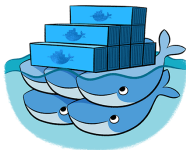


Fondements de la technologie de conteneurisations docker

Jean-Mathieu Chantrein

LERIA

27 novembre 2017



Licence

Ce document est mis à disposition selon les termes de la licence Creative Commons “Attribution - Pas d'utilisation commerciale - Partage dans les mêmes conditions 3.0 non transposé” .



Plan

1 Introduction générale

- Conteneur versus machine virtuelle
- Des technologies de conteneurisations différentes pour des objectifs différents

2 Bases

- Introduction
- Définitions et prérequis
- Premiers pas
- Volatilité, persistance et sécurité des données
- Sécurité des conteneurs
- Conteneurs graphiques

3 Utilisation avancées

- Créations d'images : méthode naïve
- Créations d'images : Dockerfile

4 Toujours plus loin

- Travaux pratiques

Plan

1 Introduction générale

- Conteneur versus machine virtuelle
- Des technologies de conteneurs différentes pour des objectifs différents

2 Bases

- Introduction
- Définitions et prérequis
- Premiers pas
- Volatilité, persistance et sécurité des données
- Sécurité des conteneurs
- Conteneurs graphiques

3 Utilisation avancées

- Créations d'images : méthode naïve
- Créations d'images : Dockerfile

4 Toujours plus loin

- Travaux pratiques

Introduction

Dans cette section nous allons définir quelques notions qui nous permettront de mieux appréhender les principales différences qu'il y a entre une machine physique (réel), une machine virtuelle et un conteneur.

Au commencement il y avait ...

... la machine physique : l'ordinateur (serveur, PC, Mac, ...)

- Il est fabriqué principalement avec des composants électroniques
- Il est programmable (on peut lui faire exécuter des algorithmes)
- Il est très puissant pour effectuer des calculs binaires (mais il ne sait faire que ça)

Système d'exploitation (Operating System, abrégé OS)

Un système d'exploitation est un programme qui a pour but :

- d'assurer la communication entre des logiciels et les différents composants d'une machine (device (RAM, HDD, Carte réseau, ...))
- d'organiser/planifier l'exécution d'autres programmes

Au commencement il y avait ...

... la machine physique : l'ordinateur (serveur, PC, Mac, ...)

- Il est fabriqué principalement avec des composants électroniques
- Il est programmable (on peut lui faire exécuter des algorithmes)
- Il est très puissant pour effectuer des calculs binaires (mais il ne sait faire que ça)

Système d'exploitation (Operating System, abrégé OS)

Un système d'exploitation est un programme qui a pour but :

- d'assurer la communication entre des logiciels et les différents composants d'une machine (device (RAM, HDD, Carte réseau, ...))
- d'organiser/planifier l'exécution d'autres programmes

Au commencement il y avait ...

... la machine physique : l'ordinateur (serveur, PC, Mac, ...)

- Il est fabriqué principalement avec des composants électroniques
- Il est programmable (on peut lui faire exécuter des algorithmes)
- Il est très puissant pour effectuer des calculs binaires (mais il ne sait faire que ça)

Système d'exploitation (Operating System, abrégé OS)

Un système d'exploitation est un programme qui a pour but :

- d'assurer la communication entre des logiciels et les différents composants d'une machine (device (RAM, HDD, Carte réseau, ...))
- d'organiser/planifier l'exécution d'autres programmes

Au commencement il y avait ...

... la machine physique : l'ordinateur (serveur, PC, Mac, ...)

- Il est fabriqué principalement avec des composants électroniques
- Il est programmable (on peut lui faire exécuter des algorithmes)
- Il est très puissant pour effectuer des calculs binaires (mais il ne sait faire que ça)

Système d'exploitation (Operating System, abrégé OS)

Un système d'exploitation est un programme qui a pour but :

- d'assurer la communication entre des logiciels et les différents composants d'une machine (device (RAM, HDD, Carte réseau, ...))
- d'organiser/planifier l'exécution d'autres programmes

Au commencement il y avait ...

... la machine physique : l'ordinateur (serveur, PC, Mac, ...)

- Il est fabriqué principalement avec des composants électroniques
- Il est programmable (on peut lui faire exécuter des algorithmes)
- Il est très puissant pour effectuer des calculs binaires (mais il ne sait faire que ça)

Système d'exploitation (Operating System, abrégé OS)

Un système d'exploitation est un programme qui a pour but :

- d'assurer la communication entre des logiciels et les différents composants d'une machine (device (RAM, HDD, Carte réseau, ...))
- d'organiser/planifier l'exécution d'autres programmes

Ensuite, il y a eu la virtualisation ...

... de machine (machine virtuelle, virtual machine, abrégé VM)

- Une machine virtuelle est une machine émulée par un logiciel.
- Le logiciel d'émulation simule la présence de ressources matérielles telles que la mémoire, le(s) processeur(s), le(s) disque(s) dur.
- Le logiciel d'émulation est capable de simuler un ordinateur.
- On appelle hyperviseur le logiciel ayant pour mission de gérer les VM (QEMU/KVM, HyperV, VMWare, VirtualBox, Xen, Proxmox, ...)
- Il y a 2 types d'hyperviseur :
 - type 1 (bare metal ou natif) meilleur performance, l'hyperviseur est l'os (Ex : Xen, Qemu/KVM) (VM "aware")
 - type 2 (hosted) : Hyperviseur sur OS (Virtual Box) (VM "no aware")

Ensuite, il y a eu la virtualisation ...

... de machine (machine virtuelle, virtual machine, abrégé VM)

- Une machine virtuelle est une machine émulée par un logiciel.
- Le logiciel d'émulation simule la présence de ressources matérielles telles que la mémoire, le(s) processeur(s), le(s) disque(s) dur.
- Le logiciel d'émulation est capable de simuler un ordinateur.
- On appelle hyperviseur le logiciel ayant pour mission de gérer les VM (QEMU/KVM, HyperV, VMWare, VirtualBox, Xen, Proxmox, ...)
- Il y a 2 types d'hyperviseur :
 - type 1 (bare metal ou natif) meilleur performance, l'hyperviseur est l'os (Ex : Xen, Qemu/KVM) (VM "aware")
 - type 2 (hosted) : Hyperviseur sur OS (Virtual Box) (VM "no aware")

Ensuite, il y a eu la virtualisation ...

... de machine (machine virtuelle, virtual machine, abrégé VM)

- Une machine virtuelle est une machine émulée par un logiciel.
- Le logiciel d'émulation simule la présence de ressources matérielles telles que la mémoire, le(s) processeur(s), le(s) disque(s) dur.
- Le logiciel d'émulation est capable de simuler un ordinateur.
- On appelle hyperviseur le logiciel ayant pour mission de gérer les VM (QEMU/KVM, HyperV, VMWare, VirtualBox, Xen, Proxmox, ...)
- Il y a 2 types d'hyperviseur :
 - type 1 (bare metal ou natif) meilleur performance, l'hyperviseur est l'os (Ex : Xen, Qemu/KVM) (VM "aware")
 - type 2 (hosted) : Hyperviseur sur OS (Virtual Box) (VM "no aware")

Ensuite, il y a eu la virtualisation ...

... de machine (machine virtuelle, virtual machine, abrégé VM)

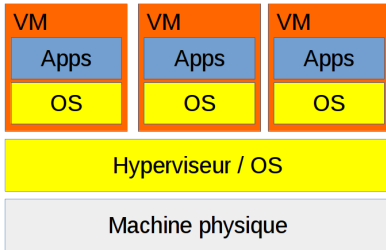
- Une machine virtuelle est une machine émulée par un logiciel.
- Le logiciel d'émulation simule la présence de ressources matérielles telles que la mémoire, le(s) processeur(s), le(s) disque(s) dur.
- Le logiciel d'émulation est capable de simuler un ordinateur.
- On appelle hyperviseur le logiciel ayant pour mission de gérer les VM (QEMU/KVM, HyperV, VMWare, VirtualBox, Xen, Proxmox, ...)
- Il y a 2 types d'hyperviseur :
 - type 1 (bare metal ou nativ) meilleur performance, l'hyperviseur est l'os (Ex : Xen, Qemu/KVM) (VM "aware")
 - type 2 (hosted) : Hyperviseur sur OS (Virtual Box) (VM "no aware")

Ensuite, il y a eu la virtualisation ...

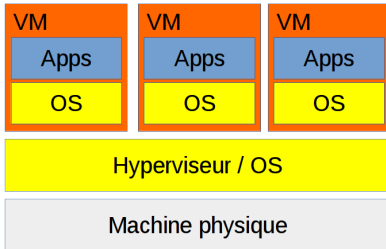
... de machine (machine virtuelle, virtual machine, abrégé VM)

- Une machine virtuelle est une machine émulée par un logiciel.
- Le logiciel d'émulation simule la présence de ressources matérielles telles que la mémoire, le(s) processeur(s), le(s) disque(s) dur.
- Le logiciel d'émulation est capable de simuler un ordinateur.
- On appelle hyperviseur le logiciel ayant pour mission de gérer les VM (QEMU/KVM, HyperV, VMWare, VirtualBox, Xen, Proxmox, ...)
- Il y a 2 types d'hyperviseur :
 - type 1 (bare metal ou natif) meilleur performance, l'hyperviseur est l'os (Ex : Xen, Qemu/KVM) (VM "aware")
 - type 2 (hosted) : Hyperviseur sur OS (Virtual Box) (VM "no aware")

Architecture simplifiée du modèle VM/hyperviseur



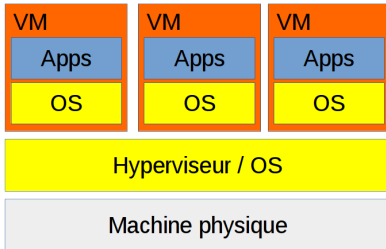
Architecture simplifiée du modèle VM/hyperviseur



Avantages

- Portabilité des VM
- Isolations des applications
- Mutualisations des ressources
- Ressources à la demande
- Démarrage plus rapide qu'une machine réelle

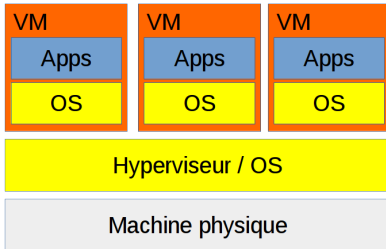
Architecture simplifiée du modèle VM/hyperviseur



Avantages

- Portabilité des VM
- Isolations des applications
- Mutualisations des ressources
- Ressources à la demande
- Démarrage plus rapide qu'une machine réelle

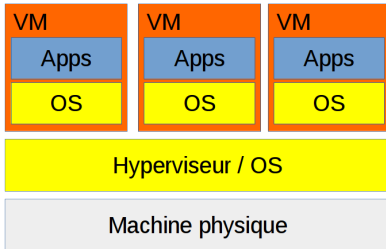
Architecture simplifiée du modèle VM/hyperviseur



Avantages

- Portabilité des VM
- Isolations des applications
- Mutualisations des ressources
 - Ressources à la demande
 - Démarrage plus rapide qu'une machine réelle

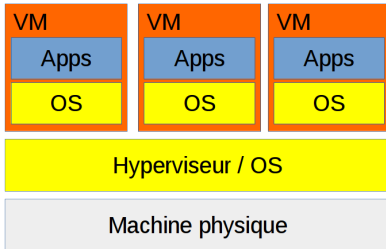
Architecture simplifiée du modèle VM/hyperviseur



Avantages

- Portabilité des VM
- Isolations des applications
- Mutualisations des ressources
- Ressources à la demande
- Démarrage plus rapide qu'une machine réelle

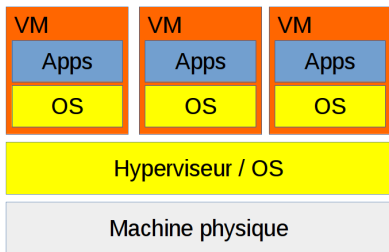
Architecture simplifiée du modèle VM/hyperviseur



Avantages

- Portabilité des VM
- Isolations des applications
- Mutualisations des ressources
- Ressources à la demande
- Démarrage plus rapide qu'une machine réelle

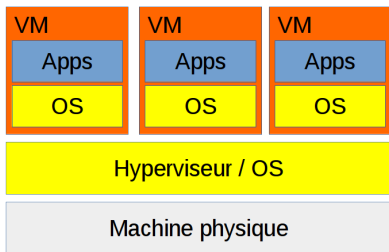
Architecture simplifiée du modèle VM/hyperviseur



Inconvénients

- Performance inférieur à une exécution native
- Consommation de mémoire vive importante
- Consommation de mémoire de masse importante
- Plus lent qu'un conteneur

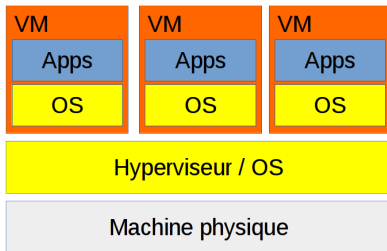
Architecture simplifiée du modèle VM/hyperviseur



Inconvénients

- Performance inférieur à une exécution native
- Consommation de mémoire vive importante
- Consommation de mémoire de masse importante
- Plus lent qu'un conteneur

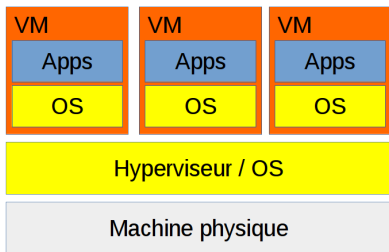
Architecture simplifiée du modèle VM/hyperviseur



Inconvénients

- Performance inférieure à une exécution native
- Consommation de mémoire vive importante
- Consommation de mémoire de masse importante
- Plus lent qu'un conteneur

Architecture simplifiée du modèle VM/hyperviseur



Inconvénients

- Performance inférieur à une exécution native
- Consommation de mémoire vive importante
- Consommation de mémoire de masse importante
- Plus lent que un conteneur

... et la virtualisation d'environnement : la conteneurisation

Terminologie : virtualisation légère versus conteneurisation

- On désigne souvent la conteneurisation comme étant de la virtualisation légère.
- Le terme de virtualisation légère est bien adapté lorsque il s'agit de mettre en opposition la conteneurisation face à la virtualisation (appelé aussi "virtualisation lourde").
- De manière générale, le terme est mal choisi. En effet, lorsque nous utilisons des techniques de conteneurisation, il n'y a pas de virtualisation.
- Pour cette raison, nous n'utiliserons plus le terme de virtualisation lorsque nous parlerons de conteneurisation dans la suite de cet exposé.

... et la virtualisation d'environnement : la conteneurisation

Terminologie : virtualisation légère versus conteneurisation

- On désigne souvent la conteneurisation comme étant de la virtualisation légère.
- Le terme de virtualisation légère est bien adapté lorsque il s'agit de mettre en opposition la conteneurisation face à la virtualisation (appelé aussi "virtualisation lourde").
- De manière générale, le terme est mal choisis. En effet, lorsque nous utilisons des techniques de conteneurisation, il n'y a pas de virtualisation.
- Pour cette raison, nous n'utiliserons plus le terme de virtualisation lorsque nous parlerons de conteneurisation dans la suite de cet exposé.

... et la virtualisation d'environnement : la conteneurisation

Terminologie : virtualisation légère versus conteneurisation

- On désigne souvent la conteneurisation comme étant de la virtualisation légère.
- Le terme de virtualisation légère est bien adapté lorsque il s'agit de mettre en opposition la conteneurisation face à la virtualisation (appelé aussi "virtualisation lourde").
- De manière générale, le terme est mal choisi. En effet, lorsque nous utilisons des techniques de conteneurisation, il n'y a pas de virtualisation.
- Pour cette raison, nous n'utiliserons plus le terme de virtualisation lorsque nous parlerons de conteneurisation dans la suite de cet exposé.

... et la virtualisation d'environnement : la conteneurisation

Terminologie : virtualisation légère versus conteneurisation

- On désigne souvent la conteneurisation comme étant de la virtualisation légère.
- Le terme de virtualisation légère est bien adapté lorsque il s'agit de mettre en opposition la conteneurisation face à la virtualisation (appelé aussi "virtualisation lourde").
- De manière générale, le terme est mal choisi. En effet, lorsque nous utilisons des techniques de conteneurisation, il n'y a pas de virtualisation.
- Pour cette raison, nous n'utiliserons plus le terme de virtualisation lorsque nous parlerons de conteneurisation dans la suite de cet exposé.

