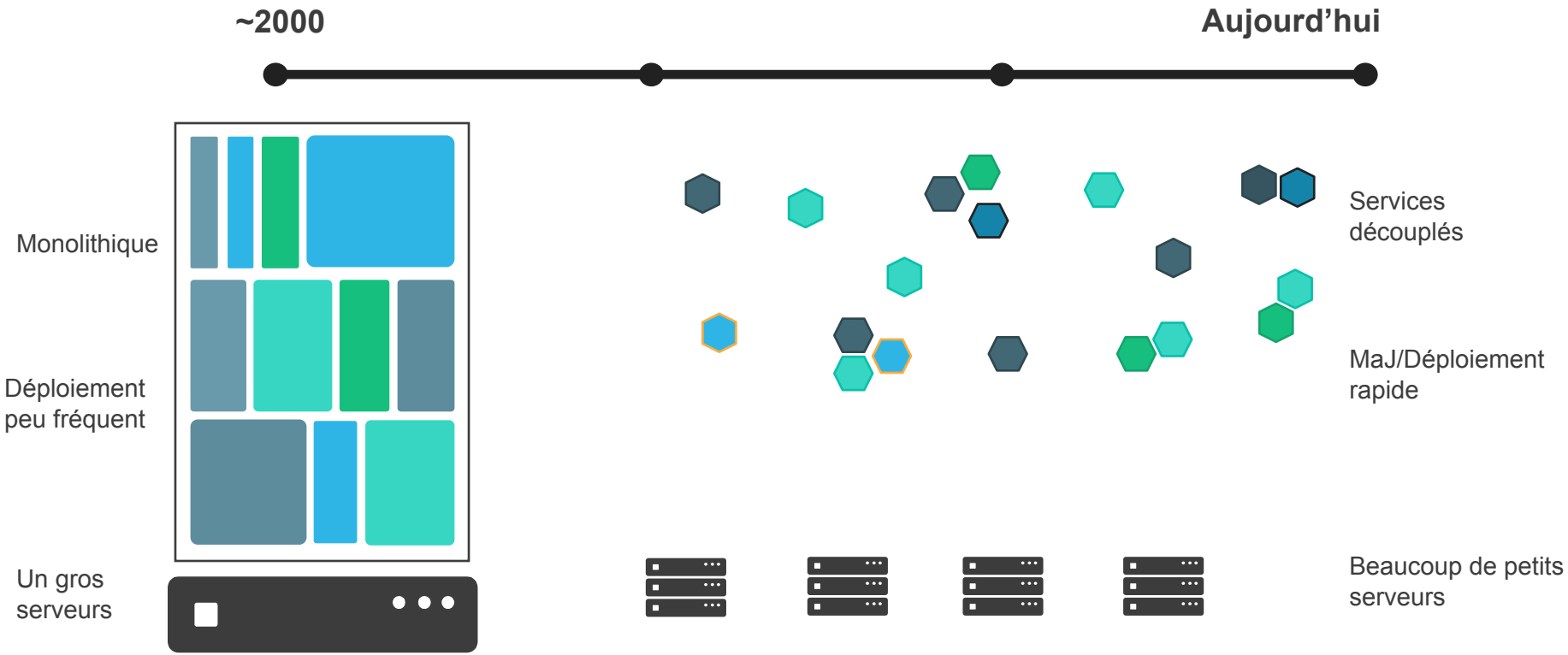
DOCKER

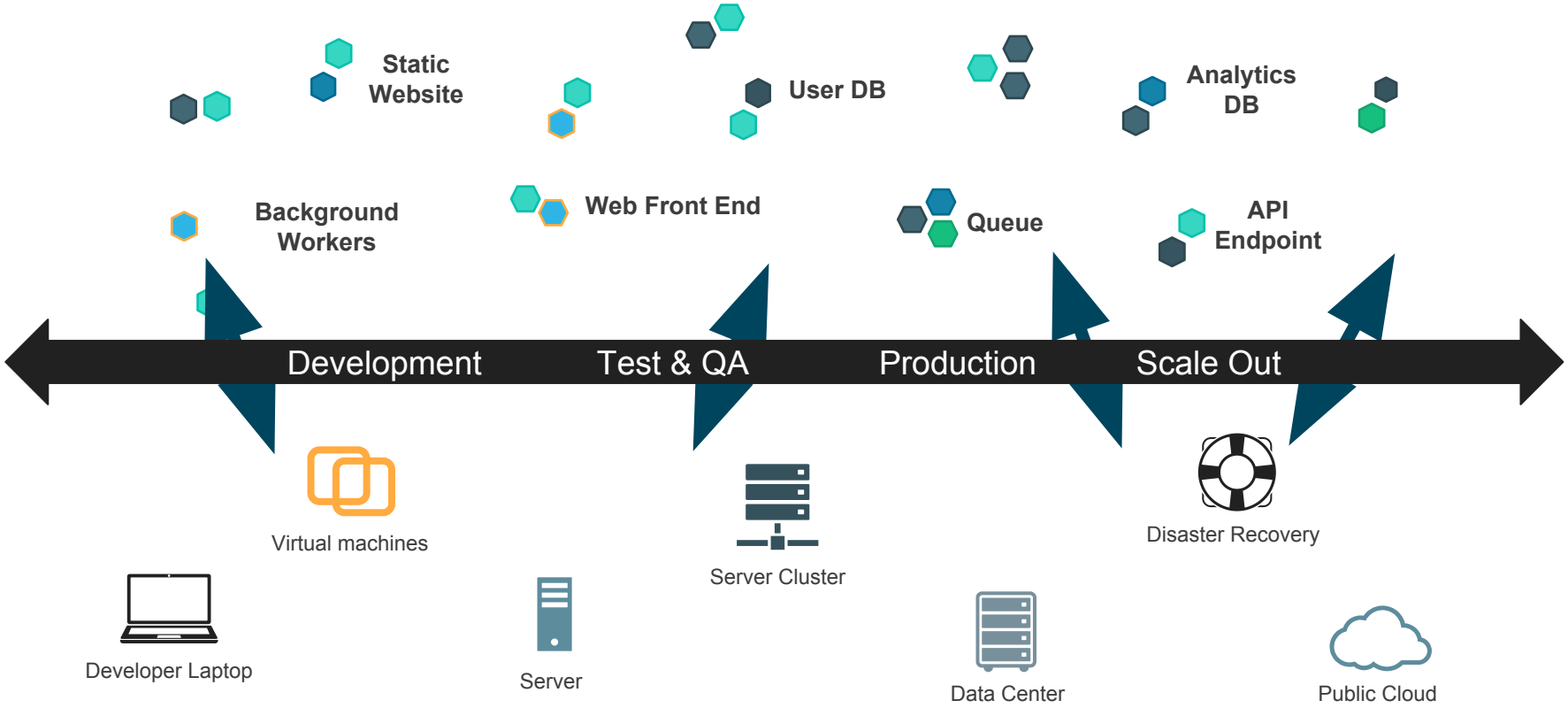
créer et administrer vos conteneurs virtuels d'applications

La problématique

L’architecture des applications change rapidement



Environnements et technologies ultra hétérogènes

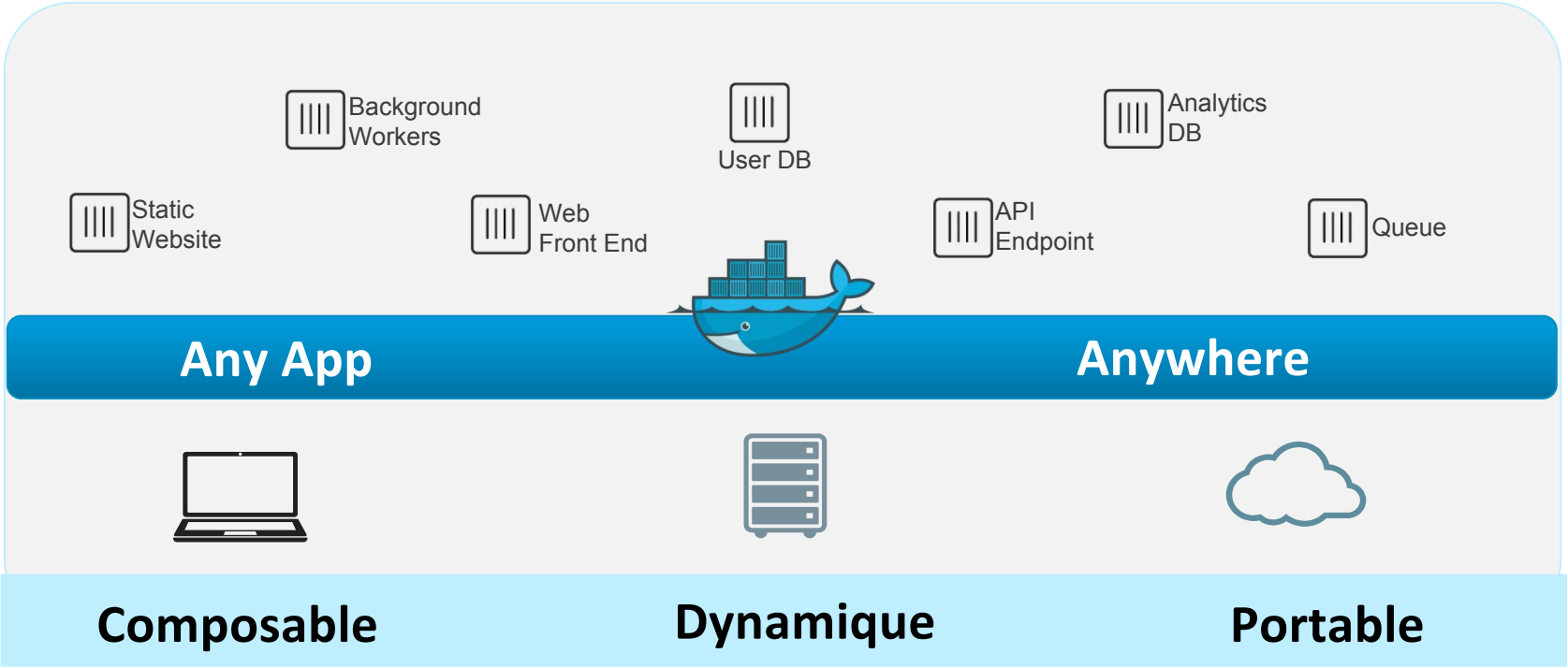


Solution: Docker containers

Container

- Package une application et ces dépendances
- Isoler les applications les une des autres
- Agnostique sur le contenu
- Création d’un standard pour le format des containers (OCI)
- Facilement portable sur différents environnements
- Flexibilité/modularité des composants. Ex. toolkit comme LinuxKit / Moby
- Automatisable/Scriptable

Solution: Docker containers



Du point de vue du développeur ...

Build once… run anywhere

- un environnement portable pour l'exécution des apps
- Pas de risque d’oublier des dépendances, packages, … durant les déploiements.
- Chaque application s'exécute dans son propre container : avec ses propre **version des librairies**
- Facilite l’automatisation des tests
- élimine les problèmes de compatibilité entres les plate-forme.
- Coût ressource très bas pour lancer un container. On peut en lancer des dizaines sur poste développeur (laptop)
- Permet de tester des technologie ou faire des prototypes rapidement et à très bas coût.

