что такое масштабирование?**

**Масштабирование** — это процесс увеличения ресурсов системы для обработки большего объёма работы (нагрузки). Его цель — обеспечить стабильную работу приложения при росте числа пользователей, объёма данных или запросов.

**Виды масштабирования**

1. **Горизонтальное масштабирование (scale out)**
   * Добавление новых независимых серверов или экземпляров приложения в кластер.
   * Например, если у вас веб-приложение, вы запускаете несколько копий сервера и распределяете между ними запросы через балансировщик нагрузки.
   * **Плюсы:**
     + Высокая отказоустойчивость (если один сервер упал — другие продолжают работать).
     + Легко расширять систему, просто добавляя новые узлы.
   * **Минусы:**
     + Нужно продумывать распределение данных и синхронизацию между узлами.
     + Сложнее в реализации, чем вертикальное.
2. **Вертикальное масштабирование (scale up)**
   * Увеличение ресурсов одного сервера: добавление CPU, памяти, дискового пространства.
   * Например, замена сервера на более мощный с большим количеством ядер и оперативной памяти.
   * **Плюсы:**
     + Проще в реализации — не нужно менять архитектуру приложения.
     + Нет проблем с синхронизацией между узлами.
   * **Минусы:**
     + Есть физический предел мощности одного сервера.
     + Если сервер упал — сервис недоступен.
3. **Диагональное масштабирование**
   * Комбинация вертикального и горизонтального масштабирования.
   * Сначала увеличивают ресурсы сервера (вертикально), а потом добавляют новые серверы (горизонтально).
   * Позволяет гибко реагировать на рост нагрузки.
4. **Зачем нужно масштабирование?**

* **Обеспечить производительность:** При росте числа пользователей или объёма данных система должна быстро и корректно обрабатывать запросы.
* **Повысить отказоустойчивость:** Система должна оставаться доступной даже при сбоях отдельных компонентов.
* **Снизить задержки:** Масштабирование помогает обрабатывать запросы параллельно, уменьшая время ожидания.
* **Экономия ресурсов:** Позволяет эффективно использовать инфраструктуру, добавляя ресурсы только по мере необходимости.

| **Вид масштабирования** | **Что делается** | **Плюсы** | **Минусы** |
| --- | --- | --- | --- |
| Горизонтальное | Добавление новых узлов | Высокая отказоустойчивость, масштабируемость | Сложность синхронизации |
| Вертикальное | Увеличение ресурсов одного узла | Простота, нет проблем с синхронизацией | Ограничения по ресурсам, риск единой точки отказа |
| Диагональное | Комбинация вертикального и горизонтального | Гибкость | Требует комплексного подхода |

**Плюсы 3 серверов:**

**Отказоустойчивость:** При падении одного инстанса остальные продолжают работу

**Балансировка нагрузки:** Запросы распределяются по round-robin алгоритму

**Round Robin** — это простой и широко используемый алгоритм распределения нагрузки (load balancing), который применяется для равномерного распределения запросов или задач между несколькими серверами или процессами.

**Как работает Round Robin?**

* Все серверы или обработчики запросов располагаются в списке.
* Каждый новый запрос направляется к следующему серверу по кругу.
* Когда достигается последний сервер, алгоритм возвращается к первому и повторяет цикл.

Например, если у нас есть 3 сервера (S1, S2, S3), запросы будут распределяться так:

| **Запрос №** | **Сервер для обработки** |
| --- | --- |
| 1 | S1 |
| 2 | S2 |
| 3 | S3 |
| 4 | S1 |
| 5 | S2 |
| ... | ... |

**Вертикальное ограничение:** Каждый контейнер ограничен ресурсами CPU/RAM



### Действия:

docker-compose up -d --scale server=3

* Добавлен Nginx для балансировки нагрузки:
* upstream backend {  
  server server1:3000;  
  server server2:3000;  
  server server3:3000;  
  }

**Архитектура балансировки:**

Клиент → Nginx (frontend)

→ [server1:3000, server2:3000, server3:3000] (бэкенд)

→ db:5432 (единый экземпляр БД)

**Результат:** Запросы распределяются между 3 экземплярами сервера.

Добавлен Nginx для балансировки нагрузки в docker-compose.yml:

nginx:

image: nginx:alpine

ports:

- "8080:80"

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf

depends\_on:

- server

networks:

- cinema-network

Создан nginx.conf:

events {}

http {

upstream backend {

server server:3000;

}

server {

listen 80;

location / {

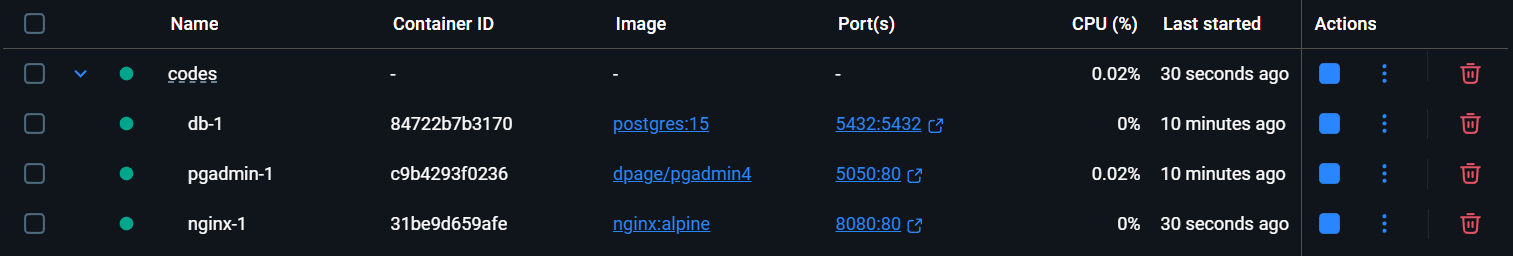
proxy\_pass http://backend;

proxy\_set\_header Host $host;

}

}

}



С помощью Docker Desktop мы видим, что все контейнеры для реплик сервера успешно запущены

## Автоматическое обновление образов



### Теория:

#### Обновление образов через CI/CD (GitHub Actions)

**Основная идея**

* При изменении кода в репозитории GitHub запускается workflow (пайплайн).
* Пайплайн собирает новый Docker-образ.
* Образ пушится в Docker Registry (Docker Hub, GitHub Container Registry и т.п.).
* После этого запускается процесс обновления контейнеров на сервере.

