<https://habr.com/ru/companies/ruvds/articles/599929/>

<https://proglib.io/p/git-for-half-an-hour>

<https://contentim.ru/post/git>

# Коммиты

**git commit -m "Name of commit"**  # зафиксировать в коммите проиндексированные изменения (закоммитить), добавить сообщение

.gitignore тоже добавляется в индекс

Добавляются к примеру:

\*.log

/storage

/config/database.php

/config/app.php

.htaccess

/node\_modules

/themes/sambiahotel/content

**git rm —cached filename** удалить из отслеживания после добавления в .gitignore

**—cached** без удаления из файловой системы

## Ветки

**git status** показывает файлы находящиеся в индексе, на какой ветке в данный момент находимся и последние коммиты

git log история коммитов

**git checkout –b dev1** создать ветку с именем dev1, одновременно переключившись на нее

**git branch**  посмотреть список доступных веток. Зеленый цвет обозначает что это локальная ветка

**git branch -a** посмотреть список доступных веток, включая в удаленных репозиториях, помечены красным

**git branch new-branch-name** создать новую ветку

# удаленные репозитории

**git remote -v** список удаленных репозиториев

**git push origin master**  # отправить в удалённый репозиторий (с сокр. именем origin) данные своей ветки master

когда на локальном есть ветка а на удаленном нет соответствующей

**git push –u origin dev1** отправляем именно ветку **dev1** в удаленный репозиторий **origin**

**git push –u origin main** отплавяем ветку **main** в удаленный репозиторий **origin**

**origin** указатель на удаленный репозиторий

**-u** отправить в ветку с таким же названием как локальная и создать связь между этими ветками, в ином случае выдаст ошибку если на удаленном нет ветки с соответствующим именем. После этого можно просто использовать находясь на данной ветке, просто **git push**

**git pull origin main** получить изменения с удаленного репозитория origin и его ветки main

# указана последовательность действий:

# создана директория проекта, мы в ней

git init # создаём репозиторий в этой директории

touch readme.md # создаем файл readme.md

git add readme.md # добавляем файл в индекс

git commit -m "Старт" # создаем коммит

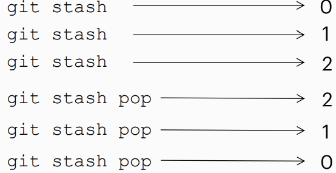
git remote add origin https://github.com:nicothin/test.git # добавляем предварительно созданный пустой удаленный репозиторий

git push -u origin main # отправляем данные из локального репозитория в удаленный (в ветку master)

**Слияние веток**

**git merge new-branch** вливаем ветку new-branch в текущую ветку, в которой находимся

**Откладывание изменений**



git stash отложить пак изменений

git stash list просмотреть список отложенных паков изменений

git stash pop вернуть последний пак изменений

# Отмена изменений

## Не или из стейджа

**git diff filename** просмотреть сделанные изменения. Между файлом и стейджом

**git diff** просмотреть изменения всех файлов. Между файлом и стейджом

**git restore filename** безвозвратноеудаление незакомиченных изменений в конкретном файле

**git restore –staged** безвозвратное удаление не закомиченных изменений в конкретном файле из стейджа

**git reset --hard** безвозвратноудаляет все незакомиченные изменения во всех файлах, даже если они не были добавлены в стейдж

**git clean -f** удаляет все новые неотслеживаемые незакомиченные изменения

**git rm --cached**

## Отмена закомиченных изменений

**git checkout commit\_hash filename** возвращает файл в состояние на указанный коммит и сразу добавляется в стейдж…. т.е восстановить в рабочей директории указанный файл на момент указанного коммита (и добавить это изменение в индекс) (git reset index.html для удаления из индекса, но сохранения изменений в файле. git restore –staged для удаления изменений и удаление из индекса )

**git revert commit\_name** отменяет указанный коммит, при этом все последующие коммиты остаются без изменений

**git revert --no-commit commit\_name** можно отменить несколько коммитов подряд, не делая каждый раз коммит и сделав единый коммит в конце

## Сброс коммитов

**git reset --soft commit\_name** если указываем предпоследний коммит, то мы сбросим изменения последнего коммита и отправим их в индекс

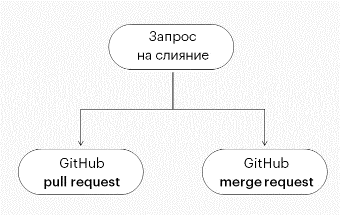
**git reset –mixed** то же что и мягкий режим, но не добавляем в индекc Можно просто писать **git reset** без этого параметра, т.к. он по умолчанию идет

**git reset HEAD~1** сбрасывает один последний коммит. Если другая цифра, то соотвественно

**git reset –hard** удаляет все изменения и коммиты позднее указанного коммита

**git commit --amend -m** редактирование комментария последнего коммита

**git commit --amend --no-edit** редактирование файлов последнего коммита



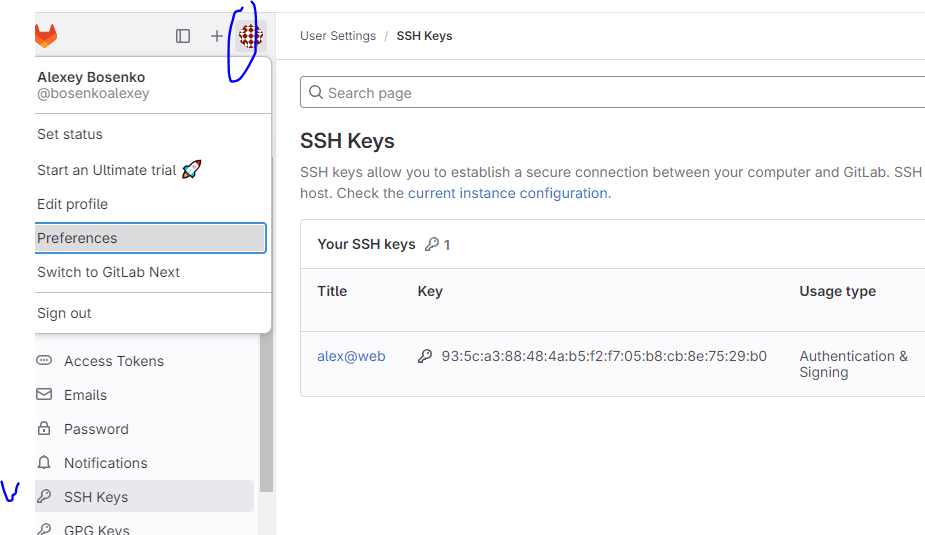
**git mv filename1 filename2** переименование файла

Начинаем работать с того что забираем главный проект к себе предварительно настроив ssh соединение по ключу

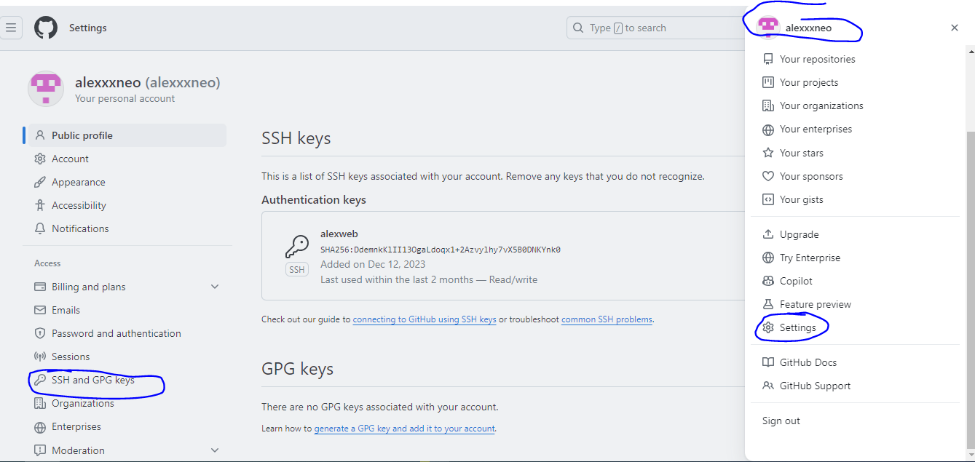
**keygen-ssh**  везде жмем энтер и пропускаем все. Находимся в домашней директории

id\_rsa.pub публичный ключ отправляем на сервер

для гитлаба:



для гитхаба



DEVOPS

# Получение и доставка кода

Предварительно имеем главный репозиторий на гитлабе. И такой же репозиторий на гитхабе но с изменениями от аутсорсеров

1. Забирать новый код от аутсорсеров
   1. **git clone** [**git@gitlab.com:sambia1/site.git**](mailto:git@gitlab.com:sambia1/site.git)Клонируем главный репозиторий и переходим в эту папку
   2. **git remote add out** [**git@github.com:alexxxneo/glmain.git**](mailto:git@github.com:alexxxneo/glmain.git)подключаем второй удаленный репозиторий с именем **out,** которое будет служить указателем на этот репозиторий
   3. **git remote -v** Проверим список удаленных репозиториев
   4. **git branch -a** посмотреть список доступных веток, включая в удаленных репозиториях, помечены красным
   5. **git fetch out** получаем коммиты от аутсорсеров с гитхаба
2. Доставлять его к нам в develop-ветку
   1. **git branch develop out/main** Создаем ветку **develop**, которая ссылается на источник кода **out** и веткой **main** 
   2. **git checkout develop** переключаемся на эту ветку, чтобы далее с ней работать
   3. **git pull** получаем все изменения кода в нашу **develop** ветку из удаленного репозитория **out**
3. Создавать merge-реквест и отправляем полученные от аутсорсеров изменения в наш репозиторий на гитлабе
   1. **git push -u origin -o merge\_request.create** Отправляем изменения в удаленный репозиторий origin.
      1. Ключ **-u** указывает на то что нужно автоматически создать в удаленном репозитории такие же ветки как на локальном.
      2. Ключ **-o** указывает на то что создаем merge request
4. Автоматизируем
   1. Создаем bash скрипт

**#!/bin/bash**

**cd ../projects/glmain**

**git checkout develop**

**git pull**

**git push -u origin -o merge\_request.create**

**cd ..**

* 1. **chmod +x autogit.sh** Выдаем права на выполнение

1. Запланируем выполнение скрипта каждые 10 мин
   1. crontab -e **минута час день месяц день\_недели /путь/к/исполняемому/файлу**и обязательно **enter**

## **git remote -v** Проверим список удаленных репозиториев

## Нужны автоматические проверки

## модульные тесты пишут разработчики

## статический анализ кода выполняют специальные программы

## существует много видов тестирования, у каждого своя задача

## Тестирование кода Python

## **pytest** - модульные тесты

## **flake8** и **pylint** - статический анализ

## **mypy** - проверка типов переменных Тестирование кода Python

## экономят время на ненужную доставку кода

## помогают программисту в поиске ошибок

## Непрерывная интеграция

## CI - Continuous Integration

## **написание кода**

## **слияние**

## **сборка и тестирование**

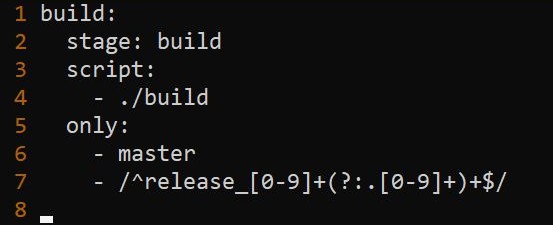
## **отчеты**

## **выпуск новой версии**

## Триггерные события

* **Автоматизировали** запуск при получении нового кода
* Запуск при **мердж-реквесте**
* Запуск в **специальной ветке или тэге**
* **Ручной запуск**

## Триггер на запуск в нужных ветках

****

  - /^release\_[0-9]+(?:.[0-9]+)+$/

Пример .gitlab-ci.yml

stages:

  - test

  - build

  - test2

  - dev

  - stage

  - prod

unit test:

    stage: test

    script:

        - echo "test"

linter:

    stage: test

    script:

        - echo "linter test"

type test:

    stage: test

    script:

        - echo "type test"

build:

    stage: build

    script:

        - echo 'run build ./deploy.sh'

    when: manual

test-after-build:

    stage: test2

    script:

        - echo "test 2 after build"

deploy dev:

    stage: dev

    script:

        - echo 'run dev ./deploy.sh'

    when: manual

    only:

        refs:

            - tags

            - main

            - /^release.\*$/ выполняем только в релизных ветках

deploy stage:

    stage: stage

    script:

        - echo 'run stage ./deploy.sh'

    when: manual выполняем в ручном режиме

    only:

        refs:

            - tags

            - main

            - /^release.\*$/

deploy prod:

    stage: prod

    script:

        - echo 'run prod ./deploy.sh'

    when: manual

    only:

        refs: выполняем только в главной мастер ветке и в тагах

            - tags

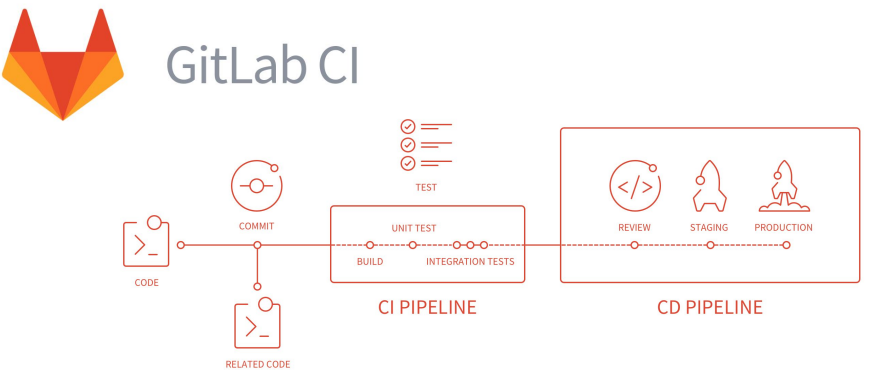
            - main

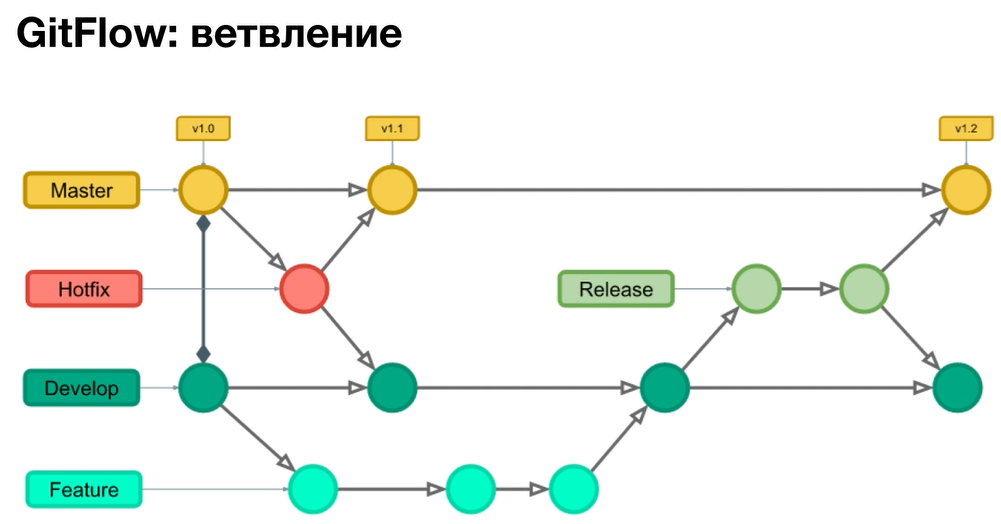
Теперь нам надо построить релизный бранч

