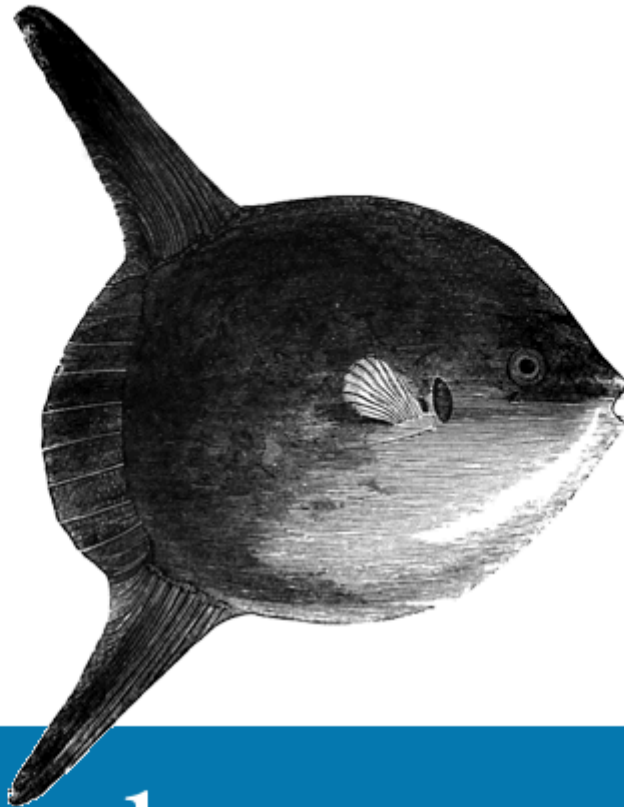


# Utilisation de Docker

*The best !!!*



# Docker

*The Ultimate Guide*

O RLY?

*XataZ*

Par XataZ

Date de publication : 18 juillet 2017

Pour réagir sur cet article, un espace de dialogue vous est proposé sur le forum :  
**Commentez**

I - Information.....	6
I-A - Historique des modifications.....	6
I-B - À faire.....	6
I-C - Contribution.....	6
I-D - Questions.....	6
II - Introduction.....	6
II-A - Qu'est-ce que Docker (pour notre ami Wikipédia) ?.....	7
II-B - Conteneurs VS machines virtuelles.....	7
II-C - Docker, pour quoi faire ?.....	8
III - Installation.....	8
III-A - Installation sous Linux.....	9
III-B - Installation sous Windows.....	10
III-B-1 - Docker4Windows.....	10
III-B-2 - Docker-toolbox.....	10
III-C - Installation sous Mac.....	11
III-C-1 - Docker4Mac.....	11
III-C-2 - Docker-toolbox.....	11
III-D - Mon environnement de test.....	12
IV - Le docker Hub.....	12
IV-A - Qu'est-ce que le docker Hub ?.....	12
IV-B - Chercher une image.....	12
V - Gérer les images.....	15
V-A - Télécharger des images.....	15
V-B - Lister les images.....	17
V-C - Supprimer les images.....	17
V-D - Conclusion.....	19
VI - Gérer les conteneurs.....	19
VI-A - Lancer, arrêter et lister des conteneurs.....	19
VI-B - Voir les logs des conteneurs.....	22
VI-C - Supprimer les conteneurs.....	23
VI-D - Cas concrets.....	24
VI-D-1 - Premier cas : le développeur.....	24
VI-D-2 - Deuxième cas : installer une application.....	26
VI-D-3 - Troisième cas : le déploiement.....	27
VI-E - Conclusion.....	28
VII - Créer une image.....	28
VII-A - Création d'un Dockerfile.....	28
VII-A-1 - Créons une image apache.....	28
VII-A-2 - Exemple d'une image lutim.....	30
VII-A-3 - Créons une image de base.....	32
VII-B - Les bonnes pratiques.....	34
VII-B-1 - Limiter les layers.....	34
VII-B-2 - Limiter la taille d'une image.....	35
VII-B-3 - La lisibilité.....	36
VII-B-4 - Éviter les processus root.....	37
VII-C - Conclusion.....	40
VIII - Déployer/partager une image.....	41
VIII-A - Via un dockerfile.....	41
VIII-B - Via le docker hub.....	41
VIII-C - Via une image tar.....	41
VIII-D - Conclusion.....	42
IX - Limiter les ressources d'un conteneur.....	42
IX-A - La mémoire.....	42
IX-B - Le CPU.....	43
IX-C - L'écriture disque.....	45
X - Docker Volume.....	45
X-A - Création d'un volume simple.....	46
X-B - Un peu plus loin.....	47

X-C - Encore et toujours plus loin avec les plugins.....	47
X-C-1 - Prérequis.....	47
X-C-2 - Installation du plugin.....	48
X-C-3 - Utilisation.....	48
X-D - Conclusion.....	48
XI - docker network.....	48
XI-A - Les types de réseaux.....	49
XI-B - Création d'un network.....	49
XI-C - Utilisation des networks.....	51
XI-D - Conclusion.....	53
XII - Docker compose.....	53
XII-A - Installation.....	54
XII-A-1 - Sous Windows.....	54
XII-A-2 - Sous GNU/Linux.....	54
XII-A-2-a - Installation via pip.....	54
XII-A-2-b - Installation par un conteneur.....	55
XII-B - Utilisation de docker-compose.....	55
XII-C - Créer une stack web.....	58
XII-D - Conclusion.....	60
XIII - Docker-machine.....	60
XIII-A - Qu'est-ce que docker-machine ?.....	60
XIII-B - Installation.....	60
XIII-B-1 - Sous Windows.....	61
XIII-B-2 - Sous GNU/Linux et OS X.....	61
XIII-C - Utilisation.....	61
XIII-C-1 - Créer une machine sous virtualbox.....	62
XIII-C-2 - Utilisons notre machine.....	65
XIII-C-3 - Gérer nos machines.....	67
XIII-D - Les Plugins.....	69
XIII-D-1 - Installation.....	69
XIII-D-1-a - Avec les binaires.....	69
XIII-D-1-b - Avec homebrew.....	69
XIII-D-1-c - Via les sources .....	69
XIII-D-2 - Utilisation.....	71
XIII-E - Conclusion.....	74
XIV - Clustering avec Swarm.....	74
XIV-A - Qu'est-ce que Swarm ?.....	74
XIV-B - Docker swarm (docker >= 1.12.X).....	74
XIV-B-1 - Créons notre cluster.....	74
XIV-B-2 - Les services.....	75
XIV-C - Conclusion.....	77
XV - Registry.....	77
XV-A - Installation de registry.....	77
XV-B - Utilisation.....	78
XV-C - Conclusion.....	79
XVI - Bonus.....	79
XVI-A - L'autocomplétion.....	79
XVI-B - Docker avec btrfs.....	80
XVII - Conclusion.....	81
XVIII - Ressources.....	81
XIX - Note de la rédaction de Developpez.com.....	82

## I - Information

### I-A - Historique des modifications

- **2017-05-15** : Suppression de docker swarm via des conteneurs - Màj du tuto vers Docker 17.05.0-ce
- **2016-12-05** : Ajout d'explications sur les options de lancement d'un conteneur - Ajout partie plugin pour docker volume (netshare)
- **2016-11-26** : Ajout de katacoda (Merci @xavier)
- **2016-11-25** : Ajout partie plugin pour docker-machine (scaleway)
- **2016-11-22** : Passe sur l'orthographe (Merci flashcode)
- **2016-11-16** : Publication de la version 2 du Tutoriel
- **2016-09-08** : Rédaction d'un chapitre sur docker network - Rédaction d'un chapitre sur docker volume
- **2016-09-06** : Màj du tuto vers Docker 1.12.X - Màj de docker swarm
- **2016-09-02** : Ajout des bonnes pratiques pour la création d'un dockerfile - Refonte du chapitre Limiter les ressources d'un conteneur
- **2016-09-01** : Refonte du chapitre Création d'une image - Ajout d'informations dans la présentation de Docker
- **2016-06-30** : Ajout de l'installation de docker4mac (magicalalex)
- **2016-06-24** : Ajout de l'installation de docker4windows
- **2016-05-30** : Amélioration de l'installation sous Mac OS (magicalalex)
- **2016-05-25** : Màj du tuto vers Docker 1.11.X - Rédaction d'un chapitre sur docker-machine - Rédaction d'un chapitre sur Swarm
- **2016-03-12** : Mise à jour de l'environnement de test
- **2016-02-19** : Ajout de l'installation sous Mac OS (Merci Magicalalex)
- **2016-01-30** : Ajout du paquet (aufs-tools) pour éviter les erreurs dans les logs système du type « Couldn't run auplink before unmount: exec: »auplink« : *executable file not found in \$PATH* »
- **2015-11-05** : Ajout de l'utilisation de btrfs
- **2015-11-01** : Ajout de quelques liens
- **2015-10-29** : Ajout de l'autocomplétion
- **2015-10-17** : Version initiale

### I-B - À faire

- Ajout d'une partie sur glusterfs (en version Docker)
- Refactor avec les nouveautés

### I-C - Contribution

Toute contribution est la bienvenue. N'hésitez pas à contribuer aux tutoriels, ajout d'informations, correction de fautes (et il y en a), améliorations, etc. Ça se passe [ici](#)

### I-D - Questions

Toute question sur la [discussion](#) ou sur [github](#)

## II - Introduction

Tout le monde a déjà entendu parler de Docker, mais peu ont décidé de passer le cap. Docker est le truc qui monte en ce moment, et il le mérite (avis purement personnel).

J'ai décidé de rédiger ce tutoriel afin de montrer ce qu'on peut en faire.

Pour ce tutoriel, je me base principalement sur mon expérience, sur mon apprentissage, sur les problématiques que j'ai pu rencontrer (que je rencontre encore pour certaines). J'ai essayé de structurer au mieux ce tutoriel, et j'espère que celui-ci vous conviendra.

Pour simplifier, Docker est un outil permettant la création d'un environnement (appelé conteneurs, *containers en anglais*) afin d'isoler des applications pour ne pas en gêner d'autres. Docker utilise des fonctionnalités natives au noyau Linux, comme les cgroups ou les namespaces, mais offre les outils pour le faire de manière simplifiée.



**ATTENTION :** Je serais incapable d'évaluer le niveau requis pour ce tutoriel, mais en fonction de votre utilisation, il vous faudra peut-être un peu plus que quelques notions de base en administration GNU/Linux.

## II-A - Qu'est-ce que Docker (pour notre ami Wikipédia) ?

*Docker est un logiciel libre qui automatise le déploiement d'applications dans des conteneurs logiciels. Selon la firme de recherche sur l'industrie 45 Research, « Docker est un outil qui peut empaqueter une application et ses dépendances dans un conteneur virtuel, qui pourra être exécuté sur n'importe quel serveur Linux ». Ceci permet d'étendre la flexibilité et la portabilité d'exécution d'une application, que ce soit sur la machine locale, un cloud privé ou public, une machine nue, etc.*



*Docker étend le format de conteneur Linux standard, LXC, avec une API de haut niveau fournissant une solution de virtualisation qui exécute les processus de façon isolée. Docker utilise LXC, cgroups, et le noyau Linux lui-même. Contrairement aux machines virtuelles traditionnelles, un conteneur Docker n'inclut pas de système d'exploitation, s'appuyant sur les fonctionnalités du système d'exploitation fournies par l'infrastructure sous-jacente.*

*La technologie de conteneur de Docker peut être utilisée pour étendre des systèmes distribués de façon à ce qu'ils s'exécutent de manière autonome depuis une seule machine physique ou une seule instance par nœud. Cela permet aux nœuds d'être déployés au fur et à mesure que les ressources sont disponibles, offrant un déploiement transparent et similaire aux PaaS pour des systèmes comme Apache Cassandra, Riak ou d'autres systèmes distribués.*

## II-B - Conteneurs VS machines virtuelles

Il faut savoir que Docker n'a rien inventé, la technologie de conteneurisation existe depuis un moment, notamment avec les *jails (prisons)* sous BSD, les *zones* sous Solaris, même Linux a eu son lot, avec *openvz*, *vserver* ou plus récemment *LXC*.

Docker permet de simplifier l'utilisation des outils présents dans le noyau Linux, à savoir les namespaces et les cgroups.



*Mais en fait, les conteneurs c'est comme les machines virtuelles ?*

Oui, mais NON, la finalité est quasiment la même : isoler nos applications. Mais le fonctionnement est totalement différent.

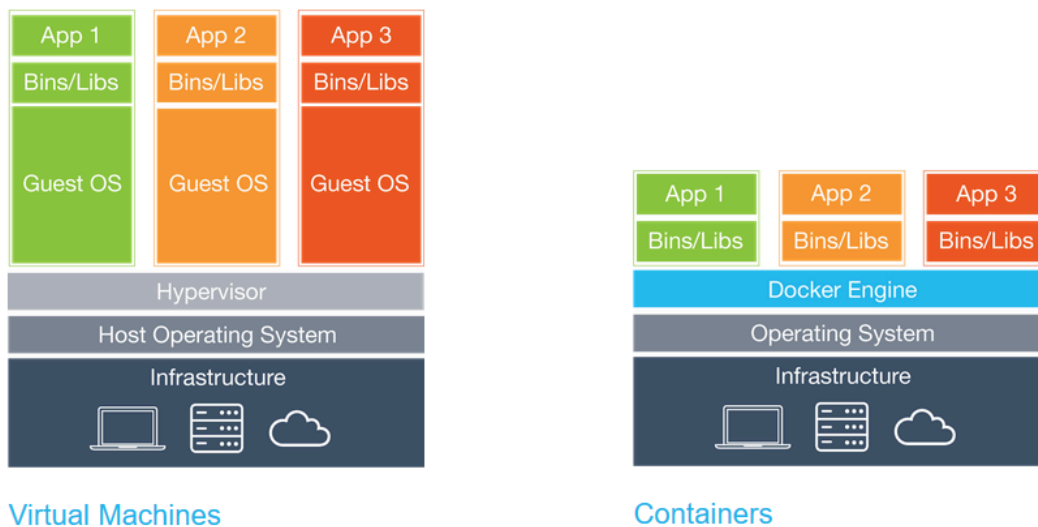
Pour une machine virtuelle, vous créez comme son nom l'indique, une machine virtuelle, c'est-à-dire, vous lui indiquez la ram à utiliser, le nombre de cpu, et vous créez un disque dur virtuel pour installer un OS. Votre machine dispose de ses propres ressources, et n'a aucunement conscience d'être virtualisée.

Pour les conteneurs c'est différent, on n'installe pas d'OS à proprement parler, mais un rootfs (le / d'un unix/Linux) qui est appelé image, qui contient les bibliothèques et les binaires nécessaires. Le noyau quant à lui, est partagé avec le système hôte. Nous pouvons évidemment limiter les ressources des conteneurs.

Machines virtuelles et Conteneurs ont leurs avantages et bien évidemment leurs inconvénients. Par exemple lancer ou créer un conteneur est vraiment plus rapide que lancer une VM. Mais une VM offre une meilleure isolation. Et ils ne sont pas forcément incompatibles, bien souvent, Docker est simplement utilisé dans une VM pour uniformiser une application entre les différents environnements (prod, préprod, intégration, etc.). Il arrive même de trouver dans une VM un seul conteneur.

Le plus gros défaut des conteneurs, c'est le fait que ce n'est pas *cross-platform*. On lance des conteneurs Linux sous Linux, des conteneurs BSD sous BSD ou des conteneurs Windows sous Windows.

Voici une image illustrant la différence entre VM et Docker :



## II-C - Docker, pour quoi faire ?

Docker n'a pas pour vocation de remplacer la virtualisation, voici plusieurs cas d'utilisation possibles.

- Le déploiement : puisque Docker a pour vocation de conteneuriser des applications, il devient simple de créer un conteneur pour notre application, et la dispatcher où bon nous semble. Un conteneur qui fonctionne sur une machine avec une distribution X, fonctionnera sur une autre machine avec une distribution Y.
- Le développement : cela permet de facilement avoir le même environnement de développement qu'en production, si ça marche quelque part, ça marchera partout. Cela permet également de pouvoir sur la même machine, tester avec plusieurs versions d'un même logiciel. Par exemple pour une application PHP, on pourrait facilement tester sur plusieurs versions de PHP, puis plusieurs versions de nginx et d'autres serveurs web.
- Installer des applications : étant donné que Docker propose une multitude d'outils, vous allez voir à quel point il est facile et rapide d'installer une application, bien souvent une seule ligne de commande suffit pour avoir par exemple notre nextcloud fonctionnel.

## III - Installation

Docker n'est pour l'instant compatible qu'avec GNU/Linux (et BSD en compatibilité Linux). Windows travaille par contre sur le projet, et une version custom de Docker verra le jour pour la prochaine *Release Candidate* de Windows Server 2016. Cela ne veut pas dire qu'il n'y a aucun moyen de l'utiliser sur Windows ou Mac.



## III-A - Installation sous Linux

Il existe des paquets tout prêts pour la plupart des distributions. Je vous renvoie vers ces paquets avec les procédures d'installation : <https://docs.docker.com/installation/>

Nous allons partir sur une debian Jessie (parce que !!!) : on commence par installer les prérequis puis on en profite pour faire une mise à jour :

```
1. $ apt-get update
2. $ apt-get upgrade
3. $ apt-get install apt-transport-https ca-certificates xz-utils iptables aufs-tools git
```

Puis on ajoute le dépôt, ainsi que la clé GPG de celui-ci :

```
1. $ echo "deb [arch=amd64] https://download.docker.com/linux/debian $(lsb_release -cs) edge" > /
etc/apt/sources.list.d/docker.list
2. $ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
```

**Nous utilisons ici le dépôt edge, qui est le dépôt testing de Docker avec une nouvelle version par mois, il est possible d'utiliser le dépôt stable, qui lui est mis à jour une fois tous les trois mois.**

Puis on installe :

```
1. $ apt-get update
2. $ apt-get install docker-ce
```

Il ne nous reste plus qu'à lancer Docker :

```
1. $ systemctl start docker
2. $ systemctl enable docker
```

Pour que Docker fonctionne dans les meilleures conditions, il faut ajouter ceci sous Debian dans le `/etc/default/grub` :

```
GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

Ceci permet de limiter la RAM utilisable par un conteneur.

Puis on régénère notre grub :

```
1. $ grub-mkconfig -o /boot/grub/grub.cfg
2. Création du fichier de configuration GRUB#8230;
3. Image Linux trouvée : /boot/vmlinuz-3.16.0-4-amd64
4. Image mémoire initiale trouvée : /boot/initrd.img-3.16.0-4-amd64
5. fait
```

On reboot, et c'est good.

On va tester avec une image de test :

```
1. $ docker run hello-world
2. Unable to find image 'hello-world:latest' locally
3. latest: Pulling from library/hello-world
4.
5. 03f4658f8b78: Pull complete
6. a3ed95caeb02: Pull complete
7. Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
8. Status: Downloaded newer image for hello-world:latest
9.
```

```
10. Hello from Docker.
11. This message shows that your installation appears to be working correctly.
12.
13. To generate this message, Docker took the following steps:
14. 1. The Docker client contacted the Docker daemon.
15. 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
16. 3. The Docker daemon created a new container from that image which runs the
17.    executable that produces the output you are currently reading.
18. 4. The Docker daemon streamed that output to the Docker client, which sent it
19.    to your terminal.
20.
21. To try something more ambitious, you can run an Ubuntu container with:
22. $ docker run -it ubuntu bash
23.
24. Share images, automate workflows, and more with a free Docker Hub account:
25. https://hub.docker.com
26.
27. For more examples and ideas, visit:
28. https://docs.docker.com/userguide/
```

Si quelque chose du genre s'affiche, c'est bon, Docker est fonctionnel.

## III-B - Installation sous Windows

Sous Windows il existe trois manières de l'installer :

- À la main => Une VM docker, avec le client docker sur Windows
- Docker-toolbox => Exécutable qui installe tout, virtualbox, une VM, et les clients
- Docker4Windows => Comme docker-toolbox, mais en mieux, utilise hyper-v au lieu de virtualbox (encore en version bêta, seulement compatible à partir de Windows 10 Pro).

Nous ne verrons ici qu'avec docker-toolbox, ainsi que docker4windows. L'installation manuelle étant la même chose que sous Linux.

### III-B-1 - Docker4Windows

Docker4Windows est encore en version bêta, mais reste totalement utilisable au quotidien. Pour le moment il n'est compatible qu'avec Windows 10 (version pro, Enterprise et Education), donc si vous êtes sur une autre version de Windows, il faudra passer par docker-toolbox ou faire une installation manuelle.

Avant de commencer l'installation de docker4windows, nous devons activer hyper-v. Pour ce faire, clic droit sur le menu d'application -> panneau de configuration -> Programmes et fonctionnalités -> Activer ou désactiver des fonctionnalités Windows -> Cocher Hyper-V -> OK.

On redémarre le PC, et normalement c'est bon, nous avons Hyper-V.

On peut passer à l'installation de docker4windows, et là c'est vraiment simple, télécharger l'exécutable [ici](#), ensuite c'est du next-next.

Vous aurez normalement dans la zone des notifications, une petite baleine, cela veut dire que Docker est bien installé.

### III-B-2 - Docker-toolbox

L'installation de docker-toolbox est rapide, il suffit de télécharger **docker-toolbox**, ensuite c'est du next-next. Ceci vous installera toute la panoplie du super-docker, c'est-à-dire, virtualbox avec une VM boot2docker, le client docker, docker-machine et docker-compose pour Windows. Puis également, si vous le souhaitez, kitematic, qui est un GUI pour installer des applications via docker.

## III-C - Installation sous Mac

Il y a deux solutions pour installer docker sur Mac OS X :

- *Docker4Mac* ;
- Docker-toolbox.

### III-C-1 - Docker4Mac

Docker4Mac est encore en version bêta, mais reste totalement utilisable au quotidien.

L'installation de docker4Mac est vraiment simple, téléchargez l'exécutable [ici](#) et ensuite c'est du next-next.

### III-C-2 - Docker-toolbox

Il faut télécharger l'installateur « docker toolbox » ici : <https://www.docker.com/products/docker-toolbox>

Ensuite vous exécutez le pkg et installez docker comme indiqué ici : <https://docs.docker.com/engine/installation/mac/>

Ensuite il faut créer une VM docker, ça va créer une vm dans virtualbox qui aura pour nom docker (vous pourrez vérifier dans virtualbox)

```
$ docker-machine create --driver virtualbox docker
```

Pour connecter notre shell à chaque fois avec la vm docker

```
$ echo 'eval "$(docker-machine env docker)"' >> ~/.bash_profile
```

N. B. Il se peut que le partage et la synchronisation de volumes soient lents, voire très lents (suite à un ralentissement du système de fichier NFS natif à OS X). Dans ce cas, la solution « Dinghy » peut vous aider à résoudre ces problèmes. <https://github.com/codekitchen/dinghy>

**Nous avons ici utilisé docker-machine pour créer notre machine, nous verrons dans un autre chapitre comment utiliser cet outil.**

Et pour finir on teste si ça fonctionne :

```
1. $ docker run hello-world
2. Unable to find image 'hello-world:latest' locally
3. latest: Pulling from library/hello-world
4.
5. 03f4658f8b78: Pull complete
6. a3ed95caeb02: Pull complete
7. Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
8. Status: Downloaded newer image for hello-world:latest
9.
10. Hello from Docker.
11. This message shows that your installation appears to be working correctly.
12.
13. To generate this message, Docker took the following steps:
14. 1. The Docker client contacted the Docker daemon.
15. 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
16. 3. The Docker daemon created a new container from that image which runs the
17.    executable that produces the output you are currently reading.
18. 4. The Docker daemon streamed that output to the Docker client, which sent it
19.    to your terminal.
```

```
20.  
21. To try something more ambitious, you can run an Ubuntu container with:  
22. $ docker run -it ubuntu bash  
23.  
24. Share images, automate workflows, and more with a free Docker Hub account:  
25. https://hub.docker.com  
26.  
27. For more examples and ideas, visit:  
28. https://docs.docker.com/userguide/
```

## III-D - Mon environnement de test

Je suis sous archlinux, et utilise docker directement sur ma machine.

Version de docker :

```
1. $ docker version  
2. Client:  
3. Version:      17.05.0-ce  
4. API version:  1.29  
5. Go version:   go1.8.1  
6. Git commit:   89658bed64  
7. Built:        Fri May 5 22:40:58 2017  
8. OS/Arch:      linux/amd64  
9.  
10. Server:  
11. Version:     17.05.0-ce  
12. API version:  1.29 (minimum version 1.12)  
13. Go version:   go1.8.1  
14. Git commit:   89658bed64  
15. Built:        Fri May 5 22:40:58 2017  
16. OS/Arch:      linux/amd64  
17. Experimental: false
```

## IV - Le docker Hub

### IV-A - Qu'est-ce que le docker Hub ?

Le docker Hub est un store où les utilisateurs de docker peuvent partager leurs images. Les images de base ont été créées par l'équipe de docker. Il est accessible ici : <https://hub.docker.com/explore/>

Ceci fait partie des forces de docker, beaucoup d'images sont disponibles (peut-être même trop), allant d'une simple debian, à une debian préconfigurée pour nextcloud par exemple. C'est justement cette méthode que j'appelle la méthode feignasse ^^ . Je veux nextcloud, je télécharge l'image et je crée un conteneur, vu que j'ai une bonne connexion, en moins d'une minute, j'ai un nextcloud fonctionnel, elle est pas belle la vie ?!

Aucun compte n'est nécessaire pour télécharger une image, mais bien évidemment pour pouvoir envoyer vos images, il faut un compte.

### IV-B - Chercher une image







Il existe deux méthodes pour chercher sur le Hub, la première par le site web. Gardons mon exemple, je veux un nextcloud. Sur le site je cherche donc nextcloud, et j'obtiens plusieurs résultats :

nextcloud

Dashboard Explore Organizations Create xataz

## Repositories (217)

All

 nextcloud official	87 STARS	500K+ PULLS	<a href="#">DETAILS</a>
 wonderfall/nextcloud public   automated build	187 STARS	500K+ PULLS	<a href="#">DETAILS</a>
 greyltc/nextcloud public   automated build	31 STARS	10K+ PULLS	<a href="#">DETAILS</a>
 rootlogin/nextcloud public   automated build	9 STARS	50K+ PULLS	<a href="#">DETAILS</a>
 aheimsbakk/nextcloud public   automated build	2 STARS	864 PULLS	<a href="#">DETAILS</a>
 indiehosters/nextcloud public   automated build	9 STARS	4.5K PULLS	<a href="#">DETAILS</a>

Nous avons donc plusieurs résultats, avec plusieurs informations :

- Le nom de l'image => Généralement sous la forme USER/IMAGE\_NAME, sauf dans le cas d'une image officielle, où c'est seulement IMAGE\_NAME
- Le nombre de stars => Le système de notation
- Le nombre de pulls => Le nombre de téléchargements

On en choisit l'officielle. Nous allons avoir plusieurs informations :

OFFICIAL REPOSITORY

nextcloud ☆

Last pushed: 2 days ago

Repo Info Tags

Short Description

A safe home for all your data

Docker Pull Command

