

из докер файла собирается образ, а на основе образа запускается контейнер  
докер файл - набор инструкций  
образ - все наши файлы  
docker compose - для объединения и запуска нескольких контейнеров  
следить за нагрузкой контейнеров следит кубер

## ЛЕКЦИЯ 1

### Разработка бывает каскадной(водопад) и гипкой SCRUM - пример гипкой

спринт обычно 2 недели

есть владелец продукта, скрам-мастер и команда, команда выбирает тимлида, но обычно тимлид - это и есть скрам-мастер

бэклог продукта - разбили на мелкие задачи (правильно разбивать учитывая capacity людей)

обзор спринта - это показ демо-версии заказчику

ретро - извлечение смысла прошлого спринта

#### Scrum



#### Плюсы Scrum

- Гибкое изменения требований
- Снижение рыночных рисков
- Рабочий продукт в конце каждого спринта
- Универсальная мотивированная команда

## Минусы Scrum

- Долгое время разработки до первого релиза
- Высокая цена найма многофункциональных работников
- Нет фиксированного бюджета и технического задания
- Сложности планирования крупных проектов

но сейчас на многофункциональных разрабов забивают и лучше больше но которые в чем-то одном, так как это дешевле

**DEV**

### Development — разработка

- ★ Разрабатывают продукт и новый функционал
- ★ Хотят как можно скорее его увидеть на серверах

**OPS**

### Operations — эксплуатация (системы)

- ★ Отвечают за стабильность серверов
- ★ Хотят, чтобы серверы реже выключались
- ★ Хотят, чтобы приложение было доступно клиенту бесперебойно



## Нефункциональные требования

определяющие свойства, которые система должна демонстрировать, или ограничения, которые она должна соблюдать, не относящиеся к поведению системы. Например, производительность, удобство сопровождения, расширяемость, надежность, факторы эксплуатации.



## Time to Market (TTM)

время вывода продукта на рынок

## Назначение DevOps

сокращение Time To Market без потери  
нефункциональных требований

ТТМ важнее для разработчиков, нефункциональные требования важнее для сисадминов

## Решаем хронический конфликт



то есть получается такая система, что если ошибок нет, то все сразу уедет на прод, а если ошибки есть, то не пойдет, это и делает devops, разработчики сразу получают обратную связь, так как все это делается машиной а не людьми

- Создание контейнеров → 
- Автоматизация действий → 
- Настройка серверов → 
- Оркестрация контейнеров → 

Зачем нужен докер:

1

Упаковка любого приложения в контейнер

2

Возможность быстрого тестирования на ранних стадиях

3

Возможность локальной разработки

4

Повышение скорости применения

## Виртуализация и контейнеризация

### Зачем нужна виртуализация

- Повышение изоляции и уровня безопасности
- Распределение ресурсов
- Постоянная доступность
- Повышение качества администрирования

1) хочешь что-то спрятать, прячь в виртуалку и туда не попасть из другой виртуалки

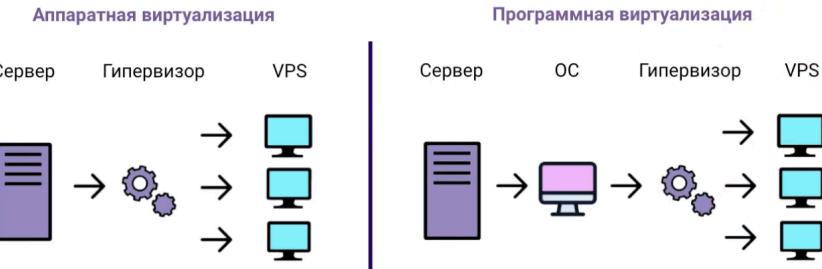
2) какую мощность дадим виртуалке, столько и будет и не сожрет больше чем дали

3) можно перенести легко на другой сервер

4) админить виртуалку легче чем железную тачку

## Виды виртуализации

- Программная
- Аппаратная
- Контейнерная

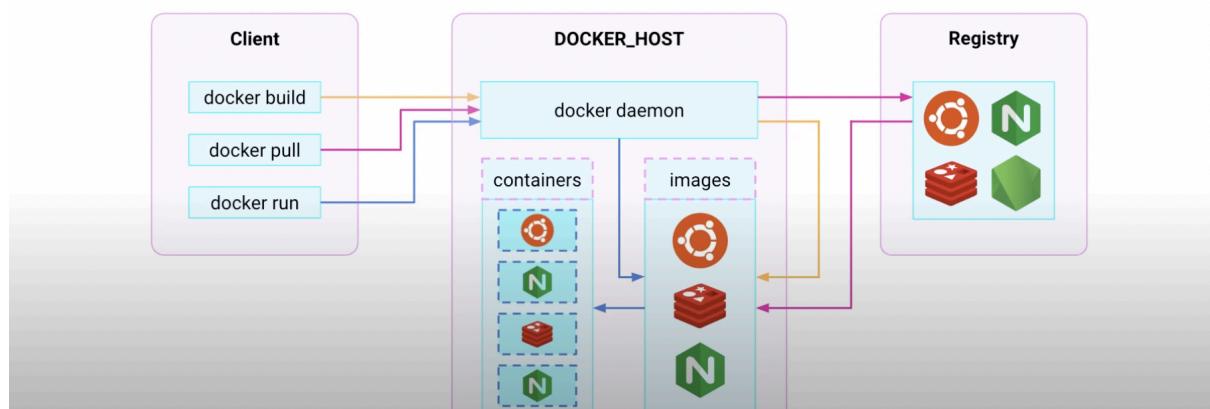


Контейнерная виртуализация - сейчас нет такого понятия. Это контейнеризация!!!!!!  
виртуалка = эмуляция

в контейнеризации ничего не эмулируется  
контейнер - окружение, запущенное поверх ядра  
гипервизор отвечает за эмулирование оборудования

на маке и винде нужно ставить docker desktop, который в себе содержит прослойку линуксовую, которая позволит работать с докером

## Концепция Docker



по умолчанию docker host это localhost, но можно поменять на адрес вообще другого компа/сервера, а у себя на компе писать только клиентские команды

images - образы, их них поднимаем сколько угодно контейнеров (локальное хранилище)

registry - хранилище образов (глобальное, там уже есть набор часто нужных)  
docker hub - общий registry, там также можно создавать свои репозитории

# Docker

Система контейнерной виртуализации и управления  
контейнерами с упакованным приложением

## Основные возможности

- Упаковка приложений и зависимостей в контейнеры
- Минимальное потребление ресурсов
- Скоростное развертывание
- Изоляция кода и процессов
- Простое масштабирование
- Удобный запуск

минимальное потребление ресурсов - использует ровно столько, сколько нужно, а в виртуалке столько, сколько выделили

## Из чего вообще строится docker-контейнер

- **Dockerfile**

Это основа будущего образа, содержит поэтапные инструкции

- **Docker-образ**

шаблон для создания Docker-контейнеров. Исполняемый пакет, содержащий все необходимое для запуска приложения. Docker-образ состоит из слоев.

Каждое изменение записывается в новый слой.

- **Docker-контейнер**

виртуализированная среда выполнения, в которой пользователи могут изолировать приложения от хостовой системы

## Dockerfile

- Основа будущего docker-образа
- Текстовый файл с последовательным описанием **инструкций** для сборки образа
- Каждая инструкция создает промежуточный слой образа
- Инструкции кешируются в образах
- Сам сборку делает **демон Docker**, а не Docker CLI

## Базовые команды docker-cli

- ↳ **docker build**: сборка из dockerfile и “контекста сборки”
- ↳ **docker images**: просмотр локально имеющихся образов
- ↳ **docker tag**: задать тэг для образа
- ↳ **docker ps**: просмотр имеющихся контейнеров
- ↳ **docker run**: запуск контейнера из образа
- ↳ **docker rm**: удалить контейнер
- ↳ **docker start/stop/restart**: управление контейнером
- ↳ **docker logs**: просмотр логов контейнера
- ↳ **docker pull/push**: скачать или закачать docker-образ в хранилище (registry)



контекст сборки - с чем делать (корень репозитория часто, в этом же корне репозитории лежит dockerfile, поэтому контекст обозначают точкой - текущая директория)

dockerfile - что делать

tag - имя образа, с командой run пишем tag а не id так как без тега мы просто забудем где какой образ(теги всегда писать)

tag состоит из repository и tag

A screenshot of a terminal window on a Mac OS X system. The window title is 'Work'. The command 'docker images' is being run. The output shows three Docker images:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	latest	760b7cbba31e	5 days ago	192MB
2048	latest	311890579be4	7 weeks ago	182MB
<none>	<none>	8e8e45a01455	7 weeks ago	1.2GB

Пример практики:

```
docker build -f Dockerfile -t deusops/ansible:240220 .
```

-f DockerFile - найти dockerfile

-t - тэг назначили

. - контекст

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
deusops/ansible	240220	f40ac8da6cf6	About a minute ago	480MB
deusops/ansible	latest	f40ac8da6cf6	About a minute ago	480MB
nginx	latest	760b7cbbab31e	5 days ago	192MB
postgres	16	0d774dc2e01c	7 days ago	453MB
2048	latest	311890579be4	7 weeks ago	182MB

docker ps -a - какие контейнеры запущены

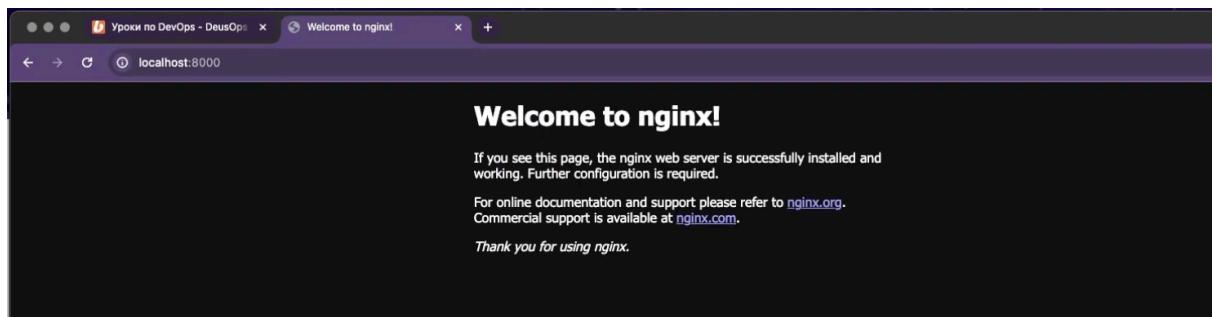
если просто запустить контейнер, то занимается фоном, ничего делать в терминале нельзя, а если завершить, то выйдем из контейнера

