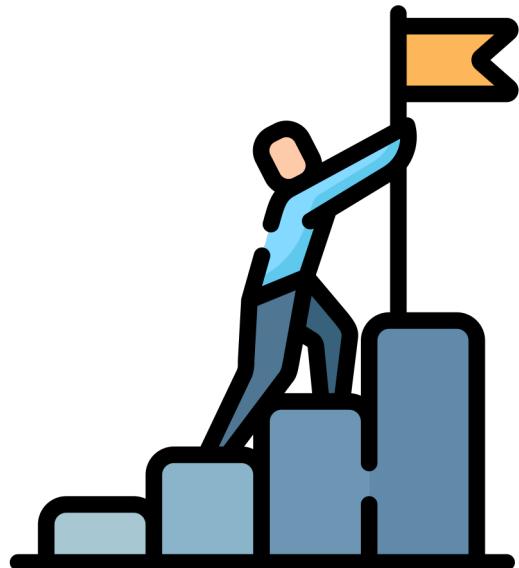




# Docker

# Objectifs

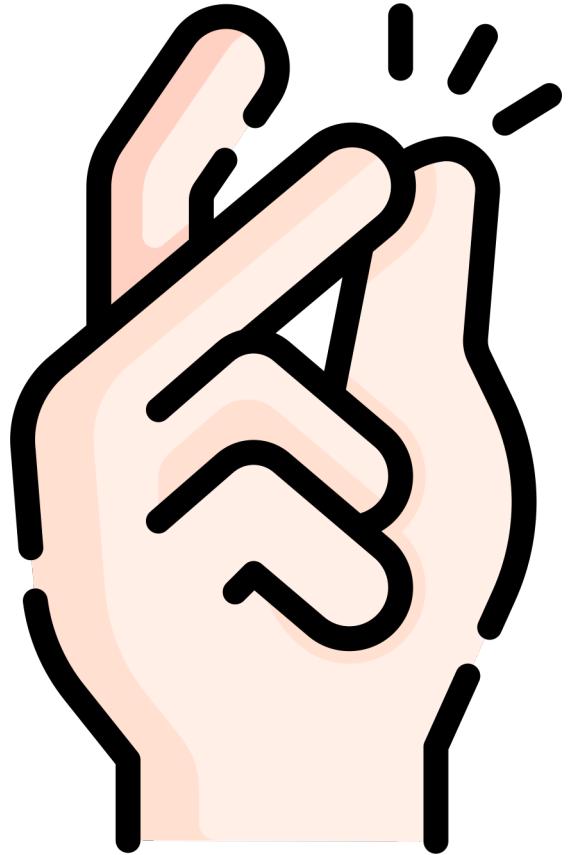


Concepts

Utilisation

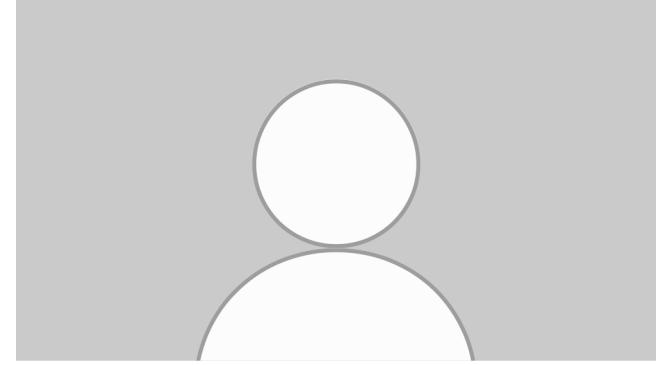
Fondamentaux techniques

Utilité

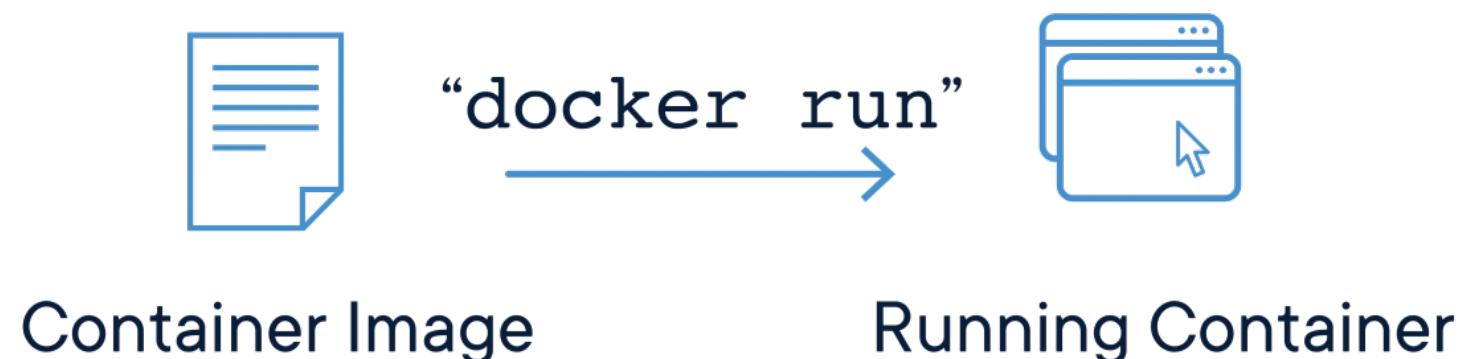


# Docker en quelques mots

# Qu'est ce que Docker ?



- Docker permet de « dev/deploy/run » des applications
- Docker une plateforme pour les dev et sysadmins (DevOps)
- Un container se lance a partir d'une image
- Une image contient code, binaire, bibliothèques, fichiers de config

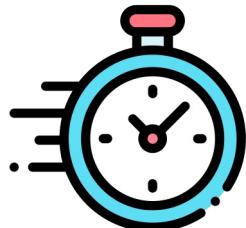


# Héberger un site web

```
$ docker run -p 8001:80 --rm nginxdemos/hello
```

Accessible en localhost:8001

Image à lancer



Rapide



Lisible/Documenté



Reproductible

# Démo : le container, un process isolé

Commade exécutée dans l'hôte

```
$ uname -a  
Darwin benoits-imac-5k.3ie.fr 20.3.0 Darwin Kernel Version 20.3.0: Thu Jan  
21 00:07:06 PST 2021; root:xnu-7195.81.3~1/RELEASE_X86_64 x86_64
```

Hôte sous mac

Commande à executer

Commande exécutée dans le Container

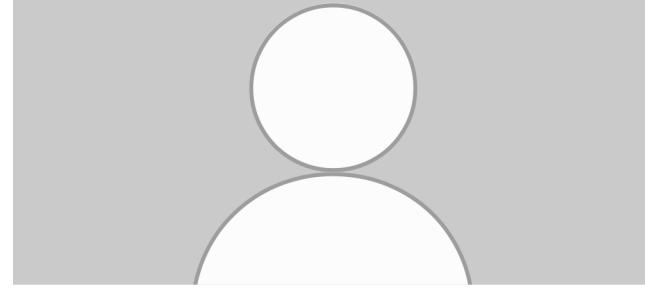
```
$ docker run --rm alpine uname -a  
Linux 3a08f6aa73e2 4.19.121-linuxkit #1 SMP Thu Jan 21 15:36:34 UTC 2021  
x86_64 Linux
```

Container sous Linux



# L'avant docker

# Contexte : les préoccupations du déploiement

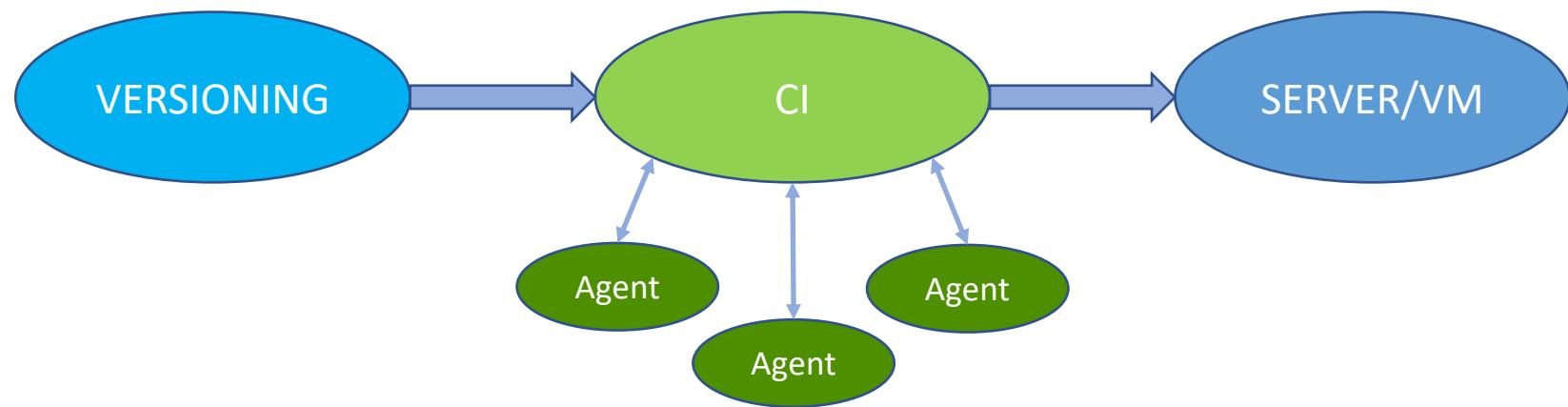


- Comment garantir que le code fonctionne ailleurs que chez le dev ?
- Comment déployer avec fiabilité ?
- Comment maximiser l'utilisation des infra ?
- Comment isoler les solutions ?



