# Основы DevOps

Контейнеризация при помощи Docker

# Введение в контейнеризацию

# Плюсы и минусы виртуализации

- В рамках наших уроков мы сталкивались с таким понятием как **виртуализация**, т. е. со способностью компьютера одновременно поддерживать работу разных операционных систем на разных виртуальных машинах (программах, которые эмулируют работу компьютера). В результате мы получаем полностью изолированные окружения, где можно запускать любые программы без опасения, что это нанесет вред другим окружениям или самому компьютеру.

- В числе прочего есть возможность делать снимки состояния виртуальных машин, что позволяет восстанавливаться после возможных сбоев, упрощает процесс резервного копирования и облегчает переносимость данных между разными компьютерами.

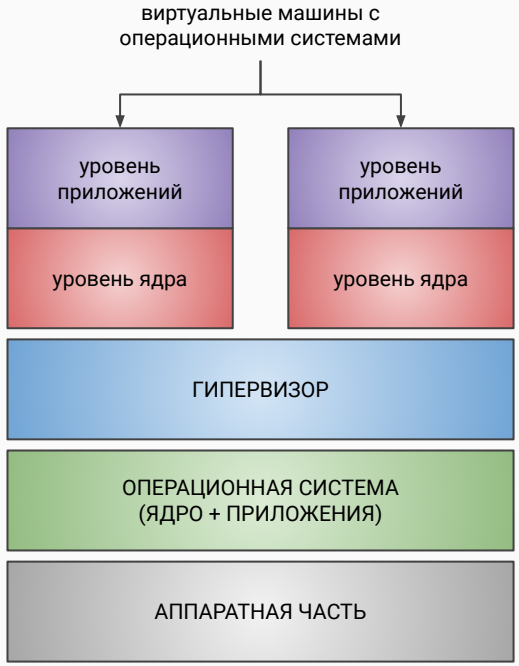
- Однако все вышесказанное достается нам очень дорогой ценой - для полной эмуляции виртуальной машины требуется большое количество вычислительных ресурсов, что замедляет работу самой виртуальной машины. Кроме того, высокий уровень изоляция одной системы от другой создает дополнительные проблемы при попытках настроить совместную работу этих систем. Наконец, виртуализация не позволяет переиспользовать в одних системах те ресурсы, которые уже используются в рамках других изолированных окружений.

# Альтернатива виртуализации - контейнеризация

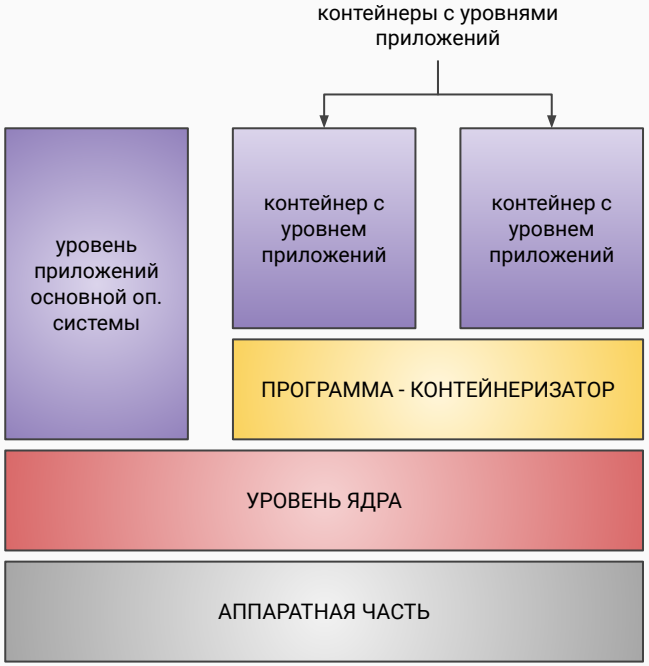
- Для решения проблем, связанных с виртуализацией, было придумано новое решение - **контейнеризация**. По своей сути, данная технология является виртуализацией на уровне приложений операционной системы. Дело в том, внутри операционной системы работа происходит на двух уровнях - на уровне ядра (взаимодействие непосредственно с аппаратной частью - “железом”) и на уровне приложений (взаимодействие с пользовательскими программами - текстовыми редакторами, браузерами и т.д.). В контексте контейнеризации эмулирование работы ядра не происходит - операционная система опирается на распараллеливание уже изначально запущенных процессов ядра, в свою очередь для уровня приложения создается специальная среда (своего рода контейнер), которая изолирована от других систем.

- Контейнеризация возможна даже в том случае, если основная и эмулированная система отличаются (например, при запуске Linux контейнеров на Windows). Сначала внутри Windows с помощью виртуализации (посредством Windows Subsystem for Linux или гипервизора Hyper-V) создается специальная среда для запуска ядра Linux, а уже потом поверх этого ядра запускаются изолированные контейнеры с разными дистрибутивами Linux. Однако в этом случае возрастают накладные расходы и быстродействие изолированных сред ухудшается.

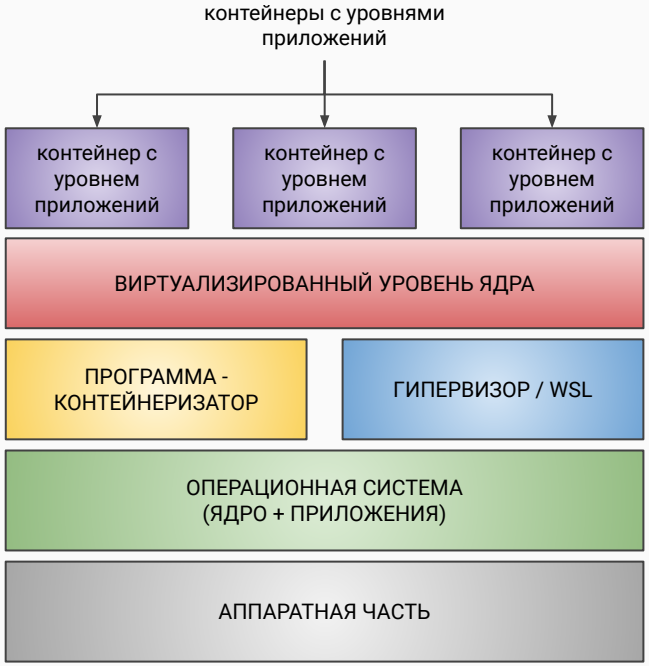
# Визуальное сравнение виртуализации и контейнеризации




ВИРТУАЛИЗАЦИЯ




КОНТЕЙНЕРИЗАЦИЯ ДЛЯ СИСТЕМ С ОДИНАКОВЫМ ЯДРОМ




КОНТЕЙНЕРИЗАЦИЯ ДЛЯ СИСТЕМ С РАЗНЫМ ЯДРОМ

# Преимущества контейнеризации

- Контейнеры (в отличие от виртуальных машин) имеют более тесную связь с ядром операционной системы, поэтому можно рассчитывать на довольно высокий уровень быстродействия и более эффективное использование доступных ресурсов (это особенно актуально для систем с одинаковым ядром).