Du point de vue de l’admin sys ...

Configure once…run anything

- Rends le workflow plus consistant, prédictible, répétable
- Élimine les inconsistances entre les environnements de dev/test/prod
- Améliore la rapidité et la fiabilité du déploiement continu (continuous deployment)
- Réduction des pb de performances (Ex. avec les VM); réduction des coûts (hébergements cloud, …)

12 Factor App

https://12factor.net/fr/
https://blog.eleven-labs.com/fr/12-factor-app/
Lab pour Docker autour de ces bonnes pratiques : https://github.com/docker/labs/tree/master/12factor

Méthodologie de 12 bonnes pratiques pour créer des applications modernes, portables, sous forme de (micro-)services.
Quelques exemples :

- Code base : 1 aplication == 1repository code source
  - Si il y a plusieurs dépot, c'est un système distribuée, chaque composant du système distribué est une application, et chacun peut individuellement respecter la méthodologie 12 facteurs.
  - Plusieurs applications partageant le même code est une violation des 12 facteurs. La solution dans ce cas est de factoriser le code partagé dans des bibliothèques qui peuvent être intégrées via un gestionnaire de dépendances.
- Dépendances : Une application 12 facteurs ne dépend jamais de l'existence implicite de packages au niveau du système.
  - Utiliser un gestionnaire de packages/dépendances (NodeJS npm, python pip, ruby bundler, PHP composer). Pour PHP les version exacte des lib sont dans composer.lock pour être consistant lors du déploiement en préprod/prod.
  - Les applications 12 facteurs ne s'appuient pas sur l'existence implicite d'outils système (Ex. curl). Si l'application dépend d'un outil système, cet outil doit être distribué avec l'application.

12 Factor App

- Configuration
  - Au lieu de maintenir plusieurs fichiers de config (dev, test, prod,...), qui peuvent en plus contenir des mots de passes, API security token etc. il est recommandé de passer la configuration via des variables d'environnements
    - **EX.**
```
module.exports.connections = {
  mongo: {
    adapter: 'sails-mongo',
    url: process.env.MONGO_URL
  }
};
```
    - NB : Approche symfony avec un fichier parameters.yml.dist
- Services externes : Traitez les services externes comme des ressources attachées
  - L'application 12 facteurs ne fait pas de distinction entre les services locaux et les services tiers.
    - Ex. Le remplacement d'une base de données PostgreSQL locale par une autre gérée chez un tiers doit pouvoir se faire sans modifications dans le code de l'application, seul la configuration doit changer.

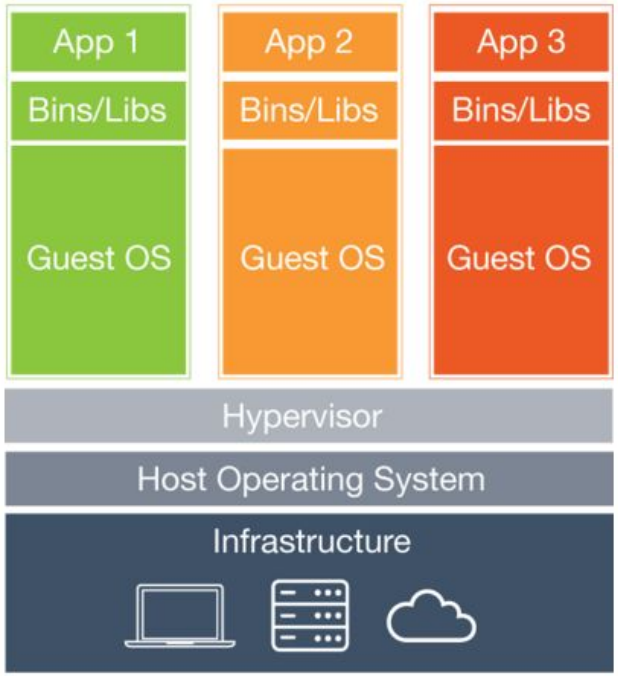
Voir liste exhaustive des 12 facteurs là : https://12factor.net/fr/

…

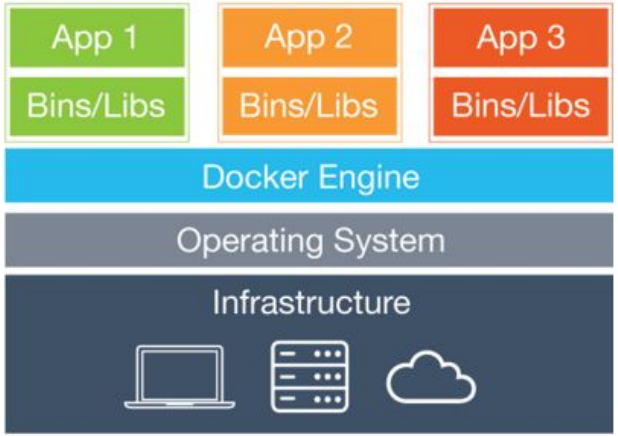
...

De la virtualisation à docker

Container vs. Virtualisation



Virtual Machines



Containers

Historique et concepts de base

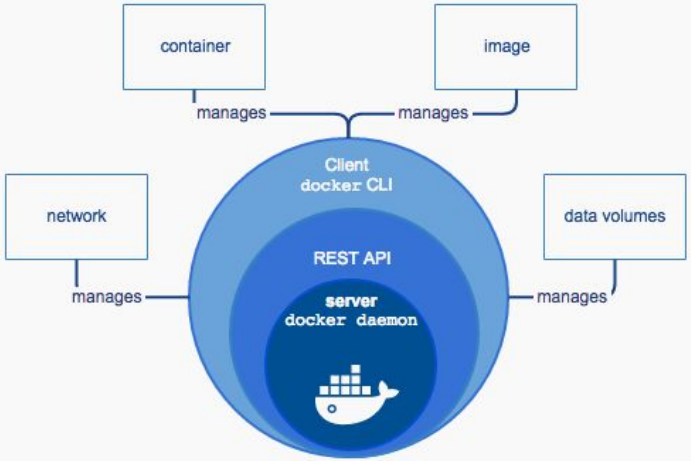
- 1979: Unix V7 introduit le concept de `"chroot"` i.e. modification du root directory (filesystem) d'un process et de ses enfants (ex. forks). C'est les début de tentative d'isolation d'un processus : confinement d'un processus afin de l'empêcher d'accéder à certain fichiers.
- 2000: FreeBSD Jails : permet à l'admin de partitionner un système FreeBSD en plusieurs système indépendants (appelés "jails") avec la possibilité d'assigner une adresse IP différente à chacun.
- 2004: Oracle Solaris Containers
- 2006: Process Containers. Créer par Google. Fais pour limiter, comptabiliser et isoler l'utilisation des ressources : CPU, RAM, disk IO, network, d'un collection de processus. C'est ce qui deviendra les **cgroups** et sera mergé dans le noyaux linux 2.6.24
- 2008: LXC (LinuX Containers) la plus aboutie des implémentation linux pour la containerisation .
- 2013: Let Me Contain That For You (LMCTFY) c'est la mise en open source de la stack de containerisation de Google
- Docker utilise LXC au départ pour ensuite créer sa propre implémentation : **libcontainer**
- **OCI** **https://www.opencontainers.org/** Open Container Initiative : standards industriel pour container et image

Architecture

Docker engine

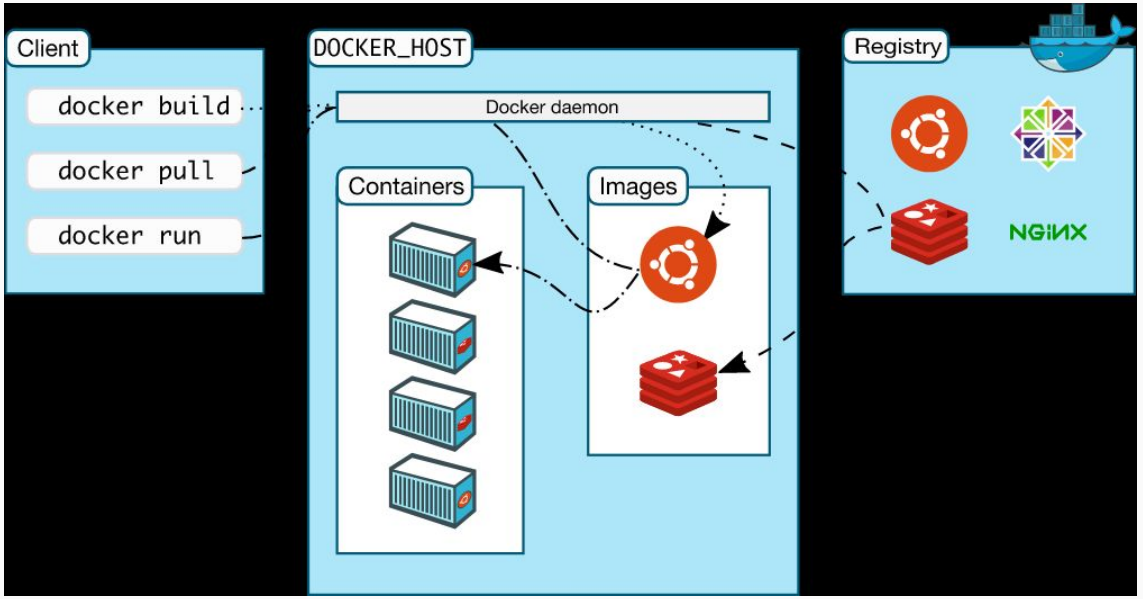
- Une partie “serveur” (unix daemon `dockerd`)
- une API REST pour communiquer avec le serveur.
- Un “client” : interface ligne de commande (commande `docker`)

beaucoup d’autre applications de l'écosystème docker utilisent l’API REST pour contrôler le daemon `dockerd`.



Docker engine

Le client peut communiquer avec un ou plusieurs serveur(s) (daemon *dockerd*) en local ou via le réseau à travers l'API REST exposée par le serveur.



La registry

Dépôt public (ou privé) ou sont stockées les images.

Images

C’est un **template de système de fichiers en “lecture seule” (*immutable*)** (on peut voir ça comme une sorte d’archive `Tar`)
avec en plus des **méta-informations pour la création d’un container.**

Pour créer une nouvelle image on utilise un Dockerfile, chaque instruction du Dockerfile créer une nouvelle couche (layer) dans l’image.

Quand vous modifiez le Dockerfile et recréez l’image, seule la/les couche(s) modifié sont reconstruite (très léger et performant).

```
1  FROM alpine:3.1
2  RUN apk add --update php-cli && rm -rf /var/cache/apk/*
3  ADD index.php /var/www/index.php
4  EXPOSE 80
5  CMD ["php", "-S", "0.0.0.0:80", "-t", "/var/www"]
```

```
$ docker history revollat/micro-php-app-docker
IMAGE                CREATED              CREATED BY
SIZE                 COMMENT
74d1fc40c34f         2 years ago          /bin/sh -c #(nop) CMD ["php" "-S" "0.0.0.0...   0B
<missing>            2 years ago          /bin/sh -c #(nop) EXPOSE 80/tcp                 0B
<missing>            2 years ago          /bin/sh -c #(nop) ADD file:7d46011101766c9...
40.3kB
<missing>            2 years ago          /bin/sh -c apk add --update php-cli && rm ...
11.3MB
<missing>            2 years ago          /bin/sh -c #(nop) ADD file:b9238a47014404d...
5.03MB
```

Images

2 manières de créer des images

Cf. TP Layer01

docker commit

- Sauvegarde les changement effectués au sein du container dans un nouveau layer
- Crée une nouvelle image (qui est une copie du container)

docker build

- Build répétable à partir d'un Dockerfile
- c'est la méthode à préférer.