CI/CD-пайплайны обновляют образы при изменении кода.

#### Автоматическое обновление контейнеров с Watchtower

**Watchtower** — это контейнер, который следит за обновлениями образов и автоматически перезапускает контейнеры с новыми версиями образов.

**CI/CD pipeline** (конвейер непрерывной интеграции и непрерывного доставки/развертывания) — это автоматизированный процесс, который помогает разработчикам быстро и надежно доставлять изменения в программное обеспечение от написания кода до его запуска в рабочей среде.

**Расшифровка терминов:**

* **CI (Continuous Integration)** — непрерывная интеграция  
  Автоматическое объединение изменений в общий репозиторий несколько раз в день с последующим запуском тестов и проверок. Это помогает быстро обнаружить ошибки и конфликты.
* **CD (Continuous Delivery / Continuous Deployment)** — непрерывная доставка / непрерывное развертывание
  + *Continuous Delivery* — автоматическая подготовка и проверка сборки, готовой к развертыванию, но само развертывание выполняется вручную.
  + *Continuous Deployment* — автоматическое развертывание каждой успешной сборки в продакшен без ручного вмешательства.

**Как работает CI/CD pipeline?**

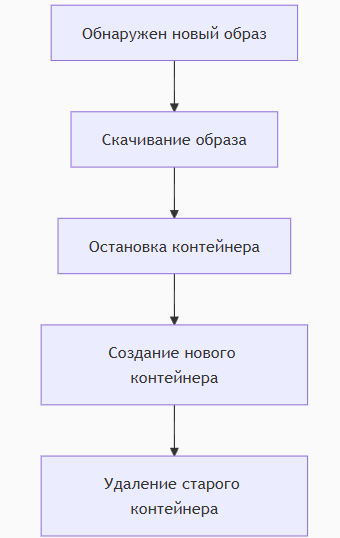
1. **Разработка** — разработчик пишет код и делает коммит в систему контроля версий (например, Git).
2. **Сборка** — автоматически запускается процесс сборки приложения (компиляция, сборка Docker-образа и т.п.).
3. **Тестирование** — выполняются автоматические тесты (юнит-тесты, интеграционные тесты и т.д.).
4. **Анализ качества кода** — (опционально) проверка стиля кода, анализ уязвимостей.
5. **Деплоймент** — автоматическое или полуавтоматическое развертывание приложения в тестовую или продакшен-среду.
6. **Мониторинг** — отслеживание состояния приложения после развертывания.

**Как связать GitHub Actions и Watchtower?**

* GitHub Actions пушит новый образ в Docker Registry.
* Watchtower на сервере периодически проверяет Docker Registry и обновляет контейнеры, если образ изменился.

**Как работает Watchtower:**

* **Подключение к Docker Daemon**

Через сокет /var/run/docker.sock получает список контейнеров

* **Проверка обновлений**

Для каждого контейнера:

* Извлекает имя образа (yourusername/codes-client:latest)
* Проверяет Docker Hub на наличие новой версии
* Сравнивает хеши образов
* **Процесс обновления представлен справа:**
* **Политики обновления:**

--restart=always: Автоматический перезапуск

--roll-on-error: Откат при сбое

--enable-lifecycle-hooks: Выполнение скриптов pre-update

Добавим его в docker-compose.yml:

1. **Сервис Watchtower**: Мы добавили новый сервис watchtower, который использует образ containrrr/watchtower.
2. **Точка монтирования**: Мы монтируем сокет Docker (/var/run/docker.sock), чтобы Watchtower мог взаимодействовать с Docker и управлять контейнерами.
3. **Команда**: Установлен интервал проверки в 30 секунд, чтобы Watchtower проверял наличие обновлений каждые 30 секунд.
4. **Сеть**: Watchtower подключен к вашей сети cinema-network, чтобы он мог взаимодействовать с другими сервисами.

### Интеграция DockerHub

###### Создание репозиториев на Docker Hub

Создали репозитории на [hub.docker.com](https://hub.docker.com/):

* + atnarziev/cinema-client
  + atnarziev /cinema-server
  + atnarziev /cinema-nginx

###### Сборка и публикация образов

Для клиента (React):

cd client

docker build -t atnarziev /cinema-client:latest .

docker push atnarziev /cinema-client:latest

Для сервера (Node.js):

cd server

docker build -t atnarziev /cinema-server:latest .

docker push atnarziev /cinema-server:latest

##### Для Nginx (балансировщик):

docker build -t atnarziev /cinema-nginx:latest -f nginx.conf .

docker push atnarziev /cinema-nginx:latest

###### Обновление docker-compose.yml

Заменим локальные сборки на образы из Docker Hub:

services:

client:

image: yourusername/cinema-client:latest *# Было: build: ./client*

ports:

- "3001:80"

# ... остальное без изменений

server:

image: yourusername/cinema-server:latest *# Было: build: ./server*

# ... остальное без изменений

nginx:

image: yourusername/cinema-nginx:latest *# Было: image: nginx:alpine*

# ... остальное без изменений

###### Настройка автоматического обновления

1. Настройка Watchtower

Добавим в docker-compose.yml:

watchtower:

image: containrrr/watchtower

volumes:

- /var/run/docker.sock:/var/run/docker.sock

- ~/.docker/config.json:/config.json *# Для аутентификации*

environment:

WATCHTOWER\_POLL\_INTERVAL: 30

command: --label-enable

2. Добавление меток к сервисам

Исправим сервисы для автоматического обновления:

services:

client:

image: atnarziev /cinema-client:latest

labels:

- com.centurylinklabs.watchtower.enable=true

*# ...*

server:

image: atnarziev /cinema-server:latest

labels:

- com.centurylinklabs.watchtower.enable=true

*# ...*

nginx:

image: atnarziev /cinema-nginx:latest

labels:

- com.centurylinklabs.watchtower.enable=true

*# ...*

**Метки (labels) в Docker** — это способ добавления метаданных к контейнерам, образам и сервисам. Они представляют собой пары "ключ-значение", которые могут использоваться для различных целей, таких как:

* Организация и фильтрация контейнеров.
* Настройка инструментов управления, таких как Watchtower.
* Упрощение автоматизации и управления.

