**Docker**

Образы Docker создают на основе файлов Dockerfile, описывающих всё то, что нужно для сборки образов. Кроме того, не будем забывать и о том, что контейнер — это образ Docker, вызванный к жизни.

* Images (образы) - Схемы нашего приложения, которые являются основой контейнеров. В примере выше мы использовали команду docker image pull чтобы скачать образ busybox.
* Containers (контейнеры) - Создаются на основе образа и запускают само приложение. Мы создали контейнер командой docker container run, и использовали образ busybox, скачанный ранее. Список запущенных контейнеров можно увидеть с помощью команды docker ps.
  + Docker Daemon (демон Докера) - Фоновый сервис, запущенный на хост-машине, который отвечает за создание, запуск и уничтожение Докер-контейнеров. Демон — это процесс, который запущен на операционной системе, с которой взаимодействует клиент.
* Docker Client (клиент Докера) - Утилита командной строки, которая позволяет пользователю взаимодействовать с демоном. Существуют другие формы клиента, например, [Kitematic](https://kitematic.com/), с графическим интерфейсом.
* Docker Hub - [Регистр](https://hub.docker.com/explore/) Докер-образов. Грубо говоря, архив всех доступных образов. Если нужно, то можно содержать собственный регистр и использовать его для получения образов.

Важно понимать разницу между базовыми и дочерними образами:

* Base images (базовые образы) — это образы, которые не имеют родительского образа. Обычно это образы с операционной системой, такие как ubuntu, busybox или debian.
* Child images (дочерние образы) — это образы, построенные на базовых образах и обладающие дополнительной функциональностью.

Существуют официальные и пользовательские образы, и любые из них могут быть базовыми и дочерними.

* Официальные образы — это образы, которые официально поддерживаются командой Docker. Обычно в их названии одно слово. В списке выше python, ubuntu, busybox и hello-world — базовые образы.
* Пользовательские образы — образы, созданные простыми пользователями вроде меня и вас. Они построены на базовых образах. Обычно, они называются по формату user/image-name.

## Часть 0.2 Процессы в контейнерах

Теперь, когда мы выяснили что такое docker, посмотрим в бинокль что там внутри. Нужно запомнить следующие постулаты:

1. Контейнер живет, пока живет процесс, вокруг которого рождается контейнер.
2. Внутри контейнера этот процесс имеет pid=1
3. Рядом с процессом с pid=1 можно порождать сколько угодно других процессов (в пределах возможностей ОС, естественно), но убив (рестартовав) именно процесс с pid=1, контейнер выходит. (см п.1)
4. Внутри контейнера вы увидите привычное согласно стандартам [FHS](https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard) расположение директорий. Расположение это идентично исходному дистрибутиву (с которого взят контейнер).
5. Данные, создаваемые внутри контейнера остаются в контейнере и нигде более не сохраняются (ну, еще к этому слою есть доступ из хостовой ОС). удалив контейнер — потеряете все ваши изменения. Поэтому данные в контейнерах не хранят, а выносят наружу, на хостовую ОС.

## Общие сведения о командах Docker

Вот кое-что, о чём полезно знать тем, кто хочет работать с Docker:

* Команды интерфейса командной строки Docker, используемые для управления чем-либо, начинаются с ключевого слова docker, за которым идёт пробел, затем идёт указание на то, на что именно будет направлена некая команда, потом ещё один пробел, а потом следует сама команда. Например, именно так построена такая команда: docker container stop.
* Если команда направлена на конкретный образ или контейнер, то в ней используется имя или идентификатор такого образа или контейнера.

Например, команда docker container run my\_app — это команда для создания и запуска контейнера с именем my\_app.

## Команды для управления контейнерами

Общая схема команд для управления контейнерами выглядит так:

docker container my\_command

Вот команды, которые могут быть подставлены туда, где мы использовали my\_command:

* create — создание контейнера из образа.
* start — запуск существующего контейнера.
* run — создание контейнера и его запуск.
* ls — вывод списка работающих контейнеров.
* inspect — вывод подробной информации о контейнере.
* logs — вывод логов.
* stop — остановка работающего контейнера с отправкой главному процессу контейнера сигнала SIGTERM, и, через некоторое время, SIGKILL.
* kill — остановка работающего контейнера с отправкой главному процессу контейнера сигнала SIGKILL.
* rm — удаление остановленного контейнера.

## Команды для управления образами

Для управления образами используются команды, которые выглядят так:

docker image my\_command

Вот некоторые из команд этой группы:

* build — сборка образа.
* push — отправка образа в удалённый реестр.
* ls — вывод списка образов.
* history — вывод сведений о слоях образа.
* inspect — вывод подробной информации об образе, в том числе — сведений о слоях.
* rm — удаление образа.