```
docker pull nextcloud
```

Full Description

Supported tags and respective Dockerfile links

- 10.0.5-apache, 10.0-apache, 10-apache, 10.0.5, 10.0, 10 (10.0/apache/Dockerfile)
- 10.0.5-fpm, 10.0-fpm, 10-fpm (10.0/fpm/Dockerfile)
- 11.0.3-apache, 11.0-apache, 11-apache, apache, 11.0.3, 11.0, 11, latest (11.0/apache/Dockerfile)
- 11.0.3-fpm, 11.0-fpm, 11-fpm, fpm (11.0/fpm/Dockerfile)
- 9.0.58-apache, 9.0-apache, 9-apache, 9.0.58, 9.0, 9 (9.0/apache/Dockerfile)
- 9.0.58-fpm, 9.0-fpm, 9-fpm (9.0/fpm/Dockerfile)

Quick reference

- Where to get help:  
the Docker Community Forums, the Docker Community Slack, or Stack Overflow
- Where to file issues:  
<https://github.com/nextcloud/docker/issues>

L'onglet Repo Info est divisé en trois parties. La première est une description brève de l'image.

Sur la droite, nous avons la commande qui permet de la télécharger :

```
docker pull nextcloud
```

Puis dans le corps, plusieurs informations sur l'image, les versions des applications par exemple, puis souvent, les commandes/variables pour lancer un conteneur avec cette image, par exemple ceci :

```
docker run -d nextcloud
```

Ceci lance le conteneur, mais nous verrons ceci tout à l'heure.

Dans l'onglet Tags, ce sont les numéros de tags disponibles, souvent apparentés au numéro de version de l'application.

Nous avons parfois deux autres onglets, Dockerfile et Build Details. Ces onglets apparaissent quand les images sont « autobuildées » par le dockerhub, et donc il s'agit de la « recette » de l'image, c'est le fichier qui a permis de la construire. Nous verrons ceci plus loin dans le tutoriel. Autrement nous avons Build Details, qui permet de voir quand et comment se sont passées les constructions de l'image.

Autre possibilité pour trouver une image, en ligne de commande avec docker :

```
1. $ docker search nextcloud
2. NAME                                DESCRIPTION                                STARS
   OFFICIAL    AUTOMATED
3. wonderfall/nextcloud                Nextcloud - a safe home for all your data.... 187
   [OK]
4. nextcloud                            A safe home for all your data                88      [OK]
5. greyltc/nextcloud                  Nextcloud: a safe home for all your data. ... 31
   [OK]
6. rootlogin/nextcloud                Nextcloud docker image running on top of N... 9
   [OK]
7. indiehosters/nextcloud             Docker image for Nextcloud application.        9
   [OK]
8. sameersbn/nextcloud                Dockerized Nextcloud                          3
   [OK]
9. freenas/nextcloud                  Access & share your files, calendars, cont... 3
   [OK]
10. aheimsbakk/nextcloud               Nextcloud - a safe home for all your data.... 2
   [OK]
11. skybosh/nextcloud                  A simple Nextcloud image.                     2
   [OK]
12. cyphar/nextcloud                  NextCloud is a fork of OwnCloud. This is a... 1
   [OK]
13. whatever4711/nextcloud             Image for Nextcloud with php-fpm                0
   [OK]
14. bennibu/nextcloud                  nextcloud on php 7                             0
   [OK]
15. jefferyb/nextcloud                 Docker Image packaging for Nextcloud - A s... 0
   [OK]
16. mkuron/nextcloud                  Nextcloud with PHP IMAP extension              0
   [OK]
17. dklein/nextcloud                   Added cron to the official nextcloud Image    0
   [OK]
18. gorlug/nextcloud                   https://nextcloud.com/                        0
   [OK]
19. icewind1991/nextcloud-dev          Docker image for NextCloud for development    0
   [OK]
20. aknaebel/nextcloud                 This docker image provide a nextcloud serv... 0
   [OK]
21. whatwedo/nextcloud                 Nextcloud powered by Apache                    0
   [OK]
22. vger/nextcloud                     Nextcloud image based on Debian Jessie.        0
   [OK]
23. martmaiste/nextcloud               Nextcloud Docker using php-fpm and Nginx r... 0
   [OK]
24. georgehrke/nextcloud               Dockerized Nextcloud                          0
   [OK]
```

25. asannou/nextcloud	A safe home <b>for</b> all your data	0
[OK]		
26. ianusic/nextcloud	Nextcloud	0
[OK]		
27. servercontainers/nextcloud	nextcloud container on debian jessie with ...	0
[OK]		

Nous avons par contre ici beaucoup moins d'informations, personnellement je n'utilise cette méthode que pour rechercher des images de base (debian, centos, fedora, etc.).

## V - Gérer les images

Dans cette partie, nous allons voir comment gérer nos images, c'est-à-dire les télécharger, les lister, et bien sûr les supprimer.

Nous utiliserons ici `docker image [subcommand]` :

```

1. $ docker image --help
2.
3. Usage:  docker image COMMAND
4.
5. Manage images
6.
7. Options:
8.     --help    Print usage
9.
10. Commands:
11.  build      Build an image from a Dockerfile
12.  history    Show the history of an image
13.  import     Import the contents from a tarball to create a filesystem image
14.  inspect   Display detailed information on one or more images
15.  load      Load an image from a tar archive or STDIN
16.  ls        List images
17.  prune     Remove unused images
18.  pull      Pull an image or a repository from a registry
19.  push      Push an image or a repository to a registry
20.  rm        Remove one or more images
21.  save      Save one or more images to a tar archive (streamed to STDOUT by default)
22.  tag       Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
23.
24. Run 'docker image COMMAND --help' for more information on a command.
```

### V-A - Télécharger des images

Pour télécharger une image, on utilise cette commande :

```
$ docker image pull [nom image]:[tag]
```

Ce qui donne pour télécharger notre owncloud :

```

1. $ docker image pull nextcloud
2. Using default tag: latest
3. latest: Pulling from library/nextcloud
4. 10a267c67f42: Pull complete
5. 370377701f89: Pull complete
6. 455c73a122bc: Pull complete
7. fb71bac61c47: Pull complete
8. 288a1d91ad4e: Pull complete
9. 86e0256ba4b0: Pull complete
10. f14fbe7a9dfb: Pull complete
11. 0f36dd91c0ab: Pull complete
12. c2e4a1f87acc: Pull complete
13. ca5541ee478f: Pull complete
14. afb657ecb370: Pull complete
```

```

15. 8769771ac5f4: Pull complete
16. b08c0f680a7a: Pull complete
17. 7248dd69b572: Pull complete
18. 694f7bad4667: Pull complete
19. 7a4238c1b120: Pull complete
20. 727952036373: Pull complete
21. b3fd60530d47: Pull complete
22. 1b9a58bad45d: Pull complete
23. ddi11b8b6245: Pull complete
24. c9ba8440391e: Pull complete
25. Digest: sha256:dcdd3f4feeacedfb936b802a5c05885db3abcb909315aed162c2d8938f4ab29
26. Status: Downloaded newer image for nextcloud:latest

```

Si on ne met pas de tag, il télécharge automatiquement la latest. Comme nous avons vu dans la partie sur le dockerhub, nextcloud possède plusieurs tags. En spécifiant un tag, par exemple 10.0 ça donnerait :

```

1. $ docker image pull nextcloud:10.0
2. 10.0: Pulling from library/nextcloud
3. 10a267c67f42: Already exists
4. 370377701f89: Already exists
5. 455c73a122bc: Already exists
6. fb71bac61c47: Already exists
7. 288ald91ad4e: Already exists
8. 86e0256ba4b0: Already exists
9. f14fbe7a9dfb: Already exists
10. 0f36dd91c0ab: Already exists
11. 97d77fc1fd2c: Pull complete
12. d8c0a0fd7605: Pull complete
13. 79601c211937: Pull complete
14. 48bba15d4000: Pull complete
15. 96a57b907249: Pull complete
16. fee2a4169fe8: Pull complete
17. aeel6e9f06fe: Pull complete
18. 660947d2c7c3: Pull complete
19. 53dca82293cb: Pull complete
20. 45482c69893c: Pull complete
21. 70e2079f73d1: Pull complete
22. 4a2509f39f50: Pull complete
23. 54531e3804ca: Pull complete
24. Digest: sha256:b08e617ea87d39a9498ecd6ba2635017b8f1c661b057dae092df57c0c0eab968
25. Status: Downloaded newer image for nextcloud:10.0

```

Nous pouvons voir qu'il avait déjà des éléments, en fait une image est souvent basée sur une autre image, qui peut être basée sur une autre et ainsi de suite. Ce sont des layers (couches). Vous comprendrez mieux ceci lorsque nous apprendrons à créer des images. Chaque couche possède un id unique, c'est ce qui permet de savoir s'il est déjà présent ou non. Sur certaines images, comme les officielles, plusieurs tags peuvent être associés à une même image pour une même version, par exemple on peut voir sur le hub, que latest correspond également à 11.0.3-apache, 11.0-apache, 11-apache, apache, 11.0.3, 11.0, et 11.

Donc si maintenant je télécharge la version 11.0.3, puisqu'il a déjà toutes les couches, il ne devrait pas les retélécharger :

```

1. $ docker image pull nextcloud:11.0.3
2. 11.0.3: Pulling from library/nextcloud
3. Digest: sha256:dcdd3f4feeacedfb936b802a5c05885db3abcb909315aed162c2d8938f4ab29
4. Status: Downloaded newer image for nextcloud:11.0.3

```

Donc effectivement, tout était déjà présent, donc il n'a rien téléchargé.



*Pourquoi dans ce cas, mettre plusieurs tags ?*

En fait c'est tout bête, prenons par exemple, si je veux rester dans la branche 11.X.X de nextcloud, il me suffit d'utiliser le tag **11**, qui correspondra toujours à la dernière version 11.X.X, sans se soucier du nouveau numéro de version.



## V-B - Lister les images

Pour lister les images téléchargées, donc disponibles en local, nous utiliserons cette commande :

```
1. $ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
------------	-----	----------	---------	--------------

- REPOSITORY : Le nom de l'image
- TAG : Version de l'image
- IMAGE ID : Identifiant unique de l'image
- CREATED : Date de création de l'image
- VIRTUAL SIZE : Taille de l'image + toutes ses images dépendantes

Ce qui donne avec ce que l'on a téléchargé :

```
1. $ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nextcloud	11.0.3	05b0dd17351b	2 days ago	627MB
nextcloud	latest	05b0dd17351b	2 days ago	627MB
nextcloud	10.0	65f984a7a402	2 days ago	596MB

Nous voyons nos trois images. Comme nous pouvons le voir, nextcloud:11.0.3 et nextcloud:latest ont le même ID, mais rassurez-vous, ce sont juste des alias, elles ne prennent pas toutes les deux 627 MB d'espace disque.

Petite astuce pour ne pas afficher les doublons :

```
1. $ docker image ls | uniq -f 3
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
owncloud	8.2.5	52f7d60d34bd	13 days ago	699.2 MB
owncloud	9.0.2	4e0dc7be3d39	3 weeks ago	698.5 MB

Vous pouvez également afficher seulement l'image (ou les images) voulue :

```
1. $ docker images owncloud
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
owncloud	8.2.5	52f7d60d34bd	13 days ago	699.2 MB
owncloud	9.0.2	4e0dc7be3d39	3 weeks ago	698.5 MB
owncloud	latest	4e0dc7be3d39	3 weeks ago	698.5 MB

Ou si vous ne vous rappelez plus du nom complet, on peut jouer un peu avec les regex :

```
1. $ docker images */*cloud
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
wonderfall/owncloud	latest	233e6e0c61de	3 days ago	201 MB

## V-C - Supprimer les images

Pour supprimer une image, c'est plutôt simple :

```
$ docker image rm [nom Image ou ID image]:[tag]
```

Voici un exemple :

```
1. $ docker image rm nextcloud:10.0
2. Untagged: nextcloud:10.0
3. Untagged: nextcloud@sha256:b08e617ea87d39a9498ecd6ba2635017b8f1c661b057dae092df57c0c0eab968
4. Deleted: sha256:65f984a7a402e076e8c859a6198d295a1ce587e7583c4665770d6499b5bf7f2d
```

```

5. Deleted: sha256:e90e466e6f2dc093e5975c8550eb1981a9ebb635cc47b0bd81aeac1b5aca448f
6. Deleted: sha256:a2fcd4a803fdb8e2bef7064c3294b73c212778179b74b28600b9153933aa226a
7. Deleted: sha256:7a0e4dbef3ff7ff0edaa72de731f0919f287949a589156c46822aa30905079ce
8. Deleted: sha256:f23609f40dfb2fa2b9b2c998df6d0321aa6977dd23c20c6459b3cb067f78e020
9. Deleted: sha256:071048d8d0169cfaccf1f413da432818c20bc044664ade97e6ab8a639b611856
10. Deleted: sha256:40aab88ad21c9abf2932cc8a6ac14f42e159bde78b090989c49bd6479fbc2058
11. Deleted: sha256:dba4449ae481bb25cb8c58fe01724cd9d084fa3b60d5473c264dabe972fa8bbf
12. Deleted: sha256:389511eec4eafd25dacbcfa8164ea21411ca1d133ad20819875f85d4014cff1e
13. Deleted: sha256:2ad68ca5b887a6492d212f9b41a659aeb7f35de3c2352afe062254c5c9ea87af
14. Deleted: sha256:616fa657caf50c79f1dba57ab3030dafbcc377d8744ccdcee62b5f9afeed7cb2
15. Deleted: sha256:7cb5088120bb5f77d16b5451cf52ad157e506de215b031575689d41a008e8974
16. Deleted: sha256:29c7a1e9b9e530cbad0ea17b3b23c47b8186939363c04396efb948aeba99cfff
17. Deleted: sha256:2155b268258a477900f1ee8624d238c637db24b235f547e876b2696622d8137

```

On vérifie :

1. \$ docker image ls	2. REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
3. nextcloud		11.0.3	05b0dd17351b	2 days ago	627MB
4. nextcloud		latest	05b0dd17351b	2 days ago	627MB

Par contre si on supprime une image que l'on possède avec plusieurs tags, il ne supprime pas l'image, mais l'alias :

```

1. $ docker image rm nextcloud:11.0.3
2. Untagged: nextcloud:11.0.3

```

On peut vérifier :

1. \$ docker image ls	2. REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
3. nextcloud		latest	05b0dd17351b	2 days ago	627MB

Nous pouvons bien sûr supprimer plusieurs images :

```
$ docker image rm [image] [image] [image]
```

Petit bonus pour supprimer toutes les images (oui cela peut être utile) :

```

1. $ docker image prune -a
2. WARNING! This will remove all images without at least one container associated to them.
3. Are you sure you want to continue? [y/N] y
4. Deleted Images:
5. untagged: nextcloud:latest
6. untagged: nextcloud@sha256:dcdd3f4feeacedfb936b802a5c05885db3abcb909315aed162c2d8938f4ab29
7. deleted: sha256:05b0dd17351b75f9b818c0f8075e2d2b9f2ef25873422e2f52c20b12cf5c10bf
8. deleted: sha256:83e371d71cfbc5ac783026006959b8a38f582947ac7fc9d54f871de3df4be9f6
9. deleted: sha256:04be7a548f626aa9ab5e17a492194b51ff61df8c0545eb89f8fc35f1b073e70f
10. deleted: sha256:da435d4b91262cfe06ae635135399124e63afa4bd87ec4e1a161feba38a05036
11. deleted: sha256:82f6d723072ee7aad7cb4e95e6bc37d8760e055122487bfb32d92182551bc5f7
12. deleted: sha256:58603b6a763e5604d0fa3a858a891c7f17a23c5a10dafel6160e14b70f05f269
13. deleted: sha256:14b5b1d7e7f962b42dcbcb0413897bdb7d38584ad3e21f74ec8137306eee14db
14. deleted: sha256:176076b65ddaddff95ef2c19d65478746b61dd26973933fbc18c8d058ef06eb3
15. deleted: sha256:23befc96c53800701fa02c5f7e989c6122256143edcc2c84a8275b01e57f9dba
16. deleted: sha256:73585d624d5a4a0885e4c6de8463a6642dfb0fbef1b5847a300e0097bdbac064
17. deleted: sha256:9716cef6be433295bf3cd6ab7a9bd0500555bfdbaedf953c4637e18c8a59ba7
18. deleted: sha256:9b03fealb03a4c74a105dcef8d0df1e89b73124a7f4190a6703d615bff41dd6b
19. deleted: sha256:30963d11485e5a28117e91075497b16386a4d0b30b3ff263b57d19fb3388f3df
20. deleted: sha256:631b87d9cd40ecce449ddb56fb13c233e2ca779b0f6bd5ff3d857bdc6be82464
21. deleted: sha256:083157adf43b6c275b1612c03a0274f1ba1cde60f0a2d8139ae88f06d3846617
22. deleted: sha256:51ce5c8f7ebd5571e95a06f14049b0c7cd7de4bdf9ba6ceaec476b3586c99775
23. deleted: sha256:708b30f6a3f3cea053d1195467bd9ce08cca95581b4e7f55bc94d8516501aa7f
24. deleted: sha256:63d4aa293876898c55c282643fd737bb738f3afe8b1aa287c6fe12ad17411e5a
25. deleted: sha256:a5260b90ccbb5b6c6916ebadaf68fad4ccd98ac067a59440461592df8d3b719b
26. deleted: sha256:ad9f11b3ac807160c9963732f05e74e3fe2c173b4d88ff7324b315f2452e5c3b
27. deleted: sha256:53a165900cc96169bbfe9ab3d41511f180b6f6136c45ee9697cd03958d670445
28. deleted: sha256:8d4d1ab5ff74fc361fb74212fff3b6dc1e6c16d1e1f0e8b44f9a9112b00b564f

```

```
29.
30. Total reclaimed space: 626.5MB
```

## V-D - Conclusion

Même si ce chapitre n'apporte pas grand-chose, il est tout de même utile (voire indispensable) de savoir gérer ces images, ne serait-ce que pour un souci d'espace disque. Comme vous avez pu le voir, il est vraiment simple de gérer ces images, et je vous rassure, docker est simple en règle générale.

## VI - Gérer les conteneurs

Dans cette partie, nous verrons comment créer un conteneur (enfin on y vient !!! ^^), mais aussi comment le supprimer, comment les gérer, les relancer et plein d'autres choses indispensables.

Tout d'abord, un (tout petit) peu de théorie. Comme je l'ai dit tout au début, un conteneur ne peut se lancer que s'il a une commande à exécuter. Les images applicatives ont généralement une commande par défaut. Une fois cette commande terminée, le conteneur s'arrête. En fait ce principe est le même qu'une machine lambda, qui exécute un système d'init (systemd par exemple), et si celui-ci se termine, la machine s'arrête. En fait c'est là, la grosse différence avec une VM, même s'il est possible de le faire, un conteneur docker exécute seulement sa commande, pas besoin d'init pour gérer les points de montage, le réseau, le matériel, etc. seulement une commande.

### VI-A - Lancer, arrêter et lister des conteneurs

La première commande que nous utiliserons, sera `docker container run` qui s'utilise comme ceci :

```
$ docker container run [OPTIONS] IMAGE [COMMANDE]
```

Nous allons commencer par un petit conteneur, basé sur debian (pourquoi pas), et nous lui dirons d'afficher « bonjour mondedie !!! » :

```
1. $ docker container run debian echo "bonjour mondedie"
2.
3. Unable to find image 'debian:latest' locally
4. latest: Pulling from library/debian
5. 10a267c67f42: Pull complete
6. Digest: sha256:476959f29a17423a24a17716e058352ff6fbf13d8389e4a561c8ccc758245937
7. Status: Downloaded newer image for debian:latest
8. bonjour mondedie
```

Euh ?! il s'est passé quoi là ? Nous avons créé et exécuté notre conteneur, mais puisqu'il n'a pas trouvé l'image debian en local, il l'a téléchargée de lui même (sans avoir à utiliser `docker image pull`), pratique hein ?! Ensuite il a exécuté la commande qu'on lui a passée, à savoir écrire « bonjour mondedie ». Et c'est tout, puisque l'echo est terminé, il a éteint le conteneur.

Nous allons maintenant vérifier mes dires, nous allons vérifier si ce conteneur est démarré ou pas, pour ce faire nous utiliserons `docker container ls` :

```
1. $ docker container ls
2. CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
   PORTS              NAMES
```

Nous n'avons aucun conteneur en cours.



*Mais il doit bien être quelque part ce conteneur !! non ?!*

Oui et nous pouvons bien évidemment le voir, il suffit d'ajouter l'option `-a`, qui permet de voir tous les conteneurs :

```
1. $ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
	PORTS	NAMES		
3. ce198d73aafc	debian	"echo 'bonjour mon..."	4 minutes ago	
Exited (0)	4 minutes ago	pedantic_snyder		

Le voici, petite explication de ce tableau.

- **CONTAINER ID** : ID du conteneur, généré de manière à ce qu'il soit unique.
- **IMAGE** : l'image utilisée pour ce conteneur.
- **COMMAND** : la commande exécutée.
- **CREATED** : temps depuis création du conteneur.
- **STATUS** : le statut actuel du conteneur, ici exited avec un code retour 0 (sans erreur) depuis 8 minutes.
- **PORTS** : liste des ports écoutés (nous verrons ceci plus tard).
- **NAMES** : nom du conteneur, ici c'est un nom aléatoire, car nous n'en avons pas défini à notre conteneur.

Relançons notre conteneur plusieurs fois, avec une boucle et un time :

```
1. $ time sh -c 'i=1; while [ $i -le 20 ]; do docker container run debian echo "bonjour mondedie $i"; i=$((i+1)); done'
```

```
2. bonjour mondedie 1 !!!
```

```
3. bonjour mondedie 2 !!!
```

```
4. bonjour mondedie 3 !!!
```

```
5. bonjour mondedie 4 !!!
```

```
6. bonjour mondedie 5 !!!
```

```
7. bonjour mondedie 6 !!!
```

```
8. bonjour mondedie 7 !!!
```

```
9. bonjour mondedie 8 !!!
```

```
10. bonjour mondedie 9 !!!
```

```
11. bonjour mondedie 10 !!!
```

```
12. bonjour mondedie 11 !!!
```

```
13. bonjour mondedie 12 !!!
```

```
14. bonjour mondedie 13 !!!
```

```
15. bonjour mondedie 14 !!!
```

```
16. bonjour mondedie 15 !!!
```

```
17. bonjour mondedie 16 !!!
```

```
18. bonjour mondedie 17 !!!
```

```
19. bonjour mondedie 18 !!!
```

```
20. bonjour mondedie 19 !!!
```

```
21. bonjour mondedie 20 !!!
```

```
22. real    0m 3.53s
```

```
23. user    0m 0.00s
```

```
24. sys     0m 0.00s
```

Déjà on voit que c'est plus rapide, puisque l'image est en local, plus besoin de la télécharger, moins de 4 secondes pour 20 lancements.

Vérifions son état :

```
1. $ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
	PORTS	NAMES		
3. 8464c7bb5c96	debian	"echo 'bonjour mon..."	About a minute ago	
Exited (0)	About a minute ago	jolly_swartz		
4. e8d80d7dc23a	debian	"echo 'bonjour mon..."	About a minute ago	
Exited (0)	About a minute ago	reverent_galileo		
5. 56199baee7f9	debian	"echo 'bonjour mon..."	About a minute ago	
Exited (0)	About a minute ago	optimistic_lichterman		
6. 64ae77f60124	debian	"echo 'bonjour mon..."	About a minute ago	
Exited (0)	About a minute ago	condescending_mayer		
7. 379c6fcfb53f	debian	"echo 'bonjour mon..."	About a minute ago	
Exited (0)	About a minute ago	confident_roentgen		

```

8. 94cdbdf938cb      debian
   Exited (0) About a minute ago
9. 938300e4e31f      debian
   Exited (0) About a minute ago
10. 933b9542fd06     debian
   Exited (0) About a minute ago
11. 8e674629468f     debian
   Exited (0) About a minute ago
12. 2ecbda8dd3b4     debian
   Exited (0) About a minute ago
13. 3dlb01dfe606     debian
   Exited (0) About a minute ago
14. d69d98bf2aa7     debian
   Exited (0) About a minute ago
15. da2eb71d7eb5     debian
   Exited (0) About a minute ago
16. 27688ff57cc0     debian
   Exited (0) About a minute ago
17. 64c31a7323b8     debian
   Exited (0) About a minute ago
18. d1f9f173ef1d     debian
   Exited (0) About a minute ago
19. 0387ccf1092a     debian
   Exited (0) About a minute ago
20. eaf59f436d28     debian
   Exited (0) About a minute ago
21. aac609b95f89     debian
   Exited (0) About a minute ago
22. 6a6be6f146eb     debian
   Exited (0) About a minute ago
23. ce198d73aa7c     debian
   Exited (0) 6 minutes ago

"echo 'bonjour mon...' About a minute ago
   sad_easley
"echo 'bonjour mon...' About a minute ago
   agitated_mcclintock
"echo 'bonjour mon...' About a minute ago
   goofy_colden
"echo 'bonjour mon...' About a minute ago
   eloquent_meninsky
"echo 'bonjour mon...' About a minute ago
   pedantic_lamport
"echo 'bonjour mon...' About a minute ago
   vigorous_sinoussi
"echo 'bonjour mon...' About a minute ago
   heuristic_raman
"echo 'bonjour mon...' About a minute ago
   happy_elion
"echo 'bonjour mon...' About a minute ago
   wonderful_tesla
"echo 'bonjour mon...' About a minute ago
   laughing_euler
"echo 'bonjour mon...' About a minute ago
   naughty_jepsen
"echo 'bonjour mon...' About a minute ago
   mystifying_khorana
"echo 'bonjour mon...' About a minute ago
   hardcore_murdock
"echo 'bonjour mon...' About a minute ago
   infallible_nobel
"echo 'bonjour mon...' About a minute ago
   tender_minsky
"echo 'bonjour mon...' 6 minutes ago
   pedantic_snyder

```



*Oula, c'est quoi tout ça ?!*

En fait nous n'avons pas relancé notre conteneur, mais nous en avons créé d'autres. Cela vous montre la rapidité de création d'un conteneur.



*Mais comment le relancer ?*

Nous utiliserons `docker container start` :

```

1. $ docker container start 8464c7bb5c96
2. 8464c7bb5c96

```



*Euh oui, mais là, ça n'a pas marché ?*

En fait si, mais par défaut, il relance en arrière-plan, donc on ne voit rien s'afficher, mais on peut vérifier :

```

1. $ docker container ls -a | grep 8464c7bb5c96
2. 8464c7bb5c96      debian
   Exited (0) 2 seconds ago
   "echo 'bonjour monded" 6 minutes ago
   jolly_swartz

```

Donc là, on voit qu'il a été créé il y a 6 minutes, mais qu'il s'est terminé il y a 2 secondes, donc il vient de tourner.

Nous pouvons par contre le relancer en avant-plan, avec l'option `-a` :

```

1. $ docker container start -a 8464c7bb5c96

```

```
2. bonjour mondedie 20
```

Là on voit la commande.

