

Virtualisation et Containerisation avec Docker

Institut F2i

Jaber

Janvier 2019

Programme

- Installation
- Docker proxy
- Premier conteneur
- Image personnalisée
- Création d'image
- *Upload* sur Docker hub
- Options basiques de docker run
- Dockerfile

Programme

- Volumes
- Lien entre conteneur
- Ressources
- *Private Registry*
- Docker-compose
- Docker-machine
- Docker Swarm
- Docker Network

Les Technologies de Virtualisation

Systeme invité / Systeme Hôte

- Le paradigme *systeme invité / systeme hôte* représente la *virtualisation classique* ou *virtualisation hébergée*. Ce type de virtualisation repose sur système existant – le système d'exploitation hôte, une solution de virtualisation tierce et la création de divers *systeme d'exploitation invités*.
- Chaque invité fonctionne sur l'hôte en utilisant des ressources partagées qui lui sont attribuées par l'hôte. L'avantage principal de ce type de virtualisation est qu'il existe un nombre limité de périphériques et de pilotes à gérer. Chaque machine virtuelle – invitée – dispose d'un ensemble cohérent de matériel.

Les Technologies de Virtualisation

- L'inconvénient majeur est que les entrées-sorties disques souffrent dans le cadre de cette technologie.
- Le temps d'exécution des opérations ne faisant pas appel au disque est proche de celui des opérations natives. Par conséquent, il est préférable dans ce cas d'interagir avec les machines virtuelles via le réseau en utilisant *Windows Terminal Services* – Remote Desktop Protocol – pour les machines virtuelles Windows ou ssh pour les systèmes Unix/Linux/Mac OS.

Les Technologies de Virtualisation

VMware Server

- *VmWare Server* fonctionne selon le paradigme *système invité / système hôte*. Il est considéré comme un produit d'introduction à utiliser dans les petits environnements. Son utilité est limitée dans des environnements plus grands en raison des limites de mémoire pour les machines virtuelles et des basses performances disque. *VmWare Server* prend en charge les machines 64 bits en tant qu'hôtes et en tant qu'invités.

Les Technologies de Virtualisation

VirtualBox

- *VirtualBox*, comme *VMware Server* est multiplate-forme, mais à la différence de ce dernier, il est libre. Avec sa mémoire vidéo ajustable, sa connectivité aux périphériques distants, sa connectivité RDP et ses bonnes performances, il s'agit de l'un des outils de virtualisation hébergée le plus intéressant.
- *VirtualBox* est plus adapté pour les petits réseaux et pour les particuliers, pour les mêmes raisons que *VMware Server*.

Les Technologies de Virtualisation

Hyperviseur

- Un *hyperviseur* est une approche *bare-metal* ou *bare-machine* de la virtualisation. *Bare-metal* fait référence au matériel du serveur sans système d'exploitation installé. La meilleure manière de décrire la technologie d'un *hyperviseur* est de le comparer à la virtualisation hébergée. Au premier abord, l'hyperviseur peut sembler comparable à la virtualisation hébergée, mais il est significativement différent.
- Un *hyperviseur* est un logiciel de virtualisation qui fait fonctionner un système d'exploitation. À l'inverse, la virtualisation hébergée utilise un système d'exploitation et y fait fonctionner un logiciel de virtualisation comme application.

Les Technologies de Virtualisation

- L'*hyperviseur* est installé directement sur le matériel, puis le système d'exploitation est installé ; il est lui-même une *machine virtuelle paravirtualisée*. Le système d'exploitation hôte, si on l'appelle ainsi, est désigné sous le nom de *machine virtuelle zéro*.
- Des produits comme *VMware ESXi*, implémentent un hyperviseur bare-metal sans interface par système d'exploitation traditionnel. Il s'installe sur le matériel pour une empreinte faible de 32Mo.

Les Technologies de Virtualisation

Citrix Xen

- Les version 3.0 et précédentes de Xen n'étaient pas faciles à utiliser et ne semblaient pas fonctionner si bien.
- À partir de la version 4.x, en revanche est plus stable. L'interface graphique est intuitive, rapide et bien présentée. La mise en place de machines est rapide. À étudier pour des besoins de virtualisation haut de gamme.

VMware ESX/ VMware ESXi

- Des outils de virtualisation pour les entreprises. ESX est un produit mûr qui n'a pour concurrence que Xen à ce niveau de virtualisation. Les deux produits nécessitent une architecture 64 bits, mais ESXi a des demandes matérielles spécifiques au delà de celle d'ESX. ESXi est un produit gratuit :

<https://www.vmware.com/fr/products.html>

Les Technologies de Virtualisation

Microsoft Hyper-V

- *Microsoft* présente sa solution de virtualisation avec *Hyper-V*, afin de proposer une solution basée sur Windows en plus là où Citrix et VMware n'ont pas tout à fait réussi. Ces deux derniers sont basés sur Linux, ce qui signifie que, si vous n'êtes pas familier de l'environnement Unix/Linux, le produit *Microsoft* peut s'avérer un bon choix.

Les Technologies de Virtualisation

Émulation

- L'*émulation* fait référence à la capacité de mimer un type particulier de matériel pour un système d'exploitation indépendamment du système d'exploitation hôte. Par exemple, grâce à une solution d'émulation, vous pouvez installer une version Sparc du système *Solaris* sur un ordinateur hôte non Sparc.
- Le logiciel d'émulation fonctionne comme une application du système hôte, mais *émule* un ordinateur entier d'une autre plate-forme. Le système invité n'a pas conscience de son statut de système invité, ni qu'il fonctionne dans un environnement étranger.

Les Technologies de Virtualisation

- Dans certains cas, l'émulation matérielle peut être lente, mais les technologies récentes, les logiciels d'émulation, les pilotes récents et les processeurs hôtes 64 bits plus rapides rendent l'émulation viable en tant qu'options de virtualisation, en particulier pour ceux qui doivent développer des pilotes ou des technologies pour d'autres plate-formes sans pouvoir investir dans le matériel.

- Les meilleurs exemples de logiciels d'émulation logicielle sont *Bochs* et *QEMU*.

<http://bochs.sourceforge.net/>

<http://wiki.qemu.org/>

Les Technologies de Virtualisation

Bochs

- Bochs est un émulateur libre et gratuit d'architecture Intel x86 qui fonctionne sous Unix, Linux, Windows et Mac OS X, mais qui ne prend en charge que les systèmes x86.
- Il prend en charge un éventail de matériel pour émuler toutes les architectures de processeurs x86. Il prend également en charge les processeurs multiples, mais il ne tire pas complètement avantage du *SMP*.

Qemu

- Qemu est un autre programme libre et gratuit d'émulation qui fonctionne sur un nombre limité d'architectures hôtes (x86, x86_64 et PowerPC) mais qui permet d'émuler des systèmes invités pour x86, x86_64, ARM, Sparc, PowerPC, MIPS et m68k.

Les Technologies de Virtualisation

Microsoft Virtual PC et Virtual Server

- Virtual PC est une solution gratuite de virtualisation de Microsoft. Virtual PC utilise l'émulation pour fournir son environnement de machines virtuelles. C'est une solution pour héberger des machines virtuelles sous Windows XP Workstation ou sous Windows Server. Il ne s'agit pas de solution adaptée pour un large environnement, mais il peut faire fonctionner des machines virtuelles rapidement et pas cher.
- Les performances des machines virtuelles sur ces produits sont correctes pour des machines virtuelles Windows. Il est difficile, sinon impossible de savoir que vous utilisez une machine virtuelle lorsque vous vous connectez au réseau. Les performances de la console sont parfois décevantes. Lorsque c'est possible, minimisez la console et utilisez RDP pour se connecter à vos systèmes Windows virtualisés.

Virtualisation au Niveau Noyau

- La virtualisation au niveau noyau est un cas particulier dans le monde de la virtualisation au sens où chaque machine virtuelle utilise son propre noyau unique pour démarrer la machine virtuelle invitée – appelée système de fichier racine – indépendamment du noyau de l'hôte.

Kvm

- Kvm – *Kernel Virtual Machine* – est un Qemu modifié. À la différence de Qemu, Kvm utilise les extensions de virtualisation du processeur (Intel-VT et AMD-V).
- Kvm prend en charge de nombreux systèmes d'exploitation invités sous x86 et x86_64, y compris Windows, Linux et FreeBSD. Il utilise le noyau Linux comme hyperviseur et fonctionne comme module que l'on peut charger dans le noyau.

Virtualisation au Niveau Noyau

User-Mode Linux

- User-Mode Linux – UML – utilise un noyau exécutable et un système de fichiers racine pour créer une machine virtuelle. Pour créer une machine virtuelle, vous avez besoin d'un noyau exécutable en espace utilisateur – *noyau invité* – et d'un système de fichiers racine créé par UML.
- Ces deux composants forment une machine virtuelle UML. La session de terminal en ligne de commande que vous utilisez pour vous connecter au système hôte distant devient votre console de machine virtuelle. UML est inclus dans tous les noyaux 2.6.x.

Les Technologies de Virtualisation

À Noyau Partagé

- La virtualisation à noyau partagé, aussi appelée virtualisation de système d'exploitation ou virtualisation au niveau système, tire avantage de la possibilité, unique sous Unix et Linux, de partager le noyau avec d'autres processus du système. Cette virtualisation à noyau partagé utilise une fonctionnalité nommée `chroot`, pour changer `root`, ou modifier la racine.
- Cette fonctionnalité modifie le système de fichiers racine d'un processus pour l'isoler de manière à fournir une certaine sécurité. On appelle souvent cela *chroot jail* ou de la virtualisation par *conteneurs*.

Les Technologies de Virtualisation

- Un programme, ensemble de programmes ou système dans le cas de virtualisation à noyau partagé fonctionnant dans un environnement chroot est protégé en faisant croire au système emprisonné qu'il fonctionne sur une machine réelle avec son propre système de fichiers.
- Le mécanisme de chroot a été amélioré pour mimer un système de fichier entier, de sorte qu'un système entier peut fonctionner dans un chroot, ce qui constitue une machine virtuelle.
- Les avantages de la virtualisation à noyau partagé sont :
 - sécurité et isolation accrues ;
 - performances natives ;
 - densité plus élevée de systèmes virtualisés.
- L'inconvénient : toutes les machines virtuelles doivent être compatibles avec le noyau en cours d'exécution.

Les Technologies de Virtualisation

Solaris Containers Zones

- Solaris 10 fournit un système de virtualisation intégré. Le système d'exploitation Solaris 10 est nommé *zone globale*. Les zones de Solaris sont en fait des *jails BSD*, chacune contenant sa racine virtuelle propre qui imite un système d'exploitation et un système de fichiers complet.
- Lorsque vous créez une nouvelle zone, un système de fichier complet est copié dans le répertoire de la nouvelle zone. Chaque zone ne voit que ses propres processus et systèmes de fichiers.

Les Technologies de Virtualisation

- La zone croit qu'elle est un système d'exploitation complet et indépendant ; seule la zone globale a conscience de la virtualisation.
- Chaque zone est, globalement, un bac à sable propre dans lequel vous pouvez installer des applications, fournir des services ou tester des correctifs. Les zones Solaris constituent une solution de virtualisation extensible et de niveau professionnel, facile d'utilisation et aux performances natives.

Les Technologies de Virtualisation

OpenVZ

- Le noyau OpenVZ est optimisé pour la virtualisation et s'avère être efficace pour gérer les performances de machines virtuelles, y compris pour d'autres produits de virtualisation.
- OpenVZ est comparable aux zones de Solaris, à ceci près que vous pouvez faire fonctionner plusieurs distributions Linux avec le même Noyau : <http://openvz.org/>

Introduction à Docker

Présentation de Docker

- Docker est une plateforme ouverte de *développement, livraison, et exécution* d'applications.
- Docker vous permet de séparer vos applications de votre infrastructure afin de pouvoir livrer des applications logiciels rapidement.
- Avec Docker vous pouvez gérer votre infrastructure de la même manière que vos applications. En tirant parti des méthodologies de Docker pour la livraison, le test et le déploiement rapide de code. Le délai entre l'écriture du code et sa mise en production est réduit considérablement.

Introduction à Docker

La plateforme Docker

- Docker permet de *packager* et exécuter une application dans un environnement isolé (librement) nommé *conteneur*.
- L'isolation et la sécurité vous permettent d'exécuter plusieurs conteneurs simultanément sur un hôte donné.
- Les conteneurs sont légers car ils n'ont pas besoin de la charge supplémentaire d'un *hyperviseur*, mais s'exécute directement dans le noyau de la machine hôte.
- Docker vous permet ainsi d'exécuter plus de conteneurs sur un système réel donnée que des machines virtuelles. Vous pouvez même exécuter des conteneurs Docker sur des machines hôtes qui sont des machines virtuelles.

Introduction à Docker

La plateforme Docker

- Docker fournit des outils et une plateforme pour gérer le cycle de vie de vos conteneurs :
 - Développer votre application et ces composants de support en utilisant des conteneurs ;
 - Le conteneur devient l'unité pour la distribution et le test de votre application ;
 - Lorsque vous êtes prêt, déployez votre application dans votre environnement de production, en tant que conteneur ou un service orchestré. Cela fonctionne de la même manière que votre environnement de production soit un centre de données local, un fournisseur de *cloud* ou un hybride des deux.

Introduction à Docker

Docker Engine

- Docker Engine est une application *client-serveur* avec les composants majeurs suivants :
 - Un serveur représenté par la commande `dockerd` et exécuté en tant que processus démon ;
 - Une API REST qui spécifie les interfaces que les programmes peuvent utiliser afin de communiquer avec le démon `dockerd` et lui indiquer les demandes ;
 - Une interface ligne de commande — *Command Line Interface* (CLI) — représentée par la commande `docker`.

