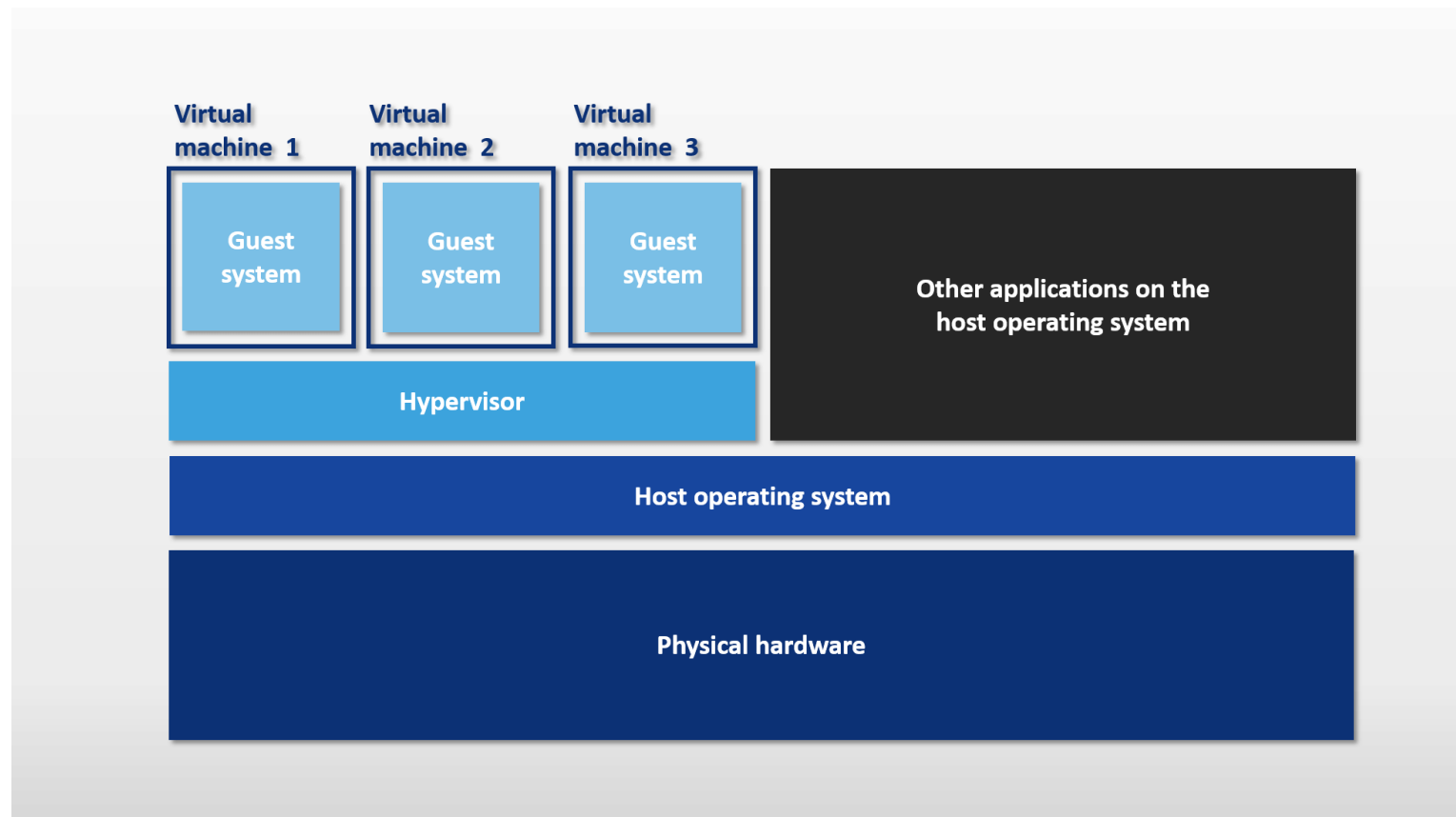


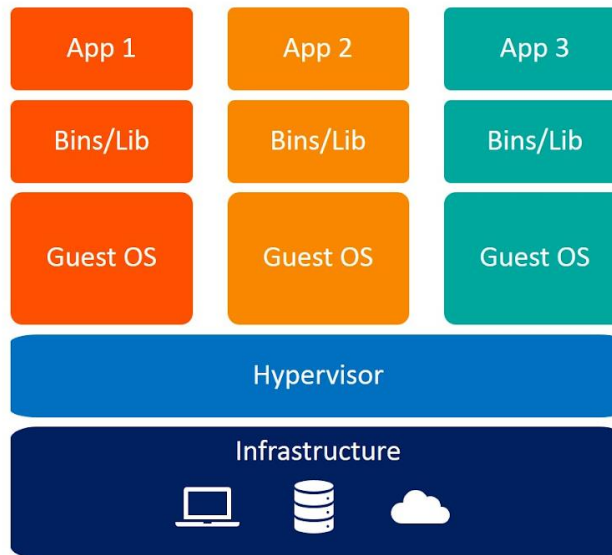
Архитектура ПО

контейнеры

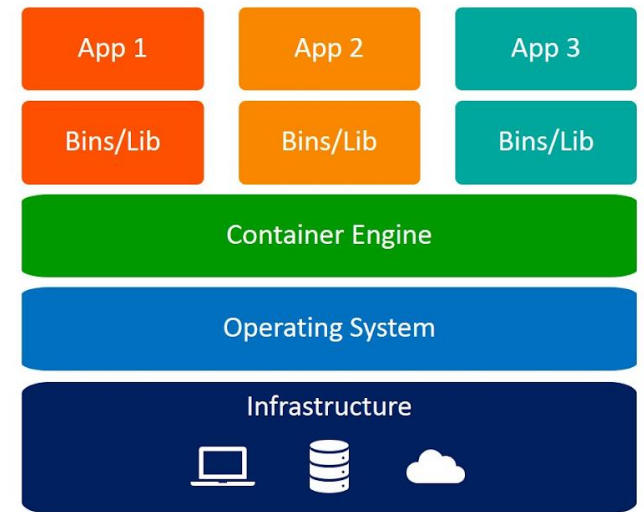
Виртуальные машины



Контейнеры

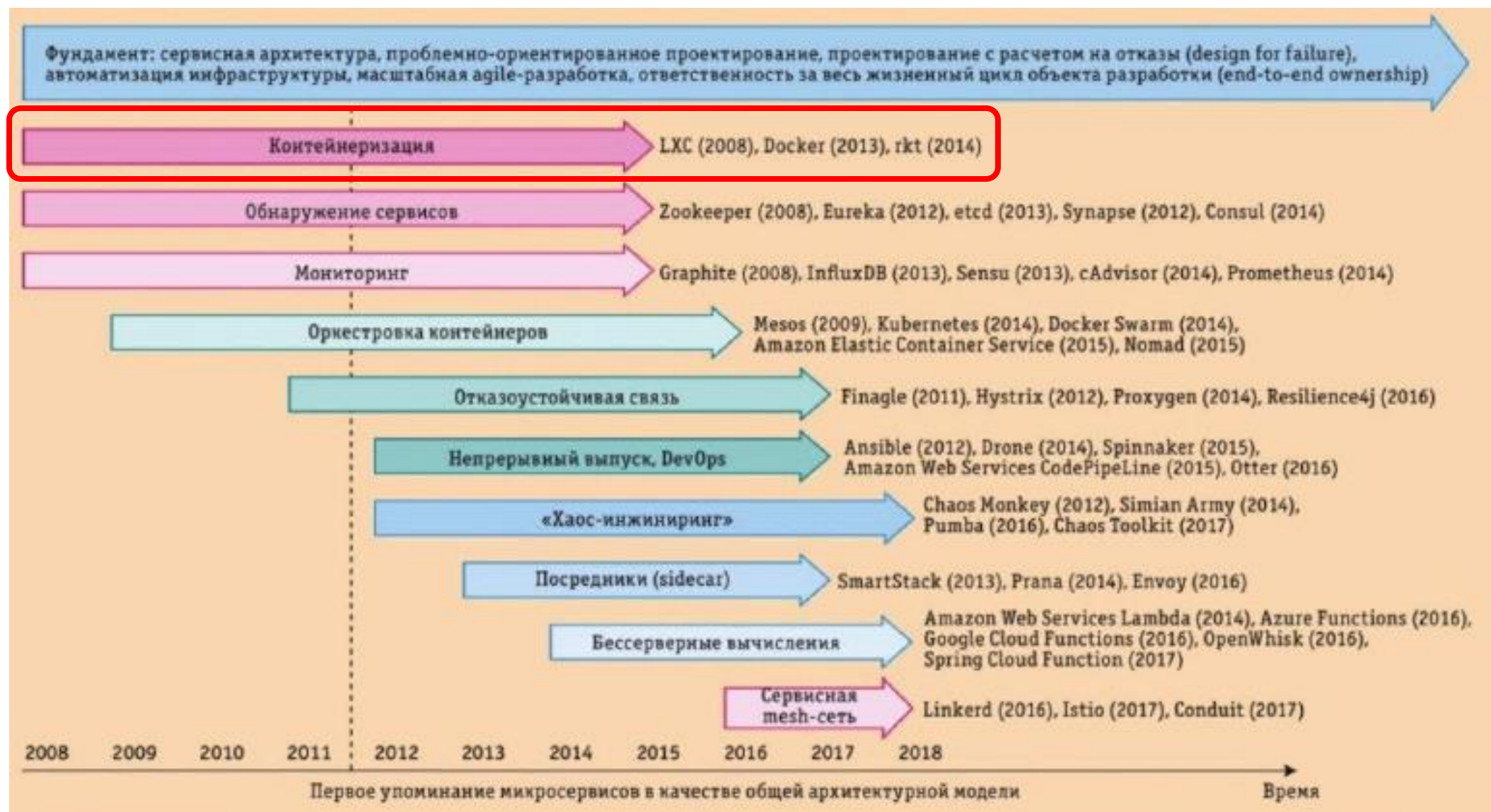


Virtual Machines



Containers

Паттерны микросервисов



Docker- контейнеры

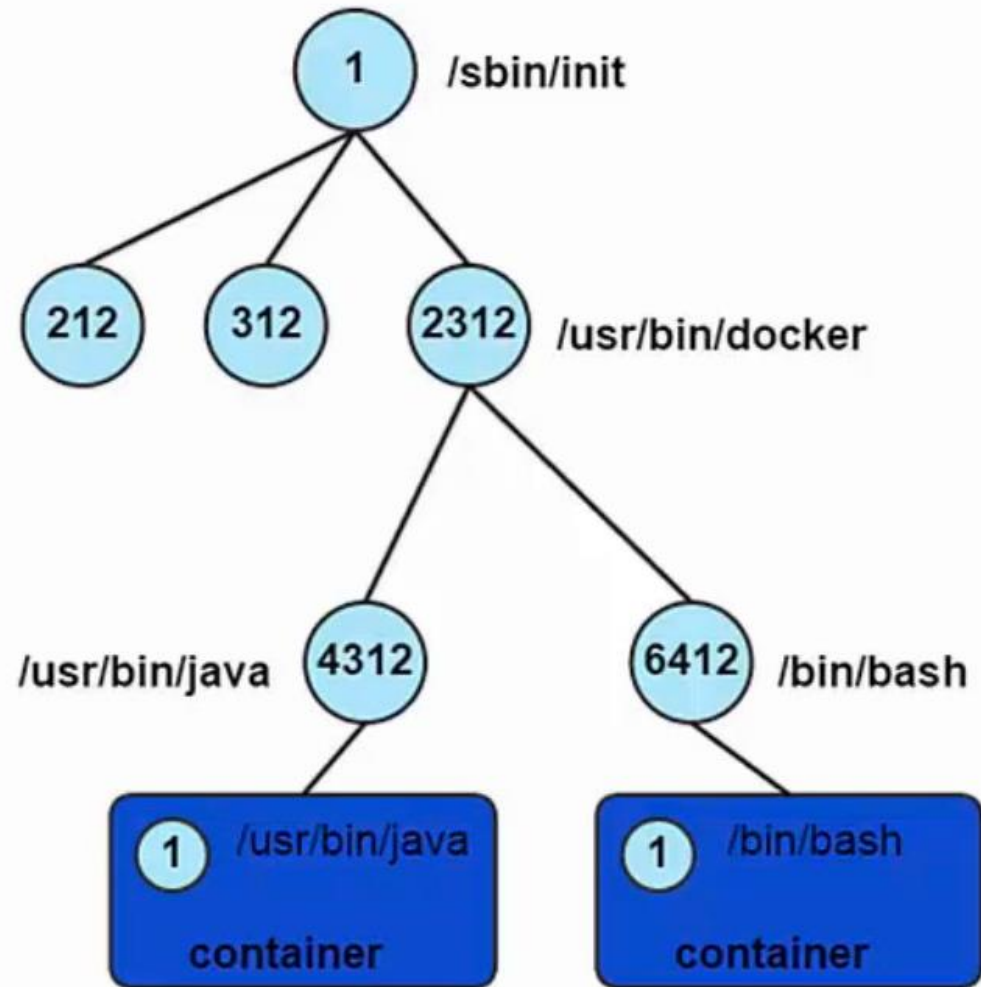


Namespaces

- **PID**
изоляция процессов
- **NET**
изоляция сетей
- **MNT**
изоляция файлов
- **UTS**
изоляция имен и идентификаторов
- **IPC**
изоляция межпроцессорного взаимодействия
- **CGROUPS**
ограничения на процессорные, сетевые, ресурсы памяти, ресурсы ввода-вывода