Nous allons maintenant voir comment arrêter un conteneur, rien de bien méchant, pour ce faire, je vais créer un conteneur qui fait un ping de google.fr en arrière-plan, comme ceci :

```
1. $ docker container run -d debian ping google.fr
2. 03b1d375ac58955f439867cfe84d5635064e357d814a7e1977ee536f42fe7616
```

Nous pouvons vérifier que le conteneur tourne :

```
1. $ docker container ls
2. CONTAINER ID      IMAGE          COMMAND          CREATED          STATUS
   PORTS            NAMES
3. 03b1d375ac58      debian        "ping google.fr" 16 seconds ago   Up 16 seconds
   adoring_thompson
```

Comme on peut le voir, il est démarré depuis 16 secondes.

Nous allons d'abord le redémarrer puis directement afficher son statut, pour cela nous utiliserons `docker container restart` :

```
1. $ docker container restart 03b1d375ac58 && docker container ls
2. 03b1d375ac58
3. CONTAINER ID      IMAGE          COMMAND          CREATED          STATUS
   PORTS            NAMES
4. 03b1d375ac58      debian        "ping google.fr" About a minute ago Up Less than a
   second          adoring_thompson
```

On voit bien qu'il a redémarré.

Maintenant on peut l'arrêter, parce qu'un conteneur qui fait une boucle qui ne sert à rien, bah ça sert à rien, pour cela nous utiliserons `docker container stop` :

```
1. $ docker container stop 03b1d375ac58 && docker container ls
2. 03b1d375ac58
3. CONTAINER ID      IMAGE          COMMAND          CREATED          STATUS
   PORTS            NAMES
```

Voilà il est bien éteint. Il arrive parfois qu'un conteneur rencontre des difficultés à s'arrêter, vous pouvez utiliser `docker container kill` qui permet d'ajouter le signal d'arrêt.

Je ne l'ai pas encore précisé, mais toutes les actions effectuées sur les conteneurs peuvent l'être avec l'ID (complet ou les premiers caractères uniques), ou avec le nom du conteneur, autogénéré ou non.

## VI-B - Voir les logs des conteneurs

Tout sysadmin/devs doit penser, et même rêver des logs, c'est indispensable. Avec docker c'est assez spécial, les logs d'un conteneur sont en fait ce qui est en output (stdin et stderr) du shell.

C'est plutôt simple, même très simple, nous utiliserons `docker container logs` :

```
$ docker container logs [conteneur]
```

Exemple :

```
1. $ docker container logs 03b1d375ac58
2. PING google.fr (216.58.213.131): 56 data bytes
3. 64 bytes from 216.58.213.131: icmp_seq=0 ttl=54 time=23.563 ms
4. 64 bytes from 216.58.213.131: icmp_seq=1 ttl=54 time=23.317 ms
5. 64 bytes from 216.58.213.131: icmp_seq=2 ttl=54 time=22.317 ms
6. 64 bytes from 216.58.213.131: icmp_seq=3 ttl=54 time=22.888 ms
7. 64 bytes from 216.58.213.131: icmp_seq=4 ttl=54 time=22.600 ms
8. 64 bytes from 216.58.213.131: icmp_seq=5 ttl=54 time=22.595 ms
9. 64 bytes from 216.58.213.131: icmp_seq=6 ttl=54 time=23.009 ms
10. 64 bytes from 216.58.213.131: icmp_seq=7 ttl=54 time=22.187 ms
```

Il est possible de faire comme **tail** :

```
1. $ docker container logs --tail=20 03b1d375ac58 # Affiche les 20 dernières lignes
2. $ docker container logs -f 03b1d375ac58 # Affiche les logs au fur et à mesure
```

Et en fait c'est tout, c'est très simple.

## VI-C - Supprimer les conteneurs

Maintenant que nous avons vu comment créer, lister, démarrer, redémarrer et arrêter un conteneur, il ne nous reste plus qu'à... les supprimer. Pour cela, nous allons utiliser la commande :

```
$ docker container rm [conteneur]
```

Ce qui donnerait pour notre conteneur :

```
1. $ docker container rm 0387ccf1092a
2. 0387ccf1092a
```

Vous pouvez également supprimer un conteneur qui tourne, avec l'option **-f**.

Pas grand-chose d'autre à dire sur la suppression, à part comment supprimer tous les conteneurs arrêtés :

```
1. $ docker container prune
2. WARNING! This will remove all stopped containers.
3. Are you sure you want to continue? [y/N] y
4. Deleted Containers:
5. 8464c7bb5c966a3f970dc913698635265aa01f97cceedb454e256c25933d7015
6. e8d80d7dc23ae15f58571acc98d99859580b29ac4e5c131854008f38f2ee9456
7. 56199baee7f9f53be84bbea7d29f2a19bc4cacefa2c8c025ba463fb5c84d0c6e
8. 64ae77f60124af6b9f07d3f1be6849508b8f0401bd79494adf488a3c00bc88853
9. 379c6fcfb53fa41a4ef3dc824c7f81338a1a0ff0d94f9267b23a38f900db8b48
10. 94cdbdf938cb94098dc724792a2b973731f19b8d4d1de64e9a0ac2ab6eec8114
11. 938300e4e31fffb59d902329d4dad22160df6b0dc9b29a60780dcc9d809ee8f0d
12. 933b9542fd06126f7dd8453fca83ea11efec73d113a5ec36a739dc281816685f
13. 8e674629468fb61abedbc39819536825dbd1e87dbf3bde15c326a94c26285d41
14. 2ecbda8dd3b4721313e52eef3f443c7921730b35013a8d196a5a1df0bef8146d
15. 3dlb01dfe6064e6f0aa6bd9adecce144e1c001b546eaa66dbf598c73df4f4a10
16. d69d98bf2aa7e0450e857b8459f7f77f4adf32a500048415e2477dce9cedc363
17. da2eb71d7eb5a755dbeffffcdab759077d2a7c2ee5db3de7f7570444845f91c6c
18. 27688ff57cc0a9428e51a85635a3ce705fdd7ccd78358c5218be6e1090266b07
19. 64c31a7323b824782ff427909659824495988c8d780d7d46c2f46ffa7e200f6f
20. dl1f9f173ef1d0a0798ef9bb9c9924612541206a9dba1fd856fb293c4946734213
21. eaf59f436d289f1b390dc16d66491a65552bedaae1574309591d8fd44fadf206
22. aac609b95f89264d11e77a970911a86cc86b629ac5d7722b7b1932c78dfde634
23. 6a6be6f146eb4d7b1687d84eb1697227dd174b532646097f83fd0feac75ef308
24. ce198d73aafcde404e3134ee338cd327dd1b6566e6c0587cdeff8495560dff0a
25.
26. Total reclaimed space: 0B
```

Ou même tous les conteneurs, via la commande **docker container rm -f \$(docker container ls -aq)**

Et voilà pour la gestion basique des conteneurs.

Passons aux choses sérieuses.

## VI-D - Cas concrets

Jusqu'ici, nous n'avons rien fait de bien excitant, créer ou supprimer un conteneur c'est marrant cinq minutes, mais si celui-ci ne sert à rien... bah ça ne sert à rien. Nous allons donc maintenant voir des utilisations concrètes de conteneurs docker.

Avant de commencer, voici la liste des arguments que nous utiliserons dans cette partie :

- `-t` : fournit un terminal au docker ;
- `-i` : permet d'écrire dans le conteneur (couplé à `-t`) ;
- `-d` : exécute le conteneur en arrière-plan ;
- `-v` : permet de monter un répertoire local sur le conteneur ;
- `-p` : permet de binder un port sur le conteneur vers un port sur le host ;
- `-e` : permet l'ajout d'une variable d'environnement ;
- `--name` : donne un nom au conteneur ;
- `--rm` : détruit le conteneur une fois terminé ;
- `-w` : choisit le répertoire courant (dans le conteneur) ;
- `--link` : permet de faire un lien entre deux conteneurs.

Bien évidemment, beaucoup d'autres options existent, je vous renvoie à la [documentation](#) de docker run.

### VI-D-1 - Premier cas : le développeur

Admettons que j'ai développé une application nodejs, et je dois tester mon application sous différentes versions de node pour le rendre le plus « portable » possible. Installer plusieurs versions de nodejs peut être plutôt compliqué (sauf avec nvm) ou long si on utilise une VM par version, mais pas avec docker.

On commence par écrire notre code, un simple hello world :

```
1. // vim app.js
2. console.log("Hello World");
```

Puis on pull la version 6 et 7 de node :

```
1. $ docker image pull xataz/node:6
2. $ docker image pull xataz/node:7
```

Puis on peut faire nos tests, pour commencer avec node 6 :

```
1. $ docker container run -t --rm -v $(pwd):/usr/src/app -w /usr/src/app xataz/node:6 node app.js
2. Hello World
```

Puis node 7 :

```
1. $ docker container run -t --rm -v $(pwd):/usr/src/app -w /usr/src/app xataz/node:7 node app.js
2. Hello World
```

C'est cool, notre code fonctionne avec les deux versions.





### Qu'avons-nous fait ici ?

Nous avons lancé un conteneur via une image disposant de node (xataz/node:x), sur lequel nous avons mis plusieurs paramètres, un `-t` pour pouvoir voir le retour de la commande, ici nous n'avons pas besoin du `-i` puisque nous n'avons pas besoin d'interactivité avec le terminal. Nous avons monté le répertoire courant `$(pwd)` avec le paramètre `-v` dans `/usr/src/app`, nous avons donc choisi ce répertoire en répertoire de travail (workdir) avec l'option `-w`. Pour finir nous avons exécuté la commande `node app.js`.

Ici c'est une application plutôt simple, utilisons une application plus complète, comme un petit site, qui affichera *Hello Mondedie avec la version vX.X.X*. Donc voici le code :

```
1. // vim app.js
2. var http = require('http');
3.
4. var server = http.createServer(function (request, response) {
5.   response.writeHead(200, {"Content-Type": "text/plain"});
6.   response.end("Hello Mondedie avec la version " + process.version + "\n");
7. });
8.
9. server.listen(8000);
10.
11. console.log("Server running at 0.0.0.0:8000");
```

Et nous lançons nos conteneurs, mais cette fois-ci en arrière-plan :

```
1. $ docker container run -d -v $(pwd):/usr/src/app -w /usr/src/app -p 8001:8000 --name node5
   xataz/node:6 node app.js
2. 7669bef4b5c06b08a6513ed1ce8b8b036ad5285236a9e21a969897e5a9a8c537
3. $ docker container run -d -v $(pwd):/usr/src/app -w /usr/src/app -p 8002:8000 --name node6
   xataz/node:7 node app.js
4. 0e02e0844dd1b70a7e53e9e185831a05f93d9ed4f4a31f17d066b3eea38be90b
```

Ici nous n'avons que les id des conteneurs qui s'affichent, et nous rend la main directement, mais cela ne veut pas dire qu'ils ne tournent pas. Vérifions :

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
0e02e0844dd1	xataz/node:7	"node app.js"	About a minute ago	Up About a minute
7669bef4b5c0	xataz/node:6	"node app.js"	About a minute ago	Up About a minute

Nous pouvons tester dans notre navigateur, en tapant <http://XX.XX.XX.XX:8001> et <http://XX.XX.XX.XX:8002> (XX.XX.XX.XX étant l'ip de l'hôte docker), et nous voyons donc clairement que les deux pages affichent un numéro de version différent. J'ai donc ajouté trois paramètres ici, `-d` à la place de `-t`, pour lancer le conteneur en arrière-plan, `-p` pour rediriger un port de l'hôte vers le port du conteneur, c'est pour cela que nous avons utilisé les ports 8001 et 8002 pour accéder aux applications au lieu du port 8000. Ainsi que l'option `--name` qui donne un nom plus simple à notre conteneur, ce qui permet de mieux les gérer. J'ai également supprimé le `--rm`, qui logiquement n'est pas compatible avec un conteneur lancé en arrière-plan.

Maintenant je peux les supprimer avec leurs noms :

```
1. $ docker container rm -f node6 node7
2. node6
3. node7
```

Et voilà, on peut voir à quel point c'est pratique d'utiliser docker dans ce cas présent.

## VI-D-2 - Deuxième cas : installer une application

Nous allons maintenant voir comment installer/déployer une application. Sur le docker hub, on trouve toutes sortes d'images, comme des images pour ghost, ou pour wordpress, mais également des images plus spécifiques comme oracle. Ces images sont souvent des images AllinOne (Tout en un), c'est-à-dire qu'une fois le conteneur créé, c'est fonctionnel.

Nous allons ici créer un conteneur lutim. Nous prendrons ma propre image ([ici](#)).

Nous lançons donc notre application :

```
1. $ docker container run -d --name lutim -p 8181:8181 -e UID=1000 -e GID=1000 -e SECRET=mysecretcookie -e WEBROOT=/images -v /docker/config/lutim:/usr/lutim/data -v /docker/data/lutim:/usr/lutim/files xataz/lutim
2. Unable to find image 'xataz/lutim:latest' locally
3. latest: Pulling from xataz/lutim
4. c1c2612f6b1c: Already exists
5. 0e00ee3bbf34: Pull complete
6. 58fda08c5f8a: Pull complete
7. 1bb27614a217: Pull complete
8. 0dff0105dd58: Pull complete
9. Digest: sha256:a71eb9f0cfa205083029f0170aa5184a5fc9e844af292b44832dbd0b9e8fdeba
10. Status: Downloaded newer image for xataz/lutim:latest
11. 766be7bdb450d42b45a56d4d1c11467825e03229548dc9110c1e46e0d3fbf033
```

On vérifie que ça tourne :

```
1. $ docker ps
2. CONTAINER ID          IMAGE          COMMAND                  CREATED              STATUS
3. 766be7bdb450          xataz/lutim   "/usr/local/bin/start"   7 minutes ago       Up 7
   minutes           0.0.0.0:8181->8181/tcp   lutim
```

Nous avons ici ajouté des `-e`, ceci permet d'ajouter des variables d'environnement au conteneur. Ces variables seront utilisées soit directement par l'application, soit par le script d'init de l'image (que nous verrons dans la partie [Créer une image](#)). Dans notre cas nous avons ajouté quatre variables, mais il en existe d'autres (cf. [README](#)) :

- `UID` et `GID` sont des variables que vous trouverez dans toutes mes images, qui permet de choisir avec quels droits sera lancée l'application.
- `WEBROOT` est une variable qui permettra la modification du webroot du fichier de configuration de l'application, donc ici nous y accéderons via <http://XX.XX.XX.XX:8181/images>.
- `SECRET` est une variable qui permettra la modification du secret du fichier de configuration de l'application. Ces variables sont spécifiques à l'image

Nous pouvons vérifier les variables d'environnement via `docker container inspect lutim`, mais cette commande retourne toute la configuration de notre conteneur, nous allons donc le formater :

```
1. $ docker container inspect -f '{{.Config.env}}' lutim
2. [UID=1000 GID=1000 SECRET=mysecretcookie WEBROOT=/images PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin CONTACT=contact@domain.tld MAX_FILE_SIZE=1000000000 DEFAULT_DELAY=1 MAX_DELAY=0]
```

Nous avons ici également des variables que nous n'avons pas indiquées lors du lancement du conteneur, mais c'est normal, lors de la création d'une image, nous pouvons mettre des valeurs par défaut (nous verrons également ceci dans la partie [Créer une image](#)).

Puisque tout semble OK, on teste donc avec <http://XX.XX.XX.XX:8181/images>. Bon je ne vais pas rentrer dans les détails de fonctionnement de lutim, mais on voit que ça fonctionne.

Comme vous pouvez le voir, en quelques secondes nous avons installé un lutim, alors qu'il est normalement plus compliqué et plus long de le faire manuellement.

Cette partie ne vous apprendra rien de technique, je suppose, mais c'est simplement pour vous montrer ce que docker peut vous apporter si vous n'avez pas forcément la technique pour le faire à la main, ou tout simplement pour tester une application.

## VI-D-3 - Troisième cas : le déploiement

Dans ce troisième cas, nous allons partir sur quelque chose d'un peu plus complexe, et spécifique. Nous allons ici voir comment faire un déploiement en blue/green (version simplifiée), c'est-à-dire sans interruption de service (ou très peu, moins d'une seconde).

Dans ce scénario, nous aurons trois conteneurs, un conteneur nginx qui servira de reverse proxy, et deux conteneurs nodejs avec des versions différentes.

Nous allons donc reprendre notre code de tout à l'heure et lancer avec node5 et node6 :

```
1. $ docker container run -d -v $(pwd):/usr/src/app -w /usr/src/app -p 8001:8000 --name node-blue
   xataz/node:6 node app.js
2. e2a392d5b0ee7c65683dc277eb47c67dd93804ef36458968b2e5d34afc154957
3. $ docker container run -d -v $(pwd):/usr/src/app -w /usr/src/app -p 8002:8000 --name node-
   green xataz/node:7 node app.js
4. 18ff8c5b4c4d9c37cd2ee14eadd75e4addc10e04324cd513c77ae55b4912b042
```

node-blue est actuellement notre production, et node-green notre machine de test. Nous appellerons ceci des branches. Notre but est donc de mettre à jour notre node de la version 6 à la version 7, en s'assurant que tout fonctionne correctement.

Pour cela nous utiliserons nginx en reverse proxy. Nous commençons par créer notre fichier de configuration nginx :

```
1. # mkdir -p /docker/config/nginx
2. # vim /docker/config/nginx/bluegreen.conf
3. server {
4.     listen 8080;
5.
6.     location / {
7.         proxy_pass http://toto:8000;
8.     }
9. }
```

On part sur un fichier de configuration plutôt simple. Pour vous expliquer rapidement, tout ce qui arrivera sur le port 8080 sera retransmis au conteneur node-blue qui répondra à nginx qui nous le retransmettra. Nous utilisons ici directement le port de l'application, puisque nous « attaquons » directement le conteneur. Nous verrons juste en dessous à quoi correspond le toto.

Puis on lance notre nginx :

```
$ docker container run -d -v /docker/config/nginx:/sites-enabled -p 80:8080 --name reverse --link
   node-blue:toto --link node-green:tata xataz/nginx:mainline
```

Nous voyons ici un nouveau paramètre, le `--link`, celui-ci permet de créer un alias, au sein du conteneur lancé, afin de communiquer avec un autre conteneur, via cet alias. `toto` est le nom de l'alias qui pointe vers le conteneur node-blue, c'est donc identique avec `tata` et `node-green`. J'ai volontairement appelé les alias comme ceci, pour différencier le nom du conteneur et l'alias.

Si nous testons notre appli, avec l'URL <http://XX.XX.XX.XX>, nous devrions avoir affiché :

```
Hello Mondedie avec la version v6.10.3
```

Maintenant que j'ai bien testé mon application sur node.js 7 (via l'URL <http://XX.XX.XX.XX:8002>), je peux facilement faire un basculement de branche, il me suffit de modifier le fichier de configuration de nginx, et de relancer le conteneur :

```
1. # vim /docker/config/nginx/bluegreen.conf
2. server {
3.     listen 8080;
4.
5.     location / {
6.         proxy_pass http://tata:8000;
7.     }
8. }
```

On relance nginx :

```
1. $ docker restart reverse
2. reverse
```

Et on reteste la même URL ( <http://XX.XX.XX.XX>), nous avons maintenant la version 6 de node :

```
Hello Mondedie avec la version v7.10.0
```

Maintenant, node-green est devenu notre production, et node-blue notre dev dans laquelle nous testerons la version 8 de node (par exemple). Et quand celle-ci sera prête, nous referons un basculement de branche sur notre nginx.

Bien sûr, ceci n'est qu'une ébauche du basculement blue/green, mais le principe est là. Nous pourrions améliorer ceci en utilisant un réseau docker, que nous verrons dans un prochain chapitre, ou avec l'utilisation d'un serveur DNS interne à notre réseau de conteneur.

## VI-E - Conclusion

Cette partie fut plus concrète que les précédentes, nous savons maintenant comment créer un conteneur, et le gérer. À partir de ce moment, vous êtes totalement capable d'installer une application via docker.

## VII - Créer une image

Utiliser des images docker c'est bien, mais les fabriquer soi-même c'est mieux. Nous avons plusieurs façons de faire une image. Nous pouvons le faire à partir d'un conteneur existant (via `docker save`), facile à mettre en place, mais compliqué à maintenir. From scratch, complexe, et difficile à maintenir. Puis via un dockerfile, un fichier qui comporte les instructions de la création de l'image (la recette), en se basant sur une image existante, c'est la meilleure méthode, et c'est facile à maintenir.

### VII-A - Création d'un Dockerfile

#### VII-A-1 - Créons une image apache

Le Dockerfile (toujours avec une majuscule) est un fichier qui contient toutes les instructions pour créer une image, comme des métadonnées (Mainteneur, label, etc.), ou même les commandes à exécuter pour installer un logiciel.

Voici la liste des instructions d'un Dockerfile :

```
1. FROM # Pour choisir l'image sur laquelle on se base, toujours en premier
2. RUN # Permet d'exécuter une commande
```

```
3. CMD # Commande exécutée au démarrage du conteneur par défaut
4. EXPOSE # Ouvre un port
5. ENV # Permet d'éditer des variables d'environnement
6. ARG # Un peu comme ENV, mais seulement le temps de la construction de l'image
7. COPY # Permet de copier un fichier ou répertoire de l'hôte vers l'image
8. ADD # Permet de copier un fichier de l'hôte ou depuis une URL vers l'image, permet également de décompresser une archive tar
9. LABEL # Des métadonnées utiles pour certains logiciels de gestion de conteneurs, comme rancher ou swarm, ou tout simplement pour mettre des informations sur l'image.
10. ENTRYPOINT # Commande exécutée au démarrage du conteneur, non modifiable, utilisée pour package une commande
11. VOLUME # Crée une partition spécifique
12. WORKDIR # Permet de choisir le répertoire de travail
13. USER # Choisit l'utilisateur qui lance la commande du ENTRYPOINT ou du CMD
14. ONBUILD # Crée un step qui sera exécuté seulement si notre image est choisie comme base
15. HEALTHCHECK # Permet d'ajouter une commande pour vérifier le fonctionnement de votre conteneur
16. STOPSIGNAL # permet de choisir le [signal] (http://man7.org/linux/man-pages/man7/signal.7.html) qui sera envoyé au conteneur lorsque vous ferez un docker container stop
```

Pour plus d'informations, je vous conseille de consulter la [documentation officielle](#)

Et la commande pour construire l'image :

```
docker image build -t [imagename]:[tag] [dockerfile folder]
```

Pour pouvoir construire une image, il faut connaître un minimum le logiciel que l'on souhaite conteneuriser, par exemple ici je vais conteneuriser apache, et je sais qu'il lui faut certaines variables d'environnement pour fonctionner. On commence par créer le répertoire de notre projet :

```
1. $ mkdir /home/xataz/superapache
2. $ cd /home/xataz/superapache
3. $
```

Puis on crée notre Dockerfile (le D toujours en majuscule) :

```
$ vim Dockerfile
```

On commence par mettre le FROM :

```
FROM ubuntu
```

Puis on ajoute les variables d'environnement (qui sont normalement gérées par le système d'init) :

```
1. ENV APACHE_RUN_USER www-data
2. ENV APACHE_RUN_GROUP www-data
3. ENV APACHE_LOG_DIR /var/web/log/apache2
4. ENV APACHE_PID_FILE /var/run/apache2.pid
5. ENV APACHE_RUN_DIR /var/run/apache2
6. ENV APACHE_LOCK_DIR /var/lock/apache2
```

On installe apache :

```
RUN export DEBIAN_FRONTEND=noninteractive && apt-get update && apt-get -y -q upgrade && apt-get -y -q install apache2
```

Dans un Dockerfile, il ne faut aucune interactivité, donc on utilise : `DEBIAN_FRONTEND=noninteractive` et surtout l'option `-y` de `apt-get`

On expose les ports (facultatif) :

```
EXPOSE 80 443
```

Et pour finir, on ajoute la commande par défaut :

```
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

le Dockerfile au complet :

```
1. FROM ubuntu
2.
3. ENV APACHE_RUN_USER www-data
4. ENV APACHE_RUN_GROUP www-data
5. ENV APACHE_LOG_DIR /var/web/log/apache2
6. ENV APACHE_PID_FILE /var/run/apache2.pid
7. ENV APACHE_RUN_DIR /var/run/apache2
8. ENV APACHE_LOCK_DIR /var/lock/apache2
9.
10. RUN export DEBIAN_FRONTEND=noninteractive && apt-get update && apt-get -y -q upgrade && apt-
    get -y -q install apache2
11.
12. EXPOSE 80 443
13.
14. CMD ["apache2ctl", "-D", "FOREGROUND"]
```

Puis on construit notre image :

```
1. $ docker image build -t xataz/superapache .
2. Sending build context to Docker daemon 3.072 kB
3. Step 0 : FROM ubuntu
4. latest: Pulling from library/ubuntu
5. d3a1f33e8a5a: Pull complete
6. c22013c84729: Pull complete
7. d74508fb6632: Pull complete
8. 91e54dfb1179: Pull complete
9. [...]
10. Step 10 : CMD apache2ctl -D FOREGROUND
11. ---> Running in d435f9e2db87
12. ---> 1f9e7590d11b
13. Removing intermediate container d435f9e2db87
14. Successfully built 1f9e7590d11b
```

On peut la tester :

```
1. $ docker container run -ti -p 80:80 xataz/superapache
2. AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using
   172.17.0.2. Set the 'ServerName' directive globally to suppress this message
```

Si on se connecte maintenant au site, vous devriez avoir la page *Apache2 Ubuntu Default Page*.

## VII-A-2 - Exemple d'une image lutim

Nous allons ici créer une image lutim, basée sur le tuto de **solinvictus**. Pour ceci, nous nous baserons sur Debian Jessie.

On commence par créer un dossier pour notre image :

```
$ mkdir lutim
```

Voici le Dockerfile :

```
1. FROM debian:jessie
```

```

2.
3. ENV GID=991
4. ENV UID=991
5. ENV CONTACT=contact@domain.tld
6. ENV WEBROOT=/
7. ENV SECRET=e7c0b28877f7479fe6711720475dcbbd
8. ENV MAX_FILE_SIZE=1000000000
9.
10. LABEL description="lutim based on debian" \
11.     maintainer="xataz <https://github.com/xataz>"
12.
13. RUN apt-get update && apt-get install -y --no-install-recommends --no-install-suggests perl
    ca-certificates shared-mime-info perlmagick make gcc ca-certificates libssl-dev git
14. RUN cpan install Carton
15. RUN cd / && git clone https://git.framasoft.org/luc/lutim.git
16. RUN cd /lutim && carton install
17.
18. VOLUME /lutim/files /data
19.
20. EXPOSE 8181
21.
22. COPY lutim.conf /lutim/lutim.conf
23. COPY startup /usr/bin/startup
24. RUN chmod +x /usr/bin/startup
25.
26. CMD ["startup"]

```

C'est plutôt simpliste, j'ai suivi exactement le tutoriel. J'y ai ajouté des variables d'environnement qui seront utilisées par le script `startup` afin de générer le fichier `lutim.conf`. Je crée deux volumes, `/lutim/files` qui contiendra les images hébergées, et `/data` qui contient la base de données de lutim. Le port exposé est le 8181.

On écrit donc le `lutim.conf` :