для этого ключ -d чтобы контейнер был в фоне

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
4dcf29a92f6d	nginx	"/docker-entrypoint..."	6 seconds ago	Up 5 seconds	0.0.0.0:8000->80/tcp	jovial_liskov

-р - указать порт

напишем в браузере localhost:8000



зайти в контейнер и выполнить команду какую-то:

total 76
drwxr-xr-x 1 root root 4096 Feb 20 17:53 .
drwxr-xr-x 1 root root 4096 Feb 20 17:53 ..
-rwxr-xr-x 1 root root 0 Feb 20 17:53 .dockerenv
lrwxrwxrwx 1 root root 7 Feb 11 00:00 bin -> usr/bin
drwxr-xr-x 2 root root 4096 Jan 28 21:20 boot
drwxr-xr-x 5 root root 340 Feb 20 17:53 dev
drwxr-xr-x 1 root root 4096 Feb 14 22:24 docker-entrypoint.d
-rwxr-xr-x 1 root root 1620 Feb 14 22:24 docker-entrypoint.sh
drwxr-xr-x 1 root root 4096 Feb 20 17:53 etc
drwxr-xr-x 2 root root 4096 Jan 28 21:20 home
lrwxrwxrwx 1 root root 7 Feb 11 00:00 lib -> usr/lib
drwxr-xr-x 2 root root 4096 Feb 11 00:00 media
drwxr-xr-x 2 root root 4096 Feb 11 00:00 mnt
drwxr-xr-x 2 root root 4096 Feb 11 00:00 opt
dr-xr-xr-x 260 root root 0 Feb 20 17:53 proc
drwx----- 1 root root 4096 Feb 20 17:56 root
drwxr-xr-x 1 root root 4096 Feb 20 17:53 run
lrwxrwxrwx 1 root root 8 Feb 11 00:00 sbin -> usr/sbin
drwxr-xr-x 2 root root 4096 Feb 11 00:00 srv
dr-xr-xr-x 12 root root 0 Feb 20 17:53 sys
drwxrwxrwt 2 root root 4096 Feb 11 00:00 tmp

вообще свой html запустили (-P - сам порт выберется)

```
apple:~/Work/deusops/university/101 docker run -v $(pwd)/html:/usr/share/nginx/html -d -P nginx 83d96bbda30f36450e76a1e0e8b548de192b91323a41882a64edbfbce3b59b81
```

## Контекст сборки

- **Контекст** – это набор файлов расположенных по пути или URL. Процесс сборки может обращаться к любым файлам в контексте, например, копировать их внутрь образа.
- **PATH-контекст**  
путь до директории с файлами. Чаще всего устанавливается значение точки – “.”
- **URL-контекст**  
В качестве URL могут быть переданы git-репозитории, tarball-архивы и текстовые файлы. Git-репозиторий может передавать ветки и тэги.

Инструкция **FROM** указывает базовый образ, на основе которого мы строим свою сборку

```
FROM <image>[:<tag>]  
  
# <image> – имя базового образа  
# <tag> – опциональный атрибут указывающий на версию образа
```

Примеры:

```
FROM ubuntu:22.04  
FROM python:3  
FROM nginx  
FROM quay.io/vektorlab/ctop  
FROM my_base_image:1.0
```

базовый образ один!!!

по умолчанию тэг latest

чтобы обновить устаревшие версии нужно делать pull, автоматически не получится

## Инструкция: LABEL

Инструкция **LABEL** задает метаданные для нашего образа:

```
LABEL <key>==<value> [<key>==<value> ...]  
  
# <key> - ключ  
# <value> - значение
```

Примеры:

```
LABEL maintainer="konstantin@deusops.com" version="1.0.3"
```

label - только для докерфайла, никуда не идет

## Инструкция: COPY

Инструкция **COPY** копирует файлы из “контекста сборки” в образ:

```
COPY <src> [<src> ...] <dest>  
# или  
COPY [<src>, ... <dest>]  
  
# <src> - файл или директория внутри build контекста  
# <dest> - целевая директория внутри контейнера
```

Примеры:

```
COPY start* /startup/  
COPY httpd.conf magicfile /etc/httpd/conf/
```

Инструкция Dockerfile COPY позволяет на этапе сборки контейнера скопировать в него с локальной машины любые файлы. Чаще всего это файлы приложения, конфигурационные файлы, скрипты, статическое содержимое: файлы стилей, изображения.

## Инструкция: ENV

Инструкция **ENV** задает переменные окружения в контейнере:

```
ENV <key> <value>

# <key> – имя переменной окружения
# <value> – присваиваемое значение
```

Примеры:

```
ENV LOG_LEVEL debug
ENV DB_HOST 10.10.50.2:3389
```

Инструкция **ARG** задает переменные окружения при сборке образа, которые может получить из командной строки:

```
ARG <key>[=<default value>]

# <key> – имя переменной окружения
# <default value> – значение по-умолчанию, опционально
```

Примеры:

```
ARG VERSION=latest
FROM ubuntu:${VERSION}
ARG BUILD
ENV BUILD=v1.0.0
RUN echo $BUILD
```

```
$ docker build \
  --build-arg VERSION=22.04 \
  --build-arg BUILD=v2.0.0 .
```

отличие env и arg: arg берет снаружи

пример 1) если параметр version не передастся, то будет latest

пример 2) ждем параметр build из вне, если не передастся, то берем из env

## Инструкции WORKDIR и USER

Инструкция **WORKDIR** задает рабочую директорию при сборке, а **USER** пользователя, от которого будут выполняться дальнейшие инструкции:

```
WORKDIR <path>
```

# <path> – путь внутри контейнера

```
USER <user>[ :<group>]
```

# или

```
USER <UID>[ :<GID>]
```

# <user> / <UID> – имя пользователя или UID

# <group> / <GID> – имя группы или GID (опционально)

Примеры:

```
WORKDIR /app
```

```
USER www-data
```

```
USER postgres
```

## Инструкция: RUN

Инструкция **RUN** задает команды, которые выполняются при **сборке** контейнера:

```
RUN <command>
```

# <command> – команда, которая будет выполнена при создании образа

Примеры:

```
RUN apt update && apt install python
RUN [ "bash", "-c", "rm", "-rf", "/tmp/dir"]
RUN [ "myscript.py", "argument1", "argument2"]
```

Инструкция **CMD** задает команду, которая выполняется при **старте** контейнера:

```
CMD <command>
```

# <command> – команда, которая будет выполнена при старте контейнера

Примеры:

```
CMD /start.sh
CMD [ "echo", "Hello World"]
CMD [ "python3", "manage.py", "runserver"]
```

- В одном файле Dockerfile может присутствовать лишь одна инструкция **CMD**. Если в файле есть несколько таких инструкций, система проигнорирует все кроме последней.

## Инструкция: ENTRYPPOINT

Инструкция **ENTRYPPOINT** задает команду, которая выполняется при **старте** контейнера:

```
ENTRYPPOINT <command>
# <command> – команда, которая будет выполнена при старте контейнера
```

Примеры:

```
ENTRYPPOINT exec top -b
ENTRYPPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
ENTRYPPOINT ["/bin/sh", "/docker-entrypoint.sh"]
```

cmd - это атрибут, который мы передаем в точку входа  
entrypoint - инструкция, которая запускается в контейнере, точка входа

## Как работают ENTRYPPOINT и CMD

Обе инструкции имеют две формы - **exec** и **shell**, при этом CMD имеет третью форму – в качестве параметров по-умолчанию для ENTRYPPOINT

	Без ENTRYPPOINT	ENTRYPPOINT exec_entry p1_entry	ENTRYPPOINT ["exec_entry", "p1_entry"]
Без CMD	не поддерживается	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

## Инструкция: VOLUME

Инструкция **VOLUME** позволяет указать точки для монтирования томов внутри образа и создает том с содержимым директории на хостовой машине:

```
VOLUME <dest> [<dest> ...]
# <dest> – директория монтирования для тома
```

Примеры:

```
VOLUME /app /db /data
# или
VOLUME [ "/var/www", "/var/log/apache", "/etc/apache2" ]
```

Этот инструмент отключает привязку данных к жизненному циклу контейнера, позволяя получить доступ к контейнерным данным в любой момент. Таким образом, сделанные в контейнерах записи остаются доступными после уничтожения содержавшего их контейнера и могут повторно использоваться в других. 29 дек. 2022 г.

Если немного упростить, то Docker volume — это просто папка хоста, примонтированная к файловой системе контейнера. Так как технически она больше не принадлежит контейнеру, то последний можно смело удалять, пересоздавать заново, снова прикручивать к нему хостовые папки, и ничего с данными внутри не случится. Если несколько способов, как этой функциональностью воспользоваться, и сегодня мы рассмотрим целых три из них.

## Инструкция: EXPOSE

Инструкция **EXPOSE** информирует, какие порты слушает сервис в запущенном контейнере. Используется, как средство коммуникации между тем, кто собирает образ и запускает контейнер. Не отвечает за реальное открытие портов.

```
EXPOSE <port>[/proto] [<port>/[<proto>] ...]  
# <port> – порт сервиса внутри контейнера  
# <proto> – tcp или udp, дополнительно
```

Примеры:

```
EXPOSE 5050  
EXPOSE 8080/tcp 3389/udp
```

## Инструкция: HEALTHCHECK

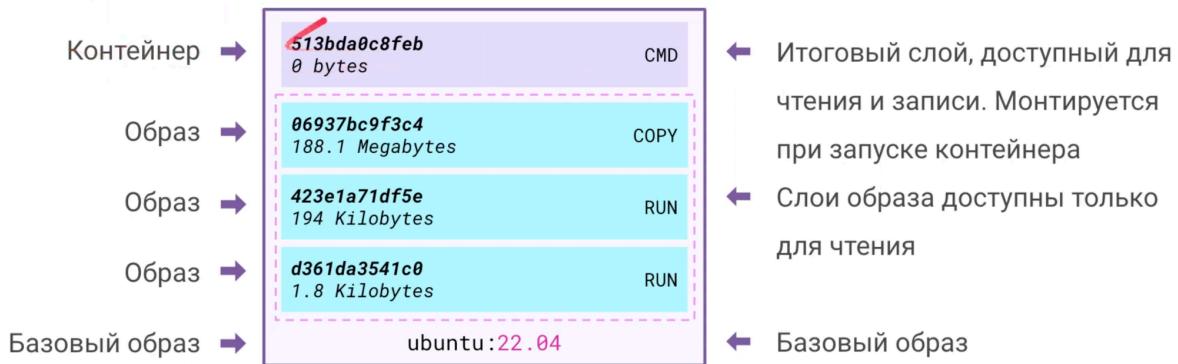
Инструкция **HEALTHCHECK** говорит docker как проверять, что контейнер всё еще работает.

```
HEALTHCHECK [OPTIONS] CMD command  
# OPTIONS – возможные параметры проверки:  
# --interval=DURATION (по-умолчанию: 30s)  
# --timeout=DURATION (по-умолчанию: 30s)  
# --start-period=DURATION (по-умолчанию: 0s)  
# --retries=N (по-умолчанию: 3)  
  
# <command> – команда, которая будет выполнена для проверки работоспособности
```

