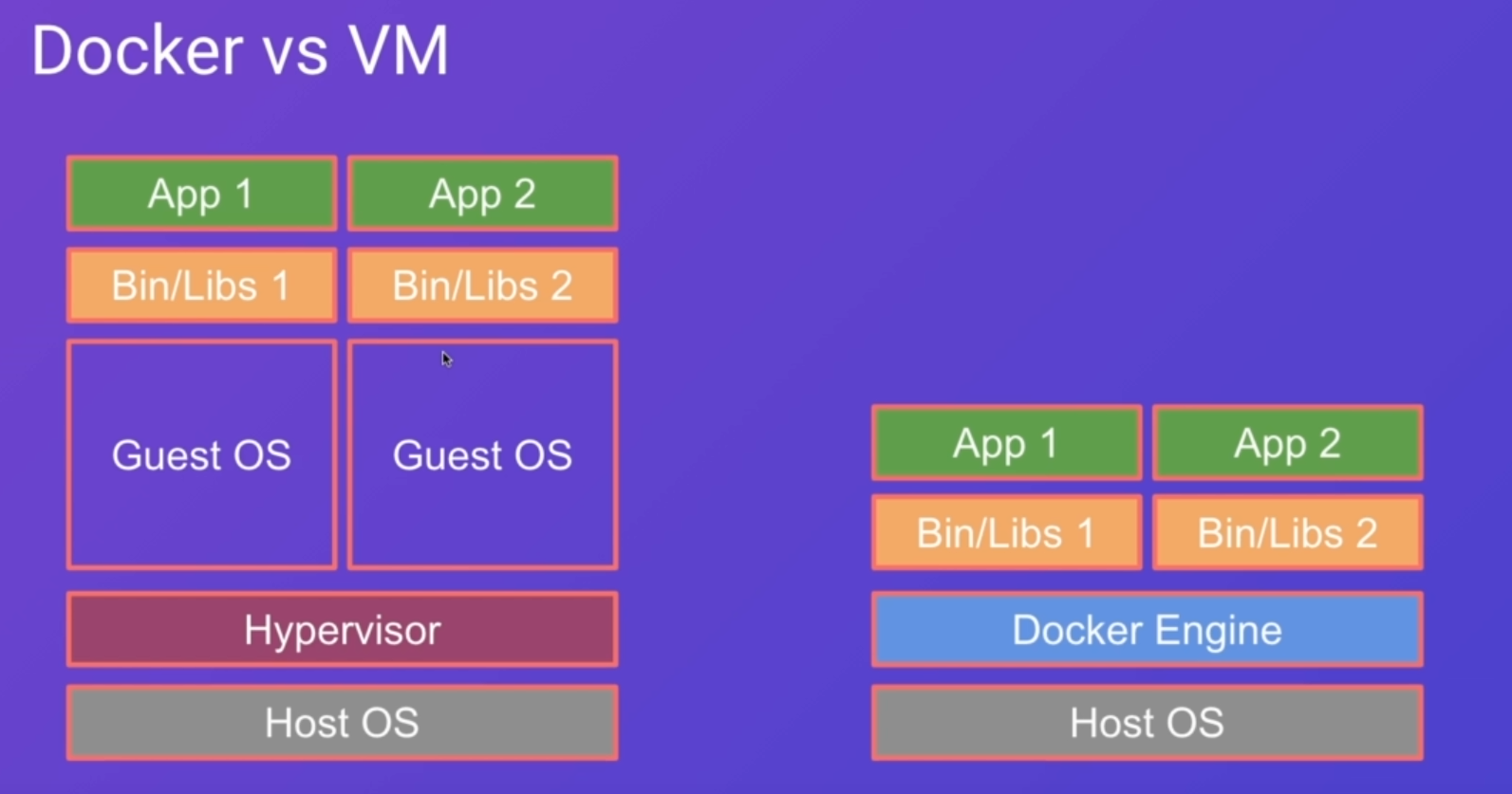
***DOCKER***



Сравнение с виртуальной машиной.

У нас есть хостовоя ОС на которой установлен Docker Engine или Hypervisor (гипервизор тоже может быть хостовой ОС).

Далее на Hypervisor-е запускаются гостевые ОС, которые внутри уже содержат бинарники и библиотеки необходимые для запуска приложения и само приложение. Если необходимо запустить два приложения, запускается дополнительная гостевая ОС, на которую уже размещают бинарники и библиотеки второго приложения и само второе приложение.

У Docker-а на хостовую ОС устанавливается Docker Engine (один) и на него устанавливаются библиотеки и приложения, при этом нет прослойки в виде гостевой ОС.

Docker не имеет ничего общего с виртуальной машиной (если только изолированность).

рассмотрим как работают процессы в линукс

есть процесс, в котором мы сейчас находимся и мы хотим запустить процесс на node(напр). Что будет происходить, когда мы запустим процесс на node:

1. создастся новый процесс при этом скопировав предыдущий(дешёвая операция)был баш мы делаем форк и появляется второй баш,
2. далее как только до него доходит очередь он запускается и подменяется необходимым процессом. это делается с помощью execv.
3. у нового процесса появится procesID и он будет функционировать в рамках этого процесса.

В данном случае первый процесс не изолирован, он может получать доступ к диску, может что-то удалить, а если второй процесс будет использовать ту же область диска, то первый может удалить нужную второму информацию.