```

1. {
2.     hypnotoad => {
3.         listen => ['http://0.0.0.0:8181'],
4.     },
5.     contact      => '<contact>',
6.     secrets      => ['<secret>'],
7.     length       => 8,
8.     crypto_key_length => 8,
9.     provis_step  => 5,
10.    provisioning  => 100,
11.    anti_flood_delay => 5,
12.    max_file_size  => <max_file_size>,
13.    default_delay  => 1,
14.    max_delay      => 0,
15.    always_encrypt => 1,
16.    token_length   => 24,
17.    stats_day_num  => 365,
18.    keep_ip_during => 365,
19.    policy_when_full => 'warn',
20.    #broadcast_message => 'Maintenance',
21.    prefix          => '<webroot>',
22.    db_path         => '/data/lutim.db',
23.    delete_no_longer_viewed_files => 90
24. };

```

Ce fichier est presque un copier-coller de celui du tutoriel. Toutes les valeurs entre `<>` seront remplacées avec les variables d'environnement par le script `startup`.

Voici donc le `startup` :

```

1. #!/bin/bash
2.
3. grep lutim /etc/group > /dev/null 2>&1; [[ $? -eq 1 ]] && addgroup --gid ${GID} lutim

```

```
4. grep lutim /etc/passwd > /dev/null 2>&1; [[ $? -eq 1 ]] && adduser --system --shell /bin/sh --
no-create-home --ingroup lutim --uid ${UID} lutim
5.
6. chown -R lutim:lutim /data /lutim
7.
8. sed -i -e 's|<secret>|'${SECRET}'|' \
9.      -e 's|<contact>|'${CONTACT}'|' \
10.     -e 's|<max_file_size>|'${MAX_FILE_SIZE}'|' \
11.     -e 's|<webroot>|'${WEBROOT}'|' /lutim/lutim.conf
12.
13.
14. su - lutim -c "cd /lutim; /usr/local/bin/carton exec hypnotoad -f /lutim/script/lutim"
```

Le script est plutôt simpliste, il crée un utilisateur et un groupe lutim, puis lui donne les droits au répertoire /lutim et /data, ensuite il modifie le fichier de conf avec les bonnes valeurs et exécute lutim avec l'utilisateur lutim.

On peut tenter de construire l'image :

```
$ docker image build -t xataz/lutim .
```

ça va prendre un petit moment, c'est plutôt long à installer les dépendances.

Testons notre image :

```
1. $ docker container run -d -P xataz/lutim
2. bb40fd7df491b224a73146981fff831f9bc5d61efde8c040cd48fa2418450a54
3. $ docker ps
4. CONTAINER ID      IMAGE          COMMAND                  CREATED             STATUS
   PORTS            NAMES
5. bb40fd7df491     xataz/lutim   "startup"               2 seconds ago      Up 1 seconds
   0.0.0.0:32770->8181/tcp    suspicious_hamilton
```

J'ai ici utilisé l'option `-P`, qui permet d'attribuer un port disponible à tous les ports EXPOSE, ici 32770->8181.

On teste dans le navigateur ( <http://XX.XX.XX.XX:32770>), et normalement, ça fonctionne.

## VII-A-3 - Créons une image de base

Nous allons ici créer une image de base, c'est-à-dire une image qui servira pour créer une autre image, comme debian, ubuntu ou alpine.

Ici nous créerons une image alpine.

On crée le répertoire de notre projet :

```
1. $ mkdir alpine
2. $ cd alpine
```

On commence par créer un rootfs, pour ceci nous utiliserons l'outil officiel de alpine, c'est-à-dire apk (version actuelle 2.6.8-r2, à vérifier au moment de la lecture [ici](#)), la méthode est différente pour chaque distribution (pour debian c'est debootstrap, pour archlinux ou gentoo on télécharge directement le rootfs, etc.), il faut par contre faire ceci en root :

```
1. $ wget http://dl-cdn.alpinelinux.org/alpine/latest-stable/main/x86_64/apk-tools-static-2.6.7-r0.apk
2. $ tar xzvf apk-tools-static-2.6.7-r0.apk
3. .SIGN.RSA.alpine-devel@lists.alpinelinux.org-4a6a0840.rsa.pub
4. .PKGINFO
5. sbin/
6. tar: Ignoring unknown extended header keyword 'APK-TOOLS.checksum.SHA1'
7. sbin/apk.static.SIGN.RSA.alpine-devel@lists.alpinelinux.org-4a6a0840.rsa.pub
```



```

8. tar: Ignoring unknown extended header keyword 'APK-TOOLS.checksum.SHA1'
9. $ ./sbin/apk.static -X http://dl-cdn.alpinelinux.org/alpine/3.4/main -U --allow-untrusted --
   root rootfs --initdb add alpine-base
10. fetch http://dl-cdn.alpinelinux.org/alpine/latest-stable/main/x86_64/APKINDEX.tar.gz
11. (1/16) Installing musl (1.1.14-r11)
12. (2/16) Installing busybox (1.24.2-r11)
13. Executing busybox-1.24.2-r11.post-install
14. (3/16) Installing alpine-baselayout (3.0.3-r0)
15. Executing alpine-baselayout-3.0.3-r0.pre-install
16. Executing alpine-baselayout-3.0.3-r0.post-install
17. (4/16) Installing openrc (0.21-r2)
18. Executing openrc-0.21-r2.post-install
19. (5/16) Installing alpine-conf (3.4.1-r2)
20. (6/16) Installing zlib (1.2.8-r2)
21. (7/16) Installing libcrypto1.0 (1.0.2h-r1)
22. (8/16) Installing libssl1.0 (1.0.2h-r1)
23. (9/16) Installing apk-tools (2.6.7-r0)
24. (10/16) Installing busybox-suid (1.24.2-r11)
25. (11/16) Installing busybox-initscripts (3.0-r3)
26. Executing busybox-initscripts-3.0-r3.post-install
27. (12/16) Installing scanelf (1.1.6-r0)
28. (13/16) Installing musl-utils (1.1.14-r11)
29. (14/16) Installing libc-utils (0.7-r0)
30. (15/16) Installing alpine-keys (1.1-r0)
31. (16/16) Installing alpine-base (3.4.3-r0)
32. Executing busybox-1.24.2-r11.trigger
33. OK: 7 MiB in 16 packages

```

Notre rootfs est maintenant créé :

```

1. $ ls
2. apk-tools-static-2.6.7-r0.apk  rootfs                                sbin
3. $ ls rootfs/
4. bin      dev      etc.      home      lib      linuxrc  media    mnt      proc      root      run
   sbin     srv      sys      tmp      usr      var

```

Nous n'avons plus besoin de apk, on le supprime donc :

```
$ rm -rf apk-tools-static-2.6.7-r0.apk sbin
```

Afin de gagner de l'espace, nous créons une archive de rootfs, et nous supprimons le dossier :

```

1. $ tar czf rootfs.tar.gz -C rootfs .
2. $ rm -rf rootfs

```

Pour finir on crée notre Dockerfile :

```

1. FROM scratch
2.
3. ADD rootfs.tar.gz /

```

scratch étant une image spécifique qui est vide, spécialement pour créer une image de base, ou à partir de rien.

On construit l'image et on teste :

```

1. $ docker image build -t superalpine .
2. Sending build context to Docker daemon 3.342 MB
3. Step 1 : FROM scratch
4. --->
5. Step 2 : ADD rootfs.tar.gz /
6. ---> fa8cc3e04a21
7. Removing intermediate container 385884a4f0c5
8. Successfully built fa8cc3e04a21
9. $ docker container run -ti superalpine /bin/sh
10. / # ls

```

```

11. bin      dev      etc.      home      lib      linuxrc  media      mnt      proc      root
    run      sbin     srv       sys       tmp      usr       var
  
```

Nous pouvons bien sûr l'améliorer, comme ajouter des dépôts, des paquets, une commande par défaut, etc. :

```

1. FROM scratch
2.
3. ADD rootfs.tar.gz /
4.
5. RUN echo "http://dl-cdn.alpinelinux.org/alpine/v3.4/main" > /etc/apk/repositories
6.
7. RUN apk add -U wget git
8.
9. CMD "/bin/sh"
  
```

Puis on reconstruit et teste :

```

1. $ docker image build -t superalpine .
2. [...]
3. $ docker container run -ti superalpine
4. / # ls
5. bin      dev      etc.      home      lib      linuxrc  media      mnt      proc      root      run
   sbin     srv       sys       tmp      usr       var
  
```

Et voilà vous avez votre image qui pourra vous servir de base à toutes vos autres images.

## VII-B - Les bonnes pratiques

Cette partie vous donnera des conseils pour optimiser vos images. Je ne prétends pas avoir la science infuse, et ces astuces/conseils sont plutôt personnels, mais je pense que ce sont de bonnes pratiques.

### VII-B-1 - Limiter les layers



#### Qu'est-ce qu'un layer

Les images docker sont créées avec des couches de filesystems, chaque instruction d'un dockerfile est une couche (chaque étape d'un build). Ces couches sont des layers.

Reprenons notre image de base :

```

1. FROM scratch
2.
3. ADD rootfs.tar.gz /
4.
5. RUN echo "http://dl-cdn.alpinelinux.org/alpine/v3.4/main" > /etc/apk/repositories
6.
7. RUN apk add -U wget git
8.
9. CMD "/bin/sh"
  
```

Nous avons ici cinq étapes.

- Étape 1 : je pars sur la base d'une image vide => Layer 1 ;
- Étape 2 : le layer 1 passe en lecture, je crée un layer 2 et copie le contenu de *rootfs.tar.gz* dedans. => layer 2 ;
- Étape 3 : le layer 2 passe en lecture, je crée un layer 3 et crée le fichier */etc/apk/repositories*. => layer 3 ;
- Étape 4 : le layer 3 passe en lecture, j'installe *wget* et *git*. => layer 4 ;

- Étape 5 : le layer 4 passe en lecture, j'ajoute une commande par défaut. => layer 5.

Chaque layer comporte les modifications apportées par rapport au layer précédent. Si j'ajoute une étape entre la 4 et 5 pour supprimer wget et git. Lors du build je repars du cache de l'étape 4, par contre l'étape 5 sera rejouée, puisque j'ai ajouté une étape entre les deux, et son layer précédent n'est maintenant plus disponible. Chaque layer est donc le différentiel du layer précédent.

Lorsque l'on crée un conteneur, on crée une nouvelle couche sur l'image, qui est en écriture, les couches précédentes ne sont qu'en lecture. C'est ce qui permet d'utiliser la même image pour plusieurs conteneurs, sans perte d'espace.

Mais multiplier les layers diminue les performances en lecture, en effet, admettons que dans un des premiers layers, vous installez wget, et que vous avez 50 layers après. Dans votre conteneur, quand vous demanderez d'exécuter wget, il passera par chaque layer pour chercher cette commande. Dans notre exemple, si je demande à mon conteneur d'exécuter wget, il va d'abord le chercher dans le layer 6 (le fs du conteneur), il n'est pas ici, donc on passe au layer 5, mais il n'est pas ici, donc il cherche dans le layer 4 et là, il le trouve. Dans une image qui comporte une vingtaine de layers, les performances ne sont pas trop impactées, mais avec une centaine de layers à remonter, cela se sent.

Pour corriger cela, on met plusieurs commandes dans la même étape :

```
1. FROM scratch
2.
3. ADD rootfs.tar.gz /
4.
5. RUN echo "http://dl-cdn.alpinelinux.org/alpine/v3.4/main" > /etc/apk/repositories && apk add
  -U wget git
6.
7. CMD "/bin/sh"
```

## VII-B-2 - Limiter la taille d'une image

Plusieurs pistes pour diminuer la taille d'une image :

- utiliser une image de base minimaliste comme alpine par exemple (5 Mo pour alpine contre 120 Mo pour Debian) ;
- supprimer le cache des gestionnaires de paquets ou autres applications ;
- désinstaller les applications qui ne sont plus utiles.

Pour la première étape pas de souci, il faut juste changer l'image de base, ou pas, c'est au choix.

Pour les deux autres étapes, c'est encore une histoire de layers.

Reprenons notre exemple, nous allons supprimer git et wget, de deux manières différentes :

```
1. FROM scratch
2.
3. ADD rootfs.tar.gz /
4.
5. RUN echo "http://dl-cdn.alpinelinux.org/alpine/v3.4/main" > /etc/apk/repositories && apk add
  -U wget git
6. RUN apk del wget git && rm -rf /var/cache/apk/*
7.
8. CMD "/bin/sh"
```

Je la construis en la nommant superalpine1 :

```
$ docker image build -t superalpine1 .
```

Puis je crée un superalpine2 :

```

1. FROM scratch
2.
3. ADD rootfs.tar.gz /
4.
5. RUN echo "http://dl-cdn.alpinelinux.org/alpine/v3.4/main" > /etc/apk/repositories && apk add
   -U wget git && apk del wget git && rm -rf /var/cache/apk/*
6.
7. CMD "/bin/sh"

```

Que je construis en superalpine2 :

```
$ docker image build -t superalpine2 .
```

Regardons la différence maintenant :

```

1. $ docker images "superalpine*"

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
superalpine2	latest	7808b8c75444	5 minutes ago	4.813 MB
superalpine1	latest	886e22de4865	5 minutes ago	23.54 MB



*Pourquoi superalpine1 est-il plus lourd que superalpine2 ? Comme précédemment dit, c'est une histoire de layers. Sur superalpine1, nous avons supprimé les paquets depuis un autre layer, ce qui fait que wget et git (ainsi que les dépendances) sont toujours présents dans le layer précédent, et donc utilisent de l'espace. Ce nouveau layer indique simplement que tel ou tel fichier a été supprimé.*

## VII-B-3 - La lisibilité

Nous avons vu précédemment qu'il fallait limiter les layers, le problème c'est que cela peut vite devenir illisible. Ce que je conseille (ce n'est pas une obligation), c'est de faire une commande par ligne, pour notre superalpine, cela donnerait ceci :

```

1. FROM scratch
2.
3. ADD rootfs.tar.gz /
4.
5. RUN echo "http://dl-cdn.alpinelinux.org/alpine/v3.4/main" > /etc/apk/repositories \
6.   && apk add -U wget \
7.   git \
8.   && apk del wget \
9.   git \
10.  && rm -rf /var/cache/apk/*
11.
12. CMD "/bin/sh"

```

Cela permet de voir en un seul coup d'œil, les différentes commandes. Ne surtout pas oublier le caractère d'échappement \ pour chaque nouvelle ligne.

Un autre exemple, en reprenant notre lutim, et en appliquant les précédentes règles :

```

1. FROM debian:jessie
2.
3. ENV GID=991 \
4.   UID=991 \
5.   CONTACT=contact@domain.tld \
6.   WEBROOT=/ \
7.   SECRET=e7c0b28877f7479fe6711720475dcbbd \

```

```

8.     MAX_FILE_SIZE=1000000000
9.
10.  LABEL description="lutim based on debian" \
11.      maintainer="xataz <https://github.com/xataz>"
12.
13.  RUN apt-get update \
14.      && apt-get install -y --no-install-recommends --no-install-suggests perl \
15.                                          ca-certificates \
16.                                          shared-mime-info \
17.                                          perlmagick \
18.                                          make \
19.                                          gcc \
20.                                          ca-certificates \
21.                                          libssl-dev \
22.                                          git \
23.      && cpan install Carton \
24.      && cd / \
25.      && git clone https://git.framasoft.org/luc/lutim.git \
26.      && cd /lutim \
27.      && carton install \
28.      && apt-get purge -y make \
29.                                          gcc \
30.                                          ca-certificates \
31.                                          libssl-dev \
32.                                          git \
33.      && apt-get autoremove --purge -y \
34.      && apt-get clean \
35.      && rm -rf /var/lib/apt/lists/* /root/.cpan* /lutim/local/cache/* /lufi/utilities
36.
37.  VOLUME /lutim/files /data
38.
39.  EXPOSE 8181
40.
41.  COPY lutim.conf /lutim/lutim.conf
42.  COPY startup /usr/bin/startup
43.  RUN chmod +x /usr/bin/startup
44.
45.  CMD ["startup"]

```

Si on tente de construire cette image :

```
$ docker image build -t lutim2 .
```

Ce qui donne pour comparer :

1. \$ docker images				
2. lutim2	latest	ae75fa0elf8a	2 minutes ago	252.6 MB
3. xataz/lutim	latest	3d72d9158f68	17 hours ago	577.1 MB

Nous avons diminué de moitié la taille de l'image.

## VII-B-4 - Éviter les processus root

Ici c'est plus de l'administration qu'une spécificité docker.

Nous avons bien sûr la possibilité d'utiliser l'instruction USER directement dans un dockerfile, mais ceci comporte un défaut, c'est qu'on peut avoir besoin de root pour exécuter un script, pour par exemple créer un utilisateur, modifier des permissions, etc.

Pour pallier ceci, nous utiliserons `su-exec` et `exec`, tout se jouera dans le script de démarrage.

- `su-exec` est un outil qui fait la même chose que `su`, mais à la différence que `su-exec` ne crée qu'un seul processus. `su-exec` s'utilise de manière très simple : `su-exec <user>:<group> <command>`.

- `exec` est une fonction, qui permet d'exécuter une commande afin qu'elle remplace le PID actuel. `exec` s'utilise comme ceci : `exec <command>`

Pour l'exemple, reprenons notre script startup de lutim :

```
1. #!/bin/bash
2.
3. grep lutim /etc/group > /dev/null 2>&1; [[ $? -eq 1 ]] && addgroup --gid ${GID} lutim
4. grep lutim /etc/passwd > /dev/null 2>&1; [[ $? -eq 1 ]] && adduser --system --shell /bin/sh --
no-create-home --ingroup lutim --uid ${UID} lutim
5.
6. chown -R lutim:lutim /data /lutim
7.
8. sed -i -e 's|<secret>|${SECRET}||' \
9.     -e 's|<contact>|${CONTACT}||' \
10.    -e 's|<max_file_size>|${MAX_FILE_SIZE}||' \
11.    -e 's|<webroot>|${WEBROOT}||' /lutim/lutim.conf
12.
13.
14. su - lutim -c "cd /lutim; /usr/local/bin/carton exec hypnotoad -f /lutim/script/lutim"
```

Avec ceci, nous aurons deux processus root :

```
1. $ docker container run -d --name lutim lutim
2. $ docker container top lutim
3. UID PID PPID C STIME
   TTY TIME CMD
4. root 22898 22887 0 11:50
   ? 00:00:00 /bin/bash /usr/bin/startup
5. root 22942 22898 0 11:50
   ? 00:00:00 su - lutim -c cd /lutim; /usr/local/bin/carton exec
hypnotoad -f /lutim/script/lutim
6. 991 22947 22942 0 11:50
   ? 00:00:00 -su -c cd /lutim; /usr/local/bin/carton exec
hypnotoad -f /lutim/script/lutim
7. 991 22950 22947 9 11:50
   ? 00:00:00 /lutim/script/lutim
8. 991 22951 22950 0 11:50
   ? 00:00:00 /lutim/script/lutim
9. 991 22952 22950 0 11:50
   ? 00:00:00 /lutim/script/lutim
10. 991 22953 22950 0 11:50
   ? 00:00:00 /lutim/script/lutim
11. 991 22954 22950 0 11:50
   ? 00:00:00 /lutim/script/lutim
12. $ docker container rm lutim
```

Sous debian, malheureusement il n'y a pas de paquet précompilé, il va donc falloir le faire à la main, ci-dessous le dockerfile avec l'ajout de l'installation de `su-exec` :

```
1. FROM debian:jessie
2.
3. ENV GID=991 \
4.     UID=991 \
5.     CONTACT=contact@domain.tld \
6.     WEBROOT=/ \
7.     SECRET=e7c0b28877f7479fe6711720475dcbbd \
8.     MAX_FILE_SIZE=1000000000
9.
10. LABEL description="lutim based on debian" \
11.     maintainer="xataz <https://github.com/xataz>"
12.
13. RUN apt-get update \
14.     && apt-get install -y --no-install-recommends --no-install-suggests perl \
15.                                     ca-certificates \
16.                                     shared-mime-info \
17.                                     perlmagick \
```

```

18.                                     make \
19.                                     gcc \
20.                                     ca-certificates \
21.                                     libssl-dev \
22.                                     git \
23.                                     libv6-dev \
24.     && cpan install Carton \
25.     && cd / \
26.     && git clone https://git.framasoft.org/luc/lutim.git \
27.     && git clone https://github.com/ncopa/su-exec \
28.     && cd /su-exec \
29.     && make \
30.     && cp su-exec /usr/local/bin/su-exec \
31.     && cd /lutim \
32.     && rm -rf /su-exec \
33.     && carton install \
34.     && apt-get purge -y make \
35.                                     gcc \
36.                                     ca-certificates \
37.                                     libssl-dev \
38.                                     git \
39.                                     libc6-dev \
40.     && apt-get autoremove --purge -y \
41.     && apt-get clean \
42.     && rm -rf /var/lib/apt/lists/* /root/.cpan* /lutim/local/cache/* /lufi/utilities
43.
44. VOLUME /lutim/files /data
45.
46. EXPOSE 8181
47.
48. COPY lutim.conf /lutim/lutim.conf
49. COPY startup /usr/bin/startup
50. RUN chmod +x /usr/bin/startup
51.
52. CMD ["startup"]

```

Nous commençons par remplacer notre su par su-exec dans le fichier startup :

```

1. #!/bin/bash
2.
3. grep lutim /etc/group > /dev/null 2>&1; [[ $? -eq 1 ]] && addgroup --gid ${GID} lutim
4. grep lutim /etc/passwd > /dev/null 2>&1; [[ $? -eq 1 ]] && adduser --system --shell /bin/sh --
no-create-home --ingroup lutim --uid ${UID} lutim
5.
6. chown -R lutim:lutim /data /lutim
7.
8. sed -i -e 's|<secret>|'${SECRET}'|' \
9.         -e 's|<contact>|'${CONTACT}'|' \
10.        -e 's|<max_file_size>|'${MAX_FILE_SIZE}'|' \
11.        -e 's|<webroot>|'${WEBROOT}'|' /lutim/lutim.conf
12.
13. cd /lutim
14. su-exec lutim /usr/local/bin/carton exec hypnotoad -f /lutim/script/lutim

```

Puis on reconstruit :

```
$ docker image build -t lutim .
```

On teste notre première modification :

```

1. $ docker container run -d --name lutim lutim
2. $ docker container top lutim
3. UID          PID          PPID          C          STIME
   TTY          TIME         CMD
4. root         2142         2129         0          13:03
   ?            00:00:00    /bin/bash /usr/bin/startup
5. 991          2183         2142         1          13:03
   ?            00:00:00    /lutim/script/lutim

```

```

6. 991          2185          2183          0          13:03
   ?           00:00:00      /lutim/script/lutim
7. 991          2186          2183          0          13:03
   ?           00:00:00      /lutim/script/lutim
8. 991          2187          2183          0          13:03
   ?           00:00:00      /lutim/script/lutim
9. 991          2188          2183          0          13:03
   ?           00:00:00      /lutim/script/lutim
10. $ docker container rm lutim
  
```

On voit qu'il nous reste un seul processus root.

Et si on ajoute, exec :

```

1. #!/bin/bash
2.
3. grep lutim /etc/group > /dev/null 2>&1; [[ $? -eq 1 ]] && addgroup --gid ${GID} lutim
4. grep lutim /etc/passwd > /dev/null 2>&1; [[ $? -eq 1 ]] && adduser --system --shell /bin/sh --
no-create-home --ingroup lutim --uid ${UID} lutim
5.
6. chown -R lutim:lutim /data /lutim
7.
8. sed -i -e 's|<secret>|${SECRET}||' \
9.      -e 's|<contact>|${CONTACT}||' \
10.     -e 's|<max_file_size>|${MAX_FILE_SIZE}||' \
11.     -e 's|<webroot>|${WEBROOT}||' /lutim/lutim.conf
12.
13. cd /lutim
14. exec su-exec lutim /usr/local/bin/carton exec hypnotoad -f /lutim/script/lutim
  
```

On reconstruit :

```
$ docker image build -t lutim .
```

Puis on teste :

```

1. $ docker container run -d --name lutim lutim
2. $ docker container top lutim
3. UID          PID          PPID          C          STIME
   TTY          TIME          CMD
4. 991          2312          2299          11          13:04
   ?           00:00:00      /lutim/script/lutim
5. 991          2353          2312          0          13:04
   ?           00:00:00      /lutim/script/lutim
6. 991          2354          2312          0          13:04
   ?           00:00:00      /lutim/script/lutim
7. 991          2355          2312          0          13:04
   ?           00:00:00      /lutim/script/lutim
8. 991          2356          2312          0          13:04
   ?           00:00:00      /lutim/script/lutim
9. $ docker container rm lutim
  
```

Et voilà, plus de processus root, un bon pas niveau sécurité.

## VII-C - Conclusion

Nous avons vu ici comment créer une image applicative, mais aussi comment faire une image de base. L'optimisation de l'image est importante, que ce soit pour la sécurité, ou pour la taille de celle-ci. Normalement à partir de ce moment, vous devriez pouvoir créer vos propres images, et de manière propre.



## VIII - Déployer/partager une image

Il existe trois façons de partager/déployer une image, la plus simple est sans doute de donner un dockerfile. Mais nous pouvons également l'envoyer sur le hub, ou tout simplement en envoyant une archive de l'image.

### VIII-A - Via un dockerfile

Je doute avoir vraiment besoin de l'expliquer, vous avez créé votre dockerfile, ainsi que les scripts qui vont avec, il vous suffit de l'envoyer à la personne que vous souhaitez, ou de le partager sur github par exemple.

### VIII-B - Via le docker hub

Je ne ferai pas une présentation complète du Hub, mais juste une explication sur comment envoyer votre image. Pour commencer, il faut se créer un compte sur le Hub, rendez-vous [ici](#). Une fois inscrit, on se connecte depuis notre docker :

```
1. $ docker login
2. Username: xataz
3. Password:
4. Email: xataz@mondedie.fr
5. WARNING: login credentials saved in /home/xataz/.docker/config.json
6. Login Succeeded
```

Maintenant que l'on est connecté, on peut pousser notre image, pour ce faire, il faut que votre image soit nommée sous cette forme : username/imagename:tag :

```
1. $ docker image push xataz/lutim:latest
2. The push refers to a repository [docker.io/xataz/lutim] (len: 1)
```

Et voilà, notre image est dans les nuages. Vous pouvez vous rendre sur votre Hub, et vous trouverez un nouveau repository. Vous pouvez donc ajouter une description et un mini tuto.

Bien évidemment, il existe d'autres options pour les envoyer sur le Hub, par exemple la possibilité de lier un repository à un github, ceci permet un build automatique (pratique, mais lent). Il est également possible de créer un hub privé (payant).