Namespace images

- Officiels (Ex. ubuntu, busybox, …)
- Utilisateur et Organisations (Ex. jpetazzo/clock)
- Repo privée (Ex. registry.example.com:5000/my-private/image)

Container

C’est une instance **exécutable** d’une image.

Analogie programmation objet *:*

- Images are conceptually similar to *classes*.
- Image Layers are conceptually similar to *inheritance*.
- Containers are conceptually similar to *instances*.

En contactant le serveur via l’API ou via le client (commande docker) nous pouvons démarrer, arrêter, déplacer, supprimer des containers.

Le container exécuté peut être attaché
à un réseau, un volume de stockage. On peut aussi prendre un snapshot du container pour créer une nouvelle image.

On pourras aussi contrôler l’***isolation*** du container (vis-à-vis des autres containers, vis-à-vis de la machine hôte)

Container

Un container est définie par les métas informations de son image correspondante et par les éventuelles options passées lors de l'exécution du container.

Ex.

```
docker run -i -t ubuntu /bin/bash
```

Lorsqu'un container est arrêté et supprimé, les changements d'état qui n'ont pas été sauvegardés dans un volume de stockage persistant sont perdus.

Les technologies qui sous tendent les containers

“Container” (ou Conteneur en français) est juste un terme générique qui désigne sous ce nom unique un ensemble de technologies matures intégré dans le noyaux linux depuis une dizaine d’année.

PS : une version Microsoft Windows des container est en train de voir le jour dans les version de windows serveur.

Les namespaces

Au coeur des technologies qui rende possible l'isolation dans un container on retrouve les ***Namespaces***.
Au lancement d'un container plusieurs namespaces sont créés.

- **The pid namespace**
  - Process isolation (PID: Process ID).
  - PID 1 init-like par namespace
  - chaque namespace a sa propre numération PID (isolation de l'hôte)
  - le process d'un namespace ne peut envoyer de systemcall sur un autre process d'un autre PID namespace
  - gestion de pseudo-filesystem (ex : /proc) vu par le PID namespace qui le monte
  - un process possède plusieurs PID : un dans le namespace, un en dehors (process vu par l'hôte), davantage en cas de namespaces imbriqués

- **The net namespace:**
  - ses interfaces réseau
  - ses ports
  - sa table de routage
  - ses règles de firewall (iptables)
  - son répertoire /proc/net
  - INADDR_ANY (0.0.0.0)

Les namespaces

- **The ipc namespace:** Management des ressources IPC (InterProcess Communication).
  - semaphores
  - message queues
  - shared memory segments

- **The mnt namespace:** Gère l'isolation des points de montage du système de fichier vus par un groupe de process :
  - les points de montage ne sont plus globaux mais spécifiques au namespace
  - racine propre (chroot)

- **The uts namespace:** Gère l'isolation des identifiants de nom et de domaine. (UTS: Unix Timesharing System).
  - Gère l'isolation des identifiants de nom et de domaine.
  - UTS vient de la structure "utsname" passée à l'appel système uname(). UTS voulant dire "UNIX Time-sharing System".

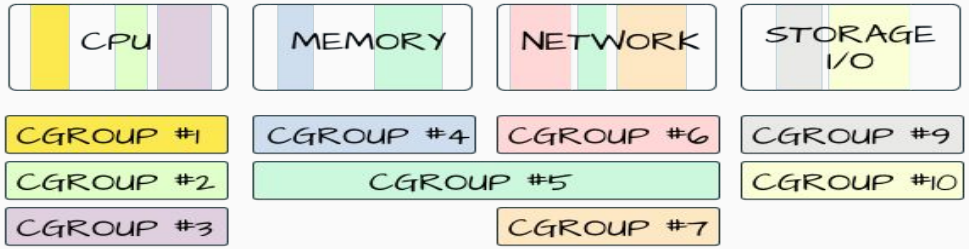
- **The user namespace** mapping UID/GID entre container et machine hote (Ex. root dans container peut être un utilisateurs lambda dans la machine hôte)

Ex. création d'un namespace "pid" grâce à la commande "unshare"

```
$ sudo unshare --fork --pid --mount-proc bash
# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  1.0  0.0  21460  5176 pts/2    S    11:52   0:00 bash
root         11  0.0  0.0  36084  3328 pts/2    R+   11:53   0:00 ps aux
```

Control Groups (a.k.a Cgroups)

Permet de contrôler l'allocation des ressources système à des groupes de processus (group of tasks) : CPU, RAM, bande passante réseau, …
On peut voir *cgroups* comme un *ulimit* pour une groupe de processus.
(NB : systemd utilise les cgroups de façon intensive, cf. commande pour visualiser les cgroups)



Ex.

```
$ sudo cgcreate  -a bork -g memory:mycoolgroup
$ ls -l /sys/fs/cgroup/memory/mycoolgroup/
-rw-r--r-- 1 bork root  0 Okt 10 23:16 memory.kmem.limit_in_bytes
-rw-r--r-- 1 bork root  0 Okt 10 23:14 memory.kmem.max_usage_in_bytes
$ sudo echo 10000000 >  /sys/fs/cgroup/memory/mycoolgroup/memory.kmem.limit_in_bytes
$ sudo cgexec  -g memory:mycoolgroup bash
$ root@kiwi:~/work/ruby-stacktrace# PAS ASSEZ DE MEMOIRE (10 Mo limité par CGROUPS)
error: Could not execute process `rustc -vV` (never executed)
Caused by:
  Cannot allocate memory (os error 12)
```

TP : https://www.katacoda.com/courses/docker-security/cgroups-and-namespaces

Control Groups (a.k.a Cgroups)

```
$ docker run --help
--cpu-shares                CPU shares (relative weight)
--cpuset-cpus               CPUs in which to allow execution (0-3, 0,1)
--pids-limit                Tune container pids limit (set -1 for unlimited)
```

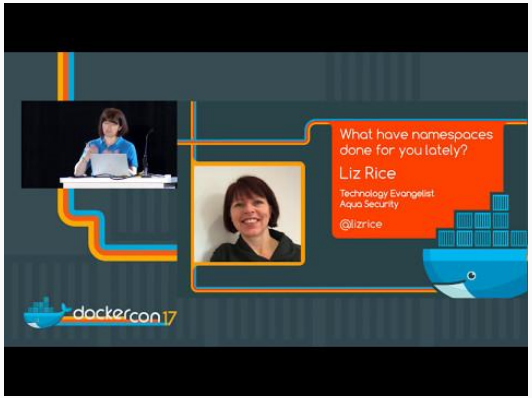
CF TP cgroups

Ressources Namespace et Cgroups : “nano-container” runtime ecrit en go

https://github.com/lizrice/containers-from-scratch/blob/master/main.go

Fork process avec flags pour namespaces

```
cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall.CLONE_NEWNS,
    Unshareflags: syscall.CLONE_NEWNS,
}
```



Chroot du process et mount de /proc

```
syscall.Chroot("/home/liz/ubuntufs")
syscall.Mount("proc", "proc", "proc", 0, "")
```

Assignation CGroups (qui limite le nb de processus à 20 dans ce cgroup)

```
cgroups := "/sys/fs/cgroup/"
pids := filepath.Join(cgroups, "pids")
os.Mkdir(filepath.Join(pids, "liz"), 0755)
must(ioutil.WriteFile(filepath.Join(pids, "liz/pids.max"), []byte("20"), 0700))
```

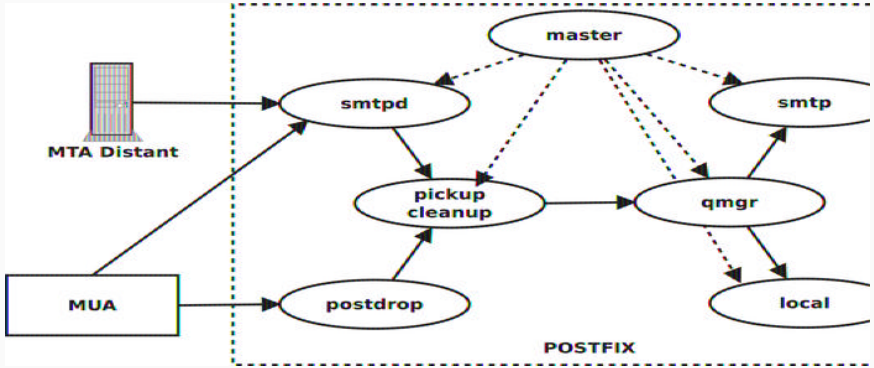
Linux capabilities

Les DAC (Discretionary access control, i.e. système de permission UNIX par défaut) sont limités et les MAC (Mandatory access control) comme selinux sont compliqués à mettre en place.

Avec les capabilities, un processus qui s'exécute en root peut abandonner (drop) les capabilities dont il n'a pas besoin (afin de prévenir une escalade de privilège si le programme est vulnérable)

"root" within a container has much less privileges than the real "root"

Permet d'assurer un confinement d'exécution pour les processus : entrée/sorties, envoi de signaux, passage interface en mode promiscuous, bind d'un port < 1024, …
Les architecture modulaire comme celle de postfix (contrement à Sendmail) permettent d'appliquer finement les capabilities (moindre privilège) aux différents binaires et ainsi réduire la surface d'attaque.



Linux capabilities

Exemple de Linux Capabilities

SYS_MODULE
Load and unload kernel modules

SETGID
Make arbitrary manipulations of process GIDs and supplementary GID list.

SETUID
Make arbitrary manipulations of process UIDs.

MKNOD
Create special files using mknod(2).

NET_RAW
Use RAW and PACKET sockets.

KILL
Bypass permission checks for sending signals.

SYS_CHROOT
Use chroot(2), change root directory.

SYS_NICE
Raise process nice value (nice(2), setpriority(2)) and change the nice value for arbitrary processes.

