INTRODUCTION À DOCKER

Benoît Benedetti

OBJECTIFS

SÉANCE 1

Savoir ce que c’est que Docker et à quoi ça sert

- Concepts et terminologie,
- Exécuter des conteneurs,
- Gérer des images,
- Créer ses propres images.

SÉANCE 2

Utilisation avancée

- Interconnecter des conteneurs
- Orchestration

PLAN DE CETTE SÉANCE

- Problématique
- Conteneur Linux
- Docker

LEVEZ VOTRE MAIN SI...

- Vous avez installé Docker sur votre machine?
- Vous en aviez déjà entendu parler?
- Vous avez lu, regardé des ressources?
- Vous avez déjà testé?

PROBLÉMATIQUES

LE DÉVELOPPEMENT AVANT

- Longue phase de développement
- Longue phase de recette/test
- Mise en production
- On touche plus!

MAINTENANT

- Développement Agile
- Cycles courts (MVP, Release early/Release often)

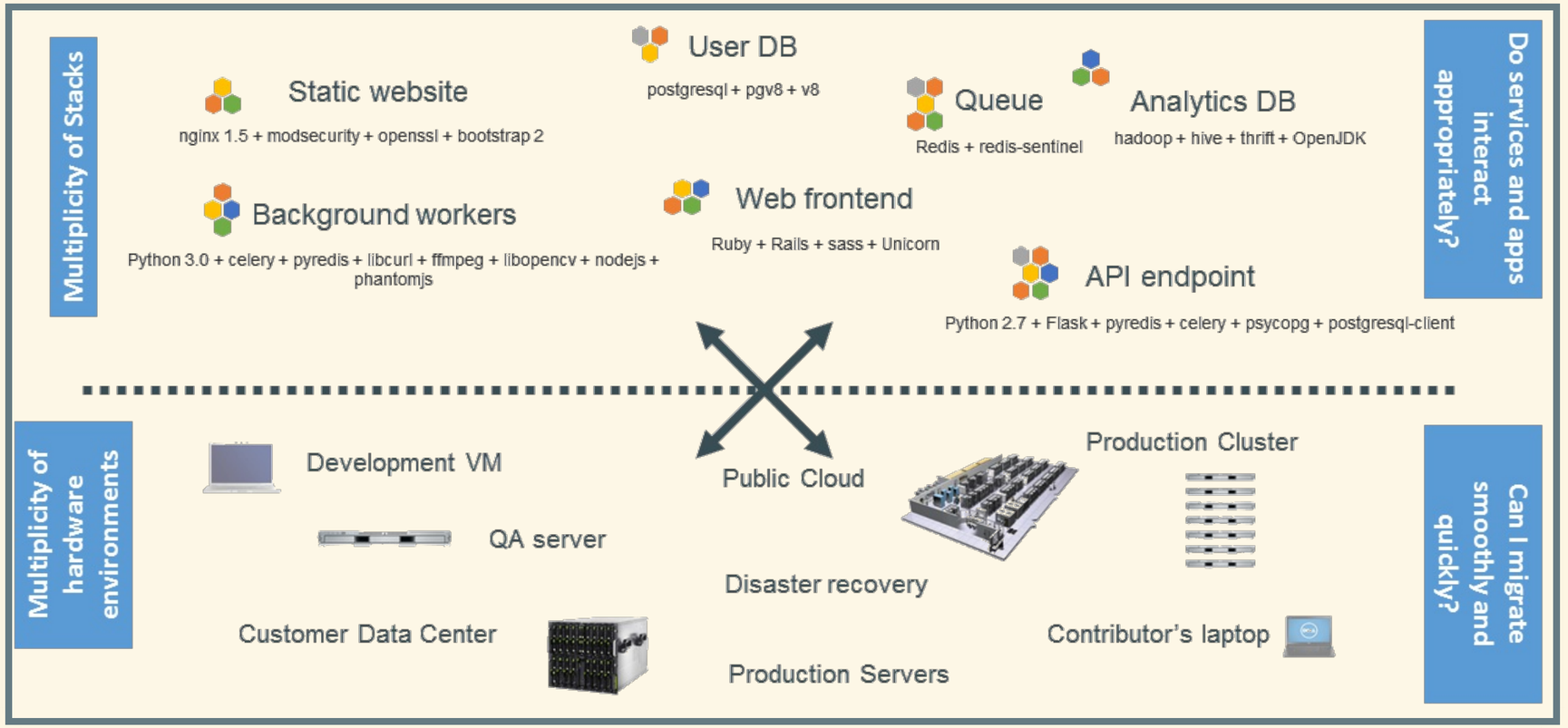
AVANT

- Applications monolithiques
- Evolutivité verticale (vertical scalability)
- Dev VS Ops

MAINTENANT

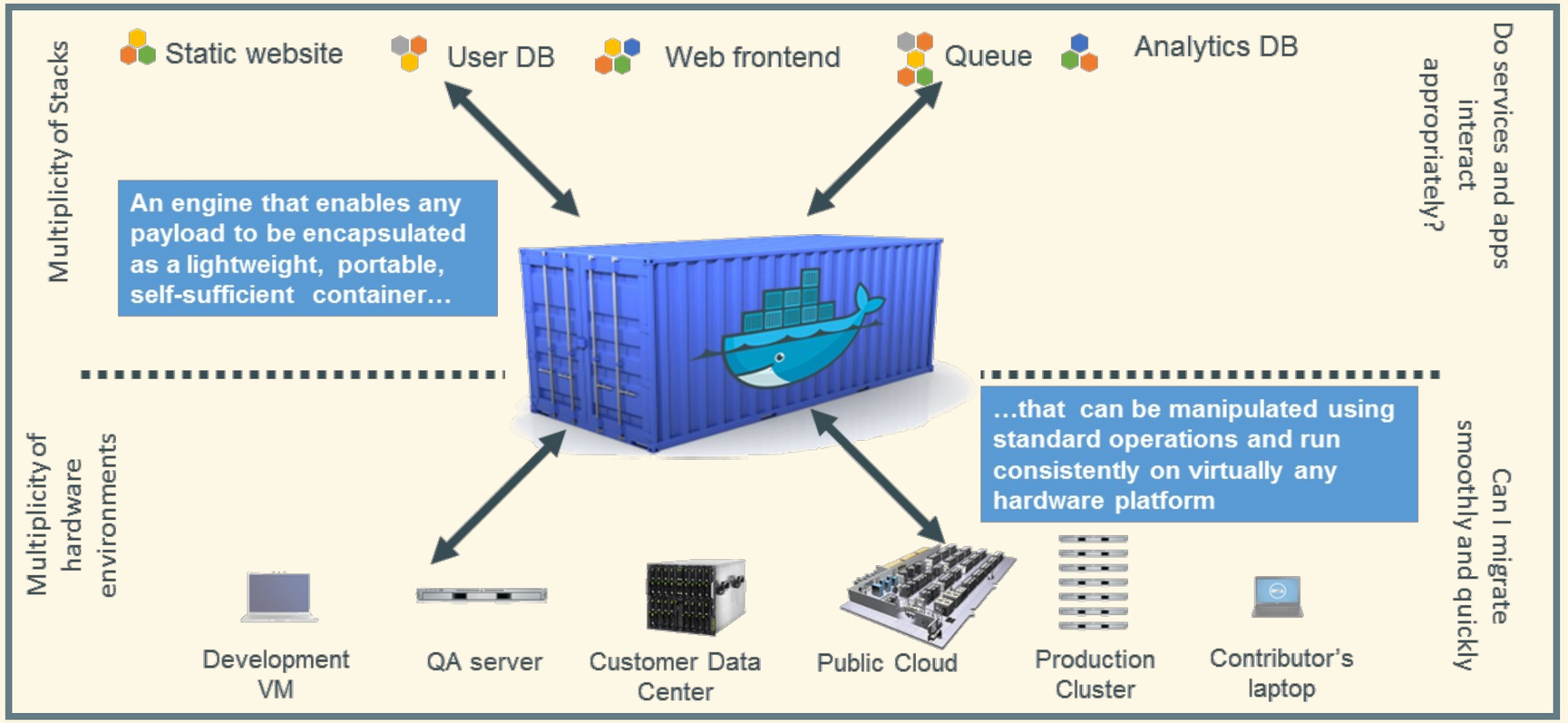
- Environnements polyglottes, différentes stacks/Frameworks
- Evolutivités horizontale(répandue) et architecturale(micro-services)
- DevOps, environnements identiques à la production sur le poste de dev