## Разные команды

* docker version — вывод сведений о версиях клиента и сервера Docker.
* docker login — вход в реестр Docker.
* docker system prune — удаление неиспользуемых контейнеров, сетей и образов, которым не назначено имя и тег.

Теперь рассмотрим эти команды подробнее.  
Контейнеры

### ▍Начало существования контейнера

На начальном этапе работы с контейнерами используются команды create, start и run. Они применяются, соответственно, для создания контейнера, для его запуска, и для его создания и запуска.  
  
Вот команда для создания контейнера из образа:

docker container create my\_repo/my\_image:my\_tag

В следующих примерах конструкция my\_repo/my\_image:my\_tag будет сокращена до my\_image.  
Команда create принимает множество [флагов](https://docs.docker.com/engine/reference/commandline/container_create/). Например, её можно записать в таком виде:

docker container create -a STDIN my\_image

Флаг -a представляет собой краткую форму флага --attach. Этот флаг позволяет подключить контейнер к STDIN, STDOUT или STDERR.  
  
После того, как контейнер создан, его можно запустить следующей командой:

docker container start my\_container

Обратите внимание на то, что сослаться на контейнер в команде можно либо используя его ID, либо имя.  
  
Теперь взглянем на команду, которая позволяет создать и запустить контейнер:

docker container run my\_image

Эта команда тоже способна принимать множество [аргументов](https://docs.docker.com/engine/reference/commandline/container_run/) командной строки. Рассмотрим некоторые из них на примере такой конструкции:

docker container run -i -t -p 1000:8000 --rm my\_image

Флаг -i — это сокращение для --interactive. Благодаря этому флагу поток STDIN поддерживается в открытом состоянии даже если контейнер к STDIN не подключён.  
  
Флаг -t — это сокращение для --tty. Благодаря этому флагу выделяется псевдотерминал, который соединяет используемый терминал с потоками STDIN и STDOUT контейнера.  
  
Для того чтобы получить возможность взаимодействия с контейнером через терминал нужно совместно использовать флаги -i и -t.  
  
Флаг -p представляет собой сокращение для --port. Порт — это интерфейс, благодаря которому контейнер взаимодействует с внешним миром. Конструкция 1000:8000 перенаправляет порт Docker 8000 на порт 1000 компьютера, на котором выполняется контейнер. Если в контейнере работает некое приложение, способное выводить что-то в браузер, то, для того, чтобы к нему обратиться, в нашем случае можно перейти в браузере по адресу localhost:1000.  
  
Флаг --rm автоматически удаляет контейнер после того, как его выполнение завершится.  
  
Рассмотрим ещё некоторые примеры команды run:

docker container run -it my\_image my\_command

В подобной конструкции может применяться команда sh, которая создаст сессию терминала в контейнере, с которой можно взаимодействовать через ваш терминал. При работе с образами, основанными на Alpine, лучше ориентироваться на использование sh а не bash, так как в этих образах, по умолчанию, оболочка bash не установлена. Для выхода из интерактивной сессии воспользуйтесь командой exit.  
  
Обратите внимание на то, что здесь мы скомбинировали флаги -i и -t в -it.  
  
Вот ещё один пример работы с командой run:

docker container run -d my\_image

Флаг -d — это сокращение для --detach. Эта команда запускает контейнер в фоновом режиме. Это позволяет использовать терминал, из которого запущен контейнер, для выполнения других команд во время работы контейнера.

### ▍Проверка состояния контейнера

Если у вас имеются запущенные контейнеры Docker и вы хотите узнать о том, что это за контейнеры, вам понадобится вывести их список. Сделать это можно такой командой:

docker container ls

Эта команда выводит список выполняющихся контейнеров и снабжает этот список некоторыми полезными сведениями о них. Вот ещё один пример этой команды:

docker container ls -a -s

Ключ -a этой команды — это сокращение для --all. Благодаря использованию этого ключа можно вывести сведения обо всех контейнерах, а не только о выполняющихся.  
  
Ключ -s — это сокращение для --size. Он позволяет вывести размеры контейнеров.  
  
Вот команда, которая выводит подробные сведения о контейнере:

docker container inspect my\_container

Вот команда, выводящая логи контейнера:

docker container logs my\_container

### ▍Завершение работы контейнера

Иногда работающий контейнер надо остановить. Для этого используется такая команда:

docker container stop my\_container

Она позволяет останавливать работающие контейнеры, позволяя им корректно завершить работу. У контейнера есть, по умолчанию, 10 секунд, на то, чтобы завершить работу.  
  