В линуксе предвестником изоляции была команда chroot

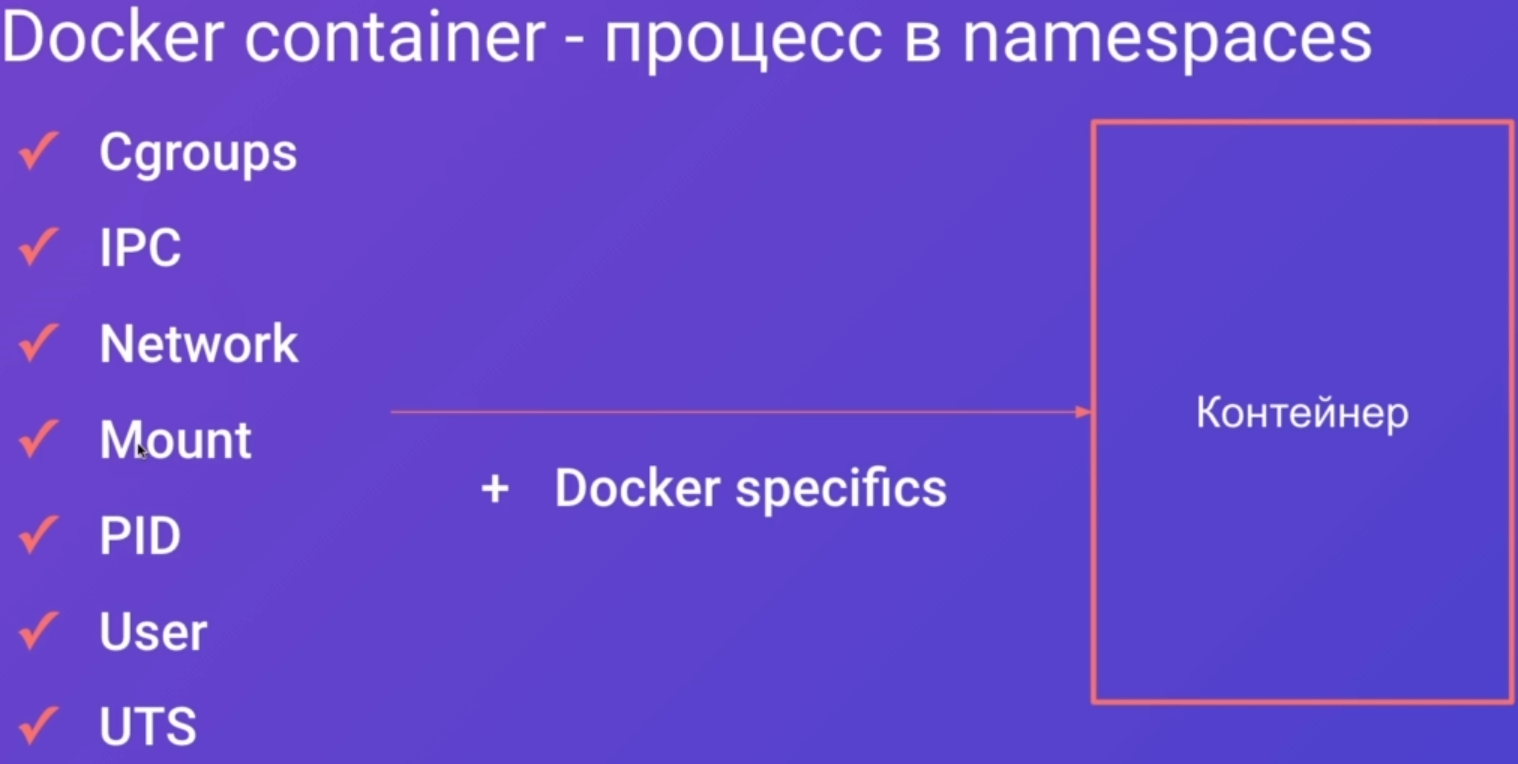
позволяет сменить корневую директорию

docker для изоляции использует namespace.

Но по аналогии можно сказать, что мы запустим процесс, а затем сменим ему root-овую директорию на к-л кастомную и он не будет видеть другие бинарники и библиотеки внутри своего изолированного пространства. В рамках этой изоляции мы можем скопировать другие бинарники и запускаться с ними в рамках этой изоляции не получая доступ к дисковому пространству.

что есть namespace?

когда запускается контейнер, создается новый namespace. namespace имеет несколько наборов это



* Cgroups управляет ограничениями по памяти и процессору для конкретного namespace.
* IPC – коммуникация между процессами
* Свой Network
* Mount – говорит какие директории доступны какие нет.
* processID который может повторяться с processID текущего Если мы запускаем на хосте processID должен быть уникален, если мы запускаем в namespace, то processID может быть свой
* User может быть аналогичный и он никак не будет конфликтовать с корневым user host машины
* и UTS

Кроме всего этого есть специфичные для docker обвязки

Контейнер - это некая изолированная часть, изолированный namespace, который запускается на ядре хостовой машины и фунциклирует изолированно, за исключением того, что он получает доступ к ресурсам машины.

Благодаря изоляции, можно запускать разные библиотеки и это не будет никак конфликтовать из-за изолированности контейнеров.

Контейнер - это изолированный namespace с дополнительными обвязками docker, в котором запускаются приложения, кладутся библиотеки для работы с ядром нашей ОС, после запуска этот контейнер работает изолированно. При этом docker дает удобный api для взаимодействия.

ОГРАНИЧЕНИЕ: МЫ НЕ МОЖЕМ ЗАПУСКАТЬ ARM-ОВСКУЮ ОС НА x86 И ОБРАТНО. Поэтому если контейнер сбилджен для ARM, то и запускать его нужно на ARM-е.

Из чего состоит docker?