- По сравнению с виртуальными машинами контейнеры запускаются гораздо быстрее, т.к. им не надо ждать загрузки ядра (оно уже загружено на момент старта контейнера).

- Контейнеры обычно занимают гораздо меньше места на диске по сравнению с виртуальными машинами, так как они используют общие компоненты - это упрощает их развертывание и масштабирование.

- Использование общих компонентов также облегчает потенциальное взаимодействие между двумя или более контейнерами (легче наладить общую сеть, подмонтировать общую файловую систему и т.д.).

- В современном мире контейнеры обладают удобными средствами управления (т.н. контейнеризаторами) - из них самым популярным является Docker.

# Что нам может дать контейнеризация?

- Контейнеры играют огромную роль в т.н. **архитектуре микросервисов** - каждая часть системы находится в отдельном контейнере, что позволяет ей быть написанной на своем языке программирования и использовать специальные технологии (которые могли бы конфликтовать с другими частями системы, если бы они работали в одном контейнере).

- Еще одним важным преимуществом контейнеров является **переносимость окружения**. После разработки нашей системы нам теперь не надо думать о том, как ее внедрять у конечного пользователя. Мы производим разработку внутри контейнеров, где окружение настроено под нужды нашей системы, а для развертывания системы просто запускаем копии контейнеров (у которых внутри находится система и правильно настроенное окружение) в нужном для клиента дата-центре.

- Также контейнеры могут оказать содействие при **масштабировании системы**. Если для решения некой задачи не хватает вычислительной мощности, то не будет проблемой развернуть дополнительные контейнеры для балансировки нагрузки (горизонтальное масштабирование) или перенаправить ресурсы для работы критически важного контейнера (вертикальное масштабирование).

# Установка Docker

# Введение

- **Docker** - платформа для создания и управления контейнерами, появившаяся в 2013 году и быстро ставшая общепринятым стандартом при разработке и внедрении программного обеспечения. В основе Docker находится т.н. **Docker Engine** - технология, которая состоит из трех частей. Во-первых, это **Docker Daemon** или **dockerd** - фоновый процесс, который управляет контейнерами и их взаимодействием. Во-вторых, это **Docker API** - специальный интерфейс (набор методов взаимодействия), позволяющий подключаться к dockerd. И наконец, в-третьих, это **docker** - специальный клиент командной строки, который используется для отправки инструкций dockerd.

- Контейнеры создаются на основе образов, которые обычно загружаются из специального репозитория **Docker Hub** (https://hub.docker.com/), но можно использовать и другие репозитории. Образ содержит базовый функционал какой-либо операционной системы (Debian, Ubuntu, Fedora и т.д.), на основе которой можно запускать свои программы и настраивать свое окружение.

- Docker позволяет подключаться к контейнерам, автоматически присоединяет их к специальной общей сети, разрешает пробрасывать внутрь контейнера порты, а также предоставляет возможность монтировать внешние файловые системы.

# Установка Docker для Ubuntu

- Установка Docker для Ubuntu подробно описана на сайте создателей Docker'а (https://docs.docker.com/engine/install/ubuntu/) - весь процесс происходит в командной строке и не требует особых усилий:

```
# создание цифровой подписи для репозитория Docker:
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# добавление Docker репозитория:
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

# установка зависимостей для Docker:
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin

# добавление пользователя в группу docker:
sudo usermod -aG docker $USER
```

# Установка Docker для Windows (зависимости)

Чтобы установить Docker для Windows, сначала необходимо определимся с тем типом виртуализации, который будет использоваться для запуска ядра Linux - как мы помним, их может быть всего два - через Windows Subsystem for Linux (WSL) или через гипервизор Hyper-V:

- Если мы выбрали WSL (предпочтительный вариант), то в разделе “**Windows Features**” необходимо включить две опции - “**Windows Subsystem for Linux**” и “**Virtual Machine Platform**”. Затем следует перезапустить компьютер.

- Если нам больше нравится гипервизор Hyper-V, то в разделе “**Windows Features**” надо включить следующие опции - “**Hyper-V**” и “**Containers**”. После этого также стоит перезапустить компьютер.

Какой бы из вариантов мы не выбрали, также нам надо убедиться, что у нашего компьютера достаточно оперативной памяти - как минимум 4 Gb, процессор имеет 64-х битную разрядность, а в UEFI включена аппаратная виртуализация - иначе мы не сможем запустить ядро Linux. Работать с Windows контейнерами мы не будем, т.к. они совместимы только с Windows 10 или Windows 11 Professional или Enterprise версиями.

# Установка Docker для Windows (инсталляция)

- Для установки Docker на Windows сначала надо загрузить программу Docker Desktop for Windows: https://docs.docker.com/desktop/install/windows-install/.

- После запуска инсталлятора нам будет предложено выбрать WSL вместо Hyper-V - лучше всего сделать это, но надо помнить о нашем выборе на предыдущем шаге - для WSL должны быть включены соответствующие зависимости.

- Дождемся окончания установки и запустим Docker Desktop. После этого перейдет в раздел настроек “**Settings**” (иконка шестеренки вверху справа), выберем секцию “**General**”, а затем выключаем опции “**Start Docker Desktop when you log in**” и “**Open Docker Dashboard at startup**”. После этого нажимаем синюю кнопку “**Apply & restart**” внизу слева.

- После всех вышеописанных манипуляций мы можем приступать к работе с Docker контейнерами, но надо помнить, что Docker Desktop бесплатен только для физических лиц и небольших компаний с годовым доходом менее 10 миллионов долларов и количеством сотрудников, не превышающим 250 человек.

# Начало работы с Docker

# Загрузка образов

- Как мы уже знаем, Docker работает с контейнерами, но контейнеры должны быть созданы на базе образа операционной системы. По умолчанию Docker предлагает загружать их со своего репозитория Docker Hub, что мы и сделаем. Загрузка осуществляется командой **docker pull**, после которой идет название образа (название можно поискать на сайте репозитория по ключевым словам):

```
docker pull hello-world
```

- Важнейший момент - у образов есть версии, которые могут сильно отличаться одна от другой. Если мы просто указываем название образа без дополнительных параметров, то будет загружен самый свежий образ - это также можно указать явно, подставив специальный тэг **latest** (т.е. последняя версия):

```
docker pull hello-world:latest
```

- Если нам нужна конкретная версия образа, то в виде тэга мы можем подставить уже ее. Например, явно укажем версию образа Ubuntu - **22.04**:

```
docker pull ubuntu:22.04
```

- Явное указание версии образа и отказ от использования тэга **latest** считается наиболее правильным подходом, поэтому отныне будем делать только так!

