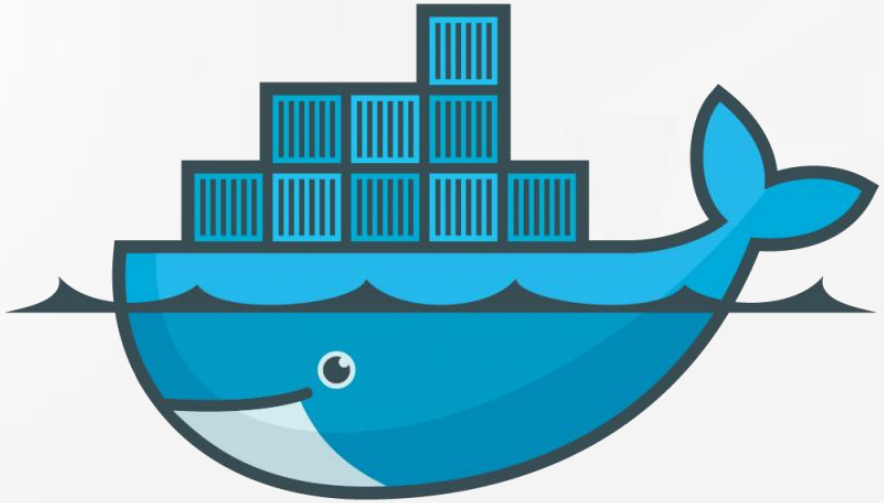


docker

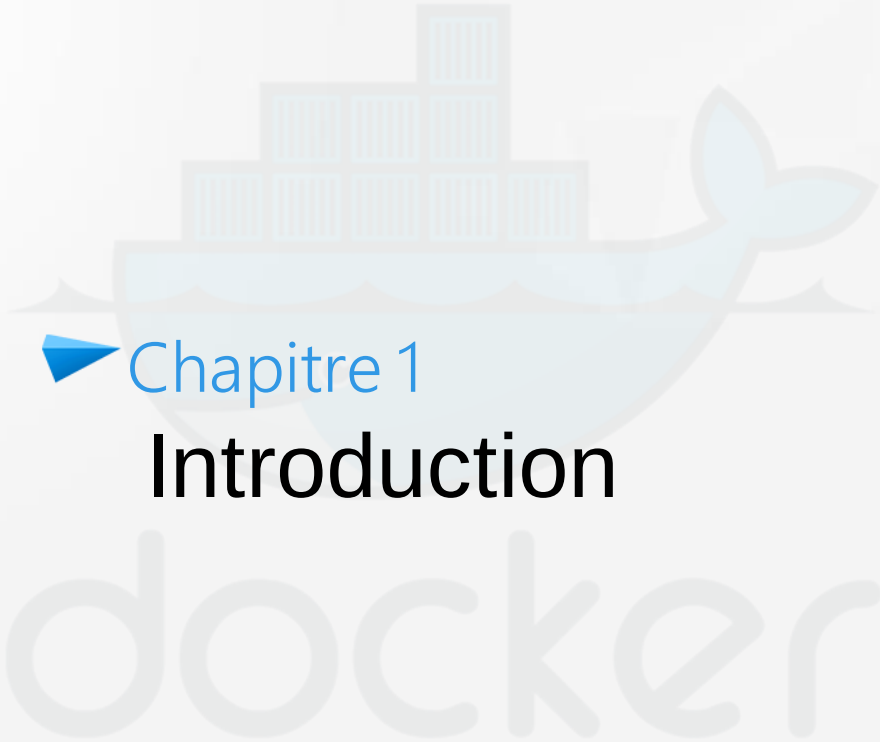
Sommaire

1. Introduction
2. Les différences entre la virtualisation et la conteneurisation
3. C'est quoi exactement un conteneur ?
4. Découverte et installation de Docker
5. Fonctionnement et manipulation des images Docker
6. Fonctionnement et manipulation des conteneurs Docker
7. Créer ses propres images Docker avec le Dockerfile
8. Fonctionnement et manipulation des volumes dans Docker
9. Gérez vos conteneurs avec le Docker Compose
10. Fonctionnement et manipulation du réseau dans Docker

SOLYTI

Chapitre 1

Introduction



- Docker est une plateforme lancé en mars 2013 permettant aux développeurs et aux administrateurs système de développer, déployer et exécuter des applications avec des conteneurs.

- Plus précisément la plateforme permet d'embarquer une application avec toutes ses dépendances dans un processus isolé (nommé conteneur) qui peut être ensuite exécutée sur n'importe quelle machine avec n'importe quel système d'exploitation compatible avec le moteur Docker.

- Docker a été fondée en France par un diplômé de l'école d'Epitech nommé Solomon Hykes. Par manque d'investissement en France, l'entreprise a souhaité évoluer dans la Silicon Valley, où elle a pu enchaîner des levées de fonds spectaculaires, illustrant ainsi le potentiel attendu de cette technologie.

Chapitre 2

Les différences entre la virtualisation et la conteneurisation

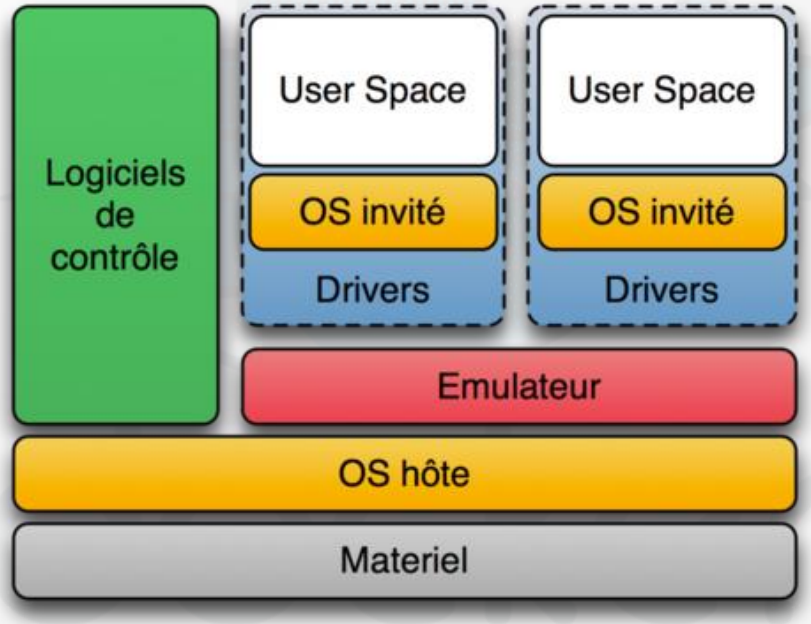
SOLYTI

- Le fonctionnement de la virtualisation reste assez simple, c'est qu'au lieu d'avoir un serveur avec un système d'exploitation faisant tourner une ou plusieurs application(s), on préférera mutualiser plusieurs serveurs virtuels depuis un serveur physique grâce à un logiciel nommé l'hyperviseur.

- L’hyperviseur permet d’émuler intégralement les différentes ressources matérielles d'un serveur physique (tels que l'unité centrale, le CPU, la RAM, le disque dur, carte réseau etc ...), et permet à des machines virtuelles de les partager.

- Les machines virtuelles nommées aussi VM (Virtual Machine) bénéficieront de ressources matérielles selon leurs besoins
- L'avantage c'est qu'il est possible de modifier les ressources physiques de ces VMs en quelques clics. De plus elles possèdent leur propre système d’exploitation ainsi que leurs propres applications.

Les différences entre la virtualisation et la conteneurisation

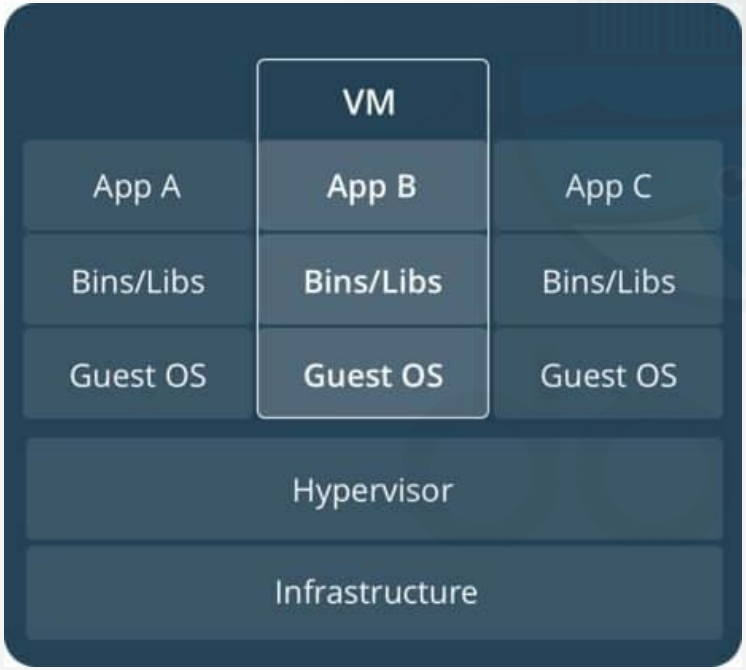


L’isolation :

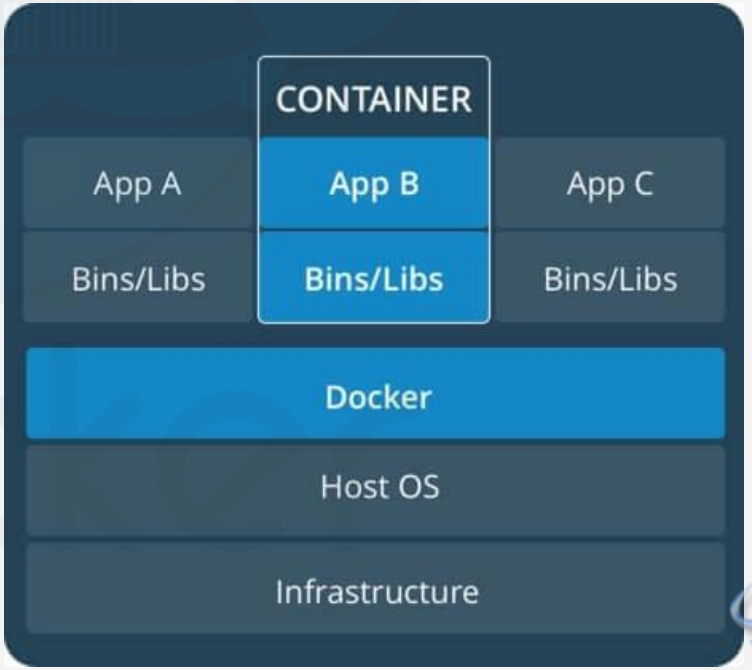
- Dans le cas de la virtualisation l’isolation des VMs se fait au niveau matérielles (CPU/RAM/Disque) avec un accès virtuel aux ressources de l'hôte via un hyperviseur. De plus, généralement les ordinateurs virtuels fournissent un environnement avec plus de ressources que la plupart des applications n'en ont besoin.

- Dans le cas de la conteneurisation, l'isolation se fait au niveau du système d'exploitation. Un conteneur va s'exécuter sous Linux de manière native et va partager le noyau de la machine hôte avec d'autres conteneurs. ne prenant pas plus de mémoire que tout autre exécutable, ce qui le rend léger.

Virtualisation



Conteneurisation



SOLYTI

Avantages de la conteneurisation par rapport à la virtualisation :

- Comme vu plus haut les machines virtuelles intègrent elles-mêmes un OS pouvant aller jusqu'à des Giga-octets. Ce n'est pas le cas du conteneur. Le conteneur appel directement l'OS pour réaliser ses appels système et exécuter ses applications. Il est beaucoup moins gourmand en ressources

Avantages de la conteneurisation par rapport à la virtualisation :

- Comme vu plus haut les machines virtuelles intègrent elles-mêmes un OS pouvant aller jusqu'à des Giga-octets. Ce n'est pas le cas du conteneur. Le conteneur appel directement l'OS pour réaliser ses appels système et exécuter ses applications. Il est beaucoup moins gourmand en ressources

- Le déploiement est un des points clés à prendre en compte de nos jours. On peut déplacer les conteneurs d'un environnement à l'autre très rapidement (en réalité c'est encore plus simple et rapide avec Docker, car il suffit juste de partager des fichiers de config qui sont en général très légers).