Если же контейнер нужно остановить быстро, не заботясь о корректном завершении его работы, можно воспользоваться такой командой:

docker container kill my\_container

Команда kill, если сравнить работающий контейнер с включенным телевизором, напоминает выключение телевизора путём отключения его от электричества. Поэтому, в большинстве ситуаций, для остановки контейнеров рекомендуется использовать команду stop.  
  
Вот команда, которая позволяет быстро остановить все работающие контейнеры:

docker container kill $(docker ps -q)

Для удаления остановленного контейнера можно воспользоваться такой командой:

docker container rm my\_container

Вот команда, которая позволяет удалить все контейнеры, которые на момент вызова этой команды не выполняются:

docker container rm $(docker ps -a -q)

Подведём итоги этого раздела. Сначала контейнер создают, потом его запускают, или комбинируют эти два шага, используя команду вида docker run my\_container. После этого запускается контейнеризированное приложение.  
  
Потом контейнер останавливают командой docker stop my\_container. Для удаления контейнера используется команда docker rm my\_container.  
  
Поговорим теперь о командах, используемых для работы с образами, с теми самыми шаблонами, из которых создают контейнеры.

## Образы

### ▍Создание образов

Вот команда, которая позволяет собирать образы Docker:

docker image build -t my\_repo/my\_image:my\_tag .

В данном случае создаётся образ с именем my\_image, при его сборке используется файл Dockerfile, находящийся по указанному пути или URL.  
  
Флаг -t — это сокращение для --tag. Он указывает Docker на то, что создаваемому образу надо назначить предоставленный в команде тег. В данном случае это my\_tag.  
  
Точка в конце команды указывает на то, что образ надо собрать с использованием файла Dockerfile, находящегося в текущей рабочей директории.  
  
После того, как образ собран, его можно отправить в удалённый реестр. Благодаря этому им смогут воспользоваться другие люди, его можно будет загрузить и запустить на другом компьютере. Предположим, вы хотите использовать [Docker Hub](https://hub.docker.com/). Если так — вам понадобится завести там учётную запись. Пользоваться этим ресурсом можно бесплатно.   
  
После того, как вы зарегистрируетесь на Docker Hub, вам нужно войти в систему. И хотя команда, которая для этого используется, напрямую к командам, предназначенным для работы с образами, не относится, её полезно будет рассмотреть именно здесь. Речь идёт о следующей команде:

docker login

Она позволяет войти в учётную запись на Docker Hub. Для входа в систему вам понадобится ввести имя пользователя и пароль.  
  
После входа в систему можно будет отправлять образы в реестр. Делается это так:

docker image push my\_repo/my\_image:my\_tag

Теперь, когда у вас наберётся несколько образов, вы можете их исследовать с помощью специальных команд.

### ▍Исследование образов

Вот команда, которая выводит список образов, выводя, в том числе, и сведения об их размере:

docker image ls

Следующая команда позволяет вывести сведения о промежуточных образах, входящих в состав образа, в частности — данные об их размерах и о том, как они были созданы:

docker image history my\_image

Вот команда, которая выводит подробные сведения об образе, в том числе — данные о слоях, из которых состоит образ:

docker image inspect my\_image

Если вы создадите очень много образов, может случиться так, что некоторые из них понадобится удалить.

### ▍Удаление образов

Вот команда, которая позволяет удалить указанный образ:

docker image rm my\_image

Если образ хранится в удалённом репозитории, он оттуда удалён не будет.  
  
Вот команда, которая позволяет удалить все локальные образы:

docker image rm $(docker images -a -q)

Пользоваться этой командой стоит с осторожностью, но надо заметить, что при её использовании образы, хранящиеся в удалённом репозитории, удалены не будут. В этом заключается одно из преимуществ хранения образов в репозиториях.  
  
Мы рассмотрели основные команды, используемые для управления контейнерами и образами. Поговорим теперь ещё о некоторых командах.

### ▍Создание тома

Создать самостоятельный том можно следующей командой:

docker volume create —-name my\_volume

### ▍Выяснение информации о томах

Для того чтобы просмотреть список томов Docker, воспользуйтесь следующей командой:

docker volume ls

Исследовать конкретный том можно так:

docker volume inspect my\_volume

### ▍Удаление тома

Удалить том можно так:

docker volume rm my\_volume

Для того чтобы удалить все тома, которые не используются контейнерами, можно прибегнуть к такой команде:

docker volume prune

Перед удалением томов Docker запросит у вас подтверждение выполнения этой операции.  
  
