DevOps, Docker and Gitlab-CI

Part 2: Docker

Version 1.1.0 (2023-02-15)

CRI

CENTRE DES RESSOURCES INFORMATIQUES

-- Cyril zarak Duval, root CRI/ACU 2020

Docker with an example

Let’s have a look first at the top level

Quick docker overlook

- I want to run a webserver quickly
- I don't really know in details any
- I don't want to mess with the things installed on my computer
  - Libraries, general packages …
- I just need it for some time and then forget about it
- Maybe I'll need it again in some months

With docker

What are Docker and containers ?

Let’s try to understand with more or less details

What is Docker ?

- Docker is a container engine
- It allows you to:
  - Create images
  - Start containers from those images
  - Manage containers
  - Exchange images

What are containers and images ?

- A container is “kinda” like a virtual machine
- A container is not a universal definition
  - We’re talking about linux containers in this course
- A container is essentially a process (and its sub-processes if any) which is isolated
- A container is ephemeral by design
- An image (in docker/OCI) is the source of a container
  - From an image you can create multiple containers
  - Each container is created from an image
  - See the relation like class/object in OOP

Container vs VM

- VM uses CPU mechanisms (+ bits of hypervisor)
- VM needs its own kernel
- VM can be of different architecture (x86, ARM, RISC-V, …)
  - Virtualization, paravirtualization, emulation
- Host (hypervisor) doesn't have much access in the VM
  - i.e. can't see natively its process, load, etc
- Container is simply a linux process isolated with kernel mechanisms
- Host has full access on the container

Container vs VM

- VM needs to be setup with RAM amount, CPU count, disk, etc
- Container is a process. You can limit resources but not mandatory
- Containers are lighter:
  - No kernel
  - Faster to start
  - Can even run without an OS
- Containers are less secure
- Containers can't run everything (i.e. no windows on linux)
- Containers are ephemeral by nature

Container vs regular process

- What makes a process a container ?
- Isolation of:
  - Filesystem
  - Other running processes
  - Users
  - And also: network, mountpoints, UTS (hostname), …
- Can also have limitations (CPU, memory, etc)
- No clear way of identification
  - No “container id” or anything provided by the kernel

Why do even need containers ?

Don’ t only take my word, but there are useful

Why do we use containers ?

- Control your OS (it's in the image)
  - No dependency issue from a laptop to a server: everything is in the image
  - Can have multiple libs in parallel (in different images)
- Common interface to build and run applications
- Share easily the images
  - The app and all its dependencies
- Version control
- Isolation

Why do we use containers ?

- ➔ Cheap
  - ◆ Quick to build
  - ◆ Quick to start
  - ◆ No overhead (unlike VMs)
- ➔ No difference between your laptop, dev server and prod server
- ➔ Follow the 12 factors principles

How does Docker work ?

Let’s have a look at the daemon and the CLI

How does Docker work ?

- Docker works with a daemon: dockerd
- dockerd manages everything
- The user can contact dockerd in multiple ways
  - UNIX socket, TCP, …
- Most people use the Docker CLI client, the command docker
- The docker command will contact the docker daemon to execute the user inputed command

How does Docker work ?

- ➔ The Docker daemon manages running, stopped containers, but also images, volumes, etc…
- ➔ If the daemon is not running, or you don't have the permissions to contact it, you might get some error like
- ➔ Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/json": dial unix /var/run/docker.sock: connect: permission denied

Let's get started with docker

Creating containers

Docker CLI - run containers

- To run a container with docker, we use docker run
- To check for running containers, we use docker ps
- Let's check docker common operations with containers:
  - pull, start, stop, ps, image ls, exec

How to build docker images ?

Stop using and start creating

What is a docker image ?

- ➔ We said that docker containers are created from docker image
- ➔ Like an instance from a class, an object and a template
- ➔ What defines an image then ?
- ➔ What’s inside an image ?

What is a docker image ?

- A docker image is essentially a combination of a few things:
  - A filesystem
    - The “main” binary of the image
    - Its libraries, essential files, …
    - Some other binaries
      - Dependencies
      - Utilities
    - All sets of files deemed worthy of being shipped in the image

What is a docker image ?

- A docker image is essentially a combination of a few things:
  - A filesystem
  - Some metadatas :
    - How it was built
    - What commands to run by default
    - Some environment variable to set
    - ...