La conteneurisation

- Un conteneur fournit un environnement d'exécution isolé pour une/des applications
- Un conteneur partage le noyau du système d'exploitation de l'hôte
- L'isolation est effectuée par des fonctionnalités du noyau (cgroups, namespace, capabilities)
- Il existe différents types de conteneurs (chroot, Docker, LXC, OpenVZ, Rocket, Singularity, ...)
- On appelle hyperviseur ou orchestrateur le logiciel ayant pour mission de gérer les conteneurs (LXD, proxmox (pct), compose, swarm, kubernetes)

La conteneurisation

- Un conteneur fournit un environnement d'exécution isolé pour une/des applications
- Un conteneur partage le noyau du système d'exploitation de l'hôte
- L'isolation est effectuée par des fonctionnalités du noyau (cgroups, namespace, capabilities)
- Il existe différents types de conteneurs (chroot, Docker, LXC, OpenVZ, Rocket, Singularity, ...)
- On appelle hyperviseur ou orchestrateur le logiciel ayant pour mission de gérer les conteneurs (LXD, proxmox (pct), compose, swarm, kubernetes)

La conteneurisation

- Un conteneur fournit un environnement d'exécution isolé pour une/des applications
- Un conteneur partage le noyau du système d'exploitation de l'hôte
- L'isolation est effectuée par des fonctionnalités du noyau (cgroups, usernamespace, capabilities)
- Il existe différents types de conteneurs (chroot, Docker, LXC, OpenVZ, Rocket, Singularity, ...)
- On appelle hyperviseur ou orchestrateur le logiciel ayant pour mission de gérer les conteneurs (LXD, proxmox (pct), compose, swarm, kubernetes)

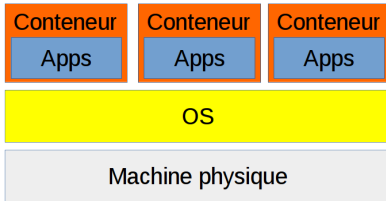
La conteneurisation

- Un conteneur fournit un environnement d'exécution isolé pour une/des applications
- Un conteneur partage le noyau du système d'exploitation de l'hôte
- L'isolation est effectuée par des fonctionnalités du noyau (cgroups, namespace, capabilities)
- Il existe différents types de conteneurs (chroot, Docker, LXC, OpenVZ, Rocket, Singularity, ...)
- On appelle hyperviseur ou orchestrateur le logiciel ayant pour mission de gérer les conteneurs (LXD, proxmox (pct), compose, swarm, kubernetes)

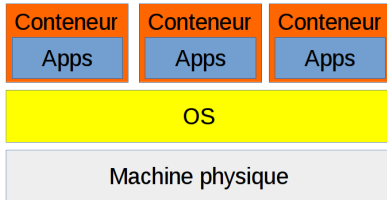
La conteneurisation

- Un conteneur fournit un environnement d'exécution isolé pour une/des applications
- Un conteneur partage le noyau du système d'exploitation de l'hôte
- L'isolation est effectuée par des fonctionnalités du noyau (cgroups, namespace, capabilities)
- Il existe différents types de conteneurs (chroot, Docker, LXC, OpenVZ, Rocket, Singularity, ...)
- On appelle hyperviseur ou orchestrateur le logiciel ayant pour mission de gérer les conteneurs (LXD, proxmox (pct), compose, swarm, kubernetes)

Architecture simplifiée du modèle conteneur/OS



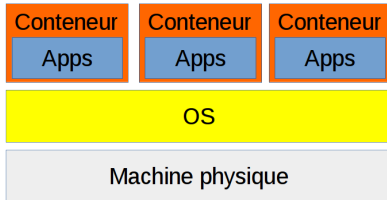
Architecture simplifiée du modèle conteneur/OS



Avantages

- Portabilité des conteneurs (+ ou -)
- Isolations des applications (+ ou -)
- Mutualisations des ressources
- Ressources à la demande
- Démarrage quasiment aussi rapide qu'une application native
- Contrairement aux VMs, pas d'overhead (économie des ressources matériels)

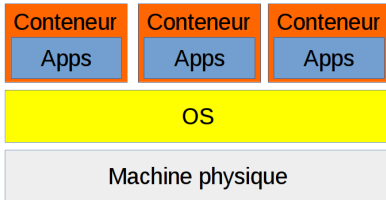
Architecture simplifiée du modèle conteneur/OS



Avantages

- Portabilité des conteneurs (+ ou -)
- Isolations des applications (+ ou -)
- Mutualisations des ressources
- Ressources à la demande
- Démarrage quasiment aussi rapide qu'une application native
- Contrairement aux VMs, pas d'overhead (économie des ressources matériels)

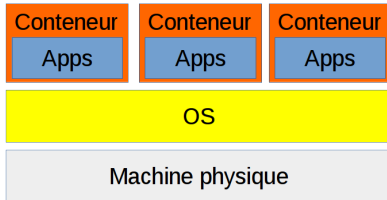
Architecture simplifiée du modèle conteneur/OS



Avantages

- Portabilité des conteneurs (+ ou -)
- Isolations des applications (+ ou -)
- Mutualisations des ressources
- Ressources à la demande
- Démarrage quasiment aussi rapide qu'une application native
- Contrairement aux VMs, pas d'overhead (économie des ressources matériels)

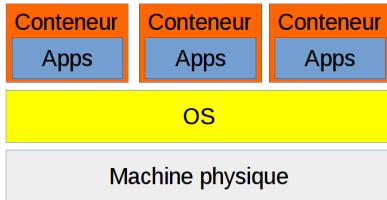
Architecture simplifiée du modèle conteneur/OS



Avantages

- Portabilité des conteneurs (+ ou -)
- Isolations des applications (+ ou -)
- Mutualisations des ressources
- Ressources à la demande
- Démarrage quasiment aussi rapide qu'une application native
- Contrairement aux VMs, pas d'overhead (économie des ressources matériels)

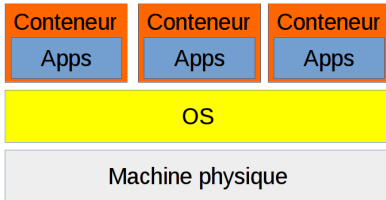
Architecture simplifiée du modèle conteneur/OS



Avantages

- Portabilité des conteneurs (+ ou -)
- Isolations des applications (+ ou -)
- Mutualisations des ressources
- Ressources à la demande
- Démarrage quasiment aussi rapide qu'une application native
- Contrairement aux VMs, pas d'overhead (économie des ressources matériels)

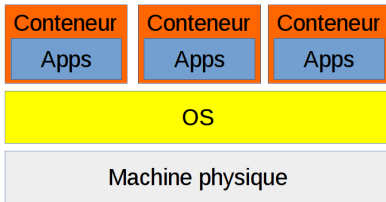
Architecture simplifiée du modèle conteneur/OS



Avantages

- Portabilité des conteneurs (+ ou -)
- Isolations des applications (+ ou -)
- Mutualisations des ressources
- Ressources à la demande
- Démarrage quasiment aussi rapide qu'une application native
- Contrairement aux VMs, pas d'overhead (économisation des ressources matériels)

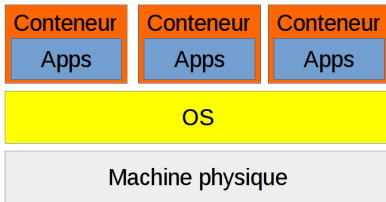
Architecture simplifiée du modèle conteneur/OS



Inconvénients

- Ne permet d'exécuter que des applications compatibles avec le système hôte
- Peut-être sans état, volatile (dépend des technologies employées) ⇒ pas de snapshot RAM incluse, pas de persistance des données nativement

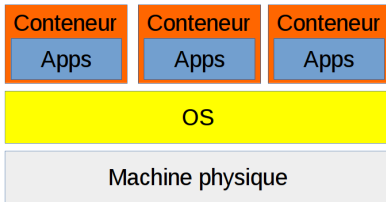
Architecture simplifiée du modèle conteneur/OS



Inconvénients

- Ne permet d'exécuter que des applications compatibles avec le système hôte
- Peut-être sans état, volatile (dépend des technologies employées) ⇒ pas de snapshot RAM incluse, pas de persistance des données nativement

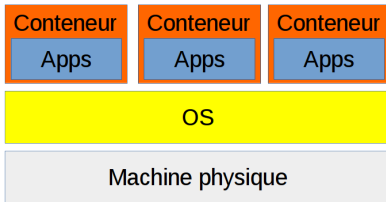
Architecture simplifiée du modèle conteneur/OS



Inconvénients (suite)

- Certaines de ces technologies sont encore jeune (docker, singularity)
- Courbe d'apprentissage un peu plus élevé que pour des VM
- Évolution très (trop ?) rapide
- Va-t-on vraiment vers un changement de paradigme ?

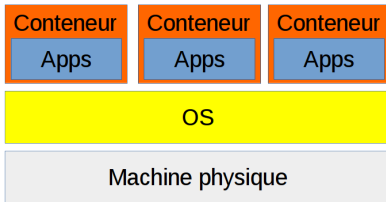
Architecture simplifiée du modèle conteneur/OS



Inconvénients (suite)

- Certaines de ces technologies sont encore jeune (docker, singularity)
- Courbe d'apprentissage un peu plus élevé que pour des VM
- Évolution très (trop ?) rapide
- Va-t-on vraiment vers un changement de paradigme ?

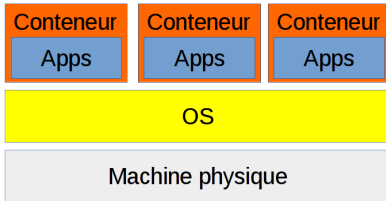
Architecture simplifiée du modèle conteneur/OS



Inconvénients (suite)

- Certaines de ces technologies sont encore jeune (docker, singularity)
- Courbe d'apprentissage un peu plus élevé que pour des VM
- Évolution très (trop ?) rapide
- Va-t-on vraiment vers un changement de paradigme ?

Architecture simplifiée du modèle conteneur/OS



Inconvénients (suite)

- Certaines de ces technologies sont encore jeune (docker, singularity)
- Courbe d'apprentissage un peu plus élevé que pour des VM
- Évolution très (trop ?) rapide
- Va-t-on vraiment vers un changement de paradigme ?

Divers

On peut imbriquer plusieurs VM ou plusieurs conteneurs les uns dans les autres
(en anglais nested virtualisation/containers)

Chroot : le prémice de la conteneurisation

- Réalisez un chroot pour un utilisateur d'uid 2000 sur un répertoire nommé new_root, vous aurez besoin de :
 - chroot (et de man chroot)
 - cp
 - ldd
 - mkdir

Chroot : le prémice de la conteneurisation

Terminal:

```
jmc@laptop/home/jmc $ mkdir new_root  
jmc@laptop/home/jmc $ cp -r /bin/ new_root/  
jmc@laptop/home/jmc $ sudo chroot new_root /bin/bash  
jmc@laptop/home/jmc $ # Zut alors
```

Chroot : le prémice de la conteneurisation

Terminal:

```
jmc@laptop/home/jmc $ ldd /bin/bash
jmc@laptop/home/jmc $ cp -r /lib new_root/
jmc@laptop/home/jmc $ cp -r /lib64 new_root/
jmc@laptop/home/jmc $ sudo chroot new_root /bin/bash
bash-4.3# pwd
/
bash-4.3# echo $UID
0
bash-4.3# exit
jmc@laptop/home/jmc $
```

Chroot : le prémice de la conteneurisation

Terminal:

```
jmc@laptop/home/jmc $ sudo chroot --userspec 2000 :2000 new_root /bin/bash
bash-4.3# echo $UID
2000
bash-4.3# ls -l / # A vous de voir
bash-4.3# rm -r --no-preserve-root / # A votre avis ? Et lorsque on ne met pas
l'option userspec ?
bash-4.3# exit
jmc@laptop/home/jmc $
```

Plan

1 Introduction générale

- Conteneur versus machine virtuelle
- Des technologies de conteneurs différentes pour des objectifs différents

2 Bases

- Introduction
- Définitions et prérequis
- Premiers pas
- Volatilité, persistance et sécurité des données
- Sécurité des conteneurs
- Conteneurs graphiques

3 Utilisation avancées

- Créations d'images : méthode naïve
- Créations d'images : Dockerfile

4 Toujours plus loin

- Travaux pratiques

Les conteneurs applicatifs

Il s'agit des conteneurs qui ont pour objectif de n'exécuter qu'une application, qu'un service, voir même qu'un seul processus. Les deux principaux concurrents sont :

- Docker (Docker Inc.)
- Rocket (CoreOS Inc.)

Ce sont des technologies très prisées par le monde du DevOps (relation entre les développeurs et les administrateurs systèmes/réseaux).

L'utilisation de ces technologies met en avant l'utilisation des architectures de micro-services (1 conteneur = 1 micro-service, multi-conteneurisations du service).

Il s'agit de conteneur sans état (stateless), il n'y a pas de persistance de donnée par défaut, car ces conteneurs sont volatiles.

Les conteneurs pour l'administration système

Ce sont des conteneurs qui ont pour objectif de remplacer les VM. Ils diffèrent des conteneurs applicatifs dans le sens où ils peuvent fournir un **ensemble** de (micro) services. Ce sont des conteneurs avec état (statefull) : ils offrent une persistance des données et ils ont pour vocation d'être manipulés comme des VMs (snapshot, backup, migration à chaud). Les principaux concurrents sont :

- LXC (via LXD, Canonical)
- LXC (via Proxmox pct, Proxmox)
- OpenVZ

Les conteneurs LXC sont capables d'héberger des conteneurs Docker (sous conditions : voir

<https://stgraber.org/2016/04/13/lxd-2-0-docker-in-lxd-712/>).

Les conteneurs pour le calcul haute performance

L'idée est de se servir du mécanisme des conteneurs pour offrir des environnements isolés mais performants dans le cadre du calcul haute performance. Singularity permet l'utilisation de conteneur à destination du HPC (utilisation sur des clusters de calculs). Les conteneurs singularity sont en mesure d'exécuter des conteneurs docker. Ils sont également plus sécurisés nativement et permettent l'utilisation native de certains protocoles (infiniband, openMPI, ...)

Plan

- 1 Introduction générale
 - Conteneur versus machine virtuelle
 - Des technologies de conteneurisations différentes pour des objectifs différents
- 2 Bases
 - Introduction
 - Définitions et prérequis
 - Premiers pas
 - Volatilité, persistance et sécurité des données
 - Sécurité des conteneurs
 - Conteneurs graphiques
- 3 Utilisation avancées
 - Créations d'images : méthode naïve
 - Créations d'images : Dockerfile
- 4 Toujours plus loin
 - Travaux pratiques

La puissance de docker ?

Démonstration wordpress PWD

Contexte d'utilisation de docker

- Dans le cadre du cours, notre utilisation de Docker se fera uniquement sur un noyau Linux. En effet, il existe une possibilité d'utiliser docker nativement sur Windows server 2016 et pour conteneuriser des applications windows.
- On peut utiliser docker nativement sur un OS GNU / Linux.
- On peut utiliser docker dans une machine virtuelle avec un OS GNU/Linux.
- On peut utiliser docker via une virtualisation Virtual Box minimal de l'OS "boot2docker" via l'outil "docker toolbox" sur Windows et macOS.
- On peut avoir un environnement "bac à sable" docker en ligne avec l'excellent site "play with docker" (nécessite un compte docker)

Contexte d'utilisation de docker

- Dans le cadre du cours, notre utilisation de Docker se fera uniquement sur un noyau Linux. En effet, il existe une possibilité d'utiliser docker nativement sur Windows server 2016 et pour conteneuriser des applications windows.
- On peut utiliser docker nativement sur un OS GNU / Linux.
- On peut utiliser docker dans une machine virtuelle avec un os GNU/linux.
- On peut utiliser docker via une virtualisation Virtual Box minimal de l'os "boot2docker" via l'outil "docker toolbox" sur windows et macOS.
- On peut avoir un environnement "bac à sable" docker en ligne avec l'excellent site "play with docker" (nécessite un compte docker)

Contexte d'utilisation de docker

- Dans le cadre du cours, notre utilisation de Docker se fera uniquement sur un noyau Linux. En effet, il existe une possibilité d'utiliser docker nativement sur Windows server 2016 et pour conteneuriser des applications windows.
- On peut utiliser docker nativement sur un OS GNU / Linux.
- On peut utiliser docker dans une machine virtuelle avec un OS GNU/Linux.
- On peut utiliser docker via une virtualisation Virtual Box minimal de l'OS "boot2docker" via l'outil "docker toolbox" sur Windows et macOS.
- On peut avoir un environnement "bac à sable" docker en ligne avec l'excellent site "play with docker" (nécessite un compte docker)

Contexte d'utilisation de docker

- Dans le cadre du cours, notre utilisation de Docker se fera uniquement sur un noyau Linux. En effet, il existe une possibilité d'utiliser docker nativement sur Windows server 2016 et pour conteneuriser des applications windows.
- On peut utiliser docker nativement sur un OS GNU / Linux.
- On peut utiliser docker dans une machine virtuelle avec un OS GNU/Linux.
- On peut utiliser docker via une virtualisation Virtual Box minimal de l'OS "boot2docker" via l'outil "docker toolbox" sur Windows et macOS.
- On peut avoir un environnement "bac à sable" docker en ligne avec l'excellent site "play with docker" (nécessite un compte docker)

Contexte d'utilisation de docker

- Dans le cadre du cours, notre utilisation de Docker se fera uniquement sur un noyau Linux. En effet, il existe une possibilité d'utiliser docker nativement sur Windows server 2016 et pour conteneuriser des applications windows.
- On peut utiliser docker nativement sur un OS GNU / Linux.
- On peut utiliser docker dans une machine virtuelle avec un OS GNU/Linux.
- On peut utiliser docker via une virtualisation Virtual Box minimal de l'OS "boot2docker" via l'outil "docker toolbox" sur Windows et macOS.
- On peut avoir un environnement "bac à sable" docker en ligne avec l'excellent site "play with docker" (nécessite un compte docker)

Historique

- Solomon Hykes débute le projet Docker en interne pour DotCloud en France.
- Docker a été distribué en tant que projet open source à partir de mars 2013
- À partir de la version 0.9, Docker abandonne l'utilisation de LXC et développe sa propre librairie de conteneur : libcontainer
- Docker est principalement développé par la société Docker Inc
- En 3 ans, 10 000 forks et 1400 contributeurs sur github pour le projet Docker

Carte d'identité

Docker



Développeur	Docker, Inc. ^(en) et Solomon Hykes 
Première version	13 mars 2013
Dernière version	17.05.0 (5 mai 2017) ¹ 
Version avancée	17.04.0 (6 avril 2017) ^{2,3,4} 
État du projet	En développement actif
Écrit en	Go 
Environnement	Multi-plateforme
Type	Gestionnaire de containers
Politique de distribution	Gratuit
Licence	Apache-2.0 ^(d) ⁵ 
Site web	www.docker.com  [archive] 

[modifier](#) 

N.B : Versions de docker

Il y a eu 2 grands changements majeurs dans l'historique de développements de docker :

- docker 0.9 \Rightarrow fork de LXC pour création de libcontainer
- docker 1.10 \Rightarrow Changement de gestion des images (4 février 2016)

Ce cours a été fait sur la base de la version 17.03.1-ce. Les explications données ici peuvent être **incompatibles** avec les versions antérieures à docker 1.10. Si vous êtes amenés à consulter des forums sur internet, pensez à regarder la date du post et/ou la version de docker utilisé. Plus d'informations sur ces changements majeurs :

- <https://blog.docker.com/2016/02/docker-1-10/>
- <http://windsock.io/explaining-docker-image-ids/>

N.B : Versions de docker

Il y a eu 2 grands changements majeurs dans l'historique de développements de docker :

- docker 0.9 \Rightarrow fork de LXC pour création de libcontainer
- docker 1.10 \Rightarrow Changement de gestion des images (4 février 2016)

Ce cours a été fait sur la base de la version 17.03.1-ce. Les explications données ici peuvent être **incompatibles** avec les versions antérieures à docker 1.10. Si vous êtes amenés à consulter des forums sur internet, pensez à regarder la date du post et/ou la version de docker utilisé. Plus d'informations sur ces changements majeurs :

- <https://blog.docker.com/2016/02/docker-1-10/>
- <http://windsock.io/explaining-docker-image-ids/>