LA MATRICE INFERNALE



source: docker.com @Docker, inc

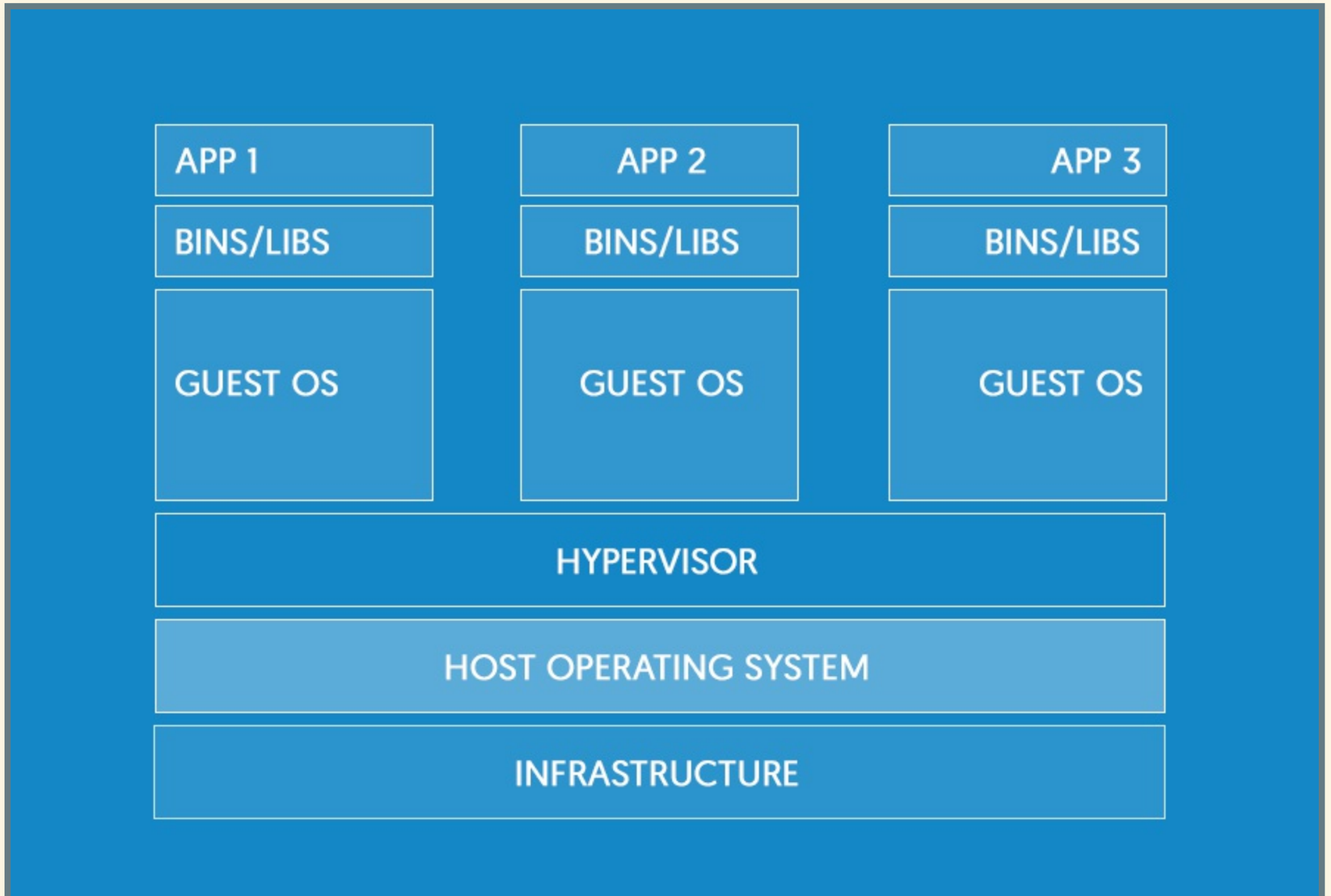
APERÇU DOCKER



source: docker.com @Docker, inc

GÉNÉRALITÉS SUR LES CONTAINEURS

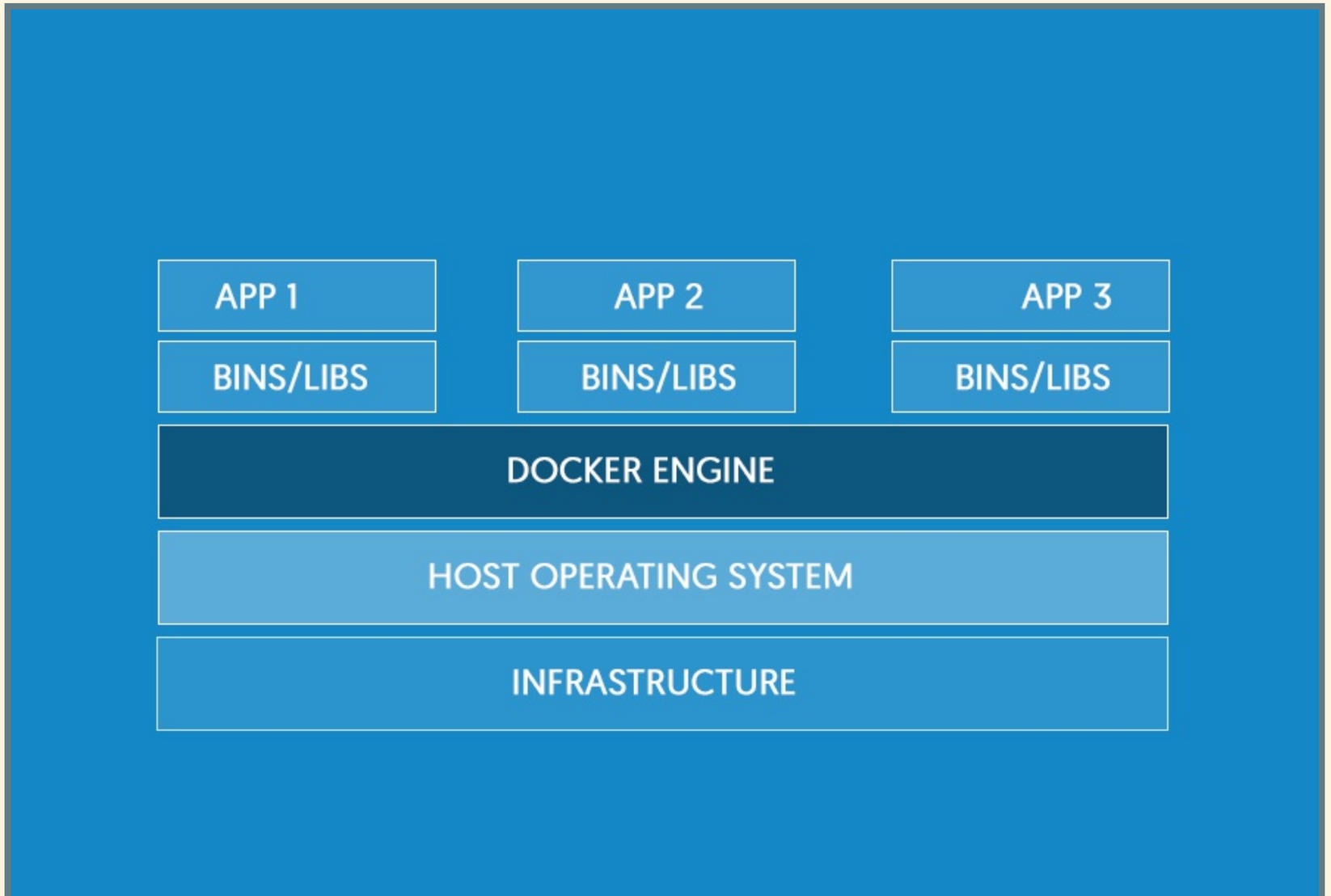
MACHINE VIRTUELLE



VM

- Avantages
  - Emulation bas niveau
  - Sécurité/compartimentation forte hôte/VMs et VMs/VMs
- Inconvénients
  - Usage disque important
  - Impact sur les performances

CONTENEUR DOCKER



CONTENEUR

- Avantages:
  - Espace disque optimisé
  - Impact quasi nul sur les performances CPU, réseau et I/O
- Inconvénients
  - Fortement lié au kernel Hôte
  - Ne peut émuler un OS différent que l’hôte
  - Sécurité

CONTENEUR LINUX

Processus isolé par des mécanismes noyaux.

3 éléments fondamentaux

- Namespaces
- CGroups
- Copy-On-Write

NAMESPACES

FONCTIONNALITÉ DU KERNEL, AVEC SON API

- Apparue dans le noyau 2.4.19 en 2002, réellement utiles en 2013 dans le noyau 3.8
- Limite ce qu'un processus peut voir du système:
  - un processus ne peut utiliser/impacter que ce qu'il peut voir
- Types: pid, net, mnt, uts, ipc, user
- Chaque processus est dans un namespace de chaque type

NET NAMESPACE

Le processus ne voit que la pile réseau du Namespace dont il fait partie:

- ses interfaces (eth0, l0, différentes de l’hôte)
- Table de routage sépararée.
- règles iptables
- socket (ss, netstat)

NAMESPACE UTS

Identification propre au Namespace:

- {get,set} hostname

NAMESPACE IPC

Permettent à un groupe de processus d'un même namespace d'avoir leurs propres:

- ipc semaphore
- ipc message queues
- ipc shared memory

Sans risque de conflit avec d'autres groupes d'autres namespaces

NAMESPACE USER

- mappe uid/gid vers différents utilisateurs de l’hôte
  - uid 0 ⇒ 9999 du C1 correspond à uid 10000⇒ 119999 sur l’hôte
  - uid 0 ⇒ 9999 du C2 correspond à uid 12000⇒ 139999 sur l’hôte

NAMESPACE MOUNT

Un namespace dispose de son propre rootfs
(conceptuellement proche d’un chroot)

- peut masquer /proc, /sys
- peut aussi avoir ses mounts "privés"
  - /tmp (par utilisateur, par service)

NAMESPACE PID

les processus d’un namespace pid ne voit que les processus de celui-ci

- Chaque namespace pid a sa propre numérotation, débutant à 1
- Si le PID 1 disparaît, le namespace est détruit
- Les namespaces peuvent être imbriqués
- Un processus a donc plusieurs PIDs
  - Un pour chaque namespace dans lequel il est imbriqué
- Chaque namespace pid débute avec le PID 1 et lui et les suivants sont les seuls visibles depuis ce namespace.

UTILISATION DES NAMESPACES

- Créés à l'aide de clone() ou unshare()
- Matérialisés par des pseudo-files dans /proc/$PID/ns
- fichiers dans /proc/{pid}/ns

Voir : http://iutsa.unice.fr/~urvoy/ext/Cours/Virtualisation/TPs/docker-lp.pdf

CONTROL GROUPS

CGROUPS LINUX

- Fonctionnalité du noyau, apparue en 2008 (noyau 2.6.24)
- Contrôle les ressources d’un processus:
  - allocation
  - monitoring
  - limite
- type:
  - cpu, memory, network, block io, device

HIÉRARCHIE

Notion de hiérarchie:

- Chaque sous système a une hiérarchie
  - hiérarchies différente pour CPU, memory, block I/O, etc…
- Hiérarchies indépendantes:
  - Les arbres pour MEMORY et CPU sont différents
- Chaque processus est dans un noeud de chaque hiérarchie
- Chaque hiérarchie part d'une racine
- A l'origine chaque processus part de la racine

Chaque noeud équivaut à un groupe de processus * partagent les mêmes ressources

MEMORY CGROUP

- Limites possibles
  - sur la mémoire physique, du noyau et totale
  - Limites soft/hard
- Surveillance de l’utilisation des ressources
- Notifications OOM possibles

CGROUP CPU

- Surveillance du temps CPU utilisateur/système, de l’utilisation par CPU
- Possibilité de définir un poids
- Allocation de processus à un ou plusieurs CPU(s) spécifique(s).

CGROUP BLKIO

- Surveillance des I/O de chaque groupe/device
- Définition de limite d’utilisation par périphérique
- Définition de poids

CGROUP NETWORK

- Définit automatiquement une priorité ou classe pour un trafic, trafic généré par les processus du groupe.
- Fonctionne uniquement pour le trafic généré, sortant.
- net_cls pour assigner une classe (à lier avec iptables)
- net_prio pour assigner une priorité

CGROUP DEVICE

- Contrôle les permissions d'un groupe sur un node device
  - permissions: read/write/mknod
- Typiquement: permettre /dev/{tty,zero,random,null} rejeter tout le reste
- nodes intéressantes:
  - /dev/net/tun (manipulation des interfaces réseau)
  - /dev/fuse (filesystems en espace utilisateurs)
  - /dev/kvm (vm dans des conteneurs)
  - /dev/dri (gpu)

SUBTILITÉS CGROUPS

- Pid 1 est placé à la racine de chaque hiérarchie
- Les nouveaux processus sont démarrés dans le groupe de leur parent
- Les groupes sont matérialisés par des pseudos systèmes de fichiers
  - généralement montés dans /sys/fs/cgroup
- Les groupes sont créés dans ces pseudos systèmes de fichiers:
  - mkdir /sys/fs/cgroup/memory/somegroup/subgroup
- Pour déplacer un processus dans un group:

*** echo $PID > /sys/fs/cgroup/..**
- Différents outils pour faire cela
systemd vs cgmanager vs ...

COPY-ON-WRITE

COW

Si de multiples entités ont besoin de la même ressource, plutôt que de leur donner une copie, on leur donne un pointeur vers celle-ci:

- Réduit l’usage disque et le temps de création
- Plusieurs options disponibles
  - device mapper (niveau fichier)
  - btrfs, zfs (niveau fs)
  - aufs, overlay (niveau fichiers)
- Le type de stockage surveille les modifications
  - lorsque l’appelant modifie sa "copie" de la ressource,

une copie privée spécifique lui est attribuée

DIFFÉRENTES IMPLÉMENTATIONS DE CONTENEURS LINUX

LXC

- Ensemble d’outils en espace utilisateur
- Un conteneur est un dossier dans /var/lib/lxc
- Petit fichier de configuration + root fs
- Les premières versions n’avaient pas de support CoW
- Les premières versions ne supportaient pas le déplacement d’images
- Nécessite beaucoup d’huile de coude

SYSTEMD-NSPAWN

- Pour debugging, testing
- similaire à chroot, mais plus puissant
- Beaucoup de manipulations à faire
- implémente une interface de conteneur
  - support des images docker depuis peu

DOCKER ENGINE

- daemon accessible via une api rest
- Gestion des conteneurs, images, builds, et plus

création image

RKT, RUNC

Retour à la simplicité

- Dédié à l'exécution de conteneur
- Pas d'API Rest, de gestion des images, de build,
- implémentent différentes spécifications
  - rkt implémente appc (app container spec)
  - runc implémente ocp (open container project)
  - runc utilise libcontainer de Docker

LE MEILLEUR?