Примеры:

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

## Структура слоёв



## Кэширование

- Кэширование очень важно для реализации быстрых сборок
- **ADD, COPY** - файлы по умолчанию кешируются (в случае изменений файлов, кеш сбрасывается)
- Для остальных инструкций, включая **RUN**, проверяется только изменение параметров самой инструкции
- **RUN apt-get -y update** не проверяет обновления постоянно, только в первый раз
- Пропустить и перестроить кэш можно командой

```
docker build --no-cache
```

все слои после измененного слоя пересобираются, так как их хэш тоже поменяется  
так как **RUN apt-get -y update** инструкция не поменялась, то постоянно обновляться не будет, только при первом запуске или при очистке кэша

# Кэширование

- Сначала **COPY**, потом **RUN**:

<b>ENV</b> LOG_LEVEL INFO	# cache hit! ✓
<b>COPY</b> app /src/app	# cache miss! ✗
<b>RUN</b> apt install -y nginx php-fpm	# cache miss! ✗
<b>CMD</b> [ "/bin/app" ]	# cache miss! ✗

- Сначала **RUN**, потом **COPY**:

<b>ENV</b> LOG_LEVEL INFO	# cache hit! ✓
<b>RUN</b> apt install -y nginx php-fpm	# cache hit! ✓
<b>COPY</b> app /src/app	# cache miss! ✗
<b>CMD</b> [ "/bin/app" ]	# cache miss! ✗

то есть порядок должен быть логичный, в первом примере показан плохой вариант - любое изменение app заставит пересобираться nginx, хотя они вообще не связаны

## Уменьшение размера образа

- Используйте меньшее количество слоёв

<b>RUN</b> apt update	# one image!
<b>RUN</b> apt install -y nginx	# two image!
<b>RUN</b> apt update && \	
apt install -y nginx	# one image!

- Удаляйте неиспользуемые файлы, до их записи в образ

<b>RUN</b> apt update && \ apt install -y nginx	# one image!
<b>RUN</b> rm -rf /var/lib/apt/lists/*	# two image!
<b>RUN</b> apt update && \ apt install -y nginx && \ rm -rf /var/lib/apt/lists/*	# one image!

Иначе эти файлы навсегда останутся в образе

•

## Уменьшение размера образа

- Имеем скомпилированное приложение, которое не имеет зависимостей:

```
FROM ubuntu:20.04
COPY ./hello-world .
EXPOSE 8080
CMD ["/./hello-world"]
```

- Проверим размер образа созданного из такого Dockerfile:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-app	1.0	513bda0c8feb	15 seconds ago	194MB
ubuntu	20.04	d361da3541c0	8 days ago	188MB

- А нужен ли нам образ ОС? Он же в 30 раз больше самого приложения!

## Уменьшение размера образа

**scratch** – один из зарезервированных образов Docker, в котором ничего нет (пустой Dockerfile). В случае со **scratch**-образом, следующая инструкция создаст первый слой с файловой системой

```
FROM scratch
COPY ./hello-world .
EXPOSE 8080
CMD ["/./hello-world"]
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-app	2.0	423e1a71df5e	30 seconds ago	5.85MB
hello-app	1.0	513bda0c8feb	15 minutes ago	194MB

## Общие рекомендации

- Избегайте установки лишних пакетов и упаковки лишних данных в образы при сборке (*build context bloating*)
- Используйте связанные команды для **RUN**-инструкций
- Следим за последовательностью описания **Dockerfile**, избегаем **cache miss**
- Уменьшайте количество слоев
- Один контейнер – одна задача
- Чистите за собой
- Используйте multi-stage сборки

## Секретный слайд!

- У **docker build** есть параметр **--squash**, который в finale собирает все слои в один (как будто инструкции **Dockerfile** были выполнены в одном слое), так что можно сильно не заморачиваться.



## ПРАКТИКА

если скачать образ питона то это 1г(там уже с убунтой итп) но не всегда нужно это все есть легковесные версии alpine

python	3	d8be44680b2e	2 weeks ago	1.02GB
python	3-alpine	80f904359cf6	2 weeks ago	56.5MB

python:<version>-alpine

This image is based on the popular [Alpine Linux project](#), available in the [alpine](#) official image. Alpine Linux is much smaller than most distribution base images (~5MB), and thus leads to much slimmer images in general.

This variant is useful when final image size being as small as possible is your primary concern. The main caveat to note is that it does use `musl` libc instead of `glibc` and friends, so software will often run into issues depending on the depth of their libc requirements/assumptions. See [this Hacker News comment thread](#) for more discussion of the issues that might arise and some pro/con comparisons of using Alpine-based images.

To minimize image size, it's uncommon for additional related tools (such as `git` or `bash`) to be included in Alpine-based images. Using this image as a base, add the things you need in your own Dockerfile (see the [alpine](#) image description for examples of how to install packages if you are unfamiliar).

все образы нужно запустить до того как писать в dockerfile

```
Dockerfile+
1 FROM python:3-alpine
2
3 WORKDIR /app
4
5 COPY . /app
6
7 RUN pip install -r requirements.txt
8 RUN python manage.py migrate blog
9 #RUN python manage.py createsuperuser
10
11 CMD python3 manage.py runserver 0.0.0.0:8000
~
```

```
apple > ~ / W / d / u / d / 0301 > v main !1 ?1 > docker build .
```

улучшаем образ:

```
Dockerfile+
1 FROM python:3-alpine
2
3 WORKDIR /app
4
5 COPY requirements.txt /app/requirements.txt
6 RUN pip install -r requirements.txt --no-cache-dir
7 COPY . /app
8
9 RUN python manage.py migrate blog
10 #RUN python manage.py createsuperuser
11
12 ENTRYPOINT ["python", "3"]
13 CMD ["manage.py", "runserver", "0.0.0.0:8000"]
~
```

- 1) зависимости устанавливаем до сору . /app, так как в случае добавления каких-то файлов у нас раньше всегда зависимости перезапускались
- 2) добавили entrypoint для симпатичности
- 3) еще добавили ключ чтобы не было ключа по cache

### Dockerfile+

```
1 FROM python:3-alpine
2
3 WORKDIR /app
4
5 COPY requirements.txt /app/requirements.txt
6 RUN pip install -r requirements.txt --no-cache-dir
7 COPY . /app
8
9 RUN python manage.py migrate blog
10 #RUN python manage.py createsuperuser
11
12 EXPOSE 8000
13 VOLUME ["/app/db"]
14
15 ENTRYPOINT ["python", "3"]
16 CMD ["manage.py", "runserver", "0.0.0.0:8000"]
```

- ~ 1) добавили expose чтобы не гадали какой порт
- 2) добавили volume

все запускается и работает

```
apple: ~ /W/d/u/d/0301 > ⌘ p main !1 ?1 ➤ docker run -p 8000:8000 d699c9262a02
Watching for file changes with StatReloader
[27/Feb/2024 20:59:59] "GET / HTTP/1.1" 200 912
[27/Feb/2024 20:59:59] "GET /static/css/blog.css HTTP/1.1" 200 702
Not Found: /favicon.ico
[27/Feb/2024 20:59:59] "GET /favicon.ico HTTP/1.1" 404 2686
[27/Feb/2024 21:00:08] "GET /admin/login/?next=/ HTTP/1.1" 200 1905
```

создавать базу данных в момент билда нельзя, но если ее закомментировать, то работать ничего не будет

```

Dockerfile+
1 FROM python:3-alpine
2
3 WORKDIR /app
4
5 ARG user=admin
6 ARG password=pass
7 ARG email=admin@admin.com
8 ENV user $user
9 ENV password $password
10 ENV email $email
11
12 COPY requirements.txt /app/requirements.txt
13 RUN pip install -r requirements.txt --no-cache-dir
14 COPY . /app
15
16 #RUN python manage.py migrate blog
17 #RUN python manage.py createsuperuser
18
19 EXPOSE 8000
20 VOLUME ["/app/db"]
21
22 CMD sh init.sh && python3 manage.py runserver 0.0.0.0:8000
23
24 #ENTRYPOINT ["python3"]
25 #CMD ["manage.py", "runserver", "0.0.0.0:8000"]
~
```



```

init.sh+
1#!/bin/sh
2
3 # User credentials
4 #user=admin
5 #email=admin@example.com
6 #password=pass
7
8 file=db/db.sqlite3
9
10 python3 manage.py migrate
11
12 echo "from django.contrib.auth.models import User; User.objects.create_superuser('$user', '$email', '$password')"
   python3 manage.py shell || true
~
~
```

```

apple:~/W/d/u/d/0301 apple:~ 2 ?1 docker build -t blog:1 .
[+] Building 0.0s (10/10) FINISHED
=> [internal] load .dockerignore
=> => transferring context: 115B
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 512B
=> [internal] load metadata for docker.io/library/python:3-alpine
=> [1/5] FROM docker.io/library/python:3-alpine
=> [internal] load build context
=> => transferring context: 7.05kB
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY requirements.txt /app/requirements.txt
=> CACHED [4/5] RUN pip install -r requirements.txt --no-cache-dir
=> CACHED [5/5] COPY . /app
=> exporting to image
=> => exporting layers
=> => writing image sha256:7164a8754b52b3f2bf165b20847a3b4a38378db6b826146c08df9b9a7b4dd85b
=> => naming to docker.io/library/blog:1

What's Next?
View a summary of image vulnerabilities and recommendations - docker scout quickview
apple:~/W/d/u/d/0301 apple:~ 2 ?1 docker run -p 8000:8000 blog:1
```

теперь все работает, но уже и регистрируется юзер и можно писать на сайте