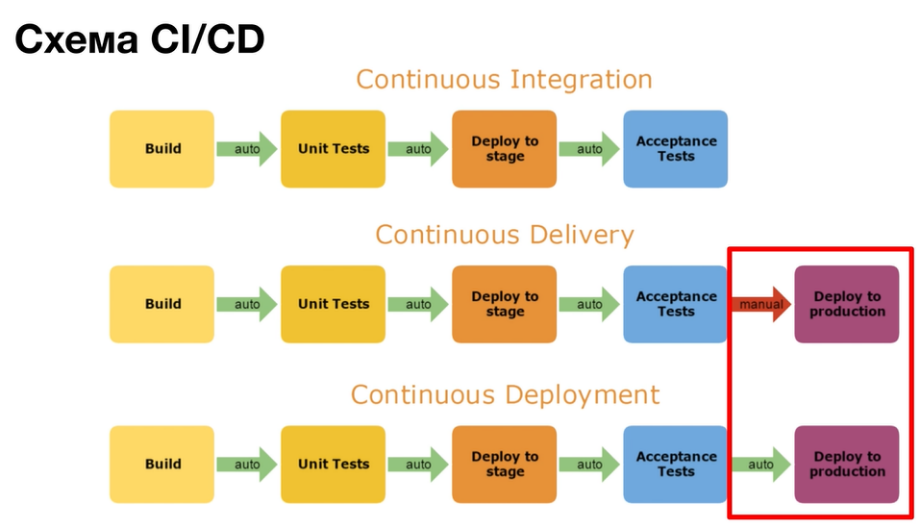
## Непрерывное развертывание

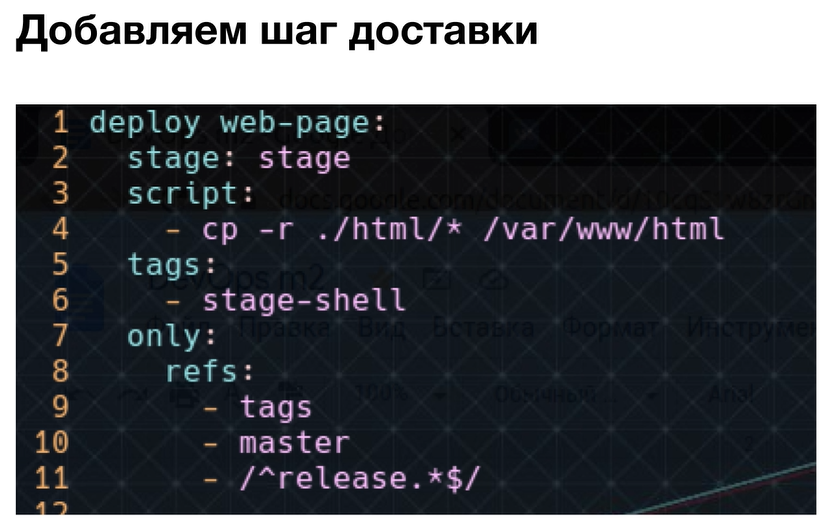
## CD - continuous delivery и continuous deployment

* **написание кода**
* **слияние**
* **сборка и тестирование**
* **выпуск новой версии**
* **автоматическое развертывание в тестовой среде**
* **ручное/автоматическое развертывание в продакшене**









Раннеры

<https://habr.com/ru/companies/flant/articles/795475/>

1. Устанавливаем гитлаб раннеры на собственный сервер. Настройи проекта, ci-cd. Выбираем Runners. ИнструкцияКонфигурация раннера
2. Для выполнения каких-либо заданий в рамках пайплайна CI/CD GitLab’у требуется раннер, на котором он будет выполнять работу. Это такая утилита, устанавливаемая на машину, которая связывается с GitLab и позволяет выполнять на этой машине задачи, указанные в пайплайне CI/CD. Раннер может быть развернут как на ВМ с Linux, macOS, FreeBSD или Windows на борту, так и в Docker-контейнере или даже в самом кластере Kubernetes.

# Gitlab runners

## Установка сервиса gitlab runners

Берем инструкции с гитлаба в разделе ci/cd, runners, expend и открываем инструкции

# Download the binary for your system

sudo curl -L --output /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64

# Give it permission to execute

sudo chmod +x /usr/local/bin/gitlab-runner

# Create a GitLab Runner user

sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash

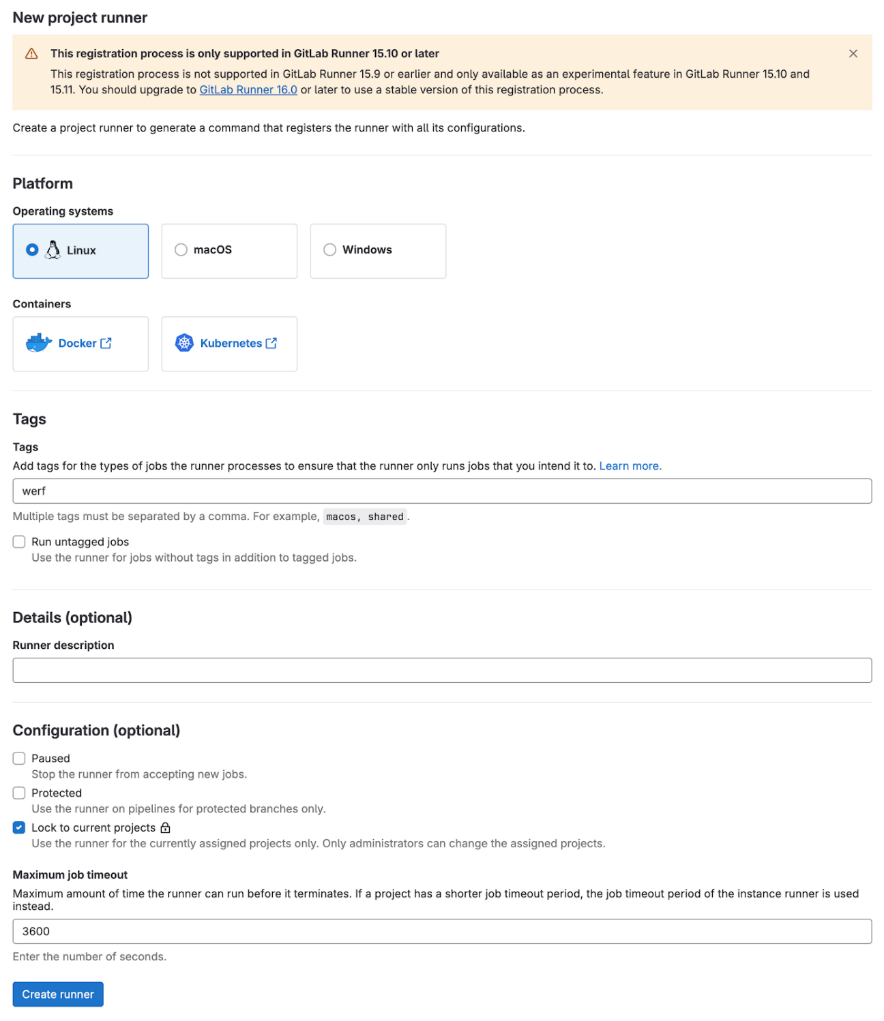
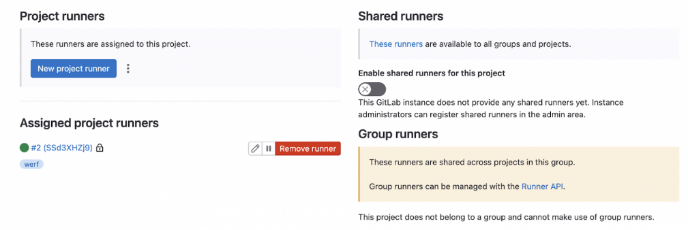
# Install and run as a service

sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner

sudo gitlab-runner start

## Регистрация раннера.

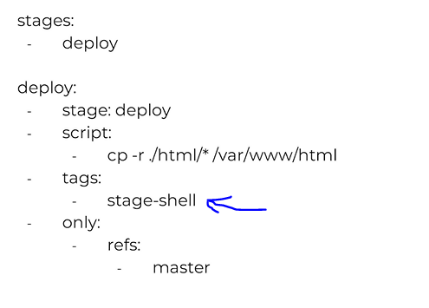
1. Создаем раннер на странице гитлаба по инструкции



На следующей странице выберем параметры нашего раннера: ОС Linux, [тег](https://ru.werf.io/guides/java_springboot/400_ci_cd_workflow/030_gitlab_ci_cd.html" \l "настройка-runner)*«stage-shell»*, галочка «Lock to current projects» (это сделает его закрытым и доступным только для этого проекта) и время таймаута джобы 3600 секунд (1 час). Задать можно любое время в зависимости от степени сложности задачи.

**Указываем tag раннера. Тэги говорят пайплайну на какой раннер ему отправить задачу**

**Выбираем executor из доступных shell, говорит о том что команды будут выполняться на stage сервере в виде команды линукса**

1. sudo gitlab-runner register --url https://gitlab.com/ --registration-token GR1348941fHMyCH4aJP4zM2NoFznn
2. Вводим адрес **https://gitlab.com/**
3. Вводим токен, к примеру **GR1348941fHMyCH4aJP4zM2NoFznn**
4. Вводим имя раннера. К примеру **stage-shell**, явно указывающий что команды будут выполняться на стейдж сервере
5. gitlab-runner run
6. Проверяем чтобы на странице со списком раннеров наш был подсвечен зеленым
7. Вводим тэги. Тэги говорят пайплайну на какой раннер ему отправить задачу. К примеру задим тег такой же как имя **stage-shell**
8. Вводим executor - указать к примеру shell, который сразу выполняет инструкции в командной строке сервера.

Проверяем работу раннера в гитлабеб -.

Если ошибка:

ERROR: Job failed: prepare environment: exit status 1. Check https://docs.gitlab.com/runner/shells/index.html#shell-profile-loading for more information

То в **файле /home/gitlab-runner/.bash\_logout** в домашнем каталоге пользователя под которым работает раннер проверяем отсутствуют или закомментированы следующие строки должны быть:

**if [ "$SHLVL" = 1 ]; then**

**[ -x /usr/bin/clear\_console ] && /usr/bin/clear\_console -q**

**fi**

Самое время доставлять на него приложения через пайплайны

Устанавливаем на наш сервер инструменты

* Устанавливаем nginx
* Устанавливаем гитлаб раннеры

gitlab-ci приходит к следующему виду

stages:

- test

- build

- test2

- dev

- stage

- prod

unit test:

stage: test

script:

- echo "test"

linter:

stage: test

script:

- echo "linter test"

type test:

stage: test

script:

- echo "type test"

build:

stage: build

script:

- echo 'run build ./deploy.sh'

when: manual

test-after-build:

stage: test2

script:

- echo "test 2 after build"

deploy website:

stage: stage

script:

- cp -rf ./html/\* /var/www/wordpress/

- rm -f html.7z; 7z a /var/www/wordpress/website.zip /var/www/wordpress/

when: manual

tags:

- stage-shell

only:

refs:

- tags

- main

- /^release.\*$/

deploy prod:

stage: prod

script:

- echo 'run prod ./deploy.sh'

when: manual

only:

refs:

- tags

- main

# Yarn установка

Установка

[https://classic.yarnpkg.com/en/docs/install#debian-stable](https://classic.yarnpkg.com/en/docs/install%23debian-stable)

**curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -**

**echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/sources.list.d/yarn.list**

**sudo apt update && sudo apt install yarn**

If using nvm you can avoid the node installation by doing:

**sudo apt update && sudo apt install --no-install-recommends yarn**

# **Nodejs установка**

**sudo apt purge nodejs**

**sudo apt install build-essential checkinstall**

**sudo apt install libssl-dev**

**wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash**

открываем новый терминал

**nvm ls-remote** Затем смотрим список доступных версий Node js: **nvm install 18.20.3**

**nvm list** список установленных версий

Создание приложения

1. yarn global add create-react-app
2. ~/.yarn/bin/create-react-app test-react-app
3. Заливаем его на гитлаб.
   1. Согдаем новый проект на гитлабе
   2. git remote add origin [git@gitlab.com:sambia1/test-app.git](mailto:git@gitlab.com:sambia1/test-app.git)
   3. git push --set-upstream origin --all

Добавляем приложение

yarn global add create-react-app

~/.yarn/bin/create-react-app test-react-app

переходим в приложение и

yarn start

Создаем в гитлабе новый репозиторий пустой и привязываем наш локальный репозтиторий

git remote add origin git@gitlab.com:sambia1/test-create-app.git

docker run -d –name gitlab-runner –restart always \

# DOCKER ТЕОРИЯ

## Из уроков

### Docker установка docker engine

<https://docs.docker.com/engine/install/ubuntu/>

помимо команд из инструкции чтобы не было ошибки с правами или не запускать постоянно от sudo еще нужно

sudo groupadd docker

sudo usermod -aG docker $USER

перезагружаем систему

### Основные команды Docker

docker run -d -p 80:80 docker/getting-started

* run запуск контейнера
* -d в фоновом режиме
* - p 80:80 порт внутри и снаружи контейнера

### Манипуляции с Docker-контейнером

● docker images — смотрим images в наличии и узнаём ID нашего

● docker images -a — смотрим images, в том числе не активные

● docker ps — смотрим запущенные контейнеры

● docker ps -a — смотрим список всех контейнеров, включая неактивные

● docker login username darkbenladan --password password — логинимся по логину и паролю, которые мы создали на hub.docker.com ранее

● docker save 3c156928aeec > start.tar — сохраняем локально наш контейнер, теперь мы можем его перенести куда угодно

● docker rm -f $(docker ps -a -q) — удалим и принудительно остановим все контейнеры

● docker load < start.tar — мы вернули наш контейнер

### Ещё немного полезных команд

● docker run -d -p 80:80 docker/getting-started

● docker ps — узнаем новый ID нашего контейнера

● docker port 8c71b40dc003 — узнаем порт приложения

● docker exec -it 8c71b40dc003 sh — зайдём в терминал нашего контейнера

● / # cat /etc/os-release — узнаем какая OS внутри нашего контейнера NAME="Alpine Linux"

ID=alpine VERSION\_ID=3.11.5

PRETTY\_NAME="Alpine Linux v3.11" HOME\_URL="https://alpinelinux.org/" BUG\_REPORT\_URL="https://bugs.alpinelinux.org/"

● exit — разлогиниться из контейнера

● Так же могут быть полезны команды docker logs, docker stop\start, docker logs

# Шпаргалка docker

## docker run флаги

**-d (Run container in background and print container ID)**

Запуск контейнера в фоновом режиме.

docker run -d nginx

**-p (Publish a container’s port(s) to the host)**

Публикация портов контейнера на хосте.

docker run -d -p 8080:80 nginx

**--name (Assign a name to the container)**

Назначение имени контейнеру.

docker run -d --name my\_nginx nginx

**-v (Bind mount a volume)**

Монтирование тома.

docker run -d -v /my/local/data:/data nginx

**-e (Set environment variables)**

Установка переменных окружения.

docker run -d -e MY\_ENV\_VAR=my\_value nginx

**--rm (Automatically remove the container when it exits)**

Автоматическое удаление контейнера после завершения работы.

docker run --rm ubuntu echo "This container will be removed after it exits"

**-it (Interactive mode with a pseudo-TTY)**

Интерактивный режим с псевдо-TTY.

docker run -it ubuntu /bin/bash

**--network (Connect a container to a network)**

Подключение контейнера к сети.

docker run -d --network my\_network nginx

**--restart (Restart policy to apply when a container exits)**

Политика перезапуска контейнера.

docker run -d --restart always nginx

**--cpus (Limit CPU usage)**

Ограничение использования процессора.

docker run -d --cpus="1.5" nginx

**--memory (Memory limit)**

Ограничение использования памяти.

docker run -d --memory="500m" nginx

**--link (Add link to another container)**

Добавление ссылки на другой контейнер.

docker run -d --link another\_container nginx

## Флаги для команды **docker build**

**-t или --tag**: Задает имя и тег для создаваемого образа.

docker build -t my-image:latest .

Этот флаг присваивает образу имя **my-image** и тег **latest**.