Dockerfile

- Docker images are built with a Dockerfile*
- It's a recipe-like config file
- It has multiple kind of instructions
  - FROM selects the docker image to start from
  - RUN let you run arbitrary shell commands
  - ...
    - More details to come with the practicum

* there are other ways not to be mentioned in this course

Dockerfile

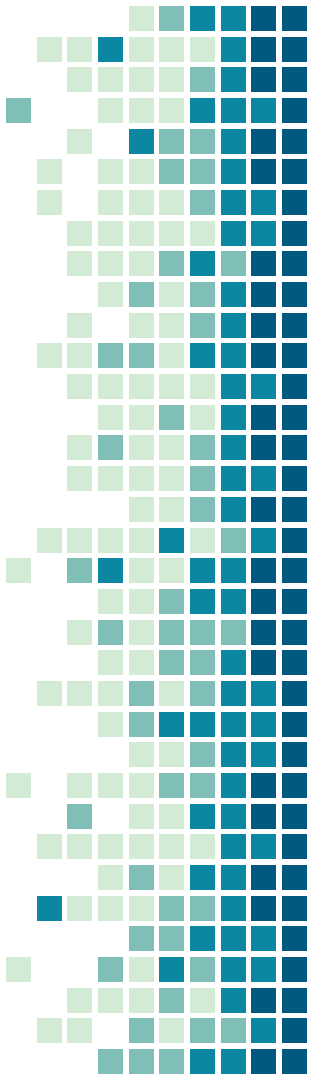
- A Dockerfile starts by a source docker image
  - FROM instruction
- Let's say you want to create an image based on debian
  - FROM debian:11
- Everything from the image stated in FROM will be imported
- The rest of the commands will create another image on top of the initial FROM

Dockerfile

- Each instruction will perform some modifications on the image
  - Add a file
  - Run a command
  - Set some variable
  - ....
- Once they are all successfully executed, a new image is built

Image, tags, repository

- A docker image is defined by a hash (sha256)
- But it's not convenient for most people
- So a name can be set on a hash for references purposes
- But because a name could have multiple version, we can append a tag
  - debian:11, python:3, nginx:alpine, …



Image, tags, repository

- To be shared, images need to have a name that includes a registry
- Default registry: docker.io
- Default directory: library
- When referencing an image debian:11, in fact is *real* name is docker.io/library/debian:11

Docker image and build workflow

Really make the difference between building and running

Workflow

1. Find a starting appropriate image for your project
2. Find a correct tag for the image
3. Write a Dockerfile that starts from said image
4. `docker build` to create the image
5. Check information about the image
6. Create one or multiple container(s) from said image with `docker run`

Having a look at containers mechanisms

What does it look like ?

Container isolation

- Isolation is done via 2 syscalls:
  - chroot(2)
  - namespaces(7)
- chroot:
  - Change the root directory for a process
  - Prevent the process from accessing anything not in its root
  - Example

Container isolation - chroot

- Changing a process root directory means preventing it from accessing host libraries
  - /usr/lib for example might be needed and then provided
- A good way to control installed libraries and their version
- Needs to provide an “OS” in the chrooted directory
  - Needed binaries, libs, FHS, …
  - Tends to make a container VM-ish

Container isolation – namespaces

- Other syscall namespaces(7)
- Create a namespace of a kind for a process (and its children)
- Kind of namespace:
  - Network
  - Mount
  - PID
  - User
  - …
- Hierarchical approach

Container isolation – namespaces

- ➔ Example of network namespace
- ➔ Example of PID (and user) namespace

Container limitation

- A container shall be limitable
- Like VM : allow max resources
  - Avoid CPU burst, OOM, …
- Linux mechanism: cgroups

Cgroups (v2)

- Linux mechanism to add process in a control group
- Control groups allow to set limits on various resources
  - Limits are hierarchical, a sub cgroup cannot exceed its parent limits
- 2 versions of cgroups:
  - v2 used on modern systems
  - v1 still widely used
- Exposed as a pseudo filesystem
  - Check mount(1) output

Cgroups (v2)

➔ Cgroups example with cpuset cgroup

Container isolation – how to share ?