N.B : Versions de docker

Il y a eu 2 grands changements majeurs dans l'historique de développements de docker :

- docker 0.9 \Rightarrow fork de LXC pour création de libcontainer
- docker 1.10 \Rightarrow Changement de gestion des images (4 février 2016)

Ce cours a été fait sur la base de la version 17.03.1-ce. Les explications données ici peuvent être **incompatibles** avec les versions antérieures à docker 1.10. Si vous êtes amenés à consulter des forums sur internet, pensez à regarder la date du post et/ou la version de docker utilisé. Plus d'informations sur ces changements majeurs :

- <https://blog.docker.com/2016/02/docker-1-10/>
- <http://windsock.io/explaining-docker-image-ids/>

Installation

- Ubuntu 16.04 (VM ou natif)
- Dernière version stable de docker en CE (Community Edition)
- <https://docs.docker.com/engine/installation/linux/ubuntu/#uninstall-old-versions>
- <https://docs.docker.com/engine/installation/linux/ubuntu/#install-using-the-repository>

Pour ne pas avoir à écrire *sudo* à chaque commande docker

Terminal:

```
jmc@laptop/home/jmc $ sudo usermod -aG docker $USER
jmc@laptop/home/jmc $ # Déconnexion, reconnexion
jmc@laptop/home/jmc $ systemctl enable docker # Activation au démarrage de
l'hôte
```

- Considérez que vous êtes root à chaque fois que votre commande est préfixé par docker

Pour ne pas avoir à écrire *sudo* à chaque commande docker

Terminal:

```
jmc@laptop/home/jmc $ sudo usermod -aG docker $USER
jmc@laptop/home/jmc $ # Déconnexion, reconnexion
jmc@laptop/home/jmc $ systemctl enable docker # Activation au démarrage de
l'hôte
```

- Considérez que vous êtes root à chaque fois que votre commande est préfixé par docker

Plan

- 1 Introduction générale
 - Conteneur versus machine virtuelle
 - Des technologies de conteneurisations différentes pour des objectifs différents
- 2 Bases
 - Introduction
 - Définitions et prérequis
 - Premiers pas
 - Volatilité, persistance et sécurité des données
 - Sécurité des conteneurs
 - Conteneurs graphiques
- 3 Utilisation avancées
 - Créations d'images : méthode naïve
 - Créations d'images : Dockerfile
- 4 Toujours plus loin
 - Travaux pratiques

Définitions formelles

Les images et les couches : bases des conteneurs docker

- Une image désigne une collection **ordonnée** (par empilement) de **modifications** d'un système de fichiers racine.
- On conserve chacune de ces modifications du FS dans **une couche ou un layer** identifiable par le hash sha256 de son contenu. \Rightarrow Une image désigne donc une collection **ordonnée** de couche.
- A chaque couche **intermédiaire** est associé une image **intermédiaire**.
- Les couches sont créées grâce à l'exécution de conteneurs qui sont transformés en image. (Pas de panique, on va faire un schéma)
- Il peut y avoir des couches vide(empty layer).
- Les couches désignées par une image sont nécessairement en lecture seule.
- Une image est **unique**. Elle est identifiée par une fonction de hachage sha256 de son descriptif. Elle **n'a pas d'état** (i.e. : on n'exécute pas une image) et elle est **immuable**.

Définitions formelles

Les images et les couches : bases des conteneurs docker

- Une image désigne une collection **ordonnée** (par empilement) de **modifications** d'un système de fichiers racine.
- On conserve chacune de ces modifications du FS dans **une couche ou un layer** identifiable par le hash sha256 de son contenu. \Rightarrow Une image désigne donc une collection **ordonnée** de couche.
- A chaque couche **intermédiaire** est associé une image **intermédiaire**.
- Les couches sont créées grâce à l'exécution de conteneurs qui sont transformés en image. (Pas de panique, on va faire un schéma)
- Il peut y avoir des couches vide(empty layer).
- Les couches désignées par une image sont nécessairement en lecture seule.
- Une image est **unique**. Elle est identifiée par une fonction de hachage sha256 de son descriptif. Elle **n'a pas d'état** (i.e. : on n'exécute pas une image) et elle est **immuable**.

Définitions formelles

Les images et les couches : bases des conteneurs docker

- Une image désigne une collection **ordonnée** (par empilement) de **modifications** d'un système de fichiers racine.
- On conserve chacune de ces modifications du FS dans **une couche ou un layer** identifiable par le hash sha256 de son contenu. \Rightarrow Une image désigne donc une collection **ordonnée** de couche.
- A chaque couche **intermédiaire** est associé une image **intermédiaire**.
- Les couches sont créées grâce à l'exécution de conteneurs qui sont transformés en image. (Pas de panique, on va faire un schéma)
- Il peut y avoir des couches vide(empty layer).
- Les couches désignées par une image sont nécessairement en lecture seule.
- Une image est **unique**. Elle est identifiée par une fonction de hachage sha256 de son descriptif. Elle **n'a pas d'état** (i.e. : on n'exécute pas une image) et elle est **immuable**.

Définitions formelles

Les images et les couches : bases des conteneurs docker

- Une image désigne une collection **ordonnée** (par empilement) de **modifications** d'un système de fichiers racine.
- On conserve chacune de ces modifications du FS dans **une couche ou un layer** identifiable par le hash sha256 de son contenu. \Rightarrow Une image désigne donc une collection **ordonnée** de couche.
- A chaque couche **intermédiaire** est associé une image **intermédiaire**.
- Les couches sont créées grâce à l'exécution de conteneurs qui sont transformés en image. (Pas de panique, on va faire un schéma)
- Il peut y avoir des couches vide(empty layer).
- Les couches désignées par une image sont nécessairement en lecture seule.
- Une image est **unique**. Elle est identifiée par une fonction de hachage sha256 de son descriptif. Elle **n'a pas d'état** (i.e. : on n'exécute pas une image) et elle est **immuable**.

Définitions formelles

Les images et les couches : bases des conteneurs docker

- Une image désigne une collection **ordonnée** (par empilement) de **modifications** d'un système de fichiers racine.
- On conserve chacune de ces modifications du FS dans **une couche ou un layer** identifiable par le hash sha256 de son contenu. \Rightarrow Une image désigne donc une collection **ordonnée** de couche.
- A chaque couche **intermédiaire** est associé une image **intermédiaire**.
- Les couches sont créées grâce à l'exécution de conteneurs qui sont transformés en image. (Pas de panique, on va faire un schéma)
- Il peut y avoir des couches vide(empty layer).
- Les couches désignées par une image sont nécessairement en lecture seule.
- Une image est **unique**. Elle est identifiée par une fonction de hachage sha256 de son descriptif. Elle **n'a pas d'état** (i.e. : on n'exécute pas une image) et elle est **immuable**.

Définitions formelles

Les images et les couches : bases des conteneurs docker

- Une image désigne une collection **ordonnée** (par empilement) de **modifications** d'un système de fichiers racine.
- On conserve chacune de ces modifications du FS dans **une couche ou un layer** identifiable par le hash sha256 de son contenu. \Rightarrow Une image désigne donc une collection **ordonnée** de couche.
- A chaque couche **intermédiaire** est associé une image **intermédiaire**.
- Les couches sont créées grâce à l'exécution de conteneurs qui sont transformés en image. (Pas de panique, on va faire un schéma)
- Il peut y avoir des couches vide(empty layer).
- Les couches désignées par une image sont nécessairement en lecture seule.
- Une image est **unique**. Elle est identifiée par une fonction de hachage sha256 de son descriptif. Elle **n'a pas d'état** (i.e. : on n'exécute pas une image) et elle est **immuable**.

Définitions formelles

Les images et les couches : bases des conteneurs docker

- Une image désigne une collection **ordonnée** (par empilement) de **modifications** d'un système de fichiers racine.
- On conserve chacune de ces modifications du FS dans **une couche ou un layer** identifiable par le hash sha256 de son contenu. \Rightarrow Une image désigne donc une collection **ordonnée** de couche.
- A chaque couche **intermédiaire** est associé une image **intermédiaire**.
- Les couches sont créées grâce à l'exécution de conteneurs qui sont transformés en image. (Pas de panique, on va faire un schéma)
- Il peut y avoir des couches vide(empty layer).
- Les couches désignées par une image sont nécessairement en lecture seule.
- Une image est **unique**. Elle est identifiée par une fonction de hachage sha256 de son descriptif. Elle **n'a pas d'état** (i.e. : on n'exécute pas une image) et elle est **immuable**.

Un conteneur : l'instanciation d'une image

- Le système de fichier racine du conteneur **est** le système de fichier racine de l'image-sous-jacente, **mais** celui-ci est en lecture seule.
- On a l'**illusion** de modifier le système de fichier racine dans le conteneur, car les changements sont effectués dans **une nouvelle couche en lecture/écriture**. (technologie de Copy On Write (COW))
- La suppression du conteneur entraîne **systématiquement la suppression de cette nouvelle couche en lecture/écriture**.
- On peut pérenniser les modifications effectuées dans le FS d'un conteneur en créant une **nouvelle** image qui va ajouter aux couches en lecture seule la nouvelle couche ainsi créée. La nouvelle couche qui était en lecture/écriture passe ainsi en lecture seule.
- Cette pérennisation se fait via la commande *docker commit*

Un conteneur : l'instanciation d'une image

- Le système de fichier racine du conteneur **est** le système de fichier racine de l'image-sous-jacente, **mais** celui-ci est en lecture seule.
- On a **l'illusion** de modifier le système de fichier racine dans le conteneur, car les changements sont effectués dans **une nouvelle couche en lecture/écriture**. (technologie de Copy On Write (COW))
- La suppression du conteneur entraîne **systématiquement la suppression de cette nouvelle couche en lecture/écriture**.
- On peut pérenniser les modifications effectuées dans le FS d'un conteneur en créant une **nouvelle** image qui va ajouter aux couches en lecture seule la nouvelle couche ainsi créée. La nouvelle couche qui était en lecture/écriture passe ainsi en lecture seule.
- Cette pérennisation se fait via la commande *docker commit*

Un conteneur : l'instanciation d'une image

- Le système de fichier racine du conteneur **est** le système de fichier racine de l'image-sous-jacente, **mais** celui-ci est en lecture seule.
- On a **l'illusion** de modifier le système de fichier racine dans le conteneur, car les changements sont effectués dans **une nouvelle couche en lecture/écriture**. (technologie de Copy On Write (COW))
- La suppression du conteneur entraîne **systématiquement la suppression de cette nouvelle couche en lecture/écriture**.
- On peut pérenniser les modifications effectuées dans le FS d'un conteneur en créant une **nouvelle** image qui va ajouter aux couches en lecture seule la nouvelle couche ainsi créée. La nouvelle couche qui était en lecture/écriture passe ainsi en lecture seule.
- Cette pérennisation se fait via la commande *docker commit*

Un conteneur : l'instanciation d'une image

- Le système de fichier racine du conteneur **est** le système de fichier racine de l'image-sous-jacente, **mais** celui-ci est en lecture seule.
- On a **l'illusion** de modifier le système de fichier racine dans le conteneur, car les changements sont effectués dans **une nouvelle couche en lecture/écriture**. (technologie de Copy On Write (COW))
- La suppression du conteneur entraîne **systématiquement la suppression de cette nouvelle couche en lecture/écriture**.
- On peut pérenniser les modifications effectuées dans le FS d'un conteneur en créant une **nouvelle** image qui va ajouter aux couches en lecture seule la nouvelle couche ainsi créée. La nouvelle couche qui était en lecture/écriture passe ainsi en lecture seule.
- Cette pérennisation se fait via la commande *docker commit*

Un conteneur : l'instanciation d'une image

- Le système de fichier racine du conteneur **est** le système de fichier racine de l'image-sous-jacente, **mais** celui-ci est en lecture seule.
- On a **l'illusion** de modifier le système de fichier racine dans le conteneur, car les changements sont effectués dans **une nouvelle couche en lecture/écriture**. (technologie de Copy On Write (COW))
- La suppression du conteneur entraîne **systématiquement la suppression de cette nouvelle couche en lecture/écriture**.
- On peut pérenniser les modifications effectuées dans le FS d'un conteneur en créant une **nouvelle** image qui va ajouter aux couches en lecture seule la nouvelle couche ainsi créée. La nouvelle couche qui était en lecture/écriture passe ainsi en lecture seule.
- Cette pérennisation se fait via la commande *docker commit*

Ce qu'il faut bien comprendre

- Image \neq Couche \neq Conteneur
- Une couche contient **uniquement** les **différences** d'un système de fichier.
- Un **ensemble** de couche peut former un système de fichier racine.
- Les couches ne savent rien des images qui les exploitent.
- Les couches peuvent être utilisés par **plusieurs** images.
- Une image peut faire référence à **plus d'1** couche.
- Une image est quelque chose que vous pouvez instancier. Le **résultat** de cette instanciation est un conteneur.
- Lorsque un conteneur démarre, certaines informations de l'image ayant permis son instanciation permettent de construire son système de fichier racine.

Ce qu'il faut bien comprendre

- Image \neq Couche \neq Conteneur
- Une couche contient **uniquement** les **différences** d'un système de fichier.
- Un **ensemble** de couche peut former un système de fichier racine.
- Les couches ne savent rien des images qui les exploitent.
- Les couches peuvent être utilisés par **plusieurs** images.
- Une image peut faire référence à **plus d'1** couche.
- Une image est quelque chose que vous pouvez instancier. Le **résultat** de cette instanciation est un conteneur.
- Lorsque un conteneur démarre, certaines informations de l'image ayant permis son instanciation permettent de construire son système de fichier racine.

Ce qu'il faut bien comprendre

- Image \neq Couche \neq Conteneur
- Une couche contient **uniquement** les **différences** d'un système de fichier.
- Un **ensemble** de couche peut former un système de fichier racine.
- Les couches ne savent rien des images qui les exploitent.
- Les couches peuvent être utilisés par **plusieurs** images.
- Une image peut faire référence à **plus d'1** couche.
- Une image est quelque chose que vous pouvez instancier. Le **résultat** de cette instanciation est un conteneur.
- Lorsque un conteneur démarre, certaines informations de l'image ayant permis son instanciation permettent de construire son système de fichier racine.

Ce qu'il faut bien comprendre

- Image \neq Couche \neq Conteneur
- Une couche contient **uniquement** les **différences** d'un système de fichier.
- Un **ensemble** de couche peut former un système de fichier racine.
- Les couches ne savent rien des images qui les exploitent.
- Les couches peuvent être utilisés par **plusieurs** images.
- Une image peut faire référence à **plus d'1** couche.
- Une image est quelque chose que vous pouvez instancier. Le **résultat** de cette instanciation est un conteneur.
- Lorsque un conteneur démarre, certaines informations de l'image ayant permis son instanciation permettent de construire son système de fichier racine.

Ce qu'il faut bien comprendre

- Image \neq Couche \neq Conteneur
- Une couche contient **uniquement** les **différences** d'un système de fichier.
- Un **ensemble** de couche peut former un système de fichier racine.
- Les couches ne savent rien des images qui les exploitent.
- Les couches peuvent être utilisés par **plusieurs** images.
- Une image peut faire référence à **plus d'1** couche.
- Une image est quelque chose que vous pouvez instancier. Le **résultat** de cette instanciation est un conteneur.
- Lorsque un conteneur démarre, certaines informations de l'image ayant permis son instanciation permettent de construire son système de fichier racine.

Ce qu'il faut bien comprendre

- Image \neq Couche \neq Conteneur
- Une couche contient **uniquement** les **différences** d'un système de fichier.
- Un **ensemble** de couche peut former un système de fichier racine.
- Les couches ne savent rien des images qui les exploitent.
- Les couches peuvent être utilisés par **plusieurs** images.
- Une image peut faire référence à **plus d'1** couche.
- Une image est quelque chose que vous pouvez instancier. Le **résultat** de cette instanciation est un conteneur.
- Lorsque un conteneur démarre, certaines informations de l'image ayant permis son instanciation permettent de construire son système de fichier racine.

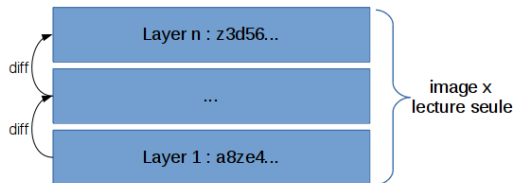
Ce qu'il faut bien comprendre

- Image \neq Couche \neq Conteneur
- Une couche contient **uniquement** les **différences** d'un système de fichier.
- Un **ensemble** de couche peut former un système de fichier racine.
- Les couches ne savent rien des images qui les exploitent.
- Les couches peuvent être utilisés par **plusieurs** images.
- Une image peut faire référence à **plus d'1** couche.
- Une image est quelque chose que vous pouvez instancier. Le **résultat** de cette instanciation est un conteneur.
- Lorsque un conteneur démarre, certaines informations de l'image ayant permis son instanciation permettent de construire son système de fichier racine.

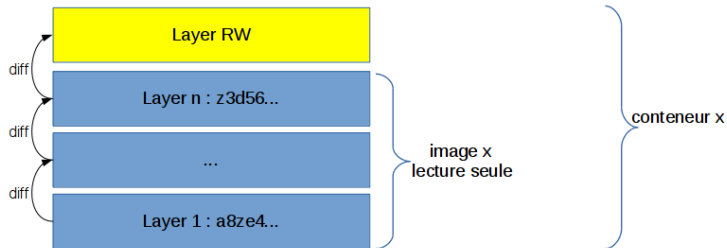
Ce qu'il faut bien comprendre

- Image \neq Couche \neq Conteneur
- Une couche contient **uniquement** les **différences** d'un système de fichier.
- Un **ensemble** de couche peut former un système de fichier racine.
- Les couches ne savent rien des images qui les exploitent.
- Les couches peuvent être utilisés par **plusieurs** images.
- Une image peut faire référence à **plus d'1** couche.
- Une image est quelque chose que vous pouvez instancier. Le **résultat** de cette instanciation est un conteneur.
- Lorsque un conteneur démarre, certaines informations de l'image ayant permis son instanciation permettent de construire son système de fichier racine.

