**Введение автора**

**Данное пособие написано Козловым Михаилом в апреле месяце 2020 года находясь в самоизоляции от коронавируса**

**Видео можно найти на канале YouTube:** <https://www.youtube.com/channel/UCR6j95DKI80MFXIw2iJ3f1w>

**Сайт автора: <http://snakeproject.ru/>**

**Если Вам понравилось пособие и видео, Вы всегда сможете связаться со мной с помощью:**

**1. Формы обратной связи на сайте:** <http://snakeproject.ru/contacts.php>

**2. Странички в ВК:** <https://vk.com/sot_rbr>

**Отблагодарить автора можно абсолютно любой суммой на Ваше усмотрение.**

**Это поможет в развитии проекта.**

**Сам проект задумывался как абсолютно бесплатный, с 2013 года он работает для Вас в сети.**

**На IT в таком формате ничего не заработаешь, с натягом на услуги хостинга окупается.**

**Поддержать можно через эту форму: <https://money.yandex.ru/to/410012210709233>**

**Всем успехов в освоении новых технологий.**

**Щупаем Docker часть 1**

Без воды, особой теории и прочего в период самоизоляции от коронавируса в 2020 году сел освежить знания по **Docker**

Не претендую на слог Пушкина и прочие красивые объяснения терминов

Картинки взяты из открытых источников в интернете

Пособие написано автором проекта: **<http://snakeproject.ru> - Михаилом Козловым**

Определение об отличии контейнера **Docker** от классической виртуальной машины:

**Docker контейнер** - совокупность программного кода приложения и библиотек в изолированном пространстве

**Virtual machine** - совокупность программного кода приложения, библиотек + операционная система





Образы в **Docker** создаются из **Dockerfile** или берутся готовые из репозиториев, локальных или общедоступных типа **dockerhub**



По сути Docker нам дает некоторые преимущества.

Быстрое развертывание приложений, меньшая потребляемость ресурсов, независимость от ОС

**Основные понятия в Docker:**

**Docker Engine** - среда выполнения, управляет образами и контейнерами.

**Docker Daemon** - выполняет команды, отданные клиентом в docker. (сборка образа, запуск контейнера)

**Dockerﬁle** - файл с набором инструкций, применяется для сборки образов.

**Docker Image** - образ - файл, на основе которого создаются контейнеры. Состоит из множества слоев.

**Docker Container** - сущность, состоящая из кода приложения и все его зависимостей. Создается на основе образов.

**Docker Volumes** - пробрасываемые файловые носители для сохранения данных. Например сохранения файлов или базы данных, которые создаются или меняются в контейнерах.

**Рассмотрим установку Docker на Ubuntu 18.04**

**1. Установите по инструкции:**

<https://docs.docker.com/install/linux/docker-ce/ubuntu/>

**2. Убедитесь, что он работает:**

sudo systemctl status docker

**3. Добавьте текущего пользователя в группу Docker (для запуска команд docker без sudo):**

sudo usermod -aG docker ${USER}

**Базовые команды**

**Справка или помощь:**

docker --help

docker **команда\_докера** --help

**Частые в использовании команды:**

attach

build

commit cp create

diﬀ

events exec export

history

images import info inspect

kill

load login logout logs

pause port ps pull push

rename restart rm rmi run

save search start stats stop

tag top

unpause update

version

wait

**Для проверки работоспособности Docker дадим пару команд**

**Проверим версию docker:**

docker version

**Скачаем образ и запустим на его основе контейнер:**

docker run hello-world

**Щупаем Docker часть 2**

**После показательного примера с первым контейнером изучим основные команды**

**Выводим конфиг docker:**

docker info

Мы увидим в выводе информацию по сервису в т.ч. и счетчик контейнеров

**Запустим контейнер с последующим удалением после остановки:**

docker run --rm hello-world

**или:**

docker container run --rm hello-world

**Рассмотрим пример запуска веб-сервера nginx**

Сразу договоримся в дальнейшем, подразумевается, что команды идентичны:

**docker run** и **docker container run**

Используйте по вашему вкусу

**Запустим контейнер с nginx (используйте -p или --publish) с пробросом порта 80:**

docker container run -p 80:80 nginx

**Проверяем работу контейнера(откройте вторую ssh сессию):**

curl 127.0.0.1:80

Завершим работу контейнера - CTRL + C

**Что произошло в процессе запуска контейнера?**

Скачался образ Nginx из Docker Hub

Запустился контейнер на основе скачанного образа Nginx

Пробросился 80 порт локальной машины на 80 порт контейнера

Обратите внимание, контейнер запускался в интерактивном режиме

**Запустить контейнер в фоновом режиме:**

docker container run -p 80:80 -d nginx

**Посмотреть на него можно с помощью команд:**

docker ps

**или:**

docker container ls

**Запустим контейнер в фоне с определенным именем "web\_server":**

docker container run --name web\_server -d nginx

**Поставить контейнер "web\_server" на паузу и снять с нее:**

docker container pause web\_server

docker container unpause web\_server

**Остановить контейнер "web\_server" и стартовать:**

docker container stop web\_server

docker container start web\_server

**Статистику контейнеров смотреть командой:**

docker container stats

**Данные контейнера "web\_server" смотреть командой:**

docker container inspect web\_server

**Процессы контейнера "web\_server" смотреть командой:**

docker container top web\_server

**Удалить контейнер "web\_server" (даже запущенный "f" и его volume "v"):**

docker container rm -fv web\_server

**Создание контейнера без запуска имеет следующий синтаксис:**

docker create <options> <image name:tag>

**или:**

docker container create <options> <image name:tag>

**Варианты запусков контейнера**

**С рестартом контейнера в случае остановки (always - всегда):**

docker container rm -fv web\_server

docker container run -d -p 80:80 --name web\_server --restart always nginx

**Возможные опции:**

**no** - по умолчанию

**on-failure** - перезагрузка при ошибке

**always** - перезагрузка всегда

**unless-stopped** - пока контейнер не остановлен

**С healthcheck (необходимо иметь curl внутри контейнера):**

docker container rm -fv web\_server

docker container run -d -p 80:80 --name web\_server --health-cmd 'curl -sS 127.0.0.1:80 || exit 1' --health-interval=5s --health-timeout=10s --health-retries=3 nginx

Т.е. **Docker** отслеживает "работоспособность" контейнера командой с интервалом опроса, таймаутом, счетсчиком попыток

**С жестким лимитом памяти в 100 мегабайт(указывается в байтах ~ 100 \* 1024 \* 1024):**

docker container rm -fv web\_server

docker container run -d -p 80:80 --name web\_server -m 104857600 nginx

**С жестким лимитом памяти в 100 мегабайт и мягким (memory-reservation) 50:**

docker container rm -fv web\_server

docker container run -d -p 80:80 --name web\_server -m 104857600 --memory-reservation 52428800 nginx

**С жестким лимитом памяти в 100 мегабайт и swap (memory-swap) 50:**

docker container rm -fv web\_server

docker container run -d -p 80:80 --name web\_server -m 104857600 --memory-swap 52428800 nginx

**С подключением в интерактивном режиме в терминал контейнера с оболочкой bash:**

docker container rm -fv web\_server

docker container run -it --name web\_server nginx bash

Выйти из контейнера - CTRL + D

**Подключим stdout и stderr:**

docker container rm -fv web\_server

docker container run -d -p 80:80 --name web\_server --restart always nginx

docker container attach web\_server

Можете из соседней сессии дать команду "curl 127.0.0.1:80" и увидите stdout

Выйти из контейнера - CTRL + C контейнер при этом остановится, и перезапуститься (restart always)

**Выполнить команду (например ls) внутри контейнера или войти в оболочку bash:**

docker container exec -it web\_server ls

docker container exec -it web\_server bash

Выйти из контейнера - CTRL + D

**Вывод логов контейнера:**

docker container logs web\_server

**Вывод определенной информации о контейнере:**

docker container inspect web\_server --format "IP: {{ .NetworkSettings.IPAddress }} | Gateway: {{.NetworkSettings.Networks.bridge.Gateway}}"

**Создать образ на основе контейнера:**

docker container rm -fv web\_server

CID=$(docker container run -d -p 80:80 --name web\_server nginx)

docker container commit --author "Kozlov Mikhail snakeproject.ru" --message "Create New Image" "$CID" nginx-new-image:0.1

**Увидеть созданный образ можно командой:**

docker images

**Скопировать файлы или папки в контейнер:**

docker container cp /vmlinuz web\_server:/tmp/

**Вывод изменений в контейнере:**

docker container diff web\_server

**Удалить остановленные контейнеры:**

docker container prune

**Щупаем Docker часть 3**

**Типы сетей в Docker:**

**bridge**: сетевой драйвер по умолчанию.

Если вы не указываете драйвер, это тип сети, которую вы создаете.