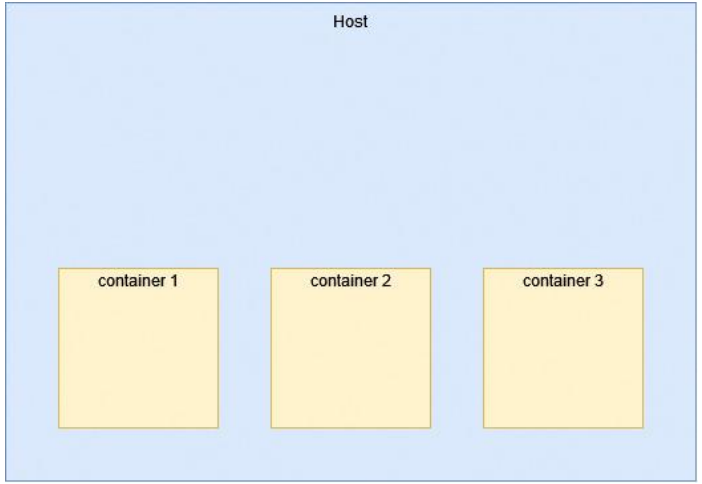
- What if you need to share a directory ?
  - Ephemeral containers aren't suitable for persistent data
- What if your container must be network accessible ?
- Docker offers way to share resources
  - Let's have a look at its CLI

Understand implications of such isolation through network

Maybe a schema will help ?

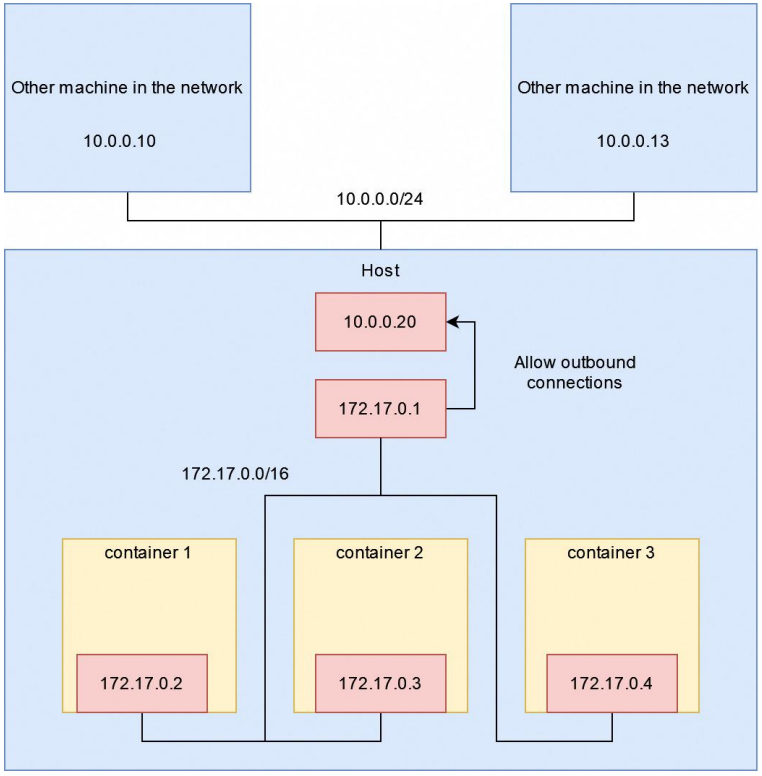
Container isolation boxes

- Most important part of container isolation to understand is the box model
- The host is the bigger box, and contains the rest
- Asymmetrical relation



