

Introduction du cours pour apprendre la technologie Docker

Ce cours complet pour débutants sur la technologie Docker vous expliquera pas à pas les différentes notions de Docker.

Histoire

Hello world, aujourd'hui je vais vous présenter l'une de mes technologies préférées, à savoir **Docker**.

Docker est une plateforme lancée 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 **process 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. (plus d'explications seront fournies dans un chapitre dédié aux conteneurs).

Avant de commencer, il faut savoir que le terme Docker signifie à la fois une technologie et à la fois une entreprise, ce qui pourrait parfois porter à confusion.

Docker a été fondée en France (cocorico ☐☐☐), 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.

Son rythme d'adoption serait l'un des plus rapide de toutes les technologies récentes, elle est déjà largement répandue chez les acteurs leaders des nouvelles technologies.

Après dix ans chez Docker, le fondateur et CTO de Docker Solomon Hykes, a annoncé son départ de l'entreprise le 28 mars 2018. Il reste tout de même membre du conseil d'administration et l'actionnaire principal.

Déclaration du fondateur de Docker concernant son départ

« Aujourd'hui, j'annonce mon départ de Docker, la société que j'ai aidé à créer il y a dix ans et que je construis depuis ».

Docker blog

Public visé

Ce tutoriel est conçu pour les débutants ayant besoin de comprendre la technologie Docker à partir de zéro. Ce tutoriel vous donnera une compréhension suffisante de la technologie, qui vous permettra plus tard d'atteindre des niveaux d'expertise beaucoup plus élevés.

Prérequis

Avant de poursuivre ce cours, vous devez au minimum avoir une compréhension de base sur les commandes Linux. Si vous maîtrisez Linux et la virtualisation, il vous sera très facile de comprendre les concepts de Docker et d'avancer rapidement sur la piste d'apprentissage, mais ce n'est pas non plus indispensable.

Êtes-vous prêt pour de nouvelles découvertes ? alors c'est parti !

Les différences entre la virtualisation et la conteneurisation

Cet article vous explique en détail, les différences entre la virtualisation et la conteneurisation. Vous comprendrez ainsi pourquoi la technologie Docker est devenue si prisée.

Introduction

Pour comprendre pourquoi la technologie Docker est devenue de nos jours si populaire que ça, il est important d'abord de comprendre quel est l'intérêt des conteneurs en le comparant à la virtualisation.

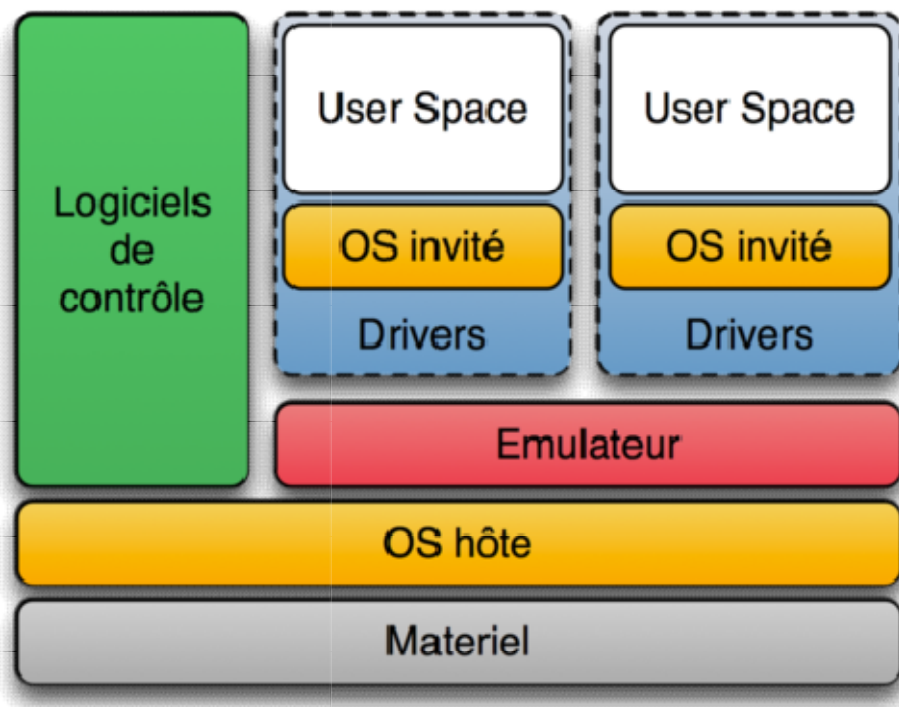
Commençons d'abord par expliquer le fonctionnement de la virtualisation.

La virtualisation

Le fonctionnement de la virtualisation

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.

Ainsi ces machines virtuelles nommées aussi VM (Virtual Machine) bénéficieront de ressources matérielles selon leurs besoins (par exemple plus de puissance processeur et plus de mémoire vive mais avec moins d'espace disque). 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 avantages de virtualisation

- Consacrer les **ressources adaptées** selon les applications qu'on souhaite mettre en place.
- Les machines virtuelles restent simples à manier. Il est possible par exemple de basculer une VM d'un lieu à l'autre voire même de sauvegarder et de dupliquer une VM à volonté sans aucun impact visible pour les utilisateurs.
- La virtualisation **réduit les dépenses** en abaissant le besoin de systèmes matériels physiques. Elle permet ainsi de **réduire la quantité d'équipement** nécessaire et les coûts de maintenance d'alimentation et de refroidissement des composants.
- Les machines virtuelles apportent également une aisance à l'administration car un matériel virtuel n'est pas sujet aux défaillances. Les administrateurs profitent des environnements virtuels pour faciliter les sauvegardes, la **reprise après catastrophe**.

Les inconvénients de la virtualisation

- Le fait d'accéder aux ressources de façon virtuelle affaiblit les performances, cela est dû car on passe par une **couche d'abstraction matérielle** qui malheureusement doit faire des interprétations entre le matériel en place et celui simulé dans la machine virtuelle.
- Comme expliqué plus haut la virtualisation consiste à faire fonctionner sur un seul ordinateur physique plusieurs VMs avec différents systèmes d'exploitation, comme s'ils fonctionnaient sur des ordinateurs distincts. Mais malheureusement cette **couche d'OS** consomme à lui tout seul énormément de ressources alors qu'au final, ce qui nous intéresse c'est là où les applications vont tourner dessus.

Conclusion

On se trouve alors avec une technologie très utile, malléable et économique pour les professionnels, mais malheureusement elle possède aussi son lot d'inconvénients, heureusement que d'autres personnes ont pensé à aller encore plus loin, et d'être encore plus efficace, et pour cela, la conteneurisation a été créée et par la suite la technologie Docker a permis de la populariser.

La conteneurisation vs virtualisation

Information

L'utilisation de conteneurs Linux pour déployer des applications s'appelle 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.

Par contre 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**.

L'image ci-dessous illustre cette phase d'abstraction de l'OS.



Avantages de la conteneurisation par rapport à la virtualisation traditionnelle

- 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 appelle 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 sûr faire la même chose pour une machine virtuelle en la déplaçant entièrement de serveurs en serveurs mais n'oubliez pas qu'il existe 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.

Information

La virtualisation reste tout de même une technologie profitable qui a encore de beaux jours devant elle. Pour rappel nous avons besoin d'une machine avec un OS pour faire tourner nos conteneurs. Je vais me répéter mais pour moi ça reste une notion très importante, la conteneurisation permet d'optimiser l'utilisation de vos ressources tout en profitant des avantages qu'offre la virtualisation.

Les avantages de conteneurisation

La conteneurisation est de plus en plus populaire car les conteneurs sont :

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

Pourquoi Docker est si populaire ?

La conteneurisation est loin d'être une technologie récente. En réalité les conteneurs ne sont pas si nouveaux que ça, comme on pourrait le croire. Je peux en citer quelques technologies comme Chroot sur Unix (1982), Jail sur BSD (2000), conteneurs sur Solaris (2004), [LXC](#) (Linux conteneurs) sur Linux (2008). La célébrité de docker vient du fait qu'il a su permettre aux utilisateurs de gérer facilement leurs conteneurs avec une interface en ligne de commande simple.

Les conteneurs ne sont pas nouveaux, mais leur utilisation pour déployer facilement des applications l'est.

Hors de ce cours pour s'amuser un peu, on aura l'occasion sur d'autres articles de créer un conteneur Linux en utilisant des fonctionnalités nativement disponible sur Linux ☐.

C'est quoi exactement un conteneur ?

Sur ce chapitre, je réponds aux questions suivantes : qu'est-ce que c'est un conteneur ? Comment fonctionne-t-il ? À quoi servent-ils ? Et Docker dans tout ça ?

Introduction

Depuis le début de ce cours on parle de conteneur, mais on parle de qui/quoi exactement ? Qu'est-ce que c'est un conteneur ? Comment fonctionne-t-il ? À quoi servent-ils ? Je vais tenter de répondre à toutes ces questions à travers cet article.

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 .

Information

Docker tire parti de plusieurs ces fonctionnalités proposées par le noyau Linux pour fournir ses fonctionnalités.

Voyons voir plus en détail ces fonctionnalités.

Les namespaces : limiter les vues

Supposons que nous voulons créer une sorte de machine virtuelle. Une des caractéristiques que vous exigerez sera la suivante : "mes processus doivent être séparés des autres processus de l'ordinateur"

Pour réussir à atteindre notre but, on utilisera une fonctionnalité que Linux fournit à savoir les namespaces !

Les namespaces (ou "espaces de noms" en français) isolent les ressources partagées. Ils donnent à chaque processus sa propre vue unique du système, limitant ainsi leur accès aux ressources système sans que le processus en cours ne soit au courant des limitations.

Il existe différents types de namespaces, que je vais vous expliquer sur la liste ci-dessous :

- Le **namespace PID** : fournit un ensemble indépendant d'identifiants de processus (PID). Il s'agit d'une structure hiérarchique dans laquelle le namespace parent peut afficher tous les PID des namespaces enfants. Lorsqu'un nouveau namespace est créé, le premier processus obtient le PID 1 et constitue une sorte de **processus init** de ce namespace. Cela signifie également que si on tue ce processus PID 1 alors on mettra immédiatement fin à tous les processus de son namespace PID et à tous ses descendants.
- Le **namespace IPC** : empêche la communication avec les autres processus, plus simplement il interdit l'échange d'informations avec les autres processus.
- Le **namespace NET** : crée une pile réseau entièrement nouvelle, y compris : un ensemble privé d'adresses IP, sa propre table de routage, liste de socket, table de suivi des connexions, pare-feu et autres ressources liées au réseau.
- Le **namespace MOUNT** : monte un système de fichier propre au processus qui est différent du système de fichier de la machine hôte. Vous pouvez ainsi monter et démonter des systèmes de fichiers sans que cela n'affecte le système de fichiers hôte.
- Le **namespace USER** : fournit à la fois l'isolation des privilèges et la séparation des identifications d'utilisateurs entre plusieurs ensembles. Il permet par exemple de donner un accès root dans le conteneur sans qu'il soit root sur la machine hôte.
- Le **namespace UTS** : associe un nom d'hôte et de domaine au processus pour avoir son propre hostname.

Ce n'est pas vraiment le but de ce cours mais pour s'amuser un peu, on utilisera la commande **unshare** pour créer un namespace PID du programme **bash**.

Juste avant de lancer la commande **unshare**, je vais vous montrer les processus qui tournent déjà sur ma machine hôte :

```
ps aux
```

Copier

Résultat :

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME
root /init	1	0.0	0.0	8324	156	?	Ss	09:18	0:00
root /init	3	0.0	0.0	8328	156	tty1	Ss	09:18	0:00
hatim bash	4	0.0	0.0	16796	3424	tty1	S	09:18	0:00 -
root /init	16	0.0	0.0	8332	160	tty2	Ss	10:08	0:00
hatim bash	17	0.0	0.0	16788	3420	tty2	S	10:08	0:00 -
root /usr/sbin/apache2 -k start	75	1.8	0.1	225388	16196	?	Ss	11:24	0:00
www-data /usr/sbin/apache2 -k start	80	0.0	0.0	225680	2796	?	S	11:24	0:00
www-data /usr/sbin/apache2 -k start	81	0.0	0.0	225680	2796	?	S	11:24	0:00
www-data /usr/sbin/apache2 -k start	82	0.0	0.0	225680	2796	?	S	11:24	0:00
www-data /usr/sbin/apache2 -k start	83	0.0	0.0	225680	2796	?	S	11:24	0:00
www-data /usr/sbin/apache2 -k start	84	0.0	0.0	225680	2796	?	S	11:24	0:00
mysql /bin/sh /usr/bin/mysqld_safe	130	2.0	0.0	10660	800	?	S	11:24	0:00

```
mysql      493  8.4  1.0 1934168 129464 ?      Sl   11:24   0:00
/usr/sbin/mysqld --basedir=/usr --datadir=/var/lib/mysq

hatim      551  0.0  0.0  17380   1924 tty2    R    11:24   0:00 ps
aux
```

Maintenant exécutant notre namespace PID avec la commande unshare :

```
sudo unshare --fork --pid --mount-proc bash
```

Copier

Je vais maintenant afficher les processus en cours au sein de ce mini conteneur :

```
ps aux
```

Copier

Résultat :

USER COMMAND	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME
root bash	1	0.5	0.0	16692	3292	tty2	S	11:27	0:00
root aux	9	0.0	0.0	17380	1920	tty2	R	11:27	0:00 ps

Il n'y a que 2 processus en cours d'exécution bash et ps. Preuve que les namespaces permettent de limiter la vue d'un processus.

En tout bravo vous venez de créer un tout mini conteneur ☐.

Tapez ensuite la commande `exit` pour quitter votre mini conteneur.

Les Cgroups : limiter les ressources

Bon jusqu'ici, nous avons vu comment fonctionnent les namespaces, mais que faire si je veux limiter la quantité de mémoire ou de processeur utilisée par l'un de mes processus ?

Heureusement que des personnes en 2007 ont développé spécialement pour nous les **groupes de contrôle**.

Information

Il existe aussi l'outil nommé nice (et son petit frère renice) permettant de contrôler la priorité des processus sur Linux, sauf que les groupes de contrôle proposent plus de fonctionnalités.

Je vous présente ci-dessous quelques types de cgroup :

- **cgroup cpuset** : assigne des processeurs individuels et des nœuds de mémoire à des groupes de contrôle
- **cgroup cpu** : planifie un accès aux ressources du processeur
- **cgroup cpuacct** : génère des rapports sur les ressources du processeur utilisées
- **cgroup devices** : autorise ou refuse l'accès aux périphériques
- **cgroup net_prio** : permet de définir la priorité du trafic réseau
- **cgroup memory** : définit la limite d'utilisation de la mémoire
- **cgroup blkio** : limite de la vitesse E/S (lecture/écriture) sur périphériques de type bloc (ex : disque dur)
- **cgroup pid** : limite le nombre de processus

Allé pour s'amuser encore un peu plus, créons un cgroup qui limite la mémoire !

Commençons d'abord par créer un cgroup limitant l'utilisation de la mémoire :

```
sudo cgcreate -a <nom_d_utilisateur> -g memory:<nom_du_cgroup>
```

Copier

Voyons ce qu'il y a dedans :

```
ls -l /sys/fs/cgroup/memory/<nom_du_cgroup>/
```

Copier

Résultat :

```
-rw-r--r-- 1 <nom_d_utilisateur> root 0 Okt 10 23:16  
memory.kmem.limit_in_bytes  
  
-rw-r--r-- 1 <nom_d_utilisateur> root 0 Okt 10 23:14  
memory.kmem.max_usage_in_bytes
```

Ensuite, on va limiter notre cgroup à 20 mégaoctets :

```
sudo echo 20000000 > /sys/fs/cgroup/memory/<nom_du_cgroup>/memory.kmem.limit_in_bytes
```

Copier

Maintenant utilisons notre cgroup sur notre programme bash :

```
sudo cgexec -g memory:<nom_du_cgroup> bash
```

Copier

Voilà le processus bash ne peut plus dépasser 20 Mo de mémoire.

Conclusion

Pour résumer, la technologie Docker possède de nombreuses fonctionnalités de nos jours, mais beaucoup d'entre elles reposent sur les fonctionnalités de base du noyau Linux vues plus haut.

Pour rentrer plus dans les détails, les conteneurs contiennent généralement un ou plusieurs programme(s) de manière à les maintenir isolées du système hôte sur lequel elles s'exécutent. Ils permettent à un développeur de conditionner une application avec toutes ses dépendances, et de l'expédier dans un package unique.

En outre, ils sont conçus pour faciliter la mise en place d'une expérience cohérente lorsque les développeurs et les administrateurs système déplacent le code des environnements de développement vers la production de manière rapide et reproductible.

Découverte et installation de Docker

Dans cet article, je vais vous présenter les différentes éditions et versions de docker et par la suite nous procéderons à l'installation et post installation de Docker.

Introduction

Cette partie du cours va être très rapide mais aussi très importante. Vous allez connaître les différentes versions et éditions existantes de Docker. Une fois qu'on aura fini les présentations, on passera ensuite directement à l'étape d'installation et de vérification d'install.

