

# IDVOC - Docker

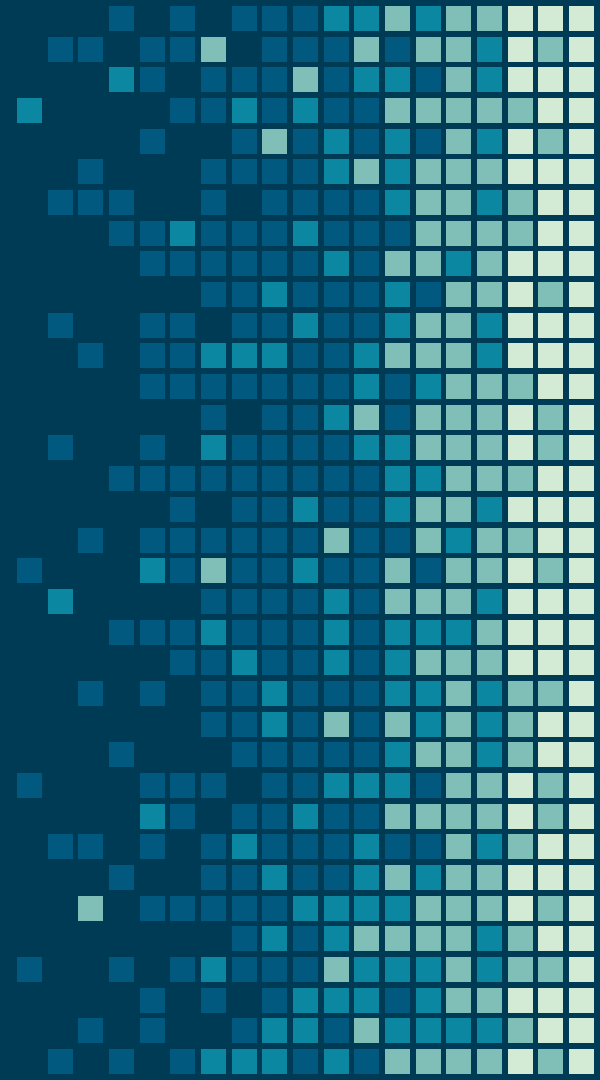


**CRI**  
CENTRE DES RESSOURCES INFORMATIQUES



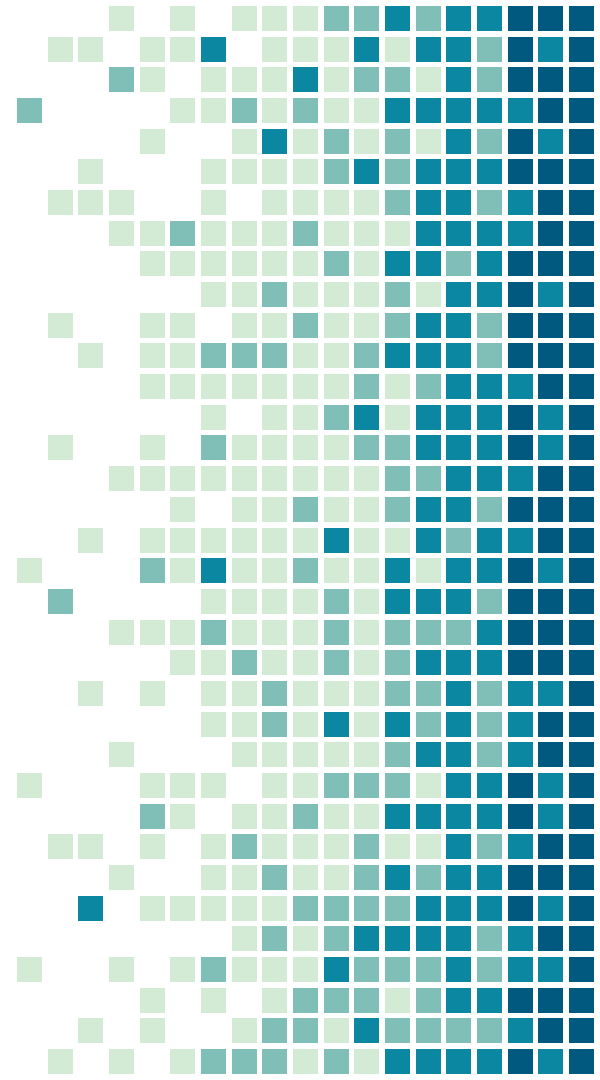
-- Cyril zarak Duval, root CRI/ACU 2020

Let's dive in  
Docker



# Docker with an example

Let's have a look first at the top level