Мостовые сети обычно используются, когда ваши приложения работают в автономных контейнерах, которые должны взаимодействовать.

**host**: для автономных контейнеров, удаляет сетевую изоляцию между контейнером и **Docker**-хостом.

Хост доступен только для служб **swarm** в **Docker** версии 17.06 и выше.

**overlay**: **overlay** сети соединяют несколько демонов **Docker** вместе и позволяют сервисам **Swarm** связываться друг с другом.

Вы также можете использовать **overlay** сети для облегчения связи между сервисом Swarm и автономным контейнером.

Или между двумя автономными контейнерами на разных демонах **Docker**.

Эта стратегия устраняет необходимость выполнять маршрутизацию на уровне ОС между этими контейнерами.

**macvlan**: сети **Macvlan** позволяют назначать **MAC**-адрес контейнеру, делая его физическим устройством в вашей сети.

Демон **Docker** направляет трафик в контейнеры по их **MAC**-адресам.

Использование драйвера **macvlan** иногда является лучшим выбором при работе с устаревшими приложениями.

Приложениям, которые ожидают прямого подключения к физической сети, а не маршрутизации через сетевой стек хоста **Docker**.

**none**: для этого контейнера отключит все сети.

Обычно используется в сочетании с пользовательским сетевым драйвером. Ни один не доступен для **swarm** услуг.

**Сетевые плагины**: вы можете устанавливать и использовать сторонние сетевые **плагины** с **Docker**.

Эти **плагины** доступны в **Docker Hub** или у сторонних поставщиков.

**Команды, для работы с сетью:**

docker network create

docker network connect

docker network disconnect

docker network inspect

docker network ls

docker network rm

docker network prune

**Примеры использования**

**Вывести список сетей:**

docker network ls

**Создание сети с именем "**web\_net**" типа bridge:**

docker network create -d bridge web\_net

**Подключаем контейнер к сети:**

docker container rm -fv web\_server

docker container run -d -p 80:80 --name web\_server --restart always nginx

docker network connect web\_net web\_server

**Вывести информацию о сети контейнера:**

docker container inspect web\_server --format "{{ .NetworkSettings.Networks }}"

**Вывести подробную информацию о сети контейнера:**

docker network inspect web\_net

**Отключить контейнер от сети:**

docker network disconnect web\_net web\_server

**Запуск контейнера сразу в указанной сети:**

docker container rm -f web\_server

docker network rm web\_net

docker network create -d bridge web\_net

docker container run -d -p 80:80 --name web\_server --net web\_net nginx

**Пример round-robin использования:**

docker container rm -f web\_server

docker container rm -f web\_server1

docker container rm -f web\_server2

docker network rm web\_net

docker network create web\_net

**Создадим два веб-сервера в одной сети с одним сетевым алиасом "web\_app":**

docker container run -d --net web\_net --net-alias web\_app --name web\_server1 nginx

docker container run -d --net web\_net --net-alias web\_app --name web\_server2 nginx