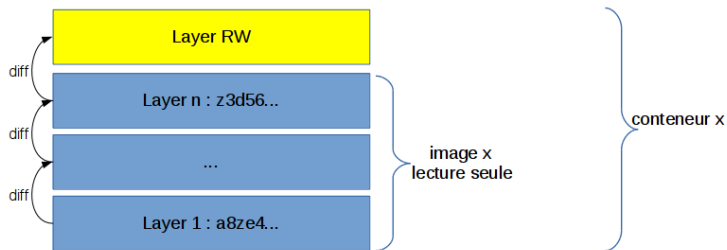
Représentation graphique



Représentation graphique

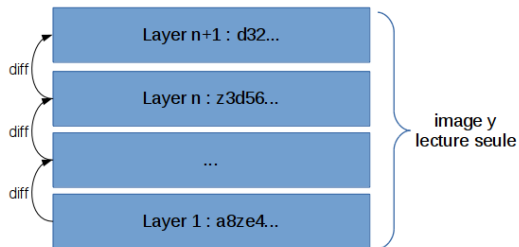


Représentation graphique



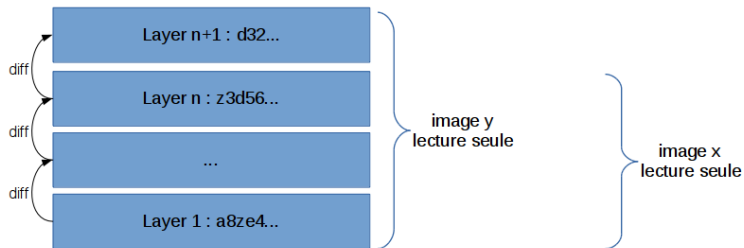
- Les couches de l'image sont obtenues grâce à une succession de conteneurs commités

Représentation graphique



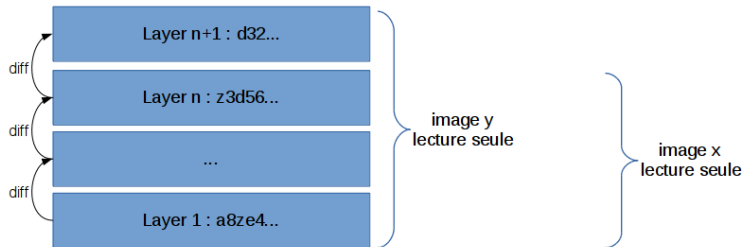
- docker commit x y (On crée une image y à partir du conteneur x (équivalent d'un snapshot à froid))

Représentation graphique



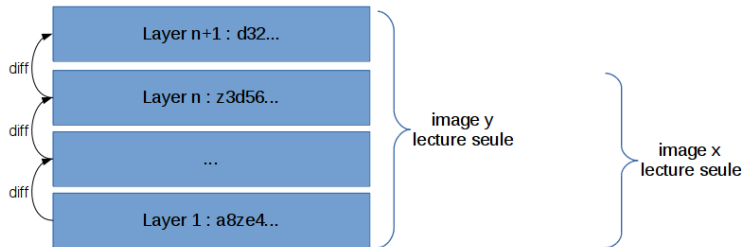
- Avantage : image x et y ont n layer en commun (mutualisation des ressources)

Représentation graphique



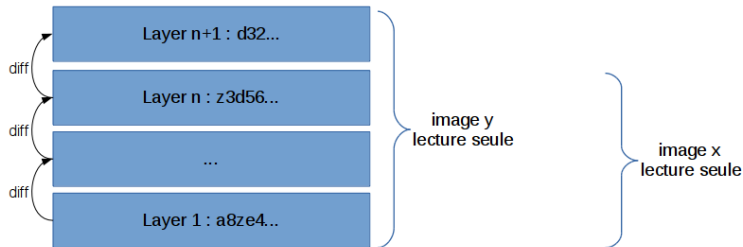
- Avantage : image x et y ont n layer en commun (mutualisation des ressources)
- Chaque layer construit **localement** a une image correspondante, on parle d'image intermédiaire (docker images --all)

Représentation graphique



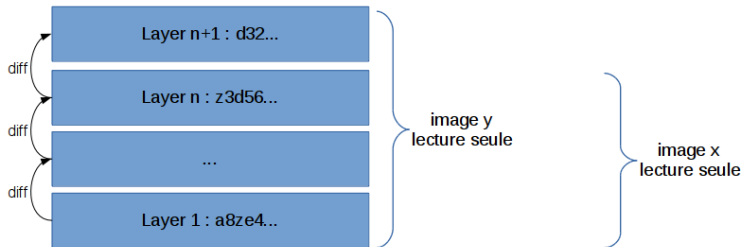
- **Avantage :** image x et y ont n layer en commun (mutualisation des ressources)
- Chaque layer construit localement a une image correspondante, on parle d'image intermédiaire (docker images –all)
- Lorsque l'on supprime une image :
 - Si les layers sont utilisés par d'autres images, ceux ci ne seront pas supprimés.

Représentation graphique



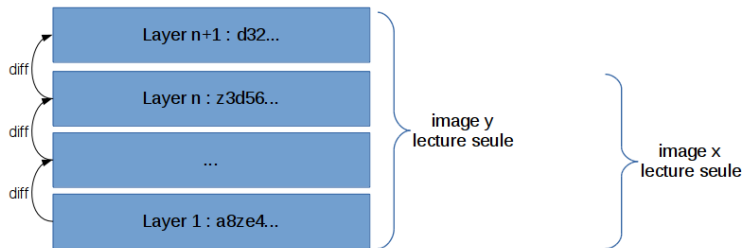
- **Avantage :** image x et y ont n layer en commun (mutualisation des ressources)
- Chaque layer construit localement a une image correspondante, on parle d'image intermédiaire (docker images –all)
- Lorsque l'on supprime une image :
 - Si les layers sont utilisés par d'autres images, ceux ci ne seront pas supprimés.
 - Sinon, il y a une destruction en cascade des images parentes intermédiaire et de leurs layers associés.

Représentation graphique



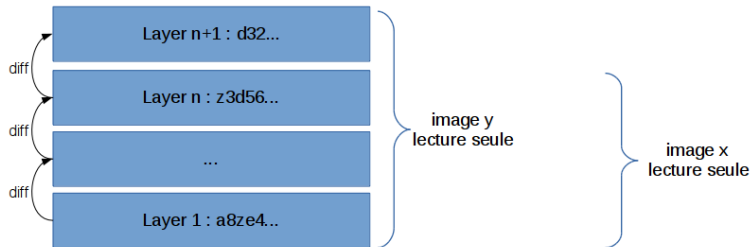
- **Avantage :** image x et y ont n layer en commun (mutualisation des ressources)
- Chaque layer construit localement a une image correspondante, on parle d'image intermédiaire (docker images --all)
- Lorsque l'on supprime une image :
 - Si les layers sont utilisés par d'autres images, ceux ci ne seront pas supprimés.
 - Sinon, il y a une destruction en cascade des images parentes intermédiaire et de leurs layers associés.
 - La suppression de x (resp. y) n'empêche pas l'utilisation de y (resp. x),

Représentation graphique



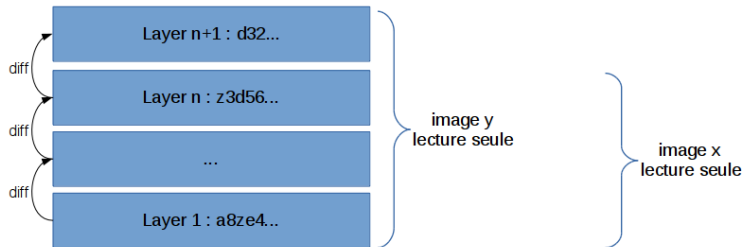
- On peut visualiser l'architecture des images sous la forme d'un arbre : les feuilles sont les images finales, elles n'ont pas d'enfants.

Représentation graphique



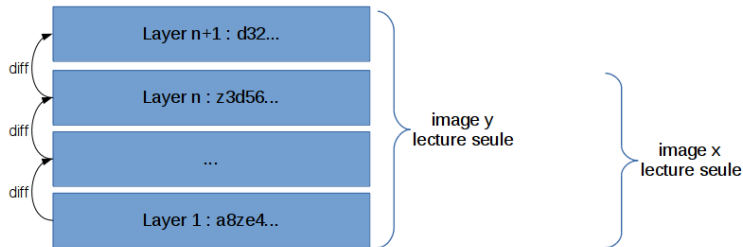
- On peut visualiser l'architecture des images sous la forme d'un arbre : les feuilles sont les images finales, elles n'ont pas d'enfants.
- On dit que l'image y **à pour parent** l'image x

Représentation graphique



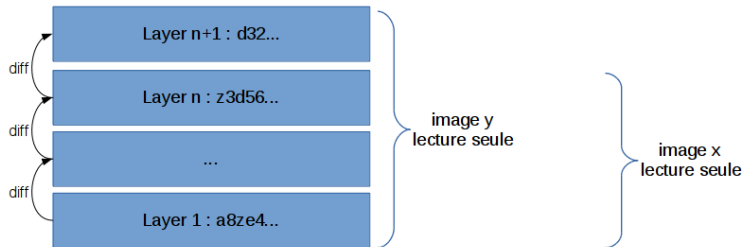
- On peut visualiser l'architecture des images sous la forme d'un arbre : les feuilles sont les images finales, elles n'ont pas d'enfants.
- On dit que l'image y **à pour parent** l'image x
- Une image **feuille** (top level image) qui n'a pas de tag (<none>) est dite « **pendante** » (dangling)

Représentation graphique



- On peut visualiser l'architecture des images sous la forme d'un arbre : les feuilles sont les images finales, elles n'ont pas d'enfants.
- On dit que l'image y **à pour parent** l'image x
- Une image **feuille** (top level image) qui n'a pas de tag (<none>) est dite « **pendante** » (dangling)
- Une image **feuille** destiné à être utilisée devrait toujours être taggé

Représentation graphique



- On peut visualiser l'architecture des images sous la forme d'un arbre : les feuilles sont les images finales, elles n'ont pas d'enfants.
- On dit que l'image y **à pour parent** l'image x
- Une image **feuille** (top level image) qui n'a pas de tag (<none>) est dite « **pendante** » (dangling)
- Une image **feuille** destiné à être utilisée devrait toujours être taggé
- Lors de la conception d'une architecture, il faudrait **maximiser le nombre de couches partagés et minimiser le nombre d'image de base**

Images particulières

- Une "top level image" est une image sans enfant
- Une image taguée "none" est une image intermédiaire, sinon c'est une image pendante (dangling)
- Une image taguée "missing" est une image manquante mais dont on a obtenu le layer associé (via *docker pull*)

Avant la version 1.10

- Chaque image était identifié par un nombre aléatoire
- Chaque image était associé à 1 seul couche.
- ⇒ Chaque couche était repéré par le même identifiant que l'image
- ⇒ Problèmes de sécurité à cause du nombre aléatoire
- Pour plus d'informations :
<http://windsock.io/explaining-docker-image-ids/>

Images particulières

- Une "top level image" est une image sans enfant
- Une image taguée "none" est une image intermédiaire, sinon c'est une image pendante (dangling)
- Une image taguée "missing" est une image manquante mais dont on a obtenu le layer associé (via *docker pull*)

Avant la version 1.10

- Chaque image était identifié par un nombre aléatoire
- Chaque image était associé à 1 seul couche.
- ⇒ Chaque couche était repéré par le même identifiant que l'image
- ⇒ Problèmes de sécurité à cause du nombre aléatoire
- Pour plus d'informations :
<http://windsock.io/explaining-docker-image-ids/>

Images particulières

- Une "top level image" est une image sans enfant
- Une image taguée "none" est une image intermédiaire, sinon c'est une image pendante (dangling)
- Une image taguée "missing" est une image manquante mais dont on a obtenu le layer associé (via *docker pull*)

Avant la version 1.10

- Chaque image était identifié par un nombre aléatoire
- Chaque image était associé à **1 seul** couche.
- ⇒ Chaque couche était repéré par le même identifiant que l'image
- ⇒ Problèmes de sécurité à cause du nombre aléatoire
- Pour plus d'informations :
<http://windsock.io/explaining-docker-image-ids/>

Images particulières

- Une "top level image" est une image sans enfant
- Une image taguée "none" est une image intermédiaire, sinon c'est une image pendante (dangling)
- Une image taguée "missing" est une image manquante mais dont on a obtenu le layer associé (via *docker pull*)

Avant la version 1.10

- Chaque image était identifié par un nombre aléatoire
- Chaque image était associé à **1 seul** couche.
- ⇒ Chaque couche était repéré par le même identifiant que l'image
- ⇒ Problèmes de sécurité à cause du nombre aléatoire
- Pour plus d'informations :
<http://windsock.io/explaining-docker-image-ids/>

Images particulières

- Une "top level image" est une image sans enfant
- Une image taguée "none" est une image intermédiaire, sinon c'est une image pendante (dangling)
- Une image taguée "missing" est une image manquante mais dont on a obtenu le layer associé (via *docker pull*)

Avant la version 1.10

- Chaque image était identifié par un nombre aléatoire
- Chaque image était associé à **1 seul** couche.
- ⇒ Chaque couche était repéré par le même identifiant que l'image
- ⇒ Problèmes de sécurité à cause du nombre aléatoire
- Pour plus d'informations :
<http://windsock.io/explaining-docker-image-ids/>

Images particulières

- Une "top level image" est une image sans enfant
- Une image taguée "none" est une image intermédiaire, sinon c'est une image pendante (dangling)
- Une image taguée "missing" est une image manquante mais dont on a obtenu le layer associé (via *docker pull*)

Avant la version 1.10

- Chaque image était identifié par un nombre aléatoire
- Chaque image était associé à **1 seul** couche.
- ⇒ Chaque couche était repéré par le même identifiant que l'image
- ⇒ Problèmes de sécurité à cause du nombre aléatoire
- Pour plus d'informations :
<http://windsock.io/explaining-docker-image-ids/>

Images particulières

- Une "top level image" est une image sans enfant
- Une image taguée "none" est une image intermédiaire, sinon c'est une image pendante (dangling)
- Une image taguée "missing" est une image manquante mais dont on a obtenu le layer associé (via *docker pull*)

Avant la version 1.10

- Chaque image était identifié par un nombre aléatoire
- Chaque image était associé à **1 seul** couche.
- ⇒ Chaque couche était repéré par le même identifiant que l'image
- ⇒ Problèmes de sécurité à cause du nombre aléatoire
- Pour plus d'informations :
<http://windsock.io/explaining-docker-image-ids/>

Images particulières

- Une "top level image" est une image sans enfant
- Une image taguée "none" est une image intermédiaire, sinon c'est une image pendante (dangling)
- Une image taguée "missing" est une image manquante mais dont on a obtenu le layer associé (via *docker pull*)

Avant la version 1.10

- Chaque image était identifié par un nombre aléatoire
- Chaque image était associé à **1 seul** couche.
- ⇒ Chaque couche était repéré par le même identifiant que l'image
- ⇒ Problèmes de sécurité à cause du nombre aléatoire
- Pour plus d'informations :
<http://windsock.io/explaining-docker-image-ids/>

Quelques commandes de bases

Gestion des images

- *docker pull* : télécharger une image présente sur un registre (docker hub par défaut)
- *docker push* : pousser (upload) une image sur un registre
- *docker search* : chercher une image sur un registre
- *docker images* : liste les images présentent localement
- *docker history* : affiche l'historique de création d'une image
- *docker rmi* : supprimer une image
- *docker commit* : construire une image à partir d'un conteneur
- *docker build* : construire une image à partir d'un dockerfile

Quelques commandes de bases

Gestion des conteneurs

- *docker run* : démarrer un conteneur
- *docker stop/start/restart/pause/unpause* :
arrête/démarre/redémarre/pause/reprise d'un conteneur
- *docker logs* : obtenir les journaux système d'un conteneur
- *docker kill* : tuer un conteneur
- *docker rm* : supprimer un conteneur
- *docker ps* : liste les conteneurs actifs

Quelques commandes de bases

Plusieurs commandes pour le même résultat

- Les commandes vues précédemment sont les commandes historiques de docker
- Avec le temps, docker a regroupé ses commandes en fonction des "objets" concernés
- Par exemple, toutes les commandes concernant :
 - les conteneurs sont regroupés dans *docker container cmd*
 - les images sont regroupées dans *docker image cmd*
 - et ainsi de suite pour : config, container, image, network, node, plugin, secret, service, stack, swarm, system et volume
- Mais *docker container ls -aq* **est identique** à *docker ps -aq*
- Etc, etc

Quelques commandes de bases

Gestion de docker

- *docker info* : affiche des informations relatives à l'écosystème docker sur la machine
- *docker inspect* : affiche des informations de bas niveau sur des objets docker (images, conteneurs, réseaux, ...)
- *docker version* : affiche la version de docker

"Philosophie" de docker : Bonnes pratiques recommandés

- Un conteneur doit être éphémère (séparation et point de montage des données dans le conteneur)
- Un conteneur = 1 (micro) service (= 1 processus pour les puristes)
- On ne devrait pas être root dans un conteneur (sshd, crond, firewall, ... ⇒ Hôte)
- Si jamais on doit être root, utilisation d'un setuid, gosu ou su-exec
- Minimiser la taille des images (pas de superflu)
- Minimiser le nombre de couche dans une image (Dockerfile ⇒ multiligne argument, pipe)
- Maximiser le partage des couches
- Les couches les plus volumineuses au plus près de la racine (utilisation du cache de *docker build*)

"Philosophie" de docker : Bonnes pratiques recommandés

- Un conteneur doit être éphémère (séparation et point de montage des données dans le conteneur)
- Un conteneur = 1 (micro) service (= 1 processus pour les puristes)
- On ne devrait pas être root dans un conteneur (sshd, crond, firewall, ... ⇒ Hôte)
- Si jamais on doit être root, utilisation d'un setuid, gosu ou su-exec
- Minimiser la taille des images (pas de superflu)
- Minimiser le nombre de couche dans une image (Dockerfile ⇒ multiligne argument, pipe)
- Maximiser le partage des couches
- Les couches les plus volumineuses au plus près de la racine (utilisation du cache de *docker build*)

"Philosophie" de docker : Bonnes pratiques recommandés

- Un conteneur doit être éphémère (séparation et point de montage des données dans le conteneur)
- Un conteneur = 1 (micro) service (= 1 processus pour les puristes)
- On ne devrait pas être root dans un conteneur (sshd, crond, firewall, ... ⇒ Hôte)
- Si jamais on doit être root, utilisation d'un setuid, gosu ou su-exec
- Minimiser la taille des images (pas de superflu)
- Minimiser le nombre de couche dans une image (Dockerfile ⇒ multiligne argument, pipe)
- Maximiser le partage des couches
- Les couches les plus volumineuses au plus près de la racine (utilisation du cache de *docker build*)