Состоит из 3 частей:

1 - клиент CLI

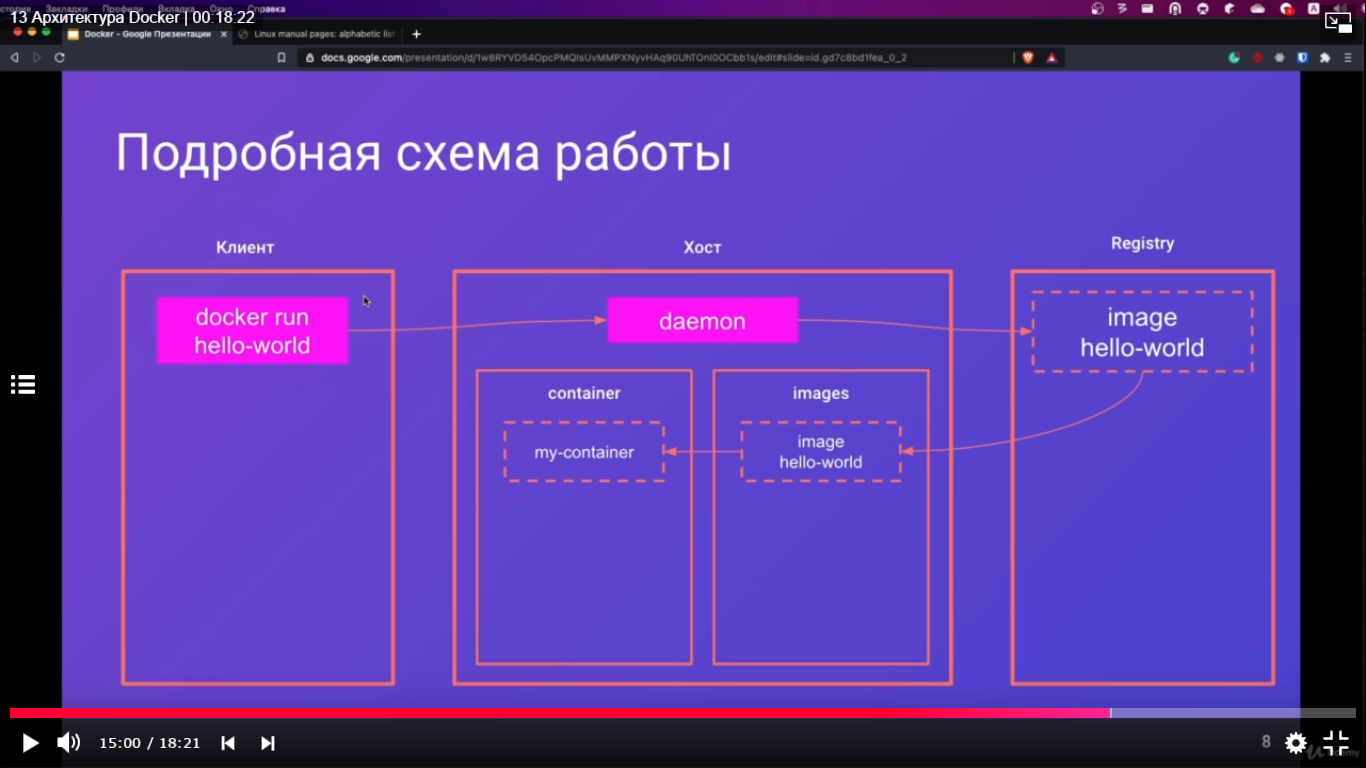
2 - API

3 - Docker host

Когда мы нажимаем docker ps мы на самом деле запускаем не docker, а пользуемся клиентом к docker. Это удобная CLI утилита, которая позволяет взаимодействовать с API docker-a.

Docker host это процесс хостовый (docker demon) который и работает на этой машине поверх него есть API и когда мы пишем docker ps, мы отправляем команду из нашей CLI на api и docker её обрабатывает. Можно даже написать свой CLI для взаимодействия с docker.

Схема работы



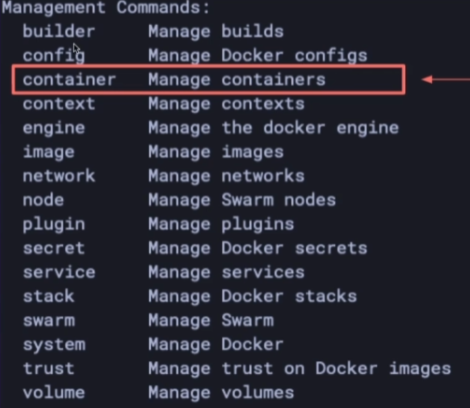
У нас есть клиент, в рамках которого мы делаем docker run hello-world

После этого он делает запрос по api к хосту, на этом хосте крутится докер демон, он проверяет есть ли у нас такой image локально, если его нет идет в общий registry(интернет, также мы можем сами создавать registry) после этого он скачивает нужный ему image и после этого он начинает запускать контейнер (создается новый namespace и передает туда этот image, он распаковывается ( бинарники, либы) всё что нужно чтобы этот image запустить и уже в этом запускается наше приложение.



и их жизненным циклом.

Если выполнить docker help то можно увидеть различные объекты, над которыми мы можем выполнить различные команды

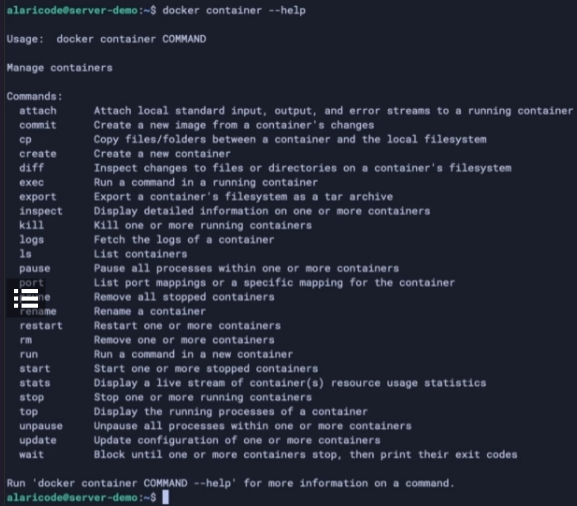


все эти сущности находятся внутри docker

Начнём с команды container – это сущность, отвечающая за работу с докер контейнерами, но при выполнении команд мы её будем опускать. Почему?

