|  |
| --- |
| [Nom de la société] |
| Cours de Docker |
| [Sous-titre du document] |

|  |
| --- |
| Mohamed Ziane CHIBANE  15/09/2020 |

Table des matières

[II - Introduction 2](#_Toc51087157)

[II-A - Qu'est-ce que Docker (Wikipédia) ? 2](#_Toc51087158)

[II-C - Docker, pour quoi faire ? 3](#_Toc51087159)

[III - Installation 4](#_Toc51087160)

[III-A - Installation sous Linux 5](#_Toc51087161)

[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. 5](#_Toc51087162)

[III-B - Installation sous Windows 6](#_Toc51087163)

[III-B-1 - Docker4Windows 6](#_Toc51087164)

[III-B-2 - Docker-toolbox 6](#_Toc51087165)

[III-C - Installation sous Mac 7](#_Toc51087166)

[III-C-1 - Docker4Mac 7](#_Toc51087167)

[III-C-2 - Docker-toolbox 7](#_Toc51087168)

[Nous avons ici utilisé docker-machine pour créer notre machine, nous verrons dans un autre chapitre comment utiliser cet outil. 7](#_Toc51087169)

[III-D - Mon environnement de test 8](#_Toc51087170)

[IV - Le docker Hub 8](#_Toc51087171)

[IV-B - Chercher une image 8](#_Toc51087172)

[V - Gérer les images 12](#_Toc51087173)

[V-A - Télécharger des images 12](#_Toc51087174)

[V-B - Lister les images 15](#_Toc51087175)

[V-C - Supprimer les images 15](#_Toc51087176)

[V-D - Conclusion 17](#_Toc51087177)

[VI - Gérer les conteneurs 17](#_Toc51087178)

[VI-A - Lancer, arrêter et lister des conteneurs 17](#_Toc51087179)

[VI-B - Voir les logs des conteneurs 20](#_Toc51087180)

[VI-C - Supprimer les conteneurs 21](#_Toc51087181)

[VI-D - Cas concrets 22](#_Toc51087182)

[VI-D-1 - Premier cas : le développeur 22](#_Toc51087183)

[VI-D-2 - Deuxième cas : installer une application 24](#_Toc51087184)

[VI-D-3 - Troisième cas : le déploiement 25](#_Toc51087185)

[VI-E - Conclusion 26](#_Toc51087186)

[VII - Créer une image 26](#_Toc51087187)

[VII-A - Création d'un Dockerfile 26](#_Toc51087188)

[VII-A-2 - Exemple d'une image lutim 28](#_Toc51087189)

[VII-A-3 - Créons une image de base 30](#_Toc51087190)

[VII-B - Les bonnes pratiques 32](#_Toc51087191)

[VII-B-1 - Limiter les layers 32](#_Toc51087192)

[VII-B-2 - Limiter la taille d'une image 33](#_Toc51087193)

# II - Introduction

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.

# II-A - Qu'est-ce que Docker (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.

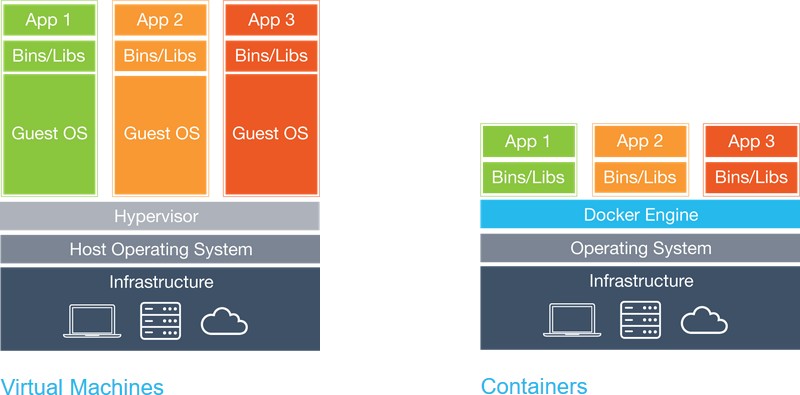
Pour une machine virtuelle, vous créez comme son nom l'indique, une machine virtuelle, c'est-à-dire, vous lui indiquez la ram à utiliser, le nombre de cpu, et vous créez un disque dur virtuel pour installer un OS. Votre machine dispose de ses propres ressources, et n'a aucunement conscience d'être virtualisée.

