Docker

Docker permet la simplification des workflows et de la communication. Autrement dit, il permet un déploiement plus simple des applications.

Ce n'est évidemment pas le seul avantage de Docker ; il permet également d'augmenter la productivité, de gérer les dépendances plus facilement,etc. De plus, de plus en plus de cloud (comme AWS) le supporte.

Fonctionnement

Docker fonctionne sur un modèle client/serveur :

- **client:** a un rôle de communication, il dit au serveur quoi faire
- **serveur:** a un rôle de host, il run et manage les containers

Un client peut s'adresser à n'importe quel nombre de serveur qu'il le souhaite, pour un bonne et simple raison ; ils possèdent tous deux le même binary.

Pour qu'un serveur Docker écoute les connections entrantes, il suffit de le lancer avec le flag `-d` :

```
docker -d
```

> **Note:** *Une daemon est un processus qui s'execute en arriere plan*

On peut donc en venir à la question suivante ; comment s'effectue la communication client/serveur ?

Communication client / serveur

Il existe deux cas :

- Le client et le serveur sont sur la même machine, on utilise alors un socket Unix, nommé `docker.sock`.

> **Note:** *Un socket Unix permet de partager des données entre deux processus Unix via le système de fichier (plus particulièrement grâce à un fichier `*.sock`)*

- Le serveur est sur une machine distante, on utilise un socket TCP pour y accéder. Le port par défaut est le 2375, ou le 2346 pour du encrypted.

Pour faciliter la communication avec un serveur Docker, le deamon possède une API. De même, le networking de Docker est un bridge par l'interface Docker.

D'ailleurs, il est possible d'utiliser le réseau de l'hôte grâce à la commande:

```
docker -d --net host
```

Qu'est ce que Docker ?

Docker n'est pas une Virtual Machine ; c'est un wrapper autour d'un process Unix. Il est donc extrêmement léger, et peu servir pour une tâche éphémère, c'est à dire démarrer un Docker, faire une courte exécution et s'éteindre. Il est d'ailleurs d'usage de faire en sorte que les applications exécutées dans un Docker soient stateless ; les data sont conservées de manière externes.

Docker utilise les ressources du host, donc plusieurs containers sur une même machine entrent en compétition pour l'acquisition des ressources du host ; cela veut dire aussi que les containers utilisent le même kernel que l'host, ce qui peut induire des failles de sécurités.

Comme dit précédemment, les applications sont supposées être stateless et peuvent se lancer et s'éteindre en un instant. On veut donc souvent qu'elle ai la même configuration a chaque demarrage. On parvient à faire cela grâce aux variables d'environnements.

Il est également possible de sauver des data dans le file system, mais cette pratique est déconseillée pour de nombreuse raisons :

- Peu de performance
- Peu de place
- Le state n'est pas conservé a chaque redémarrage
- Le container dépend du host (et de son fs) et n'est donc plus scalable

Images Docker

Une image Docker est une pile de couche de *filesystem*. Chaque instruction permettant de construire une telle image genere une nouvelle couche, dependante de celle qui la precede. Ces couches pouvant etre reutilise entre differentes images, cela permet de sauver du space disk et du reseau.

> **Note:** Ce systeme de *layers*, couple aux *tags* qu'il est possible de , permet de faire du *Revision Control*.

Dockerfile

Afin de creer une image Docker, on decrit chaque layer grace a une instruction dans le fichier `Dockerfile`. Chaque instruction genere un layer, sauvegarde par Docker lors du build. Un layer peut etre reutilise par une autre image si les instructions consecutives sont les memes *en partant de la premiere instruction*.

Exemple 1: *Utilisation des layers build par l'image 1 par l'image 2 pour les instructions consecutives a partir de la premiere instruction*

Instructions img 1	Layer img 1	Build	Instructions img 2	Layer img 2	Build
FROM ubuntu	Layer A	Building	**FROM** ubuntu	Layer A	Using cache
RUN echo "a"	Layer B	Building	**RUN** echo "a"	Layer B	Using cache
RUN echo "b"	Layer C	Building	**RUN** echo "poulet"	Layer Z	Building

Exemple 2: *L'image 2 n'utilise pas les layers build par l'image 1 car la premiere instruction est differente et chaque layer depend des layers qui le precede*

Instructions img 1	Layer img 1	Build	Instructions img 2	Layer img 2	Build
FROM ubuntu	Layer A	Building	**FROM** alpine	Layer Q	Building
RUN echo "a"	Layer B	Building	**RUN** echo "a"	Layer R	Building
RUN echo "b"	Layer C	Building	**RUN** echo "b"	Layer S	Building

Instructions

- **FROM** `image:tag` : image sur laquelle se base la nouvelle image
- **MAINTAINER** `prenom nom <mail>` : informations pour contacter l'auteur du Dockerfile. Ces informations se retrouvent dans les metadata de l'image build.

- **LABEL** `"key"="value"` : ajoute des informations dans les metadata de l'image pour identifier / rechercher des images. On peut voir les labels d'une image grace a la commande `docker inspect`
- **USER** `username` : definit quel user va run les processes. Par defaut, cet user est `root`

> **WARNING:** En production, il est conseille de run les processes avec un user *sans* privileges, pour des questions de securite.

- **ENV** `VARNAME value` ou `VARNAME1=value1 VARNAME2=value2` : image sur laquelle se base la nouvelle image
- **RUN** `command` : execute une commande shell

> **NOTES:**
>
> - Il vaut mieux ne pas *RUN* des commandes de type `apt update` car cela rallonge les temps de build. Mieux vaut se baser sur des images contenant deja les updates.
> - Chaque instruction creant un *image layer*, c'est une bonne idee de combiner les commandes ou de faire et ajouter un script avec *ADD* et l'executer avec *RUN* pour limiter le nombre d'instruction (et donc la taille de l'image)

- **ADD** `sourcePath destinationPath` : inclus des fichiers du host vers l'image. Ils sont *copies*
- **WORKDIR** `path` : change le dossier actuel dans l'image. Toutes les instructions qui suivent seront executees dans ce `path`
- **CMD** `commandArray` : Defini la commande qui va lancer les process dans le container. Il est vivement conseille de n'avoir qu'un process par container

> **Exemple: CMD** ["echo", "'a'"]

Docker Mirror Registry

Interet

Un *Docker Mirror Registry* permet de mettre en place un registry local, qui stocke les images que l'on pull. Si l'image que l'on pull existe dans le mirror registry, il sera pull depuis lui, et non depuis internet, ce qui prend beaucoup moins de temps.

> **Note:**
> Le Docker Mirror Registry doit etre sur le meme fs ou le meme reseau local que le(s) Docker Server pour que ca ait de l'interet. Cela est particulierement utile lorsqu'on utiliser `docker-machine`

Mise en place

Pour les systemes utilisant systemd comme gestionnaire de service, il faut ajouter la ligne suivante au fichier `/etc/docker/daemon` (JSON):

```
"registry-mirrors": [ "http://<mirror_host>:<mirror_port>" ]
```

Pour la modification soit prise en compte par Docker, il faut restart le service Docker.

Swarm

Lors d'un update d'un service, si la configuration du service reste la meme, le service ne sera pas reload. Pour cela, il faut rajouter l'option `-f`. Dans ce cas, le container sera relance !

> **Exemple:**
>
> ```
> docker service update -f myService
> ```