- Utilisent tous les mêmes fonctionnalités noyau
- Performances globalement équivalentes
- Prendre en compte:
  - Design
  - Fonctionnalités
  - Ecosystème

DOCKER

BANALISER L'UTILISATION DES CONTENEURS

Une plate-forme ouverte pour créer, déployer et exécuter des applications réparties, empaquetées, de manière isolée et portable.

NOUVEAUTÉS

- Standardiser et rendre portable les formats de conteneurs
- Rendre l’utilisation des conteneurs simples pour les développeurs (et pas que)
- API
- Standardiser les outils

EXÉCUTER DES CONTENEURS PARTOUT

- Sur n’importe quelle plate-forme: physique, virtuel, cloud,
- Pouvoir passer de l’une à l’autre des plate-formes,
- Maturité des technologies (cgroups, namespaces, copy-on-write)

EXÉCUTER TOUTE SORTE D'APPLICATION

- Services: web, bdd,
- CLI
- GUI

DISTRIBUTION EFFICACE DES CONTENEURS

- Distribuer des images, au format standard
- Optimiser l’utilisation disque, mémoire et réseau

GENÈSE

DotCloud

- Paas en Python
- Basé sur LXC

PIVOT

Réécriture en Go:

- 03/2013: V0.1
- 06/2014: V1.0

AUJOURD'HUI

- Version 1.9.1 **Séparation docker communiy (CE) et docker entreprise (EE) depuis ~ 2017**
- Docker Engine
  - Démon gérant les conteneurs
- Docker CLI
  - peut gérer un ou plusieurs démon locaux ou distants
- Format standardisé d'images
- Registre et protocole libre pour héberger et distribuer les images de manière optimisée
- Outillage **Builder, Orchestration, cluster**

Source : https://nickjanetakis.com/blog/docker-community-edition-vs-enterprise-edition-and-their-release-cycle
Prior to March 2017, the latest version of Docker was Docker 1.13, and at the time of writing this article, the latest version of Docker is 17.07.

There's a pretty big difference between 1.13 and 17.07, and no, they didn't jump 16 major versions in a few months.

What happened was, on March 2nd 2017, Docker completely changed their version format, and also changed the name of the Docker Engine package to either Docker Community Edition or Docker Enterprise Edition.

CONTENEURS: LES BASES

CONTENEUR BASIQUE

```
$ docker run debian /bin/echo "Salut"
Salut
$ docker run debian /bin/echo "Coucou"
Coucou
```

COMMANDES PS

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS

$ docker ps -a
CONTAINER ID        IMAGE               COMMAND               CREATED             STATUS
d0683f6462a5        debian              "/bin/echo Salut"     About a minute ago  Exited (0) About
e1794g7573b6        debian              "/bin/echo Coucou"    About a minute ago  Exited (0) Abo
```

RÉ-EXÉCUTER UN CONTENEUR

```
$ docker start -i hungry_visvesvaraya
Salut
```

nom donné par docker à la création si vous n'en avez pas mis
Le message ici : un container qui s'est arrêté n'est pas détruit

LOGS

```
$ docker logs hungry_visvesvaraya
Salut
Salut
```

MÉNAGE

```
$ docker rm hungry_visvesvaraya
```

rm uniquement sur un container arrêté! Sinon, il faut d’abord le stopper puis le détruire

CONTENEUR EN COURS D'EXÉCUTION

```
$ docker run -it debian /bin/bash
root@2c666d3ae783:/# ps -a
  PID TTY          TIME CMD
    6 ?        00:00:00 ps
```

AFFICHAGE AVEC PS

Depuis une autre console

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
2c666d3ae783        debian              "/bin/bash"         44 seconds ago      Up 42 seconds
```

DOCKER TOP

```
$ docker top happy_mietner
UID                 PID                 PPID                C                   STIME               TTY                 TIM
root                23338               867                 0                   15:06               pts/15              00:0(
```

numéro des processus dans la machine hôte, pas dans le container où cela recommence à 1.

INTERROMPRE LE CONTENEUR ET SON PROCESSUS

```
$ docker stop $conteneur
$ docker kill $conteneur
$ docker pause $conteneur #unpause
```

Ou tout simplement arrêter le processus (exit pour bash)

SE DÉTACHER

Pour se détacher du processus d’un conteneur sans l’arrêter

```
root@2c666d3ae783:/# ^P^Q
```

SE RATTACHER À UN CONTENEUR

Le conteneur doit être en cours d'exécution, on se rattache au processus exécuté:

```
$ docker attach happy_mietner
root@2c666d3ae783:/#
```

DOCKER EXEC

```
$ docker exec -it happy_mietner  /bin/bash
root@2c666d3ae783:/# exit
```

Le exit ne tuera que le bash en cours

docker exec permet d’exécuter une commande (ici bash mais on aurait pu faire un ifconfig) dans l’espace du container.

IMAGES

NOUS ALLONS VOIR

- Ce qu’est une image
- Ce qu’est une couche
- Les espaces de nom des images
- Rechercher et récupérer des images
- Images tags

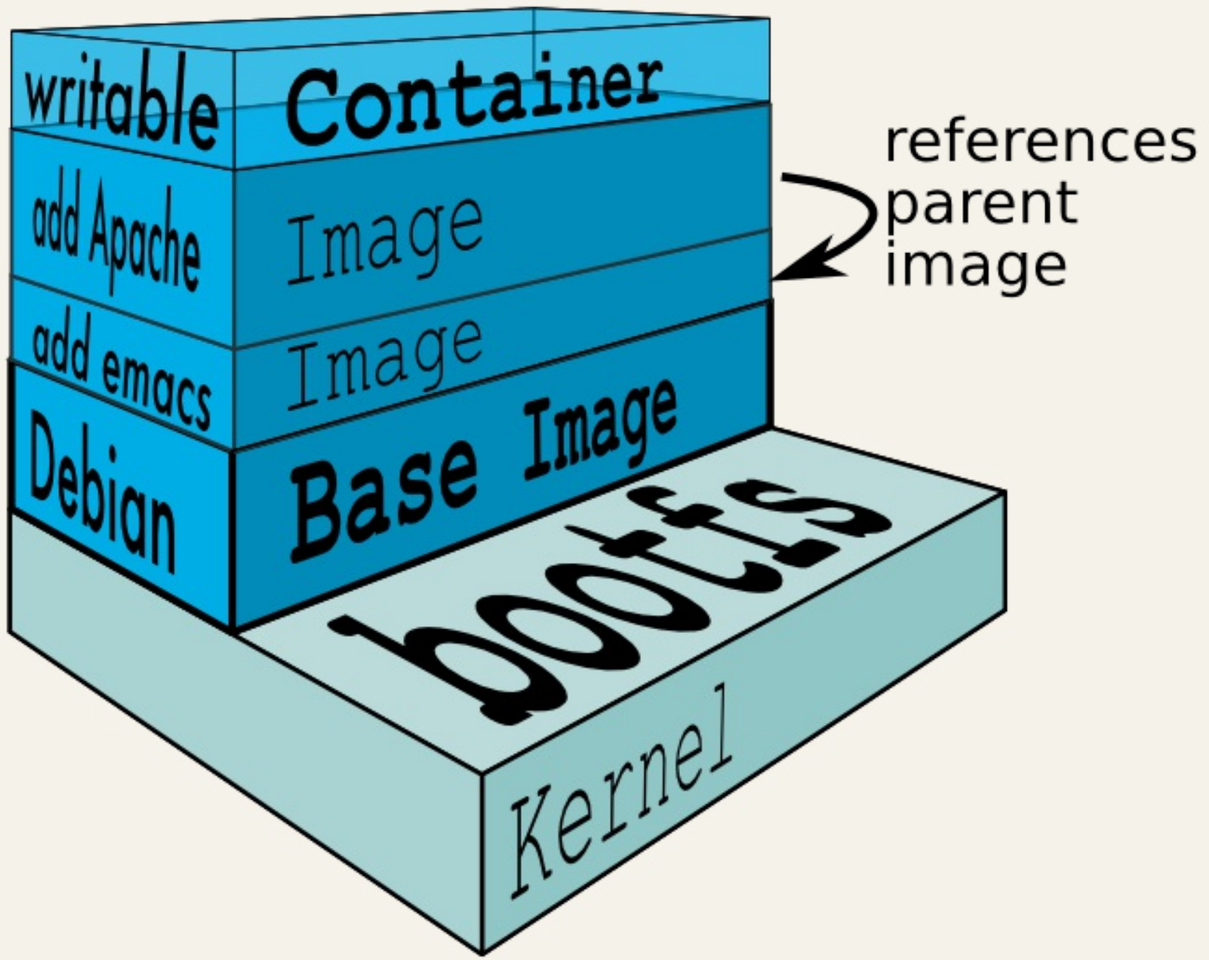
IMAGE?

- Collection de fichiers + méta-données
  - Ces fichiers forment le FS racine du conteneur
- Composées de couches, en théorie superposées
- Chaque couche peut rajouter, modifier, supprimer des fichiers
- Des images peuvent partager des couches pour optimiser
  - L’usage disque
  - Les temps de transfert

DIFFERENCES AVEC LES CONTENEURS

- Systèmes de fichiers lecture-seule
- Un conteneur est un ensemble de processus s’exécutant dans une copie en lecture-écriture de ce système de fichiers
- Pour optimiser les ressources, le CoW est utilisé au lieu de copier le système de fichiers.
- docker run démarre un conteneur depuis une image

COUCHES


writable
Container
add Apache
Image
add emacs
Image
Debian
Base Image
bootfs
Kernel
references
parent
image

MÉTAPHORES

Les images sont des patrons depuis lesquelles vous créez des conteneurs.

En Progammation Orientée Objet:

- Les Images sont conceptuellement similaires aux classes,
- Les couches sont conceptuellement similaires à l’héritage,
- Les Conteneurs sont conceptuellement similaires à des instances d’image.

EXEMPLES IMAGES

- hello-world
- ubuntu
- mysql
- Wordpress
- Application JAVA
- ...

REGISTRE

Héberge les images, et les met à disposition.

Les images sont récupérées en local depuis un registre distant.

Il existe 2 types de registre:

- Docker Hub
- Auto-hébergés

DOCKER HUB

Service en ligne, officiel de Docker, pour distribuer les images

- https://hub.docker.com/
- composants:
  - Un index: indexe toutes les méta-donnés des images hébergées pour recherche.
  - Un registre: stocke les couches des images pour récupération et upload.

Par défaut, les commandes de Docker liées aux images utilisent le Docker Hub.

REGISTRE AUTO-HÉBERGÉ

- Registre hébergé en interne.
- Docker fournit un conteneur pour héberger son propre registre.
- Protocole open-source: différentes implémentations existent.

ESPACES DE NOM

Il existe 3 manières de nommer des images:

- Images officielles:
  - Ex: ubuntu, debian
- Images utilisateur:
  - Ex: benedetti/myapp
- Images auto-hébergées:
  - Ex: registry.example.com:5000/my-private/image

ESPACE DE NOMS RACINE

Cet espace de nom est pour les images distribuées officiellement par Docker, mais généralement créées par des tiers.