### VIII-C - Via une image tar

On peut également faire un tar d'une image, et ensuite la partager. Nous allons créer une image tar de notre lutim, pour ceci c'est plutôt simple :

```
1. $ docker image save -o lutim.tar xataz/lutim
2. $ ls
3. Dockerfile  lutim.conf  lutim.tar  startup
```

Et voilà on peut la partager.

L'avantage de cette méthode, c'est qu'il n'y a plus besoin de taper sur le hub pour installer l'image, car toutes les dépendances sont également dans le tar :

```
1. $ docker image rm -f $(docker images -q)
2. $ docker image ls -a
3. REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
4. $ docker image import lutim.tar xataz/lutim
5. sha256:a8dfc72dbef32cab0cd57a726f65c96c38f5e09736f46962bba7dbb3a86876d8
6. $ docker image ls
7. REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
```

```

8. xataz/lutim          latest          a8dfc72dbef3          6 seconds ago          598.2 MB
9. $ docker image history xataz/lutim
10. IMAGE              CREATED              CREATED BY              SIZE              COMMENT
11. a8dfc72dbef3       12 seconds ago
-

```

## VIII-D - Conclusion

Nous avons vu ici trois méthodes pour partager notre travail, bien évidemment celle à privilégier est celle du **Dockerfile**, qui permet de fournir la source de notre image.

La partager sur le dockerhub est bien, et si possible, il vaut mieux faire un autobuild (pas toujours facile).

## IX - Limiter les ressources d'un conteneur

### IX-A - La mémoire

Pour limiter la mémoire, nous utiliserons trois options :

- `-m --memory` : on choisit la limite de mémoire (ex : `-m 200M`) ;
- `--memory-swap` : on choisit la limite de mémoire + swap (ex : `-m 200M --memory-swap 1G`, ceci limite la mémoire à 200 M et le swap à 800 M) ;
- `--oom-kill-disable` : OOM pour Out Of Memory, permet d'éviter qu'une instance plante si elle dépasse l'allocation mémoire accordée, peut être pratique pour une application gourmande.

D'autres options existent,

```

1. $ docker container run --help | grep -Ei 'mem|oom'
2.      --cpuset-mems string          MEMS in which to allow execution (0-3, 0,1)
3.      --kernel-memory bytes        Kernel memory limit
4.      -m, --memory bytes            Memory limit
5.      --memory-reservation bytes    Memory soft limit
6.      --memory-swap bytes           Swap limit equal to memory plus swap: '-1' to enable
unlimited swap
7.      --memory-swappiness int       Tune container memory swappiness (0 to 100) (default
-1)
8.      --oom-kill-disable            Disable OOM Killer
9.      --oom-score-adj int           Tune host's OOM preferences (-1000 to 1000)

```

Pour tester la mémoire nous utiliserons l'image debian :

```

1. $ docker container run -ti --rm -m 500M debian
2. root@1044c936c209:/# free -h
3.      total          used          free          shared          buffers          cached
4. Mem:           23G           7.8G           15G           97M           457M           1.6G
5. -/+ buffers/cache:           5.8G           17G
6. Swap:           0B           0B           0B

```

On ne voit aucune différence avec la machine hôte, en fait la limitation se fait au niveau du processus du conteneur.

Nous utiliserons stress pour nos tests, nous allons stresser le conteneur avec 500 M, qui devraient représenter environ 50 % de la mémoire de ma machine :

```

1. root@1044c936c209:/# apt-get update && apt-get install stress
2. [...]
3. root@1044c936c209:/# stress --vm 1 --vm-bytes 500M &
4. [1] 54
5. root@1044c936c209:/# stress: info: [54] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
6.
7. root@1044c936c209:/# ps aux
8. USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND

```

```

9. root          1  0.0  0.1 20244 1644 ?      Ss   15:08   0:00 /bin/bash
10. root         54  0.0  0.0  7172    0 ?      S    15:12   0:00 stress --vm 1 --vm-bytes
500M
11. root         55 70.5 49.7 519176 506868 ?      R    15:12   0:04 stress --vm 1 --vm-bytes
500M
12. root         56  0.0  0.1 17500  1992 ?      R+   15:13   0:00 ps aux
13. root@1044c936c209:/# kill 54 55
14. root@1044c936c209:/#

```

On voit que c'est correct.

Maintenant testons avec 900 M de ram :

```

1. root@1044c936c209:/# stress --vm 1 --vm-bytes 900M &
2. [1] 68
3. root@1044c936c209:/# stress: info: [68] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
4.
5. root@1044c936c209:/# ps aux
6. USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
7. root           1  0.0  0.1 20244  1696 ?      Ss   15:08   0:00 /bin/bash
8. root          68  0.0  0.0  7172    0 ?      S    15:15   0:00 stress --vm 1 --vm-bytes 900M
9. root          69 77.6 47.2 928776 482072 ?      D    15:15   0:02 stress --vm 1 --vm-bytes 900M
10. root          70  0.0  0.1 17500  2016 ?      R+   15:15   0:00 ps aux
11. root@1044c936c209:/# kill 68 69

```

Comme on peut le voir, stress n'utilise que 47.2 % de la mémoire disponible sur l'hôte, alors qu'on lui a dit d'en utiliser environ 90 %. On voit donc que la limitation fonctionne.

Mais je sens des septiques parmi vous, nous allons tester cette même commande dans un conteneur non limité :

```

1. $ docker container run -ti --rm debian
2. root@e3ba516add96:/# apt-get update && apt-get install stress
3. [...]
4. root@e3ba516add96:/# stress --vm 1 --vm-bytes 900M &
5. [1] 51
6. root@e3ba516add96:/# stress: info: [51] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
7. root@e3ba516add96:/# ps aux
8. USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
9. root           1  0.0  0.1 20244  1584 ?      Ss   15:17   0:00 /bin/bash
10. root          51  0.0  0.0  7172    0 ?      S    15:18   0:00 stress --vm 1 --vm-bytes
900M
11. root          52 41.1 77.0 928776 785696 ?      D    15:18   0:03 stress --vm 1 --vm-bytes
900M
12. root          54  0.0  0.1 17500  1940 ?      R+   15:18   0:00 ps aux

```

## IX-B - Le CPU

Pour limiter le CPU nous utiliserons trois options :

- `-c-cpu-shares` : permet le partage des ressources CPU, c'est une proportion, si on met tous les conteneurs à 1024, ils se partageront équitablement les ressources, si un conteneur à 1024 et deux autres à 512, cela donnera 50 % pour le conteneur 1 et 25 % pour les deux autres ;
- `--cpu-quota` : permet la limitation de l'utilisation CPU (50000 pour 50 %, 0 pour pas de limite) ;
- `--cpuset-cpus` : permet de choisir les CPU/core utilisés (0,1 utilise les cpus 0 et 1, 0-2 utilise les cpus 0, 1, et 2)

Il existe d'autres options, que vous pouvez trouver avec `docker container run --help | grep -E 'cpu'` :

```

1.      --cpu-period int          Limit CPU CFS (Completely Fair Scheduler) period
2.      --cpu-quota int           Limit CPU CFS (Completely Fair Scheduler) quota
3.      --cpu-rt-period int       Limit CPU real-time period in microseconds
4.      --cpu-rt-runtime int      Limit CPU real-time runtime in microseconds
5.  -c, --cpu-shares int          CPU shares (relative weight)
6.      --cpus decimal            Number of CPUs

```

- |    |                      |   |
|----|----------------------|---|
| 7. | --cpuset-cpus string | CPUs in which to allow execution (0-3, 0,1) |
| 8. | --cpuset-mems string | MEMS in which to allow execution (0-3, 0,1) |

Nous allons également utiliser debian avec stress :

```
1. $ docker container run -ti --rm --cpuset-cpus 0 debian
2. root@0cfcada740e4:/# apt-get update && apt-get install stress
3. root@0cfcada740e4:/# stress -c 2 &
4. stress: info: [75] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd
5. root@0cfcada740e4:/# top
6. top - 23:10:14 up 2 days, 15:45, 0 users, load average: 1.09, 0.86, 0.44
7. Tasks: 4 total, 2 running, 2 sleeping, 0 stopped, 0 zombie
8. %Cpu(s): 50.0 us, 0.0 sy, 0.0 ni, 50.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
9. KiB Mem: 1019768 total, 214828 used, 804940 free, 1564 buffers
10. KiB Swap: 1186064 total, 789032 used, 397032 free. 30952 cached Mem
11.
12. PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
13. 76 root 20 0 7164 88 0 R 49.8 0.0 0:17.90 stress
14. 77 root 20 0 7164 88 0 R 50.0 0.0 0:17.90 stress
15. 1 root 20 0 20236 3272 2792 S 0.0 0.2 0:00.02 bash
16. 75 root 20 0 7164 868 788 S 0.0 0.0 0:00.00 stress
17. 78 root 20 0 21968 2416 2024 R 0.0 0.1 0:00.00 top
```

Comme on peut le voir, stress n'utilise qu'un cpu. Nous allons tester avec les deux cœurs :

```
1. $ docker container run -ti --rm --cpuset-cpus 0,1 debian
2. root@d19a45b0cb82:/# apt-get update && apt-get install stress
3. root@d19a45b0cb82:/# stress -c 2 &
4. [1] 60
5. stress: info: [60] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
6. root@d19a45b0cb82:/# top
7. top - 23:14:47 up 2 days, 15:50, 0 users, load average: 0.70, 0.64, 0.45
8. Tasks: 5 total, 3 running, 2 sleeping, 0 stopped, 0 zombie
9. %Cpu(s):100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
10. KiB Mem: 1019768 total, 214828 used, 804940 free, 1564 buffers
11. KiB Swap: 1186064 total, 789032 used, 397032 free. 30952 cached Mem
12.
13. PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
14. 62 root 20 0 7164 88 0 R 100.0 0.0 0:20.66 stress
15. 61 root 20 0 7164 88 0 R 99.9 0.0 0:20.66 stress
16. 1 root 20 0 20236 3168 2688 S 0.0 0.2 0:00.03 bash
17. 60 root 20 0 7164 864 784 S 0.0 0.0 0:00.00 stress
18. 63 root 20 0 21968 2460 2060 R 0.0 0.1 0:00.00 top
```

Maintenant on va tester l'option --cpu-quota :

```
1. $ docker container run -ti --cpu-quota 50000 --rm debian
2. root@1327f5aa6e13:/# apt-get update && apt-get install stress
3. root@1327f5aa6e13:/# stress -c 2 &
4. [1] 53
5. root@1327f5aa6e13:/# stress: info: [53] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
6.
7. root@1327f5aa6e13:/# top
8. top - 15:28:43 up 2 days, 5:46, 0 users, load average: 0.49, 0.23, 0.19
9. Tasks: 5 total, 3 running, 2 sleeping, 0 stopped, 0 zombie
10. %Cpu(s): 50.5 us, 0.0 sy, 0.0 ni, 49.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
11. KiB Mem: 1019768 total, 240840 used, 778928 free, 3208 buffers
12. KiB Swap: 1186064 total, 788908 used, 397156 free. 51508 cached Mem
13.
14. PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
15. 54 root 20 0 7172 88 0 R 25.0 0.0 0:05.70 stress
16. 55 root 20 0 7172 88 0 R 25.0 0.0 0:05.71 stress
17. 1 root 20 0 20244 3212 2728 S 0.0 0.3 0:00.02 bash
18. 53 root 20 0 7172 936 856 S 0.0 0.1 0:00.00 stress
19. 56 root 20 0 21988 2412 1964 R 0.0 0.2 0:00.00 top
```

On voit donc que c'est bien limité à 50 %.

## IX-C - L'écriture disque

Nous pouvons également limiter la vitesse d'écriture du disque, ceci grâce à une option :

- `--blkio-weight` : ceci est une priorisation des I/O. (un chiffre en 10 et 1000) ;
- `--blkio-weight-device` : pareil que la précédente, mais pour un device particulier (`--blkio-weight-device /dev/sda:100`) ;
- `--device-read-bps` : limite la vitesse de lecture sur un device par seconde. (`--device-read-bps /dev/sda:1mb`) ;
- `--device-read-iops` : limite les IO en lecture sur un device par seconde. (`--device-read-iops /dev/sda:1000`) ;
- `--device-write-bps` : limite la vitesse d'écriture sur un device par seconde. (`--device-write-bps /dev/sda:1mb`) ;
- `--device-write-iops` : limite les IO en écriture sur un device par seconde. (`--device-write-iops /dev/sda:1000`).

Sans vous montrez toutes les options, voici par exemple comment limiter l'écriture disque :

```
1. $ docker container run -ti --device-write-bps /dev/sda:1mb --rm debian bash
2. root@43677eafb3f6:/# time dd if=/dev/zero of=/tmp/test bs=1M count=10 oflag=direct
3. 10+0 records in
4. 10+0 records out
5. 10485760 bytes (10 MB) copied, 13.4905 s, 777 kB/s
6.
7. real    0m13.491s
8. user    0m0.000s
9. sys     0m0.000s
```

Et sans limiter l'écriture :

```
1. $ docker container run -ti --rm debian bash
2. root@cc2079d07eba:/# time dd if=/dev/zero of=/tmp/test bs=1M count=10 oflag=direct
3. 10+0 records in
4. 10+0 records out
5. 10485760 bytes (10 MB) copied, 0.0250823 s, 418 MB/s
6.
7. real    0m0.026s
8. user    0m0.000s
9. sys     0m0.003s
```

## X - Docker Volume

Depuis la version 1.9.0, docker a introduit une nouvelle sous-commande à docker, `docker volume`. Celle-ci permet de créer des volumes, facilement réutilisables pour plusieurs conteneurs, et depuis différentes sources avec une multitude de **plugins** (glusterfs, flocker, AFS etc.). N'hésitez pas à consulter la **documentation**.

Nous allons voir ici quelques possibilités que nous offre cette commande.

La syntaxe reste dans l'esprit de docker :

```
1. $ docker volume --help
2.
3. Usage:  docker volume COMMAND
4.
5. Manage Docker volumes
6.
7. Options:
8.     --help    Print usage
9.
10. Commands:
11.  create      Create a volume
12.  inspect     Display detailed information on one or more volumes
13.  ls          List volumes
14.  rm          Remove one or more volumes
15.
```

```
16. Run 'docker volume COMMAND --help' for more information on a command.
```

## X-A - Création d'un volume simple

Jusque-là, nous utilisons l'option `-v` avec un `docker run`, genre `docker run -d -v /path/on/host:/path/on/container image`.

Nous allons commencer par créer un volume, voyons ce que `docker volume create` prend comme arguments :

```
1. $ docker volume create --help
2.
3. Usage:  docker volume create [OPTIONS]
4.
5. Create a volume
6.
7. Options:
8.   -d, --driver string      Specify volume driver name (default "local")
9.   --help                  Print usage
10.  --label value            Set metadata for a volume (default [])
11.  --name string            Specify volume name
12.  -o, --opt value          Set driver specific options (default map[])
```

Donc on va créer notre premier volume :

```
1. $ docker volume create --name test
2. test
```

Que nous utiliserons comme ceci :

```
1. $ docker run -ti -v test:/test alpine:3.4 sh
2. / # ls /test
3. / # touch /test/bidule
4. / # exit
5. $
```

Si je crée un autre conteneur, on retrouve notre fichier bidule :

```
1. $ docker run -ti -v test:/test alpine:3.4 sh
2. / # ls /test
3. bidule
```

On retrouve bien notre fichier, les données ont donc correctement persisté.



*C'est bien beau, mais les fichiers, ils sont où sur l'hôte ?!*

Les fichiers se retrouvent dans `/var/lib/docker/volumes/<volumename>/_data`.

```
1. $ sudo ls /var/lib/docker/volumes/test/_data
2. bidule
```



*Et si je veux choisir où les mettre, comme un `-v /path:/path` ?*

C'est possible. Par exemple, je veux que mon volume `test` pointe vers `/data/test`, je peux le monter via cette commande :

```
$ docker volume create --name test -o type=none -o device=/data/test -o o=bind
```

Je recrée un conteneur où je vais créer un fichier :

```
1. $ docker run -ti -v test:/test alpine:3.4 touch /test/fichier  
2. $ sudo ls /data/test  
3. fichier
```

Et voilà, mon fichier est correctement dans /data/test.

## X-B - Un peu plus loin

Avec `docker volume`, il nous est possible de créer un volume via un device (par exemple /dev/sdb1).

Nous avons plein de possibilités.

Création d'un tmpfs, qui permettra de faire passer des données entre deux conteneurs, attention cependant, une fois ce volume non utilisé par un conteneur, les données sont effacées, puisqu'en ram :

```
1. $ docker volume create -o type=tmpfs -o device=tmpfs -o o=size=100M,uid=1000 --name tmpfile  
2. tmpfile
```

Ou alors monter une partition complète (attention, la partition ne doit pas être montée par l'hôte) :

```
1. $ docker volume create -o type=ext4 -o device=/dev/sdb1 --name extpart  
2. extpart
```

## X-C - Encore et toujours plus loin avec les plugins

Comme précédemment cité, il nous est possible d'ajouter des plugins à docker. Nous utiliserons ici le plugin **netshare**. Le plugin netshare permet l'utilisation de NFS, AWS EFS, Samba/CIFS et ceph.

Pour les besoins du test, j'ai créé deux machines de test, une sans docker avec un serveur nfs, et l'autre avec docker.

Ma machine servant de serveur de partage aura comme IP 10.2.81.71, et l'autre 10.2.155.205.

Je ne partirai pas sur l'explication de l'installation d'un serveur nfs, vous trouverez ceci [ici](#).

### X-C-1 - Prérequis

Il nous faudra nfs sur la machine docker :

```
$ apt-get install nfs-common
```

Comme indiqué dans le README du plugin, nous testons si notre partage fonctionne :

```
1. $ mount -t nfs 10.2.81.71:/shared /mnt  
2. $ root@scw-docker:~# ls /mnt  
3. fichier partager  
4. $ umount /mnt
```

c'est bon tout fonctionne.

## X-C-2 - Installation du plugin

Nous avons de la chance, il existe un dpkg pour debian et ubuntu, on l'installe donc comme ceci :

```
1. $ wget https://github.com/ContainX/docker-volume-netshare/releases/download/v0.20/docker-volume-netshare_0.20_amd64.deb
2. $ dpkg -i docker-volume-netshare_0.20_amd64.deb
```

Et c'est tout ^^.

## X-C-3 - Utilisation

Il faut d'abord lancer le daemon :

```
$ service docker-volume-netshare start
```

Puis on crée notre volume :

```
$ docker volume create -d nfs --name 10.2.81.71/shared
```

Puis on le monte :

```
1. $ docker run -i -t -v 10.2.81.71/shared:/mount xataz/alpine:3.4 sh
2. $ ls /mount
3. fichier_partager
```

Et voilà, nous voyons notre fichier partagé. Je n'ai absolument rien inventé ici, tout ce que j'ai écrit est indiqué dans le readme du plugin.

## X-D - Conclusion

Nous avons vu ici comment utiliser `docker volume`, et comment gérer plusieurs volumes, dans plusieurs conteneurs. Nous avons même vu comment installer un plugin (ici netshare), pour pouvoir étendre les possibilités de la gestion de volume.

## XI - docker network

Tout comme `docker volume`, `docker network` est apparu avec la version 1.9.0 de docker. Les networks ont plusieurs utilités, créer un réseau overlay entre plusieurs machines par exemple, ou alors remplacer les links en permettant à tous les conteneurs d'un même réseau de communiquer par leurs noms.

Voici la syntaxe :

```
1. $ docker network
2.
3. Usage:  docker network COMMAND
4.
5. Manage Docker networks
6.
7. Options:
8.     --help    Print usage
9.
10. Commands:
11.   connect    Connect a container to a network
12.   create     Create a network
13.   disconnect Disconnect a container from a network
```



```

14. inspect      Display detailed information on one or more networks
15. ls           List networks
16. rm           Remove one or more networks
17.
18. Run 'docker network COMMAND --help' for more information on a command.
```

## XI-A - Les types de réseaux

Nous avons de base, quatre types de networks :

- **bridge** : crée un réseau interne pour vos conteneurs ;
- **host** : ce type de réseau permet au conteneur d'avoir la même interface que l'hôte ;
- **none** : comme le nom l'indique, aucun réseau pour les conteneurs ;
- **overlay** : réseau interne entre plusieurs hôtes.

Bien évidemment il existe des **plugins** pour étendre ces possibilités.

Par défaut, nous avons déjà un réseau *bridge*, un réseau *host* et un réseau *none*. Nous ne pouvons pas créer de réseau *host* ou *none* supplémentaire. Ce chapitre expliquera donc l'utilisation des réseaux *bridge*. Pour le réseau *overlay*, ce type étant utilisé pour la communication interhôte, nous verrons ceci dans la partie swarm de ce tutoriel.

## XI-B - Création d'un network

Voici les arguments que prend `docker network create` :

```

1. $ docker network create --help
2.
3. Usage:  docker network create [OPTIONS] NETWORK
4.
5. Create a network
6.
7. Options:
8.     --aux-address value      Auxiliary IPv4 or IPv6 addresses used by Network driver (default
    map[])
9.     -d, --driver string      Driver to manage the Network (default "bridge")
10.    --gateway value          IPv4 or IPv6 Gateway for the master subnet (default [])
11.    --help                   Print usage
12.    --internal               Restrict external access to the network
13.    --ip-range value         Allocate container ip from a sub-range (default [])
14.    --ipam-driver string      IP Address Management Driver (default "default")
15.    --ipam-opt value         Set IPAM driver specific options (default map[])
16.    --ipv6                   Enable IPv6 networking
17.    --label value            Set metadata on a network (default [])
18.    -o, --opt value          Set driver specific options (default map[])
19.    --subnet value           Subnet in CIDR format that represents a network segment (default
    [])
```

Plus d'informations dans la [documentation](#).

Nous allons directement attaquer en créant un réseau, que nous appellerons *test* :

```

1. $ docker network create test
2. 1b8d8e2fae05224702568d71a7d8e128601250018795d06dba884d860e124e65
```

Nous pouvons vérifier s'il est bien créé :

```

1. $ docker network ls
2. NETWORK ID          NAME           DRIVER          SCOPE
3. 461b26287559        bridge        bridge         local
4. 9a2cd2ffcb62        host         host           local
5. d535bd59f11a        none         null           local
```

6. 1b8d8e2fae05	test	bridge	local
-----------------	------	--------	-------

Nous pouvons obtenir quelques informations sur ce réseau :

```
1. $ docker network inspect test
2. [
3.   {
4.     "Name": "test",
5.     "Id": "1b8d8e2fae05224702568d71a7d8e128601250018795d06dba884d860e124e65",
6.     "Scope": "local",
7.     "Driver": "bridge",
8.     "EnableIPv6": false,
9.     "IPAM": {
10.      "Driver": "default",
11.      "Options": {},
12.      "Config": [
13.        {
14.          "Subnet": "172.18.0.0/16",
15.          "Gateway": "172.18.0.1/16"
16.        }
17.      ]
18.    },
19.    "Internal": false,
20.    "Containers": {},
21.    "Options": {},
22.    "Labels": {}
23.  }
24. ]
```

Comme nous pouvons le voir, le réseau a comme sous-réseau 172.18.0.0/16, et comme gateway 172.18.0.1. Les conteneurs auront donc des IP attribuées entre 172.18.0.2 et 172.18.255.254 (172.18.255.255 étant le broadcast).

Nous pouvons évidemment choisir ce sous-réseau, les IP à attribuer aux conteneurs, etc. Pour ceci nous avons plusieurs options, comme `--subnet`, `--gateway`, ou `--ip-range` par exemple, ce qui donnerait :

```
1. $ docker network create --subnet 10.0.50.0/24 --gateway 10.0.50.254 --ip-range 10.0.50.0/28
   test2
2. 6d1a863a8dbadbabb2fb2d71c87d856309447e446837ba7a06dcde66c70614de
3. $ docker network inspect test2
4. [
5.   {
6.     "Name": "test2",
7.     "Id": "6d1a863a8dbadbabb2fb2d71c87d856309447e446837ba7a06dcde66c70614de",
8.     "Scope": "local",
9.     "Driver": "bridge",
10.    "EnableIPv6": false,
11.    "IPAM": {
12.      "Driver": "default",
13.      "Options": {},
14.      "Config": [
15.        {
16.          "Subnet": "10.0.50.0/24",
17.          "IPRange": "10.0.50.0/28",
18.          "Gateway": "10.0.50.254"
19.        }
20.      ]
21.    },
22.    "Internal": false,
23.    "Containers": {},
24.    "Options": {},
25.    "Labels": {}
26.  }
27. ]
```

Là j'ai créé un réseau 10.0.50.0/24 (donc de 10.0.50.0 à 10.0.50.255), j'ai mis la passerelle en 10.0.50.254, et un IPRange en 10.0.50.0/28 (donc une attribution de 10.0.50.1 à 10.0.50.14).

Pour chaque réseau créé, docker nous crée une interface :

```
1. $ ifconfig
2. br-1b8d8e2fae05 Link encap:Ethernet HWaddr 02:42:E1:AC:C5:1A
3.      inet addr:172.18.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
4.      UP BROADCAST MULTICAST MTU:1500 Metric:1
5.      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
6.      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
7.      collisions:0 txqueuelen:0
8.      RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
9.
10. br-6d1a863a8dba Link encap:Ethernet HWaddr 02:42:B8:7E:FE:A8
11.      inet addr:10.0.50.254 Bcast:0.0.0.0 Mask:255.255.255.0
12.      UP BROADCAST MULTICAST MTU:1500 Metric:1
13.      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
14.      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
15.      collisions:0 txqueuelen:0
16.      RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

Personnellement je ne suis pas fan de ces notations pour le nom des interfaces. Nous pouvons également choisir ce nom, avec l'argument `-o` :

```
1. $ docker network create --subnet 192.168.200.0/24 -o "com.docker.network.bridge.name=br-test"
   test3
2. d9955df665853315647ed3987f06768199c309255b5e5f17ffe1a24b7bd5f388
3. docker@default:~$ ifconfig
4. br-test Link encap:Ethernet HWaddr 02:42:1E:7A:E4:0E
5.      inet addr:192.168.200.1 Bcast:0.0.0.0 Mask:255.255.255.0
6.      UP BROADCAST MULTICAST MTU:1500 Metric:1
7.      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
8.      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
9.      collisions:0 txqueuelen:0
10.      RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

C'est bien beau de créer des networks, mais il faut bien les utiliser avec les conteneurs.