**-f или --file**: Указывает путь к Dockerfile.

docker build -f /path/to/Dockerfile .

В этом примере используется Dockerfile, расположенный по пути **/path/to/Dockerfile**.

**--build-arg**: Передает аргументы сборки в Dockerfile.

docker build --build-arg VERSION=1.0 .

Этот флаг передает аргумент **VERSION** со значением **1.0** в Dockerfile, где его можно использовать с помощью директивы **ARG**.

**--no-cache**: Отключает кэширование промежуточных слоев.

docker build --no-cache -t my-image:latest .

Эта команда выполняет сборку без использования кэша, что может быть полезно для обеспечения того, что все слои будут пересобраны.

**--pull**: Обновляет базовые образы перед сборкой.

docker build --pull -t my-image:latest .

Эта команда заставляет Docker всегда загружать новейшие версии базовых образов.

**--target**: Указывает целевой этап в многоэтапной сборке.

docker build --target builder -t my-image:builder .

В этом примере сборка завершится на этапе **builder**, указанном в Dockerfile.

**--rm**: Удаляет временные контейнеры после успешной сборки (по умолчанию включено, можно отключить с **--rm=false**).

docker build --rm=false -t my-image:latest .

В этом случае временные контейнеры не будут удаляться после сборки.

**--force-rm**: Удаляет промежуточные контейнеры даже при неудачной сборке.

docker build --force-rm -t my-image:latest .

Этот флаг гарантирует, что промежуточные контейнеры будут удалены, даже если сборка завершится с ошибкой.

**--compress**: Сжимает контекст сборки, передаваемый на Docker-демон.

docker build --compress -t my-image:latest .

Сжимает контекст сборки, что может уменьшить объем передаваемых данных.

**--label**: Добавляет метаданные к образу.

docker build --label maintainer="me@example.com" -t my-image:latest .

Добавляет метку **maintainer** с указанием контактного email.

Примеры команд, комбинирующие несколько флагов:

docker build -t my-image:latest -f /path/to/Dockerfile --build-arg VERSION=1.0 --no-cache .

Эта команда задает имя и тег образу, указывает Dockerfile, передает аргумент сборки и отключает кэширование.

## Образы

**docker search alpine**  поиск образа

d**ocker pull alpine** загрузка образа

**docker image list**  список образов

**docker images** список образов

**docker images –a** список образов включая неактивные

**docker run -it --name name\_container 05455a08881e** запускаем контейнер интерактивно, придумав ему имя и указав его id из списка образов. –it значит интерактивно

**docker image inspect ubuntu** подробности об образе

**docker pull ubuntu:20.04**  загрузка образа с нужным тегом

**docker tag ubuntu:20.04 newname:newtag** переименовать образ и тэг

**docker rm newname:newtag** возвращение к оригинальному имени и тэгу образа без удаления оригинального образа

**docker save -o ubuntu:20.04.tar ubuntu:20.04**  сохраняем образ в архив

**docker rmi ubuntu:20.04** удалим образ

**docker load -i ubuntu:20.04.ta**r загрузим образ из архива

В чем разница способов

* **save/load** выгружают и загружают образы контейнеров, которые будут использоваться для создания контейнеров;
* **export** выгружает файловую систему созданного контейнера (не образ контейнера, а изменения, которые внес контейнер в образ);
* **import** позволяет создать образ тома файловой системы из архива, созданного с помощью **export**, который может использоваться, например, при запуске контейнера с помощью **—volumes-from**.

В целом, разница заключается, что save/load относятся к данным образов, а export/import к данным контейнеров.

## Контейнеры

**docker run -it ubuntu** запуск контейнера убунту в интерактивном режиме.

**docker container run -it ubuntu** запуск нового экземпляра контейнера убунту

**docker ps** или **docker container list**  список запущенных контейнеров

**docker stop <container\_id>** остановка запущенного контейнера

**docker rm id\_container**  удаление контейнера

**docker container rename c84b0d053103 alexcontainer**  переименование контейнера

**docker container commit -m "added alex.txt" alexcontainer aleximage:alextag** сохранение контейнера в образ

## **Установка Docker**

curl -fsSL https://get.docker.com -o get-docker.sh

sudo sh get-docker.sh

## **Информация и регистрация**

* **docker info** - Информация обо всём в установленном Docker
* **docker history** - История образа
* **docker tag** - Дать тег образу локально или в registry
* **docker login** - Залогиниться в registry
* **docker search** - Поиск образа в registry
* **docker pull** - Загрузить образ из Registry себе на хост
* **docker push** - Отправить локальный образ в registry

## **Управление контейнерами (Container Management)**

* **docker ps -а** - Посмотреть все контейнеры
* **docker start container-name** - Запустить контейнер
* **docker kill/stop container-name** - Убить (SIGKILL) /Остановить (SIGTERM) контейнер
* **docker logs --tail 100 container-name** - Вывести логи контейнера, последние 100 строк
* **docker inspect container-name** - Вся инфа о контейнере + IP
* **docker rm container-name**- Удалить контейнер (поле каждой сборки Dockerfile)
* **docker rm -f $(docker ps -aq)** - Удалить все запущенные и остановленные контейнеры
* **docker events container-name** - Получения событий с контейнера в реальном времени.
* **docker port container-name** - Показать публичный порт контейнера
* **docker top container-name** - Отобразить процессы в контейнере
* **docker stats container-name** - Статистика использования ресурсов в контейнере
* **docker diff container-name** - Изменения в ФС контейнера

## **Управление образами (Images Management)**

* **docker build -t my\_app .** - Билд контейнера в текущей папке, Скачивает все слои для запуска образа
* **docker images / docker image ls** - Показать все образы в системе
* **docker image rm / docker rmi image** - Удалить image
* **docker commit lepkov/debian11slim:version3** - Создает образ из контейнера
* **docker insert URL** - вставляет файл из URL в контейнер
* **docker save -o backup.tar** - Сохранить образ в backup.tar в STDOUT с тегами, версиями, слоями
* **docker load -i backup.tar** - Загрузить образ в .tar в STDIN с тегами, версиями, слоями
* **docker import** - Создать образ из .tar
* **docker image history --no-trunc** - Посмотреть историю слоёв образа
* **docker system prune -f** - Удалит все, кроме используемого (лучше не использовать на проде, ещё кстати из-за старого кеша может собираться cтарая версия контейнера)

### **Отличия Save, Load от Export, Import[¶](https://devopsgu.ru/guides/docker-cheatsheet/" \l "save-load-export-import)**

Docker поддерживает команды export и import, которые часто могут быть перепутаны с save, load. Ключевое отличие данных команд в том, для чего они предназначены:

* **save/load** выгружают и загружают образы контейнеров, которые будут использоваться для создания контейнеров;
* **export** выгружает файловую систему созданного контейнера (не образ контейнера, а изменения, которые внес контейнер в образ);
* **import** позволяет создать образ тома файловой системы из архива, созданного с помощью export, который может использоваться, например, при запуске контейнера с помощью —volumes-from.

В целом, разница заключается, что save/load относятся к данным образов, а export/import к данным контейнеров.

## **Запуск docker (Run)[¶](https://devopsgu.ru/guides/docker-cheatsheet/" \l "docker-run)**

* **docker run -d -p 80:80 -p 22:22 debian:11.1-slim sleep infinity** (*--rm* удалит после закрытия контейнера, *--restart unless-stopped* добавит автозапуск контейнера) - Запуск контейнера интерактивно или как демона/detached (-d), Порты: слева хостовая система, справа в контейнере, пробрасывается сразу 2 порта 80 и 22, используется легкий образ Debian 11 и команда бесконечный сон
* **docker update --restart unless-stopped redis** - добавит к контейнеру правило перезапускаться при закрытии, за исключением команды стоп, автозапуск по-сути
* **docker exec -it container-name /bin/bash** (ash для alpine) - Интерактивно подключиться к контейнеру для управления, exit чтобы выйти
* **docker attach container-name** - Подключиться к контейнеру чтоб мониторить ошибки логи в реалтайме

## **Хранилище (Volumes)[¶](https://devopsgu.ru/guides/docker-cheatsheet/" \l "volumes)**

Скопировать в корень контейнера file

docker cp file <containerID>:/

Скопировать file из корня контейнера в текущую директорию командной строки

docker cp <containerID>:/file .

Создать volume для постоянного хранения файлов

docker volume create todo-db

Добавить named volumу todo-db к контейнеру (они ok когда мы не заморачиваемся где конкретно хранить данные)

docker run -dp 3000:3000 --name=dev -v todo-db:/etc/todos container-name

docker run -dp 3000:3000 --name=dev --mount source=todo-db,target=/etc/todos container-name

# тоже самое что команда сверху

Отобразить список всех volume’ов

docker volume ls

Инспекция volume’ов

docker volume inspect

Удалить volume

docker volume rm

## **Сеть (Network)**

## Команды работы с сетью

Основные команды для работы с сетью

1. Создание сети:

docker network create [OPTIONS] NETWORK

docker network create my-network

2. Список сетей:

docker network ls

3. Удаление сети:

docker network rm NETWORK

**docker network rm my-network**

4. Информация о сети:

docker network inspect NETWORK

**docker network inspect my-network**

5. Подключение контейнера к сети:

docker network connect [OPTIONS] NETWORK CONTAINER

**docker network connect my-network my-container**

6. Отключение контейнера от сети:

docker network disconnect [OPTIONS] NETWORK CONTAINER

**docker network disconnect my-network my-container**

Флаги и опции для создания сети

1. --driver: Задает драйвер для сети (по умолчанию bridge).

docker network create --driver bridge my-bridge-network

2. --subnet: Указывает подсеть в формате CIDR.

docker network create --subnet 192.168.1.0/24 my-subnet-network

3. --gateway: Указывает шлюз для сети.

docker network create --subnet 192.168.1.0/24 --gateway 192.168.1.1 my-gateway-network

4. --ip-range: Указывает диапазон IP-адресов для выделения контейнерам.

docker network create --subnet 192.168.1.0/24 --ip-range 192.168.1.128/25 my-ip-range-network

5. --ipam-driver: Указывает драйвер IPAM для управления IP-адресами.

docker network create --ipam-driver default my-ipam-network

6. --ipam-opt: Передает дополнительные опции драйверу IPAM.

docker network create --ipam-driver default --ipam-opt foo=bar my-ipam-option-network

1. Создание моста (bridge) сети:

docker network create --driver bridge my-bridge-network

2. Создание overlay сети для использования в Swarm:

docker network create --driver overlay my-overlay-network

3. Создание сети с указанной подсетью и шлюзом:

docker network create --subnet 192.168.10.0/24 --gateway 192.168.10.1 my-custom-network

4. Подключение контейнера к сети:

docker run -d --name my-container --network my-network nginx

5. Подключение существующего контейнера к сети:

docker network connect my-network my-container

6. Отключение контейнера от сети:

docker network disconnect my-network my-container

## Драйвера сети

**Bridge (мост)**

* **Когда использовать:**
  + Для стандартной изолированной сети, создаваемой по умолчанию для контейнеров на одном хосте.
  + Хорошо подходит для простых приложений, где все компоненты работают на одном Docker-хосте.
* **Преимущества:**
  + Легкость настройки и использования.
  + Контейнеры могут общаться друг с другом на одном хосте через IP-адреса или имена контейнеров.
* **Пример использования:**

docker network create --driver bridge my-bridge-network

**Host (хост)**

* **Когда использовать:**
  + Для приложений, которым требуется высокая производительность и минимальная задержка сети.
  + Подходит для сетевых служб, которые должны использовать сетевые интерфейсы хоста напрямую.
* **Преимущества:**
  + Меньшая задержка, так как контейнеры используют сеть хоста напрямую.
  + Полный доступ ко всем интерфейсам хоста.
* **Ограничения:**
  + Меньшая изоляция между контейнерами.
  + Портовые конфликты, так как контейнеры используют те же порты, что и хост.

docker run --network host my-container

**Overlay (наложенная сеть)**

* **Когда использовать:**
  + Для соединения контейнеров, работающих на разных Docker-хостах в кластере Swarm.
  + Полезно для распределенных приложений и микросервисов.
* **Преимущества:**
  + Соединяет контейнеры на разных хостах.
  + Поддерживает скалируемость и распределение нагрузки.
* **Ограничения:**
  + Требует настройки и управления кластером Swarm или Kubernetes.
  + Более сложная настройка и поддержка.
* **Пример использования:**

docker network create --driver overlay my-overlay-network

**Macvlan**

* **Когда использовать:**
  + Для контейнеров, которым нужны свои собственные MAC-адреса и доступ к локальной сети хоста.
  + Подходит для интеграции с физическими сетевыми устройствами или для случаев, когда требуется, чтобы контейнеры имели свои собственные IP-адреса на той же сети, что и хост.
* **Преимущества:**
  + Каждый контейнер получает уникальный MAC-адрес.
  + Контейнеры могут выглядеть как физические устройства в локальной сети.
* **Ограничения:**
  + Более сложная настройка.
  + Ограниченная поддержка в некоторых сетевых инфраструктурах.
* **Пример использования:**

docker network create -d macvlan \ --subnet=192.168.1.0/24 \ --gateway=192.168.1.1 \ -o parent=eth0 my-macvlan-network

**None**

* **Когда использовать:**
  + Когда требуется полная изоляция контейнера от любой сети.
  + Полезно для выполнения задач, не требующих сетевого взаимодействия.
* **Преимущества:**
  + Полная изоляция от сети.
  + Максимальная безопасность.
* **Ограничения:**
  + Нет сетевого взаимодействия с другими контейнерами или хостом.
* **Пример использования:**

docker run --network none my-container

Создать сеть

docker network create todo-app

Удалить сеть

docker network rm

Отразить все сеть

docker network ls

Вся информация о сети

docker network inspect

Соединиться с сетью

docker network connect

Отсоединиться от сети

docker network disconnect

Пробросить текущую папку в контейнер и работать на хосте, -w working dir, sh shell

docker run -dp 3000:3000 \

-w /app -v "$(pwd):/app" \

node:12-alpine \

sh -c "yarn install && yarn run dev"

Запуск контейнера с присоединением к сети и заведение переменных окружения

docker run -d \

--network todo-app --network-alias mysql \ # (алиас потом сможет резолвить докер для других контейнеров)

-v todo-mysql-data:/var/lib/mysql \ # (автоматом создает named volume)

-e MYSQL\_ROOT\_PASSWORD=secret \ # (в проде нельзя использовать, небезопасно)

-e MYSQL\_DATABASE=todos \ # (в проде юзают файлы внутри конейнера с логинами паролями)

mysql:5.7

Запуск контейнера с приложением

docker run -dp 3000:3000 \

-w /app -v "$(pwd):/app" \

--network todo-app \

-e MYSQL\_HOST=mysql \

-e MYSQL\_USER=root \

-e MYSQL\_PASSWORD=secret \

-e MYSQL\_DB=todos \

node:12-alpine \

sh -c "yarn install && yarn run dev"

## DOCKER CREATE

Команда **docker create** используется для создания нового контейнера, но в отличие от **docker run**, она не запускает контейнер сразу после создания. Это позволяет настроить контейнер до его запуска. Вот основные аспекты использования команды **docker create**:

Основной синтаксис

**docker create [OPTIONS] IMAGE [COMMAND] [ARG...]**

IMAGE: Это имя или идентификатор образа Docker, который будет использоваться для создания контейнера.

[COMMAND] [ARG...]: (Необязательно) Команда и аргументы, которые будут выполнены внутри контейнера при его запуске.

Примеры использования

1. Создание контейнера из образа Nginx:

**docker create nginx**

Этот пример создаёт контейнер из образа nginx, но не запускает его.

1. Создание контейнера с именем и пробросом порта:

**docker create --name my-nginx -p 8080:80 nginx**

Здесь создается контейнер с именем my-nginx и пробросом порта 8080 на хосте к порту 80 в контейнере.