"Philosophie" de docker : Bonnes pratiques recommandés

- Un conteneur doit être éphémère (séparation et point de montage des données dans le conteneur)
- Un conteneur = 1 (micro) service (= 1 processus pour les puristes)
- On ne devrait pas être root dans un conteneur (sshd, crond, firewall, ... ⇒ Hôte)
- Si jamais on doit être root, utilisation d'un setuid, gosu ou su-exec
- Minimiser la taille des images (pas de superflu)
- Minimiser le nombre de couche dans une image (Dockerfile ⇒ multiligne argument, pipe)
- Maximiser le partage des couches
- Les couches les plus volumineuses au plus près de la racine (utilisation du cache de *docker build*)

"Philosophie" de docker : Bonnes pratiques recommandés

- Un conteneur doit être éphémère (séparation et point de montage des données dans le conteneur)
- Un conteneur = 1 (micro) service (= 1 processus pour les puristes)
- On ne devrait pas être root dans un conteneur (sshd, crond, firewall, ... ⇒ Hôte)
- Si jamais on doit être root, utilisation d'un setuid, gosu ou su-exec
- Minimiser la taille des images (pas de superflu)
- Minimiser le nombre de couche dans une image (Dockerfile ⇒ multiligne argument, pipe)
- Maximiser le partage des couches
- Les couches les plus volumineuses au plus près de la racine (utilisation du cache de *docker build*)

"Philosophie" de docker : Bonnes pratiques recommandés

- Un conteneur doit être éphémère (séparation et point de montage des données dans le conteneur)
- Un conteneur = 1 (micro) service (= 1 processus pour les puristes)
- On ne devrait pas être root dans un conteneur (sshd, crond, firewall, ... ⇒ Hôte)
- Si jamais on doit être root, utilisation d'un setuid, gosu ou su-exec
- Minimiser la taille des images (pas de superflu)
- Minimiser le nombre de couche dans une image (Dockerfile ⇒ multiligne argument, pipe)
- Maximiser le partage des couches
- Les couches les plus volumineuses au plus près de la racine (utilisation du cache de *docker build*)

"Philosophie" de docker : Bonnes pratiques recommandés

- Un conteneur doit être éphémère (séparation et point de montage des données dans le conteneur)
- Un conteneur = 1 (micro) service (= 1 processus pour les puristes)
- On ne devrait pas être root dans un conteneur (sshd, crond, firewall, ... ⇒ Hôte)
- Si jamais on doit être root, utilisation d'un setuid, gosu ou su-exec
- Minimiser la taille des images (pas de superflu)
- Minimiser le nombre de couche dans une image (Dockerfile ⇒ multiligne argument, pipe)
- Maximiser le partage des couches
- Les couches les plus volumineuses au plus près de la racine (utilisation du cache de *docker build*)

"Philosophie" de docker : Bonnes pratiques recommandés

- Un conteneur doit être éphémère (séparation et point de montage des données dans le conteneur)
- Un conteneur = 1 (micro) service (= 1 processus pour les puristes)
- On ne devrait pas être root dans un conteneur (sshd, crond, firewall, ... ⇒ Hôte)
- Si jamais on doit être root, utilisation d'un setuid, gosu ou su-exec
- Minimiser la taille des images (pas de superflu)
- Minimiser le nombre de couche dans une image (Dockerfile ⇒ multiligne argument, pipe)
- Maximiser le partage des couches
- Les couches les plus volumineuses au plus près de la racine (utilisation du cache de *docker build*)

"Philosophies" de docker ?

- Adapter les bonnes pratiques en fonction de ses besoins et de ses usages
- Pour plus d'un processus par conteneur, voir du côté de supervisor
- Utilisez-vous la bonne technologie de conteneurisation ?

"Philosophies" de docker ?

- Adapter les bonnes pratiques en fonction de ses besoins et de ses usages
- Pour plus d'un processus par conteneur, voir du côté de supervisor
- Utilisez-vous la bonne technologie de conteneurisation ?

"Philosophies" de docker ?

- Adapter les bonnes pratiques en fonction de ses besoins et de ses usages
- Pour plus d'un processus par conteneur, voir du côté de supervisor
- Utilisez-vous la bonne technologie de conteneurisation ?

Plan

- 1 Introduction générale
 - Conteneur versus machine virtuelle
 - Des technologies de conteneurisations différentes pour des objectifs différents
- 2 Bases
 - Introduction
 - Définitions et prérequis
 - Premiers pas
 - Volatilité, persistance et sécurité des données
 - Sécurité des conteneurs
 - Conteneurs graphiques
- 3 Utilisation avancées
 - Créations d'images : méthode naïve
 - Créations d'images : Dockerfile
- 4 Toujours plus loin
 - Travaux pratiques

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker run hello-world
```

```
Unable to find image 'hello-world :latest' locally
```

```
latest : Pulling from library/hello-world
```

```
78445dd45222 : Downloading [=====> ]
```

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker run hello-world
Unable to find image 'hello-world :latest' locally
latest : Pulling from library/hello-world
78445dd45222 : Extracting [=====>]
```

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker run hello-world
Unable to find image 'hello-world :latest' locally
latest : Pulling from library/hello-world
78445dd45222 : Pull complete
Digest :
sha256 :c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status : Downloaded newer image for hello-world :latest
```

...

Notre premier conteneur

Hello world !

Terminal:

Conteneur:

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

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://cloud.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/engine/userguide/>

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker ps
jmc@laptop/home/jmc $ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ccdd8f78bf58	hello-world	"/hello"	15 minutes ago	Exited (0) 15 minutes ago		furious_pare

```
jmc@laptop/home/jmc $ docker ps -aq
ccdd8f78bf58
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{.ID}}' furious_pare
ccdd8f78bf58e744968462d979bde759155dcd6892cad94bea2bf64fe3fa0012
```

- 1 Aucun conteneur actif
- 2 1 conteneur inactif (-a = all), si nom non spécifié, attribution d'un nom aléatoire
- 3 (-q = quiet)
- 4 CONTAINER ID affecté aléatoirement

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker ps
jmc@laptop/home/jmc $ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ccdd8f78bf58	hello-world	"/hello"	15 minutes ago	Exited (0) 15 minutes ago		furious_pare

```
jmc@laptop/home/jmc $ docker ps -aq
ccdd8f78bf58
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{.ID}}' furious_pare
ccdd8f78bf58e744968462d979bde759155dcd6892cad94bea2bf64fe3fa0012
```

1. Aucun conteneur actif
2. 1 conteneur inactif (-a = all), si nom non spécifié, attribution d'un nom aléatoire
3. (-q = quiet)
4. CONTAINER ID affecté aléatoirement

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker ps
jmc@laptop/home/jmc $ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ccdd8f78bf58	hello-world	"/hello"	15 minutes ago	Exited (0) 15 minutes ago		furious_pare

```
jmc@laptop/home/jmc $ docker ps -aq
ccdd8f78bf58
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{ .ID }}' furious_pare
ccdd8f78bf58e744968462d979bde759155dcd6892cad94bea2bf64fe3fa0012
```

- 1 Aucun conteneur actif
- 2 1 conteneur inactif (-a = all), si nom non spécifié, attribution d'un nom aléatoire
- 3 (-q = quiet)
- 4 CONTAINER ID affecté aléatoirement

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker ps
jmc@laptop/home/jmc $ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ccdd8f78bf58	hello-world	"/hello"	15 minutes ago	Exited (0) 15 minutes ago		furious_pare

```
jmc@laptop/home/jmc $ docker ps -aq
ccdd8f78bf58
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{.ID}}' furious_pare
ccdd8f78bf58e744968462d979bde759155dcd6892cad94bea2bf64fe3fa0012
```

- 1 Aucun conteneur actif
- 2 1 conteneur inactif (-a = all), si nom non spécifié, attribution d'un nom aléatoire
- 3 (-q = quiet)
- 4 CONTAINER ID affecté aléatoirement

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	48b5124b2768	4 months ago	1.84 kB

```
jmc@laptop/home/jmc $ docker images -aq  
48b5124b2768
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{.ID}}' hello-world  
sha256 :48b5124b2768d2b917edcb640435044a97967015485e812545546cbcd5cf0233  
jmc@laptop/home/jmc $ docker inspect --format='{{.RootFS.Layers}}' hello-world  
[sha256 :98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08]
```

- 1 Liste les images de plus haut niveau (top level image)
- 2 Liste de façon silencieuse (-q) toutes les images (intermédiaires également)
- 3 IMAGE ID affecté par fonction de hachage
- 4 Fonction de hachage du layer décompressé (N.B. : présence des [])

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	48b5124b2768	4 months ago	1.84 kB

```
jmc@laptop/home/jmc $ docker images -aq  
48b5124b2768
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{.ID}}' hello-world  
sha256 :48b5124b2768d2b917edcb640435044a97967015485e812545546cbcd5cf0233  
jmc@laptop/home/jmc $ docker inspect --format='{{.RootFS.Layers}}' hello-world  
[sha256 :98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08]
```

- 1 Liste les images de plus haut niveau (top level image)
- 2 Liste de façon silencieuse (-q) toutes les images (intermédiaires également)
- 3 IMAGE ID affecté par fonction de hachage
- 4 Fonction de hachage du layer décompressé (N.B. : présence des [])

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	48b5124b2768	4 months ago	1.84 kB

```
jmc@laptop/home/jmc $ docker images -aq  
48b5124b2768
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{.ID}}' hello-world  
sha256 :48b5124b2768d2b917edcb640435044a97967015485e812545546cbcd5cf0233
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{.RootFS.Layers}}' hello-world  
[sha256 :98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08]
```

- 1 Liste les images de plus haut niveau (top level image)
- 2 Liste de façon silencieuse (-q) toutes les images (intermédiaires également)
- 3 IMAGE ID affecté par fonction de hachage
- 4 Fonction de hachage du layer décompressé (N.B. : présence des [])

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	48b5124b2768	4 months ago	1.84 kB

```
jmc@laptop/home/jmc $ docker images -aq  
48b5124b2768
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{.ID}}' hello-world  
sha256 :48b5124b2768d2b917edcb640435044a97967015485e812545546cbcd5cf0233
```

```
jmc@laptop/home/jmc $ docker inspect --format='{{.RootFS.Layers}}' hello-world  
[sha256 :98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08]
```

- 1 Liste les images de plus haut niveau (top level image)
- 2 Liste de façon silencieuse (-q) toutes les images (intermédiaires également)
- 3 IMAGE ID affecté par fonction de hachage
- 4 Fonction de hachage du layer décompressé (N.B. : présence des [])

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker rm furious_pare # docker rm cdd8 (debut CONTAINER ID)
furious_pare
```

- Suppression du conteneur par son nom ou son identifiant

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker rmi hello-world # docker rmi 48b (debut IMAGE ID)
Untagged : hello-world :latest
Untagged : hello-world@sha256 :c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Deleted : sha256 :48b5124b2768d2b917edcb640435044a97967015485e812545546cbcd5cf0233
Deleted : sha256 :98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08
```

- latest considéré comme tag par défaut (untagged by user)
- hello-world@sha256 :c55... considéré comme tag par défaut (untagged by user)(Hash Image Compressed)
- Deleted : sha256 :48b... ⇒ Suppression de l'image
- Deleted : sha256 :98c... ⇒ Suppression des couches de l'image (1 seul en l'occurrence) car couches non partagés par d'autres images.
- A quoi correspond 78445dd45222 lors du pull ? ⇒ Hash Layer Compressed

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker rmi hello-world # docker rmi 48b (debut IMAGE ID)
Untagged : hello-world :latest
Untagged : hello-world@sha256 :c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Deleted : sha256 :48b5124b2768d2b917edcb640435044a97967015485e812545546cbcd5cf0233
Deleted : sha256 :98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08
```

- latest considéré comme tag par défaut (untagged by user)
- hello-world@sha256 :c55... considéré comme tag par défaut (untagged by user)(Hash Image Compressed)
- Deleted : sha256 :48b... ⇒ Suppression de l'image
- Deleted : sha256 :98c... ⇒ Suppression des couches de l'image (1 seul en l'occurrence) car couches non partagés par d'autres images.
- A quoi correspond 78445dd45222 lors du pull ? ⇒ Hash Layer Compressed

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker rmi hello-world # docker rmi 48b (debut IMAGE ID)
Untagged : hello-world :latest
Untagged : hello-world@sha256 :c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Deleted : sha256 :48b5124b2768d2b917edcb640435044a97967015485e812545546cbcd5cf0233
Deleted : sha256 :98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08
```

- latest considéré comme tag par défaut (untagged by user)
- hello-world@sha256 :c55... considéré comme tag par défaut (untagged by user)(Hash Image Compressed)
- Deleted : sha256 :48b... ⇒ Suppression de l'image
- Deleted : sha256 :98c... ⇒ Suppression des couches de l'image (1 seul en l'occurrence) car couches non partagés par d'autres images.
- A quoi correspond 78445dd45222 lors du pull ? ⇒ Hash Layer Compressed

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker rmi hello-world # docker rmi 48b (debut IMAGE ID)
Untagged : hello-world :latest
Untagged : hello-world@sha256 :c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Deleted : sha256 :48b5124b2768d2b917edcb640435044a97967015485e812545546cbcd5cf0233
Deleted : sha256 :98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08
```

- latest considéré comme tag par défaut (untagged by user)
- hello-world@sha256 :c55... considéré comme tag par défaut (untagged by user)(Hash Image Compressed)
- Deleted : sha256 :48b... ⇒ Suppression de l'image
- Deleted : sha256 :98c... ⇒ Suppression des couches de l'image (1 seul en l'occurrence) car couches non partagés par d'autres images.
- A quoi correspond 78445dd45222 lors du pull ? ⇒ Hash Layer Compressed

Notre premier conteneur

Hello world !

Terminal:

```
jmc@laptop/home/jmc $ docker rmi hello-world # docker rmi 48b (debut IMAGE ID)
Untagged : hello-world :latest
Untagged : hello-world@sha256 :c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Deleted : sha256 :48b5124b2768d2b917edcb640435044a97967015485e812545546cbcd5cf0233
Deleted : sha256 :98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08
```

- latest considéré comme tag par défaut (untagged by user)
- hello-world@sha256 :c55... considéré comme tag par défaut (untagged by user)(Hash Image Compressed)
- Deleted : sha256 :48b... ⇒ Suppression de l'image
- Deleted : sha256 :98c... ⇒ Suppression des couches de l'image (1 seul en l'occurrence) car couches non partagés par d'autres images.
- A quoi correspond 78445dd45222 lors du pull ? ⇒ Hash Layer Compressed

Quelques cas faciles

Quelques conteneurs à exécuter

Prenez le temps de lancer quelques conteneurs simplement, par exemple :

- ➊ Exécutez `"/bin/bash"` dans un conteneur interactif ubuntu
- ➋ Exécutez `"/bin/ls"` dans un conteneur non interactif ubuntu
- ➌ Exécutez `"/bin/ps -eflH"` dans un conteneur non interactif ubuntu
- ➍ Exécutez `"/bin/bash"` dans un conteneur interactif nommé `"mon_conteneur_c0"` ubuntu
- ➎ Détachez-vous du conteneur `"mon_conteneur"` avec `Ctrl+p Ctrl+q`
- ➏ Rattachez-vous au conteneur `"mon_conteneur"`
- ➐ Sortez du conteneur `"mon_conteneur"`
- ➑ Listez les conteneurs actifs, listez tous les conteneurs
- ➒ Supprimez tous les conteneurs inactifs en 1 commande (*`docker container help`*)

Quelques cas faciles : commentaires

Quelques conteneurs à exécuter

Prenez le temps de lancer quelques conteneurs simplement, par exemple :

- ❶ `docker run -it ubuntu /bin/bash`
- ❷ `docker run ubuntu /bin/ls`
- ❸ `docker run ubuntu /bin/ps -eflH`
- ❹ `docker run -it --name mon_conteneur_c0 ubuntu`
- ❺ `Ctrl+p Ctrl+q`
- ❻ `docker container attach mon_conteneur`
- ❼ `exit`
- ❽ `docker container ls; docker container ls -a`
- ❾ `docker container prune`

Quelques cas pénibles

Quelques conteneurs à exécuter

Vous utiliserez les commandes *docker run* et *docker container*.

docker cmd help pour l'aide sur la commande *cmd*

- ❶ Essayez d'exécuter `"/bin/bash"` dans un conteneur interactif alpine
- ❷ Exécutez `"echo hello world!"` dans un conteneur non interactif centos `"c1"`
- ❸ Essayez de démarrer 1 fois le conteneur `"c1"`. Faites *docker logs c1*
- ❹ Exécutez `"/bin/bash"` dans un conteneur non interactif centos `"c2"`
- ❺ Essayez de démarrer 1 fois le conteneur `"c2"`. Faites *docker logs c2*
- ❻ Exécutez `"/bin/bash"` dans un conteneur interactif centos `"c3"` et *exit* de `c3`
- ❼ Démarrez, connectez vous avec un nouveau bash et détachez vous du conteneur `"c3"`
- ❽ Listez les processus du conteneur `"c3"` depuis l'hôte **et** depuis le conteneur (*ps*)

Quelques cas pénibles : commentaires

Vous utiliserez les commandes *docker run* et *docker container*.
docker cmd help pour l'aide sur la commande *cmd*

- ❶ `/bin/bash` n'est pas présent par défaut dans alpine (Idée sous-jacente :
 - Minimiser la taille des images, on installe que ce que l'on a besoin
 - La distribution Alpine fait 5 Mo !
- ❷ *docker run centos echo hello world !*
- ❸ `c1` est en mode non interactif et `"echo"` est sans boucle d'événement, mais `"hello"` effectué réellement 2 fois.
- ❹ `c2` est en mode non interactif et il n'y a pas de pseudo-terminal alloué
- ❺ La commande `"/bin/bash"` est effectué réellement 2 fois
- ❻ *docker run -it --name c3 centos /bin/bash ; exit ;*
- ❼ *docker start c3 ; docker exec -it c3 /bin/bash ; Ctrl+p Ctrl+q*
- ❽ *docker container top c3 ; docker exec c3 ps ;*