## XI-C - Utilisation des networks

Pour attacher un réseau à un conteneur, il suffit d'utiliser l'argument `--network` avec `docker run` :

```
1. $ docker run -ti --network test3 --name ctest alpine:3.4 sh
2. / # ifconfig
3. eth0      Link encap:Ethernet HWaddr 02:42:C0:A8:C8:02
4.      inet addr:192.168.200.2 Bcast:0.0.0.0 Mask:255.255.255.0
5.      inet6 addr: fe80::42:c0ff:fea8:c8022674/64 Scope:Link
6.      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
7.      RX packets:11 errors:0 dropped:0 overruns:0 frame:0
8.      TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
9.      collisions:0 txqueuelen:0
10.      RX bytes:926 (926.0 B) TX bytes:418 (418.0 B)
11.
12. lo        Link encap:Local Loopback
13.      inet addr:127.0.0.1 Mask:255.0.0.0
14.      inet6 addr: ::12674/128 Scope:Host
15.      UP LOOPBACK RUNNING MTU:65536 Metric:1
16.      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
17.      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
18.      collisions:0 txqueuelen:1
19.      RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

Nous sommes bien sur le réseau `test3` que nous avons précédemment créé.

Nous pouvons également ajouter un réseau supplémentaire avec `docker network connect` : Dans un autre terminal, taper :

```
$ docker network connect test ctest
```

Puis dans le conteneur :

```
1. / # ifconfig
2. eth0      Link encap:Ethernet  HWaddr 02:42:C0:A8:C8:02
3.          inet addr:192.168.200.2  Bcast:0.0.0.0  Mask:255.255.255.0
4.          inet6 addr: fe80::42:c0ff:fea8:c8022557/64 Scope:Link
5.          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
6.          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
7.          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
8.          collisions:0 txqueuelen:0
9.          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)
10.
11. eth1      Link encap:Ethernet  HWaddr 02:42:AC:13:00:02
12.          inet addr:172.18.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
13.          inet6 addr: fe80::42:acff:fe13:22557/64 Scope:Link
14.          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
15.          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
16.          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
17.          collisions:0 txqueuelen:0
18.          RX bytes:1296 (1.2 KiB)  TX bytes:648 (648.0 B)
19.
20. lo        Link encap:Local Loopback
21.          inet addr:127.0.0.1  Mask:255.0.0.0
22.          inet6 addr: ::12557/128 Scope:Host
23.          UP LOOPBACK RUNNING  MTU:65536  Metric:1
24.          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
25.          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
26.          collisions:0 txqueuelen:1
27.          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

L'un des avantages des networks, c'est qu'il n'est plus nécessaire de créer des liens entre les conteneurs, et donc deux conteneurs peuvent communiquer ensemble sans souci. Pour faire un test, créons un conteneur *dest* puis un conteneur qui le ping qui s'appellera *ping* :

```
1. $ docker run -d --network test3 --name dest alpine:3.4 ping 8.8.8.8 # ping 8.8.8.8 est là
   juste pour faire tourner le conteneur en arrière-plan
2. $ docker run -ti --network test3 --name ping alpine:3.4 ping dest
3. PING dest (192.168.200.2): 56 data bytes
4. 64 bytes from 192.168.200.2: seq=0 ttl=64 time=0.073 ms
5. 64 bytes from 192.168.200.2: seq=1 ttl=64 time=0.118 ms
6. 64 bytes from 192.168.200.2: seq=2 ttl=64 time=0.076 ms
7. 64 bytes from 192.168.200.2: seq=3 ttl=64 time=0.071 ms
8. 64 bytes from 192.168.200.2: seq=4 ttl=64 time=0.068 ms
9. 64 bytes from 192.168.200.2: seq=5 ttl=64 time=0.172 ms
10. ^C
11. --- dest ping statistics ---
12. 6 packets transmitted, 6 packets received, 0% packet loss
13. round-trip min/avg/max = 0.068/0.096/0.172 ms
```

Et voilà, ça marche sans `--link`.

Maintenant que nos tests sont terminés, nous pouvons supprimer nos réseaux :

```
1. $ docker network rm test test2 test3
2. test
3. test2
4. Error response from daemon: network test3 has active endpoints
```



**OUPS !!!**

Le réseau test3 est toujours utilisé par des conteneurs, nous pouvons voir les conteneurs actifs sur ce réseau :

```
1. $ docker network inspect test3
2. [
3.   {
4.     "Name": "test3",
5.     "Id": "971ffa0d0660547e309da67111d3826abff69f053dfba44b22ad05358bd78202",
6.     "Scope": "local",
7.     "Driver": "bridge",
8.     "EnableIPv6": false,
9.     "IPAM": {
10.      "Driver": "default",
11.      "Options": {},
12.      "Config": [
13.        {
14.          "Subnet": "192.168.200.0/24"
15.        }
16.      ]
17.    },
18.    "Internal": false,
19.    "Containers": {
20.      "edadf47462885d2433cc3bb8df17f262c3b9f8d57842951d148d31e746d2a155": {
21.        "Name": "dest",
22.
23.        "EndpointID": "eede64502ee056b20f57f39e3cc6631bac801a424faed475d2d90d1f71cde7a1",
24.        "MacAddress": "02:42:c0:a8:c8:02",
25.        "IPv4Address": "192.168.200.2/24",
26.        "IPv6Address": ""
27.      }
28.    },
29.    "Options": {
30.      "com.docker.network.bridge.name": "br-test"
31.    },
32.    "Labels": {}
33.  ]
```

Il s'agit du conteneur *dest* :

```
1. $ docker stop dest && docker rm -f dest
2. dest
3. $ docker network rm test3
4. test3
```

## XI-D - Conclusion

Nous avons vu ici, comment créer des réseaux pour nos conteneurs, ceci est plutôt pratique pour une isolation encore plus poussée des conteneurs, et surtout supprimer les limitations de `--link`, à savoir l'intercommunication entre conteneurs. Je vous invite une fois de plus à consulter la **documentation**, qui est comme toujours, très bien faite.

## XII - Docker compose

Docker compose est un outil anciennement tiers puis racheté par docker, qui permet de composer une stack ou une infrastructure complète de conteneurs. Celui-ci permet de simplifier la création, l'interconnexion et la multiplication de conteneurs. En gros nous créons un fichier yml qui nous permettra de gérer un ensemble de conteneurs en quelques commandes.

Maintenant que vous connaissez bien docker (si j'ai bien fait mon boulot), je serai plus succinct sur les explications, surtout que docker-compose reprend les mêmes options que docker.

## XII-A - Installation

### XII-A-1 - Sous Windows

Si vous avez installé Docker via docker-toolbox ou docker4windows, docker-compose est déjà installé et fonctionnel.

Sinon nous pouvons l'installer à la main, il suffit de télécharger le binaire sur [github](#) Exemple pour la version 1.8.1 (dernière version présentement) **docker-compose**

Si vous avez installé Docker en suivant la méthode manuelle de ce tutoriel, vous devriez avoir un répertoire c:\docker\bin, copiez-y le binaire et renommez-le en *docker-compose.exe*.

Et c'est tout, normalement tout est OK, on peut tester :

```
1. $ docker-compose version
2. docker-compose version 1.12.0, build unknown
3. docker-py version: 2.2.1
4. CPython version: 3.6.1
5. OpenSSL version: OpenSSL 1.1.0e 16 Feb 2017
```

### XII-A-2 - Sous GNU/Linux

Nous avons trois possibilités d'installer docker-compose sous GNU/Linux, avec pip, à la main, ou avec un conteneur. Nous ne verrons ici que l'installation via pip, et l'installation via un conteneur. Je vous conseille tout de même l'installation par conteneur, plus simple et plus rapide à mettre à jour.

#### XII-A-2-a - Installation via pip

Pour ceux qui ne connaissent pas pip, pip est un gestionnaire de paquets pour python, qui permet l'installation d'applications, ou de bibliothèques. C'est un peu comme apt-get.

On installe d'abord pip, sous debian cela donne :

```
1. $ apt-get install python-pip
2. Lecture des listes de paquets... Fait
3. Construction de l'arbre des dépendances
4. Lecture des informations d'état... Fait
5. python-pip est déjà la plus récente version disponible.
6. 0 mis à jour, 0 nouvellement installés, 0 à enlever et 0 non mis à jour.
```

*(de mon côté j'avais déjà installé pip)*

Il suffit maintenant de lancer cette commande :

```
$ pip install docker-compose
```

On peut tester :

```
1. $ docker-compose version
2. docker-compose version 1.8.0, build d988a55
3. docker-py version: 1.9.0
4. CPython version: 2.7.11
5. OpenSSL version: OpenSSL 1.0.2d 9 Jul 2015
```

## XII-A-2-b - Installation par un conteneur

C'est vraiment très simple, nous allons créer un alias qui lancera un conteneur docker-compose. Nous utiliserons ma propre image : **xataz/compose**

On édite le fichier *.profile* de notre utilisateur :

```
$ vim ~/.profile
```

Et on ajoute ceci dedans :

```
1. alias docker-compose='docker run -v "$(pwd)": "$(pwd)" \
2.     -v /var/run/docker.sock:/var/run/docker.sock \
3.     -e UID=$(id -u) -e GID=$(id -g) \
4.     -w "$(pwd)" \
5.     -ti --rm xataz/compose:1.8'
```

On recharge le profile :

```
$ . ~/.profile
```

Et on peut faire un essai :

```
1. $ docker-compose version
2. Unable to find image 'xataz/compose:1.8' locally
3. 1.8: Pulling from xataz/compose
4. c0cb142e4345: Already exists
5. a0bbf809363b: Pull complete
6. 6d4c02e2941c: Pull complete
7. Digest: sha256:8a56828af12467a6f7ac55c54d6dd877d51e50026ff91d9fc88d0d0cedbadb3f
8. Status: Downloaded newer image for xataz/compose:1.8
9. docker-compose version 1.8.1, build 878cff1
10. docker-py version: 1.10.3
11. CPython version: 2.7.12
12. OpenSSL version: OpenSSL 1.0.2i 22 Sep 2016
```

## XII-B - Utilisation de docker-compose

Comme je le disais plus haut, docker-compose permet de simplifier la gestion de plusieurs conteneurs. Cette partie sera purement théorique, nous verrons dans la partie suivante, un cas concret.

Voyons d'abord les commandes de docker-compose :

```
1. $ docker-compose --help
2. Define and run multi-container applications with Docker.
3.
4. Usage:
5.   docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
6.   docker-compose -h|--help
7.
8. Options:
9.   -f, --file FILE           Specify an alternate compose file (default: docker-compose.yml)
10.  -p, --project-name NAME     Specify an alternate project name (default: directory name)
11.  --verbose                  Show more output
12.  -v, --version              Print version and exit
13.  -H, --host HOST            Daemon socket to connect to
14.
15.  --tls                      Use TLS; implied by --tlsverify
16.  --tlscacert CA_PATH        Trust certs signed only by this CA
17.  --tlscert CLIENT_CERT_PATH Path to TLS certificate file
18.  --tlskey TLS_KEY_PATH      Path to TLS key file
```

```
19. --tlsverify          Use TLS and verify the remote
20. --skip-hostname-check Don't check the daemon's hostname against the name specified
21.                      in the client certificate (for example if your docker host
22.                      is an IP address)
23.
24. Commands:
25. build                Build or rebuild services
26. bundle               Generate a Docker bundle from the Compose file
27. config               Validate and view the compose file
28. create               Create services
29. down                 Stop and remove containers, networks, images, and volumes
30. events               Receive real time events from containers
31. exec                 Execute a command in a running container
32. help                 Get help on a command
33. kill                 Kill containers
34. logs                 View output from containers
35. pause                Pause services
36. port                 Print the public port for a port binding
37. ps                   List containers
38. pull                 Pulls service images
39. push                 Push service images
40. restart              Restart services
41. rm                   Remove stopped containers
42. run                  Run a one-off command
43. scale                 Set number of containers for a service
44. start                Start services
45. stop                 Stop services
46. unpause              Unpause services
47. up                   Create and start containers
48. version              Show the Docker-Compose version information
```

Nous n'utiliserons pas toutes ces commandes, je ne vais pas vous faire une description de chaque commande, je pense que malgré l'anglais, elles sont claires.

Pour que docker-compose fonctionne, il nous faut un *docker-compose.yml*. Ce fichier, au format yaml, comportera les informations pour créer notre infrastructure, comme les conteneurs, les networks et les volumes. Le format yaml est vraiment très susceptible au niveau de l'indentation, j'utilise personnellement une règle, à chaque sous-configuration, je fais deux espaces.

Nous avons deux versions de *docker-compose.yml*, la version 1 et la version 2. Nous utiliserons ici la version 2, qui remplacera complètement la version 1 bientôt.

Bon on attaque !!!

Nous commençons notre fichier par préciser que nous utiliserons la version 2 :

```
version: '2'
```

Puis nous aurons trois parties :

```
1. volumes:
2. networks:
3. services:
```

Le nom de ces trois parties est plutôt clair, volumes pour définir nos volumes (équivalent à docker volume), networks pour définir nos réseaux (équivalent à docker network) et services pour définir nos conteneurs (équivalent à docker run).

Par exemple, pour créer un conteneur nginx, avec un volume spécifique et un réseau spécifique, nous devrions faire :

```
1. $ docker volume create --name vol_nginx
2. $ docker volume ls
3. DRIVER          VOLUME NAME
```



```

4. local                vol_nginx
5. $ docker network create net_nginx
6. $ docker network ls
7. NETWORK ID          NAME                DRIVER            SCOPE
8. 4d9b424c1abb        bridge             bridge            local
9. 9a2cd2ffcb62        host               host              local
10. 0a21f7e5462b        net_nginx          bridge            local
11. d535bd59f11a        none               null              local
12. $ docker run -d -p 80:80 \
13.     -v vol_nginx:/var/www \
14.     --network net_nginx \
15.     -e VARIABLE=truc \
16.     --name nginx nginx
17. 996626cbde9b5245a7c4bf22e9bc033379eda094fe2e9c758096730d07866d36
  
```

Et voilà la comparaison avec docker-compose, on commence par le fichier :

```

1. version: '2'
2.
3. volumes:
4.   vol_nginx:
5.
6. networks:
7.   net_nginx:
8.
9. services:
10.  nginx:
11.    image: nginx
12.    container_name: nginx
13.    volumes:
14.      - vol_nginx:/var/www
15.    environment:
16.      - VARIABLE=truc
17.    ports:
18.      - "80:80"
19.    networks:
20.      - net_nginx
  
```

Puis on lance le tout :

```

1. $ docker-compose up -d
2. Creating network "projects_net_nginx" with the default driver
3. Creating volume "projects_vol_nginx" with default driver
4. Creating nginx
  
```

*Le nom des éléments est nommé en fonction du répertoire courant, ici mon répertoire était **projects**.*

Et voilà, nous avons lancé notre conteneur, avec son réseau et son volume. Pas forcément utile pour un seul conteneur, mais pour plusieurs, cela prend tout son sens. Volumes et networks sont bien évidemment optionnels, si vous ne souhaitez pas créer un réseau particulier, ou si vous utilisez les volumes directement avec le chemin complet.

Je vous invite à regarder cette [page](#) pour plus d'information sur le fichier de configuration.

Une fois notre fichier fait, nous pouvons gérer nos conteneurs avec docker-compose. Toutes ces actions se font dans le répertoire du yaml.

Pour lancer les conteneurs : `docker-compose up`

En arrière-plan : `docker-compose up -d`

Pour arrêter les conteneurs : `docker-compose stop` ou `docker-compose stop [containername]`

On les relance : `docker-compose start` ou `docker-compose start [containername]`

Et on les redémarre : `docker-compose restart` ou `docker-compose restart [containername]`

Pour voir les conteneurs créés : `docker-compose ps`

Nous pouvons aussi voir les logs : `docker-compose logs`

Et nous pouvons aussi supprimer les conteneurs : `docker-compose rm`

## XII-C - Créer une stack web

Nous allons créer une stack web, c'est-à-dire un ensemble de conteneurs pour faire un serveur web, nous utiliserons nginx, PHP-fpm et mariadb. Pour tester notre mixture, nous utiliserons *adminer.php*. Je passerai sur les détails de la configuration des applications, car ceci n'est pas le but du tutoriel.

Voici les images que nous utiliserons :

- nginx : nginx
- php : PHP:fpm
- mariadb : mariadb

Vous pouvez les pull manuellement, cela permettra de gagner du temps.

Si vous avez suivi mes conseils, nous avons normalement un répertoire Docker à la racine de notre machine docker. Nous créons un répertoire web, qui accueillera nos fichiers de configuration nginx ainsi que nos sites :

```
1. $ mkdir -p /Docker/web/{conf.d,sites,www}
2. $ ls /Docker/web
3. conf.d  sites www
```

On copie ou télécharge *adminer.php* dans le répertoire `/Docker/web/www` :

```
$ wget https://www.adminer.org/static/download/4.2.5/adminer-4.2.5.php -O /Docker/web/www/adminer.php
```

On crée un fichier *adminer.conf* dans `/Docker/web/sites` et on y colle ceci :

```
1. server {
2.     listen 80;
3.     root /var/www/html;
4.
5.     location / {
6.         index adminer.php;
7.     }
8.
9.     location ~ /\.PHP$ {
10.         try_files $uri =404;
11.         fastcgi_pass  php:9000;
12.         fastcgi_index  adminer.php;
13.         fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
14.         include fastcgi_params;
15.     }
16. }
```

Pour PHP, il va falloir créer notre propre image, à partir de l'image officielle, afin d'y ajouter le module mysql :

```
1. # vim php.dockerfile
2. FROM php:fpm
3. RUN docker-php-ext-install mysqli
4. CMD ["php-fpm"]
```

En fait nous utilisons un script disponible dans l'image pour créer notre image.

On build :

```
$ docker build -t php-mysql:fpm -f php.dockerfile .
```

Et on crée notre stack avec un fichier *docker-compose.yml* :

```

1. version: '2'
2.
3. networks:
4.   default:
5.     driver: bridge
6.
7. services:
8.   web:
9.     container_name: web
10.    image: nginx
11.    ports:
12.      - "8080:80"
13.    volumes:
14.      - /Docker/web/www:/var/www/html
15.      - /Docker/web/sites:/etc/nginx/conf.d
16.
17.   PHP:
18.     container_name: PHP
19.     image: php-mysql:fpm
20.     volumes:
21.       - /Docker/web/www:/var/www/html
22.
23.   mariadb:
24.     container_name: mariadb
25.     image: mariadb
26.     environment:
27.       MYSQL_ROOT_PASSWORD: root
  
```

Puis on lance notre stack en arrière-plan :

```

1. $ docker-compose up -d
2. Creating web
3. Creating PHP
4. Creating mariadb
5. $
  
```

Et voilà notre stack est prête, on peut essayer de s'y connecter avec : **http://IPDocker:8080**, et normalement cela devrait fonctionner, si la page de connexion de adminer s'affiche, c'est que l'interaction entre le conteneur **web** et le conteneur **PHP** est correcte. Pour se connecter, il suffit de remplir comme ceci (le mot de passe est **root** comme indiqué dans le docker-compose) :

## Authentification

<b>Système</b>	MySQL ▼
<b>Serveur</b>	mariadb
<b>Utilisateur</b>	root
<b>Mot de passe</b>	••••
<b>Base de données</b>	

☐ Authentification permanente

Et si la connexion fonctionne, c'est que **PHP** et **mariadb** fonctionne correctement.

Nous allons modifier notre docker-compose, afin d'y ajouter un alias pour **mariadb**, car c'est fastidieux de taper **mariadb** tout le temps :

```
1. [...]
2.   mariadb:
3.     container_name: mariadb
4.     image: mariadb
5.     environment:
6.       MYSQL_ROOT_PASSWORD: root
7.     networks:
8.       default:
9.         aliases:
10.          - db
```

Puis on relance notre stack :

```
1. $ docker-compose up -d
2. web is up-to-date
3. Recreating mariadb
4. PHP is up-to-date
```

Comme on peut le voir, docker-compose est intelligent, et ne touche qu'aux conteneurs modifiés, ici mariadb. Puis on peut retester adminer, en mettant cette fois-ci dans *serveur* seulement **db**, et normalement, cela fonctionne correctement.

## XII-D - Conclusion

Ceci n'est qu'une ébauche, nous avons ici créé une stack vraiment simple. Mais il est possible de faire des infras complètes via docker-compose. Je vous invite de nouveau à regarder cette [page](#) pour plus d'informations sur docker-compose.

## XIII - Docker-machine

### XIII-A - Qu'est-ce que docker-machine ?

Docker-machine est un outil qui permet de provisionner des machines (physiques ou virtuelles) afin d'y installer le nécessaire pour faire fonctionner docker. Machine permet de provisionner sur virtualbox, mais également beaucoup de services cloud, comme digitalOcean, Azure, Google ou plus génériques sur du openstack. Il installera donc docker, mais génère également un certificat ssl, un jeu de clés pour une connexion ssh, et ceux sur de nombreuses distributions GNU/Linux (debian, centos, archlinux ...).

Je n'ai personnellement commencé à utiliser docker-machine que très récemment, et je trouve cela vraiment indispensable.

### XIII-B - Installation

Lors de la rédaction de ce chapitre, nous sommes à la version 0.11.0 de docker-machine, les liens indiqués sont donc contextualisés avec cette version. Il se peut donc que lors de votre lecture, une autre version soit sortie, pour le vérifier, vous pouvez regarder sur [les releases du github de docker-machine](#).

## XIII-B-1 - Sous Windows

Si vous avez installé boot2docker, docker-machine est déjà préinstallé, sinon l'installation est plutôt simple, il suffit de télécharger l'exécutable : **docker-machine 32bits** **docker-machine 64 bits**

Il faut ensuite le placer dans un endroit stratégique, personnellement c:\docker\bin, je vous conseille également de le renommer en *docker-machine.exe*, car ce n'est pas très pratique de toujours taper *docker-machine-Windows-x86\_64.exe*.

Si vous avez suivi mon tutoriel pour l'installation sous Windows, normalement vous ne devriez pas avoir à modifier les variables d'environnement, sinon n'oubliez pas de rajouter l'emplacement de votre binaire dans la variable d'environnement PATH afin qu'il soit utilisable sans devoir être dans le répertoire c:\docker\bin.

Et normalement cela fonctionne, on teste :

```
1. $ docker-machine.exe version
2. docker-machine version 0.7.0, build 61388e9
```

## XIII-B-2 - Sous GNU/Linux et OS X

L'installation est encore plus simple sous un système Unix, il suffit de télécharger le binaire (sauf si vous avez utilisé la toolbox pour OS X) :

```
1. $ wget https://github.com/docker/machine/releases/download/v0.11.0/docker-machine-$(uname -s)-$(uname -m) -O /usr/local/bin/docker-machine
2. $ chmod +x /usr/local/bin/docker-machine
3. # $(uname -s) : Permet d'obtenir le type d'OS (Linux ou darwin)
4. # $(uname -m) : Permet d'obtenir l'architecture de l'OS (i386 ou x86_64)
5. # Exemple sous OS X : docker-machine-$(uname -s)-$(uname -m) devient docker-machine-darwin-x86_64
```

Normalement c'est bon, pour tester :

```
1. $ docker-machine version
2. docker-machine version 0.11.0, build 61388e9
```

## XIII-C - Utilisation

Nous utiliserons docker-machine avec le driver virtualbox, le principe reste cependant le même avec les autres drivers (cf. [liste](#)).

Pour voir les commandes de docker-machine :

```
1. $ docker-machine
2. Usage: docker-machine.exe [OPTIONS] COMMAND [arg...]
3.
4. Create and manage machines running Docker.
5.
6. Version: 0.7.0, build a650a40
7.
8. Author:
9.   Docker Machine Contributors - <https://github.com/docker/machine>
10.
11. Options:
12.   --debug, -D           Enable debug mode
13.   -s, --storage-path "C:\Users\xataz\.docker\machine" Configures storage path
14.   --tls-ca-cert [ $MACHINE_TLS_CA_CERT ] CA to verify remotes against
```

```

15. --tls-ca-key Private key to generate certificates
    [$MACHINE_TLS_CA_KEY]
16. --tls-client-cert Client cert to use for TLS
    [$MACHINE_TLS_CLIENT_CERT]
17. --tls-client-key Private key used in client TLS auth
    [$MACHINE_TLS_CLIENT_KEY]
18. --github-api-token Token to use for requests to the
    Github API [$MACHINE_GITHUB_API_TOKEN]
19. --native-ssh Use the native (Go-based) SSH
    implementation. [$MACHINE_NATIVE_SSH]
20. --bugsnag-api-token BugSnag API token for crash reporting
    [$MACHINE_BUGSNAG_API_TOKEN]
21. --help, -h show help
22. --version, -v print the version
23.
24. Commands:
25. active Print which machine is active
26. config Print the connection config for machine
27. create Create a machine
28. env Display the commands to set up the environment for the Docker client
29. inspect Inspect information about a machine
30. ip Get the IP address of a machine
31. kill Kill a machine
32. ls List machines
33. provision Re-provision existing machines
34. regenerate-certs Regenerate TLS Certificates for a machine
35. restart Restart a machine
36. rm Remove a machine
37. ssh Log into or run a command on a machine with SSH.
38. scp Copy files between machines
39. start Start a machine
40. status Get the status of a machine
41. stop Stop a machine
42. upgrade Upgrade a machine to the latest version of Docker
43. url Get the URL of a machine
44. version Show the Docker Machine version or a machine docker version
45. help Shows a list of commands or help for one command
46.
47. Run 'docker-machine.exe COMMAND --help' for more information on a command.

```

Comme on peut le voir, les commandes sont toujours similaires à compose ou docker. Si vous êtes sous Windows, installer avec docker-toolbox, vous devriez déjà avoir une machine, pour vérifier, faites :

```

1. $ docker-machine ls
2. NAME ACTIVE DRIVER STATE URL SWARM DOCKER
   ERRORS
3. mydocker - generic Running tcp://192.168.1.201:2376 v1.10.3
4. swarm - virtualbox Running tcp://192.168.99.100:2376 v1.11.0

```

Sinon, nous y reviendrons.

### XIII-C-1 - Créer une machine sous virtualbox

Comme dit précédemment, nous utiliserons virtualbox, nous allons donc créer notre machine avec docker-machine create. Pour voir les arguments possibles pour create il suffit de faire :

```

1. $ docker-machine create
2. Usage: docker-machine create [OPTIONS] [arg...]
3.
4. Create a machine
5.
6. Description:
7. Run 'C:\Program Files\Docker Toolbox\docker-machine.exe create --driver name' to include
   the create flags for that driver in the help text.
8.
9. Options:
10.