Types d’images:

- images de distribution à utiliser comme base: debian, ubuntu
- services prêt à l’emploi: mysql, tomcat

ESPACE DE NOM UTILISATEUR

Images mises à disposition par Docker sans vérification.

- Ex benedetti/mysql ou benedetti/myapp
  - benedetti est mon nom d’utilisateur chez Docker
  - Le nom de l’image est mysql ou myapp

ESPACE DE NOM AUTO-HÉBERGÉ

Images auto-hébergées et distribuées non officiellement.

- Ex: exemple.fr:5000/wordpress
  - exemple.fr:5000 : hôte et port du registre auto-hébergé
  - wordpress: nom de l’image

⇒ vous n’utiliserez sûrement pas de registre et image auto-hébergés.

RECHERCHER UNE IMAGE

Vous pouvez le faire via l'interface web, ou via la CLI:

```
$ docker search nginx
NAME                      DESCRIPTION                                     STARS     OFFICIAL   AUTOMATE
nginx                     Official build of Nginx.                        4172      [OK]
jwilder/nginx-proxy       Automated Nginx reverse proxy for docker c...   800                  [OK]
richarvey/nginx-php-fpm   Container running Nginx + PHP-FPM capable ...   274                  [OK]
million12/nginx-php       Nginx + PHP-FPM 5.5, 5.6, 7.0 (NG), CentOS...   76                   [OK]
maxexcloo/nginx-php       Framework container with nginx and PHP-FPM...   58                   [OK]
webdevops/php-nginx       Nginx with PHP-FPM                              51                   [OK]
```

RÉCUPÉRER UNE IMAGE

Faire un pull pour récupérer l'image en local.

```
$ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
709f78077458: Pull complete
5f5490fb32ee: Pull complete
```

Elle sera ensuite utilisable pour créer un conteneur avec docker run

NB: docker run fait un pull si l'image n'est pas disponible en local

LISTER LES IMAGES

Images disponibles localement

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
debian              latest              93a2e30f1000        3 days ago          123 MB
nginx               latest              e8b9e1a0dfbe        7 days ago          183.5 MB
hello-world         latest              95f1eedc264a        11 weeks ago        1.848 kB
```

TAGS

Les images peuvent avoir des tags

- Un tag définira une version, une variante différente d'une image
- par défaut le tag est latest:
  - docker run ubuntu == docker run ubuntu:latest
- Un tag est juste un alias, un surnom pour un identifiant d'image
  - plusieurs tags différents == une image
  - ex: ubuntu:latest == ubuntu:16.04

EXEMPLE

```
$ docker images debian
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
debian              latest              93a2e30f1000        4 days ago          123 MB
debian              8.5                 f854eed3f31f        3 months ago        125.1 MB
debian              8.4                 32f2a4cccab8        5 months ago        125 MB
debian              8.2                 140f9bdfeb97        8 months ago        125.1 MB
```

EXEMPLE TAG

```
$ docker tag debian benedetti/debian:8.6
docker images | grep debian
debian              latest              93a2e30f1000        4 days ago          123 MB
benedetti/debian    8.6                 93a2e30f1000        4 days ago          123 MB
debian              8.5                 f854eed3f31f        3 months ago        125.1 MB
debian              8.4                 32f2a4cccab8        5 months ago        125 MB
debian              8.2                 140f9bdfeb97        8 months ago        125.1 MB
```

UTILISATION DES TAGS

- On n’utilisera pas les tags
  - durant les tests et prototypage
  - expérimentations
  - quand vous avez simplement besoin de la dernière version
- On les utilisera
  - pour utiliser une image spécifique en production
  - pour créer une image qui évolue

HISTORIQUE D'UNE IMAGE

On peut afficher l'historique de la création d'une image, et des couches qui la constituent:

```
$ docker history debian
IMAGE               CREATED             CREATED BY                                      SIZE                COM
93a2e30f1000        4 days ago          /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
d5daf556aca7        4 days ago          /bin/sh -c #(nop) ADD file:cae7a35a0d8c43d5ba    123 MB
```

HISTORIQUE NGINX

```
$ docker history nginx
IMAGE               CREATED             CREATED BY                                      SIZE                COM
e8b9e1a0dfbe        8 days ago          /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon     0 B
e856b4c52d15        8 days ago          /bin/sh -c #(nop)  EXPOSE 443/tcp 80/tcp        0 B
c7066b7f861d        8 days ago          /bin/sh -c ln -sf /dev/stdout /var/log/nginx/    0 B
69afb4ee6173        8 days ago          /bin/sh -c apt-key adv --keyserver hkp://pgp.    58.36 MB
452019be0ab7        8 days ago          /bin/sh -c #(nop)  ENV NGINX_VERSION=1.11.4-1    0 B
c95559aa6ca8        3 weeks ago         /bin/sh -c #(nop)  MAINTAINER NGINX Docker Ma    0 B
5f5490fb32ee        3 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]             0 B
709f78077458        3 weeks ago         /bin/sh -c #(nop) ADD file:f2453b914e7e026efd    125.1 ME
```

MODIFIER UNE IMAGE

Si une image est en lecture-seule, comment est-ce que l'on la modifie?

- On ne la modifie pas,
- On crée un conteneur depuis cette image,
- On fait nos modifications dans ce conteneur,
- On transforme ces modifications en une couche,
- On crée une nouvelle image en validant cette couche par-dessus celles de l'image de base.

CRÉATION USUELLE D'IMAGES

Deux méthodes:

- docker commit:
  - sauvegarde les modifications apportées à un conteneur dans une nouvelle couche
  - crée l'image
- docker build:
  - séquence d'instructions répétables,
  - Dockerfile
  - Méthode recommandée

CRÉATION D'IMAGE INTERACTIVE

interactive == manuellement:

- On lance un shell dans conteneur
- On fait les modifications voulues:
  - ajout de paquets, de fichiers, etc..
- On commit ces modifications en image
- On peut éventuellement tagger l'image

EN PRATIQUE

Installation de Nginx

```
$ docker run -it debian /bin/bash
root@05739348bc4e:/#
root@05739348bc4e:/# apt update && apt install -y nginx
```

COUCHE COW

```
$ docker diff 05739348bc4e
A /etc/rc1.d/K01nginx
C /etc/default
A /etc/default/nginx
C /etc/ld.so.cache
A /etc/nginx
```

VALIDATION EN UNE COUCHE

```
$ docker commit 05739348bc4e
4f98b27dcc19d60c913ba29516bc31c9b0c80d2ebfdd28f99f43521db3caffed
```

UTILISATION DE NOTRE IMAGE

```
docker run -it 4f98b27dcc19d60c /bin/bash
root@9fc1ee35e2a4:/# nginx -v
nginx version: nginx/1.6.2
```

TAGGER NOTRE IMAGE

```
$ docker tag  4f98b27dcc19d6  benedetti/nginx
$ docker tag  benedetti/nginx benedetti/nginx:0.1
$ docker images benedetti/nginx
REPOSITORY          TAG                 IMAGE ID          CREATED           VIRTUAL SIZE
benedetti/nginx     0.1                 4f98b27dcc19      12 minutes ago    194.3 MB
benedetti/nginx     latest              4f98b27dcc19      12 minutes ago    194.3 MB
$ docker run -it benedetti/nginx nginx -v
nginx version: nginx/1.6.2
```

HISTORIQUE DE NOTRE IMAGE

```
$ docker history benedetti/nginx
IMAGE               CREATED             CREATED BY                                      SIZE                COM
4f98b27dcc19        14 minutes ago      /bin/bash                                       71.33 MB
93a2e30f1000        4 days ago          /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
d5daf556aca7        4 days ago          /bin/sh -c #(nop) ADD file:cae7a35a0d8c43d5ba    123 MB
```

CRÉATION INTERACTIVE

- Intérêt
  - manipulation usuelle
  - bien pour tester rapidement quelque chose
- Désavantage
  - Manuelle
  - Non répétable

CRÉATION AUTOMATISÉE D'IMAGE

- Ecrire un Dockerfile
- Créer l'image à partir du Dockerfile

DOCKERFILE

Principe:

- Recette automatisée de création d’images
- Contient une suite d’instructions
- La commande docker build utilise le Dockerfile pour créer l’image

NOTRE PREMIER DOCKERFILE

On travaille dans un dossier qui va contenir le Dockerfile propre à notre future image:

```
$ mkdir -p ~/docker/nginx
```

On se place dans ce dossier et on ouvre un fichier Dockerfile

```
$ cd ~/docker/nginx/
$ edit Dockerfile
```

Contenu du Dockerfile:

```
FROM debian
RUN apt-get update
RUN apt-get install -y nginx
```

INSTRUCTIONS

Un Dockerfile est composé d’instruction, une par ligne.

- FROM: image de base à utiliser pour notre future image
  - Un seul FROM par Dockerfile
- RUN: commande shell à exécuter
  - seront exécutées durant le processus de build
  - utilisable à volonté
  - non-interactive: aucun input possible durant le build

BUILD IT!

Depuis le dossier contenant le Dockerfile

```
$ docker build -t benedetti/nginx:0.2 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM debian
---> 93a2e30f1000
Step 2 : RUN apt-get update
---> Running in 20f50a8284f5
 Get:1 http://security.debian.org jessie/updates InRelease [63.1 kB]
...
Processing triggers for sgml-base (1.26+nmu4) ...
---> 95f9e15fa8d2
 Removing intermediate container e99f0c6f5bd2
 Successfully built 95f9e15fa8d2
```

RE BUILD IT!

Si on relance le build, il sera instantané

- A chaque étape, Docker prend un instantané dans un conteneur
- Avant d’exécuter une étape, Docker vérifie s’il n’a pas déjà exécuté cette séquence

EXÉCUTION

L’image obtenue permet de démarrer un conteneur, de manière similaire à celle créée manuellement:

```
$ docker run -it benedetti/nginx:0.2 nginx -v
nginx version: nginx/1.6.2
```

HISTORIQUE DIFFÉRENT

L’historique entre une image créée interactivement et manuellement diffère néanmoins:

```
$ docker history benedetti/nginx:0.2
IMAGE               CREATED             CREATED BY                                      SIZE                COM
95f9e15fa8d2        7 minutes ago       /bin/sh -c apt-get install -y nginx             61.5 MB
b8b4c747628c        10 minutes ago      /bin/sh -c apt-get update                       9.828 MB
93a2e30f1000        4 days ago          /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
d5daf556aca7        4 days ago          /bin/sh -c #(nop) ADD file:cae7a35a0d8c43d5ba    123 MB
```

CMD

Avec l'instruction CMD, on peut définir une commande à exécuter par défaut lorsque l'on lance un conteneur.

Par exemple:

```
$ docker run -it benedetti/nginx
nginx version: nginx/1.6.2
```

on obtient la version du nginx embarqué dans le conteneur à cause du build…transparent suivant

UTILISATION DE CMD

```
FROM debian
RUN apt-get update
RUN apt-get install -y nginx
CMD nginx -v
```

BUILD ET TEST DE CMD

```
$ docker build -t benedetti/nginx:0.3 .
$ docker run -it benedetti/nginx:0.3
nginx version: nginx/1.6.2
```

OUTREPASSER CMD

```
$ docker run -it benedetti/nginx:0.3 echo salut
salut
```

ENTRYPOINT

- Définit une commande de base à exécuter par le conteneur,
- Les paramètres de la ligne commande sont ajoutés à ces paramètres.

UTILISATION DE ENTRYPOINT