1. Создание контейнера с томом и переменными окружения:

**docker create --name my-app -v /my/local/data:/data -e "ENV=production" my-app-image**

Этот пример создаёт контейнер с именем my-app, монтирует локальную директорию /my/local/data в контейнер по пути /data и устанавливает переменную окружения ENV в значение production.

Основные опции

**--name**: Задаёт имя контейнера. **docker create --name my-container my-image**

**-p, --publish**: Публикует порты контейнера на хосте. **docker create -p 8080:80 my-image**

**-v, --volume**: Монтирует томы или директории в контейнер. **docker create -v /host/path:/container/path my-image**

**-e, --env**: Устанавливает переменные окружения. **docker create -e "ENV=production" my-image**

**--network**: Подключает контейнер к указанной сети. **docker create --network my-network my-image**

**--restart**: Устанавливает политику перезапуска контейнера. **docker create --restart unless-stopped my-image**

**-d, --detach**: Создает контейнер в фоновом режиме. **docker create -d my-image**

Дополнительные возможности

1. Изменение параметров контейнера перед запуском:

После создания контейнера можно использовать команды **docker container update** и **docker container rename** для изменения его параметров до запуска.

1. Запуск созданного контейнера:

Используйте **docker start** для запуска контейнера: **docker start my-container**

1. Просмотр списка созданных, но не запущенных контейнеров:

**docker container ls -a**

Команда **docker create** полезна для случаев, когда требуется создать контейнер с предварительной настройкой и не запускать его сразу. Это позволяет настраивать контейнер перед его запуском, что может быть полезно в сложных сценариях развертывания.

**CMD VS** ENTRYPOINT

Разница в том, что CMD выполняется из под /bin/sh по дефолту, а ENTRYPOINT без него. В случае с CMD, команда и параметры к ней захардкожены в образ, пример запуска с переопределением команды CMD ["sleep","10"]

docker run ubuntu sleep 5

Контейнер проспит 5 секунд вместо 10.

В случае с ENTRYPOINT, только команда захардкожена в образ, пример запуска с переопределением команды ENTRYPOINT ["sleep"] CMD ["10"] (Есл используются обе директивы, то в энтрипоинте команда, а в cmd параметры к ней)

docker run ubuntu 5

Контейнер проспит 5 секунд вместо 10.

Чтобы переопределить ENTRYPOINT:

docker run --entrypoint another-command ubuntu 20

Обычно практика такая, всегда используй CMD, если только не требуется каждый раз запускать контейнер с разным параметром (экономия времени, чтоб каждый раз не вводить строчку с командой)

### Основные отличия:

* **RUN**: Выполняется на этапе сборки образа. Создает новый слой в образе.
* **CMD**: Задает команду по умолчанию, которая будет выполнена при запуске контейнера. Может быть переопределена при запуске контейнера.
* **ENTRYPOINT**: Задает команду, которая всегда будет выполняться при запуске контейнера. Аргументы, переданные при запуске контейнера, добавляются к ENTRYPOINT.

## **Лучшая практика(Best Practice)[¶](https://devopsgu.ru/guides/docker-cheatsheet/" \l "best-practice)**

Следуй принципу минимальных привилегий, процессы в контейнере никогда не должны выполняться из под рута, кроме редких случаев, нужно добавлять команду user и менять юзера на non-root.

Не привязываться к UID, он динамичен, можно записать во временную папку UID.

Сделать все исполняемые файлы владельцем рута, чтобы никто не изменил исполняемые файлы, а пользователю достаточно только права на выполнение.

Чем меньше компонентов и открытых портов, тем меньше поверхность для атак.

Использовать multistage для промежуточного контейнера для компиляции всего, зависимостей, временных файлов, образ может весить на треть меньше.

Distroless с чистого листа, использовать минимальный набор пакетов, например избавиться от образа Ubuntuи выбрать Debian-base, наши контейнеры содержат уязвимости изначального образа, чекать это.

Нужно обновлять всё до того, как выйдет из под поддержки.

Оставлять только те порты, которые реально нужны, избегать 22 и 21 3389 (ssh & ftp & rdp).

Никогда не помещайте логины/пароли в команде, в докерфайлах, переменных, docker secret или любой другой менеджер секретов ok.

Не использовать ADD, только COPY (когда используем точку - это воркдир где лежит докерфайл).

При сборке используйте .dockerignore чтобы убрать сенситив дату, это как .gitignore.

При сборке вначале команд лучше кешировать команду ран, а потом скопировать исходные данные.

Метадату записать.

Использовать тесты типа Linter и сканеры образов для CI.

Время от времени делать prune, докер любит много места жрать

### Сохранение образа в архив

Для начала рассмотрим как можно выполнить сохранение образа контейнера из среды. Для выполнения данной операции используется команда docker save (man docker-save):

docker pull ubuntu:18.04

docker save -o ubuntu-1804.tar ubuntu:18.04

После выполнения данной команды образ со всеми слоями будет выгружен из пространства образов демона Docker в файловую систему в виде архива tar.

Интересно, что таким образом можно выгрузить более одного образа:

docker pull ubuntu:16.04

docker save -o ubuntu-16+18.tar ubuntu:18.04 ubuntu:16.04

Таким образом, вы можете за одну команду выгрузить все необходимые для распространения контейнеры в один tar-архив.

### Загрузка образа в среду Docker

Для выполнения загрузки ипользуется обратная операция **docker load**:

# удалим образы

docker rmi ubuntu:18.04 ubuntu:16.04

# убедимся, что нет доступных образов

docker images | grep ubuntu | wc -l

# загрузим образы из архива в среду Docker

docker load -i ubuntu-16+18.tar

aa54c2bc1229: Loading layer [==================================================>] 121.6MB/121.6MB

7dd604ffa87f: Loading layer [==================================================>] 15.87kB/15.87kB

2f0d1e8214b2: Loading layer [==================================================>] 11.78kB/11.78kB

297fd071ca2f: Loading layer [==================================================>] 3.072kB/3.072kB

Loaded image: ubuntu:16.04

Loaded image: ubuntu:18.04

# проверим доступное количество образов

docker images | grep ubuntu | wc -l

2

Таким образом вы можете распространять образы контейнеров Docker между узлами без использования реестра и даже сети, например, на DVD или накопителе USB.

### Отличия Save, Load от Export, Import

Docker поддерживает команды **export** и **import**, которые часто могут быть перепутаны с **save**, **load**. Ключевое отличие данных команд в том, для чего они предназначены:

* **save/load** выгружают и загружают образы контейнеров, которые будут использоваться для создания контейнеров;
* **export** выгружает файловую систему созданного контейнера (не образ контейнера, а изменения, которые внес контейнер в образ);
* **import** позволяет создать образ тома файловой системы из архива, созданного с помощью **export**, который может использоваться, например, при запуске контейнера с помощью **—volumes-from**.

В целом, разница заключается, что save/load относятся к данным образов, а export/import к данным контейнеров.

# ПРАКТИКА 1 Создание контейнера с nginx с помощью Dockerfile

Создадим простой Dockerfile с OS ubuntu:18.04, внутри которого будет nginx:

FROM ubuntu:18.04

COPY . /app

RUN apt-get update && apt-get upgrade -y && apt-get install -y nginx

EXPOSE 80

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

Запустим Dockerfile с ngnix

**docker build ./** собираем докерфайл

**docker images -a** смотрим что наш образ присутствует в списке всех образов.

**docker run -d -p 80:80 id\_images**  запускаем контейнер из образа на 80 внешнем и внутреннем порту. Запускаем как демон в фоновом режиме

docker ps смотрим список запущенных контейнеров

Убеждаемся что nginx работает. Переходим по ссылке локалхоста



**docker stop a96076966e8b** останавливаем запущенный контейнер

Вариант Создание контейнера с nginx из командной строки

**docker create --name nginx\_based -p 80:80 nginx:alpine**  создает образ на базе образа nginx:alpine

**docker ps -a**  проверяем список всех контейнеров, в т.ч. и незапущенных

**docker start 3f72eb5f48af**  запускаем контейнер

**docker cp index.html nginx\_based:/usr/share/nginx/html/index.html**  Создаем рядом с докерфайлом файл index.html и копируем его в созданный ранее контейнер **nginx\_based.**  Проверяем работу.

Сохраняем наш контейнер в образ

**docker commit nginx\_based my\_nginx\_image**

**docker rm**

**docker run -d -p 80:80 my\_nginx\_image** запускаем новый контейнер из нашего образа ранее созданного

Volume

В dockerfile

VOLUME /my\_volume

Создание тома **docker volume create my\_volume**

Выяснить информацию о томах -- **docker volume ls**

Исследовать конкретный том - **docker volume inspect my\_volume**

Удаление тома - **docker volume rm my\_volume**

Удалить все тома - **docker volume prune**

Очистка peсурсов **Docker - docker system prune**

**Использовать надо флаг --mount, а не --volume, который устарел**

docker container run --mount source=my\_volume, target=/container/path/for/volume my\_image запускаем докер и примаунчиваем созданный ранее том

Параметры --mount

● type — тип монтирования. Значением для соответствующего ключа могут выступать bind,

volume или tmpfs. Мы тут говорим о томах, то есть нас интересует значение volume

● source — источник монтирования. Для именованных томов — это имя тома.

Для неименованных томов этот ключ не указывают. Он может быть сокращён до src

● destination — путь, к которому файл или папка монтируется в контейнере.

Этот ключ может быть сокращён до dst или target

● readonly — монтирует том, который предназначен только для чтения. Использовать этот ключ необязательно, значение ему не назначают

Пример использования --mount с множеством параметров:

**docker run --mount type=volume,source=volume\_name,destination=/path/in/container,readonly**

**my\_image**

А теперь рассмотрим 2 примера запуска контейнера с nginx с volume с read\write

**$ docker run -d \**

**--name nginxtest \**

**--mount source=myvol2,target=/app \**

**nginx:latest**

Пример запуска контейнера с nginx с volume в read-only:

**$ docker run -d \**

**--name=nginxtest \**

**--mount source=nginx-vol,destination=/usr/share/nginx/html,readonly \**

**nginx:latest**

## php – fpm

### dockerfile классический вариант

FROM php:7.4-cli

COPY . /usr/src/myapp

WORKDIR /usr/src/myapp

CMD [ "php", "./your-script.php" ]

$ docker build -t my-php-app .

$ docker run -it --rm --name my-running-app my-php-app

### Второй вариант

$ docker run -it --rm --name my-running-script -v "$PWD":/usr/src/myapp -w /usr/src/myapp php:7.4-cli php your-script.php

Результат одинаковый что в первом что во втором варианте

### Как установить дополнительные расширения

PHP FROM php:7.4-cli

RUN docker-php-source extract \

# do important things \

&& docker-php-source delete

### Расширения ядра

PHP FROM php:7.4-fpm

RUN apt-get update && apt-get install -y \

libfreetype6-dev \

libjpeg62-turbo-dev \

libpng-dev \

&& docker-php-ext-configure gd --with-freetype --with-jpeg \

&& docker-php-ext-install -j$(nproc) gd

При такой сборке мы сразу установим все расширения, все разрешения и все сконфигурирем автоматом

Далее берем файлы из моего репозитория <https://github.com/alexxxneo/docker-php>

Разбираем dockerfile:

FROM alpine:3.11

# ADD https://dl.bintray.com/php-alpine/key/php-alpine.rsa.pub /etc/apk/keys/php-alpine.rsa.pub old link

ADD https://packages.whatwedo.ch/php-alpine.rsa.pub /etc/apk/keys/php-alpine.rsa.pub

# make sure you can use HTTPS

RUN apk --update add ca-certificates

RUN echo "https://packages.whatwedo.ch/php-alpine/v3.11/php-7.4" >> /etc/apk/repositories

# Install packages

RUN apk --no-cache add php php-fpm php-opcache php-openssl php-curl \

nginx supervisor curl

# https://github.com/codecasts/php-alpine/issues/21

RUN ln -s /usr/bin/php7 /usr/bin/php

# Configure nginx

COPY config/nginx.conf /etc/nginx/nginx.conf

# Remove default server definition

RUN rm /etc/nginx/conf.d/default.conf

# Configure PHP-FPM

COPY config/fpm-pool.conf /etc/php7/php-fpm.d/www.conf

COPY config/php.ini /etc/php7/conf.d/custom.ini

# Configure supervisord

COPY config/supervisord.conf /etc/supervisor/conf.d/supervisord.conf

# Setup document root

RUN mkdir -p /var/www/html

# Make sure files/folders needed by the processes are accessable when they run under the nobody user

RUN chown -R nobody.nobody /var/www/html && \

chown -R nobody.nobody /run && \

chown -R nobody.nobody /var/lib/nginx && \

chown -R nobody.nobody /var/log/nginx

# Switch to use a non-root user from here on

USER nobody

# Add application

WORKDIR /var/www/html

COPY --chown=nobody src/ /var/www/html/

# Expose the port nginx is reachable on

EXPOSE 8080

# Let supervisord start nginx & php-fpm

CMD ["/usr/bin/supervisord", "-c", "/etc/supervisor/conf.d/supervisord.conf"]

# Configure a healthcheck to validate that everything is up&running

HEALTHCHECK --timeout=10s CMD curl --silent --fail <http://127.0.0.1:8080/fpm-ping>

рассмотрим, что делает каждая строчка в предоставленном Dockerfile:

FROM alpine:3.11

* Использует базовый образ **alpine:3.11**. Alpine Linux — это легковесный дистрибутив Linux, оптимизированный для безопасности и простоты использования.

# ADD https://dl.bintray.com/php-alpine/key/php-alpine.rsa.pub /etc/apk/keys/php-alpine.rsa.pub old link ADD https://packages.whatwedo.ch/php-alpine.rsa.pub /etc/apk/keys/php-alpine.rsa.pub

* Загружает файл ключа для репозитория PHP Alpine и добавляет его в **/etc/apk/keys/**, чтобы Alpine мог доверять этому репозиторию.

# make sure you can use HTTPS RUN apk --update add ca-certificates

* Устанавливает сертификаты CA, необходимые для работы с HTTPS.

RUN echo "https://packages.whatwedo.ch/php-alpine/v3.11/php-7.4" >> /etc/apk/repositories

* Добавляет сторонний репозиторий PHP для Alpine Linux в список репозиториев пакетов.

# Install packages RUN apk --no-cache add php php-fpm php-opcache php-openssl php-curl \ nginx supervisor curl

* Устанавливает PHP и необходимые расширения, а также nginx, supervisor и curl. Флаг **--no-cache** предотвращает сохранение промежуточных данных, что уменьшает размер конечного образа.

# https://github.com/codecasts/php-alpine/issues/21 RUN ln -s /usr/bin/php7 /usr/bin/php

* Создает символическую ссылку **php**, указывающую на **php7**, чтобы можно было использовать команду **php** для запуска PHP 7.

# Configure nginx COPY config/nginx.conf /etc/nginx/nginx.conf

* Копирует пользовательский файл конфигурации nginx в контейнер.

# Remove default server definition RUN rm /etc/nginx/conf.d/default.conf

* Удаляет стандартную конфигурацию nginx, чтобы избежать конфликтов с пользовательской конфигурацией.

# Configure PHP-FPM COPY config/fpm-pool.conf /etc/php7/php-fpm.d/www.conf COPY config/php.ini /etc/php7/conf.d/custom.ini

* Копирует пользовательские файлы конфигурации для PHP-FPM и PHP в соответствующие директории.

# Configure supervisord COPY config/supervisord.conf /etc/supervisor/conf.d/supervisord.conf

* Копирует конфигурационный файл supervisord в контейнер. Supervisor используется для управления процессами nginx и PHP-FPM.

# Setup document root RUN mkdir -p /var/www/html

* Создает корневую директорию для веб-сервера.

# Make sure files/folders needed by the processes are accessible when they run under the nobody user RUN chown -R nobody.nobody /var/www/html && \ chown -R nobody.nobody /run && \ chown -R nobody.nobody /var/lib/nginx && \ chown -R nobody.nobody /var/log/nginx

* Устанавливает права собственности для необходимых директорий, чтобы они были доступны пользователю **nobody**, под которым будут запускаться процессы.