Vue d'ensemble des éditions et des versions de Docker

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).

La version communautaire Docker dispose de trois types de canaux de mise à jour, **stable**, **test** et **Nightly** :

- **Stable** : cette version vous donne les dernières releases pour une disponibilité générale
- **Test** : cette version vous fournit des pré-versions prêtes à être testées avant la disponibilité générale
- **Nightly** : cette version vous présente les dernières versions de build en cours pour la prochaine release, elle fournit donc un accès plus rapide aux nouvelles fonctionnalités et correctifs pour les tests.

Pour ce cours, nous utiliserons la version communautaire de Docker

Installation Linux

Dans cette partie je vais **installer le moteur Docker** sur mon OS **Fedora version 29** avec la version 18.09.6 du moteur Docker. Si vous avez une autre distribution, je vous conseille alors de vous référer directement à la documentation qui vous expliquera pas à pas quelles sont les commandes à lancer pour votre installation.

On va commencer par ajouter le repository officiel de Docker sur notre machine. Cette étape va nous permettre de profiter des dernières mises à jour stable directement depuis le repository Docker.

voici la commande à lancer :

```
sudo dnf config-manager \
    --add-repo \
    https://download.docker.com/linux/fedora/docker-ce.repoCopier
```

Dans mon cas je n'utiliserai que la version stable de docker mais si vous souhaitez activer le repository et les mises à jour des versions Nightly et Test depuis le repository de Docker, alors il faut lancer les commandes suivantes :

Pour la version **Nightly** :

```
sudo dnf config-manager --set-enabled docker-ce-nightlyCopier
```

Pour la version **Test** :

```
sudo dnf config-manager --set-enabled docker-ce-testCopier
```

Information

Si vous souhaitez désactiver la version Nightly ou Test, il suffit de remplacer `--set-enabled` par `--set-disabled`.

Une fois le repository Docker ajouté, l'étape suivante est l'installation de Docker :

```
sudo dnf install docker-ce docker-ce-cli containerd.ioCopier
```

Une fois votre installation finie, l'étape suivante est de **vérifier que Docker CE est correctement installé** en vérifiant d'abord la version du moteur :

```
sudo docker --versionCopier
```

Résultat:

```
Docker version 18.09.6, build 481bc77
```

Maintenant il faut **activer le service docker** en tapant la commande suivante :

```
sudo systemctl start dockerCopier
```

Pour activer le service docker automatiquement après un boot, il suffit de lancer la commande suivante :

```
sudo systemctl enable dockerCopier
```

Pour finir nos tests de vérifications d'install, on va lancer la commande suivante :

```
sudo docker run hello-worldCopier
```

Cette commande va télécharger une image (vous comprendrez ce terme plus tard) de test et l'exécute dans un conteneur. Lorsque le conteneur va s'exécuter, il vous affichera un message d'information et se fermera automatiquement

Résultat:

```
Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

1b930d010525: Pull complete

Digest:
sha256:0e11c388b664df8a27a901dce21eb89f11d8292f7fca1b3e3c4321bf7897b
ffe

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/>

Bonus Linux

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`.

En ce qui me concerne et pour la suite de ce cours, je vais **autoriser mon compte utilisateur Unix à lancer les commandes docker sans passer par le `sudo`**. Donc à partir de maintenant je ne vais plus utiliser le `sudo` dans mes exemples, sauf si obligation.

Cette manipulation est assez simple à faire, il suffit de créer un **groupe Unix appelé docker** et de lui ajouter notre utilisateur.

On va commencer par créer notre groupe Unix avec la commande suivante :

```
sudo groupadd dockerCopier
```

Puis on rajoute notre utilisateur au groupe docker

```
sudo usermod -aG docker $USERCopier
```

Ensuite, déconnectez-vous et reconnectez-vous (si ça ne fonctionne pas alors il faut redémarrer complètement votre machine) pour que votre appartenance au groupe soit réévaluée.

Enfin, vérifiez que vous pouvez exécuter la commande docker suivante sans sudo :

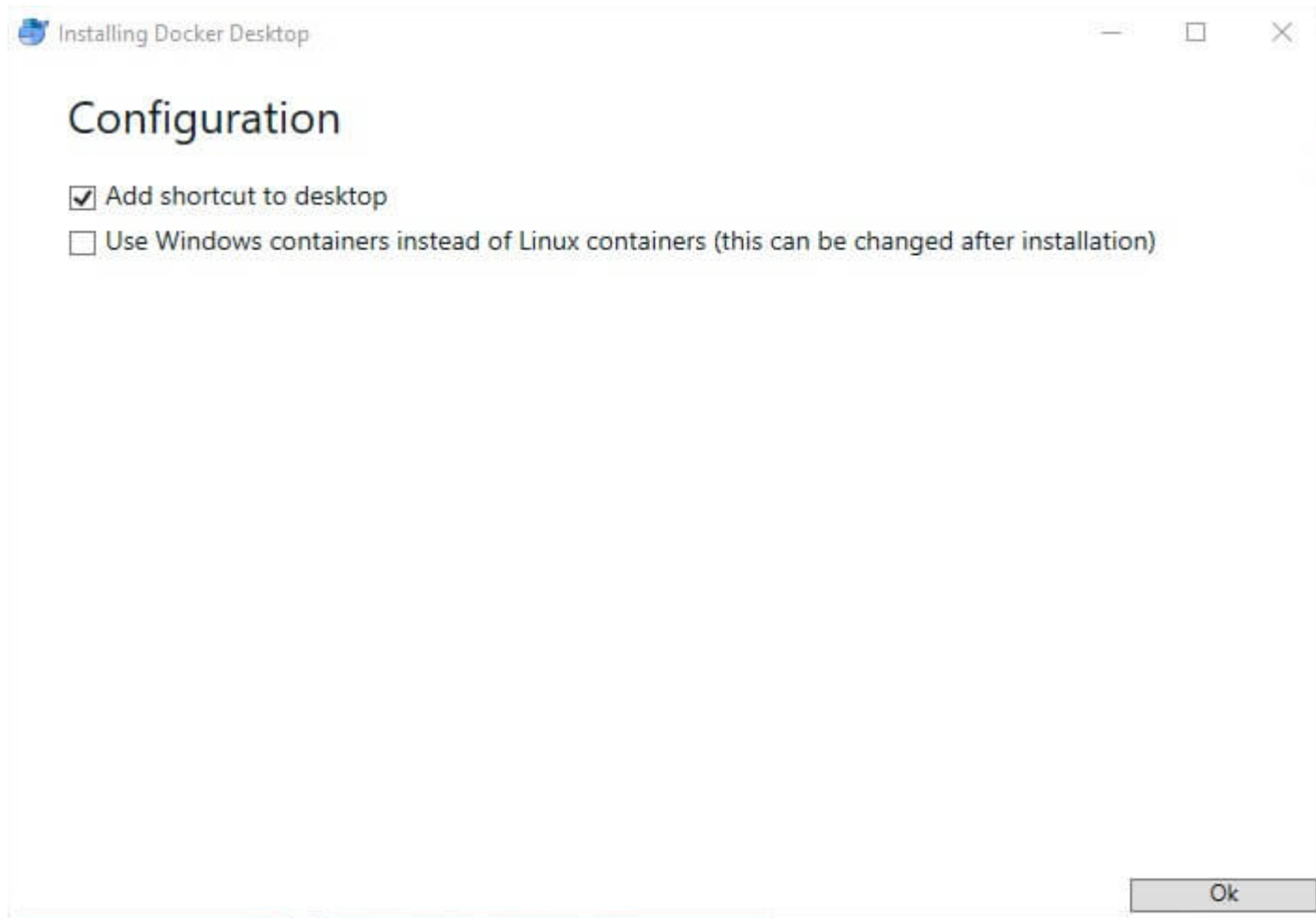
```
docker psCopier
```

Installation Windows

Voici la **configuration requise pour une installation windows native** :

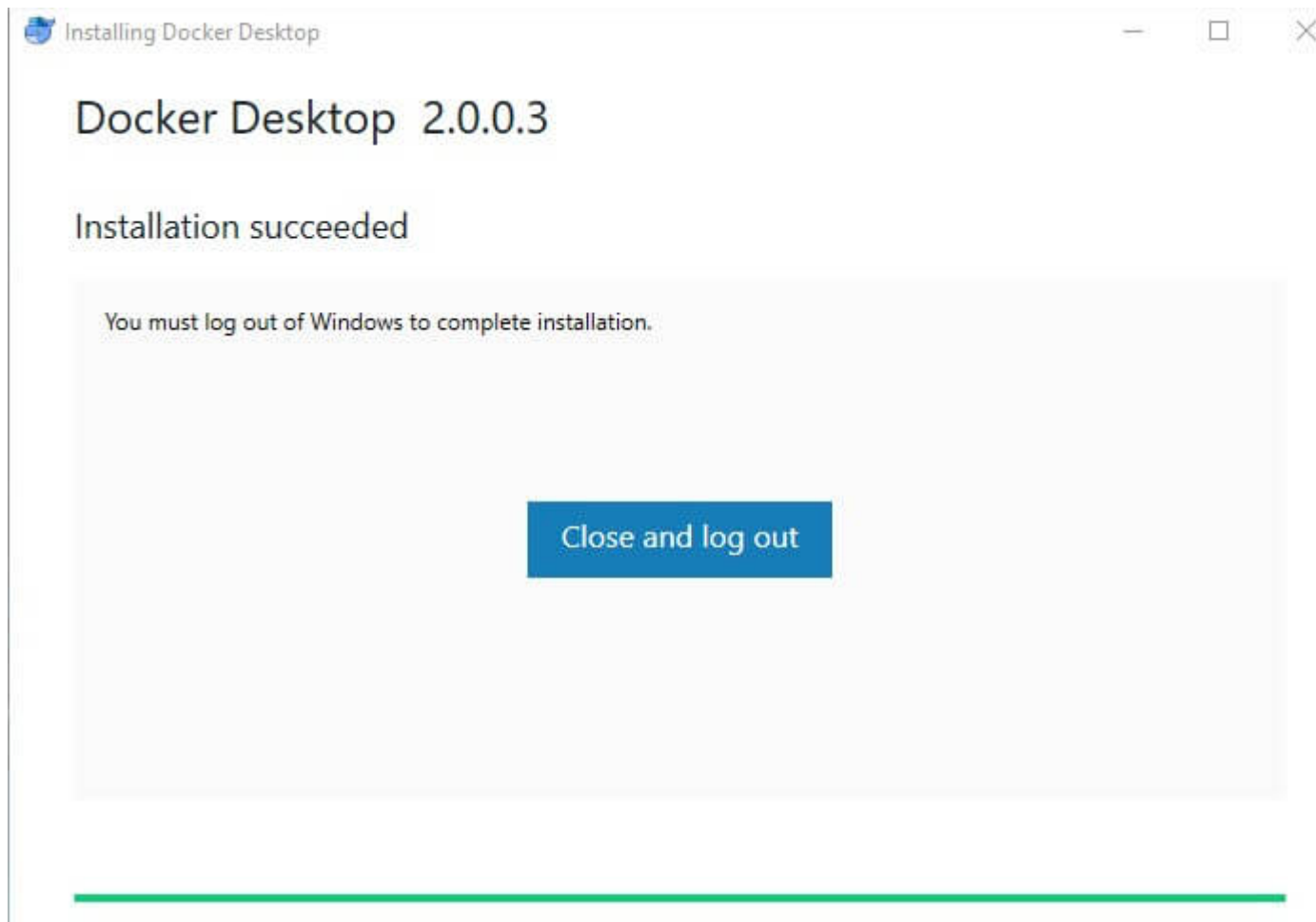
- Windows 10 64 bits: Pro, Entreprise ou Education (version 15063 ou ultérieure).
- La virtualisation est activée dans le BIOS (normalement elle est activée par défaut).
- Au moins 4 Go de RAM

Une fois les conditions satisfaites, rendez-vous et inscrivez sur le Docker Hub en cliquant [ici](#). Une fois inscrit et connecté, cliquez sur le bouton "Get Docker" pour télécharger l'exécutable Docker CE et suivez les instructions ci-dessous :



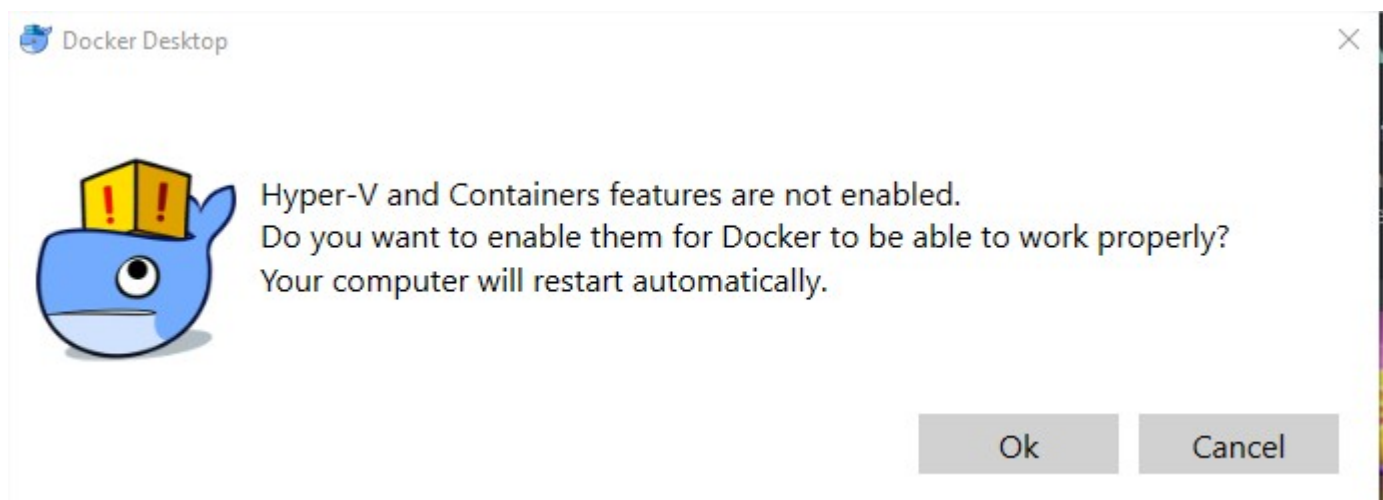
Cliquez sur le bouton "OK".

Une fois votre installation terminée, vous recevrez le message suivant :



Cliquez sur "Close and log out". Vous allez ensuite être automatiquement déconnecté de votre session Windows.

Si vous n'avez pas activé Hyper-V, alors Docker s'en chargera en l'activant automatiquement pour vous :

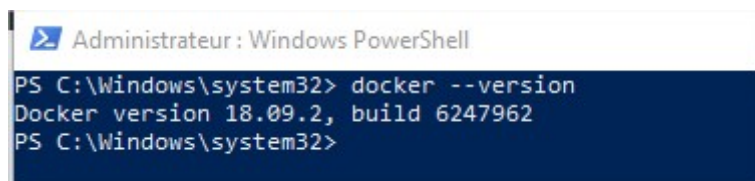


Cliquez sur "OK" pour activer Hyper-V. Par la suite votre machine va automatiquement se redémarrer à la fin de l'activation d'Hyper-V.

Après votre redémarrage, Vous verrez la fenêtre suivante indiquant que le moteur Docker est bien installé :



Dernière étape, lancez votre powershell en tant qu'administrateur et exécutez la commande suivante afin de vérifier que votre Docker CE c'est correctement installé :



Si votre système ne remplit pas les conditions requises vous pouvez alors installer **Docker Toolbox**, qui utilise Oracle Virtual Box au lieu de Hyper-V en cliquant [ici](#).

Fonctionnement et manipulation des images Docker

Aujourd'hui, vous allez apprendre le fonctionnement et la manipulation des images Docker. Vous saurez ainsi comment lister, télécharger, supprimer et rechercher des images Docker. Enfin vous allez découvrir quelques commandes pour récolter des informations sur votre installation Docker.

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**

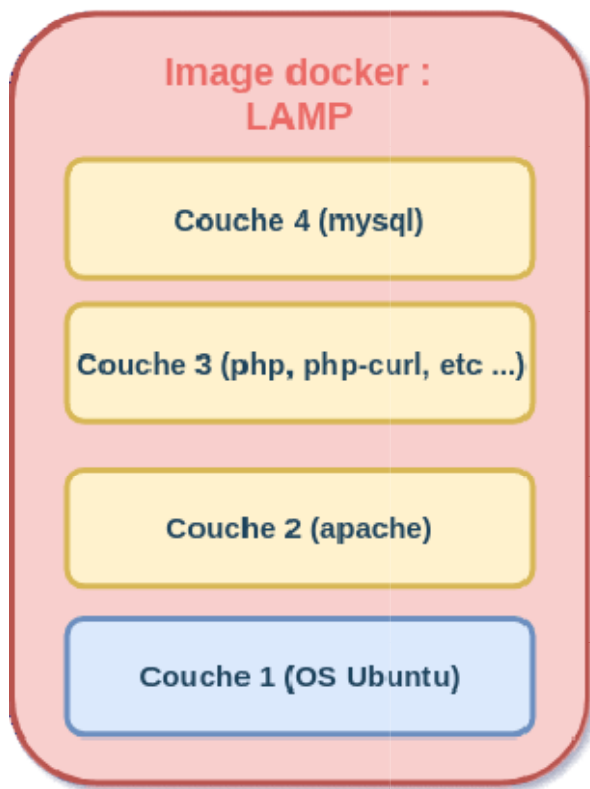
Dans le chapitre suivant vous allez en savoir plus sur les conteneurs. Pour le moment tout ce que vous devez retenir, c'est qu'une image docker est créée à partir d'un fichier nommé le **Dockerfile**. Une image est un modèle **composé de plusieurs couches**, ces couches contiennent notre application 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, Sur le chapitre précédent nous avons téléchargé et exécuté l'image "hello-world", je vais m'appuyer au début sur cette image pour vous dévoiler quelques **commandes de manipulation d'images Docker** et par la suite nous téléchargerons d'autres images.

Quelques commandes

Récupérer des informations

Pour commencer on va d'abord récupérer la liste des commandes possible :

```
docker helpCopier
```

Résultat:

```
Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:

  -v, --version          Print version information and quit
  ...