Plan

- 1 Introduction générale
 - Conteneur versus machine virtuelle
 - Des technologies de conteneurisations différentes pour des objectifs différents
- 2 Bases
 - Introduction
 - Définitions et prérequis
 - Premiers pas
 - Volatilité, persistance et sécurité des données
 - Sécurité des conteneurs
 - Conteneurs graphiques
- 3 Utilisation avancées
 - Créations d'images : méthode naïve
 - Créations d'images : Dockerfile
- 4 Toujours plus loin
 - Travaux pratiques

Volatilité, persistance et sécurité des données

Something more ambitious

Terminal:

```
jmc@laptop/home/jmc $ docker run -it ubuntu /bin/bash
```

```
Unable to find image 'ubuntu :latest' locally
```

```
latest : Pulling from library/ubuntu
```

```
bd97b43c27e3 : Pull complete
```

```
6960dc1aba18 : Pull complete
```

```
2b61829b0db5 : Pull complete
```

```
1f88dc826b14 : Pull complete
```

```
73b3859b1e43 : Pull complete
```

```
Digest :
```

```
sha256 :ea1d854d38be82f54d39efe2c67000bed1b03348bcc2f3dc094f260855dff368
```

```
Status : Downloaded newer image for ubuntu :latest
```

Volatilité, persistance et sécurité des données

Something more ambitious

Terminal:

Conteneur:

```
root@dfdf734253a9 :/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr
var
root@dfdf734253a9 :/# rm -rf / # Aucune incidence sur l'hôte car pas de
montage de volume
rm : it is dangerous to operate recursively on '/'
rm : use --no-preserve-root to override this failsafe
```

Volatilité, persistance et sécurité des données

Something more ambitious

Terminal:

Conteneur:

```
root@dfdf734253a9 :/# rm -rf --no-preserve-root /  
...  
rm : cannot remove '/proc/fs/aufs/plink_maint' : Read-only file system  
...  
root@dfdf734253a9 :/# ls  
bash : /bin/ls : No such file or directory  
root@dfdf734253a9 :/# exit
```

jmc@laptop/home/jmc \$

Volatilité, persistance et sécurité des données

Something more ambitious

Terminal:

```
jmc@laptop/home/jmc $ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
dfdf734253a9	ubuntu	"/bin/bash"	26 minutes ago	Exited (127) 6 minutes ago		stoic_engelbart

```
jmc@laptop/home/jmc $ docker rm dfd  
dfd
```

```
jmc@laptop/home/jmc $
```

Volatilité, persistance et sécurité des données

Something more ambitious

Terminal:

```
jmc@laptop/home/jmc $ docker run -it ubuntu /bin/bash
```

Conteneur:

```
root@63f0abcc808a :/# #Ceci est un nouveau conteneur
```

```
root@63f0abcc808a :/# ls
```

```
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr  
var
```

```
root@63f0abcc808a :/# exit
```

```
jmc@laptop/home/jmc $
```

Volatilité, persistance et sécurité des données

Something more ambitious

Terminal:

```
jmc@laptop/home/jmc $ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
63f0abcc808a	ubuntu	"/bin/bash"	32 minutes ago	Exited (0) 27 seconds ago		suspicious_goldstine

```
jmc@laptop/home/jmc $ docker rm suspicious_goldstine
suspicious_goldstine
jmc@laptop/home/jmc $
```

Volatilité, persistance et sécurité des données

Something more ambitious

Terminal:

```
jmc@laptop/home/jmc $ docker run -it -v / :/danger_racine_hote --hostname myhostname --name myname ubuntu /bin/bash
```

Conteneur:

```
root@myhostname :/# ls
bin boot dev etc home lib lib64 media mnt opt proc danger_racine_hote root run
sbin srv sys tmp usr var
root@myhostname :/# ls /danger_racine_hote
bin boot cdrom dev etc home initrd.img lib lib64 lost+found media mnt opt proc
root run sbin srv sys tmp usr var vmlinuz
root@myhostname :/# # Ne pas faire rm -rf /danger_racine_hote :-0
```


Volatilité, persistance et sécurité des données

Something more ambitious

Terminal:

```
jmc@laptop/home/jmc $ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
5c2eacbfcee	ubuntu	"/bin/bash"	8 minutes ago	Exited (0) 28 seconds ago		myname

```
jmc@laptop/home/jmc $ docker rm myname  
myname  
jmc@laptop/home/jmc $
```

Volatilité, persistance et sécurité des données

Something more ambitious

Terminal:

```
jmc@laptop/home/jmc $ docker run -it -rm ubuntu /bin/bash
```

Conteneur:

```
root@30481297b94c :/# exit
```

```
jmc@laptop/home/jmc $ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

```
jmc@laptop/home/jmc $
```

Volatilité, persistance et sécurité des données

Résumé

- Il peut être dangereux d'exécuter des conteneurs contenant des processus root
- On peut monter des répertoires de l'hôte dans le conteneur afin d'assurer la persistance de certaines données
- Cette technologie de persistance s'appelle data volume
- Pour des montages sur des ressources partagés (NFS, ISCSI, FC) : voir docker volume plugin
- Il existe une technologie de persistance à travers des conteneurs : data volume container

Volatilité, persistance et sécurité des données

Résumé

- Il peut être dangereux d'exécuter des conteneurs contenant des processus root
- On peut monter des répertoires de l'hôte dans le conteneur afin d'assurer la persistance de certaines données
- Cette technologie de persistance s'appelle data volume
- Pour des montages sur des ressources partagés (NFS, ISCSI, FC) : voir docker volume plugin
- Il existe une technologie de persistance à travers des conteneurs : data volume container

Volatilité, persistance et sécurité des données

Résumé

- Il peut être dangereux d'exécuter des conteneurs contenant des processus root
- On peut monter des répertoires de l'hôte dans le conteneur afin d'assurer la persistance de certaines données
- Cette technologie de persistance s'appelle data volume
- Pour des montages sur des ressources partagés (NFS, ISCSI, FC) : voir docker volume plugin
- Il existe une technologie de persistance à travers des conteneurs : data volume container

Volatilité, persistance et sécurité des données

Résumé

- Il peut être dangereux d'exécuter des conteneurs contenant des processus root
- On peut monter des répertoires de l'hôte dans le conteneur afin d'assurer la persistance de certaines données
- Cette technologie de persistance s'appelle data volume
- Pour des montages sur des ressources partagés (NFS, ISCSI, FC) : voir docker volume plugin
- Il existe une technologie de persistance à travers des conteneurs : data volume container

Volatilité, persistance et sécurité des données

Résumé

- Il peut être dangereux d'exécuter des conteneurs contenant des processus root
- On peut monter des répertoires de l'hôte dans le conteneur afin d'assurer la persistance de certaines données
- Cette technologie de persistance s'appelle data volume
- Pour des montages sur des ressources partagés (NFS, ISCSI, FC) : voir docker volume plugin
- Il existe une technologie de persistance à travers des conteneurs : data volume container

Plan

- 1 Introduction générale
 - Conteneur versus machine virtuelle
 - Des technologies de conteneurisations différentes pour des objectifs différents
- 2 Bases
 - Introduction
 - Définitions et prérequis
 - Premiers pas
 - Volatilité, persistance et sécurité des données
 - Sécurité des conteneurs
 - Conteneurs graphiques
- 3 Utilisation avancées
 - Créations d'images : méthode naïve
 - Créations d'images : Dockerfile
- 4 Toujours plus loin
 - Travaux pratiques

Sécurité des conteneurs

Namespace : isolation de base

- Les processus exécutés dans un conteneur ne peuvent ni voir ni affecter les processus exécutés dans un autre conteneur ou dans le système hôte.
- Chaque conteneur possède son propre réseau. Si l'hôte est configuré en conséquence, les conteneurs peuvent interagir les uns avec les autres (et avec l'hôte) au travers de leurs interfaces de réseau respectives
- Tous les conteneurs d'un hôte Docker sont connectés à des bridges (virtual switch).
- Il est tout de même déconseillé d'utiliser des applications root (ssh, vpn, dhcp, ...). L'hôte étant supposé subvenir à ces besoins.
- Si on souhaite modifier le comportement par défaut de l'isolation, cela implique une intervention et une connaissance des capacités (c'est un prix à payer lorsque l'on partage le même noyau)
- Plus d'info sur <https://docs.docker.com/engine/security/security/>

Sécurité des conteneurs

Namespace : isolation de base

- Les processus exécutés dans un conteneur ne peuvent ni voir ni affecter les processus exécutés dans un autre conteneur ou dans le système hôte.
- Chaque conteneur possède son propre réseau. Si l'hôte est configuré en conséquence, les conteneurs peuvent interagir les uns avec les autres (et avec l'hôte) au travers de leurs interfaces de réseau respectives
- Tous les conteneurs d'un hôte Docker sont connectés à des bridges (virtual switch).
- Il est tout de même déconseillé d'utiliser des applications root (ssh, vpn, dhcp, ...). L'hôte étant supposé subvenir à ces besoins.
- Si on souhaite modifier le comportement par défaut de l'isolation, cela implique une intervention et une connaissance des capacités (c'est un prix à payer lorsque l'on partage le même noyau)
- Plus d'info sur <https://docs.docker.com/engine/security/security/>

Sécurité des conteneurs

Namespace : isolation de base

- Les processus exécutés dans un conteneur ne peuvent ni voir ni affecter les processus exécutés dans un autre conteneur ou dans le système hôte.
- Chaque conteneur possède son propre réseau. Si l'hôte est configuré en conséquence, les conteneurs peuvent interagir les uns avec les autres (et avec l'hôte) au travers de leurs interfaces de réseau respectives
- Tous les conteneurs d'un hôte Docker sont connectés à des bridges (virtual switch).
- Il est tout de même déconseillé d'utiliser des applications root (ssh, vpn, dhcp, ...). L'hôte étant supposé subvenir à ces besoins.
- Si on souhaite modifier le comportement par défaut de l'isolation, cela implique une intervention et une connaissance des capacités (c'est un prix à payer lorsque l'on partage le même noyau)
- Plus d'info sur <https://docs.docker.com/engine/security/security/>

Sécurité des conteneurs

Namespace : isolation de base

- Les processus exécutés dans un conteneur ne peuvent ni voir ni affecter les processus exécutés dans un autre conteneur ou dans le système hôte.
- Chaque conteneur possède son propre réseau. Si l'hôte est configuré en conséquence, les conteneurs peuvent interagir les uns avec les autres (et avec l'hôte) au travers de leurs interfaces de réseau respectives
- Tous les conteneurs d'un hôte Docker sont connectés à des bridges (virtual switch).
- Il est tout de même déconseillé d'utiliser des applications root (ssh, vpn, dhcp, ...). L'hôte étant supposé subvenir à ces besoins.
- Si on souhaite modifier le comportement par défaut de l'isolation, cela implique une intervention et une connaissance des capacités (c'est un prix à payer lorsque l'on partage le même noyau)
- Plus d'info sur <https://docs.docker.com/engine/security/security/>

Sécurité des conteneurs

Namespace : isolation de base

- Les processus exécutés dans un conteneur ne peuvent ni voir ni affecter les processus exécutés dans un autre conteneur ou dans le système hôte.
- Chaque conteneur possède son propre réseau. Si l'hôte est configuré en conséquence, les conteneurs peuvent interagir les uns avec les autres (et avec l'hôte) au travers de leurs interfaces de réseau respectives
- Tous les conteneurs d'un hôte Docker sont connectés à des bridges (virtual switch).
- Il est tout de même déconseillé d'utiliser des applications root (ssh, vpn, dhcp, ...). L'hôte étant supposé subvenir à ces besoins.
- Si on souhaite modifier le comportement par défaut de l'isolation, cela implique une intervention et une connaissance des capacités (c'est un prix à payer lorsque l'on partage le même noyau)
- Plus d'info sur <https://docs.docker.com/engine/security/security/>

Sécurité des conteneurs

Namespace : isolation de base

- Les processus exécutés dans un conteneur ne peuvent ni voir ni affecter les processus exécutés dans un autre conteneur ou dans le système hôte.
- Chaque conteneur possède son propre réseau. Si l'hôte est configuré en conséquence, les conteneurs peuvent interagir les uns avec les autres (et avec l'hôte) au travers de leurs interfaces de réseau respectives
- Tous les conteneurs d'un hôte Docker sont connectés à des bridges (virtual switch).
- Il est tout de même déconseillé d'utiliser des applications root (ssh, vpn, dhcp, ...). L'hôte étant supposé subvenir à ces besoins.
- Si on souhaite modifier le comportement par défaut de l'isolation, cela implique une intervention et une connaissance des capacités (c'est un prix à payer lorsque l'on partage le même noyau)
- Plus d'info sur <https://docs.docker.com/engine/security/security/>

Sécurité des conteneurs

Capabilities

- Il peut être dangereux d'exécuter des conteneurs contenant des processus root
- En cas d'escalade de privilège depuis le conteneur sur l'hôte : par défaut le root dans le conteneur n'a pas les mêmes droits que le potentiel root sur l'hôte.
- En effet celui-ci ne dispose pas de toutes les capabilities.
- Plus d'info sur <https://linux.die.net/man/7/capabilities>

Sécurité des conteneurs

Capabilities

- Il peut être dangereux d'exécuter des conteneurs contenant des processus root
- En cas d'escalade de privilège depuis le conteneur sur l'hôte : par défaut le root dans le conteneur n'a pas les mêmes droits que le potentiel root sur l'hôte.
- En effet celui-ci ne dispose pas de toutes les capabilities.
- Plus d'info sur <https://linux.die.net/man/7/capabilities>

Sécurité des conteneurs

Capabilities

- Il peut être dangereux d'exécuter des conteneurs contenant des processus root
- En cas d'escalade de privilège depuis le conteneur sur l'hôte : par défaut le root dans le conteneur n'a pas les mêmes droits que le potentiel root sur l'hôte.
- En effet celui-ci ne dispose pas de toutes les capabilities.
- Plus d'info sur <https://linux.die.net/man/7/capabilities>

Sécurité des conteneurs

Capabilities

- Il peut être dangereux d'exécuter des conteneurs contenant des processus root
- En cas d'escalade de privilège depuis le conteneur sur l'hôte : par défaut le root dans le conteneur n'a pas les mêmes droits que le potentiel root sur l'hôte.
- En effet celui-ci ne dispose pas de toutes les capabilities.
- Plus d'info sur <https://linux.die.net/man/7/capabilities>

Sécurité des conteneurs

Capabilités par défaut

- CAP_CHOWN
- CAP_DAC_OVERRIDE
- CAP_FSETID
- CAP_FOWNER
- CAP_MKNOD
- CAP_NET_RAW
- CAP_SETGID
- CAP_SETUID
- CAP_SETFCAP
- CAP_SETPCAP
- CAP_NET_BIND_SERVICE
- CAP_SYS_CHROOT
- CAP_KILL
- CAP_AUDIT_WRITE

Sécurité des conteneurs

CGroups : limitations des ressources

- Comptabilité et limitations des ressources par conteneurs
 - Par défaut : part équitable de ressources (CPU, RAM, I/O) par conteneur
 - Empêche l'utilisation de toutes les ressources par un conteneur (démon fork bomb `-cpuset-cpus 0`)
 - D'une certaine manière, ils empêchent certaines attaques de type DOS
 - "Garantit" une disponibilité et une performance quasiment constante

Sécurité des conteneurs

CGroups : limitations des ressources

- Comptabilité et limitations des ressources par conteneurs
- Par défaut : part équitable de ressources (CPU, RAM, I/O) par conteneur
- Empêche l'utilisation de toutes les ressources par un conteneur (démon fork bomb `-cpuset-cpus 0`)
- D'une certaine manière, ils empêchent certaines attaques de type DOS
- "Garantit" une disponibilité et une performance quasiment constante

Sécurité des conteneurs

CGroups : limitations des ressources

- Comptabilité et limitations des ressources par conteneurs
- Par défaut : part équitable de ressources (CPU, RAM, I/O) par conteneur
- Empêche l'utilisation de toutes les ressources par un conteneur (démon fork bomb –cpuset-cpus 0)
- D'une certaine manière, ils empêchent certaines attaques de type DOS
- "Garantit" une disponibilité et une performance quasiment constante

Sécurité des conteneurs

CGroups : limitations des ressources

- Comptabilité et limitations des ressources par conteneurs
- Par défaut : part équitable de ressources (CPU, RAM, I/O) par conteneur
- Empêche l'utilisation de toutes les ressources par un conteneur (démon fork bomb –cpuset-cpus 0)
- D'une certaine manière, ils empêchent certaines attaques de type DOS
- "Garantit" une disponibilité et une performance quasiment constante

Sécurité des conteneurs

CGroups : limitations des ressources

- Comptabilité et limitations des ressources par conteneurs
- Par défaut : part équitable de ressources (CPU, RAM, I/O) par conteneur
- Empêche l'utilisation de toutes les ressources par un conteneur (démon fork bomb –cpuset-cpus 0)
- D'une certaine manière, ils empêchent certaines attaques de type DOS
- "Garantit" une disponibilité et une performance quasiment constante

Démon docker

Surface d'attaque

- Le démon docker est exécuté en root !
- Seul un administrateur devrait pouvoir contrôler ce démon. (quid wrapper)
- Quelle confiance accorder vous à l'image docker que vous téléchargez ?
- Il est recommandé de n'utiliser que docker sur un serveur (à part ssh, dhcp, etc) (s'il y a d'autres applications, il faut les conteneuriser ou il faut les déplacer)
- Il ne faut pas faire ses tests de conteneur sur une machine de prod !

Démon docker

Surface d'attaque

- Le démon docker est exécuté en root !
- Seul un administrateur devrait pouvoir contrôler ce démon. (quid wrapper)
- Quelle confiance accorder vous à l'image docker que vous téléchargez ?
- Il est recommandé de n'utiliser que docker sur un serveur (à part ssh, dhcp, etc) (s'il y a d'autres applications, il faut les conteneuriser ou il faut les déplacer)
- Il ne faut pas faire ses tests de conteneur sur une machine de prod !

Démon docker

Surface d'attaque

- Le démon docker est exécuté en root !
- Seul un administrateur devrait pouvoir contrôler ce démon. (quid wrapper)
- Quelle confiance accorder vous à l'image docker que vous téléchargez ?
- Il est recommandé de n'utiliser que docker sur un serveur (à part ssh, dhcp, etc) (s'il y a d'autres applications, il faut les conteneuriser ou il faut les déplacer)
- Il ne faut pas faire ses tests de conteneur sur une machine de prod !