Если том связан с каким-либо контейнером, такой том нельзя удалить до тех пор, пока не удалён соответствующий контейнер. При этом, даже если контейнер удалён, Docker не всегда это понимает. Если это случилось — можете воспользоваться следующей командой:

docker system prune

Она предназначена для очистки ресурсов Docker. После выполнения этой команды у вас должна появиться возможность удалить тома, статус которых до этого определялся неправильно.

## Флаги --mount и --volume

Для работы с томами вам, при вызове команды docker, часто придётся пользоваться флагами. Например, для того чтобы создать том во время создания контейнера можно воспользоваться такой конструкцией:

docker container run --mount source=my\_volume, target=/container/path/for/volume my\_image

В давние времена (до 2017 года) популярен был флаг --volume. Изначально этот флаг (ещё им можно пользоваться в сокращённом виде, тогда он выглядит как -v) использовался для самостоятельных контейнеров, а флаг --mount — в среде Docker Swarm. Однако, начиная с Docker 17.06, флаг --mount можно использовать в любых сценариях.  
  
Надо отметить, что при использовании флага --mount увеличивается объём дополнительных данных, которые приходится указывать в команде, но, по нескольким причинам, лучше использовать именно этот флаг, а не --volume. Флаг --mount — это единственный механизм, который позволяет работать с сервисами или указывать параметры драйвера тома. Кроме того, работать с этим флагом проще.  
  
В существующих примерах команд, направленных на работу с данными в Docker, вы можете встретить множество примеров употребления флага -v. Пытаясь адаптировать эти команды для себя, учитывайте то, что флаги --mount и --volume используют различные форматы параметров. То есть, нельзя просто заменить -v на --mount и получить рабочую команду.  
  
Главное различие между --mount и --volume заключается в том, что при использовании флага --volume все параметры собирают вместе, в одном поле, а при использовании --mount параметры разделяются.  
  
При работе с --mount параметры представлены как пары вида ключ-значение, а именно, это выглядит как key=value. Эти пары разделяют запятыми. Вот часто используемые параметры --mount:

* type — тип монтирования. Значением для соответствующего ключа могут выступать [bind](https://docs.docker.com/storage/bind-mounts/), [volume](https://docs.docker.com/storage/volumes/) или [tmpfs](https://docs.docker.com/storage/tmpfs/). Мы тут говорим о томах, то есть — нас интересует значение volume.
* source — источник монтирования. Для именованных томов это — имя тома. Для неименованных томов этот ключ не указывают. Он может быть сокращён до src.
* destination — путь, к которому файл или папка монтируется в контейнере. Этот ключ может быть сокращён до dst или target.
* readonly — монтирует том, который предназначен [только для чтения](https://docs.docker.com/storage/volumes/" \l "use-a-read-only-volume). Использовать этот ключ необязательно, значение ему не назначают.

Вот пример использования --mount с множеством параметров:

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

The -v and --mount examples below produce the same result. You can’t run them both unless you remove the devtest container and the myvol2 volume after running the first one.

* --mount
* -v

$ docker run -d \

--name devtest \

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

nginx:latest

Use docker inspect devtest to verify that the volume was created and mounted correctly. Look for the Mounts section:

"Mounts": [

{

"Type": "volume",

"Name": "myvol2",

"Source": "/var/lib/docker/volumes/myvol2/\_data",

"Destination": "/app",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

],

**После установки, Докер автоматически создает три сети:**

**docker network ls**  посмотреть какие сети

Сеть bridge — это сеть, в которой контейнеры запущены по умолчанию.

docker network inspect bridge проверка кто в сети bridge

docker network create --driver bridge pkNetwork создание своей сети pkNetwork. создает новую сеть типа bridge

запустить наши контейнеры внутри сети с помощью флага —net

docker container run -dp 9200:9200 --net foodtrucks --name es elasticsearch

запускает контейнер в сети foodtruck, дали контейнеру название es

|  |  |
| --- | --- |
|  | docker container run -i -t --container rm ubuntu /bin/bash |

-t присваивает псевдо-tty или терминал внутри нового контейнера.

-i позволяет создавать интерактивное соединение, захватывая стандартный вход (STDIN) контейнера.

—rm требуется для автоматического удаления контейнера при выходе из процесса. По умолчанию контейнеры не удаляются.

Чтобы отсоединить tty, не выходя из оболочки, используйте escape-последовательность Ctrl + p + Ctrl + q.

Если вы хотите, чтобы контейнер работал после окончания сеанса, вам необходимо его «демонизировать»:

docker container run --name daemon -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done"

—name daemon назначает имя новому контейнеру. Если вы его не укажете, имя сгенерируется и назначится автоматически.

