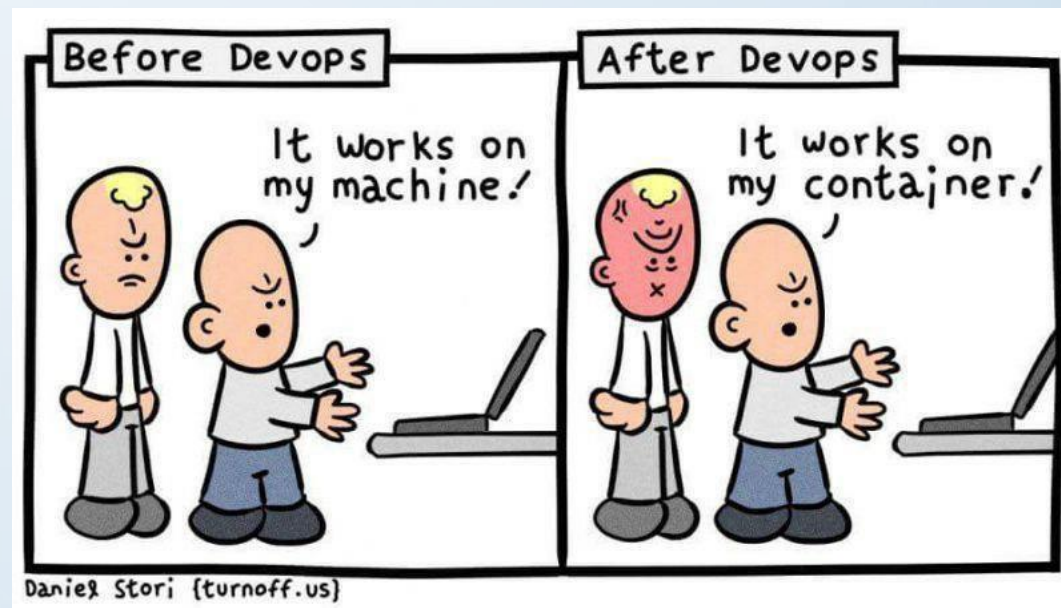




Docker for users

Лекция №2

Контейнеры...



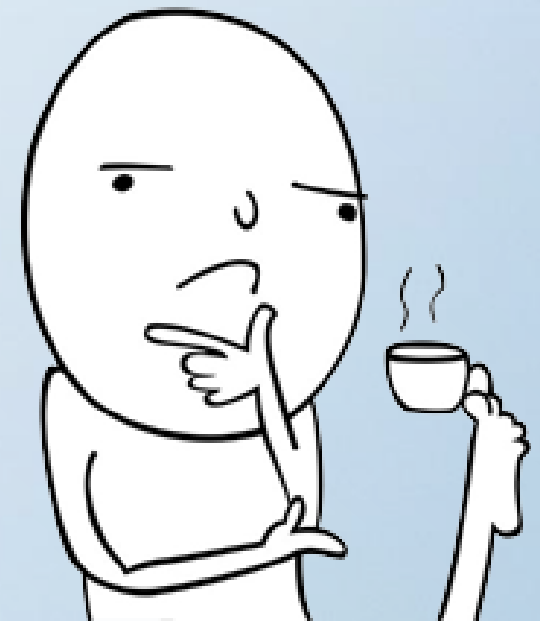
Еще раз о namespaces

- User Namespaces - позволяет связать процесс с другим набором user id
- Mount namespaces - позволяет управлять точками монтирования
- PID Namespaces - изолирует ID процессов в системе
- Network namespaces - изолирует ресурсы, связанные с сетью
- Unix Timesharing System NS – позволяет переопределять время и имя хоста
- cgroup namespace – изоляция ресурсов