- On peut bien sur faire la même chose pour une machine virtuelle en la déplaçant entièrement de serveurs en serveurs mais n'oubliez pas qu'il éxiste cette couche d'OS qui rendra le déploiement beaucoup plus lent, sans oublier le processus d'émulation de vos ressources physiques, qui lui-même demandera un certain temps d'exécution et donc de la latence en plus.

Récapitulation des avantages de la conteneurisation :

- **Flexible**: même les applications les plus complexes peuvent être conteneurisées.
- **Léger**: les conteneurs exploitent et partagent le noyau hôte.
- **Interchangeable**: vous pouvez déployer des mises à jour à la volée

Récapitulation des avantages de la conteneurisation :

- **Portable**: vous pouvez créer localement, déployer sur le cloud et exécuter n'importe où votre application.
- **Évolutif**: vous pouvez augmenter et distribuer automatiquement les réplicas (les clones) de conteneur.
- **Empilable**: Vous pouvez empiler des services verticalement et à la volée.

Chapitre 3

C’est quoi exactement un conteneur ?

- Avant toute chose, il faut savoir que le noyau Linux offre quelques fonctionnalités comme les namespaces (ce qu'un processus peut voir) et les cgroups (ce qu'un processus peut utiliser en terme de ressources), qui vont vous permettre d'isoler les processus Linux les uns des autres. Lorsque vous utilisez ces fonctionnalités, on appelle cela des conteneurs.

- Prenons un exemple simple, si jamais on souhaite créer un conteneur contenant la distribution Ubuntu. Fondamentalement, ces fonctionnalités d'isolation proposées par le noyau Linux, vont vous permettent de prétendre d'avoir quelque chose qui ressemble à une machine virtuelle avec l'OS Ubuntu, sauf qu'en réalité ce n'est pas du tout une machine virtuelle mais un processus isolé s'exécutant dans le même noyau Linux .

Chapitre 4

Découverte et installation de Docker

SOLYTI

Docker est disponible en deux éditions:

- Docker Community Edition (CE)
- Docker Enterprise Edition (EE)

- Docker Community Edition (CE) est idéale pour les développeurs individuels et les petites équipes cherchant à se familiariser avec Docker et à expérimenter des applications basées sur des conteneurs. De plus cette version est gratuite. Ça sera la version que nous utiliserons.

- Docker Enterprise Edition (EE) est conçue pour les équipes de développement d'entreprise et les équipes système qui créent, expédient et exécutent des applications critiques pour la production à grande échelle (Elle n'est pas gratuite).

Initialement, Docker était exclusivement utilisé sur les distributions Linux, la version actuelle du moteur de conteneur (Container-Engine) est largement indépendante de la plateforme. Des packages d'installation sont disponibles pour Microsoft Windows et macOs ainsi que les services Cloud comme Amazon Web Services (AWS) et Microsoft Azure.

Les distributions Linux prises en charge incluent :

- CentOS
- Debian
- Fedora
- Oracle Linux
- Red Hat Enterprise Linux
- Ubuntu
- openSUSE
- SUSE Linux Enterprise

Il y a aussi des distributions de Dockers maintenues par les communautés pour :

- Arch Linux
- CRUX Linux
- Gentoo Linux

Il existe trois façons différentes d'installer la plateforme de conteneurs Docker sur votre système Ubuntu en fonction des conditions générales et préalables :

- Installation manuelle via le package DEB
- Installation à partir du dépôt Docker
- Installation aus dem Ubuntu-Repository

En principe, Docker peut être téléchargé sous forme de paquet DEB et être installé manuellement. Le paquet d'installation requis est disponible à l'adresse URL suivante :

- *https://download.docker.com/linux/ubuntu/dists/*

Choisir la version de ubuntu puis allez dans pool/stable, ensuite choisissez l'architecture du processeur et téléchargé le paquet .deb

Téléchargez le fichier DEB de la version Ubuntu souhaitée et lancez le processus d'installation avec cette commande via le terminal Ubuntu :

```
$ sudo dpkg -i /path/to/package.deb
```

Les utilisateurs qui ne souhaitent pas utiliser le repository Docker ont également la possibilité de charger la plate-forme de conteneurs à partir du repository du système d'exploitation.

Utilisez la ligne de commande suivante pour installer un paquet Docker fourni par la communauté Ubuntu :

```
$ sudo apt-get install -y docker.io
```

Pour installer Docker depuis le depot officiel suivez ce tuto :

https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04-fr

Par défaut on ne peut lancer les commandes docker qu'avec l'utilisateur root et les autres utilisateurs ne peuvent y accéder qu'en utilisant le sudo.

Pour autoriser mon compte utilisateur Unix à lancer les commandes docker sans passer par le sudo il faut ajouter son utilisateur au groupe docker :

```
$ sudo usermod -aG docker $USER
```

Voici la configuration requise pour une installation windows native :

- Windows 10 64 bits: Pro, Entreprise ou Education (version 15063 ou ultérieure).
- La virtualisation est activée dans le BIOS (normalement elle est activée par défaut).
- Au moins 4 Go de RAM

Rendez-vous sur cette page web pour télécharger docker desktop pour windows :

https://hub.docker.com/editions/community/docker-ce-desktop-windows/

Une fois docker installé, nous allons lancer notre premier conteneur :

```
$ docker run hello-world
```

```
vagrant@debian10:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:37a0b92b08d4919615c3ee023f7ddb068d12b8387475d64c622ac30f45c29c51
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Chapitre 5

Fonctionnement et manipulation des images Docker

SOLYTI

Qu'est qu'une image Docker ?

Sur Docker, un conteneur est lancé en exécutant une image.

"Attends mais c'est quoi une image Docker" ?

Une image est un package qui inclut tout ce qui est nécessaire à l'exécution d'une application, à savoir :

- Le code
- L'exécution
- Les variables d'environnement
- Les bibliothèques
- Les fichiers de configuration

Une image docker est créée à partir d'un fichier nommé le Dockerfile.

Une image est un modèle composé de plusieurs couches, ces couches contiennent notre application ainsi que les fichiers binaires et les bibliothèques requises.

Lorsqu'une image est instanciée, son nom est un conteneur, un conteneur est donc une image en cours d'exécution.

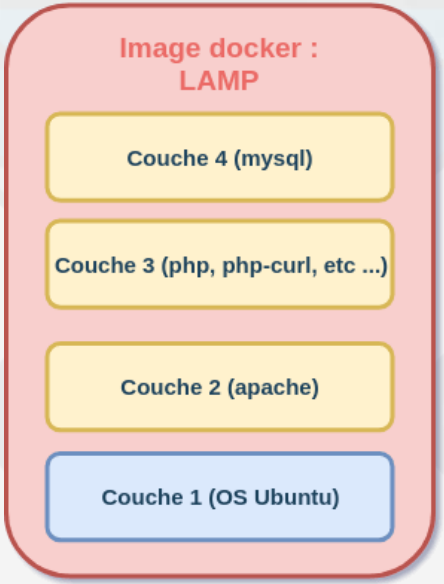
Pour mieux comprendre le système de couche, imaginons par exemple qu'on souhaite déployer notre application web dans un serveur LAMP (Linux Apache MySQL PHP) au moyen de Docker.

Pour créer notre stack (pile en français), nous aurons besoin de :

- Une couche OS pour exécuter notre Apache, MySQL et Php
- Une couche Apache pour démarrer notre serveur web et pourquoi pas la config qui va avec (.htaccess, apache2.conf, site-available/, etc ...)

- Une couche php qui contiendra un interpréteur Php mais aussi les bibliothèques qui vont avec (exemple : php-curl)
- Une couche Mysql qui contiendra notre système de gestion de bases de données Mysql

Au total, notre image docker sera composée de quatre couches, en schéma ceci nous donnerai :



Il est important de bien différencier une image Docker d'un conteneur Docker car ce sont deux choses distinctes.

Une image sert de base à la création d'un conteneur.

Un conteneur est le résultat de l'exécution de cette image.

Voici la commande qui permet de répertorier les images Docker téléchargées sur votre ordinateur :

```
$ docker image ls
```

Ou

```
$ docker images
```

Exemple de résultat :

```
vagrant@debian10:~$ docker images
REPOSITORY                                    TAG                        IMAGE ID       CREATED          SIZE
telegraf                                      latest                     e36bdd348ca3   22 hours ago     334MB
influxdb                                      latest                     3d28107b0899   25 hours ago     342MB
influxdb                                      1.8                        5c8e0adc1031   25 hours ago     287MB
grafana/grafana                               latest                     092a480a2531   2 days ago       249MB
atlassian/bitbucket                           latest                     b46046edf770   4 days ago       1.2GB
jenkins/jenkins                               lts-jdk11                  e4ebf98bd6ca   7 days ago       441MB
hello-world                                   latest                     feb5d9fea6a5   2 weeks ago      13.3kB
elleflorio/svn-server                         latest                     b8cc2d411214   5 months ago     49.7MB
atlassian/bamboo-server                       7.0.4                      12dbdf08d2c9   17 months ago    941MB
docker.elastic.co/logstash/logstash           7.4.2                      642b82780655   23 months ago    889MB
elk-cec-docker_logstash                       latest                     642b82780655   23 months ago    889MB
docker.elastic.co/kibana/kibana               7.4.2                      230d3ded1abc   23 months ago    1.1GB
elk-cec-docker_kibana                         latest                     230d3ded1abc   23 months ago    1.1GB
docker.elastic.co/elasticsearch/elasticsearch 7.4.2                      b1179d41a7b4   23 months ago    855MB
elk-cec-docker_elasticsearch                  latest                     b1179d41a7b4   23 months ago    855MB
mesosphere/mesos-slave                        0.28.0-2.0.16.ubuntu1404   b25ca94d42f4   5 years ago      513MB
mesosphere/mesos-master                       0.28.0-2.0.16.ubuntu1404   84e10630bd66   5 years ago      513MB
netflixoss/exhibitor                          1.5.2                      07572922ce64   6 years ago      705MB
```

Maintenant si on souhaite supprimer une image Docker, on aura besoin soit de son ID soit de son nom.

Une fois qu'on aura récupéré ces informations, on peut passer à la suite en lançant la commande suivante :

```
$ docker rmi <IMAGE ID>
```

Ou

```
$ docker rmi <REPOSITORY>
```

Exemple :

Je lance ma commande pour supprimer l'image hello-world. Je dois donc récupérer les infos de cette image :

```
hello-world                    latest              feb5d9fea6a5        2 weeks ago         13.3kB
```

Je peux alors lancer ma commande :

```
$ docker rmi hello-world
```

Ou

```
$ docker rmi feb5d9fea6a5
```

Le resultat est donc le suivant :

```
vagrant@debian10:~$ docker rmi hello-world
Error response from daemon: conflict: unable to remove repository reference "hello-world" (must force) - container
12eca3c17ecc is using its referenced image feb5d9fea6a5
```

Ce message d'erreur nous explique, qu'on ne peut pas supprimer notre image Docker car des conteneurs ont été instanciés depuis notre image "hello-world". En gros si on jamais on supprime notre image "hello-world", ça va aussi supprimer nos conteneurs car ils se basent sur cette image.

Dans notre cas ça ne pose aucun problème, d'ailleurs pour résoudre ce problème l'erreur nous informe qu'on doit forcer la suppression pour éliminer aussi les conteneurs liés à notre image (must force).

Pour forcer la supprimer on va utiliser l'option --force ou -f.