Démon docker

Surface d'attaque

- Le démon docker est exécuté en root !
- Seul un administrateur devrait pouvoir contrôler ce démon. (quid wrapper)
- Quelle confiance accorder vous à l'image docker que vous téléchargez ?
- Il est recommandé de n'utiliser que docker sur un serveur (à part ssh, dhcp, etc) (s'il y a d'autres applications, il faut les conteneuriser ou il faut les déplacer)
- Il ne faut pas faire ses tests de conteneur sur une machine de prod !

Démon docker

Surface d'attaque

- Le démon docker est exécuté en root !
- Seul un administrateur devrait pouvoir contrôler ce démon. (quid wrapper)
- Quelle confiance accorder vous à l'image docker que vous téléchargez ?
- Il est recommandé de n'utiliser que docker sur un serveur (à part ssh, dhcp, etc) (s'il y a d'autres applications, il faut les conteneuriser ou il faut les déplacer)
- Il ne faut pas faire ses tests de conteneur sur une machine de prod !

Plan

- 1 Introduction générale
 - Conteneur versus machine virtuelle
 - Des technologies de conteneurisations différentes pour des objectifs différents
- 2 Bases
 - Introduction
 - Définitions et prérequis
 - Premiers pas
 - Volatilité, persistance et sécurité des données
 - Sécurité des conteneurs
 - Conteneurs graphiques
- 3 Utilisation avancées
 - Créations d'images : méthode naïve
 - Créations d'images : Dockerfile
- 4 Toujours plus loin
 - Travaux pratiques

C'est possible

- Il est tout à fait possible d'exécuter des conteneurs graphiques (skype dans un conteneur par exemple).
- En général, on fait un mapping de `/tmp/.X11-unix` et de `/.Xauthority` de l'hôte vers le conteneur avec Xorg.
- Certains utilisent vnc (beaucoup moins efficace, mais plus sécurisé ...).
- Quid de Wayland ?
- Nous n'aborderons pas plus cet aspect dans ce cours.

Plan

- 1 Introduction générale
 - Conteneur versus machine virtuelle
 - Des technologies de conteneurisations différentes pour des objectifs différents
- 2 Bases
 - Introduction
 - Définitions et prérequis
 - Premiers pas
 - Volatilité, persistance et sécurité des données
 - Sécurité des conteneurs
 - Conteneurs graphiques
- 3 Utilisation avancées
 - Créations d'images : méthode naïve
 - Créations d'images : Dockerfile
- 4 Toujours plus loin
 - Travaux pratiques

La méthode naïve

3 cas d'usage

- Pour un usage pédagogique
- Lorsque l'on est pas en mesure de configurer un environnement autrement qu'en mode graphique :- (éclipse par exemple)
- Pour déboguer une construction automatisée (*docker build*)

La méthode naïve

Utilisation de docker commit

Terminal:

```
jmc@laptop/home/jmc $ docker run -it -h conteneur --name  
conteneur_avant_commit ubuntu /bin/bash
```

Conteneur:

```
root@conteneur :/# echo "hello world !"  
>/nouveau_fichier_dans_le_systeme_de_fichier_racine.txt  
root@conteneur :/# exit  
exit
```

La méthode naïve

Utilisation de docker commit

Terminal:

```
jmc@laptop/home/jmc $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	7b9b13f7b9c0	2 weeks ago	118 MB

```
jmc@laptop/home/jmc $ docker commit conteneur_avant_commit
```

```
image_apres_commit_du_conteneur
```

```
sha256 :ca0028fa88252c281db33655b17517f3484b82ae42057e26193a6b91045bf2b8
```

```
jmc@laptop/home/jmc $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
image_apres_commit_du_conteneur	latest	ca0028fa8825	16 minutes ago	118 MB
ubuntu	latest	7b9b13f7b9c0	2 weeks ago	118 MB

La méthode naïve

Utilisation de docker commit

Terminal:

```
jmc@laptop/home/jmc $ docker run --rm image_apres_commit_du_conteneur cat /nouveau_fichier_dans_le_systeme_de_fichier_racine.txt
```

Conteneur:

hello world !

```
jmc@laptop/home/jmc $
```

La méthode naïve

A vous

- Faire une image "mon_image" à partir d'une ubuntu 16.04
- Cette nouvelle image contiendra le paquet inetutils-ping
- Tester la commande "*docker run mon_image ping localhost*"

Problème dns dans certains sous-réseaux

Il se peut que votre réseau vous oblige à passer par un dns précis. Ce dns précis ne correspond pas forcément aux adresses présentent dans /etc/resolv.conf de l'hôte (surcharge effectué par network manager). Dans ce cas, la résolution de nom ne se fera pas dans le conteneur (et vous ne pourrez pas installer de paquets), car la surcharge de network manager n'affectera pas le conteneur (qui s'en tiendra au fichier /etc/resolv.conf). De fait, il vous faut soit changer le contenu de /etc/resolv.conf, soit passer l'argument `-dns IP_DNS` lors du lancement du conteneur. Pour obtenir IP_DNS, il suffit de voir quel serveur DNS répond lorsque vous faites une requête dns sur l'hôte avec nslookup.

Plan

- 1 Introduction générale
 - Conteneur versus machine virtuelle
 - Des technologies de conteneurisations différentes pour des objectifs différents
- 2 Bases
 - Introduction
 - Définitions et prérequis
 - Premiers pas
 - Volatilité, persistance et sécurité des données
 - Sécurité des conteneurs
 - Conteneurs graphiques
- 3 Utilisation avancées
 - Créations d'images : méthode naïve
 - Créations d'images : Dockerfile
- 4 Toujours plus loin
 - Travaux pratiques

La méthode efficace : Dockerfile

- Un Dockerfile est un document texte contenant toutes les commandes que vous exécuteriez normalement manuellement sur un OS (apt/yum, configurations de logiciels, créations d'utilisateurs, variables d'environnements, ...).
- Avec la commande *docker build*, docker peut créer des images **automatiquement** en lisant les instructions d'un Dockerfile.
- A **chaque** instructions du dockerfile, docker build génère une image et un layer si nécessaire (car potentiellement vide), on parle d'image intermédiaire, sauf pour la dernière instruction, qui correspond à l'image finale (top level image).
- Les images intermédiaires sont générées par un mécanisme de commit des conteneurs dans lesquels les instructions ont été exécutés.
- On peut créer plusieurs images avec un seul Dockerfile (mais on ne devrait pas)

La méthode efficace : Dockerfile

- Un Dockerfile est un document texte contenant toutes les commandes que vous exécuteriez normalement manuellement sur un OS (apt/yum, configurations de logiciels, créations d'utilisateurs, variables d'environnements, ...).
- Avec la commande *docker build*, docker peut créer des images **automatiquement** en lisant les instructions d'un Dockerfile.
- A **chaque** instructions du dockerfile, docker build génère une image et un layer si nécessaire (car potentiellement vide), on parle d'image intermédiaire, sauf pour la dernière instruction, qui correspond à l'image finale (top level image).
- Les images intermédiaires sont générées par un mécanisme de commit des conteneurs dans lesquels les instructions ont été exécutés.
- On peut créer plusieurs images avec un seul Dockerfile (mais on ne devrait pas)

La méthode efficace : Dockerfile

- Un Dockerfile est un document texte contenant toutes les commandes que vous exécuteriez normalement manuellement sur un OS (apt/yum, configurations de logiciels, créations d'utilisateurs, variables d'environnements, ...).
- Avec la commande *docker build*, docker peut créer des images **automatiquement** en lisant les instructions d'un Dockerfile.
- A **chaque** instructions du dockerfile, docker build génère une image et un layer si nécessaire (car potentiellement vide), on parle d'image intermédiaire, sauf pour la dernière instruction, qui correspond à l'image finale (top level image).
- Les images intermédiaires sont générées par un mécanisme de commit des conteneurs dans lesquels les instructions ont été exécutés.
- On peut créer plusieurs images avec un seul Dockerfile (mais on ne devrait pas)

La méthode efficace : Dockerfile

- Un Dockerfile est un document texte contenant toutes les commandes que vous exécuteriez normalement manuellement sur un OS (apt/yum, configurations de logiciels, créations d'utilisateurs, variables d'environnements, ...).
- Avec la commande *docker build*, docker peut créer des images **automatiquement** en lisant les instructions d'un Dockerfile.
- A **chaque** instructions du dockerfile, docker build génère une image et un layer si nécessaire (car potentiellement vide), on parle d'image intermédiaire, sauf pour la dernière instruction, qui correspond à l'image finale (top level image).
- Les images intermédiaires sont générées par un mécanisme de commit des conteneurs dans lesquels les instructions ont été exécutés.
- On peut créer plusieurs images avec un seul Dockerfile (mais on ne devrait pas)

La méthode efficace : Dockerfile

- Un Dockerfile est un document texte contenant toutes les commandes que vous exécuteriez normalement manuellement sur un OS (apt/yum, configurations de logiciels, créations d'utilisateurs, variables d'environnements, ...).
- Avec la commande *docker build*, docker peut créer des images **automatiquement** en lisant les instructions d'un Dockerfile.
- A **chaque** instructions du dockerfile, docker build génère une image et un layer si nécessaire (car potentiellement vide), on parle d'image intermédiaire, sauf pour la dernière instruction, qui correspond à l'image finale (top level image).
- Les images intermédiaires sont générées par un mécanisme de commit des conteneurs dans lesquels les instructions ont été exécutés.
- On peut créer plusieurs images avec un seul Dockerfile (mais on ne devrait pas)

La méthode efficace : Dockerfile

- Il faut dédier un répertoire pour chaque création d'image par dockerfile. En effet, docker build fonctionne avec une notion de contexte qui s'applique récursivement à partir du chemin (path) du répertoire contenant le Dockerfile.
- Il ne faut surtout pas créer un Dockerfile à la racine du système. En effet, le contexte envoyé au démon docker concernerait l'intégralité du système de fichier !
- À l'instar de git, vous pouvez utiliser un fichier .dockerignore.
- docker build ignorera tous les fichiers/répertoires indiqués dans .dockerignore (i.e. ils ne feront pas partie du contexte).
- Les commentaires dans un Dockerfile sont préfixés par #.

La méthode efficace : Dockerfile

- Il est possible d'utiliser des variables interprétables par certaines instructions dans un Dockerfile.
- Ces variables sont des variables d'environnement (comme HOME, PATH, ...)

La méthode efficace : Dockerfile

Liste des instructions possible

- FROM
- RUN
- CMD
- LABEL
- MAINTAINER (déprécié, utiliser LABEL)
- EXPOSE
- ENV
- ADD
- COPY
- ENTRYPOINT
- VOLUME
- USER
- WORKDIR
- ARG
- ONBUILD
- STOPSIGNAL
- HEALTHCHECK
- SHELL

Capables d'interpréter les variables d'environnement

La méthode efficace : Dockerfile

Dockerfile d'un serveur ssh

```
# Image de départ
FROM debian:testing-slim

# Variables d'environnement
ENV VERSION=1 \
    DEBIAN_FRONTEND=noninteractive

# Metadonnées de l'image
LABEL maintainer="Jean-Mathieu Chantrein" \
    maintainer_email="jean-mathieu.chantrein@univ-angers.fr" \
    version="${VERSION}"

# Création d'un utilisateur lambda
RUN useradd lambda --create-home --shell /bin/bash && \
    echo "lambda:password" | chpasswd

# Installation des packages, suppression du cache apt
RUN apt update && \
    apt install -yq openssh-server && \
    rm -rf /var/lib/apt/lists/* && \
    mkdir /var/run/ssh

# setuid sur le démon ssh
RUN chmod u+s /usr/sbin/sshd

# root => lambda
USER lambda

# Commande par défaut au lancement du conteneur
CMD ["/usr/sbin/sshd", "-D"]
```

La méthode efficace : Dockerfile

Dockerfile serveur ssh : /home/jmc/sshd/Dockerfile

Terminal:

```
jmc@laptop/home/jmc/sshd $ docker build . -t sshd
```

```
Sending build context to Docker daemon 8.192 kB
```

```
Step 1/8 : FROM debian:testing-slim
```

```
testing-slim: Pulling from library/debian
```

```
8f99754b7fe0: Pull complete
```

```
Digest: sha256:a00ac0aa75d79b5db3e5b0c66ddae4ccb6a9b0d28a856607de7ef2ac6527cdc7
```

```
Status: Downloaded newer image for debian:testing-slim
```

```
---> f2020c7a5565
```

```
Step 2/8 : ENV VERSION 1 DEBIAN_FRONTEND noninteractive
```

```
---> Running in 358b9c6ec81e
```

```
---> e3ef9a7b6a23
```

```
Removing intermediate container 358b9c6ec81e
```

```
Step 3/8 : LABEL maintainer "Jean-Mathieu Chantrein" maintainer_email "jean-mathieu.chantrein@univ-angers.fr" version "${VERSION}"
```

```
---> Running in f4d7a947fe58
```

```
---> fbb7230757c2
```

```
Removing intermediate container f4d7a947fe58
```

```
Step 4/8 : RUN useradd lambda --create-home --shell /bin/bash && echo "lambda:password" | chpasswd
```

```
---> Running in a9c930c23c4a
```

```
---> a17fecb71d55
```

```
Removing intermediate container a9c930c23c4a
```


La méthode efficace : Dockerfile

Dockerfile serveur ssh : /home/jmc/sshd/Dockerfile

Terminal:

```
Step 5/8 : RUN apt update &&          apt install -yq openssh-server &&          rm -rf /varlib/apt/lists/* &&          mkdir /var/run/sshd
--> Running in c378b292a0d0

[update, install, config ...]

--> 15bd3d357785
Removing intermediate container c378b292a0d0
Step 6/8 : RUN chmod u+s /usr/sbin/sshd
--> Running in a697d7d0bcde
--> 174bd751e04d
Removing intermediate container a697d7d0bcde
Step 7/8 : USER lambda
--> Running in e512834f2b07
--> c7002394e3de
Removing intermediate container e512834f2b07
Step 8/8 : CMD /usr/sbin/sshd -D
--> Running in 83b6f14485f3
--> 1c1305bc7c28
Removing intermediate container 83b6f14485f3
Successfully built 1c1305bc7c28
```

jmc@laptop/home/jmc/sshd \$

La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Image de départ  
FROM debian:testing-slim
```

Pull debian:testing-slim

Step 1

f2020c7a5565

La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Variables d'environnement  
ENV VERSION=1 \  
    DEBIAN_FRONTEND=noninteractive
```

Step 2

f2020c7a5565

358b9c6ec81e

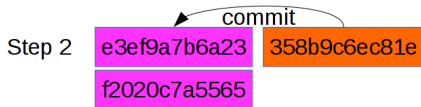
run



La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Variables d'environnement  
ENV VERSION=1 \  
    DEBIAN_FRONTEND=noninteractive
```



La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Variables d'environnement
ENV VERSION=1 \
    DEBIAN_FRONTEND=noninteractive
```

Step 2

e3ef9a7b6a23

f2020c7a5565

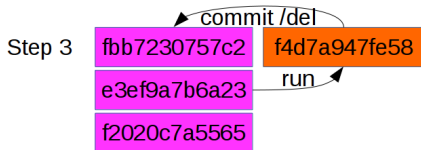
Suppression

~~358b9c8ec81e~~

La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Metadonnées de l'image
LABEL maintainer="Jean-Mathieu Chantrein" \
    maintainer_email="jean-mathieu.chantrein@univ-angers.fr" \
    version="${VERSION}"
```



La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Création d'un utilisateur lambda  
RUN useradd lambda --create-home --shell /bin/bash && \  
    echo "lambda:password" | chpasswd
```

Step 4

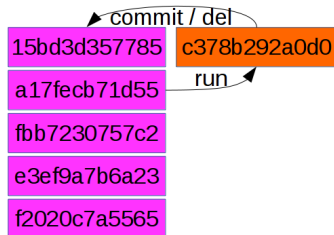


La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Installation des packages, suppression du cache apt
RUN apt update && \
    apt install -y openssh-server && \
    rm -rf /var/lib/apt/lists/* && \
    mkdir /var/run/sshd
```

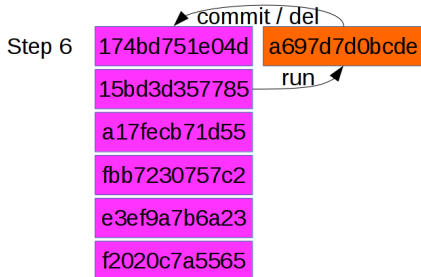
Step 5



La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# setuid sur le démon ssh  
RUN chmod u+s /usr/sbin/sshd
```



La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# root => lambda  
USER lambda
```



La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique



```
# Commande par défaut au lancement du conteneur  
CMD ["/usr/sbin/sshd", "-D"]
```

La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Image de départ
FROM debian:testing-slim

# Variables d'environnement
ENV VERSION=1 \
    DEBIAN_FRONTEND=noninteractive

# Metadonnées de l'image
LABEL maintainer="Jean-Mathieu Chantrein" \
    maintainer_email="jean-mathieu.chantrein@univ-angers.fr" \
    version="${VERSION}"

# Création d'un utilisateur lambda
RUN useradd lambda --create-home --shell /bin/bash && \
    echo "lambda:password" | chpasswd

# Installation des packages, suppression du cache apt
RUN apt update && \
    apt install -yq openssh-server && \
    rm -rf /var/lib/apt/lists/* && \
    mkdir /var/run/ssh

# setuid sur le démon ssh
RUN chmod u+s /usr/sbin/sshd

# root => lambda
USER lambda

# Commande par défaut au lancement du conteneur
CMD ["/usr/sbin/sshd", "-D"]
```

Step 8

1c1305bc7c28

sshd

Step 7

c7002394e3de

Step 6

174bd751e04d

Step 5

15bd3d357785

Step 4

a17fecb71d55

Step 3

fbf7230757c2

Step 2

e3ef9a7b6a23

Step 1

f2020c7a5565

debian:testing

La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Image de départ
FROM debian:testing-slim

# Variables d'environnement
ENV VERSION=1 \
    DEBIAN_FRONTEND=noninteractive

# Metadonnées de l'image
LABEL maintainer="Jean-Mathieu Chantrein" \
    maintainer_email="jean-mathieu.chantrein@univ-angers.fr" \
    version="${VERSION}"