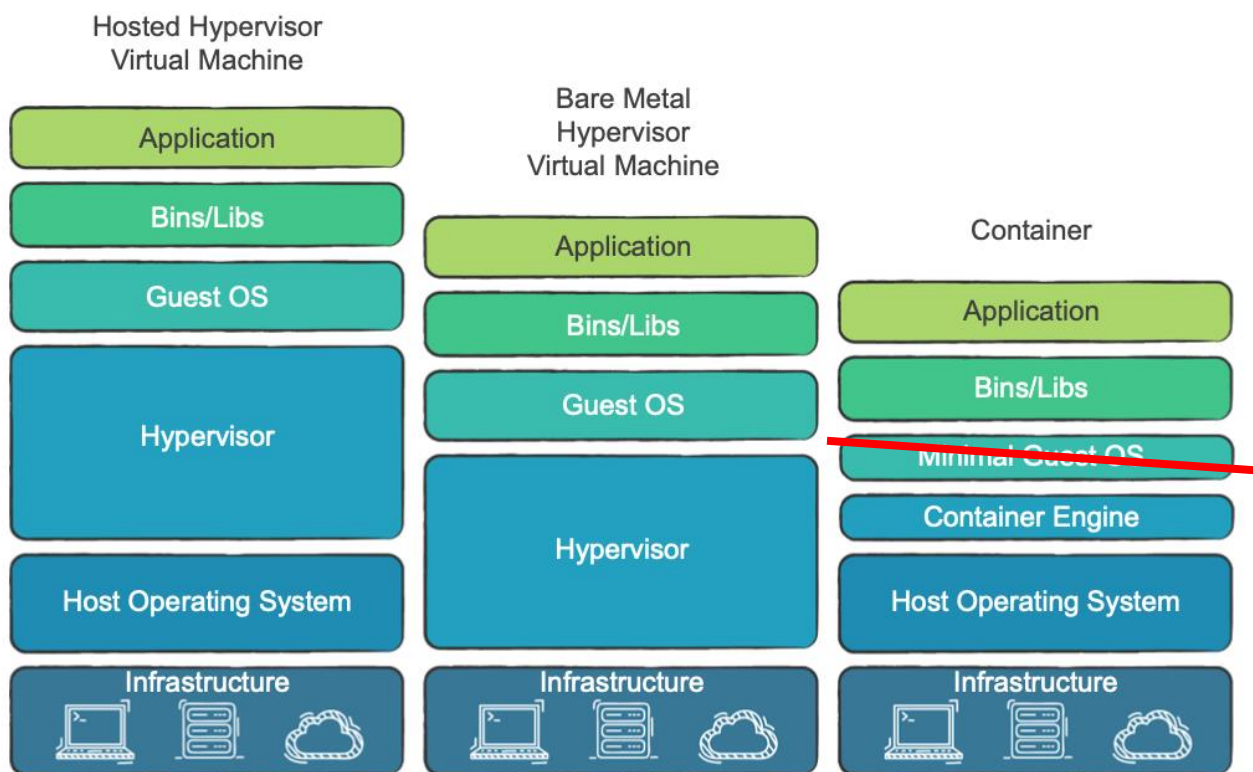
А что если...

...взять файлы, смонтировать их с помощью `mount ns` и запустить в собственном `pid`, `user`, `uts` и `cgroups ns`?

Поздравляю! Мы изобрели докер 😊
(и много других систем контейнеризации)



Связь с виртуализацией



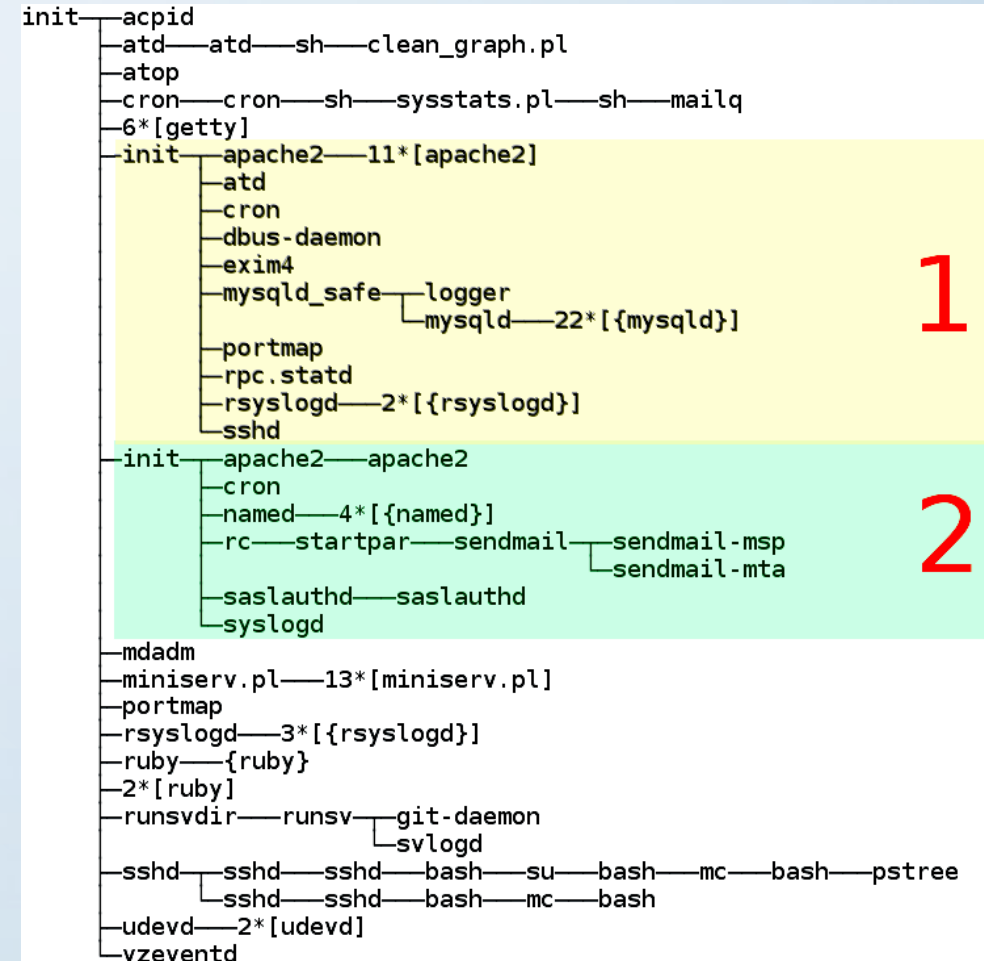
Эволюция контейнеризации

- o Unix, 1971
- o **chroot**, 1982
- o Linux, 17 сентября 1991
- o **FreeBSD Jail**, 2003, еще нет контроля ресурсов
- o **OpenVZ**, 2005, компания Parallels, зарождение cgroups
- o **CGroups**, 2006, ранее “*process containers*”, компания Google
- o **LXC**, 2008, позже **LXC/LXD** стали поддерживать Canonical
- o **Docker**, 20 марта 2013, PyCon, компания dotCloud