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/**](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.
4. 03f4658f8b78: Pull complete
5. a3ed95caeb02: Pull complete

7. Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7

8. Status: Downloaded newer image **for** hello-world:latest 9.

1. Hello from Docker.
2. This message shows that your installation appears to be working correctly. 12.
3. To generate this message, Docker took the following steps:
4. 1. The Docker client contacted the Docker daemon.
5. 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
6. 3. The Docker daemon created a new container from that image which runs the
7. executable that produces the output you are currently reading.
8. 4. The Docker daemon streamed that output to the Docker client, which sent it
9. to your terminal. 20.
10. To try something more ambitious, you can run an Ubuntu container with:
11. ***$*** docker run -it ubuntu bash 23.
12. Share images, automate workflows, and more with a free Docker Hub account:
13. https://hub.docker.com 26.
14. **For** more examples and ideas, visit:
15. 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**](https://download.docker.com/win/beta/InstallDocker.msi), 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**](https://download.docker.com/win/stable/DockerToolbox.exe), 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**](https://download.docker.com/mac/beta/Docker.dmg) 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**](https://www.docker.com/products/docker-toolbox)

Ensuite vous exécutez le pkg et installez docker comme indiqué ici : [**https://docs.docker.com/engine/installation/**](https://docs.docker.com/engine/installation/mac/)[**mac/**](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**](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.
4. 03f4658f8b78: Pull complete
5. a3ed95caeb02: Pull complete

7. Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7

8. Status: Downloaded newer image **for** hello-world:latest 9.

1. Hello from Docker.
2. This message shows that your installation appears to be working correctly. 12.
3. To generate this message, Docker took the following steps:
4. 1. The Docker client contacted the Docker daemon.
5. 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
6. 3. The Docker daemon created a new container from that image which runs the
7. executable that produces the output you are currently reading.
8. 4. The Docker daemon streamed that output to the Docker client, which sent it
9. to your terminal.

20.

1. To try something more ambitious, you can run an Ubuntu container with:
2. ***$*** docker run -it ubuntu bash 23.
3. Share images, automate workflows, and more with a free Docker Hub account:
4. https://hub.docker.com 26.
5. **For** more examples and ideas, visit:
6. 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.

4.

5.

6.

7.

8.

9.

Version:

API version: Go version: Git commit: Built: OS/Arch:

17.05.0-ce

1.29 go1.8.1 89658bed64

Fri May 5 22:40:58 2017

linux/amd64

10. Server:

11.

12.

13.

14.

15.

16.

17.

Version:

API version: Go version: Git commit: Built: OS/Arch:

17.05.0-ce

1.29 (minimum version 1.12) go1.8.1

89658bed64

Fri May 5 22:40:58 2017

linux/amd64

Experimental: false

# IV - Le docker Hub

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

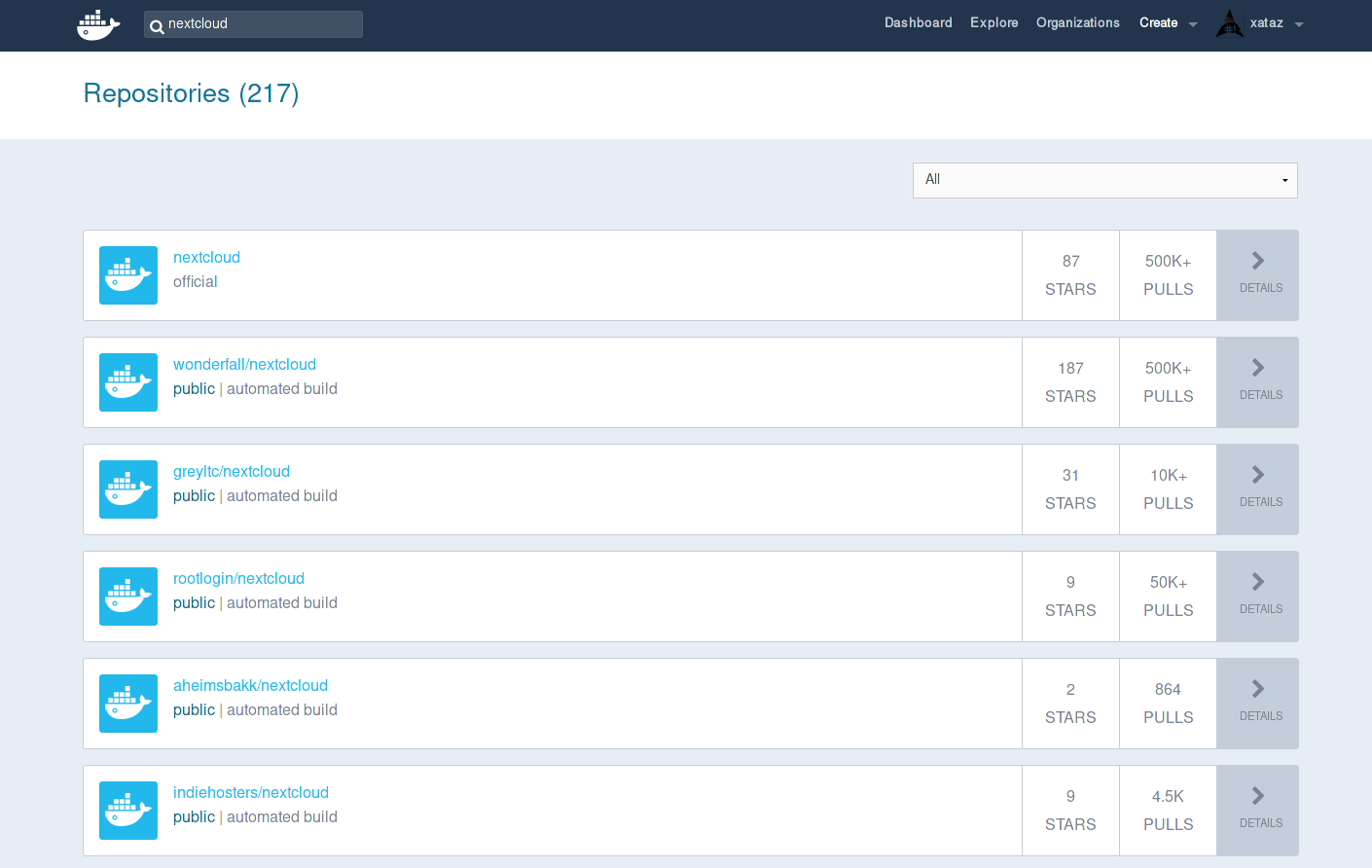
Le docker Hub est un store ou les utilisateurs de docker peuvent partager leurs images. Les images de base ont été créées par l'équipe de docker. Il est accessible ici : [**https://hub.docker.com/explore/**](https://hub.docker.com/explore/)