# Un début de réponse

## Les débuts de l'intégration continue

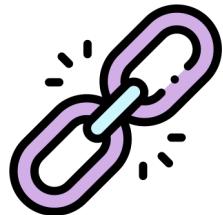


Mais aussi : Début des solutions **PaaS** + Emergence du **DevOps**

Rencontre entre le dev et l'opérationnel

Platform as a Service

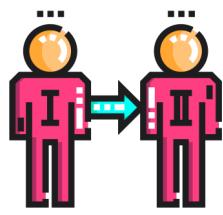
# Des regrets



Pas une bonne séparation entre infra et code

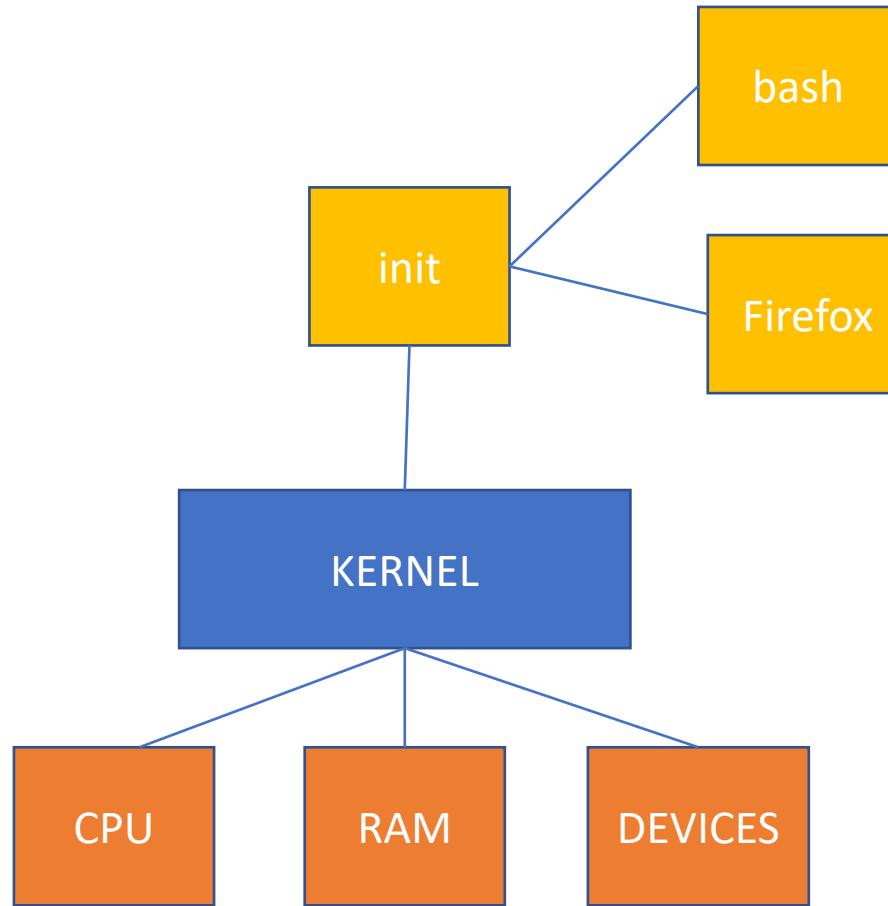


La VM consomme beaucoup de ressources

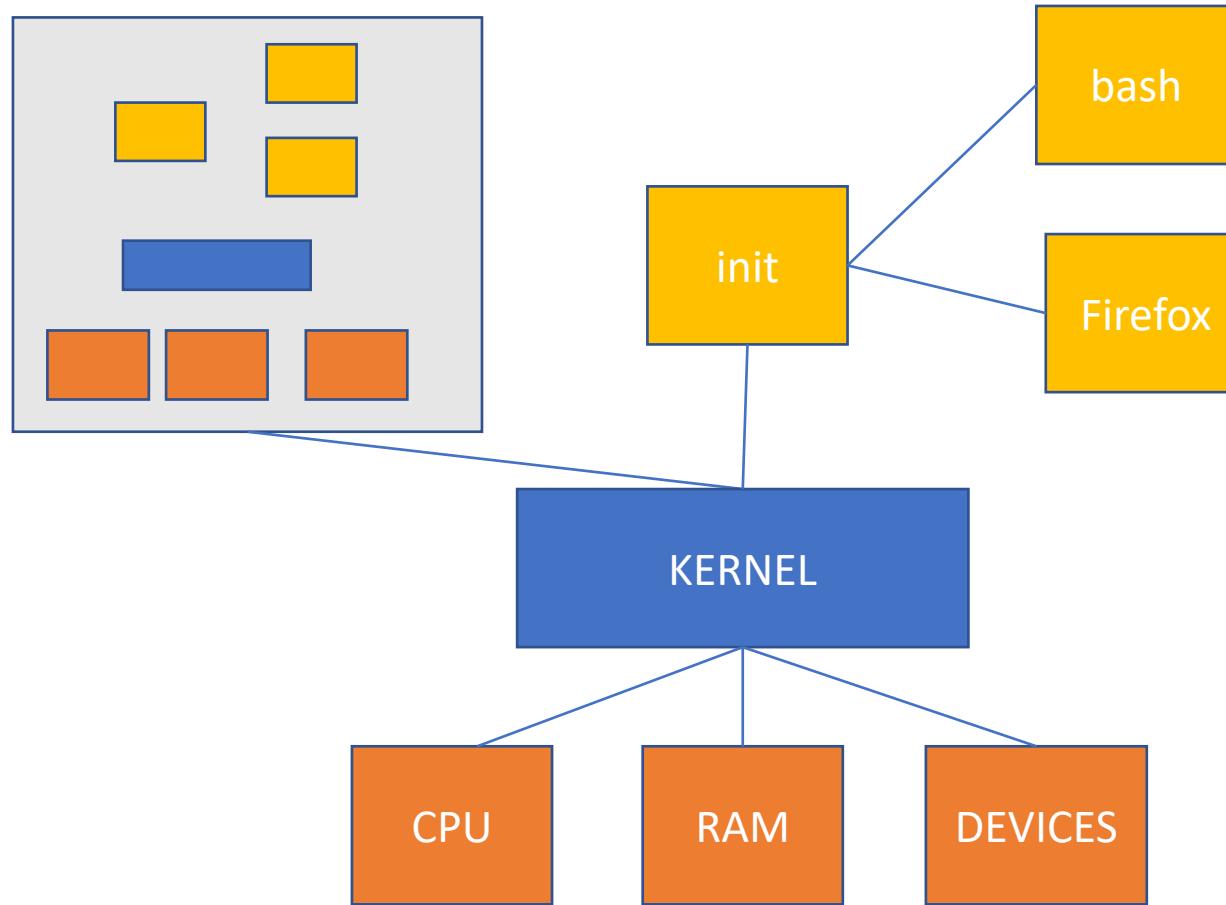


Difficile d'avoir le même environnement partout

# Rappel système : qu'est ce qu'une VM ?



# Rappel système : qu'est ce qu'une VM ?



# Inconvénients d'une VM



C'est lourd

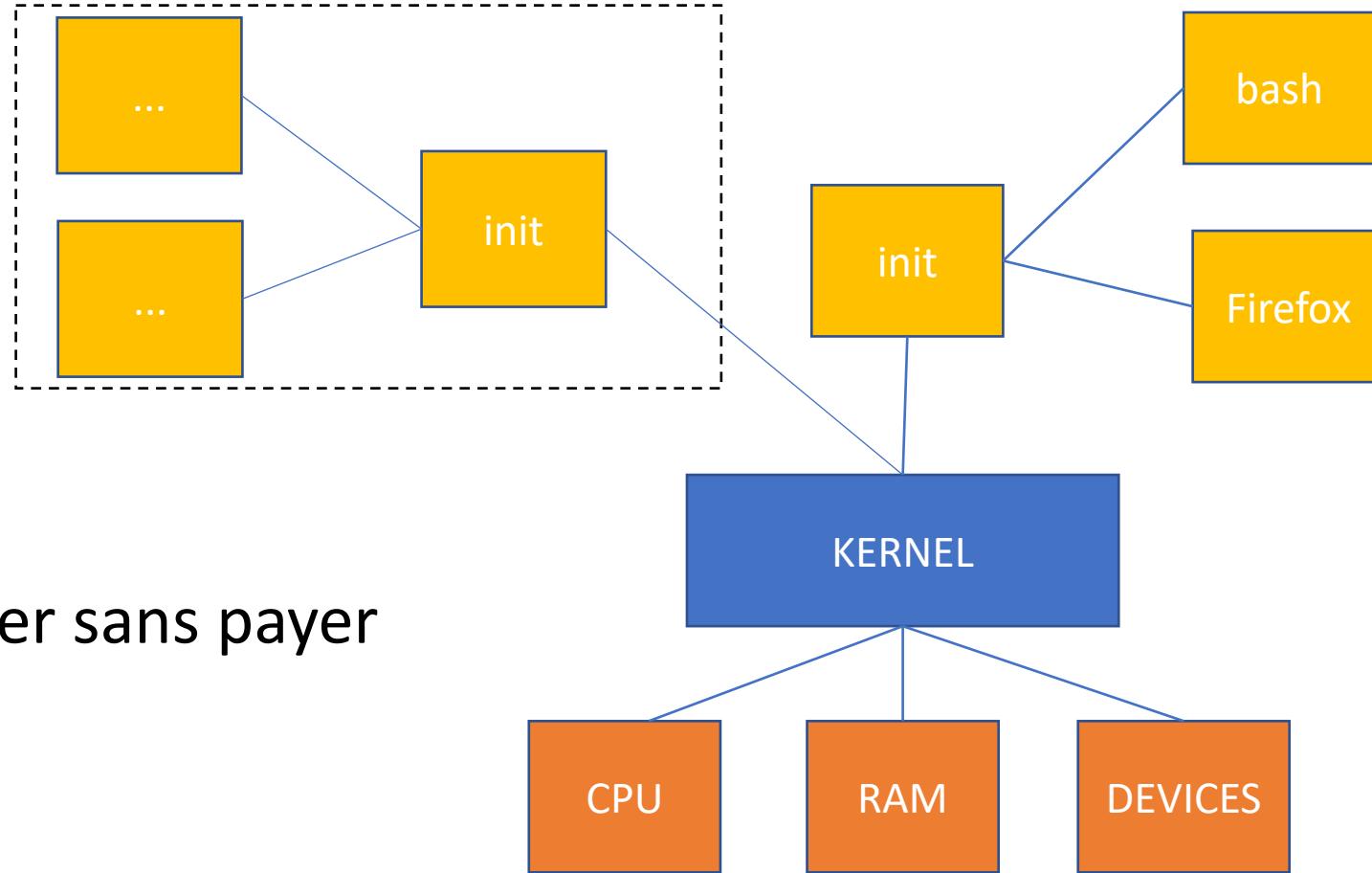


On paye le coût d'une virtualisation matérielle ...  
... alors que le Kernel nous protège déjà

Tout ce qu'on veux, c'est isoler les processus



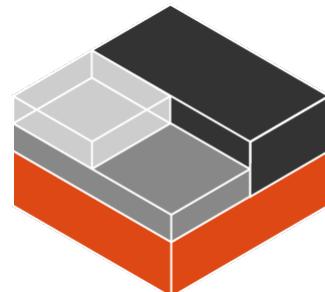
# Et si ...



Et si on pouvait isoler sans payer  
le cout de la VM ?

# Le conteneur système

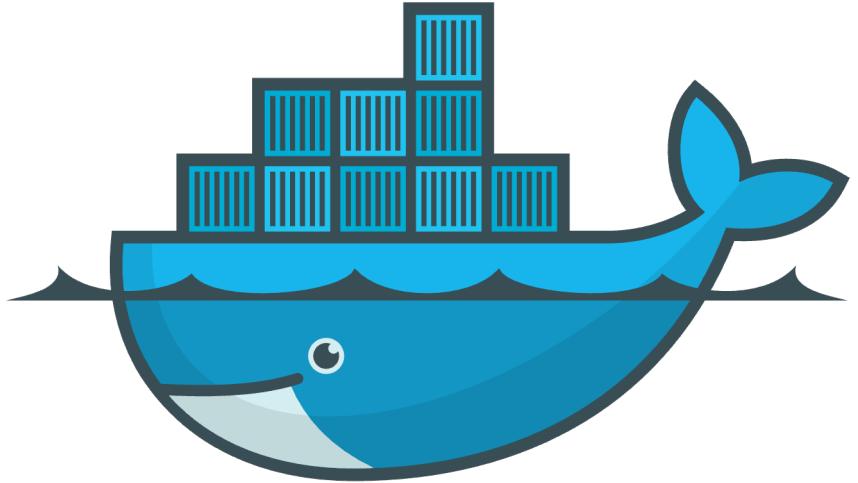
- « OS-level virtualization » => le kernel gère plusieurs user space
- Isolation des ressources matérielles et des processus
- Pas de hardware émulé
- Même OS que la machine hôte (pas de Windows sur un Linux)



LXC : Linux Containers

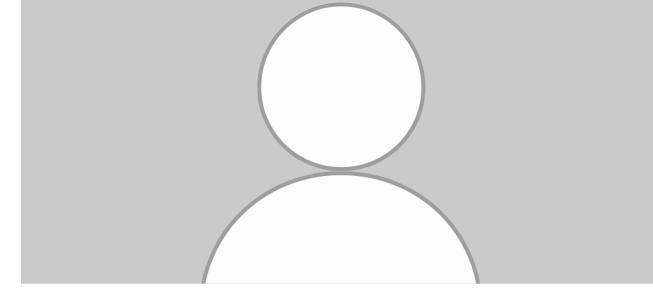
# Le conteneur applicatif





# Fondamentaux techniques de Docker

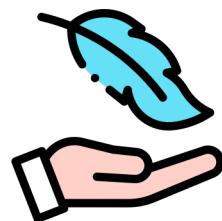
# Les bases techniques



Docker exécute le programme en 1<sup>er</sup> process  
(et non pas init)

C'est des Container applicatif avec les même fondamentaux que LXC  
(LXC = container système)

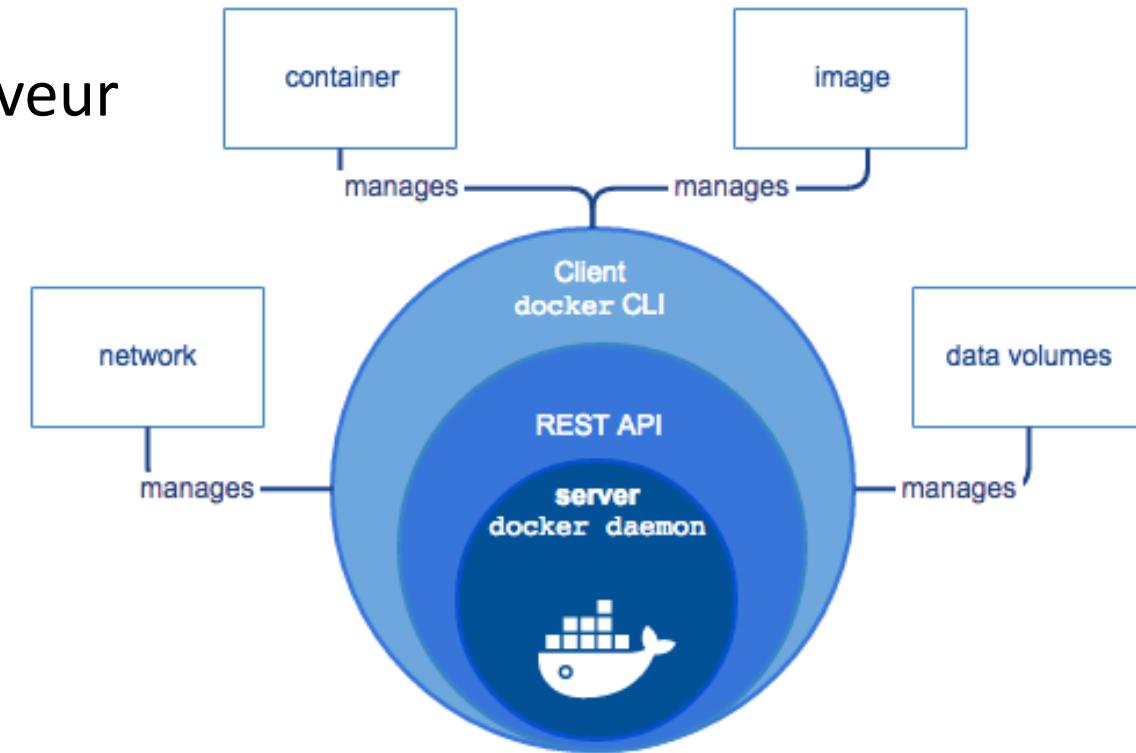
Pas besoin de simuler l'OS, ni le hardware, ni les applications



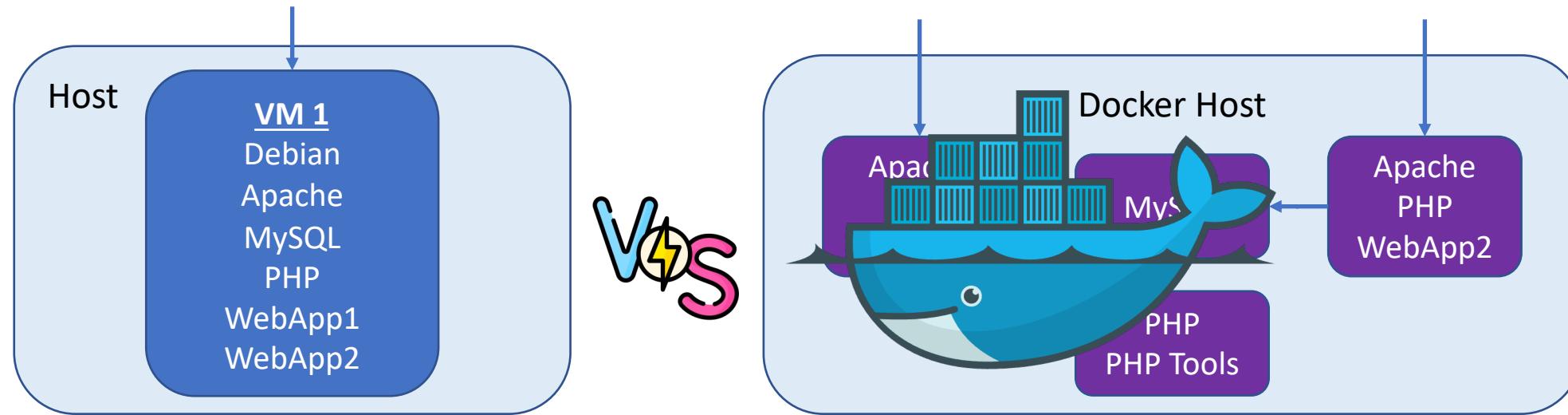
# Docker engine

Docker est en fait une application client-serveur avec 3 composants :

- Un serveur (processus dockerd)
- Une API REST utilisée pour communiquer avec le daemon
- Un client : le CLI