-d запускает контейнер в фоновом режиме («демонизирует» его).

$ docker container run -d -P --name static-site prakhar1989/static-site

Флаг -d открепит (detach) терминал, флаг -P сделает все открытые порты публичными и случайными, и, наконец, флаг --name это имя, которое мы хотим дать контейнеру.

Теперь можно увидеть порты с помощью команды docker container port [CONTAINER].

$ docker container port static-site

80/tcp -> 0.0.0.0:32769

443/tcp -> 0.0.0.0:32768

docker container stop 2567dc2cd836 останавливает конетйне с типом id

docker ps -a

docker ps – команда для перечисления контейнеров.

-a показывает все контейнеры (без -a ps покажет только запущенные контейнеры).

|  |  |
| --- | --- |
| 1 | docker container logs -f daemon |

docker container logs получают журналы контейнера.

* -f следит за выходом журнала.

|  |  |
| --- | --- |
|  | docker container stop daemon |

Остановка контейнера

|  |  |
| --- | --- |
|  | docker container start daemon |

Старт контейнера

docker container rm daemon

ручное удаление контейнера, предварительно нужно остановаить

docker container rm -f $(docker ps -aq) автоматиеское удаление всех контейнеров

docker container rm – команда удаления контейнера.

* -f (для container rm) должен остановить контейнер, если он работает (принудительное удаление).
* -q (для ps) – это вывод только идентификаторов контейнера.

$ docker container rm $(docker ps -a -q -f status=exited)

удаляет все контейнеры, у которых статус exited.

docker search elasticsearch поиск контейнера

docker container run -d -p 9200:9200 elasticsearch:6.5.4 tail -f /dev/null

docker image pull ubuntu:14.04 скачать ubuntu

## Файлы Dockerfile

https://kapeli.com/cheat\_sheets/Dockerfile.docset/Contents/Resources/Documents/index

FROM ubuntu:18.04

COPY . /app

Слои в итоговом образе создают только инструкции FROM, RUN, COPY, и ADD. Другие инструкции что-то настраивают, описывают метаданные, или сообщают Docker о том, что во время выполнения контейнера нужно что-то сделать, например — открыть какой-то порт или выполнить какую-то команду.

## Дюжина инструкций Dockerfile

1. FROM — задаёт базовый (родительский) образ.
2. LABEL — описывает метаданные. Например — сведения о том, кто создал и поддерживает образ.
3. ENV — устанавливает постоянные переменные среды.
4. RUN — выполняет команду и создаёт слой образа. Используется для установки в контейнер пакетов.
5. COPY — копирует в контейнер файлы и папки.
6. ADD — копирует файлы и папки в контейнер, может распаковывать локальные .tar-файлы.
7. CMD — описывает команду с аргументами, которую нужно выполнить когда контейнер будет запущен. Аргументы могут быть переопределены при запуске контейнера. В файле может присутствовать лишь одна инструкция CMD.
8. WORKDIR — задаёт рабочую директорию для следующей инструкции.
9. ARG — задаёт переменные для передачи Docker во время сборки образа.
10. ENTRYPOINT — предоставляет команду с аргументами для вызова во время выполнения контейнера. Аргументы не переопределяются.
11. EXPOSE — указывает на необходимость открыть порт.
12. VOLUME — создаёт точку монтирования для работы с постоянным хранилищем.

FROM python:3.7.2-alpine3.8

LABEL maintainer="jeffmshale@gmail.com"

ENV ADMIN="jeff"

RUN apk update && apk upgrade && apk add bash

COPY . ./app

ADD https://raw.githubusercontent.com/discdiver/pachy-vid/master/sample\_vids/vid1.mp4 \

/my\_app\_directory

RUN ["mkdir", "/a\_directory"]

CMD ["python", "./my\_script.py"]

Инструкция [COPY](https://docs.docker.com/engine/reference/builder/" \l "copy) представлена в нашем файле так: COPY . ./app. Она сообщает Docker о том, что нужно взять файлы и папки из локального контекста сборки и добавить их в текущую рабочую директорию образа. Если целевая директория не существует, эта инструкция её создаст.

Инструкция [ADD](https://docs.docker.com/engine/reference/builder/" \l "add) позволяет решать те же задачи, что и COPY, но с ней связана ещё пара вариантов использования. Так, с помощью этой инструкции можно добавлять в контейнер файлы, загруженные из удалённых источников, а также распаковывать локальные .tar-файлы.  
  