PID Namespace

- Процессы внутри pid namespace'a видят только процессы из этого же namespace'a
- Каждый pid namespace имеет свою нумерацию процессов (начиная с 1)
- Когда процесс с pid 1 умирает, то умирает весь Namespace
- PID namespace'ы могут быть вложенными



Net Namespace

- Процессы в net namespace'е имеют свой собственный сетевой стек, а именно:
 - Сетевые интерфейсы
 - Таблицу маршрутизации
 - Правила iptables
 - Socket'ы

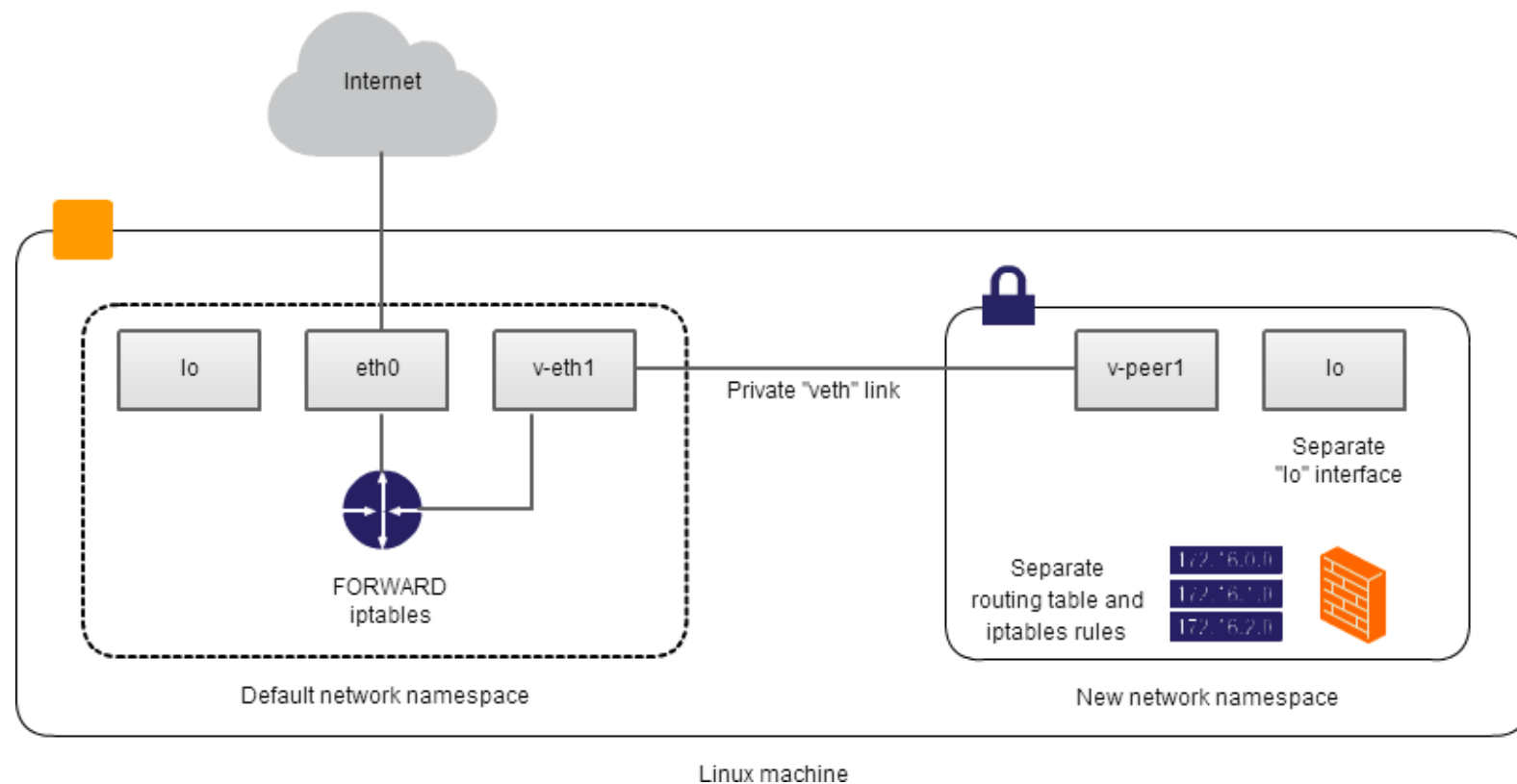
Net Namespace

Создаются два
виртуальных сетевых
интерфейса

Eth0 внутри контейнера

v-ethXXX на хост системе

Все v-ethXXX соединены в
один bridge-интерфейс
(dockero)



MNT Namespace

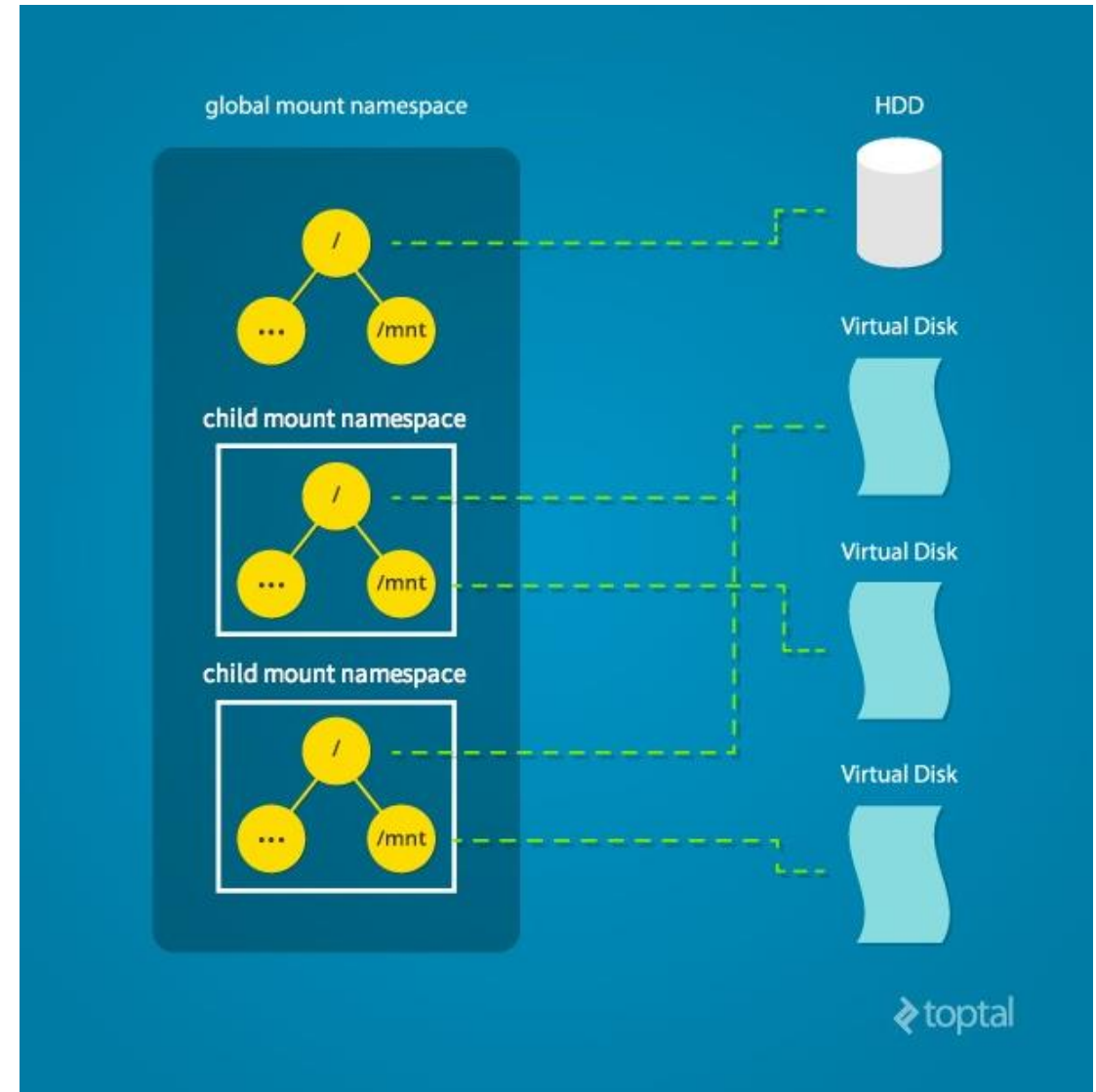
Процессы могут иметь свой собственный root (для chroot)

У процессов могут быть свои приватные "Точки монтирования" (mounts)

`/tmp`

`/proc, /sys`

"Точки монтирования" (mounts) могут быть приватными, а могут быть доступны в нескольких namespace'ах



UTS

UTS используется для изоляции системных идентификаторов: имени узла (nodename) и имени домена (domainname), возвращаемых системным вызовом [uname\(\)](#).

Ipc Namespace

interprocess communications

Процессы или группы процессов могут иметь свои наборы:

- IPC семафоров
- IPC очередей сообщений
- IPC совместно доступной памяти

<https://www.opennet.ru/docs/RUS/lpg/node6.html>

cgroups

- Ограничивает доступ к ресурсам (в т.ч. устройствам)
- Ограничивает доступ к системным вызовам

Хранение данных в **Docker**

Storage (storage drivers)

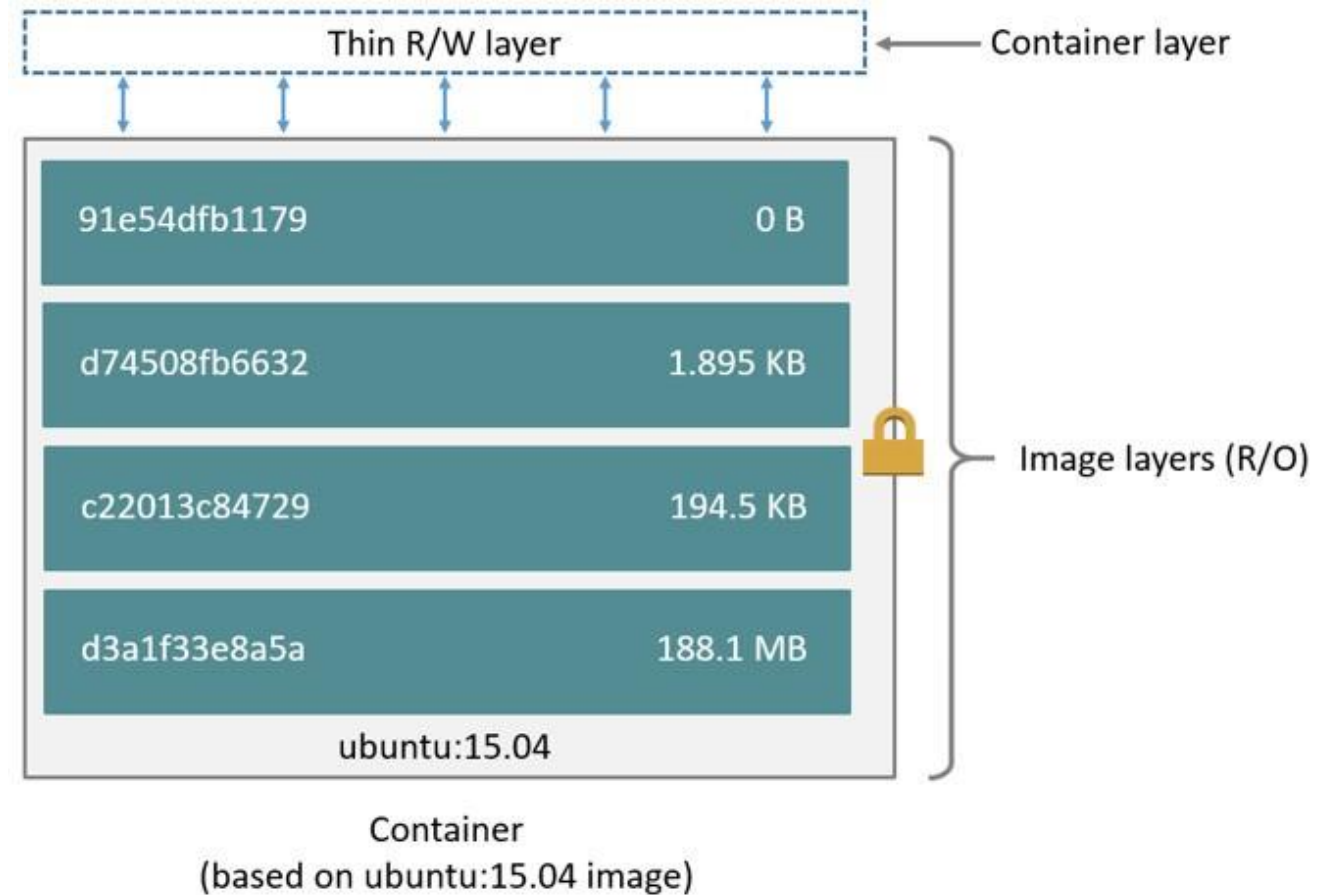
Data Volumes (volume drivers)