**Создадим тестовый файл для различия:**

echo "web\_server1 hello" > /tmp/index.html

docker container cp /tmp/index.html web\_server1:/usr/share/nginx/html/

echo "web\_server2 hello" > /tmp/index.html

docker container cp /tmp/index.html web\_server2:/usr/share/nginx/html/

**Посмотрим их ip обращаясь к сетевому алиасу "web\_app":**

docker container run --rm --net web\_net alpine nslookup web\_app

**Делаем запросы, они будут уходить к разным серверам:**

docker container run --rm --net web\_net centos curl -s web\_app:80

**Удалить неиспользуемые сети:**

docker network prune

**Щупаем Docker в самоизоляции от коронавируса ч4**

Поработаем образами и их созданием

**По умолчанию образ скачивается из репозитория docker hub:**

docker run -it ubuntu /bin/bash

Unable to find image 'ubuntu:latest' locally

latest: Pulling from library/ubuntu

**84ed7d2f608f**: Pull complete

**be2bf1c4a48d**: Pull complete

**a5bdc6303093**: Pull complete

**e9055237d68d**: Pull complete

Digest: sha256:868fd30a0e47b8d8ac485df174795b5e2fe8a6c8f056cc707b232d65b8a1ab68

Status: Downloaded newer image for ubuntu:latest

**В выводе мы видим:**

**84ed7d2f608f**, **be2bf1c4a48d**, **a5bdc6303093**, **e9055237d68d**

Это - слои, образ в докере состоит из слоев

**Т.е. образ имеет:**

**Базовый образ 0** (файловая система и директории минимально нужные для запуска)

**Образ 1** (на пример какое-то установленное ПО, веб-сервер, базы данных и т.п.)

**Образ 2** (на пример еще какое-то установленное ПО или обновления)

и т.д.

Т.е. мы будем иметь в нашем случае 4 слоя одного образа...

Все слои монтируются в режиме для чтения, поверх монтируется слой с возможностью перезаписи

При запуске контейнера перед **базовым** монтируется слой **bootfs**, после запуска контейнера он исчезает

Unable to find image 'ubuntu:**latest**' locally

Где 'ubuntu:**latest**':

Означает версию, если явно не указывать, то будет скачана последняя версия - **latest**

docker image pull ubuntu - **скачает последнюю версию образа ubuntu**

docker image pull ubuntu:14.04 - **скачает 14.04 версию образа ubuntu**

docker image pull -a ubuntu - с**качает все версии образов ubuntu**

**Список скачанных образов выводится с помощью команды:**

docker image ls

**Посмотреть список определенных образов ubuntu:**

docker image ls ubuntu

**Создать тэг образу можно данной командой:**

docker image tag <repository\_name/source\_image\_name:tag> <repository\_name/new\_image\_name:tag>

**Пример:**

docker image tag nginx new\_nginx:1.1

**Тэги говорят нам о версии образа**

ID образа, например 5b117edd0b76 повторяется дважды, указывает на один и тот-же образ

На диске образы будут храниться в /var/lib/docker/{driver-name}

Это может быть: **overlay2,** **aufs**, **overlay**, **btrfs**, **devicemapper** или **zfs**

**Если зарегистрироваться в docker hub, создать общедоступный репозиторий, можно запушить в него образ:**

docker image push <options> <repository\_name/image\_name:tag>

**Пример:**

docker image push kozlov\_repo/super\_nginx:1.1

**Скачать идентично:**

docker image pull kozlov\_repo/super\_nginx:1.1

**Посмотреть информацию по образу можно данной командой:**

docker image inspect <options> <repository\_name/image\_name:tag>

**Пример с форматированием:**

docker image inspect centos --format "{{.Config.Labels}}"

**История образа (слои) выводится так:**

docker image history <options> <repository\_name/image\_name:tag>

**Пример:**

docker image history centos

**Или:**

docker image history --no-trunc centos

**Удалить все не используемые образы:**

docker system prune -a -f

**Щупаем Docker в самоизоляции от коронавируса ч5**

**Dockerfile** - файл инструкций, которые выполняются с верху в низ

С помощью него создаются образы

В файле есть синтаксис, инструкция - значение

**Рассмотрим самые распространенные:**

# - комментарий

FROM - обязательная, указывается базовый образ

MAINTAINER - контакт разработчика данного образа

RUN - что будем делать, команды, каждый RUN по сути создает свой слой

CMD - команда в запущенном контейнере (возможно переопределить при старте контейнера)

ENTRYPOINT - команда в запущенном контейнере, аналогична CMD(невозможно переопределить при старте контейнера)

EXPOSE - трансляция указанных портов

ENV - переменные среды контейнера

COPY - копирование локальных файлов в контейнер

ADD - добавление файлов в контейнер, допускает указание url

VOLUME - указание пробрасываемого пути файловой системы для сохранения результатов работы в ней

USER - указание учетной записи, от которой будут работать следующие команды

WORKDIR - указание рабочей директории, в которой будут работать следующие команды