OpenVZ

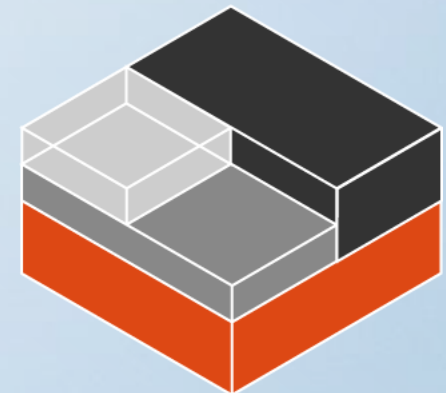


- Создало концепцию VPS и VE (virtual env)
- Состоит из модифицированного ядра Linux и пользовательских утилит
- Является базовой платформой для Virtuozzo — проприетарного продукта Parallels, Inc
- Есть checkpoint и миграции



LXC/LXD

- LXC – *Linux Containers* - предшественник docker
- LXD – надстройка над LXC, демон для управления
- Содержит в себе систему инициализации, и пр.
- Если нужно запихнуть legacy-решение в контейнер
- Если нужно запустить что-то очень не типичное
- Ныне даже более гибкий, чем Docker



Docker

- ПО для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации
- Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер
- Первая версия основывалась на LXC и Aufs
- С 2015 года начал использовать собственную библиотеку, абстрагирующую виртуализационные возможности ядра Linux — libcontainer



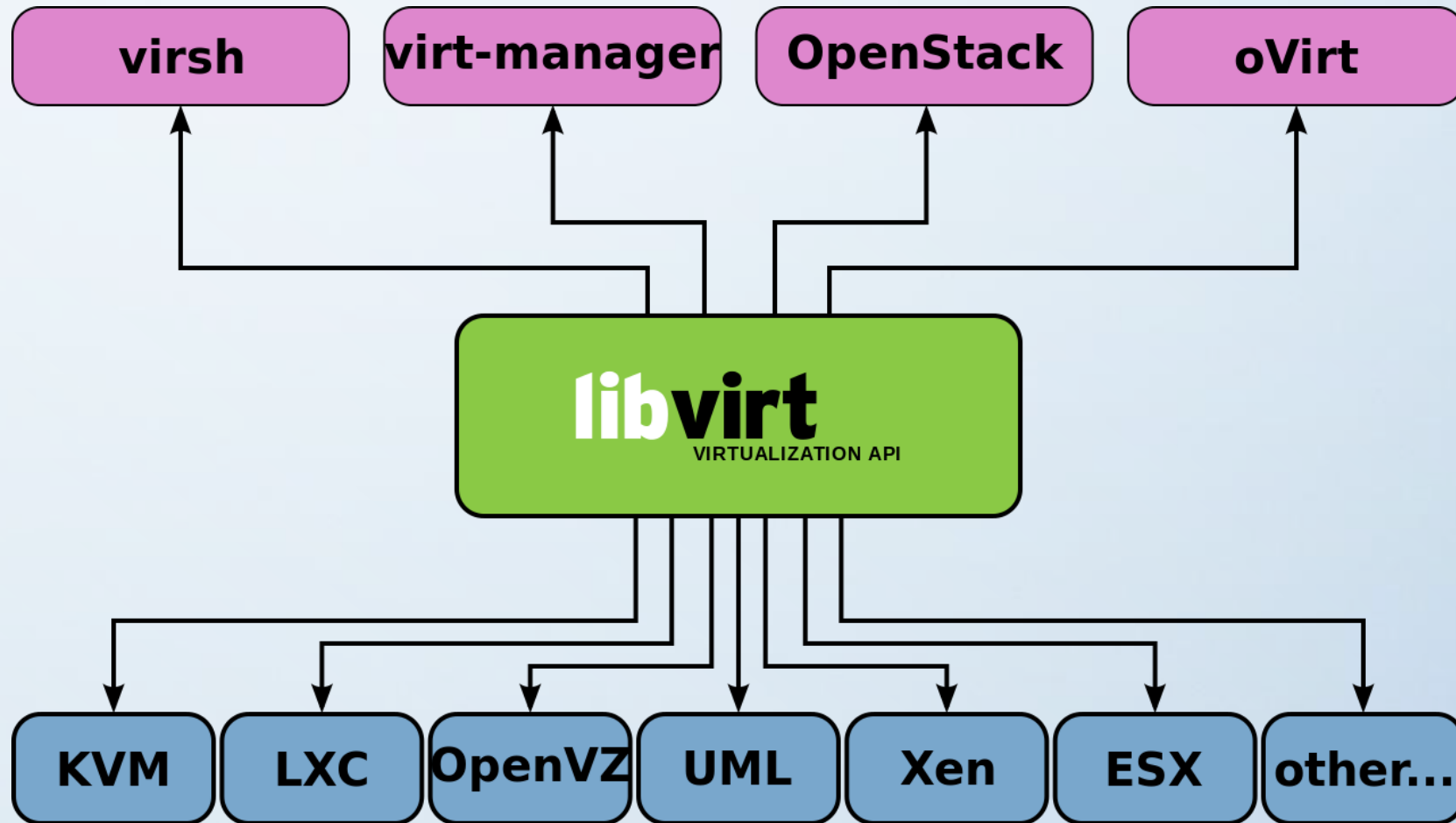
У вас шизофрения, если вы...