```
vagrant@debian10:~$ docker rmi -f hello-world
Untagged: hello-world:latest
Untagged: hello-world@sha256:37a0b92b08d4919615c3ee023f7ddb068d12b8387475d64c622ac30f45c29c51
Deleted: sha256:feb5d9fea6a5e9606aa995e879d862b825965ba48de054caab5ef356dc6b3412
```

Bonus :

Voici une commande qui permet de supprimer toutes les images disponibles sur notre machine :

```
$ docker rmi -f $(docker images -q)
```

Télécharger une image depuis le Docker Hub Registry

Maintenant rentrons dans du concret et téléchargeons une image plus utile comme par exemple l'image officielle d'Ubuntu. "Oui mais ou est-ce que je peux retrouver la liste des images disponibles ?"

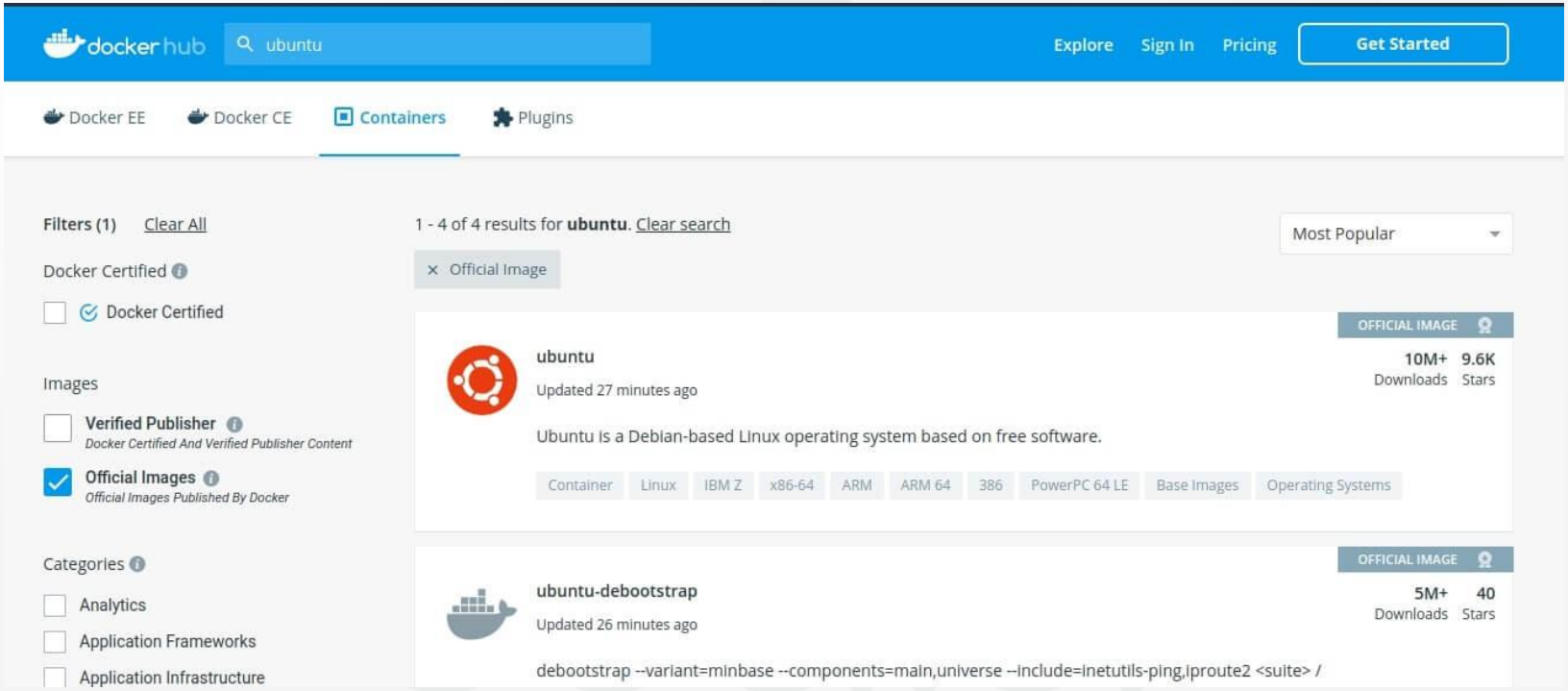
Rendons nous sur le Docker hub registry :

https://hub.docker.com/

Pour faire simple un Registry (registre en français) est une application côté serveur qui permet de stocker et de distribuer des images Docker, le Docker Hub Registry est le registre officiel de Docker.

Il existe deux façons pour voir si une image est disponible dans le Docker Hub Registry, la première consiste à visiter le site web et vous recherchez directement le nom de l'image dans la barre de recherche :

Fonctionnement et manipulation des images Docker



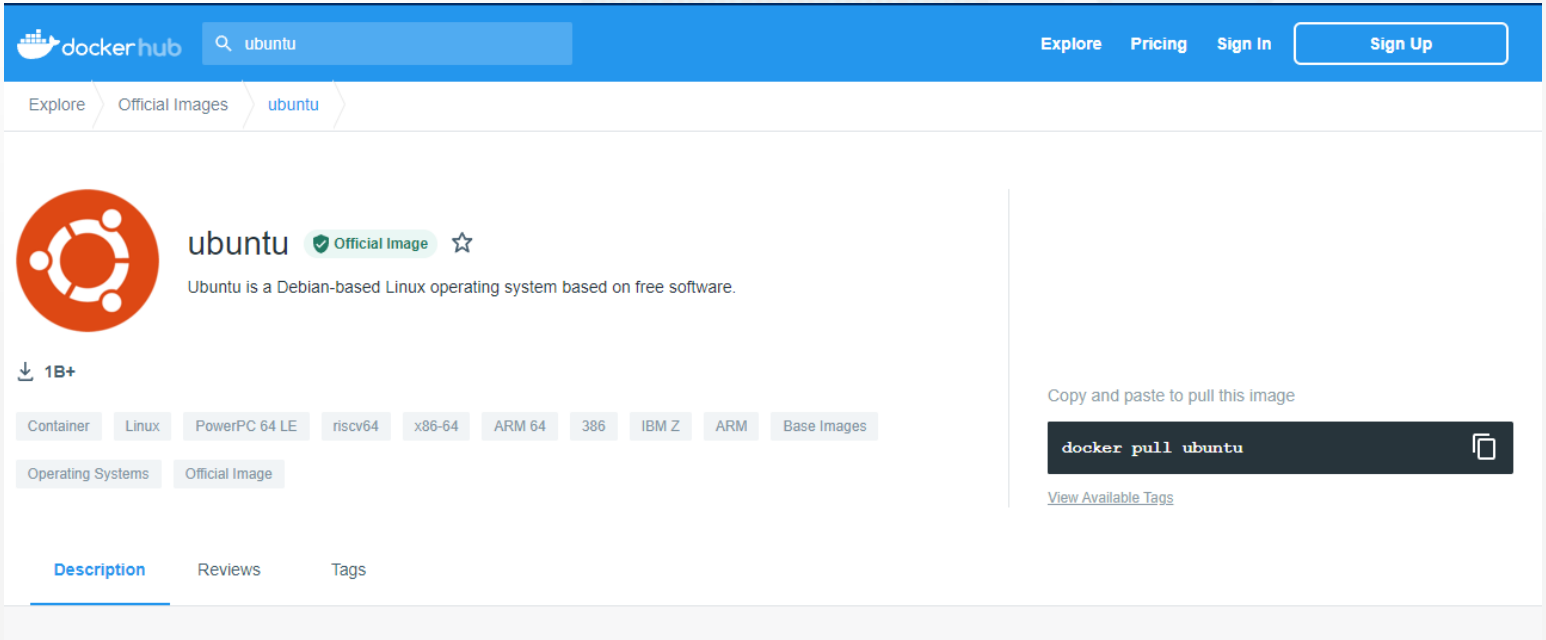
SOLYTI

Vous remarquerez sur l'image que j'ai coché la case "Official Image" pour ne m'afficher que les images officielles. Ainsi je m'assure que l'image Ubuntu que je vais télécharger a bien été crée par l'équipe gérant la distribution Ubuntu.

Il faut bien faire attention aux images qu'on télécharge sur le net.

Il faut toujours vérifier au préalable le code source de l'image plus précisément le fichier Dockerfile (ne vous inquiétez on verra le Dockerfile dans un prochain chapitre), car on est jamais à l'abri d'avoir des images contenant des programmes vulnérables voire même des images malhonnêtes.

Si je clique sur l'image officielle d'ubuntu je tombe sur la page suivante :



Plus bas on retrouve, un menu de navigation contenant :

- **DESCRIPTION** : Description de l'image, souvent on retrouve quelques tags, la configuration de votre conteneur (par exemple la config de votre base de données pour une image basé sur du mysql) et les liens github vers les sources du projet.
- **REVIEWS** : l'avis des utilisateurs
- **TAGS** : les différents tags disponible pour cette image

La deuxième manière pour lister les images disponibles dans le Docker hub Registry, c'est de passer par la ligne de commande :

```
$ docker search ubuntu
```

```
vagrant@debian10:~$ docker search ubuntu
NAME                                                      DESCRIPTION                                     STARS
          OFFICIAL            AUTOMATED
ubuntu                                                    Ubuntu is a Debian-based Linux operating sys…   12940
          [OK]
dorowu/ubuntu-desktop-lxde-vnc                            Docker image to provide HTML5 VNC interface …   578
                              [OK]
websphere-liberty                                         WebSphere Liberty multi-architecture images …   280
          [OK]
rastasheep/ubuntu-sshd                                    Dockerized SSH service, built on top of offi…   255
                              [OK]
consol/ubuntu-xfce-vnc                                    Ubuntu container with "headless" VNC session…   243
                              [OK]
ubuntu-upstart                                            DEPRECATED, as is Upstart (find other proces…   113
          [OK]
neurodebian                                               NeuroDebian provides neuroscience research s…   86
          [OK]
1and1internet/ubuntu-16-nginx-php-phpmyadmin-mysql-5      ubuntu-16-nginx-php-phpmyadmin-mysql-5          50
                              [OK]
open-liberty                                              Open Liberty multi-architecture images based…   48
          [OK]
ubuntu-debootstrap                                        DEPRECATED; use "ubuntu" instead                44
          [OK]
i386/ubuntu                                               Ubuntu is a Debian-based Linux operating sys…   25

solita/ubuntu-systemd                                     Ubuntu + systemd                                24
                              [OK]
1and1internet/ubuntu-16-apache-php-5.6                    ubuntu-16-apache-php-5.6                        14
                              [OK]
1and1internet/ubuntu-16-apache-php-7.0                    ubuntu-16-apache-php-7.0                        13
```

SOLYTI

Si vous souhaitez n'afficher que les images officielles, il est possible de filtrer le résultat avec la commande suivante :

```
$ docker search --filter "is-official=true" ubuntu
```

Vous l'aurez deviné, pour télécharger une image depuis le Docker hub Registry il faut utiliser la commande suivante (**précisez le tag si** vous souhaitez un tag différent de latest) :

```
$ docker pull ubuntu
```

Pour télécharger une image ubuntu avec un autre tag différent de latest par exemple le tag 16.04:

```
$ docker pull ubuntu:16.04
```

Recap :

Copier

```
## Afficher de l'aide
docker help
docker <sous-commande> --help

## Afficher des informations sur l'installation de Docker
docker --version
docker version
docker info

## Executer une image Docker
docker run hello-world

## Lister des images Docker
docker image ls
# ou
docker images

## Supprimer une image Docker
docker images rmi <IMAGE_ID ou IMAGE_NAME>  # si c'est le nom de l'image qui est spécifié alors il prendra par déf
    -f ou --force : forcer la suppression

## Supprimer tous les images Docker
docker rmi -f $(docker images -q)

## Rechercher une image depuis le Docker hub Registry
docker search ubuntu
    --filter "is-official=true" : Afficher que les images officielles

## Télécharger une image depuis le Docker hub Registry
docker pull <IMAGE_NAME>  # prendra par défaut le tag latest
docker pull ubuntu:16.04 # prendra le tag 16.04
```

SOLYTI

Chapitre 6

Fonctionnement et manipulation des conteneurs Docker

SOLYTI

Différence entre image et conteneur :

Pour rappel lorsque vous utilisez des fonctionnalités permettant une isolation du processus (comme les namespaces et les cgroups), on appelle cela des conteneurs.

Dans ces conteneurs on peut retrouver généralement un ou plusieurs programme(s) avec toute leurs dépendances de manière à les maintenir isolées du système hôte sur lequel elles s'exécutent.

Nous avons aussi vu que sur docker un conteneur est une instance d'exécution d'une image.

Plus précisément un conteneur est ce que l'image devient en mémoire lorsqu'elle est exécutée (c'est-à-dire une image avec un état ou un processus utilisateur).

Pour créer une instance de notre image, ou autrement dit créer un conteneur, alors nous utiliserons une commande que nous avions déjà vue sur les chapitres précédents, soit :

```
$ docker run [OPTIONS] <IMAGE_NAME ou ID>
```

Avouons-le, l'image hello-world n'est pas vraiment utile, car elle n'est prévue que pour afficher un petit message d'information et en plus de ça elle se ferme directement après.

Téléchargeons une image plus utile, comme par exemple l'image Ubuntu, et pourquoi pas la manipuler avec l'interpréteur de commandes bash et de télécharger dessus des outils comme l'outil git.

Étape 1 : Téléchargement de l'image ubuntu :