Linux capabilities

Exemple commande

```
root@capng:/testcap# ls -al /bin/ping
-rwsr-xr-x 1 root root 36136 avril 12 2011 /bin/ping
```

Le SUID est identifié par le bit « s » mis en évidence dans la sortie de **ls**. Supprimons-le :

```
root@capng:/testcap# chmod u-s /bin/ping
root@capng:/testcap# ls -al /bin/ping
-rwxr-xr-x 1 root root 36136 avril 12 2011 /bin/ping
```

Testons en tant que l'utilisateur « toto » :

```
toto@capng:/testcap$ ping www.google.fr
ping: icmp open socket: Operation not permitted
```

Nous pouvons ajouter la capacité **cap_net_raw** dans les sets « permitted » et « effective » (pe) du fichier exécutable **ping**. Cet ajout se fait avec la commande **setcap** (**getcap** pour afficher les capabilities d'un exécutable) :

```
root@capng:/testcap# setcap cap_net_raw=pe /bin/ping
root@capng:/testcap# getcap /bin/ping
/bin/ping = cap_net_raw+ep
```

Testons à nouveau en tant que l'utilisateur « toto » :

```
toto@capng:~$ ping -c 1 www.google.fr
PING www.google.fr (173.194.40.152) 56(84) bytes of data.
64 bytes from par10s10-in-f24.1e100.net (173.194.40.152): icmp_req=1 ttl=63 time=1.83 ms
--- www.google.fr ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.838/1.838/1.838/0.000 ms
```

CF TP cap01

Seccomp

Fonctionnalité de sandboxing apporté par le noyaux linux qui agit comme un **firewall pour les syscalls**. Seccomp place un process dans un état “secure” où le processus ne peut plus faire que les appels systems suivants : *exit(), sigreturn(), read() et write()*

CF TP seccomp01

AppArmor

AppArmor permet à l'administrateur système d'associer à chaque programme un profil de sécurité qui restreint ses accès au système d'exploitation. Il complète le traditionnel modèle d'Unix du contrôle d'accès discrétionnaire (DAC, Discretionary access control) en permettant d'utiliser le contrôle d'accès obligatoire (MAC, Mandatory access control).

Le profile par defaut utilisé pour loes container docker (pas pour le demon) est dans ***/etc/apparmor.d/docker***

En plus de pouvoir gérer les syscall et les capabilities (attention aux recouvrements avec seccomp, cap_add, cap_drop, …) AppArmor peut limiter l'accès aux fichiers dans un répertoire donné.

Exemple avec un extrait de ce profil qui empeche un code malveillant d'installer un plugin wordpress

```
deny /var/www/html/wp-content/plugins/** wlx,
deny /var/www/html/wp-content/themes/** wlx,
owner /var/www.html/wp-content/uploads/** rw,
```

Plus d'infos https://github.com/docker/labs/tree/master/security/apparmor

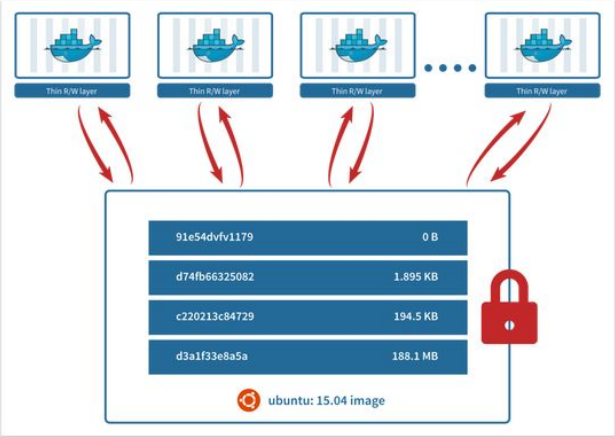
Union file systems et COW

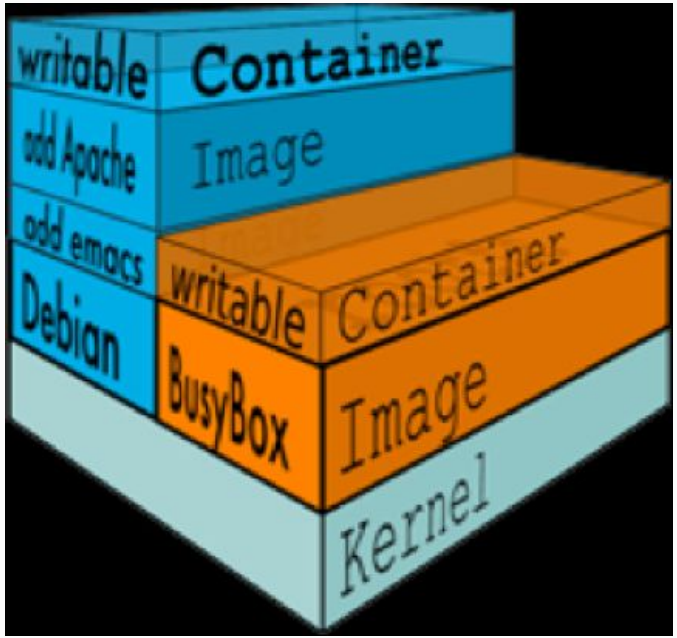
Ce sont des systèmes de fichiers qui fonctionnent avec des couches (layers). Docker peut utiliser plusieurs UnionFS, dont : AUFS, btrfs, vfs, et DeviceMapper.

Lorsque un container est exécuté l'image correspondante est montée en lecture seule et une couche "writable" est montée par dessus.

Le système de fichier global résultant est l'union ("U")
du système en lecture seule et couche writable.

Plusieurs containers peuvent utiliser la même images de base (mutualisation pour éviter de prendre trop de place sur la machine hôte)





Union file systems et COW

Le Copy-On-Write

L'idée fondamentale : si de multiples appelants demandent des ressources initialement impossibles à distinguer, vous pouvez leur donner des **pointeurs vers la même ressource**. Cette fiction peut être maintenue jusqu'à ce qu'un appelant modifie sa « copie » de la ressource. À ce moment-là, **une copie privée est créée**. Cela évite que le changement soit visible ailleurs. Ceci se produit de manière **transparente** pour les appelants. L'avantage principal est que si un appelant ne fait jamais de modifications, la copie privée n'est jamais créée.

Ex en C++ :

```
std::string x("Hello");
std::string y = x;      // x et y utilisent le même tampon
y += ", World!";        // maintenant y utilise un tampon différent
                        // x continue à utiliser le même buffer
```

Dans le cas des container dès que le système de fichier de l'image est modifié dans le container en exécution une **copie privée du bloc modifié est créée dans la couche writable du container.** Si on veut conserver les modifications après arrêt et suppression du container, Il faudra commiter cette modification ce qui créera une nouvelle image à partir de ce nouvel état du container.

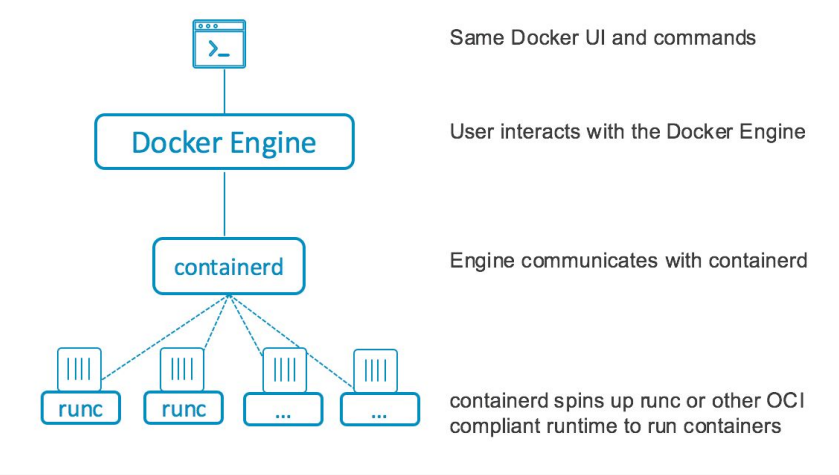
Aller plus loin :
https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/#the-copy-on-write-cow-strategy

Une architecture modulaire : GRPC, libcontainer, runc, containerd, OCI, ...

Le format des images, comment doit être exécuté un container, etc… a été décrit sous forme de spécifications ouvertes et réunies sous le nom de OCI : Open Container Initiative (https://www.opencontainers.org/)

https://github.com/opencontainers/image-spec/releases
https://github.com/opencontainers/runtime-spec/releases

DotCloud a été un contributeur majeur de cette standardisation. Le résultat est que le docker engine, auparavant monolithique (implémenté majoritairement en langage Go), a été découpé en plusieurs modules indépendants et interchangeables.
Ce changement (majeur au niveau de l'architecture bien que transparent pour l'utilisateur) a été introduit dans la version 1.11 de Docker (cf. https://blog.docker.com/2016/04/docker-engine-1-11-runc/)



Une architecture modulaire : GRPC, libcontainer, runc, containerd, OCI, ...

`runC` est une implémentation de Open Containers Runtime specification qui utilise ***libcontainer*** (qui s'interface avec les fonctionnalités kernel namespaces, cgroups, capabilities, … pour garantir l'isolation)

c'est **`runC`** qui est utilisé par défaut dans docker pour l'exécution des container, ce n'est plus l'*Engine* qui se charge de ça. De plus cette brique peut être remplacée par une autre implémentation (principe "*batteries included but swappable*").

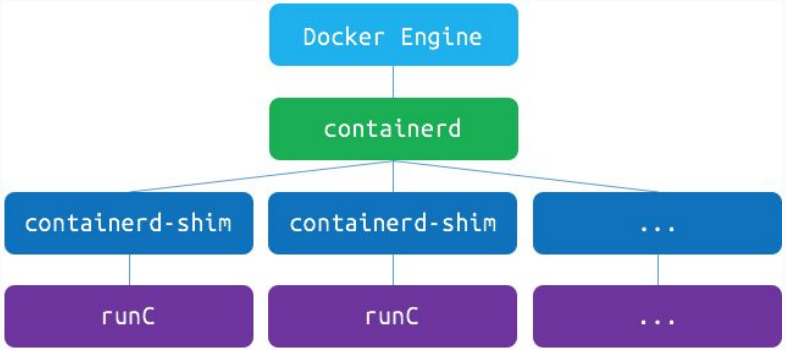
Grâce à ces changement, depuis la version 1.11 on peut par exemple redémarrer/mettre à jour l'*Engine* sans affecter les containers en cours d'exécution (découplage).

Une architecture modulaire : GRPC, libcontainer, runc, containerd, OCI, ...

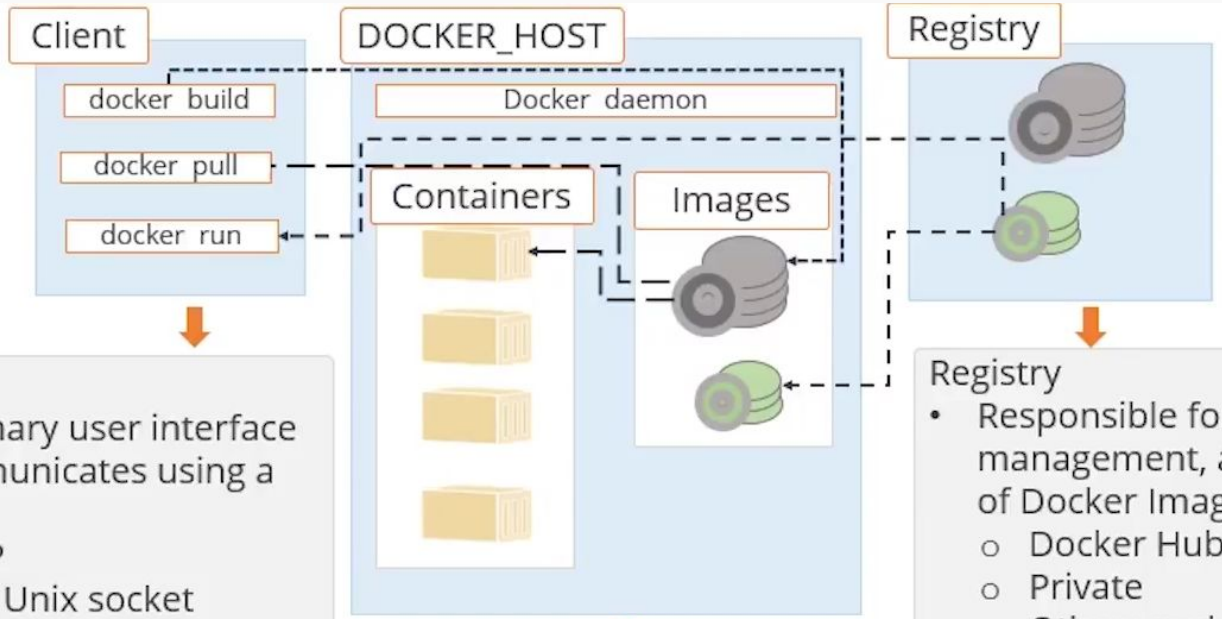
`containerd` est un daemon qui utilise runC (ou une autre implémentation compatible OCI) pour créer, démarrer, arrêter, supprimer les containers.