Management Commands:

  builder      Manage builds
  ...

Commands:

  ...

  info          Display system-wide information
  ...

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

Je n'ai pas listé toutes les commandes car il y en a beaucoup, mais sur votre terminal, vous verrez beaucoup plus d'options que moi.

Sur l'output de la commande help, nous avons une information super intéressante et croyez-moi elle vous sera d'une grande utilité et vous permettra de gagner beaucoup de temps ☐. Je parle de la ligne suivante : `Run 'docker COMMAND --help' for more information on a command.` Cette ligne nous informe qu'il est possible d'avoir de l'**auto-complétion** sur n'importe quelles

sous-commandes/options de Docker, je m'explique, si vous tapez la commande suivante (avec un espace à la fin et sans la lancer) :

```
docker volumes
```

Si à ce moment vous appuyez sur la touche tabulation, alors ça vous affichera toutes les options possibles pour la sous-commande `volumes`, ainsi que leurs descriptions.

Résultat:

```
create  -- Create a volume
inspect -- Display detailed information on one or more volumes
ls      -- List volumes
prune   -- Remove all unused volumes
rm      -- Remove one or more volumes
```

Nice non ☐ ?

Sur le chapitre précédent, nous avons vu comment afficher la version du moteur Docker avec la commande suivante :

```
docker --version
```

Résultat:

```
Docker version 18.09.6, build 481bc77
```

C'est cool d'avoir une telle information, mais il existe une autre commande qui permet d'afficher encore plus de détails sur votre installation de Docker:

```
docker info
```

Cette commande, nous fournit plusieurs informations concernant les spécifications du moteur Docker. Elle nous transmet aussi d'autres informations telles que le nombre total de conteneurs tournant sur notre machine ainsi que leur état :

```
Containers: 3
```

```
Running: 0
```

```
Paused: 0
```

```
Stopped: 3
```

Voir le nombre total d'images disponibles sur notre machine :

```
Images: 1
```

Hmm, par contre ça ne m'affiche que le nombre d'images disponibles sur ma machine, "Est-ce que c'est possible d'avoir davantage d'informations sur mes images ?" La réponse est oui ! On va voir ça tout de suite ☐.

Lister les images Docker téléchargées

Sur le chapitre précédent vous aviez lancé la commande `docker run hello-world`. Pour information cette commande télécharge une image depuis le [Docker Hub Registry](#) et l'exécute dans un conteneur. Lorsque le conteneur s'exécute, il vous affiche un petit message d'information et se ferme directement.

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

```
docker image lsCopier
```

ou bien la commande :

```
docker imagesCopier
```

Résultat:

REPOSITORY SIZE	TAG	IMAGE ID	CREATED
hello-world ago 1.84kB	latest	fce289e99eb9	5 months

Elle nous donne différentes informations dont :

REPOSITORY	TAG	IMAGE ID	C
Le titre REPOSITORY peut porter à confusion, c'est essentiellement le nom de l'image.	un tag ici est une façon de faire référence à votre image, ils sont utilisés principalement pour affecter une version à une image	L'identifiant de l'image (unique pour chaque image téléchargée)	Date de modification de l'image

Supprimer une image Docker

Maintenant si on souhaite supprimer une image Docker, on aura besoin soit de son IMAGE ID soit de son nom. Une fois qu'on aura récupéré ces informations, on peut passer à la suite en lançant la commande suivante :

avec l'id de l'image :

```
docker rmi fce289e99eb9Copier
```

avec le nom de l'image :

```
docker rmi hello-worldCopier
```

Si vous lancez cette commande vous aurez le message d'erreur suivant :

```
Error response from daemon: conflict: unable to remove repository
reference "hello-world" (must force) - container 3e444920f82d is using its referenced image
fce289e99eb9
```

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**.

```
docker rmi -f hello-worldCopier
```

Information

Si jamais vous possédez plusieurs images avec le même nom mais avec des tags différents alors vous devez préciser le tag dans votre commande `rmi`, pour ainsi être sûr de supprimer la bonne image. Pour information par défaut Docker prend le tag `latest`.

Bonus

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

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

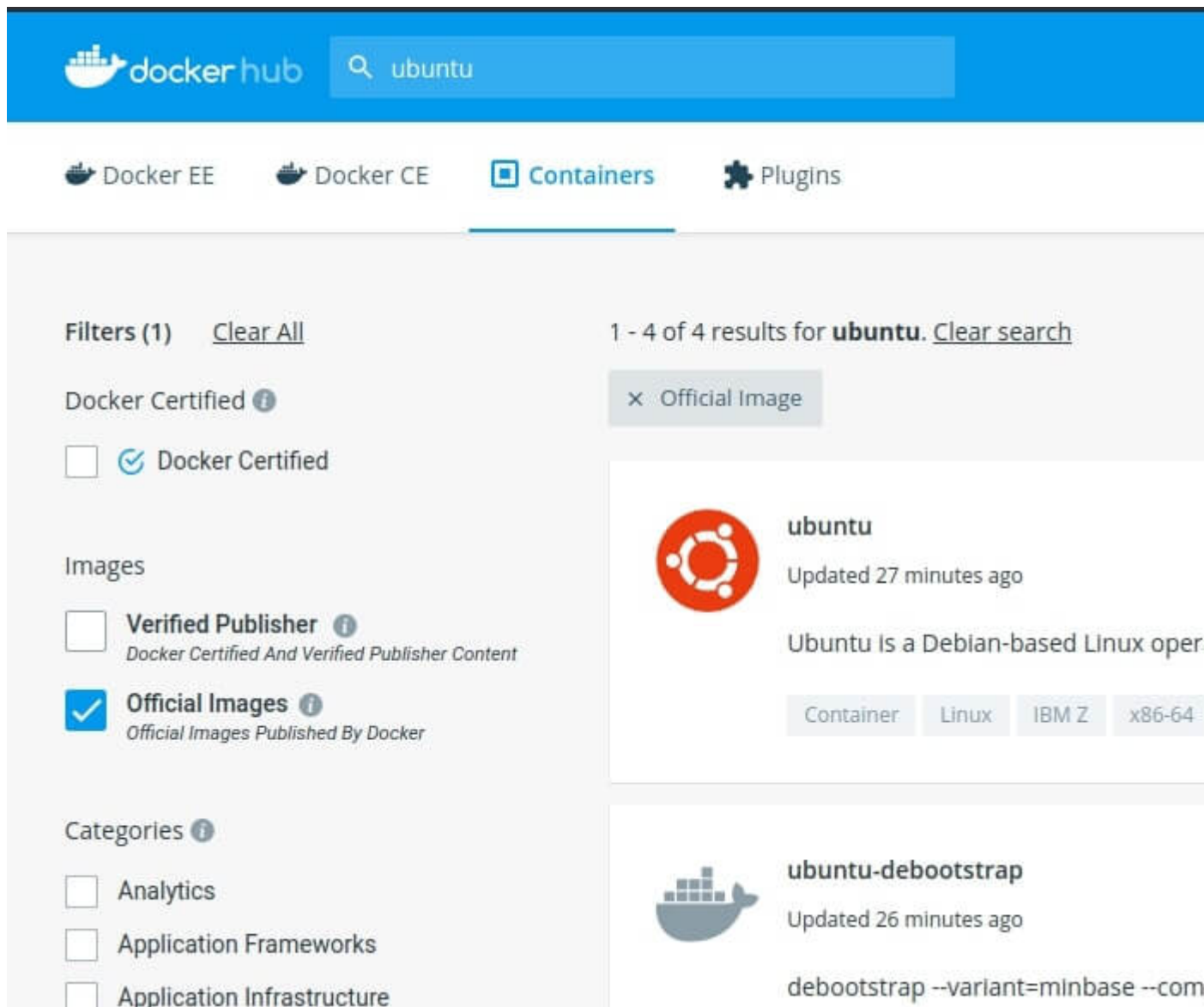
Télécharger une image depuis le Docker Hub Registry

Lister les images disponibles sur 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 où est-ce que je peux retrouver la liste des images disponibles ?"

Très bonne question au tout début je vous ai parlé du [Docker Hub Registry](#). 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 :



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 créée 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.

Je ne dis pas non plus qu'il ne faut télécharger que des images officielles mais juste faire attention à ce qu'on télécharge sur le net. Car d'un autre côté il ne faut pas oublier qu'il existe dans le Docker Hub Registry une multitude d'images créées par des utilisateurs expérimentés indépendants. Ces images sont parfois bien plus optimisées que les images officielles car

certaines images n'utilisent que le strict minimum pour faire fonctionner leur application permettant ainsi de réduire la taille de l'image.

Bref revenons à nos moutons, si je clique sur l'image officielle d'ubuntu je tombe sur la page suivante :



ubuntu ☆

Docker Official Images

Ubuntu is a Debian-based Linux operating system based on free software.

10M+

Container

Linux

386

PowerPC 64 LE

IBM Z

x86-64

ARM

ARM 64

Base Images

Operating Systems

Official Image

DESCRIPTION

REVIEWS

TAGS

Supported tags and respective Dockerfile links

- 18.04, bionic-20190515, bionic, latest (*bionic/Dockerfile*)
- 18.10, cosmic-20190515, cosmic (*cosmic/Dockerfile*)
- 19.04, disco-20190515, disco, rolling (*disco/Dockerfile*)
- 19.10, eoan-20190508, eoan, devel (*eoan/Dockerfile*)
- 14.04, trusty-20190515, trusty (*trusty/Dockerfile*)
- 16.04, xenial-20190515, xenial (*xenial/Dockerfile*)

Quick reference

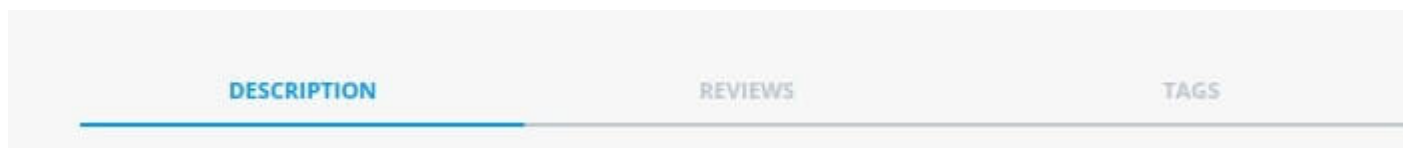
- **Where to get help:**
the Docker Community Forums, the Docker Community Slack, or Stack Overflow
- **Where to file issues:**
the cloud-images bug tracker (include the `docker` tag)
- **Maintained by:**
Canonical and Tianon (Debian Developer)
- **Supported architectures:** (more info)
amd64, arm32v7, arm64v8, i386, ppc64le, s390x

en haut à droite on retrouve le nom de l'image avec une toute petite description



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



Enfin en haut à droit nous avons la commande à lancer permettant de télécharger l'image sur.



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

```
docker search ubuntuCopier
```

Résultat:

NAME	DESCRIPTION	STARS
OFFICIAL	AUTOMATED	
ubuntu	a Debian-based Linux operating sys...	9616
...		
pivotaldata/ubuntu-gpdb-dev	images for GPDB development	0
		Ubuntu is [OK]
		Ubuntu

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" ubuntuCopier
```

Vous aurez ainsi beaucoup moins de résultats.

Télécharger une images depuis le Docker Hub Registry

Je pense que 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 ubuntuCopier
```

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

```
docker pull ubuntu:16.04Copier
```

Je vais faire un petit coup de `docker images` pour si mon image s'est bien téléchargée.

REPOSITORY	TAG	IMAGE ID	CREATED
SIZE			
ubuntu	16.04	2a697363a870	3 weeks
ago 119MB			
ubuntu	latest	7698f282e524	3 weeks
ago 69.9MB			

hello-world	latest	fce289e99eb9	5 months
ago	1.84kB		

Youpi !

Conclusion

Vous savez maintenant **lister, télécharger, supprimer et rechercher une image Docker**, je vous mets en bas un **aide-mémoire** des commandes vu ensemble :

```
## 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 défaut le tag latest

    -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
```

Fonctionnement et manipulation des conteneurs Docker

Aujourd'hui, vous allez apprendre le fonctionnement et la manipulation des conteneurs Docker. Vous saurez ainsi comment créer, lister, lancer, supprimer, transformer, déboguer vos conteneurs.

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).

Quelques commandes

Créer un conteneur

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

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

Nous allons pour le moment créer un conteneur basé sur l'image **hello-world**, et pour faire les choses dans les règles de l'art, nous allons d'abord commencer par télécharger notre image depuis le **Docker Hub Registry**, et ensuite on va exécuter notre image afin de créer notre conteneur.

Étape 1 : Téléchargement de l'image hello-world :

```
docker pull hello-world:latestCopier
```

Étape 2 : Instanciation de l'image hello-world :

```
docker run hello-world:latestCopier
```

Avouons-le, cette image 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.

Normalement si on suit la logique vu précédemment, on devrait exécuter les commandes suivantes :

Étape 1 : Téléchargement de l'image ubuntu :

```
docker pull ubuntu:latestCopier
```

Étape 2 : Instanciation de l'image ubuntu :

```
docker run ubuntu:latestCopier
```

Vous : "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 -ti ubuntu:latestCopier
```

héhé, vous êtes maintenant à l'intérieur de votre conteneur ubuntu avec un jolie shell ☐.

Profitons-en pour télécharger l'outil git, mais juste avant n'oubliez pas de récupérer les listes de paquets depuis les sources afin que votre conteneur sache quels sont les paquets

disponibles en utilisant la commande `apt-get update`, ce qui nous donnera la commande suivante :

```
apt-get update -y && apt-get install -y gitCopier
```

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

```
git --versionCopier
```

Résultat :

```
git version 2.17.1
```

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 /*`. Si vous souhaitez faire comme moi, alors **ASSUREZ-VOUS BIEN AVANT QUE VOUS ETES DANS UN CONTENEUR**. Après avoir lancé cette commande je peux vous affirmer d'ores et déjà dire que j'ai bien détruit mon conteneur, preuve à l'appui je ne peux même plus lancer la commande `ls` :

```
root@7cfb553ebcc2:/# ls
bash: /bin/ls: No such file or directory
```

Vous : "Mais tu es fou de lancer cette commande destructrice !"

Ne vous inquiétez pas, je vais réparer mon conteneur en même pas 1 seconde. Je vais d'abord commencer par **quitter mon conteneur** avec la commande `exit` et ensuite je vais demander à mon moteur Docker la phrase suivante : "abra cadabra Docker crée-moi un nouveau conteneur" :

```
docker run -ti ubuntu:latestCopier
```

Maintenant si je relance ma commande `ls` :

```
root@7cfb553ebcc2:/# ls
bin  etc  lib64  opt  run  sys  var
```

```
boot  home  media  proc  sbin  tmp
dev   lib   mnt    root  srv   usry
```

Tout est en ordre chef ! Mon conteneur est de nouveau fonctionnel et devinez quoi cette réparation ne m'a même pas pris 1 seconde à faire (oui je tape vite au clavier ☐). Cette puissance de pouvoir **créer rapidement des conteneurs** à la volée avec une latence quasi inexistante est possible, car nous avons préalablement téléchargé notre image Ubuntu sur notre machine locale. Et comme notre image contient déjà son **propre environnement d'exécution**, alors elle est déjà prête à l'emploi et elle n'attend plus qu'à être exécutée !

Par contre dans mon nouveau conteneur, si je lance la commande `git --version`, alors je vais obtenir une belle erreur m'indiquant que git est inexistant dans mon conteneur.