```

/app #
apple:~/W/d/u/d/0301 apple:~ 2 ?1 docker build --build-arg user=sanya -t blog:2 .
[+] Building 0.1s (10/10) FINISHED
```

```
view a summary of image vulnerabilities and recommendations → docker scout quick
Apple > ~/W/d/u/devops/0301 > ⌘ ⌘ main !2 ?1 docker run -p 8000:8000 blog:2
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
```

с user тоже все работает

единственное что можно создать, это создавать суперюзера только если нет еще

```
init.sh+
1 #!/bin/sh
2
3 file=/app/db/db.sqlite
4
5 python3 manage.py migrate
6
7 if [ $file -e ]; then
8   migrate
9   echo "from django.contrib.auth.models import User; User.objects.create_superuser('$user', '$email', '$password')"
| python3 manage.py shell || true
10 else
11   miggrate
```



## Многоступенчатая сборка

### Зачем нужен multi-stage build

- С компилируемыми языками, такими как Golang, люди, как правило, создают свои Dockerfile из образа Golang SDK, добавляют исходный код, выполняют сборку, а затем отправляют ее в Docker Hub. К сожалению, размер полученного образа был довольно большим - не менее 670 мегабайт

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
go-app-full	1.7.3	ef15416724f6	4 months ago	864MB
go-app-light	1.0	513bda0c8feb	15 seconds ago	13.8MB

- Непродуктивно хранить весь SDK в контейнере запускаемого приложения

## Как уменьшить размер образа

- Получить базовый образ Golang SDK
- Добавить исходный код
- Получить готовый бинарный файл
- Получить легковесный базовый образ **Scratch** или **Alpine**
- *Перенести бинарный файл из одного образа в другой*
- Разместить полученный легковесный образ с приложением в registry

## Как уменьшить размер образа

- Получить базовый образ Golang SDK
- Добавить исходный код
- Получить готовый бинарный файл
- Получить легковесный базовый образ **Scratch** или **Alpine**
- *Перенести бинарный файл из одного образа в другой*
- Разместить полученный легковесный образ с приложением в registry

## Как выполнить multi-stage build

- Пример многостадийной сборки

```
FROM golang:1.16 AS builder
WORKDIR /code/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /code/app .
CMD [ "./app" ]
```

- На выходе получаем легковесный образ на базе alpine

## Возможности стейджирования

- Скопировать файл из публичного образа

```
COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf
```

- Использовать предыдущий stage как новый

```
FROM alpine:latest AS builder
RUN apk --no-cache add build-base

FROM builder AS build-stage1
RUN g++ -o /binary source.cpp
```

- Управление через ARG

```
FROM alpine
ARG src=stage1
COPY --from=build-${src} . .
```

## Docker-compose

### Проблематика

- Одно приложение состоит из нескольких контейнеров
- Один контейнер зависит от другого
- Порядок запуска имеет значение
- docker build/run/create – много и нигде не хранится

## Docker-compose!

- Отдельная утилита
- Декларативное описание Docker инфраструктуры в YAML-формате
- Управление многоконтейнерными приложениями
- docker-compose.yml – конфигурация проекта

## docker-compose.yml

```
version: "3"
services:
  mongo_db:
    image: mongo:3.2
    volumes:
      - db:/data/db
    networks:
      - reddit
  ui:
    build: ./ui
    image: ${USERNAME}/ui:1.0
    ports:
      - 9292:9292/tcp
    networks:
      - reddit
  volumes:
    db:
      sample_volume:
...
...
```

## Сети

```
networks:
  reddit:
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 172.16.238.0/24
```

## Работа с окружениями

- Один docker-compose.yml на все окружения (под проект или приложение)
- Файл docker-compose.override.yml с:
  - Произвольными точками монтирования для быстрого изменения кода внутри контейнера
  - ENV переменными, характерными для окружения
  - Заменой command инструкций для образов
  - Перезаписью выводимых наружу портов

## Healthcheck

```
healthcheck:  
  test: ["CMD", "curl", "-f", "http://localhost:5000/healthcheck"]  
  interval: 1m30s  
  timeout: 10s  
  retries: 3
```

## Типичный сценарий использования Docker Compose

Docker Compose — это, в умелых руках, весьма мощный инструмент, позволяющий очень быстро развёртывать приложения, отличающиеся сложной архитектурой. Сейчас мы рассмотрим пример практического использования Docker Compose, разбор которого позволит вам оценить те преимущества, которые даст вам использование Docker Compose.

Представьте себе, что вы являетесь разработчиком некоего веб-проекта. В этот проект входит два веб-сайта. Первый позволяет людям, занимающимся бизнесом, создавать, всего в несколько щелчков мышью, интернет-магазины. Второй нацелен на поддержку клиентов. Эти два сайта взаимодействуют с одной и той же базой данных.

Ваш проект становится всё популярнее, и оказывается, что мощности сервера, на котором он работает, уже недостаточно. В результате вы решаете перевести весь проект на другую машину.

К сожалению, нечто вроде Docker Compose вы не использовали. Поэтому вам придётся переносить и перенастраивать сервисы по одному, надеясь на то, что вы, в процессе этой работы, ничего не забудете.

Если же вы используете Docker Compose, то перенос вашего проекта на новый сервер — это вопрос, который решается выполнением нескольких команд. Для того чтобы завершить перенос проекта на новое место, вам нужно лишь выполнить кое-какие настройки и загрузить на новый сервер резервную копию базы данных.

### 5. Сборка проекта

После того, как в `docker-compose.yml` внесены все необходимые инструкции, проект нужно собрать. Этот шаг нашей работы напоминает использование команды `docker build`, но соответствующая команда имеет отношение к нескольким сервисам:

```
$ docker-compose build
```

### 6. Запуск проекта

Теперь, когда проект собран, пришло время его запустить. Этот шаг нашей работы соответствует шагу, на котором, при работе с отдельными контейнерами, выполняется команда `docker run`:

```
$ docker-compose up
```

## Полезные команды

Рассмотрим некоторые команды, которые могут вам пригодиться при работе с Docker Compose.

Эта команда позволяет останавливать и удалять контейнеры и другие ресурсы, созданные командой `docker-compose up`:

```
$ docker-compose down
```

Эта команда выводит журналы сервисов:

```
$ docker-compose logs -f [service name]
```

Например, в нашем проекте её можно использовать в таком виде: `$ docker-compose logs -f [service name]`.

С помощью такой команды можно вывести список контейнеров:

```
$ docker-compose ps
```

Данная команда позволяет выполнить команду в выполняющемся контейнере:

```
$ docker-compose exec [service name] [command]
```

Например, она может выглядеть так: `docker-compose exec server ls`.

Такая команда позволяет вывести список образов:

```
$ docker-compose images
```

В вашем `docker-compose` файле указаны два порта для сервиса `server`:

```
yaml  
  
ports:  
  - 1234:1234
```

 Копировать код

Это синтаксис для перенаправления порта между хост-машиной и контейнером. Первый порт (слева от двоеточия) — это порт хост-машины, а второй порт (справа от двоеточия) — это порт контейнера.

Вот почему это сделано:

- Порт на контейнере (правый 1234):** Это порт, на котором серверное приложение внутри контейнера будет слушать запросы. В вашем случае, серверное приложение, которое запускается командой `python ./server.py`, настроено слушать на порту 1234.
- Порт на хост-машине (левый 1234):** Это порт на вашей локальной машине (или на сервере, где запускается Docker), который будет перенаправлен к порту контейнера. Это позволяет вам обращаться к вашему серверному приложению с хост-машины, используя указанный порт.

Когда вы пишете `1234:1234`, это значит, что порт 1234 на хост-машине будет направлен к порту 1234 на контейнере. Таким образом, если вы отправите запрос на `localhost:1234` на вашем компьютере, он будет перенаправлен к приложению, работающему на порту 1234 внутри контейнера.

## Пример использования

Если бы вы хотели, чтобы хост-машина использовала другой порт для доступа к вашему приложению, вы могли бы изменить первую цифру. Например, запись:

```
yaml
ports:
  - 8080:1234
```

означала бы, что обращения к `localhost:8080` на хост-машине будут перенаправляться к `localhost:1234` внутри контейнера.

### Важные моменты:

- **Смысл перенаправления:** Это удобно, если порт 1234 на хост-машине уже занят, и вам нужно использовать другой порт, или если вы хотите использовать стандартные порты для различных приложений.
- **Обеспечение доступа:** Перенаправление портов также важно для доступа к приложениям, работающим внутри контейнеров, из внешнего мира (вне Docker).



Перенаправление портов между хост-машиной и контейнером необходимо по нескольким причинам:

### 1. Доступ к приложениям внутри контейнеров

Контейнеры изолированы друг от друга и от хост-машины. Чтобы получить доступ к приложению, запущенному внутри контейнера, снаружи контейнера (например, из браузера на вашем компьютере или с другой машины), необходимо настроить перенаправление портов. Это позволяет маршрутизировать трафик, поступающий на определенный порт хост-машины, в соответствующий порт внутри контейнера.

### 2. Избежание конфликтов портов

На хост-машине может быть несколько контейнеров, каждый из которых слушает на одном и том же порту внутри своих изолированных окружений. Перенаправление портов позволяет каждому контейнеру иметь уникальный внешний порт на хост-машине, предотвращая конфликты. Например, два разных контейнера могут оба слушать на порту 80 внутри себя, но через перенаправление портов они будут доступны на хост-машине через разные порты:

```
yaml
```

Копировать код

```
services:  
  web1:  
    build: ./web1  
    ports:  
      - "8080:80"  
  web2:  
    build: ./web2  
    ports:  
      - "8081:80"
```

В этом примере оба контейнера слушают на порту 80, но снаружи доступны через порты 8080 и 8081 соответственно.

<https://habr.com/ru/companies/ruvds/articles/450312/>

## МОДЕЛИ ВЕТВЛЕНИЕ И КОНЦЕПЦИЯ CI/CD

### Зачем это уметь

1

Хранение и версионирование любого кода

2

Организация работы в командах разработки

3

Понимание механик работы фундаментальных инструментов разработки

4

Понимание организации процессов доставки и развертывания программного обеспечения

## Знакомство с Git

### Система контроля версий

- Единый источник правды
- Распределенная система хранения кода
- Хранение истории проекта
- Основа для коллективной работы – возможность “отпочковать” свою версию и потом влить её обратно
- Основа для DevOps-практик

# Распределенная система хранения кода

- Полноценная локальная копия проекта
- Резервная копия
- Можно работать оффлайн
- Скорость

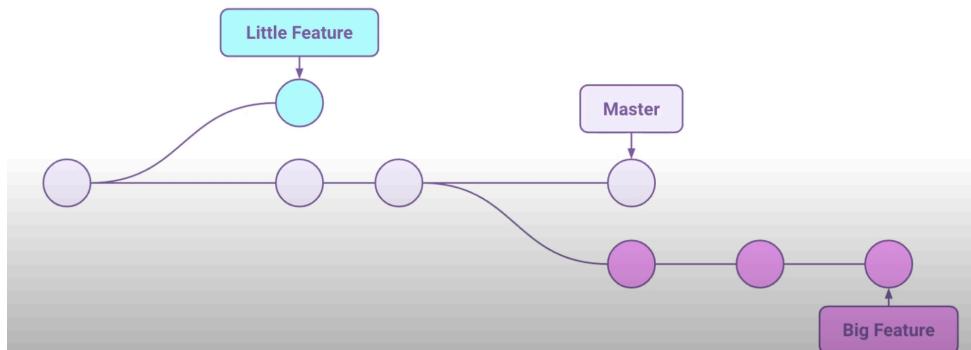
Локальная копия хранит все комментарии, кто что делал и тд

## Ветвление

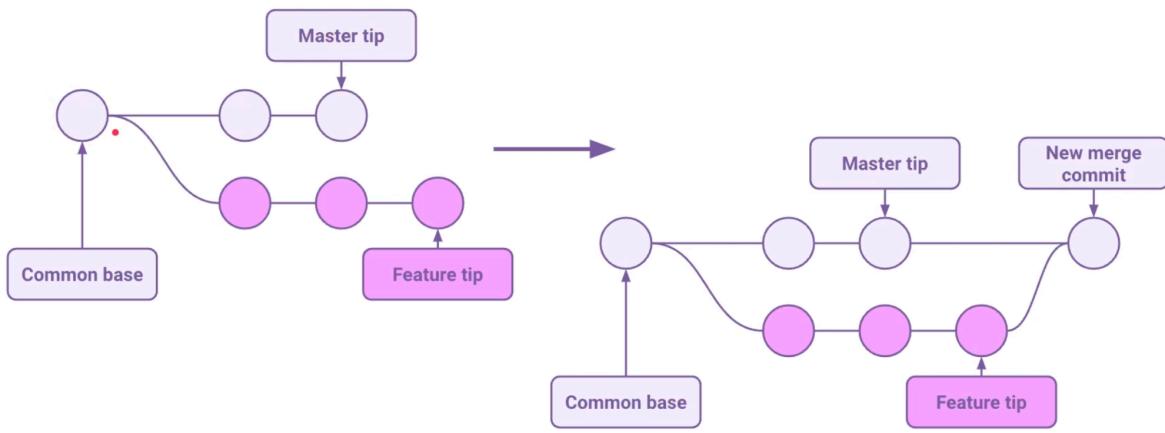
- Основная ветка – **main**
- Репозиторий может содержать множество веток, в которых могут быть отличия между файлами
- На основе любой ветки можно сделать новую ветку
- Любую ветку можно соединить с любой другой веткой
- Можно посмотреть разницу файлов и решить конфликты

Главная ветка - всегда работающее, то что в любой момент можно показать заказчику(то что уже проверено и уже на проде)

### Ветвление



## Ветвление

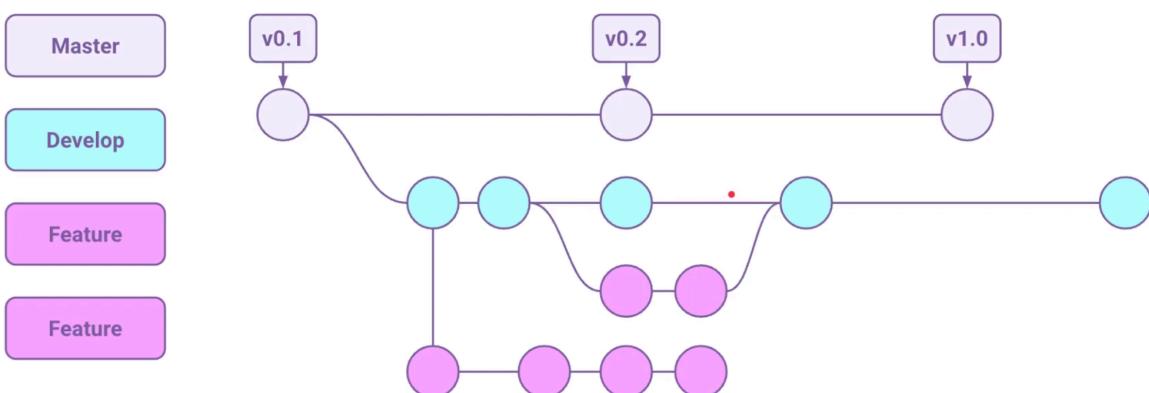


11

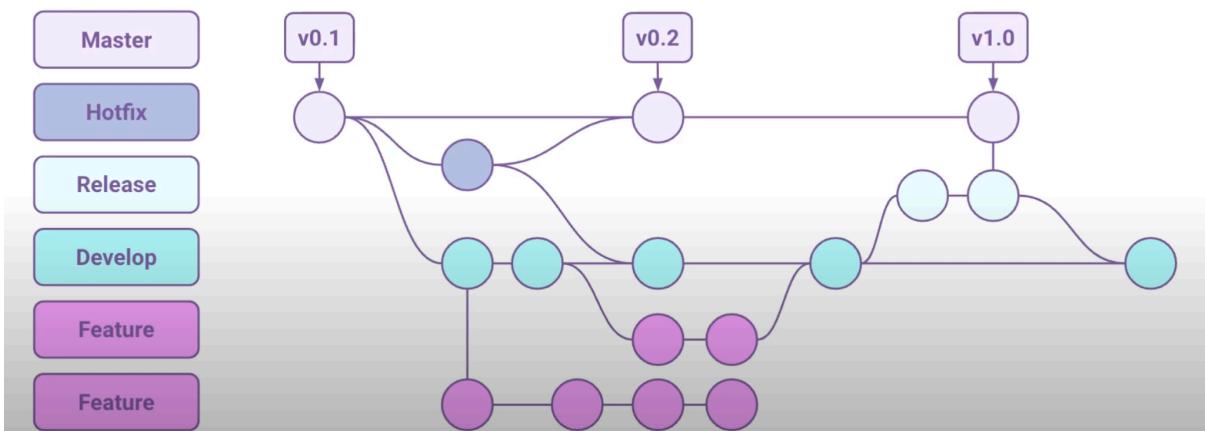
Модель ветвления GitFlow - самая популярная:

- 1) одна главная ветка master(main).
- 2) В master есть тэги у коммитов - это версия
- 3) обновление мастера сразу влечет обновление прода
- 4) разработка ведется в ветке development (название просто выбрали). Разрабатывать в тупую в ней не верно, так как людей много. Разбранчиваться сразу от мастер пропустив development тоже нельзя, так как несколько фич надо будет объединить, протестировать, а только потом влиять в мастере. То есть мастер отображает самую новую и рабочую версию, а development самую новую и возможно с багами

## Ветвление



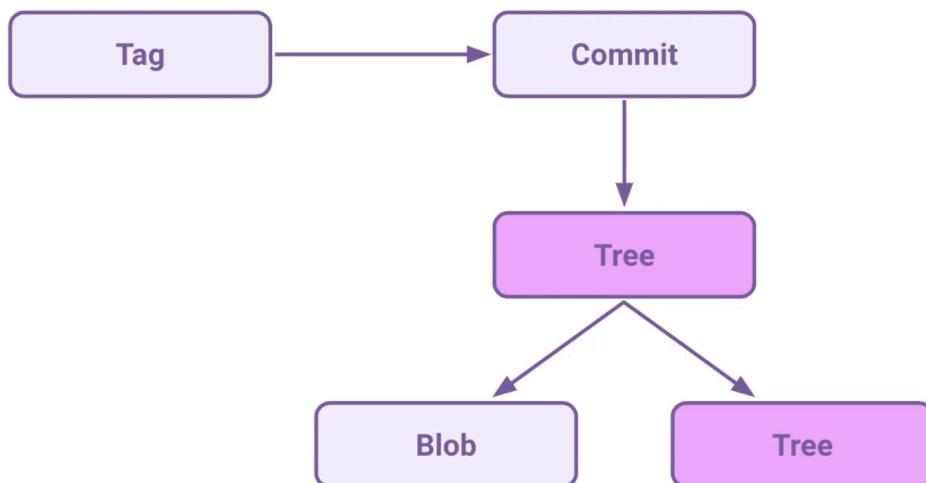
## Ветвление



Нашли баг на проде - делается ветка hotfix. Исправили и замержили в мастер, теперь дев ветка отстает от мастер, а она всегда должна быть впереди. Поэтому в дев вмерживаем тоже.

Если фича работает, то она вкатывается в дев. В дев тоже тестируем, так как у нас там есть изменение с hostfix, которое было после ответвления фичи. Если все окей, то можно на прод. Но для начала нужно создать релизную ветку. На релизной еще протестировали (например перекинули 10% пользователей на эту обнову и если им не понравилось, то не выкатились). Если все ок, то с релизной мерджим в мастер. и в дев, чтобы дев не отставал.

## Структура хранимых данных



## Blob

- Базовая единица хранения данных в Git
- Хранит снимок содержимого файла
- В качестве имени объекта берется хеш

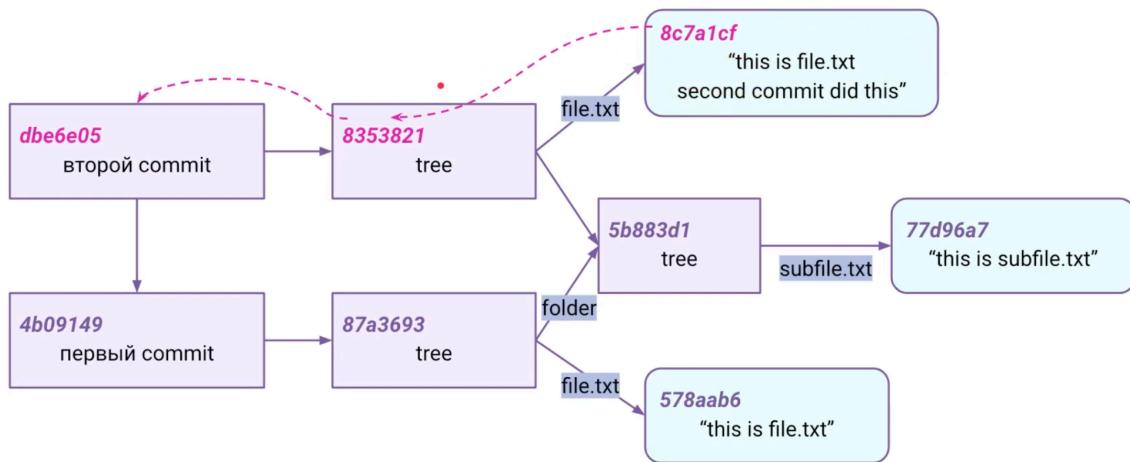
## Tree

- Деревья содержат информацию о блобах и других поддеревьях
- Решают проблему хранения имён файлов и их группировки

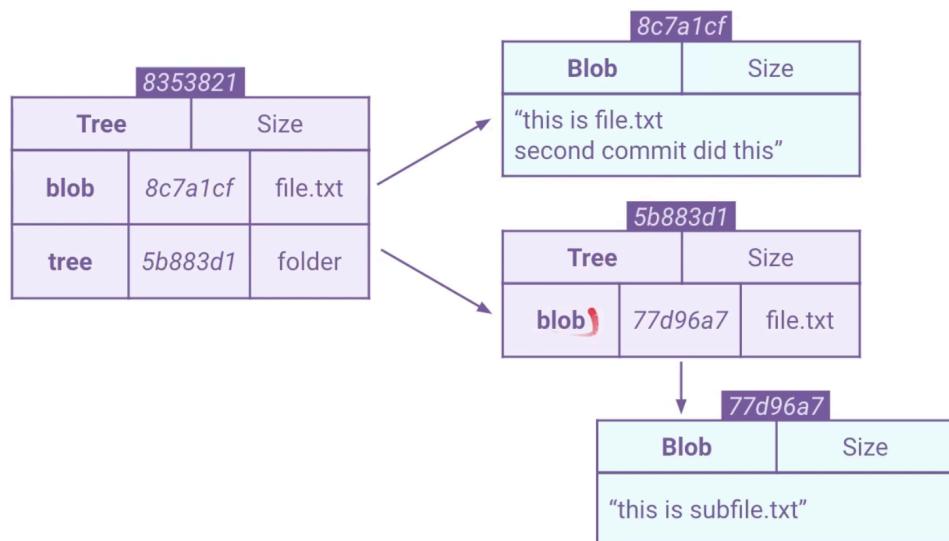
## Commit

- Содержит информацию об авторе и дате
- Содержит сообщение о коммите
- Ссылка на дерево
- Информация о родительском коммите

## Commit



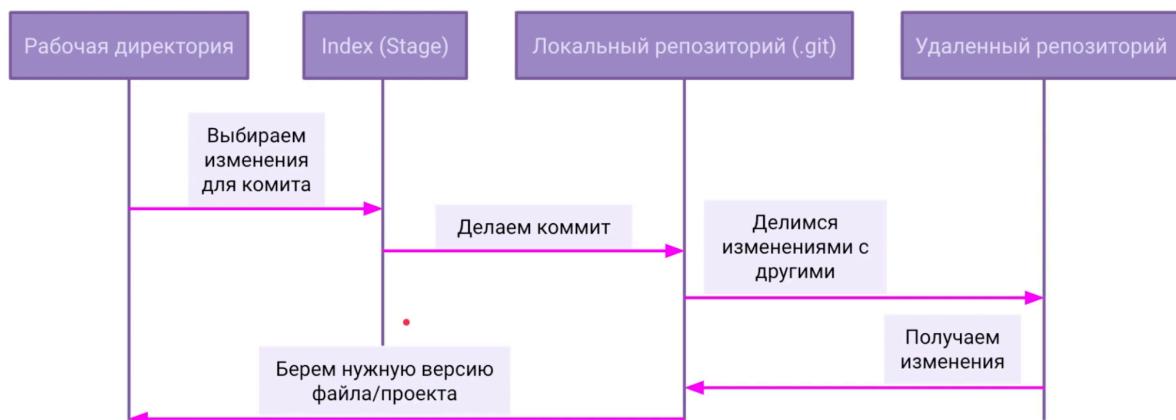
## Commit



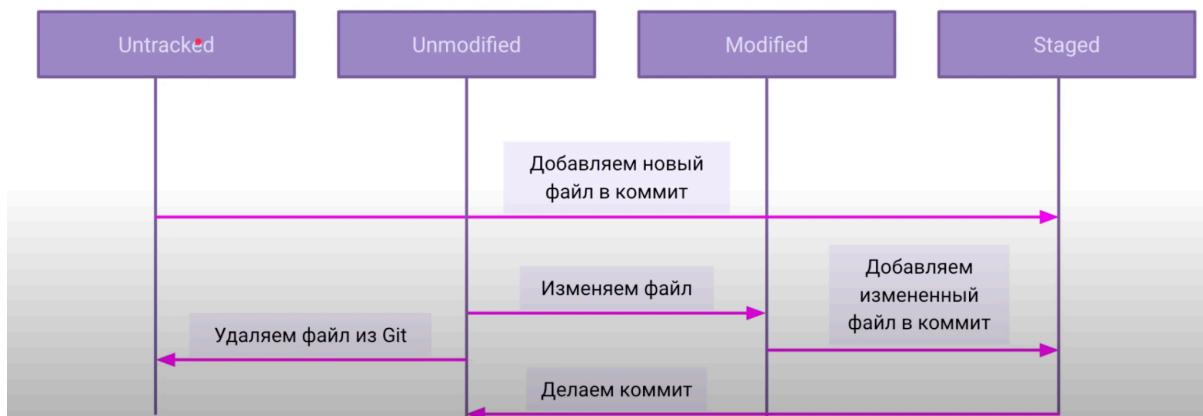
## Tags / Тэги

- Ярлык
- Указывает на определенный коммит
- Аннотированные тэги: могут содержать имя, дату и данные – хранятся как полноценные объекты

## Работа с Git: состояние коммитов



## Работа с Git: состояние файлов



## Работа с репозиторием

`git clone / git init`

скопировать удалённый  
репозиторий или создать  
локальный

`git pull / git push`

подгрузить изменения из  
интернета и отправить  
изменения в удалённый  
репозиторий

`git tag`

протегировать  
КОММІТ

## Работа с ветками

`git branch`

создать или удалить ветку

`git checkout`

переключиться на ветку

`git merge`

соединить несколько  
веток в одну

## Работа с файлами

`git add <file>`  
`git rm <file>`

добавить,  
удалить файл в индекс

`git mv <file>`

переместить файл или  
директорию

`git commit`

зарегистрировать коммит

## Работа с историей

`git status / git show`

посмотреть состояние  
репозитория и объектов

`git log`

показать логи коммитов

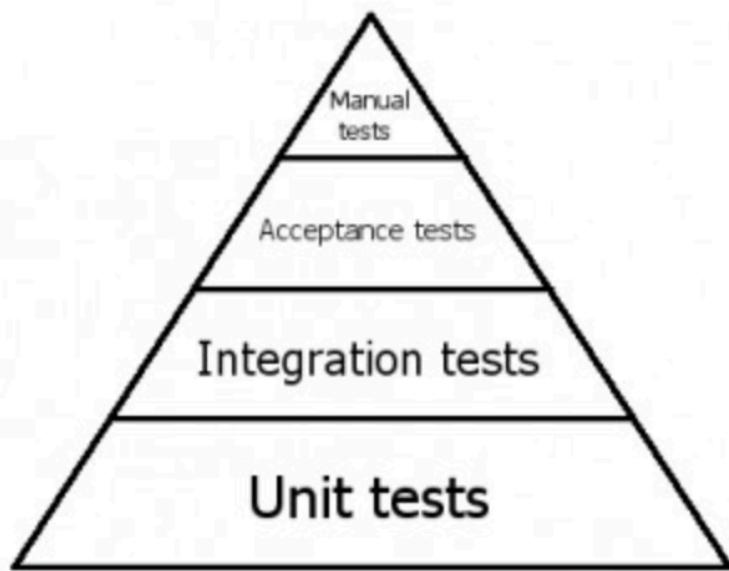