```
$ docker pull ubuntu
```

Étape 2 : Instanciation de l'image ubuntu :

```
$ docker run ubuntu
```

"Ah mais attend mais moi quand je lance mon image Ubuntu, mon conteneur se quitte directement après, est-ce que les commandes sont bonnes ? !«

La réponse est oui, vos commandes sont bien les bonnes mais ce n'est pas le cas pour vos options, car oui cette commande ’docker run’ possède beaucoup d'options consultables soit depuis la doc officielle, soit depuis commande ’docker run –help’

Comme vous pouvez le constater il existe beaucoup d'options, mais rassurez-vous, car vous n'avez pas besoin de tous les connaître, voici pour le moment les options qui nous intéressent pour ce chapitre :

- -t : Allouer un pseudo TTY (terminal virtuel)
- -i : Garder un STDIN ouvert (l'entrée standard plus précisément l'entrée clavier)
- -d : Exécuter le conteneur en arrière-plan et afficher l'ID du conteneur

Dans notre cas nous avons besoin d'une Tty (option -t) et du mode interactif (option -i) pour interagir avec notre conteneur basé sur l'image Ubuntu. Tentons alors l'exécution de ces deux options :

```
$ docker run –it ubuntu
```

Vous êtes maintenant à l'intérieur de votre conteneur ubuntu avec un jolie shell

Profitons-en pour télécharger l'outil git :

$ apt update –y && apt install –y git

Maintenant, si j'essaie de vérifier mon installation git alors vous constaterez que je n'ai aucune erreur :

```
root@9281506cc74a:/# git --version
git version 2.25.1
```

Je profite un peu de cette partie pour vous montrer la puissance des conteneurs. Je vais détruire mon conteneur Ubuntu avec la commande ’rm -rf /*’ , **ASSUREZ-VOUS BIEN AVANT QUE VOUS ETES DANS UN CONTENEUR**.

Après avoir lancé cette commande je peux vous affirmer que j'ai bien détruit mon conteneur, preuve à l'appui je ne peux même plus lancer la commande ls :

```
root@9281506cc74a:/# ls
bash: ls: command not found
```

SOLYTI

Je vais donc quitter mon conteneur avec la commande ’exit’ ou ctrl+d pour ensuite relancer un nouveau conteneur ubuntu :

```
$ docker run –it ubuntu
```

J’ai donc crée un nouveau conteneur et ma commande ls est de nouveau disponible, en revanche la commande ‘git –version’ n’est plus disponible car git n’est pas installer dans le nouveau conteneur.

Pourquoi ? Car les données et les fichiers dans un conteneur sont éphémères !

Il existe deux façons pour stocker les données d'un conteneur, soit on transforme notre conteneur en image soit on utilise le système de volume, nous verrons cette notion dans un autre chapitre dédié aux volumes.

Il y a trois autres options utiles de la commande ’docker run’ :

- --name : Attribuer un nom au conteneur
- --expose : Exposer un port ou une plage de ports (on demande au firewall du conteneur de nous ouvrir un port ou une plage de port)
- -p ou --publish : Mapper un port déjà exposé, plus simplement ça permet de faire une redirection de port

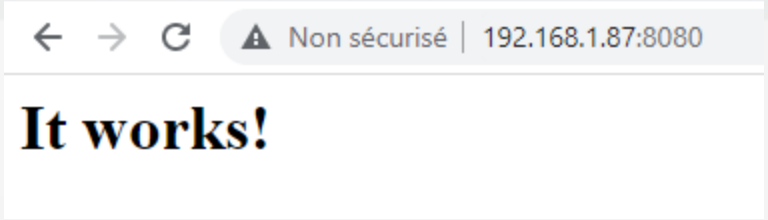
Nous allons maintenant vous montrer l'utilité de l'option ’-d’ de la commande docker run.

Pour mieux comprendre cette option je vais utiliser l'image officielle d’Apache.

Par défaut l'image apache expose déjà le port 80 donc pas besoin de lancer l'option --expose, pour notre exemple nous allons mapper le port 80 vers le 8080 et nommer notre conteneur en "monServeurWeb", ce qui nous donnera comme commande :

```
$ docker run --name monServeurWeb -p 8080:80 httpd
```

Une fois votre image executée, visitez http://localhost:8080 et vous verrez la phrase "It works"!



Seulement le seul souci, c'est que sur notre terminal, on aperçoit les logs d'Apache qui tournent en boucle :

```
vagrant@debian10:~$ docker run --name monServeurWeb -p 8080:80 httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
b380bbd43752: Pull complete
d0c6942edac3: Pull complete
d027638c026c: Pull complete
17082a2122d5: Pull complete
0b7d42e498cc: Pull complete
Digest: sha256:f03a63735d3653045a3b1f5490367415e9534d8abfe0b21252454dc85ca09800
Status: Downloaded newer image for httpd:latest
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'S
erverName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'S
erverName' directive globally to suppress this message
[Thu Oct 14 11:38:23.593846 2021] [mpm_event:notice] [pid 1:tid 140219829064832] AH00489: Apache/2.4.51 (Unix) conf
igured -- resuming normal operations
[Thu Oct 14 11:38:23.593985 2021] [core:notice] [pid 1:tid 140219829064832] AH00094: Command line: 'httpd -D FOREGR
OUND'
192.168.1.185 - - [14/Oct/2021:11:40:16 +0000] "GET / HTTP/1.1" 200 45
192.168.1.185 - - [14/Oct/2021:11:41:09 +0000] "-" 408 -
```

Ce qui serait intéressant c'est de laisser notre conteneur tourner en arrière-plan avec l'option -d :

```
$ docker run --name monServeurWeb -d -p 8080:80 httpd
```

Cependant, si vous relancez la commande, vous obtenez l'erreur suivante :

```
vagrant@debian10:~$ docker run --name monServeurWeb -d -p 8080:80 httpd
docker: Error response from daemon: Conflict. The container name "/monServeurWeb" is already in use by container "b
cea7568a40865fe1ad77fc38f162b3f068741a8ee78438373c9cd643eeea1d5". You have to remove (or rename) that container to
be able to reuse that name.
See 'docker run --help'.
```

Pour faire simple, il nous indique qu'il n'est pas possible de démarrer deux conteneurs sous le même nom

Dans ce cas vous avez deux solutions, soit vous choisissez la facilité en renommant votre conteneur.

Soit la deuxième solution, qui est de supprimer mon conteneur, et d'en recréer un nouveau avec le bon nom.

Ça tombe bien car c'est l'occasion de vous montrer d'autres commandes utiles pour manipuler les conteneurs Docker.

Pour supprimer notre conteneur, il faut d'abord l'identifier et par la suite récolter soit son id, soit son nom.

Ça sera l'occasion de vous dévoiler une commande que vous utiliserez beaucoup, cette commande vous permettra de lister les conteneurs disponibles sur votre machine :

```
$ docker ps
```

Voici l'explication des différentes colonnes :

- **CONTAINER ID** : id du conteneur
- **IMAGE** : L'image sur laquelle c'est basé le conteneur
- **COMMAND** : Dernière commande lancée lors de l'exécution de votre image (ici la commande httpd-foreground permet de lancer le service apache en premier plan)
- **CREATED** : date de création de votre conteneur

- **STATUS** : statut de votre conteneur, voici une liste des différents états d'un conteneur :
  - created : conteneur créé mais non démarré (cet état est possible avec la commande docker create)
  - restarting : conteneur en cours de redémarrage
  - running : conteneur en cours d'exécution
  - paused : conteneur stoppé manuellement (cet état est possible avec la commande docker pause)

Par défaut la commande ’docker ps’ ne vous affichera que les conteneurs en état running, pour pouvoir afficher les conteneurs peu importe leur état, alors il faut utiliser l'option -a ou --all :

```
$ docker ps -a
```

Maintenant que nous avons pu récupérer l'id ou le nom du conteneur, on est capable de supprimer notre conteneur avec la commande suivante :

```
$ docker rm <CONTAINER NAME ou ID>
```

Donc :

```
$ docker rm monServeurWeb
```

Une fois mon conteneur supprimé, je peux enfin créer mon conteneur Apache avec l'option ’-d’ :

```
$ docker run --name monServeurWeb -d -p 8080:80 httpd
```

Une fois mon conteneur supprimé, je peux enfin créer mon conteneur Apache avec l'option ’-d’ :

```
$ docker run --name monServeurWeb -d -p 8080:80 httpd
```

Vous ne voyez plus les logs d'apache et votre conteneur tourne en arrière-plan

Une fois mon conteneur supprimé, je peux enfin créer mon conteneur Apache avec l'option ’-d’ :

$ docker run --name monServeurWeb -d -p 8080:80 httpd

```
vagrant@debian10:~$ docker run --name monServeurWeb -d -p 8080:80 httpd
0ed069f3888b586ff8ff7bedd22c99bd1b689f4b4649610e3767086496a23691
```

Vous ne voyez plus les logs d'apache et votre conteneur tourne en arrière-plan

Il existe une commande ’docker exec’ qui permet de lancer n'importe quelle commande dans un conteneur déjà en cours d'exécution.

Nous allons l'utilisée pour récupérer notre interpréteur de commande /bin/bash, ce qui aura pour but de se connecter directement à notre conteneur Apache :

```
$ docker exec –it monServeurWeb /bin/bash
```

Vous êtes maintenant dans votre conteneur. Pour s'amuser un peu, on va changer le message de la page d'accueil :

```
$ echo "<h1>Docker c'est vraiment cool</h1>" > /usr/local/apache2/htdocs/index.html
```

Revisitez la page, http://localhost:8080 et vous verrez votre nouveau message.

SOLYTI

Voici la commande qui permet de transformer un conteneur en image, afin retrouver l'état du conteneur au moment de la transformation :

$ docker commit <CONTAINER NAME or ID> <NEW IMAGENAME>

Cette commande est peu utilisée, on préfèrera utiliser un Dockerfile pour construire son propre conteneur.

Recap :

```
## Exécuter une image Docker
docker run <CONTAINER_ID ou CONTAINER_NAME>
    -t ou --tty : Allouer un pseudo TTY
    --interactive ou -i : Garder un STDIN ouvert
    --detach ou -d : Exécuter le conteneur en arrière-plan
    --name : Attribuer un nom au conteneur
    --expose: Exposer un port ou une plage de ports
    -p ou --publish : Mapper un port  "<PORT_CIBLE:PORT_SOURCE>"
    --rm : Supprimer automatiquement le conteneur quand on le quitte

## Lister des conteneurs en état running Docker
docker container ls
# ou
docker ps
    -a ou --all : Afficher tous les conteneurs peut-importe leur état

## Supprimer un conteneur Docker
docker rm <CONTAINER_ID ou CONTAINER_NAME>
    -f ou --force : forcer la suppression

## Supprimer tous les conteneurs Docker
docker rm -f $(docker ps -aq)
```

Recap :

```
## Exécuter une commande dans un conteneur Docker
docker exec <CONTAINER_ID ou CONTAINER_NAME> <COMMAND_NAME>
    -t ou --tty : Allouer un pseudo TTY
    -i ou --interactive : Garder un STDIN ouvert
    -d ou --detach : lancer la commande en arrière plan

## sorties/erreurs d'un conteneur
docker logs <CONTAINER_ID ou CONTAINER_NAME>
    -f : suivre en permanence les logs du conteneur
    -t : afficher la date et l'heure de la réception de la ligne de log
    --tail <NOMBRE DE LIGNE> = nombre de lignes à afficher à partir de la fin (par défaut "all")