# Switch to use a non-root user from here on USER nobody

* Переключается на пользователя **nobody**, чтобы избежать запуска процессов от имени root.

# Add application WORKDIR /var/www/html COPY --chown=nobody src/ /var/www/html/

* Устанавливает рабочую директорию и копирует файлы приложения в контейнер, устанавливая владельца **nobody**.

# Expose the port nginx is reachable on EXPOSE 8080

* Указывает Docker, что контейнер будет прослушивать порт 8080.

# Let supervisord start nginx & php-fpm CMD ["/usr/bin/supervisord", "-c", "/etc/supervisor/conf.d/supervisord.conf"]

* Устанавливает команду по умолчанию для запуска supervisord с указанной конфигурацией, что позволит запустить nginx и PHP-FPM.

# Configure a healthcheck to validate that everything is up & running HEALTHCHECK --timeout=10s CMD curl --silent --fail http://127.0.0.1:8080/fpm-ping

* Настраивает проверку состояния контейнера. Команда curl проверяет доступность PHP-FPM по указанному URL и возвращает статус.

## Сети docker

### Управление пользовательскими сетями bridge

Создание определяемой пользователем сети bridge — $ docker network create my-net Вы можете указать подсеть, диапазон IP-адресов, шлюз и другие параметры Подсказку можно посмотреть через команду — $ docker network create --help Чтобы удалить определённую пользователем мостовую сеть — **$ docker network rm my-net** (Если контейнеры в настоящее время подключены к сети, сначала отключите их)

**$ docker create --name my-nginx \**

**--network my-net \**

**--publish 8080:80 \**

**nginx:latest**

Подключить контейнер к сети — **$ docker network connect my-net my-nginx**

Отключить контейнер от сети — **$ docker network disconnect my-net my-nginx**

# Практика 2

Ставим доккер только по официальной инструкции

<https://docs.docker.com/engine/install/ubuntu/>

## Образы

**docker search alpine**  поиск образа

d**ocker pull alpine** загрузка образа

**docker image list**  список образов

**docker images** список образов

**docker images –a** список образов включая неактивные

**docker run -it --name name\_container 05455a08881e** запускаем контейнер интерактивно, придумав ему имя и указав его id из списка образов. –it значит интерактивно

**docker image inspect ubuntu** подробности об образе

**docker pull ubuntu:20.04**  загрузка образа с нужным тегом

**docker tag ubuntu:20.04 newname:newtag** переименовать образ и тэг

**docker rm newname:newtag** возвращ

ение к оригинальному имени и тэгу образа без удаления оригинального образа

**docker save -o ubuntu:20.04.tar ubuntu:20.04**  сохраняем образ в архив

**docker rmi ubuntu:20.04** удалим образ

**docker load -i ubuntu:20.04.ta**r загрузим образ из архива

В чем разница способов

* **save/load** выгружают и загружают образы контейнеров, которые будут использоваться для создания контейнеров;
* **export** выгружает файловую систему созданного контейнера (не образ контейнера, а изменения, которые внес контейнер в образ);
* **import** позволяет создать образ тома файловой системы из архива, созданного с помощью **export**, который может использоваться, например, при запуске контейнера с помощью **—volumes-from**.

В целом, разница заключается, что save/load относятся к данным образов, а export/import к данным контейнеров.

## Контейнеры

**docker run -it ubuntu** запуск контейнера убунту в интерактивном режиме.

**docker container run -it ubuntu** запуск нового экземпляра контейнера убунту

**docker ps** или **docker container list**  список запущенных контейнеров

**docker stop <container\_id>** остановка запущенного контейнера

**docker rm id\_container**  удаление контейнера

**docker container rename c84b0d053103 alexcontainer**  переименование контейнера

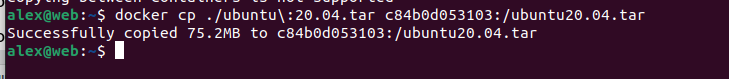
**docker container commit -m "added alex.txt" alexcontainer aleximage:alextag** сохранение контейнера в образ

docker run -it aleximage:alextag запуск нашего созданного образа. Все данные из прошлого созданного контейнера сохранены в этом образе

docker cp 96fc0c6fec6c:/alex.txt . Копирование с контейнера на хост в текущую папку

docker cp ./ubuntu\:20.04.tar c84b0d053103:/ubuntu20.04.tar скопировали файл с хоста в контейнер

docker cp alex.txt c84b0d053103:/alexnew.txt скопировали файл с хоста в контейнер



**docker container stats**  статистика по занимаемым ресурсам контейнерами. Можно использовать для чистки

**docker container exec c84b0d053103 touch alexfile.txt**  выполнение команды в из хоста, но в указанном контейнере

d**ocker container exec -it c84b0d053103 /bin/bash** войти в запущенный контейнер из хоста и сразу запустить баш

## Volume

**docker run -it -v /home/alex/devops/:/mount ubuntu**  запуск контейнера и одновременно замаунтить хостовую папку в контейнер

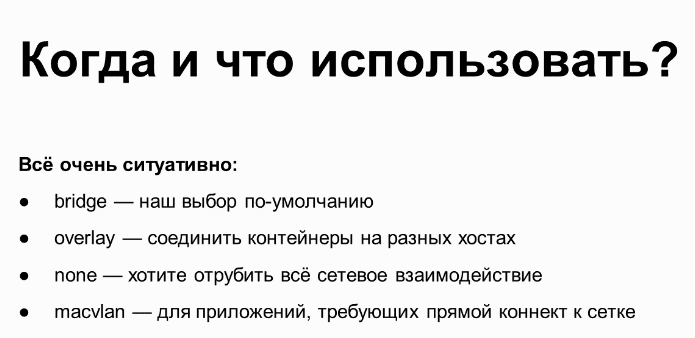
docker volume create —name toster создать новый том в докере

docker volume ls

docker volume inspect toster подробности тома

docker run -it --mount type=volume,source=toster,destination=/toster ubuntu запускаем докер и примаунчиваем созданный ранее том

## Сети



**docker network create my\_network** или **docker network create –d bridge my\_network** Эта команда позволяет создавать новые сети для ваших контейнеров. Например:

**docker network ls**: Эта команда отображает список всех сетей Docker на вашей машине:

**docker network inspect my\_network** Позволяет просматривать подробную информацию о сети Docker, включая список подключенных к ней контейнеров:

**docker network connect / disconnect** **my\_network container\_name** : Эти команды позволяют подключать или отключать контейнеры к существующим сетям. Например:

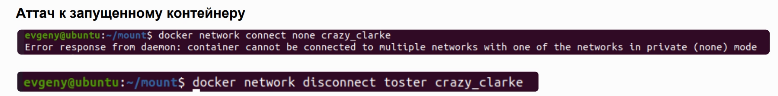
**docker network rm my\_network** Удаляет сеть Docker. Например:

**docker network prune**: Удаляет все неиспользуемые сети Docker. Это может быть полезно для очистки системы от ненужных ресурсов:

**Подключение контейнера к сети при его запуске**: При запуске контейнера вы можете указать, к каким сетям его подключить, используя опцию **--network**. Например:

docker run --network=my\_network image\_name

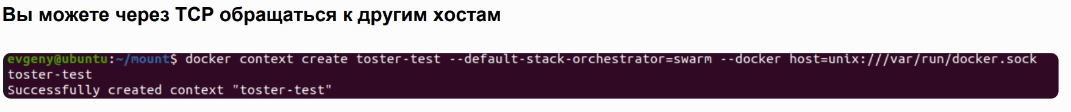
**docker network connect my\_network container\_name** docker network connect с помощью имен контейнеров: Вы также можете использовать имена контейнеров вместо идентификаторов контейнеров при подключении их к сети. Например:



Проброс портов **docker run -d -p 2000:80 nginx**

Проверить можно с хоста **curl 127.0.0.1:2000**

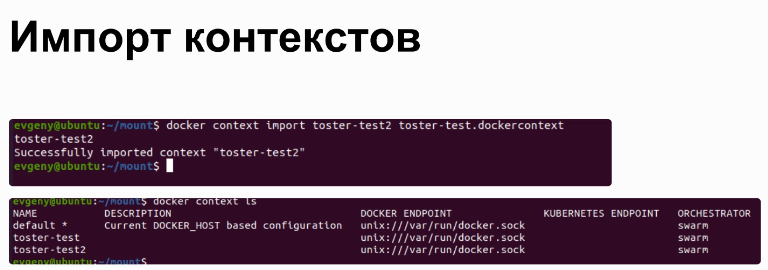
## Docker Context

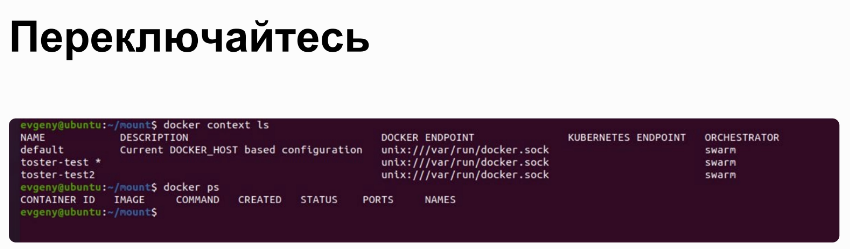


docker context create toster-test --default-stack-orchestrator=swarm --docker host=unix:///var/run/docker.sock toster-test

docker context inspect context-name подробности о контексте







Переключаться через команду use

## DOCKERFILE

**FROM:** Эта инструкция указывает базовый образ, на основе которого будет создан ваш образ. Обычно это минимальный образ операционной системы или другой базовый образ.

**FROM ubuntu:20.04**

**LABEL:** Используется для добавления метаданных к образу. Это может включать автора, описание и версию образа

**LABEL maintainer=**[**author@example.com**](mailto:author@example.com)

**RUN:** Запускает команду внутри контейнера во время сборки образа. Это может быть любая команда, которая может быть выполнена в командной строке, например установка пакетов.

# Обновляем список пакетов и устанавливаем несколько пакетов

**RUN apt-get update && \**

**apt-get install -y \**

**build-essential \**

**wget \**

**curl \**

**git \**

**&& rm -rf /var/lib/apt/lists/\***

**# Клонируем репозиторий из GitHub и переключаемся на определенную ветку**

**RUN git clone https://github.com/username/repository.git /opt/repository && \**

**cd /opt/repository && \**

**git checkout development**

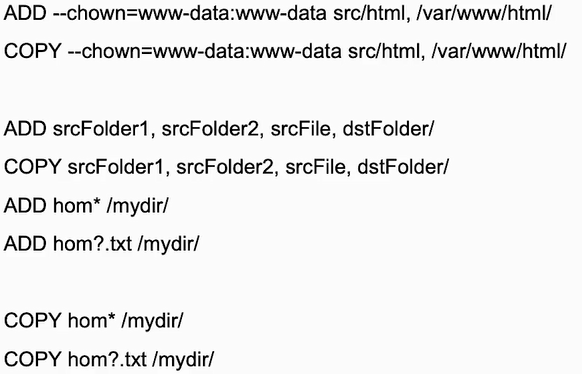
**ADD или COPY**

**COPY:** Эта инструкция копирует файлы и директории из файловой системы хоста в файловую систему контейнера. Она просто копирует файлы и каталоги в целевую директорию в контейнере.

**COPY ./app /app**

**ADD:** Эта инструкция аналогична COPY, но имеет некоторые дополнительные возможности. Она может копировать файлы, а также распаковывать архивы, копировать файлы с удаленных URL и т.д. Поэтому ADD предпочтительнее COPY, если вам нужны такие возможности. Однако в общем случае, если вам нужно только скопировать локальные файлы, рекомендуется использовать COPY для простоты и ясности.Пример использования:

**ADD http://example.com/file.tar.gz /path/in/container/**



## ENTRYPOIN И CMD

Инструкция ENTRYPOINT определяет исполняемую программу или команду, которая будет запущена при запуске контейнера. Она также позволяет передавать аргументы этой программе в командной строке при запуске контейнера. Если в Dockerfile определены и ENTRYPOINT, и CMD, то CMD аргументы будут переданы как аргументы ENTRYPOINT. Если вы определите ENTRYPOINT как массив строк (JSON-массив), то команду и аргументы можно передавать вместе с использованием docker run.

**ENTRYPOINT ["echo", "Hello"]**

В этом примере при запуске контейнера с использованием команды docker run, echo Hello будет выполнено внутри контейнера.

**ENTRYPOINT** следует использовать для исполняемых файлов, а **CMD** для аргументов

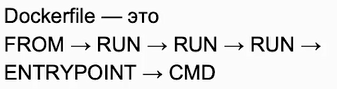
**ENTRYPOINT ["ping"]**

**CMD ["127.0.0.1"]**

Мы можем переопределить параметр инструкции cmd при запуске контейнера:

**docker run -rm -it myubuntu 1.1.1.1**

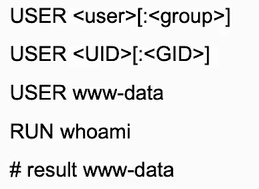
**docker run -rm -it myubuntu -- 1.1.1.1**



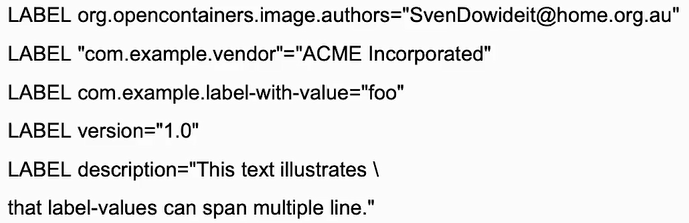
## WORKDIR



## USER



## LABEL



EXPOSE

Открывает порт

Особо не используется и служит для документирования в основном. Пробрасываются порты через флаг -**p** при запуске контейнера

## Пример dockerfile для запуска nginx практика

### Сложный вариант для обучения

--------------------------------------------------------------------------------------------------------------------------------------

FROM ubuntu

ENV NODE\_VERSION=v16.18.0

ENV NODE\_DISTRO=linux-x64

RUN apt update && apt install -y curl xz-utils

WORKDIR /tmp RUN curl <https://nodejs.org/dist/> $NODE\_VERSION/node-$NODE\_VERSION-$NODE\_DISTRO.tar.xz -o node-$NODE\_VERSION-$NODE\_DISTRO. tar. xz

RUN mkdir -p /usr/local/lib/nodejs && tar -xJvf node-$NODE\_VERSION-$NODE\_DISTRO.tar.xz -C /usr/local/lib/nodejs

RUN in -s /usr/local/lib/nodejs/node-$NODE\_VERSION-$NODE\_DISTRO/bin/node /usr/bin/node && \

ln -s /usr/local/lib/nodejs/node-$NODE\_VERSION-$NODE\_DISTRO/bin/npm /usr/bin/npm && \ ln -s /usr/local/lib/nodejs/node-$NODE\_VERSION-$NODE\_DISTRO/bin/npx /usr/bin/npx && \ node -v && npm version && npx -v

RUN npx create-react-app my-app

WORKDIR /tmp/my-app

EXPOSE 3000

ENTRYPOINT ["npm", "start"]

### Простой и оптимальный вариант dockerfile практика

------------------------------------------------------------------------------------------------------------------------------------

#берем за базу прогугленный на сайте докера элпайновский нод

**FROM node:16-alpine**

# устанавливаем переменную среды – нужно для самой ноды

**ENV NODE\_ENV=production**

# Указываем рабочий каталог для контейнера, если его нет он создается

**WORKDIR /app**

# Копируются 2 файла в текущую директорию **./** а это на данный момент **/app**

**COPY ["package.json", "package-lock.json\*", "./"]**

# Установка модулей. Таким образом, команда npm install --production установит только те пакеты, которые находятся в секции "dependencies" вашего package.json файла, игнорируя те, что находятся в секции "devDependencies". Это часто используется при развертывании приложений в производственной среде, когда вам нужно минимизировать размер установленных зависимостей.

