



# Контейнеризация: зачем?

Нужна изоляция приложений друг от друга

Возможность поднять что-то быстро

Неизменяемость результата

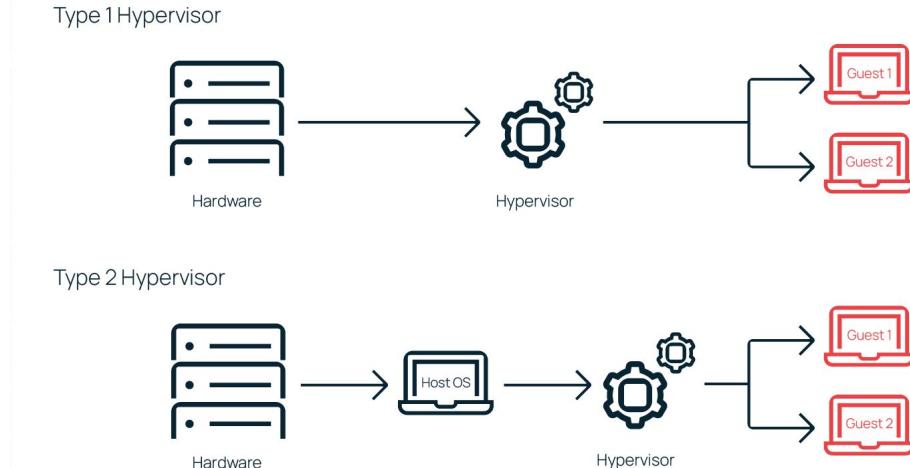


# Но до этого были виртуалки

Запускаем **полнценную машинку**,  
которая думает что у нее есть прямой  
доступ до ресурсов хоста  
(еще все зависит от типа  
гипервизора)

**У нас есть свое ядро и  
операционная система.**

Всё это не зависит от OS хостовой  
машины (разве только от  
архитектуры процессора)



## Но тут есть минусы

- слишком много ресурсов потребляет один экземпляр
- хочется чего-то более легковесного и изолированного

# Контейнеризация

это технология позволяющая запускать процессы в изолированном окружении на **Ядре хоста** (но мы изолируем ресурсы которые может использовать процесс)

технология реализуется **только за счет механизмом ядра Linux:**

1. **Cgroups** (Ограничение количественное)
2. **Namespaces** (Область видимости процесса)

**Container Runtime** для запуска контейнеров:

1. Docker
2. containerd
3. CRI-O

# Основной подход использования docker

- Один контейнер - один процесс (сервис)
- Повторяемость сборки (указываем версии важных пакетов для единства сборки)

# Синтаксис Dockerfile

```
FROM ubuntu:24.04 as base
```

```
RUN echo 'Hello world'
```

```
COPY ./test /tmp/test-2
```

```
WORKDIR /tmp
```