# Просмотр и удаление образов

- На данный момент у нас загружено два образа - *hello-world* и *ubuntu*. Чтобы просмотреть все загруженные образы, надо применить команду **docker images**:

```
docker images
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|---|---|---|---|---|
| ubuntu | 22.04 | e4c58958181a | 4 weeks ago | 77.8MB |
| hello-world | latest | 9c7a54a9a43c | 6 months ago | 13.3kB |

- Для того, чтобы удалить образ, следует использовать команду **docker rmi**, после которой надо указать либо идентификатор образа (**IMAGE ID**), либо его название (**REPOSITORY**), либо название вместе с тэгом (разделив их двоеточием):

```
docker rmi hello-world
```

- Если мы хотим удалить абсолютно все образы, то применяется команда **docker image prune -a**. Она потребует подтверждение наших действий, поэтому даже после ввода команды можно передумать и отменить решение об удалении:

```
docker image prune -a
```

# Запуск нашего первого контейнера

- Настало время запустить наш первый простейший контейнер. Для этого будем использовать образ ubuntu - мы удалили его на прошлом шаге, но при старте контейнера Docker сможет подгрузить его сам. Контейнер запускается командой **docker run**, после которой надо указать необходимый нам образ:

```
docker run ubuntu:22.04
```

- Контейнер должен был запуститься и после этого сразу завершить свою работу. Почему же он сразу остановился? Дело в том, что контейнерам при старте нужен хотя бы один процесс,  который может выполнять задачи. Если процесса нет - контейнер останавливается. В образе *ubuntu* нет процесса по умолчанию, поэтому без дополнительных команд контейнер всегда прекращает работу.

- Получается, мы должны запустить внутри контейнера процесс. В качестве примера попросим его при старте открыть оболочку командной строки (**/bin/bash**) и ждать наших инструкций. Кроме того, мы должны дополнить нашу команду параметром **-i** (позволяет контейнеру читать ввод с нашей клавиатуры) и параметром **-t** (позволяет подключаться к контейнеру в качестве терминала):

```
docker run -it ubuntu:22.04 /bin/bash
```

# Запуск фонового контейнера

- После манипуляций, произведенных в предыдущем шаге, контейнер запустится и продолжит работать, но работать он будет только то время, когда мы будем находится в командной строке. Если же мы ее закроем (командой **exit** или как-то иначе), то контейнер немедленно остановится.

- Чтобы решить вышеупомянутую проблему, необходимо запустить контейнер в фоновом режиме - для этого используется параметр **-d**. После этого мы можем свободно покидать командную строку - контейнер продолжит свою работу:

```
docker run -dit ubuntu:22.04 /bin/bash
```

# Просмотр списка контейнеров

- Для просмотра статусов используется команда **docker ps** - если мы ее применим, то увидим, что нам доступен только один контейнер:

```
docker ps
```

```
CONTAINER ID   IMAGE          COMMAND     CREATED          STATUS          PORTS     NAMES
e409aa3ae57f   ubuntu:22.04   "/bin/bash"   2 minutes ago    Up 2 minutes              kind_zhukovsky
```

- Однако до этого мы запускали еще два контейнера - куда же они пропали? Дело в том, что команда docker ps показывает только активные контейнеры, а остановленные - игнорирует. Для отображения всех контейнеров используется параметр **-a**:

```
docker ps -a
```

```
CONTAINER ID   IMAGE          COMMAND     CREATED          STATUS                      PORTS     NAMES
e409aa3ae57f   ubuntu:22.04   "/bin/bash"   2 minutes ago    Up 2 minutes                          kind_zhukovsky
ce431d272765   ubuntu:22.04   "/bin/bash"   23 minutes ago   Exited (0) 23 minutes ago             priceless_bohr
902d528c0aeb   ubuntu:22.04   "/bin/bash"   23 minutes ago   Exited (0) 23 minutes ago             intelligent_panini
```

# Остановка, запуск и перезапуск существующих контейнеров

- Если мы хотим остановить работающий контейнер, надо применить команду **docker stop**, после которой следует указать либо ID контейнера, либо имя (имя генерируется автоматически, но дальше мы увидим, как это делать вручную):

```
docker stop e409aa3ae57f
```

- Для повторного запуска контейнера (именно для повторного запуска, а не для создания и запуска) используется команда **docker start**:

```
docker start e409aa3ae57f
```

- Если по какой-то причине нам нужно перезапустить контейнер, то следует задействовать команду **docker restart**:

```
docker restart e409aa3ae57f
```

# Удаление контейнеров

- Если мы хотим удалить контейнер, то следует применить команду **docker rm** после которой можно указать те имена или ID контейнеров (разделенные пробелом), которые нам больше не нужны:

```
docker rm ce431d272765 902d528c0aeb
```

- При попытке удалить работающий контейнер Docker выдаст ошибку - т.е. по умолчанию удалять действующие контейнеры запрещено. Однако мы можем применить параметр **-f** - тогда все пройдет успешно:

```
docker rm -f e409aa3ae57f
```

- Также существует возможность уничтожить все контейнеры одновременно - это осуществляется командой **docker container prune** - в числе прочего надо знать, что эта команда потребует подтверждения наших действий:

```
docker container prune
```

# Назначение имени контейнера

- При запуске контейнера ему будет назначен уникальный идентификатор и некое случайное имя. Чтобы назначить это имя вручную и пользоваться им для старта, перезапуска и других операций с контейнером, следует применить параметр **--name**, после которого указывается имя контейнера:

```
docker run -dit --name=my_container ubuntu:22.04 /bin/bash
```

# Подключение к работающему контейнеру

- Если мы хотим подключиться к работающему контейнеру и произвести в нем некоторые манипуляции, нам поможет команда **docker exec**, после которой надо указать ID или имя контейнера, а потом указать нужную команду:

```
docker exec my_container ls -la
```

- Для более активной работы с контейнером нам надо подключиться к оболочке командной строки (*bash* или *sh*) внутри него. Для этого опять же используется команда **docker exec**, но с параметрами **-i** (позволяет контейнеру читать ввод с нашей клавиатуры) и **-t** (позволяет подключаться к контейнеру в качестве терминала), а в конце подставляется обращение к оболочке.

