2 - Docker

2. Docker

Aujourd'hui, difficile de parler conteneur sans parler de **Docker**. Cependant, il existe d'autres solutions comme rkt, à prononcer rocket ou encore le tout nouveau cri-o qui est un projet incubateur de Kubernetes.

2.1 Historique

Docker est un programme informatique **open source** qui **exécute des conteneurs** au niveau du **système d'exploitation**. Il est principalement développé et maintenu par « Docker, Inc », une société américaine.

Docker est principalement développé pour Linux, où il utilise les **fonctions d'isolation des ressources du noyau Linux** comme les « cgroups » et les « espaces de noms » du kernel, ainsi qu'un système de fichiers comme OverlayFS pour permettre à ses **conteneurs indépendants** de fonctionner dans **une seule instance Linux**.

Ce fonctionnement permet d'**éviter les coûts de démarrage** et de **maintenance** que l'on retrouve sur les **machines virtuelles**.

Le support du noyau Linux pour les espaces de noms (« namespace » en anglais) isole principalement la vue d'une application de l'environnement d'exploitation, y compris les arbres de processus, le réseau, les ID utilisateur et les systèmes de fichiers montés.
Tandis que les « cgroups » du noyau fournissent une limitation des ressources pour la mémoire et le CPU.

Figure 1 : Logo Docker

Solomon Hykes a lancé **Docker** en **France** en tant que projet interne au sein de dotCloud, une société de PaaS, avec les contributions initiales d'autres ingénieurs de dotCloud, dont Andrea Luzzardi et Francois-Xavier Bourlet, mais également Jeff Lindsay en tant que collaborateur indépendant.
Docker représente une évolution de la technologie propriétaire de dotCloud, qui est elle-même construite sur des projets open source antérieurs tels que Cloudlets.

2.2 Les dates clés

Le logiciel a été **présenté pour la première fois au grand public** à la PyCon de Santa Clara en **2013**.

Docker a été publié sous forme de **logiciel libre** à partir de **mars 2013**. Un an plus tard, avec la sortie de la version 0.9,

Docker abandonne LXC comme environnement d'exécution par défaut pour le remplacer par sa propre librairie écrite dans le langage de programmation **Go** : **« libcontainer »**.

Le 19 septembre 2013, Red Hat et Docker ont annoncé une collaboration autour de Fedora, Red Hat Enterprise Linux et OpenShift, leur solution de PaaS pour les entreprises.

En novembre 2014, les services de conteneurs Docker ont été annoncés pour l'Amazon Elastic Compute Cloud (EC2). Un mois plus tard, IBM a annoncé un partenariat stratégique avec Docker qui permet à Docker de s'intégrer plus étroitement avec IBM Cloud.

Le 22 juin 2015, Docker et plusieurs autres sociétés ont annoncé qu'ils travaillent sur une nouvelle norme indépendante du fournisseur et du système d'exploitation pour les conteneurs logiciels.

Une analyse de mai 2016 a montré que les organisations suivantes sont les **principaux contributeurs** de Docker : **The Docker team, Cisco, Google, Huawei, IBM, Microsoft et Red Hat**.

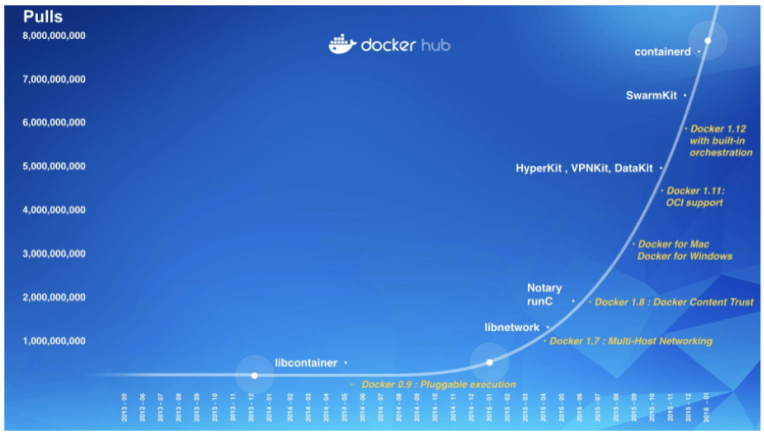


Figure 2 : Timeline des technologies Docker

Le 4 octobre 2016, Solomon Hykes a annoncé InfraKit en tant que nouvelle infrastructure de conteneurs à auto-guérison pour les environnements de conteneurs Docker.

Une dernière analyse de janvier 2017 montre que la présence de Docker a augmenté de 160% sur les profils LinkedIn en 2016.