# Quick docker overlook

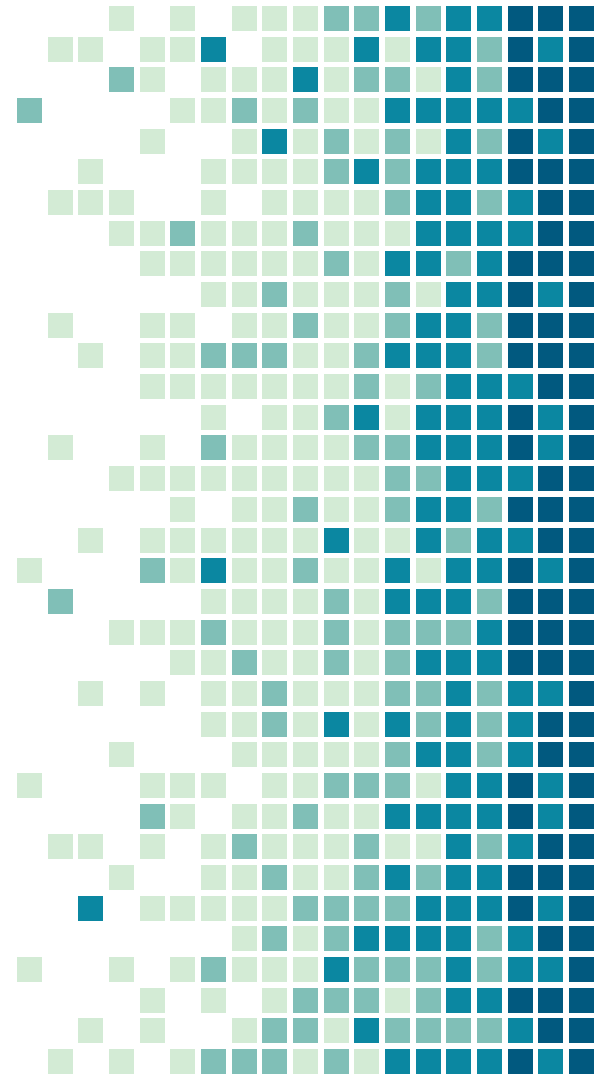
- I want to run a webserver quickly
- I don't really know in details any
- I don't want to mess with the things installed on my computer
  - ◆ Libraries, general packages ...
- I just need it for some time and then forget about it
- Maybe I'll need it again in some months