- ставите внутрь docker-контейнера более одного приложения или запустить несколько **независимых** процессов
- ставите внутрь docker-контейнера systemd или init
- меняете схему сборки логов внутри docker-контейнера с использованием journald или rsyslog
- накапливаете изменения внутри docker-контейнера



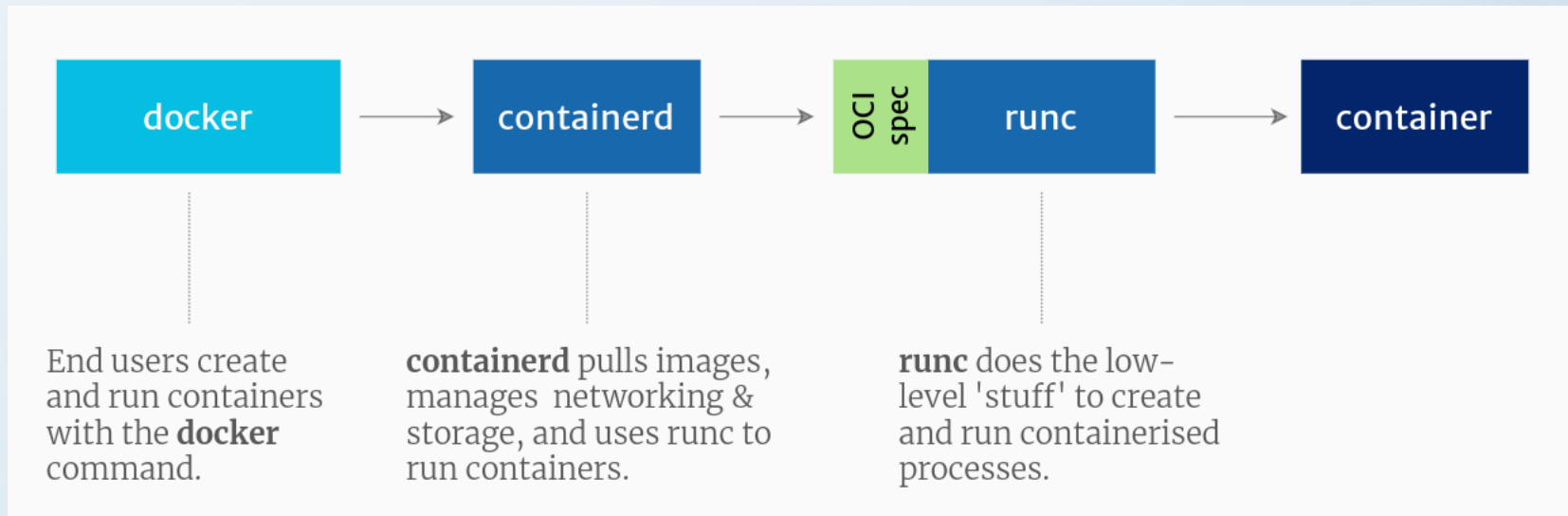
Libvirt

набор инструментов для управления виртуализацией



OCI - Open Container Initiative

- [Open Container Initiative](#) (OCI) – это проект Linux Foundation, основанный в 2015 году компанией Docker, Inc, целью которого является разработка стандартов контейнеризации
- Не путать с **Container Runtime Interface (CRI)**, определяет API между Kubernetes и Container Runtime (средой выполнения контейнеров)

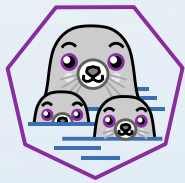


Docker? containerd? runc?