## Transformer un conteneur en image
docker commit <CONTAINER_NAME ou CONTAINER_ID> <NEW IMAGENAME>
    -a ou --author <string> : Nom de l'auteur (ex "John Hannibal Smith <hannibal@a-team.com>")
    -m ou --message <string> : Message du commit
```

Chapitre 7

Créer ses propres images Docker avec le Dockerfile

SOLYTI

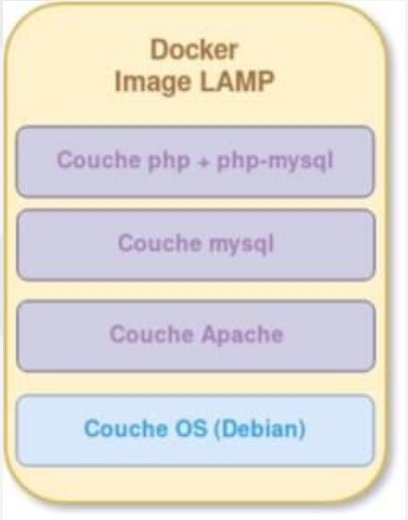
Il est temps de créer vos propres images Docker à l'aide du fichier Dockerfile.

Petit rappel, une image est un modèle composé de plusieurs couches, ces couches contiennent notre application ainsi que les fichiers binaires et les bibliothèques requises.

Pour s'exercer, nous allons créer notre propre stack LAMP (Linux Apache MySQL PHP) au moyen de Docker. Voici les différentes couches de cette image :

- Une couche OS pour exécuter notre Apache, MySQL et Php, je vais me baser sur la distribution Debian.
- Une couche Apache pour démarrer notre serveur web.

- Une couche php qui contiendra un interpréteur Php mais aussi les bibliothèques qui vont avec.
- Une couche Mysql qui contiendra notre système de gestion de bases de données.



Avant de créer notre propre image, je vais d'abord vous décrire les instructions Dockerfile les plus communément utilisées

- FROM : Définit l'image de base qui sera utilisée par les instructions suivantes.
- LABEL : Ajoute des métadonnées à l'image avec un système de clés-valeurs, permet par exemple d'indiquer à l'utilisateur l'auteur du Dockerfile.

- ARG : Variables temporaires qu'on peut utiliser dans un Dockerfile.
- ENV : Variables d'environnements utilisables dans votre Dockerfile et conteneur.
- RUN : Exécute des commandes Linux ou Windows lors de la création de l'image. Chaque instruction RUN va créer une couche en cache qui sera réutilisée dans le cas de modification ultérieure du Dockerfile.

- COPY : Permet de copier des fichiers depuis notre machine locale vers le conteneur Docker.
- ADD : Même chose que COPY mais prend en charge des liens ou des archives (si le format est reconnu, alors il sera décompressé à la volée).
- ENTRYPOINT : comme son nom l'indique, c'est le point d'entrée de votre conteneur, en d'autres termes, c'est la commande qui sera toujours exécutée au démarrage du conteneur. Il prend la forme de tableau JSON (ex : CMD ["cmd1","cmd1"]) ou de texte.

- CMD : Spécifie les arguments qui seront envoyés au ENTRYPOINT, (on peut aussi l'utiliser pour lancer des commandes par défaut lors du démarrage d'un conteneur). Si il est utilisé pour fournir des arguments par défaut pour l'instruction ENTRYPOINT, alors les instructions CMD et ENTRYPOINT doivent être spécifiées au format de tableau JSON.
- EXPOSE : Expose un port

- WORKDIR : Définit le répertoire de travail qui sera utilisé pour le lancement des commandes CMD et/ou ENTRYPOINT et ça sera aussi le dossier courant lors du démarrage du conteneur.
- VOLUMES : Crée un point de montage qui permettra de persister les données.
- USER : Désigne quel est l'utilisateur qui lancera les prochaines instructions RUN, CMD ou ENTRYPOINT (par défaut c'est l'utilisateur root).

Pour la création de notre image, commencez par créer un dossier, positionnez vous dans ce dossier et téléchargez les sources de l'image :

```
$ wget https://devopssec.fr/documents/docker/dockerfile/sources.zip
```

Ensuite dézipper le fichier .zip avec la commande unzip :

```
$ unzip sources.zip
```

Notre dossier doit donc avoir les éléments suivants :

- db : contient un fichier articles.sql, qui renferme toute l'architecture de la base de données.
- app : comporte les sources php de notre l'application web.

Ensuite dans la racine du dossier que vous venez de créer, créez un fichier et nommez le Dockerfile, puis rajoutez le contenu suivant :

```
vagrant@debian10:~/build_docker$ cat Dockerfile
# --------------- DÉBUT COUCHE OS -------------------
FROM debian:stable-slim
# --------------- FIN COUCHE OS ---------------------


# MÉTADONNÉES DE L'IMAGE
LABEL version="1.0" maintainer="MOTREFF Dayan <dayan.motreff@solyti.fr"


# VARIABLES TEMPORAIRES
ARG APT_FLAGS="-q -y"
ARG DOCUMENTROOT="/var/www/html"



# --------------- DÉBUT COUCHE APACHE ---------------
RUN apt-get update -y && \
    apt-get install ${APT_FLAGS} apache2
# --------------- FIN COUCHE APACHE -----------------
```

```
# --------------- DÉBUT COUCHE MYSQL ----------------
RUN apt-get install ${APT_FLAGS} mariadb-server

COPY db/articles.sql /
# --------------- FIN COUCHE MYSQL ------------------


# --------------- DÉBUT COUCHE PHP ------------------
RUN apt-get install ${APT_FLAGS} \
    php-mysql \
    php && \
    rm -f ${DOCUMENTROOT}/index.html && \
    apt-get autoclean -y

COPY app ${DOCUMENTROOT}
# --------------- FIN COUCHE PHP --------------------


# OUVERTURE DU PORT HTTP
EXPOSE 80


# RÉPERTOIRE DE TRAVAIL
WORKDIR  ${DOCUMENTROOT}


# DÉMARRAGE DES SERVICES LORS DE L'EXÉCUTION DE L'IMAGE
ENTRYPOINT service mariadb start && mysql < /articles.sql && apache2ctl -D FOREGROUND
```

SOLYTI

Au final vous devez obtenir l'architecture suivante :

```
vagrant@debian10:~/build_docker$ tree
.
├── app
│   ├── db-config.php
│   └── index.php
├── db
│   └── articles.sql
├── Dockerfile
└── sources.zip

2 directories, 5 files
```

Pour créer ma couche OS, je me suis basée sur l'image debian-slim. Vous pouvez, choisir une autre image si vous le souhaitez (il existe par exemple une image avec une couche OS nommée alpine, qui ne pèse que 5 MB !), sachez juste qu'il faut adapter les autres instructions si jamais vous choisissez une autre image de base.

```
# -------------- DÉBUT COUCHE OS -------------------
FROM debian:stable-slim
# -------------- FIN COUCHE OS ---------------------
```

Ensuite, j'ai rajouté les métadonnées de mon image. Comme ça, si un jour je décide de partager mon image avec d'autres personnes, alors ils pourront facilement récolter des métadonnées sur l'image (ex: l'auteur de l'image) depuis la commande docker inspect <IMAGE_NAME>.

```
# MÉTADONNÉES DE L'IMAGE
LABEL version="1.0" maintainer="MOTREFF Dayan <dayan.motreff@solyti.fr"
```

Ici, j'ai créé deux variables temporaires qui ne me serviront qu'au sein de mon Dockerfile, d'où l'utilisation de l'instructionARG. La première variable me sert comme arguments pour la commande apt, et la seconde est le répertoire de travail de mon apache.

```
# VARIABLES TEMPORAIRES
ARG APT_FLAGS="-q -y"
ARG DOCUMENTROOT="/var/www/html"
```

Par la suite, j'ai construit ma couche Apache. Pour cela j'ai d'abord commencé par récupérer la liste de paquets et ensuite j'ai installé mon Apache.

```
# --------------- DÉBUT COUCHE APACHE ---------------
RUN apt-get update -y && \
    apt-get install ${APT_FLAGS} apache2
# --------------- FIN COUCHE APACHE -----------------
```

Ici, je commence d'abord par télécharger le service mysql et ensuite je rajoute mon fichier articles.sql pour mon futur nouveau conteneur.

```
# -------------- DÉBUT COUCHE MYSQL ----------------
RUN apt-get install ${APT_FLAGS} mariadb-server

COPY db/articles.sql /
# -------------- FIN COUCHE MYSQL -----------------
```

Ici j'installe l'interpréteur php ainsi que le module php-mysql. j'ai ensuite vidé le cache d'apt-get afin de gagner en espace de stockage. J'ai aussi supprimé le fichier index.html du DocumentRoot d'Apache (par défaut /var/www/html), car je vais le remplacer par mes propres sources.

```
# -------------- DÉBUT COUCHE PHP ----------------
RUN apt-get install ${APT_FLAGS} \
    php-mysql \
    php && \
    rm -f ${DOCUMENTROOT}/index.html && \
    apt-get autoclean -y

COPY app ${DOCUMENTROOT}
# -------------- FIN COUCHE PHP -------------------
```

J’ouvre le port http.

```
# OUVERTURE DU PORT HTTP
EXPOSE 80
```

Je met le dossier /var/www/html en tant que répertoire de travail, comme ça, quand je démarrerai mon conteneur, alors je serai directement sur ce dossier.

```
# RÉPERTOIRE DE TRAVAIL
WORKDIR  ${DOCUMENTROOT}
```

Ici, lors du lancement de mon conteneur, le service mariadb démarrera et construira l'architecture de la base de données grâce à mon fichier articles.sql . Maintenant, il faut savoir qu'un conteneur se ferme automatiquement à la fin de son processus principal. Il faut donc un processus qui tourne en premier plan pour que le conteneur soit toujours à l'état running, d'où le lancement du service Apache en premier plan à l'aide de la commande ’apache2 -D FOREGROUND’.

```
# DÉMARRAGE DES SERVICES LORS DE L'EXÉCUTION DE L'IMAGE
ENTRYPOINT service mariadb start && mysql < /articles.sql && apache2ctl -D FOREGROUND
```

Voici la commande pour qui nous permet de construire une image docker depuis un Dockerfile :

```
$ docker build -t <IMAGE_NAME> .
```

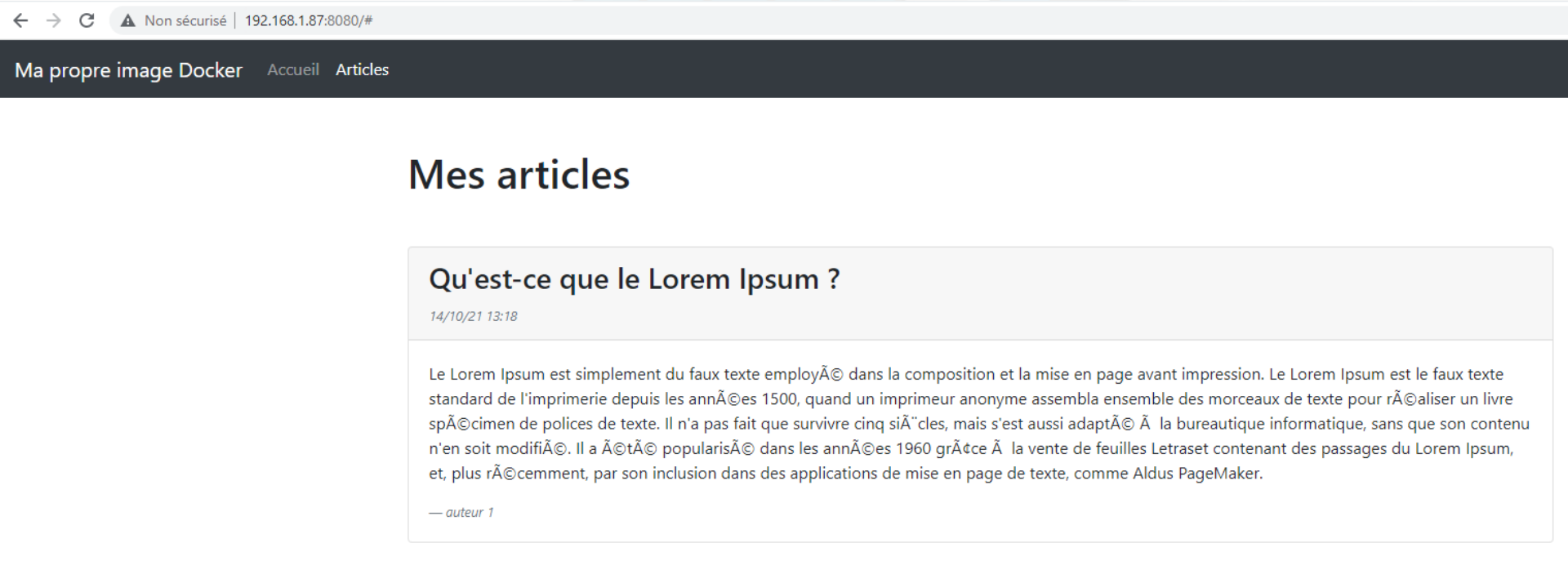
Ce qui nous donnera :

```
$ docker build -t my_lamp .
```

Ensuite, exécutez votre image personnalisée :