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



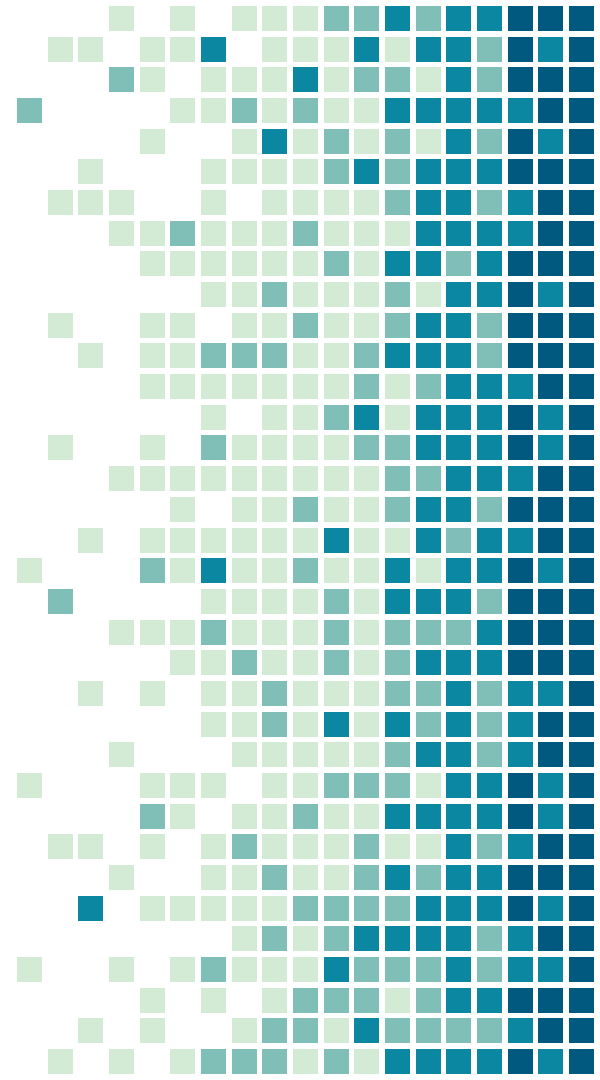
# Container vs regular process

- 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



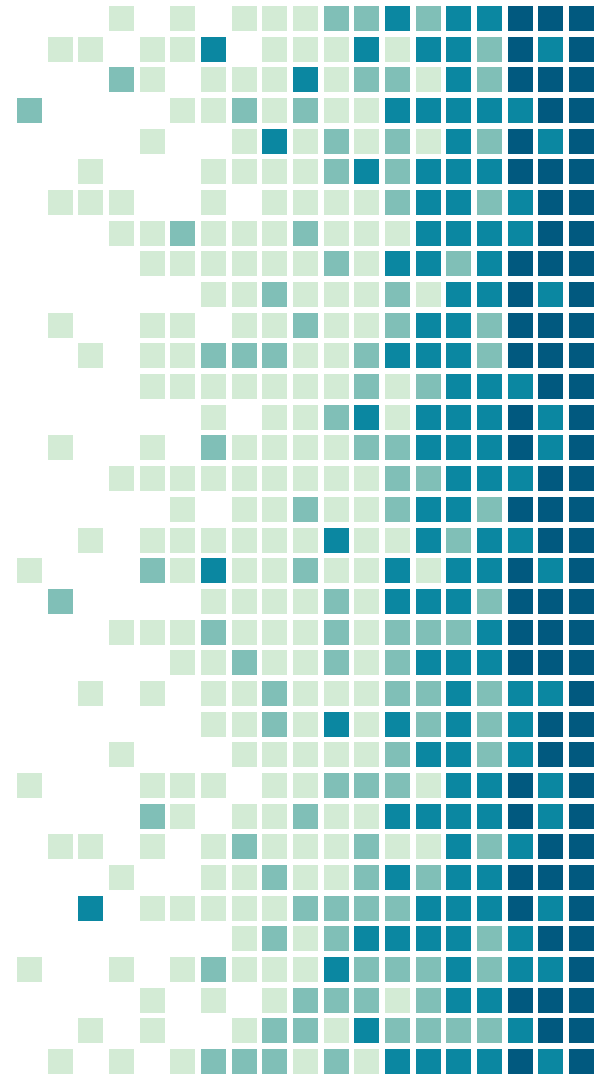
# Why do we use containers ?