LABEL - метаинформация образа

**Создание образа из Dockerfile (. - указание текущего каталога с Dockerfile):**

docker image build . -t <image name:tag>

**Перейдем в тестовую папку для эксперимента:**

 mkdir test && cd test

**Пример Dockerfile:**

FROM nginx:latest

LABEL version="latest\_local\_0.1" maintainer="kozlovma"

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

# команда создаст символическую ссылку, будет выводиться access.log на консоль

RUN ln -sf /dev/stdout /var/log/nginx/access.log

ENV WORKDIR /usr/share/nginx/html

WORKDIR ${WORKDIR}

COPY ./index.html ./index.html

EXPOSE 80 443

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

**Тестовый файл index.html:**

echo '<html><head><title>Hello</title></head><body><h2>Hello!</h2></body></html>' > index.html

**Скачиваем образ nginx для Dockerfile:**

docker image pull nginx

**Создаем образ на основе Dockerfile:**

docker build . -t nginx:latest\_local\_0.1

**Стартуем и проверяем:**

docker run -d -p 80:80 nginx:latest\_local\_0.1

curl 127.0.0.1:80

**Щупаем Docker в самоизоляции от коронавируса ч6**

Ранее мы работали с публичным репозиторием образов - **Docker Hub**

Создадим локальный репозиторий для хранения образов

**Скачаем образ репозитория "registry" с docker hub:**

docker container run -d -p 5000:5000 --restart always --name my\_local\_registry registry

**Зададим имя образу в локальном репозитории:**

docker image tag nginx:latest\_local\_0.1 localhost:5000/nginx:latest\_local\_0.1

**Зальем образ на локальный репозиторий:**

docker push localhost:5000/nginx:latest\_local\_0.1

**Скачать образ из локального репозитория данной командой:**

docker pull localhost:5000/nginx:latest\_local\_0.1

Немного забегая вперед создадим запускальщик **"docker-compose"** нашего репозитория, который заставит его хранить свои данные в определенной директории

**Инструкция по установке docker-compose в ubuntu:**

sudo apt install docker-compose

sudo mkdir /var/lib/registry

**docker-compose.yml:**

my\_local\_registry:

 restart: always

 image: registry

 container\_name: my\_local\_registry

 ports:

 - 5000:5000

 volumes:

 - /var/lib/registry:/var/lib/registry

**Удалим контейнер предыдущего репозитория:**

docker container stop my\_local\_registry

docker container rm -fv my\_local\_registry

**Запустим новый репозиторий с указанием рабочей директории и проверим:**

docker-compose up -d

docker push localhost:5000/nginx:latest\_local\_0.1

**Щупаем Docker в самоизоляции от коронавируса ч7**

Рассмотрим работу с **volumes**

**Основные понятия:**

**Bind Mount** - файл \ директория на диске хостовой системы

**TMPFS** - механизм временного хранения данных, генерируемых \ используемых контейнером в оперативной памяти хостовой системы

**Docker Volume** - механизм сохранения данных, генерируемых \ используемых контейнером

**Схема выглядит так:**



**Команды для работы с volumes:**

docker volume create

docker volume ls

docker volume inspect

docker volume rm

docker volume prune

**Синтаксис создания и применения именнованного volume:**

docker volume create <options> <volume\_name>

docker container run -d -v <volume\_name>:/path <image\_name>

**Пример создания и подключения volumes**

Допустим, мы хотим сохранить данные, генерируемые контейнером

Рассмотрим на примере образа БД **PostgreSQL**

 **Удалим ненужные volumes:**

docker volume prune -f

**Скачиваем образ "postgres":**

docker image pull postgres

**Инспектирование образа, какие пути он использует при работе контейнера:**

docker image inspect postgres --format "{{.Config.Volumes}}"

map[/var/lib/postgresql/data:{}]

**Создаем volume:**

docker volume create volume\_data

**Обратите внимание, вы можете задать размер volume(например в 100 мегабайт):**

docker volume create --opt o=size=100m volume\_data

**Выведим список volumes:**

docker volume ls

DRIVER VOLUME NAME

local volume\_data

**Если мы хотим сохранить данные работы контейнера, то должны подключить volumes к этим путям:**

docker container run -d -v volume\_data:/var/lib/postgresql/data -p 5432:5432 -e POSTGRES\_PASSWORD=pass --name postgresql\_server postgres

**Где сохранятся данные:**

docker volume inspect volume\_data

...