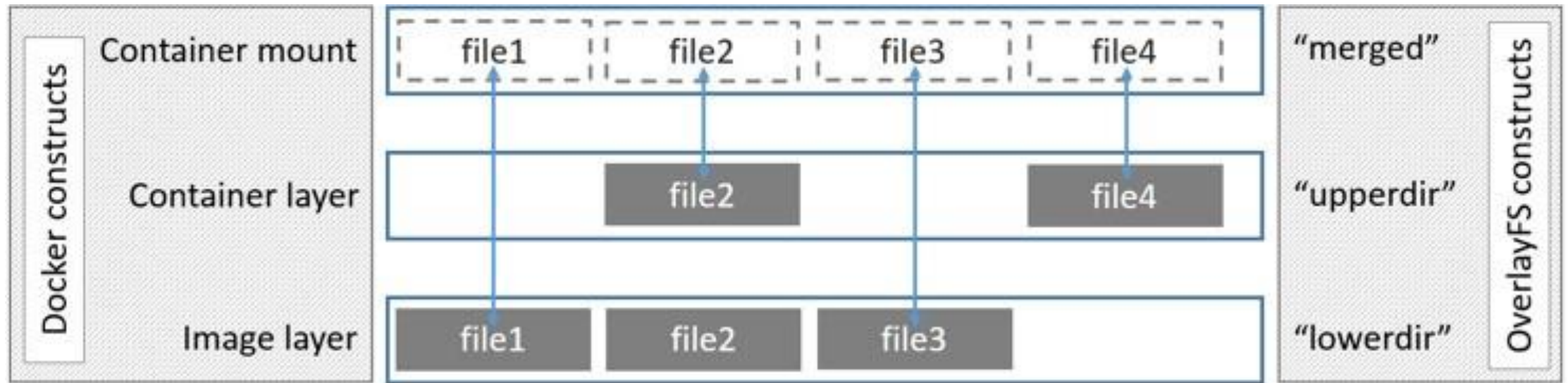
Storage

Обеспечивает хранение слоев образов

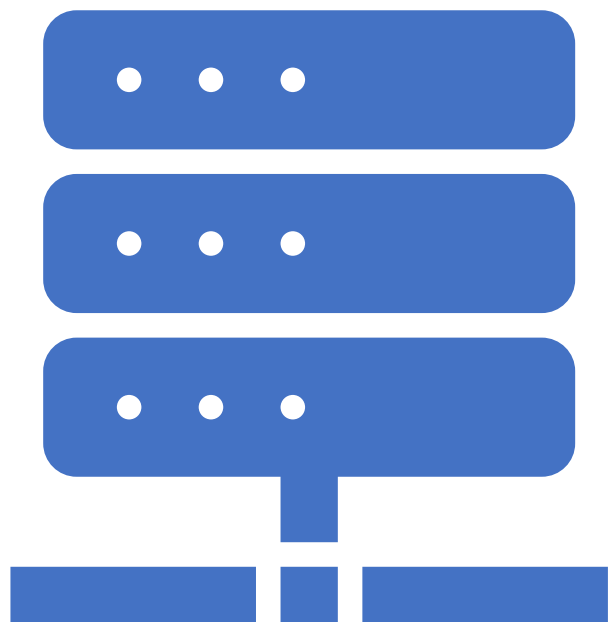
Обеспечивает слой для контейнера

Можно выбрать в зависимости от потребностей и возможностей (но чаще нет необходимости)





Storage



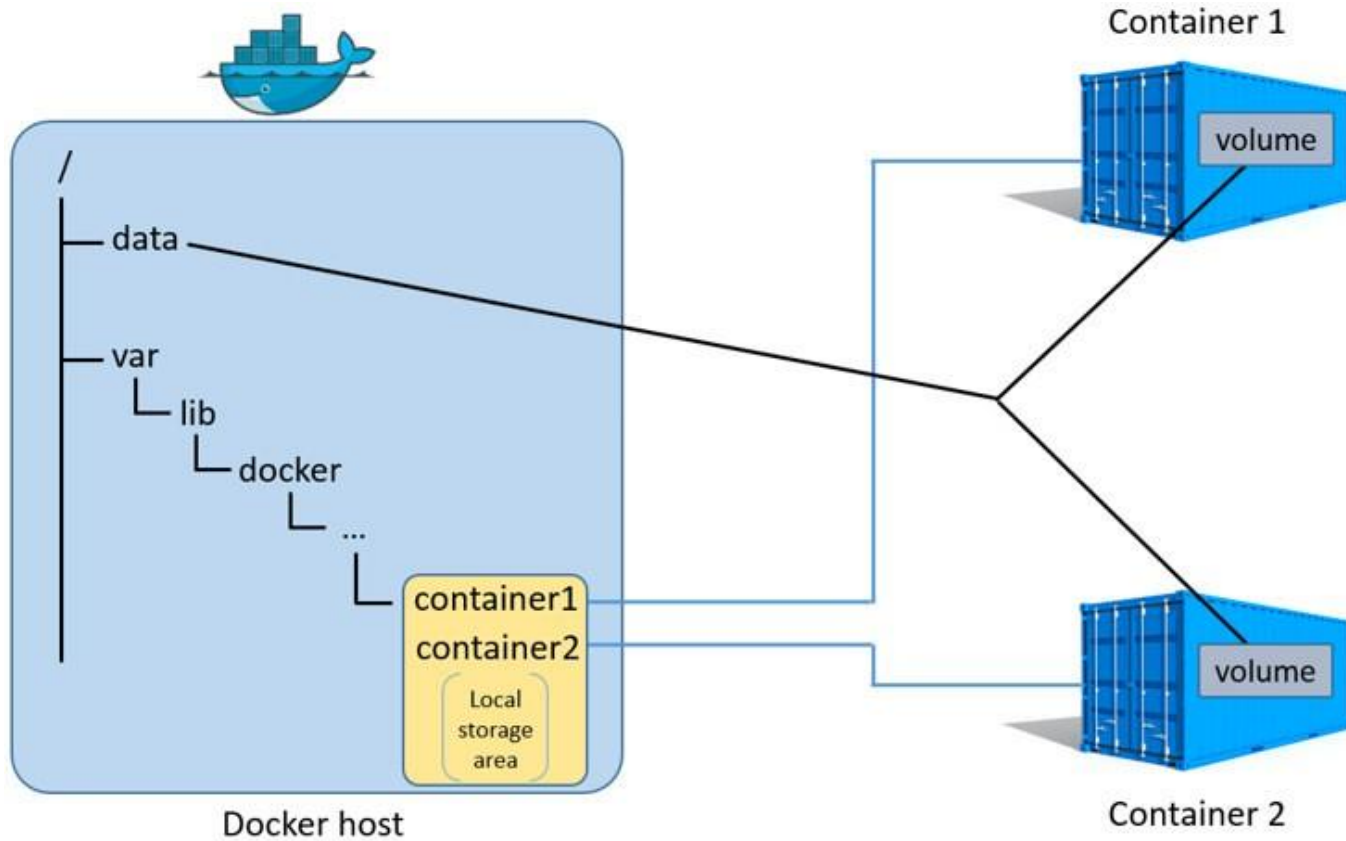
Storage

ИТОГО

Данные могут быть потеряны вместе с остановкой контейнера

Верхний слой (RW-слой) тесно связан с контейнером

Меньше производительность по отношению к data volumes



Data Volumes

Позволяют отделить жизненный цикл данных, которые он в себе хранит, от жизни самого контейнера, который эти данные создал

Типы Data Volumes

Volumes

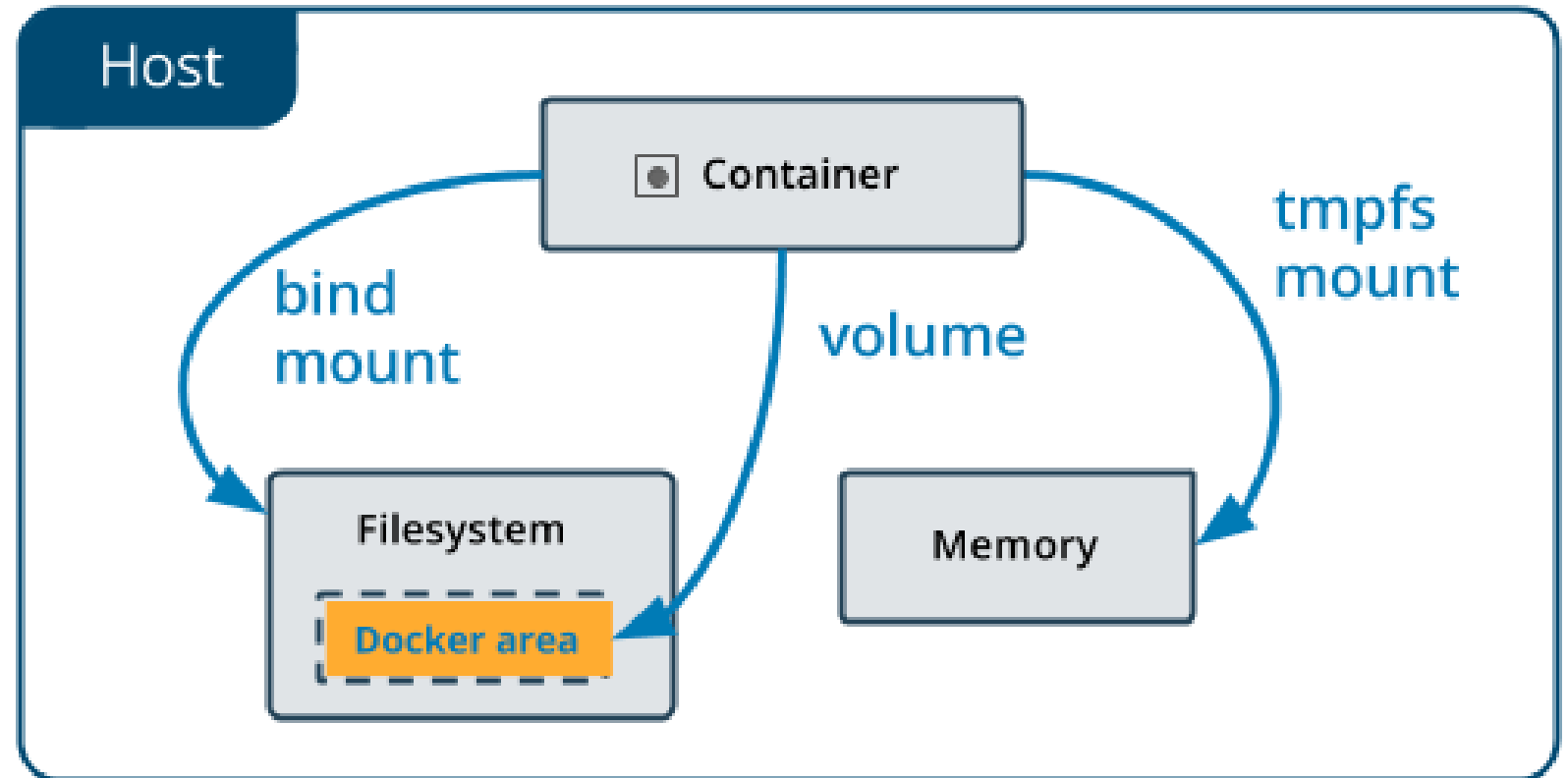
тома управляемые Docker'ом. Другие процессы не должны иметь к ним доступ

Bind mount

директории на файловой системе. Любой процесс может получить к ним доступ

tmpfs

тома расположенные в памяти хоста. Никогда не записываются на диск



Docker Volumes

- Доступ к данным из нескольких контейнеров
- Когда неизвестна файловая структура на хосте
- Когда храним данные удаленно
- Предпочтительнее при миграции с хоста на хост
- Бывают именнованные и неименованные