Sur l'année 2017, les containers du logiciel ont été téléchargés plus de **13 milliards de fois**.

2.3 Spécificités des conteneurs Docker

Docker peut être considéré comme un **outil de virtualisation**, car il permet de **faire tourner différentes distributions Linux** sur le **même OS** en **partageant son noyau**.

Il devient donc possible de faire tourner un OS Ubuntu 16.04 grâce à un conteneur Docker spécifique sur une distribution CentOS 7. Le noyau Linux sera commun aux deux systèmes d'exploitation, mais le conteneur Ubuntu embarquera uniquement les librairies et binaires spécifiques à Ubuntu sans utiliser celle de l'OS hôte : CentOS.

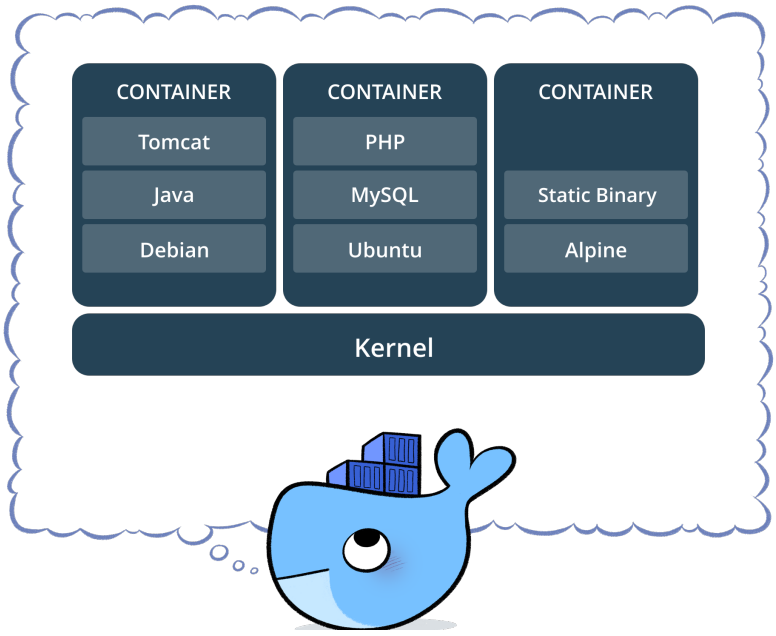
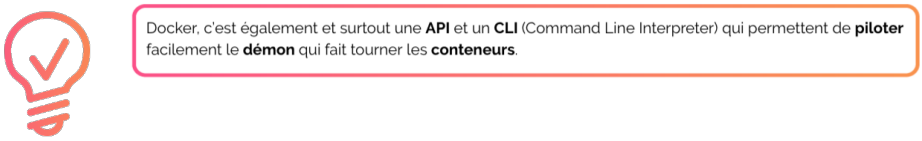


Figure 3 : Concept de conteneur Docker

a. Gestion de révision et API

L'une des principales forces de Docker, c'est d'avoir su implémenter les **mécaniques des VCS** (Version Control System) tel que Git avec des **notions de dépôts** (« registry » en anglais) privée de ou distant public tel que DockerHub, un système de tag de version, et de réutilisation de code existant.



Avec seulement **quatre commandes « build », « push », « pull » et « run »** vous aurez construit puis poussé votre propre image Docker sur un dépôt distant.
Par la suite, vous allez **rapatrier cette image** sur un **serveur** afin de l'**instancier** pour faire tourner un conteneur basé sur l'image.

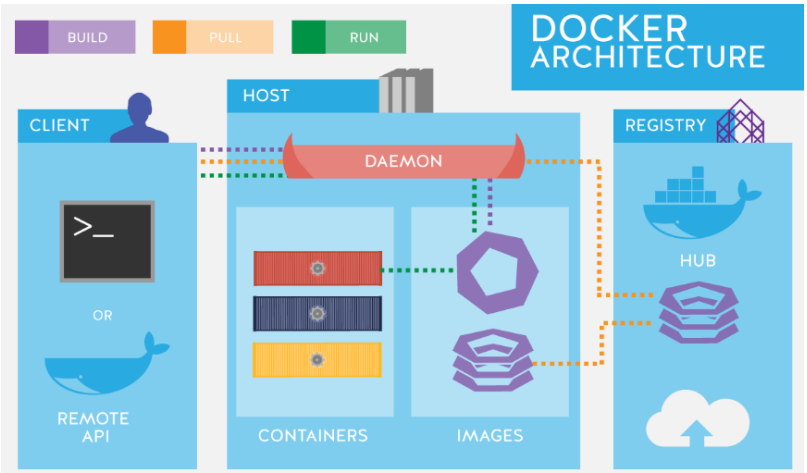


Figure 4 : Architecture de construction et de déploiement de conteneurs Docker

C'est notamment grâce à ces outils que la communauté a pu adopter très vite les mécanismes de Docker.

b. Infrastructure as Code

Docker a pris le pari de **décrire les conteneurs** à travers un fichier appelé **Dockerfile**. Ce dernier permet de construire un conteneur Docker à l'aide de plusieurs instructions telles que FROM, RUN, CMD ....