Introduction à Docker

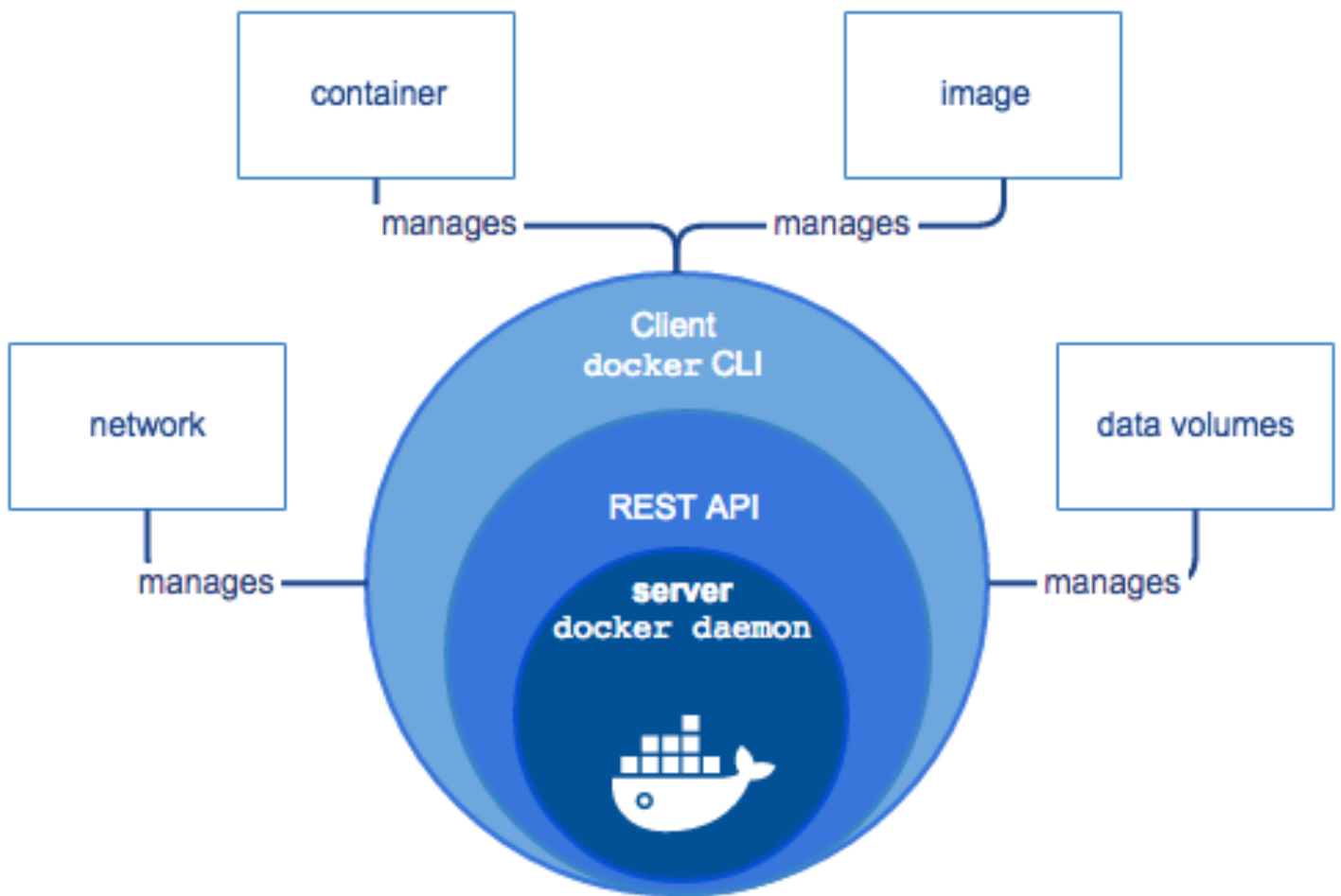


Fig 1 : Les composants de Docker Engine

Introduction à Docker

- L'interface ligne de commande utilise l'API REST Docker pour contrôler ou interagir avec le démon Docker — `dockerd` — par le biais de *scripts* ou des commandes directes. De nombreuses autres applications Docker utilisent les API et CLI sous-jacentes.
- Le démon crée et gère des *objets* Docker, tels que des *images*, des *conteneurs*, des *réseaux*, et des *volumes*.

Introduction à Docker

Qu'est-ce-qu'un conteneur ?

- Un *conteneur* est une *unité logiciel standardisée* qui regroupe *le code et toutes ses dépendances* afin que l'application s'exécute rapidement et de manière fiable d'un environnement à un autre.
- Une *image de conteneur Docker* est un *paquet logiciel léger, autonome et exécutable*, qui inclut tout le nécessaire pour exécuter une application :
 - code ;
 - temps d'exécution? (environnement d'exécution) ;
 - outils système ;
 - bibliothèques systèmes ;
 - paramètres.

Introduction à Docker

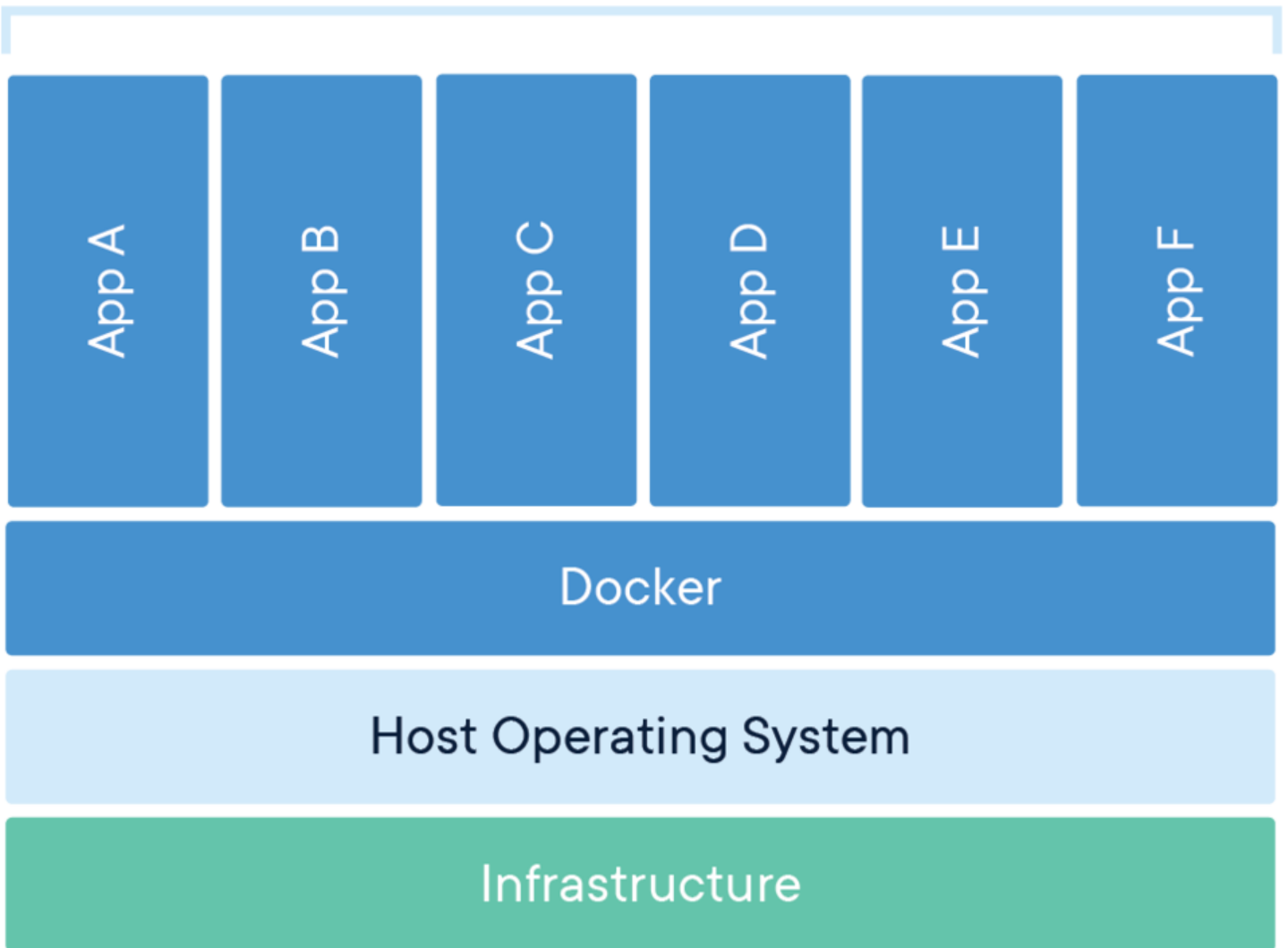


Fig 1 : Conteneurisation d'applications avec Docker

Introduction à Docker

Image & conteneur

- Les images de conteneur deviennent des conteneurs à l'exécution et dans le cas des conteneurs Docker les images deviennent des conteneurs lorsqu'elles s'exécutent sur Docker Engine.

Plateforme applicative

- Disponible pour les applications Linux et Windows, l'application logicielle conteneurisée fonctionnera toujours de la même manière, quelle que soit l'infrastructure.
- Les conteneurs isolent les logiciels de leur environnement et garantissent un fonctionnement uniforme, malgré les différences.

Introduction à Docker

Docker Engine

- Les conteneurs qui fonctionnent sur Docker Engine sont :
 - *Standard*. Docker a créé la norme de conteneurs, de sorte qu'ils puissent être portables ;
 - *Léger*. Les conteneurs partagent le noyau du système d'exploitation de la machine et ne nécessitent donc pas de système d'exploitation par application, ce qui permet d'optimiser l'efficacité du système et/ou du serveur et de réduire les coûts du serveur et de licences ;
 - *Sécurisé*. Les applications sont plus sûres dans les conteneurs et Docker offre les meilleures capacités d'isolation dans le domaine.

Introduction à Docker

Des conteneurs Docker multi-plateformes

- La technologie de conteneur Docker a été lancée en 2013 en tant que projet *Open Source* avec Docker Engine.
- Docker Engine a exploité les concepts existant autour des conteneurs et particulièrement ceux de l'environnement Linux, des mécanismes tels que cgroups et namespaces.
- La technologie de Docker sépare les dépendances des applications de l'infrastructure afin de répondre aux exigences des développeurs et des administrateurs systèmes.

Introduction à Docker

Des conteneurs Docker multi-plateformes

- Des différents environnements utilisent la technologie de conteneur Docker :
 - les conteneurs Windows Docker, le résultat de l'intégration des techniques de conteneur Docker dans un environnement Windows ;
 - les conteneurs dans des environnements *cloud* comme offres *IaaS* natives ;
 - les *frameworks open source*.

Introduction à Docker

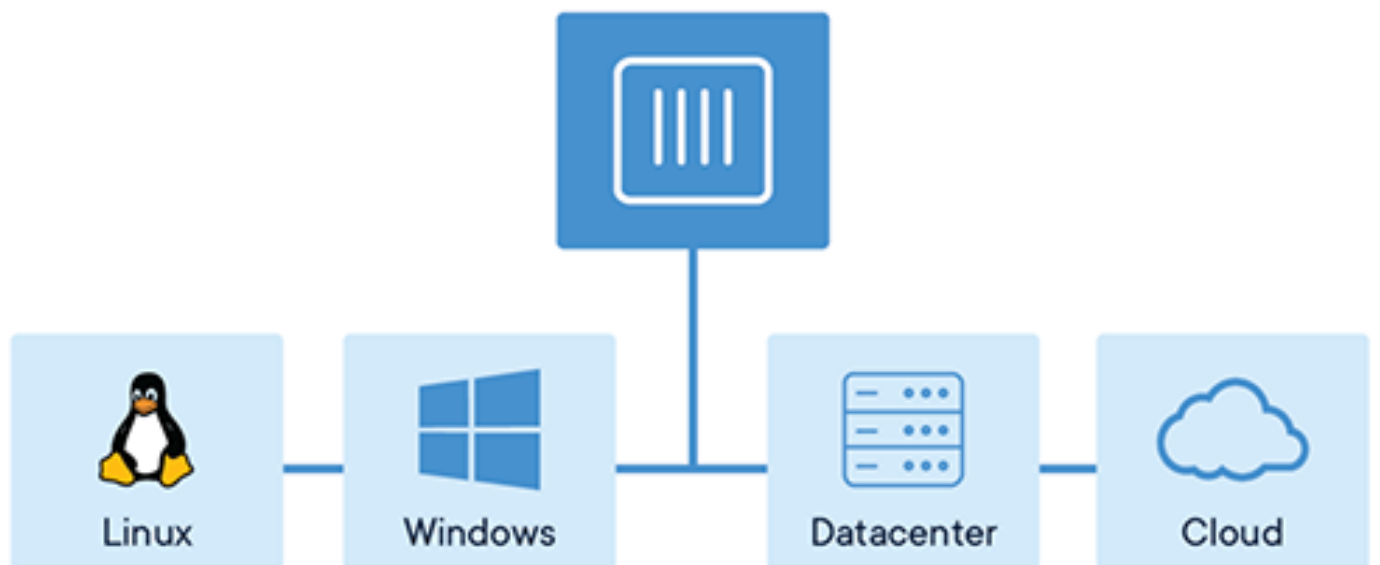


Fig 2 : Conteneur Docker multi-plateformes

Introduction à Docker

Conteneurs & machines virtuelles : comparaison

- Les conteneurs et les machines virtuelles présentent des avantages similaires en termes d'isolation et d'allocation de ressources, mais fonctionnent différemment.
- Les conteneurs virtualisent le système d'exploitation au lieu du matériel.
- Les conteneurs sont plus portables et efficaces.