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

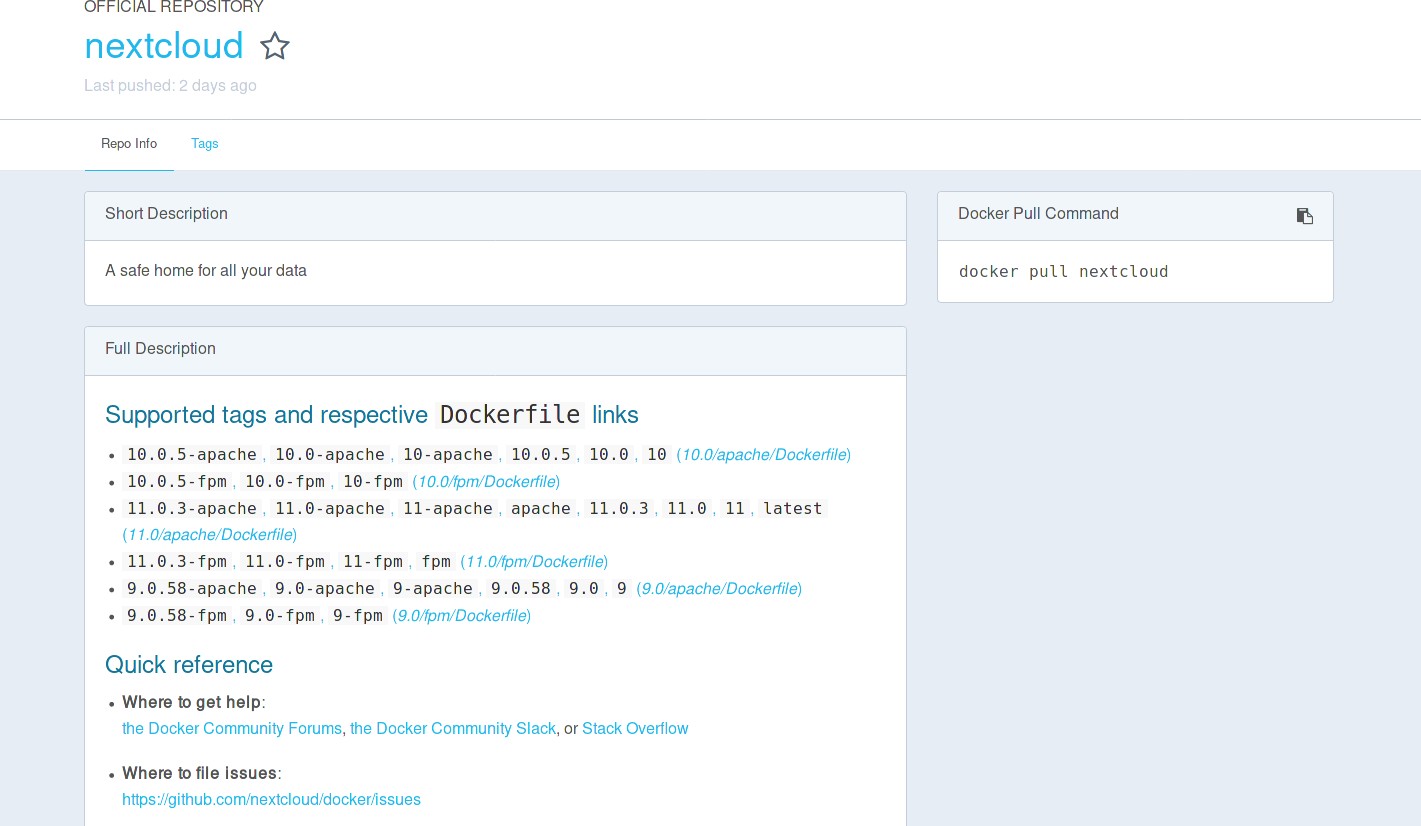
Il existe deux méthodes pour chercher sur le Hub, la première par le site web. Gardons mon exemple, je veux un nextcloud. Sur le site je cherche donc nextcloud, et j'obtiens plusieurs résultats :



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 :



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   OFFICIAL AUTOMATED   1. wonderfall/nextcloud | DESCRIPTION  Nextcloud - a safe home **for** all your data.... | STARS 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]   1. rootlogin/nextcloud Nextcloud docker image running on top of N... 9 [OK] 2. indiehosters/nextcloud Docker image **for** Nextcloud application. 9   [OK]   1. sameersbn/nextcloud Dockerized Nextcloud 3   [OK]   1. freenas/nextcloud Access & share your files, calendars, cont. 3   [OK]   1. aheimsbakk/nextcloud Nextcloud - a safe home **for** all your data. 2   [OK]   1. skybosh/nextcloud A simple Nextcloud image. 2   [OK]   1. cyphar/nextcloud NextCloud is a fork of OwnCloud. This is a. 1   [OK]   1. whatever4711/nextcloud Image **for** Nextcloud with php-fpm 0   [OK]   1. bennibu/nextcloud nextcloud on php 7 0   [OK]   1. jefferyb/nextcloud Docker Image packaging **for** Nextcloud - A s. 0   [OK]   1. mkuron/nextcloud Nextcloud with PHP IMAP extension 0   [OK]   1. dklein/nextcloud Added cron to the official nextcloud Image 0   [OK]   1. gorlug/nextcloud https://nextcloud.com/ 0   [OK]   1. icewind1991/nextcloud-dev Docker image **for** NextCloud **for** development 0   [OK]   1. aknaebel/nextcloud This docker image provide a nextcloud serv. 0   [OK]   1. whatwedo/nextcloud Nextcloud powered by Apache 0   [OK]   1. vger/nextcloud Nextcloud image based on Debian Jessie. 0   [OK]   1. martmaiste/nextcloud Nextcloud Docker using php-fpm and Nginx r. 0   [OK]   1. georgehrke/nextcloud Dockerized Nextcloud 0   [OK] | | | |

1. asannou/nextcloud [OK]
2. ianusit/nextcloud [OK]
3. servercontainers/nextcloud [OK]

A safe home **for** all your data

0

Nextcloud

0

nextcloud container on debian jessie with ...