 "Mountpoint": "/var/lib/docker/volumes/volume\_data/\_data",

 "Name": "volume\_data",

...

**И после удаления контейнера, данные останутся:**

docker exec -it postgresql\_server psql -U postgres -c "CREATE DATABASE test;"

docker exec -it postgresql\_server psql -U postgres -c "\l test"

 List of databases

 Name | Owner | Encoding | Collate | Ctype | Access privileges

 test | postgres | UTF8 | en\_US.utf8 | en\_US.utf8 |

docker container rm -f -v postgresql\_server

docker container run -d -v volume\_data:/var/lib/postgresql/data -p 5432:5432 -e POSTGRES\_PASSWORD=pass --name postgresql\_server postgres

docker exec -it postgresql\_server psql -U postgres -c "\l test"

 List of databases

 Name | Owner | Encoding | Collate | Ctype | Access privileges

 test | postgres | UTF8 | en\_US.utf8 | en\_US.utf8 |

**Примеры команд**

**Или указание типа "tmpfs" (оперативная память):**

docker container run -d --mount type=tmpfs,destination=/usr/share/nginx/html nginx

**Или указание volume source и destination для созданного volume:**

docker container run -d --mount source=volume\_www,destination=/usr/share/nginx/html nginx

**Или в режиме readonly:**

docker container run -d --mount source=volume\_www,destination=/usr/share/nginx/html,readonly nginx

**Синтаксис применения "bind" без mount, source, destination:**

docker container run -d -v <host\_path\_dir>:/usr/share/nginx/html nginx

**Пример указания типа "bind" с mount, source, destination:**

mkdir /www\_dir

docker container run -d --mount type=bind,source=/www\_dir,destination=/usr/share/nginx/html nginx

**Инспектирование именованного volume "volume\_www":**

docker volume inspect volume\_www

**Обращаем внимание, где хранятся данные:**

"Mountpoint": "/var/lib/docker/volumes/volume\_www/\_data"

**Щупаем Docker в самоизоляции от коронавируса ч8**

Ранее был приведен простейший пример запуска сервисов с помощью **docker-compose**

Теперь рассмотрим пример, приближенный к жизни

**Пример связки Nginx + PHP-FPM + MySQL8**

Мы создадим связку, в которой будет разделение по сетям

**frontend** и **backend** будут доступны только **php-fpm** сервису

Сервис **php-fpm** будет использовать образ, описанный в локальном **Dockerfile**

Так-же будет применяться проверка **healthcheck** с помощью **curl**

**Создадим каталоги для проекта:**

sudo mkdir -p /data/web/{nginx,logs,public,php-fpm,mysql}

**Я работаю под пользователем peasant в системе:**

sudo chown -R peasant:peasant /data

**Создадим файлы для логов nginx:**

touch /data/web/logs/{nginx-error.log,nginx-access.log}

**Создадим конфиг для nginx:**

cat > /data/web/nginx/default << EOF

server {

 listen 80 default;

 root /application/public;

 index index.php;

 location ~ \.php$ {

 fastcgi\_pass php-fpm:9000;

 fastcgi\_index index.php;

 include fastcgi\_params;

 fastcgi\_param SCRIPT\_FILENAME \$document\_root\$fastcgi\_script\_name;

 }

}

EOF

**Приводим тестовый файл /data/web/public/index.php к такому виду:**

<?php

 $db = new PDO("mysql:host=mysql;dbname=dbname", "root", "32167");

 $db -> exec("set names utf8");

?>

<html>

<head><title>Test</title></head>

<body>

 <?php

 $body = '';

 foreach ($db->query('SELECT id, name FROM dbname.test') as $row){

 $body .= "<p>";

 $body .= $row['id'];

 $body .= " ";

 $body .= $row['name'];

 $body .= "</p>";

 }

 echo $body;

 ?>

</body></html>

**Создадим образ php-fpm для проекта:**

cat > /data/web/php-fpm/Dockerfile << EOF

FROM php:7.3-fpm

RUN apt-get update && apt-get install -y curl mariadb-client && \

 /usr/local/bin/docker-php-ext-install pdo pdo\_mysql

WORKDIR "/application"