Le docker *Engine* gère toujours les images, les build (Dockerfiles), volumes, network, … et *containerd* gère les containers.

Engine et *Containerd* communiquent via GRPC.



Recap



Client
- Is the primary user interface that communicates using a REST API
- Over HTTP
- Over local Unix socket

Registry
- Responsible for the storage, management, and delivery of Docker Images
  - Docker Hub
  - Private
  - Other vendors

Server
- Is the Docker daemon
- Is responsible for building, running, and distributing containers

surface d’attaque du daemon docker

https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface

Docker CE vs. Docker EE

- Toutes les fonctionalités extra de la EE peuvent etre intégrés dans la CE via des "integration points" comme les plugin d'auth par exemple.
- EE Applications critiques <==> CE dévelopeurs et petite équipe
- UCP (Universal Control Plane <==> Portainer
- authentification et autorisations fine intégrée à UCP <==> On peut développer ses propres plugins d'autorisation
- Dans EE inclus un ReverseProxy/LoadBalancer HTTP <==> en CE on peut utiliser nginx/haproxy/Traeffic, ...
- EE possède une "Trusted Regitry" intégré avec user/group/permissions de l'UCP <==> CE on utilise hub publique ou le container de la registry en privée
- EE inclus un security scanner des images <==> https://github.com/coreos/clair ?

Mise en oeuvre

```
$ git clone https://github.com/revollat/hello-world
$ tar cv --files-from /dev/null | docker import - vide
sha256:f3bd550d2f8f8f8b3822cbdafeb7f0b4887efe317edc65b85d33f2e663a45de1

$ make
nasm -o hello hello.asm
chmod +x hello

$ docker build -t revollat/hello .
Sending build context to Docker daemon  99.84kB
Step 1/3 : FROM vide
 ---> f3bd550d2f8f
Step 2/3 : COPY hello /
 ---> 30ac18d9034b
Removing intermediate container 0e20c55c45db
Step 3/3 : CMD /hello
 ---> Running in ac888e24778e
 ---> f9538c6ddbe9
Removing intermediate container ac888e24778e
Successfully built f9538c6ddbe9
Successfully tagged revollat/hello:latest

$ docker run revollat/hello

Hello formation docker  :)
```

TP intro01.md

Ecriture d'un Dockerfile

Documentation de référence : https://docs.docker.com/engine/reference/builder/

La commande *`docker build`* crée une image à partir d'une Dockerfile (fichier texte) et d'un *contexte* (chemin local a.k.a PATH ou URL d'un dépôt git par exemple)

Format

```
# Commentaire
INSTRUCTION arguments
```

FROM

```
FROM <image>[:<tag>] [AS <name>]
```

Choisir la "base image" à partir de laquelle construire la nouvelle image (exploite les capacités des UnionFS)
TIPS : Recommandé de partir de https://hub.docker.com/_/alpine/ qui est très légère

Ecriture d'un Dockerfile

RUN

2 façon de l'écrire :

- `RUN <command>` (*la forme "shell", executée par un shell "/bin/sh -c ..." par defaut sous linux)*
- `RUN ["executable", "param1", "param2"]` (Forme "exec", appelé via un system call exec)

Chaque commande `RUN` va créer une nouvelle couche (layer) par dessus l'image courante et va **"commiter"** une nouvelle version de l'image. Cette nouvelle image sert de base aux instructions suivantes du Dockerfile.

Les différents "commits" peuvent servir de pour reprendre le build à partir d'une instruction modifiée sans avoir à recréer toute l'image de zéro (mise en cache des build steps)

NB : Si vous utilisez de pipes "|" comme dans `RUN wget -O - https://some.site | wc -l > /number`
c'est exécuté dans un shell via "/bin/sh -c" qui renvoi un code erreur seulement si la dernière commande a échouée. Si vous voulez envoyer une erreur si n'importe quelle commande échoue il faut faire :

```
RUN set -o pipefail && wget -O - https://some.site | wc -l > /number
```

Ecriture d’un Dockerfile

CMD

- `CMD ["executable","param1","param2"]` (*exec* form, this is the preferred form)
- `CMD ["param1","param2"]` (as *default parameters to ENTRYPOINT*)
- `CMD command param1 param2` (*shell* form)

Permet de définir ce que doit exécuter par défaut un container lors de son lancement.

Ex. shell form
```
CMD echo "This is a test." | wc -
```

Ex. exec form
```
CMD ["python", "app.py"]
```

Il ne peut y avoir qu'une instruction `CMD` par Dockerfile.

`CMD` peut être surchargé via un argument passé à “`docker run`”.

Attention à ne pas confondre RUN et CMD :

- RUN exécute des commande et créer des commit et nouvelles images
- CMD n'exécute rien au moment du build (ça définit dans l’image, la commande a exécuter par un container)

Ecriture d’un Dockerfile

ENTRYPOINT

- `ENTRYPOINT ["executable", "param1", "param2"]` (*exec* form, preferred)
- `ENTRYPOINT command param1 param2` (*shell* form)

ENTRYPOINT vs. CMD

- Un Dockerfile doit spécifier au moins un des deux : CMD ou ENTRYPOINT
- ENTRYPOINT doit être utilisé quand on utilise un container comme un exécutable
- CMD doit être utilisé pour définir des arguments par défaut pour un ENTRYPOINT ou pour exécuter une commande ad-hoc dans le container.
- CMD sera surchargé en lançant un container avec des arguments

Ecriture d'un Dockerfile

Exemples (VOIR AUSSI EXEMPLE AVEC STRACE DU TP SECCOMP)

```
Dans un Dockerfile on a : CMD echo "Hello world"
```

```
run -it <image>
```
Produit
```
Hello world
```

```
run -it <image> /bin/bash
```
Va renvoyer un shell (CMD du Docker file sera ignoré par l'argument passé à docker run

---

```
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["world"]
```

```
run -it <image>
```
Produit
```
Hello world
```

```
docker run -it <image> Olivier
```
Produit
```
Hello olivier
```

La forme "shell" de entrypoint ignore CMD ou tout argument passé à "docker run". Cf. Tableau récapitulatif page suivante.

Ecriture d’un Dockerfile

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT [“exec_entry”, “p1_entry”]
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD [“exec_cmd”, “p1_cmd”]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

Ecriture d'un Dockerfile

EXPOSE

```
EXPOSE <port> [<port>...]
```

Permet de dire quel ports seront écouté dans le container exécuté.
Au lancement du container on pourra **mapper un port de la machine hôte vers un port (exposé) du container.**

ADD/COPY

- `ADD <src>... <dest>`
- `ADD ["<src>",... "<dest>"]` (si les chemins contiennent des espaces)

Copie des fichiers depuis la machine hôte vers le filesystem d'une image.
NB : copy est plus basique et plus performant que ADD (avec laquelle on peut faire par exemple
`ADD http://example.com/big.tar.xz /usr/src/things/`)

ENV
Utile quand on veut gérer des constantes pour facilement mettre à jour de version, Ex.

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJC /usr/src/postgress && …
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

Ecriture d'un Dockerfile

ONBUILD

Ajoute à l'image un déclencheur (trigger) qui sera déclenché plus tard : lorsque cette image sera utilisée comme *base image* (*FROM*) pour la construction (*build*) d'une nouvelle image.

Cas d'utilisation :
On a une image réutilisable pour nos applications python.
Quand on la créer la première fois avec tout nos outils python, on ne peut pas ajouter des source avec ADD ni builder l'application python avec un script via RUN car à ce moment nous n'avons pas encore d'application.

Solution : dans notre Dockerfile de base nous allons mettre

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

Au build de notre image de base des triggers sont ajouté dans les metadatas de l'image mais les instructions ne sont pas exécutés.

Plus tard quand l'image de base est utilisée (FROM) pour un nouveau build. Le builder vérifie s'il y a des trigger enregistrés et ils sont exécutés.
Les trigger sont nettoyés de l'image résultante : donc les trigger ne sont pas hérités par les "arrières petits-enfants".

Bonnes pratiques (Best practices) pour l’écriture d’un Dockerfile

https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

Bonnes pratiques (Best practices) pour l'écriture d'un Dockerfile

Le processus principal d'un container est l'**`ENTRYPOINT`** et/ou **`CMD`** du Dockerfile.

Pour mettre en oeuvre la "Separation Of Concerns" il est conseillé de lancer **un processus par container**. Bien sur ce processus peut forker des enfants (comme les serveur web nginx/apache par exemple : master et worker processes)

Il n'est pas recommandé de lancer plusieurs services (par exemple une instance d'une base de donnée et d'un serveur web) dans le même container, il est préférable (dans la plupars des cas) d'utiliser 2 containers liés entre eux par réseau (user-defined network) et/ou volumes partagés.