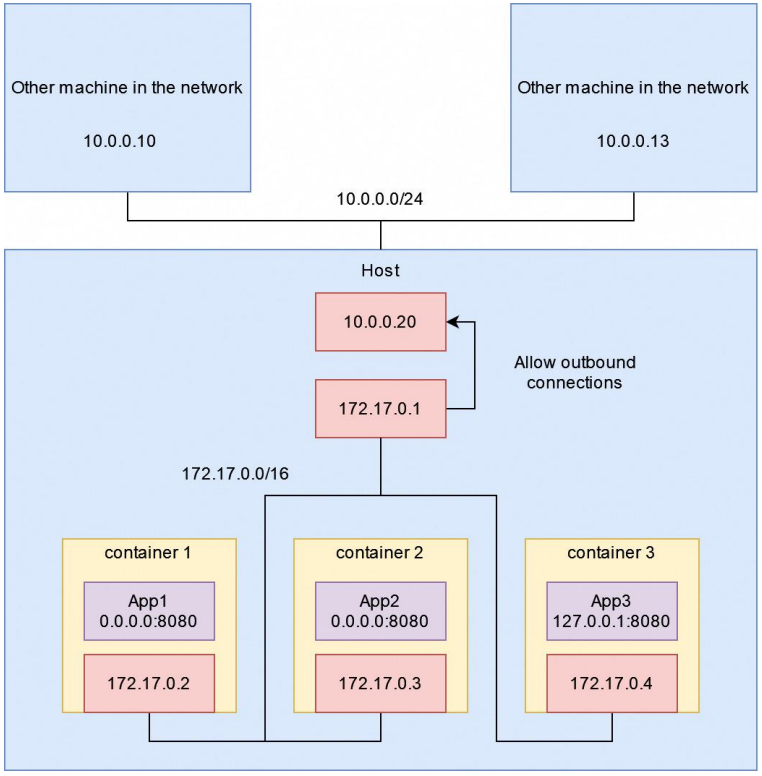
Container isolation boxes – network

- For network aspect, docker creates a subnet by default
- Each container is put in this default subnet
- Allow to access internet
  - But by default not accessible from outside
- Can communicate with each others



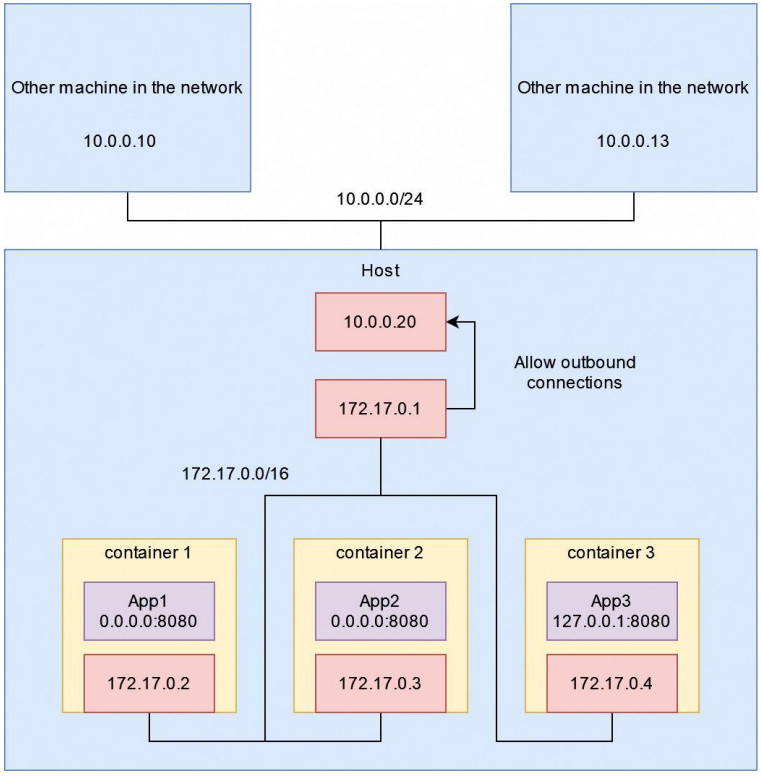
Container isolation boxes – network

- Let's have some network services listening and awaiting connections: App1, 2 & 3
- Listening on IP:port
  - 0.0.0.0:8080
  - 0.0.0.0:8080
  - 127.0.0.1:8080
- Listening on IP 0.0.0.0 means listening on all IP addresses