```

```

11. --driver, -d "none"
    Driver to create machine with. [$MACHINE_DRIVER]
12. --engine-install-url "https://get.docker.com"
    Custom URL to use for engine installation [$MACHINE_DOCKER_INSTALL_URL]
13. --engine-opt [--engine-opt option --engine-opt option]
    Specify arbitrary flags to include with the created engine in the form flag=value
14. --engine-insecure-registry [--engine-insecure-registry option --engine-insecure-registry
    option] Specify insecure registries to allow with the created engine
15. --engine-registry-mirror [--engine-registry-mirror option --engine-registry-mirror option]
    Specify registry mirrors to use [$ENGINE_REGISTRY_MIRROR]
16. --engine-label [--engine-label option --engine-label option]
    Specify labels for the created engine
17. --engine-storage-driver
    Specify a storage driver to use with the engine
18. --engine-env [--engine-env option --engine-env option]
    Specify environment variables to set in the engine
19. --swarm
    Configure Machine with Swarm
20. --swarm-image "swarm:latest"
    Specify Docker image to use for Swarm [$MACHINE_SWARM_IMAGE]
21. --swarm-master
    Configure Machine to be a Swarm master
22. --swarm-discovery
    Discovery service to use with Swarm
23. --swarm-strategy "spread"
    Define a default scheduling strategy for Swarm
24. --swarm-opt [--swarm-opt option --swarm-opt option]
    Define arbitrary flags for swarm
25. --swarm-host "tcp://0.0.0.0:3376"
    ip/socket to listen on for Swarm master
26. --swarm-addr
    addr to advertise for Swarm (default: detect and use the machine IP)
27. --swarm-experimental
    Enable Swarm experimental features
28. --tls-san [--tls-san option --tls-san option]
    Support extra SANs for TLS certs
  
```

Et plus particulièrement pour le driver virtualbox :

```

1. $ docker-machine create -d virtualbox --help
2. Usage: docker-machine create [OPTIONS] [arg...]
3.
4. Create a machine
5.
6. Description:
7.   Run 'C:\Program Files\Docker Toolbox\docker-machine.exe create --driver name' to include
   the create flags for that driver in the help text.
8.
9. Options:
10.
11. --driver, -d "none"
    Driver to create machine with. [$MACHINE_DRIVER]
12. --engine-env [--engine-env option --engine-env option]
    Specify environment variables to set in the engine
13. --engine-insecure-registry [--engine-insecure-registry option --engine-insecure-registry
    option] Specify insecure registries to allow with the created engine
14. --engine-install-url "https://get.docker.com"
    Custom URL to use for engine installation [$MACHINE_DOCKER_INSTALL_URL]
15. --engine-label [--engine-label option --engine-label option]
    Specify labels for the created engine
16. --engine-opt [--engine-opt option --engine-opt option]
    Specify arbitrary flags to include with the created engine in the form flag=value
17. --engine-registry-mirror [--engine-registry-mirror option --engine-registry-mirror option]
    Specify registry mirrors to use [$ENGINE_REGISTRY_MIRROR]
18. --engine-storage-driver
    Specify a storage driver to use with the engine
19. --swarm
    Configure Machine with Swarm
20. --swarm-addr
    addr to advertise for Swarm (default: detect and use the machine IP)
  
```

```

21.  --swarm-discovery
      Discovery service to use with Swarm
22.  --swarm-experimental
      Enable Swarm experimental features
23.  --swarm-host "tcp://0.0.0.0:3376"
      ip/socket to listen on for Swarm master
24.  --swarm-image "swarm:latest"
      Specify Docker image to use for Swarm [$MACHINE_SWARM_IMAGE]
25.  --swarm-master
      Configure Machine to be a Swarm master
26.  --swarm-opt [--swarm-opt option --swarm-opt option]
      Define arbitrary flags for swarm
27.  --swarm-strategy "spread"
      Define a default scheduling strategy for Swarm
28.  --tls-san [--tls-san option --tls-san option]
      Support extra SANs for TLS certs
29.  --virtualbox-boot2docker-url
      The URL of the boot2docker image. Defaults to the latest available version
      [$VIRTUALBOX_BOOT2DOCKER_URL]
30.  --virtualbox-cpu-count "1"
      number of CPUs for the machine (-1 to use the number of CPUs available)
      [$VIRTUALBOX_CPU_COUNT]
31.  --virtualbox-disk-size "20000"
      Size of disk for host in MB [$VIRTUALBOX_DISK_SIZE]
32.  --virtualbox-host-dns-resolver
      Use the host DNS resolver [$VIRTUALBOX_HOST_DNS_RESOLVER]
33.  --virtualbox-hostonly-cidr "192.168.99.1/24"
      Specify the Host Only CIDR [$VIRTUALBOX_HOSTONLY_CIDR]
34.  --virtualbox-hostonly-nicpromisc "deny"
      Specify the Host Only Network Adapter Promiscuous Mode
      [$VIRTUALBOX_HOSTONLY_NIC_PROMISC]
35.  --virtualbox-hostonly-nictype "82540EM"
      Specify the Host Only Network Adapter Type [$VIRTUALBOX_HOSTONLY_NIC_TYPE]
36.  --virtualbox-import-boot2docker-vm
      The name of a Boot2Docker VM to import [$VIRTUALBOX_BOOT2DOCKER_IMPORT_VM]
37.  --virtualbox-memory "1024"
      Size of memory for host in MB [$VIRTUALBOX_MEMORY_SIZE]
38.  --virtualbox-nat-nictype "82540EM"
      Specify the Network Adapter Type [$VIRTUALBOX_NAT_NICTYPE]
39.  --virtualbox-no-dns-proxy
      Disable proxying all DNS requests to the host [$VIRTUALBOX_NO_DNS_PROXY]
40.  --virtualbox-no-share
      Disable the mount of your home directory [$VIRTUALBOX_NO_SHARE]
41.  --virtualbox-no-vtx-check
      Disable checking for the availability of hardware virtualization before the vm is
      started [$VIRTUALBOX_NO_VTX_CHECK]

```

Bon maintenant que nous avons les arguments, nous pouvons créer notre machine :

```

1. $ docker-machine create -d virtualbox --virtualbox-cpu-count "2" --virtualbox-memory "2048" --
   virtualbox-disk-size "25000" tutorial
2. Running pre-create checks...
3. Creating machine...
4. (tutorial) Copying C:\Users\xataz\.docker\machine\cache\boot2docker.iso to C:\Users\xataz
   \.docker\machine\machines\tutorial\boot2docker.iso...
5. (tutorial) Creating VirtualBox VM...
6. (tutorial) Creating SSH key...
7. (tutorial) Starting the VM...
8. (tutorial) Check network to re-create if needed...
9. (tutorial) Waiting for an IP...
10. Waiting for machine to be running, this may take a few minutes...
11. Detecting operating system of created instance...
12. Waiting for SSH to be available...
13. Detecting the provisioner...
14. Provisioning with boot2docker...
15. Copying certs to the local machine directory...
16. Copying certs to the remote machine...
17. Setting Docker configuration on the remote daemon...
18. Checking connection to Docker...
19. Docker is up and running!

```



20. To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: C:\Program Files\Docker Toolbox\docker-machine.exe env tutorial

Nous avons donc créé notre machine, comme nous pouvons le vérifier :

```
1. $ docker-machine ls
2. NAME          ACTIVE  DRIVER      STATE    URL                                     SWARM   DOCKER
   ERRORS
3. mydocker      -       generic     Running  tcp://192.168.1.201:2376              v1.10.3
4. tutorial      *       virtualbox  Running  tcp://192.168.99.100:2376             v1.11.0
```

## XIII-C-2 - Utilisons notre machine

Créer une machine c'est bien, mais l'utiliser c'est mieux.

Si nous tentons de lister les images par exemple, nous avons une erreur (ici sous Windows) :

```
1. $ docker images
2. An error occurred trying to connect: Get http://localhost:2376/images/json:
   open //pipe/docker_engine: Le fichier spécifié est introuvable.
```

Pour ce faire il faut indiquer à Docker sur quelle machine se connecter, comme ceci :

```
1. $ docker --tlsverify -H tcp://192.168.99.100:2376 --tlscacert=/c/Users/xataz/.docker/machine/
   machines/tutorial/ca.pem --tlscert=/c/Users/xataz/.docker/machine/machines/tutorial/cert.pem
   --tlskey=/c/Users/xataz/.docker/machine/machines/tutorial/key.pem info
2. Containers: 0
3. Running: 0
4. Paused: 0
5. Stopped: 0
6. Images: 0
7. Server Version: 1.11.0
8. Storage Driver: aufs
9. Root Dir: /mnt/sda1/var/lib/docker/aufs
10. Backing Filesystem: extfs
11. Dirs: 0
12. Dirperm1 Supported: true
13. Logging Driver: json-file
14. Cgroup Driver: cgroupfs
15. Plugins:
16. Volume: local
17. Network: host bridge null
18. Kernel Version: 4.1.19-boot2docker
19. Operating System: Boot2Docker 1.11.0 (TCL 7.0); HEAD : 32ee7e9 - Wed Apr 13 20:06:49 UTC 2016
20. OSType: Linux
21. Architecture: x86_64
22. CPUs: 2
23. Total Memory: 1.955 GiB
24. Name: tutorial
25. ID: ZYFO:VNOS:UWNZ:LKHI:WC7A:D2XC:RFKD:GAMN:VF42:GI5Y:D27G:HSIK
26. Docker Root Dir: /mnt/sda1/var/lib/docker
27. Debug mode (client): false
28. Debug mode (server): true
29. File Descriptors: 12
30. Goroutines: 30
31. System Time: 2016-04-20T18:03:14.189404964Z
32. EventsListeners: 0
33. Registry: https://index.docker.io/v1/
34. Labels:
35. provider=virtualbox
```

Admettez-le, c'est plutôt fastidieux à faire, nous avons une autre méthode, qui consiste à « sourcer » les informations de la machine, de mettre ceci en variable d'environnement avec l'option env :

```
1. $ docker-machine env tutorial
```

```
2. export DOCKER_TLS_VERIFY="1"
3. export DOCKER_HOST="tcp://192.168.99.100:2376"
4. export DOCKER_CERT_PATH="C:\Users\xataz\.docker\machine\machines\tutoriel"
5. export DOCKER_MACHINE_NAME="tutoriel"
6. # Run this command to configure your shell:
7. # eval $( "C:\Program Files\Docker Toolbox\docker-machine.exe" env tutoriel)
```

Nous avons ici les différentes informations pour pouvoir s'y connecter, et même la commande à taper :

```
$ eval $( "C:\Program Files\Docker Toolbox\docker-machine.exe" env tutoriel)
```

Sous l'invite de commande Windows (j'utilise mingw64), la commande sera : **@FOR** /f "tokens=" %i **IN** ('docker-machine env tutoriel') **DO @%**i, mais elle est également indiquée via la commande.

Et c'est tout, nous pouvons tester :

```
1. $ docker info
2. Containers: 0
3. Running: 0
4. Paused: 0
5. Stopped: 0
6. Images: 0
7. Server Version: 1.11.0
8. Storage Driver: aufs
9. Root Dir: /mnt/sdal/var/lib/docker/aufs
10. Backing Filesystem: extfs
11. Dirs: 0
12. Dirperm1 Supported: true
13. Logging Driver: json-file
14. Cgroup Driver: cgroupfs
15. Plugins:
16. Volume: local
17. Network: bridge null host
18. Kernel Version: 4.1.19-boot2docker
19. Operating System: Boot2Docker 1.11.0 (TCL 7.0); HEAD : 32ee7e9 - Wed Apr 13 20:06:49 UTC 2016
20. OSType: Linux
21. Architecture: x86_64
22. CPUs: 2
23. Total Memory: 1.955 GiB
24. Name: tutoriel
25. ID: ZYFO:VNOS:UWNZ:LKHI:WC7A:D2XC:RFKD:GAMN:VF42:GI5Y:D27G:HSIK
26. Docker Root Dir: /mnt/sdal/var/lib/docker
27. Debug mode (client): false
28. Debug mode (server): true
29. File Descriptors: 12
30. Goroutines: 30
31. System Time: 2016-04-20T18:08:33.287063691Z
32. EventsListeners: 0
33. Registry: https://index.docker.io/v1/
34. Labels:
35. provider=virtualbox
```

Simple non ?! Nous pouvons maintenant utiliser Docker comme si on était en local.

```
1. $ docker run -d -P xataz/nginx:1.9
2. Unable to find image 'xataz/nginx:1.9' locally
3. 1.9: Pulling from xataz/nginx
4. 420890c9e918: Pulling fs layer
5. 49453f6fdf36: Pulling fs layer
6. 14a932cbdb93: Pulling fs layer
7. 179d8f2a0f72: Pulling fs layer
8. de957a98ee12: Pulling fs layer
9. 4237b3506f00: Pulling fs layer
10. 87aa5a2470bc: Pulling fs layer
11. e0d4bf63eb3c: Pulling fs layer
12. 179d8f2a0f72: Waiting
13. de957a98ee12: Waiting
```

```

14. 4237b3506f00: Waiting
15. 87aa5a2470bc: Waiting
16. e0d4bf63eb3c: Waiting
17. 49453f6fdf36: Download complete
18. 420890c9e918: Verifying Checksum
19. 420890c9e918: Pull complete
20. 49453f6fdf36: Pull complete
21. 14a932cbdb93: Verifying Checksum
22. 14a932cbdb93: Download complete
23. 14a932cbdb93: Pull complete
24. 4237b3506f00: Verifying Checksum
25. 4237b3506f00: Download complete
26. 179d8f2a0f72: Verifying Checksum
27. 179d8f2a0f72: Download complete
28. 179d8f2a0f72: Pull complete
29. 87aa5a2470bc: Verifying Checksum
30. 87aa5a2470bc: Download complete
31. de957a98ee12: Verifying Checksum
32. de957a98ee12: Download complete
33. e0d4bf63eb3c: Verifying Checksum
34. e0d4bf63eb3c: Download complete
35. de957a98ee12: Pull complete
36. 4237b3506f00: Pull complete
37. 87aa5a2470bc: Pull complete
38. e0d4bf63eb3c: Pull complete
39. Digest: sha256:a04aebdf836a37c4b5de9ce32a39ba5fc2535e25c58730e1a1f6bf77ef11fe69
40. Status: Downloaded newer image for xataz/nginx:1.9
41. 818cebd0bed38966c05730b1b0a02f3a3f48adf0aea5bf52d25da7578bdfef15
42.
43. $ docker ps
44. CONTAINER ID          IMAGE               COMMAND                  CREATED              STATUS
PORTS
45. 818cebd0bed3          xataz/nginx:1.9    "tini -- /usr/bin/sta"   15 seconds ago      Up 14
seconds
0.0.0.0:32769->8080/tcp, 0.0.0.0:32768->8443/tcp hungry_hawking

```



**Attention, les volumes restent locaux à la machine créée, et non à la machine cliente.**

### XIII-C-3 - Gérer nos machines

Maintenant que nous avons créé notre machine, il va falloir la gérer, et franchement c'est simpliste.

Commençons par l'arrêter :

```

1. $ docker-machine stop tutoriel
2. Stopping "tutoriel"...
3. Machine "tutoriel" was stopped.
4.
5. $ docker-machine ls
6. NAME      ACTIVE  DRIVER      STATE     URL                  SWARM   DOCKER   ERRORS
7. mydocker  -       generic     Running   tcp://192.168.1.201:2376   v1.10.3
8. tutoriel  -       virtualbox  Stopped

```

Puis nous pouvons la démarrer :

```

1. $ docker-machine start tutoriel
2. Starting "tutoriel"...
3. (tutoriel) Check network to re-create if needed...
4. (tutoriel) Waiting for an IP...
5. Machine "tutoriel" was started.
6. Waiting for SSH to be available...
7. Detecting the provisioner...
8. Started machines may have new IP addresses. You may need to re-run the `docker-machine env`
command.
9.
10. $ docker-machine ls

```

11. NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER
ERRORS						
12. mydocker	-	generic	Running	tcp://192.168.1.201:2376		v1.10.3
13. tutorial	*	virtualbox	Running	tcp://192.168.99.100:2376		v1.11.0

Il aurait été plus simple de la redémarrer :

```

1. $ docker-machine restart tutorial
2. Restarting "tutorial"...
3. (tutorial) Check network to re-create if needed...
4. (tutorial) Waiting for an IP...
5. Waiting for SSH to be available...
6. Detecting the provisioner...
7. Restarted machines may have new IP addresses. You may need to re-run the `docker-machine env`
   command.
8.
9. $ docker-machine ls
10. NAME          ACTIVE  DRIVER      STATE     URL                  SWARM   DOCKER
    ERRORS
11. mydocker      -       generic     Running   tcp://192.168.1.201:2376   v1.10.3
12. tutorial      *       virtualbox   Running   tcp://192.168.99.100:2376   v1.11.0

```

Nous pouvons même mettre à jour docker :

```

1. $ docker-machine upgrade mydocker
2. Waiting for SSH to be available...
3. Detecting the provisioner...
4. Upgrading docker...
5. Restarting docker...
6.
7. xataz@DESKTOP-2JR2J0C MINGW64 /
8. $ docker-machine ls
9. NAME          ACTIVE  DRIVER      STATE     URL                  SWARM   DOCKER
    ERRORS
10. mydocker      -       generic     Running   tcp://192.168.1.201:2376   v1.11.0
11. tutorial      *       virtualbox   Running   tcp://192.168.99.100:2376   v1.11.0

```

Il se peut que le besoin de se connecter en ssh sur la machine se fasse, dans ce cas nous pouvons :

```

1. $ docker-machine ssh tutorial
2.
3.      ##
4.      ##  ##  ##      ==
5.      ##  ##  ##  ##  ===
6.  /#####\
7. ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~} ~~~  === ~~~
8.      \#####/
9.
10.
11. | _ _ _ _ |
12. | | o o | | _ _ _ _ |
13. | | o o | | _ _ _ _ |
14. | | o o | | _ _ _ _ |
15. Boot2Docker version 1.11.0, build HEAD : 32ee7e9 - Wed Apr 13 20:06:49 UTC 2016
16. Docker version 1.11.0, build 4dc5990
17. docker@tutorial:~$

```

Et enfin nous pouvons la supprimer :

```

1. $ docker-machine stop tutorial && docker-machine rm tutorial
2. Stopping "tutorial"...
3. Machine "tutorial" was stopped.
4. About to remove tutorial
5. Are you sure? (y/n): y
6. Successfully removed tutorial
7.
8. $ docker-machine ls

```

9. NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS
10. mydocker	-	generic	Running	tcp://192.168.1.201:2376		v1.11.0	

## XIII-D - Les Plugins

Comme précédemment cités, nous pouvons installer des plugins, nous installerons ici celui de **scaleway**.

### XIII-D-1 - Installation

Normalement, lorsqu'on vous fournit un plugin pour docker, tout est indiqué dans le *readme.md* (ce qui est ici le cas).

Ici nous avons plusieurs solutions, soit par homebrew (OS X uniquement), soit en téléchargeant les binaires (Linux/Unix/OS X), soit en compilant les sources.

#### XIII-D-1-a - Avec les binaires

Scaleway ne fournissant plus de binaire pour Windows, ceci n'est compatible qu'avec Linux/Unix/OS X.

```
1. $ wget https://github.com/scaleway/docker-machine-driver-scaleway/releases/download/v1.3/docker-machine-driver-scaleway_1.3_${uname -s}_${uname -m}.zip
2. $ unzip docker-machine-driver-scaleway_1.3_${uname -s}_${uname -m}.zip
3. $ cp docker-machine-driver-scaleway_1.3_${uname -s}_${uname -m}/docker-machine-driver-scaleway /usr/local/bin/docker-machine-driver-scaleway
4. $ chmod +x /usr/local/bin/docker-machine-driver-scaleway
```

#### XIII-D-1-b - Avec homebrew

Homebrew étant disponible que sous OS X, ceci ne fonctionne que sur celui-ci.

```
1. $ brew tap scaleway/scaleway
2. $ brew install scaleway/scaleway/docker-machine-driver-scaleway
```

#### XIII-D-1-c - Via les sources

Si vous êtes sous Windows, pas le choix, il faudra compiler, mais avant de commencer, il faudra installer **golang**, ainsi que **git** (je passe ici les détails d'installation).

On configure golang (ici sous powershell) :

```
1. PS C:\Users\xataz\go> $env:GOPATH="c:\Users\xataz\go"
2. PS C:\Users\xataz\go> $env:GOBIN="c:\Users\xataz\go\bin"
3. PS C:\Users\xataz\go> $env:PATH=$env:PATH+";"+$env:GOBIN
4. PS C:\Users\xataz\go> go env
5. set GOARCH=amd64
6. set GOBIN=c:\Users\xataz\go\bin
7. set GOEXE=.exe
8. set GOHOSTARCH=amd64
9. set GOHOSTOS=Windows
10. set GOOS=Windows
11. set GOPATH=c:\Users\xataz\go
12. set GORACE=
13. set GOROOT=F:\Programs\Go
14. set GOTOOLDIR=F:\Programs\Go\pkg\tool\windows_amd64
15. set CC=gcc
16. set GOGCCFLAGS=-m64 -mthreads -fmessage-length=0
17. set CXX=g++
18. set CGO_ENABLED=1
19. PS C:\Users\xataz\go>
```

Vous pouvez ajouter les variables d'environnement via l'interface, ici elles ne seront pas persistantes, dès que vous lancerez un nouveau powershell, il faudra les recharger.

Puis on peut compiler notre driver (compter quelques minutes) :

```
1. PS C:\Users\xataz\go> go get -u github.com/scaleway/docker-machine-driver-scaleway
2. PS C:\Users\xataz\go>
```

Et normalement c'est bon, on teste :

```
1. PS C:\Users\xataz\go> docker-machine create -d scaleway --help
2. Usage: docker-machine create [OPTIONS] [arg...]
3.
4. Create a machine
5.
6. Description:
7.   Run 'C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe create --driver name'
   to include the create flags for that driver in the help text.
8.
9. Options:
10.
11.   --driver, -d "none"
       Driver to create machine with. [$MACHINE_DRIVER]
12.   --engine-env [--engine-env option --engine-env option]
       Specify environment variables to set in the engine
13.   --engine-insecure-registry [--engine-insecure-registry option --engine-insecure-registry
   option] Specify insecure registries to allow with the created engine
14.   --engine-install-url "https://get.docker.com"
       Custom URL to use for engine installation [$MACHINE_DOCKER_INSTALL_URL]
15.   --engine-label [--engine-label option --engine-label option]
       Specify labels for the created engine
16.   --engine-opt [--engine-opt option --engine-opt option]
       Specify arbitrary flags to include with the created engine in the form flag=value
17.   --engine-registry-mirror [--engine-registry-mirror option --engine-registry-mirror option]
       Specify registry mirrors to use [$ENGINE_REGISTRY_MIRROR]
18.   --engine-storage-driver
       Specify a storage driver to use with the engine
19.   --scaleway-commercial-type "VC1S"
       Specifies the commercial type [$SCALEWAY_COMMERCIAL_TYPE]
20.   --scaleway-debug
       Enables Scaleway client debugging [$SCALEWAY_DEBUG]
21.   --scaleway-image "ubuntu-xenial"
       Specifies the image [$SCALEWAY_IMAGE]
22.   --scaleway-ip
       Specifies the IP address [$SCALEWAY_IP]
23.   --scaleway-ipv6
       Enable ipv6 [$SCALEWAY_IPV6]
24.   --scaleway-name
       Assign a name [$SCALEWAY_NAME]
25.   --scaleway-organization
       Scaleway organization [$SCALEWAY_ORGANIZATION]
26.   --scaleway-port "22"
       Specifies SSH port [$SCALEWAY_PORT]
27.   --scaleway-region "par1"
       Specifies the location (par1,ams1) [$SCALEWAY_REGION]
28.   --scaleway-token
       Scaleway token [$SCALEWAY_TOKEN]
29.   --scaleway-user "root"
       Specifies SSH user name [$SCALEWAY_USER]
30.   --scaleway-volumes
       Attach additional volume (e.g., 50G) [$SCALEWAY_VOLUMES]
31.   --swarm
       Configure Machine to join a Swarm cluster
32.   --swarm-addr
       addr to advertise for Swarm (default: detect and use the machine IP)
33.   --swarm-discovery
       Discovery service to use with Swarm
34.   --swarm-experimental
       Enable Swarm experimental features
```

```

35. --swarm-host "tcp://0.0.0.0:3376"
    ip/socket to listen on for Swarm master
36. --swarm-image "swarm:latest"
    Specify Docker image to use for Swarm [$MACHINE_SWARM_IMAGE]
37. --swarm-join-opt [--swarm-join-opt option --swarm-join-opt option]
    Define arbitrary flags for Swarm join
38. --swarm-master
    Configure Machine to be a Swarm master
39. --swarm-opt [--swarm-opt option --swarm-opt option]
    Define arbitrary flags for Swarm master
40. --swarm-strategy "spread"
    Define a default scheduling strategy for Swarm
41. --tls-san [--tls-san option --tls-san option]
    Support extra SANs for TLS certs

```

## XIII-D-2 - Utilisation

L'utilisation n'est pas si différente que sous virtualbox, mais nous aurons des options obligatoires :

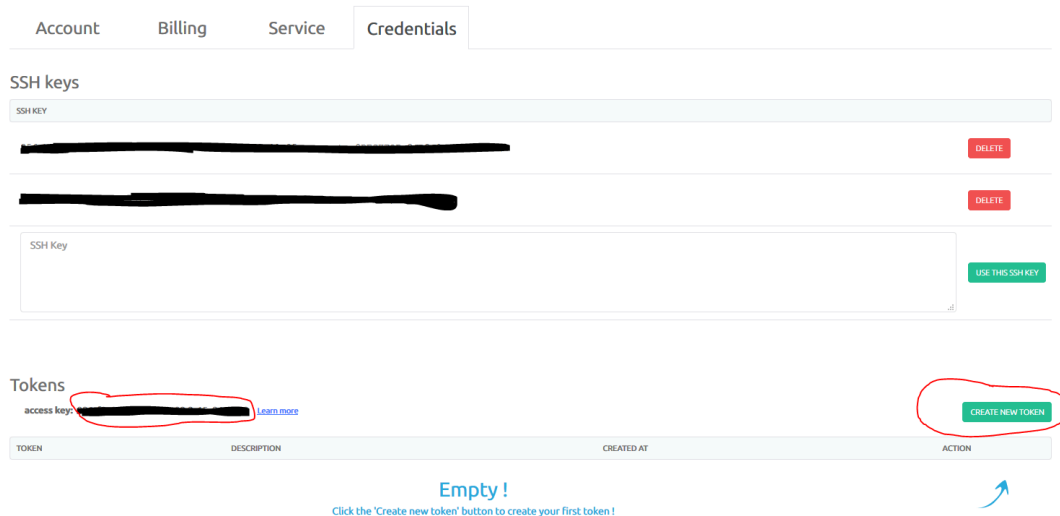
```

1. --scaleway-organization
    Scaleway organization [$SCALEWAY_ORGANIZATION]
2. --scaleway-token
    Scaleway token [$SCALEWAY_TOKEN]

```

Puis d'autres possibilités, toutes celles commençant par `--scaleway` en fait.

On commence par récupérer l'organization et le token, tout se passe sur le site de scaleway :



L'organization est l'**access key** entouré. Pour le token, il faudra le générer en cliquant sur **Create new token**

Puis on crée notre première machine :

```

1. PS C:\Users\xataz\go> docker-machine create -d scaleway --scaleway-token "montoken" --
scaleway-organization "monaccesskey" test
2. Running pre-create checks...
3. Creating machine...
4. (test) Creating SSH key...
5. (test) Creating server...
6. (test) Starting server...
7. Waiting for machine to be running, this may take a few minutes...