```
bash: git: command not found
```

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

Vous : "Quoi ? Il existe bien un moyen pour sauvegarder mes données ??? Dit moi oui please please ???"

Bien sûr que oui, sinon les conteneurs ne serviraient pas à grand-chose. Il existe deux façons pour **stocker les données d'un conteneur**, soit on transforme notre conteneur en image (cependant ça reste une méthode non recommandée, mais je vous montrerai à la fin de ce chapitre comment l'utiliser), soit on utilise le système de volume, nous verrons cette notion dans un autre chapitre dédié aux volumes.

Pour le moment je souhaite 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**. Bon, maintenant vous connaissez la routine, on télécharge l'image et on l'exécute, mais juste avant j'aimerais bien vous montrer 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**

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 pull httpd:latestCopier
```

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

Une fois votre image exécutée, 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 :

```
hatim@localhost ~ » docker run --name monServeurWeb -p 8080:80 httpd:latest
AH00558: httpd: Could not reliably determine the server's fully qualified domain name,
verName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name,
verName' directive globally to suppress this message
[Wed Jun 12 20:14:30.194632 2019] [mpm_event:notice] [pid 1:tid 140201429643328] AH0048
ured -- resuming normal operations
[Wed Jun 12 20:14:30.194815 2019] [core:notice] [pid 1:tid 140201429643328] AH00094: Co
ND'
172.17.0.1 - - [12/Jun/2019:20:14:33 +0000] "GET / HTTP/1.1" 304 -
172.17.0.1 - - [12/Jun/2019:20:14:34 +0000] "GET / HTTP/1.1" 304 -
172.17.0.1 - - [12/Jun/2019:20:14:34 +0000] "GET / HTTP/1.1" 304 -
172.17.0.1 - - [12/Jun/2019:20:14:34 +0000] "GET / HTTP/1.1" 304 -
172.17.0.1 - - [12/Jun/2019:20:14:34 +0000] "GET / HTTP/1.1" 304 -
172.17.0.1 - - [12/Jun/2019:20:14:35 +0000] "GET / HTTP/1.1" 304 -
172.17.0.1 - - [12/Jun/2019:20:14:35 +0000] "GET / HTTP/1.1" 304 -
█
```

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:latestCopier
```

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

```
docker: Error response from daemon: Conflict.
```

```
The container name "/monServeurWeb" is already in use by container
"832d83bf810a28a68ef5406743b6d604e0a1717da6bad541bd61ab83e172f6ff".
```

```
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 (ici "monServeurWeb"). Dans ce cas vous avez deux solutions, soit vous choisissez la facilité en renommant votre conteneur, mais ce n'est pas vraiment très propre comme solution, en plus moi, ce que je souhaite c'est de garder le nom "monServeurWeb" pour mon 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**.

Afficher la liste des conteneurs

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 container lsCopier
```

ou

```
docker psCopier
```

Résultat :

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
55e5ccfc1357	httpd:latest	"httpd-foreground"	19
seconds ago	Up 16 seconds	0.0.0.0:8080->80/tcp	
monServeurWeb			

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`)
- **exited** : conteneur qui a été exécuté puis terminé
- **dead** : conteneur que le service docker n'a pas réussi à arrêter correctement (généralement en raison d'un périphérique occupé ou d'une ressource utilisée par le conteneur)
- **PORTS** : les ports utilisés par votre conteneur
- **NAMES** : nom de votre conteneur

Par défaut la commande `docker ps` ou `docker container ls` 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**.

Supprimer un conteneur

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>Copier
```

Soit :

```
docker rm monServeurWebCopier
```

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:latestCopier
```

Vous ne voyez plus les logs d'apache et votre conteneur tourne en arrière-plan, preuve à l'appui, avec la commande `docker ps` :

CONTAINER ID STATUS	IMAGE PORTS	COMMAND NAMES	CREATED
f3f971625432 minutes ago monServeurWeb	httpd:latest Up 7 minutes	"httpd-foreground" 0.0.0.0:8080->80/tcp	7

Exécuter une commande dans un conteneur

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'utiliser pour récupérer notre interpréteur de commande `/bin/bash`, ce qui aura pour but de se connecter directement à notre conteneur Apache.

```
docker exec -ti monServeurWeb /bin/bashCopier
```

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.htmlCopier
```

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

Information

Pour quitter votre conteneur sans le détruire, utilisez le raccourci suivant : **Ctrl + P + Q**

Afficher les logs d'un conteneur

Dès fois, vous aurez besoin de déboguer votre conteneur en regardant les **sorties/erreurs d'un conteneur**.

Il existe pour cela la commande `docker logs` qui vient avec deux options très utiles :

- **-f** : suivre en permanence les logs du conteneur (correspond à `tail -f`)
- **-t** : afficher la date et l'heure de réception des logs d'un conteneur

```
docker logs -ft monServeurWebCopier
```

si vous visitez votre page, alors vous verrez des logs s'afficher successivement sur votre terminal (**Ctrl + C** pour quitter vos logs)

Transformer votre conteneur en image

Comme promis voici, la commande qui permet de **transformer un conteneur en image**, afin de stocker nos données

Attention

Je me répète mais c'est important, cette commande n'est pas vraiment recommandée, pour stocker vos données. Il faut pour ça utiliser les volumes, que nous verrons dans un autre chapitre.

Voici les étapes que nous allons suivre :

- Exécuter notre conteneur basé sur l'image officielle Ubuntu
- Installer l'outil git
- Mettre du texte dans un nouveau fichier
- Transformer notre conteneur en image
- Relancer notre un nouveau conteneur basé sur cette nouvelle image

```
docker run -ti --name monUbuntu ubuntu:latestCopier  
  
apt-get update -y && apt-get install -y gitCopier  
  
echo "ceci est un fichier qui contient des donnees de test" >  
test.txt && cat test.txtCopier
```

Ctrl + P + Q

Maintenant, c'est l'étape où je vais créer mon image depuis mon nouveau conteneur. Voici son prototype :

```
docker commit <CONTAINER NAME or ID> <NEW IMAGENAME>Copier
```

Ce qui nous donnera :

```
docker commit monUbuntu ubuntu:gitCopier
```

Information

Vous pouvez voir, votre nouvelle image avec la commande `docker images`

Voilà, maintenant, on va lancer notre conteneur basé sur cette nouvelle image :

```
docker run -ti --name ubuntu:git_container ubuntu:gitCopier
```

À présent, je vais vérifier si les données ont bien été stockées sur ce nouveau conteneur.

```
cat test.txtCopier
```

Résultat :

```
ceci est un fichier qui contient des donnees de test
```

```
git --version
```

Résultat :

```
git version 2.17.1
```

Notre outil git et notre fichier sont bien présents dans notre nouveau conteneur ☐.

Conclusion

Vous avez appris, à manipuler des conteneurs docker. Dans le futur chapitre, nous allons apprendre à créer notre propre image Docker avec le fameux **Dockerfile**. Comme pour chaque fin de chapitre, je vous mets un **cheat sheet** reprenant les commandes qu'on a pu voir dans ce chapitre.

```
## 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)
```

```
## 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 commitCopier
```

Créer ses propres images Docker avec le Dockerfile

Comprendre les différentes instructions du Dockerfile et apprendre à créer vos propres images Docker avec le Dockerfile, et pusher vos images vers le Hub Docker.

Introduction

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.

Voici le schéma de notre image :



Les différentes instructions du Dockerfile

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.
- **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.
- **EXPOSE** : Expose un port.
- **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).

Je pense, que vous avez sûrement quelques interrogations pour savoir quand est-ce-qu'il faut utiliser telle ou telle instruction. Ne vous inquiétez car à la fin de ce chapitre, je vais rédiger une FAQ, pour répondre à quelques une de vos interrogations.

Création de notre image

Normalement pour faire les choses dans les règles de l'art, il faut séparer l'image de l'application web par rapport à l'image de la base de données. Mais pour ce cours je vais faire une exception et je vais mettre toute notre stack dans une seule image.

Création des sources et du Dockerfile

Commencez par créer un dossier et téléchargez les sources de l'image, en cliquant [ici](#).

Désarchivez le fichier zip, et mettez les dossiers suivants dans votre nouveau dossier :

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

Création des sources et du Dockerfile

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

```
# ----- DÉBUT COUCHE OS -----
```

```
FROM debian:stable-slim

# ----- FIN COUCHE OS -----

# MÉTADONNÉES DE L'IMAGE

LABEL version="1.0" maintainer="AJDAINI Hatim
<ajdaini.hatim@gmail.com>"

# 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 && mariadb < /articles.sql &&
apache2ctl -D FOREGROUND
```

Voici l'architecture que vous êtes censé avoir :

```
├─ app
|   └─ db-config.php
|       └─ index.php
├─ db
|   └─ articles.sql
└─ Dockerfile
```

Explication du Dockerfile

```
FROM debian:stable-slim
```

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.

```
LABEL version="1.0" maintainer="AJDAINI Hatim  
<ajdaini.hatim@gmail.com>"Copier
```

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>`.

```
ARG APT_FLAGS="-q -y"  
  
ARG DOCUMENTROOT="/var/www/html"Copier
```

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

```
RUN apt-get update -y && \  
apt-get install ${APT_FLAGS} apache2Copier
```

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.

```
RUN apt-get install ${APT_FLAGS} mariadb-server  
  
COPY db/articles.sql /Copier
```

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

```
RUN apt-get install ${APT_FLAGS} \  
mysql
```

```
php-mysql \  
  
php && \  
  
rm -f ${DOCUMENTROOT}/index.html && \  
  
apt-get autoclean -y
```

```
COPY app ${DOCUMENTROOT}Copier
```

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.

```
EXPOSE 80Copier
```

J'ouvre le port HTTP.

```
WORKDIR /var/www/htmlCopier
```

Comme je suis un bon flemmard d'informaticien ☹, j'ai mis 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.

```
ENTRYPOINT service mariadb start && mariadb < /articles.sql &&  
apache2ctl -D FOREGROUNDCopier
```

Ici, lors du lancement de mon conteneur, le service mysql 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`.

Construction et Execution de notre image

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

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

Ce qui nous donnera :

```
docker build -t my_lamp .Copier
```

Ensuite, exécutez votre image personnalisée :

```
docker run -d --name my_lamp_c -p 8080:80 my_lampCopier
```

Visitez ensuite la page suivante <http://localhost:8080/>, et vous obtiendrez le résultat suivant :

Mes articles

Qu'est-ce que le Lorem Ipsum ?

01/07/19 18:44

Le Lorem Ipsum est simplement du faux texte employé dans le standard de l'imprimerie depuis les années 1500, quand un imprimeur utilisait un spécimen de polices de texte. Il n'a pas fait que survivre cinq siècles, mais a aussi été modifié. Il a été popularisé dans les années 1960 grâce à la technologie, et récemment, par son inclusion dans des applications de mise en page.

— auteur 1

Pourquoi l'utiliser?

01/07/19 18:44

On sait depuis longtemps que travailler avec du texte lisible et compréhensible en page elle-même. L'avantage du Lorem Ipsum sur un texte générique est que les lettres plus ou moins normale, et en tout cas comparable avec du texte réel. Les éditeurs de sites Web ont fait du Lorem Ipsum leur faux texte par défaut, et ceux qui n'en sont encore qu'à leur phase de construction. Plusieurs personnes ont (histoire d'y rajouter de petits clins d'oeil, voire des phrases entières).

— auteur 2

Bravo ! vous venez de créer votre propre image Docker ☐!

FAQ Dockerfile

Promesse faite, promesse tenue. Je vais tenter de répondre à quelques questions concernant certaines instructions du Dockerfile.

Quelle est la différence entre ENV et ARG dans un Dockerfile ?

Ils permettent tous les deux de stocker une valeur. La seule différence, est que vous pouvez utiliser l'instruction **ARG** en tant que variable temporaire, utilisable qu'au niveau de votre Dockerfile, à l'inverse de l'instruction **ENV**, qui est une variable d'environnements accessible depuis le Dockerfile et votre conteneur. Donc privilégiez **ARG**, si vous avez besoin d'une variable temporaire et **ENV** pour les variables persistantes.

Quelle est la différence entre COPY et ADD dans un Dockerfile ?

Ils permettent tous les deux de copier un fichier/dossier local vers un conteneur. La différence, c'est que **ADD** autorise les sources sous forme d'url et si jamais la source est une archive dans un format de compression reconnu (ex : zip, tar.gz, etc ...), alors elle sera décompressée automatiquement vers votre cible. Notez que dans les [best-practices de docker](#), ils recommandent d'utiliser l'instruction **COPY** quand les fonctionnalités du **ADD** ne sont pas requises.

Quelle est la différence entre RUN, ENTRYPOINT et CMD dans un Dockerfile ?

- L'instruction **RUN** est **exécutée pendant la construction de votre image**, elle est souvent utilisée pour installer des packages logiciels qui formeront les différentes couches de votre image.
- L'instruction **ENTRYPOINT** est **exécutée pendant le lancement de votre conteneur** et permet de configurer un conteneur qui s'exécutera en tant qu'exécutable. Par exemple pour notre stack LAMP, nous l'avons utilisée, pour démarrer le service Apache avec son contenu par défaut et en écoutant sur le port 80.
- L'instruction **CMD** est aussi **exécutée pendant le lancement de votre conteneur**, elle définit les commandes et/ou les paramètres de l'instruction **ENTRYPOINT** par défaut, et qui peuvent être surchargées à la fin de la commande `docker run`.

Comme expliqué précédemment, il est possible de combiner l'instruction **ENTRYPOINT** avec l'instruction **CMD**.

Je pense qu'un exemple sera plus explicite. Imaginons qu'on souhaite proposer à un utilisateur une image qui donne la possibilité de lister les fichiers/dossiers selon le paramètre qu'il a fournit à la fin de la commande `docker run` (Par défaut le paramètre sera la racine /).

On va commencer par créer notre image Dockerfile, en utilisant l'instruction **ENTRYPOINT** :

```
FROM alpine:latest
```

```
ENTRYPOINT ls -l /Copier
```

Ensuite on construit et on exécute notre image :

```
docker build -t test .Copier
```

```
docker run testCopier
```

Résultat :

```
drwxr-xr-x    2 root    root          4096 Jun 19 17:14 bin
...
drwxr-xr-x   11 root    root          4096 Jun 19 17:14 var
```

Par contre si je tente de surcharger mon paramètre, j'obtiendrai toujours le même résultat :

```
docker run test /etcCopier
```

Pour pouvoir régler ce problème, nous allons utiliser l'instruction **CMD**. Pour rappel l'instruction **CMD** combinée avec **ENTRYPOINT** doivent être spécifiées au format de tableau JSON. Ce qui nous donnera :

```
FROM alpine:latest
```

```
ENTRYPOINT ["ls", "-l"]
```

```
CMD ["/"]Copier
```

On va reconstruire maintenant notre image et relancer notre image avec le paramètre personnalisé.

```
docker build -t test .Copier
```

```
docker run test /etcCopier
```

Résultat :

```
-rw-r--r--    1 root    root          7 Jun 19 17:14 alpine-  
release  
  
...  
  
-rw-r--r--    1 root    root       4169 Jun 12 17:52 udhcpd.conf
```

Voilà l'objectif est atteint ☐.


J'espère, que vous avez bien compris la différence entre les différentes instructions, si ce n'est pas le cas alors n'hésitez pas à me poser des questions dans l'espace commentaire, il est prévu pour ça ☐.

Publier son image dans le Hub Docker

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.

Create Repository

 hajdaini

lamp

LAMP docker image with a demo database and web app.

Visibility

Using 0 of 1 private repositories. [Get more](#)



Public 

Public repositories appear in Docker Hub search results



Private 

Only you can view private repositories

Build Settings *(optional)*

Autobuild triggers a new build with every **git push** to your source code repository. [Learn More.](#)



Connected



Disconnected

Cancel

Create

Create & Build

Une fois que vous aurez choisi le nom et la description de votre repository, cliquez ensuite sur le bouton **create**.

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

```
docker login
```

Il va vous demander, votre nom d'utilisateur et votre mot de passe, et si tout se passe bien vous devez avoir le message suivant :

```
Login Succeeded
```

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

```
docker imagesCopier
```

Résultat :

REPOSITORY SIZE	TAG	IMAGE ID	CREATED
my_lamp seconds ago	latest 497MB	898661ad8fb2	35
alpine ago	latest 5.58MB	4d90542f0623	12 days
debian ago	stable-slim 55.3MB	7279351ce73b	3 weeks

Ensuite il faut rajouter un tag à l'id ou le nom de l'image récupérée. Il existe une commande pour ça, je vous passe d'abord son prototype et ensuite la commande que j'ai utilisée.

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

soit :

```
docker tag my_lamp hajdaini/lamp:firstCopier
```

Si vous relancez la commande `docker images`, vous verrez alors votre image avec le bon tag.

Maintenant envoyez la sauce ☑, en pushant votre image vers le Hub Docker grâce à la commande suivante :

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

soit :

```
docker push hajdaini/lamp:firstCopier
```

Conclusion

Ce chapitre vous a appris à créer des images Dockers personnalisées. Dans le prochain chapitre, nous verrons comment persister nos données avec l'utilisation des volumes Docker.

Fonctionnement et manipulation des volumes dans Docker

je vous explique comment fonctionne les volumes dans Docker et comment créer/gérer/supprimer/lister vos différents volumes Docker.