```
docker exec -it my_container /bin/bash
```

# Запуск контейнера с переменными окружения

- Некоторые контейнеры при запуске требуют указания переменных окружения - они могут использоваться для установки паролей, генерации конфигурационных параметров и т.д. Например, контейнер, содержащий СУБД MySQL, требует пароль для пользователя root с помощью указания переменной **MYSQL_ROOT_PASSWORD**. Для этого следует использовать параметр **-e**, затем название переменной, а уже в конце (через равно) - значение переменной. Если понадобится несколько переменных, то конструкцию с параметром **-e** надо повторить несколько раз. Подставим MySQL контейнеру пароль *secret*:

```
docker run --name=db -d -e MYSQL_ROOT_PASSWORD=secret mysql:8.0.35
```

# Хранение данных в контейнерах

# Введение

- Когда мы производим работу в контейнере (создаем файлы и директории, заполняем файлы текстом и т.д.), результат наших действий сохраняется не в оперативной памяти, а на диске нашего компьютера. Таким образом, если мы остановим наш контейнер, а потом запустим его вновь, все наши изменения и дополнения будут доступны для дальнейшей работы.

- Где же хранятся эти данные? Если мы создаем контейнер и не указывает опций, которые отвечают за содержимое, то данные будут находится в специальном слое контейнера. При использовании Windows и WSL все слои контейнеров расположены по пути **\\wsl.localhost\docker-desktop-data\data\docker\overlay2**.

- Данный подход к хранению данных не является надежным решением, т.к. слои контейнера автоматически удаляются, когда удаляются сами контейнеры. Поэтому для долгосрочного хранения результатов нашей работы надо использовать другие решения.

# Подключение внешних директорий к контейнерам

- Один из самых простых способов долговременного и надежного хранения данных - подключение какой-либо директории нашего основного компьютера ко внутренней директории одного или нескольких контейнеров.

- Для осуществления предложенного решение необходимо использовать параметр **-v**, после него следует указать полный путь к директории на основном компьютере, затем двоеточие, а в конце - путь к директории внутри контейнера (даже если ее нет, она будет автоматически создана). Предположим, что у нас есть директория **data**, которую мы хотим подключить к директории контейнера **/var/data**:

```
docker run -dit -v data:/var/data --name=dir_container
ubuntu:22.04 /bin/bash
```

- Теперь все изменения в директории контейнера **/var/data** будут отображаться в подключенной директории основного компьютера **data** и наоборот. Более того, при остановке и даже удалении контейнера все данные будут оставлены в целости и сохранности.

# Подключение внешних томов к контейнерам

- Еще один механизм хранения данных - использование внешних томов. Тома - директории внутри Docker, которые можно подключать к одному контейнеру или нескольким контейнерам одновременно. В этом случае данные будут в безопасности как после перезапуска контейнера, так и после его удаления. У каждого тома должно быть имя, которое может быть человекочитаемым. Первый способ создания тома - применение команды **docker volume create**, после которой надо указать название тома (название надо придумать самим):

```
docker volume create my_volume
```

- Теперь мы можем подключить наш том к контейнеру в момент создания этого контейнера. Подключим том к директории **/var/dir**:

```
docker run -dit -v my_volume:/var/dir --name=vol_container
ubuntu:22.04 /bin/bash
```

- Том необязательно создавать заранее - можно просто подставить еще неиспользованное имя в момент создания контейнера, и том будет создан автоматически (это и есть второй способ создания тома):

```
docker run -dit -v my_volume2:/var/dir2 --name=vol_container2
ubuntu:22.04 /bin/bash
```

# Альтернативный синтаксис подключения томов

- Есть еще один способ подключения томов - он осуществляется с помощью параметра **--mount**, после которого в параметре **source** надо указать том, который мы подключаем, а в параметре **target** следует подставить директорию контейнера, куда мы подключаем том:

```
docker run -dit --mount source=my_volume3,target=/var/dir3
--name=vol_container3 ubuntu:22.04 /bin/bash
```

# Подключение томов только в режиме для чтения

- Часто может так случиться, что мы хотим подключить том к контейнеру, но одновременно хотим запретить контейнеру этот том менять - т.е. подключить только в режиме для чтения. Это делается элементарно - просто применить параметр **ro** (через двоеточие) после указания пары том:директория при создании контейнера:

```
docker run -dit -v my_volume4:/var/dir4:ro
--name=vol_container4 ubuntu:22.04 /bin/bash
```

- Если мы применяем альтернативный синтаксис, то после перечисления всех опций после параметра **--mount** через запятую надо вписать слово **readonly**:

```
docker run -dit --mount
source=my_volume5,target=/var/dir5,readonly
--name=vol_container5 ubuntu:22.04 /bin/bash
```

# Просмотр и удаление томов

- Если мы хотим посмотреть список всех существующих томов, то мы должны применить команду **docker volume ls**:

```
docker volume ls
```

```
DRIVER    VOLUME NAME
local     my_volume
local     my_volume2
local     my_volume3
local     my_volume4
local     my_volume5
```

- Для удаления томов мы сначала должны убедиться, что они не используются в контейнерах. Используемые тома удалить нельзя. Когда мы удалим все связанные контейнеры, то для удаления томов применим команду **docker volume rm**, после которой можно указать название одного или нескольких томов:

```
docker volume rm my_volume
```

- Для полного удаления сразу всех томов (кроме используемых) используется команда **docker volume prune** с параметром **-a**:

```
docker volume prune -a
```

# Организация сетей в Docker

# Введение

- Мы уже научились работать с контейнерами и даже умеем хранить в них данные. Теперь пришла пора заставить наши контейнеры взаимодействовать с операционной системой, поверх которой функционирует Docker, а также наладить связи между самим контейнерами. Сразу заметим, что все это взаимодействие так или иначе связано с сетями.

- Первое, что нам надо знать - способ, с помощью которого контейнер может обратиться к основной операционной системе. На самом деле это очень просто - Docker имеет встроенный DNS сервер, у которого есть специальное зарезервированное имя - **host.docker.internal** - это и есть доменное имя основной операционной системы. Представим, что у нас на основной системе работает web-приложение (сайт), которое использует 8000 порт и которое разрешает внешние подключения. Если мы работаем внутри контейнера, то мы может обратиться к web-приложению следующим образом (если у нас нет утилиты **curl**, то можно установить ее в контейнере при помощи команды **apt install curl**):

```
curl http://host.docker.internal:8000
```

# Проброс портов внутрь контейнера

- Очень важной частью функционала Docker является его приспособленность к прослушиванию портов, открытых на основной системе. Представим, что у нас на основной системе открыт 80 порт, но нет ни одного приложения, которое этот порт бы использовало. Однако мы можем запустить контейнер с web-сервером **nginx** внутри и указать, что 80 порт основной системы будет связан с 80 портом внутри контейнера. Таким образом приложение, заключенное в контейнер, сможет слушать запросы, идущие к основной операционной системе извне.

- Проброс портов производится при создании контейнера - используется параметр **-v**, после которого указывает порт основной операционной системы, а потом через двоеточие - порт внутри контейнера. Если мы хотим пробросить несколько портов, то конструкцию с параметром -v надо повторить нужное количество раз. Теперь же реализуем описанный выше сценарий с **nginx** и 80 портом:

```
docker run -d -p 80:80 --name=server nginx:1.24
```

# Типы сетевых драйверов внутри контейнеров

- Пришло время приступить ко взаимодействию контейнеров между собой. Для начала рассмотрим основные типы сетевых драйверов (способов работы с сетевыми подключениями) в Docker:
    - **bridge** - тип по умолчанию, позволяющий создавать свою изолированную сеть. Все контейнеры, подключенные к сети с этим драйвером могут общаться между собой без каких-либо ограничений.
    - **host** - использует сетевые настройки операционной системы, поверх которой запущен Docker. Таким образом, у контейнера будет такой-же IP адрес и MAC-адрес, как и у основной системы.
    - **none** - контейнер, подключенный к сети с этим драйвером, не сможет работать с внешними сетями. Однако будет сохранена возможность пользоваться локальной сетью внутри самого контейнера.
- Также существуют драйвера **macvlan** (одинаковый с основной системой IP-адрес, но уникальный MAC-адрес) и **ipvlan** (одинаковый с системой MAC-адрес, но уникальный IP-адрес), но они используются редко, и мы их рассматривать не будем.

# Предустановленные сети в Docker

- Для каждого основного сетевого драйвера (**bridge**, **host**, **none**) в Docker есть заранее предустановленная сеть. Увидеть их можно при помощи команды **docker network ls**:

```
docker network ls
```

```
NETWORK ID     NAME      DRIVER    SCOPE
1e3f4a210ab    bridge    bridge    local
6a2f0b677zx    host      host      local
6g5f3ed83zh    none      null      local
```

- Три перечисленные сети не могут быть удалены, кроме того - может быть только одна сеть с типом **host**, и только одна сеть с типом **none**. Поэтому в будущем мы сможем создавать/удалять (кроме первоначальной) только сети с типом **bridge**.

- Сеть **bridge** является сетью по умолчанию, поэтому при запуске контейнера, если мы не указываем сеть явно (при помощи параметра **--net**), будет применена именно она, а самому контейнеру будет назначен IP в диапазоне 172.17.0.0/16:

```
docker run -dit --name=container1 ubuntu:22.04 /bin/bash
```

```
docker run -dit --name=container2 --net=bridge ubuntu:22.04 /bin/bash
```

# Особенности предустановленной сети **bridge**

- У встроенной сети **bridge** есть два больших недостатка:
  - Во-первых, мы не можем явно назначить IP адрес контейнеру (даже если он находится в разрешенном диапазоне и пока не занят). В сетях, создаваемых вручную, такое будет возможно.
  - Во-вторых, внутри контейнера мы можем обращаться к другому контейнеру только при помощи IP-адреса (который, как уже было сказано, нельзя назначать вручную). Например, представим, что мы находимся внутри контейнера **container1** (172.17.0.5) и хотим обратиться к контейнеру **container2** (172.17.0.9). В данной ситуации мы не можем использовать имя **container2** - docker позволит использовать только полный IP-адрес:

```
ping 172.17.0.9
```

- Если мы хотим увидеть сгенерированный IP-адрес контейнера, то мы должны применить команду **docker inspect**, после которой следует подставить ID или название контейнера - в отображенной структуре в разделе **NetworkSettings -> IPAddress** и будет находится интересующее нас значение:

```
docker inspect container2
```

# Создание своих сетей

- Чтобы полноценно использовать сеть, нам надо создать ее самим со всеми необходимыми настройками. Это осуществляется командой **docker network create**, после чего следует указать уникальное названия сети. В этом случае будет создана сеть с драйвером bridge и случайной сетью и случайным шлюзом. Если мы хотим все указать точно, мы должны применить следующие параметры:
  - **--driver** - тип сетевого драйвера
  - **--subnet** - адрес сети и маска подсети
  - **--gateway** - адрес шлюза по умолчанию

- Создадим вручную настроенную сеть с названием **my_network**:

```
docker network create --driver=bridge --subnet=172.20.0.0/24
--gateway=172.20.0.1 my_network
```

# Использование своих сетей

- Наконец, создадим два контейнера (**net_container1** и **net_container2**) с нашей собственной сетью и укажем каждому явный IP-адрес (если адрес не указывать, то он будет назначен автоматически, однако сейчас мы этого не хотим):

```
docker run -dit --name=net_container1 --net=my_network --ip=172.20.0.5
ubuntu:22.04 /bin/bash

docker run -dit --name=net_container2 --net=my_network --ip=172.20.0.9
ubuntu:22.04 /bin/bash
```

- Теперь внутри контейнеров мы можем общаться с другими контейнерами как при помощи IP-адресов (четко прописанных и неизменных), так и при помощи имен контейнеров:

```
ping net_container2
```

```
ping 172.20.0.9
```

# Отключение и подключение сетей в контексте запущенных контейнеров

- У нас есть возможность отключать сети у уже работающих контейнеров и подключать к ним новые сети. Отключим сеть my_network у контейнера net_container1. Для этого применяется команда **docker network disconnect**, потом идет название сети, а потом - название контейнера:

```
docker network disconnect my_network net_container2
```

- Теперь создадим совершенно новую сеть my_network2 и подключим ее к контейнеру net_container2 (команда **docker network connect**) с явно указанным IP-адресом (параметр **--ip**):

```
# создание сети
docker network create --driver=bridge
--subnet=192.168.255.0/24 --gateway=192.168.255.1 my_network2

# подключение сети
docker network connect --ip=192.168.255.10 my_network2
net_container2
```

# Удаление сетей

- Для того, чтобы удалить сеть, нам надо убедиться, что все контейнеры, которые используют эту сеть, были остановлены. Для удаления используется команда **docker network rm**, после которой надо указать название сети. Например, удалим сеть my_network2. Но сначала остановим контейнер net_container2, который эту сеть использует:

```
# остановка контейнера
docker stop net_container2

# удаление сети
docker network rm my_network2
```

# Создание собственных образов (**Dockerfile**)

# Введение

- Использование готовых образов - отличное решение для развертывания изолированных окружений и работы в рамках этих окружений. Например, мы можем запустить несколько контейнеров с разными версиями языка Java, которые не будут конфликтовать между собой и эффективно обслуживать различные части нашего приложения.

- Однако все то, что мы делали до этого момента, имеет один недостаток. Мы применяем образы, которые нельзя использовать для работы сразу, их необходимо настраивать - устанавливать языки программирования, запускать сервера и т.д. Было бы гораздо удобнее, если бы мы сами смогли создать свой образ со всеми предустановленными технологиями, а затем развертывать уже его (без всяких дополнительных ручных манипуляций, которые могут привести к ошибкам и отличиям от ранее произведенных настроек).

- Чтобы решить обозначенную проблему, следует использовать т.н. **Dockerfile**. В рамках этого файла можно указать необходимую операционную систему, языки программирования и другие зависимости. Затем, после запуска этого файла с помощью Docker, мы получим новый образ, настроенный нужным нам образом и содержащий все необходимые нам технологии.

# Создание **Dockerfile** (теория)

- Прежде всего нам надо создать файл с названием Dockerfile и записать в него несколько нужных нам инструкций. Инструкции пишутся следующим образом: сначала некая директива, а после нее через пробел - значение для этой директивы. Далее перечислим самые простые директивы:
  - **FROM** - используется для обозначение того образа (операционной системы), на основе которого будет создаваться наш собственный образ.
  - **RUN** - директива для исполнения команд внутри командной оболочки базового образа, например, *apt update*, *apt install nginx*, *mkdir /srv/example*. Директива может применяться несколько раз.
  - **LABEL** - позволяет задать некую информацию об образе (автор, время создания, цель создания и т.д.). Также может применяться несколько раз.
  - **ENV** - предоставляет возможность для установки переменных окружения (например, для определения временной зоны) внутри образа.
  - **CMD** - после этой директивы можно задать команду, выполняемую при инициализации контейнера.

# Создание **Dockerfile** (написание файла с инструкциями)

- Создадим Dockerfile на основе операционной системы Ubuntu 22.04 с рижской временной зоной, предустановленным и запущенным сервером nginx и технологиями nodejs и npm:

```
FROM ubuntu:22.04

LABEL maintainer="John Smith"

RUN apt update && DEBIAN_FRONTEND=noninteractive apt install -y nodejs npm nginx tzdata

ENV TZ Europe/Riga

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

# Создание образа с помощью **Dockerfile**

- Чтобы создать образ из Dockerfile, мы должны выполнить команду **docker build**, затем с помощью **-t** задать название образа, а в завершение указать путь к той директории, где находится сам Dockerfile. Предположим, что мы уже находимся в директории с Dockerfile и хотим назвать наш образ server:1.0 - тогда полная команда будет следующей:

```
docker build -t server:1.0 .
```

- Теперь, когда мы выполним команду **docker images**, то среди списка образов будет присутствовать наш собственный образ *server* с версией *1.0*:

```
docker images
```

```
REPOSITORY   TAG      IMAGE ID       CREATED          SIZE
server       1.0      74f6b52a5e76   5 minutes ago    955MB
```

# Запуск образа, созданного с помощью **Dockerfile**

- Запуск наших образов ничем не отличается от запуска готовых образов - применяется все та же команда **docker run**, а после нее подставляются все стандартные параметры. Единственный нюанс - нам следует подставить название нашего образа. Запустим контейнер site-dev из образа server:1.0 и одновременно перебросим внутрь 80 порт:

```
docker run -d --name=site-dev-1 -p 80:80 server:1.0
```

Теперь, если мы перейдем в браузере по адресу http://localhost, то увидим начальную страницу web-сервера nginx. Поздравляем, вы запустили первый контейнер, который основан на вашем личном образе!

# Использование аргументов внутри Dockerfile

- При использовании Dockerfile часто используются аргументы - допустим нам необходим nginx, но мы хотим разрешить создателям образа самим определять версию. Для этого в Dockerfile надо применить параметр **ARG**, после которого следует вписать название аргумента. В результате мы получим возможность подставлять эти названия на то место, где необходимо указать версию сервера:

```
FROM ubuntu:22.04

LABEL maintainer="John Smith"

ARG NGINX_VERSION

RUN apt update && DEBIAN_FRONTEND=noninteractive apt install -y nodejs npm nginx=${NGINX_VERSION} tzdata

ENV TZ Europe/Riga

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

Теперь при создании образа надо указывать аргумент NGINX_VERSION именно той версии, которая нам нужна в данный момент:

```
docker build --build-arg="NGINX_VERSION=1.18.*" -t server:2.0 .
```

# Подгрузка файлов в образ при помощи Dockerfile

- Иногда нам может потребоваться подготовить образ, который содержит не только установленные программы, а уже части нашего приложения или даже все приложение. Самый простой пример - сейчас у нас отображается страничка nginx по умолчанию, а хорошо было бы пробросить внутрь полноценный сайт. Чтобы это реализовать, надо использовать директиву **COPY** (может использоваться сколько угодно раз), после которой следует указать директорию на основной системе, а потом (через пробел) директорию внутри контейнера:

```
FROM ubuntu:22.04

LABEL maintainer="John Smith"

ARG NGINX_VERSION

RUN apt update && DEBIAN_FRONTEND=noninteractive apt install -y nodejs npm nginx=${NGINX_VERSION} tzdata

ENV TZ Europe/Riga

COPY ./site /var/www/html

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

```
docker build --build-arg="NGINX_VERSION=1.18.*" -t server:3.0 .
```

# Предложение для открытия портов

- Внутри Dockerfile мы можем предложить операционной системе, использующей контейнер на основе нашего образа, открыть определенные порты - для этого используется директива **EXPOSE** (может быть использована многократно):

```
FROM ubuntu:22.04
LABEL maintainer="John Smith"
ARG NGINX_VERSION

RUN apt update && DEBIAN_FRONTEND=noninteractive apt install -y nodejs npm nginx=${NGINX_VERSION} tzdata

ENV TZ Europe/Riga
COPY ./site /var/www/html

EXPOSE 80
EXPOSE 443

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

- При создании контейнера (из нашего образа) мы можем указать параметр **-P** - в результате случайные порты системы будут переброшены в порты контейнера:

```
docker build --build-arg="NGINX_VERSION=1.18.*" -t server:4.0 .
```

```
docker run -d --name=site-dev-4 -P server:4.0
```

# Установка рабочей директории

- На этапе создания образа нам могут потребоваться переходы по директориям. Для этого есть директива **WORKDIR** - она разрешает перейти в те директории, которые пока не созданы (т.е. перед переходом она их создаст автоматически). Предположим, что в директории сайта надо установить и скомпилировать JavaScript зависимости - применим **WORKDIR**, чтобы это сделать:

```
FROM ubuntu:22.04
LABEL maintainer="John Smith"
ARG NGINX_VERSION

RUN apt update && DEBIAN_FRONTEND=noninteractive apt install -y nodejs npm nginx=${NGINX_VERSION} tzdata