Introduction à Docker

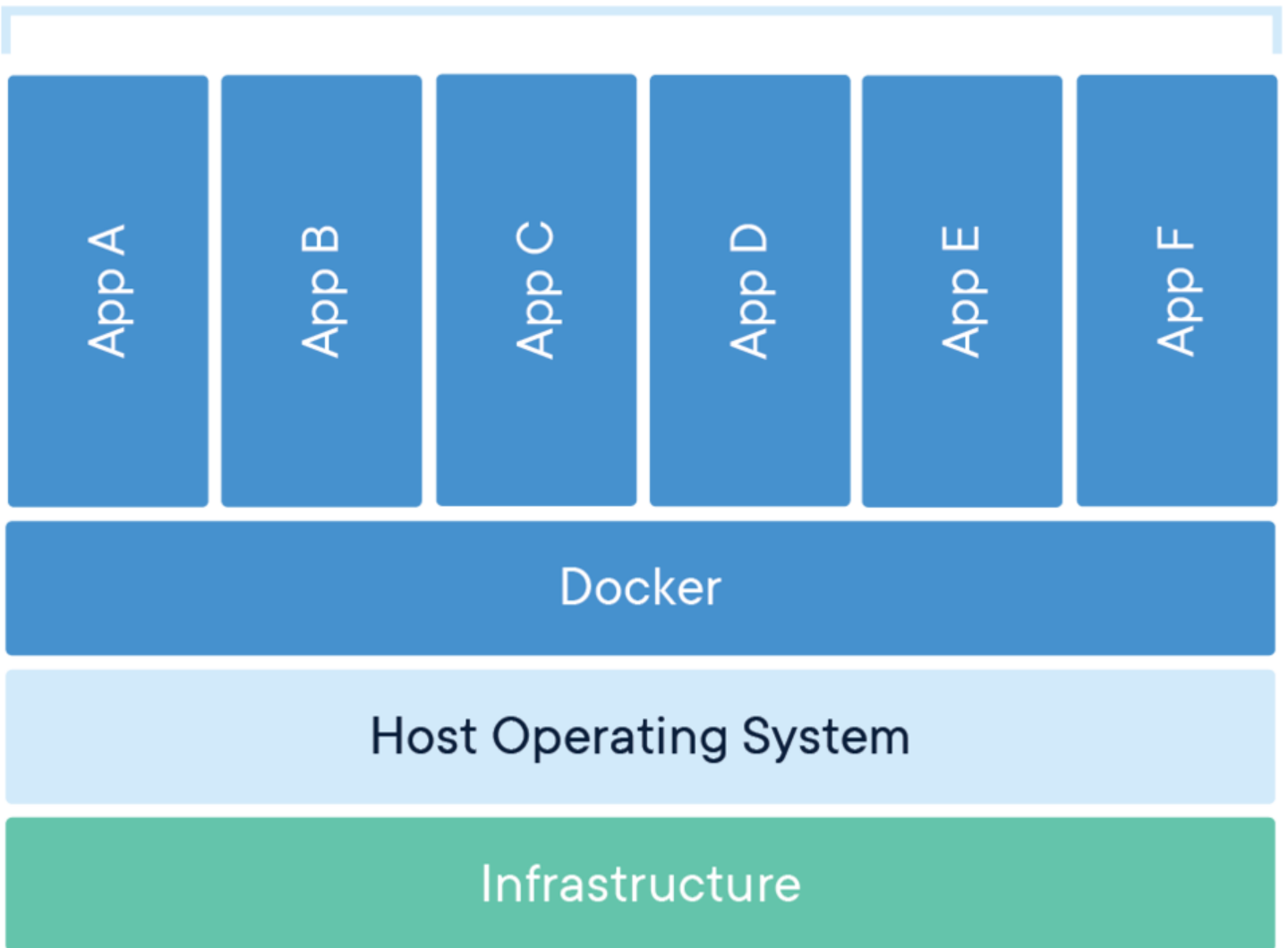


Fig 3 : Conteneurisation avec Docker

Introduction à Docker

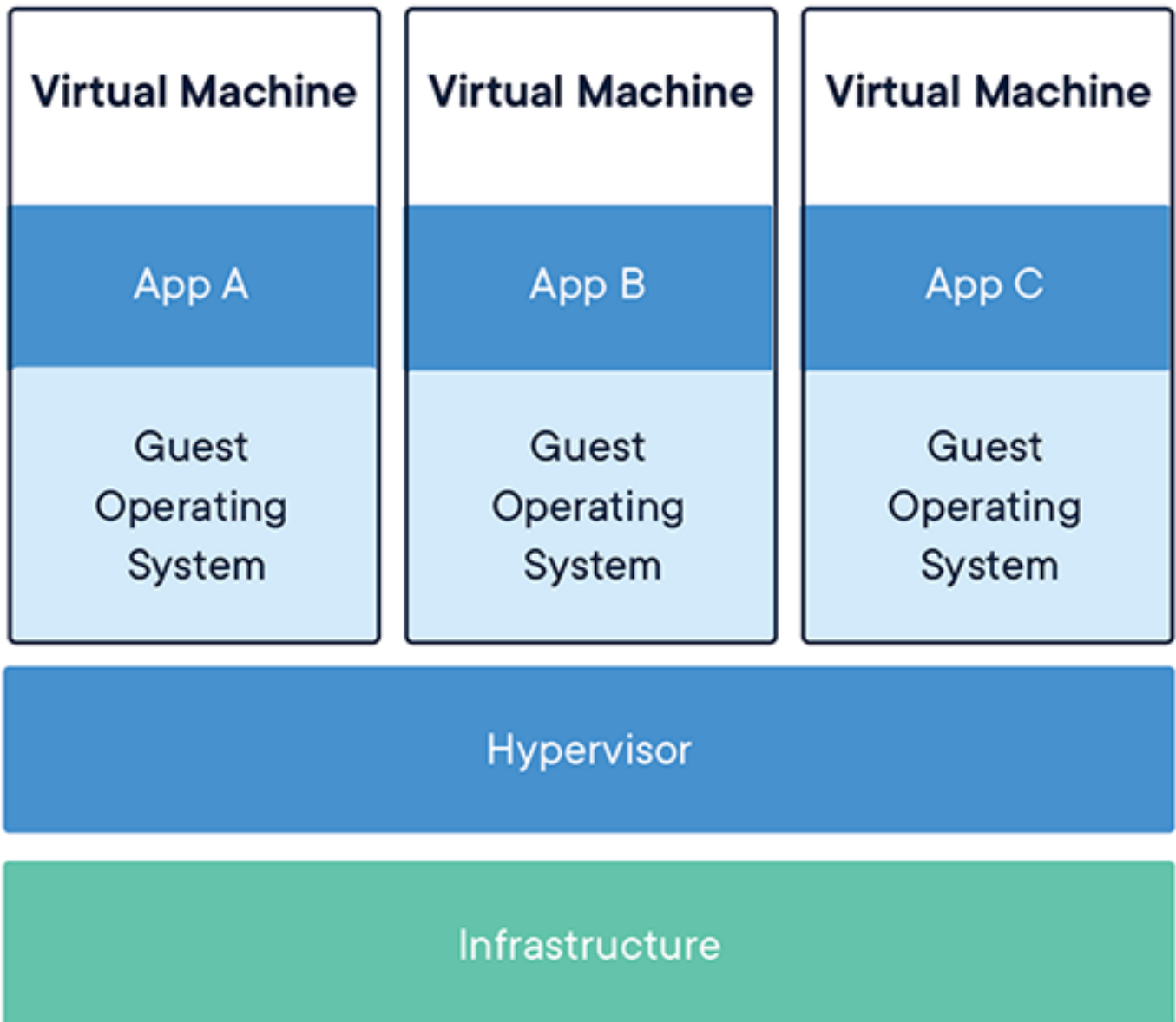


Fig 4 : Machines virtuelles

Introduction à Docker

Les conteneurs

- Les conteneurs sont une abstraction au niveau de la couche d'application qui regroupe le code et les dépendances.
- Plusieurs conteneurs peuvent être exécutés sur la même machine et partager le noyau du système d'exploitation avec d'autres conteneurs. Chaque conteneur s'exécute en tant que *processus isolé* dans l'espace utilisateur.
- Les conteneurs occupent moins d'espace que les machines virtuelles (les images de conteneurs ont généralement une taille de dizaines de Mo), peuvent gérer plus d'applications et nécessitent moins de machines virtuelles et de systèmes d'exploitation.

Introduction à Docker

Les machines virtuelles

- Les machines virtuelles (VM) sont une abstraction du matériel physique transformant un serveur en plusieurs serveurs.
- L'hyperviseur permet à plusieurs machines virtuelles de s'exécuter sur une seule machine.
- Chaque machine virtuelle comprend une copie complète du système d'exploitation, de l'application, des fichiers binaires et des bibliothèques nécessaires, ce qui représente des dizaines de Go.
- Les machines virtuelles peuvent également être lentes à démarrer.

Introduction à Docker

Conteneurs & machines virtuelles

- Les conteneurs et les machines virtuelles utilisés conjointement facilitent le déploiement et la gestion d'applications.
- Les conteneurs et les machines virtuelles offrent une souplesse dans la gestion et le déploiement d'applications.

Introduction à Docker

Normes & standards de conteneur

- Démocratisation de conteneurs logiciel : le lancement de Docker en 2013 a marqué le début d'une révolution dans le *développement d'application*.
- Docker a développé une technologie de conteneur Linux, *portable, flexible* et *facile* à déployer.
- Docker a ouvert l'accès aux sources de `libcontainer` et s'est associé à une communauté mondiale de contributeurs pour poursuivre son développement.
- En juin 2015, Docker a offert la spécification d'image de conteneur et du code d'exécution, nommé actuellement `runc`, à la *Open Container Initiative (OCI)* afin d'aider à la normalisation au fur et à mesure de la croissance et de la maturation de l'écosystème de conteneurs.

Introduction à Docker

- Suite à cette évolution, Docker continue avec le projet containerd, que Docker a fourni à la *Cloud Native Computing Foundation (CNCF)* en 2017.
- containerd est un environnement d'exécution standard qui s'appuie sur runc et a été créé avec un accent mis sur *la simplicité, la robustesse, et la portabilité*.
- containerd est le principal environnement d'exécution de Docker Engine :

| <https://containerd.io/>

Introduction à Docker

Une plateforme moderne pour toutes les applications

- Docker offre aux développeurs et aux services informatiques la possibilité de créer, gérer et sécuriser des applications sans contraintes technologiques ou d'infrastructure.
- En combinant la technologie de gestionnaire de conteneur, Docker permet d'intégrer des applications natives traditionnelles et des applications *cloud* construites sur Windows Server, Linux et autres systèmes dans une chaîne logistique automatisée et sécurisée, afin de faciliter la collaboration entre les développeurs *Dev* et les administrateurs de productions et d'opérations *Ops*.

Introduction à Docker

Une plateforme moderne pour toutes les applications

- Docker augmente la productivité et réduit le temps nécessaire au développement d'applications, ainsi permet de disposer désormais des ressources nécessaires pour développer des projets de numérisation couvrant l'ensemble de la chaîne de valeur.
- La chaîne de valeur est constituée par la modernisation des applications, la migration dans le *cloud* et la consolidation de serveurs.
- Avec Docker, on dispose d'une solution qui aide à gérer les diverses applications, les *cloudset* l'infrastructure d'aujourd'hui, tout en offrant aux projets une voie à suivre pour les applications futures.

Introduction à Docker

Les Besoins

Livraison rapide et cohérente d'applications

- Docker rationalise le cycle de développement en permettant aux développeurs de travailler dans des environnements standardisés en utilisant des conteneurs locaux fournissant vos applications et services.
- Les conteneurs sont adaptés pour l'intégration et la livraison continues — *continuous integration and continuous delivery*, CI/CD.

Introduction à Docker

Les Besoins

Déploiement et calibrage dynamique

- La plateforme de conteneurs Docker permet des charges de travail portable. Les conteneurs peuvent s'exécuter dans un environnement local du développeur, sur des machines physiques, virtuelles, des *cloud* ou dans plusieurs environnements.
- La portabilité et la légèreté de Docker facilitent la gestion dynamique des charges de travail, l'extension ou la suppression des applications et des services en fonction des besoins en temps quasi réel.

Introduction à Docker

Les Besoins

Exécution de plusieurs charges sur le même matériel

- Docker est rapide et léger. Il fournit une alternative aux machines virtuelles basées sur l'hyperviseur. Vous pouvez ainsi utiliser pleinement la capacité de calcul de votre machine.
- Docker est adapté pour les environnements à haute densité et pour les déploiements de petite et moyenne taille où vous devez faire plus avec moins de ressources.

Introduction à Docker

L'architecture Docker

- Docker utilise une architecture *client-serveur*. Le client Docker communique avec le démon Docker, qui se charge de la construction, l'exécution et la distribution des conteneurs.
- Le client et le démon Docker peuvent être exécutés sur le même système, ou vous pouvez connecter un client Docker à un démon Docker distant.
- Le client et le démon Docker communiquent en utilisant une API REST, via des *sockets UNIX* ou une interface réseau.

Introduction à Docker

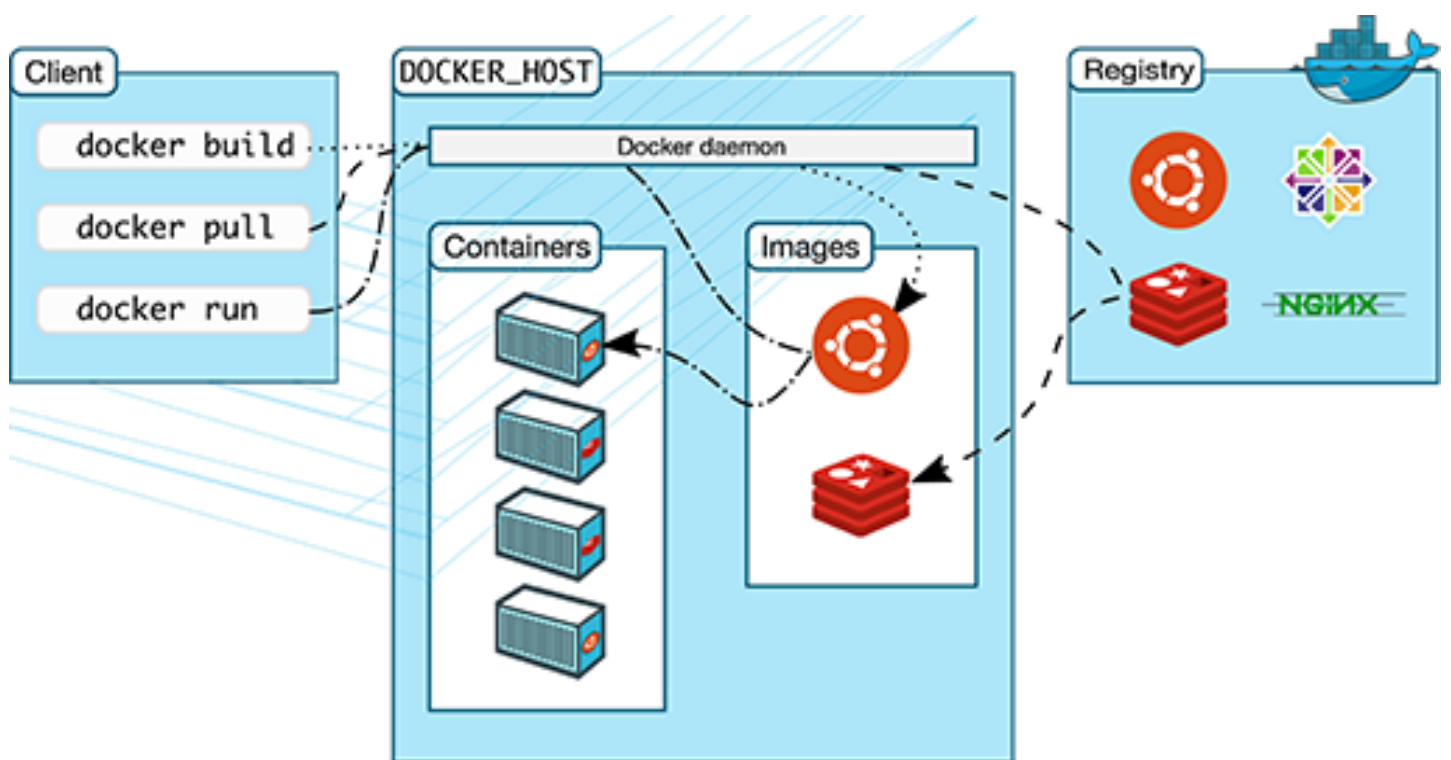


Fig 2 : L'architecture de Docker

Introduction à Docker

Le démon Docker

- Le démon Docker (dockerd) écoute les requêtes de l'API Docker et gère les objets Docker tels que les images, les conteneurs, les réseaux, et les volumes.
- Un démon peut également communiquer avec d'autres démons afin de gérer les services Docker.

Introduction à Docker

Le client Docker

- Le client Docker (`docker`) est le moyen principal utilisé par les utilisateurs Docker afin d'interagir avec Docker.
- Lorsque vous utilisez des commandes telles que :

```
| docker run
```

le client envoie ces commandes au démon `dockerd` qui les exécute.

- La commande `docker` utilise l'API Docker.
- Le client Docker peut communiquer avec plusieurs démons.

Introduction à Docker

Les registres Docker

- Un *registre* Docker stocke des *images* Docker.
- Docker Hub et Docker Cloud sont des *registres publics* que tout le monde peut utiliser, et Docker est configuré pour chercher des images sur le Docker Hub par défaut.
- Il est possible de g'érer son propre registre privé. Si vous utilisez Docker Datacenter (DDC), il inclut Docker Trusted Registry (DTR).

Introduction à Docker

Les registres Docker

- Quand vous utilisez les commandes :

```
| docker pull  
| docker run
```

les images requises sont extraites de votre registre configuré.