Les volumes

Introduction

Comme vous le savez déjà, les données dans un conteneur sont éphémères. Il faut donc trouver un moyen à notre solution "**comment sauvegarder les données d'un conteneur**". Nous avons vu une méthode qui n'est pas très recommandée, qui consistait à transformer notre conteneur en image et de relancer un nouveau conteneur basé sur cette nouvelle image. Je vous ai aussi parlé d'une autre méthode qui repose sur les volumes, et ça tombe bien car nous allons voir cette méthode plus en détail sur ce chapitre.

Pourquoi les données d'un conteneur sont éphémères ?

Afin de comprendre ce qu'est un volume Docker, nous devons d'abord préciser le fonctionnement normal du **système de fichiers dans Docker**.

En effet, une image Docker se compose d'un ensemble de **layers (calques) en lecture seule**. Lorsque vous lancez un conteneur à partir d'une image, Docker ajoute au sommet de cette pile de layers un nouveau **layer en lecture-écriture**. Docker appelle cette combinaison de couches un "**Union File System**".

Voyons voir comment le moteur Docker gère les modifications de vos fichiers au sein de votre conteneur :

- Lors d'une modification de fichier, Docker crée une copie depuis les couches en lecture seule vers le layer en lecture-écriture.

- Lors d'une création de fichier, Docker crée le fichier que sur le layer en lecture-écriture, et ne touche pas au layer en lecture seule.
- Lors d'une suppression de fichier, Docker supprime le fichier que sur le layer en lecture-écriture, et si il est déjà existant dans le layer en lecture seule alors il le garde.

Les données dans le layer de base sont en lecture seule, elles sont donc protégées et intactes par toutes modifications, seul le layer en lecture-écriture est impacté lors de modifications de données.

Lorsqu'un conteneur est supprimé, le layer en lecture-écriture est supprimé avec. Cela signifie que toutes les modifications apportées après le lancement du conteneur auront disparus avec.

La création des volumes

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.

Créer et gérer des volumes

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>Copier
```

Soit :

```
docker volume create volume-testCopier
```

Pour **lister les volumes** :

```
docker volume lsCopier
```

Résultat :

```
DRIVER          VOLUME NAME
local
0af7c41b62cf782b4d053e03b4b11575078bb01bbda90edfa73fbc88ac1703ec
...
local          volume-test
```

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

```
docker volume inspect volume-testCopier
```

Résultat sous format JSON:

```
[
  {
    "CreatedAt": "2019-07-03T10:03:20+02:00",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/volume-test/_data",
    "Name": "volume-test",
    "Options": {},
    "Scope": "local"
  }
]
```

Dans ce résultat, on peut remarquer que toutes les nouvelles données de notre conteneur seront stockées sur le point de montage `/var/lib/docker/volumes/volume-test/_data`.

Pour **supprimer un volume** :

```
docker volume rm volume-testCopier
```

Démarrer un conteneur avec un volume

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`.

Pour ce chapitre, nous allons créer une petite image pour tester cette option, commencez d'abord par créer un Dockerfile avec le contenu suivant :

```
FROM alpine:latest

RUN mkdir /data

WORKDIR /dataCopier
```

Ensuite buildez notre image

```
docker build -t vtest .Copier
```

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 vtestCopier
```

Ouvrez un autre terminal et dans celui-ci inspectez le nouveau volume :

```
docker volume inspect data-testCopier
```

Résultat sous format JSON:

```
[
  {
    "CreatedAt": "2019-07-03T10:28:55+02:00",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/data-test/_data",
    "Name": "data-test",
    "Options": null,
    "Scope": "local"
  }
]
```

Nous allons essayer de voir en temps réel le contenu de ce volume, pour cela utilisez la commande suivante sur votre nouveau terminal :

```
sudo watch -n 1 ls /var/lib/docker/volumes/data-test/_dataCopier
```

Résultat :

```
sEvery 1,0s: ls /var/lib/dock... localhost.localdomain: Wed Jul 3
10:48:52 2019
```

Pour le moment le dossier est vide, maintenant retournez vers le terminal avec le shell du conteneur et créez dans le dossier */data* un fichier avec le texte suivant :

```
echo "ceci est un test" > test.txtCopier
```

Si vous retournez sur le nouveau terminal, vous verrez dessus votre nouveau fichier :

```
sEvery 1,0s: ls /var/lib/dock... localhost.localdomain: Wed Jul 3  
10:48:59 2019
```

```
test.txt
```

Maintenant, je vais quitter mon conteneur avec la commande `exit` et le supprimer :

```
docker rm -f vtest_cCopier
```

Je vais recréer un nouveau conteneur, pour vérifier que les données ont bien été sauvegardées :

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

Dans ce même nouveau conteneur, vérifie je le contenu du fichier crée précédemment :

```
cat test.txtCopier
```

Résultat :

```
ceci est un test
```

Cool, nos données sont maintenant bien sauvegardées ☐.

Amélioration de notre image LAMP

Dans le chapitre précédent, nous avons créé une image avec stack LAMP, malheureusement, c'est pas top niveau persistance de données car, lors d'un redémarrage du conteneur, nous allons rencontrer les deux problèmes suivants :

- Les données de notre base de données ne sont pas sauvegardées.
- Les modifications des sources de notre application ne seront pas appliquées.

Pour résoudre ce problème, nous allons utiliser les volumes !

Commencez par télécharger le projet , en cliquant [ici](#). Dans ce projet, j'ai gardé le même Dockerfile, mais j'ai juste changé les fichiers sources en rajoutant un formulaire.

Désarchivez le fichier zip et buildez votre image :

```
docker build -t my_lamp .Copier
```

Concernant la base de données, nous allons créer un volume nommé "mysqldata" qui sera par la suite basé sur le dossier */var/lib/mysql* du conteneur:

```
docker volume create --name mysqldataCopier
```

Pour les sources de mon application, je vais faire les choses différemment. Je vais juste changer le dossier source du volume (les volumes nous donne cette possibilité). Lançons donc notre conteneur :

```
docker run -d --name my_lamp_c -v $PWD/app:/var/www/html -v  
mysqldata:/var/lib/mysql -p 8080:80 my_lampCopier
```

La commande `$PWD` prendra automatiquement le chemin absolu du dossier courant, donc faites bien attention à exécuter votre image depuis le dossier du projet où mettez le chemin complet si vous souhaitez lancer votre commande depuis n'importe quelle autre chemin.

Articles

Nouveau article

Titre *

Nom de l'auteur *

Contenu *

Liste d'articles

test 2

03/07/19 11:42

ceci est un article de test 2

— test 2

test 1

Vous pouvez dès à présent modifier vos sources, depuis votre conteneur ou votre machine local et vos changements seront permanents et immédiatement traités par l'interpréteur php. Les données de votre base de données seront aussi automatiquement sauvegardées dans le volume mysqldata.

Bon bah bravo, nous avons atteint notre objectif ☐!

Conclusion

Avec les volumes, nous avons pu créer une image assez stable et exploitable dans un environnement de production. Vous l'avez sûrement remarqué mais notre commande `docker run` commence à devenir vraiment longue et nous n'avons pas encore résolu le problème qui est de séparer notre conteneur d'application web de notre conteneur de base de données. Et c'est pour ces raisons que dans le prochain chapitre, nous verrons le `docker-compose.yml`, c'est un fichier qui va nous permettre de définir le comportement de nos conteneurs et d'exécuter des applications Docker à conteneurs multiples.

Comme d'habitude, voici un petit récapitulatif des commandes liées aux volumes :

```
## 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 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
```

Gérez vos conteneurs avec le Docker Compose

Docker Compose est un outil permettant de faciliter et centraliser la gestion des applications Docker à conteneurs multiples. Nous allons apprendre à l'utiliser à travers cet article.

Introduction

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**.

Installation du docker-compose

Docker Compose n'est pas installé par défaut et s'appuie sur le moteur Docker pour fonctionner. Au jour d'aujourd'hui, la dernière version de Docker Compose est la 1.24.0.

Voici la procédure à suivre pour **télécharger Docker Compose sous un environnement Linux** :

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.24.0/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-
composeCopier

sudo chmod +x /usr/local/bin/docker-composeCopier

sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-
composeCopier
```

Vérifiez ensuite votre installation :

```
docker-compose --versionCopier

docker-compose version 1.24.0, build 0aa59064
```

Si vous n'avez pas d'erreur, alors vous pouvez poursuivre la lecture de ce chapitre.

Définition des besoins du Docker Compose et amélioration du Dockerfile

Le but de cet article est d'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 du projet en cliquant [ici](#) et désarchivez ensuite le projet.

Amélioration du Dockerfile

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

Conseil

Si vous souhaitez conteneuriser une application assez connue, alors je vous conseille de toujours fouiller dans le Hub Docker, afin de savoir si une image officielle de l'application existe déjà.

En cherchant dans le Hub Docker, j'ai pu dénicher les images adéquates, notamment :

- Une [image officielle php](#) avec le tag 7-apache
- Une [image officielle mysql](#)

Une fois que j'ai 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 :

```
FROM php:7-apache

LABEL version="1.0" maintainer="AJDAINI Hatim "

# Activation des modules php

RUN docker-php-ext-install pdo pdo_mysql

WORKDIR /var/www/htmlCopier
```

Buildez ensuite votre image avec la commande suivante :

```
docker build -t myapp .Copier
```

Les besoins pour notre Docker Compose

Avant de créer notre fichier docker-compose.yml, il faut auparavant **définir les comportements de nos conteneurs**.

Nos besoins pour le conteneur de la base de données

On va débiter 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.

Avant de foncer tête baissée dans la création/modification de notre fichier sql, 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.

Nos besoins pour le conteneur de l'application web

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).

Lancer les conteneurs sans le docker-compose

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:5.7Copier
```

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 myappCopier
```

Dans cet exemple, on peut vite remarquer que les commandes `docker run` sont assez longues et par conséquent elles ne sont pas assez lisibles. De plus, vous aurez à lancer cette commande pour chaque nouveau démarrage de l'application. Mais vous aurez aussi à gérer vos différents conteneurs séparément. C'est pour ces raisons, que nous utiliserons le fichier `docker-compose.yml` afin de **centraliser la gestion de nos multiples conteneurs d'une application Docker depuis un seul fichier**. Dans notre cas il va nous permettre d' **exécuter et définir les services, les volumes et la mise en relation des différents conteneurs** de notre application.

Création du docker-compose

Contenu du docker-compose

Commencez d'abord par créer un fichier et nommez le `docker-compose.yml`, ensuite copiez collez le contenu ci-dessous. Par la suite, plus bas dans l'article, je vais vous fournir les explications des différentes lignes de ce fichier :

```
version: '3.7'

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:Copier
```

Explication du fichier docker-compose.yml

```
version: '3.7'
```

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 3.7.

```
services:Copier
```

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.

```
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: testCopier
```

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:

    - dbCopier
```

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 :
 - Les services démarrent en ordre de dépendance. Dans notre cas, le service `db` est démarré avant le service `app`
 - Les services s'arrêtent selon l'ordre de dépendance. Dans notre cas, le service `app` est arrêté avant le service `db`

```
volumes:
```

```
db-volume:Copier
```

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.

Lancer l'application depuis docker-compose.yml

Pour être sur le même pied d'estale, voici à quoi doit ressembler votre arborescence :

```
├─ app
|   ├── db-config.php
|   ├── index.php
|   └─ validation.php
├─ articles.sql
├─ docker-compose.yml
└─ Dockerfile
```

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 -dCopier
```

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

Si vous le souhaitez, vous pouvez **vérifier le démarrage des conteneurs issus du docker-compose.yml** :

```
docker psCopier
```

Résultat :

CONTAINER ID CREATED	IMAGE STATUS	COMMAND PORTS	NAMES
26bb6e0dd252 seconds ago	myapp Up 5 seconds	"docker-php-entrypoi..." 0.0.0.0:8080->80/tcp	34 myapp_c
b5ee22310ebc seconds ago	mysql:5.7 Up 6 seconds	"docker-entrypoint.s..." 3306/tcp, 33060/tcp	35 mysql_c

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

```
docker-compose ps
```

Résultat :

Name	Command	State	Ports

myapp_c	docker-php-entrypoint apac ...	Up	0.0.0.0:8080->80/tcp
mysql_c	docker-entrypoint.sh mysqld	Up	3306/tcp, 33060/tcp

Si jamais vos conteneurs ne sont pas dans l'état **UP**, alors **vérifiez les logs des services de votre Docker Compose** en tapant la commande suivante :

```
docker-compose logs
```

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

Articles

Nouveau article

Titre *

Nom de l'auteur *

Contenu *

Liste d'articles

test2

06/07/19 18:38

Je test l'article test 2

— test2

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.

Détails de la communication inter-conteneurs dans les sources de l'application

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)Copier
```

Résultat :

```
/myapp_c - 172.18.0.2  
/mysql_c - 172.18.0.3
```

Pour faire communiquer notre application web avec la base de données, on peut 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';Copier
```

Dans ces cas, j'ai utilisé le nom du conteneur de la base de données pour communiquer avec ce dernier.

Conclusion

Je pense que vous l'aurez compris, le Docker Compose est un outil permettant de faciliter la gestion des applications Docker à conteneurs multiples, comme :

- Démarrer, arrêter et reconstruire des services
- Afficher le statut des services en cours d'exécution

- Diffuser la sortie des logs des services en cours d'exécution
- Exécuter une commande unique sur un service
- etc ...

Comme pour chaque fin de chapitre, je vous liste ci-dessous un récapitulatif de quelques commandes intéressantes du Docker Compose:

```
## 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 imagesCopier
```

[Chapitre précédent](#)

Fonctionnement et manipulation du réseau dans Docker

Cet article vous décrit les différents types de driver dans docker et comment créer/gérer/supprimer/lister vos différents réseaux Docker.

Introduction

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, que nous allons voir à travers ce chapitre.

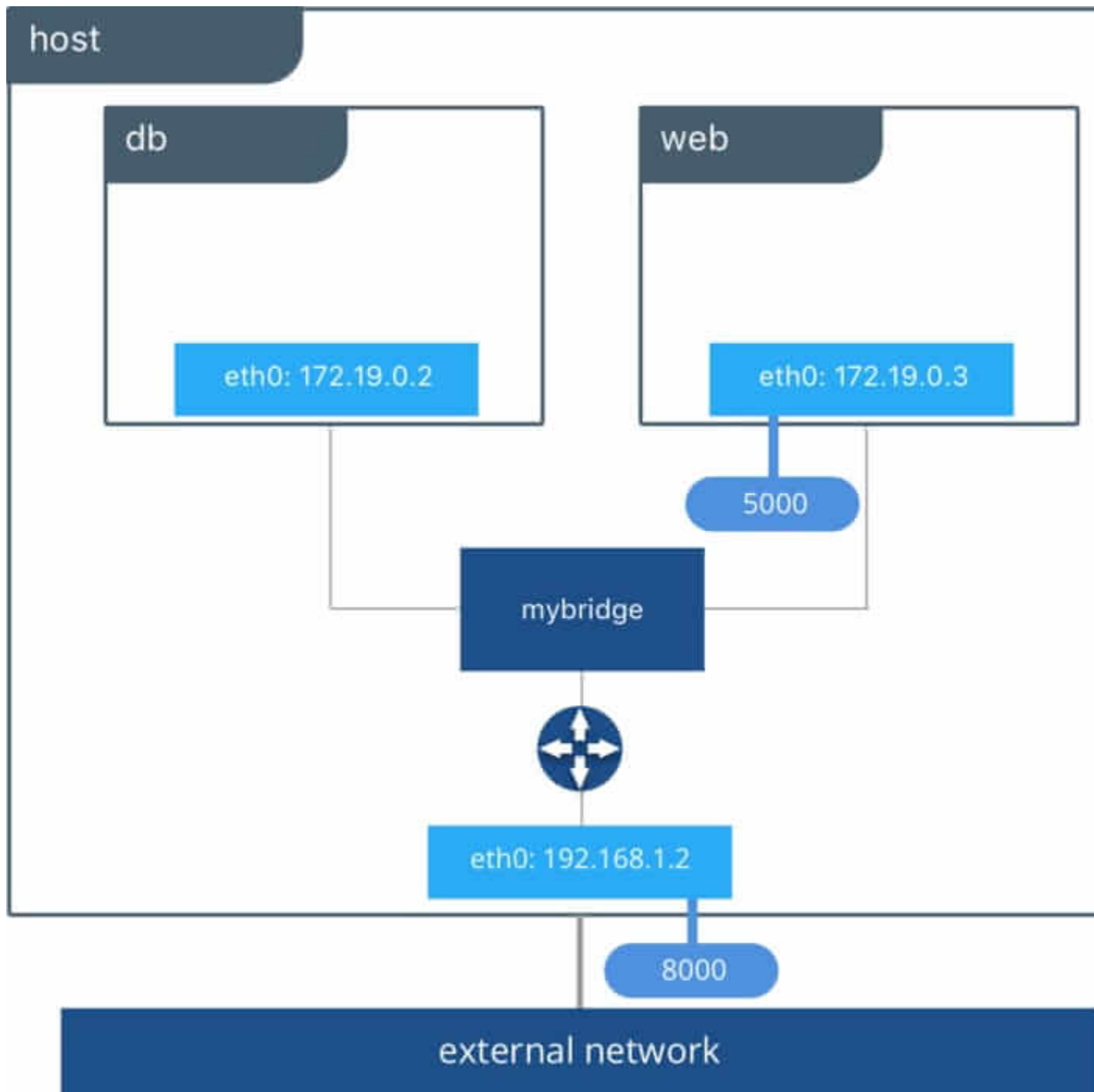
Présentation des différents types de réseau Docker

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

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.



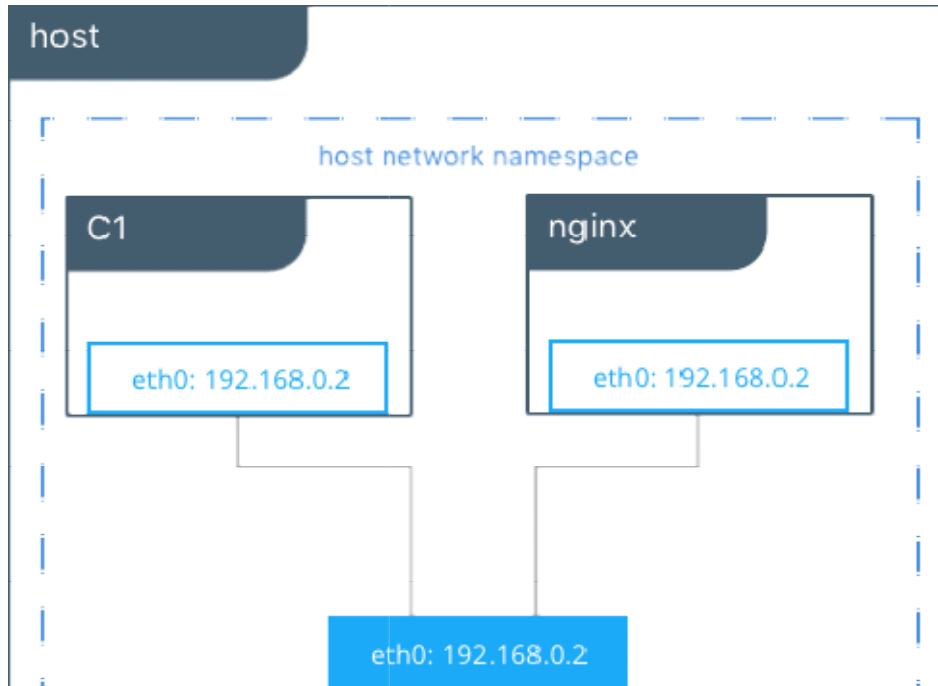
Pour que les conteneurs sur le réseau bridge puissent communiquer ou être accessibles du monde extérieur, vous devez configurer le mappage de port.

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.



Me concernant sur mon pc perso j'utilise le réseau wifi, plus précisément l'interface **wlp3s0**. Voici les informations retournées par la commande `ip add show wlp3s0` depuis ma machine hôte :