- Хорошая [статья](#) по теме
- Из Docker выделили отдельный проект containerd – демон, который управляет контейнерами, образами, и т.д.
- Docker теперь является консольной утилитой и GUI к containerd
- containerd работает со средой исполнения контейнеров, эталонная реализация - runc

Podman, Buildah, CRI-O

- *Podman* — непосредственное взаимодействие с контейнерами и хранилищем образов через процесс runC
- *Buildah* — сборка и загрузка в реестр образов
- *CRI-O* — исполняемая среда для систем оркестрации контейнеров. Т.е. по-сути замена Docker для Kubernetes



podman



buildah



cri-o

Терминология Docker

- Образ (image) – шаблон для создания контейнеров
- Тег образа (tag) – идентификатор, описывающий состояние образа
- Контейнер (container) – запущенный инстанс докер образа
- Том (volume) – персистентное место для хранения данных в контейнере
- Реестр/репозиторий образов (registry) – облачное хранилище образов

Docker образ (препарируем busybox)

Образ – это архив с файлами. Архив состоит из:

- файла с метаданными manifest.json
- Конфига что и как запускать (длинный хеш.json)
- И слоев (папочка с хешом), слой содержит все измененные файлы. Слой может быть много, первый слой содержит все базовые файлы операционной системы

```
.├── c98db043bed913c5a7b59534cbf8d976122f98b75cb00baabf8af888041e4f9d.json
├── e9589b7a23ce99365f3b7a2823b7510460a1dbe565b49f9147aefdbfb4457742
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── manifest.json
└── repositories
```

```
layer
├── bin
├── dev
├── etc
├── home
├── root
├── tmp
├── usr
└── var
```

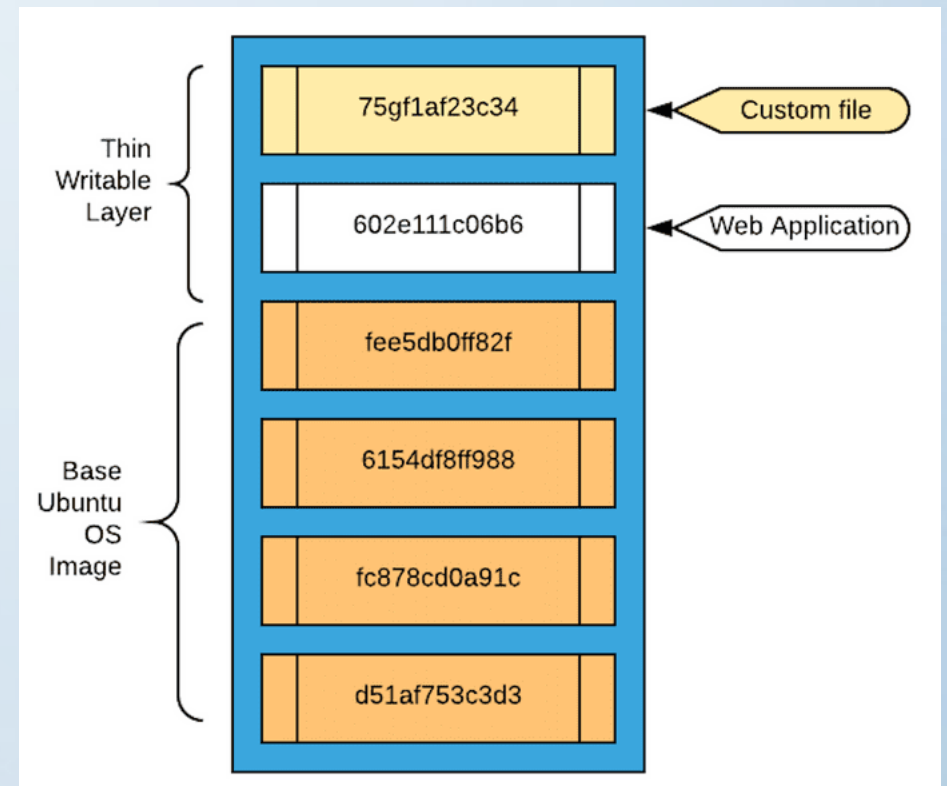

OverlayFS

- **Union mount** — это тип файловой системы, которая создает иллюзию слияния содержимого нескольких каталогов в один без изменения исходных (физических) данных в оригинальных источниках
- **OverlayFS** — включена в ядро Linux с версии 3.18 (26 октября 2014 года), файловая система, использующая дефолтный драйвер Docker **overlay2**
- Благодаря объединенной файловой системе, в Docker нужно создать только один слой поверх образа, а остальная его часть может использоваться всеми контейнерами

Смотрим отличия от образа

- Как только мы внесем изменения в работающий контейнер – мы добавим новый слой изменений к базовому образу, с которого его запустили
- Посмотрим список изменений:
`docker diff <container_id_or_name>`

```
student@debian:~$ docker diff checkintegriy
A /data
A /data/output
C /root
A /root/.bash_history
C /.dockerenv
student@debian:~$
```



Docker образ и registry

Образ можно и нужно качать из интернета. Для скачивания используется команда **docker pull <имя образа>**

Обычно образ имеет название в следующем формате:
<registry>/<имя>:<tag>

Например: *quay.io/coreos/etcd:v3.5.4*

Но можно сделать и **docker pull nginx**, это тоже самое, что **docker pull registry.hub.docker.com/library/nginx:latest**

hub.docker.com - это репозиторий по умолчанию для всех образов

Docker push

- Свой образ можно загрузить обратно в Интернет
- **docker push <image_name_or_id>**
- Предварительно образ можно перетегать с помощью:
docker image tag <old> <new>
- Создать свой образ можно либо сборкой, либо прямо из работающего контейнера:
docker container commit <container_id> <new_image_name>
- Куда?
hub.docker.com, local docker registry, gitlab-registry, nexus, ...