- Quand vous utilisez :

```
| docker push
```

votre image est transférée dans le registre configuré.

Introduction à Docker

Docker Store

- Docker Store vous permet d'acheter et de vendre des images Docker ou les distribuer gratuitement.
- Par exemple, vous pouvez acheter une image Docker contenant une application ou un service auprès d'un fournisseur de logiciels et utiliser l'image pour déployer l'application dans vos environnements de test, de stockage et de production.
- Vous pouvez mettre à jour l'application en extrayant la nouvelle version de l'image et redéployer les conteneurs.

| <https://store.docker.com/>

Introduction à Docker

Les objets Docker

- Lorsque vous utilisez Docker, vous créez et utilisez des images, des conteneurs, des réseaux, des volumes, des *plugins* et autres objets tels que :
 - les images, représentant un modèle en lecture seule avec des instructions pour créer un conteneur Docker ;
 - les conteneurs, représentant une instance exécutable d'une image ;
 - les services, permettant de redimensionner les conteneurs sur plusieurs démons Docker, qui fonctionnent tous ensemble en groupe (*swarm*) avec plusieurs gestionnaires et conteneurs (*travailleurs*).

Installation de Docker

Docker sur Linux

- **Prérequis** Sur les versions récentes, la plupart des prérequis logiciels font partie des distributions. Cependant, les deux points principaux à prendre en compte sont un noyau récent (3.10) minimum et une installation 64 bits.

- **Installation par gestionnaire de paquets**

La méthode la plus simple consiste à utiliser le gestionnaire de paquets Aptitude pour installer Docker, ce dernier étant désormais incorporé dans certaines distributions.

- **Documentation en ligne**

<https://docs.docker.com/install/>

Installation de Docker

Docker sur Debian

- **Debian Jessie/Wheezy**

```
uname -r  
uname -m  
lsb_release -a  
getconf LONG_BIT
```

- **Les paquets supplémentaires**

```
sudo apt-get update  
sudo apt-get install curl
```

Installation de Docker

Docker sur Debian

- **Installation par script** Il est possible d'installer Docker à partir d'un script fourni par Docker en fonction de la distribution qu'il détecte. Ce script est obtenu depuis :

`https://get.docker.com`

- Il est possible possible de télécharger le script depuis un navigateur et le sauvegarder dans un fichier ou bien le récupérer en ligne de commande :

```
| wget -O dockerinstall.sh https://get.docker.com
```

- Il est instructif de consulter les opérations d'installation réalisées par le script, et constitue également une bonne habitude à prendre lorsqu'on exécute un fichier téléchargé depuis l'Internet, pour des raisons de sécurité.

Démarrage d'un Conteneur

Hello World, Docker

Validation de l'installation Un exemple de conteneur qui affiche un message de bienvenue. L'intérêt de ce conteneur est de valider l'installation de Docker et de vérifier que l'ensemble de la chaîne logicielle fonctionne.

- **Démarrer un conteneur Hello World** implique un certain nombre d'opérations afin d'afficher le résultat :

```
| sudo docker run hello-world
```

- **Récupération de l'image** Pour commencer, Docker a besoin de lire l'image nommée `hello-world` qui lui a été passée en paramètre de la commande `run`.

Démarrage d'un Conteneur

- **Docker dispose d'un dépôt d'images** auxquelles il peut accéder par Internet. Ce dépôt, connu sous le nom de *registre*, regroupe toutes les images *officielles* fournies par Docker, et accessible par l'URL:

<https://registry.hub.docker.com>

- Il est possible de stocker vos propres images si vous souhaitez les partager, ou de créer des *dépôts privés*.

Démarrage d'un Conteneur

Hello World, Docker

- Constatant que l'image n'était pas déjà présente sur la machine hôte, Docker l'a téléchargé. Il est toutefois possible de récupérer l'image au préalable.

- **Télécharger une image sans l'exécuter**

```
| sudo docker pull hello-world
```

- Dans le cas de l'image hello-world, dont la taille est de moins 1Ko pour certaine versions, la différence de temps est infime, mais pour des images volumineuses, il peut être utile de séparer le téléchargement et le lancement.

- Docker gère un cache des images sur la machine, et le prochain démarrage d'une instance de conteneur sur la même image n'aboutira donc pas nécessairement au rechargement de celle-ci, sauf si elle a été modifiée entre temps.

Démarrage d'un Conteneur

Anatomie de l'image hello-world

- Une recherche sur le registre Docker permet de localiser la page correspondant à l'image en ligne, à savoir depuis :

`https://hub.docker.com`

- Il est aussi possible de passer par un nouveau service nommé Docker Store qui permet de localiser l'image :

`https://store.docker.com`

Démarrage d'un Conteneur

Anatomie de l'image hello-world

- La page fait référence à différentes caractéristiques de l'image, et en particulier à un fichier qui se nomme `Dockerfile`. Ce fichier correspond à la définition textuelle du contenu de l'image. En suivant le lien vers ce fichier, on accède à sa version actuelle, qui est stockée sur le dépôt de code source GitHub :

<https://github.com/docker-library/hello-world/...>

Démarrage d'un Conteneur

Anatomie de l'image hello-world

- Le contenu du fichier Dockerfile :

```
1 | FROM scratch
2 | COPY hello /
3 | CMD ["/hello"]
```

- La ligne 1 définit sur quelle image la présente image va se baser. Dans ce cas, le mot clé `scratch` correspond à une machine vide
- La ligne 2 fait en sorte que, lorsque le fichier `Dockerfile` sera lu pour générer l'image, le contenu du fichier `hello` sera ajouté dans la racine du système de fichiers de l'image
- Enfin, la ligne 3 établit que le démarrage de l'image provoquera le lancement de la commande `/hello`, qui exécutera le contenu du fichier précédemment copié.

Démarrage d'un Conteneur

Anatomie de l'image hello-world

- En effet, le fichier `Dockerfile` doit être transformé en une image utilisable (grâce à la commande `build`), mais pour cet exemple de simple lancement d'un conteneur à partir d'une image préexistante, cette opération de compilation a déjà été réalisée en amont par Docker, et le registre nous expose une image déjà compilé.
- En remontant d'un niveau dans l'arborescence du projet, nous retrouvons le fichier `hello`. Il s'agit d'un *fichier binaire* généré par le `Makefile`, celui-ci étant en charge de produire un fichier exécutable à partir du code assembleur stocké dans le fichier `hello.asm`.
- Pour information, ce choix d'utiliser du code assembleur est dans le but de produire une image réduite tout en étant compatible avec n'importe quel système Linux 64 bits.

Démarrage d'un Conteneur

Lancement du processus

- Une fois que l'image a été fournie puis récupérée dans le cache local, la commande `run` demande à Docker de créer un conteneur, autrement un espace étanche dans le système de la machine hôte, avec ses propres espace de nommage pour la gestion des processus, des entrées/sorties, du réseau et autres ressources.
- L'instanciation se poursuit avec le montage d'un système de fichier en couches, de façon que le conteneur final dispose des fichiers contenus dans l'image, mais en ajoutant également une dernière couche qui est la seule en écriture.
- Si des modifications sont apportées dans le système de fichiers par le processus hébergé dans le conteneur, c'est cette couche qui sera modifiée en fonction.

Démarrage d'un Conteneur

Lancement du processus

- Dans le cas de l'image Hello-World, aucune écriture n'est réalisée, mais la couche existe.
- La couche `scratch` est en fait inexistante, car il s'agit d'un identifiant particulier précisant que l'image ne se base sur aucune autre image existante.
- Docker procède ensuite à la création d'une interface réseau qui permettra la communication entre le conteneur et la machine hôte, il place le conteneur dans une configuration réseau ainsi que dans le bon système de fichiers en utilisant la commande `chroot` et peut passer à la dernière étape, à savoir le lancement du processus proprement dit.

Démarrage d'un Conteneur

Lancement du processus

- Le processus démarré est celui qui avait été prévu dans l'image par la commande `CMD` du fichier `Dockerfile`, dans ce cas `/hello`.
- Le conteneur va donc exécuter cette commande, qui déclenchera le fichier exécutable en charge d'afficher le texte de bienvenue contenu dans le fichier `hello.asm`. La redirection des entrées/sorties de console permettra à la machine hôte de manipuler plus facilement le processus isolé.

Arrêt d'un Conteneur

Lister les conteneurs

- L'image `hello-world` contient un processus prévu pour réaliser une tâche – en l'occurrence afficher un message de bienvenue – puis s'arrêter.
- Il est possible de retrouver la trace des conteneurs exécutés même après que ceux-ci se soient arrêtés.
- Les commandes suivantes permettent d'afficher la liste des conteneurs en cours ainsi que tous les conteneurs respectivement:

```
sudo docker ps
sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED
94acebe02ea8	hello-world	"/hello"	10 seconds ago
STATUS	PORTS	NAMES	
Exited (0) 9 seconds ago		awesome_beaver	

Les Conteneurs

Lister les conteneurs

- **CONTAINER ID** (94acebe02ea8): Représente l'identifiant unique du conteneur. Docker crée automatiquement un nouvel identifiant à chaque lancement d'un conteneur, même s'ils instancient la même image. Cet identifiant peut être fourni, y compris dans sa forme réduite (les quatre ou cinq premiers caractères suffisent en général), lorsqu'on souhaite mettre en pause ou arrêter un conteneur.
- **IMAGE** (hello-world:latest): Il s'agit du nom de l'image utilisée pour instancier le conteneur. Le suffixe :latest indique qu'il s'agit de la dernière version disponible. Il est possible de spécifier une version particulière d'une image. Par défaut la plus récente est utilisée.

Les Conteneurs

Lister les conteneurs

- **COMMAND** (`"/hello"`): Cette information reprend le contenu de la ligne CMD du fichier Dockerfile, dans la commande lançant le processus dans le conteneur.
- **CREATED** (`10 seconds ago`): Donne la date d'initialisation du conteneur.
- **STATUS** (`Exited(0) 9 seconds ago`): le statut du conteneur contient plusieurs informations. Dans ce cas, le conteneur s'est arrêté, et le processus interne est sorti avec le code 0, qui correspond par convention à un déroulement sans erreur.
- **PORTS**: Les ports réseau exposés par le conteneur, en l'occurrence aucune dans le cas de cet exemple n'affichant que du texte.

Les Conteneurs

Lister les conteneurs

- `NAMES (awesome_beaver)`: Il est possible d'affecter un nom à un conteneur lors de son lancement, pour pouvoir les manipuler plus facilement que par leur identifiant. En l'absence de nom spécifié, `Docker` crée un nom à partir d'une combinaison de noms propres et d'adjectifs.
- Un autre signe de l'exécution du conteneur est que l'image `hello-world` a été téléchargée, et se trouve donc dans la liste des images présentes sur la machine hôte.

Les Conteneurs

Lister les images

```
| sudo docker images
```

- Le résultat fourni les informations suivantes:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	48b5124b2768	2 weeks ago	1.84 kB

- REPOSITORY (hello-world): le nom de l'image dans le dépôt.
- TAG (latest): sa version, qui peut être un nombre ou une dénomination textuelle. Ces *tags* peuvent être multiples par rapport à une même image.
- IMAGE ID (48b5124b2768): un identifiant unique pour l'image

Les Conteneurs

Lister les images

- `CREATED (2 weeks ago)`: la date de création de l'image dans la machine hôte Docker. Il s'agit de la date de création dans le cache local et non pas celle par le dépôt.
- `SIZE (1.84 kB)`: la taille de l'image. Il s'agit d'une taille virtuelle représentant toutes les couches composant successivement l'image. La taille réellement occupé sur le disque dur par une image est donc souvent inférieur.

Lancement d'un Conteneur Interactif

Récupération d'une image dans sa dernière version

```
| sudo docker pull ubuntu:latest
```

- Le chargement d'image a lieu une fois pour toutes, et tout lancement suivant se basera sur cette image, y compris d'ailleurs – et c'est l'avantage de l'architecture par couche – toutes les images se basant sur celle-ci. Le client Docker renvoie des informations sur le téléchargement en cours comme suit une fois finalisé :

```
latest: Pulling from library/ubuntu
8aec416115fd: Pull complete
695f074e24e3: Pull complete
946d6c48c2a7: Pull complete
bc7277e579f0: Pull complete
2508cbcade94b: Pull complete
Digest: sha256:71cd81252a3563a03ad8daee81047b62ab5d892ebbfbf
Status: Downloaded newer image for ubuntu:latest
```

Lancement d'un Conteneur Interactif

Récupération d'une image dans sa dernière version

- Le téléchargement réalisé, une commande permet de voir les images disponibles localement:

```
| sudo docker images
```

- Le résultat correspond aux images comme suit:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	f49eec89601e	8 days ago	129 MB
hello-world	latest	48b5124b2768	2 weeks ago	1.84 kB

Lancement d'un Conteneur Interactif

Présentation des TAGS

- Le *tag* associé à l'image `ubuntu` est `latest`, comme précisé dans la commande `pull`. Le *tag* est un marqueur de version de l'image. Toutefois, contrairement à un système de version, il est possible d'avoir plusieurs *tags* pour une même version.

- **Le lancement du conteneur** sans option peut être réalisé par la ligne de commande suivante:

```
| docker run ubuntu
```

- Sans l'utilisation d'un *tag* après le nom de l'image implique à la commande `run` qu'il s'agit de la version `latest` par défaut.

Lancement d'un Conteneur Interactif

Liste des conteneurs

- Pour lister les conteneurs actifs et inactifs respectivement :

```
| docker ps
```

```
| docker ps -a
```

- Le résultat indique qu'aucun conteneur est actif (la commande `ps` sans option). Pour la seconde commande, le résultat correspond à tous les conteneurs y compris ceux qui ne sont pas actifs.

- Le `STATUS` indique que le processus s'est arrêté sans erreur et la `COMMAND` correspond à `/bin/bash`.

Lancement d'un Conteneur Interactif

- Le contenu du fichier `Dockerfile` nous indique que le processus lancé correspond par défaut est `/bin/bash`.

| `https://hub.docker.com/\_/ubuntu/`

- Sans console associée – entrées/sorties – le processus `shell` s'arrête automatiquement. Donc ce conteneur n'a aucune utilité.

- **Suppression d'un conteneur**

| `docker ps -a`
| `docker rm nom_conteneur`

Lancement d'un Conteneur Interactif

- **Pour exploiter le conteneur ubuntu**, et interagir avec le processus `shell` lancé, nous allons exécuter le conteneur en mode interactif, en lui associant une console TTY pour assurer les entrées/sorties.