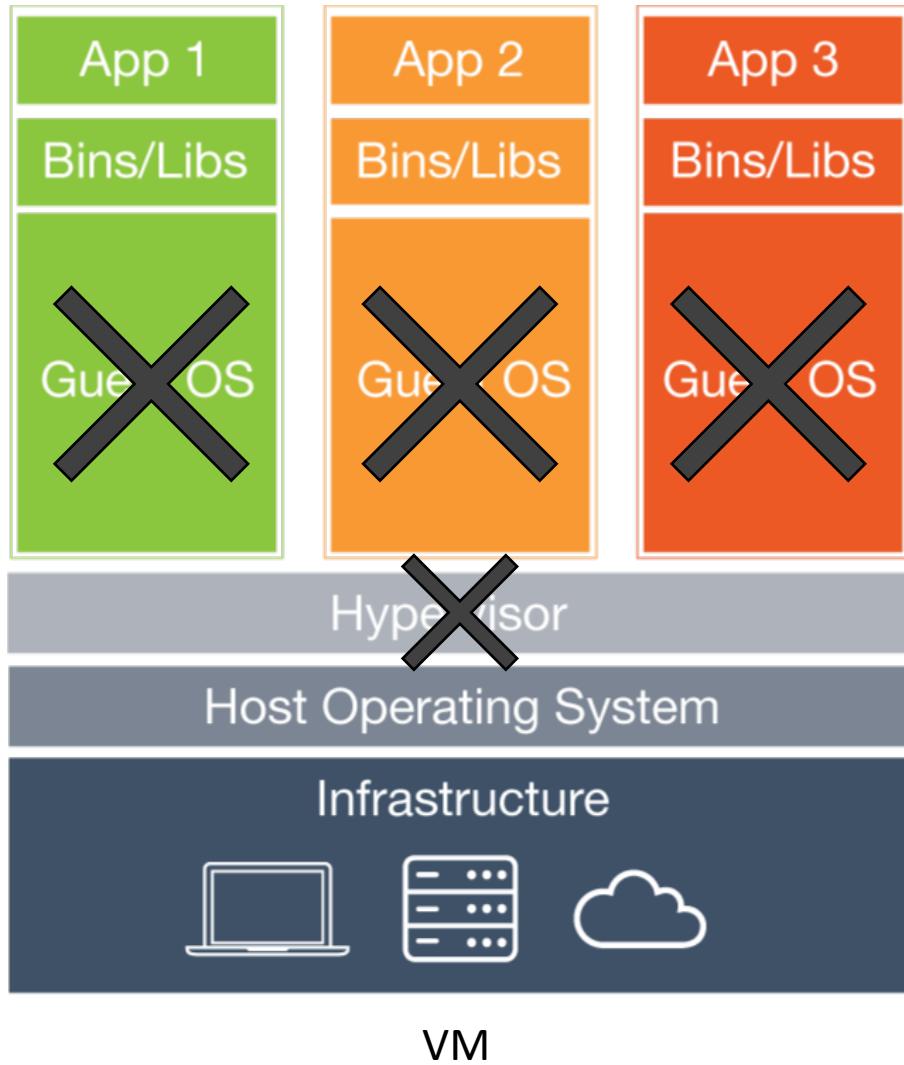
# L'effet obtenu : un livrable universel



## Avec Docker

- Le container n'est plus un système entier
- Container = application isolée et packagée avec toutes ses dépendances
- Toutes ces applications se gèrent et se lancent de la même manière
- Découplage : Pas besoin de maîtriser les composants dont on dépend

# VM vs Docker



©GalaxyProject

# Pets vs Cattle



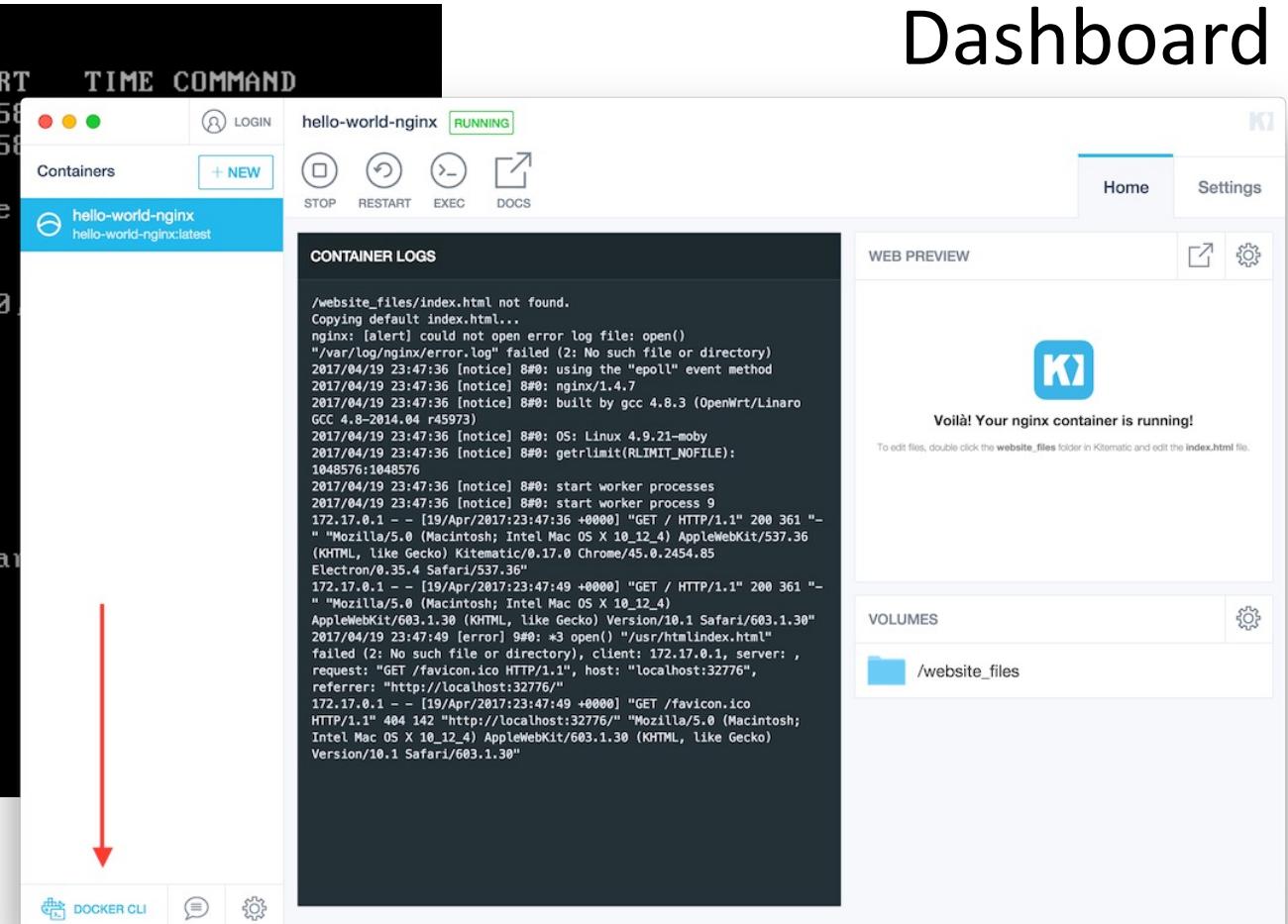
VS



# Les outils

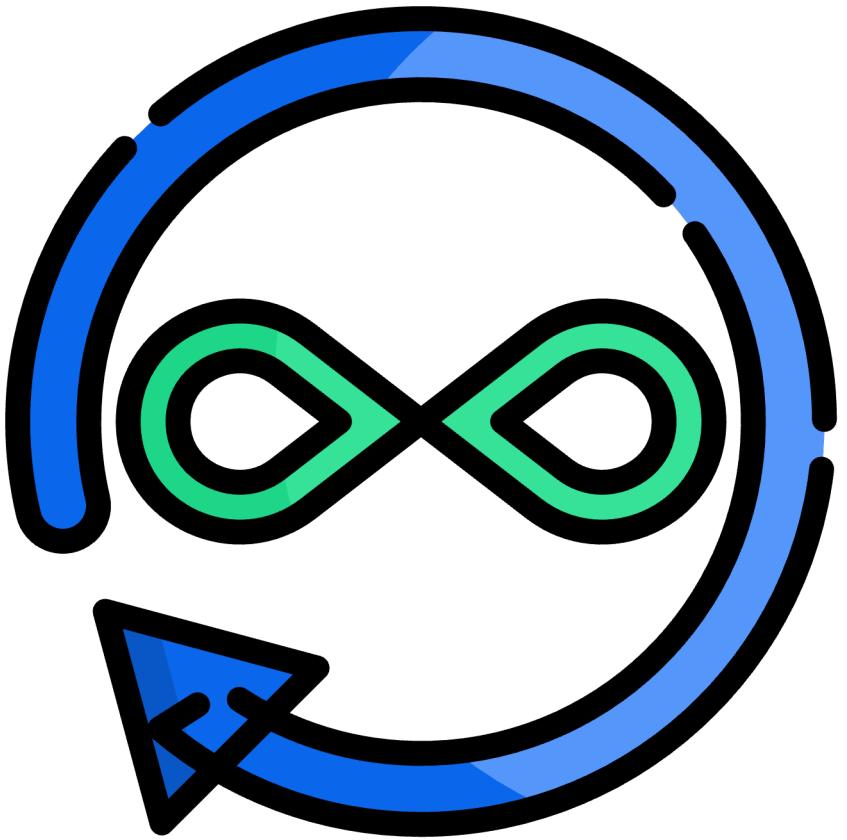
## CLI

```
[root@tecmint ~]# docker run -it ubuntu bash
[root@ff3cccd9fb856: /root@ff3cccd9fb856:/# ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.1  18112  1972 ?        Ss   06:58
root        15  0.0  0.1 15512  1144 ?        R+   06:58
[root@ff3cccd9fb856: /root@ff3cccd9fb856:/# uname -a
Linux ff3cccd9fb856 2.6.32-573.12.1.el6.x86_64 #1 SMP Tue
5 x86_64 x86_64 x86_64 GNU/Linux
[root@ff3cccd9fb856: /root@ff3cccd9fb856:/# w
 06:58:47 up 43 min,  0 users,  load average: 0.00, 0.00
USER     TTY      FROM          LOGIN@ IDLE   JCPU
[root@ff3cccd9fb856: /root@ff3cccd9fb856:/# cat /etc/de
debconf.conf default/      depmod.d/
debian_version deluser.conf
[root@ff3cccd9fb856: /root@ff3cccd9fb856:/# cat /etc/de
debconf.conf default/      depmod.d/
debian_version deluser.conf
[root@ff3cccd9fb856: /root@ff3cccd9fb856:/# cat /etc/debian
jessie/sid
[root@ff3cccd9fb856: /root@ff3cccd9fb856:/# exit
exit
[root@tecmint ~]# _
```



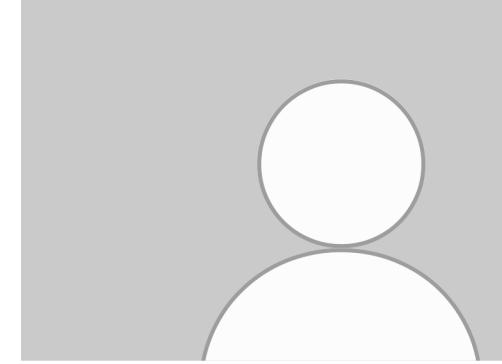
## En résumé

- Une image est un **livrable universel**
- Pas de dépendances !
- Un pas de plus vers une **standardisation** des briques techniques