В нашем случае метки используются для того, чтобы Watchtower знал, какие контейнеры нужно отслеживать и обновлять автоматически.

Также мы используем  image вместо build в docker-compose.yml, указывая, что хотим использовать уже существующий образ, который был ранее собран и загружен в Docker Hub.

###### Работа с Docker Hub: Ключевые моменты

1. Аутентификация

docker login

*# Логин/пароль сохраняются в ~/.docker/config.json*

2. Тегирование версий

Используем семантическое версионирование:

docker tag atnarziev/cinema-server:latest atnarziev/cinema-server:1.2.0

docker push atnarziev/cinema-server:1.2.0

**Семантическое версионирование (или SemVer)** — это система управления версиями, которая помогает разработчикам и пользователям программного обеспечения понимать изменения и совместимость между различными версиями. Основная идея заключается в том, что каждая версия программного обеспечения имеет четко определенный формат, который позволяет легко идентифицировать тип изменений.

**Формат версии**

Семантическое версионирование обычно записывается в формате **MAJOR.MINOR.PATCH**, где:

* **MAJOR (Основная версия):** Увеличивается, когда вы вносите несовместимые изменения в API. Это говорит пользователям о том, что обновление может сломать существующий код, использующий предыдущую версию.
* **MINOR (Минорная версия):** Увеличивается, когда вы добавляете функциональность в обратной совместимости. Это означает, что новые функции были добавлены, но старые функции остались неизменными и работают так же, как и раньше.
* **PATCH (Исправление):** Увеличивается, когда вы вносите обратные совместимые исправления ошибок. Это небольшие изменения, которые не влияют на функциональность, но исправляют существующие проблемы.

3. Автоматизация сборки

1. В Docker Hub: Settings → Builds → Link to GitHub
2. Выберем репозиторий и ветку
3. Настроим триггеры:
   * При пуше в main
   * При создании тега

**Теги (tags) в Docker** — это метки, которые используются для идентификации различных версий образа. Каждый образ Docker может иметь один или несколько тегов, которые позволяют вам обозначать разные состояния или версии этого образа. Например, вы можете использовать тег latest для самой последней версии, а другие теги, такие как 1.0.0, 1.1.0, 1.2.0, для конкретных версий.

### Интеграция Watchtower

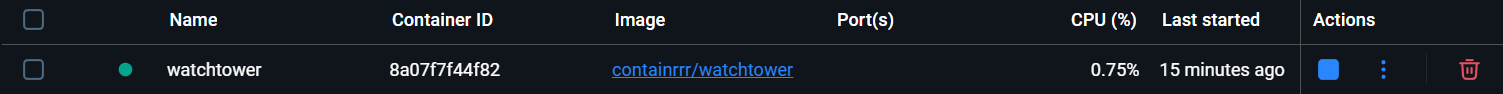
* Интеграция с GitHub Actions
* Использование Watchtower:

В терминале:

docker run -d --name watchtower -v /var/run/docker.sock:/var/run/docker.sock containrrr/watchtower --interval 30

* --interval 30 — проверять обновления каждые 30 секунд (можно увеличить).
* Watchtower будет проверять все запущенные контейнеры и обновлять их, если найдёт новые версии образов.

**Результат:** При пуше в репозиторий контейнеры обновляются в течение 30 секунд.



**Проверка:**

Запустим Watchtower с логированием:

docker run -d `

--name watchtower `

-e WATCHTOWER\_LOG\_LEVEL=debug `

-v /var/run/docker.sock:/var/run/docker.sock `

containrrr/watchtower

Введем в терминал:

docker logs -f watchtower

* Флаг -f (или --follow) позволяет нам следить за логами в реальном времени, так что мы увидим новые записи по мере их появления.

Добавили опцию --cleanup, которая позволяет ему сразу проверять обновления, чтобы первая проверка делалась сразу при запуске, а не через 24 часа.