Выведем все команды для работы с контейнером для этого наберем

docker container –help



но если вызвать ту же команду без префикса container, то мы получим



т.о. мы видим большинство команд и без этого префикса. Эти команды вынесены на верхний уровень для удобства из-за их оч. частого применения.

Теперь перейдем к жизненному циклу:

когда мы делаем docker run у нас скачивается image, этот image затем запускается и превращается в контейнер, который уже содержит процесс, библиотеки, все что нужно. какой жизненный цикл происходит дальше?

После docker run контейнер принял состояние running. если мы хотим его остановить мы делаем команду docker stop, которая переводит его в состояние stopped, если мы хотим его убить(потому что процесс повис или мы не можем его остановить) мы делаем docker kill это тоже приведет контейнер в состояние stopped.

Из состояния stopped мы можем опять его вернуть в состояние running командой docker start. Также есть команда docker restart, которая сделает последовательно docker stop, затем docker start.

Также процесс можно поставить на паузу командой docker pause и он перейдет в положение paused. Вернуть обратно – docker unpause.

В жизненный цикл мы можем войти не только через docker run, запустив контейнер, но и создав контейнер из image выполнив docker create. тогда он у нас появится в состоянии stopped.



Если мы хотим совсем избавиться от контейнера мы делаем docker remove, который убирает контейнер из состояния stopped или docker removed с флагом -f (форсед), которая даже если контейнер запущен его убьет и удалит.

Контейнер является statefull, т.е. если мы в нем что-то удалили / изменили то эта информация внутри контейнера сохраняется, но если мы его ремувим, мы теряем все состояние контейнера и следующий docker run создаст контейнер «С нуля». а при перемещении между состояниями running, stopped и paused его состояние сохраняется.

РАССМОТРИМ КОМАНДЫ ДЛЯ РАБОТЫ С КОНТЕЙНЕРОМ

- **docker ps** – просмотр всех запущенных контейнеров



- здесь отображаются идентификатор контейнера,

- image из которого он был построен,

- команда, которая у него выполняется

- когда он был создан,

- его текущий статус,

- порты и

- идополнительные имена контейнера.

- **docker ps -а** – отображает все контейнеры (даже остановленные)



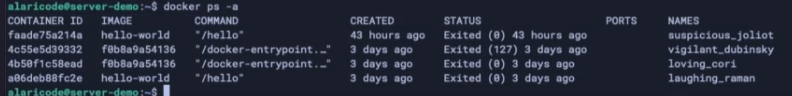
каждый раз, когда мы делаем docker run, мы создаем из image новый контейнер.

если у него в status отображается exited(0) это значит, что он вышел нормально, если не ноль, то эта цифра обозначает код ошибки(напр. 127)

если мы хотим удалить контейнер. мы к нему можем обратиться либо по имени, либо по идентификатору. Имена, если мы их не задаем ему, генерятся сами.

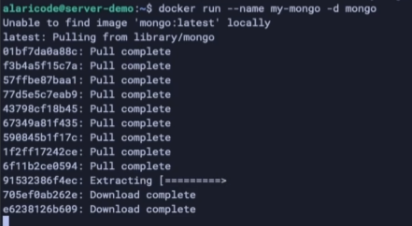
 - удаление по имени

 - удаление по id



запустим контейнер и поиграемся с его жизненным циклом

запустим mongo db, укажем ему имя, параметр -d (detached)(что значит, что мы отделяем этот процесс от текущего процесса Bush) если не указать этот параметр он будет здесь располагаться и как только мы сделаем ctrl + C остановив процесс у нас остановится и контейнер. флаг -d позволяет сделать форк от текущего процесса и запустить его в отдельном процессе, а не в текущей нашей сессии. и укажем наш контейнер mongo(пока без прокидывания портов)



выполним docker ps и увидим запущенный контейнер



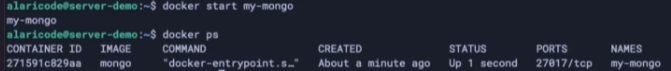
остановим контейнер при выполнении команды нажать Tab (Позволяет быстро останавливать контейнер)



убедимся в этом набрав docker ps



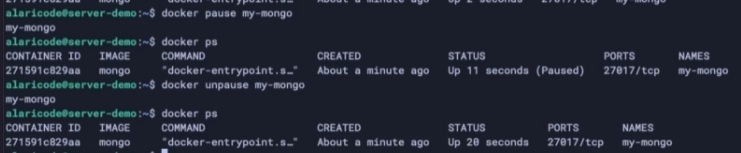
запустим его и проверим



рестартанем и проверим



поставим на паузу проверим и вернем как было



если мы хотим убить процесс можем выполнить

**docker kill** –



посылаемые системные сигналы

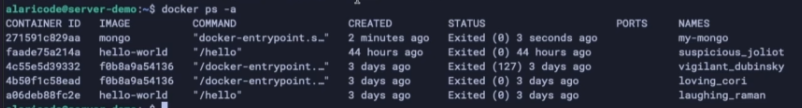
когда мы говорим **docker stop** мы посылаем системный сигнал SIGTERM, который говорит завершить процесс, но в если в течении какого-то времени контейнер не останавливается, то docker stop сам пошлет ему сигнал **docker kill**

**docker pause** остановит все процессы внутри контейнера

при отправке **docker kill** мы можем передать еще и конкретный сигнал например – signal=1 в наш монго



и это убьет процесс



удаление всех контейнеров, которые остановлены

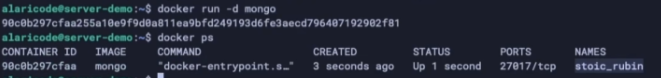
**docker container prune**

проверим



допустим мы запустили контейнер и забыли указать ему имя,

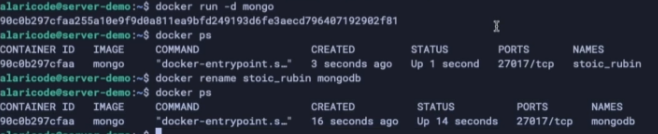
тогда запустится контейнер с каким-то рандомным названием