```
$ docker run –d --name my_lamp_c –p 8080:80 my_lamp
```

On obtient alors :



Si vous souhaitez partager votre image avec d'autres utilisateurs, une des possibilités est d'utiliser le Hub Docker.

Pour cela, commencez par vous inscrire sur la plateforme et créez ensuite un repository public.

L'étape suivante est de se connecter au hub Docker à partir de la ligne de commande :

```
$ docker login
```

Récupérer ensuite l'id ou le nom de votre image :

```
$ docker images
```

Ensuite il faut rajouter un tag à l'id ou le nom de l'image récupérée :

```
$ docker tag <IMAGENAME OU ID> <HUB-USER>/<REPONAME>[:<TAG>]
```

Maintenant vous pouvez push votre image vers le Hub Docker grâce à la commande suivante :

```
$ docker push <HUB-USER>/<REPONAME>[:<TAG>]
```

Chapitre 8

Fonctionnement et manipulation des volumes dans Docker

SOLYTI

Afin de pouvoir sauvegarder (persister) les données et également partager des données entre conteneurs, Docker a développé le concept de volumes.

Les volumes sont des répertoires (ou des fichiers) qui ne font pas partie du système de fichiers Union mais qui existent sur le système de fichiers hôte.

En outre, les volumes constituent souvent le meilleur choix pour persistance des données pour le layer en lecture-écriture, car un volume n'augmente pas la taille des conteneurs qui l'utilisent et son contenu existe en dehors du cycle de vie d'un conteneur donné.

Contrairement à un montage lié, vous pouvez créer et gérer des volumes en dehors de la portée de tout conteneur.

Pour créer un volume, nous utiliserons la commande suivante :

```
$ docker volume create <VOLUMENAME>
```

Pour lister les volumes :

```
$ docker volume ls
```

Pour récolter des informations sur un volume, il faut utiliser la commande suivante :

```
$ docker volume inspect <VOLUMENAME>
```

Pour supprimer un volume :

```
$ docker volume rm <VOLUMENAME>
```

```
vagrant@debian10:~/build_docker2$ docker volume create volume-test
volume-test
vagrant@debian10:~/build_docker2$ docker volume ls
DRIVER              VOLUME NAME
local               3a6f7440793152650928446 3d1af73e3f28702133eb417c1dd7b290067d15c1a
local               36ef929f7608e8436caf964329d19a2b8df52a46d76d361ca823033fecb48ab8
local               44ff9b0dcfa07399a4d3bedc57e2d21dc39b4dda5a3cb6701f4095fa79d88126
local               49dc8d29f6e46128b94f766c89f3bce622df275972a3642bf3ac82a43b0d9195
local               60cb332209566ac99d87d27a433ce9371e26bca4c9a2a7a7849d65f61d2f2946
local               71f7191a73ab519b2eeaa7f69eb433317307a6d311c1741ab412a2f26d204433
local               922e995920dcff9a98d198aaad672f47ab7672014c555c2239015ab56abe8889
local               2509aea7ab3c72ebc8ac33c798f538b3fe8005dcb4abd16870293424f7d20702
local               d4656fdd66a41e17185eedac5a0344d5646154319006a46cc48a058095843f5a
local               d170756e687fb58001a23b56db780d4d248246ab368c94fb809a185f687836a1
local               e435a91d0f579586b02f69bef30cb47d00e6a41d8dddce4dcbb5abc232c8a218
local               elk-cec-docker_elasticsearch
local               f93c03fa204b93262df54d351e73feaba9b0a551745944f22a804cf57103cc6a
local               svn_config
local               svnadmin_config
local               volume-test
vagrant@debian10:~/build_docker2$ docker volume inspect volume-test
[
    {
        "CreatedAt": "2021-10-14T14:15:45Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/volume-test/_data",
        "Name": "volume-test",
        "Options": {},
        "Scope": "local"
    }
]
```

SOLYTI

Si vous démarrez un conteneur avec un volume qui n'existe pas encore, Docker le créera pour vous.

Pour démarrer un conteneur avec un volume, il faut utiliser l'option -v de la commande docker run.

La commande suivante va créer et monter le volume data-test dans le dossier /data du conteneur :

```
$ docker run -ti --name vtest_c -v data-test:/data vtest
```

Si vous démarrez un conteneur avec un volume qui n'existe pas encore, Docker le créera pour vous.

Pour démarrer un conteneur avec un volume, il faut utiliser l'option -v de la commande docker run.

La commande suivante va créer et monter le volume data-test dans le dossier /data du conteneur :

```
$ docker run -ti --name vtest_c -v data-test:/data vtest
```

Les informations concernant les volumes pour la persistance des données sont souvent détaillées sur le docker hub repo.

Vous pouvez utiliser aussi les dossiers de votre système pour le montage des volumes :

```
$ docker run --name some-mariadb -v $PWD/mariadb_data:/var/lib/mysql -e MARIADB_ROOT_PASSWORD=my-secret-pw -d mariadb
```

Recap :

```
## Créer une volume
docker volume create <VOLUME NAME>

# Lister les volumes
docker volume ls

## Supprimer un ou plusieurs volume(s)
docker volume rm <VOLUME NAME>
    -f ou --force : forcer la suppression

## Récolter des informations sur une volume
docker volume inspect <VOLUME NAME>

## Supprimer tous les volumes locaux non inutilisés
docker volume prune
    -f ou --force : forcer la suppression

## Supprimer un conteneur Docker avec le/les volumes associés
docker rm -v <CONTAINER_ID ou CONTAINER_NAME>
    -f ou --force : forcer la suppression
    -v ou --volume : supprime les volumes associés au conteneur
```

Chapitre 9

Gérez vos conteneurs avec le Docker Compose

SOLYTI

Docker Compose est un outil permettant de définir le comportement de vos conteneurs et d'exécuter des applications Docker à conteneurs multiples.

La config se fait à partir d'un fichier YAML, et ensuite, avec une seule commande, vous créez et démarrez tous vos conteneurs de votre configuration.

Docker Compose n'est pas installé par défaut et s'appuie sur le moteur Docker pour fonctionner.

Pour installer docker-compose plusieurs choix s’offre à vous, dans ce cours j’utilise le gestionnaire de paquet apt de debian 10 :

```
$ apt install docker-compose
```

Dans cette partie nous allons améliorer notre ancienne application LAMP. Par la suite nous allons séparer le conteneur de notre application web par rapport au conteneur de notre base de données.

Au préalable, commencez par télécharger les sources :

$ wget https://devopssec.fr/documents/docker/docker-compose/sources.zip

Ensuite dezipper le fichier telechargé :

```
$ unzip sources.zip
```

Pour améliorer le Dockerfile de notre stack LAMP en réduisant son nombre d'instructions, on se basera sur une nouvelle image.

En cherchant dans le Hub Docker, on trouve les images adéquates, notamment :

- Une image officielle php avec le tag 7-apache
- Une image officielle mysql

Une fois que l'on a trouvé les bonnes images, je peux alors m'attaquer à la modification du Dockerfile.

Pour le moment, nous utiliserons ce Dockerfile seulement pour construire une image avec une couche OS, Apache et Php sans implémenter aucun service de base de données. Cette image se basera sur l'image officielle php avec le tag 7-apache qui vient déjà avec un OS (distribution Debian). Concernant l'image mysql nous l'utiliserons plus tard dans notre docker-compose.yml..

Dans le même dossier que vous avez désarchivé, créez un fichier Dockerfile et mettez dedans le contenu suivant :

```
vagrant@debian10:~/compose_cours/sources$ cat Dockerfile
FROM php:7-apache

LABEL version="1.0" maintainer="Motreff Dayan "

# Activation des modules php
RUN docker-php-ext-install pdo pdo_mysql

WORKDIR  /var/www/html
```

Buildez ensuite votre image avec la commande suivante :

```
$ docker build –t myapp .
```

On va débuter par la récolte des besoins du conteneur de la base de données. Pour celle-ci, il nous faudra :

- Un fichier sql pour créer l'architecture de notre base de données.
- Un volume pour stocker les données.

Il est toujours important de vérifier avant ce que nous propose la page Docker Hub de l'image mysql. En lisant sa description, les informations qui m'ont le plus captivé sont ses variables d'environnements qu'on peut surcharger, notamment :

- MYSQL_ROOT_PASSWORD: spécifie le mot de passe qui sera défini pour le compte MySQL root (c'est une variable obligatoire).

- MYSQL_DATABASE: spécifie le nom de la base de données à créer au démarrage de l'image.
- MYSQL_USER et MYSQL_PASSWORD : utilisées conjointement pour créer un nouvel utilisateur avec son mot de passe. Cet utilisateur se verra accorder des autorisations de super-utilisateur pour la base de données MYSQL_DATABASE.

Ces variables d'environnements vont nous aider à créer une partie de l'architecture de notre base de données.

Dans la description de l'image mysql, il existe une autre information très utile. Lorsqu'un conteneur mysql est démarré, il exécutera des fichiers avec des extensions .sh, .sql et .sql.gz qui se trouvent dans /docker-entrypoint-initdb.d.

Nous allons profiter de cette information pour déposer le fichier articles.sql (disponible dans les sources téléchargées) dans le dossier /docker-entrypoint-initdb.d afin de créer automatiquement notre table SQL.

Concernant le conteneur de l'application web, nous aurons besoin de :

- Une communication avec le conteneur de la base de données.
- Un volume pour stocker les sources de l'application web.

Me concernant la seule information utile dans la description de la page Docker Hub de l'image php, est qu'il est possible d'installer et d'activer les modules php dans le conteneur php avec la commande docker-php-ext-install (C'est la commande utilisée dans notre Dockerfile afin d'activer le module pdo et pdo_mysql).

Histoire de vous donner une idée sur la longueur de la commande docker run sans utiliser le fichier docker-compose.yml. Je vais alors l'utiliser pour démarrer les différents conteneurs de notre application.

Premièrement je vais vous dévoiler, deux nouvelles options de la commande docker run :

- -e : définit/surcharge des variables d'environnement
- --link : ajoute un lien à un autre conteneur afin de les faire communiquer entre eux.

Voici à quoi va ressembler la commande docker run pour la création du conteneur de la base de données :

```
docker run -d -e MYSQL_ROOT_PASSWORD='test' \
-e MYSQL_DATABASE='test' \
-e MYSQL_USER='test' \
-e MYSQL_PASSWORD='test' \
--volume db-volume:/var/lib/mysql \
--volume $PWD/articles.sql:/docker-entrypoint-initdb.d/articles.sql \
--name mysql_c mysql:5.7
```

Voici à quoi va ressembler la commande docker run pour la création du conteneur de l'application web :

```
docker run -d --volume $PWD/app:/var/www/html -p 8080:80 --link mysql_c --name myapp_c myapp
```

Commencez d'abord par créer un fichier et nommez le docker-compose.yml, ensuite copiez collez le contenu ci-dessous :

```
vagrant@debian10:~/compose_cours/sources$ cat docker-compose.yml
version: '3'

services:
    db:
        image: mysql:5.7
        container_name: mysql_c
        restart: always
        volumes:
            - db-volume:/var/lib/mysql
            - ./articles.sql:/docker-entrypoint-initdb.d/articles.sql
        environment:
            MYSQL_ROOT_PASSWORD: test
            MYSQL_DATABASE: test
            MYSQL_USER: test
            MYSQL_PASSWORD: test

    app:
        image: myapp
        container_name: myapp_c
        restart: always
        volumes:
            - ./app:/var/www/html
        ports:
            - 8080:80
        depends_on:
            - db

volumes:
    db-volume:
```

Il existe plusieurs versions rétrocompatibles pour le format du fichier Compose (voici la liste des versions de Docker Compose selon la version moteur Docker). Dans mon cas je suis sous la version 18.09.7 du moteur Docker, donc j'utilise la version 3.

```
version: '3'
```

Dans une application Docker distribuée, différentes parties de l'application sont appelées services. Les services ne sont en réalité que des conteneurs. Dans notre cas nous aurons besoin d'un service pour notre base de données et un autre pour notre application web.