Запуск контейнера из образа

`docker image ls`

`docker run -it busybox sh`

- **-it** - флаги включающие перенаправление stdin и псевдо tty, позволяют работать с контейнером в интерактивном режиме
- **busybox** – имя образа
- **sh** – команда для запуска в контейнере

```
[/ # hostname
6a12f40a17d6
[/ # ps -a
PID    USER    TIME    COMMAND
   1   root      0:00    sh
   8   root      0:00    ps -a
[/ # ls -la /
total 44
drwxr-xr-x  1 root    root      4096 Sep 11 19:56 .
drwxr-xr-x  1 root    root      4096 Sep 11 19:56 ..
-rwxr-xr-x  1 root    root         0 Sep 11 19:56 .dockerenv
drwxr-xr-x  2 root    root     12288 Sep  1 16:42 bin
drwxr-xr-x  5 root    root      340 Sep 11 19:56 dev
drwxr-xr-x  1 root    root      4096 Sep 11 19:56 etc
drwxr-xr-x  2 nobody  nobody     4096 Sep  1 16:42 home
dr-xr-xr-x 179 root    root         0 Sep 11 19:56 proc
drwx----- 1 root    root      4096 Sep 11 19:56 root
dr-xr-xr-x 13 root    root         0 Sep 11 19:56 sys
drwxrwxrwt  2 root    root      4096 Sep  1 16:42 tmp
drwxr-xr-x  3 root    root      4096 Sep  1 16:42 usr
drwxr-xr-x  4 root    root      4096 Sep  1 16:42 var
[/ #
```

Основные команды

- **docker ps** – список запущенный контейнеров
- **docker attach <id или имя контейнера>** - подключиться к контейнеру в интерактивном режиме
- **docker exec <id или имя контейнера> <команда>** - выполнить команду в рабочем контейнере
- **docker stop <id или имя контейнера>** - остановить рабочий контейнер
- **docker kill <id или имя контейнера>** - принудительно остановить контейнер
- **docker start <id или имя контейнера>** - повторно запустить остановленный контейнер
- **docker rm <id или имя контейнера>** - удалить остановленный контейнер

```
[sh-3.2$ docker run -d --name=hello busybox sleep 600
7c6f587447fd12f0b061657fecb4de74b0a6384c029011b99d0c3d41a3262677
[sh-3.2$ docker stop hello
hello
[sh-3.2$ docker start hello
hello
[sh-3.2$ docker rm -f hello
hello
```

Docker и данные

По умолчанию данные в контейнере не сохраняются между запусками!

```
docker run -it -e \
"MYSQL_ALLOW_EMPTY_ROOT_PASSWORD=1" mysql:5.7
```

Чтобы это исправить используются volume

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql      |
| performance_schema |
| sys       |
+-----+
4 rows in set (0.00 sec)

mysql> create database test;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql      |
| performance_schema |
| sys       |
| test      |
+-----+
5 rows in set (0.00 sec)

mysql> sh-3.2$
sh-3.2$ docker exec -it 7d8beed2fdd4 mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.37 MySQL Community Server (GPL)

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql      |
| performance_schema |
| sys       |
+-----+
4 rows in set (0.00 sec)
```


Типы volume

При запуске контейнера, подмонтировать volume можно с помощью опции **-v**

Существуют несколько типов volume:

- bind – пробрасывает в контейнер директорию с хоста
- volume для докера – хранилище внутри докера
- tmpfs – создает том в оперативной памяти, не сохраняет данные, но удобно для временных данных с большим iops

Bind volume

Bind позволяет пробросить директорию на хосте в контейнер

Указываем `-v <путь на хосте>:<путь в контейнере>`

Да, так можно пробросить целый device, docker-сокеты или даже всю иерархию фс хоста. **Да, так делать нельзя.**

```
[sh-3.2$  
[sh-3.2$ ls  
[sh-3.2$ docker run --rm busybox touch /data/hello_bind  
touch: /data/hello_bind: No such file or directory  
[sh-3.2$ ls  
[sh-3.2$ docker run --rm -v $(pwd):/data busybox touch /data/hello_bind  
[sh-3.2$ ls  
hello_bind  
sh-3.2$
```

Стандартный docker volume

Докер позволяет создавать именованное хранилище данных с помощью команды

docker volume create <имя volume>

В примере ниже, **docker volume create hello**

```
sh-3.2$ docker run --rm -it -v hello:/data busybox sh
[/ # echo "hello volume!" > /data/message
[/ #
sh-3.2$ docker run --rm -it -v hello:/data busybox cat /data/message
hello volume!
sh-3.2$
```

Docker и сеть

Разумеется, в docker можно запускать сетевые приложения, такие как веб сервера. Например, **docker run --rm nginx**

Приложения будут доступны внутри докер сети, но чтобы получить к ним доступ с компьютера, необходимо пробросить порт. Это делается с помощью опции

-p <порт на хосте куда пробросить>:<какой порт из контейнера пробросить>

Например, **docker run --rm -p 8080:80 nginx**

Кроме того, для встроенного “service discovery” в docker существует **локальный DNS** (127.0.0.11:53)

Для параметра **name** контейнера автоматически настраивается преобразование имени в адрес контейнера внутри виртуальной сети