чтобы его переименовать пишем

**docker rename old\_name new\_name**

и т.о. мы можем изменить имя контейнеру на необходимое





посмотрим, как можно смотреть логи контейнера и как можно его инспектировать

мы можем посмотреть статистику по всем запущенным контейнерам с помощью команды

**docker stats**

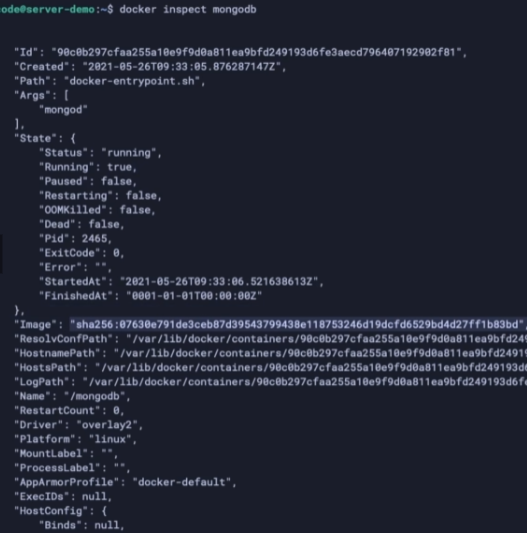


данная команда в режиме реального времени будет выводить загрузку ЦПУ, memory юзач network input output и id-к процесса

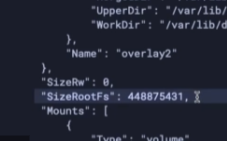
если выполнить команду

**docker inspect mongodb**

мы получим json в котором описана вся информация о нашем контейнере(его стэйт, image и пр.)



если передать параметр -s можно получить его занятый size



из этого json-а можно доставать отдельные элементы и их характеристики, например



это полезно, когда мы делаем утилиты позволяющие получить статусы контейнеров и благодаря этому из inspect мы можем вытащить только нужную информацию.

ЛОГИ

контейнер, когда работает он генерит в течении своей жизни логи

чтобы их посмотреть набираем:



мы видим все логи, которые на текущий момент написал контейнер

чтобы искать какую-то конкретную информацию нужно использовать линуксовские команды, например мы хотим вытащить всё, где в логах есть id

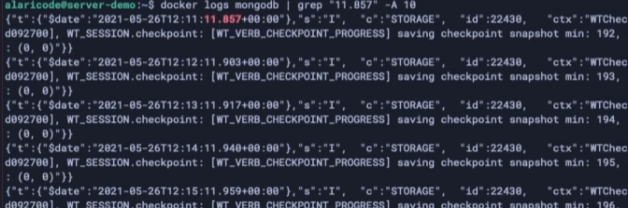


или часть какую-то

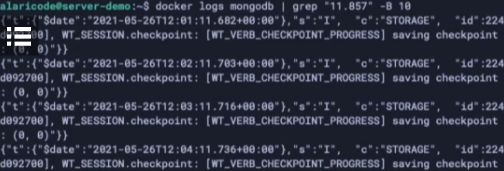


так мы можем искать например отдельные ошибки.

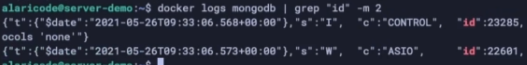
также можно передать доп. параметры (напр. -А 10 - говорящее, что мы вытащим всё, что будет after текущей строки(10 строк после текущей строки)



или -B 10 (вытащит 10 сток до найденной строки)



мы можем ограничить число выводов напр. двумя



В «» мы на самом деле передаем regex чем можем задавать сам запрос



если использовать флаг -f то мы будем следовать за логами и когда произойдет, мы увидим его вживую без дополнительной команды

выйти из этого режима: CTRL + C

можно сохранять логи в файл (например test.txt)



и если мы посмотрим в этот файл, мы получим те же логи

также можно совместить и искать с выводом результата в отдельный файл



вывести результат поиска

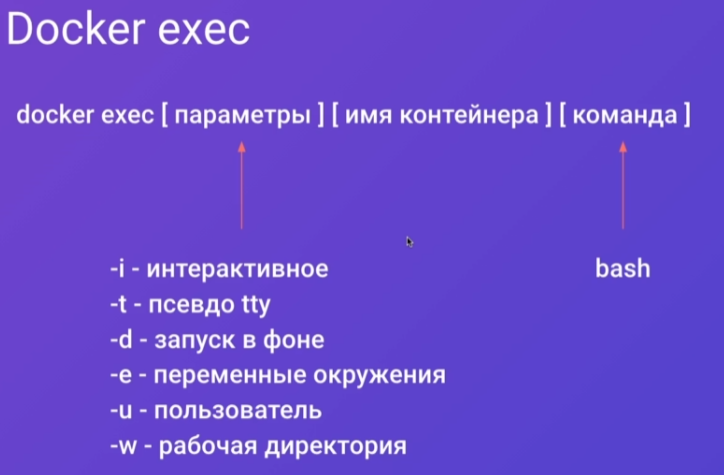


Т.О. можно контролировать сколько контейнер занимает оперативной памяти и ест процессора, и вытаскивать логи для анализа и контроля.



разберем как мы можем выполнять команды вручную внутри контейнера и залезать внутрь контейнера и смотреть что происходит.

это часто необходимо, когда нужно провести отладку запущенного приложения, например нам нужно посмотреть файлы, которые положились в рамках билда, на сколько они соответствуют тому, что мы хотели, может мы неправильно написали докер файл, по которому это сбилдилось.



поэтому часто приходится залезать в контейнер для контроля.

Пример исполнения команды: когда нам нужно забэкапить mongo db мы это можем сделать, исполнив команду бэкапа внутри контейнера, затем вытащить оттуда этот бэкап.