```
services:
```

```
db:
    image: mysql:5.7
    container_name: mysql_c
    restart: always
    volumes:
        - db-volume:/var/lib/mysql
        - ./articles.sql:/docker-entrypoint-initdb.d/articles.sql
    environment:
        MYSQL_ROOT_PASSWORD: test
        MYSQL_DATABASE: test
        MYSQL_USER: test
        MYSQL_PASSWORD: test
```

Dans cette partie, on crée un service nommé db. Ce service indique au moteur Docker de procéder comme suit :

1. Se baser sur l'image mysql:5.7
2. Nommer le conteneur mysql_c
3. Le restart: always démarrera automatiquement le conteneur en cas de redémarrage du serveur

4. Définir les volumes à créer et utiliser (un volume pour exécuter automatiquement notre fichier sql et un autre pour sauvegarder les données de la base de données)
5. Surcharger les variables d'environnements à utiliser

```
app:
    image: myapp
    container_name: myapp_c
    restart: always
    volumes:
        - ./app:/var/www/html
    ports:
        - 8080:80
    depends_on:
        - db
```

Ici, on crée un service nommé app. Ce service indique au moteur Docker de procéder comme suit :

1. Se baser sur l'image nommée myapp qu'on avait construit depuis notre Dockerfile
2. Nommer le conteneur myapp_c
3. Le restart: always démarrera automatiquement le conteneur en cas de redémarrage du serveur
4. Définir les volumes à créer et à utiliser pour sauvegarder les sources de notre application

5. Mapper le port 8080 sur le port 80
6. Le depends_on indique les dépendances du service app. Ces dépendances vont provoquer les comportements suivants :
    1. Les services démarrent en ordre de dépendance. Dans notre cas, le service db est démarré avant le service app
    2. Les services s'arrêtent selon l'ordre de dépendance. Dans notre cas, le service app est arrêté avant le service db

Enfin, je demande au moteur Docker de me créer un volume nommé db-volume, c'est le volume pour stocker les données de notre base de données.

```
volumes:
    db-volume:
```

Placez vous au niveau du dossier qui contient le fichier docker-compose.yml. Ensuite lancez la commande suivante pour exécuter les services du docker-compose.yml :

```
docker-compose up -d
```

Ici l'option -d permet d'exécuter les conteneur du Docker compose en arrière-plan.

Pour seulement lister les conteneurs du docker-compose.yml, il suffit d'exécuter la commande suivante :

```
docker-compose ps
```

Si tout c'est bien passé, alors visitez la page suivante http://localhost:8080/, et vous obtiendrez le résultat suivant :


Non sécurisé | 192.168.1.87:8080
Ma propre image Docker
Accueil
Articles
Articles
Nouveau article
Titre *
titre de votre article
Nom de l'auteur *
Nom de l'auteur
Contenu *
Envoyer

Remplissez le formulaire de l'application, et tuez les conteneurs du docker-compose.yml, avec la commande suivante :

```
docker-compose kill
```

Relancez ensuite vos services, et vous verrez que vos données sont bel et bien sauvegardées.

Je ne vais pas trop rentrer dans les détails sur la partie réseau, car je vais rédiger un article qui sera dédié à cette partie. Mais sachez juste qu'un réseau bridge est créé par défaut, plus précisément c'est l'interface docker0 (ip addr show docker0), c'est un réseau qui permet une communication entre les différents conteneurs.

Donc les conteneurs possèdent par défaut une adresse ip. Vous pouvez récolter cette information grâce à la commande suivante :

```
docker inspect -f '{{.Name}} - {{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' $(docker ps -aq)
```

Résultat :

```
/myapp_c - 192.168.16.3
/mysql_c - 192.168.16.2
```

Pour faire communiquer notre application web avec la base de données, on peux utiliser dans le conteneur de l'app web soit l'ip, le nom du service (ici db) ou le nom du conteneur (ici mysql_c) de la base de données.

Si vous ouvrez le fichier db-config.php dans le dossier app, alors vous verrez la ligne suivante :

```
const DB_DSN = 'mysql:host=mysql_c;dbname=test';
```

Recap :

```
## Exécuter les services du docker-compose.yml
docker-compose up
    -d : Exécuter les conteneurs en arrière-plan

## Lister des conteneurs du Docker Compose
docker-compose ls
    -a ou --all : afficher aussi les conteneurs stoppés

## Sorties/erreurs des conteneurs du Docker Compose
docker-compose logs
    -f : suivre en permanence les logs du conteneur
    -t : afficher la date et l'heure de la réception de la ligne de log
    --tail=<NOMBRE DE LIGNE> = nombre de lignes à afficher à partir de la fin pour chaque conteneur.

## Tuer les conteneurs du Docker Compose
docker-compose kill
```

```
## Stopper les conteneurs du Docker Compose
docker-compose stop
    -t ou --timeout : spécifier un timeout en seconde avant le stop (par défaut : 10s)

## Démarrer les conteneurs du Docker Compose
docker-compose start

## Arrêtez les conteneurs et supprimer les conteneurs, réseaux, volumes, et les images
docker-compose down
    -t ou --timeout : spécifier un timeout en seconde avant la suppression (par défaut : 10s)

## Supprimer des conteneurs stoppés du Docker Compose
docker-compose rm
    -f ou --force : forcer la suppression

## Lister les images utilisées dans le docker-compose.yml
docker-compose images
```

Chapitre 10

Fonctionnement et manipulation du réseau dans Docker

SOLYTI

Pour que les conteneurs Docker puissent communiquer entre eux mais aussi avec le monde extérieur via la machine hôte, alors une couche de mise en réseau est nécessaire.

Cette couche réseau rajoute une partie d'isolation des conteneurs, et permet donc de créer des applications Docker qui fonctionnent ensemble de manière sécurisée.

Docker prend en charge différents types de réseaux qui sont adaptés à certains cas d'utilisation

Le système réseau de Docker utilise des drivers (pilotes). Plusieurs drivers existent et fournissent des fonctionnalités différentes.

1. Le driver Bridge
2. Le driver none
3. Le driver host
4. Le driver overlay
5. Le driver macvlan

Le driver Bridge

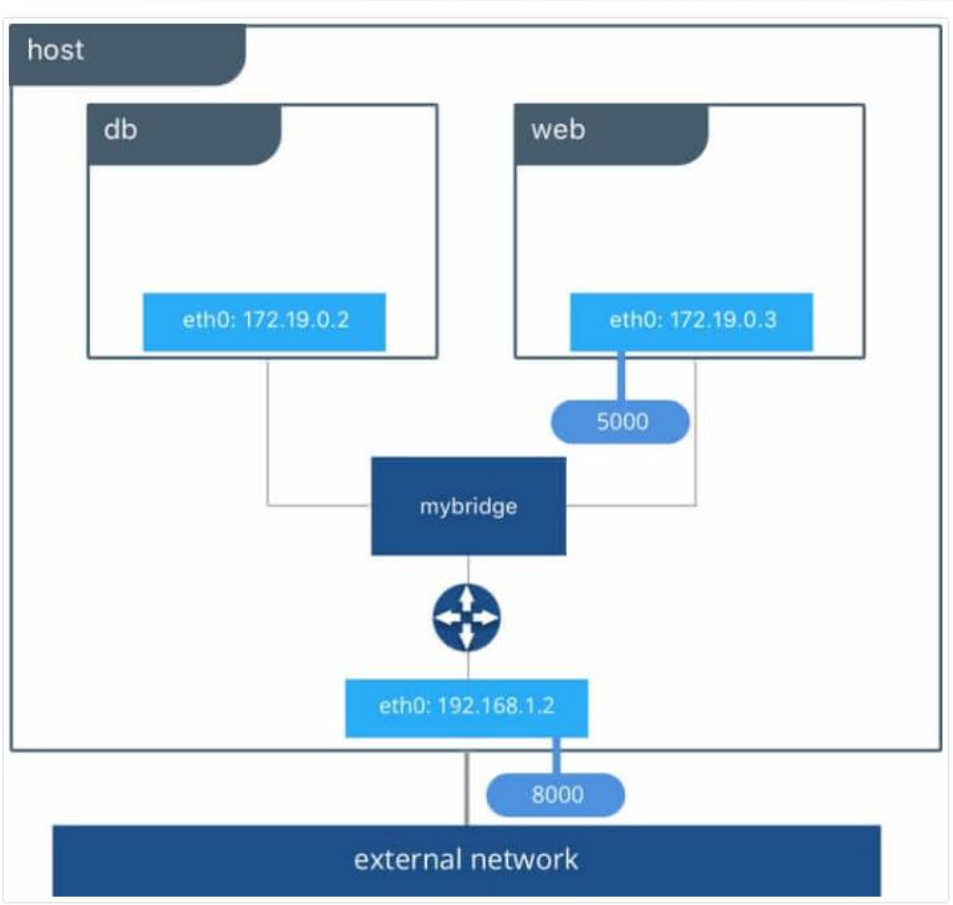
Tout d'abord, lorsque vous installez Docker pour la première fois, il crée automatiquement un réseau bridge nommé bridge connecté à l'interface réseau docker0 (consultable avec la commande ip addr show docker0).

Chaque nouveau conteneur Docker est automatiquement connecté à ce réseau, sauf si un réseau personnalisé est spécifié.

Par ailleurs, le réseau bridge est le type de réseau le plus couramment utilisé. Il est limité aux conteneurs d'un hôte unique exécutant le moteur Docker. Les conteneurs qui utilisent ce driver, ne peuvent communiquer qu'entre eux, cependant ils ne sont pas accessibles depuis l'extérieur.

Pour que les conteneurs sur le réseau bridge puissent communiquer ou être accessibles du monde extérieur, vous devez configurer le mappage de port.

Fonctionnement et manipulation du réseau dans Docker

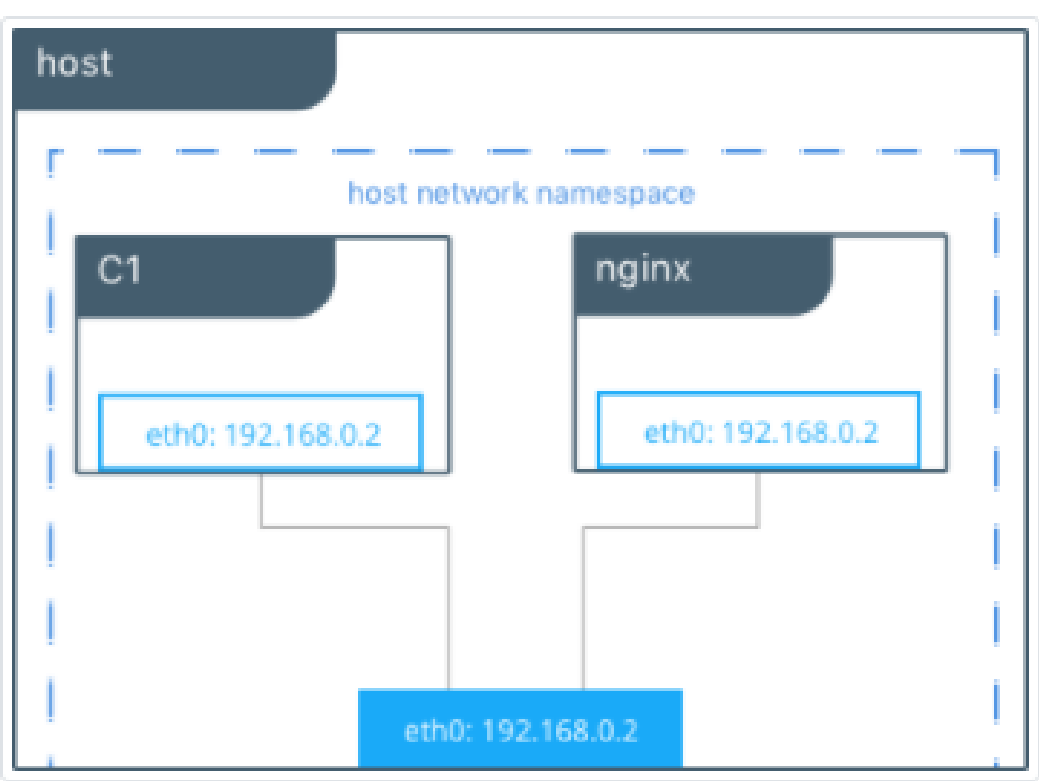


Le driver none

C'est le type de réseau idéal, si vous souhaitez interdire toute communication interne et externe avec votre conteneur, car votre conteneur sera dépourvu de toute interface réseau (sauf l'interface loopback).

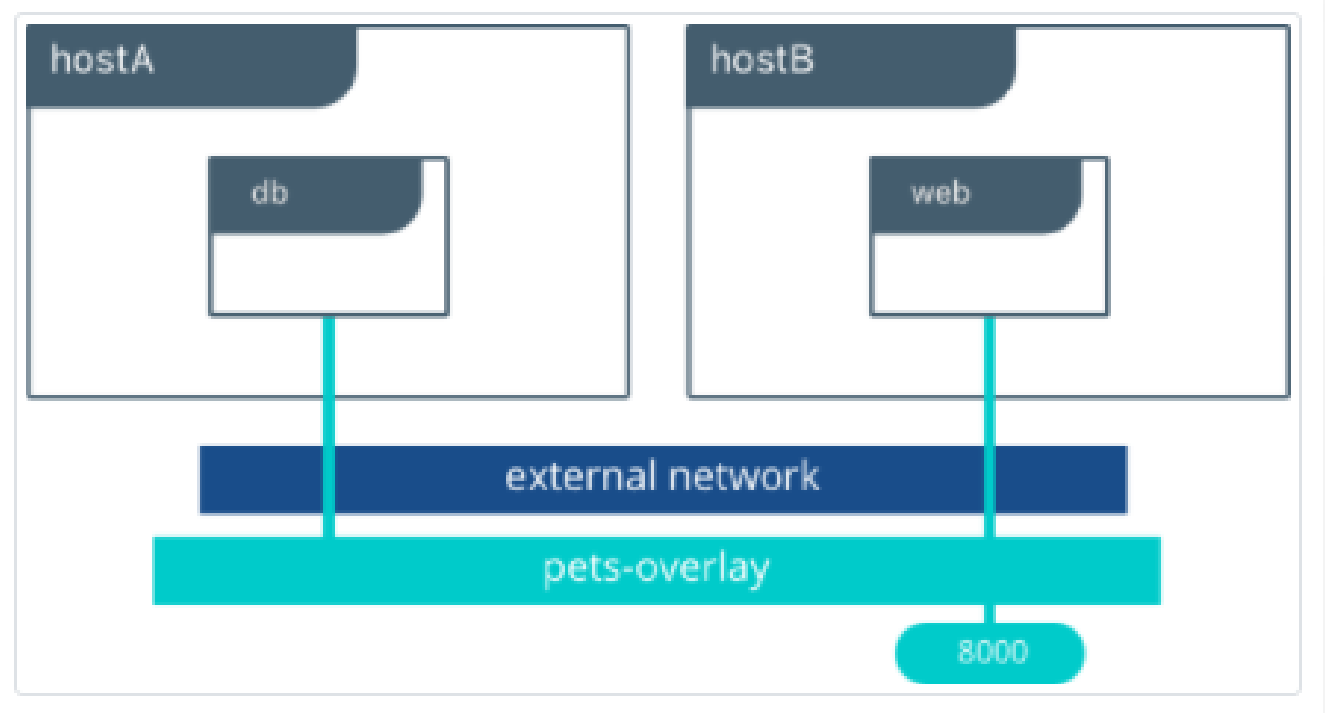
Le driver host

Ce type de réseau permet aux conteneurs d'utiliser la même interface que l'hôte. Il supprime donc l'isolation réseau entre les conteneurs et seront par défaut accessibles de l'extérieur. De ce fait, il prendra la même IP que votre machine hôte.


host
host network namespace
C1
eth0: 192.168.0.2
nginx
eth0: 192.168.0.2
eth0: 192.168.0.2

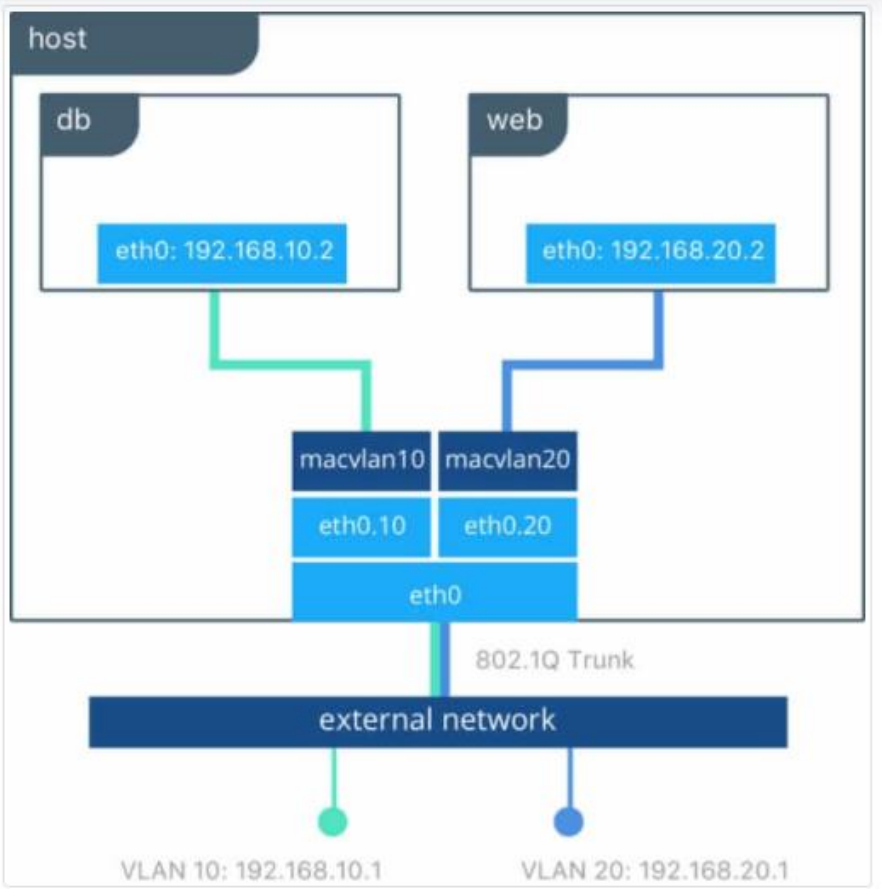
Le driver overlay

Si vous souhaitez une mise en réseau multi-hôte native, vous devez utiliser un driver overlay. Il crée un réseau distribué entre plusieurs hôtes possédant le moteur Docker. Docker gère de manière transparente le routage de chaque paquet vers et depuis le bon hôte et le bon conteneur.


hostA
db
hostB
web
external network
pets-overlay
8000

Le driver macvlan

L'utilisation du driver macvlan est parfois le meilleur choix lorsque vous utilisez des applications qui s'attendent à être directement connectées au réseau physique, car le driver Macvlan vous permet d'attribuer une adresse MAC à un conteneur, le faisant apparaître comme un périphérique physique sur votre réseau. Le moteur Docker route le trafic vers les conteneurs en fonction de leurs adresses MAC.


host
db
eth0: 192.168.10.2
web
eth0: 192.168.20.2
macvlan10
macvlan20
eth0.10
eth0.20
eth0
802.1Q Trunk
external network
VLAN 10: 192.168.10.1
VLAN 20: 192.168.20.1

La commande pour créer un réseau Docker est la suivante :