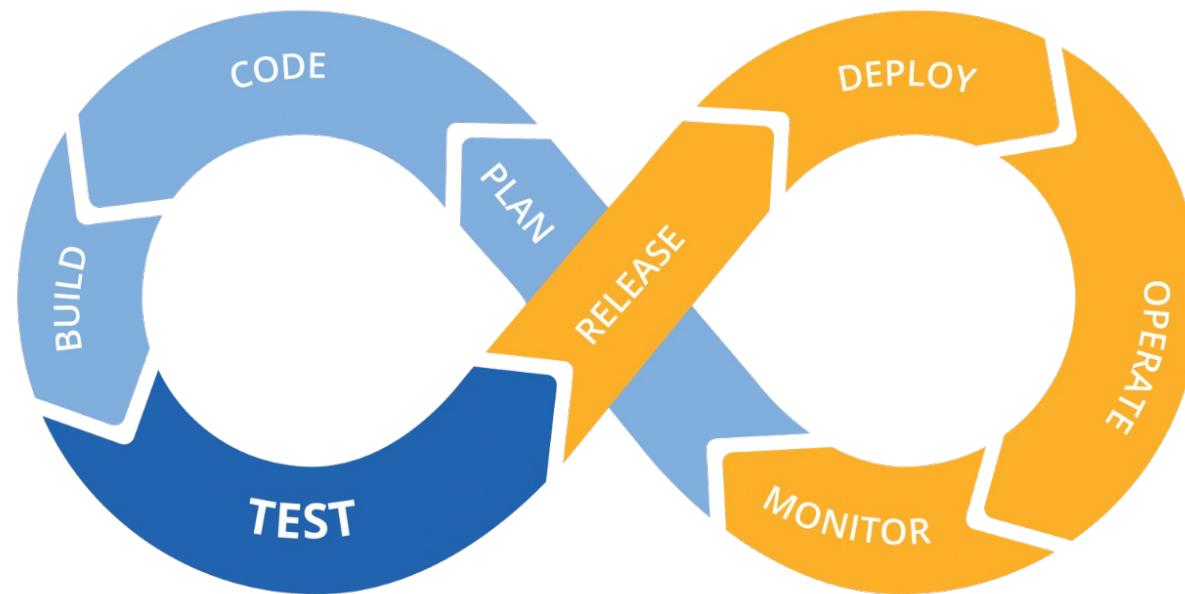
# CI/CD

# La CI/CD en quelques mots

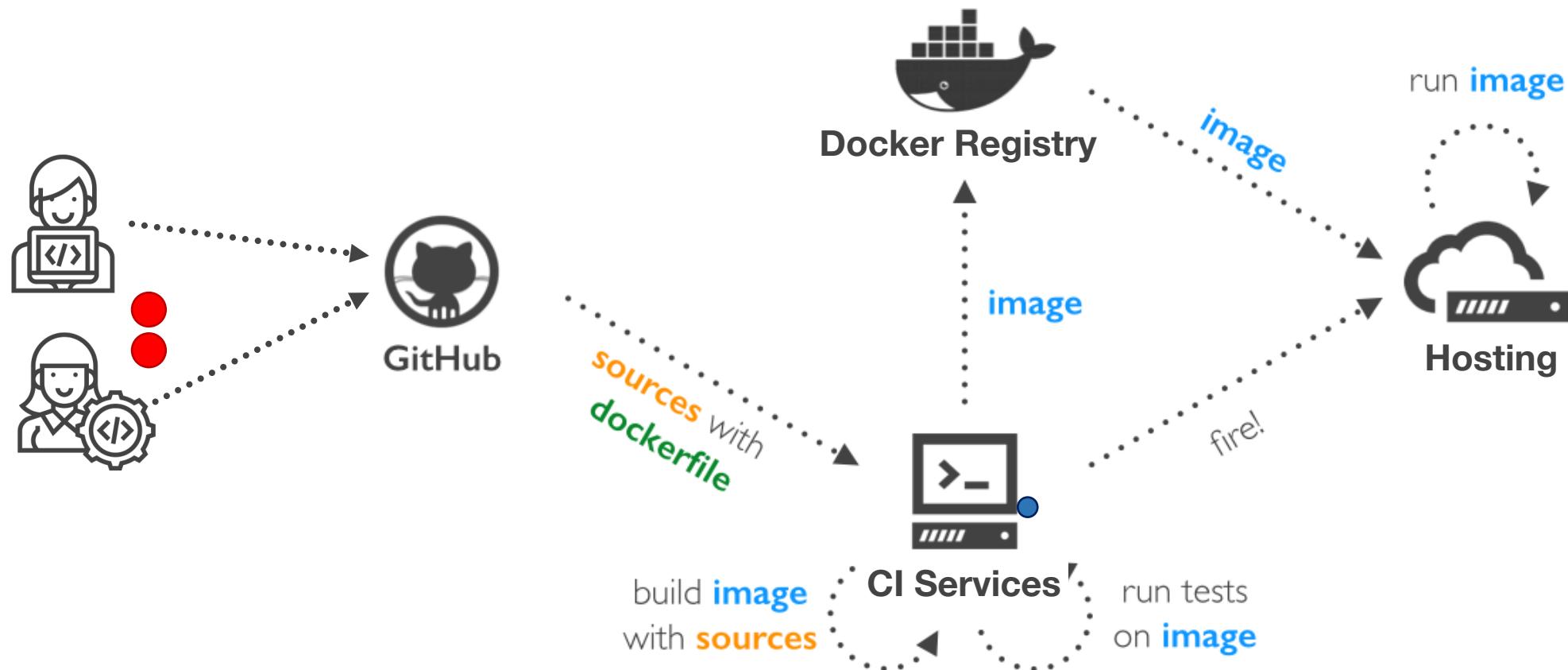


**Continuous Integration** : vérifier que chaque modification ne fait pas régresser

**Continuous Delivery** : dev en cycle court et pouvoir déployer n'importe quand, de manière fiable

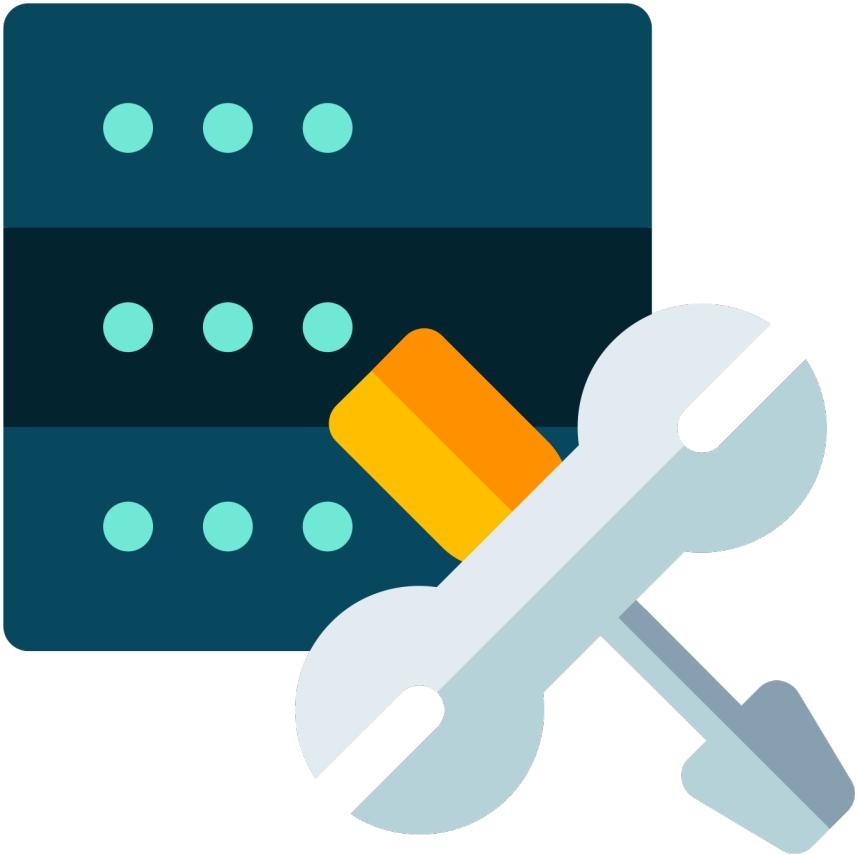


# Docker parfait pour la CI/CD



# Docker parfait pour la CI/CD

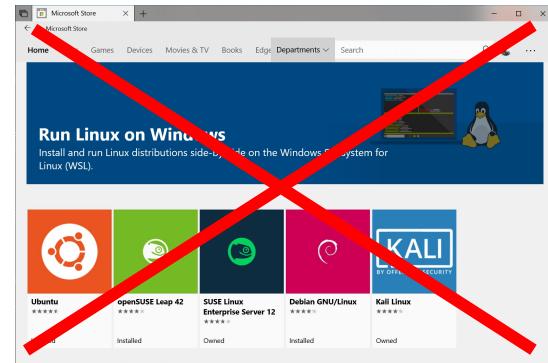
1. Docker n'a pas besoin de savoir où il fonctionne
2. La CI peut générer l'image et l'upload si les tests sont OK
3. Même image utilisable en dev/test/preprod/prod
4. Docker utilisable *on premise* comme dans le cloud
5. Lancement rapide des containers => bien pour test comme prod
6. Image *immutable*, parfait pour la reproductibilité



# Installation

# Docker Desktop for Windows

WSL 1



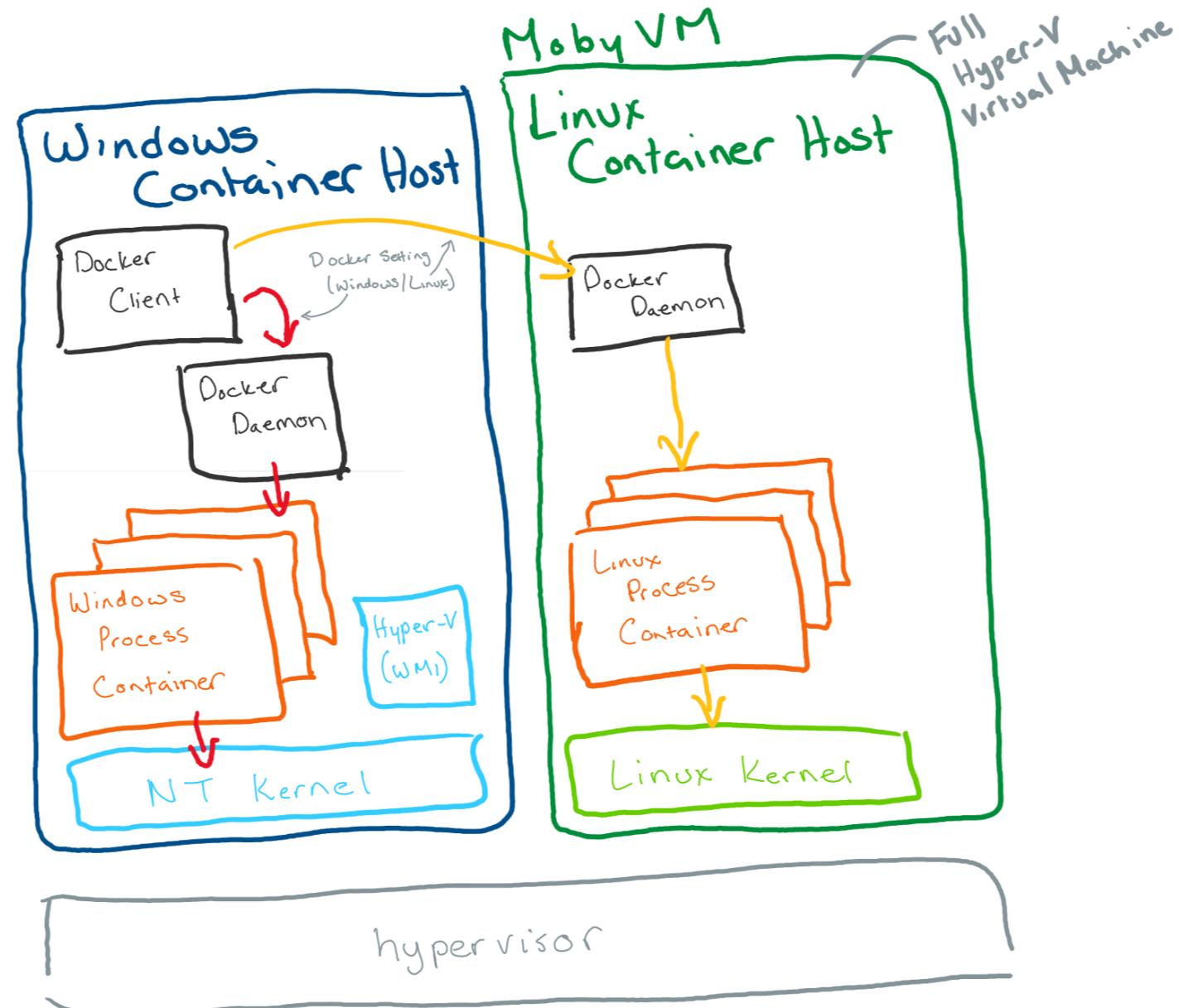
Windows PowerShell

WSL 2



# Docker For Windows on WSL1

- Il faut une VM pour faire tourner les containers Linux
- Compliqué pour du partage de dossier (Volume)
- Compliqué à intégrer avec du WSL

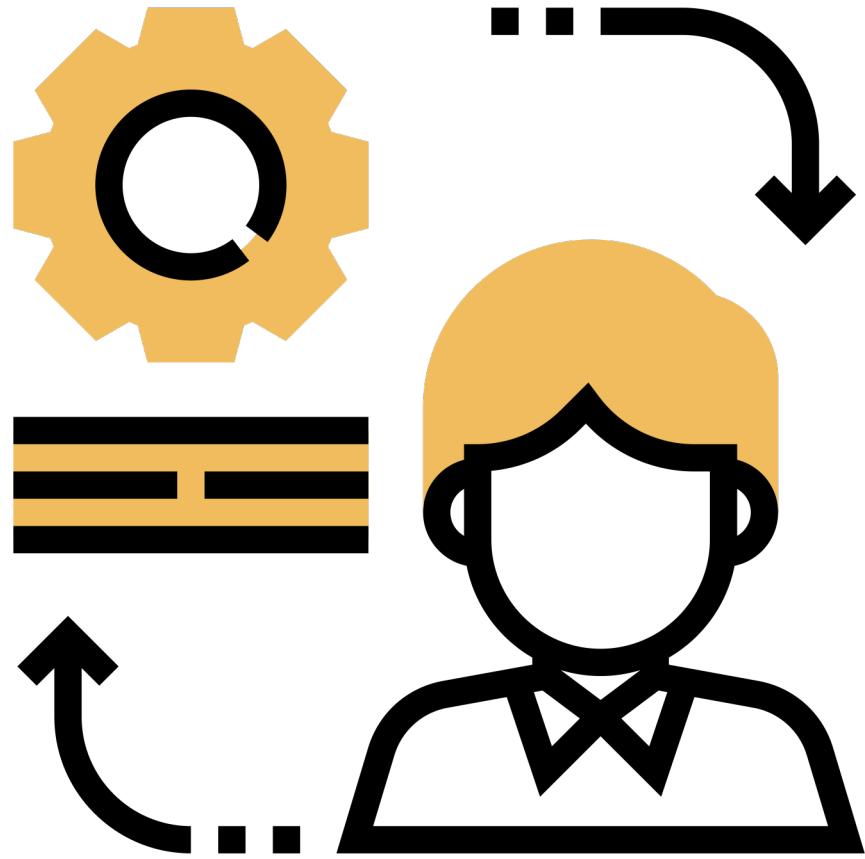


Src <https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers>

# Docker For Windows on WSL2

- Pas de Hyper-V
- Une VM légère
- Docker Windows en Shell ☺

Feature	WSL 1	WSL 2
Integration between Windows and Linux	✓	✓
Fast boot times	✓	✓
Small resource foot print	✓	✓
Runs with current versions of VMware and VirtualBox	✓	✓
Managed VM	✗	✓
Full Linux Kernel	✗	✓
Full system call compatibility	✗	✓



# Mise en pratique



# Mise en pratique

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

```
$ docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:ca0eeb6fb05351dfc8759c20733c91def84cb8007aa89a5bf606bc8b315b9fc7
Status: Downloaded newer image for hello-world:latest

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

```
$ docker image ls
```

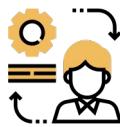
Listing des images déjà téléchargées

```
$ docker container ls
```

=

```
$ docker ps
```

Listing des containers (qui sont « up »)



Utilisation de `docker run` avec l'image alpine (qui ne fait rien, c'est une image de base)

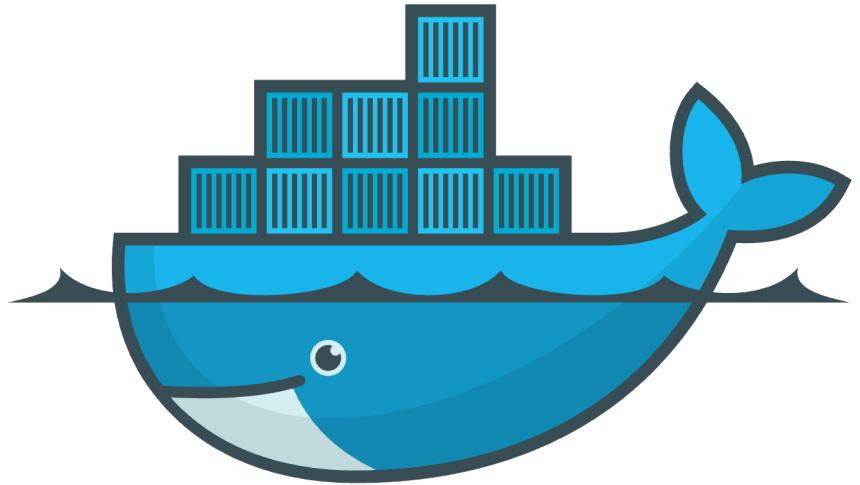
```
~$ docker run alpine  
~$
```

Utilisation de `docker run` pour lancer une commande custom

```
~$ docker run alpine echo hello  
hello
```

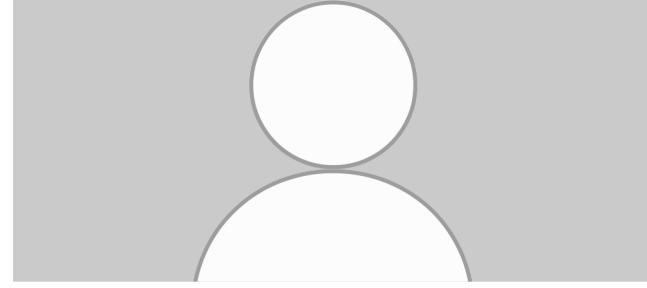
On voit que le `docker run` notre affiche bien le contenu du container, pas celui de notre machine