- 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



# 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



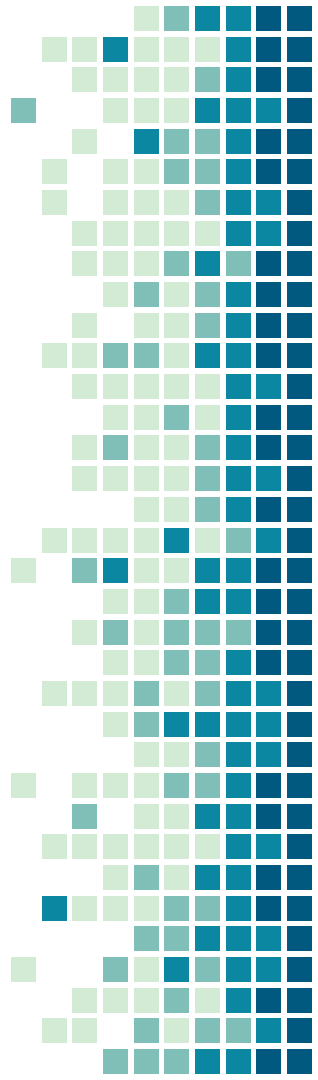
# Cgroups (v2)

→ [Cgroups example with cpuset cgroup](#)



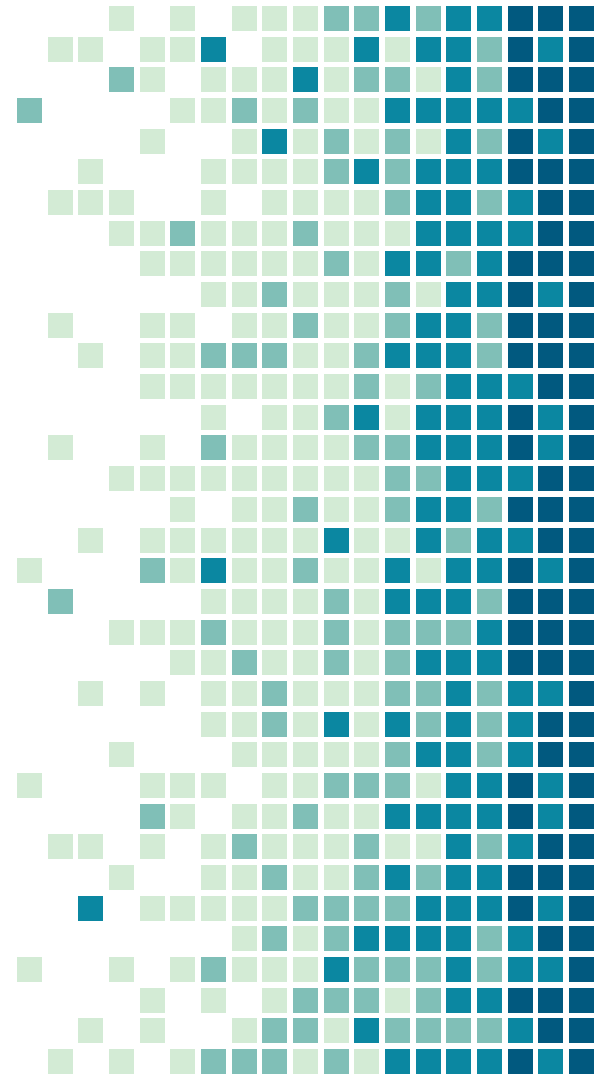
# Container isolation – how to share ?

- 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



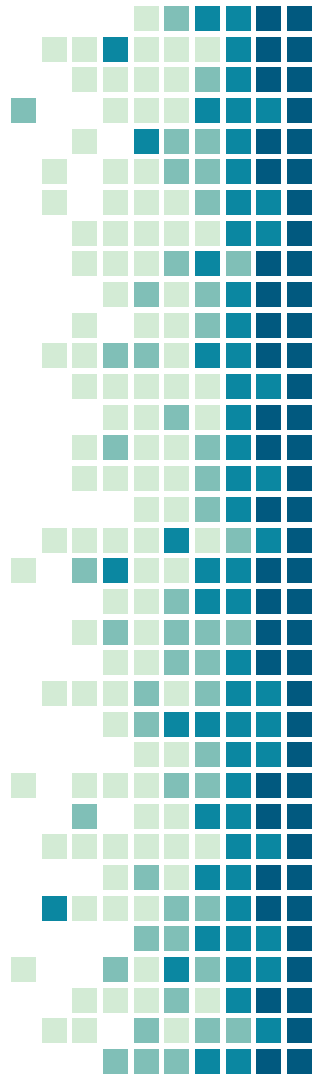
# How to run docker containers ?