```
sh-3.2$ curl -I localhost:8080
HTTP/1.1 200 OK
Server: nginx/1.23.1
Date: Sun, 11 Sep 2022 20:57:01 GMT
Content-Type: text/html
Content-Length: 615
Last-Modified: Tue, 19 Jul 2022 14:05:27 GMT
Connection: keep-alive
ETag: "62d6ba27-267"
Accept-Ranges: bytes
```

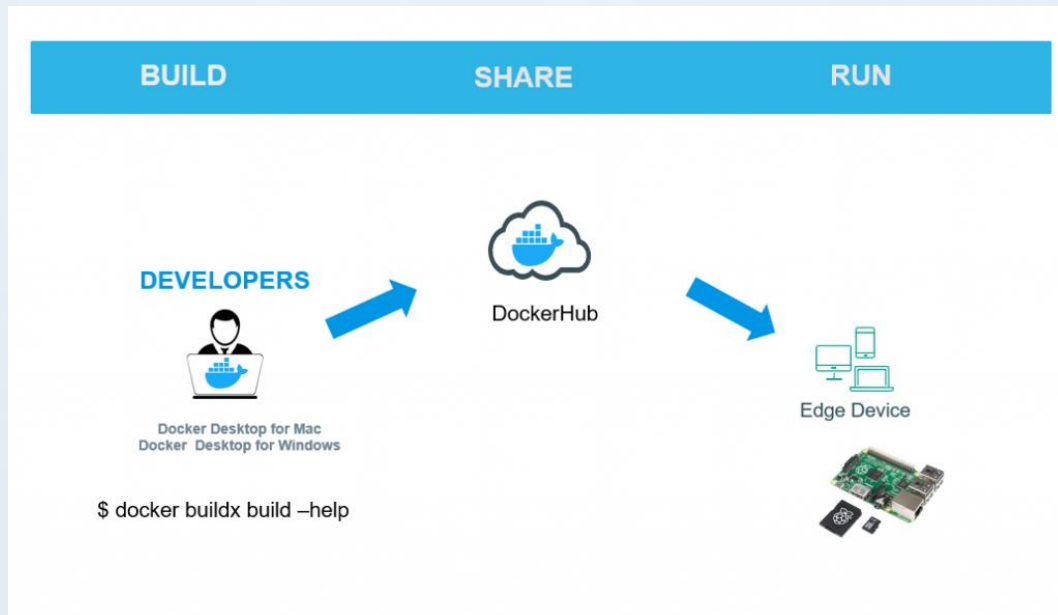
Создание своего образа

Для сборки своего образа используется команда **docker build** и особый файл **Dockefile**, который содержит набор инструкций, которые описывают слои в образе.

Для сборки образа используется контекст, указанный в команде `docker build`, все содержимое загружается в докер демон, если какие-то файлы не требуются, то их можно обавить в файл **.dockerignore**


buildx

- Сборка под другие платформы, например вы работаете на маке, а использовать его планируете на Raspberry Pi
- `docker buildx build --platform linux/arm64 -f Dockerfile.alpine -t devdotnetorg/alpine-ssh:aarch64 . --load`



docker/buildx

Docker CLI plugin for extended build capabilities with BuildKit



60 Contributors 99 Used by 20 Discussions 2k Stars 300 Forks

Основы Dockerfile

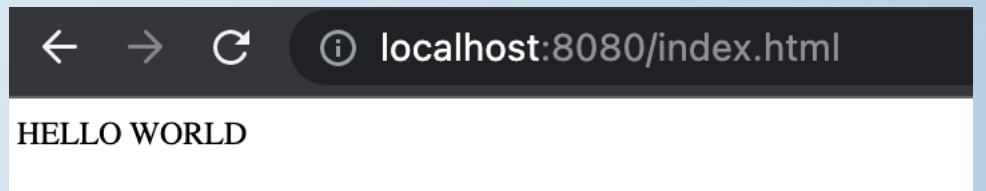
- Всегда надо указывать базовый образ с помощью **FROM**.
Самый базовый образ scratch
- Чтобы скопировать файл из контекста **COPY** /что-то в контексте /куда-то в образе
- Чтобы запустить команду в образе **RUN** команда
- **ENV** позволяет определять переменные окружения для будущего контейнера

```
<> index.html ×
<> index.html > h1
1  <h1>HELLO WORLD</h1>
2
3

Dockerfile ×
Dockerfile > ...
1  FROM nginx
2
3  COPY index.html /usr/share/nginx/html/

TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
alexey-yakubov:devops alexey.yakubov$ docker build
[+] Building 0.2s (7/7) FINISHED
=> [internal] load build definition from Dockerf
=> => transferring dockerfile: 93B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/
=> [internal] load build context
=> => transferring context: 59B
=> CACHED [1/2] FROM docker.io/library/nginx
=> [2/2] COPY index.html /usr/share/nginx/html/
=> exporting to image
=> => exporting layers
=> => writing image sha256:5542ddce9da39bc570905
=> => naming to docker.io/library/hello_world

Use 'docker scan' to run Snyk tests against image
alexey-yakubov:devops alexey.yakubov$ docker run
```



Использование слоев

Команды RUN и COPY порождают слои, слой – это изменения в файловой системе, вызванные вышей командой (новые файлы или удаления) ВАЖНО! Слой «запекается» в момент вызова команды.