```
~$ docker run alpine ls -al  
total 56  
drwxr-xr-x    2 root      root            4096 Aug 20 10:30 bin  
...
```



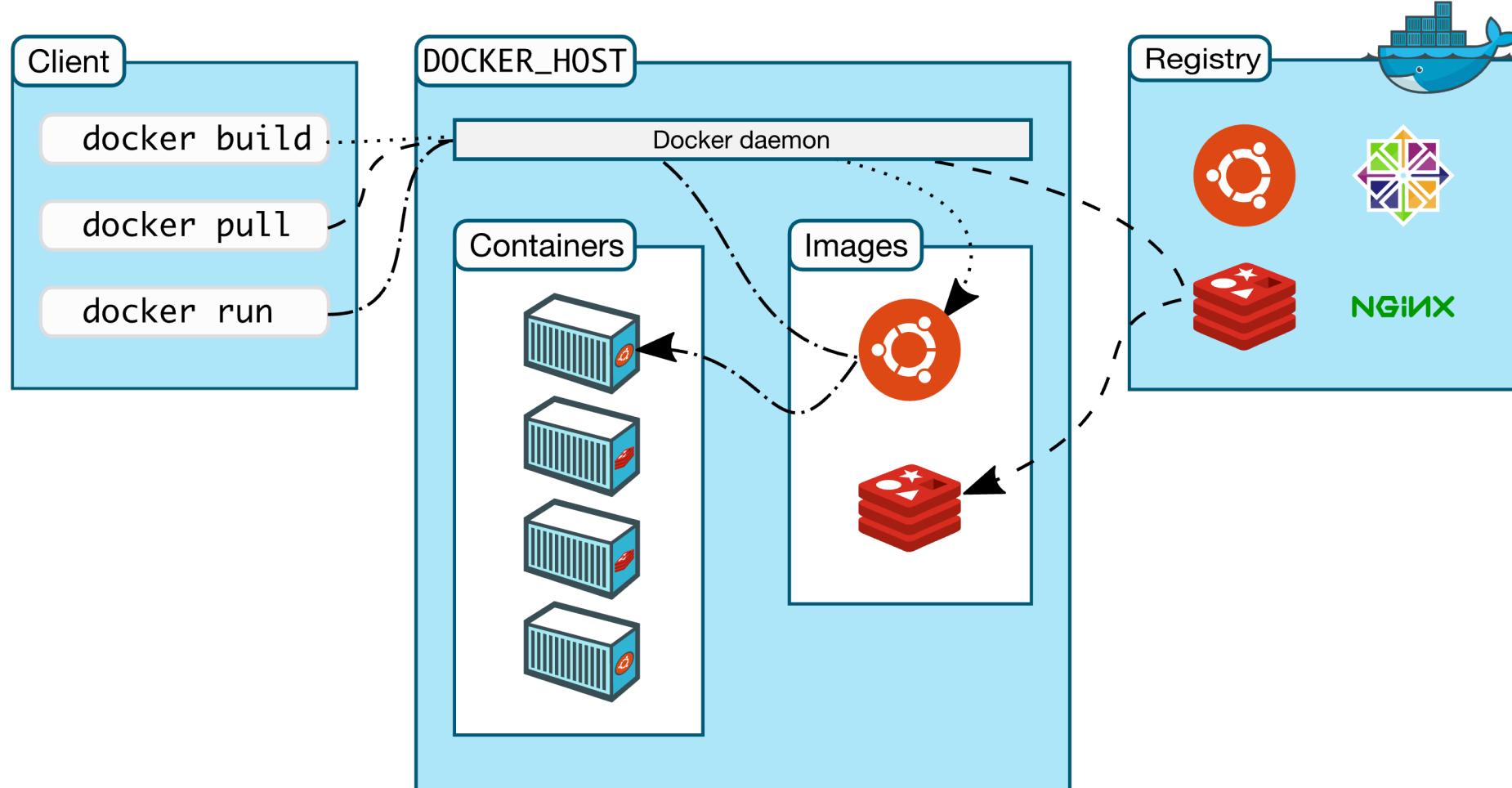
# Concepts Docker

# Registry



- Application serveur qui stocke et distribue les images Docker
- Possibilité d'authentification
- Stockage intelligent par rapport aux *layer*
- Image associées à un tag
- Possibilité d'utiliser du SaaS ou du On Premise

# Registry



# Nomenclature d'une image

```
[registry/]username/imagename[:tag]
```

- **registry**: si absent, pointe vers le **DockerHub**
- **username**: seules les images officielles n'en ont pas
- **tag**: si absent, défaut à **latest**

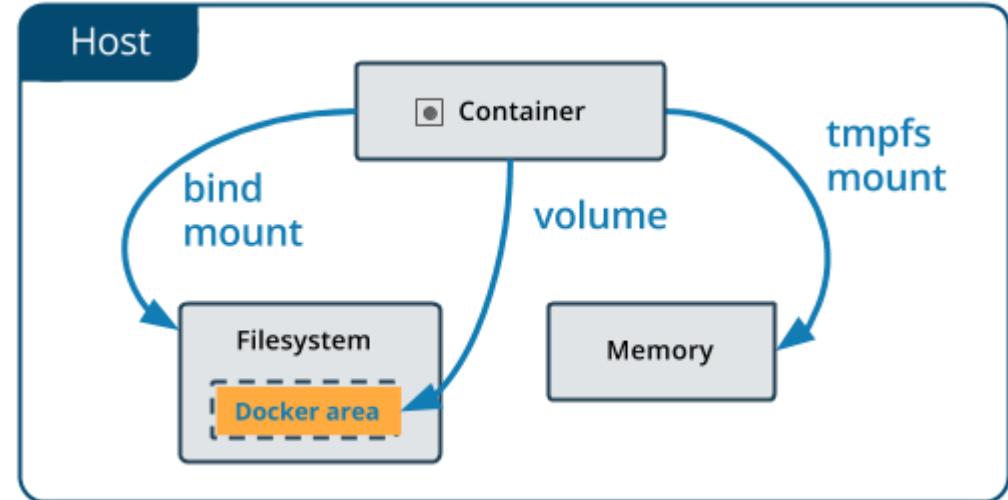
# Docker hub



- *Registry* officiel de Docker
- Bien plus qu'un *registry*, c'est une offre SaaS
- Stockage des images officielles
- Build automatique des images après 1 push sur Git
- Gestion des teams

# Volume

Les images sont immutables et les containers sont fait pour mourir  
=> il faut des volumes !

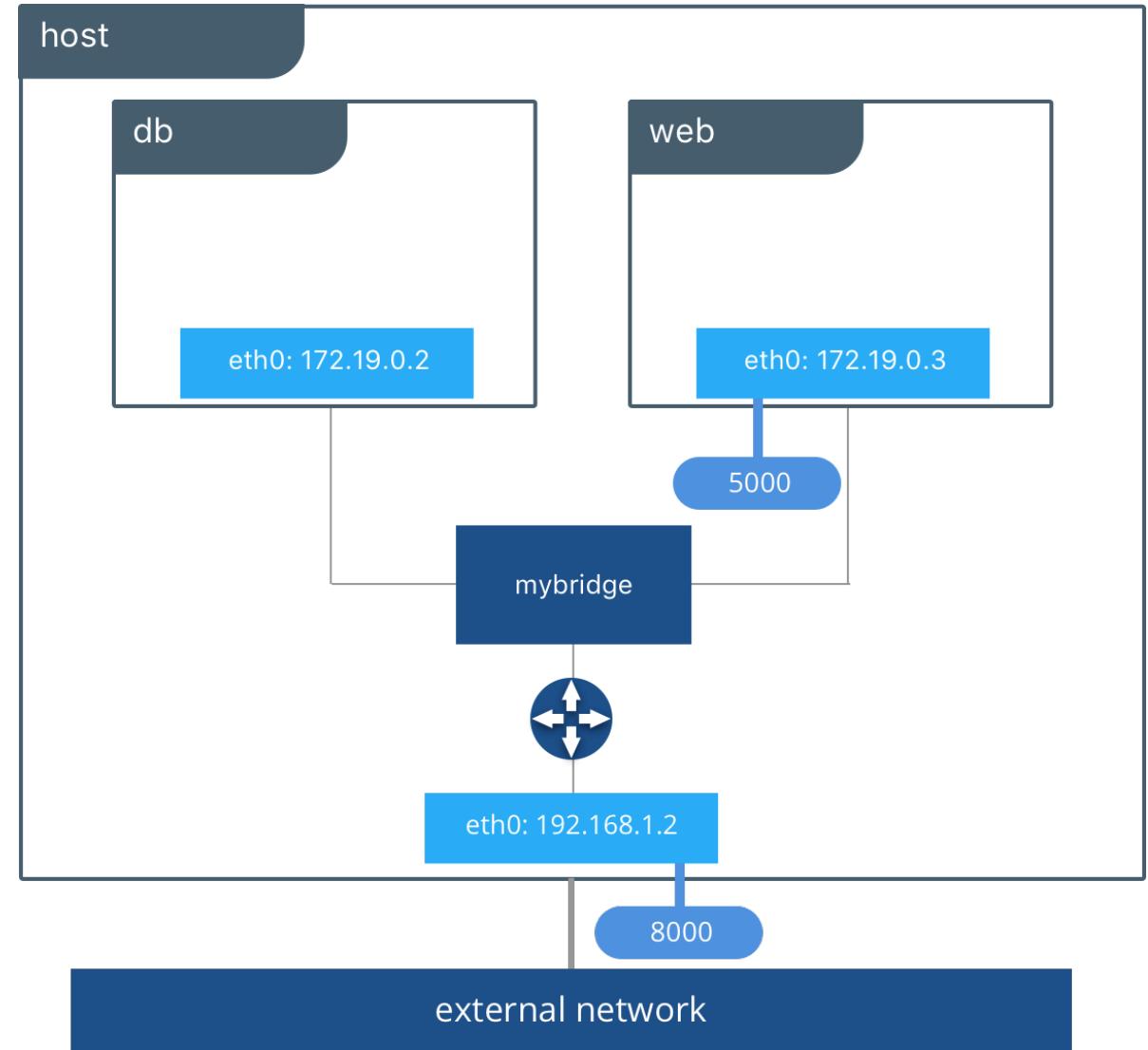


- Mécanisme pour faire persister des données sur la machine hôte
- Complètement géré par Docker
- Possibilité de le partager entre plusieurs containers
- Peut être créé avant 1 container
- Peut être en Read Only

# Networking

## Port

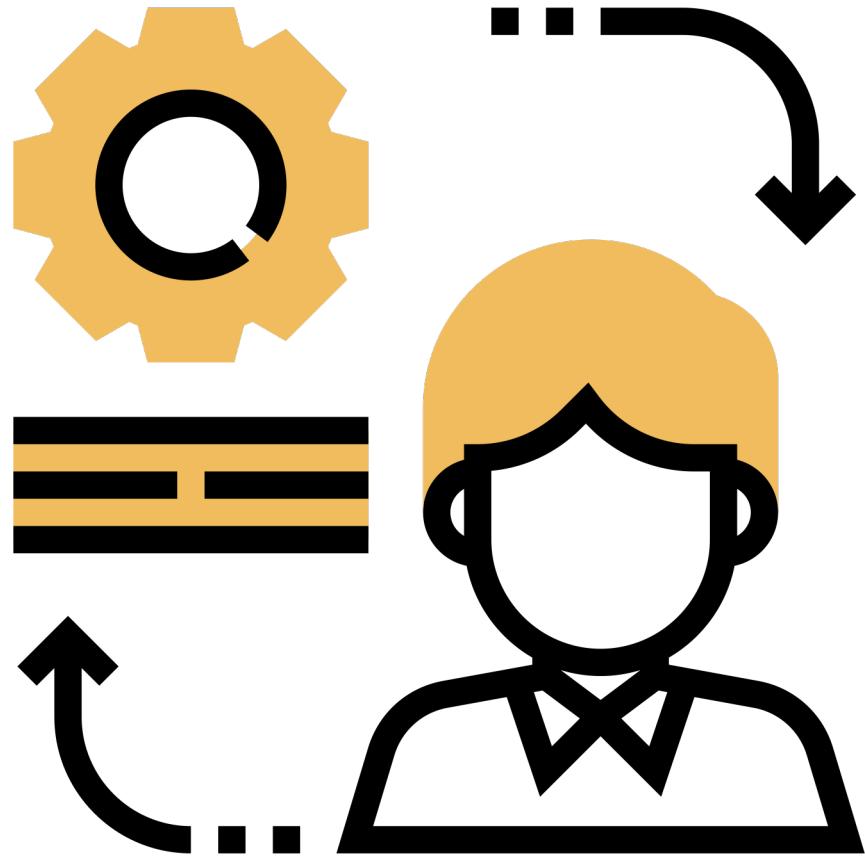
- Par défaut, le container ne publie aucun port vers l'extérieur
- Il faut publier explicitement le port
- Pas de conflit de port entre plusieurs instances de la même image, c'est la machine hôte qui gère



# Networking

## DNS & Network

- Les containers peuvent se contacter entre eux (même sans publier un port)
- On peut créer des réseaux pour isoler les containers entre eux



# Mise en pratique



Listing de tous les container qui sont « *running* », puis de tous les containers, peu importe leur état

```
~$ docker container ps
CONTAINER   ID   IMAGE   COMMAND   CREATED   STATUS    PORTS   NAMES
~$ docker container ps --all
???
```

Docker `run` avec option `--rm` qui supprime le container dès qu'il a fini de s'exécuter.

On peut exécuter `docker ps` ensuite pour le confirmer

```
~$ docker run --rm alpine echo "hello world"
hello world
```

Docker `image ls` permet de lister les images présentes sur l'ordinateur

```
~$ docker image ls
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
alpine          latest   961769676411  4 weeks ago  5.58MB
hello-world     latest   fce289e99eb9  8 months ago 1.84kB
```



Docker run avec option --it pour lancer en mode « interactive terminal »

On se retrouve alors dans le container. Quand on fait exit, on sort du container.

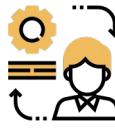
```
~$ docker run -it --name interactive_shell alpine /bin/sh  
/ # touch myuniquefile  
/ # exit  
~$
```

Si on relance un container, notre fichier n'est plus là. C'est normal, c'est un nouveau container.