```
FROM debian
RUN apt-get update
RUN apt-get install -y nginx
ENTRYPOINT ["nginx", "-g"]
```

BUILD AVEC ENTRYPOINT

```
$ docker build -t benedetti/nginx:0.4 .
```

EXÉCUTION DE ENTRYPOINT

```
$ docker run -it benedetti/nginx:0.4 "param bidon;"
nginx: [emerg] unknown directive "param" in command line
$ docker run -it benedetti/nginx:0.4 "daemon off;"
#nginx s'exécute en avant plan
```

Depuis un autre terminal:

```
$ docker ps
CONTAINER ID        IMAGE                 COMMAND                  CREATED              STATUS
29e86a41e506        benedetti/nginx:0.4   "nginx -g 'daemon off"   About a minute ago   Up About
```

CMD ET ENTRYPOINT

```
FROM debian
RUN apt-get update
RUN apt-get install -y nginx
ENTRYPOINT ["nginx", "-g"]
CMD ["daemon off;"]
```

Source : https://stackoverflow.com/questions/21553353/what-is-the-difference-between-cmd-and-entrypoint-in-a-dockerfile#21564990

Docker has a default entrypoint which is /bin/sh -c but does not have a default command.

When you run docker like this: docker run -i -t ubuntu bash the entrypoint is the default /bin/sh -c, the image is ubuntu and the command is bash.

The command is run via the entrypoint. i.e., the actual thing that gets executed is /bin/sh -c bash. This allowed Docker to implement RUN quickly by relying on the shell's parser.

Later on, people asked to be able to customize this, so ENTRYPOINT and --entrypoint were introduced.

Everything after ubuntu in the example above is the command and is passed to the entrypoint. When using the CMD instruction, it is exactly as if you were doing docker run -i -t ubuntu <cmd>. <cmd> will be the parameter of the entrypoint.

You will also get the same result if you instead type this command docker run -i -t ubuntu. You will still start a bash shell in the container because of the ubuntu Dockerfile specified a default CMD: CMD ["bash"]

BUILD CMD ET ENTRYPOINT

```
$ docker build -t benedetti/nginx:0.5 .
```

EXÉCUTION CMD ET ENTRYPOINT

```
$ docker run -d benedetti/nginx:0.5
10bb961dfe60fc4c92b45f6a1a390f62f7edc0f6d78fe2088a0cf08c6d6cb040
```

Nous avons un nginx qui tourne: comment y accéder?

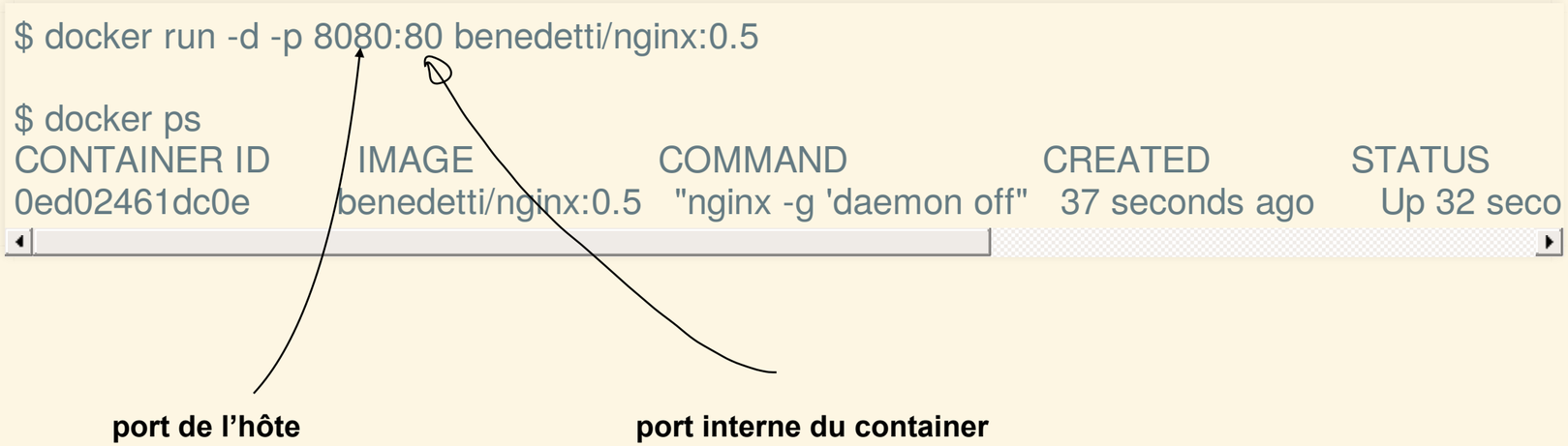
EXPOSER LES PORTS D'UN CONTENEUR

- Tous les ports sont privés par défaut
  - Un port privé n'est pas accessible de l'extérieur
- C'est au client à rendre publics ou non les ports exposés
  - Public: accessible par d'autres conteneurs et en dehors de l'hôte.

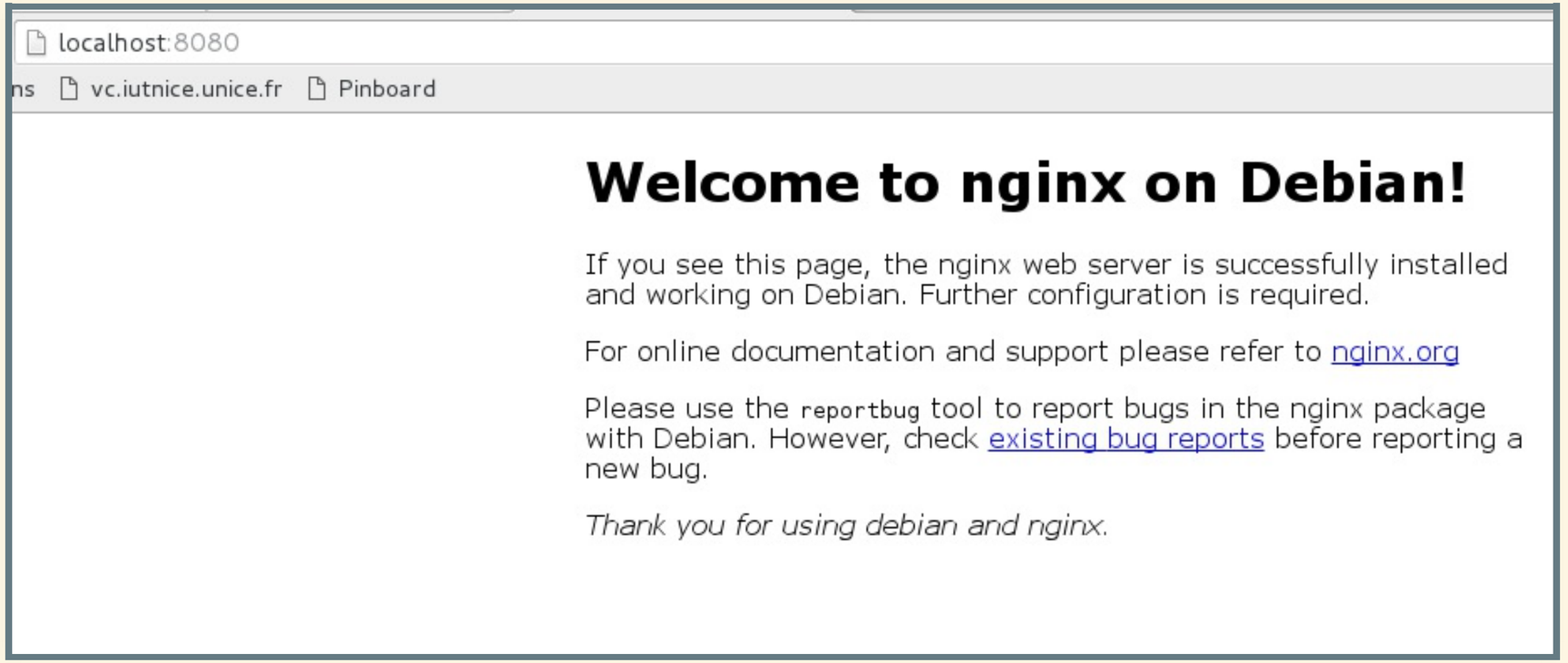
EXPOSER UN PORT VIA LA CLI

```
$ docker run -d -p 8080:80 benedetti/nginx:0.5

$ docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
0ed02461dc0e        benedetti/nginx:0.5    "nginx -g 'daemon off"   37 seconds ago      Up 32 seco
```



ACCÈS WEB


localhost:8080
ns
vc.iutnice.unice.fr
Pinboard
Welcome to nginx on Debian!
If you see this page, the nginx web server is successfully installed and working on Debian. Further configuration is required.
For online documentation and support please refer to nginx.org
Please use the reportbug tool to report bugs in the nginx package with Debian. However, check existing bug reports before reporting a new bug.
Thank you for using debian and nginx.

EXPOSE

Instruction Dockerfile qui indique à Docker quel(s) port(s) publier pour notre image.

Ces ports seront automatiquement exposés avec l’option -P au lancement d’un conteneur.

EXPOSE ET DOCKERFILE

```
FROM debian
RUN apt-get update
RUN apt-get install -y nginx
EXPOSE 80 443
ENTRYPOINT ["nginx", "-g"]
CMD ["daemon off;"]
```

BUILD AVEC EXPOSE

```
$ docker build -t benedetti/nginx:0.6 .
```

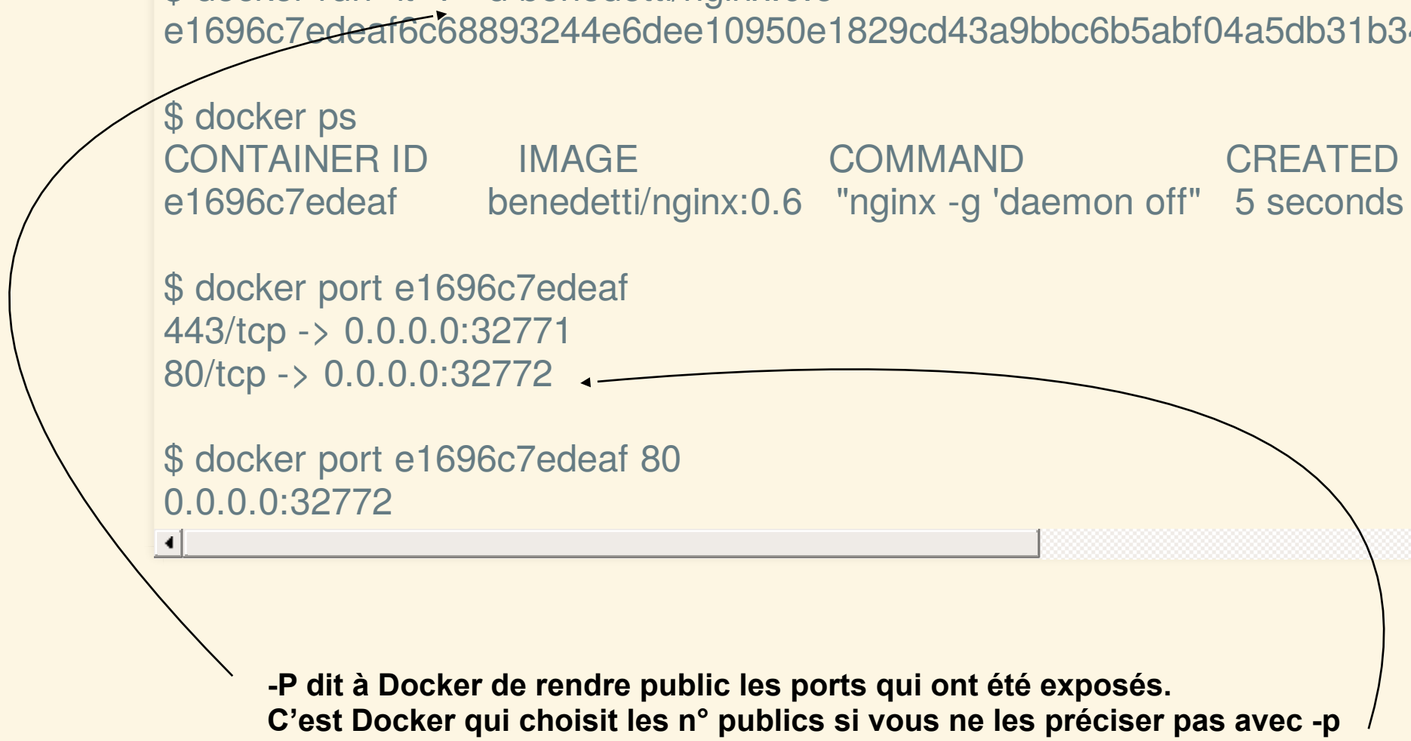
EXÉCUTION AVEC -P