```
Dockerfile > ...
1 FROM alpine
2 RUN truncate -s 10M big_file.txt
3 RUN rm -rf big_file.txt
4
```

TERMINAL	PROBLEMS	OUTPUT	DEBUG	CONSOLE
● alexey-yakubov:devops alexey.yakubov\$ docker images big_file1				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
big_file1	latest	1c22d451965c	3 minutes ago	16MB
● alexey-yakubov:devops alexey.yakubov\$ docker history big_file1				
IMAGE	CREATED	CREATED BY		SIZE
1c22d451965c	3 minutes ago	RUN /bin/sh -c rm -rf big_file.txt # buildkit		0B
<missing>	3 minutes ago	RUN /bin/sh -c truncate -s 10M big_file.txt ...		10.5MB
<missing>	4 weeks ago	/bin/sh -c #(nop) CMD ["/bin/sh"]		0B
<missing>	4 weeks ago	/bin/sh -c #(nop) ADD file:2a949686d9886ac7c...		5.54MB

Использование слоев (часть 2)

```
Dockerfile > ...
1 FROM alpine
2 RUN truncate -s 10M big_file.txt &&\
3 |rm -rf big_file.txt
4
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
alexey-yakubov:devops alexey.yakubov$ docker images big_file2
REPOSITORY TAG IMAGE ID CREATED SIZE
big_file2 latest 4f176ad5dd8d 6 seconds ago 5.54MB
alexey-yakubov:devops alexey.yakubov$ docker history big_file2
IMAGE CREATED CREATED BY SIZE
4f176ad5dd8d 31 seconds ago RUN /bin/sh -c truncate -s 10M big_file.txt ... 0B
<missing> 4 weeks ago /bin/sh -c #(nop) CMD ["/bin/sh"] 0B
<missing> 4 weeks ago /bin/sh -c #(nop) ADD file:2a949686d9886ac7c... 5.54MB
alexey-yakubov:devops alexey.yakubov$
```

Таким образом в каждом слое рекомендуется минимизировать количество создаваемых ненужных файлов, те сразу их чистить.

Например, *RUN yum install package && yum clean all*

Слои и кеш

Docker высчитывает хеш от каждого слоя, при попытке его собрать, хеш слоя считается в зависимости от хеша предыдущего слоя, т.о. если поменялся предыдущий слой, все последующие слои пересоберутся.

```
Dockerfile > ...
1 FROM nginx
2
3 RUN sleep 10
4 COPY index.html /usr/share/nginx/html/
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
alexey-yakubov:devops alexey.yakubov$ time docker build -t nginx_cache .
[+] Building 10.7s (8/8) FINISHED
```

```
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 106B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/nginx:latest
=> CACHED [1/3] FROM docker.io/library/nginx
=> [internal] load build context
=> => transferring context: 59B
=> [2/3] RUN sleep 10
=> [3/3] COPY index.html /usr/share/nginx/html/
=> exporting to image
=> => exporting layers
=> => writing image sha256:db653b8e135bd443f53aafb6d6c8a6e5d6add482f015aceeafb6261826ebb01
=> => naming to docker.io/library/nginx_cache
```

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

```
real    0m11.657s
user    0m0.143s
sys     0m0.288s
```

```
alexey-yakubov:devops alexey.yakubov$ time docker build -t nginx_cache .
[+] Building 0.1s (8/8) FINISHED
```

```
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 36B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/nginx:latest
=> [1/3] FROM docker.io/library/nginx
=> [internal] load build context
=> => transferring context: 31B
=> CACHED [2/3] RUN sleep 10
=> CACHED [3/3] COPY index.html /usr/share/nginx/html/
=> exporting to image
=> => exporting layers
=> => writing image sha256:db653b8e135bd443f53aafb6d6c8a6e5d6add482f015aceeafb6261826ebb01
=> => naming to docker.io/library/nginx_cache
```

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

```
real    0m0.480s
user    0m0.115s
sys     0m0.069s
```

Много слоев с кешом или минимизировать размер?!

Как и везде в IT нет идеального ответа на этот вопрос, все зависит от условий использования образа и задач:

- Образ часто скачивается, но мало меняется – минимизируйте размер, уменьшив число слоев.
- Образ постоянно пересобирается – выделите слои, которые меняются редко и добавляйте изменения в последнем слое (например, сначала установка зависимостей, потом копируем исходный код)

Лайфхак с компилируемыми приложениями

Докер позволяет во время сборки использовать дополнительные образы для сборки кода, отделив сборочную часть и эксплуатационную, с целью минимизации места

```
# syntax=docker/dockerfile:1
FROM golang:1.16
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go ./
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app ./
CMD [ "./app" ]
```