Рассмотрим из чего состоит эта команда.

Команда называется **exec**, далее мы можем передать набор параметров, далее имя контейнера и затем команда.

Рассмотрим параметры:

- i – говорит о том, что у нас вызывается интерактивный бот

- t – создает псевдо tty(грубо говоря мы ssh-ся внутрь контейнера и получаем интерактивное окружение в котором можем что-то делать)

- d – запуск задачи в фоне(вне текущего процесса баш в котором мы находимся)

- e – дает возможность передать переменные окружения

- u – задается пользователь, из-под которого мы хотим выполнить эту команду

- w – рабочая директория, в которой мы будем выполнять команду

по сути, мы это все можем выполнить внутри контейнера с разными параметрами.

Рассмотрим, как это работает:

запускаем mongo контейнер, в котором мы будем делать команды

допустим мы хотим указать какую-то директорию и получить текущий рабочий путь (напр. говорим, что мы будем исполнять команду, которую мы скажем дальше внутри root, далее указывает контейнер mongo и далее саму команду (напр pwd) pwd – выводит текущую рабочую директорию)

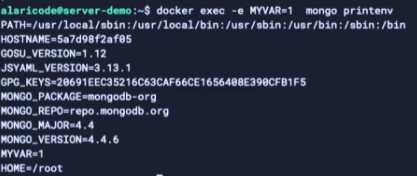


так как мы передали root то и получили root, если бы не передавали бы этот параметр, то исполняли бы совсем в корне



переменные окружения:

передадим переменную MYVAR=1 и сделаем printenv которая нам напечатает переменные окружения, которые есть внутри контейнера



как видим в перечне переменных окружения в контейнере присутствует и переменная, добавленная нами Эти переменные окружения появляются у нас в рамках сборки контейнера

теперь рассмотрим интерактивный режим, когда мы можем перейти в него и повзаимодействовать с самим контейнером

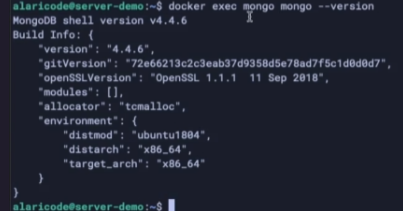
после указания bash в команде это указание того, что мы входим в bash и это bash-скриптинг



теперь рассмотрим выполнение не в интерактивном режиме, а с какими-то простыми командами

к примеру, мы хотим получить версию mongo db

(первое mongo это название контейнера, далее команда



теперь сохраним output нашего контейнера куда-то в файл

по аналогии с тем, что делали ранее пишем



с помощью ls проверим, что у нас есть

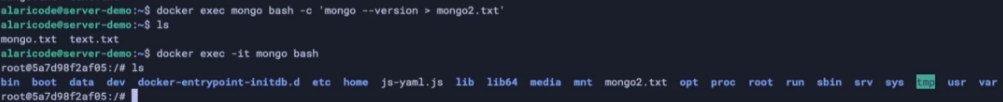


есть файл mongo.txt

при этом важно заметить, что до оператора > это одна команда, а после него другая и эта другая команда исполняется уже не в контейнере

что если мы хотим эту штуку сделать тоже внутри контейнера

мы после монго указываем bash и команду берем в кавычки (говоря т.о. исполни это всё) далее зайдем в контейнер и проверим наличие файла

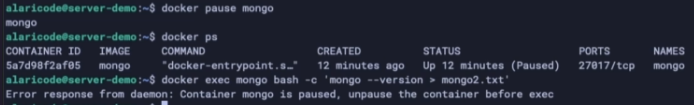


как видим, снаружи он не создался, а внутри контейнера есть.

это к тому, что нужно аккуратно относится к пайплайну, так как нужно понимать, где заканчивается команда, которая выполняется на контейнере mongo и начинается команда хостовой машины.

теперь посмотрим, что будет происходить, если мы попробуем выполнить команду на контейнере, который остановлен или на паузе

1 - на паузе



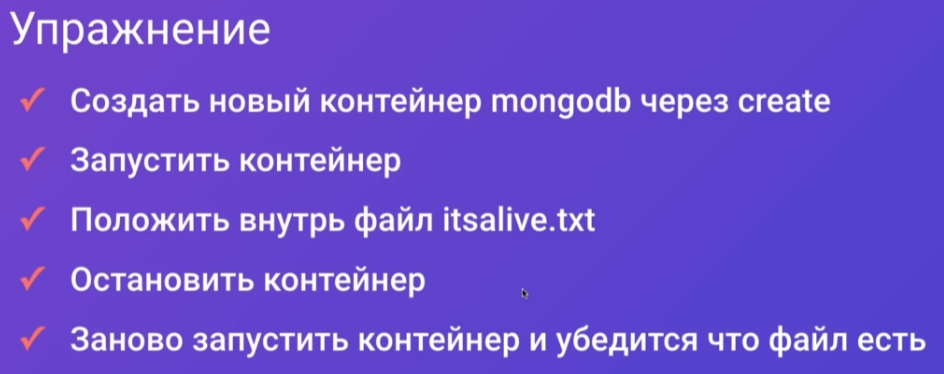
как видим, мы не можем выполнить пока он на паузе, то же и со stop



любые действия можно проводить с контейнером, только если он running.

Работа с контейнером, практика.

что будем делать:



мы не запускаем контейнер через run, а делаем create

запускаем созданный контейнер и кладем внутрь файл, затем останавливаем контейнер и запускаем его заново. после этого запуска проверяем, что state контейнера сохранился и у нас есть сохраненный ранее файл.