```
~$ docker run -it alpine /bin/sh  
/ # ls myuniquefile  
ls: myuniquefile: No such file or directory
```

Le fichier est toujours là mais il faut relancer le container qui était « *exited* »

```
~$ docker start -i interactive_shell  
/ # ls myuniquefile  
myuniquefile
```



On expose le port 80 du container vers le port 8001 de notre machine.  
Allez avec votre browser sur l'url <http://localhost:8001> pour consulter le site

```
~$ docker run -p 8001:80 --rm nginxdemos/hello
```

Depuis un autre terminal, vous pouvez confirmer que votre container est bien entrain de run.

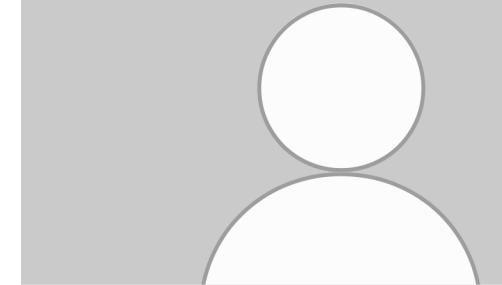
```
~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                 NAMES
69a2b5000d7e        nginxdemos/hello   "nginx -g 'daemon of..."   2 minutes ago      Up 2 minutes       0.0.0.0:8001->80/tcp   dazzling_gauss
```

Vous pouvez vous entraîner en lançant un deuxième container de la même image sur un autre port.  
Les 2 seront fonctionnels en même temps.



# Construction des images

# Dockerfile



Docker peut créer des images à partir des instructions du Dockerfile

01

On ajoute dans son dépôt un **Dockerfile**, fichier contenant la "recette" pour produire une image à partir du code

02

On obtient une **image**, sorte de tarball sous stéroïdes qui peut être déposée dans un **registre** (un serveur avec plein d'images)

03

On instancie autant de fois qu'on veut cette image, ces instances sont appelées **conteneur**

# Example de Dockerfile

```
FROM node:19-alpine

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
COPY package*.json ./

RUN npm install

# Bundle app source
COPY .

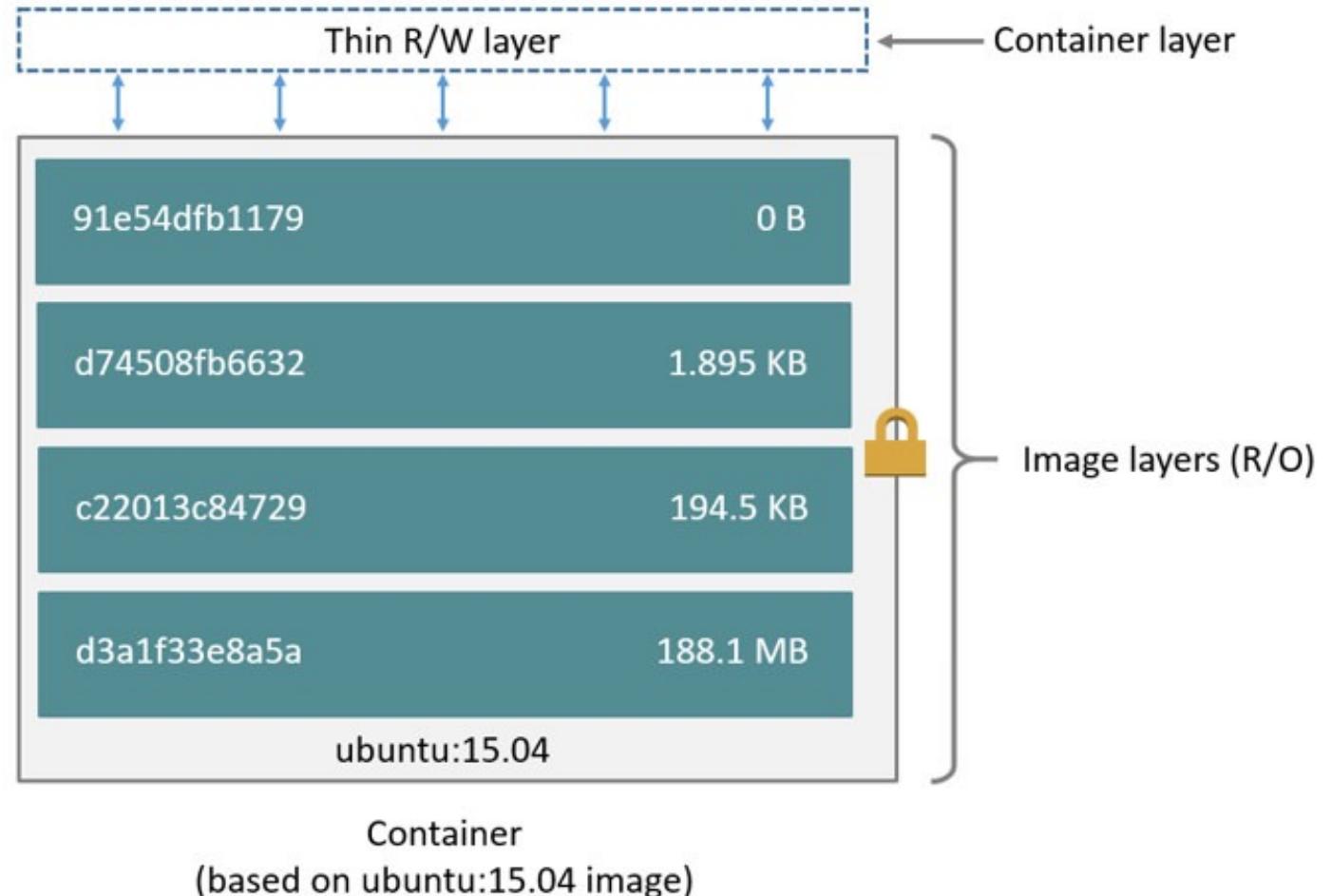
EXPOSE 8080
CMD [ "node", "server.js" ]
```

# Dockerfile

- On utilise la commande `docker build` pour construire une image
- Chaque ligne est exécutée et génère un nouveau *layer*
- On ne manipule que l'image finale
- Docker met en cache toutes les *layers* intermédiaires
- Economie stockage pour les images qui ont des *layers* communs

# Dockerfile

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```



## Build from scratch

```
$ docker build . -t node-web-app
[+] Building 9.7s (11/11) FINISHED
=> [internal] load build definition from Dockerfile                                     0.0s
=> => transferring dockerfile: 505B                                                 0.0s
=> [internal] load .dockerignore                                                 0.0s
=> => transferring context: 149B                                                 0.0s
=> [internal] load metadata for docker.io/library/node:19-alpine                     1.8s
=> [auth] library/node:pull token for registry-1.docker.io                           0.0s
=> [1/5] FROM docker.io/library/node:19-alpine@sha256:72b0f918ad76b5ef68c6243869fab5800d7393c1dcccf54ef00958c2abc8164a 2.7s
=> => resolve docker.io/library/node:19-alpine@sha256:72b0f918ad76b5ef68c6243869fab5800d7393c1dcccf54ef00958c2abc8164a 0.0s
=> => sha256:9b7c6faea308da9eac4d18543cb04c6ef9110319858a6fc81206f4bf330362c9 2.41MB / 2.41MB          0.6s
=> => sha256:72b0f918ad76b5ef68c6243869fab5800d7393c1dcccf54ef00958c2abc8164a 1.43kB / 1.43kB          0.0s
=> => sha256:0c201d0512b1b2ddd694dd0c42fb232c93a931c978607c098d6bda782ae76786 1.16kB / 1.16kB          0.0s
=> => sha256:ea92cacba269cd42a25c3b7a67b8aa105d1595115b277e738d94c0947be7fff 6.45kB / 6.45kB          0.0s
=> => sha256:a9eaa45ef418e883481a13c7d84fa9904f2ec56789c52a87ba5a9e6483f2b74f 3.26MB / 3.26MB          0.2s
=> => sha256:1e68e48f57e223c337a00d4b42cfc2e43827760b938e82b483b2f10ae9209682 47.91MB / 47.91MB         1.7s
=> => extracting sha256:a9eaa45ef418e883481a13c7d84fa9904f2ec56789c52a87ba5a9e6483f2b74f           0.1s
=> => sha256:1e826af8c749dfc5ccce6a8786276627f6b3294ea454c4cb6247fec6efc6b7d8 449B / 449B          0.9s
=> => extracting sha256:1e68e48f57e223c337a00d4b42cfc2e43827760b938e82b483b2f10ae9209682           0.7s
=> => extracting sha256:9b7c6faea308da9eac4d18543cb04c6ef9110319858a6fc81206f4bf330362c9           0.1s
=> => extracting sha256:1e826af8c749dfc5ccce6a8786276627f6b3294ea454c4cb6247fec6efc6b7d8           0.0s
=> [internal] load build context                                                 0.0s
=> => transferring context: 1.81kB                                              0.0s
=> [2/5] WORKDIR /usr/src/app                                                 0.1s
=> [3/5] COPY package*.json ./                                              0.0s
=> [4/5] RUN npm install                                                 4.9s
=> [5/5] COPY . .                                                       0.0s
=> exporting to image                                                 0.1s
=> => exporting layers                                                 0.1s
=> => writing image sha256:08fd931b5f2fbf3edc88ebc4e5a9e23a3adc2420163c2a819e31eb0559296f64 0.0s
=> => naming to docker.io/library/node-web-app                            0.0s
```

# Build après modification d'un fichier

Example: modification du code source, les étapes précédentes sont pas ré-exécutées (utilisation du cache)

```
$ docker build . -t node-web-app
[+] Building 0.7s (10/10) FINISHED
=> [internal] load build definition from Dockerfile          0.0s
=> => transferring dockerfile: 241B                         0.0s
=> [internal] load .dockerignore                            0.0s
=> => transferring context: 117B                           0.0s
=> [internal] load metadata for docker.io/library/node:19-alpine 0.5s
=> [1/5] FROM docker.io/library/node:19-alpine@sha256:72b0f918ad76b5ef68c6243869fab5800d7393c1dccc54ef00958c2abc8164a 0.0s
=> [internal] load build context                          0.0s
=> => transferring context: 1.33kB                      0.0s
=> CACHED [2/5] WORKDIR /usr/src/app                     0.0s
=> CACHED [3/5] COPY package*.json ./                   0.0s
=> CACHED [4/5] RUN npm install                         0.0s
=> [5/5] COPY . .                                       0.0s
=> exporting to image                                    0.0s
=> => exporting layers                                  0.0s
=> => writing image sha256:6f4dd390de33c8d6a38b2f169316095460ce6e84db59d5a79f4f4af92fc68c10 0.0s
=> => naming to docker.io/library/node-web-app        0.0s
```

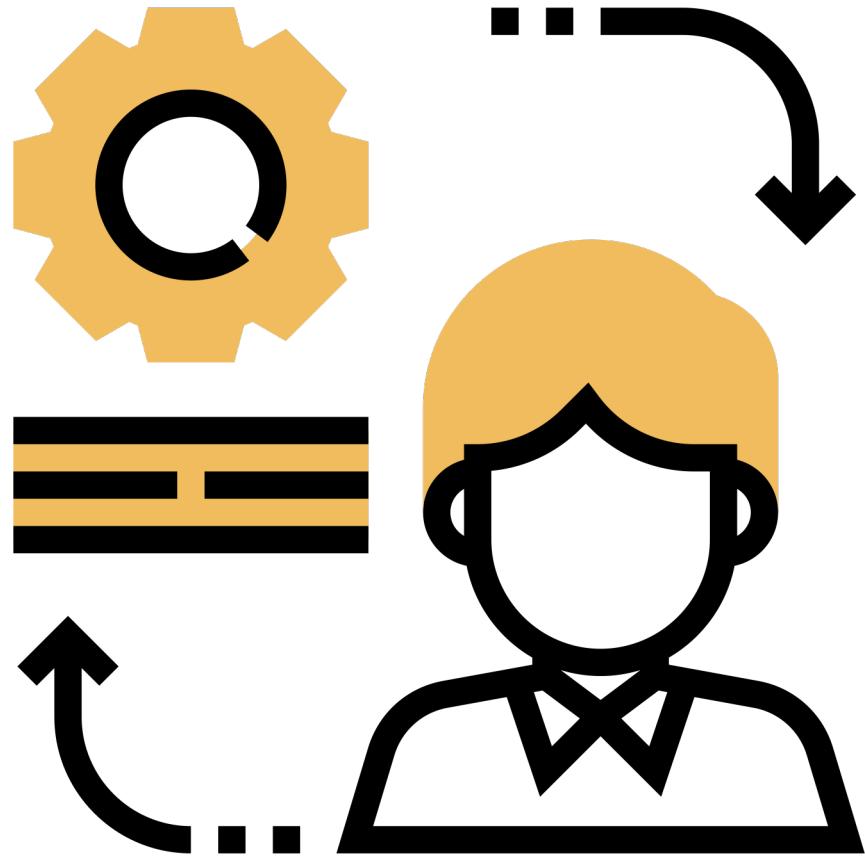
## Not cached

# Inspection des layers

Seule  
"vraie"  
image

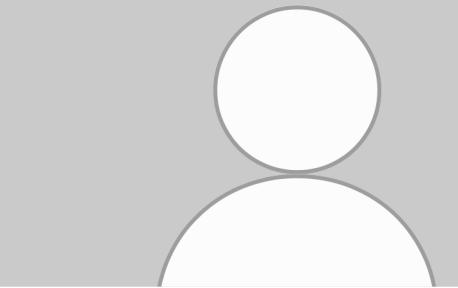
Layers de  
l'image  
source

\$ docker history node-web-app	IMAGE	CREATED	CREATED BY	SIZE	COMMENT
	6f4dd390de33	About a minute ago	CMD ["node" "server.js"]	0B	buildkit.dockerfile.v0
	<missing>	About a minute ago	EXPOSE map[8080/tcp:{}]	0B	buildkit.dockerfile.v0
	<missing>	About a minute ago	COPY . . # buildkit	841B	buildkit.dockerfile.v0
	<missing>	3 minutes ago	RUN /bin/sh -c npm install # buildkit	6.25MB	buildkit.dockerfile.v0
	<missing>	3 minutes ago	COPY package*.json ./ # buildkit	174B	buildkit.dockerfile.v0
	<missing>	3 minutes ago	WORKDIR /usr/src/app	0B	buildkit.dockerfile.v0
	<missing>	6 days ago	/bin/sh -c #(nop) CMD ["node"]	0B	
	<missing>	6 days ago	/bin/sh -c #(nop) ENTRYPOINT ["docker-entry...	0B	
	<missing>	6 days ago	/bin/sh -c #(nop) COPY file:4d192565a7220e13...	388B	
	<missing>	6 days ago	/bin/sh -c apk add --no-cache --virtual .bui...	7.84MB	
	<missing>	6 days ago	/bin/sh -c #(nop) ENV YARN_VERSION=1.22.19	0B	
	<missing>	6 days ago	/bin/sh -c addgroup -g 1000 node && addu...	159MB	
	<missing>	6 days ago	/bin/sh -c #(nop) ENV NODE_VERSION=19.6.0	0B	
	<missing>	4 weeks ago	/bin/sh -c #(nop) CMD ["/bin/sh"]	0B	
	<missing>	4 weeks ago	/bin/sh -c #(nop) ADD file:3080f19f39259a4b7...	7.46MB	



# Example de Dockerfile

# Exo : hébergement nodejs



## serveur.js : Api basique

```
'use strict';

const express = require('express');

// Constants
const PORT = 8080;
const HOST = '0.0.0.0';

// App
const app = express();
app.get('/', (req, res) => {
  res.send(`{"msg": "Hello Zorld"}`);
});
                                Faute d'ortographe !