В этом примере инструкция ADD была использована для копирования файла, доступного по URL, в директорию контейнера my\_app\_directory. Надо отметить, однако, что [документация Docker](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/) не рекомендует использование подобных файлов, полученных по URL, так как удалить их нельзя, и так как они увеличивают размер образа.

Инструкция [CMD](https://docs.docker.com/engine/reference/builder/" \l "cmd) предоставляет Docker команду, которую нужно выполнить при запуске контейнера. Результаты выполнения этой команды не добавляются в образ во время его сборки. В нашем примере с помощью этой команды запускается скрипт my\_script.py во время выполнения контейнера.  
  
Вот ещё кое-что, что нужно знать об инструкции CMD:

* В одном файле Dockerfile может присутствовать лишь одна инструкция CMD. Если в файле есть несколько таких инструкций, система проигнорирует все кроме последней.
* Инструкция CMD может иметь exec-форму. Если в эту инструкцию не входит упоминание исполняемого файла, тогда в файле должна присутствовать инструкция ENTRYPOINT. В таком случае обе эти инструкции должны быть представлены в формате JSON.
* Аргументы командной строки, передаваемые docker container run, переопределяют аргументы, предоставленные инструкции CMD в Dockerfile.

FROM python:3.7.2-alpine3.8

LABEL maintainer="jeffmshale@gmail.com"

# Устанавливаем зависимости

RUN apk add --update git

# Задаём текущую рабочую директорию

WORKDIR /usr/src/my\_app\_directory

# Копируем код из локального контекста в рабочую директорию образа

COPY . .

# Задаём значение по умолчанию для переменной

ARG my\_var=my\_default\_value

# Настраиваем команду, которая должна быть запущена в контейнере во время его выполнения

ENTRYPOINT ["python", "./app/my\_script.py", "my\_var"]

# Открываем порты

EXPOSE 8000

# Создаём том для хранения данных

VOLUME /my\_volume

Инструкция [WORKDIR](https://docs.docker.com/v17.09/engine/reference/builder/" \l "workdir) позволяет изменить рабочую директорию контейнера. С этой директорией работают инструкции COPY, ADD, RUN, CMD и ENTRYPOINT, идущие за WORKDIR. Вот некоторые особенности, касающиеся этой инструкции:

Инструкция [ENTRYPOINT](https://docs.docker.com/engine/reference/builder/" \l "entrypoint) позволяет задавать команду с аргументами, которая должна выполняться при запуске контейнера. Она похожа на команду CMD, но параметры, задаваемые в ENTRYPOINT, не перезаписываются в том случае, если контейнер запускают с параметрами командной строки.  
  
Вместо этого аргументы командной строки, передаваемые в конструкции вида docker container run my\_image\_name, добавляются к аргументам, задаваемым инструкцией ENTRYPOINT. Например, после выполнения команды вида docker container run my\_image bash аргумент bash добавится в конец списка аргументов, заданных с помощью ENTRYPOINT.

В документации к Docker есть несколько рекомендаций, касающихся того, какую инструкцию, CMD или ENTRYPOINT, стоит выбрать в качестве инструмента для выполнения команд при запуске контейнера:

* Если при каждом запуске контейнера нужно выполнять одну и ту же команду — используйте ENTRYPOINT.
* Если контейнер будет использоваться в роли приложения — используйте ENTRYPOINT.
* Если вы знаете, что при запуске контейнера вам понадобится передавать ему аргументы, которые могут перезаписывать аргументы, указанные в Dockerfile, используйте CMD.

В нашем примере использование инструкции ENTRYPOINT ["python", "my\_script.py", "my\_var"] приводит к тому, что контейнер, при запуске, запускает Python-скрипт my\_script.py с аргументом my\_var.

Документация Docker рекомендует использовать exec-форму ENTRYPOINT: ENTRYPOINT ["executable", "param1", "param2"].

**КЭШ**

Вот несколько советов, касающихся эффективного использования кэша Docker:

* Кэширование можно отключить, передав ключ --no-cache=True команде docker image build.
* Если вы собираетесь вносить изменения в инструкции Dockerfile, тогда каждый слой, созданный инструкциями, идущими после изменённых, будет достаточно часто собираться повторно, без использования кэша. Для того чтобы воспользоваться преимуществами кэширования, помещайте инструкции, вероятность изменения которых высока, как можно ближе к концу Dockerfile.
* Объединяйте команды RUN apt-get update и apt-get install в цепочки для того, чтобы исключить проблемы, связанные с неправильным использованием кэша.
* Если вы используете менеджеры пакетов, наподобие pip, с файлом requirements.txt, тогда придерживайтесь нижеприведённой схемы работы для того, чтобы исключить использование устаревших промежуточных образов из кэша, содержащих набор пакетов, перечисленных в старой версии файла requirements.txt. Вот как это выглядит:

COPY requirements.txt /tmp/

RUN pip install -r /tmp/requirements.txt

COPY . /tmp/

### Файл .dockerignore

О файлах .dockerignore нужно знать абсолютно всем, кто хочет освоить Docker. Эти файлы похожи на файлы .gitignore. Они содержат список файлов и папок, в виде имён или шаблонов, которые Docker должен игнорировать в ходе сборки образа.  
  