Voici un exemple de Dockerfile permettant de faire tourner une application Node.js :

```
FROM ubuntu

RUN apt-get install -y software-properties-common python
RUN add-apt-repository ppa:chris-lea/node.js
RUN echo
"deb http://us.archive.ubuntu.com/ubuntu/ precise universe"
 >> /etc/apt/sources.list
RUN apt-get update && apt-get install -y nodejs \
        && rm –rf /var/lib/apt/lists/*
RUN mkdir /var/www

ADD app.js /var/www/app.js

CMD ["/usr/bin/node", "/var/www/app.js"]
```

Ici, le but est de créer une image basée sur la dernière image d'Ubuntu.

La première instruction « **FROM** » permet d'indiquer que l'image à construire se base sur la dernière version de l'image Ubuntu, car il n'y a pas de version spécifiée : c'est donc le tag "latest" qui sera utilisé.
Les instructions « **RUN** » suivantes sont les équivalents des commandes « bash » qu'on exécuterait sur un serveur ou l'on voudrait installer Node.js.
Il y a tout de même de bonnes pratiques à respecter. Lorsque la commande « apt-get update » est utilisée, il faut systématiquement nettoyer le cache qui est positionné sous "/var/lib/apt/lists/*" dans la distribution Ubuntu.
« **ADD** » permet de copier des fichiers locaux dans l'image Docker directement.
La dernière instruction « **CMD** » permet de définir une action à attacher au démarrage du dock (conteneur docker).

c. Système de layer

Docker, ce n'est pas seulement un moteur de conteneurs d'applications portables : c'est aussi et surtout un **découpage des conteneurs** en **blocs de construction** appelés des **couches** ou **"layers"** en anglais.

Chaque image Docker se compose d'un **ensemble de calques** qui composent une **image finale**.
L'image Docker PHP 7.2 avec apache pèse 162 Mo, elle se base sur l'image de Debian 9 allégée qui pèse 22 Mo. L'image PHP 7.2 Apache rajoute donc une ou plusieurs couches faisant au total 140 Mo.

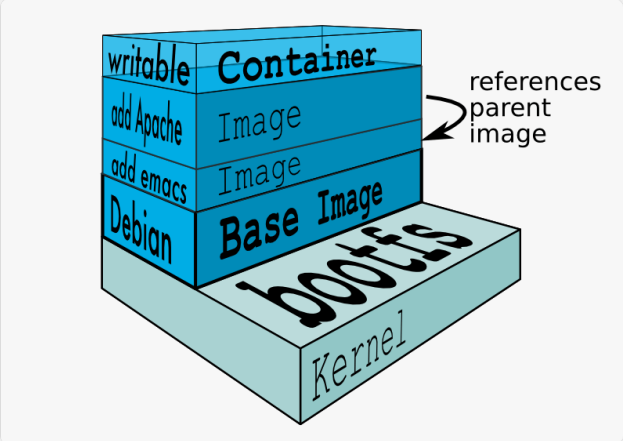


Figure 5 : Construction d'une image Docker

Considérons le Dockerfile suivant pour construire une image Ubuntu simple avec une installation d'Apache2 :

FROM ubuntu:16.04
RUN apt-get update
RUN apt-get install -y apache2
RUN touch /opt/a.txt

Si nous construisons l'image en appelant la commande "docker build -t mon-entreprise/appache2 . ", nous obtenons une image appelée "appache2", appartenant à un dépôt appelé "mon-entreprise".

Nous pouvons voir l'historique de votre image en appelant la commande suivante :

```
$ docker history mon-entreprise/appache2
IMAGE          CREATED             CREATED BY                                      SIZE
bfd131996f2d   About a minute ago  /bin/sh -c touch /opt/a.txt                      8 B
25c7ca554ca3   About a minute ago  /bin/sh -c apt-get install -y apache2            54.17MB
58ae3ba72841   2 minutes ago       /bin/sh -c apt-get update                       20.67 MB
d16f0cbdac7e   3 months ago        /bin/sh -c #(nop) ADD precise.tar.xz in /       204.4 MB
29a312113e76   3 months ago        /bin/sh -c #(nop) MAINTAINER Tianon Gravi
cb5cecbb6447   10 months ago                                                       0 B
```

L'image finale se compose de six images intermédiaires comme on peut le voir. Les trois premières couches (de bas en haut) appartiennent à l'image de base d'Ubuntu et le reste est à nous : un calque pour chaque instruction de construction.