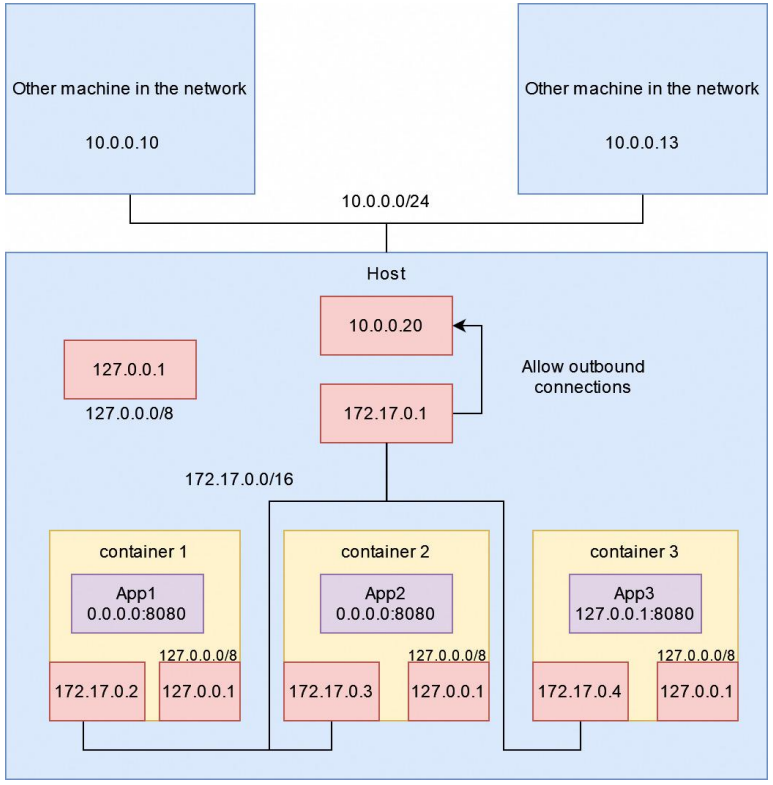
Container isolation boxes – network

- App1 and App2 don't step on each others toes
  - Different containers
  - Different IP addresses
  - They can both listen on port 8080
- Can 10.0.0.10 reach 172.17.0.2:8080 ?
- Can the host reach App3 ?
- Can container1 reach App3 ?



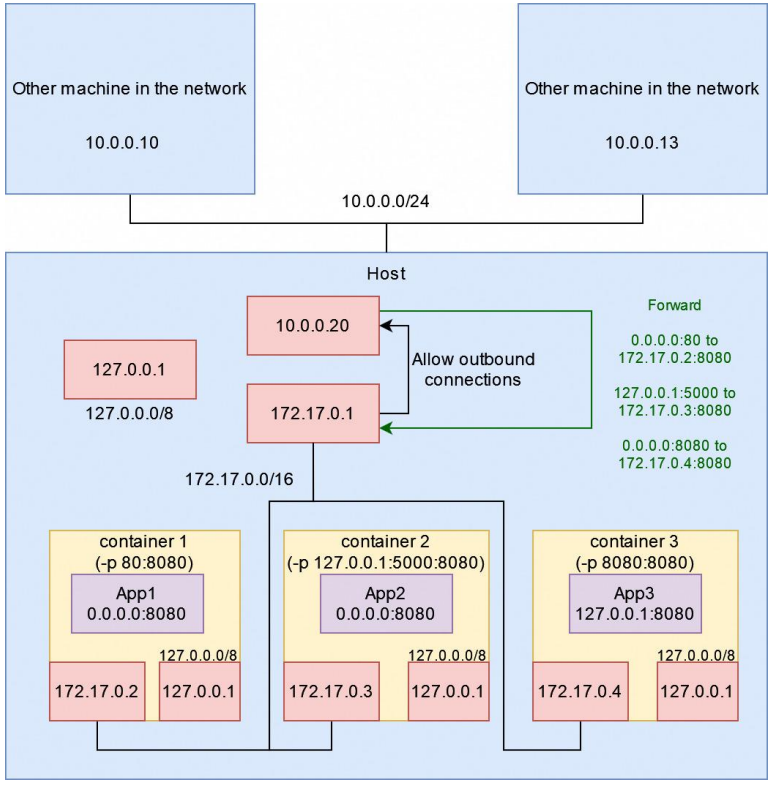
Container isolation boxes – network

- “En fait l’histoire est plus complexe”
- Each container and the host have their own localhost (127.0.0.0/8) subnet
- To expose App1 (or App2) publicly, a link must be made between 10.0.0.20 and 172.17.0.2 (172.17.0.3)