Этот файл размещают там же, где находится файл Dockerfile, и всё остальное, входящее в контекст сборки образа.

Слои независимы друг от друга. это значит что запущенная в процессе сборки какая-нибудь служба внутри контейнера существует только в пределах своего слоя. Яркий пример — попытка залить базу в mysql. Как это обычно происходит? нужно запустить mysql-сервер и следующей командой залить базу. Здесь это так не сработает. Создастся слой, который сохранит результаты запуска mysql (логи, итп) и потом mysql просто завершится. в следующем слою (при выполнении команды заливки базы) мускуль уже не будет запущен и будет ошибка. Решение этой проблемы — всего-навсего объединять команды через &&.

## Рекомендации по уменьшению размеров образов и ускорению процесса их сборки

1. Используйте всегда, когда это возможно, официальные образы в качестве базовых образов. Официальные образы регулярно обновляются, они безопаснее неофициальных образов.
2. Для того чтобы собирать как можно более компактные образы, пользуйтесь базовыми образами, основанными на Alpine Linux.
3. Если вы пользуетесь apt, комбинируйте в одной инструкции RUN команды apt-get update и apt-get install. Кроме того, объединяйте в одну инструкцию команды установки пакетов. Перечисляйте пакеты в алфавитном порядке на нескольких строках, разделяя список символами \. Например, это может выглядеть так:

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

   package-one \

   package-two \

   package-three

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

Этот метод позволяет сократить число слоёв, которые должны быть добавлены в образ, и помогает поддерживать код файла в приличном виде.

1. Включайте конструкцию вида && rm -rf /var/lib/apt/lists/\* в конец инструкции RUN, используемой для установки пакетов. Это позволит очистить кэш apt и приведёт к тому, что он не будет сохраняться в слое, сформированном командой RUN. Подробности об этом можно почитать в [документации](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/).
2. Разумно пользуйтесь возможностями кэширования, размещая в Dockerfile команды, вероятность изменения которых высока, ближе к концу файла.
3. Пользуйтесь файлом .dockerignore.
4. Взгляните на dive — отличный инструмент для исследования образов Docker, который помогает в деле уменьшения их размеров.
5. Не устанавливайте в образы пакеты, без которых можно обойтись.

### Dive

установка

wget https://github.com/wagoodman/dive/releases/download/v0.0.8/dive\_0.0.8\_linux\_amd64.deb

анализ

sudo dive ea4c82dcd15a

Список сочетаний клавиш для использования «Dive».

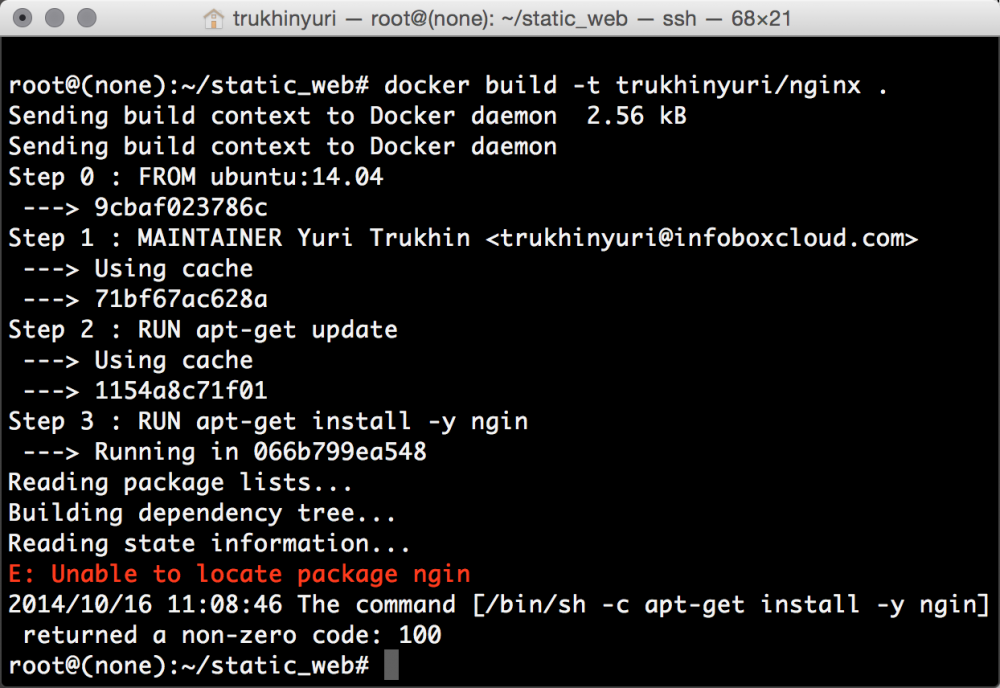
* Ctrl + Пробел — Переключение между левой и правой панелями,
* Пробел — развернуть / свернуть дерево каталогов,
* Ctrl + A — Показать / скрыть добавленные файлы,
* Ctrl + R — Показать / скрыть удаленные файлы,
* Ctrl + M — Показать / скрыть измененные файлы,
* Ctrl + U — Показать / скрыть немодифицированные файлы,
* Ctrl + L — Показать изменения слоя,
* Ctrl + A — Показать изменения слоя,
* Ctrl + / — Фильтровать файлы,
* Ctrl + C — выход.