**RUN npm install –production**

# Копирование всего остального. Т.е. весь build context копируется в **/app**

**COPY . .**

# запустить

**CMD ["npm", "start" ]**

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Запускаем сборку

**docker build -t alex\_react\_image.**

**-t** присваиваем имя будущему образу

Запускаем контейнер

**docker run -d --rm --name alex\_react\_container -p 7000:3000 alex\_react\_image**

-d означает что в фоне запукаем

--rm, что после остановки, контейнер удалится

-p прокидываем порты

## Multistage многоступенчатая сборка

Берем за основу предыдущий пример

Инструкция AS в Dockerfile используется для определения имени цели (target stage) в многоступенчатом сборочном процессе. Многоступенчатый сборочный процесс позволяет создавать более компактные и безопасные образы Docker, разделяя различные этапы сборки и используя только необходимые артефакты из каждого этапа.

Синтаксис использования инструкции AS следующий: AS <имя\_цели>

Эта инструкция используется в сочетании с инструкцией FROM, чтобы назначить имя цели определенному этапу сборки.

FROM golang:1.16 AS builder

WORKDIR /app

COPY . .

RUN go build -o myapp .

FROM alpine:latest

WORKDIR /root/

COPY --from=builder /app/myapp .

CMD ["./myapp"]

Сначала мы объявляем цель с именем builder, используя инструкцию AS.

Затем мы выполняем сборку на этапе builder.

В конце мы копируем результаты сборки из этапа builder в новый образ на основе Alpine Linux.

Использование инструкции AS позволяет явно указать имена для каждого этапа сборки, что упрощает управление и понимание Dockerfile, особенно в случае многоступенчатой сборки.

### Контейнирезация своего приложения на react

1. [Создаем](#_Yarn_установка) или берем откуда-то react приложение

Берем докерфайл со страницы официальной документации для ноды <https://docs.docker.com/language/nodejs/containerize/> и сохраняем его в папку с react приложением  
ARG NODE\_VERSION=22.2.0

FROM node:22.2.0-alpine AS first

# Use production node environment by default.

ENV NODE\_ENV production

WORKDIR /usr/src/app

# Download dependencies as a separate step to take advantage of Docker's caching.

# Leverage a cache mount to /root/.npm to speed up subsequent builds.

# Leverage a bind mounts to package.json and package-lock.json to avoid having to copy them into

# into this layer.

RUN --mount=type=bind,source=package.json,target=package.json \

    --mount=type=bind,source=package-lock.json,target=package-lock.json \

    --mount=type=cache,target=/root/.npm \

    npm ci --omit=dev

# Run the application as a non-root user.

USER node

# Copy the rest of the source files into the image.

COPY . .

# Expose the port that the application listens on.

EXPOSE 3000

# Run the application.

CMD npm start

1. Запускаем сборку **docker build -t alex\_react\_image .**
2. Запускаем контейнер **docker run -d --rm --name alex\_react\_container -p 7000:3000 alex\_react\_image** в фоновом режиме и проверяем работу. Если не заработало то смотрим логи **docker logs -f alex\_react\_container** Или смотрим запустился конейр ли в принципе **docker ps**
3. Запускаем контейнер уже интерактивно тогда **docker run -it --rm --name alex\_react\_container -p 7000:3000 alex\_react\_image**
4. В нашем случае ошибка в последней команде докерфайла. Там надо использщовать **CMD npm start**Запускаем пересборку **docker build -t alex\_react\_image .** и запускаем заново **docker run -it --rm --name alex\_react\_container -p 7000:3000 alex\_react\_image** Ошибок нет, все по факту работает

### Multistage сборка

1. Создаем dockerfile

#1 stadiya

FROM node:22.2.0-alpine AS first

ENV NODE\_ENV production

WORKDIR /myapp

COPY . .

RUN npm install && npm run build

#2 stadiya

FROM nginx:1.25.5-alpine

COPY --from=first /myapp/build /var/www/myapp

RUN ls -la /var/www

EXPOSE 80

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

1. На первом этапе (первый блок) мы соберем в nodejs контейнере bundle, т.е. создадим артефакт сборки, т.е. выполним npm install и npm build и этот результат отдадим во второй этап в nginx для запуска этих статичных файлов
2. Создадим файл default.conf для настройки nginx  
   server{

listen 80 default;

server\_name \_;

root /var/www/myapp;

index index.html;

}

1. Запускаем сборку **docker build -t alex\_react\_image -f dockerfilem .**
2. Запускаем контейнер **docker run -d --rm -p 9000:80 --name alex\_react\_nginx\_container -v $(pwd)/default.conf:/ etc/nginx/conf.d/default.conf alex\_react\_image**
3. Можем посмотреть логи **docker logs -f alex\_react\_nginx\_container**
4. Можем подключиться к шеллу контейнера **docker exec -it alex\_react\_nginx\_container /bin/sh**  
    **docker exec -it container-name /bin/bash** (sh для alpine) - Интерактивно подключиться к контейнеру для управления, exit чтобы выйти

Пример dockerfile от ИИ

# Указываем базовый образ

FROM ubuntu:20.04

# Добавляем метаданные к образу

LABEL maintainer="author@example.com"

# Устанавливаем переменные среды

ENV MY\_VAR=value

# Определяем переменные, которые могут быть переданы во время сборки образа

ARG MY\_ARG=value

# Устанавливаем оболочку по умолчанию для выполнения инструкций RUN, CMD, ENTRYPOINT, COPY и ADD

SHELL ["/bin/bash", "-c"]

# Создаем точку монтирования для постоянного хранения данных

VOLUME /data

# Устанавливаем рабочий каталог для последующих инструкций

WORKDIR /app

# Копируем файлы из файловой системы хоста в файловую систему контейнера

COPY ./app /app

# Добавляем проверку состояния контейнера

HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost/ || exit 1

# Задаем исполняемый файл, который будет запущен при запуске контейнера

ENTRYPOINT ["java", "-jar", "myapp.jar"]

# Открываем порт контейнера

EXPOSE 8080

# Задаем команду по умолчанию, которая будет выполняться при запуске контейнера

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

# Устанавливаем пользователя, под которым будут выполняться инструкции RUN, CMD и ENTRYPOINT

USER appuser

# Запускаем команду внутри контейнера во время сборки образа

RUN apt-get update && apt-get install -y package\_name

# DOCKER-COMPOSE

## ТЕОРИЯ

Docker-compose.yaml шаблон

------------------------------------------------------------------------------------------------------------------------

version: '3' # Версия синтаксиса Docker Compose

services: # Определение сервисов приложения

service1: # Название первого сервиса

image: image\_name:tag # Docker образ, используемый для сервиса

ports: # Определение портов, которые будут проброшены на хостовую машину

- "host\_port:container\_port" # Пример: "8080:80"

volumes: # Определение примонтированных томов

- "host\_path:container\_path" # Пример: "./data:/app/data"

environment: # Определение переменных окружения

- ENV\_VAR1=value1 # Пример: "DEBUG=true"

- ENV\_VAR2=value2

depends\_on: # Определение зависимостей сервисов

- service2 # Пример: service2

networks: # Определение сетей, к которым присоединяется сервис

- my\_network # Пример: my\_network

service2: # Название второго сервиса

build: ./path\_to\_Dockerfile # Путь к Dockerfile для сборки образа

volumes: # Определение примонтированных томов

- "host\_path:container\_path"

environment: # Определение переменных окружения

- ENV\_VAR=value

networks: # Определение сетей, к которым присоединяется сервис

- my\_network

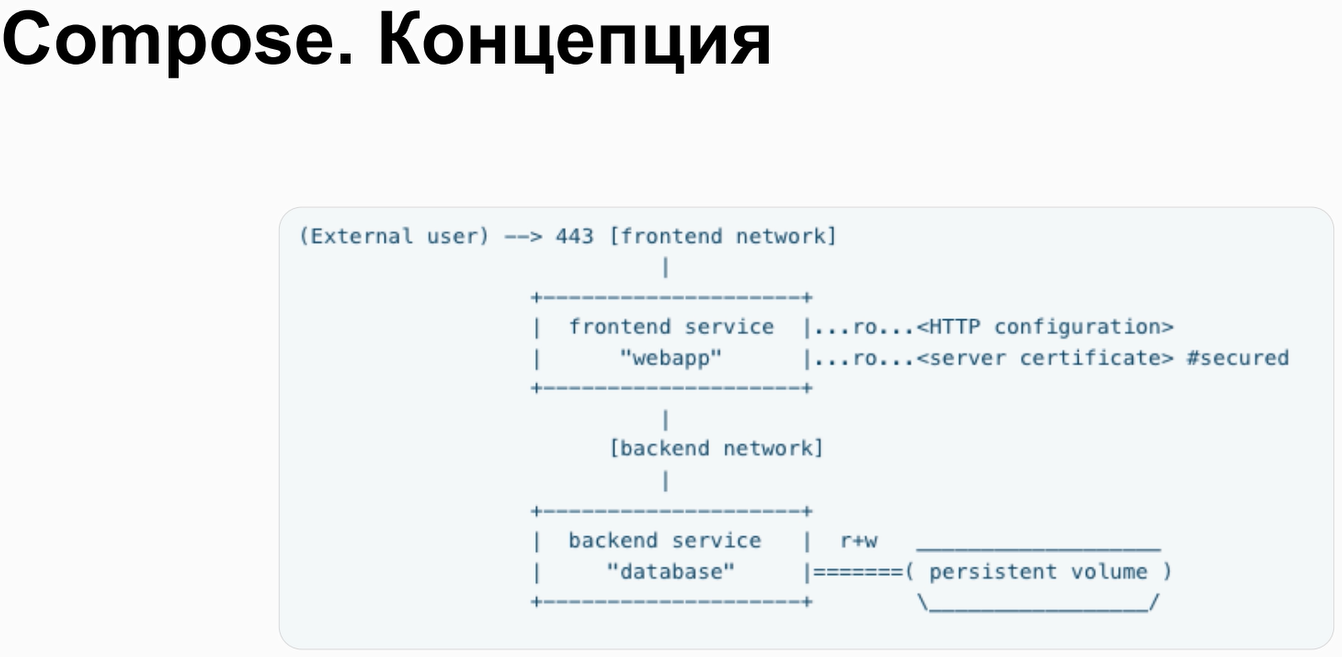
networks: # Определение сетей, используемых сервисами

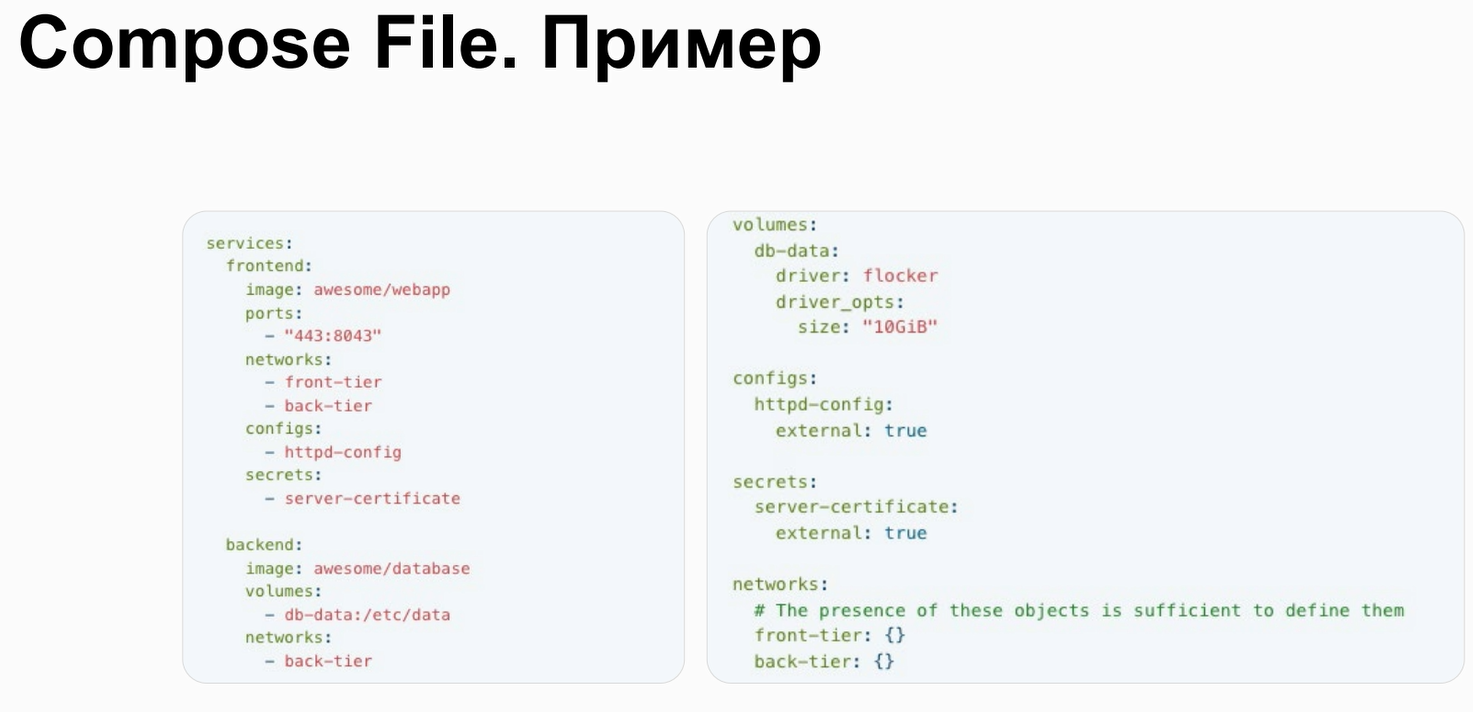
my\_network: # Название сети

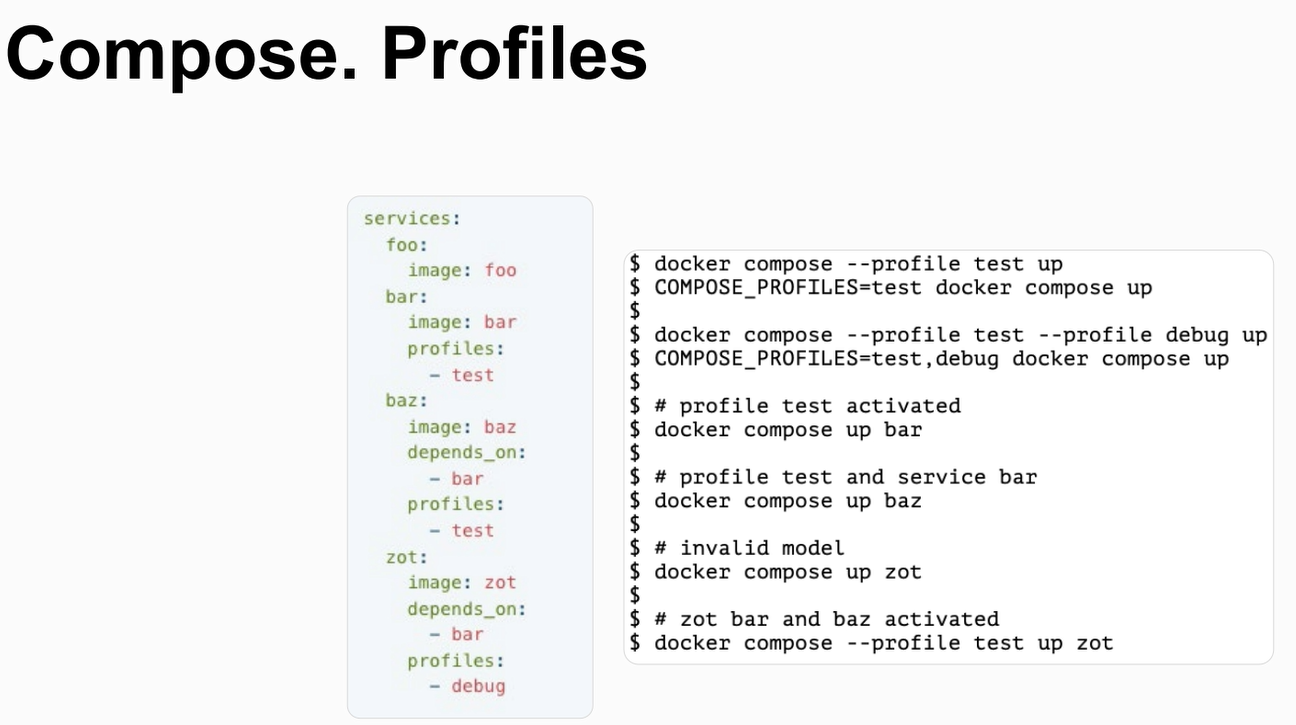
driver: bridge # Драйвер сети (по умолчанию bridge)