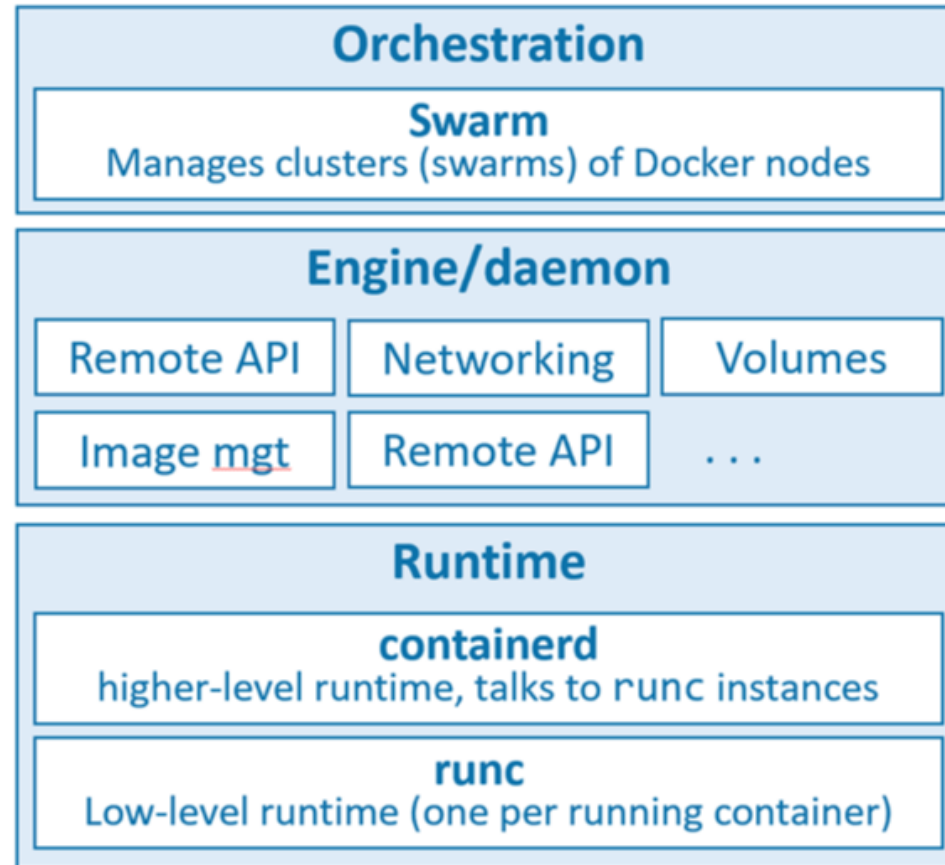
Bind mounts

- Совместный с хостом доступ к данным (исходному коду, артефактам)
- При известной структуре файловой системы хоста

tmpfs

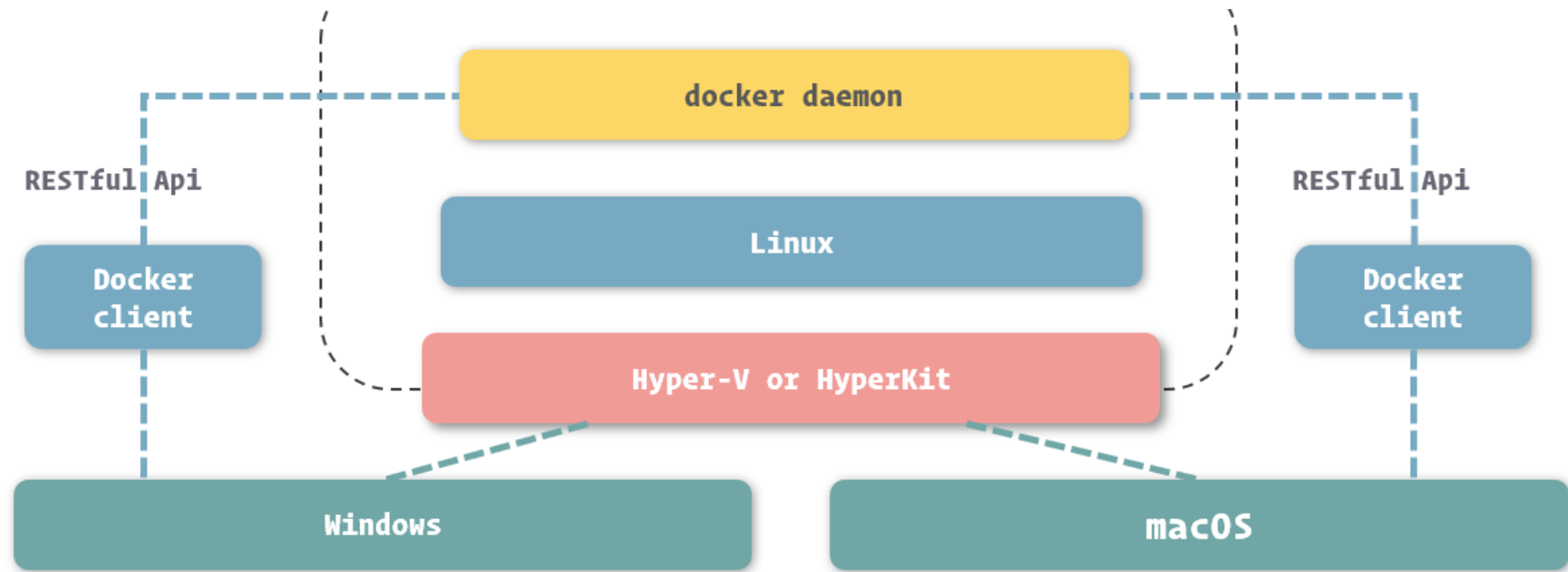
- Когда данные не должны сохраняться на хосте или в контейнере
- Виртуальный диск в памяти

Архитектура Docker



Архитектура docker

- [containerd](#) - это среда выполнения контейнера, которая может управлять полным жизненным циклом контейнера - от передачи / хранения изображений до выполнения, контроля и создания сетей контейнера.
- [runc](#) - это легкий универсальный контейнер времени выполнения, который соответствует спецификации OCI. runc используется containerd для порождения и запуска контейнеров в соответствии со спецификацией OCI. Это также переупаковка libcontainer.



Есть ли не на unix?



Установим docker

Docker
Desktop

Docker Desktop

The #1 containerization software for developers and teams

Your command center for innovative container development

Create an account

Download for Mac - Apple Chip



<https://www.docker.com/products/docker-desktop/>

для mac можно <https://github.com/abiosoft/colima>

Visual Studio Code

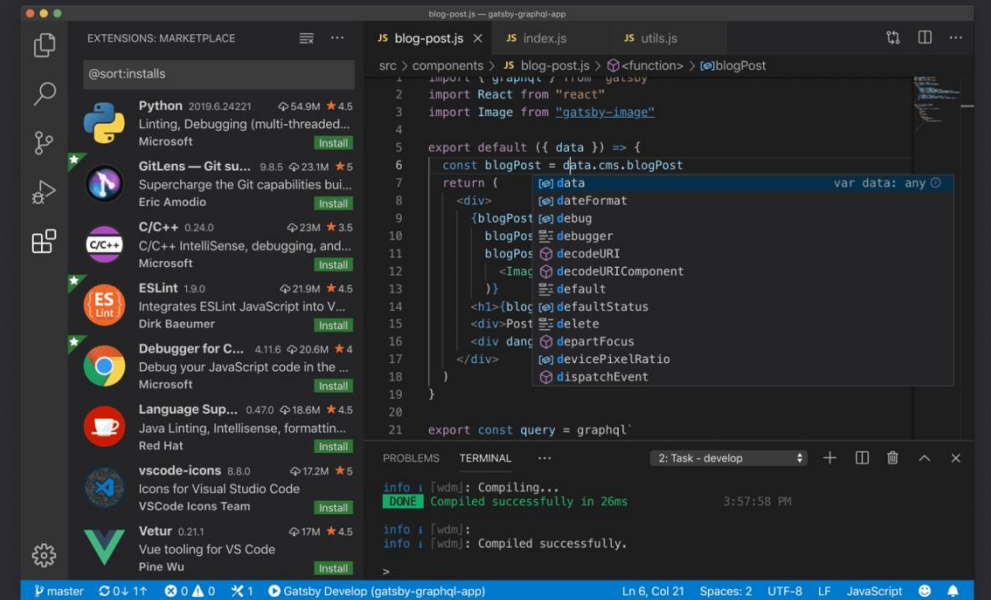
Code editing. Redefined.

Free. Built on open source. Runs everywhere.

Download Mac Universal
Stable Build

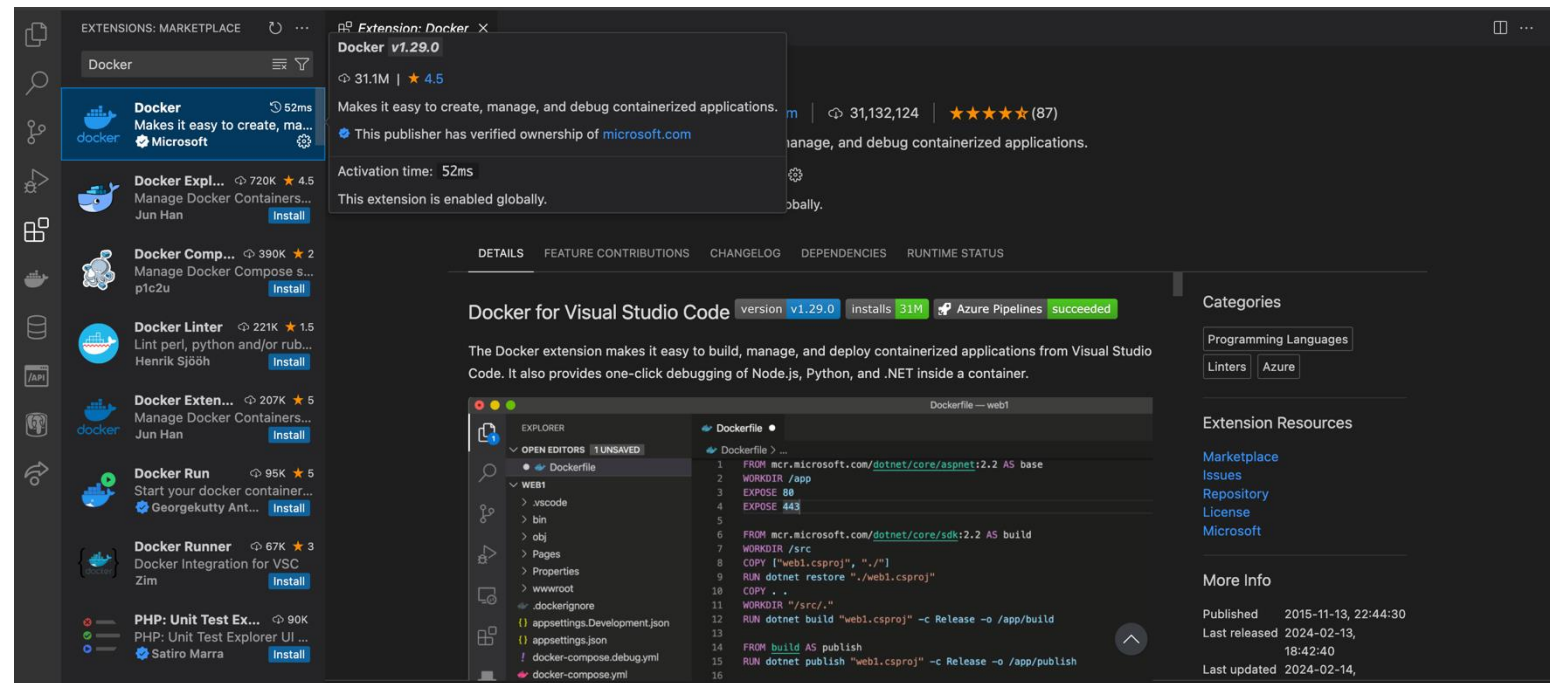
Web, Insiders edition, or other platforms

By using VS Code, you agree to its
[license and privacy statement.](#)



<https://code.visualstudio.com/>

Полезный плагин



<https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker>

Установка

<https://www.docker.com/get-started>

```
$ docker run hello-world
```

```
Hello from Docker. This message shows that  
your installation appears to be working  
correctly. ...
```

Закачиваем
контейнер из
registry

```
# https://hub.docker.com/_/busybox  
docker pull busybox  
docker images
```

Запускаем
команду в
контейнере

```
docker run busybox echo "hello from busybox"
```