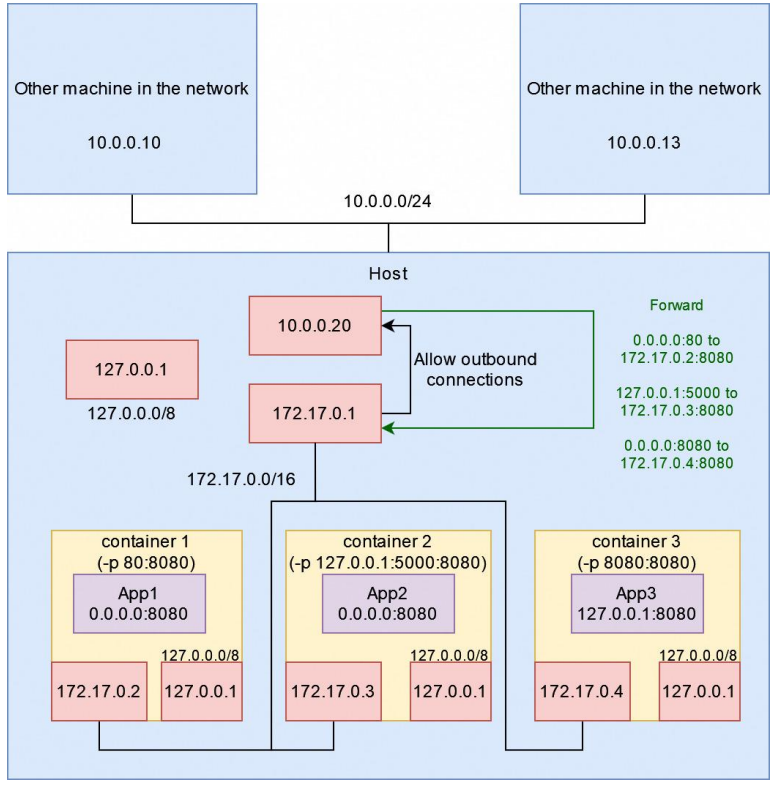
Container isolation boxes – network

- Running the containers with -p to expose ports (docker run -p) allow external connections and mapping
- App1 is reachable from 10.0.0.10 on 10.0.0.20:80
- App1 is reachable from Host on 127.0.0.1:80, 10.0.0.20:80 or 172.17.0.2:8080



Container isolation boxes – network

- App2 is not reachable from 10.0.0.10
- App2 is reachable from Host on 127.0.0.1:5000 or 172.17.0.3:8080
- App3 is reachable only* from container 3 on 127.0.0.1:8080



* actually can be reached from the Host by tricking quite a bit, but not covered by the course

A word about overlayfs

Understand this to build better images

Overlayfs

- Docker uses overlayfs to assemble images
  - Also to run containers on top of an image
- Overlayfs is interesting and a bit complex
  - Won't go into details here
  - Basically uses layers
    - A layer contains all the files changed at a step
    - An image is built with multiple steps = multiple layers
    - A container adds a final layer on an image: the runtime diffs

Overlayfs

- Overlayfs layers
  - A layer contains all the files changed at a step
  - An image is built with multiple steps = multiple layers
- Many steps = Many layers
  - It is preferable to reduce as much as possible
  - Example:
    - RUN apt-get install -y vim
      RUN echo “syntax on” > ~/.vimrc
      ->
      RUN apt-get install -y vim && echo “syntax on” > ~/.vimrc

Overlayfs

- A layer that add a 1GiB file and layer that removes it after = 1GiB still
- A layer that both adds & removes = ~no space taken
  - Important to `apt-get install` and remove cache in the same layer
- 2 Images with common instructions creates the same layers
  - Until they diverge
  - Important to put the common instructions first
  - Then packages installation (heavy)
  - Then image-specific things

Overlayfs

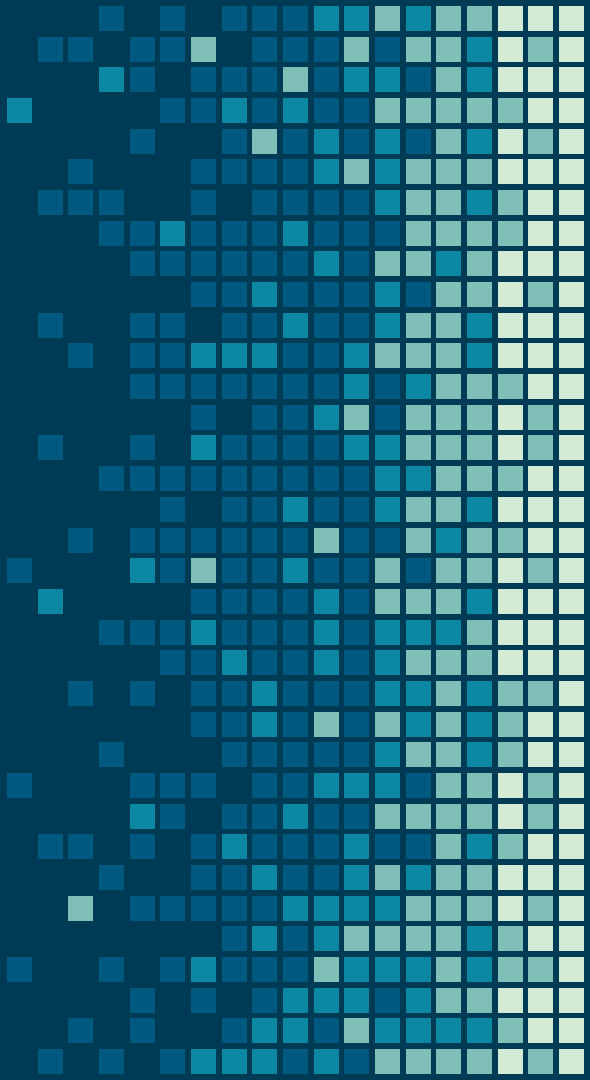
- You can see layers when building images
  - They are designated by a hash
- When pulling
- With docker inspect
- With mount if a container is running

```
$ docker build -t myapp:mytag .
Step 1/7 : FROM python:alpine
 ---> 2c167788a673
Step 2/7 : WORKDIR /app
 ---> Using cache
 ---> 8a9f6f64de7f
Step 3/7 : RUN addgroup -S app && adduser --disabled-password -s /bin/bash -h /app -u 1000 -G app app
 ---> d0e9a3442050
 ...
```