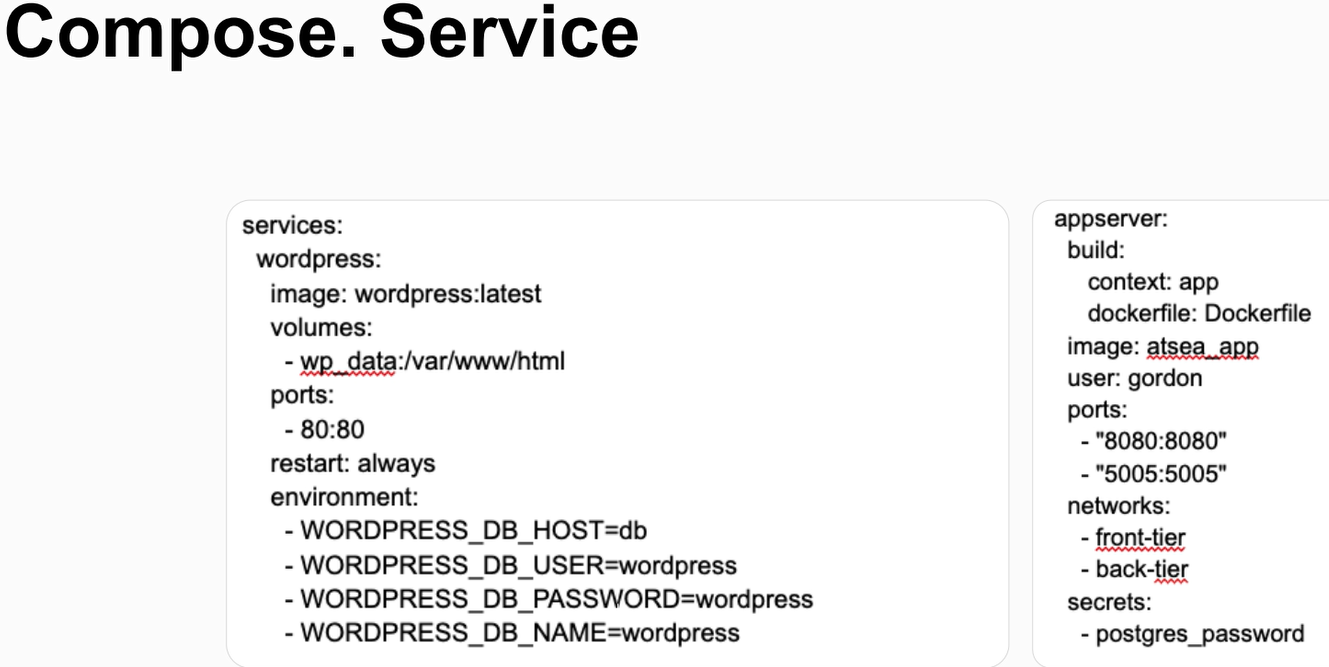
Базовые возможности Compose и ключевые особенности Compose:

* запуск, остановка и пересборка сервисов
* множество изолированных сред на одном сервере. 1 compose файл это одна изолировнная среда
* просмотр состояния запущенных сервисов
* сохранение данных на созданных докер-вольюмах, при остановки контейнеров
* просмотр потока логов событий запущенных сервисов
* пересоздание только тех контейнеров, которые изменились. При выполнении docker compose up постоянном, если какой-то образ изменился, то он будет перезапущен
* выполнение команд в контейнерах
* при помощи переменных можно задавать поведение в различных средах

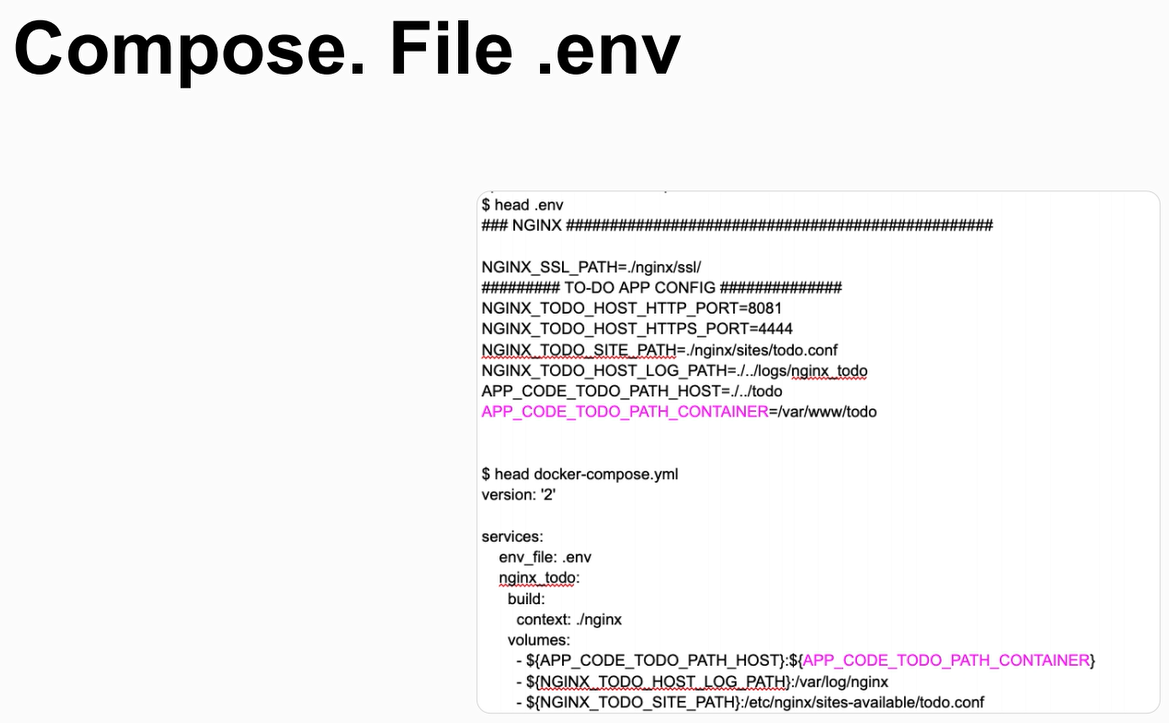


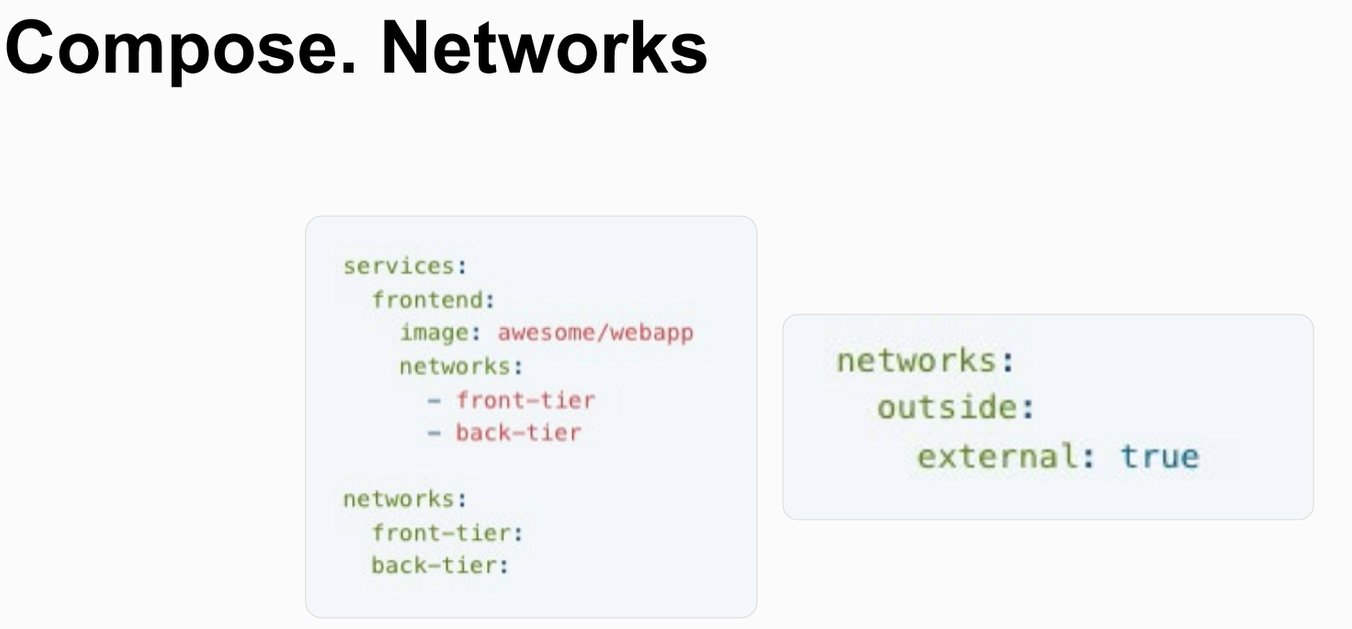


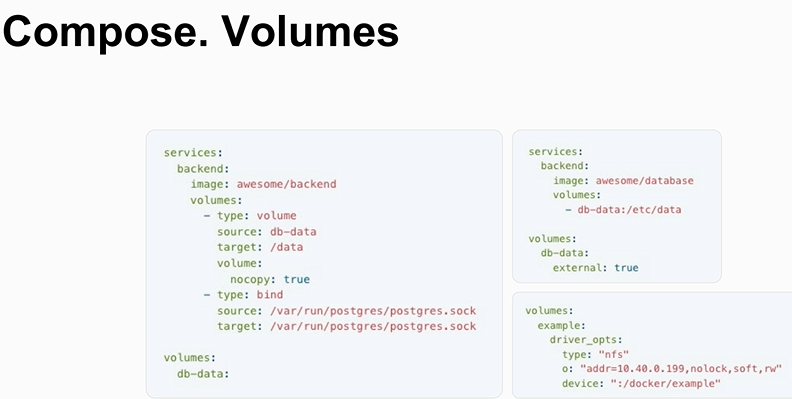


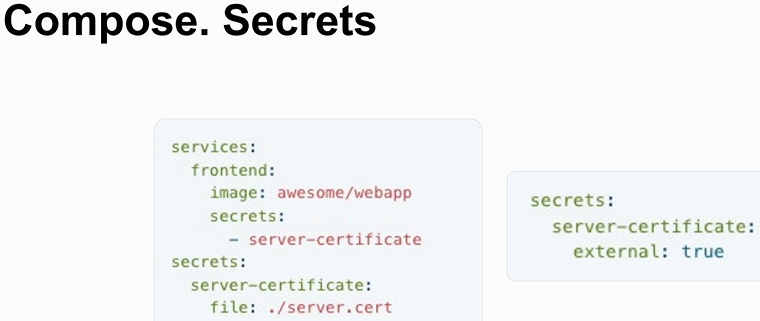














### GPT

Инструкция `services` в `docker-compose.yml` файле позволяет определить множество контейнеров, которые должны быть запущены вместе как часть одного приложения. Каждое определение сервиса включает в себя конфигурацию контейнера, его образ, порты, тома, зависимости и многое другое. Давайте рассмотрим подробности конфигурации `services`.

### Основные параметры для определения сервиса

#### 1. `image`

Указывает на образ, который будет использоваться для создания контейнера. Образ может быть локальным или из Docker Hub.

```yaml

services:

web:

image: nginx:latest

```

#### 2. `build`

Позволяет указать директорию с Dockerfile или дополнительные опции для сборки образа.

```yaml

services:

app:

build:

context: ./app

dockerfile: Dockerfile.dev

```

#### 3. `command`

Позволяет переопределить команду, которая будет выполнена внутри контейнера.

```yaml

services:

app:

image: my\_app:latest

command: python app.py

```

#### 4. `environment`

Позволяет задать переменные окружения для контейнера.

```yaml

services:

db:

image: postgres:latest

environment:

POSTGRES\_USER: user

POSTGRES\_PASSWORD: password

```

#### 5. `ports`

Позволяет опубликовать порты контейнера на хосте.

```yaml

services:

web:

image: nginx:latest

ports:

- "80:80"

- "443:443"

```

#### 6. `volumes`

Позволяет монтировать тома или директории с хоста в контейнер.

```yaml

services:

app:

image: my\_app:latest

volumes:

- ./app:/usr/src/app

- app\_data:/var/lib/data

```

#### 7. `depends\_on`

Задает зависимости между сервисами. Это указывает Docker Compose, в каком порядке запускать сервисы.

```yaml

services:

web:

image: nginx:latest

depends\_on:

- app

app:

image: my\_app:latest

```

#### 8. `networks`

Позволяет подключить сервис к одной или нескольким сетям.

```yaml

services:

web:

image: nginx:latest

networks:

- frontend

- backend

```

#### 9. `restart`

Задает политику перезапуска контейнера.

```yaml

services:

web:

image: nginx:latest

restart: always

```

### Полный пример `docker-compose.yml`

```yaml

version: '3.9'

services:

web:

image: nginx:latest

ports:

- "80:80"

volumes:

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

networks:

- frontend

depends\_on:

- app

environment:

- NGINX\_HOST=web

- NGINX\_PORT=80

app:

build:

context: ./app

dockerfile: Dockerfile

volumes:

- ./app:/usr/src/app

networks:

- backend

environment:

- APP\_ENV=development

restart: on-failure

command: python app.py

db:

image: postgres:latest

volumes:

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

environment:

POSTGRES\_USER: user

POSTGRES\_PASSWORD: password

networks:

- backend

networks:

frontend:

driver: bridge

backend:

driver: bridge

volumes:

db\_data:

```

### Подробное описание основных параметров

#### `build`

`build` позволяет задать опции для сборки образа.

- `context`: Директория, в которой находится Dockerfile.

- `dockerfile`: Имя Dockerfile, если оно отличается от стандартного `Dockerfile`.

- `args`: Параметры сборки, которые могут быть переданы в Dockerfile через `ARG`.

```yaml

services:

app:

build:

context: ./app

dockerfile: Dockerfile.dev

args:

build\_arg: value

```

#### `environment`

Задание переменных окружения может быть выполнено несколькими способами:

- В формате списка:

```yaml

services:

app:

image: my\_app:latest

environment:

- KEY=value

- ANOTHER\_KEY=another\_value

```

- В формате словаря:

```yaml

services:

app:

image: my\_app:latest

environment:

KEY: value

ANOTHER\_KEY: another\_value

```

#### `depends\_on`

Задает порядок запуска контейнеров. Например, чтобы веб-сервер ждал, пока база данных и приложение будут готовы.

```yaml

services:

web:

image: nginx:latest

depends\_on:

- db

- app

```

#### `networks`

Контейнеры могут быть подключены к одной или нескольким сетям для организации коммуникации.

```yaml

services:

web:

image: nginx:latest

networks:

- frontend

- backend

```

#### `volumes`

Тома могут быть использованы для хранения данных, которые необходимо сохранить при перезапуске контейнера, либо для совместного использования данных между контейнерами.

```yaml

services:

app:

image: my\_app:latest

volumes:

- ./app:/usr/src/app

- app\_data:/var/lib/data

volumes:

app\_data:

```

#### `restart`

Политики перезапуска определяют поведение контейнера в случае его остановки или сбоя:

- `no` (по умолчанию): Контейнер не будет перезапускаться.