The basics of docker CLI



# Docker CLI - run containers

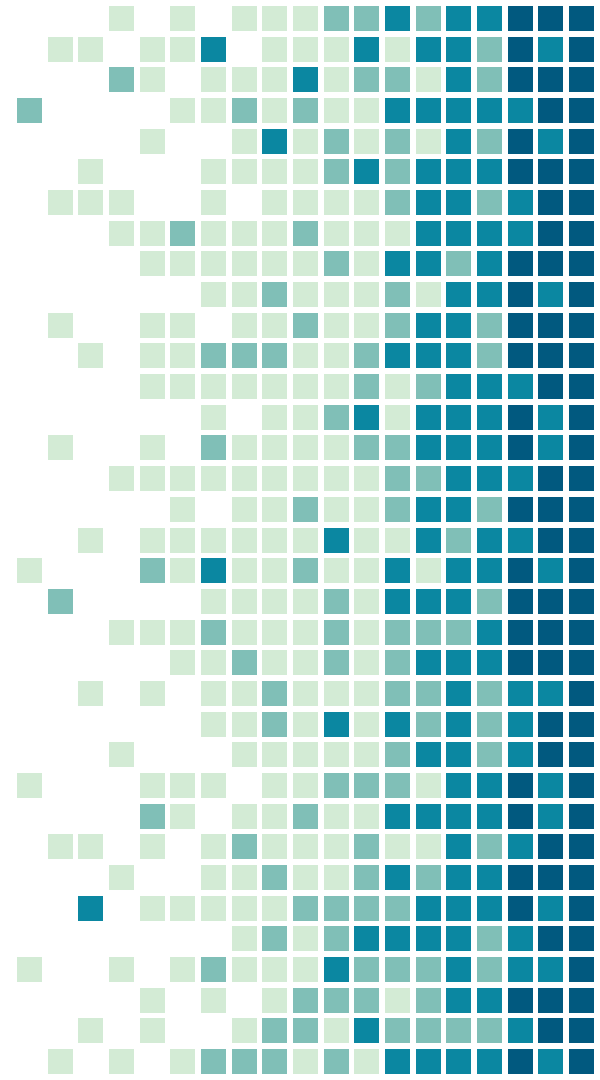
- To run a container with docker, we use `docker run`
- To check for running containers, we use `docker ps`
- Let's check docker common operations with containers:
  - ◆ [pull, start, stop, ps, image ls, exec](#)