0

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.

9.

--help

Print usage

10. Commands:

11.

12.

13.

14.

15.

16.

17.

18.

19.

20.

21.

22.

23.

build history import inspect load

ls prune pull push rm save tag

Build an image from a Dockerfile Show the history of an image

Import the contents from a tarball to create a filesystem image Display detailed information on one or more images

Load an image from a tar archive or STDIN List images

Remove unused images

Pull an image or a repository from a registry Push an image or a repository to a registry Remove one or more images

Save one or more images to a tar archive (streamed to STDOUT by default) Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

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

1. b3fd60530d47: Pull complete
2. 1b9a58bad45d: Pull complete
3. dd111b8b6245: Pull complete
4. c9ba8440391e: Pull complete
5. Digest: sha256:dcdd3f4feeacedfb936b802a5c05885db3abcbb909315aed162c2d8938f4ab29
6. 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. 288a1d91ad4e: 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. aee16e9f06fe: 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:dcdd3f4feeacedfb936b802a5c05885db3abcbb909315aed162c2d8938f4ab29
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
2. REPOSITORY VIRTUAL SIZE

TAG

IMAGE ID

CREATED

* 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 |  | | | |
| 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 |
| 5. 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 2. REPOSITORY | ls | uniq -f 3  TAG | IMAGE ID | CREATED | SIZE |
| 3. owncloud | 8.2.5 | 52f7d60d34bd | 13 days ago | 699.2 MB |
| 4. 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 2. REPOSITORY | owncloud  TAG | IMAGE ID | CREATED | SIZE |
| 3. owncloud | 8.2.5 | 52f7d60d34bd | 13 days ago | 699.2 MB |
| 4. owncloud | 9.0.2 | 4e0dc7be3d39 | 3 weeks ago | 698.5 MB |
| 5. 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 2. REPOSITORY TAG | | IMAGE ID | CREATED | SIZE |
| 3. 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:a2fcda4803fdb8e2bef7064c3294b73c212778179b74b28600b9153933aa226a

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: sha256:2155b268258a477900f1eee8624d238c637db24b235f547e876b2696622d8137

On vérifie :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. ***$*** docker image 2. REPOSITORY | ls  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:dcdd3f4feeacedfb936b802a5c05885db3abcbb909315aed162c2d8938f4ab29 7. deleted: sha256:05b0dd17351b75f9b818c0f8075e2d2b9f2ef25873422e2f52c20b12cf5c10bf

8. deleted: sha256:83e371d71cfbc5ac783026006959b8a38f582947ac7fc9d54f871de3df4be9f6

9. deleted: sha256:04be7a548f626aa9ab5e17a492194b51ff61df8c0545eb89f8fc35f1b073e70f

10. deleted: sha256:da435d4b91262cfe06ae635135399124e63afa4bd87ec4e1a161feba38a05036

11. deleted: sha256:82f6d723072ee7aad7cb4e95e6bc37d8760e055122487bfb32d92182551bc5f7

12. deleted: sha256:58603b6a763e5604d0fa3a858a891c7f17a23c5a10dafe16160e14b70f05f269

13. deleted: sha256:14b5b1d7e7f962b42dcbcb0413897bdb7d38584ad3e21f74ec8137306eee14db

14. deleted: sha256:176076b65ddaddff95ef2c19d65478746b61dd26973933fbe18c8d058ef06eb3

15. deleted: sha256:23befc96c53800701fa02c5f7e989c6122256143edcc2c84a8275b01e57f9dba

16. deleted: sha256:73585d624d5a4a0885e4c6de8463a6642dfb0fbef1b5847a300e0097bdbac064

17. deleted: sha256:9716cef6be433295bfb3cd6ab7a9bd0500555bfdbaedf953c4637e18c8a59ba7 18. deleted: sha256:9b03fea1b03a4c74a105dcef8d0df1e89b73124a7f4190a6703d615bff41dd6b 19. deleted: sha256:30963d11485e5a28117e91075497b16386a4d0b30b3ff263b57d19fb3388f3df

20. deleted: sha256:631b87d9cd40ecee449ddb56fb13c233e2ca779b0f6bd5ff3d857bdc6be82464 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.