```
docker network create --driver <DRIVER TYPE> <NETWORK NAME>
```

Dans cet exemple nous allons créer un réseau de type bridge nommé mon-bridge :

```
docker network create --driver bridge mon-bridge
```

On va ensuite lister les réseaux docker avec la commande suivante :

```
docker network ls
```

Résultat :

```
NETWORK ID          NAME                DRIVER              SCOPE
8f7d6733e8e8        bridge              bridge              local
252db255c98e        host                host                local
33ca6ac507db        jenkins_default     bridge              local
bbc00cbbdde1        mon-bridge          bridge              local
c2dd1d76d496        none                null                local
```

Il est possible de récolter des informations sur le réseau docker, comme par exemple la config réseau, en tapant la commande suivante :

```
docker network inspect mon-bridge
```

Résultat :

```
vagrant@debian10:~/compose_cours/sources$ docker network inspect mon-bridge
[
    {
        "Name": "mon-bridge",
        "Id": "bbc00cbbdde138b12304349e377245d6a9d216ebc8a5a9bb39a31928e4eba936",
        "Created": "2021-10-17T12:38:41.713810207Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "192.168.32.0/20",
                    "Gateway": "192.168.32.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

SOLYTI

Pour cet exemple, nous allons connecter deux conteneurs à notre réseau bridge créé précédemment :

```
docker run -dit --name alpine1 --network mon-bridge alpine
```

```
docker run -dit --name alpine2 --network mon-bridge alpine
```

Si on inspecte une nouvelle fois notre réseau mon-bridge, on verra nos deux nouveaux conteneurs dans les informations retournées :

```
docker network inspect mon-bridge
```

Résultat :


```
"Containers": {
    "e6017925a17dc864b22bdb356029037a90e164e3af3c1c0a19b03fb4d4903fe2": {
        "Name": "alpine1",
        "EndpointID": "b23bff7dd7dc7681faf548e6bcb46fc67241089993c2091b5bac5330e5d6db2b",
        "MacAddress": "02:42:c0:a8:20:02",
        "IPv4Address": "192.168.32.2/20",
        "IPv6Address": ""
    },
    "f355c3603cbe9e887496f858b45eb619fa1f20f0c326f9d361f7f2f64971cfdc": {
        "Name": "alpine2",
        "EndpointID": "c70953da58d0465fdb9db2316bcf1ed992b9162cc7722d802f122fac336d6d75",
        "MacAddress": "02:42:c0:a8:20:03",
        "IPv4Address": "192.168.32.3/20",
        "IPv6Address": ""
    }
},
```

D'après le résultat, on peut s'apercevoir que notre conteneur alpine1 possède l'adresse IP 192.168.32.2, et notre conteneur alpine2 possède l'adresse IP 192.168.32.3.

Tentons de les faire communiquer ensemble à l'aide de la commande ping :

```
docker exec alpine1 ping -c 1 192.168.32.3
```

Résultat :

```
PING 192.168.32.3 (192.168.32.3): 56 data bytes
64 bytes from 192.168.32.3: seq=0 ttl=64 time=0.112 ms

--- 192.168.32.3 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.112/0.112/0.112 ms
```

Pour information, vous ne pouvez pas créer un network host, car vous utilisez l'interface de votre machine hôte. D'ailleurs si vous tentez de le créer alors vous recevrez l'erreur suivante :

```
docker network create --driver host mon-host
```

Erreur :

```
Error response from daemon: only one instance of "host" network is allowed
```

On ne peut qu'utiliser le driver host mais pas le créer.

Dans cet exemple nous allons démarrer un conteneur Apache sur le port 80 de la machine hôte.

Du point de vue de la mise en réseau, il s'agit du même niveau d'isolation que si le processus Apache s'exécutait directement sur la machine hôte et non dans un conteneur.

Cependant, le processus reste totalement isolé de la machine hôte.

Cette procédure nécessite que le port 80 soit disponible sur la machine hôte :

```
docker run --rm -d --network host --name my_httpd httpd
```

Depuis votre machine hôte, vous pouvez vérifier quel processus est lié au port 80 à l'aide de la commande netstat :

```
vagrant@debian10:~/compose_cours/sources$ sudo netstat -antp | grep :80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      31102/httpd
```

C'est bien le processus httpd qui utilise le port 80 sans avoir recours au mappage de port.

Enfin arrêtez le conteneur qui sera supprimé automatiquement car il a été démarré à l'aide de l'option --rm :

```
vagrant@debian10:~/compose_cours/sources$ docker stop my_httpd
my_httpd
vagrant@debian10:~/compose_cours/sources$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
f355c3603cbe        alpine              "/bin/sh"           21 minutes ago      Up 21 minutes                           alpine2
e6017925a17d        alpine              "/bin/sh"           21 minutes ago      Up 21 minutes                           alpine1
```

Recap :

```
## Créer un réseau docker
docker network create --driver <DRIVER TYPE> <NETWORK NAME>

# Lister les réseaux docker
docker network ls

## Supprimer un ou plusieurs réseau(x) docker
docker network rm <NETWORK NAME>

## Récolter des informations sur un réseau docker
docker network inspect <NETWORK NAME>
    -v ou --verbose : mode verbose pour un meilleur diagnostique
```

```
## Supprimer tous les réseaux docker non inutilisés
docker network prune
    -f ou --force : forcer la suppression

## Connecter un conteneur à un réseau docker
docker network connect <NETWORK NAME> <CONTAINER NAME>

## Déconnecter un conteneur à réseau docker
docker network disconnect <NETWORK NAME> <CONTAINER NAME>
    -f ou --force : forcer la déconnexion

## Démarrer un conteneur et le connecter à un réseau docker
docker run --network <NETWORK NAME> <IMAGE NAME>
```