```
wlp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default qlen 1000

link/ether dc:85:de:ce:04:55 brd ff:ff:ff:ff:ff:ff

inet 192.168.0.11/24 brd 192.168.0.255 scope global dynamic
noreferrerroute wlp3s0

    valid_lft 54874sec preferred_lft 54874sec

inet6 fe80::335:f1f5:127d:b62c/64 scope link noreferrerroute

    valid_lft forever preferred_lft forever
```


Je vais lancer la même commande dans un conteneur basé sur l'image alpine avec un driver de type host :

```
docker run -it --rm --network host --name net alpine ip add show wlp3s0Copier
```

Sans surprise, j'obtiens les mêmes informations que sur ma machine hôte (normal car ils utilisent tous les deux l'interface **wlp3s0** grâce au driver host):

```
wlp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP qlen 1000

    link/ether dc:85:de:ce:04:55 brd ff:ff:ff:ff:ff:ff

    inet 192.168.0.11/24 brd 192.168.0.255 scope global dynamic
wlp3s0

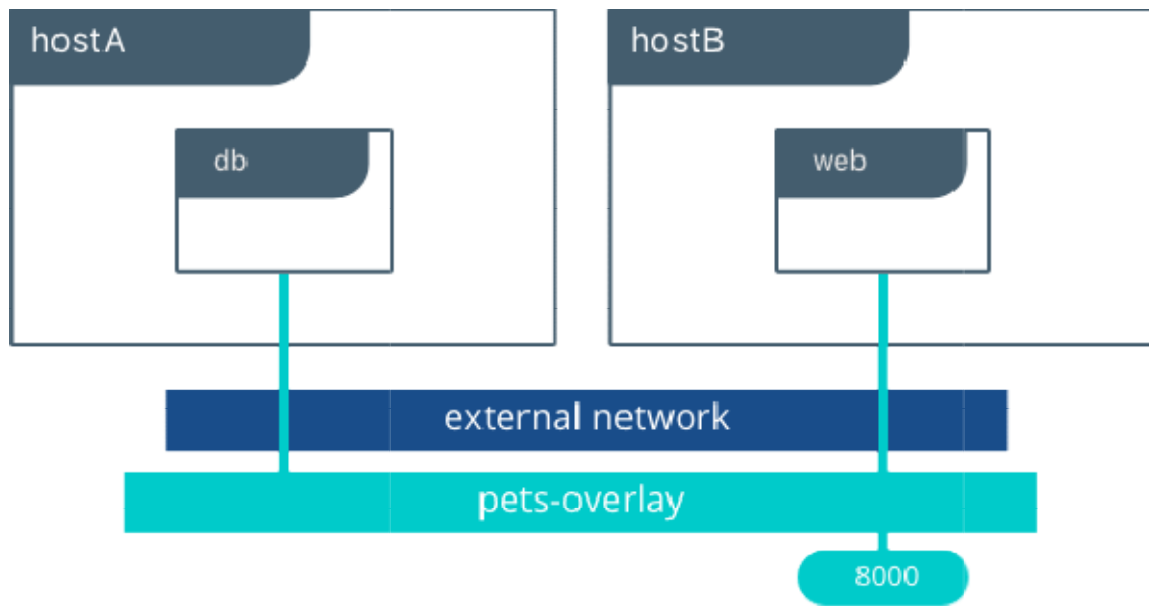
        valid_lft 54711sec preferred_lft 54711sec

    inet6 fe80::335:f1f5:127d:b62c/64 scope link

        valid_lft forever preferred_lft forever
```

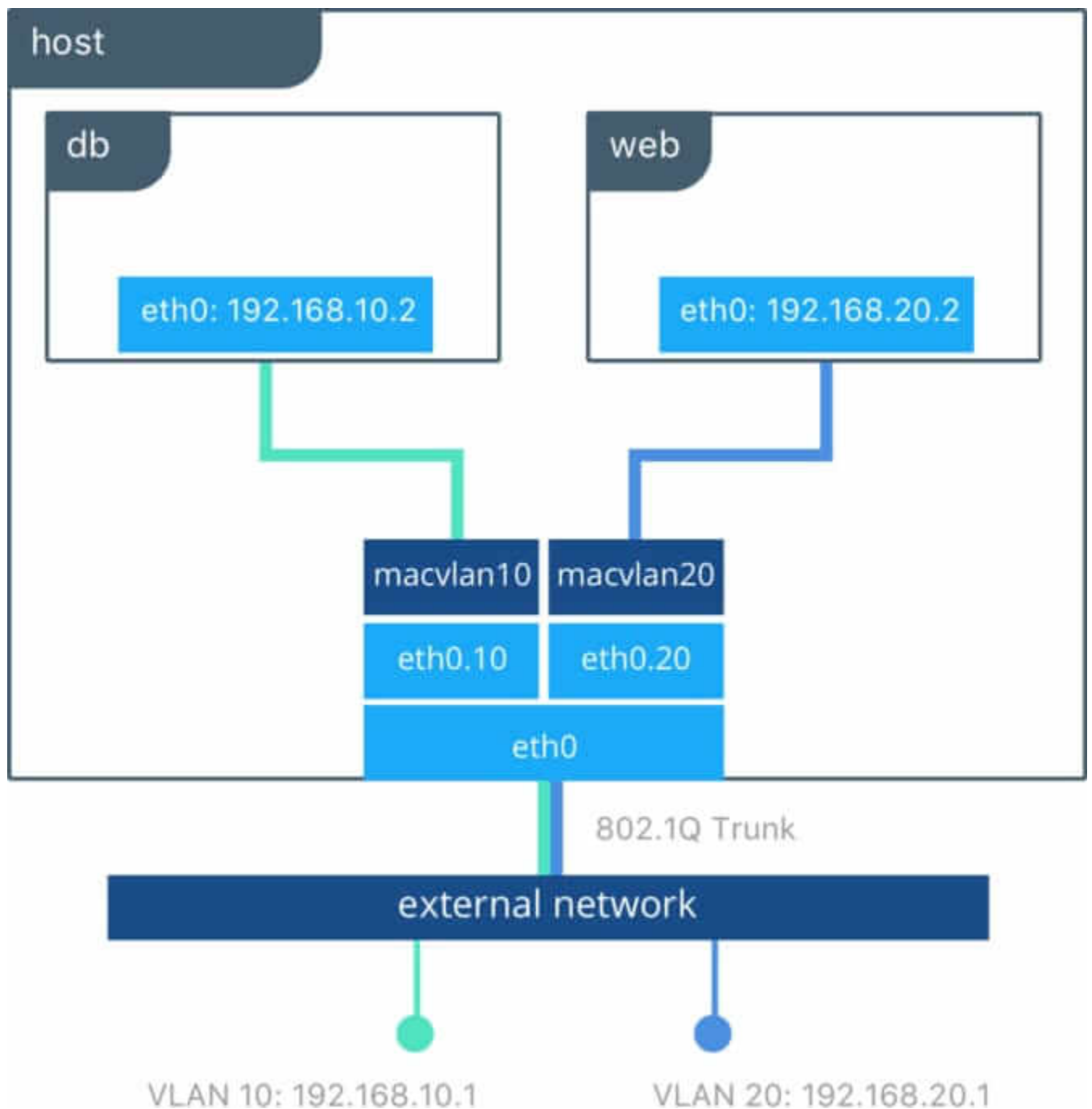
Le driver overlay

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.



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.



Manipulation du réseau dans Docker

Une fois les présentations finies, il est temps de pratiquer un peu en manipulant le réseau dans Docker.

Créer et récolter des informations d'un réseau Docker

La **commande pour créer un réseau Docker** est la suivante :

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

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

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

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

```
docker network lsCopier
```

Résultat :

NETWORK ID SCOPE	NAME	DRIVER
58b8305ce041 local	bridge	bridge
91d7f01dad50 local	host	host
ccdbdbf708db local	mon-bridge	bridge
10ee25f56420 local	monimagedocker_default	bridge
6851e9b8e06e local	none	null

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-bridgeCopier
```

Résultat :

```
[  
  
  {  
  
    "Name": "mon-bridge",
```

```

        "Id":
"ccdbdbf708db7fa901b512c8256bc7f700a7914dfaf6e8182bb5183a95f8dd9b",

        ...

        "IPAM": {

            "Driver": "default",

            "Options": {},

            "Config": [

                {

                    "Subnet": "172.21.0.0/16",

                    "Gateway": "172.21.0.1"

                }

            ]

        },

        ...

        "Labels": {}

    }

]

```

Information

Vous pouvez **surcharger la valeur du Subnet et de la Gateway** en utilisant les options **--subnet** et **--gateway** de la commande `docker network create`, comme suit :

```
docker network create bridge --subnet=172.16.86.0/24 --gateway=172.16.86.1 mon-bridge
```

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

```
docker run -dit --name alpine1 --network mon-bridge alpineCopier
docker run -dit --name alpine2 --network mon-bridge alpineCopier
```

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

```
docker network inspect mon-bridgeCopier
```

Résultat :

```
[
  {
    "Name": "mon-bridge",
    "Id": "ccdbdbf708db7fa901b512c8256bc7f700a7914dfaf6e8182bb5183a95f8dd9b",
    "...",
    "Containers": {
      "1ab5f1815d98cd492c69a63662419e0eba891c0cadb2cbdd0fb939ab25f94b33": {
        "Name": "alpine1",
        "EndpointID": "5f04963f9ec084df659cfc680b9ec32c44237dc89e96184fe4f2310ba6af7570",
        "MacAddress": "02:42:ac:15:00:02",
        "IPv4Address": "172.21.0.2/16",
        "IPv6Address": ""
```

```

    },

    "a935d2e1ddf76fe49cdb1950653f4a093928020b49ebfea4130ff9d712ffb1d6":
    {

        "Name": "alpine2",

        "EndpointID":
        "3e009b56104a1bf9106bc622043a2ee06010b102279e24b4807c7b7ffec166dd",

        "MacAddress": "02:42:ac:15:00:03",

        "IPv4Address": "172.21.0.3/16",

        "IPv6Address": ""

    }

    },

    ...

}

]

```

D'après le résultat, on peut s'apercevoir que notre conteneur **alpine1** possède l'adresse IP **172.21.0.2**, et notre conteneur **alpine2** possède l'adresse IP **172.21.0.3**. Tentons de les faire communiquer ensemble à l'aide de la commande ping :

```
docker exec alpine1 ping -c 1 172.21.0.3
```

Résultat :

```

PING 172.21.0.3 (172.21.0.3): 56 data bytes

64 bytes from 172.21.0.3: seq=0 ttl=64 time=0.101 ms

```

```
docker exec alpine2 ping -c 1 172.21.0.2
```

Résultat :

```
PING 172.21.0.2 (172.21.0.2): 56 data bytes  
64 bytes from 172.21.0.2: seq=0 ttl=64 time=0.153 mss
```

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-hostCopier
```

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 httpdCopier
```

Sans aucun mappage, vous pouvez accéder au serveur Apache en accédant à <http://localhost:80/>, vous verrez alors le message "It works!".

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

```
sudo netstat -tulpn | grep :80Copier
```

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

```
tcp        0      0 127.0.0.1:8000      0.0.0.0:*  
LISTEN    5084/php  
  
tcp6       0      0 :::80              :::*  
LISTEN    11133/httpd
```



```
tcp6      0      0 :::8080      :::*
LISTEN    3122/docker-prox
```

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

```
docker container stop my_httpdCopier
```

Supprimer, connecter et connecter un réseau Docker

Avant de supprimer votre réseau docker, il est nécessaire au préalable de supprimer tout conteneur connecté à votre réseau docker, ou sinon il suffit juste de **déconnecter votre conteneur de votre réseau docker sans forcément le supprimer**.

Nous allons choisir la méthode 2, en déconnectant tous les conteneurs utilisant le réseau docker **mon-bridge** :

```
docker network disconnect mon-bridge alpine1Copier
docker network disconnect mon-bridge alpine2Copier
```

Maintenant, si vous vérifiez les interfaces réseaux de vos conteneurs basées sur l'image alpine, vous ne verrez que l'interface loopback comme pour le driver none :

```
docker exec alpine1 ip aCopier
```

Résultat :

```
lo:  mtu 65536 qdisc noqueue state UNKNOWN qlen 1000

    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

    inet 127.0.0.1/8 scope host lo

        valid_lft forever preferred_lft forever
```

Une fois que vous avez déconnecté tous vos conteneurs du réseau docker **mon-bridge**, vous pouvez alors le supprimer :

```
docker network rm mon-bridgeCopier
```

Cependant vos conteneurs se retrouvent maintenant sans interface réseau bridge, il faut donc **reconnecter vos conteneurs au réseau bridge par défaut** pour qu'ils puissent de nouveau communiquer entre eux :

```
docker network connect bridge alpine1Copier  
docker network connect bridge alpine2Copier
```

Vérifiez ensuite si vos conteneurs ont bien reçu la bonne Ip :

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

Résultat :

```
/alpine2 - 172.17.0.3  
/alpine1 - 172.17.0.2
```

Conclusion

Vous pouvez créer autant de réseaux bridge que vous souhaitez, ça reste un bon moyen pour sécuriser la communication entre vos conteneurs, car les conteneurs connectés au bridge1 ne peuvent pas communiquer avec les conteneurs du bridge2, limitant ainsi les communications inutiles.

Concernant le driver overlay, j'essayerais de vous montrer son utilisation dans un autre article car le sujet est très vaste et demande des connaissances sur d'autres sujets que nous n'avons pas eu encore l'occasion de voir, notamment le docker swarm.

Comme d'habitude, voici l'aide-mémoire de ce cours :

```
## 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>
```

```
## Récupérer des informations sur un réseau docker
```

```
docker network inspect <NETWORK NAME>
```

```
    -v ou --verbose : mode verbose pour un meilleur diagnostic
```

```
## 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>
```

Déployer, manipuler et sécuriser un serveur Registry Docker privé

Ce chapitre, vous explique l'utilité d'un registre Docker et vous apprend à déployer, manipuler et sécuriser un serveur Registry Docker privé.

Introduction

Dans ce chapitre, nous allons nous intéresser à la partie Registry (en fr : Registre) dans Docker.