EOF

**Создадим docker-compose файл, свяжем наши сервисы в единое целое:**

cat > /data/web/docker-compose.yml << EOF

version: "3.1"

services:

 nginx:

 image:

 tutum/nginx

 restart: always

 container\_name: fb\_nginx

 ports:

 - "80:80"

 links:

 - php-fpm

 volumes:

 - ./nginx/default:/etc/nginx/sites-available/default

 - ./nginx/default:/etc/nginx/sites-enabled/default

 - ./logs/nginx-access.log:/var/log/nginx/access.log

 - ./logs/nginx-error.log:/var/log/nginx/error.log

 networks:

 - frontend

 php-fpm:

 build:

 context: .

 dockerfile: ./php-fpm/Dockerfile

 restart: always

 container\_name: fb\_php\_fpm

 ports:

 - "9000:9000"

 volumes:

 - ./public:/application/public

 depends\_on:

 - mysql

 healthcheck:

 test: curl -s nginx || exit 1

 interval: 10s

 timeout: 10s

 retries: 5

 networks:

 - frontend

 - backend

 mysql:

 image: mysql:8

 restart: always

 container\_name: fb\_mysql\_8

 volumes:

 - ./mysql:/var/lib/mysql

 environment:

 - MYSQL\_ROOT\_PASSWORD=32167

 - MYSQL\_DATABASE=dbname

 - MYSQL\_USER=dbuser

 - MYSQL\_PASSWORD=32167

 ports:

 - "3306:3306"

 networks:

 - backend

networks:

 frontend:

 driver: bridge

 backend:

 driver: bridge

EOF

**Собираем и запускаем проект:**

cd /data/web/

docker-compose up -d

**Подключимся и создадим тестовые данные в MySQL:**

docker exec -it fb\_mysql\_8 bash

**Пароль вводим указанный в файле docker-compose.yml:**

mysql -u root -p

**Выбираем базу данных из файла docker-compose.yml:**

show databases;

+--------------------+

| Database |

+--------------------+

| dbname |

| information\_schema |

| mysql |

| performance\_schema |

| sys |

+--------------------+

USE dbname;

**Создаем тестовые данные:**

CREATE TABLE test

(

 id INT NOT NULL auto\_increment,

 name VARCHAR(30) NOT NULL DEFAULT '',

 PRIMARY KEY (id)

);

INSERT INTO test (name) VALUES ("testname1");

INSERT INTO test (name) VALUES ("testname2");

INSERT INTO test (name) VALUES ("testname3");

**А эта строка, чтоб не вылезла ошибка:**

Uncaught PDOException: PDO::\_\_construct():

The server requested authentication method unknown to the client [caching\_sha2\_password]:

alter user 'root'@'%' identified with mysql\_native\_password by '32167';

**Выходим:**

\q

exit

**Теперь проверим:**

curl -s http://127.0.0.1:80/index.php

<html>

<head><title>Test</title></head>

<body>

<p>1 testname1</p><p>2 testname2</p><p>3 testname3</p>

</body></html>

**Щупаем Docker в самоизоляции от коронавируса ч9**

**Portainer** - **web** доступ управлением **Docker**

**Остановим контейнеры из предыдущего шага, тут они нам не понадобятся:**

docker container stop fb\_nginx fb\_php\_fpm fb\_mysql\_8

**Portainer** представляет из себя средство администрирования **Docker** через **web**

**Установим:**

mkdir /data/portainer

docker run -d -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.sock -v /data/portainer:/data portainer/portainer

Тут **/data/portainer** монтируется чтоб изменения сохранились при перезагрузке сервера или контейнера

Узнать сетевой адрес хостовой машины можно с помощью **ifconfig**

У меня это **192.168.0.128**

**Проверяем:**

[http://192.168.0.128:9000/](http://192.168.0.128:9000/%22%20%5Co%20%22http%3A//192.168.0.128%3A9000/)

Указываем данные доступа администратора к панели управления, локальный хост для управления

Заходим сразу в **local**

И увидим панель управления контейнерами, образами, логами и т.п.

Собственно - все, теперь у вас есть web мордочка для администрирования