Le processus principal est chargé de gérer les autres processus (Ex. fork workers) Quelquefois le processus parent ne gère pas correctement, par exemple l'arrêt des enfants, quand le container est arrêté. Pour résoudre ça on peut utiliser un mini système d'init embarqué dans docker (basé sur https://github.com/krallin/tini) avec l'option "--init" de `docker run` (c'est bien plus léger que d'utiliser sysvinit, upstart ou systemd)

Si vous avez besoin de lancer plusieurs services dans un container

1ère solution : écrire un script qui gère l'exécution des processus et qui sera lancé grâce à CMD dans le Dockerfile :

```
#!/bin/bash

# Start the first process
./my_first_process -D
status=$?
if [ $status -ne 0 ]; then
  echo "Failed to start my_first_process: $status"
  exit $status
fi

# Start the second process
./my_second_process -D
status=$?
if [ $status -ne 0 ]; then
  echo "Failed to start my_second_process: $status"
  exit $status
fi

# Naive check runs checks once a minute to see if either of the processes exited.
# This illustrates part of the heavy lifting you need to do if you want to run
# more than one service in a container. The container will exit with an error
# if it detects that either of the processes has exited.
# Otherwise it will loop forever, waking up every 60 seconds

while /bin/true; do
  ps aux |grep my_first_process |grep -q -v grep
  PROCESS_1_STATUS=$?
  ps aux |grep my_second_process |grep -q -v grep
  PROCESS_2_STATUS=$?
  # If the greps above find anything, they will exit with 0 status
  # If they are not both 0, then something is wrong
  if [ $PROCESS_1_STATUS -ne 0 -o $PROCESS_2_STATUS -ne 0 ]; then
    echo "One of the processes has already exited."
    exit -1
  fi
  sleep 60
done
```

Si vous avez besoin de lancer plusieurs services dans un container

2ème solution : utiliser un process manager comme `supervisord`.

```
FROM ubuntu:latest
RUN apt-get update && apt-get install -y supervisor
RUN mkdir -p /var/log/supervisor
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
COPY my_first_process my_first_process
COPY my_second_process my_second_process
CMD ["/usr/bin/supervisord"]
```

On trouvera aussi des solution hybride/maison/adapté aux cas particuliers (Ex. https://www.camptocamp.com/en/actualite/flexible-docker-entrypoints-scripts/)

Multi-stage build

Avant docker v17.05 on avait typiquement 2 Dockerfile (un pour le développement avec tous les outils de build et un pour la production avec juste le runtime pour exécuter l'application)

`Dockerfile.build` :

```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN go get -d -v golang.org/x/net/html \
  && CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
```

`Dockerfile` :

```
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY app .
CMD ["./app"]
```

`build.sh` :

```
#!/bin/sh
echo Building alexellis2/href-counter:build

docker build --build-arg https_proxy=$https_proxy --build-arg http_proxy=$http_proxy \
    -t alexellis2/href-counter:build . -f Dockerfile.build

docker create --name extract alexellis2/href-counter:build
docker cp extract:/go/src/github.com/alexellis/href-counter/app ./app
docker rm -f extract

echo Building alexellis2/href-counter:latest

docker build --no-cache -t alexellis2/href-counter:latest .
rm ./app
```

Le script build.sh construit d'abord le premier container pour compiler l'app. on extrait l'exécutable et on crée le deuxième container en y copiant l'exécutable créé précédemment.

Multi-stage build

Avec les multi-stage builds on utilise plusieurs FROM dans le Dockerfile.
Chaque instruction FROM peut partir d'une base image différente et initialise une nouvelle etape du build. On peut déplacer des données entre les différentes étapes.
Pour reprendre l'exemple précédent, plus besoin que d'un seule Dockerfile.

Le deuxième FROM commence une nouvelle étape dans le build.
à la fin du processus cette étape est "oubliée" il ne reste plus que l'image de "prod" avec le binaire de l'app et le runtime nécessaire pour son exécution.

```
FROM golang:1.7.3 as builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

TP Multistage01.md

Exemple de Dockerfiles

Admin

Config demon Docker

Depuis `systemd`.

```
$ systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
  Drop-In: /etc/systemd/system/docker.service.d
           └─override.conf
   Active: active (running) since Tue 2017-12-12 14:56:23 CET; 4 weeks 0 days ago
 Main PID: 17886 (dockerd)
   CGroup: /system.slice/docker.service
           ├─16534 docker-containerd -l unix:///var/run/docker/libcontainerd/docker-containerd.sock
--metrics-interval=0 --start-timeout 2m --state-dir /var/run/docker/libc
           └─17886 /usr/bin/dockerd --ipv6=false --ip=127.0.0.1 -H fd:// (<== socket activation systemd)

$ systemctl status docker.socket
● docker.socket - Docker Socket for the API
   Loaded: loaded (/lib/systemd/system/docker.socket; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2017-12-12 14:56:21 CET; 4 weeks 0 days ago
   Listen: /var/run/docker.sock (Stream)
```

Il est conseillé de passer par le fichier de config indépendant du system d'init et de l'OS par défaut sous linux dans /etc/docker/daemon.json (on peut aussi spécifier --config-file au lancement de dockerd) . Ce qu'on peut mettre dedans : https://docs.docker.com/engine/reference/commandline/dockerd//#daemon-configuration-file

EX. ***data-root*** *Root directory of persistent Docker state (default "/var/lib/docker")*

S’attacher à un container en cours d'exécution

`docker attach` pour s’attacher au STDIN/STDOUT/STDERR d’un container en cours d’execution.

Ex. Dans un 1er terminal lancer (qui fait juste un echo de STDIN) :

```
docker run -it ubuntu bash -c 'while read line; do echo "$line"; done < "${1:-/dev/stdin}"'
```

Dans un 2eme terminal lancer

```
docker attach <ID_CONTAINER>
```

Ctrl+p+q pour se détacher

Pour exécuter une commande dans un container en cours d'exécution

```
$ docker exec -it bbb7450a0b21 ps aux
USER        PID %CPU %MEM     VSZ   RSS TTY      STAT START   TIME COMMAND
root        1   0.1  0.0   18028  2716 ?    Ss+  09:09   0:00 bash -c while r
root        7   0.0  0.0   34424  2640 ?    Rs+  09:09   0:00 ps aux
```

Démarrage automatique au boot

restart policy Flag `--restart` de la commande `docker run`

Flag	Description
`no`	Do not automatically restart the container. (the default)
`on-failure`	Restart the container if it exits due to an error, which manifests as a non-zero exit code.
`unless-stopped`	Restart the container unless it is explicitly stopped or Docker itself is stopped or restarted.
`always`	Always restart the container if it stops.

Docker engine est en quelque sorte en “concurrence” avec les process manager (par exemple upstart, systemd, ...)
ça fonction est de lancer les processus (container) et de les arrêter alors que c’est normalement le “job” du PID 1 d’un système de gestion de processus.

Pour aller plus loin avec l’incompatibilité entre docker et systemd “Docker versus Systemd - Can't we just get along?”
Sinon il faut travailler avec runc (mais c’est plus bas niveau/plus compliqué)

Monitoring runtime containers

La commande *`docker stats`* donne des informations sur en temps réel sur CPU, RAM, I/O, …
Ex. `docker stats --format "table {{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}"`

Liste des formats : https://docs.docker.com/engine/reference/commandline/stats/#formatting

```
CONTAINER           CPU %               MEM USAGE / LIMIT
0ddf511dc924        0.00%               7.426MiB / 7.611GiB
```

On peut aussi récupérer directement les information brute fournies par cgroups du kernel

Ex .

```
$ docker ps --no-trunc

$ find /sys/fs/cgroup/ -type d -name 0ddf511dc924f010bed64af08c70d04d3ca588d3f013391c95eda8177883b941
...
...
/sys/fs/cgroup/cpu,cpuacct/docker/0ddf511dc924f010bed64af08c70d04d3ca588d3f013391c95eda8177883b941
/sys/fs/cgroup/blkio/docker/0ddf511dc924f010bed64af08c70d04d3ca588d3f013391c95eda8177883b941
/sys/fs/cgroup/cpuset/docker/0ddf511dc924f010bed64af08c70d04d3ca588d3f013391c95eda8177883b941
/sys/fs/cgroup/memory/docker/0ddf511dc924f010bed64af08c70d04d3ca588d3f013391c95eda8177883b941
/sys/fs/cgroup/pids/docker/0ddf511dc924f010bed64af08c70d04d3ca588d3f013391c95eda8177883b941
```

Limiter les ressources

Mémoire

Option	Description
`-m` or `--memory=`	The maximum amount of memory the container can use. If you set this option, the minimum allowed value is `4m` (4 megabyte).
`--memory-swap` *	The amount of memory this container is allowed to swap to disk. See `--memory-swap` details.
`--memory-swappiness`	By default, the host kernel can swap out a percentage of anonymous pages used by a container. You can set `--memory-swappiness` to a value between 0 and 100, to tune this percentage. See `--memory-swappiness` details.
`--memory-reservation`	Allows you to specify a soft limit smaller than `--memory` which is activated when Docker detects contention or low memory on the host machine. If you use `--memory-reservation`, it must be set lower than `--memory` in order for it to take precedence. Because it is a soft limit, it does not guarantee that the container will not exceed the limit.
`--kernel-memory`	The maximum amount of kernel memory the container can use. The minimum allowed value is `4m`. Because kernel memory cannot be swapped out, a container which is starved of kernel memory may block host machine resources, which can have side effects on the host machine and on other containers. See `--kernel-memory` details.
`--oom-kill-disable`	By default, if an out-of-memory (OOM) error occurs, the kernel kills processes in a container. To change this behavior, use the `--oom-kill-disable` option. Only disable the OOM killer on containers where you have also set the `-m/--memory` option. If the `-m` flag is not set, the host can run out of memory and the kernel may need to kill the host system's processes to free memory.

Limiter les ressources

CPU

Option	Description
`--cpus=<value>`	Specify how much of the available CPU resources a container can use. For instance, if the host machine has two CPUs and you set `--cpus="1.5"`, the container will be guaranteed to be able to access at most one and a half of the CPUs. This is the equivalent of setting `--cpu-period="100000"` and `--cpu-quota="150000"`. Available in Docker 1.13 and higher.
`--cpu-period=<value>`	Specify the CPU CFS scheduler period, which is used alongside `--cpu-quota`. Defaults to 1 second, expressed in micro-seconds. Most users do not change this from the default. If you use Docker 1.13 or higher, use `--cpus` instead.
`--cpu-quota=<value>`	Impose a CPU CFS quota on the container. The number of microseconds per `--cpu-period` that the container is guaranteed CPU access. In other words, `cpu-quota / cpu-period`. If you use Docker 1.13 or higher, use `--cpus` instead.
`--cpuset-cpus`	Limit the specific CPUs or cores a container can use. A comma-separated list or hyphen-separated range of CPUs a container can use, if you have more than one CPU. The first CPU is numbered 0. A valid value might be `0-3` (to use the first, second, third, and fourth CPU) or `1,3` (to use the second and fourth CPU).
`--cpu-shares`	Set this flag to a value greater or less than the default of 1024 to increase or reduce the container's weight, and give it access to a greater or lesser proportion of the host machine's CPU cycles. This is only enforced when CPU cycles are constrained. When plenty of CPU cycles are available, all containers use as much CPU as they need. In that way, this is a soft limit. `--cpu-shares` does not prevent containers from being scheduled in swarm mode. It prioritizes container CPU resources for the available CPU cycles. It does not guarantee or reserve any specific CPU access.

Docker inspect

Logs

`docker logs` (`docker service logs` pour les services) affiche STDOUT et STRERR du/des processus lancé dans le container (il faut veiller à ce que votre *process* ne log pas dans un fichier sinon vous ne verrez rien)

Exemple extrait du dockerfile officiel de Nginx :

```
91  # forward request and error logs to docker log collector
92  RUN ln -sf /dev/stdout /var/log/nginx/access.log \
93          && ln -sf /dev/stderr /var/log/nginx/error.log
94
```

Pour le Dockerfile Apache la methode est un peu différente, mais ça revient au même

/proc/self/fd/1 correspond à STDIN et /proc/self/fd/2 correspond à STDERR

```
86          && sed -ri \
87                  -e 's!^(\s*CustomLog)\s+\S+!\1 /proc/self/fd/1!g' \
88                  -e 's!^(\s*ErrorLog)\s+\S+!\1 /proc/self/fd/2!g' \
89                  "$HTTPD_PREFIX/conf/httpd.conf" \
```

Logs

Utilisation d’un logging driver. Ex : systemd journald