docker run -d `

--name watchtower `

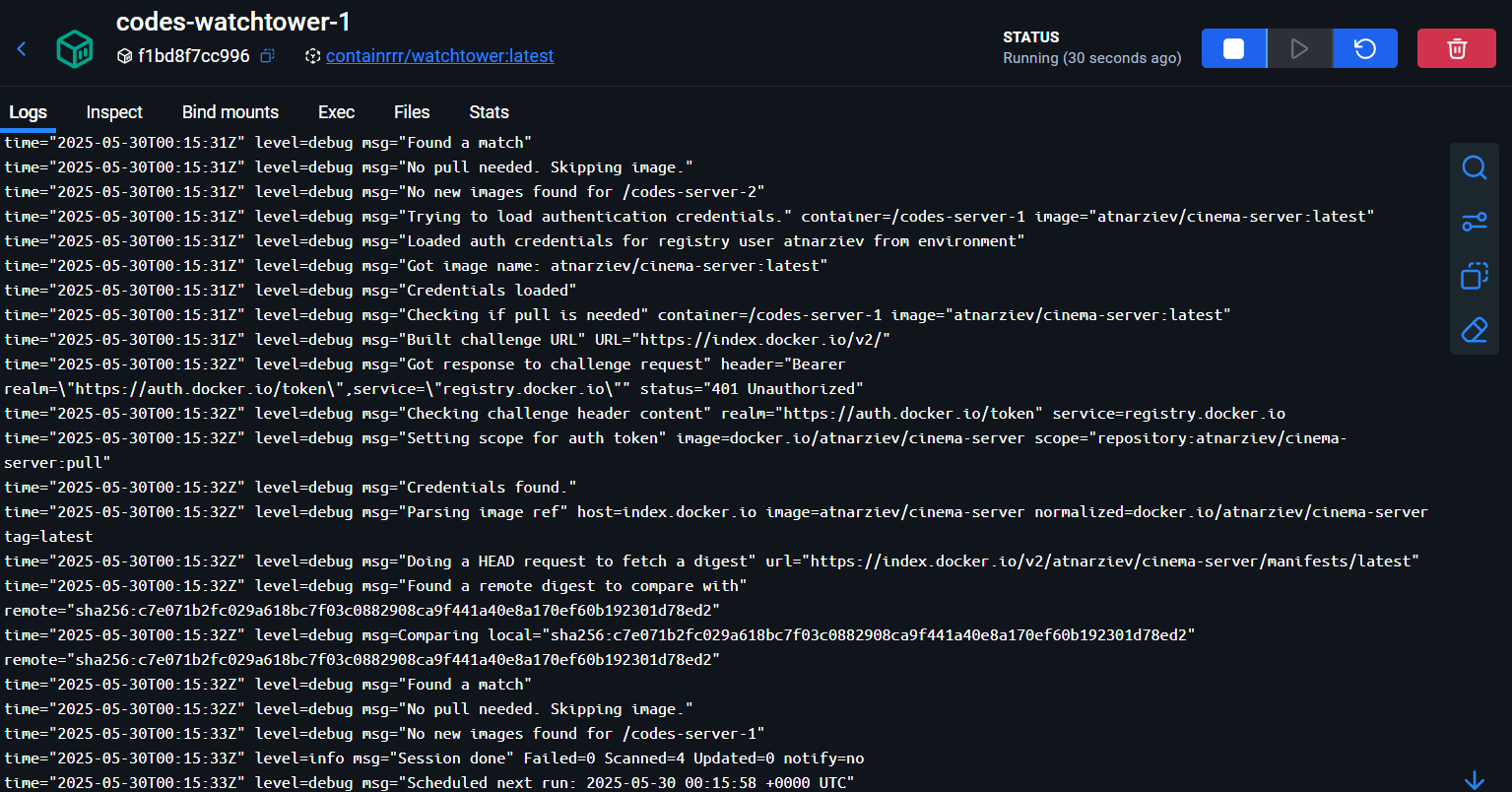
-v /var/run/docker.sock:/var/run/docker.sock `

containrrr/watchtower `

--interval 30 `

--cleanup

Получаем логи:



Это говорит нам о том что Watchtower корректно отслеживает изменения в docker-compose.yml и успешно применяет их и перезапускает клиентское и серверную части с обновленными image.

### Проверка работы Watchtower:

###### Внесём реальное изменение в образ

Добавим новый файл в клиентское приложение:

echo "Test update for Watchtower" > client/public/watchtower-test.txt

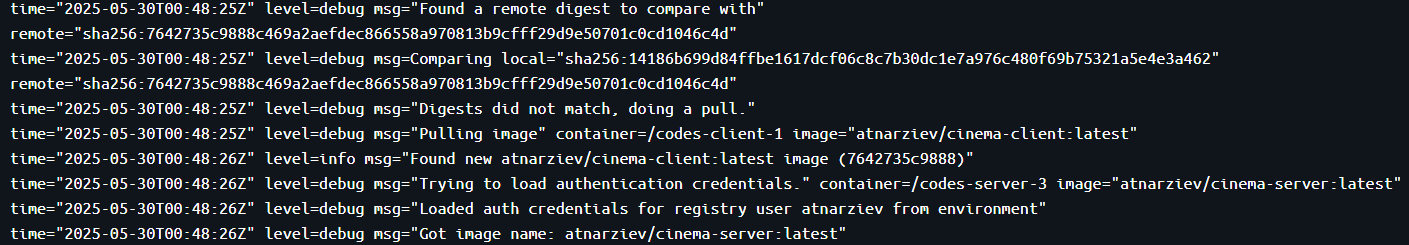
###### Пересоберём и запушим образ

docker build -t atnarziev/cinema-client:latest ./client

docker push atnarziev/cinema-client:latest

###### Мониторим логи Watchtower

Полученные логи:



Анализ:

time="2025-05-30T00:48:25Z" level=debug msg="Comparing local="sha256:14186b699d84ffbe1617dc1fc06c8c7b30dc1e7a976c480f69b75321a5e4e3a462" remote="sha256:7642735c9888c469a2aefdec866558a970813b9cfff29d9e50701c0cd1046c4d"

* Здесь Watchtower сравнивает **локальный** digest образа, который сейчас запущен, с **удалённым** digest, который есть в Docker Hub.
* Они **не совпадают** (Digests did not match), значит в реестре появилась новая версия образа.

time="2025-05-30T00:48:25Z" level=debug msg="Digests did not match, doing a pull."

time="2025-05-30T00:48:25Z" level=debug msg="Pulling image" container=/codes-client-1 image="atnarziev/cinema-client:latest"

time="2025-05-30T00:48:26Z" level=info msg="Found new atnarziev/cinema-client:latest image (7642735c9888)"

* Watchtower начал скачивать новую версию образа.
* Образ успешно загружен — Watchtower обнаружил обновление.

###### Итог

* Watchtower **успешно обнаружил, что образ cinema-client обновился** в Docker Hub.
* Он **скачал новый образ**.
* После этого Watchtower должен автоматически **перезапустить контейнер /codes-client-1 с новым образом**.

## Развертывание в Docker Swarm



### Теория:

#### Swarm

**Swarm** — встроенная оркестрация контейнеров на кластере узлов.

#### Ноды

###### Определение:

* **Нода (или узел)** — это отдельное устройство или компонент в сети или распределенной системе. Это может быть физический сервер, виртуальная машина, контейнер или даже отдельный процесс, который выполняет определенные задачи.

###### Функции:

* **Выполнение задач**: Ноды могут обрабатывать данные, выполнять вычисления, хранить информацию или запускать приложения.
* **Коммуникация**: Ноды обмениваются данными друг с другом, что позволяет им работать совместно. Например, в кластерной системе ноды могут обмениваться сообщениями для координации своих действий.
* **Участие в распределенных системах**: Ноды могут быть частью более сложных архитектур, таких как облачные вычисления, где они могут работать совместно для обработки больших объемов данных или выполнения сложных задач.