```
$ docker run -it -P -d benedetti/nginx:0.6
e1696c7edeaf6c68893244e6dee10950e1829cd43a9bbc6b5abf04a5db31b341

$ docker ps
CONTAINER ID        IMAGE                 COMMAND                  CREATED             STATUS
e1696c7edeaf        benedetti/nginx:0.6   "nginx -g 'daemon off"   5 seconds ago       Up 2 second

$ docker port e1696c7edeaf
443/tcp -> 0.0.0.0:32771
80/tcp -> 0.0.0.0:32772

$ docker port e1696c7edeaf 80
0.0.0.0:32772
```



-P dit à Docker de rendre public les ports qui ont été exposés.
C'est Docker qui choisit les n° publics si vous ne les préciser pas avec -p

TEST DE EXPOSE

```
$ curl http://localhost:32772
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx on Debian!</title>
<style>
```

COPY

L’instruction COPY permet de copier fichiers et dossiers depuis le contexte de génération, dans le conteneur.

Imaginons que l’on veuille modifier la page d’accueil de notre serveur Nginx?

DOCKERFILE AVEC COPY

```
$ echo "Bienvenue sur mon image Nginx" > index.html
```

```
FROM debian
RUN apt-get update
RUN apt-get install -y nginx
EXPOSE 80 443
COPY index.html /var/www/html/index.html
ENTRYPOINT ["nginx", "-g"]
CMD ["daemon off;"]
```

BUILD AVEC COPY

```
$ docker build -t benedetti/nginx:0.7 .
```

TEST DE COPY

```
$ docker run -it -P -d benedetti/nginx:0.7
87987bf2e06f0d14dcbf6d9d65eabb5adc34b5b56bc95f195b564b52de6f0a39

$ docker port 87987bf2 80
0.0.0.0:32776

$ curl http://localhost:32776
Bienvenue sur mon image Nginx
```

ADD

L’instruction ADD fonctionne comme COPY, avec des fonctionnalités en plus:

- Peut récupérer des fichiers en ligne (URL http://)
- Peut automatiquement décompresser des zip locaux

VOLUMES

Problèmes:

- Si je veux modifier index.html, je dois regénérer une image
  - laborieux surtout en phase de test
- Nginx génére des logs
  - Ces modifications engendrent des données dans une couche
  - Je voudrais les partager avec un autre serveur

VOLUME

- Les volumes peuvent être partagés:
  - entre conteneurs
  - entre hôte et un conteneur
- Les accès au système de fichiers via un volume outrepassent le CoW:
  - Meilleures performances
  - Ne sont pas enregitrés dans une couche pour ne pas être enregistrés par un docker commit

NOTRE PREMIER VOLUME

```
$ docker run -d -v $(pwd):/var/www/html -P benedetti/nginx:0.7
86bc6648b0bb2423adbb20c7dcdd6b3b27d2c4c5670a9330cc7571c2ec35be42

$ docker exec -it 86bc6648b0bb2423adbb20c7d ls /var/www/html
Dockerfile  index.html

$ docker port 86bc6648b0bb2423 80
0.0.0.0:32780

$ curl http://localhost:32780
Bienvenue sur mon image Nginx

$ echo "Mise à jour du fichier index.html" > index.html

$ curl http://localhost:32780
Mise à jour du fichier index.html
```

CRÉATION DE VOLUME NOMMÉ

```
$ docker volume create --name=logs
logs

$ docker run -P -v logs:/var/log/nginx -d benedetti/nginx:0.7
055ac104acf1d734ae38005e96512f666cbe483b1db87b653648d045c2c1a744

$ docker run -it --volumes-from 055ac104acf1d73 debian ls /var/log/nginx
access.log  error.log
```

PERSISTENCE

- Les volumes existent indépendamment des conteneurs
- Si un conteneur est stoppé, ses volumes sont encore disponibles
- Vous êtes responsable de la gestion, de la sauvegarde des volumes

LISTER LES VOLUMES

```
docker volume ls
DRIVER              VOLUME NAME
local               57a0848c5e5f2924be84a66157e84e830757922c7b5b856aa5bac12e494da495
local               logs
```

On peut monter ces volumes depuis un autre conteneur

MÉNAGE VOLUME

```
$ docker rm 055ac104acf1d734
$ docker volume ls -f dangling=true | grep logs
$ docker volume rm logs
```

BONUS MÉNAGE

Supprimer tous les volumes non montés (Danger!)

```
$ docker volume rm $(docker volume ls -qf dangling=true)
```

CONCLUSION

DOCKER

- Basé sur des technologies éprouvées
- Des outils, un éco-système

AVEC DOCKER...

..je peux:

- Mettre mes applicatifs dans des conteneurs,
- Exécuter ces conteneurs (presque) partout,
- Créer et automatiser la génération d’images,
- Héberger et partager mes images avec un Registre.

RÉFÉRENCES

- Documentation officielle
- Vidéos d’apprentissage officielles
- Jérôme Petazzoni - Introduction to Docker and containers - PyCon 2016
- Jérôme Petazzoni: Cgroups, namespaces, and beyond: what are containers made from? - PyCon 2016

ORCHESTRATION DOCKER

Benoît Benedetti

PROBLÉMATIQUES

Jusqu'à présent, nous avons travaillé avec une application monolithique, sur un seul hôte Docker.

Nous voulons déployer et gérer des applications de type micro-services, sur plusieurs hôtes.

PLAN DE CETTE SÉANCE

- Réseau
- Docker Compose
- Docker Machine
- Docker Swarm

RÉSEAU

INTRODUCTION

Nous avons déjà vu que les conteneurs pouvaient exposer leur port.

Docker propose d’autres moyens pour interconnecter des conteneurs:

- La fonctionnalité network, nouvelle
- Les liens, historiques

NETWORK

Quand vous installez Docker, 3 réseaux sont créés automatiquement, bridge, none, et host, suivant 3 pilotes bridge, null et host.

```
$ docker network ls
NETWORK ID          NAME                DRIVER
7fca4eb8c647        bridge              bridge
9f904ee27bf5        none                null
cf03ee007fb4        host                host
```

NONE ET HOST

- none:
  - type null: auun réseau pour un conteneur sur un réseau de ce type
- host
  - type host: stack réseau identique à l'hôte

Vous n'aurez sûrement jamais à utiliser ces réseaux, et créer des réseaux de ces types.

PILOTE ET RÉSEAU BRIDGE

Le pilote bridge interconnecte les conteneurs qui se trouvent sur un réseau de ce type de pilote.

- Vous pouvez exposer des ports sur ce type de réseau
- Les conteneurs doivent tous s'exécuter sur l'hôte du réseau (mono-hôte) Par défaut, le démon Docker connecte vos conteneurs dans le réseau bridge.

CRÉATION DE RÉSEAU

Vous pouvez créer des réseaux. Docker propose deux pilotes pour les créer:

- Bridge
- Overlay:
  - équivalent à bridge mais multi-host

EXEMPLE

```
$ docker network create -d bridge my-bridge-network
$ docker run -d --network=my-bridge-network --name db training/postgres
```

COMPOSE

BILAN

- On sait créer des images
  - de manière manuelle
  - de manière automatisée
- On sait lancer des conteneurs
  - partager les données avec des volumes
  - les interconnecter sur le réseau

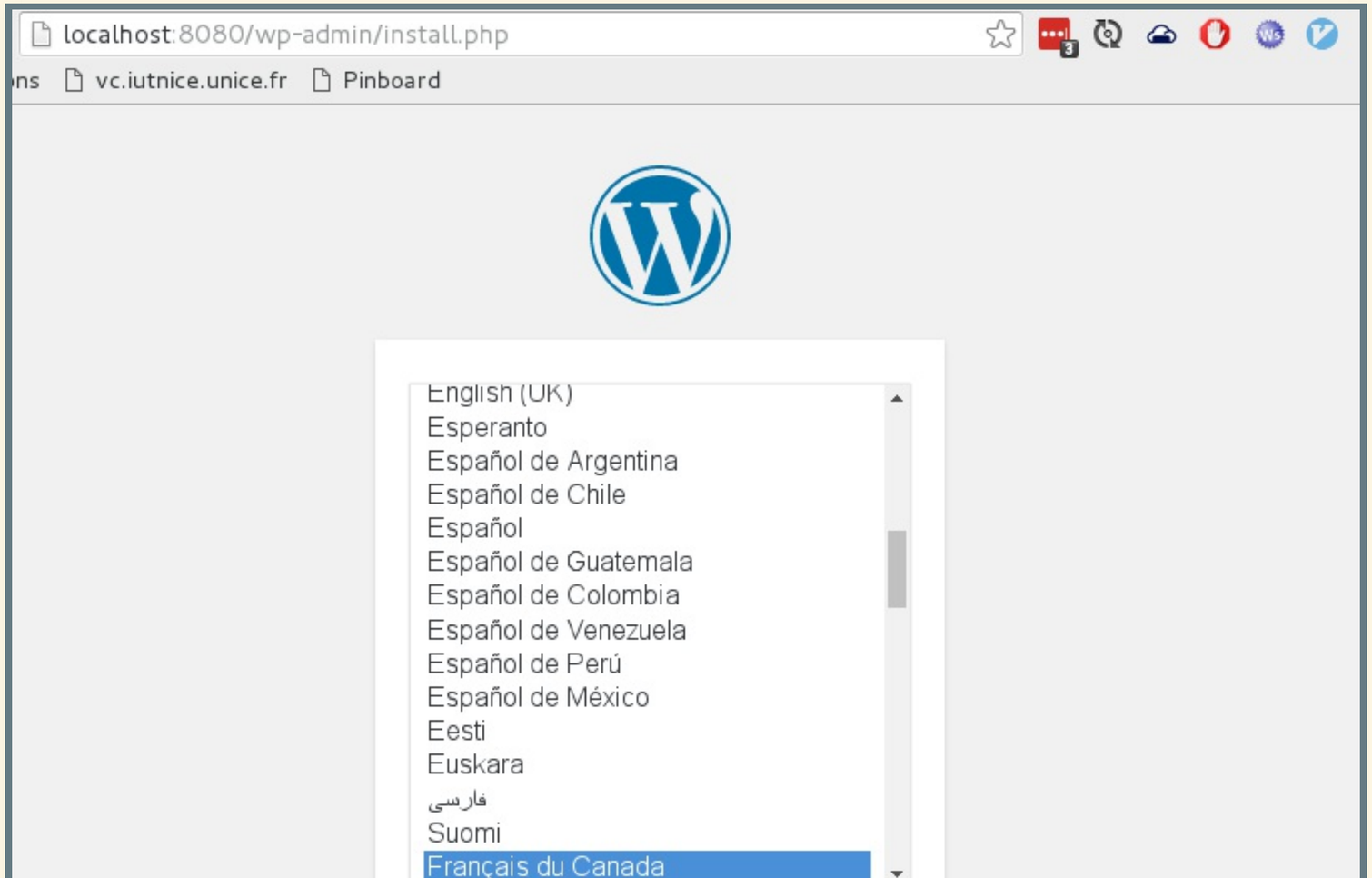
PROBLÈME

- On veut coordonner des conteneurs
- On veut simplifier la gestion multi-conteneurs
- On ne veut pas utiliser de scripts shell complexes
- On veut une interface standardisée avec l’API Docker

EXEMPLE: WORDPRESS

-e : permet le passage de paramètre
- d expose un volume
-- link récupère le volume d'un autre container