Что было запущено?

```
docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
e2f892338ee4 (0) About a minute ago	busybox	"echo 'hello from bu..." serene_gould	About a minute ago	Exited
0bf99c016137 (0) 7 minutes ago	hello-world	"/hello" competent_wing	7 minutes ago	Exited

Удалим
остатки

```
docker rm e2f892338ee4 0bf99c016137
```

Удалим
вообще все

```
docker system prune -a
```

Images

Схемы нашего приложения, которые являются основой контейнеров. В примере выше мы использовали команду `docker pull` чтобы скачать образ **busybox**.

Containers

Создаются на основе образа и запускают само приложение.

Мы создали контейнер командой `docker run`, и использовали образ **busybox**, скачанный ранее.

Список запущенных контейнеров можно увидеть с помощью команды **docker ps**.

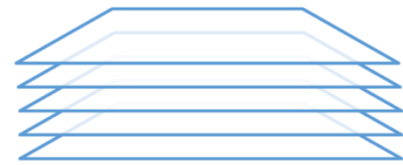
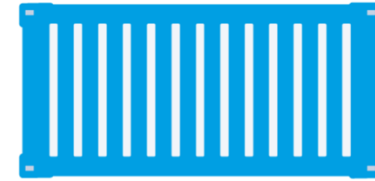


Image
(Build time)



Container
(Runtime)

Image & Container

Запуск сайта из registry

```
docker run -d -p 8080:80 --rm image_name
```

-d – detach

-p - port mapping [порт системы: порт контейнера]

-rm remove container after exit

Образы

`docker images`

TAG — это конкретный снимок или снимок (snapshot) образа, IMAGE ID — это соответствующий уникальный идентификатор образа.

`docker pull ubuntu:12.04`

если не указывать снимок, то будет подставляться latest

- **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.

Базовые команды

- `docker image ls`
- `docker image rm`
- `docker volume ls`
- `docker volume rm`
- `docker network ls`
- `docker network rm`
- `docker pull`

Dockerfile

Текстовый файл с build инструкциями (для сборки образа)

Инструкции декларативно описывают image

Каждая инструкция – промежуточный image

Сборку image делает docker daemon

91e54dfb1179	0 B
d74508fb6632	1.895 KB
c22013c84729	194.5 KB
d3a1f33e8a5a	188.1 MB
ubuntu:15.04	

Image

CMD

RUN

RUN

ADD/COPY

FROM базовый образ

```
FROM <image>[:<tag>]
```

<image> - имя базового образа

<tag> - опциональный атрибут указывающий на версию образа

Примеры:

```
FROM ubuntu:16.04
```

```
FROM quay.io/vektorlab/ctop
```

LABEL

метаданные

```
LABEL <key>=<value> [<key>=<value> ...]
```

<key> - ключ

<value> - значение

Примеры:

```
LABEL maintainerr="user@example.com"
```

```
LABEL description="This text illustrates \ that label-values  
can span multiple lines."
```

COPY

копирование данных

```
COPY <src> [<src> ...] <dst>
```

<src> - файл или директория внутри build контекста

<dst> - файл или директория внутри контейнера

Примеры:

```
COPY start* /startup/
```

```
COPY httpd.conf magic /etc/httpd/conf/
```

ADD

«крутое копирование», может распаковывать архивы

```
ADD <src> [<src> ...] <dst>
```

<src> - файл, директория, архив, сайт внутри build контекста

<dst> - файл или директория внутри контейнера

Примеры:

```
ADD web-page-config.tar /
```

```
ADD http://example.com/foobar /
```

ENV

устанавливает постоянные переменные среды

```
ENV <key> <value>
```

<key> - имя переменной окружения

<value> - присваиваемое значение

Примеры:

```
ENV LOG_LEVEL debug
```

```
ENV DB_HOST 127.0.0.1:3389
```

WORKDIR

задаёт рабочую
директорию для
следующей
инструкции

`WORKDIR <path>`

`<path>` – путь внутри
контейнера

Примеры:

`WORKDIR /app`

VOLUME

создаёт точку
монтирования для
работы с
постоянным
хранилищем

```
VOLUME <dst> [<dst> ...]
```

<dst> – директория монтирования для
volume 'a

Примеры:

```
VOLUME /app /db /data
```

```
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
```

EXPOSE

указывает необходимость открыть порт

```
EXPOSE <port>[/<proto>] [<port>[/<proto>] ...]
```

<port> - порт по которому контейнер будет слушать

<proto> - tcp или udp

Примеры:

```
EXPOSE 5000
```

```
EXPOSE 8080/tcp 3389/udp
```

RUN

`RUN <command>`

`<command>` - команда которая будет выполнена при создании образа

Примеры:

```
RUN apt-get update && apt-get install nginx
```

```
RUN ["bash", "-c", "rm", "-rf", "/tmp/abc"]
```

ENTRYPOINT

предоставляет команду с аргументами для вызова во время выполнения контейнера. аргументы не переопределяются

```
ENTRYPOINT <command>
```

<command> – команда которая будет выполнена при старте контейнера

Примеры:

```
ENTRYPOINT exec top -b
```

```
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D",  
"FOREGROUND"]
```

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

CMD <command>

<command> - команда которая будет выполнена при старте контейнера (параметр для `entrypoint`, по умолчанию `/bin/sh -c`)

Примеры:

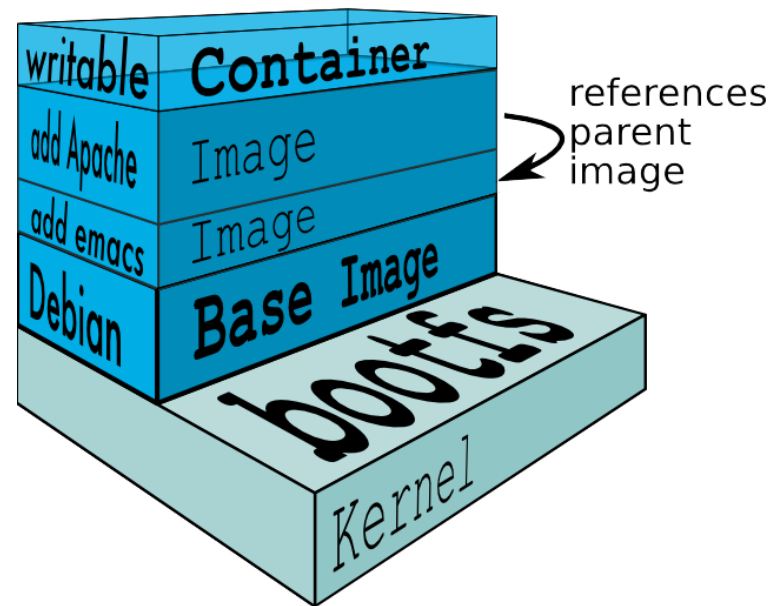
CMD `/start.sh`

CMD `["echo", "Dockerfile CMD demo"]`

Пример example_01

```
# our base image
FROM ubuntu:latest
RUN apt-get update -y
RUN apt-get install nginx wget curl openssl apt-transport-https
-y && apt-get clean -y
RUN echo "daemon off;" >> /etc/nginx/nginx.conf
COPY index.html /var/www/html/
EXPOSE 80 443
CMD ["nginx"]
#docker build . -t my_nginx
#docker images
#docker run -p 80:80 -i my_nginx
```

Оптимизация сборки Docker Image



Оптимизация сборки

- Каждая инструкция в Dockerfile это отдельный image
- Инструкции кешируются с помощью images

Оптимизация сборки

Каждая команда в Dockerfile, новый image

ENV myvar true

<- image!!!

RUN apt-get install -y nginx
RUN apt-get install -y php-fpm

<- image!

<- image!

RUN apt-get install -y imagemagick

<- image!

ADD https://some-site.com/soft/master.tar.gz /bin/

<- image!!!!

CMD ["/bin/cool-soft"]

<- image!

Оптимизация сборки

Каждая команда в Dockerfile, новый image

ENV myvar true

<- image

RUN apt-get install -y nginx && \ apt-get install -y
php-fpm && \ apt-get install -y imagemagick

<- image

ADD https://some-site.com/soft/master.tar.gz /bin/

<- image

CMD ["/bin/cool-soft"]

<- image

Поменяем порядок инструкций

Есть ли разница?

ENV myvar false

ADD https://some-site.com/soft/master.tar.gz /bin/ RUN apt-get

install -y nginx && \
apt-get install -y php-fpm && \ apt-get install -
y imagemagick

CMD ["/bin/cool-soft"]

Кеширование сборки

Как работала упаковка новой версии кода в образ до изменений:

ENV myvar true

<- image, **cache hit**

RUN apt-get install -y nginx && \ apt-get install -y
php-fpm && \ apt-get install -y imagemagick

<- image, **cache hit**

ADD https://some-site.com/soft/master.tar.gz /bin/

<- image, **cache miss**

CMD ["/bin/cool-soft"]

<- image, **cache miss**

Кеширование сборки

Как будет работать после изменений?

ENV myvar false

<- image, **cache hit**

ADD https://some-site.com/soft/master.tar.gz /bin/

<- image, **cache miss**

RUN apt-get install -y nginx && \ apt-get install -y
php-fpm && \ apt-get install -y imagemagick

<- image, **cache miss**

<- image, **cache miss**

CMD ["/bin/cool-soft"]

Вывод: порядок инструкций важен!

Работа с кешем

Кеширование очень важна для реализации быстрых сборок

ADD, COPY - файлы кешируется, в случае изменений файлов, кеш сбрасывается

Для остальных команда(в т.ч. RUN) проверяется только изменение команды.

Например: RUN apt-get -y update, не проверяет обновления постоянно, только первый раз

```
docker build --no-cache
```

Отправим в
registry

```
docker login -u "myusername" -p "mypassword"  
docker.io
```

```
docker container ps -a
```

```
docker container commit 518382b41c7a  
myusername /server2021
```

```
docker push ddzuba/server2021
```

Debug

```
> docker image ls
```

```
> docker container run --rm -ti my_image1  
/bin/bash
```


.dockerignore

- Содержимое директории указанной при docker build попадает в build контекст
- Лучшая практика - держать в директории только необходимое
- Иначе используйте .dockerignore

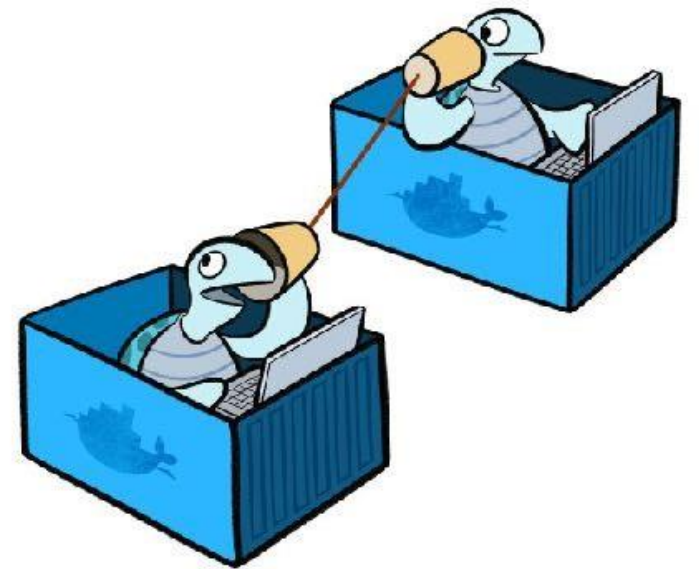
Общие рекомендации

- Избегайте установки лишних пакетов
- Уменьшайте количество слоев
- Один контейнер - одна задача
- Чистите за собой
- Разные контейнеры для сборки и запуска

Выбор базового образа

- Должен содержать необходимое ПО для сборки образа и запуска приложения
- Полезно иметь утилиты для дебага: telnet, ping
- Популярный базовый образ alpine

Как заставить
контейнеры
общаться друг с
другом?



Как заставить контейнеры общаться друг с другом?

- Docker Network Drivers - подключаемые модули для управления сетью контейнеров
 - **Native (встроенные в Docker)**
 - Remote (сторонние)

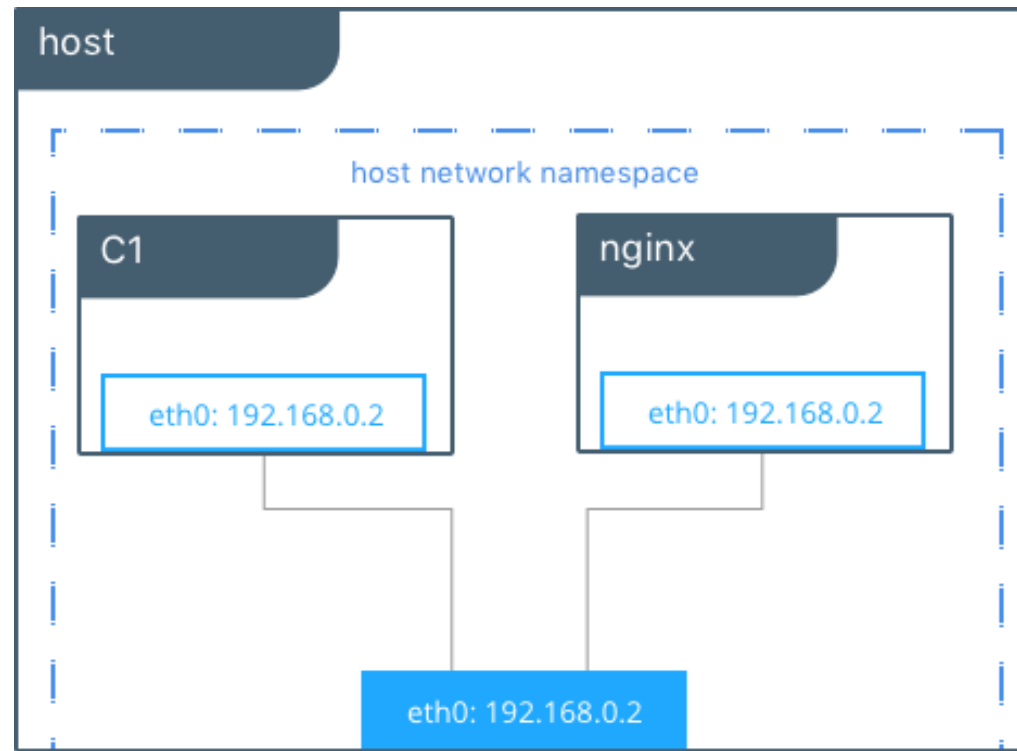
Native Docker network drivers

- None
- Host
- Bridge
- Overlay
- MACVLAN