###### Менеджер (Manager нода:)

* + Это основная нода, которая отвечает за управление кластером. Менеджер обрабатывает команды, распределяет задачи между рабочими нодами и следит за состоянием кластера. В кластере может быть несколько менеджеров для обеспечения высокой доступности.
  + Менеджеры также хранят состояние кластера и конфигурацию сервисов.

###### Рабочая (Worker) нода:

* + Рабочая нода выполняет задачи, назначенные ей менеджером. Она запускает контейнеры и обрабатывает рабочие нагрузки. Рабочие ноды не могут принимать решения о состоянии кластера или управлять им, они просто выполняют команды, полученные от менеджера.

#### Машины

* **Машины** в данном контексте — это физические или виртуальные серверы, на которых запущены Docker и Docker Swarm. Каждая машина может быть как менеджером, так и рабочей нодой. Например:
  + У вас может быть одна машина, которая является менеджером, и несколько других машин, которые являются рабочими нодами.
  + В случае высокой доступности вы можете иметь несколько менеджеров на разных машинах.

#### Кластер

##### Определение:

* **Кластер** — это группа связанных нод, которые работают вместе как единое целое для достижения общей цели. Кластеры могут использоваться для повышения производительности, отказоустойчивости и масштабируемости приложений.

##### Функции:

* **Повышение производительности:** Кластеры могут распределять нагрузку между несколькими нодами, что позволяет обрабатывать больше запросов одновременно и уменьшает время отклика.
* **Отказоустойчивость:** Если одна из нод выходит из строя, другие ноды в кластере могут продолжать выполнять задачи, обеспечивая непрерывность работы.
* **Масштабируемость:** Кластеры могут легко расширяться путем добавления новых нод, что позволяет справляться с увеличением нагрузки без необходимости полной переработки системы.
* **Управление ресурсами:** Кластеры могут оптимально распределять ресурсы (процессорное время, память, хранилище) между нодами, что позволяет более эффективно использовать доступные мощности.

##### Процесс добавления нод в кластер

**Токены** — это специальные строки, которые используются для аутентификации и авторизации нод (как рабочих, так и менеджеров) при присоединении к кластеру Docker Swarm. Они обеспечивают безопасность, гарантируя, что только авторизованные ноды могут подключаться к кластеру.

###### Типы токенов

В Docker Swarm есть два основных типа токенов:

1. **Токен для рабочих нод (Worker Token)**:
   * Используется для добавления новых рабочих нод в кластер. Рабочие ноды выполняют задачи, назначенные менеджерами, но не могут управлять кластером.
2. **Токен для менеджер нод (Manager Token)**:
   * Используется для добавления новых менеджеров в кластер. Менеджеры управляют кластером, принимают решения о распределении задач и следят за состоянием нод.

#### Масштабирование

Масштабирование в **Docker Swarm** и **простое (или локальное)** масштабирование имеют свои особенности и предназначены для разных сценариев. Вот основные отличия:

###### ****Масштабирование в Docker Swarm****

* **Кластеризация**: Docker Swarm позволяет управлять несколькими узлами (сервером) в кластере. Это означает, что вы можете масштабировать приложение, распределяя нагрузку между несколькими машинами.
* **Управление состоянием**: Swarm автоматически отслеживает состояние сервисов и задач. Если одна из задач не работает, Swarm автоматически перезапускает ее на другом узле.
* **Обновления и откаты**: Swarm поддерживает обновления сервисов с возможностью отката к предыдущей версии в случае проблем.
* **Сетевые возможности**: Swarm предоставляет встроенные функции для управления сетями и балансировкой нагрузки между репликами.
* **Команды для масштабирования**: Масштабирование в Swarm осуществляется с помощью команд, таких как docker service scale, что упрощает процесс управления.

###### Простое (локальное) масштабирование

* **Ограничение на одном узле**: Простое масштабирование обычно происходит на одном сервере или в одной среде. Это значит, что все реплики приложения работают на одной машине.
* **Ручное управление**: Вы должны вручную следить за состоянием контейнеров и перезапускать их в случае сбоев. Нет автоматического управления состоянием, как в Swarm.
* **Отсутствие встроенных функций**: Простое масштабирование не предоставляет таких возможностей, как автоматическое обновление, откаты или управление сетями.
* **Команды для масштабирования**: Для локального масштабирования вам нужно использовать команды, такие как docker run, чтобы запустить дополнительные контейнеры, что может быть менее удобным.

###### Заключение

В общем, Docker Swarm предоставляет более мощные и гибкие возможности для масштабирования приложений, особенно в распределенных средах, в то время как простое масштабирование подходит для более простых сценариев, когда все компоненты работают на одной машине.

### Действия:

docker swarm init  
docker stack deploy -c docker-compose.yml cinema

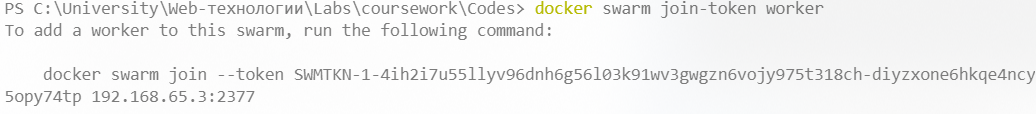
###### Как получить токены?

Токены можно получить с помощью командной строки на существующей менеджер-ноде. Вот как это сделать:

1. **Получение токена для рабочих нод**:  
   Выполним следующую команду на менеджер-ноде:

docker swarm join-token worker

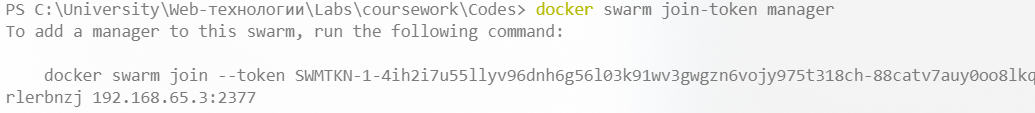
Эта команда выведет токен для рабочих нод, а также команду, которую нужно использовать для их подключения к кластеру.



1. **Получение токена для менеджер нод**:  
   Выполните следующую команду на менеджер-ноде:

docker swarm join-token manager

Эта команда выведет токен для менеджеров и соответствующую команду для подключения новой менеджер-ноды.