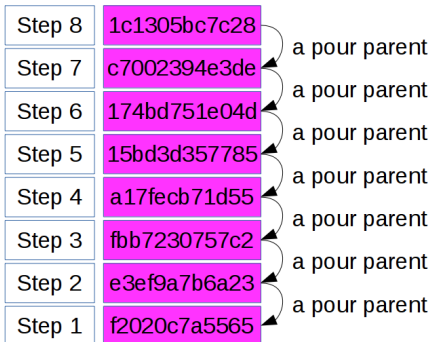
# Création d'un utilisateur lambda
RUN useradd lambda --create-home --shell /bin/bash && \
    echo "lambda:password" | chpasswd

# Installation des packages, suppression du cache apt
RUN apt update && \
    apt install -yq openssh-server && \
    rm -rf /var/lib/apt/lists/* && \
    mkdir /var/run/ssh

# setuid sur le démon ssh
RUN chmod u+s /usr/sbin/sshd

# root => lambda
USER lambda

# Commande par défaut au lancement du conteneur
CMD ["/usr/sbin/sshd", "-D"]
```



La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Image de départ
FROM debian:testing-slim

# Variables d'environnement
ENV VERSION=1 \
    DEBIAN_FRONTEND=noninteractive

# Metadonnées de l'image
LABEL maintainer="Jean-Mathieu Chantrein" \
    maintainer_email="jean-mathieu.chantrein@univ-angers.fr" \
    version="{VERSION}"

# Création d'un utilisateur lambda
RUN useradd lambda --create-home --shell /bin/bash && \
    echo "lambda:password" | chpasswd

# Installation des packages, suppression du cache apt
RUN apt update && \
    apt install -y openssh-server && \
    rm -rf /var/lib/apt/lists/* && \
    mkdir /var/run/ssh

# setuid sur le démon ssh
RUN chmod u+s /usr/sbin/sshd

# root => lambda
USER lambda

# Commande par défaut au lancement du conteneur
CMD ["/usr/sbin/sshd", "-D"]
```

1c1305bc7c28

c7002394e3de

174bd751e04d

15bd3d357785

a17fecb71d55

fbf7230757c2

e3ef9a7b6a23

f2020c7a5565

La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

```
# Image de départ
FROM debian:testing-slim
```

```
# Variables d'environnement
ENV VERSION=1 \
    DEBIAN_FRONTEND=noninteractive
```

```
# Metadonnées de l'image
LABEL maintainer="Jean-Mathieu Chantrein" \
    maintainer_email="jean-mathieu.chantrein@univ-angers.fr" \
    version="${VERSION}"
```

```
# Création d'un utilisateur lambda
RUN useradd lambda --create-home --shell /bin/bash && \
    echo "lambda:password" | chpasswd
```

```
# Installation des packages, suppression du cache apt
RUN apt update && \
    apt install -y openssh-server && \
    rm -rf /var/lib/apt/lists/* && \
    mkdir /var/run/ssh
```

```
# setuid sur le démon ssh
RUN chmod u+s /usr/sbin/sshd
```

```
# root => lambda
USER lambda
```

```
# Commande par défaut au lancement du conteneur
CMD ["/usr/sbin/sshd", "-D"]
```

1c1305bc7c28

c7002394e3de

174bd751e04d

15bd3d357785

a17fecb71d55

fb7230757c2

e3ef9a7b6a23

f2020c7a5565

La méthode efficace : Dockerfile

Dockerfile serveur ssh : explication graphique

TOP LEVEL IMAGE → dernière ligne du dockerfile

```
# Commande par défaut au lancement du conteneur  
CMD ["/usr/sbin/sshd", "-D"]
```

```
# root => lambda  
USER lambda
```

```
# setuid sur le démon ssh  
RUN chmod u+s /usr/sbin/sshd
```

```
# Installation des packages, suppression du cache apt  
RUN apt update && \  
apt install -y openssh-server && \  
rm -rf /var/lib/apt/lists/* && \  
mkdir /var/run/sshd
```

```
# Création d'un utilisateur lambda  
RUN useradd lambda --create-home --shell /bin/bash && \  
echo "lambda:password" | chpasswd
```

```
# Metadonnées de l'image  
LABEL maintainer="Jean-Mathieu Chantrein" \  
maintainer_email="jean-mathieu.chantrein@univ-angers.fr" \  
version="${VERSION}"
```

```
# Variables d'environnement  
ENV VERSION=1 \  
DEBIAN_FRONTEND=noninteractive
```

```
# Image de départ  
FROM debian:testing-slim
```

Step-8 → 1c1305bc7c28

Step-7 → c7002394e3de

Step-6 → 174bd751e04d

Step-5 → 15bd3d357785

Step-4 → a17fecb71d55

Step-3 → fbb7230757c2

Step-2 → e3ef9a7b6a23

Step-1 → f2020c7a5565

a pour parent

a pour parent

a pour parent

a pour parent

a pour parent

a pour parent

a pour parent

La méthode efficace : Dockerfile

Dockerfile d'un serveur ssh : notions de contexte et de cache

```
# Image de départ
FROM debian:testing-slim

# Variables d'environnement
ENV VERSION=1 \
    DEBIAN_FRONTEND=noninteractive

# Metadonnées de l'image
LABEL maintainer="Jean-Mathieu Chantrein" \
    maintainer_email="jean-mathieu.chantrein@univ-angers.fr" \
    version="${VERSION}"

# Création d'un utilisateur lambda
RUN useradd lambda --create-home --shell /bin/bash && \
    echo "lambda:password" | chpasswd

# Installation des packages, suppression du cache apt
RUN apt update && \
    apt install -y openssh-server && \
    rm -rf /var/lib/apt/lists/* && \
    mkdir /var/run/sshd

# setuid sur le démon ssh
RUN chmod u+s /usr/sbin/sshd && \
    echo "Une modification de cette instruction"

# root => lambda
USER lambda

# Commande par défaut au lancement du conteneur
CMD ["/usr/sbin/sshd", "-D"]
```

La méthode efficace : Dockerfile

Dockerfile serveur ssh : notions de contexte et de cache

Terminal:

```
jmc@laptop/home/jmc/sshd $ docker build . -t sshd
```

```
Sending build context to Docker daemon 8.192 kB
```

```
Step 1/8 : FROM debian:testing-slim
```

```
----> f2020c7a5565
```

```
Step 2/8 : ENV VERSION 1 DEBIAN_FRONTEND noninteractive
```

```
----> Using cache
```

```
----> e3ef9a7b6a23
```

```
Step 3/8 : LABEL maintainer "Jean-Mathieu Chantrein" maintainer_email "jean-mathieu.chantrein@univ-angers.fr" version "${VERSION}"
```

```
----> Using cache
```

```
----> fbb7230757c2
```

```
Step 4/8 : RUN useradd lambda --create-home --shell /bin/bash &&      echo "lambda:password" | chpasswd
```

```
----> Using cache
```

```
----> a17fecb71d55
```

```
Step 5/8 : RUN apt update &&      apt install -yq openssh-server &&      rm -rf /varlib/apt/lists/* &&      mkdir /var/run/sshd
```

```
----> Using cache
```

```
----> 15bd3d357785
```

La méthode efficace : Dockerfile

Dockerfile serveur ssh : notions de contexte et de cache

Terminal:

```
Step 6/8 : RUN chmod u+s /usr/sbin/sshd && echo "Une modification de cette instruction"
--> Running in a74783d3335a
Une modification de cette instruction
--> 1e8030a5f34b
Removing intermediate container a74783d3335a
Step 7/8 : USER lambda
--> Running in 6275d6897c41
--> 07c952331a94
Removing intermediate container 6275d6897c41
Step 8/8 : CMD /usr/sbin/sshd -D
--> Running in 35b3a68cb6e8
--> 8f8b2b932bc7
Removing intermediate container 35b3a68cb6e8
Successfully built 8f8b2b932bc7
```

jmc@laptop/home/jmc/sshd \$

La méthode efficace : Dockerfile

Dockerfile serveur ssh : notions de contexte et de cache

Terminal:

```
jmc@laptop/home/jmc/sshd $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
sshd	latest	8f8b2b932bc7	17 minutes ago	143 MB
<none>	<none>	1c1305bc7c28	2 days ago	143 MB
debian	testing-slim	f2020c7a5565	5 days ago	55.2 MB

```
jmc@laptop/home/jmc/sshd $ docker images -a # Combien d'images ?
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	07c952331a94	9 minutes ago	143 MB
<none>	<none>	1e8030a5f34b	9 minutes ago	143 MB
sshd	latest	8f8b2b932bc7	9 minutes ago	143 MB
<none>	<none>	1c1305bc7c28	2 days ago	143 MB
<none>	<none>	c7002394e3de	2 days ago	143 MB
<none>	<none>	15bd3d357785	2 days ago	142 MB
<none>	<none>	174bd751e04d	2 days ago	143 MB
<none>	<none>	a17fecb71d55	2 days ago	55.6 MB
<none>	<none>	e3ef9a7b6a23	2 days ago	55.2 MB
<none>	<none>	fbb7230757c2	2 days ago	55.2 MB
debian	testing-slim	f2020c7a5565	5 days ago	55.2 MB

La méthode efficace : Dockerfile

Dockerfile serveur ssh : notions de contexte et de cache

Terminal:

```
jmc@laptop/home/jmc/sshd $ docker images -a --filter dangling=true
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	1c1305bc7c28	2 days ago	143 MB

```
jmc@laptop/home/jmc/sshd $ docker rmi 1c1
```

```
Deleted: sha256:1c1305bc7c283f1e5644209d9144184f9b1d4fd15020486eda7217227f7ad3e4  
Deleted: sha256:c7002394e3de48d4e085b957c747fbb1f82123b09ccc1987045b4692c6834eff  
Deleted: sha256:174bd751e04da05124ec3b7c42cd04f22e2507b11afde8782e772a755d1ee4b5
```

- Suppression automatique des images non utilisé (i.e. : n'étant parente d'aucune top level image tagué)

La méthode efficace : Dockerfile

Dockerfile serveur ssh : notions de contexte et de cache

Terminal:

```
jmc@laptop/home/jmc/sshd $ docker images -a # Combien d'images ?
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
sshd	latest	8f8b2b932bc7	29 minutes ago	143 MB
<none>	<none>	07c952331a94	29 minutes ago	143 MB
<none>	<none>	1e8030a5f34b	29 minutes ago	143 MB
<none>	<none>	15bd3d357785	2 days ago	142 MB
<none>	<none>	a17fecb71d55	2 days ago	55.6 MB
<none>	<none>	fb7230757c2	2 days ago	55.2 MB
<none>	<none>	e3ef9a7b6a23	2 days ago	55.2 MB
debian	testing-slim	f2020c7a5565	5 days ago	55.2 MB

- Les images non utilisées ont bel et bien été supprimées

La méthode efficace : Dockerfile

Dockerfile serveur ssh : notions de contexte et de cache

Terminal:

```
jmc@laptop/home/jmc/sshd $ docker history sshd
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
8f8b2b932bc7	About an hour ago	/bin/sh -c #(nop) CMD ["/usr/sbin/sshd" "...	0 B	
07c952331a94	About an hour ago	/bin/sh -c #(nop) USER [lambda]	0 B	
1e8030a5f34b	About an hour ago	/bin/sh -c chmod u+s /usr/sbin/sshd && ec...	791 kB	
15bd3d357785	3 days ago	/bin/sh -c apt update && apt install -...	86.8 MB	
a17fecb71d55	3 days ago	/bin/sh -c useradd lambda --create-home -...	333 kB	
ffb7230757c2	3 days ago	/bin/sh -c #(nop) LABEL maintainer=Jean-M...	0 B	
e3ef9a7b6a23	3 days ago	/bin/sh -c #(nop) ENV VERSION=1 DEBIAN_FR...	0 B	
f2020c7a5565	5 days ago	/bin/sh -c #(nop) CMD ["bash"]	0 B	
<missing>	5 days ago	/bin/sh -c #(nop) ADD file:2bad6c695080ae4...	55.2 MB	

- Les images de taille 0 n'ont pas généré de layer et repose sur les layers des images parentes (on parle de layer vide (empty layer))
- Pourquoi y a-t-il une différence de 3 jours dans la création des images ?

La méthode efficace : Dockerfile

Encore quelques informations

- Il est important de taguer/versionner ses images de production
- Les images les plus lourdes devraient être les plus basses (utilisation efficace du mécanisme du cache)
- Les instructions susceptibles de changer régulièrement devraient se situer dans les couches basses
- Attention aux directives "update" et "install" lors de l'utilisation du cache
- Tout changement de contexte (modifications d'un fichier ajouté par ADD ou COPY) entraîne une reconstruction des images à partir de la couche affectée
- Chaque construction (build) d'un Dockerfile génère une **nouvelle** image (modulo une utilisation totale du cache).
- Chaque instruction est indépendante (cela n'a pas de sens d'écrire `RUN cd /tmp`)

La méthode efficace : Dockerfile

Encore quelques informations

- Il est important de taguer/versionner ses images de production
- Les images les plus lourdes devraient être les plus basses (utilisation efficace du mécanisme du cache)
- Les instructions susceptibles de changer régulièrement devraient se situer dans les couches basses
- Attention aux directives "update" et "install" lors de l'utilisation du cache
- Tout changement de contexte (modifications d'un fichier ajouté par ADD ou COPY) entraîne une reconstruction des images à partir de la couche affectée
- Chaque construction (build) d'un Dockerfile génère une **nouvelle** image (modulo une utilisation totale du cache).
- Chaque instruction est indépendante (cela n'a pas de sens d'écrire RUN cd /tmp)

La méthode efficace : Dockerfile

Encore quelques informations

- Il est important de taguer/versionner ses images de production
- Les images les plus lourdes devraient être les plus basses (utilisation efficace du mécanisme du cache)
- Les instructions susceptibles de changer régulièrement devraient se situer dans les couches basses
- Attention aux directives "update" et "install" lors de l'utilisation du cache
- Tout changement de contexte (modifications d'un fichier ajouté par ADD ou COPY) entraîne une reconstruction des images à partir de la couche affectée
- Chaque construction (build) d'un Dockerfile génère une **nouvelle** image (modulo une utilisation totale du cache).
- Chaque instruction est indépendante (cela n'a pas de sens d'écrire RUN cd /tmp)

La méthode efficace : Dockerfile

Encore quelques informations

- Il est important de taguer/versionner ses images de production
- Les images les plus lourdes devraient être les plus basses (utilisation efficace du mécanisme du cache)
- Les instructions susceptibles de changer régulièrement devraient se situer dans les couches basses
- Attention aux directives "update" et "install" lors de l'utilisation du cache
- Tout changement de contexte (modifications d'un fichier ajouté par ADD ou COPY) entraîne une reconstruction des images à partir de la couche affectée
- Chaque construction (build) d'un Dockerfile génère une **nouvelle** image (modulo une utilisation totale du cache).
- Chaque instruction est indépendante (cela n'a pas de sens d'écrire RUN cd /tmp)

La méthode efficace : Dockerfile

Encore quelques informations

- Il est important de taguer/versionner ses images de production
- Les images les plus lourdes devraient être les plus basses (utilisation efficace du mécanisme du cache)
- Les instructions susceptibles de changer régulièrement devraient se situer dans les couches basses
- Attention aux directives "update" et "install" lors de l'utilisation du cache
- Tout changement de contexte (modifications d'un fichier ajouté par ADD ou COPY) entraîne une reconstruction des images à partir de la couche affectée
- Chaque construction (build) d'un Dockerfile génère une **nouvelle** image (modulo une utilisation totale du cache).
- Chaque instruction est indépendante (cela n'a pas de sens d'écrire RUN cd /tmp)

La méthode efficace : Dockerfile

Encore quelques informations

- Il est important de taguer/versionner ses images de production
- Les images les plus lourdes devraient être les plus basses (utilisation efficace du mécanisme du cache)
- Les instructions susceptibles de changer régulièrement devraient se situer dans les couches basses
- Attention aux directives "update" et "install" lors de l'utilisation du cache
- Tout changement de contexte (modifications d'un fichier ajouté par ADD ou COPY) entraîne une reconstruction des images à partir de la couche affectée
- Chaque construction (build) d'un Dockerfile génère une **nouvelle** image (modulo une utilisation totale du cache).
- Chaque instructions est indépendante (cela n'a pas de sens d'écrire RUN cd /tmp)

La méthode efficace : Dockerfile

Encore quelques informations

- Il est important de taguer/versionner ses images de production
- Les images les plus lourdes devraient être les plus basses (utilisation efficace du mécanisme du cache)
- Les instructions susceptibles de changer régulièrement devraient se situer dans les couches basses
- Attention aux directives "update" et "install" lors de l'utilisation du cache
- Tout changement de contexte (modifications d'un fichier ajouté par ADD ou COPY) entraîne une reconstruction des images à partir de la couche affectée
- Chaque construction (build) d'un Dockerfile génère une **nouvelle** image (modulo une utilisation totale du cache).
- Chaque instruction est indépendante (cela n'a pas de sens d'écrire RUN cd /tmp)

La méthode efficace : Dockerfile

Terminal:

```
jmc@laptop/home/jmc $ docker built . -t independant_instruction
```

```
Step 1/4 : FROM alpine:latest
```

```
---> 7328f6f8b418
```

```
Step 2/4 : RUN cd /tmp
```

```
---> Running in d172a17e8705
```

```
---> 1eb259884a9f
```

```
Removing intermediate container d172a17e8705
```

```
Step 3/4 : RUN pwd
```

```
---> Running in 80a06bfd0434
```

```
/
```

```
---> 9f59028b0464
```

```
Removing intermediate container 80a06bfd0434
```

```
Step 4/4 : [...]
```

Pour info

- Pour changer de répertoire pour les instructions à suivre (y compris pour d'éventuelles images enfants) ⇒ Utiliser l'instruction WORKDIR

Plan

- 1 Introduction générale
 - Conteneur versus machine virtuelle
 - Des technologies de conteneurisations différentes pour des objectifs différents
- 2 Bases
 - Introduction
 - Définitions et prérequis
 - Premiers pas
 - Volatilité, persistance et sécurité des données
 - Sécurité des conteneurs
 - Conteneurs graphiques
- 3 Utilisation avancées
 - Créations d'images : méthode naïve
 - Créations d'images : Dockerfile
- 4 Toujours plus loin
 - Travaux pratiques

Travaux pratiques : de dockerfile à docker swarm en passant par docker compose

Réaliser un service wordpress conteneurisé

- Nous changeons de support afin de passer (enfin) à la pratique
- Merci de vous rendre sur la page web suivante : https://github.com/jmchantrein/formation_docker/blob/master/travaux_pratiques.md