```
| docker run -i -t ubuntu
```

- Dans ce cas Docker rend la main mais à travers l'invite de commande du `shell` du système `ubuntu` du conteneur.
- L'utilisateur n'est plus celui du système hôte, mais l'utilisateur `root` de la machine nommé à travers l'identifiant du conteneur. Comme toute machine qui serait effectivement distincte, nous pouvons appeler toute commande par le `shell`.

Lancement d'un Conteneur Interactif

Les modifications par rapport à son image

- Pour trouver la liste des modifications d'un conteneur par rapport à son image de lancement:

```
| docker diff identifiant_conteneur | nom_conteneur
```

- D pour la suppression, C pour la création de répertoire, A pour ajout d'un fichier.
- Ces modifications concernent unique l'image du conteneur et non l'image téléchargée.
- Afin de garder l'état d'un conteneur en une nouvelle image:

```
| docker commit identifiant_conteneur | nom_conteneur
```

Manipulation des Conteneurs

Supprimer une image

```
| docker rmi nom_image
```

Suppression automatique à la sortie

```
| docker run rmi hello-world
```

Affectation d'un nom de conteneur

```
| docker run --name=nom_conteneur_choisi image_instanciée
```

Modification du point d'entrée par défaut

```
| docker run --rm ubuntu env
```

Ajout de variable d'environnement

```
| docker run --rm -e HOSTNAME=dada.com ubuntu env
```

Manipulation des Conteneurs

Lancement en mode bloquant

- Ce mode de fonctionnement est adapté aux images qui fournissent le `shell` comme point d'entrée par défaut par exemple, ou n'importe quel processus avec une durée limitée à son usage immédiat telle que l'exécution d'une commande.

Lancement en arrière plan

```
docker run -i -t -p 88:80 nginx  
docker run -d -p 88:80 nginx
```

Gestion de cycle de vie des conteneurs

```
docker stop nom_conteneur | identifiant  
docker start nom_conteneur | identifiant  
docker restart nom_conteneur | identifiant  
docker log nom_conteneur | identifiant  
docker top nom_conteneur | identifiant
```

Manipulation des Conteneurs

Exposition de fichiers

- **Remplacer une partie d'arborescence** du conteneur par un emplacement hôte est possible par l'option de gestion de volume `-v` comme suit:

```
| docker run -d -p 88:80 --name web \  
| -v /var/www/:/usr/share/nginx/html:ro nginx
```

- Il s'agit de mettre en correspondance le répertoire `/usr/share/nginx/html` – à savoir la racine par défaut de `nginx` – avec `/var/www/` de la machine hôte.
- Le fonctionnement par défaut de l'image a été modifié sans la création d'une nouvelle image.
- L'accès au répertoire de la machine hôte est autorisé uniquement en lecture seule (`:ro`).

Création d'Images avec Docker

Création manuelle d'une nouvelle image

- L'approche la plus simple pour créer une nouvelle image est d'utiliser la commande `commit` pour persister l'état d'un conteneur sous forme d'une nouvelle image, après avoir ajouté à celui-ci les modifications nécessaires.
- **Une nouvelle image** à partir d'une image `ubuntu` comme base, en installant un serveur base de données non relationnelle `MongoDB` dans le conteneur. Ensuite, persister le conteneur sous forme d'une image réutilisable.

Création d'Images avec Docker

Installation d'un serveur MongoDB

- Les procédures d'installation d'un serveur MongoDB sur ubuntu sont décrites dans la documentation suivante :

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/>

- Les procédures :

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80\
--recv 0C49F3730359A14518585931BC711F9BA15703C6
echo "deb [ arch=amd64 ] http://repo.mongodb.org/apt/ubuntu\
precise/mongodb-org/3.4 multiverse" | \
sudo tee /etc/apt/sources.list.d/mongodb-org-3.4.list
sudo apt-get update
sudo apt-get install -y mongodb-org
mongod --config /etc/mongod.conf &
ps
```

Création d'Images avec Docker

Installation d'un serveur MongoDB

- L'appel à la commande `ps` permet de vérifier que le démon `mongod` est bien lancé. La commande précédente permet de lancer le démon en pointant sur le fichier de configuration par défaut.
- Une fois exécuté, il sera possible d'utiliser un client `mongo`, également installé par la procédure d'installation, pour se connecter au serveur, insérer une donnée et lancer une requête, de façon à valider l'installation.

Création d'Images avec Docker

Installation d'un serveur MongoDB

```
# mongo
...
> use testdb
switched to db testdb
> db.utilisateurs.insert( { nom : "test", prenom : "reussi" } )
WriteResult({ "nInserted" : 1 })
> db.utilisateurs.find()
{ "_id" : ObjectId("589758a1ad9a417fe7052d08"), "nom" : "test",
> db.utilisateurs.drop()
true
> exit
# exit
```

- Les appels à la commande `exit` respectivement permet de sortir du client `mongo` et de sortir du conteneur, qui arrêtera le `shell` hébergé.

Création d'Images avec Docker

Persistance de l'image

- Afin de créer une image à partir du conteneur et l'utiliser dans la suite, il faut l'identifier par la commande `ps` de Docker.
- La commande `commit` est ensuite utilisée pour créer une image, et la validation de sa création sera faite par appel à la commande `images`.

```
sudo docker ps -a  
sudo docker commit nom_conteneur doclab/mongodb  
sudo docker images
```

Création d'Images avec Docker

Utilisation de l'image créée

- L'image nommée `doclab/mongodb` va être utilisée pour lancer un nouveau conteneur – le précédent, qui nous a servi à créer l'image, peut être supprimé.
- Afin d'identifier ce nouveau conteneur, il sera nommé `mongo`. Il sera démarré en mode interactif afin de vérifier que le serveur MongoDB est bien exécuté:

```
$ sudo docker run -it --name mongo doclab/mongodb
# ls /usr/bin/mongo*
# ps
#
```

Création d'Images avec Docker

Utilisation de l'image créée

- L'exécutable `mongod` existe dans `/usr/bin/`, mais le processus démon n'est pas démarré comme il était lorsque nous avons arrêté le précédent conteneur.
- Le comportement est bien attendu, à savoir que toutes les modifications sur le disque ont été reprises dans l'image, mais pour ce qui est du processus, un nouveau conteneur exécute un nouveau processus.
- Dans notre cas, le processus correspond à `/bin/bash` qui est la commande par défaut de notre nouvelle image. Lorsque le conteneur est démarré, nous prenons donc bien la main en mode interactif.

Création d'Images avec Docker

Utilisation de l'image créée

- Cependant, pour que le serveur MongoDB soit accessible, nous devons exécuter à nouveau la commande `mongod --config /etc/mongod.conf`.
- Ce mode de fonctionnement n'est pas recommandable : une procédure de déploiement doit être automatique.
- Une façon de faire serait de rajouter la commande dans le fichier `/etc/bash.bashrc` par exemple comme suit:

```
echo "/usr/bin/mongod --config /etc/mongod.conf &">>/etc/bash.bashrc
tail /etc/bash.bashrc
```

Création d'Images avec Docker

Utilisation de l'image créée

- Pour tester la nouvelle configuration, il faut relancer le `shell`, donc un conteneur sans oublier de valider les modifications dans l'image, sous peine de ne voir aucun changement.

```
# exit
$ sudo docker commit mongo doclab/mongodb
$ sudo docker rm -f mango
$ sudo docker run -it --name mango doclab/mongodb
# ps
```

- Ainsi, lorsque le conteneur `mongo` est exécuté, le processus démon est bien lancé.

Création d'Images avec Docker

Connexion depuis la machine hôte

- Afin d'interagir avec la base MongoDB, il convient de le faire depuis l'extérieur du conteneur, par exemple depuis la machine hôte.
- Il s'agit de configurer le serveur mongod en modifiant le fichier `/etc/mongod.conf` en commentant la variable `bindIP=127.0.0.1`.
- Une fois le conteneur arrêté, reprendre la commande `commit` que précédemment. Ensuite, supprimer le conteneur avec la commande:
| `sudo docker rm mongo`
- Relancer un nouveau conteneur en mode interactif avec la commande:
| `sudo docker run -it doclab/mongodb`

Création d'Images avec Docker

Connexion depuis la machine hôte

- L'étape suivante consiste à utiliser un client MongoDB sur la machine depuis laquelle nous souhaitons nous connecter, par exemple la machine hôte.

- Enfin, avant la connexion il s'agit de retrouver l'adresse IP du conteneur Docker, comme suit:

```
| sudo docker inspect mongo | grep IP
```

- La connexion sera établie ainsi:

```
| $ mongo --host IP
```

Création d'Images avec Docker

Résultat

- Le mode de fonctionnement mis en œuvre n'est pas totalement satisfaisant. L'exécution du `client` montre des avertissements sur le fait que le processus serveur est lancé avec l'utilisateur `root`, ce qui est à éviter en pratique de point de vue de la sécurité.
- Ensuite, le conteneur est lancé en mode interactif, ce qui n'est pas logique vu le type d'utilisation avec un serveur de base de données. L'arrêt du `shell` implique l'arrêt du conteneur et ainsi le serveur.

Création d'Images avec Docker

Résultat

- Cette configuration peut être améliorée avec la création d'un utilisateur dédié pour le serveur MongoDB, et donc nécessite plus de temps dans la documentation et les tests de mises en place.
- Le résultat final n'est peut-être pas garanti en comparant une mise en œuvre d'un spécialiste de cette base de données.
- Cependant, cette mise en œuvre propre existe déjà, sous la forme d'une image Docker officiel pour MongoDB.
- Cette image fournit les instructions à utiliser sous forme d'un fichier contenant toutes les commandes nécessaires à une installation robuste.

Utilisation d'un Dockerfile

Intérêt des fichiers Dockerfile

- Le fichier `Dockerfile` contient toutes les opérations nécessaires à la préparation d'une image Docker.
- Ainsi, au lieu de construire une image manuellement, il s'agit de compiler une image depuis une description textuelle de ces opérations.
- Considérant le contenu du fichier `Dockerfile` correspondant à l'image officielle MongoDB
| https://hub.docker.com/r/_/mongo/
- En comparaison à une installation manuelle, la mise en place d'un utilisateur et d'un groupe dédiés a été utilisée.

Utilisation d'un Dockerfile

Intérêt des fichiers Dockerfile

- L'affectation des droits correspondants sur les répertoires pour lesquels cette opération est nécessaire.
- L'utilisation d'un ENTRYPOINT dédié, qui permet de spécifier le `script` à exécuter lors du démarrage du conteneur.
- La mise en œuvre d'un processus par défaut différent du shell.

Utilisation d'un Dockerfile

- Bien que, dans le cas d'une image officielle, le fichier `Dockerfile` serve au producteur pour réaliser l'image et montrer publiquement la façon dont celle-ci a été réalisée.

- Le lien entre les deux étant assuré en particulier par le fait que c'est Docker Hub qui s'occupe de compiler les images, il est possible d'utiliser ce fichier pour créer une image.

- Les procédures à suivre :

- Sur la machine hôte, création d'un répertoire nommé `mongodb`
- Se placer dans ce répertoire.
- Recopier les deux fichiers `Dockerfile` et `docker-entrypoint.sh`
- Affecter les droits de lecture sur les deux fichiers.
- Affecter les droits d'exécution sur `docker-entrypoint.sh`.

- **La commande de compilation de l'image**
depuis le répertoire `mongodb`

- | `sudo docker build .`

Utilisation d'un Dockerfile

- L'image créée n'a pas de nom explicite

```
sudo docker images  
sudo docker tag image_ID doclab/mongodb  
sudo docker images
```

- L'image créée est identique à l'image officielle que nous aurions pu télécharger par le registre Docker Hub.

- Un appel à `docker ps` montre que le conteneur est actif, avec le processus serveur à l'écoute.

```
sudo docker run -d --name mongo doclab/mongodb  
sudo docker ps  
sudo docker inspect mongo | grep IP  
mongo --host IP
```

Anatomie d'un Dockerfile

- **FROM** Les fichiers Dockerfile commencent toujours par la commande FROM, qui permet de définir une image de base par-dessus laquelle la nouvelle image sera construite. Dans ce cas, il s'agit d'une Debian, version Wheezy.

```
| FROM debian:wheezy
```

- Si cette image n'est pas présente sur la machine hôte, elle sera téléchargée lors de l'exécution de la commande build. Dans le cas où une image est construite sans partir d'une image existante : FROM scratch.

- **RUN** La commande est une des plus utilisées dans les Dockerfile. Elle permet d'exécuter n'importe quel processus qui va participer à l'élaboration de l'image en cours de compilation.

```
| RUN ["exécutable", "paramètre1", ..., "paramètren"]
```

Anatomie d'un Dockerfile

- **ENV** permet de déterminer des variables d'environnement, qui seront disponibles non seulement pour le reste de l'opération de compilation du Dockerfile, mais aussi dans les conteneurs qui seront exécutés depuis l'image générée.

```
ENV MONGO_MAJOR 3.0  
ENV MONGO_VERSION 3.0.1
```

- Cette commande correspond aux variables d'environnement d'un conteneur:

```
sodu docker inspect mongo | head -22
```

- Il est aussi possible de passer des variables d'environnement lors de l'exécution d'un conteneur avec l'option `--env` de la commande `docker run`.

- **VOLUME** Dans le cas d'un serveur de base de données comme MongoDB, il est important que les fichiers de données soient détachés du conteneur de base de données lui-même.

```
sudo docker run -d --name mongo doclab/mongodb
sudo docker inspect mongo | grep IP
...
> use testdb
switched to db testdb
> db.utilisateurs.insert( { nom : "test", prenom : "reussi" } )
WriteResult({ "nInserted" : 1 })
> db.utilisateurs.find()
{ "_id" : ObjectId("589758a1ad9a417fe7052d08"), "nom" : "test",
> exit
sudo docker run -d --name mongo doclab/mongodb
sudo docker inspect mongo | grep IP
...
> use testdb
switched to db testdb
> db.utilisateurs.find()
> exit
sudo docker rm -fv mongo
mkdir ~/donnees
sudo docker run -d --name mongo -v ~/donnees:/data/db doclab/mongo
... sudo docker rm -fv mongo
sudo docker run -d --name mongo -v ~/donnees:/data/db doclab/mongo
```

Anatomie d'un Dockerfile

- **COPY** La commande `COPY` permet de récupérer des fichiers locaux à la machine sur laquelle la commande `docker build` est lancée, et de les intégrer dans l'image en cours de création.
- Dans notre exemple, le fichier `docker-entrypoint.sh` se trouvait au même endroit que le fichier `Dockerfile` livré par les mainteneurs de l'image officielle de MongoDB.
- Comme ce fichier est nécessaire à l'exécution du conteneur, la commande `COPY` le recopie dans la racine de l'image compilée.
- En pratique, les fichiers se trouvant au même répertoire que `Dockerfile` sont copiés dans ce qu'on appelle le *contexte*, et donc chargés par la commande `build`.

Anatomie d'un Dockerfile

- **ENTRYPOINT** Ce mot clé permet de définir la commande qui va être exécutée lors du démarrage d'un conteneur, autrement le processus primaire qui sera porté par ce dernier.
- Une machine *Non Uniform Memory Access* – l'architecture NUMA permet la mise en commun de ressources mémoire entre plusieurs machine.
- **EXPOSE** permet d'exposer un port réseau à l'extérieur, un peu comme le fait **VOLUME** avec les systèmes de fichiers.
- **CMD** permet de passer au processus spécifié par **ENTRYPOINT** les paramètres par défaut, en l'absence de paramètres explicitement spécifiés en fin de commande `docker run`

Un exemple de Dockerfile

- Il s'agit d'une image qui une fois instanciée sous forme de conteneur, émettra des signaux textuels sur la console de sortie à intervalles réguliers.
- Le message sera paramétrable au démarrage du conteneur, avec une valeur par défaut portée par CMD.
- ENTRYPOINT sera utilisé pour que l'émission de message puisse être arrêtée, puis reprise.