1. Unable to find image 'debian:latest' locally
2. latest: Pulling from library/debian
3. 10a267c67f42: Pull complete

6. Digest: sha256:476959f29a17423a24a17716e058352ff6fbf13d8389e4a561c8ccc758245937

1. Status: Downloaded newer image **for** debian:latest
2. 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 2. CONTAINER ID IMAGE PORTS | COMMAND NAMES | CREATED | STATUS |
| 3. ce198d73aafc debian  Exited (0) 4 minutes ago | "echo 'bonjour mon..."  pedantic\_snyder | 4 minutes ago |  |

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'   1. bonjour mondedie 1 !!! 2. bonjour mondedie 2 !!! 3. bonjour mondedie 3 !!! 4. bonjour mondedie 4 !!! 5. bonjour mondedie 5 !!! 6. bonjour mondedie 6 !!! 7. bonjour mondedie 7 !!! 8. bonjour mondedie 8 !!! 9. 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
2. CONTAINER ID IMAGE PORTS

debian Exited (0) About a minute ago

4. e8d80d7dc23a debian

COMMAND NAMES

"echo 'bonjour mon..."

jolly\_swartz "echo 'bonjour mon..."

CREATED

STATUS

3. 8464c7bb5c96

About a minute ago

About a minute ago

Exited (0) About a minute ago reverent\_galileo

1. 56199baee7f9 debian "echo 'bonjour mon..." About a minute ago Exited (0) About a minute ago optimistic\_lichterman
2. 64ae77f60124 debian "echo 'bonjour mon..." About a minute ago Exited (0) About a minute ago condescending\_mayer
3. 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

"echo 'bonjour mon..."

sad\_easley "echo 'bonjour mon..."

About a minute ago

About a minute ago

Exited (0) About a minute ago

agitated\_mcclintock

1. 933b9542fd06 debian "echo 'bonjour mon..." About a minute ago Exited (0) About a minute ago goofy\_colden
2. 8e674629468f debian "echo 'bonjour mon..." About a minute ago Exited (0) About a minute ago eloquent\_meninsky
3. 2ecbda8dd3b4 debian Exited (0) About a minute ago

13. 3d1b01dfe606 debian

"echo 'bonjour mon..."

pedantic\_lamport "echo 'bonjour mon..."

About a minute ago

About a minute ago

Exited (0) About a minute ago

vigorous\_sinoussi

1. d69d98bf2aa7 debian Exited (0) About a minute ago
2. da2eb71d7eb5 debian debian

debian

"echo 'bonjour mon..."

heuristic\_raman "echo 'bonjour mon..."

happy\_elion "echo 'bonjour mon..."

wonderful\_tesla "echo 'bonjour mon..."

laughing\_euler "echo 'bonjour mon..."

naughty\_jepsen "echo 'bonjour mon..."

About a minute ago

About a minute ago

Exited (0) About a minute ago

16. 27688ff57cc0

About a minute ago

Exited (0) About a minute ago

17. 64c31a7323b8

debian

About a minute ago

Exited (0) About a minute ago

18. d1f9f173ef1d

About a minute ago

Exited (0) About a minute ago

19. 0387ccf1092a

debian

About a minute ago

Exited (0) About a minute ago mystifying\_khorana

1. eaf59f436d28 debian Exited (0) About a minute ago
2. aac609b95f89 debian Exited (0) About a minute ago

22. 6a6be6f146eb debian

"echo 'bonjour mon..."

hardcore\_murdock "echo 'bonjour mon..."

infallible\_nobel "echo 'bonjour mon..."

tender\_minsky "echo 'bonjour mon..."