```
$ docker run -d --log-driver=journald --name nginx --net my_bridge -p 5010:80 nginx
$ curl -I http://127.0.0.1:5010/pagequinexistepas
HTTP/1.1 404 Not Found
$ sudo journalctl CONTAINER_NAME=nginx
Sep 21 16:43:33 vps319836 dockerd[1051]: 10.0.0.1 - - [21/Sep/2017:14:43:33 +0000] "HEAD /pagequinexistepas HTTP/1.1" 404 0 "-" "curl/7.47.0" "-"
Sep 21 16:43:33 vps319836 dockerd[1051]: 2017/09/21 14:43:33 [error] 6#6: *2 open() "/usr/share/nginx/html/pagequinexistepas" failed (2: No such file or directory), client:
```

GELF + Logstash + ES

https://gist.github.com/shreyu86/735f2871460a2b068080

Google : docker gelf logstash gelf-address

TP Prometheus01.md

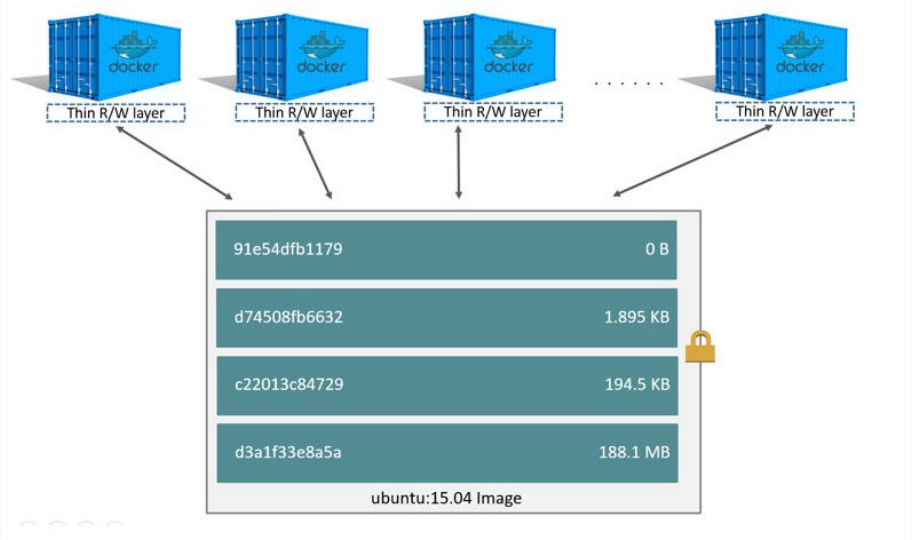
Data / Volume

Data Storage

On peut écrire des données dans le layer RW d'une image mais :

Les données ne vont pas persister si on arrête le container.

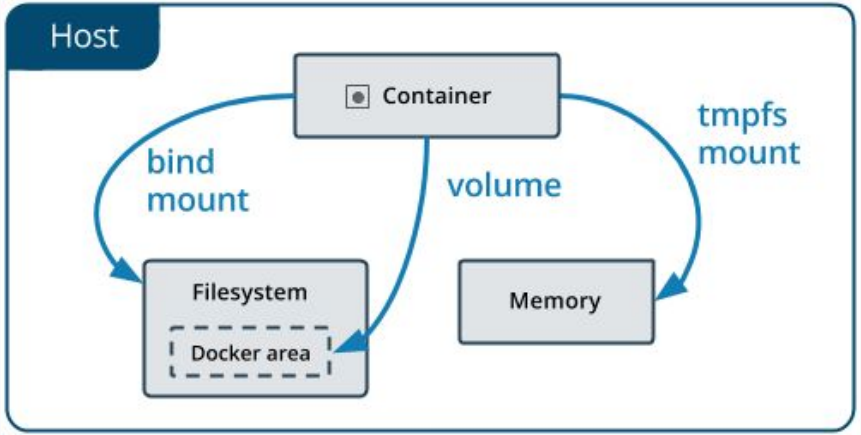
L'écriture dans la couche RW nécessite un ***storage driver*** (UnionFS, COW) cette couche d'abstraction réduit la performance d'écriture sur le disque comparé à un volume.



Data Storage

On peut monter des données dans un container à l'aide de volumes

- Volumes : les données persistent dans un filesystem géré par docker. Les processus de la machine hôte ne modifie pas ces données. Peut être partagé entre plusieurs containers.
- Bind mount : on monte une partie du FS de la machine hôte. Les processus de la machine hôte peuvent modifier ces données. Utile pour le développement ou partage config (hote->container. Ex. montage de /etc/resolv.conf pour le DNS des container)
- tmpfs : filesystem dans la RAM de la machine hôte.



Networking

Networking

Briques de bases Linux Networking utilisé par Docker (et d'autres ...)

Linux Bridge

Device couche OSI 2 Ethernet/MAC. Agit comme un "*switch virtuel*".

Network namespaces

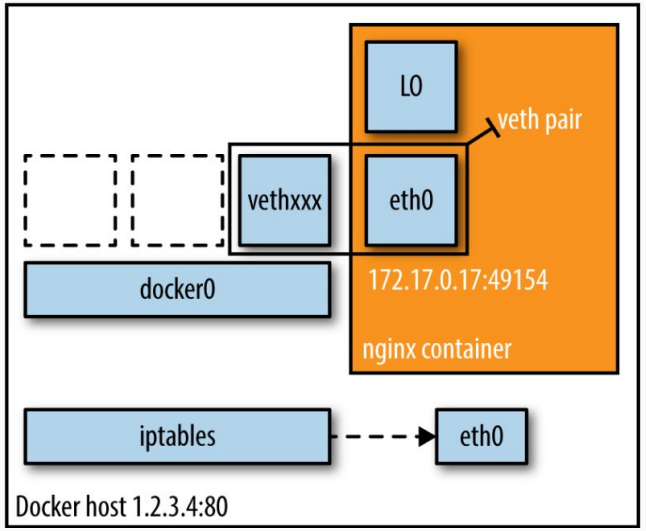
Stack réseau isolée avec ces propres interfaces, routes, règle firewall, …
Un namespace est créé pour chaque container, bien qu'il puisse aussi utiliser le namespace d'un autre container ou encore le namespace réseau de l'hôte.

Virtual ethernet device (veth)

Connecte des namespaces réseau. C'est un lien full-duplex qui possède une interface dans chacun des namespaces à connecter.

Iptables

Filtrage de paquet natif du système (interface à netfilter du kernel). Docker l'utilise pour faire du NAT et du port mapping.



Networking

4 type de réseaux pour le *sigle host networking*

- “none” : uniquement l’interface loopback dans le container
  - Pas de connection réseau possible
- “host” pour partager la stack réseau de l'hôte dans le container.
  - Pas de bridge : plus performant
- “container” pour partager la stack réseau d’un autre container.
- “Bridge” Le bridge `docker0`, fournit un réseau dans la machine hôte où les containers qui y sont connecté se voient entre eux.

CNM (Container Network Model)

The CNM introduit dans l'engine Docker 1.9.0 (Novembre 2015) (cf. commandes `docker network`)

- Un réseau peut être local ou distribué sur un cluster (overlay network)
- Un subnet IP privé est associé au network
- Une IP est attribuée au container connecté au network
- Un container peut être connecté à plusieurs réseaux
- Un container peut avoir un nom/alias sur chacun de ces réseaux
- Les nom/alias sont résolu via un resolveur DNS embarqué

```
# cat /etc/resolv.conf
nameserver 127.0.0.11
```

Custom networks

When creating a network, extra options can be provided.

- --internal disables outbound traffic (the network won't have a default gateway).
- --gateway indicates which address to use for the gateway (when utbound traffic is allowed).
- --subnet (in CIDR notation) indicates the subnet to use.
- --ip-range (in CIDR notation) indicates the subnet to allocate from.
- --aux-address allows to specify a list of reserved addresses (which won't be allocated to containers).

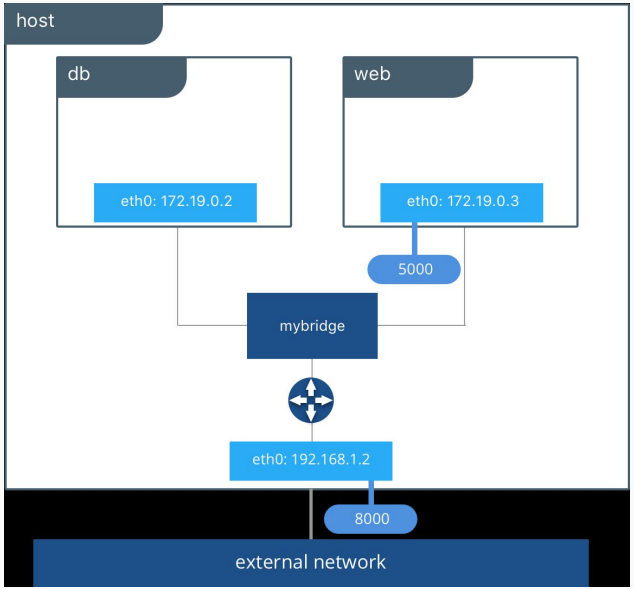
Il est possible d'attribuer une ip spésifique à un container :

```
$ docker network create --subnet 10.66.0.0/16 pubnet
$ docker run --net pubnet --ip 10.66.66.66 -d nginx
```

Networking

Bridge Network

```
docker network create -d bridge mybridge
docker run -d --net mybridge --name db redis
docker run -d --net mybridge -e DB=db -p 8000:5000 --name web chrch/web
```



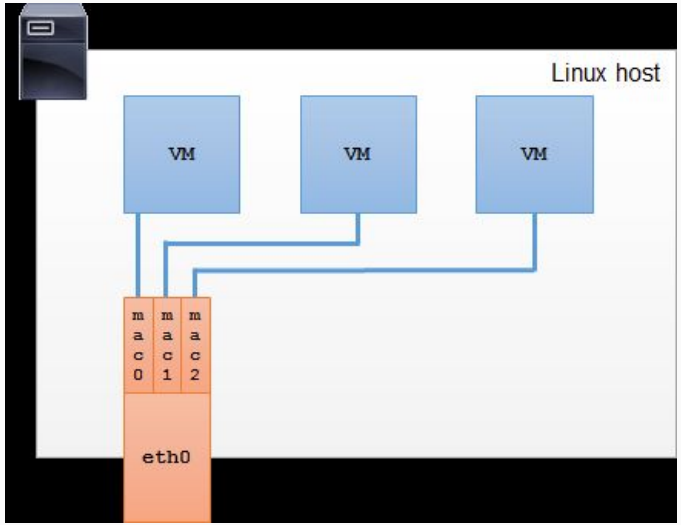
Networking

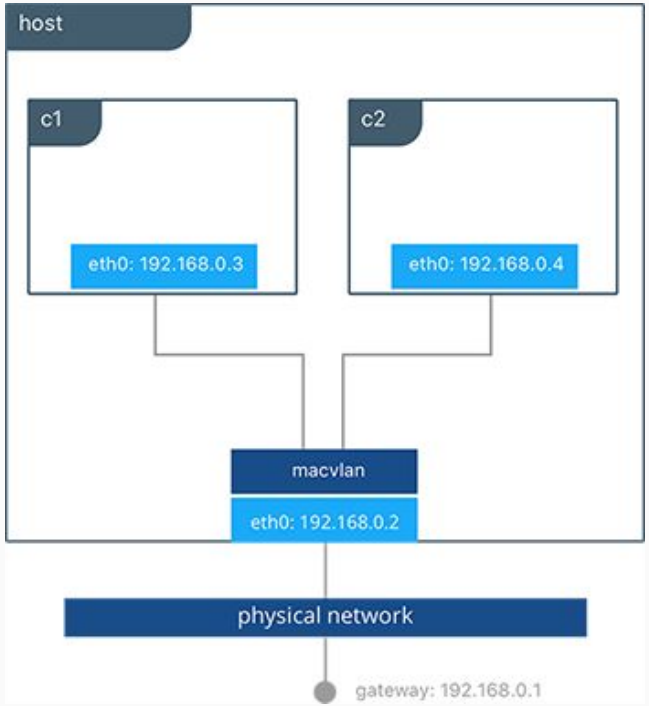
```
$ docker run -d -P nginx:alpine
6520542f3b2e60134b3513ba10ec0754a8ed763236f57356ee6b48b7c2fa1953