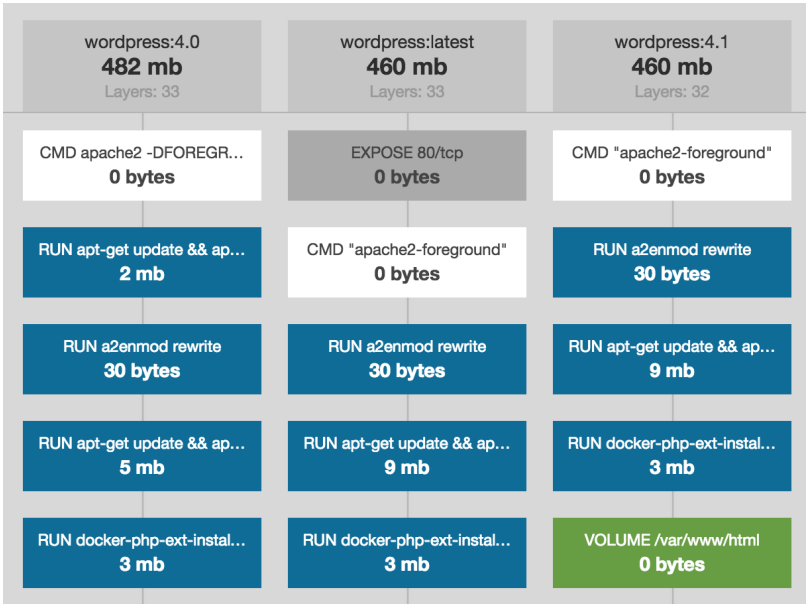


Figure 6 : Le concept de layer des images Docker

Pour bien comprendre l'avantage de cette superposition de couches, il faut construire une image légèrement différente. Considérons le Dockerfile ci-dessous pour construire presque la même image (seul le fichier texte de la dernière instruction a un nom différent) :

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y apache2
RUN touch /opt/b.txt.
```

Quand nous construisons ce fichier, la première chose que nous remarquerons est que la construction est beaucoup plus rapide. Comme nous avons déjà créé des images intermédiaires pour les trois premières instructions (FROM, RUN et RUN), Docker réutilisera ces layers pour la nouvelle image. Seul le dernier layer sera créé à partir de zéro. Cette image ressemblera à ceci :

```
$ docker history mon-entreprise/appache2-b
IMAGE          CREATED             CREATED BY                                      SIZE
dcad42b561a4   About a minute ago  /bin/sh -c touch /opt/b.txt                     8 B
25c7ca554ca3   About a minute ago  /bin/sh -c apt-get install -y apache2           54.17MB
58ae3ba72841   2 minutes ago       /bin/sh -c apt-get update                       20.67 MB
d16f0cbdac7e   3 months ago        /bin/sh -c #(nop) ADD precise.tar.xz in /       204.4 MB
29a312113e76   3 months ago        /bin/sh -c #(nop) MAINTAINER Tianon Gravi
cb5cecbb6447   10 months ago                                                       0 B
```

Comme nous le voyons, tous les calques sont les mêmes que pour la première image, à l'exception du premier où l'on a créé un fichier différent !
Ces layers ont certains avantages. Une fois que nous les construisons, Docker les réutilisera pour de nouvelles constructions. Cela rend les builds beaucoup plus rapides. C'est idéal pour l'intégration continue, où nous voulons construire une image à la fin de chaque construction réussie. Mais la construction n'est pas seulement plus rapide, les images sont aussi plus petites, puisque les images intermédiaires sont partagées entre les images.

d. DevOps

L'autre **intérêt** de **Docker**, c'est qu'il **s'intègre parfaitement** dans la **mouvance DevOps**.

Contrairement à des outils très orientés « Ops » comme Puppet ou Chef, Docker a su attirer les développeurs, car il a des **mécanismes très proches** de l'outil de gestion de version **Git** et essaye de garder ses mêmes forces : la **rapidité** grâce à un cache et un annuaire d'images à la GitHub.

Pour voir la fiche complète et les documents attachés, rendez-vous sur
https://elearning.26academy.com/course/play/5ac4f38a5bdbeb48ce2758cf