Un exemple de Dockerfile

Le script nommé `percussion.sh`

```
#!/bin/bash
if [ -z "$RYTHMEPERCUSSION" ] ; then
    echo "RYTHMEPERCUSSION non définie"
    return 1;
fi

while true;
do
    echo $1 \($(date +%H:%M:%S)\);
    sleep "$RYTHMEPERCUSSION";
done
```

Dockerfile

```
FROM ubuntu:latest
MAINTAINER IngeSup "doclab@ingesup.com"
COPY percussion.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENV RYTHMEPERCUSSION 2
ENTRYPOINT ["/entrypoint.sh"]
CMD ["percussion"]
```

Construction d'une Image

- **Test du script** par son exécution sur la machine hôte

```
. ./percussion.sh bonjour
RYTHMEPERCUSSION non définie
echo $?
1
```

- **Pour un test fonctionnel**

```
RYTHMEPERCUSSION=2 . ./percussion.sh
(12:24:57)
(12:24:59)
(12:25:01)
(12:25:03)
(12:25:05)
(12:25:07)
(12:25:09)
^C
```

Construction d'une Image

Génération de l'image

- La génération de l'image est réalisée par la commande `docker build`

```
sudo docker build -t doclab/percussion .  
sudo docker images -a
```

- Lancement du conteneur

```
sudo docker run -it --name percussion doclab/percussion  
(12:24:37)  
(12:24:39)  
(12:25:41)  
(12:25:43)  
(12:25:45)  
(12:25:47)  
(12:25:49)  
^p^q  
sudo docker logs percussion | tail
```

Construction d'une Image

Génération de l'image

- Arrêt et relance du conteneur

```
sudo docker start percussion
sudo docker stop percussion
sudo docker logs percussion | tail -20
sudo docker pause percussion
sudo docker ps
sudo docker unpause percussion
sudo docker ps
sudo docker logs percussion | tail -20
```

Construction d'une Image

Génération de l'image

- Gestion des paramètres

```
sudo docker rm -fv percussio
sudo docker run -d --name percussio doclab/percussio
sudo docker rm -fv percussio
sudo docker run -d --name percussio doclab/percussio "Dom Tek"
sudo docker rm -fv percussio
sudo docker run -d --name percussio --env RYTHMEPERCUSSION=1 \
    doclab/percussio
```

Construction d'une Image

Reconstruction d'une Image

- La modification de la variable d'environnement souligne les capacités d'amélioration de la robustesse de fonctionnement fournies par les conteneurs Docker.
- Il est en effet possible de modifier la variable d'environnement, et en même temps, la valeur par défaut est assurée par le Dockerfile.
- Ainsi, il serait possible de simplifier le script `percussion.sh`, en supprimant les premières lignes car elles ne serviront pas dans une image Docker:

```
#!/bin/bash
while true;
do
    echo $1 \($(date +%H:%M:%S)\);
    sleep "$RYTHMEPERCUSSION";
done
```

Construction d'une Image

Reconstruction d'une Image

- Le lancement de la commande `build` permet de recréer l'image avec cette nouvelle version du script:

```
sudo docker -t doclab/percussion .  
:  
---> Using cache  
...
```

- Docker utilise le cache chaque fois qu'il est possible afin d'éviter de relancer des opérations longues et identiques.
- Il est possible par nécessité de passer outre le cache avec l'option `--no-cache`.

Construction d'une Image

Reconstruction d'une Image

- Le mécanisme de cache est en lien avec les images intermédiaires qui s'affichent lors de la création d'image.
- Les images intermédiaires peuvent être gardées après une compilation correcte, avec l'option `-rm=false` de la commande `docker build`.

Construction d'une Image

Commande additionnelle

- **ADD** Cette commande réalise la même action que celle de `COPY` dans un `Dockerfile`, mais offre deux fonctionnalités additionnelles.
- Si la source est un fichier archive `tar`, son contenu sera désarchivé vers la destination.
- Si la source est une URL, le contenu sera téléchargé et déposé dans la destination.

Construction d'une Image

Notion de contexte

- La compilation d'une image Docker se fait à l'intérieur d'un contexte, qui est le répertoire contenant le Dockerfile qui pilote cette génération d'image.
- Ainsi, la commande COPY ne pourra pas aller chercher de fichiers sur la machine hôte, mais seulement ceux qui se trouvent dans le même répertoire que Dockerfile.
- Le chargement du contexte est signalé lors de la compilation d'une image avec la commande `docker build`.
- Le contexte permet de sécuriser la production d'image, en ne donnant pas accès à d'autres fichiers. Ainsi ceci implique de bien s'assurer d'intégrer uniquement les fichiers nécessaires pour la construction d'image.

Construction d'une Image

Processus de démarrage

- Dans l'exemple précédent ENTRYPOINT correspondait au processus à exécuter, et CMD au paramètre qui devait lui être passé par défaut:

```
ENTRYPOINT ["/entrypoint.sh"]  
CMD ["percussion"]
```

- La différence entre RUN et CMD/ENTRYPOINT, est que la commande RUN est exécutée au moment de la compilation. RUN ne doit pas être utilisée pour lancer des processus qu'on souhaite voir actifs.
- La commande RUN est utilisée pour lancer des commandes de préparation de l'image.

Construction d'une Image

Processus de démarrage

- Cependant, les commandes CMD peut être utilisée sans ENTRYPOINT. L'exécutable à démarrer lors du lancement du conteneur sera alors spécifié dans CMD, comme le montre l'exemple:

```
FROM ubuntu:latest
MAINTAINER IngeSup "doclab@ingesup.com"
COPY percussion.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENV RYTHMEPERCUSSION 2
CMD ["/entrypoint.sh", "percussion"]
```

Construction d'une Image

Processus de démarrage

- L'exécution d'un conteneur sans paramètre donne le résultat attendu. Par contre, avec paramètre le résultat ne correspond plus à celui de la version précédente.

```
sudo docker run -it --name percussio clab/percussio
...
sudo docker rm -fv percussio
sudo docker run -it --name percussio clab/percussio \
    "Dom Tek"
...
```

Construction d'une Image

Processus de démarrage

- En effet, Docker considère alors que la *surcharge* remplace tout le tableau JSON – *JavaScript Object Notation* – avec les deux paramètres passés à la commande CMD.

```
http://www.json.org/  
http://json.org/example.html
```

- Une façon de lancer la surcharge est de remplacer la totalité de la commande à exécuter avec le paramètre associé:

```
sudo docker run -it --name percussioin doclab/percussioin \
    ./entrypoint.sh "Dom Tek"
...
sudo docker run -it --name percussioin --entrypoint \
    ./entrypoint.sh doclab/percussioin "Dom Tek"
...
```

Construction d'une Image

Processus de démarrage

- Bien que ce passage de paramètres soit utile, cet usage n'est pas recommandé car il s'éloigne de l'objectif initial de la partition entre `ENTRYPOINT` et `CMD`.
- `ENTRYPOINT` doit porter ce qui ne change pas d'une exécution à l'autre du conteneur. `CMD` est un opérateur précisément dédié à la fourniture d'une valeur par défaut destinée à être échangée par une valeur de paramètre fournie lors de l'exécution.

Construction d'une Image

Processus de démarrage

- Dans les cas où une image doit pouvoir changer de processus, il est recommandé de créer un script de lancement qui réagit au passage d'un paramètre et lance lui-même le bon processus, comme cela a été montré dans l'exemple de MongoDB.

Format ligne de commande ou exécution

- Soit le Dockerfile suivant qui utilise un tableau JSON de paramètres:

```
FROM ubuntu:latest
MAINTAINER IngeSup "doclab@ingesup.com"
COPY percussion.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENV RYTHMEPERCUSSION 2
CMD ["ps", "-aux"]
```

Construction d'une Image

Processus de démarrage

- L'exécution du conteneur montre que le processus principal est `ps`:

```
| sudo docker run -it --name percussion doclab/percussion
```

- Par contre l'utilisation de la forme dite `shell` de l'opérateur `CMD` donne le `Dockerfile` suivant:

```
FROM ubuntu:latest
MAINTAINER IngeSup "doclab@ingesup.com"
COPY percussion.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENV RYTHMEPERCUSSION 2
CMD ps -aux
```

- Le résultat d'exécution montre qu'un `shell` a été effectivement lancé pour porter la commande.

Construction d'une Image

Processus de démarrage

- ENTRYPOINT et CMD supportent les deux formes `exec` et `shell`. Dans sa forme `shell` – un simple texte passé comme paramètre au lieu d'un tableau au format JSON, c'est `/bin/sh -c` qui sera utilisé pour lancer les paramètres de ENTRYPOINT.
- La forme `shell` a l'avantage que les variables comme `$HOME` ou autres seront substituées, mais l'inconvénient est que les signaux ne seront pas correctement remontés, car la valeur passée dans ENTRYPOINT est alors une sous-commande du processus de PID 1, qui est le `/bin/sh`.

Construction d'une Image

Processus de démarrage

- Afin d'accepter par exemple le signal SIGTERM envoyé par les touches Ctrl-C en mode interactif, une nouvelle version du script `percussion.sh`

```
#!/bin/bash
set -e
trap "echo SIGNAL" HUP INT QUIT KILL TERM

while true;
do
    echo $1 \($(date +%H:%M:%S)\);
    sleep "$RYTHMEPERCUSSION";
done
```

- Il est de bon usage que les conteneurs réagissent de manière propre à un SIGTERM, en annulant la tâche en cours et en nettoyant ce qui a besoin de l'être avant de s'arrêter.

Construction d'une Image

Commandes diverses

- **WORKDIR** est utilisé pour fixer le répertoire courant dans le déroulement du fichier `Dockerfile`. Cette commande peut être utilisée plusieurs fois dans le `Dockerfile`. Le répertoire de travail évolue alors au fur et à mesure de la compilation.
- **LABEL** permet de fournir des couples clé/valeur qui serviront de *métadonnées* décrivant l'image lorsque celle-ci sera diffusée et utilisée. Une utilisation classique est de faire porter la version de l'image
| `LABEL version="2.0"`
- La commande `docker inspect` qui permet de retrouver les métadonnées sur une image en paramètre.

Construction d'une Image

Export et import sous forme de fichiers

- Un moyen manuel d'exporter une image sous forme d'un fichier archive:

```
| sudo docker save -o /tmp/percussion.tar doclab/percussion
```

- Le moyen d'importer une archive d'image

```
| sudo docker load -i image_archive
```

Principe & Architecture

Union file systems

- *Union file systems* ou UnionFS, sont des systèmes de fichiers qui fonctionnent en créant des couches, ce qui les rend légers et rapides.
- Docker Engine utilise UnionFS pour fournir les blocs de construction des conteneurs. Docker Engine peut utiliser plusieurs variantes de UnionFS, notamment AUFS, btrfs, vfs et DeviceMapper.

Principe & Architecture

Le format de conteneur

- Docker Engine combine les espaces de nommage, les contrôles groupes, et UnionFS dans une enveloppe (*wrapper*) nommé *format de conteneur*.
- Le format de conteneur par défaut est `libcontainer`. À l'avenir, Docker pourrait prendre en charge d'autres formats de conteneur en s'intégrant à des technologies telles que BSD Jails ou Solaris Zones.

Principe & Architecture

Présentation de Docker

- La technologie de conteneurs et la conteneurisation de données par `Docker` libère le potentiel de développement (*Dev*) et d'exploitation (*Ops*).
- `Docker` fournit pour les applications une liberté de choix, des opérations agiles et une sécurité intégrée de conteneur.

Principe & Architecture

Gestion de données dans Docker

- Par défaut tous les fichiers créés dans un conteneur sont stockés sur une couche de conteneur en écriture. Ce qui implique :
 - Les données ne sont pas persistentes à l'arrêt de l'exécution du conteneur et il peut être difficile d'extraire les données du conteneur si un autre processus en a besoin.
 - Une couche de conteneur en écriture dépend de la machine hôte sur laquelle le conteneur est exécuté. On ne peut pas facilement déplacer les données ailleurs.

Principe & Architecture

Gestion de données dans Docker

- L'écriture dans une couche en écriture d'un conteneur nécessite un *pilote de stockage* afin de gérer le système de fichiers. Le pilote de stockage fournit un système de fichiers union, utilisant le noyau `Linux`.
- Cette abstraction supplémentaire réduit les performances par rapport à l'utilisation de données de volumes, qui écrivent directement dans le système de fichier de l'hôte.

Principe & Architecture

Gestion de données dans Docker

- Docker a deux options pour les conteneurs pour stocker les fichiers dans la machine hôte, pour que les fichiers soient persistents même après l'arrêt du conteneur :
 - volumes ;
 - montages liés (*bind mounts*) ;
 - montage tmpfs uniquement sur Linux.

Principe & Architecture

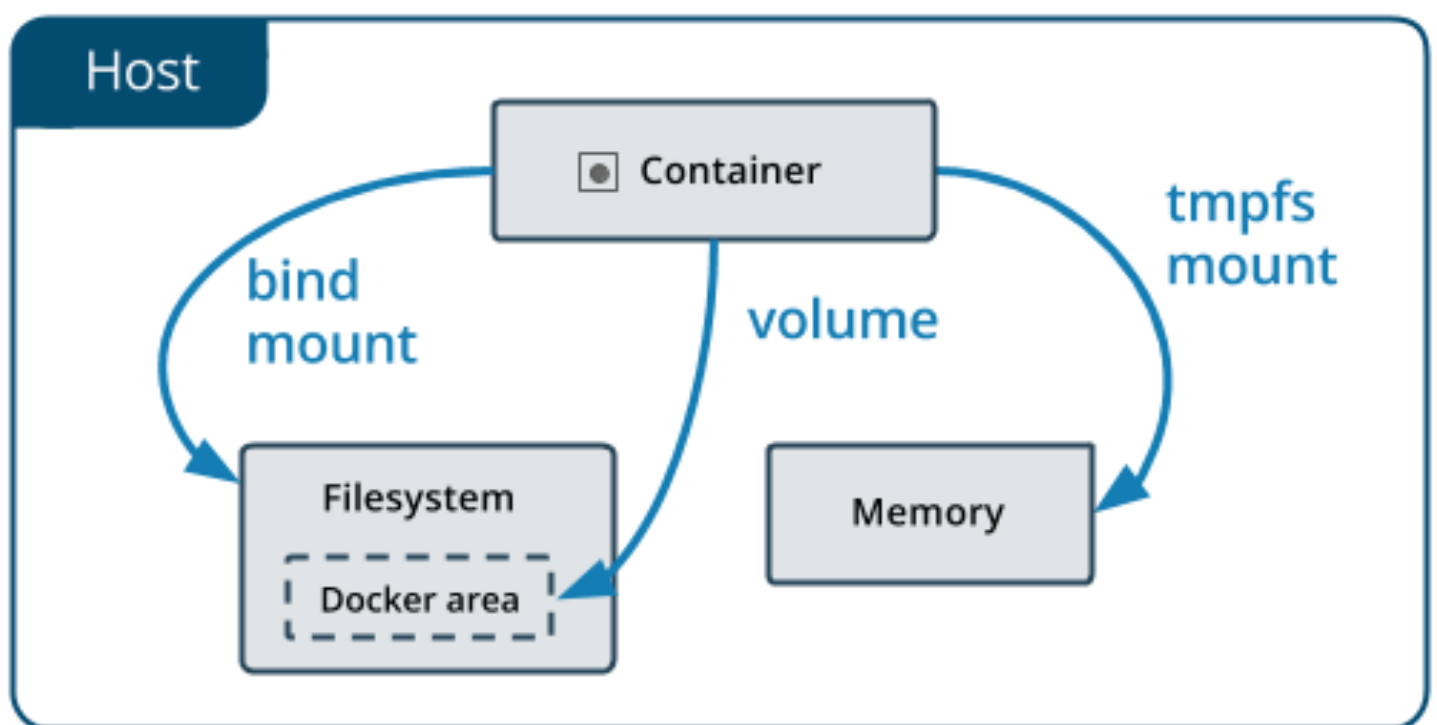


Fig 3 : Stockage de données dans Docker

Principe & Architecture

Choix de montage

- Quel que soit le type de montage utilisé, les données sont organisées sous forme d'une arborescence logique de répertoires et de fichiers dans le système de fichier du conteneur.
- Pour comprendre la différence entre les différents types de montage, volumes, *bind mounts* et *tmpfs*, il suffit de déterminer l'emplacement de données sur l'hôte Docker.