# How to build docker images ?

Stop using and start creating



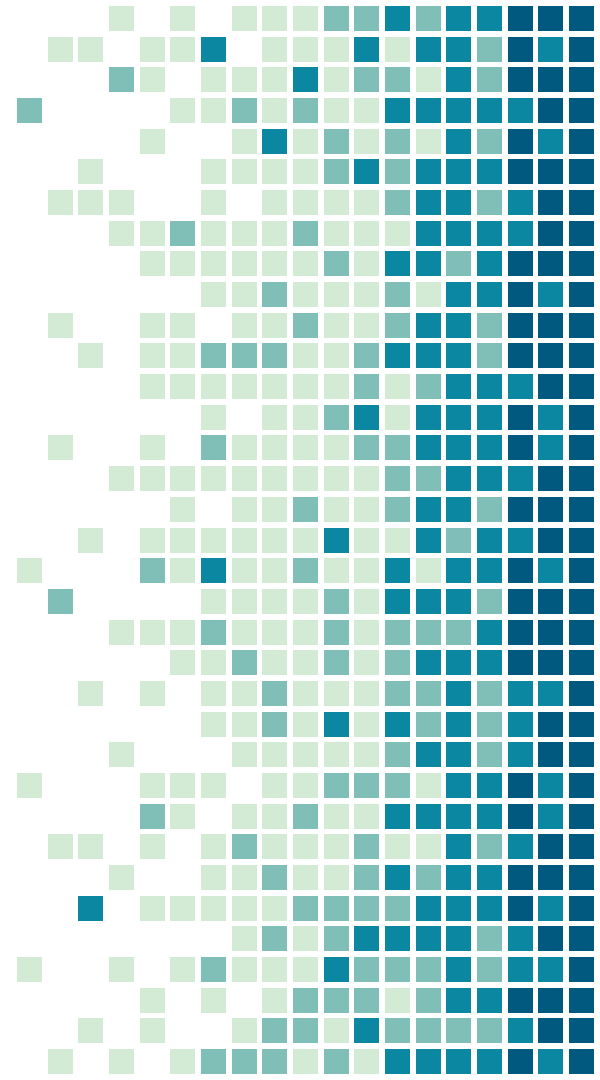
# Dockerfile

- 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



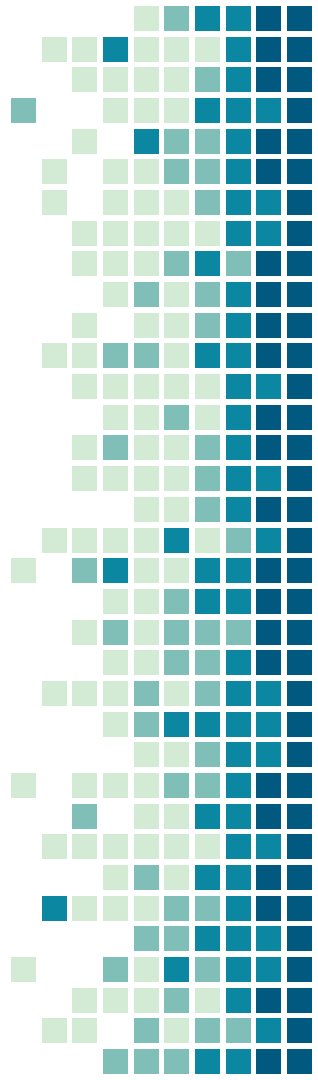
# A word about overlays

Understand this to build better images



# Overlays

- Docker uses overlays to assemble images
  - ◆ Also to run containers on top of an image
- Overlays 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



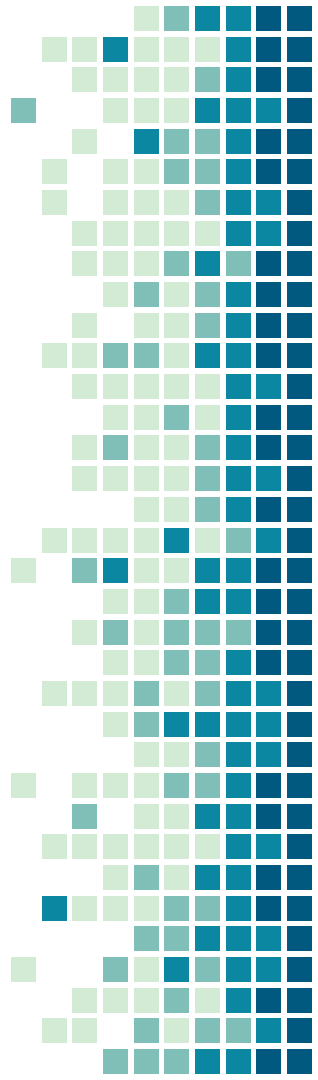
# Overlayfs

- 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



# Overlays

- 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



# Overlays

- 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

```
1 $ docker build -t myapp:mytag .
2 Step 1/7 : FROM python:alpine
3 ----> 2c167788a673
4 Step 2/7 : WORKDIR /app
5 ----> Using cache
6 ----> 8a9f6f64de7f
7 Step 3/7 : RUN addgroup -S app && adduser --disabled-password -s /bin/bash -h /app -u 1000 -G app app
8 ----> d0e9a3442050
9 ...
```