`git diff`

отразить разницу текущего  
состояния и исходного

## CI/CD

**Модульное тестирование, или юнит-тестирование** (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.



**Acceptance tests:** (то есть уже не функции тестировать, а прям в игру поиграть и проверить)

Существует несколько причин, по которым необходимо проводить приёмочное тестирование:

- Получить уверенность в продукте перед релизом.
- Убедиться в правильной работе продукта.
- Убедиться, что продукт соответствует текущим рыночным стандартам и составляет достойную конкуренцию другим подобным продуктам на рынке.

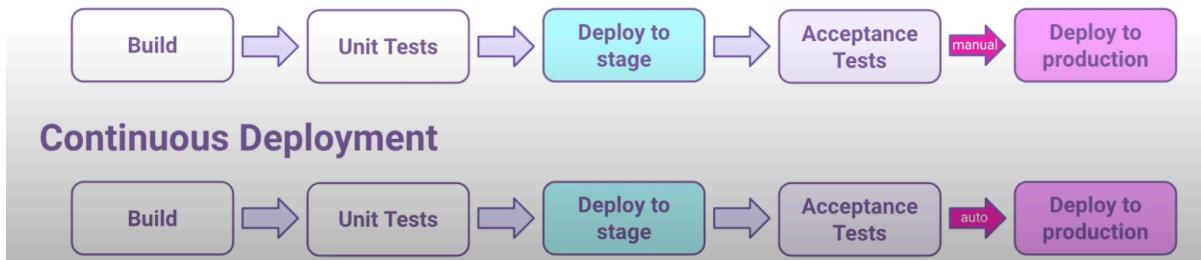
**Deploy to stage** - разворачивать на сервер. Это процесс, в ходе которого приложение или система готовится и запускается на целевой платформе или инфраструктуре для пользователей.

В итоге все можно автоматизировать и если все ок, то каждый следующий этап будет идти друг за другом. Если нет то будет посыпаться сообщение.

### Continuous Integration



### Continuous Delivery



### Continuous Deployment



Момент когда разработчики сделали фичу и это оказалось в самой новой ветке dev мы автоматизировали и никакой из этих этапов никого не ждет

Процесс когда фича ветка въезжает в dev - интеграция

И только потом уже раскатывается окружение близкое к проду и нужно все проверять. И это тоже научились делать автоматизировано (разворачивать окружение, чтобы тестировщики не трогали разрабов, а все само разворачивалось, а они уже тестировали)

Приемочные тесты тоже можно автоматизировать (селениум и тд) - то есть оптимизация прям до мастера

Далее нужно самим деплоить на прод, но по факту мы уже деплоили, просто на дев и тестовое окружение. И это тоже автоматизируем

Финал работы CI - build (готовое к запуску приложение)  
CD - это только деплоить на прод (не имеет отношения к исходному коду)

то есть CI берет исходники, собирает build, CD берет build и разворачивает на проде

Есть два вида CD, отличие лишь в том чтобы отправить на прод - руками или автоматизировано.

### Continuous Delivery vs Continuous Deployment

- В случае **Continuous Deployment** каждый следующий шаг, будет выполнен автоматически если предыдущий был успешный, включая деплой на Production.
- Если же у вас **Continuous Delivery**, то шаги будут выполняться автоматически только в безопасной среде, а перед деплоем на Production пайплайн остановится и будет ждать ручного подтверждения. Механизм, как это будет реализовано может быть разным. От самого простого, когда ответственный человек должен зайти в пайплайн и нажать кнопку Next, до интерактивного бота с кнопками в корпоративном мессенджере.

### Преимущества CD (любого) - малый TimeToMarket:

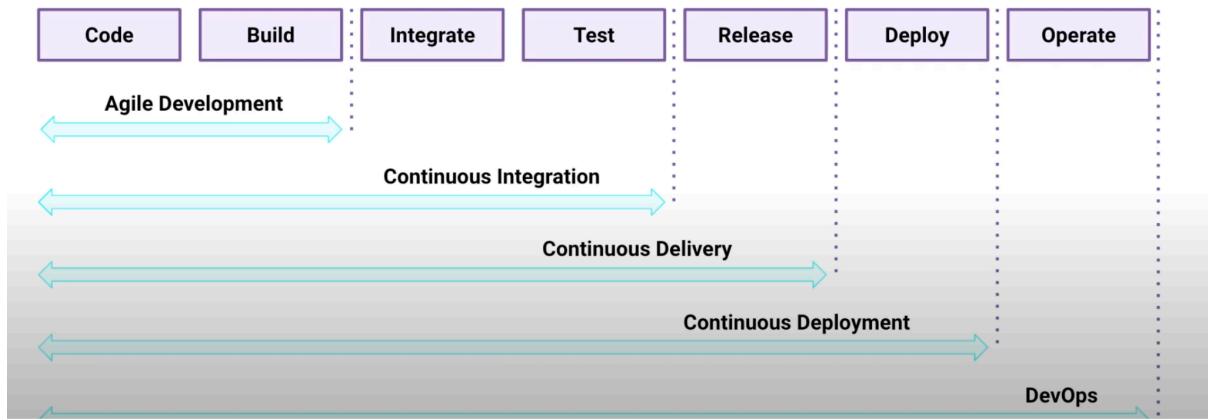
Это позволяет получить сразу отклик от клиента и, при необходимости, сделать некоторые изменения

- Внесение нового функционала в back-end для проверки совместимости с системой
- Быстрое реагирование на потребности рынка
- Возможность подстраивания под изменение бизнес-стратегии
- Низкое количество потенциальных ошибок

Continuous Integration	Continuous Delivery	Continuous Deployment
Необходимы автотесты на каждую новую фичу или баг-фикс	Тестами должен быть покрыт достаточный объём кода	Качество тестов влияет на качество готового продукта
Из-за ранней регрессии меньше багов попадает на продакшн	Развёртывание должно быть автоматизировано, обновления должны быть выпущены вручную	Развёртывание автоматизировано и триггеры настроены на каждое изменение
Разработчики должны мерджиться как можно чаще (минимум раз в день)	Команда может использовать фича-тоглинг	Фича-тоглинг является неотъемлемой частью больших изменений

Фича-тоглинг - коммитишь, но функционал выключен. То есть есть в проде, но выключен. Потом потестируем, включим на проде и все будет работать

## Как всё это работает вместе



Где цена ошибки велика (банки например), там используют Continuous Delivery. Также в онлайн играх, чтобы не было такого, что ты играешь и хоп обнова. Релизы ставятся в определенное время.

девопс - это когда все+operate тоже хорошо автоматизирован, а не так, что руками на месте чинить когда тормозит или еще чего

## GITLAB VS GITHAB

GitLab имеет свои особенности и преимущества, которые могут быть важными для некоторых проектов и команд разработчиков. Например, GitLab предоставляет больше возможностей для управления правами доступа, имеет интеграцию с CI/CD и другими инструментами разработки.

GitLab и GitHub на самом деле очень похожи и отличаются несущественно, но есть несколько ключевых отличий:

- GitLab отличается от GitHub тем, что это проект с открытым исходным кодом, что позволяет сообществу разработчиков улучшать и расширять его функциональность. В то же время, только внутренние разработчики могут улучшать и дополнять платформу GitHub.
- Существует второе отличие GitLab от GitHub, заключающееся в том, что начиная с двух тысяч восемнадцатого года компания Microsoft стала овнером GitHub. Это вызвало неоднозначную реакцию в сообществе разработчиков.
- Тем не менее, несмотря на это, популярность GitHub остается выше, чем у GitLab. GitHub был создан в 2008 году и не имел конкурентов до появления GitLab в 2011 году, который на старте был не очень активный.
- Гитлаб и Гитхаб обладают схожими возможностями для хранения и управления Git-репозиториями, а еще для работы в команде. Однако, каждая платформа имеет свои уникальные особенности плюсы, и конечный выбор между ними должен опираться на конкретные потребности и задачи Dev Teams.

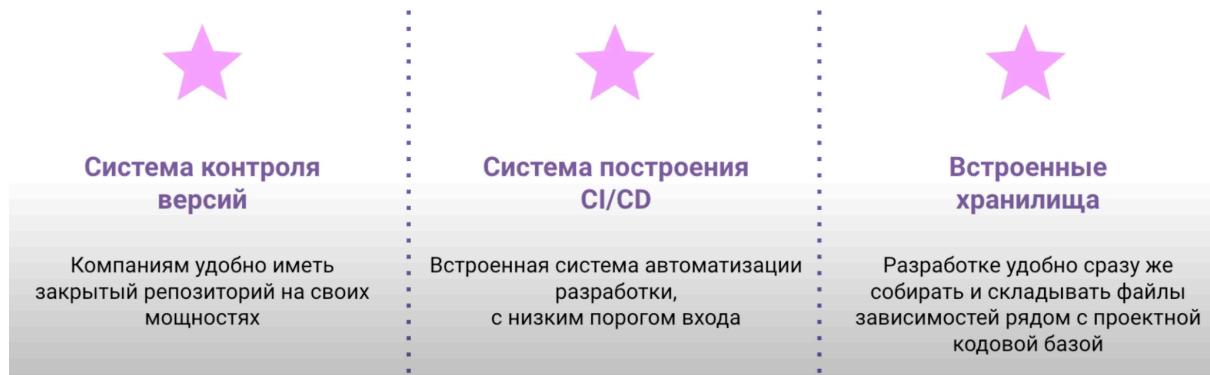
[https://softlist.kz/ru/news/gitlab-vs-github--zachem-ispolzovat-git-repozitoriy-i-cto-luchshe-ispolzova#:~:text=%D0%9D%D0%B0%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80%2C%20GitLab%20CI%2FCD%20%D0%BF%D1%80%D0%B5%D0%B4%D0%BE%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D1%8F%D0%B5%D1%82,GitHub%20Actions%20%E2%80%93%20%D1%8D%D1%82%D0%BE%20%D0%BF%D0%BB%D0%B0%D1%82%D0%BD%D1%8B%D0%B9%20%D1%81%D0%B5%D1%80%D0%B2%D0%BB%D1%8F">https://softlist.kz/ru/news/gitlab-vs-github--zachem-ispolzovat-git-repozitoriy-i-cto-luchshe-ispolzova#:~:text=%D0%9D%D0%B0%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80%2C%20GitLab%20CI%2FCD%20%D0%BF%D1%80%D0%B5%D0%B4%D0%BE%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D1%8F%D0%B5%D1%82,GitHub%20Actions%20%E2%80%93%20%D1%8D%D1%82%D0%BE%20%D0%BF%D0%BB%D0%B0%D1%82%D0%BD%D1%8B%D0%B9%20%D1%81%D0%B5%D1%80%D0%B2%D0%BB%D1%8F%D0%B5%D1%82.](https://softlist.kz/ru/news/gitlab-vs-github--zachem-ispolzovat-git-repozitoriy-i-cto-luchshe-ispolzova#:~:text=%D0%9D%D0%B0%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80%2C%20GitLab%20CI%2FCD%20%D0%BF%D1%80%D0%B5%D0%B4%D0%BE%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D1%8F%D0%B5%D1%82,GitHub%20Actions%20%E2%80%93%20%D1%8D%D1%82%D0%BE%20%D0%BF%D0%BB%D0%B0%D1%82%D0%BD%D1%8B%D0%B9%20%D1%81%D0%B5%D1%80%D0%B2%D0%BB%D1%8F)

GitHub делает упор на высокую доступность и производительность своей инфраструктуры и делегирует другие сложные функции сторонним инструментам. GitLab, наоборот, фокусируется на включении всех функций на одной проверенной и хорошо интегрированной платформе; он обеспечивает все для полного жизненного цикла DevOps под одной крышей. Что касается популярности, GitHub определенно превосходит GitLab.

В GitLab меньше разработчиков внедряют на платформу открытые исходные коды. Кроме того, что касается цен, GitHub стоит дороже, что делает его неподходящим для пользователей с небольшим бюджетом.



## Что это такое?



## Gitlab CI - встроенная в GitLab система управления пайплайнами CI/CD

ГитЛаб под каждый репо создает свое хранилище, как пакетный, так и для образов докера

The screenshot shows the GitLab interface with the following annotations:

- Git-репозиторий**: Points to the repository sidebar and the repository list.
- Кодовая база**: Points to the list of files in the repository, including `app`, `bootstrap`, `config`, `database`, `dockerfiles`, `messaging/config`, `public`, `resources`, `routes`, `storage`, and `stubs`.
- Автоматизация проекта**: Points to the CI/CD pipeline section, which includes the `cicd` tab and the `core` pipeline.
- Хранилища**: Points to the packages and registries section, which lists `PHP`, `SERVICES`, `CORE`, and `Repository`.

если есть доступ к репо, то есть доступ к registry который с ним идем. Это нужно, чтобы разраб который запушил докер образ, мог в момент деплоя его спулить

Для каждого проекта свой registry

Доступ можно получать к хранилищу не только по учетной записи, но и по CI -токену (генерируется в момент запуска CI)

## Что такое Gitlab Container Registry

Дополнительная функция гитлаба, которая позволяет хранить Docker-образы для проектов, с упрощенным доступом к ним

- Для каждого проекта - свой реестр
- Доступ по учетным данным в Gitlab
- Доступ через CI по токену

## Что такое Gitlab Registry

Container Registry

5 Image repositories | Expiration policy will run in about 23 hours

With the GitLab Container Registry, every project can have its own space to store images.

Image Repositories

- vtb-web1/lib-core/lib-core-pschannel 35 Tags
- vtb-web1/lib-core/lib-core-auth 31 Tags
- vtb-web1/lib-core/lib-core-soap 5 Tags