pedantic\_snyder

About a minute ago

About a minute ago

About a minute ago

Exited (0) About a minute ago

23. ce198d73aafc

debian

6 minutes ago

Exited (0) 6 minutes ago



*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

"echo 'bonjour monded" jolly\_swartz

6 minutes ago

Exited (0) 2 seconds ago

Donc là, on voit qu'il a été créé il y a 6 minutes, mais qu'il s'est terminé il y 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 PORTS NAMES | | COMMAND | CREATED | STATUS |
| 3. 03b1d375ac58 | debian "ping google.fr"  adoring\_thompson | | 16 seconds ago | Up 16 seconds |

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  PORTS 4. 03b1d375ac58  second | IMAGE COMMAND  NAMES  debian "ping google.fr" adoring\_thompson | CREATED  About a minute ago | STATUS  Up Less than a |

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. 64ae77f60124af6b9f07d3f1be6849508b8f0401bd79494adf488a3c0bc88853 9. 379c6fcfb53fa41a4ef3dc824c7f81338a1a0ff0d94f9267b23a38f900db8b48 10. 94cdbdf938cb94098dc724792a2b973731f19b8d4d1de64e9a0ac2ab6eec8114 11. 938300e4e31ffb59d902329d4dad22160df6b0dc9b29a60780dcc9d809ee8f0d 12. 933b9542fd06126f7dd8453fca83ea11efec73d113a5ec36a739dc281816685f 13. 8e674629468fb61abedbc39819536825dbd1e87dbf3bde15c326a94c26285d41 14. 2ecbda8dd3b4721313e52eef3f443c7921730b35013a8d196a5a1df0bef8146d 15. 3d1b01dfe6064e6f0aa6bd9adecce144e1c001b546eaa66dbf598c73df4f4a10 16. d69d98bf2aa7e0450e857b8459f7f77f4adf32a500048415e2477dce9cedc363 17. da2eb71d7eb5a755dbefffcdab759077d2a7c2ee5db3de7f7570444845f91c6c 18. 27688ff57cc0a9428e51a85635a3ce705fdd7ccd78358c5218be6e1090266b07 19. 64c31a7323b824782ff427909659824495988c8d780d7d46c2f46ffa7e200f6f 20. d1f9f173ef1d0a0798ef9bbc9924612541206a9dba1fd856fb293c4946734213 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**](https://docs.docker.com/engine/reference/run/) 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.
3. **var** server = http.createServer(function (request, response) {
4. response.writeHead(200, {"Content-Type": "text/plain"});
5. 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 :

|  |  |  |  |
| --- | --- | --- | --- |
| 1. ***$*** docker ps |  | | |
| 2. CONTAINER ID IMAGE | COMMAND | CREATED | STATUS |
| PORTS |  | NAMES |  |
| 3. 0e02e0844dd1 xataz/node:7 | "node app.js" | About a minute ago | Up About |
| a minute 0.0.0.0:8002->8000/tcp |  | node7 |  |
| 4. 7669bef4b5c0 xataz/node:6 | "node app.js" | About a minute ago | Up About |
| a minute 0.0.0.0:8001->8000/tcp |  | node6 |  |

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**](https://hub.docker.com/r/xataz/lutim/)). 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

1. Unable to find image 'xataz/lutim:latest' locally
2. latest: Pulling from xataz/lutim
3. c1c2612f6b1c: Already exists
4. 0e00ee3bbf34: Pull complete
5. 58fda08c5f8a: Pull complete
6. 1bb27614a217: Pull complete
7. 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 |
| PORTS |  | NAMES |  |  |  |
| 3. 766be7bdb450 xataz/lutim  minutes 0.0.0.0:8181->8181/t | | "/usr/local/bin/start"  cp lutim | | 7 minutes ago | Up 7 |

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**](#_bookmark0)[**une image**](#_bookmark0)). Dans notre cas nous avons ajouté quatre variables, mais il en existe d'autres (cf. [**README**](https://hub.docker.com/r/xataz/lutim/)) :

* 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](mailto:CONTACT%3Dcontact@domain.tld) ***MAX\_FILE\_SIZE***=10000000000 ***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.

5.

6.

7.

8.

9. }

listen 8080;

location / {

proxy\_pass http://toto:8000;

}

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,**](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.

4.

5.

6.

7.

8. }

listen 8080;

location / {

proxy\_pass http://tata:8000;

}

On relance nginx :

1. ***$*** docker restart reverse
2. reverse

Et on reteste la même URL ( [**http://XX.XX.XX.XX),**](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/](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**](https://docs.docker.com/engine/reference/builder/)

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.