ENV TZ Europe/Riga
COPY ./site /var/www/html

WORKDIR /var/www/html
RUN chmod +x start.sh

CMD ["bash", "start.sh"]
```

```
docker build --build-arg="NGINX_VERSION=1.18.*" -t server:5.0 .
```

```
docker run -d --name=site-dev-5 -p 80:80 server:5.0
```

# Выполнение команды при инициализации контейнера

- Мы уже знаем, что при инициализации контейнера отрабатывает та команда, которая прописана в **CMD**. Однако при запуске **docker run** у нас есть возможность ее переписать, подставив свою - например, **/bin/bash**. Однако у нас есть еще одна директива - **ENTRYPOINT**, которая имеет более высокий приоритет, чем **CMD**, и она не может быть перезаписана стандартным способом (только с помощью параметра **--entrypoint**). Если же мы поставим свою команду, она будет считаться аргументом к тому, что прописано в ENTRYPOINT:

```
FROM ubuntu:22.04
LABEL maintainer="John Smith"
ARG NGINX_VERSION

RUN apt update && DEBIAN_FRONTEND=noninteractive apt install -y nodejs npm nginx=${NGINX_VERSION} tzdata

ENV TZ Europe/Riga
COPY ./site /var/www/html

WORKDIR /var/www/html
RUN chmod +x start.sh

ENTRYPOINT ["bash", "start.sh"]
```

```
docker build --build-arg="NGINX_VERSION=1.18.*" -t server:6.0 .
```

```
docker run -d --name=site-dev-6 -p 80:80 server:6.0
```

# Смена пользователя в контейнере

- По умолчанию процессы в контейнере запускаются от имени суперпользователя *root*. Однако это может быть небезопасно, поэтому надежнее будет запускать процессы от имени другого пользователя. Для этого следует применить директиву **USER**, которая переключается на другого уже существующего пользователя. С этого момента все будет запускаться только от его имени.

```
FROM ubuntu:22.04
LABEL maintainer="John Smith"

RUN apt update && DEBIAN_FRONTEND=noninteractive apt install -y nodejs
COPY ./site /var/www/html

WORKDIR /var/www/html

RUN useradd john
USER john

ENTRYPOINT ["node", "index.js"]
```

```
docker build -t server:7.0 .
```

```
docker run -d --name=site-dev-7 -p 80:80 server:7.0
```

# Работа с многоконтейнерными приложениями (Docker Compose)

# Введение

- Развитие технологий не стоит на месте - программное обеспечение становится таким сложным и многослойным, что в большом количестве случаев его приходится делить на компоненты. Иногда только таким способом можно работать над одной и той же программой без опасения нарушить какой-либо важный участок кода.

- Кроме возросшей сложности есть еще одна проблема - разные части большого приложения могут использовать технологии, которые могут конфликтовать с другими технологиями (в рамках одной операционной системы) - например, разные версии языков программирования - Java или PHP.

- Дополнительная проблема состоит в том, что современные приложения работают под большой нагрузкой. Было бы разумно выделить критические места программы в отдельный компонент и уже в рамках этого компонента использовать, например, балансировщики нагрузки.

- Для решения этих проблем в Docker существует надстройка **Docker Compose**, позволяющая запускать множество контейнеров одновременно (для каждого компонента системы) и осуществлять управление этими контейнерами.

# Начало работы с Docker Compose

- Чтобы проверить, что у нас установлен Docker Compose, нам следует перейти в командную строку и выполнить команду, которая отображает версию необходимой нам программы:

```
docker compose version
```

- Для начала работы с контейнерами в контексте Docker Compose нам потребуется файл **docker-compose.yml**. В этом файлы мы и перечислим необходимые нам правила создания и взаимодействия контейнеров. Как можно понять из расширения **yml**, мы обязаны использовать формат YAML при написании файла.

- По своему синтаксису YAML похож на язык программирования Python - здесь для отделения смысловых блоков также используется система отступов, однако табы не применяются - только пробелы. Данные представлены в виде пары **ключ: значение**, где ключ - название переменной, а значение - сами данные. Строки могут быть обрамлены в кавычки - двойные или одинарные, но это необязательно, в свою очередь значения последовательностей (массивов) отображаются с помощью тире перед этими значениями с соответствующим отступом. На следующей странице представлены наиболее ходовые примеры.

# Примеры синтаксиса **YAML**

Последовательность (массив):

```
food:
 - apple
 - banana
 - bread
```

Словарь:

```
person:
  name: "John Smith"
  age: 25
  gender: "male"
```

Объявление свойства со строковым значением:

```
job: programmer
```

Объявление свойства со значением null:

```
state:
```

Объявление свойства с числовым значением:

```
salary: 1299.99
```

Объявление свойства логического типа:

```
success: true
```

# Основные свойства для конфигурационного файла **docker-compose.yml**

- **version** - корневое строковое свойство, которое указывает версию конфигурационного файла (в зависимости от версии разрешены те или иные свойства). Актуальная версия - "3.8".
- **services** - обязательное корневое свойство-словарь, содержащее описание контейнеров нашей системы, а также настройки этих контейнеров.
    - **image** - свойство внутри настроек контейнера, которое устанавливает тот образ, на основе которого будет работать конкретный контейнер.
    - **ports** - свойство-массив внутри настроек контейнера, устанавливает отношение порт на основной системе:порт внутри контейнера - 8080:80.
    - **environment** - свойство-словарь внутри настроек контейнера, где следует устанавливать переменные окружения контейнера.
    - **volumes** - свойство-массив внутри настроек контейнера, которое устанавливает отношение том/директория на основной операционной системе:директория внутри контейнера.
    - **build** - свойство-словарь, используется, если нам нужна сборка из Dockerfile, путь к которому указывается во внутреннем свойстве словаря **context**
    - **networks** - свойство-массив внутри настроек контейнера, устанавливает список подключенных к контейнеру сетей.
- **volumes** - корневое свойство-словарь, которое используется для создания подключаемых томов для контейнеров. Позволяет использовать вложенные свойства, например **driver: local**.
- **networks** - корневое свойство-словарь, которое используется для создания сетей. Может содержать внутри дополнительные параметры, например **driver: bridge** или **subnet: "172.16.238.0/24"**.

# Пример конфигурационного файла **docker-compose.yml** (WordPress)