```
>docker network ls
```

NETWORK	ID	NAME	DRIVER	SCOPE
	6d10b6cf938f	bridge	bridge	local
	a0f911148a5c	host	host	local
	a89f8bfd263a	none	null	local

Что в комплекте?



Host driver

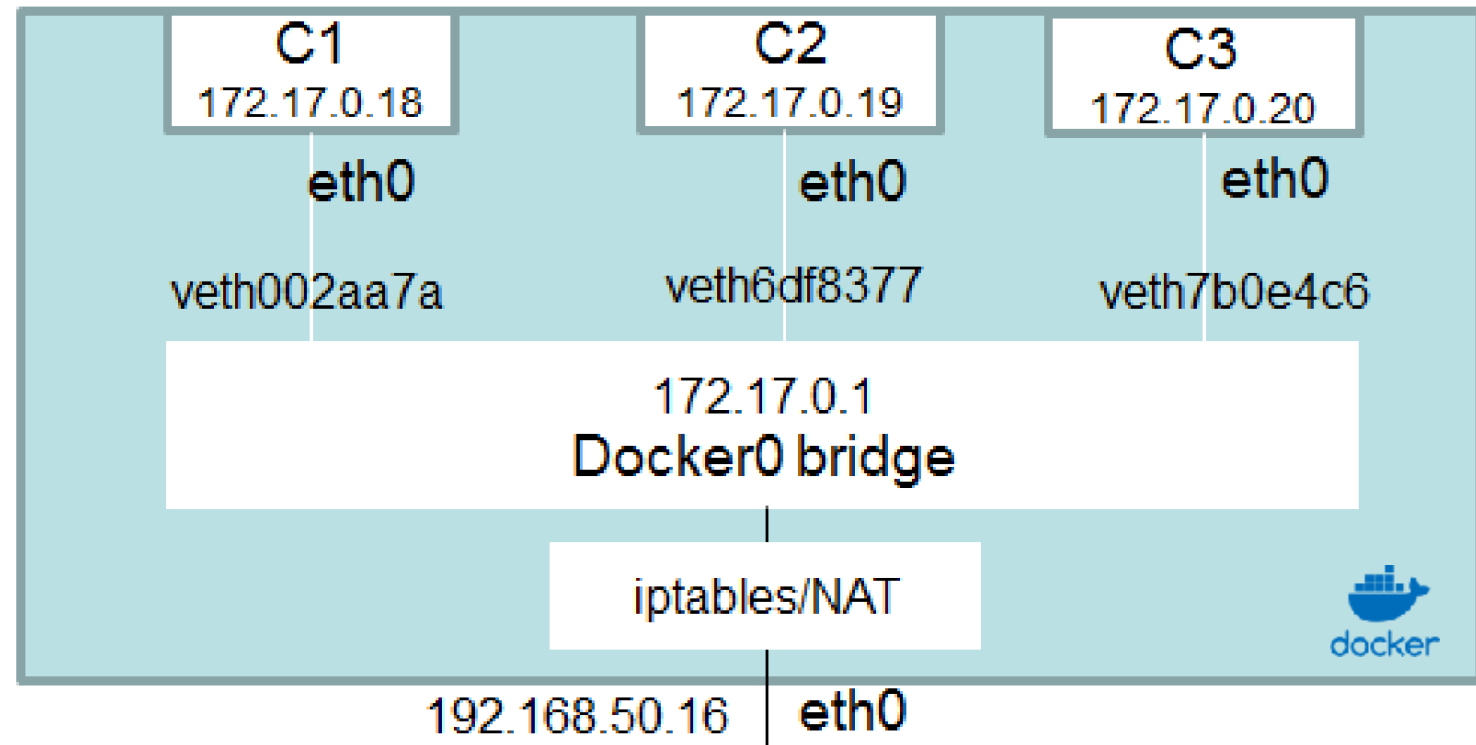
Host driver

- Контейнер использует Net-namespace хоста
- Сеть не управляется самим Docker
- Два сервиса в разных контейнерах
НЕ могут запускаться на одном порту
- Производительность сети контейнера равна
производительности сети хоста

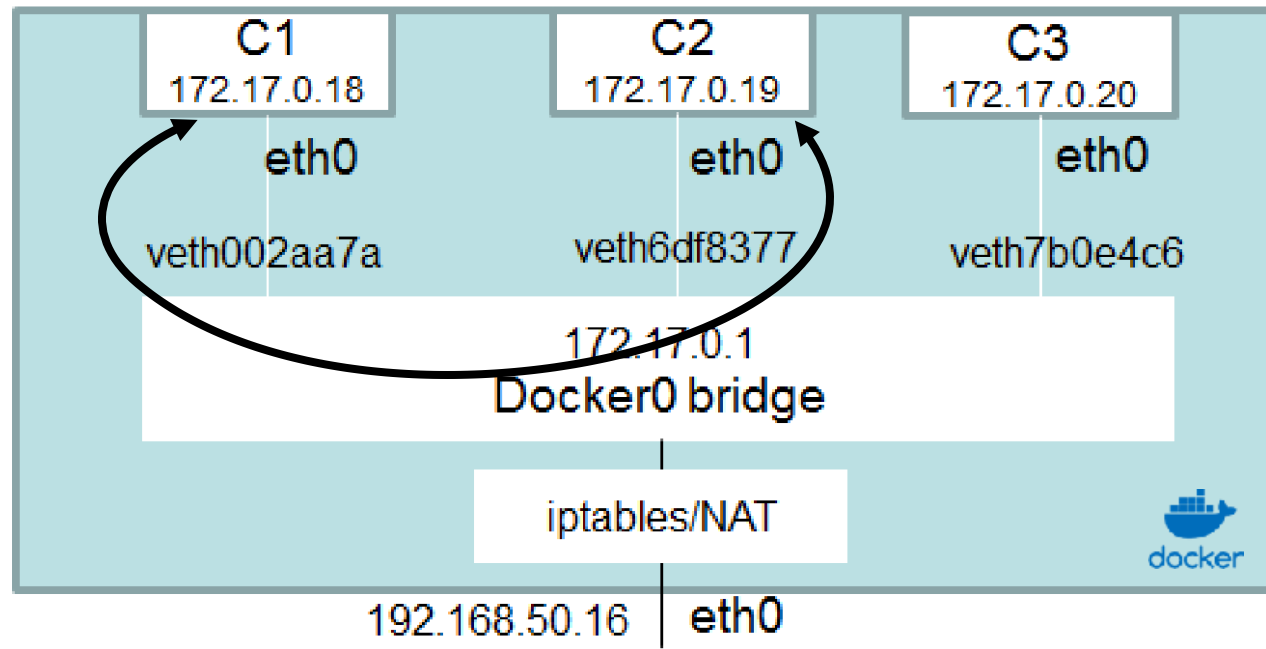
None driver

- Нет ничего, кроме loopback
- Для контейнера создается свой Net-namespace
- Сеть контейнера полностью изолирована

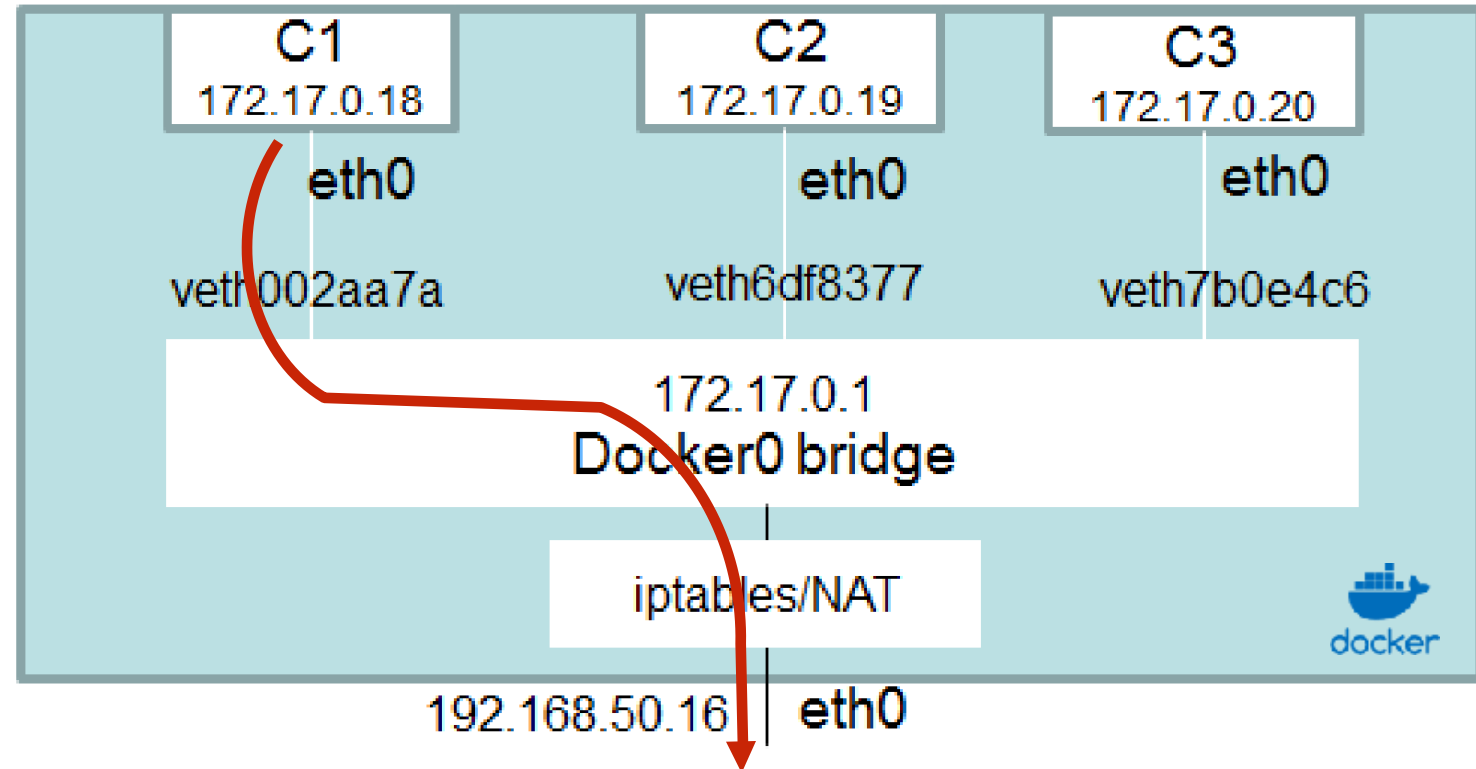
Bridge driver



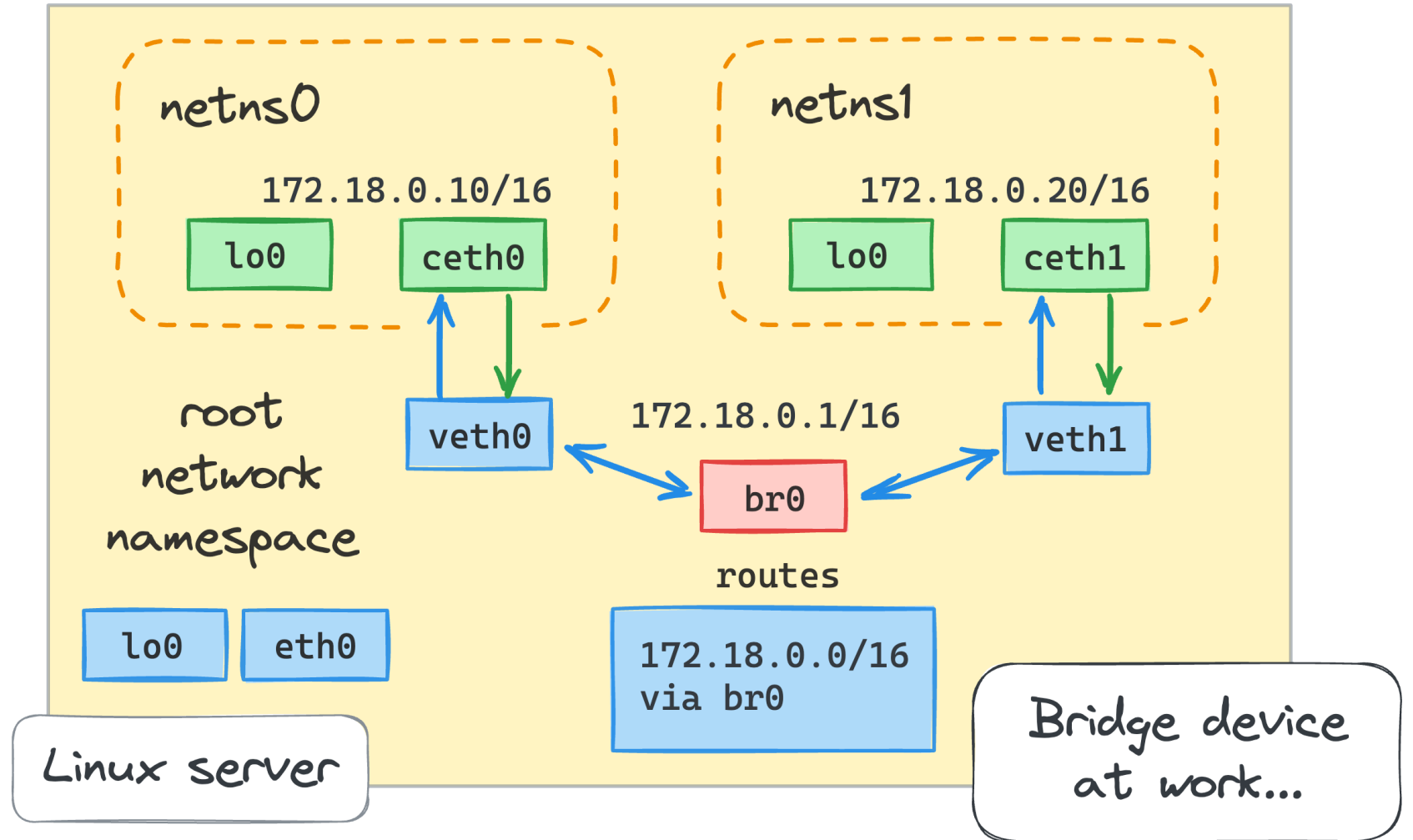
Bridge driver



Bridge driver



Bridge network



<https://labs.iximiuz.com/tutorials/container-networking-from-scratch>

Доступ к контейнерам извне

> docker run **-P** или **--publish-all** .

- распознает строки с ключевым словом **EXPOSE** в Dockerfile и флаг **--expose** при запуске контейнера
- привязывает доступный порт на хосте из диапазона 32768 - 61000
- Для избежания неявно открытых портов **НЕ РЕКОМЕНДУЕТСЯ** использовать

Доступ к контейнерам извне

`docker run -p PORT --publish = PORT.`

явно задает какой порт докер хоста мы хотим привязать к порту докер контейнера.

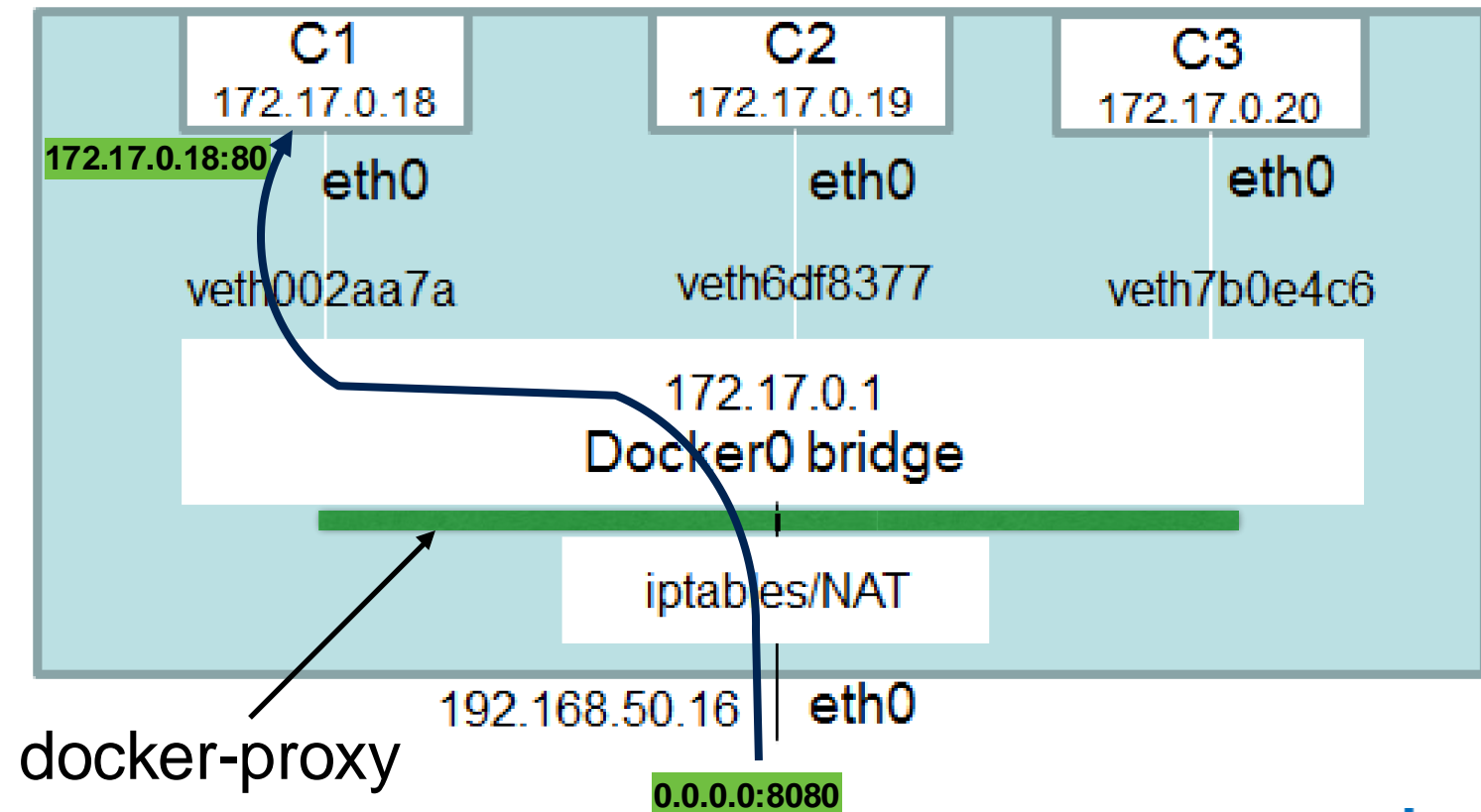
PORT может быть:

- `ip_addr:hostPort:containerPort`
- `ip_addr::containerPort`
- `hostPort:containerPort`
- `containerPort`

Обязательным для указания является порт контейнера.

Пример