```
$ docker run --name db -e MYSQL_ROOT_PASSWORD=secret -d mysql
$ docker run --name wp -p 8080:80 --link db:mysql -d wordpress
```



COMMENT FAIRE POUR

- Gérer les deux conteneurs à la volée?
- Gérer des volumes?
- Gérer des ports différents?
- Me souvenir de ces commandes?

SOLUTION

Compose vous permet d’éviter de gérer individuellement des conteneurs qui forment les différents services de votre application.

- Outil qui définit et exécute des applications multi-conteneurs
- Utilise un fichier de configuration dans lequel vous définissez les services de l’application
- A l’aide d’une simple commande, vous contrôlez le cycle de vie de tous les conteneurs qui exécutent les différents services de l’application.

UTILISATION

- Vous définissez l'environnement de votre application pour qu'il soit possible de la générer de n'importe où
  - à l'aide de Dockerfile
  - à l'aide d'image officielle
- Vous définissez vos services dans un fichier docker-compose.yml pour les exécuter et les isoler.
- Exécutez docker-compose qui se chargera d'exécuter l'ensemble de votre application

SERVICES

Compose introduit une notion de service:

- Concrétement, un conteneur exécutant un processus
- Chaque conteneur exécute un service inter-dépendant
- Le service peut-être évolutif en lançant plus ou moins d’instances du conteneur avec Compose.

Exemple Wordpress:

- Service db
- Service wordpress

EXEMPLE DOCKER-COMPOSE.YML

```
version: '2'
services:
  db:
    image: mysql:5.7
    volumes:
      - "./.data/db:/var/lib/mysql"
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    links:
```

DÉMARRAGE D'UNE APPLICATION

```
$ docker-compose up -d
Creating wordpress_db_1
Creating wordpress_wordpress_1
```

Les différents services qui composent mon application ont été démarrés, avec la configuration et l'environnement qui va bien.

INFORMATION DE MON APPLICATION

On utilise la commande ps de Compose:

```
$ docker-compose ps
        Name                      Command               State          Ports
-------------------------------------------------------------------------------------
wordpress_db_1           docker-entrypoint.sh mysqld       Up       3306/tcp
wordpress_wordpress_1    /entrypoint.sh apache2-for ...    Up       0.0.0.0:8000->80/tcp

$ docker-compose ps wordpress
        Name                      Command               State          Ports
-------------------------------------------------------------------------------------
wordpress_wordpress_1    /entrypoint.sh apache2-for ...    Up       0.0.0.0:8000->80/tcp
```

CONTENEURS CLASSIQUES

Les services s’exécutent via des conteneurs sur l’hôte. Les commandes docker classiques sont toujours fonctionnelles.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
8d086aeba614        wordpress:latest    "/entrypoint.sh apach"   7 minutes ago       Up 7 minutes
689405bb755d        mysql:5.7           "docker-entrypoint.sh"   7 minutes ago       Up 7 minutes
```

LOGS

Logs d’une application:

```
$ docker-compose logs
```

Logs d’un service

```
$ docker-compose logs db
```

PASSAGE À L'ÉCHELLE

On peut passer à l'échelle un service. Autrement dit, on peut augmenter/diminuer le nombre de conteneurs exécutant un service

Par défaut, Compose exécute chaque service avec un conteneur.

SCALE

On utilise la commande scale pour changer le nombre de réplicas d’un service:

```
$ docker-compose scale db=3
$ docker-compose ps db
     Name                  Command             State    Ports
-------------------------------------------------------------------
wordpress_db_1   docker-entrypoint.sh mysqld   Up      3306/tcp
wordpress_db_2   docker-entrypoint.sh mysqld   Up      3306/tcp
wordpress_db_3   docker-entrypoint.sh mysqld   Up      3306/tcp
$ docker-compose scale db=2
```

CYCLE DE VIE

```
$ docker-compose up
$ docker-compose stop #ou kill
$ docker-compose start
$ docker-compose restart
$ docker-compose rm
$ docker-compose down # == stop + rm
```

le kill est à faire dans le répertoire du .yml

RÉSUMÉ COMPOSE

- Compose est un outil pour définir, lancer et gérer des services qui sont définis comme une ou plusieurs instances d'un conteneur,
- Compose utilise un fichier de configuration YAML comme définition de l'environnement,
- Avec docker-compose on peut générer des images, lancer et gérer des services, …
- Certaines commandes de docker-compose sont équivalentes à l'outil docker, mais s'appliquent seulement aux conteneurs de la configuration de compose.

DOCKER MACHINE

PROBLÉMATIQUE

- Je veux pouvoir déployer des hôtes Docker à la volée.
- Je veux les utiliser de manière transparente
- Je veux configurer mon client docker facilement pour utiliser tel ou tel hôte Docker.

PRINCIPE

Outil en CLI qui vous permet d’installer Docker Engine sur des hôtes virtuels, et de les administrer avec les commandes docker-machine.

Vous pouvez l’utiliser pour créer des hôtes Docker sur votre Linux, Windows ou Mac local, sur votre réseau, dans votre datacenter, ou sur un fournisseur cloud comme Amazon AWS.

DOCKER-MACHINE

A l’aide des commandes docker-machine vous pouvez démarrer, inspecter, stopper et mettre à jour un serveur Docker, et configurer votre client local pour utiliser cet hôte. Vous pourrez ensuite utiliser docker run, docker ps, etc., comme d’habitude.

CRÉATION D'UNE MACHINE

La commande la plus importante à connaître est celle de création d'une machine:

```
$ docker-machine create --driver virtualbox host1
```

EFFET

La commande précédente a pour effet de:

- Créer un dossier de configuration pour chaque machine (~/.docker/machine/machines/host1)
- Créer une machine (virtuelle, locale, …) suivant le pilote utilisé
- D’y installer Docker
  - VirtualBox= Boot2Docker
  - Cloud: Ubuntu
- De configurer les clés ssh pour utiliser notre machine

GESTION D'UNE MACHINE

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                       SWARM   DOCKER    ERRORS
host1   -        virtualbox   Running   tcp://192.168.99.108:2376         v1.12.1

$ docker-machine ssh host1
$ docker-machine inspect host1

$ docker-machine ip host1
192.168.99.108
```

UTILISATION

99% des interactions avec un hôte Docker seront des opérations via les clients Docker. **i.e. les commandes Docker**

Inutile de se connecter en SSH.

Docker fournit un moyen de configurer son client Docker pour utiliser une machine.

CONFIGURATION DU CLIENT

Il faut configurer des variables d'environnement. Pour les connaître, on exécute:

```
$ docker-machine env host1
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.108:2376"
export DOCKER_CERT_PATH="/home/benben/.docker/machine/machines/host1"
export DOCKER_MACHINE_NAME="host1"
# Run this command to configure your shell:
# eval $(docker-machine env host1)
```

RÉSULTAT

```
$ eval $(docker-machine env host1)
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                       SWARM   DOCKER    ERRORS
host1   *        virtualbox   Running   tcp://192.168.99.108:2376           v1.12.1
```

Machine host1 est désormais active.

UTILISATION

Maintenant que votre client est configuré, l'utilisation de la machine active se fait de manière transparente:

- Utilisation du client docker
- Utilisation de docker-compose
- etc

RÉ-INITIALISER LE CLIENT

Je veux ré-initialiser le client Docker pour utiliser l'hôte local:

```
$ docker-machine env -u
unset DOCKER_TLS_VERIFY
unset DOCKER_HOST
unset DOCKER_CERT_PATH
unset DOCKER_MACHINE_NAME
# Run this command to configure your shell:
# eval $(docker-machine env -u)