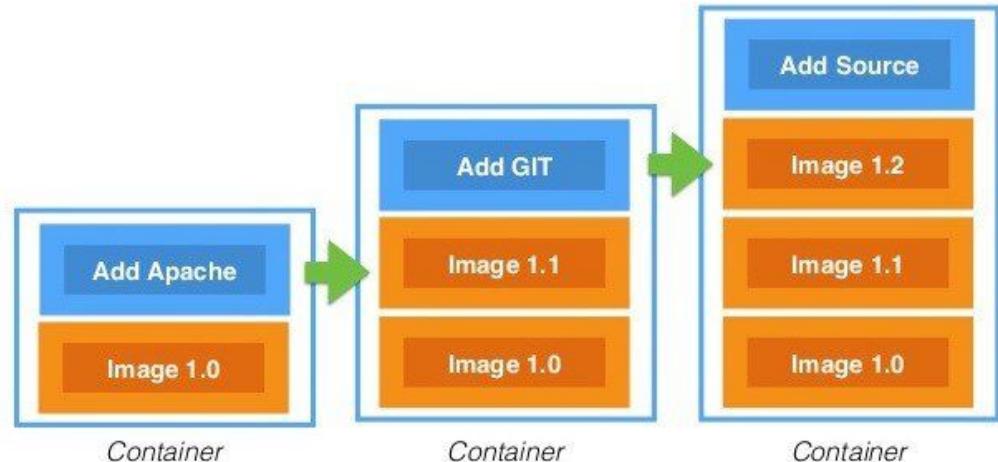
```
CMD ["bash"]
```

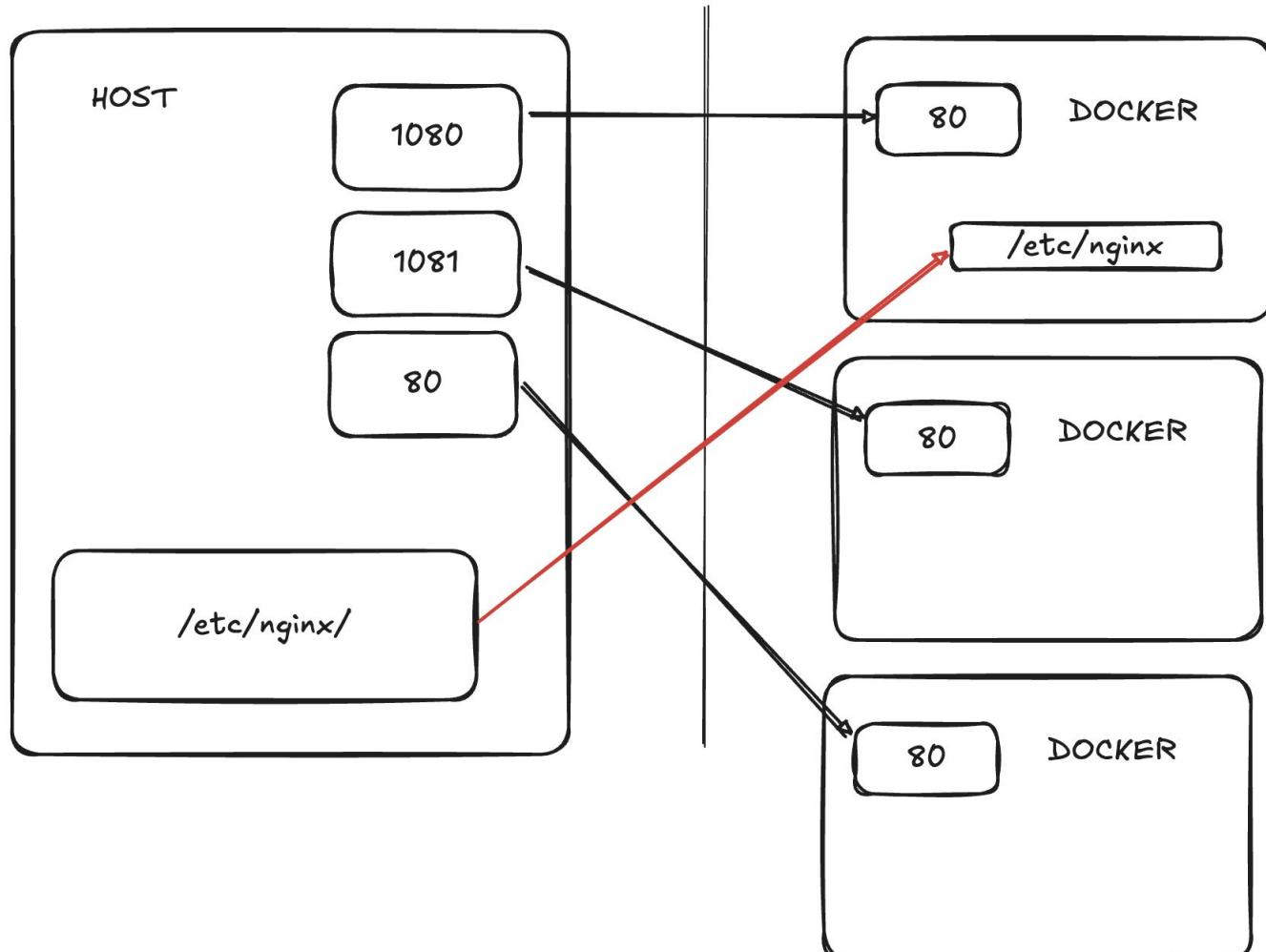
# Сборка докера

Докер собирается по слоям,  
каждая строчка в Dockerfile это  
новый слой

**Слои переиспользуются** при  
каждой сборке, поэтому надо  
выносить самые  
тяжелые/долгие операции  
наверх файла

AuFS  
*Layered Filesystem*





# Сущности docker?

- image - образ из которого запускается контейнер (грубо говоря как снапшот виртуалки)
- volume - дисковое пространство которое можно поделить между хостом и контейнером
- network - выделенная сеть под контейнер/группу контейнеров
- ports - у каждого контейнера есть свои порты, но чтобы можно было обратиться к ним из своего хоста нужно их пробросить через флаг -p

# Практика

1. Установить Docker на linux
2. Запустить hello world образ
3. Запустить любой nginx образ в docker
4. Поменять порт на котором принимает соединение nginx
5. Научиться работать с volume в докере:
  - a. Во время запуска контейнера подложить конфигурационный файл
  - b. Смонтировать папку на своем хосте в папку с логами внутри контейнера  
Надо сделать так, чтобы можно было не заходя в контейнер видеть логи
6. Собрать свой nginx на основе debian/alpine/ubuntu образов
7. docker-compose - сделать все предыдущее через docker-compose
8. Поднять 2 контейнера (1 nginx + 1 ubuntu)
9. Создать отдельную сеть в compose и поселить контейнеры в эту сеть
10. Собрать контейнер для flask приложения на питоне и поднять рядом nginx чтобы он проксировал запросы в flask

# Сам фласк

```
from flask import Flask  
  
app = Flask(__name__)  
  
@app.route('/')  
  
def hello_world():  
  
    return 'Hello, World!', 200  
  
  
if __name__ == '__main__':  
  
    app.run(debug=True)
```

Как его запустить  
pip3 install flask  
python3 main.py

## Part 2: docker compose

Полезные команды при работе с докером:

docker pull

docker build <в какой папке билдить> –tag docker-image:tag

docker push docker-image:tag

docker ps (docker ps -a)

docker run -it docker-image:tag some-command

docker exec -it docker-container-id some-command (подключаемся к рабочему контейнеру)

docker run -p 10808:80 -i --tag nginx:latest

# docker-compose

Что это такое - обертка над docker чтобы в виде конфигурационного файла описать сколько запустить контейнеров

services:

nginx:

image: nginx:stable

ports:

- 8080:80

volumes:

- ./nginx/nginx.conf:/etc/nginx/nginx.conf

- ./nginx/image.png:/tmp/minicat.png

bubuntu:

image: ubuntu:20.04

command: sleep infinity