```
docker run --name C1 -d -p 8080:80 nginx
```



Особенности default bridge network

- Назначается по-умолчанию для контейнеров
- Нельзя вручную назначать IP-адреса
- Нет Service Discovery

User Defined Networks

- Bridge
- MacVlan
- Overlay

User Defined Bridge

- Если нужно отделить контейнер или группу контейнеров
- Контейнер может быть подключен к нескольким bridge-сетям (даже без рестарта)
- Работает Service Discovery
- Произвольные диапазоны IP-адресов

User Defined Bridge

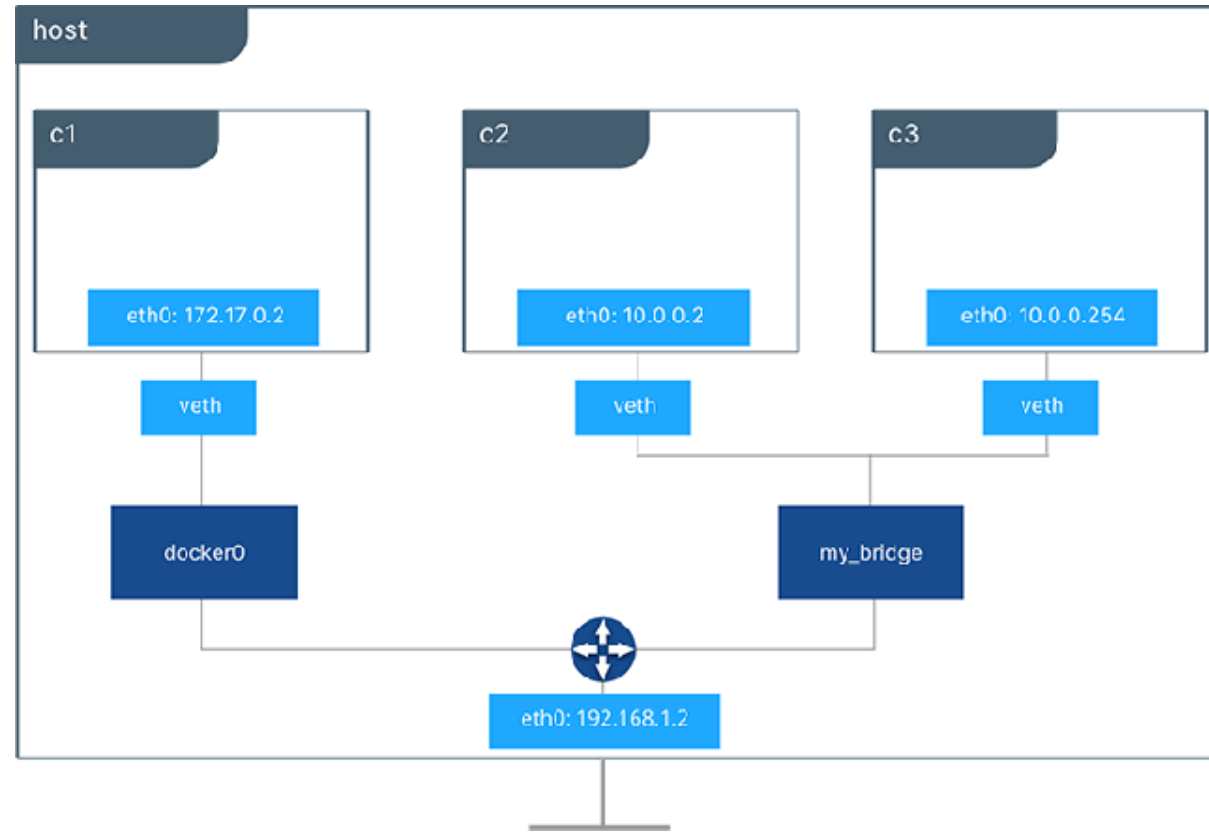
```
> docker network create -d bridge --subnet 10.0.0.0/24 my_bridge
```