C'est quoi ?

Le Registry Docker est un **système de stockage et de distribution d'image Docker** open-source (sous la licence Apache), déployé côté serveur. Il permet aux utilisateurs d'**extraire et insérer des images Docker dans un dépôt avec les autorisations d'accès appropriées**. La même image peut avoir plusieurs versions différentes, identifiées par leurs tags.

Quand l'utiliser ?

Vous devez utiliser le Registry si vous voulez :

- Contrôler étroitement l'endroit où vos images sont stockées
- Posséder pleinement le contrôle de votre pipeline de distribution d'images
- Intégrer étroitement le stockage et la distribution des images dans votre flux de travail de développement interne

Déploiement et manipulation d'un Registry privé

Création du Registry privé

Par défaut, le moteur Docker interagit avec le [DockerHub](#). DockerHub est le registre officiel de Docker offrant une solution prête à l'emploi, que nous avons déjà eu l'occasion d'utiliser [ici](#). Il ne nécessite aucune maintenance et fournit un Registry gratuit, ainsi que des fonctionnalités supplémentaires telles que des comptes d'organisation, une intégration à des solutions de contrôle de source notamment Github et Bitbucket, etc. Mais dans ce cours, nous allons nous intéresser au **déploiement d'un serveur Docker Registry privé**.

Avant de pouvoir déployer un Registry, vous devez d'abord installer le moteur Docker sur la machine hôte qui hébergera vos images Docker, car un Registry n'est rien d'autre qu'une image Docker qui attend à être exécutée. Voici la commande qui permet de **créer un Docker Registry privé** :

```
docker run -d -p 5000:5000 --restart=always --name mon-registry registry:2Copier
```

Déposer une image dans le Registry privé

Premièrement, nous allons créer une image personnalisée (alpine + git) que nous allons par la suite déposer dans notre Registry. Commencez d'abord par créer un Dockerfile et ajoutez dedans le contenu suivant :

```
FROM alpine:latest  
  
RUN apk add --no-cache gitCopier
```

Construisez ensuite votre image :

```
docker build -t alpinegit .Copier
```

En vue de pousser notre image dans notre Registry, il faut au préalable **créer un nouveau tag de votre image** en respectant le format suivant :

```
docker tag alpinegit <SERVER NAME REGISTRY>:<PORT SERVER  
REGISTRY>/<CONTAINER NAME>Copier
```

Soit :

```
docker tag alpinegit localhost:5000/alpinegitCopier
```

Dès à présent, vous pouvez **envoyer votre image vers le registry docker privé** :

```
docker push localhost:5000/alpinegitCopier
```

Visualiser les images disponibles dans le Registry privé

Une fois votre image envoyée, il est possible de la visualiser grâce à l'**API du Docker Registry**. Je vais utiliser la commande curl pour manipuler l'API du Registry.

Voici l'url qui permet de **lister les différentes images dans votre Registry Docker** :

```
curl -X GET http://localhost:5000/v2/_catalogCopier
```


Résultat :

```
{"repositories":["alpinegit"]}
```

Nous allons créer une nouvelle image, qu'on pushera ensuite dans notre registry privé, dans le but de savoir si l'API Docker Registry prend bien en compte notre nouvelle image :

Dockerfile

```
FROM alpine:latest  
  
RUN apk add --no-cache mysql-client  
  
ENTRYPOINT ["mysql"]Copier
```

Vous connaissez maintenant la musique , en construit l'image, change son tag et on la push :

```
$ docker build -t alpinemysql .  
  
$ docker tag alpinemysql localhost:5000/alpinemysql  
  
$ docker push localhost:5000/alpinemysqlCopier
```

Vérifions ensuite si l'api nous retourne bien notre nouvelle image :

```
curl -X GET http://localhost:5000/v2/_catalogCopier
```

Et c'est bien le cas :

```
{"repositories":["alpinegit","alpinemysql"]}
```

Si vous voulez, vous pouvez supprimer vos images localement à l'aide de la commande `docker rmi` et vous verrez que vos images seront toujours disponibles dans votre docker registry, car ces dernières sont stockées dans votre conteneur registry. Par la suite vous pouvez **récupérer vos images depuis votre registry privé** avec la commande suivante :

```
docker pull localhost:5000/alpinegitCopier  
docker pull localhost:5000/alpinemysqlCopier
```

Visualiser les différents tags d'une image dans le Registry privé

Dans cette partie, nous allons rajouter un nouveau tag nommé "test" sur notre image **alpinegit**,

```
docker tag alpinegit localhost:5000/alpinegit:testCopier  
docker push localhost:5000/alpinegit:testCopier
```

Nous allons maintenant, **réutiliser l'api Docker Registry pour lister les différents tags de notre image alpinegit** :

```
curl -X GET http://localhost:5000/v2/alpinegit/tags/listCopier
```

Résultat :

```
{"name":"alpinegit","tags":["test","latest"]}
```

Sécurité

Le stockage dans un Registry Docker

Par défaut aucun volume n'est créé, donc si vous quitter votre conteneur registry, vous perdrez automatiquement toutes vos images hébergées dans votre registry. Il est donc très important de **rajouter un volume**. L'exemple suivant monte le répertoire hôte *data/* vers le dossier */var/lib/registry/* du conteneur **mon-registry**.

Premièrement, on crée le dossier de stockage sur notre machine hôte :

```
mkdir dataCopier
```

Deuxièmement, on **démarre notre conteneur avec le volume adéquat** :

```
docker run -d \
    -p 5000:5000 \
    --restart=always \
    --name mon-registry \
    -v "$(pwd)"/data:/var/lib/registry \
    registry:2Copier
```

le chiffrement

L'exécution d'un Registry accessible uniquement en local a une utilité limitée. Afin de rendre un Registry accessible aux hôtes externes, vous devez d'abord **sécuriser le Registry Docker à l'aide de TLS**.

Bien qu'il soit vivement recommandé de sécuriser votre base de registre à l'aide d'un certificat TLS émis par une autorité de certification connue, vous pouvez choisir d'utiliser des certificats auto-signé ou d'utiliser votre base de registre via une connexion HTTP non chiffrée. L'un ou l'autre de ces choix implique des compromis en matière de sécurité et des étapes de configuration supplémentaires.

Information

Le Registry prend en charge l'utilisation de Let's Encrypt afin d'obtenir automatiquement un certificat approuvé par les navigateurs.

Cependant dans notre exemple, nous allons générer des certificats auto-signés, ils ne seront pas reconnus par nos navigateurs mais au moins **la communication avec notre Registry sera chiffrée**.

Conseil

En utilisant un registre en HTTP seulement, vous exposez alors votre base de registre à des attaques triviales de type "homme au milieu" (MITM).

En premier lieu, commencez par créer un dossier nommé *certs* pour stocker nos certificats :

```
mkdir certsCopier
```

En seconde partie, nous allons **générer nos propres certificats avec l'outil openssl** :

```
openssl req \
    -newkey rsa:4096 -nodes -sha256 -keyout certs/localhost.key \
    -x509 -days 365 -out certs/localhost.crtCopier
```

Openssl vous demandera quelques informations, vous pouvez laisser les options par défaut en appuyant sur la touche entrée.

Pour utiliser les certificats générés, nous allons alors surcharger des variables d'environnements proposé par l'image **registry:2** et utiliser le port 443 :

```
docker run -d \
    --restart=always \
    --name mon-registry \
    -v "$(pwd)/data:/var/lib/registry \
    -v "$(pwd)/certs:/certs \
    -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
    -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/localhost.crt \
```

```
-e REGISTRY_HTTP_TLS_KEY=/certs/localhost.key \  
  
-p 443:443 \  
  
registry:2Copier
```

Information

La ligne `REGISTRY_HTTP_ADDR=0.0.0.0:443`, permet de rendre notre registry accessible depuis n'importe IP.

Maintenant on est capable d'envoyer nos images vers notre nouveau registry chiffrée :

```
$ docker tag alpinegit localhost:443/alpinegit  
  
$ docker tag alpinemysql localhost:443/alpinemysql  
  
$ docker push localhost:443/alpinegit  
  
$ docker push localhost:443/alpinemysqlCopier
```

La prochaine étape consiste à visiter la page suivante https://localhost/v2/_catalog. Si votre navigateur vous le permet, alors il vous demandera de rajouter le certificat en question car ce dernier n'est pas reconnu par une CA. Dans mon cas j'utilise Firefox, et voici ce que j'obtiens comme résultat :



Attention : risque probable de sécurité

Firefox a détecté une menace de sécurité potentielle et n'a pas poursuivi vers des attaquants pourraient dérober des informations comme vos mots de passe bancaires.

[En savoir plus...](#)

Retour

☐ Signaler les erreurs similaires pour aider Mozilla à identifier et bloquer le


Cliquez ensuite sur le bouton "Avancé", puis sur "Accepter le risque et poursuivre" et si tout se passe comme prévu alors vous obtiendrez le résultat suivant :

```
JSON  Données brutes  En-têtes
Enregistrer Copier Tout réduire Tout développer
▼ repositories:
  0: "alpinegit"
  1: "alpinemysql"
```

Restriction d'accès

Pour rajouter une autre couche de sécurité dans le fonctionnement de vos registres sur des réseaux locaux sécurisés, vous pouvez implémenter des restrictions d'accès.

Pour le moment, nous allons **mettre en place une Authentification de base avec un utilisateur et un mot de passe globaux** à l'aide du fichier `registry.json`.

Heureusement que notre baleine (mascotte de docker ) a pensé à nous, car depuis l'image **registry**, il est possible de générer le fichier `registry.json`.

On commence d'abord par créer un dossier *auth* pour stocker notre fichier :

```
mkdir authCopier
```

Par la suite on génère notre fichier :

```
docker run --rm \
    --entrypoint htpasswd \
    registry:2 -Bbn testuser testpassword > auth/htpasswdCopier
```

Démarrer ensuite votre conteneur en créant un volume qui prend en compte le fichier :

```
docker run -d \
    -p 5000:5000 \
    --restart=always \
    --name mon-registry \
    -v "$(pwd)/data:/var/lib/registry \
    -v "$(pwd)/auth:/auth \
    -e "REGISTRY_AUTH=htpasswd" \
    -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
    -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
    -v "$(pwd)/certs:/certs \
    -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/localhost.crt \
    -e REGISTRY_HTTP_TLS_KEY=/certs/localhost.key \
    registry:2Copier
```

Maintenant, il faut **se connecter sur votre Docker Registry privé** :

```
docker login localhost:5000Copier
```

```
Username: testuser
```

```
Password: testpassword
```

Si tout se déroule comme prévu, vous devriez avoir le message suivant :

```
Login Succeeded
```

Une fois authentifié, vous pouvez alors envoyer votre image dans votre Registry privé :

```
docker push localhost:5000/alpinemysqlCopier
```

Docker Compose

Normalement, si vous avez suivi ce chapitre depuis le début, vous devez vous retrouver avec l'arborescence suivante :

```
├─ auth
|   └─ httpasswd
├─ certs
|   ├── localhost.crt
|   └─ localhost.key
└─ data
```

Reprenons les fonctionnalités que nous venons de mettre en place. Actuellement nous avons :

- Un volume pour stocker nos images envoyées par l'utilisateur
- Une communication chiffrée
- Un système d'authentification basique

Nous allons reprendre toutes ces fonctionnalités et les rajouter dans un Docker Compose :

```
version: '3.7'
```

```
services:
```

```
registry:

  restart: always

  image: registry:2

  container_name: my-web-container

  ports:

    - 5000:5000

  environment:

    REGISTRY_HTTP_TLS_CERTIFICATE: /certs/localhost.crt

    REGISTRY_HTTP_TLS_KEY: /certs/localhost.key

    REGISTRY_AUTH: htpasswd

    REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd

    REGISTRY_AUTH_HTPASSWD_REALM: Registry Realm

  volumes:

    - ./data:/var/lib/registry

    - ./certs:/certs

    - ./auth:/authCopier
```

Conclusion

Nous avons réussi à mettre en place un Registry Docker privé en rajoutant quelques couches de sécurité. Il est possible bien sûr de perfectionner encore plus ce registry basique en prenant par exemple en charge l'envoi de notifications Webhook en réponse aux événements se produisant dans le registre, ou en rajoutant des autorisations d'accès comme le readonly pour certains clients qui ne seront pas autorisés à écrire dans le registre etc ... Je vous conseille donc vivement de jeter un coup d'œil à la [documentation officielle des Registry Docker](#).

Apprendre à déboguer vos conteneurs et vos images Docker

Dans ce chapitre, nous apprendrons à déboguer des conteneurs et images Docker, de manière à ce que vous soyez capable de les réparer mais aussi d'automatiser vos tâches d'administration Docker.

Introduction

Dans ce chapitre, nous allons nous attaquer à la partie Debug dans Docker. Le but de ce chapitre c'est que vous soyez capable de récolter finement des informations sur vos conteneurs afin d'être **capable de réparer vos conteneurs** mais aussi d'utiliser ces données dans vos scripts dans l'intention d'**automatiser vos tâches d'administration Docker**.

Les commandes de débogage

la commande stats

Imaginez que vous utilisez un conteneur (un Apache par exemple), mais malheureusement il n'arrive plus à répondre malgré le fait que son statut soit toujours à l'état **UP**. Que feriez-vous si étiez dans ce cas précis ?

Dans un premier temps, il serait d'abord intéressant de **vérifier les statistiques d'utilisation des ressources de votre conteneur**. Ceci pour se faire à l'aide de la commande Docker stats.

Dans le but de manipuler cette commande, nous allons premièrement télécharger et ensuite lancer l'**image docker httpd** :

```
docker run -tid --name httpdc -p 80:80 httpdCopier
```

Si vous lancez la commande Docker stats sans argument alors elle vous affichera en temps réel les statistiques de consommation de tous vos conteneurs en cours d'exécution. Exemple :

```
docker statsCopier
```

Résultat :

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %
%	NET I/O	BLOCK I/O	PIDS	
ea4f4c869a2	ubuntuc	0.00%	1.777MiB / 11.61GiB	
0.01%	2.61kB / 0B	0B / 0B	1	
24b9fa633549	httpdc	0.00%	7.082MiB / 11.61GiB	
0.06%	4.01kB / 0B	0B / 0B	82	

Le résultat est sous forme de table, voici ci-dessous une liste d'explication des différentes colonnes de la table de la commande **Docker stats** :

- **CONTAINER ID et Name** : l'identifiant et le nom du conteneur.
- **CPU % et MEM %** : le pourcentage de CPU et de mémoire de l'hôte utilisé par le conteneur.
- **MEM USAGE / LIMIT** : la mémoire totale utilisée par le conteneur et la quantité totale de mémoire qu'il est autorisé à utiliser.
- **NET I/O** : la quantité de données que le conteneur a envoyées et reçues sur son interface réseau.
- **BLOCK I/O** : quantité de données lues et écrites par le conteneur à partir de périphériques en mode bloc sur l'hôte.
- **PIDs** : le nombre de processus ou de threads créés par le conteneur.

Vous pouvez spécifier le nom ou l'id d'un seul ou plusieurs conteneur(s), pour ne visionner que les statistiques propres à vos conteneurs :

```
docker stats httpdcCopier
```

Résultat :

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %
%	NET I/O	BLOCK I/O	PIDS	
24b9fa633549	httpdc	0.00%	7.082MiB / 11.61GiB	
0.06%	4.16kB / 0B	0B / 0B	82	

Stressons un peu notre conteneur **httpdc** avec un script shell en envoyant plusieurs requêtes, en vue de **visualiser l'augmentation de la consommation du conteneur** :

```
#!/bin/bash
```

```
curl_func () {
```



```

    curl -s "http://localhost:80/page{1, 2}.php?[1-1000]" &
}

for i in {1..4}
do
    curl_func
done

wait

echo "All done"

```

En lançant le script sur ma machine hôte, j'ai pu constater une augmentation au niveau de la consommation CPU et du flux réseau du conteneur :

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %
NET I/O	BLOCK	I/O	PIDS	
24b9fa633549	httpdc	25.24%	17.56MiB / 11.61GiB	0.15%
39.8MB / 73.1MB	0B / 0B	136		

Grâce à l'option **--format** ou **-f** vous pouvez formater le résultat de la commande Docker stats de manière à limiter l'affichage du résultat en ne représentant que les ressources souhaitées. Dans cet exemple je ne vais afficher que la consommation CPU et le flux réseau du conteneur **httpdc** sous forme de table :

```

docker stats --format "table {{.Name}}\t{{.CPUPerc}}\t{{.NetIO}}"
httpdc

```

Résultat :

NAME	CPU %	NET I/O
httpdc	19.17%	39.8MB / 73MB

Voici la liste des différents mots réservés pour l'option **--format** de la commande Docker stats :

- **.Container** : Nom ou ID du conteneur (entrée utilisateur).
- **.Name** : Nom du conteneur.
- **.ID** : Identifiant du conteneur.
- **.CPUPerc** : Pourcentage de CPU.
- **.MemUsage** : Utilisation de la mémoire.
- **.NetIO** : Utilisation du flux réseau Entrant/Sortant.
- **.BlockIO** : Utilisation du disque dur en Lecture/Écriture.
- **.MemPerc** : Pourcentage de mémoire (non disponible pour le moment sous Windows).
- **.PIDs** : Nombre de PID (non disponible pour le moment sous Windows).