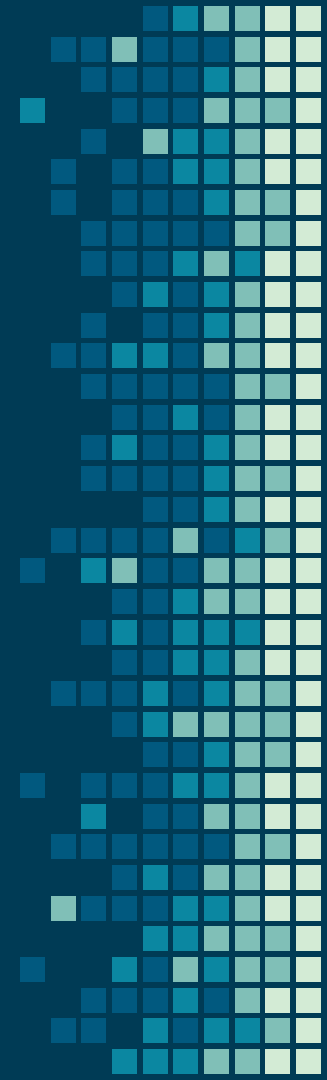
# Overlaysfs

```
1 $ $ mount | grep overlay
2 overlay on /var/lib/docker/overlay2/bc0be1d523c88451cf206a5732fed96acfa13ee7490ee7a0a351c22aa1de485e/merged type overlay
  (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/SHAS447KYIHIXRWRQTKYVJVRBJ:/var/lib/docker/overlay2
  /l/SUMUGGHAKNUNWL3TFGCTR2J04F:/var/lib/docker/overlay2/l/2C07DC6CPJHYLYEZWAFQUJEDT5:/var/lib/docker/overlay2
  /l/KEURPHRWY6XCRTNJAQPIKQ4ED0:/var/lib/docker/overlay2/l/F2BLMEBFX7C5TRX7VBP7N2RRJC:/var/lib/docker/overlay2
  /l/2KSTHP277I70R76EQ3N5NGHQZ4:/var/lib/docker/overlay2/l/EDUXZYJYT2C23ZWT5WU6F6UICP:/var/lib/docker/overlay2
  /l/CN6SZSH7Z6P70DPNRFHZS7XIE7:/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)
3 $ docker inspect nginx:1.21 | jq '.[0].RootFS.Layers'
4 [
5   "sha256:9c1b6dd6c1e6be9fdd2b1987783824670d3b0dd7ae8ad6f57dc3cea5739ac71e",
6   "sha256:4b7fffa0f0a4a72b2f901c584c1d4ffb67cce7f033cc7969ee7713995c4d2610",
7   "sha256:f5ab86d69014270bcf4d5ce819b9f5c882b35527924ffdd11fecf0fc0dde81a4",
8   "sha256:c876aa251c80272eb01eec011d50650e1b8af494149696b80a606bbeccf03d68",
9   "sha256:7046505147d7f3edb7c50c02e697d5450a2eebe5119b62b7362b10662899d85",
10  "sha256:b6812e8d56d65d296e21a639b786e7e793e8b969bd2b109fd172646ce5ebe951"
11 ]
```



# Thanks !

Questions ?



Slides available on [zarak.fr/](http://zarak.fr/)

Contact: [cyril@cri.epita.fr](mailto:cyril@cri.epita.fr)

[zarak production#5492](#)