Overlayfs

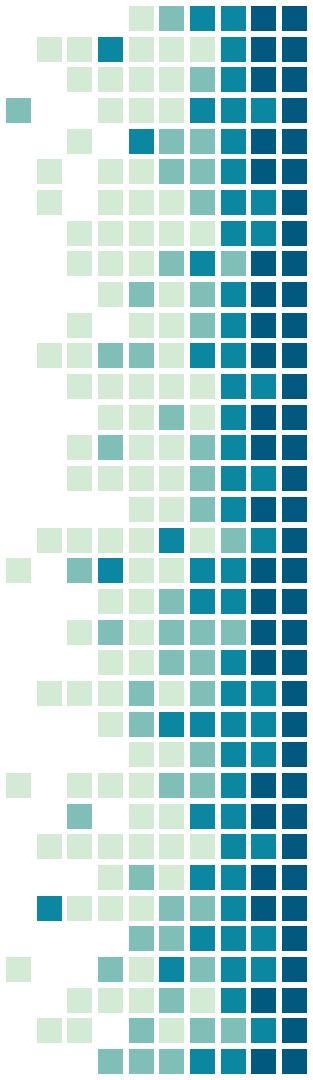
```
$ $ mount | grep overlay
overlay on /var/lib/docker/overlay2/bc0be1d523c88451cf206a5732fed96acfa13ee7490ee7a0a351c22aa1de485e/merged type overlay
(rw,relatime,lowerdir=/var/lib/docker/overlay2/l/SHAS447KYIHIXRWRQTKYVJVRBJ:/var/lib/docker/overlay2
/l/SUMUGGHAKNUNWL3TFGCTR2JO4F:/var/lib/docker/overlay2/l/2CO7DC6CPJHYLYEZWAFQUJEDT5:/var/lib/docker/overlay2
/l/KEURPHRWY6XCRTNJAQPIKQ4EDO:/var/lib/docker/overlay2/l/F2BLMEBFX7C5TRX7VBP7N2RRJC:/var/lib/docker/overlay2
/l/2KSTHP277I7OR76EQ3N5NGHQZ4:/var/lib/docker/overlay2/l/EDUXZYJYT2C23ZWT5WU6F6UICP:/var/lib/docker/overlay2
/l/CN6SZSH7Z6P7ODPNRFHZS7XIE7:/var/lib/docker/overlay2/l/YEAGNRGEUB7LTM7W7GQV7KJNPR:/var/lib/docker/overlay2
/l/HUS5KLUSFULIF4JLRA2T4MWALN,upperdir=/var/lib/docker/overlay2
/bc0be1d523c88451cf206a5732fed96acfa13ee7490ee7a0a351c22aa1de485e/diff,workdir=/var/lib/docker/overlay2
/bc0be1d523c88451cf206a5732fed96acfa13ee7490ee7a0a351c22aa1de485e/work,index=off)
 $ docker inspect nginx:1.21 | jq '.[0].RootFS.Layers'
[
  "sha256:9c1b6dd6c1e6be9fdd2b1987783824670d3b0dd7ae8ad6f57dc3cea5739ac71e",
  "sha256:4b7fffa0f0a4a72b2f901c584c1d4ffb67cce7f033cc7969ee7713995c4d2610",
  "sha256:f5ab86d69014270bcf4d5ce819b9f5c882b35527924ffdd11fecf0fc0dde81a4",
  "sha256:c876aa251c80272eb01eec011d50650e1b8af494149696b80a606bbeccf03d68",
  "sha256:7046505147d7f3edbf7c50c02e697d5450a2eebe5119b62b7362b10662899d85",
  "sha256:b6812e8d56d65d296e21a639b786e7e793e8b969bd2b109fd172646ce5ebe951"
]
```