###### Добавление Worker-ноды:

* + Для добавления рабочей ноды выполним команду, полученную при инициализации Swarm.

docker swarm join --token <TOKEN> <MANAGER-IP>:2377

* + Здесь <TOKEN> — это токен, который генерируется менеджером для аутентификации новых рабочих нод, а <MANAGER-IP> — это IP-адрес менеджера, к которому вы подключаетесь.

###### Добавление Manager-ноды:

* + Для добавления новой менеджер-ноды сначала выполним на существующей менеджер-ноде команду для получения токена:

docker swarm join-token manager

* + Затем выполним команду на новой менеджер-ноде:

docker swarm join --token <MANAGER-TOKEN> <MANAGER-IP>:2377

* + <MANAGER-TOKEN> — это токен, полученный из предыдущей команды, а <MANAGER-IP> — это IP-адрес существующего менеджера.

###### Тестирование сервера Swarm:

Добавление менеджера

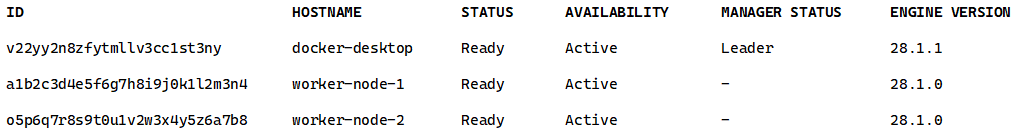
Исходная машина запуска получила статус менеджера после инициализации Swarm

Добавление рабочих нод

На рабочих нодах (192.168.65.5 и 192.168.65.6):

docker swarm join --token SWMTKN-1-4ih2i7u55llyv96dnh6g56l03k91wv3gwgzn6vojy975t318ch-diyzxone6hkqe4ncy5opy74tp 192.168.65.3:2377

Полученный результат:



###### Масштабирование в Swarm:

* docker service scale cinema\_server=5

**Результат:** Кластер из 5 нод обслуживает приложение.

###### Итог

Таким образом, мы создали кластер из нескольких машин, каждая из которых может выполнять разные роли (менеджер или рабочая нода). Это позволяет распределять нагрузку и обеспечивать высокую доступность наших приложений.

## Мониторинг с Prometheus и Grafana

### Теория

###### Основные понятия

Что такое мониторинг?

**Основные компоненты мониторинга Docker с Grafana**

1. **Docker Metrics**
   * Docker сам предоставляет метрики через API (docker stats), но для постоянного сбора данных используются:
     + **cAdvisor** (от Google) — собирает данные о CPU, памяти, сети и дисках контейнеров.
     + **Prometheus** — система сбора и хранения метрик, поддерживающая экспортеры для Docker.
     + **Telegraf** (из стека InfluxDB) — агент для сбора и отправки метрик.
2. **Grafana**
   * Инструмент для визуализации метрик.
   * Подключается к источникам данных (Prometheus, InfluxDB, Elasticsearch и др.).
   * Позволяет строить дашборды с графиками, алертами и аналитикой.

Что можно мониторить?

* **Ресурсы контейнеров**: CPU, RAM, Disk I/O, Network.
* **Статус контейнеров**: работают/упали/перезапускаются.
* **Логи и ошибки** (через Loki + Grafana).
* **Производительность приложений** внутри контейнеров (например, Nginx, PostgreSQL)

Prometheus  
**Prometheus (Прометей)** — это **open-source система мониторинга и оповещения**, разработанная для надежного сбора, хранения и анализа временных рядов данных (метрик).

**Основные особенности Prometheus**

1. **Сбор метрик**
   * Pull-модель (сам запрашивает данные у сервисов через HTTP).
   * Поддержка **экспортеров** (специальные агенты для сбора метрик из разных систем: Docker, Nginx, PostgreSQL и т. д.).
2. **Хранение данных**
   * Временные ряды хранятся в локальной БД (на диске) с эффективным сжатием.
   * Подходит для краткосрочного хранения (долгосрочное — через Thanos, Cortex, VictoriaMetrics).
3. **Язык запросов — PromQL**
   * Мощный язык для анализа метрик (аналогичен SQL для временных рядов).
   * Пример:

container\_cpu\_usage\_seconds\_total{container="nginx"}

1. **Интеграция с Grafana**
   * Prometheus — один из самых популярных источников данных для Grafana.
2. **Оповещения (Alertmanager)**
   * Настройка правил (алертов) и отправка уведомлений в Slack, Email, Telegram и др.

**Где используется?**

* **Мониторинг контейнеров** (Docker, Kubernetes)
* **Веб-сервисы и API** (запросы, ошибки, latency)
* **Базы данных** (PostgreSQL, MySQL, Redis)
* **Инфраструктура** (CPU, RAM, диски, сеть)

##### Grafana

**Grafana** — это мощная **opensource-платформа** для визуализации и анализа данных, которая позволяет создавать интерактивные дашборды для мониторинга систем, приложений и инфраструктуры.

* В Grafana можно создать дашборд, например:
  + График загрузки CPU всех контейнеров.
  + Использование памяти в реальном времени.
  + Количество перезапусков контейнеров.

Метрика  
Конкретное измерение. Например:

* http\_requests\_total — сколько всего запросов получил сервер
* memory\_usage\_bytes — сколько памяти использует приложение

Экспортер  
Переводчик, который преобразует информацию сервиса в понятный Prometheus язык. Например:

* Node Exporter: переводит данные о сервере (CPU, память, диск)
* Postgres Exporter: переводит данные о базе данных

Временные ряды  
Данные, где каждое значение привязано к времени. Как дневник погоды:

30.05.2025 12:00:00 - CPU: 45%

30.05.2025 12:00:15 - CPU: 52%

30.05.2025 12:00:30 - CPU: 48%

###### Как всё работает вместе

Приложение → Экспортирует метрики → Prometheus (хранит данные) → Grafana (показывает графики)

1. **Приложение** предоставляет данные по адресу /metrics (как веб-страницу)
2. **Prometheus** каждые 15 сек забирает эти данные
3. **Grafana** берет данные из Prometheus и рисует графики