- vtb-web1/lib-core/lib-core-soapui 2 Tags
- vtb-web1/lib-core/lib-core-superauth 2 Tags

Для проекта можно иметь несколько разных образов, например, по одному на компонент

Каждый образ можно (даже нужно) отдельно версионировать разными тэгами

Пайплайн — это регламентированный и документированный процесс выполнения типовых задач.

## Основные характеристики

- **Непрерывность процесса.** Пайплайн обеспечивает потоковый характер работы — сотрудники последовательно выполняют свои задачи, передавая результат друг другу. Это повышает скорость и эффективность разработки.
- **Гибкость.** Пайплайн легко адаптируется к изменениям — новые задачи добавляются в общий конвейер работ. Это помогает оперативно реагировать на меняющиеся требования.
- **Контроль качества.** На каждой стадии происходит проверка результатов предыдущего этапа. Это позволяет сразу выявлять и устранять ошибки, не допуская их накопления.
- **Масштабируемость.** Пайплайн одинаково эффективен как для небольших, так и для крупных проектов с участием многих команд. Его легко масштабировать при росте задач.
- **Широкие возможности мониторинга.** Они появляются за счет отслеживания прогресса каждого этапа разработки по конкретным метрикам. Это облегчает анализ эффективности процесса.

The screenshot shows the GitLab CI interface. On the left, there's a sidebar with 'CORE' and 'CI/CD' sections, with 'Pipelines' selected. The main area displays a pipeline with four stages:

- Stage 1: Passed (0:01:35, 1 week ago)
- Stage 2: Failed (0:01:14, 1 week ago)
- Stage 3: Passed (0:00:53, 1 week ago)
- Stage 4: Passed (0:01:03, 1 week ago)

Each stage has a commit message and a merge request ID. A pink arrow points from the bottom of the pipeline to the text 'Пройденный пайплайн проекта'. Another pink arrow points from the right side of the failed stage to the text 'Статусы задач пайплайна'.

Пройденный пайплайн проекта

Статусы задач пайплайна

Пайплайн - сущность, принадлежащая комиту. Он запускается для конкретного коммита на его содержании.

Нажмем на пайплайн и увидим:

The screenshot shows the GitLab CI interface for a project named 'CORE'. A successful pipeline run (#135) is displayed, triggered by 'Space Bot' 1 week ago. The pipeline is titled 'Merge MFO-MR-1306: dev'. It shows 7 jobs for the 'master' branch, queued for 52 minutes and 7 seconds. The pipeline stages are 'lccl1d1ab' and 'prod:php-fpm'. A callout box highlights the completed pipeline with the text 'Пройденный пайплайн проекта'. Below the pipeline details, there is a diagram showing two parallel job flows: one for 'php-fpm' and one for 'php-worker', each consisting of 'codecheck', 'build', and 'prod' stages.

Задачи запускает :

### Gitlab Runner

Специальный отдельный компьютер, который прикреплен к репозиторию проекту, или группе проектов



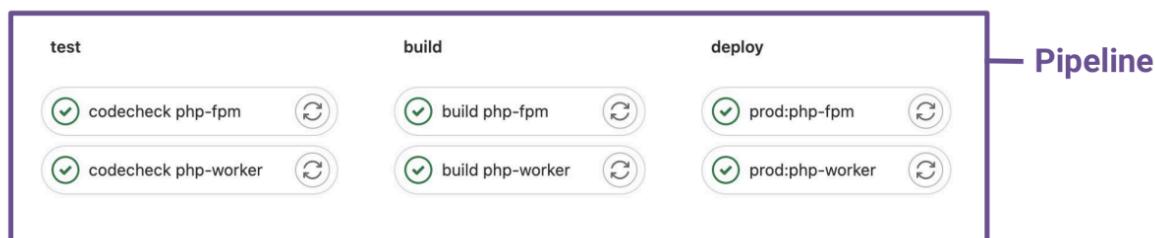
- Имеет доступ к репозиторию проекта
- Запускает заранее описанные сценарии действий с кодом
- Может работать со всеми возможностями Gitlab
- Имеет возможность работать с разными системами и разными интерпретаторами команд

выполняет задачи, описанные в GitLab CI  
может быть виртуалка, докер контейнер, отдельная машина и тд

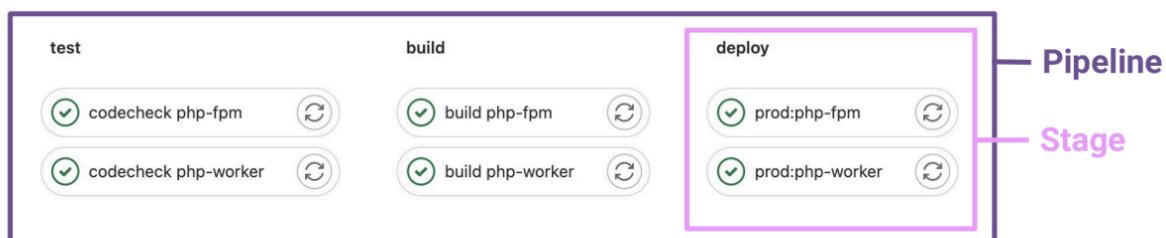
у ранеров есть экзекьютор - сущность, которая запускает описанные процессы

Ранер разу скачивает код коммита на котором он запустился. Затем он выполняет описаны нами задачи (которые мы описали в виде команд)

## Как устроен пайплайн

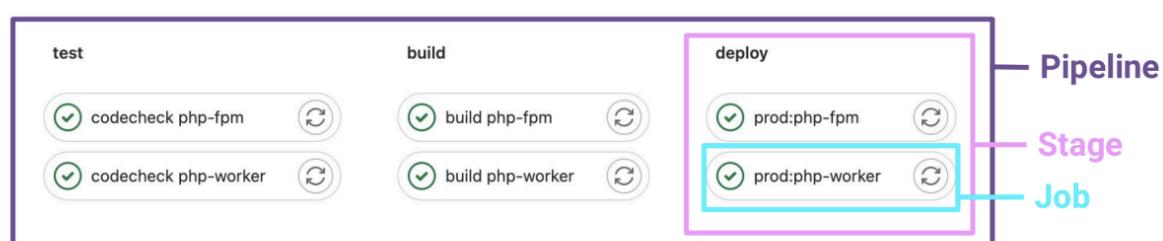


## Как устроен пайплайн



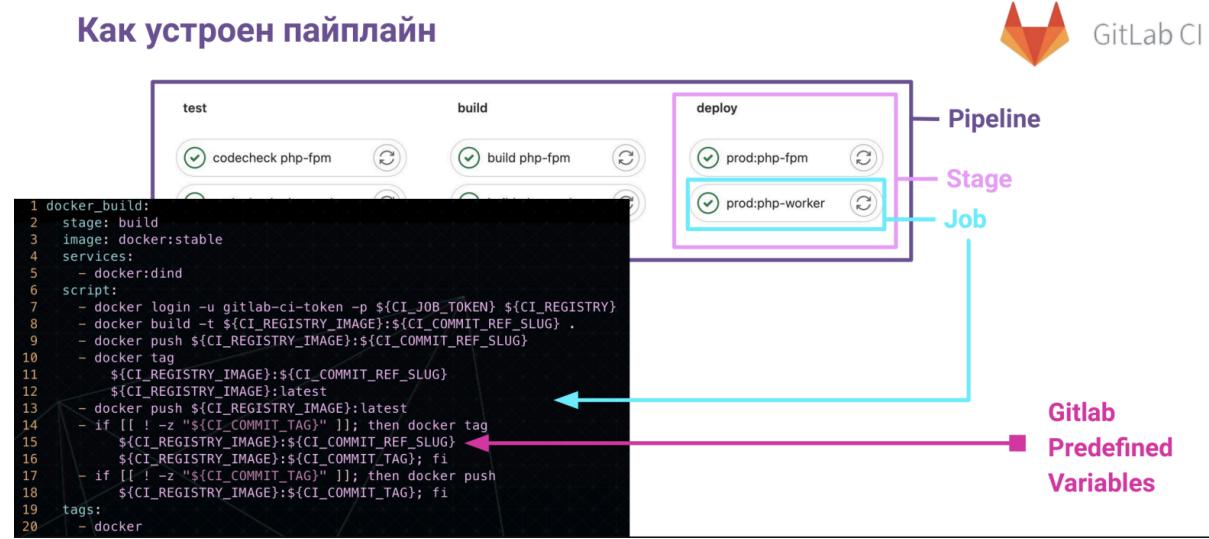
stage - колонки, имеют названия

## Как устроен пайплайн



Пайплайны описываются джобами.

Содержимое джобы тоже можно посмотреть  
(gitlabci.yml):



здесь нужно запустить в контейнере с образом docker:stable. Получается, что это джобу должен запускать докер раннер или кубернетис ранннер

сервисы как в докеркомбоуз - какие еще контейнеры нужно поднять в связки с нашим

script - тело джобы (логинимся по токену чтобы запушить)

у раннеров есть теги, чтобы выбирать какой ранер мы хотим

видно, что есть переменные окружения - GitLab Predefined Variables (это визитная карточка GilLab CI)

# Введение в Gitlab CI

## Предопределенные переменные GitLab CI

**Gitlab Predefined Variables** – Позволяют параметризовать и конфигурировать пайплайны и задачи в них, используя уникальные переменные, определяемые для каждой задачи

- **CI\_COMMIT\_REF\_NAME** – Тэг или ветка, для которой запустился пайплайн
- **CI\_COMMIT\_SHORT\_SHA** – Короткий хэш коммита
- **CI\_PROJECT\_NAME** – Имя проекта в Gitlab