docker-compose



Docker cli limitations

- ➔ Docker cli is a bit limited for some cases
  - Lots of arguments
  - Needs to remember the arguments to restart/change/move the container
  - Can be considered config, but isn't in a config file
    - Against 12 factors

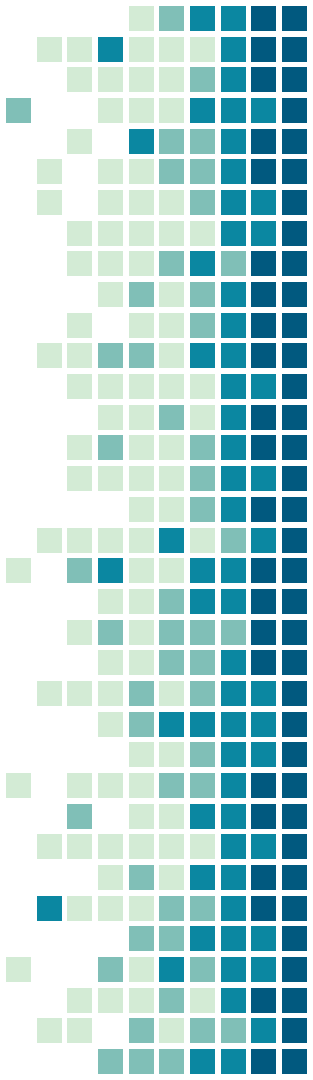


docker-compose

- ➔ Docker-compose translate a config file into docker commands
- ➔ Used to declare statically containers, networks, volumes, …
- ➔ YAML format
- ➔ docker-compose.yml

docker-compose

- ➔ docker-compose up to create and run the containers
- ➔ docker-compose down to stop and delete the containers
- ➔ docker-compose start/stop to start/stop the containers
- ➔ docker-compose logs to look at containers logs
- ➔ That's most of its CLI usage
- ➔ docker-compose is not a daemon, just a "translator" that reads YML to convert it to docker commands



About docker-compose.yml file

How to write this config file ?

```yaml
---

version: '3'

services:
  netbox:
    image: netboxcommunity/netbox:v3.0-ldap
    user: '101'
    depends_on:
      - postgres
      - redis
    env_file: netbox.env
    ports:
      - 80:8080
    volumes:
      - /srv/netbox/media:/opt/netbox/netbox/media
  postgres:
    image: postgres:13-alpine
    env_file: postgres.env
    volumes:
      - netbox-postgres-data:/var/lib/postgresql/data
    ports:
      - 5432:5432
  redis:
    image: redis:6-alpine
    command:
      - sh
      - -c # this is to evaluate the $REDIS_PASSWORD from the env
      - redis-server --appendonly yes --requirepass $$REDIS_PASSWORD ## $$ because of docker-compose
    env_file: redis.env
    volumes:
      - netbox-redis-data:/data
volumes:
  netbox-postgres-data: {}
  netbox-redis-data: {}
```

docker-compose.yml

➔ ~Equivalent to:

```
docker run --name postgres_1 --env-file postgres.env -p 5432:5432 -v netbox-postgres-data:/var/lib/postgresql/data
postgres:13-alpine && docker run --name redis_1 -v netbox-redis-data:/data --env-file redis.env --entrypoint sh redis:6-
alpine -c redis-server --appendonly yes --requirepass $REDIS_PASSWORD && docker run --name netbox_1 --env-file netbox.env
--user 101 -p 80:8080 -v /srv/netbox/media:/opt/netbox/netbox/media netboxcommunity/netbox:v3.0-ldap
```

Thanks !

Questions ?

Slides available on zarak.fr/

Contact: cyril@cri.epita.fr

zarak production#5492