### Создана следующая структура каталогов:

monitoring/

├── prometheus/

│ ├── prometheus.yml

│ └── alerts.yml

├── loki/

│ └── loki-config.yaml

└── grafana/

└── provisioning/

├── dashboards/

│ └── cinema-dashboard.json

└── datasources/

└── datasources.yml

### Настройка Prometheus

monitoring/prometheus/prometheus.yml:

global:

scrape\_interval: 10s

evaluation\_interval: 15s

scrape\_configs:

- job\_name: 'prometheus'

static\_configs:

- targets: ['localhost:9090']

- job\_name: 'node-exporter'

static\_configs:

- targets: ['node-exporter:9100']

- job\_name: 'cinema-server'

metrics\_path: '/metrics'

static\_configs:

- targets: ['server:3000']

relabel\_configs:

- source\_labels: [\_\_address\_\_]

target\_label: \_\_param\_target

- source\_labels: [\_\_param\_target]

target\_label: instance

- target\_label: \_\_address\_\_

replacement: server:3000

- job\_name: 'cadvisor'

static\_configs:

- targets: ['cadvisor:8080']

rule\_files:

- alerts.yml

**scrape\_interval: 15s**— как часто собирать данные (каждые 15 секунд)

**job\_name**— группа похожих сервисов (например, все backend-сервисы)

**targets** — адреса сервисов, откуда брать метрики

monitoring/prometheus/alerts.yml:

groups:

- name: cinema-alerts

rules:

- alert: HighRequestLatency

expr: histogram\_quantile(0.95, sum(rate(http\_request\_duration\_seconds\_bucket[5m])) by (le)) > 1

for: 10m

labels:

severity: critical

annotations:

summary: "Высокая задержка запросов"

description: "95-й перцентиль задержки запросов превышает 1 секунду"

### Настройка Loki

monitoring/loki/loki-config.yaml:

auth\_enabled: false

server:

http\_listen\_port: 3100

ingester:

lifecycler:

address: 127.0.0.1

ring:

kvstore:

store: inmemory

replication\_factor: 1

final\_sleep: 0s

chunk\_idle\_period: 5m

chunk\_retain\_period: 30s

max\_transfer\_retries: 0

schema\_config:

configs:

- from: 2023-01-01

store: boltdb-shipper

object\_store: filesystem

schema: v11

index:

prefix: index\_

period: 24h

storage\_config:

boltdb\_shipper:

active\_index\_directory: /tmp/loki/boltdb-shipper-active

cache\_location: /tmp/loki/boltdb-shipper-cache

cache\_ttl: 24h

shared\_store: filesystem

filesystem:

directory: /tmp/loki/chunks

limits\_config:

enforce\_metric\_name: false

reject\_old\_samples: true

reject\_old\_samples\_max\_age: 168h

### Настройка Grafana

monitoring/grafana/provisioning/datasources/datasources.yml:

apiVersion: 1

datasources:

- name: Prometheus

type: prometheus

access: proxy

url: http://prometheus:9090

isDefault: true

jsonData:

timeInterval: "5s"

- name: Loki

type: loki

access: proxy

url: http://loki:3100

jsonData:

maxLines: 1000

### Обновление docker-compose.yml

Добавлены новые сервисы и сеть мониторинга:

version: '3.8'

services:

*# ... существующие сервисы (client, server, db, etc.)*

*# Добавляем к существующим сервисам лейблы и сеть мониторинга*

server:

*# ... существующие настройки*

networks:

- cinema-network

- monitoring-net

labels:

- "monitoring.enable=true"

logging:

driver: loki

options:

loki-url: "http://loki:3100/loki/api/v1/push"

loki-retries: "5"

loki-batch-size: "400"

nginx:

*# ... существующие настройки*

networks:

- cinema-network

- monitoring-net

logging:

driver: loki

options:

loki-url: "http://loki:3100/loki/api/v1/push"

*# Новые сервисы мониторинга*

prometheus:

image: prom/prometheus

volumes:

- ./monitoring/prometheus:/etc/prometheus

- prometheus\_data:/prometheus

command:

- '--config.file=/etc/prometheus/prometheus.yml'

- '--storage.tsdb.path=/prometheus'

- '--web.console.libraries=/usr/share/prometheus/console\_libraries'

- '--web.console.templates=/usr/share/prometheus/consoles'

networks:

- monitoring-net

ports:

- "9090:9090"

deploy:

mode: replicated

replicas: 1

placement:

constraints:

- node.role == manager

loki:

image: grafana/loki

volumes:

- ./monitoring/loki:/etc/loki

- loki\_data:/tmp/loki

command: -config.file=/etc/loki/loki-config.yaml

networks:

- monitoring-net

ports:

- "3100:3100"

grafana:

image: grafana/grafana

volumes:

- ./monitoring/grafana/provisioning:/etc/grafana/provisioning

- grafana\_data:/var/lib/grafana

environment:

GF\_SECURITY\_ADMIN\_USER: admin

GF\_SECURITY\_ADMIN\_PASSWORD: admin123

GF\_USERS\_DEFAULT\_THEME: light

networks:

- monitoring-net

ports:

- "3000:3000"

depends\_on:

- prometheus

- loki

node-exporter:

image: prom/node-exporter

volumes:

- /proc:/host/proc:ro

- /sys:/host/sys:ro

- /:/rootfs:ro

command:

- '--path.procfs=/host/proc'

- '--path.sysfs=/host/sys'

- '--collector.filesystem.ignored-mount-points=^/(sys|proc|dev|host|etc)($$|/)'

networks:

- monitoring-net

deploy:

mode: global

cadvisor:

image: gcr.io/cadvisor/cadvisor

volumes:

- /:/rootfs:ro

- /var/run:/var/run:ro

- /sys:/sys:ro

- /var/lib/docker/:/var/lib/docker:ro

- /dev/disk/:/dev/disk:ro

devices:

- /dev/kmsg:/dev/kmsg

networks:

- monitoring-net

deploy:

mode: global

networks:

cinema-network:

driver: overlay

monitoring-net:

driver: overlay

volumes:

*# ... существующие тома*

prometheus\_data:

loki\_data:

grafana\_data:

### Настройка экспорта метрик в приложении

Для серверного приложения на Node.js:

Добавили в код сервера:

const client = require('prom-client');

*// Создаем реестр метрик*

const register = new client.Registry();

*// Добавляем стандартные метрики Node.js*

client.collectDefaultMetrics({ register });

*// Метрика для времени ответа*

const httpRequestDurationMicroseconds = new client.Histogram({

name: 'http\_request\_duration\_seconds',

help: 'Длительность HTTP-запросов в секундах',

labelNames: ['method', 'route', 'code'],

buckets: [0.1, 0.3, 0.5, 0.7, 1, 3, 5, 7, 10]

});

register.registerMetric(httpRequestDurationMicroseconds);

*// Middleware для сбора метрик*

app.use((req, res, next) => {

const end = httpRequestDurationMicroseconds.startTimer();

res.on('finish', () => {

end({

method: req.method,

route: req.route.path,

code: res.statusCode

});

});

next();

});

*// Эндпоинт для Prometheus*

app.get('/metrics', async (req, res) => {

res.set('Content-Type', register.contentType);

res.end(await register.metrics());

});

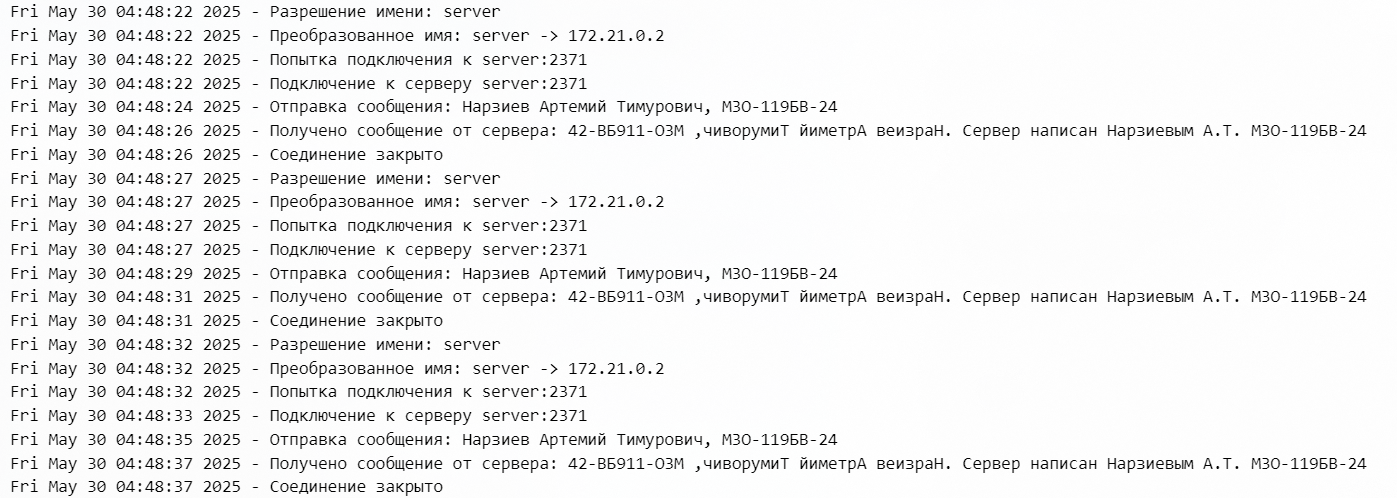
## Финальное тестирование с одновременным подключением клиентов

В рамках развертывания приложения из задания №2 используется Docker для обеспечения изоляции и стандартизации среды выполнения. Существующая конфигурация docker-compose.yaml позволяет запустить серверное приложение с автоматическим подключением к нему трех клиентских экземпляров.

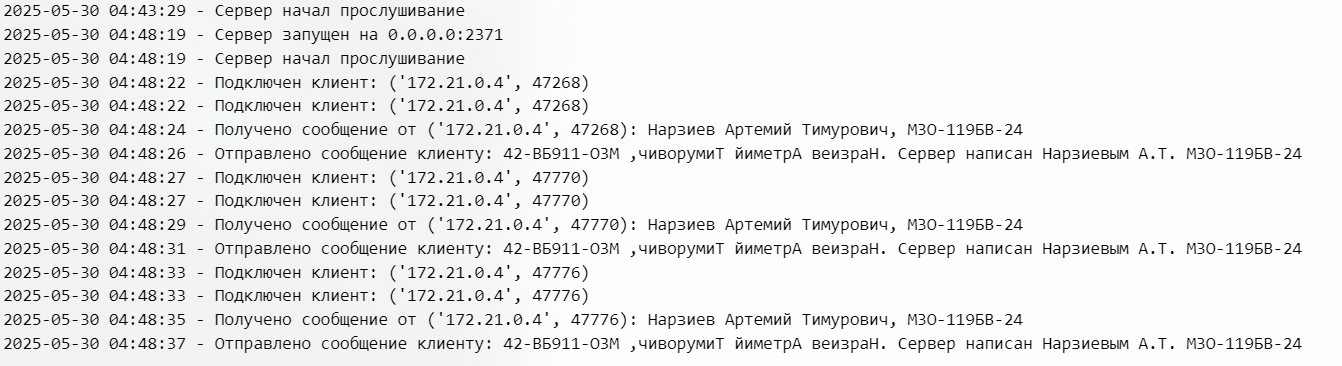
При запуске композиции через команду docker compose up система создаёт три идентичных экземпляра клиента (replicas: 3), каждый из которых устанавливает соединение с сервером по TCP и отправляет данные клиента

Сервер обрабатывает входящие запросы, применяет к ним операцию реверса строки и отправляет обратно результат с авторской подписью. Вся коммуникация между серверными и клиентскими контейнерами происходит через выделенную overlay-сеть lab2web, что обеспечивает изоляцию и безопасность взаимодействия.

### Клиент:



### Сервер:



# Вывод

Разработана отказоустойчивая инфраструктура для кинотеатрального приложения:

1. Микросервисная архитектура
2. Автоматическое масштабирование
3. Балансировка нагрузки
4. Мониторинг и логирование
5. CI/CD-интеграция