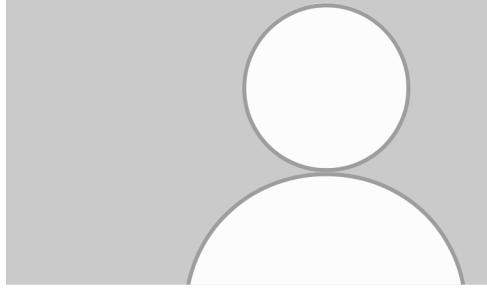
process.on('SIGINT', function() {
  console.log("Caught interrupt signal");
  process.exit();
});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

## package.json : dépendances

```
{
  "name": "docker_web_app",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.18"
  }
}
```

# Exo : hébergement nodejs



## Dockerfile

```
FROM node:19-alpine

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
COPY package*.json .
RUN npm install

# Bundle app source
COPY .

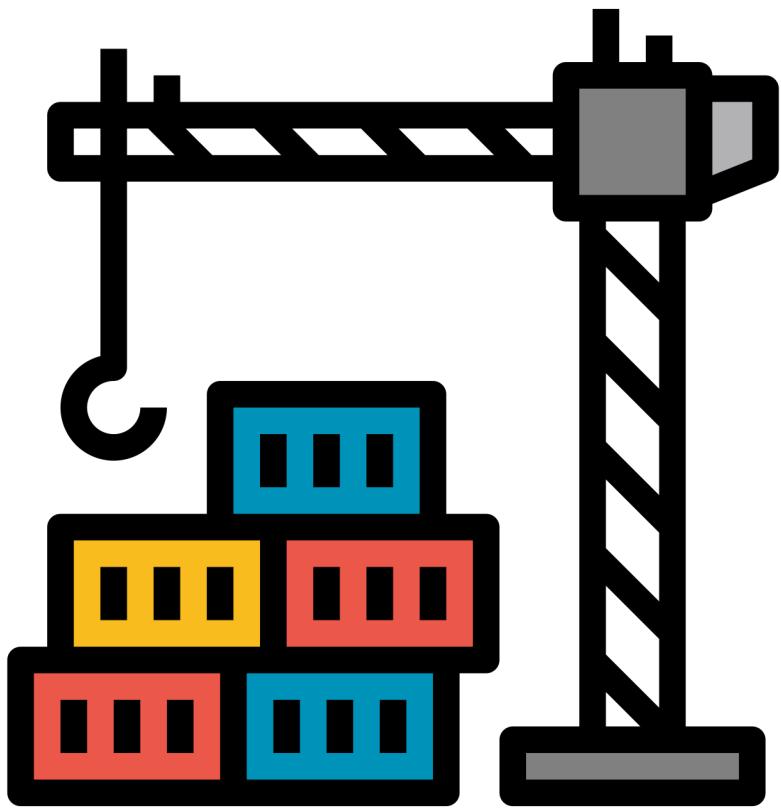
# Does not publish the port but acts as a documentation
EXPOSE 8080
# Default command
CMD [ "node", "server.js" ]
```

# Exo : hébergement nodejs

```
$ docker build -t nodejs_demo .
```

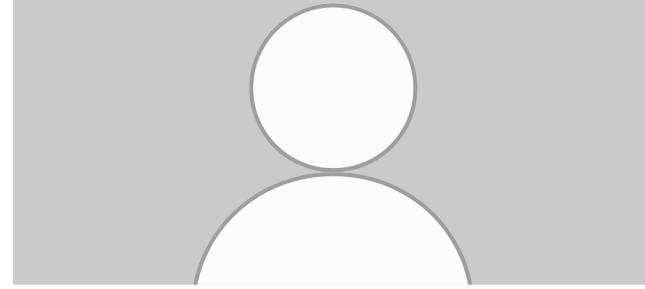
```
$ docker run --rm -p 80:8080 nodejs_demo
```

=> Api consultable sur <http://localhost>

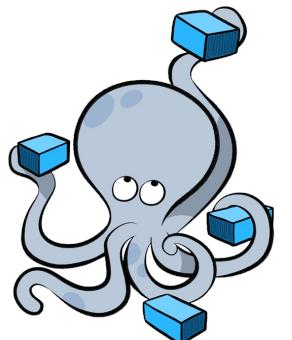


# Docker Compose

# Docker compose



- Outils pour définir et exécuter des app multi-containers
- Le *docker-compose.yml* défini comment exécuter les services et les interconnecter
- Plus besoin de retenir tous les params de lancement
- On peut définir du networking ou des volumes
- Très utile pour de la CI
- Pas vraiment d'intelligence au delà du start/stop



# Docker compose up

```
$ docker compose up
```

Lance les services non démarré ('docker run')

Détecte les modifications du yaml et create/update/delete si besoin

- Nouveau service
- Service supprimé
- Service modifié
- Maj d'un param ou d'une image

# Docker compose stop

```
$ docker compose stop
```

Stop les containers lancé mais sans les removes

```
$ docker compose start
```

Relance les containers arrêtés

```
$ docker compose rm
```

Supprime les containers arrêtés

# Compose main commands

```
$ docker compose up -d
Starting docker_nginx-golang-postgres_db_1 ... done
Starting docker_nginx-golang-postgres_backend_1 ... done
Starting docker_nginx-golang-postgres_proxy_1 ... done
```

Create and start services

```
$ docker compose stop
Stopping docker_nginx-golang-postgres_proxy_1 ... done
Stopping docker_nginx-golang-postgres_backend_1 ... done
Stopping docker_nginx-golang-postgres_db_1 ... done
```

Stop services  
Can be started again

```
$ docker compose rm
Going to remove docker_nginx-golang-postgres_proxy_1, docker_nginx-golang-
postgres_backend_1, docker_nginx-golang-postgres_db_1
Are you sure? [yN] y
Removing docker_nginx-golang-postgres_proxy_1 ... done
Removing docker_nginx-golang-postgres_backend_1 ... done
Removing docker_nginx-golang-postgres_db_1 ... done
```

Remove stopped services

# Exemple de front + back

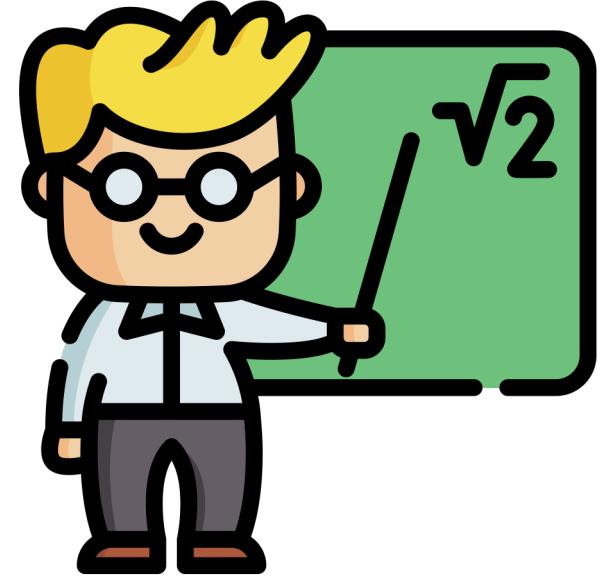
Exemples de docker compose

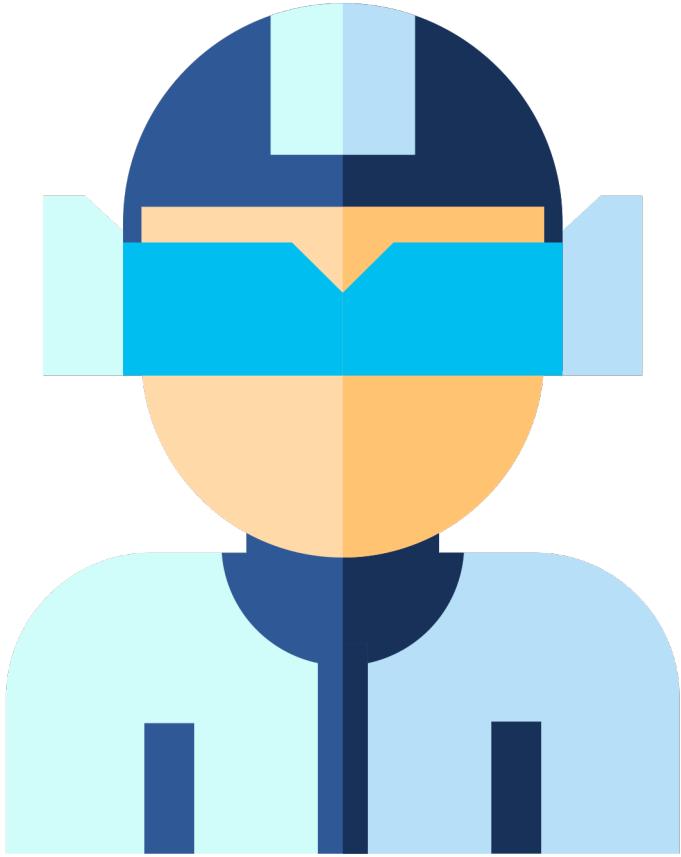
<https://github.com/docker/awesome-compose>

Exemple étudié

- Un front en JS
- Une base Redis
- Un proxy NGINX

<https://github.com/docker/awesome-compose/tree/master/nginx-nodejs-redis>





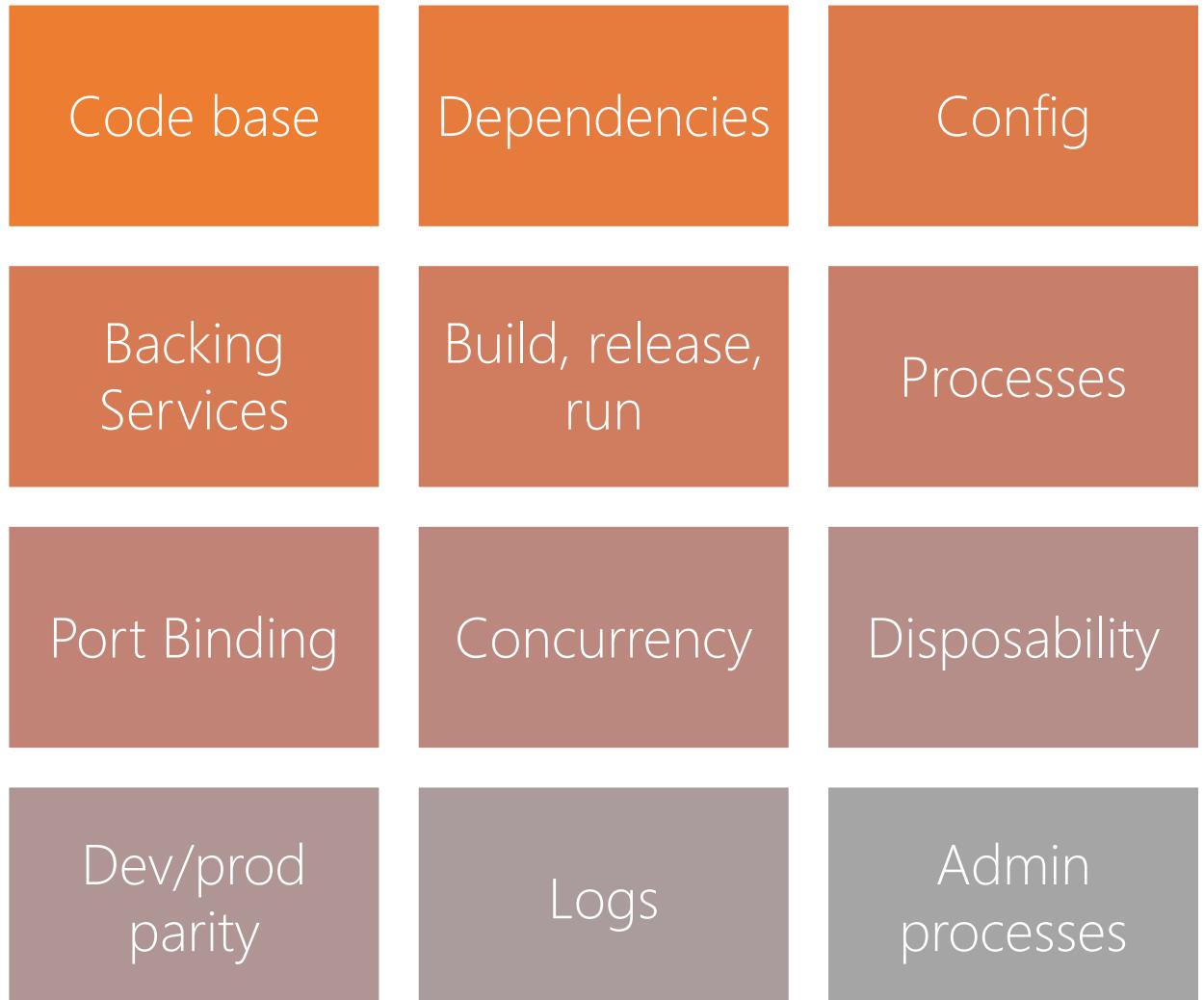
Pour aller plus  
loin

# The Twelve-Factor

Méthodologie pour concevoir  
des applications SaaS

- Format déclaratif
- Portabilité
- Adapté au cloud
- Orienté CI/CD

<https://12factor.net>

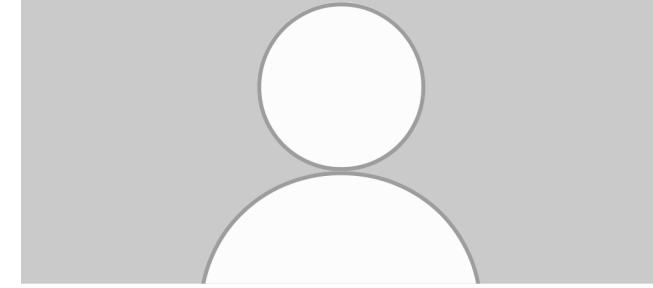




# Kubernetes

Benoit Verdier - EPITA

# Pourquoi un orchestrateur ?



- Docker run c'est bien, mais c'est pas suffisant.
- Je veux :
  - Lancer N fois mon conteneur
  - Avoir un loadbalancer
  - Surveiller la santé des conteneurs
  - Faire des rolling updates
  - Faire du service discovery
  - Gérer le state correctement
  - ...