1. Создаем контейнер (дадим ему имя mongodb, здесь мы не передаем флаг -d (дитачт), потому как мы не запускаем его, мы его просто создаем из image mongo)



1. проверяем



1. запустим его



1. проверяем



контейнер запущен, теперь положим внутрь файл

1. мы можем сделать это целой командой или зайти внутрь. Сделаем командой



1. проверим что внутри

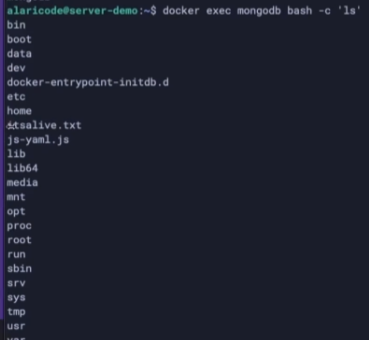


видим наш файл

1. останавливаем контейнер и снова запускаем



1. повторим команду и убедимся что вставленный ранее файл на месте



есть, файл на месте

1. также можно войти через интерактивный режим через bash и посмотреть, что внутри контейнера наш файл присутствует





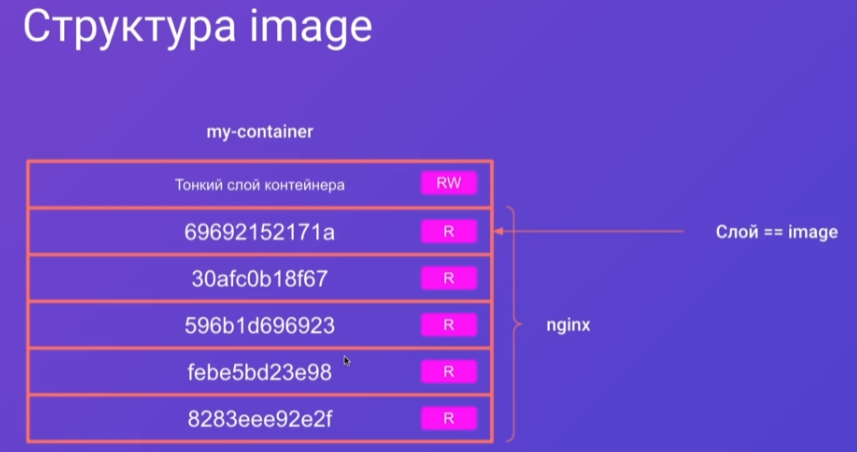
Когда мы делали docker run мы скачивали образ из registry, после этого поднимали контейнер.

Скачаем имадж nginx и посмотрим, что внутри

docker pull nginx

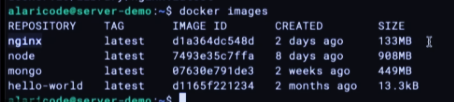
после этого образ nginx скачается к нам на машину

при этом пулится не один объект, а слои. каждый стой имеет уникальный идентификатор, благодаря этому экономится пространство на диске. Итак мы запросили имадж и докер демон скачал 6 слоев



каждый слой на самом деле тоже image. Т.е. это некоторый слепок, в котором у нас содержится информация. Все слои доступны только на чтение. После того как из имаджа создастся контейнер, добавится еще один слой, который уже будет доступен на запись. Поэтому сколько бы мы разных nginx-ов не запустили у нас имадж, на котором они будут базироваться, всегда один. это дает нам экономию пространства.

Посмотрим images, которые у нас есть



посмотрим, что у него внутри

здесь папки слоев, их 6 (это и есть скачаные слои)



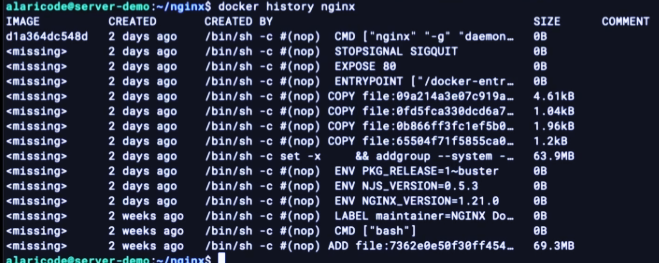
два Json-а и

В манифесте агрегируемая информация по image

во втором – более подробная информация

repositories

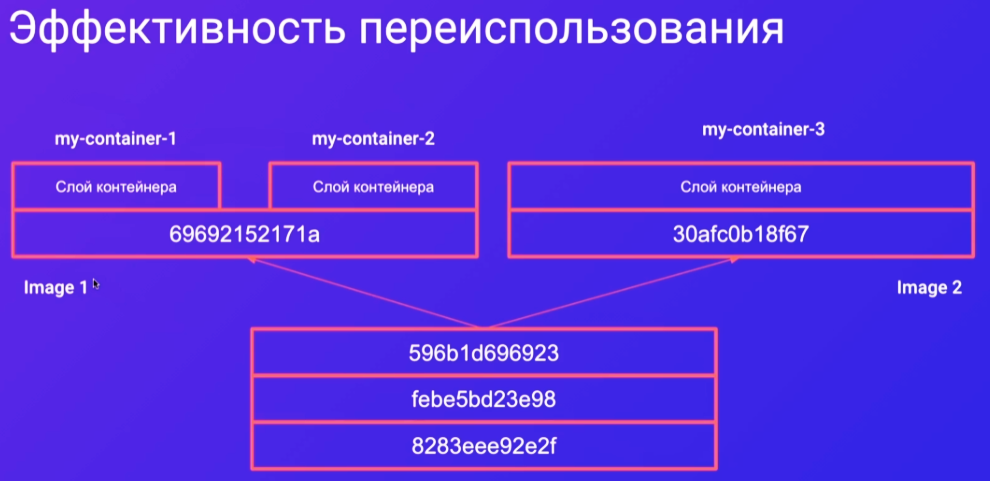
также мы можем посмотреть как был создан этот образ



и мы увидим все что происходило до момента сборки

Важно понять, что имадж состоит из слоев, слои это тоже имаджи, каждый из этих слоев мы можем проинспектировать и посмотреть, что есть внутри.