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 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. [...]
9. Step 10 : CMD apache2ctl -D FOREGROUND
10. ---> Running **in** d435f9e2db87 12. ---> 1f9e7590d11b
11. Removing intermediate container d435f9e2db87
12. 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**](https://mondedie.fr/viewtopic.php?id=7313). 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.

1. ENV GID=991
2. ENV UID=991
3. ENV [CONTACT=contact@domain.tld](mailto:CONTACT%3Dcontact@domain.tld)
4. ENV WEBROOT=/

7. ENV SECRET=e7c0b28877f7479fe6711720475dcbbd

8. ENV MAX\_FILE\_SIZE=10000000000 9.

1. LABEL description="lutim based on debian" \
2. maintainer="xataz <https://github.com/xataz>" 12.
3. 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
4. RUN cpan install Carton
5. RUN cd / && git clone https://git.framasoft.org/luc/lutim.git
6. RUN cd /lutim && carton install 17.

18. VOLUME /lutim/files /data 19.

20. EXPOSE 8181 21.

1. COPY lutim.conf /lutim/lutim.conf
2. COPY startup /usr/bin/startup
3. 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* :

20.

21.

22.

23.

24. };

#broadcast\_message => 'Maintenance',

prefix db\_path

=> '<webroot>',

=> '/data/lutim.db',

delete\_no\_longer\_viewed\_files => 90

|  |  |  |  |
| --- | --- | --- | --- |
| 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', |

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.

1. sed -i -e 's|<secret>|'***${SECRET}***'|' \
2. -e 's|<contact>|'***${CONTACT}***'|' \
3. -e 's|<max\_file\_size>|'***${MAX\_FILE\_SIZE}***'|' \
4. -e 's|<webroot>|'***${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

1. CONTAINER ID PORTS
2. bb40fd7df491

IMAGE

COMMAND

CREATED

STATUS

NAMES

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**](http://dl-cdn.alpinelinux.org/alpine/latest-stable/main/x86_64/)), 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](mailto:.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](mailto: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)
11. (2/16) Installing busybox (1.24.2-r11)
12. Executing busybox-1.24.2-r11.post-install
13. (3/16) Installing alpine-baselayout (3.0.3-r0)
14. Executing alpine-baselayout-3.0.3-r0.pre-install
15. Executing alpine-baselayout-3.0.3-r0.post-install
16. (4/16) Installing openrc (0.21-r2)
17. Executing openrc-0.21-r2.post-install
18. (5/16) Installing alpine-conf (3.4.1-r2)
19. (6/16) Installing zlib (1.2.8-r2)
20. (7/16) Installing libcrypto1.0 (1.0.2h-r1)
21. (8/16) Installing libssl1.0 (1.0.2h-r1)
22. (9/16) Installing apk-tools (2.6.7-r0)
23. (10/16) Installing busybox-suid (1.24.2-r11)
24. (11/16) Installing busybox-initscripts (3.0-r3)
25. Executing busybox-initscripts-3.0-r3.post-install
26. (12/16) Installing scanelf (1.1.6-r0)
27. (13/16) Installing musl-utils (1.1.14-r11)
28. (14/16) Installing libc-utils (0.7-r0)
29. (15/16) Installing alpine-keys (1.1-r0)
30. (16/16) Installing alpine-base (3.4.3-r0)
31. Executing busybox-1.24.2-r11.trigger
32. 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

1. Removing intermediate container 385884a4f0c5
2. Successfully built fa8cc3e04a21
3. ***$*** docker container run -ti superalpine /bin/sh
4. / # 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"](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. ***$*** docker image build -t superalpine . 2. [...]   1. ***$*** docker container run -ti superalpine 2. / # 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"](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"](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.

1. RUN echo ["http://dl-cdn.alpinelinux.org/alpine/v3.4/main"](http://dl-cdn.alpinelinux.org/alpine/v3.4/main) > /,etc./apk/repositories && apk add

-U wget git

1. 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"](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\*"

1. REPOSITORY
2. superalpine2
3. superalpine1

TAG

latest latest

IMAGE ID 7808b8c75444

886e22de4865

:( CREATED

5 minutes ago

5 minutes ago

SIZE 4.813 MB

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.*