Principe & Architecture

Choix de montage

- Les volumes sont stockés dans une partie de système de fichiers de l'hôte, géré par Docker (`/var/lib/docker/volumes/` sur Linux). Les processus n'appartenant pas à Docker ne doivent pas modifier cette partie du système de fichiers. Les volumes constituent le meilleur moyen de conserver des données dans Docker.
- *Bind mounts* peuvent être stockés n'importe où sur le système de fichiers hôte. Il peut même s'agir d'un fichier ou répertoire. Les processus n'appartenant pas à Docker peuvent modifier à tout moment les données.
- Les montages `tmpfs` sont uniquement stockés dans la mémoire du système hôte et ne sont jamais écrits dans le système de fichiers hôte.

Principe & Architecture

- Les systèmes de fichiers superposés
- Présentation de AUFS
- Apports de Docker : Docker Engine pour créer et gérer des conteneurs Dockers
- Plates-formes supportées
- L'écosystème Docker : Docker Machine, Docker Compose, Kitematic, Docker Swarm, Docker Registry

Principe & Architecture

Le stockage de données

- Les pilotes de stockage vous permettent de créer des données dans la couche accessible en écriture de votre conteneur. Les fichiers ne seront pas conservés après la suppression du conteneur et les vitesses de lecture et d'écriture sont faibles.
- Les systèmes de fichiers superposés
- Présentation de AUFS
- Apports de Docker : Docker Engine pour créer et gérer des conteneurs Dockers
- Plates-formes supportées
- L'écosystème Docker : Docker Machine, Docker Compose, Kitematic, Docker Swarm, Docker Registry

Installation d'un Registre Privé

Image Docker en local

- Pour créer notre propre serveur de registre, nous utilisons l'image officielle nommée `registry`.

| https://hub.docker.com/_/registry/

- **Démarrer un registre privé:**

| `sudo docker run -d --name registre -p88:5000 registry:2.0`

- L'URL suivant donne accès à l'API en version 2 du registre, et en l'absence de tout paramètre ou portion d'URL supplémentaire, L'API en question renvoie une liste vide, mais le fait que nous voyons les séparateurs de liste JSON – les accolades – prouve que le serveur répond correctement.

Installation d'un Registre Privé

Image Docker en local

- **Pointer sur un registre donné** Pour démontrer l'usage du registre privé, nous allons déposer une image sur ce registre.

- La commande `push` permet d'envoyer une image locale sur le registre Docker Hub

```
| sudo docker push image
```

- Logiquement on doit se connecter sur le nouveau registre et lancer une commande `push` d'une image au nom existant. Mais Docker fonctionne autrement. Lorsqu'on nomme une image, son nom complet cache le serveur registre par défaut.

Installation d'un Registre Privé

Image Docker en local

- Le serveur registre par défaut est le registre public Docker Hub. Les images que nous avons présentées en fonction de leurs contenus nécessaires à démarrer un conteneur, sont une association de ces contenus à un dépôt sur un registre.
- Pour envoyer une image sur le registre privé nouvellement créé, il va falloir commencer par lui affecter un nouveau nom complet, grâce à la commande `docker tag`, puis seulement réaliser l'opération `push` sur ce nouveau nom.

```
sudo docker tag doclab/percussion localhost:88/percussion  
sudo docker push localhost:88/percussion
```

Installation d'un Registre Privé

Image Docker en local

- Les identifiants des images sont similaires pour les deux dénominations ; Docker garde bien le contenu en commun, tant qu'il l'est.

```
| sudo docker images
```

- En revenant au navigateur avec l'URL suivie du nom de l'image et `/tags/list`, nous aurons l'image qui a été déposée avec ses *tags*, ce confirme son fonctionnement correct.

```
| http://localhost:88/v2/percussion/tags/list
```

```
| {"name": "percussion", "tags": ["latest"]}
```

- L'image étant désormais stockée sur un registre, nous pourrions en principe la récupérer depuis n'importe quelle machine du réseau, sauf que dans l'exemple il s'agit de `localhost`.

Installation d'un Registre Privé

Registre sur le réseau local

- Afin de partager le contenu d'un registre, son emplacement doit être sur un serveur accessible à toutes personnes auxquelles l'administrateur souhaite mettre à disposition ses images.
- Ce serveur pourra être un serveur central dans un réseau d'entreprise, un serveur sécurisé sur un *cloud*, ou n'importe quelle autre machine accessible au groupe de personnes ciblées comme *clientes*.
- **Scénario et préparation des machines** La machine virtuelle `Linux` qui a servi pour les exemples jusqu'à maintenant va se comporter comme fournisseur d'une image qui sera déposée sur le registre privé démarré sur une seconde machine virtuelle utilisant `CentOS`.

Installation d'un Registre Privé

Registre sur le réseau local

- Enfin, en tant que client Docker, la machine physique hôte des machines virtuelles sera utilisée pour vérifier que l'image est effectivement disponible. Les trois machines font partie du même réseau.
- Il existe un lien entre le nom du registre privé et le nom de la machine qui supporte son processus. Ainsi, la première étape sera de déclarer le nom `doclabregistre` sur les trois machines afin d'accéder à cette *machine registre*.
- La solution passe par la configuration d'un service de résolution de nom – un serveur DNS. Sinon, dans un cadre de création d'une configuration de tests, on peut se contenter d'une déclaration dans les fichiers `hosts` des différentes machines – `ping` pour validation.

Installation d'un Registre Privé

Démarrage du registre

- La seconde étape consiste à démarrer le registre privé sur la machine CentOS. Il s'agit de la même procédure que l'approche *tout en local* montrée précédemment, sauf pour le port qui sera 5000

```
ping doclabregistre
sudo docker run -d --name registre -p 5000:5000 registry:2.0
...
sudo docker ps
```

- **Dépôt de l'image depuis une autre machine** Depuis la première machine virtuelle, on peut lancer une vérification d'accès au registre par l'URL comme précédemment. Il est aussi possible de le vérifier par ligne de commande

```
curl http://doclabregistre:5000/v2/
{ }
```

Installation d'un Registre Privé

Démarrage du registre

- Une fois vérifié, la suite est identique que précédemment, à savoir utiliser la commande `tag` pour donner un nouveau nom à l'image. La vérification peut se faire en listant les images.
- Ensuite, la commande `push` pour envoyer cette image dans le registre.
- Enfin, pour valider que la commande d'envoi a bien atteint sa destination:

| `http://doclabregistre:5000/v2/percussion/tags/list`

Installation d'un Registre Privé

Utilisation de l'image depuis une troisième machine

- Il s'agit de récupérer l'image depuis la machine hôte. Dans le cas où il s'agit de machines Windows ou Mac OS X, il serait possible d'installer Boot2Docker

| `http://boot2docker.io/`

- Dans le cas d'une machine Windows ou Mac OS, il faut rajouter dans la configuration de Docker dans le `/etc/default/docker`, la variable `DOCKER_OPTS`

| `--insecure-registry=doclabregistre:5000`

- Il s'agit ensuite de relancer le serveur Docker

| `sudo service docker restart`

Installation d'un Registre Privé

Suppression de l'image sur la machine source

- Une fois que l'image a été intégrée – *poussée* – dans le registre, elle peut être supprimée du cache local.

```
sudo docker images
sudo docker rmi ID_IMAGE
sudo docker rmi -f ID_IMAGE
sudo docker images
sudo docker rmi -f ID_IMAGE
sudo docker images
```

- La liste des images fait apparaître les deux tags pour le même identifiant d'image : il s'agit de la même Docker à laquelle il a été associé un nom pour la pousser dans le registre privé.

Installation d'un Registre Privé

Suppression de l'image sur la machine source

- En utilisant la commande `rmi` avec l'identifiant et non le nom, implique la suppression de tous les tags, mais un message explique que comme le tag est associé à des dépôts multiples, il faut utiliser l'option `-f`.
- Le premier appel avec cette option `-f` supprime le premier tag.
- Le second supprime le second tag, ainsi que toutes les images intermédiaires nécessaires, puisque plus aucun tag n'y fait référence.

Installation d'un Registre Privé

Gestion de la persistance

- **Gestion locale par volume** Le stockage des images se fait dans le conteneur Docker lui-même. Dès lors, que se passe-t-il si le conteneur est arrêté et qu'on oublie de valider son état dans une nouvelle image ? Cette situation implique la perte du contenu du registre.
- L'approche la plus simple pour éviter la perte du contenu du registre consiste à utiliser les volumes pour rediriger l'endroit où le conteneur du registre stocke les images qui lui sont envoyées vers un emplacement plus pérenne.

Installation d'un Registre Privé

Gestion de la persistance

- Une solution possible serait de choisir un répertoire de la machine hôte qui sera régulièrement sauvegardé, voir synchronisé en contenu sur une machine distante. Dans un contexte professionnel, la cible pourrait être une baie SAN, un partage NFS par exemple.
- Le comportement par défaut du registre fourni par Docker est de stocker sur `/var/lib/registry`. Il s'agit donc de substituer ce répertoire par un autre.

Installation d'un Registre Privé

Gestion de la persistance

- Il s'agit de procéder par la création d'une nouvelle instance de registre comme précédemment mais en précisant la *gestion de volume* avec l'option `-v`, de façon que le stockage du registre soit réalisé sur répertoire choisi.
- Ensuite, il s'agit d'envoyer par la commande `push` du contenu dans le registre.

```
docker run -d --name registre-volume -p5000:5000 \
-v /home/doclabregistre/contenu:/var/lib/registry \
registry
docker tag doclab/percussion doclabregistre:5000/percussion
docker push doclabregistre:5000/percussion
```

Installation d'un Registre Privé

Gestion de la persistance

- Une fois la configuration finalisée, il est possible de vérifier le résultat à travers le contenu du répertoire *volume* choisi pour le stockage du registre.

```
cd /home/doclabregistre/contenu  
tree -d
```

- L'arborescence fait apparaître le nom de l'image sur laquelle nous avons réalisé l'opération de push, en l'occurrence *percussion*, dans le répertoire *repositories*.

Installation d'un Registre Privé

Gestion de la persistance

- **SELinux** pour *Security Enhanced Linux*, est un module de sécurité intégré à **Linux**, et qui permet de définir des politiques de droits d'accès aux différentes ressources du système.
- En fonction de votre configuration, **SELinux** peut bloquer l'écriture par le processus du registre sur le répertoire fourni par le mécanisme des volumes, en dehors de tout problème de droits d'écriture sur le répertoire lui-même.
- Il est possible que la machine hébergeant le conteneur registre vous envoie un avertissement **SELinux** lorsqu'on exécute la commande `push` depuis une machine distante.
- Pour une approche de test, on peut se contenter de passer **SELinux** en mode permissif ou de le désactiver temporairement.

Administration

Présentation de Compose

- Utiliser Compose est fondamentalement un processus en trois étapes :
 - Définissez l'environnement de votre application avec un fichier `Dockerfile` afin qu'elle puisse être reproduite n'importe où ;
 - Définissez les services qui composent votre application dans `docker-compose.yml` afin qu'ils puissent être exécutés ensemble dans un environnement isolé ;
 - Exécutez `docker-compose up` et Compose démarre et exécute l'intégralité de votre application.

Administration

Les caractéristiques de Compose

Plusieurs environnements isolés sur un seul hôte

- Compose utilise un nom de projet pour isoler les environnements les uns des autres. Vous pouvez utiliser ce nom de projet dans plusieurs contextes différents :
 - Sur un hôte `dev`, pour créer plusieurs copies d'un même environnement, par exemple, lorsque vous souhaitez exécuter une copie stable pour chaque branche de fonctionnalité d'un projet ;
 - Sur un serveur CI, pour éviter toute interférence des versions, vous pouvez définir le nom du projet sur un numéro de version unique ;

Administration

Plusieurs environnements isolés sur un seul hôte

-Sur un hôte partagé ou un hôte dev, pour éviter que différents projets, pouvant utiliser les mêmes noms de service, n'interfèrent entre eux.

- Le nom de projet par défaut est le nom de base du répertoire du projet. Vous pouvez définir un nom de projet personnalisé à l'aide de l'option de ligne de commande `-p` ou de la variable d'environnement `COMPOSE_PROJECT_NAME`.

Administration

Préserver les données de volume

- Préserver les données de volume lors de la création de conteneurs. `Compose` préserve tous les volumes utilisés par vos services.
- Lorsque `docker-compose up` s'exécute, s'il trouve des conteneurs d'anciennes exécutions, il copie les volumes de l'ancien conteneur dans le nouveau conteneur.
- Ce processus garantit que toutes les données que vous avez créées dans des volumes ne sont pas perdues.

Administration

Recréer uniquement les conteneurs qui ont changé

- Compose met en cache la configuration utilisée pour créer un conteneur. Lorsque vous redémarrez un service qui n'a pas changé, Compose réutilise les conteneurs existants.
- La réutilisation des conteneurs signifie que vous pouvez modifier rapidement votre environnement.

Administration

Variables et déplacement d'une composition entre environnements

- Compose prend en charge les variables dans le fichier Compose. Vous pouvez utiliser ces variables pour personnaliser votre composition pour différents environnements ou différents utilisateurs.
- Vous pouvez étendre un fichier Compose à l'aide du champ `extend` ou en créant plusieurs fichiers Compose.

Administration

Cas d'utilisation

- `Compose` peut être utilisé de différentes manières. Certains cas d'utilisation courants tels que :

-Environnements de développement : Lorsque vous développez un logiciel, il est essentiel de pouvoir exécuter une application dans un environnement isolé et d'interagir avec elle. `Compose` peut être utilisé pour créer l'environnement et interagir avec celui-ci.