## Mapping Docker Solo to Management Commands

Let’s look at how the existing top-level commands match to the management commands:

|  |  |  |
| --- | --- | --- |
| 1.12 | 1.13 | Purpose |
| attach | container attach | Attach to a running container |
| build | image build | Build an image from a Dockerfile |
| commit | container commit | Create a new image from a container’s changes |
| cp | container cp | Copy files/folders between a container and the local filesystem |
| create | container create | Create a new container |
| diff | container diff | Inspect changes on a container’s filesystem |
| events | system events | Get real time events from the server |
| exec | container exec | Run a command in a running container |
| export | container export | Export a container’s filesystem as a tar archive |
| history | image history | Show the history of an image |
| images | image ls | List images |
| import | image import | Import the contents from a tarball to create a filesystem image |
| info | system info | Display system-wide information |
| inspect | container inspect | Return low-level information on a container, image or task |
| kill | container kill | Kill one or more running containers |
| load | image load | Load an image from a tar archive or STDIN |
| login | login | Log in to a Docker registry. |
| logout | logout | Log out from a Docker registry. |
| logs | container logs | Fetch the logs of a container |
| network | network | Manage Docker networks |
| node | node | Manage Docker Swarm nodes |
| pause | container pause | Pause all processes within one or more containers |
| port | container port | List port mappings or a specific mapping for the container |
| ps | container ls | List containers |
| pull | image pull | Pull an image or a repository from a registry |
| push | image push | Push an image or a repository to a registry |
| rename | container rename | Rename a container |
| restart | container restart | Restart a container |
| rm | container rm | Remove one or more containers |
| rmi | image rm | Remove one or more images |
| run | container run | Run a command in a new container |
| save | image save | Save one or more images to a tar archive (streamed to STDOUT by default) |
| search | search | Search the Docker Hub for images |
| service | service | Manage Docker services |
| start | container start | Start one or more stopped containers |
| stats | container stats | Display a live stream of container(s) resource usage statistics |
| stop | container stop | Stop one or more running containers |
| swarm | swarm | Manage Docker Swarm |
| tag | image tag | Tag an image into a repository |
| top | container top | Display the running processes of a container |
| unpause | container unpause | Unpause all processes within one or more containers |
| update | container update | Update configuration of one or more containers |
| version | version | Show the Docker version information |
| volume | volume | Manage Docker volumes |
| wait | container wait | Block until a container stops, then print its exit code |

##### **Что произойдет, если инструкция не исполнится?**

Давайте переименуем в Dockerfile nginx в ngin и посмотрим.  
  
  
  
Мы можем создать контейнер из предпоследнего шага с ID образа 066b799ea548   
**docker run -i -t 066b799ea548 /bin/bash**  
и отладить исполнение.  
  
По-умолчанию Docker кеширует каждый шаг и формируя кеш сборок. Чтобы отключить кеш, например для использования последнего apt-get update, используйте флаг --no-cache.

docker build --no-cache -t trukhinyuri/nginx

Практика запуск пары селениум тестов

docker run -d -P --name selenium-hub -e GRID\_TIMEOUT=10000 selenium/hub

docker run -d --link selenium-hub:hub selenium/node-firefox

https://github.com/archick12/simple\_selenium\_for\_docker\_webinar