почему такая слоеная архитектура позволяет оптимизироваться – потому что позволяет переиспользовать одни и те же слои



как понять сколько занимают место контейтеры

посмотрим суммарный диск юзадж у папки var/lib/docker/overlay2

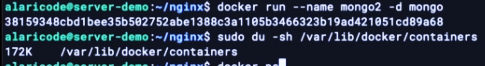


также это можно посмотреть по контейнерам



как видим контейнер почти ничего не весит и при этом он ещё переиспользуется.

запустим контейнер и посмотрим как изменится размер юзадж

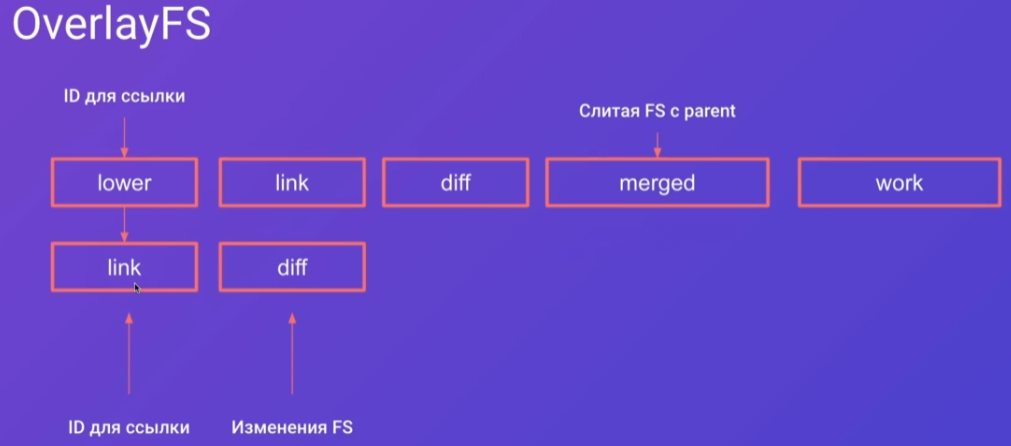


посмотрим количество запущенных контейнеров



т.о. мы видим, что два запущенных контейнера весят 172 кбайт, потому что у них есть исходные слои, на которых они и базируются

Рассмотрим Overlay файловой системы(то как она устроена/папки) и то, как она работает.



допустим, что у нас есть два слоя (самый нижний и следующий выше)

самый нижноий слой содержит id-к на который потом можем ссылаться(link) а также diff(это diff файловой системы исходной)когда мы начинаем наслаивать слои, то вторая файловая система имеет уже

lower – то что ссылается на предыдущий и говорит, сто лежит подо мной

link – id, чтобы на него мог ссылаться слой который будет еще выше

diff - diff относительно предыдущего слоя

merged – слитый второй diff с предыдущей файловой системой

work – происходят внутренние вещи для overlay

Чтобы увидеть эти папки нужно выполнить команду

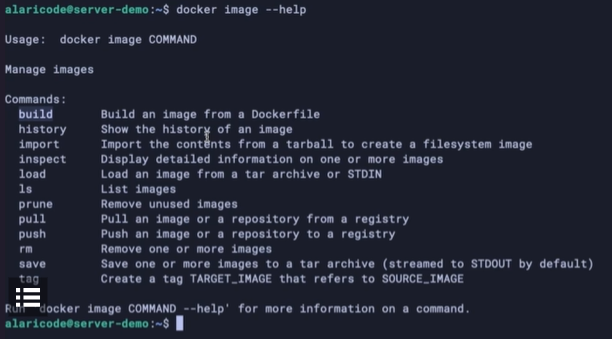


сейчас текущий драйвер, который используют в системе docker это overlay2

Итак image – это набор других слоев (images) в рамках которой мы имеем изменения нашей файловой системы и когда мы запускаем контейнер наслаивается дополнительный слой, в который мы можем что-то записывать и исполнять наши файлы, благодаря такому устройству мы имеем экономию места и удобное переиспользование слоев.



Раздерем команды которые позволяют управлять image



Рассмотрим все кроме build

**History** and **inspect** (более детально) позволяют проинспектировать наш image.

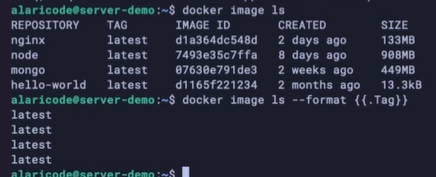
**save** – сохраняет image в архив

**import** – распаковывает из архива в image

**docker pull/docker push** тоже при работе с registry

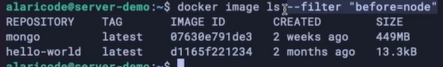
**docer image ls** – позволяет посмотреть все текущие images

можно вывести только часть информации



Кроме форматирования можно осуществить фильтрацию

Например, отобразим image которые были перед node



Удаление image, которые уже не нужны

**docker image rm** **image\_name/id**

**image** не может быть удален, если запущен контейнер основанный на этом image

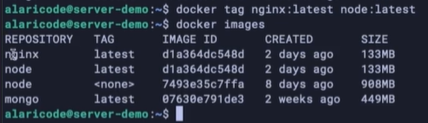
остановить – удалить контейнер – удалить image

чтобы удалить image нужно удалить все контейнеры, которые на него ссылаются

**docker images** позволяет просматривать список имеющихся image

**docker tag** – позволяет затэгать определенный image

например скажем, что nginx:latest будет теперь node:latest



Мы видим теперь новую ноду и ноду с тэгом none, потому что мы создали новый image из nginx(то есть мы сказали затэгай nginx как node:latest при этом id и size полностью совпадают)Но так как мы собрали с таким же тэгом, предыдущие все image которые имели такой же тэг теряют его и становятся <none>

То есть, если собирать несколько раз подряд image с одним и тем же тэгом, то предыдущие будут получать <none>

Чтобы удалить такие image есть команда

**docker image prune**

****