```
version: '3.8'

services:
  wordpress:
    image: wordpress
    restart: unless-stopped
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: exampleuser
      WORDPRESS_DB_PASSWORD: examplepass
      WORDPRESS_DB_NAME: exampledb

  db:
    image: mysql:5.7
    restart: unless-stopped
    environment:
      MYSQL_DATABASE: exampledb
      MYSQL_USER: exampleuser
      MYSQL_PASSWORD: examplepass
      MYSQL_RANDOM_ROOT_PASSWORD: '1'
```

# Запуск и остановка контейнеров из **docker-compose.yml**

- Для запуска приложения следует перейти в командной строке в ту директорию, где находится **docker-compose.yml** и выполнить следующую команду:

```
docker compose up
```

Контейнеры будут созданы и запущены, однако наша командная строка будет заблокирована. Для запуска контейнеров на фоне необходимо добавить к команде параметр **-d**:

```
docker compose up -d
```

- Чтобы остановить запущенные контейнеры, следует опять перейти в директорию с **docker-compose.yml** и выполнить нижеприведенную команду:

```
docker compose stop
```

- Если же мы хотим не просто остановить контейнеры, но и сразу же удалить их, то следует использовать несколько другую команду:

```
docker compose down
```

# Работа с уже созданными контейнерами

- Когда многоконтейнерное приложение запущено, нам может потребоваться посмотреть логи его работы - для этого используется нижележащая команда (ее надо выполнять в директории, содержащей файл **docker-compose.yml**):

```
docker compose logs
```

- Если по какой-то причине мы хотим перезапустить наше приложение или один из контейнеров приложения, необходимо применить команду **docker compose restart** (если хотим перезапустить только один контейнер, то после команды надо указать его название):

```
docker compose restart
```

- Теоретически для запуска приложения нам хватает команды **docker compose up -d** - если контейнеров нет, то она их создаст и запустит. Существует также команда **docker compose start** - она умеет запускать контейнеры, если они уже есть (может понадобиться, если некий контейнер аварийно выключился и его надо включить вновь - для этого после команды надо указать его имя):

```
docker compose start
```

# Поведение компонентов docker-compose.yml после остановки

- В настройках компонентов иногда мы можем увидеть следующее свойство:

```
restart: always
```

Свойство **restart** отвечает за поведение контейнера, если он остановился (в результате ошибки, отсутствия процессов или вручную со стороны пользователя), а затем главный процесс docker был перезапущен. Всего данное свойство может иметь 4 значения:

- **always** означает, что всегда будет предпринята попытка автоматически перезапустить контейнер
- **unless-stopped** - автоматический перезапуск произойдет только если контейнер не останавливали вручную (с помощью **docker stop**)
- **on-failure** - автоматический перезапуск произойдет только если контейнер остановился по причине ошибки
- **no** (значение по умолчанию) - автоматический перезапуск будет отключен

# Порядок запуска компонентов в docker-compose.yml

- Часто компоненту для запуска может понадобиться другой (уже работающий) компонент. Например, для сайта должна быть готова база данных. Чтобы решить эту проблему, Docker предлагает свойство **depends_on**, которое позволяет перечислить те компоненты, от которых зависит данный компонент.

- Ниже представлены три компонента - web, redis и db - чтобы запустить web, redis и db уже должны работать. Для осуществления этой задачи как раз используется свойство **depends_on**:

```
version: '3.8'

services:
  web:
    build:
      context: .
    depends_on:
      - db
      - redis
  redis:
    image: redis:7.2.3
  db:
    image: postgres:16.1
```

# Перенос Docker образов между компьютерами

# Перенос образов на другой компьютер с помощью архивирования

- Если нам надо перенести некий образ (уже существующий или сгенерированный нами) на другой компьютер, то одним из базовых способов является его архивирование и разархивирование в другом месте. Для осуществления этого способа применяется команда **docker save**, потом применяется параметр **-o** (чтобы сохранить данные именно в файл), затем указывается имя файла, а в самом конце - названия тех образов, которые мы хотим заархивировать. Для примера запишем образ **mysql:5.7** в файл **archive.tar**:

```
docker save -o archive.tar mysql:5.7
```

- Теперь нужно перенести получившийся файл **archive.tar** на другой компьютер (любым способом). После этого, нам надо применить команду **docker load** с параметром **-i**, а затем подставить путь к файлу **archive.tar**. В результате все образы из архива (в нашем случае - **mysql:5.7**) будут установлены на новом компьютере:

```
docker load -i archive.tar
```

# Создание образа из контейнера

- На прошлом шаге мы успешно заархивировали и перенесли наш образ на другой компьютер. Однако очень часто нам надо перенести не образ, а только контейнер, где уже произведены некоторые работы - созданы файлы, запущены программы и т.д. На самом деле это очень просто - надо всего лишь создать образ из контейнера, а потом повторить все действия из предыдущего шага. Для этого применяется команда **docker commit**, после которого идет название контейнера, а в самом конце - название нового образа. Представим, что у нас есть контейнер с названием **my_container**, из которого мы сделаем образ с названием **my_image:1.0**:

```
docker commit my_container my_image:1.0
```

# Использование Docker Hub

- Более удобную возможность перемещать образы между компьютерами предоставляет сайт Docker Hub https://hub.docker.com/. Мы можем зарегистрироваться там (например, под именем **user**) и создать токен доступа (**Account Settings -> Security -> New Access Token**). Затем мы можем создать свой репозиторий, где будем хранить версии какого-либо образа (**Repositories** -> **Create Repository**) - например, под именем **test**. Репозитории могут быть публичными (в неограниченном количестве) и приватными (бесплатно доступен только один). Для безопасности создадим именно приватный репозиторий - он не будет виден другим пользователям Docker Hub.

- Теперь на нашем компьютере аутентифицируемся в контексте Docker Hub. Для этого перейдем в командную строку и введем команду **docker login** затем в контексте параметра **-u** впишем имя нашего пользователя, в контексте параметра **-p** - токен доступа, а в самом конце ссылку **docker.io**:

```
docker login -u user -p ТОКЕН_ДОСТУПА docker.io
```

# Загрузка образов на Docker Hub

- На данный момент у нас есть репозиторий **user/test**, куда мы можем загрузить наш образ, например, **my_image:1.0**. Важный момент - мы должны загрузить образ, название которого соответствует репозиторию. Поэтому сначала надо клонировать существующий образ с новым названием - для этого применим команду **docker tag**, после которого надо сначало указать название существующего образа, а затем - имя копии:

```
docker tag my_image:1.0 user/test:1.0
```

- Теперь мы можем, наконец, загрузить наш образ на Docker Hub. Это делается при помощи команды **docker push**, после которой указывается название образа:

```
docker push user/test:1.0
```

Если данный образ нам понадобится на каком-либо другом компьютере, то мы будем должны аутентифицироваться при помощи команды **docker login**, а затем применить уже известную нам команду **docker pull** для загрузки образа:

```
docker pull user/test:1.0
```