```
> docker run --name c1 ubuntu
```

```
> docker run --network my_bridge --name c2 ubuntu
```

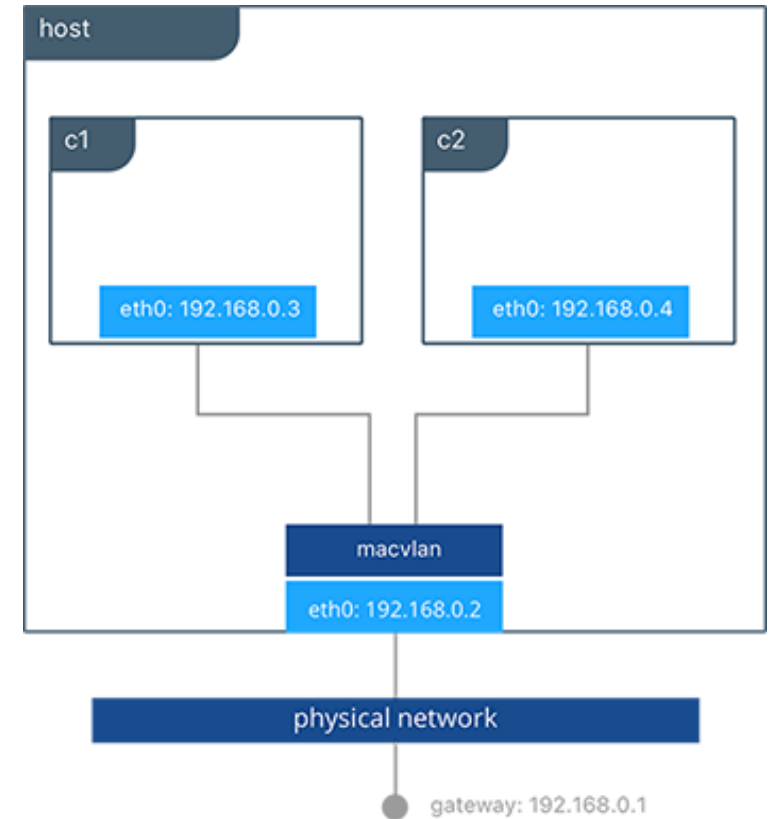
```
> docker run --network my_bridge --name c3 ubuntu
```

User Defined Bridge



Macvlan driver

- Работает на основе sub-interfaces Linux
- Более производительный, чем Linux-bridge
- Если нужно подключить контейнер к локальной сети хоста
- Поддерживается тегирование VLAN (802.1Q)
- Позволяет создавать виртуальные интерфейсы с уникальными MAC-адресами



Overlay network

- Overlay network – позволяет объединить в одну сеть контейнеры нескольких докер хостов.
- работает поверх vxlan
- Нужно хранить состояние распределенной сети

Docker-compose



Docker Compose

- Пакетный менеджер (по аналогии с composer и npm, только у docker — контейнеры), позволяющий описывать необходимую структуру в одном файле (конфиге).
- <https://docs.docker.com/compose/compose-file/compose-file-v3/>

Проблемы

- Одно приложение состоит из множества контейнеров/сервисов
- Один контейнер зависит от другого
- Порядок запуска имеет значение
- `docker build/run/create ...` (долго и много)

Docker-compose

- Отдельная утилита
- Декларативное описание docker инфраструктуры в YAML-формате
- Управление многоконтейнерными приложениями

Как работает?

- Описываем **docker-compose.yml**
- Запускаем **docker-compose up --build**

docker-compose.yml

```
version:      '2'
services:
  post_db:
    image:     mongo:3.2
    volumes:
      - post_db:/data/db
    networks:
      - reddit
  ui:
    build:     ./ui
    image:     ${USERNAME}/ui:1.0
    ports:
      - 9292:9292/tcp
    networks:
      - reddit
  post:
    build:     ./post-py
    image:     ${USERNAME}/post:1.0
    networks:
      - reddit
...
```

compose-file

Основные Секции:

```
version: '2' } (required)
services:
...
volumes:
...
networks:
...
```

version - определяет
поддерживаемый функционал

Services

Описываем конфигурацию
запуска контейнера/группы
контейнеров

```
services:
  post_db:
    image:  mongo:3.2
    volumes:
      - post_db:/data/db
    networks:
      - reddit
  ui:
    build:  ./ui
    image:  ${USERNAME}/ui:1.0
    ports:
      - 9292:9292/tcp
    networks:
      - reddit
  post:
    build:  ./post-py
    image:  ${USERNAME}/post:1.0
    networks:
      - reddit
```

...

Пример example_02

```
version: '3'
services:
  apache:
    image: httpd:2.4
    ports:
      - 8080:80
    volumes:
      - /my_path:/usr/local/apache2/htdocs
```

Что происходит?

image: httpd:2.4

указываем какой образ нам нужно и его версию (список доступных версий и модификаций можно посмотреть в соответствующем docker-hub).

Что происходит?

ports:

- 8080:80

пробрасываем порты между docker и нашей машиной, т.е. все запросы которые будут идти на 8080 порт нашей машины, будут транслироваться на 80 порт docker.

Что происходит?

volumes:

-/my_path:/usr/local/apache2/htdocs

линкуем директорию на нашей машине, с рабочей директорий apache /usr/local/apache2/htdocs, т.е. все файлы находящиеся в директории, будут доступны для apache

Запуск

```
docker-compose up --build  
docker-compose up --build -d
```

Запустим
nginx

example 03

```
version: '3'
```

```
services:
```

```
nginx:
```

```
image: nginx:1.13
```

```
ports:
```

```
- 8080:80
```

```
volumes:
```

```
- /my_path:/usr/share/nginx/html
```

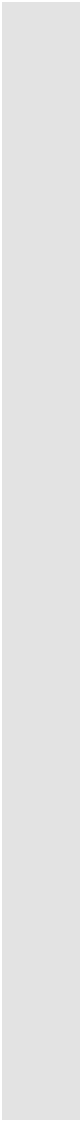

Volumes

```
services:
  post_db:
    volumes:
      - type: volume
        source: sample_vol
        target: /data/sample

      - post_db:/data/db
```

```
volumes:
  post_db:
  sample_vol:
```

} Long notation
} Short notation



Попробуем собрать
несколько сервисов

Подключаемся к СУБД в docker example 06

```
version: '3'

services:
  db-node-1:
    build:
      context: mysql
    dockerfile: Dockerfile
    container_name: db-node-1
    restart: unless-stopped
    environment:
      MYSQL_DATABASE: sql_test
      MYSQL_USER: test
      MYSQL_PASSWORD: pzjqUkMnc7vfNHET
      MYSQL_ROOT_PASSWORD: '1'
    command: >
      --sql-mode='STRICT_TRANS_TABLES,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION'
      --general-log=ON
      --log-queries-not-using-indexes=ON
    networks:
      - my_network
    ports:
      - 3306:3306
    volumes:
      - db-node-1-data:/var/lib/mysql

  sql_test:
    container_name: sql_test_main
    image: ddzuba/sql_test
    command: ./sql_test db-node-1
    networks:
      - my_network
    depends_on:
      - "db-node-1"

volumes:
  db-node-1-data:

networks:
  my_network:
    driver: bridge
```

Health checks

Назначение: обеспечить контроль над работоспособностью сервиса средствами Docker

1) Оператор Dockerfile

```
HEALTHCHECK CMD curl --fail http://localhost:5000/healthcheck || exit 1
```

2) Инструкция docker-compose

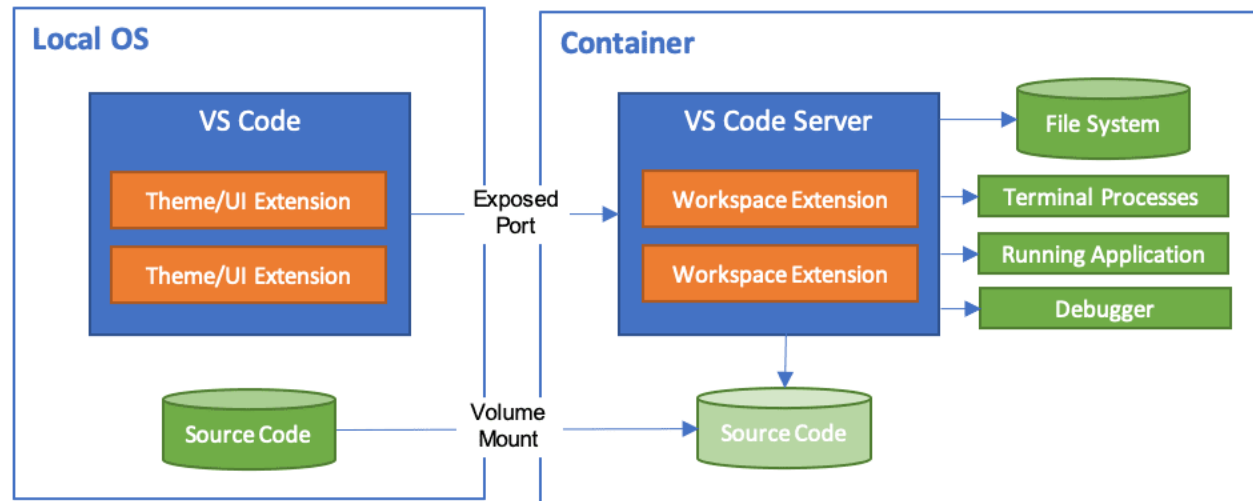
healthcheck:

```
test: ["CMD", "curl", "-f", "http://localhost"]  
interval: 1m30s  
timeout: 10s  
retries: 3
```

Privileged

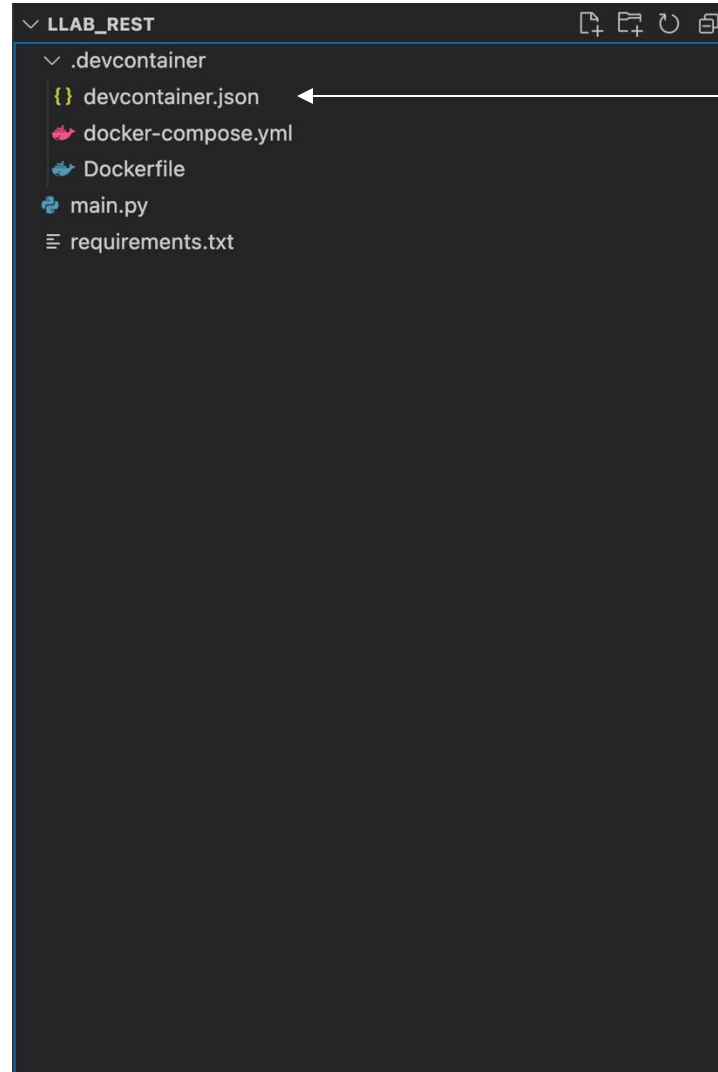
- `docker run --privileged`
- `docker run --cap-add=NET_ADMIN`
- Предоставляет доступ сравнимый с доступом обычного процесса

DevContainer



<https://code.visualstudio.com/docs/devcontainers/containers>

Структура каталога



Папка с именем ".devcontainer"
с файлом devcontainer.json

.devcontainer.json

```
{  
  "name": "My Workspace",  
  "dockerComposeFile": "docker-compose.yml",  
  "service": "app",  
  "workspaceFolder": "/workspace",  
  "settings": {  
    "terminal.integrated.shell.linux": "/bin/bash"  
  },  
  "extensions": [  
    "ms-python.python",  
    "ms-python.vscode-pylance"  
  ]  
}
```

docker-compose.yml

```
version: '3.3'

services:
  app:
    build:
      context: ..
      dockerfile: .devcontainer/Dockerfile
    networks:
      - my_network
    volumes:
      - ../workspace
    ports:
      - "8000:8000"

networks:
  my_network:
```

Dockerfile

```
# Используем официальный образ Python 3.11
FROM python:3.11-slim

# Устанавливаем рабочую директорию
WORKDIR /workspace

RUN apt-get update && apt-get install -y \
    build-essential \
    git \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Копируем зависимости
COPY requirements.txt .

# Устанавливаем зависимости
RUN pip install --no-cache-dir -r requirements.txt

# Копируем остальные файлы проекта
COPY . .

# Открываем порт 8000 для FastAPI
EXPOSE 8000

# Команда по умолчанию для запуска приложения
ENTRYPOINT ["/bin/sh", "-c", "while true; do sleep 1000; done"]
```


Что
почитать

Docker Deep Dive

Zero to Docker in a single book

May 2020



By Docker Captain
Nigel Poulton

На сегодня все

ddzuba@yandex.ru