$ docker container port $(docker container ls -lq)
80/tcp -> 127.0.0.1:32771

$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' $(docker container
ls -lq)
172.17.0.2

$ ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.324 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.127 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.095 ms
```

MACVLAN

Permet de configurer plusieurs “sub-interface” couche 2 (Ethernet/MAC) sur une seule interface réseau physique.

Les containers se connectent à ces sub-interfaces pour être directement relié au réseau physique (comme si le container était directement branché physiquement sur le même réseau que la machine hôte).



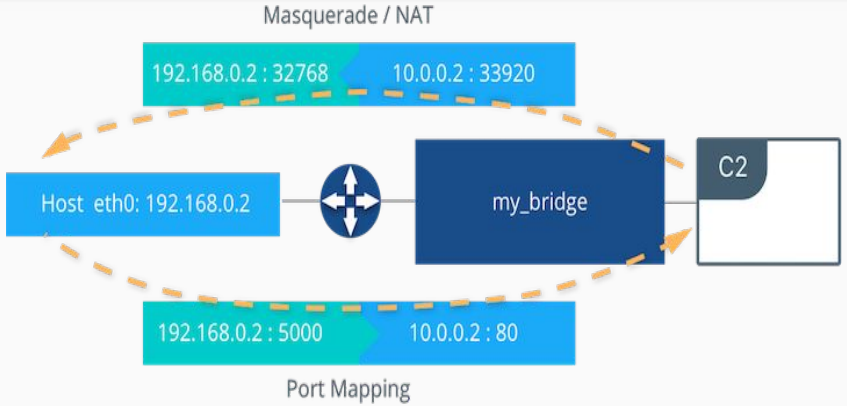


Networking

Accès externe

Par défaut les container situés sur le même réseau docker peuvent communiquer sur tous les ports.
Les communications vers l'extérieur sont autorisées explicitement par firewall via Masquerading (flux sortant) et DNAT (flux entrant)

cf. aussi
https://docs.docker.com/engine/userguide/networking/default_network/container-communication/




```
$ sudo iptables -t nat -L -n
...
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  10.0.0.0/24         0.0.0.0/0
MASQUERADE  tcp  --  10.0.0.2            10.0.0.2            tcp dpt:80

Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0           0.0.0.0/0           tcp dpt:5000 to:10.0.0.2:80
...
...
```

Networking

Overlay Network

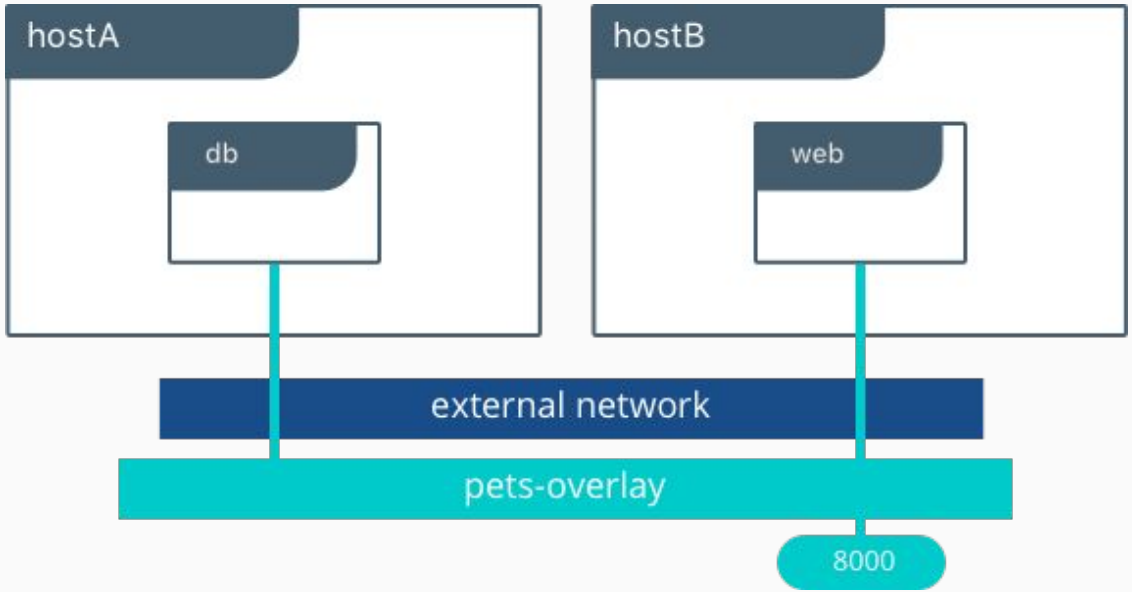
Facilite le multi-host networking.

Utilise le standard ***VXLAN*** (*Virtual Extensible LAN)*

introduit dans le kernel linux v3.7.
Encapsule la couche OSI 2 (MAC/Ethernet) dans des paquet UDP (couche 3)
(a.k.a MAC-in-UDP encapsulation)
IANA assigne à VXLAN par défaut le port 4789.
Les endpoints du VXLAN se comportent comme un switch.

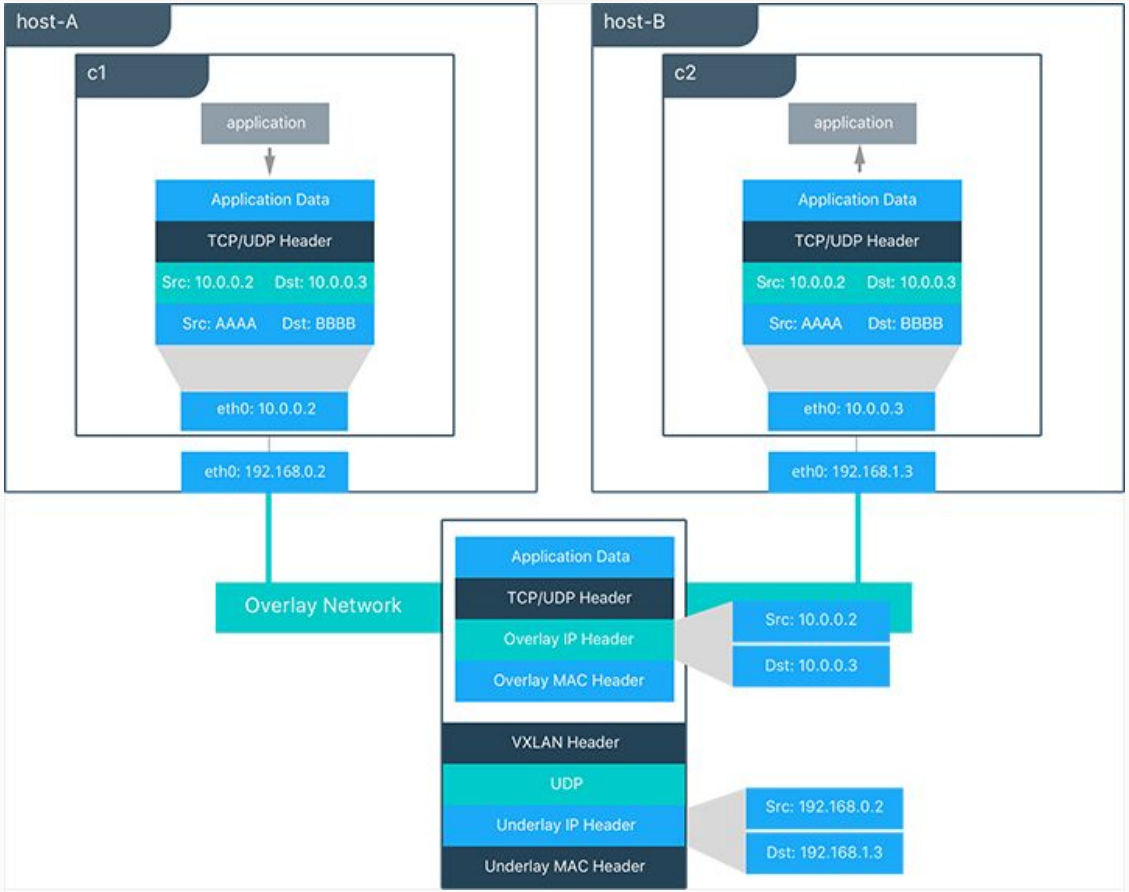
Cf. page suivante pour description complète.

```
docker network create -d overlay --opt encrypted pets-overlay

docker service create --network pets-overlay --name db redis

docker service create --network pets-overlay -p 8000:5000 -e DB=db --name web chrch/web
```



Networking

- C1 fait une requete DNS pour C2 et obtient 10.0.0.3
- C1 génère une trame de couche 2 destinée à l'adresse MAC de C2
- La trame est encapsulée dans une entête VXLAN. Le overlay network gère la localisation des VXLAN Tunnel Endpoints (VTEP), et sait donc que c2 se trouve sur la machine *host-B* qui a pour IP 192.168.1.3, qui sera utilisée comme adresse de destination dans le réseau sous-jacent "underlay network"
- Le paquet encapsulé est envoyé
- La paquet arrive sur l'interface *eth0* de la machine *host-B* et y est décapsulé par le driver de l'overlay network.
  La trame originale de c1 (couche 2) est transmise à l'interface eth0 de c2 et les couche sont dépilé jusqu'à la couche applicative(couche 7)



TP Network01.md

Applications multi-container

Compose

A l'aide de l'outil "docker compose"
Voici à quoi peut ressembler la config d'une application avec un serveur web et un serveur redis :

```
version: '3'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Compose

Fonctionnalités

- Plusieurs environnements isolés sur une machine (utilise un argument `--projet` pour isoler les environnements)
  - Ex. sur une machine de dev démarrer différentes branches d'un projet
- Récré seulement les containers qui ont changés
- Support des variables dans le fichier de conf YAML

Compose

Use cases

- Environnement de développement
- Environnement de test automatisé
  - pratique pour créer l'environement "jettable" dans lequel les tests vont se lancer.
- Déploiement en prod
  - Possibilité de déployer un compose depuis un docker engine distant (instance seule ou cluster swarm)

Exemple d'utilisation basique :
https://docs.docker.com/compose/gettingstarted/#step-4-build-and-run-your-app-with-compose

Exemple concret :
https://github.com/dockersamples/example-voting-app

https://github.com/dockersamples/example-voting-app

Applications multi-container