- `always`: Контейнер будет всегда перезапускаться.

- `on-failure`: Контейнер будет перезапускаться только при завершении с ошибкой.

- `unless-stopped`: Контейнер будет перезапускаться всегда, кроме случаев, когда он был явно остановлен пользователем.

```yaml

services:

web:

image: nginx:latest

restart: always

```

### Полезные команды

- \*\*Создать и запустить контейнеры\*\*: `docker-compose up`

- \*\*Остановить контейнеры\*\*: `docker-compose down`

- \*\*Перезапустить контейнеры\*\*: `docker-compose restart`

- \*\*Посмотреть логи контейнеров\*\*: `docker-compose logs`

- \*\*Просмотреть статус контейнеров\*\*: `docker-compose ps`

Используя `services` в `docker-compose.yml`, вы можете гибко и удобно управлять множеством контейнеров, их зависимостями, сетевыми настройками и многими другими параметрами, что упрощает развертывание и управление многокомпонентными приложениями.

Docker Compose — это инструмент для определения и запуска многоконтейнерных Docker приложений. Он использует файл `docker-compose.yml` для описания сервисов, сетей и томов, которые необходимы вашему приложению. Ниже перечислены основные команды Docker Compose, которые часто используются при работе с этим инструментом:

### Основные команды Docker Compose

1. \*\*`docker-compose up`\*\*

Запускает контейнеры, определенные в файле `docker-compose.yml`.

- `-d`, `--detach`: Запускает контейнеры в фоновом режиме (отключенном режиме).

- `--build`: Собирает образы перед запуском контейнеров.

```bash

docker-compose up

docker-compose up -d

docker-compose up --build

```

2. \*\*`docker-compose down`\*\*

Останавливает и удаляет контейнеры, сети и тома, созданные командой `up`.

- `-v`, `--volumes`: Удаляет тома, созданные для контейнеров.

- `--rmi`: Удаляет образы. Используйте `local` для удаления только локально созданных образов или `all` для удаления всех образов.

```bash

docker-compose down

docker-compose down -v

docker-compose down --rmi all

```

3. \*\*`docker-compose start`\*\*

Запускает существующие остановленные контейнеры (без создания или пересоздания).

```bash

docker-compose start

```

4. \*\*`docker-compose stop`\*\*

Останавливает запущенные контейнеры, но не удаляет их.

```bash

docker-compose stop

```

5. \*\*`docker-compose restart`\*\*

Перезапускает контейнеры.

```bash

docker-compose restart

```

6. \*\*`docker-compose build`\*\*

Собирает образы для сервисов, описанных в `docker-compose.yml`.

- `--no-cache`: Собирает образы без использования кэша.

- `--pull`: Перед сборкой выполняет команду `docker pull` для каждого базового образа.

```bash

docker-compose build

docker-compose build --no-cache

```

7. \*\*`docker-compose ps`\*\*

Показывает статус контейнеров, запущенных через Docker Compose.

```bash

docker-compose ps

```

8. \*\*`docker-compose exec`\*\*

Выполняет команду внутри работающего контейнера. Подобно `docker exec`.

```bash

docker-compose exec <service\_name> <command>

docker-compose exec web bash

```

9. \*\*`docker-compose run`\*\*

Запускает одноразовый (одноразовый) контейнер для выполнения команды. Используется для запуска краткосрочных команд, таких как миграции базы данных.

```bash

docker-compose run <service\_name> <command>

docker-compose run web python manage.py migrate

```

10. \*\*`docker-compose logs`\*\*

Показывает логи всех контейнеров или конкретного сервиса.

```bash

docker-compose logs

docker-compose logs <service\_name>

docker-compose logs -f # Следит за логами в реальном времени

```

11. \*\*`docker-compose config`\*\*

Проверяет файл `docker-compose.yml` на синтаксические ошибки и показывает его итоговую конфигурацию.

```bash

docker-compose config

```

12. \*\*`docker-compose pull`\*\*

Загружает образы из Docker Registry для сервисов, указанных в `docker-compose.yml`.

```bash

docker-compose pull

```

13. \*\*`docker-compose kill`\*\*

Принудительно завершает работу контейнеров.

```bash

docker-compose kill

```

14. \*\*`docker-compose rm`\*\*

Удаляет остановленные контейнеры, созданные командой `up`.

```bash

docker-compose rm

docker-compose rm -f # Удаляет контейнеры без запроса подтверждения

```

15. \*\*`docker-compose version`\*\*

Показывает версию Docker Compose.

```bash

docker-compose version

```

16. \*\*`docker-compose top`\*\*

Показывает запущенные процессы в контейнерах, управляемых Docker Compose.

```bash

docker-compose top

```

### Дополнительные команды

- \*\*`docker-compose scale`\*\* (Устаревшая): Масштабирует количество контейнеров для заданного сервиса. В последних версиях Docker Compose эта команда устарела, и для масштабирования используются настройки в `docker-compose.yml` и команда `up --scale`.

```bash

docker-compose up --scale <service\_name>=<number>

```

Эти команды покрывают основные сценарии использования Docker Compose для управления многоконтейнерными приложениями.

## ПРАКТИКА

1. Запустим тество nginx **docker run -d --rm -p 8080:80 --name ng1 nginx**
2. Запустим просмотр логов docker logs -f ng1 и если обновить страницу nginx, то в логах посыпется новая инфа
3. Создадим файл compose.yaml и запускаем его. Создается и контейнер и сеть

services:

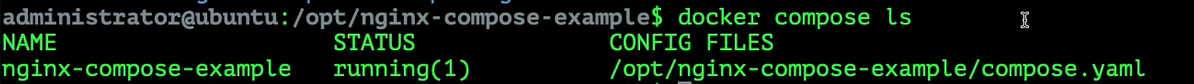
  ng1:

    image: nginx

    ports:

      - "8080:80"

1. Запустим все **docker compose up** Если ошибка и порты заняты, то удаляем мешающие контейнеры **docker rm -f ng1**
2. **docker compose ls** покажет перечень всех запущенных проектов



1. Останавливаем контейнер **Ctrl+C**
2. Удаляем сервисы, ресурсы docker compose down
3. Сервисы по названию пингуются между собой

### Запуск сервисов

1. Создадим в папке compose.yaml

services:

  ng1:

    image: nginx

    volumes:

      - type: bind

        source: /tmp

        target: /tmp/ng1\_tmp

  ng2:

    image: nginx

    volumes:

      - type: bind

        source: /tmp

        target: /tmp/ng2\_tmp

1. Подключимся к ng1. **docker compose exec -it ng1 bash**
2. Создам тестовый файл для проверки бинда **echo "my text test" >> /tmp/ng1\_tmp/test\_file**
3. В результате получается по идее общая с хостом папка на 2ух сервисах

### Запуск wordpress

Переходим на страницу документации и выбираем вордпрес <https://docs.docker.com/compose/samples-for-compose/>

Далее копируем docker compose файл в новую папку и запускаем docker compose up -d

services:

db:

# We use a mariadb image which supports both amd64 & arm64 architecture

image: mariadb:10.6.4-focal

# If you really want to use MySQL, uncomment the following line

#image: mysql:8.0.27

command: '--default-authentication-plugin=mysql\_native\_password'

volumes:

- db\_data:/var/lib/mysql

restart: always

environment:

- MYSQL\_ROOT\_PASSWORD=somewordpress

- MYSQL\_DATABASE=wordpress

- MYSQL\_USER=wordpress

- MYSQL\_PASSWORD=wordpress

expose:

- 3306

- 33060

wordpress:

image: wordpress:latest

volumes:

- wp\_data:/var/www/html

ports:

- 80:80

restart: always

environment:

- WORDPRESS\_DB\_HOST=db

- WORDPRESS\_DB\_USER=wordpress

- WORDPRESS\_DB\_PASSWORD=wordpress

- WORDPRESS\_DB\_NAME=wordpress

volumes:

db\_data:

wp\_data:

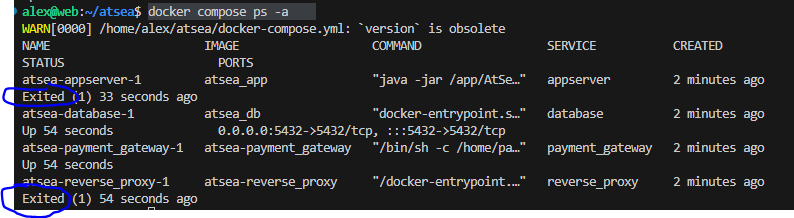
1. Проверяем. Word press готов к настройке запуску
2. Исправляем ошибки.   
   В докерфайле изменяем на  
   FROM openjdk:8-jdk-alpine

а в докер компоузе комментируем сеть оверлей

#    driver: overlay

1. **docker compose ps -a** проверка всех запущенных и упавших сервисов

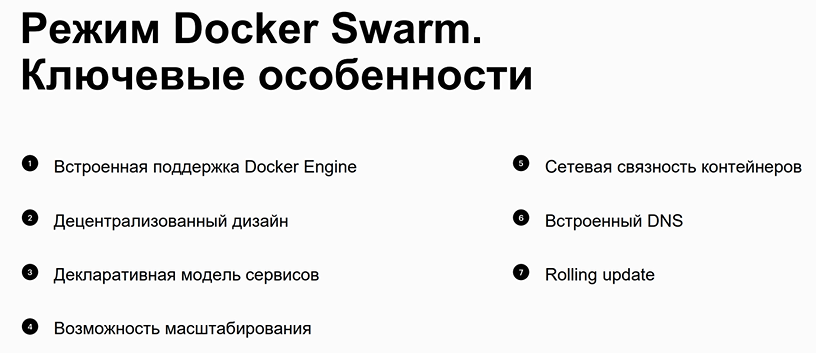
**docker compose logs -f appserver** можем посмотреть логи сервиса



1. Видим, что 2 сервиса упали
2. Находим что не хватает файла <https://jdbc.postgresql.org/download/postgresql-42.5.0.jar>

# Docker Swarm

## Теория



**Основные команды Docker Swarm**

* **docker swarm init**: Инициализация нового Swarm-кластера.
* **docker swarm join**: Присоединение узла к существующему Swarm-кластеру.
* **docker swarm leave**: Отключение узла от Swarm-кластера.
* **docker node ls**: Просмотр всех узлов в кластере.
* **docker service create**: Создание новой службы в Swarm.
* **docker service ls**: Просмотр всех служб в кластере.
* **docker service scale**: Масштабирование службы до указанного числа реплик.
* **docker service update**: Обновление конфигурации существующей службы.

Конечно! Давайте рассмотрим основные термины Docker Swarm более подробно.

### Swarm

Swarm — это кластер Docker Engine, в котором несколько узлов работают вместе, предоставляя ресурсы и балансируя задачи. Когда вы инициализируете Swarm, один узел становится менеджером и контролирует состояние кластера.

### Node (Узел)

Узел в Docker Swarm — это отдельный экземпляр Docker Engine, который может быть физическим сервером или виртуальной машиной. Узлы делятся на два типа:

1. \*\*Manager Node (Узел-менеджер)\*\*:

- Управляет состоянием и конфигурацией кластера.

- Координирует действия всех узлов, распределяет задачи и обрабатывает изменения в кластере.

- Поддерживает распределенную базу данных, чтобы отслеживать состояние всего Swarm.

- Может участвовать в голосовании и принятии решений о состоянии кластера.

- Для обеспечения отказоустойчивости рекомендуется иметь нечетное число узлов-менеджеров (например, 3 или 5).

2. \*\*Worker Node (Узел-исполнитель)\*\*:

- Выполняет задачи (контейнеры), назначенные узлом-менеджером.

- Не участвует в управлении кластером, только выполняет задания.

- Может быть добавлен или удален из кластера без остановки всего Swarm.

### Service (Служба)

Служба — это описание задачи или группы задач, которые должны быть запущены в кластере. Служба определяет состояние контейнеров, количество реплик, стратегию обновления и другие параметры.

- \*\*Replicated Service (Реплицированная служба)\*\*: Служба, в которой запущено определенное количество реплик контейнеров.

- \*\*Global Service (Глобальная служба)\*\*: Служба, в которой контейнер запускается на каждом доступном узле в кластере.

### Task (Задача)

Задача — это единица работы, которую необходимо выполнить. В контексте Docker Swarm задача представляет собой один контейнер, который должен быть запущен и команду, которую надо в ней запустить. Узел-менеджер распределяет задачи между узлами-исполнителями. Задача – это одна атомарная единица роя

### Replica (Реплика)

Реплика — это копия задачи (контейнера), которая выполняется в кластере. Количество реплик службы можно изменить, чтобы масштабировать приложение. Например, если у вас есть служба с 5 репликами, в кластере будет запущено 5 одинаковых контейнеров. Одна реплика это 1 таск

### Overlay Network (Наложенная сеть)

Наложенная сеть позволяет контейнерам, работающим на разных узлах, взаимодействовать друг с другом. Docker Swarm автоматически создает наложенные сети, чтобы обеспечить соединение между службами.

- \*\*Intra-cluster Networking (Внутрикластерная сеть)\*\*: Создает виртуальные сети поверх физических сетей, позволяя контейнерам на разных узлах общаться друг с другом.

### Ingress Network (Входящая сеть)

Входящая сеть используется для маршрутизации внешних запросов к службам, работающим в Swarm. Она предоставляет единый IP-адрес, через который внешние клиенты могут обращаться к службам внутри кластера.

### Routing Mesh (Маршрутизационная сетка)

Маршрутизационная сетка распределяет входящий трафик между всеми доступными узлами, независимо от того, где физически запущены контейнеры службы. Это позволяет обеспечить доступность служб через любой узел кластера.

### Основные команды Docker Swarm

Некоторые из основных команд для управления Docker Swarm:

- \*\*`docker swarm init`\*\*: Инициализирует новый Swarm-кластер и делает текущий узел менеджером.

- \*\*`docker swarm join`\*\*: Присоединяет узел к существующему Swarm-кластеру.

- \*\*`docker swarm leave`\*\*: Отключает узел от Swarm-кластера.

- \*\*`docker node ls`\*\*: Показывает список всех узлов в кластере.

- \*\*`docker service create`\*\*: Создает новую службу в Swarm.

- \*\*`docker service ls`\*\*: Показывает список всех служб в кластере.

- \*\*`docker service scale`\*\*: Масштабирует службу до указанного числа реплик.

- \*\*`docker service update`\*\*: Обновляет конфигурацию существующей службы.

### Пример использования Docker Swarm

Предположим, у вас есть простое приложение, которое вы хотите развернуть в Docker Swarm. Вот шаги, которые вы могли бы выполнить:

1. \*\*Инициализация Swarm\*\*:

```sh

docker swarm init

```

Это сделает текущий узел менеджером.

2. \*\*Создание службы\*\*:

```sh

docker service create --name my\_service --replicas 3 nginx

```

Это создаст службу с именем `my\_service`, запустив три реплики контейнера с образом `nginx`.

3. \*\*Просмотр служб\*\*:

```sh

docker service ls

```

Это покажет список всех служб в кластере.

4. \*\*Масштабирование службы\*\*:

```sh

docker service scale my\_service=5

```

Это увеличит количество реплик службы `my\_service` до пяти.

5. \*\*Обновление службы\*\*:

```sh

docker service update --image nginx:latest my\_service

```

Это обновит службу `my\_service`, запуская контейнеры с новым образом `nginx:latest`.

Эти концепции и команды позволяют эффективно управлять контейнерными приложениями в распределенной среде, обеспечивая масштабируемость и высокую доступность.

## ПРАКТИКА