# L'approche K8S

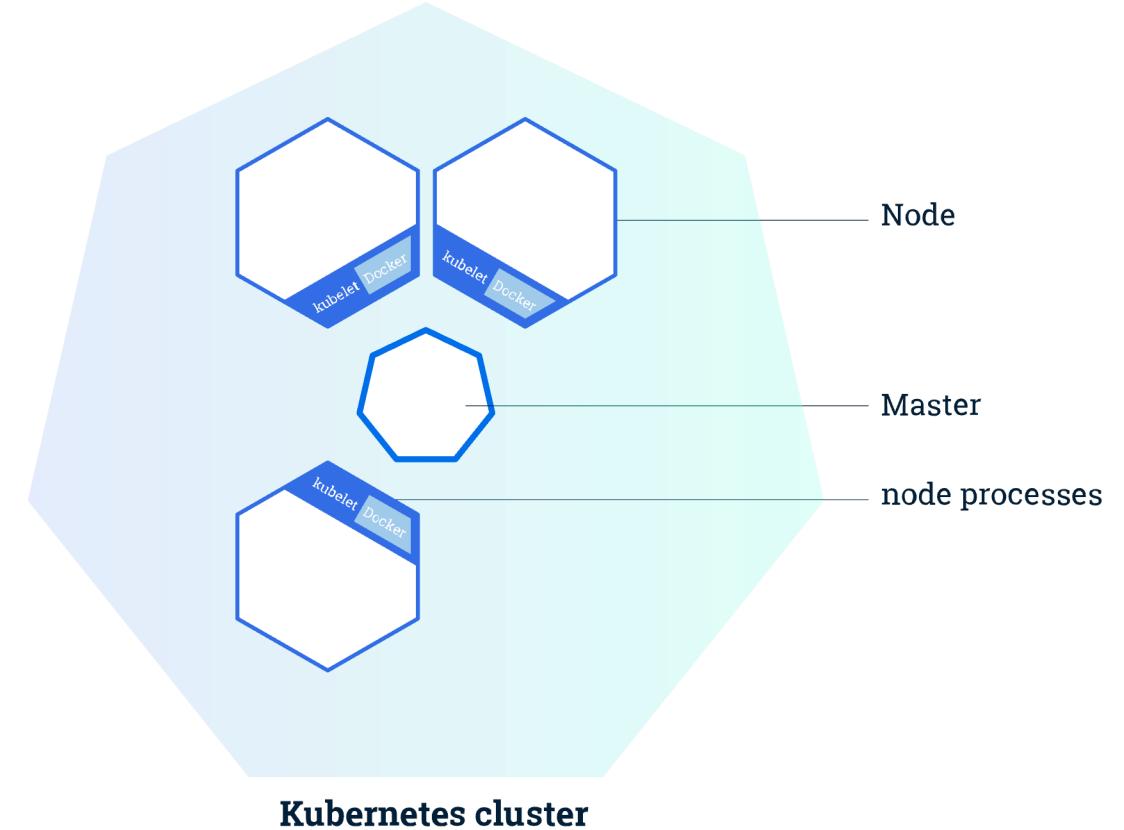
Abstraction des concepts de modélisation

Approche déclarative

Description de l'état désiré à travers des ressources dans des fichiers yaml

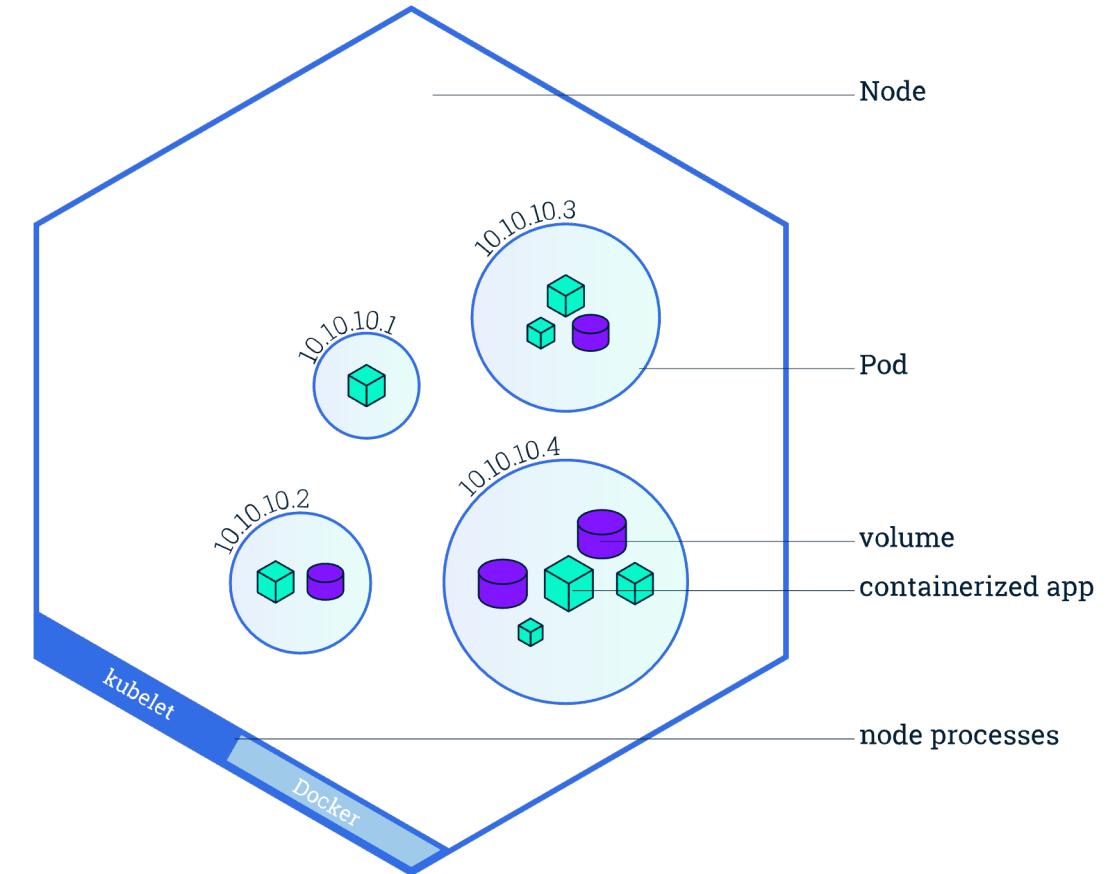
# ARCHI K8S: Le cluster

- Master gère le cluster
- Nodes communiquent avec master
- Cluster de prod a 3 nœuds ou plus



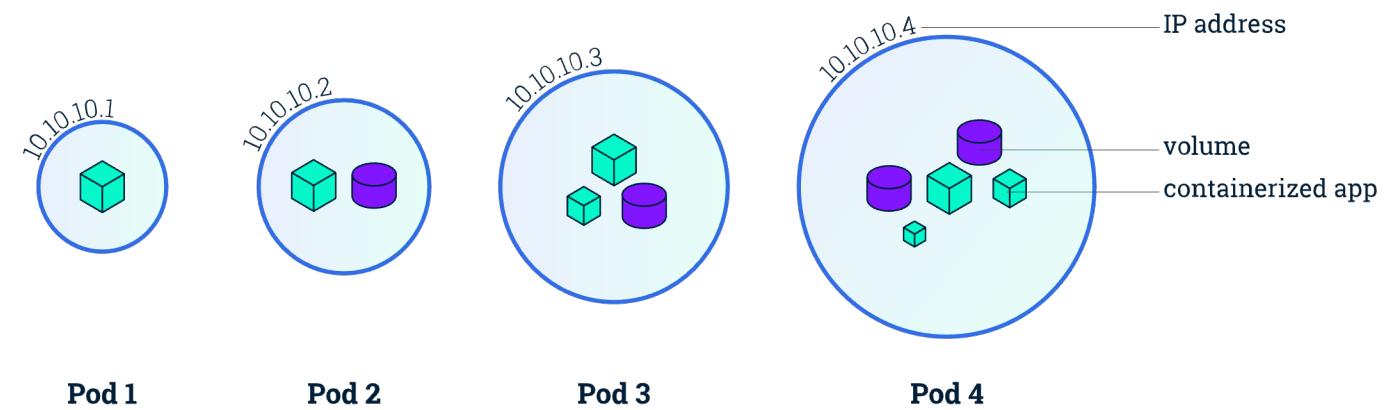
# ARCHI K8S: détail d'un NOEUD

Node  
=  
Kubelet & Docker & Pods



# ARCHI K8S: détail d'un Pod

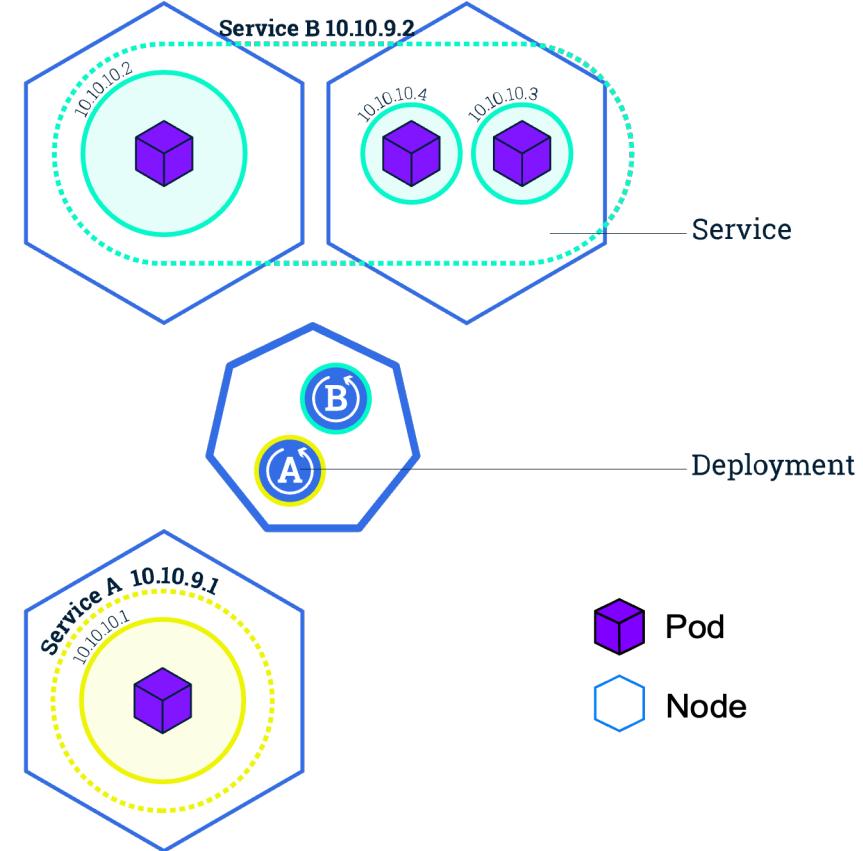
- $\geq 1$  container
- Stockage partagé (volumes)
- 1 IP unique dans le cluster
- Informations sur l'exécution



# ARCHI K8S: exposer avec les services

## Un service

- Route
- Expose les pods
- Répartis la charge



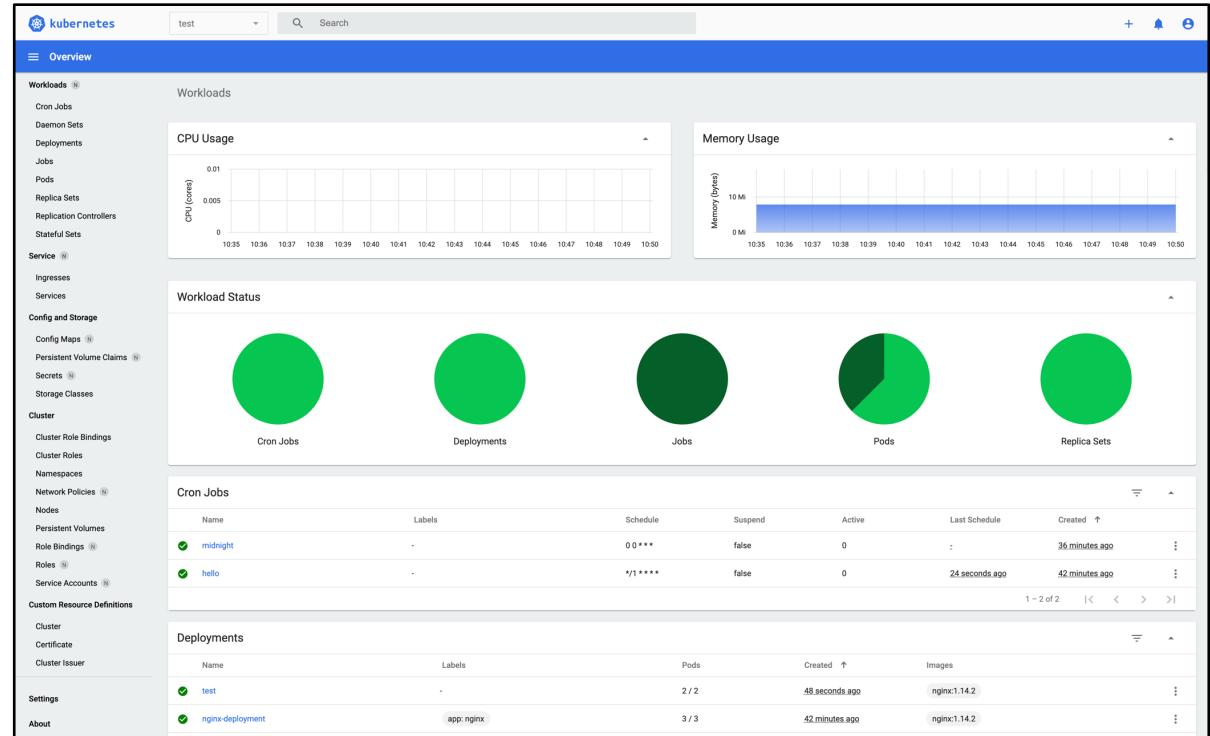
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  selector:
    matchLabels:
      app: guestbook
      tier: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google-samples/gb-frontend:v4
          resources:
            requests:
              cpu: 100m
              memory: 100Mi
          env:
            - name: GET_HOSTS_FROM
              value: dns
          ports:
            - containerPort: 80
```

## Exemple de ressource

# Pour aller plus loin

## Installer un Dashboard

<https://github.com/kubernetes/dashboard>



## Installer un sample

<https://github.com/digitalocean/kubernetes-sample-apps>

# Kubernetes, caas, paas

Beaucoup de containers as a service sont “powered by Kubernetes”

Il devient facile de faire du PaaS.

Le PaaS devient une surcouche à kubernetes en interne



*That's all Folks!*