```

*Pendant ce temps, vous pouvez vérifier sur votre dashboard scaleway, vous verrez que votre machine est en cours de création.*

Et on voit que ça se termine :

```
1. Detecting operating system of created instance...
2. Waiting for SSH to be available...
3. Detecting the provisioner...
4. Provisioning with ubuntu(systemd)...
5. Installing Docker...
6. Copying certs to the local machine directory...
7. Copying certs to the remote machine...
8. Setting Docker configuration on the remote daemon...
9. Checking connection to Docker...
10. Docker is up and running!
11. To see how to connect your Docker Client to the Docker Engine running on this virtual
    machine, run: C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe env test
```

On teste :

```
1. PS C:\Users\xataz\go> docker-machine env test | Invoke-Expression
2. PS C:\Users\xataz\go> docker version
3. Client:
4.  Version:      1.12.3
5.  API version:  1.24
6.  Go version:   gol.6.3
7.  Git commit:   6b644ec
8.  Built:        Wed Oct 26 23:26:11 2016
9.  OS/Arch:      windows/amd64
10.
11. Server:
12.  Version:      1.12.3
13.  API version:  1.24
14.  Go version:   gol.6.3
15.  Git commit:   6b644ec
16.  Built:        Wed Oct 26 22:01:48 2016
17.  OS/Arch:      linux/amd64
18. PS C:\Users\xataz\go> docker info
19. Containers: 0
20.  Running: 0
21.  Paused: 0
22.  Stopped: 0
23. Images: 0
24. Server Version: 1.12.3
25. Storage Driver: aufs
26.  Root Dir: /var/lib/docker/aufs
27.  Backing Filesystem: extfs
28.  Dirs: 0
29.  Dirperm1 Supported: true
30. Logging Driver: json-file
31. Cgroup Driver: cgroupfs
32. Plugins:
33.  Volume: local
34.  Network: null bridge host overlay
35. Swarm: inactive
36. Runtimes: runc
37. Default Runtime: runc
38. Security Options: seccomp
39. Kernel Version: 4.8.3-docker-1
40. Operating System: Ubuntu 16.04 LTS
41. OSType: Linux
42. Architecture: x86_64
43. CPUs: 2
44. Total Memory: 1.954 GiB
45. Name: test
46. ID: SPZ4:P4AY:2S6V:LYGF:4QCK:SODF:JA7S:M6MH:4K2Y:OKAO:X2XO:DVXC
47. Docker Root Dir: /var/lib/docker
48. Debug Mode (client): false
49. Debug Mode (server): false
```



```
50. Registry: https://index.docker.io/v1/
51. Labels:
52. provider=scaleway (VC1S)
53. Insecure Registries:
54. 127.0.0.0/8
```

Nous pouvons créer des environnements de variables encore une fois, ce qui nous simplifiera la tâche et de ne pas copier toujours le **token** et l'**access key**

```
1. PS C:\Users\xataz\go> $env:SCALEWAY_ORGANIZATION="ton access key"
2. PS C:\Users\xataz\go> $env:SCALEWAY_TOKEN="ton token"
```

*Petite astuce hors sujet : vous pouvez ajouter une variable d'environnement permanente en powershell, comme ceci :*



```
"[Environment]::SetEnvironmentVariable("SCALEWAY_ORGANIZATION", "ton access key", "User")
```

*Il vous faudra par contre, relancer votre console powershell.*

On recrée une machine sans ajouter ces options :

```
1. PS C:\Users\xataz\go> docker-machine create -d scaleway test2
2. Running pre-create checks...
3. Creating machine...
4. (test2) Creating SSH key...
5. (test2) Creating server...
6. (test2) Starting server...
7. Waiting for machine to be running, this may take a few minutes...
8. Detecting operating system of created instance...
9. Waiting for SSH to be available...
10. Detecting the provisioner...
11. Provisioning with ubuntu(systemd)...
12. Installing Docker...
13. [...]
```

Et voilà c'est plus simple maintenant, vous pouvez ajouter toutes les options dans des variables d'environnement.

On va aller un peu plus loin, comme nous avons vu, par défaut, il crée une instance VC1S, avec ubuntu 16.04. Nous allons créer une instance V1M, sous CentOS, et aux Pays-Bas.

```
PS C:\Users\xataz\go> docker-machine create -d scaleway --scaleway-name docker-centos --
scaleway-commercial-type "VC1M" --scaleway-image "centos" --scaleway-region "ams1" docker-centos
```

Sur le github du plugin, il explique même comment créer une instance ARM sous docker :

```
$ docker-machine create -d scaleway --scaleway-commercial-type=C1 --scaleway-image=docker --
engine-install-url="http://bit.ly/1sf3j8V" arm-machine
```

Maintenant que nos tests sont faits, on n'oublie pas de supprimer les machines :

```
1. PS C:\Users\xataz\go> docker-machine rm -y test test2 docker-centos
2. About to remove test, test2, docker-centos
3. Successfully removed test
4. Successfully removed test2
5. Successfully removed docker-centos
```

*Je vous conseille de vérifier si les machines/volumes et IP sont bien supprimés, il m'est arrivé que les IP étaient toujours présentes.*

## XIII-E - Conclusion

Nous avons vu ici comment gérer de multiples machines avec docker-machine sur un environnement VirtualBox. Mais comme nous venons de le voir, nous pouvons créer/gérer des machines sur divers hébergeurs, soit nativement comme digitalOcean ou Azure, ou via un plugin comme pour scaleway. Ceci est très rapide pour créer une nouvelle machine, et nous évite des étapes fastidieuses.

## XIV - Clustering avec Swarm

Dans ce chapitre, nous apprendrons à créer un cluster docker. Nous utiliserons virtualbox comme provider. Cette partie n'est qu'une ébauche sur l'utilisation de swarm. Ceci ne sera pas forcément utile pour tout le monde, mais il peut être utile de comprendre le principe.

### XIV-A - Qu'est-ce que Swarm ?

Swarm est l'outil natif de gestion de cluster docker. Il permet de gérer l'ordonnancement des tâches ainsi que l'allocation de ressources par conteneur. Pour simplifier, nous laissons swarm choisir la machine qui exécutera notre conteneur, nous voyons donc l'ensemble des hôtes Docker (appelés node) en un seul et unique hôte.

Nous créerons d'abord un cluster simple, mais tout à la main, ce qui permettra d'en comprendre le fonctionnement. Ensuite nous créerons un autre cluster avec docker-machine (plus rapide), mais plus complexe, avec une gestion réseau internode.

### XIV-B - Docker swarm (docker >= 1.12.X)

La commande docker swarm est une nouveauté de Docker 1.12, qui permet la simplification de l'utilisation de swarm. Ceci nécessite par contre d'apprendre de nouveaux concepts et de nouvelles commandes.

Voici les nouvelles commandes :

```
1. $ docker swarm # Permet la gestion du cluster
2. $ docker service # Permet la gestion des conteneurs
3. $ docker node # Permet la gestion des Nodes
```

#### XIV-B-1 - Créons notre cluster

Pour cet exemple, nous créerons trois machines, une maître, et deux nœuds (toujours avec notre boucle de for loop) :

```
$ for machine in master node1 node2; do docker-machine create -d virtualbox --virtualbox-memory "512" ${machine}; done
```

Maintenant que nos machines sont créées, nous allons configurer swarm, on commence par le master :

```
1. $ docker-machine ssh master
2. docker@master:~$ docker swarm init --advertise-addr 192.168.99.105 # on utilise l'IP de la machine hôte
3. Swarm initialized: current node (btb5ek9guet2ymqijqtagbv2i) is now a manager.
4.
5. To add a worker to this swarm, run the following command:
6.
7.     docker swarm join \
8.     --token
9.     SWMTKN-1-2ihwqcfv3j26q95e46gi5xp5owm3e9ggjg5aezbncio9qn21q-5gjf3y8yja2g1 \
10.     192.168.99.105:2377
```

```
11. To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Ceci nous retourne directement la commande à exécuter sur les nodes, c'est pas magnifique ?!

```
1. $ docker-machine ssh node1
2. docker@node1:~$ docker swarm join \
3. > --token
   SWMTKN-1-2ihwqcfv3jh26q95e46gi5xp5owm3e9ggjg5aezbncio9qn21q-5gjfybsp34q2rf3y8yja2t2gl \
4. > 192.168.99.105:2377
5. This node joined a swarm as a worker.
```

Et on fait de même sur le node2 :

```
1. $ docker-machine ssh node2
2. docker@node2:~$ docker swarm join \
3. > --token
   SWMTKN-1-2ihwqcfv3jh26q95e46gi5xp5owm3e9ggjg5aezbncio9qn21q-5gjfybsp34q2rf3y8yja2t2gl \
4. > 192.168.99.105:2377
5. This node joined a swarm as a worker.
```

Et voilà notre cluster est créé, pour vérifier nous nous connecterons au master :

```
1. $ docker-machine ssh master
2. docker@master:~$ docker node ls
3. ID                                HOSTNAME    STATUS    AVAILABILITY    MANAGER STATUS
4. 694gceuhyszshrb4l0v9kpmu71       node2      Ready    Active
5. 6k228dx8eteh3q4bgjumd0bks       node1      Ready    Active
6. btb5ek9guet2ymqijqtagbv2i *     master     Ready    Active           Leader
```

Nous pouvons promouvoir les nodes en manager rapidement :

```
1. docker@master:~$ docker node promote node2
2. Node node2 promoted to a manager in the swarm.
3. docker@master:~$ docker node ls
4. ID                                HOSTNAME    STATUS    AVAILABILITY    MANAGER STATUS
5. 694gceuhyszshrb4l0v9kpmu71       node2      Ready    Active           Reachable
6. 6k228dx8eteh3q4bgjumd0bks       node1      Ready    Active
7. btb5ek9guet2ymqijqtagbv2i *     master     Ready    Active           Leader
```

Le node2 est maintenant un manager, mais en slave. Nous pouvons donc maintenant contrôler nos nodes/services depuis le node2.

## XIV-B-2 - Les services

Le concept de service est nouveau, un service est un conteneur scalable, je peux par exemple créer un service nginx avec cinq conteneurs, qui sera ensuite disponible via une VIP (virtual IP), qui permet de faire un HAproxy.

Toujours connecté au serveur master, nous allons créer notre premier service :

```
1. docker@master:~$ docker service create --replicas 1 --name web -p 80:8080 xataz/nginx:mainline
2. cwrfielhxn8gb2rle8jy6safx
```

vérifions si notre service tourne :

```
1. docker@master:~$ docker service ls
2. ID                                NAME        REPLICAS  IMAGE                COMMAND
3. cwrfielhxn8g  web        1/1       xataz/nginx:mainline
4. docker@master:~$ docker service ps web
5. ID                                NAME        IMAGE                NODE        DESIRED STATE  CURRENT STATE
6. ERROR
```

```
6. 7b2osbdqc3sg84mixgzxip1fk web.1 xataz/nginx:mainline node1 Running Running 12
seconds ago
```

Nous voyons ici, avec `docker service ls` que nous avons un service qui s'appelle `web`. Puis dans ce service `web`, avec `docker service ps web`, nous voyons notre conteneur qui tourne sur le `node1`.

Nous pouvons le multiplier :

```
1. docker@master:~$ docker service scale web=5
2. web scaled to 5
```

Puis vérifions :

```
1. docker@master:~$ docker service ls
2. ID NAME REPLICAS IMAGE COMMAND
3. cwrfielhxn8g web 5/5 xataz/nginx:mainline
4. docker@master:~$ docker service ps web
5. ID NAME IMAGE NODE DESIRED STATE CURRENT STATE
ERROR
6. 7b2osbdqc3sg84mixgzxip1fk web.1 xataz/nginx:mainline node1 Running Running 5
minutes ago
7. cwpw2g0mw5gqnds0mvdem5gyx web.2 xataz/nginx:mainline master Running Running 26
seconds ago
8. 2y4mhbc6liaohkls5io76vwbu web.3 xataz/nginx:mainline node2 Running Running 26
seconds ago
9. f01kh9kn8pprj528mnl3xnzj1 web.4 xataz/nginx:mainline node2 Running Running 26
seconds ago
10. avn8ukbljt9zd7ct462h07e01 web.5 xataz/nginx:mainline node1 Running Running 26
seconds ago
```

Nous voyons maintenant, à l'aide de `docker service ls`, que le service `web` possède cinq réplicats, et qu'ils tournent tous. Puis avec `docker service ps web`, nous voyons les cinq conteneurs tourner.

Ceci était simplement pour l'exemple, nous allons aller un peu plus loin, nous commençons par supprimer le service :

```
1. docker@master:~$ docker service rm web
2. web
3. docker@master:~$ docker service ls
4. ID NAME REPLICAS IMAGE COMMAND
```

On va relancer trois conteneurs `nginx`, mais en redirigeant le port 80 vers le 8080 :

```
1. $ docker service create --name web --replicas 5 -p 80:8080 xataz/nginx:mainline
2. 9emtiwgv4jtvzj16oxg6h7og
3. $ docker service ls
4. ID NAME REPLICAS IMAGE COMMAND
5. 9emtiwgv4jtv web 5/5 xataz/nginx:mainline
6. $ docker service ps web
7. ID NAME IMAGE NODE DESIRED STATE CURRENT STATE
ERROR
8. 4ptfdheqs3yrocevde8wbsk7n web.1 xataz/nginx:mainline node1 Running Running 11
seconds ago
9. 9b8e8s5cmku9vk60s5tk2f7o0 web.2 xataz/nginx:mainline master Running Running 10
seconds ago
10. 8fty69tnwcg8szoo0lffaf0nc web.3 xataz/nginx:mainline node2 Running Running 10
seconds ago
11. 6l922h3ih828cyec65sru89ss web.4 xataz/nginx:mainline node2 Running Running 10
seconds ago
12. blk42isuztm35ou3z56npzf67 web.5 xataz/nginx:mainline master Running Running 10
seconds ago
```

Comment est-ce possible de bind plusieurs fois le port 80 ? En fait, docker service est beaucoup plus intelligent, il a en fait créé une VIP (Virtual IP) par node

pour le service web, et c'est sur cette IP qu'il bind le port 80. Celle-ci redirige à tour de rôle vers un conteneur ou un autre.

Pour tester rapidement :

```
1. i=0; while [ $i -lt 10 ]; do curl http://192.168.99.105 2> /dev/null | grep h1; sleep
1; i=$((i+1)); done
2. <h1>Nginx 1.11.3 on 1e5f3f6f3375</h1>
3. <h1>Nginx 1.11.3 on 78792c3765a5</h1>
4. <h1>Nginx 1.11.3 on e35b7b05243d</h1>
5. <h1>Nginx 1.11.3 on 49df7c284fb4</h1>
6. <h1>Nginx 1.11.3 on f638593093a3</h1>
7. <h1>Nginx 1.11.3 on 1e5f3f6f3375</h1>
8. <h1>Nginx 1.11.3 on 78792c3765a5</h1>
9. <h1>Nginx 1.11.3 on e35b7b05243d</h1>
10. <h1>Nginx 1.11.3 on 49df7c284fb4</h1>
11. <h1>Nginx 1.11.3 on f638593093a3</h1>
```

Comme on peut le voir, il tourne entre les conteneurs. Et cela fonctionne également avec l'IP d'un node :

```
1. i=0; while [ $i -lt 10 ]; do curl http://192.168.99.106 2> /dev/null | grep h1; sleep
1; i=$((i+1)); done
2. <h1>Nginx 1.11.3 on 1e5f3f6f3375</h1>
3. <h1>Nginx 1.11.3 on 78792c3765a5</h1>
4. <h1>Nginx 1.11.3 on e35b7b05243d</h1>
5. <h1>Nginx 1.11.3 on 49df7c284fb4</h1>
6. <h1>Nginx 1.11.3 on f638593093a3</h1>
7. <h1>Nginx 1.11.3 on 1e5f3f6f3375</h1>
8. <h1>Nginx 1.11.3 on 78792c3765a5</h1>
9. <h1>Nginx 1.11.3 on e35b7b05243d</h1>
10. <h1>Nginx 1.11.3 on 49df7c284fb4</h1>
11. <h1>Nginx 1.11.3 on f638593093a3</h1>
```

Comme vous pouvez le voir, c'est vraiment simple de faire un cluster avec cette nouvelle fonction.

## XIV-C - Conclusion

Nous avons vu ici comment créer un cluster swarm, mais il existe d'autres outils pour faire ceci, comme rancher, ou kubernetes.

Nous pourrions également aller beaucoup plus loin, en utilisant par exemple des outils de clustering de fs, comme ceph ou glusterfs, mais ceci serait hors sujet.

## XV - Registry

Nous allons ici créer notre propre registry, c'est-à-dire un hub (sans webUI) autohébergé. Ce sera un petit chapitre.

### XV-A - Installation de registry

L'installation est simple, il suffit de lancer un conteneur :

```
1. $ docker run -d -p 5000:5000 -v /data/registry:/var/lib/registry --name registry registry:2
2. Unable to find image 'registry:2' locally
3. 2: Pulling from library/registry
4. e110a4a17941: Already exists
5. 2ee5ed28ffa7: Pulling fs layer
6. d1562c23a8aa: Pulling fs layer
7. 06ba8e23299f: Pulling fs layer
8. 802d2a9c64e8: Pulling fs layer
9. 802d2a9c64e8: Waiting
10. 06ba8e23299f: Download complete
11. 2ee5ed28ffa7: Download complete
```

```
12. 2ee5ed28ffa7: Pull complete
13. dl562c23a8aa: Verifying Checksum
14. dl562c23a8aa: Download complete
15. 802d2a9c64e8: Download complete
16. dl562c23a8aa: Pull complete
17. 06ba8e23299f: Pull complete
18. 802d2a9c64e8: Pull complete
19. Digest: sha256:1b68f0d54837c356e353efb04472bc0c9a60ae1c8178c9ce076b01d2930bcc5d
20. Status: Downloaded newer image for registry:2
21. e24f1ad2a82396361d74b59152baff0e4fa3f67cd743450da238a76e9142f02a
```

Et voilà c'est installé, il écoute le port 5000 sur localhost (ou IP conteneur:5000).

## XV-B - Utilisation

L'utilisation est plutôt simple, tout va se jouer avec le nom de repository, où habituellement on met notre nom d'utilisateur. Nous allons donc renommer notre image lutim en localhost:5000 :

```
$ docker tag xataz/lutim localhost:5000/lutim
```

On peut maintenant la push sur notre registry :

```
1. $ docker push localhost:5000/lutim
2. The push refers to a repository [localhost:5000/lutim]
3. d581c9307ce6: Preparing
4. 232aeb6486cf: Preparing
5. 07bd11c49c2e: Preparing
6. 7d887264d1fb: Preparing
7. 92abf950c77c: Preparing
8. 2f71b45e4e25: Preparing
9. 2f71b45e4e25: Waiting
10. 232aeb6486cf: Pushed
11. 07bd11c49c2e: Pushed
12. d581c9307ce6: Pushed
13. 92abf950c77c: Pushed
14. 7d887264d1fb: Pushed
15. 2f71b45e4e25: Pushed
16. latest: digest: sha256:313f64018302aa8c3fdef7baa308c3436b067ace706067c0a0e7737bd563acd6
    size: 1573
```

Pour tester notre registry, nous allons supprimer l'image en local (l'original et la retagguée), et la retélécharger :

```
1. $ docker rmi localhost:5000/lutim xataz/lutim && docker pull localhost:5000/lutim
2. Untagged: localhost:5000/lutim:latest
3. Untagged: localhost:5000/
lutim@sha256:313f64018302aa8c3fdef7baa308c3436b067ace706067c0a0e7737bd563acd6
4. Untagged: xataz/lutim:latest
5. Untagged: xataz/lutim@sha256:313f64018302aa8c3fdef7baa308c3436b067ace706067c0a0e7737bd563acd6
6. Deleted: sha256:b22de6f27e376ce9d875cd647d0d7aca894e5ea0f2071eb83763d14092410188
7. Deleted: sha256:870865d9861d1ebb21012d5d3fa04c453fdd6a254623024e6522c2e76bc5db8e
8. Deleted: sha256:fed8ca9580ad53059c3dd107b45f991b87538260cdc4ab4fd44abae5631f6701
9. Deleted: sha256:b6622b6004255e1523dc17f3f7a5960b95ba51503838b4c7e19eff447c483d7e
10. Deleted: sha256:1d9f78dac3a778b636d02e64db2569de7e54464a50037be90facea3838390808
11. Deleted: sha256:be9cfba7137b53173b10101ce96fa17a3bedfb13cf0001a69096dee3a70b37be
12. Using default tag: latest
13. latest: Pulling from lutim
14. 357ea8c3d80b: Already exists
15. c9a3b87a9863: Pulling fs layer
16. c828912554c6: Pulling fs layer
17. a0ec173a645d: Pulling fs layer
18. 7848baf27247: Pulling fs layer
19. 5b2cd2a8ffca: Pulling fs layer
20. 7848baf27247: Waiting
21. 5b2cd2a8ffca: Waiting
22. a0ec173a645d: Verifying Checksum
23. a0ec173a645d: Download complete
```

```
24. 7848baf27247: Verifying Checksum
25. 7848baf27247: Download complete
26. 5b2cd2a8ffca: Verifying Checksum
27. 5b2cd2a8ffca: Download complete
28. c9a3b87a9863: Verifying Checksum
29. c9a3b87a9863: Download complete
30. c828912554c6: Verifying Checksum
31. c828912554c6: Download complete
32. c9a3b87a9863: Pull complete
33. c828912554c6: Pull complete
34. a0ec173a645d: Pull complete
35. 7848baf27247: Pull complete
36. 5b2cd2a8ffca: Pull complete
37. Digest: sha256:313f64018302aa8c3fdef7baa308c3436b067ace706067c0a0e7737bd563acd6
38. Status: Downloaded newer image for localhost:5000/lutim:latest
```

## XV-C - Conclusion

Ceci n'est qu'une ébauche, afin de vous montrer simplement que ceci est possible, si vous souhaitez plus d'informations, vous pouvez consulter la [documentation](#).

## XVI - Bonus

Dans cette partie que je nomme bonus, je mettrai quelques astuces, ou des trucs ^^ . En fait il y aura un peu de tout et de rien en rapport avec docker, mais qui ne rentre dans aucune des autres parties.

### XVI-A - L'autocomplétion

Ceci vous permettra d'avoir la liste des commandes qui s'affichent lorsque vous appuyez sur tab. exemple :

```
1. $ docker
2. attach cp diff export images inspect login network ps rename rmi
   search stop unpause wait
3. build create events help import kill logout pause pull restart run
   start tag version
4. commit daemon exec history info load logs port push rm save
   stats top volume
```

On installera donc l'autocomplétion pour docker et docker-compose.

En premier, et pour pas se faire avoir comme moi et passer trois heures à chercher le problème, il faut installer bash-completion :

```
$ apt-get install bash-completion
```

Ensuite on télécharge les fichiers d'autocomplétion (en root) :

```
1. $ curl -L https://raw.githubusercontent.com/docker/docker-ce/master/components/cli/contrib/
completion/bash/docker > /etc/bash_completion.d/docker
2. $ wget -O /etc/bash_completion.d/docker-compose https://raw.githubusercontent.com/docker/
compose/1.14.0/contrib/completion/bash/docker-compose
```



**Attention à votre version de docker-compose, version 1.14 à l'heure où j'écris ces lignes.**

Et on ajoute dans le .profile de notre utilisateur :

```
$ echo ". /etc/bash_completion" >> ~/.profile
```

Pour finir on source notre profile :

```
$ source ~/.profile
```

Enjoy :

```
1. $ docker r
2. rename restart rm rmi run
3. $ docker-compose
4. build kill migrate-to-labels ps restart
   run start up
5. help logs port pull rm
   scale stop version
```

Et voilà le travail !!

## XVI-B - Docker avec btrfs

Comme je vous en ai parlé, Docker fonctionne par défaut sur aufs. Mais il est possible de le faire fonctionner avec btrfs, device-mapper, zfs ou même overlay. Le principe est le même pour tous, mais nous verrons ici comment le configurer pour btrfs.

Mais pourquoi utiliser btrfs à la place de aufs ?! Pour plusieurs raisons, la première est que les images basées sur redhat (centOS, fedora etc.) n'aiment pas trop aufs, et il y a des fois des paquets qui refusent de s'installer, par exemple httpd. Donc il fallait choisir un autre FS, device-mapper, je n'aime pas trop le principe, en gros chaque layer est un disque virtuel (en gros), donc difficile de faire des modifs directement dedans. ZFS est plutôt gourmand, et pas fan de zfs sur Linux (peut-être à tort). Overlay je ne connais pas, mais apparemment c'est ce qu'il y a de plus rapide. Donc je choisis btrfs.

Pour cette partie, il vous faudra une partition vide, de mon côté, j'ai configuré le partitionnement de mon serveur avec lvm, et j'ai préparé une partition de 20 Go qui s'appelle /dev/vg1/docker.



*Si vous avez déjà des conteneurs et des images, vous perdrez tout.*

Toutes ces actions sont à exécuter en root.

On commence par installer btrfs :

```
$ apt-get install btrfs-tools
```

On arrête docker :

```
$ systemctl stop docker.service
```

On formate notre partition :

```
$ mkfs.btrfs /dev/vg1/docker
```

Puis on monte la partition (on le rajoute évidemment dans le fstab) :

```
$ mount /dev/vg1/docker /var/lib/docker
```

Maintenant, on édite le fichier default de docker :



```
1. $ vim /etc/default/docker
2.
3. DOCKER_OPTS--s btrfs
```

Et pour finir on relance docker :

```
$ systemctl start docker
```

Si on vérifie dans le répertoire de docker, normalement vous avez ceci :

```
1. $ ls /var/lib/docker/
2. btrfs containers graph linkgraph.db repositories-btrfs tmp trust volumes
```

## XVII - Conclusion

Normalement, si j'ai bien fait mon travail, vous devriez être totalement capable d'utiliser docker. Il est simple de créer un environnement, de le multiplier, de le partager, de le modifier, sans perte de performance.

En commençant l'écriture de ce tutoriel, je ne pensais pas qu'il serait aussi difficile et long d'expliquer l'utilisation de docker. Mais malgré cette taille énorme (sans sous-entendu), nous n'avons fait qu'effleurer ses possibilités, qui sont grandement suffisantes pour une utilisation personnelle.

Je finirai en beauté, avec cette image trouvée sur un article de wanadev sur docker, qui représente bien la mise en prod avec docker :



## XVIII - Ressources

- Documentation Docker (en) : <https://docs.docker.com/>
- Docker commands reference (en) : <https://docs.docker.com/engine/reference/commandline/>
- Chaîne officielle docker (en) : <https://www.youtube.com/user/dockerrun>
- Chaîne YouTube « Quoi de neuf docker » (fr) : [https://www.youtube.com/channel/UCOAhkxpryr\\_BKybt9wlw-NQ](https://www.youtube.com/channel/UCOAhkxpryr_BKybt9wlw-NQ)
- ImageLayers pour voir la composition d'une image : <https://imagelayers.io>
- Plateforme d'apprentissage (en) : <https://www.katacoda.com>

Quelques github avec des images sympa :

- **Les dockerfiles de Wonderfall** : Plein de bonnes choses
- **Les repos d'Hardware** : avec notamment mailserver et nsd
- **Les dockerfiles de jfrazelle** : Une nana qui fait des images hallucinantes
- **Mes dockerfiles** : Le meilleur (autocongrat)

## XIX - Note de la rédaction de Developpez.com

Nous tenons à remercier **Winjerome** pour la mise au gabarit Developpez.com et **Claude Leloup** pour la relecture orthographique.