Le fichier `Compose` permet de documenter et de configurer toutes les dépendances de service de l'application (bases de données, files d'attente, caches, API de service *Web* par exemple). Avec `Compose`, vous pouvez créer et démarrer un ou plusieurs conteneurs pour chaque dépendance à l'aide d'une seule commande :

```
| docker-compose up
```

Administration

Cas d'utilisation

Ensemble, ces fonctionnalités constituent un moyen pratique pour les développeurs de se lancer dans un projet.

Compose peut réduire un *guide de démarrage pour les développeurs* comprenant plusieurs pages à un seul fichier Compose lisible par machine et à quelques commandes.

Administration

Cas d'utilisation

-Environnements de test automatisés : La suite de tests automatisés est un élément important de tout processus de déploiement continu ou d'intégration continue.

Les tests automatisés de bout en bout nécessitent un environnement dans lequel exécuter les tests. `Compose` constitue un moyen pratique de créer et de détruire des environnements de test isolés pour votre suite de tests.

En définissant l'environnement complet dans un fichier `Compose`, vous pouvez créer et détruire ces environnements en quelques commandes seulement :

```
docker-compose up -d  
./run_tests  
docker-compose down
```

Administration

Cas d'utilisation

-**Déploiements à hôte unique** : Traditionnellement, Compose était axé sur les étapes de développement et de test, mais avec chaque version, nous progressons vers des fonctionnalités plus orientées production.

Vous pouvez utiliser Compose pour déployer sur un Docker Engine distant. Le Docker Engine peut être une instance unique provisionnée avec Docker Machine ou un *cluster* entier de Docker Swarm.

Administration

Docker Compose : installation

- Vous pouvez exécuter Compose sous des machines Linux/Windows/MacOs 64 bits.
- Docker Compose s'appuie sur Docker Engine. Assurez-vous donc que Docker Engine est bien installé.

| <https://docs.docker.com/compose/install/#install-compose>

Vérification de l'installation

| `docker-compose --version`

Administration

Docker Compose : Étude de cas

- Il s'agit de créer une application Web Python s'exécutant sur Docker Compose. L'application utilise le *framework* Flask et gère un compteur d'accès dans Redis.

Configuration

Définir les dépendances de l'application.

- Créez un répertoire pour le projet :

```
mkdir composetest  
cd composetest
```

- Créez un fichier nommé `app.py` dans votre répertoire de projet.

Administration

Docker Compose : le fichier app.py

```
import time
import redis
from flask import Flask
app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)
def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)
@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

Administration

Docker Compose : le fichier app.py

- Dans cet exemple, `redis` est le nom d'hôte du conteneur `redis` sur le réseau de l'application. Nous utilisons le port par défaut pour Redis, 6379.

Docker Compose : le fichier requirement.txt

- Créez un autre fichier appelé `requirement.txt` dans votre répertoire de projet contenant :

```
flask  
redis
```

Administration

Docker Compose : le fichier Dockerfile

Création de Dockerfile

- Dans cette étape, vous créez un fichier Dockerfile qui crée une image Docker. L'image contient toutes les dépendances requises par l'application Python, y compris Python :

```
FROM python:3.4-alpine
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

Administration

Docker Compose : le fichier Dockerfile

- Le fichier indique à Docker de :
 - Construire une image en commençant par l'image `Python 3.4.` ;
 - Ajouter le répertoire courant `(.)` dans le chemin `/code` de l'image ;
 - Définir le répertoire de travail sur `/code` ;
 - Installer les dépendances `Python` ;
 - Définir la commande par défaut du conteneur sur `python app.py`.

Administration

Définir les services dans un fichier de composition

- Créer un fichier nommé `docker-compose.yml` dans votre répertoire de projet contenant :

```
version:  '3'
services:
  web:
    build:  .
    ports:
      - "5000:5000"
  redis:
    image:  "redis:alpine"
```

Administration

Le fichier docker-compose.yml

- Ce fichier Compose définit deux services, Web et Redis. Le service web :

-Utilise une image construite à partir du fichier Docker dans le répertoire en cours ;

-Transférer le port exposé 5000 du conteneur au port 5000 de la machine hôte. Nous utilisons le port par défaut pour le serveur Web Flask, 5000.

- Le service redis utilise une image publique Redis extraite du registre de Docker Hub.

Administration

Construire et exécuter l' application avec Compose

- À partir du répertoire de votre projet, démarrez l'application en exécutant :

```
| docker-compose up
```

Compose extrait une image Redis, construit une image pour votre code et lance les services que vous avez définis. Dans ce cas, le code est copié de manière statique dans l'image au moment de la création.

Administration

Construire et exécuter l' application avec Compose

- Utiliser un client HTTP pour tester l'application, par exemple :

```
| curl http://0.0.0.0:5000/
```

L'application Web doit être à l'écoute sur le port 5000 de l'hôte démon Docker. On peut aussi utiliser l'adresse suivante :

```
| curl http://localhost:5000/
```

Administration

Construire et exécuter l' application avec Compose

- Actualiser la page. Le résultat devrait changer par incrémentation d'un entier.
- Utiliser un autre terminal et exécuter la commande suivante pour répertorier les images locales :

```
| docker image ls
```

La liste des images à ce stade devrait renvoyer redis et web.

- Vous pouvez inspecter les images avec :

```
| docker inspect ID
```

Administration

Construire et exécuter l' application avec Compose

- (4.5) Arrêter l'application, soit en exécutant depuis votre répertoire de projet d'un autre terminal :

```
| docker-compose down
```

soit en appuyant sur CTRL + C dans le terminal où vous avez démarré l'application.

Administration

Ajouter un volume bind mount

- Le nouveau fichier `docker-compose.yml` avec un volume pour le service web :

```
version:  '3'
services:
  web:
    build:  .
    ports:
      - "5000:5000"
    volumes:
      - ./code
  redis:
    image:  "redis:alpine"
```

- `volumes` *monte* le répertoire du projet (répertoire courant) de l'hôte sur `/code` à l'intérieur du conteneur, vous permettant de modifier le code à la volée, sans avoir à reconstruire l'image.

Administration

Reconstruire et exécuter l'application

- Pour créer l'application avec le fichier `Compose` mis à jour, exécuter dans le répertoire projet :

```
| docker-compose up
```

Mettre à jour l'application

- Comme le code de l'application est maintenant monté dans le conteneur à l'aide d'un volume, vous pouvez modifier son code et voir les modifications instantanément, sans avoir à reconstruire l'image. On peut par exemple modifier le message affiché à partir du fichier `app.py`.

Administration

Des commandes annexes

- Pour exécuter vos services en arrière-plan, vous pouvez passer l'option `-d` (pour le mode *détaché*) pour composer de manière fixe et utiliser d'autres commandes par exemple ce qui est en cours d'exécution :

```
docker-compose up -d
docker-compose ps
docker-compose run web env
docker-compose stop
docker-compose down
```

Administration

Orchestration avec Docker Machine

- Docker Machine est un outil qui vous permet d'installer Docker Engine sur des *hôtes virtuels* et de gérer les hôtes à l'aide de commandes `docker-machine`.
- Vous pouvez utiliser Docker Machine pour créer des hôtes Docker sur votre système Mac ou Windows local, sur le réseau, dans votre centre de données ou chez des fournisseurs de cloud, tels que Azure, AWS ou Digital Ocean.
- À l'aide des commandes de Docker-Machine, vous pouvez démarrer, inspecter, arrêter et redémarrer un hôte géré, mettre à jour le client et le démon Docker et configurer un client Docker pour qu'il puisse communiquer avec votre hôte.

Administration

Utilisation de Docker Machine

- Docker Machine vous permet de provisionner plusieurs hôtes Docker distants sous différentes versions de Linux.
- Docker Engine s'exécute de manière native sur les systèmes Linux. Si vous utilisez un système Linux comme système principal et souhaitez exécuter des commandes Docker, il vous suffit de télécharger et d'installer Docker Engine.
- Toutefois, si vous souhaitez un moyen efficace de provisionner plusieurs hôtes Docker sur un réseau, dans le cloud ou même localement, vous avez besoin de Docker Machine.

Administration

Utilisation de Docker Machine

- Que votre système principal soit Mac, Windows ou Linux, vous pouvez y installer Docker Machine et utiliser les commandes `docker-machine` pour provisionner et gérer un grand nombre d'hôtes Docker.



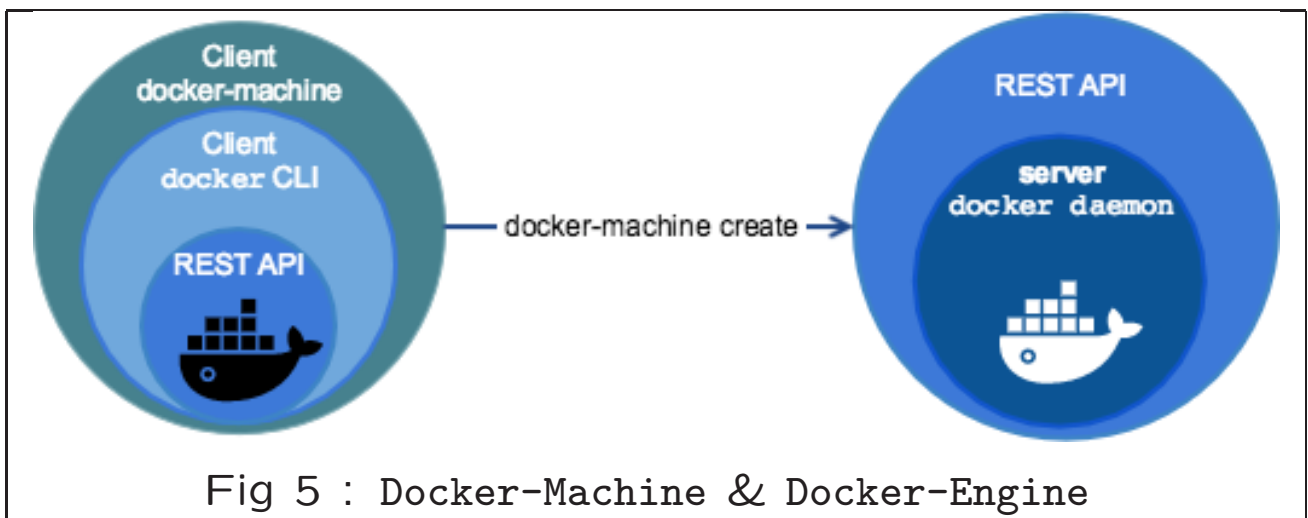
Fig 4 : Utilisation de Docker-Machine

- Il crée automatiquement des hôtes, y installe Docker Engine, puis configure les clients docker. Chaque hôte géré (*machine*) est la combinaison d'un hôte Docker et d'un client configuré.

Administration

Docker Machine & Docker Engine

- Docker désigne généralement Docker Engine, l'application *client-serveur* constituée du démon Docker, d'une API REST qui spécifie les interfaces permettant d'interagir avec le démon et d'un client d'interface de ligne de commande qui communique avec le démon.



- Docker Machine est un outil de *provisioning* et de gestion de vos hôtes *dockerisés* (hôtes sur lesquels Docker Engine est installé).

Administration

Présentation de Swarm pour le clustering

- Les versions actuelles de Docker incluent la gestion native d'un *cluster* de Docker Engine nommé `swarm`. Utiliser la ligne de commande Docker pour créer un `swarm`, déployer des services d'application sur un `swarm` et gérer son comportement.
- Les principales fonctionnalités de Docker `swarm` sont :
 - Gestion de *cluster* intégrée à Docker Engine. Utilisez la CLI Docker Engine pour créer un `swarm` de Docker Engine dans lesquels vous pouvez déployer des services d'application. Pas besoin d'application d'orchestration supplémentaire pour créer ou gérer un `swarm` ;

Administration

Docker Swarm

-Découverte de services. Les nœuds Swarm manager attribuent à chaque service de *cluster* un nom DNS unique et équilibrent la charge des conteneurs en cours d'exécution. Vous pouvez interroger tous les conteneurs exécutés dans le *cluster* via un serveur DNS intégré au cluster ;

-Equilibrage de la charge. Vous pouvez exposer les ports des services à un équilibreur de charge externe. En interne, le *cluster* vous permet de spécifier le mode de répartition des conteneurs de service entre les nœuds ;

Administration

Docker Swarm

-Sécurisé par défaut. Chaque nœud de `swarm` applique l'authentification et le cryptage mutuels TLS pour sécuriser les communications entre lui et tous les autres nœuds. Vous avez la possibilité d'utiliser des certificats *racine auto-signés* ou des certificats provenant d'une autorité de certification racine personnalisée.

-Mises à jour progressives. Au moment du déploiement, vous pouvez appliquer des mises à jour de service aux nœuds de manière incrémentielle. Le gestionnaire `swarm` vous permet de contrôler le délai entre le déploiement du service sur différents ensembles de nœuds. En cas de problème, vous pouvez rétablir une tâche dans une version précédente du service.

Administration

Configuration réseau et sécurité dans Docker

Docker inspect

- La commande `docker inspect` permet de fournir des informations détaillées de bas niveau sur les objets Docker et les constructions contrôlées par Docker. Par défaut, la commande retourne les résultats dans un tableau JSON.

Obtenir l'adresse IP d'une instance

- Dans la plupart des cas, vous pouvez sélectionner n'importe quel champ du JSON comme suit :

```
docker inspect --format='\n{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' ID
```

Administration

Obtenir l'adresse MAC d'une instance

```
docker inspect --format='\n{{range .NetworkSettings.Networks}}{{.MacAddress}}{{end}}' ID
```

Obtenir le chemin de journal d'une instance

```
| docker inspect --format='{{.LogPath}}' ID
```

Obtenir le nom de l'image d'une instance

```
| docker inspect --format='{{.Config.Image}}' ID
```

Lister toutes les liaisons de ports

- Vous pouvez parcourir les tableaux et les cartes dans les résultats pour produire une sortie texte simple :

```
docker inspect --format='{{range $p,\n    $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf 0).HostPort}}\n{{end}}' ID
```

Administration

Trouver un mappage de port spécifique

- La syntaxe `.Field` ne fonctionne pas lorsque le nom du champ commence par un nombre, contrairement à la fonction d'index du langage de template.
- La section `.NetworkSettings.Ports` contient une carte des *mappages de ports* internes sur une liste d'objets d'adresse/de port externes. Pour saisir uniquement le port public numérique, vous utilisez `index` pour rechercher la carte de port spécifique, puis l'index 0 contient le premier objet à l'intérieur de celui-ci.
- Ensuite, nous demandons au champ `HostPort` d'obtenir l'adresse publique.

```
docker inspect --format='\n    {{(index (index .NetworkSettings.Ports "8787/tcp") 0).HostPort}}' ID
```

Administration

Obtenir une sous-section au format JSON

- Si vous demandez un champ qui est lui-même une structure contenant d'autres champs, vous obtenez par défaut un *dump* de type Go des valeurs internes.
- Docker ajoute une fonction de modèle, `json`, qui peut être appliquée pour obtenir des résultats au format JSON.

```
| docker inspect --format='json .Config' ID
```