$ eval $(docker-machine env -u)
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                       SWARM   DOCKER    ERRORS
host1   -        virtualbox   Running   tcp://192.168.99.108:2376           v1.12.1
```

RÉSUMÉ MACHINE

Machine permet de:

- Créer des hôtes Docker répartis
  - Les utiliser de manière transparente
  - Configurer rapidement le client Docker pour utiliser l'un ou l'autre
- Inconvénients:
  - On ne peut utiliser qu'un hôte à la fois
  - Il faut donc mettre à jour la configuration du client pour utiliser un hôte

SWARM

PROBLÈME

Vous avez plusieurs hôtes Docker.

Vous désirez les utiliser sous forme de cluster, et répartir de manière transparente l'exécution de conteneur.

SOLUTION

Docker Engine 1.12 inclut le mode swarm pour nativement gérer un cluster de Docker Engines qu'on nomme un swarm (essaim).

On utilise la CLI Docker pour créer un swarm, déployer des service d'application sur un swarm, et gérer le comportement de votre swarm.

MODE SWARM

- Les fonctionnalités de gestion et orchestration de cluster incluses dans le Docker Engine.
- Les Moteurs Docker participant à un cluster s'exécutent en mode swarm.
- Un swarm (essaim) est un cluster de Docker Engines sur lequel vous déployez des services.
- La CLI Docker inclut la gestion des noeuds d'un swarm
  - ajout de noeuds,
  - déploiement de services,
  - gestion de l'orchestration des services

NOEUD

Un noeud est une instance Docker Engine participant à un swarm.

Noeud Worker:
*** reçoit et exécute les tâches depuis les managers**
*** un manager est également un worker**
*** Notifie les managers de son état pour l'orchestration**

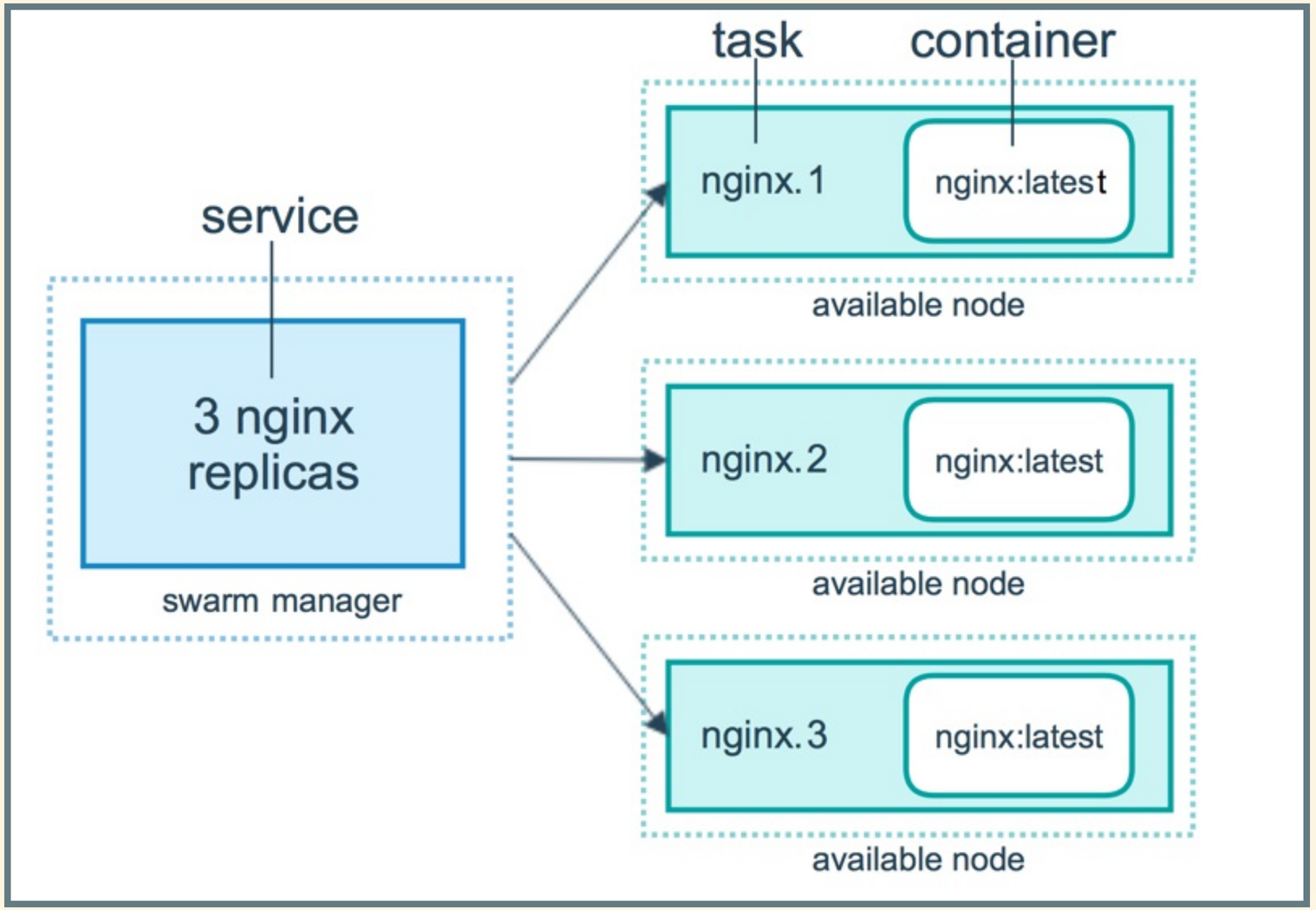
- Noeud de type manager:
  - déploie les applications suivant les définitions de services que vous lui soumettez,
  - dispatche les tâches au noeud de type worker,
  - Gestion du cluster, orchestration
  - Un leader est choisi parmi les managers pour gérer les tâches d’orchestration
- Noeud Worker:

SERVICES ET TÂCHES

- Un service est la définition de tâches à exécuter par les workers
  - exécution d’une commande via un conteneur
  - utilise une image
- deux modes d’exécution de service:
  - repliqué: un manager distribue un nombre donné de tâches sur chaque noeud
  - global: exactement une tâche est exécuté par noeud
- tâche: unité atomique d’exécution d’un swarm
  - représente le conteneur et la commande à y exécuter

tâche: unité atomique d'exécution d'un swarm
*** représente le conteneur et la commande à y exécuter**
*** assigné à un worker par un manager suivant le nombre de réplica défini par le service**
*** ne peut changer de noeud, s'exécute sur ce noeud ou échoue.**

DIAGRAMME DES SERVICES ET TÂCHES



RÉPARTITION DE CHARGE

- Le manager utilise une répartition de charge vers tous les workers pour exposer les ports des services
  - Le port rendu public est également accessible sur tout worker du swarm
- Chaque service du swarm à son propre nom DNS interne:
  - Le manager utilise une répartition de charge interne pour distribuer les requêtes des services du cluster.

COMMANDE NODE

```
$ docker node ls
Error response from daemon: This node is not a swarm manager. [...]
```

Un cluster est initialisé avec docker swarm init. A exécuter une fois sur un hôte.

CRÉER NOTRE SWARM

```
$ docker swarm init --advertise-addr <MANAGER-IP>
Swarm initialized: current node (8jud...) is now a manager.
To add a worker to this swarm, run the following command:
  docker swarm join \
  --token SWMTKN-1-59fl4ak4nqjmao1ofttrc4eprhrola2l87... \
  172.31.4.182:2377
```

Dans la sortie de la commande, un message nous indique la commande à exécuter pour ajouter un worker à notre nouveau swarm.

VÉRIFIER L'ÉTAT DE NOTRE SWARM

On utilise la classique commande docker info:

```
$ docker info
Swarm: active
 NodeID: 8jud7o8dax3zxbags3f8yox4b
 Is Manager: true
 ClusterID: 2vcw2oa9rjps3a24m91xhvv0c
```

PREMIÈRE COMMANDE EN MODE SWARM

Pour voir les informations des noeuds du swarm:

```
$ docker node ls
ID            HOSTNAME          STATUS  AVAILABILITY  MANAGER STATUS
8jud...ox4b *  ip-172-31-4-182  Ready   Active        Leader
```

SOUS LE CAPOT

lors du docker swarm init, un certificat Racine TLS a été créé.
Puis une paire de clés pour notre premier noeud, signée avec le certificat.

Pour chaque nouveau noeud sera créée sa paire de clé signée avec le certificat.

Toutes les communications sont ainsi chiffrées en TLS.

TOKEN

Docker a généré 2 tokens de sécurité (équivalent d'une passphrase ou password) pour notre cluster, à utiliser lors de l'ajout de nouveaux noeuds:

- Un token pour les workers
- Un token pour les managers

Récupération des tokens:

```
$ docker swarm join-token worker
$ docker swarm join-token manager
```

AJOUT D'UN WORKER AU CLUSTER

Se connecter à un autre serveur Docker:

```
$ docker swarm join \
  --token TOKEN-WORKER... \
  172.31.4.182:2377
```

CLUSTER DE 2 NOEUDS

```
$ docker node ls
ID            HOSTNAME          STATUS  AVAILABILITY  MANAGER STATUS
8jud...ox4b *  ip-172-31-4-182  Ready   Active        Leader
ehb0...4fvx    ip-172-31-4-180  Ready   Active
```

AJOUT D'UN MANAGER

```
$ docker swarm join \
  --token TOKEN_MANAGER... \
  172.31.4.182:2377
```

CLUSTER DE 3 NOEUDS

```
$ docker node ls
ID            HOSTNAME          STATUS  AVAILABILITY  MANAGER STATUS
8jud...ox4b *  ip-172-31-4-182  Ready   Active        Leader
abcd...1234    ip-172-31-4-181  Ready   Active        Manager
ehb0...4fvx    ip-172-31-4-180  Ready   Active
```

PROMOUVOIR UN WORKER EN MANAGER

```
$ docker node promote NODE
```

Inverse: demote

QUITTER UN SWARM

```
$ docker swarm leave node-2
$ docker node rm node-2
```

EXÉCUTER UN SERVICE

On utilise la commande service sur un manager en mode Swarm:

```
$ docker service create --replicas 1 --name helloworld alpine ping docker.com
9uk4639qpg7npwf3fn2aasksr
```

LISTER LES SERVICES

```
$ docker service ls
ID            NAME        SCALE  IMAGE   COMMAND
9uk4639qpg7n  helloworld  1/1    alpine  ping docker.com
```

PS

Exécutez ps pour savoir sur quel noeud s’exécute la tâche d’un service:

```
$ docker service ps helloworld

ID                         NAME          SERVICE     IMAGE   LAST STATE          DESIRED STATE  NOI
8p1vev3fq5zm0mi8g0as41w35  helloworld.1  helloworld  alpine  Running 3 minutes  Running
```

PASSAGE À L'ÉCHELLE

On utilise scale

```
$ docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS>
```

EXEMPLE

```
$ docker service scale helloworld=5
helloworld scaled to 5

$ docker service ps helloworld

ID                         NAME          SERVICE     IMAGE   LAST STATE          DESIRED STATE  NO
8p1vev3fq5zm0mi8g0as41w35  helloworld.1  helloworld  alpine  Running 7 minutes   Running
c7a7tcdq5s0uk3qr88mf8xco6  helloworld.2  helloworld  alpine  Running 24 seconds  Running
6crl09vdcalvtfehfh69ogfb1  helloworld.3  helloworld  alpine  Running 24 seconds  Running        wo
auky6trawmdlcne8ad8phb0f1  helloworld.4  helloworld  alpine  Running 24 seconds  Assigned
ba19kca06l18zujfwxyc5lkyn  helloworld.5  helloworld  alpine  Running 24 seconds  Running        w
```

MODE GLOBAL

Par défaut le mode répliqué est utilisé. Pour passer en mode global:

```
$ docker service create --name myservice --mode global alpine top
```

- Chaque worker exécutera un seul réplica du service
- Pour chaque nouveau worker ajouté, le réplica sera automatiquement démarré

ROLLING UPDATE

Mettre à jour un service

```
$ docker service update --image nginx:3.0.7 nginx
```

MISES À JOUR

Le manager va appliquer la mise à jour du service au noeud:

- Arrêt de la tâche
- mise à jour d’une tâche arrêtée
- démarrage du conteneur de la tâche mise à jour
- attendre un certain délai avant de passer à l’autre tâche
- etc,

Si une tâche est en échec, interrompre la mise à jour.

STRATÉGIE DE MISE À JOUR

Vous pouvez configurer le parallélisme des mises à jour, et un délai d'exécution entre chaque mise à jour de tâche:

```
$ docker service update worker --update-parallelism 2 --update-delay 5s
```

MODE MAINTENANCE

Par défaut, tout noeud est ACTIVE. Mais vous pouvez passer un noeud en mode maintenance:

```
$ docker node update --availability drain worker1
```

SORTIE DE MAINTENANCE

```
$ docker node update --availability active worker1
```

SUPPRIMER UN SERVICE

```
$ docker service rm helloworld
```

DISTRIBUTED APPLICATION BUNDLES

Nous avons vu comment gérer individuellement des services.

Nous allons voir comment optimiser la gestion de plusieurs services avec les paquets d’application répartie.

Un DAB est à Swarm, ce que Compose est à un serveur unique Docker.

DAB

Description au format JSON décrivant les services d'une application

Génération:

```
$ docker-compose bundle
```

Génére un fichier .dab

UTILISATION

La commande stack permet d’utiliser ce fichier .dab

```
$ docker stack deploy dockercoins
$ docker stack ps dockercoins
$ docker stack rm dockercoins
```

LACUNES DE STACK

Outil expérimental, certaines fonctionnalités ne sont pas encore disponibles:

- Global scheduling
- Scaling
- etc

Il faut encore passer par la commande service et gérer les services un à un pour ces opérations.

POUR CONCLURE

- Swarm est un moyen de consolider vos hôtes Docker
- Moyen simple compatible avec l'API de Docker
- Beaucoup de fonctionnalités encore en beta

NOUS AVONS VU

- Réseau
- Compose
- Machine
- Swarm

RÉFÉRENCES

- Réseau dans Docker
- Docker Compose
- Docker Machine
- Docker Swarm
- Atelier Orchestration officiel