Autre chose, si vous désirez récupérer par exemple que la valeur de la consommation CPU à l'instant T de votre conteneur dans votre script pour la stocker ensuite dans une variable et donc par la même occasion d'éviter le **résultat en mode streaming**, alors vous pouvez utiliser l'option **--no-stream**, comme suit :

```
docker stats --no-stream --format "{{.CPUPerc}}" httpdcCopier
```

Résultat :

```
24.54%
```

La commande Docker inspect

La commande Docker inspect fournit des informations détaillées sous forme de tableau JSON sur les objets Docker (image Docker, conteneur docker, volume docker, etc ...). Nous avons déjà eu l'occasion d'utiliser cette commande, mais dans ce chapitre nous allons plus nous intéresser à la **la partie filtrage de résultat de la commande Docker inspect**.

Si vous tentez de lancer cette commande sur un conteneur (par exemple sur le conteneur **httpdc** créé précédemment), vous allez alors récupérer beaucoup trop d'informations :

```
docker inspect httpdcCopier
```

Résultat (je n'affiche pas tout car c'est vraiment long) :

```
[
```

```
{  
  "Id":  
    "24b9fa6335492cb968c000a4221bcb1d503a4befc4cb8770171f5345a350cdca",  
  "Created": "2019-07-12T07:29:54.532971612Z",  
  ...  
  "EndpointID":  
    "290ebd6de9ab4f3139ee93d49462a6ca692a1f18bac7888e14e5f4ebaab26aad",  
  "Gateway": "172.17.0.1",  
  "GlobalIPv6Address": "",  
  "GlobalIPv6PrefixLen": 0,  
  "IPAddress": "172.17.0.2",  
  "IPPrefixLen": 16,  
  "IPv6Gateway": "",  
  "MacAddress": "02:42:ac:11:00:02",  
  "Networks": {  
    "bridge": {  
      "IPAMConfig": null,  
      "Links": null,  
      "Aliases": null,  
      "NetworkID":  
        "5220c685dc3d77ba5547fd853e055a66f6acffd9cc2f57acde61a6e1264ae9db",  
      "EndpointID":  
        "290ebd6de9ab4f3139ee93d49462a6ca692a1f18bac7888e14e5f4ebaab26aad",  
      "Gateway": "172.17.0.1",
```

```

        "IPAddress": "172.17.0.2",

        "IPPrefixLen": 16,

        "IPv6Gateway": "",

        "GlobalIPv6Address": "",

        "GlobalIPv6PrefixLen": 0,

        "MacAddress": "02:42:ac:11:00:02",

        "DriverOpts": null

    }

}

}

}

```

]Copier

Pour amincir le résultat, nous allons une nouvelle fois utiliser l'option de formatage **--format** ou **-f**. Nous allons commencer par récupérer les sous-éléments de l'élément **State** de la façon suivante :

```
docker inspect --format='{{json .State}}' httpdcCopier
```

Résultat :

```

{"Status":"running","Running":true,"Paused":false,"Restarting":false,
"OOMKilled":false,"Dead":false,"Pid":18532,"ExitCode":0,"Error":"","
"StartedAt":"2019-07-12T07:29:55.132385698Z","FinishedAt":"0001-01-
01T00:00:00Z"}Copier

```

Je vais utiliser la **bibliothèque json de python** (installé par défaut sur Linux) afin d'avoir un affichage plus joli :

```
docker inspect --format='{{json .State}}' httpdc | python3 -m
json.toolCopier
```

Résultat :

```
{
  "Status": "running",
  "Running": true,
  "Paused": false,
  "Restarting": false,
  "OOMKilled": false,
  "Dead": false,
  "Pid": 18532,
  "ExitCode": 0,
  "Error": "",
  "StartedAt": "2019-07-12T07:29:55.132385698Z",
  "FinishedAt": "0001-01-01T00:00:00Z"
}
```

Copier

Essayons d'aller plus loin en récupérant sous un format texte l'élément **Status** de l'élément **State**. De cette manière vous pouvez par exemple **stocker le résultat dans une variable de votre script** :

```
docker inspect --format='{{ .State.Status}}' httpdcCopier
```

Résultat :

```
running
```

Allons encore plus en profondeur et tentons de **récupérer les mappages de port d'un conteneur**. Premièrement, on va créer un conteneur utilisant différents mappages de port :

```
docker run -tid --name ubuntuuc -p 9000:8000 -p 9001:8001 -p 9002:8002 ubuntuCopier
```

Nous allons ensuite inspecter notre conteneur afin de **récupérer les informations concernant les ports** :

```
docker inspect --format='{{json .NetworkSettings.Ports}}' ubuntuuc | python3 -m json.tool
```

Nous récupérons ainsi le tableau JSON suivant :

```
{
  "8000/tcp": [
    {
      "HostIp": "0.0.0.0",
      "HostPort": "9000"
    }
  ],
  "0001/tcp": [
    {
      "HostIp": "0.0.0.0",
      "HostPort": "9001"
    }
  ],
  "8002/tcp": [
    {
      "HostIp": "0.0.0.0",
```

```

        "HostPort": "9002"
    }
]
}Copier

```

Pour récupérer des informations d'un tableau JSON depuis l'option **--format**, il faut au préalable utiliser le mot-clé **range**. Dans cet exemple nous allons d'abord récupérer la clé de chaque élément du tableau dans une variable nommée **\$p**, cette clé correspondant à tous les ports exposés par votre conteneur, ensuite nous allons aussi récupérer la valeur de chaque clé dans une variable nommée **\$conf** :

```

docker inspect --format='{{range $p, $conf :=
.NetworkSettings.Ports}}{{println "clé :" $p "=>" " | valeur :"$
$conf}}{{end}}' ubuntuCopier

```

Résultat :

```

clé : 8000/tcp => | valeur : [{0.0.0.0 9000}]
clé : 8001/tcp => | valeur : [{0.0.0.0 9001}]
clé : 8002/tcp => | valeur : [{0.0.0.0 9002}]

```

Maintenant gardons la variable **\$p** et essayons de ne récupérer que la deuxième valeur de la variable **\$conf** correspondant au port cible mapper :

```

docker inspect --format='{{range $p, $conf :=
.NetworkSettings.Ports}}{{println "Port exposé :" $p " | Port cible
:" (index $conf 0).HostPort}}{{end}}' ubuntuCopier

```

Résultat :

```

Port exposé : 8000/tcp | Port cible : 9000
Port exposé : 8001/tcp | Port cible : 9001
Port exposé : 8002/tcp | Port cible : 9002

```

la commande logs

Il y a des risques que votre conteneur soit constamment à l'état **RESTART**. Dans ce cas il est important de **vérifier les logs de votre conteneur**.

Pour nos tests, nous allons construire une image ou j'ai rajouté exprès une erreur :

```
FROM alpine:latest

RUN apk add --no-cache apache2

EXPOSE 80

ENTRYPOINT /usr/sbin/http -DFOREGROUND
```

Buildons ensuite notre image :

```
docker build -t alpineerror .
```

Démarrons subséquemment notre conteneur avec les options suivantes :

```
docker run -d --restart always --name alpineerrorc -p 80:80
alpineerror
```

Si on vérifie l'état de notre conteneur, on constatera alors qu'il essaiera toujours de redémarrer mais sans aucun succès :

```
docker ps
```

Résultat :

CONTAINER ID	IMAGE	COMMAND	PORTS
CREATED	STATUS		
NAMES			


```
31e4baf228c8      alpineerror      "/bin/sh -c '/usr/sb..."
About a minute ago  Restarting (127) 3 seconds ago
alpineerror
```

Vérifions ensuite les logs du conteneur afin de trouver la source du problème :

```
docker logs alpineerrorcCopier
```

Résultat :

```
/bin/sh: /usr/sbin/http: not found
```

Les logs nous indiquent clairement que le chemin de la commande est introuvable. Pour corriger cette erreur il suffit juste de remplacer `/usr/sbin/http` par `/usr/sbin/httpd` dans votre Dockerfile.

la commande history

Dans certains cas il est nécessaire de **diagnostiquer la construction de votre image** en vue de l'optimiser, voire la réparer. Dans ce cas la commande Docker history vous sera d'une grande aide car elle vous indiquera les couches individuelles qui constituent votre image, ainsi que les commandes qui les ont créées, leur taille, et leur temps d'exécution.

Pour utiliser cette commande, nous allons d'abord construire une nouvelle image :

Dockerfile :

```
FROM alpine:latest

RUN apk add --no-cache git

RUN apk add --no-cache mysql-clientCopier
```

Buildons ensuite notre image :

```
docker build -t myalpine .Copier
```

Exécutons maintenant la commande Docker History :

```
docker history myalpineCopier
```

Résultat :

IMAGE SIZE	CREATED COMMENT	CREATED BY
e0fee090997d cache mysql-client	4 seconds ago 35.5MB	/bin/sh -c apk add --no-
0c9541183535 cache git	7 seconds ago 15.6MB	/bin/sh -c apk add --no-
b7b28af77ffe ["/bin/sh"]	18 hours ago 0B	/bin/sh -c #(nop) CMD
alt;missing> file:0eb5ea35741d23fe3...	18 hours ago 5.58MB	/bin/sh -c #(nop) ADD

Par défaut vous n'avez pas d'horodatage de la création de l'image, cette information peut vous être utile si vous souhaitez examiner le temps d'exécution de chaque couche de votre image. Pour afficher cette information, il faut une nouvelle fois utiliser l'option **--format** avec le mot réservé **.CreatedAt** :

Avant de l'utiliser, laissez-moi d'abord vous décrire les différents mots réservés de l'option **--format** pour la commande Docker history. Par la même occasion ça vous permettra d'en savoir davantage sur chaque colonne retournée par la table de cette commande :

- **.ID** : ID de la couche de l'image.
- **.CreatedSince** : Temps écoulé depuis la création de la couche de l'image.
- **.CreatedAt** : Horodatage de la création de la couche de l'image.
- **.CreatedBy** : Commande utilisée pour créer la couche de l'image.
- **.Size** : Taille du disque de la couche de l'image.
- **.Comment** : Commentaire de la couche de l'image.

Dans l'exemple ci-dessous nous allons dans un premier temps afficher la commande utilisée par nos différentes couches de l'image **myalpine**, et dans un second temps révéler l'horodatage de création pour chaque couche de cette même image.

```
docker history -H --format="table  
{ {.ID}}\t{ {.CreatedBy}}\t{ {.CreatedAt}}" myalpineCopier
```

Résultat :

```
IMAGE          CREATED BY
CREATED AT

e0fee090997d   /bin/sh -c apk add --no-cache mysql-client
2019-07-12T18:14:46+02:00

0c9541183535   /bin/sh -c apk add --no-cache git
2019-07-12T18:14:43+02:00

b7b28af77ffe   /bin/sh -c #(nop)  CMD ["/bin/sh"]
2019-07-12T00:20:52+02:00

&tl;missing>   /bin/sh -c #(nop) ADD file:0eb5ea35741d23fe3...
2019-07-12T00:20:52+02:00
```

Exercice

Énoncé

Il est temps de pratiquer un peu ! Je vous ai préparé **un petit exercice qui reprend la base de tout ce qu'on a pu étudier** depuis le début de ce chapitre.

Le but de cet exercice est de créer un script qui affiche la configuration réseau basique et les ports mappés de tous les conteneurs de votre machine locale (peu importe leur état).

Voici un aperçu du résultat final :

```
ubuntuc :

IP : 172.17.0.3/16

MacAddress : 02:42:ac:11:00:03

Gateway : 172.17.0.1

Ports :

- 8000:9000

- 8001:9001
```

```
- 8002:9002
```

```
httpdc :
```

```
IP : 172.17.0.2/16
```

```
MacAddress : 02:42:ac:11:00:02
```

```
Gateway : 172.17.0.1
```

```
Ports :
```

```
- 80:80
```

Vous pouvez utiliser n'importe quel langage de programmation. En ce qui me concerne, j'ai utilisé un script bash.

Solution

J'espère que vous avez réussi à réaliser ce tp ! Je vous présente ici ma solution. Bien sûr, je ne détiens pas la meilleure solution donc n'hésitez à partager votre code dans les commentaires ou sur le serveur discord.

```
#!/bin/bash
```

```
# Récupération des noms des conteneurs
```

```
containers=$(docker ps -a --format={{.Names}})
```

```
for container in $containers
```

```
do
```

```
# Récupération des informations du conteneur
```

```
IP=$(docker inspect --format='{{.NetworkSettings.IPAddress}}'
$container)

IPp=$(docker inspect --format='{{.NetworkSettings.IPPrefixLen}}'
$container)

MACADDR=$(docker inspect --
format='{{.NetworkSettings.MacAddress}}' $container)

GATEWAY=$(docker inspect --format='{{.NetworkSettings.Gateway}}'
$container)

PORTS=$(docker inspect --format='{{range $p, $conf :=
.NetworkSettings.Ports}}{{println "\t  -" $p ":" (index $conf
0).HostPort}}{{end}}' $container | sed -e 's/ : /:/')

# Affichage des informations du conteneur

echo -e "$container :"

echo -e "\tIP : $IP/$IPp"

echo -e "\tMacAddress : $MACADDR"

echo -e "\tGateway : $GATEWAY"

echo -e "\tPorts : \n${PORTS//\n/tcp/}\n"

doneCopier
```

Conclusion

Il est temps de conclure ce chapitre. Dans les chapitres précédents, nous avons vu comment déboguer quelques objets docker comme les volumes et les réseaux Docker, mais cette fois-ci nous nous sommes plus concentré sur le débogage des conteneurs et images Docker.

Comme toujours, je vous partage un pense-bête des commandes que nous avons pu voir dans ce chapitre :

```
## Récupérer des informations de bas niveau d'un conteneur ou d'une image
```

```
docker inspect <CONTAINER_ID ou CONTAINER_NAME ou IMAGE_NAME ou IMAGE_ID>
```

```
-f ou --format : formater le résultat
```

```
## Afficher en temps réels les statistiques des différentes
```

```
## ressources consommées par votre conteneur en mode streaming
```

```
docker stats <CONTAINER_ID ou CONTAINER_NAME>
```

```
-f ou --format : formater le résultat
```

```
--no-stream : désactiver le mode streaming
```

```
## Visualiser des informations sur les différentes couche de votre image
```

```
docker history <IMAGE_NAME ou IMAGE_ID>
```

```
-f ou --format : formater le résultat
```

```
## Examiner les logs 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")Copier
```

[Chapitre précédent](#)

Déployer et gérer vos hôtes docker avec Docker Machine

Dans cet article vous allez apprendre à créer, démarrer, inspecter, arrêter, redémarrer, mettre à jour des hôtes Docker depuis le Docker Machine. Nous allons utiliser des drivers de plateforme de gestion de virtualisation locale mais aussi des fournisseurs de Cloud.

Introduction

Docker Machine est un **outil de provisioning et de gestion des hôtes Docker** (hôtes virtuels exécutant le moteur Docker). Vous pouvez utiliser Docker Machine pour créer des hôtes Docker sur votre ordinateur personnel ou sur le datacenter de votre entreprise à l'aide d'un logiciel de virtualisation tel que VirtualBox ou VMWare, vous pouvez aussi déployer vos machines virtuelles chez des fournisseurs de cloud, tels que Azure, AWS, Google Compute Engine, etc ..

À l'aide de la commande `docker-machine`, vous pouvez démarrer, inspecter, arrêter et redémarrer un hôte géré ou mettre à niveau le client et le moteur Docker et configurer un client Docker pour qu'il puisse communiquer avec votre hôte. En bref il **crée automatiquement des hôtes Docker**, y **installe le moteur Docker**, puis **configure les clients docker**.

Installation de Docker Machine

Voici la **commande** qui permet d'installer **Docker Machine** sous **Linux**.

```
base=https://github.com/docker/machine/releases/download/v0.16.0 &&

curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/tmp/docker-
machine &&

sudo install /tmp/docker-machine /usr/local/bin/docker-machineCopier
```

Pour pouvoir **activer l'auto-completion des commandes Docker Machine**, il suffit de créer un script qu'on va nommer dans le dossier `/etc/bash_completion.d` ou dans le dossier `/usr/local/etc/bash_completion.d` et de coller dedans le contenu ci-dessous :

```
sudo nano /etc/bash_completion.d/docker-machine-prompt.bashCopier

base=https://raw.githubusercontent.com/docker/machine/v0.16.0

for i in docker-machine-prompt.bash docker-machine-wrapper.bash
docker-machine.bash
do

    sudo wget "$base/contrib/completion/bash/${i}" -P
/etc/bash_completion.d

doneCopier
```

Enfin, il faut lancer la commande `source` pour charger votre script d'auto-completion :

```
source /etc/bash_completion.d/docker-machine-prompt.bashCopier
```

Pour ceux qui ont installé Docker sur une **machine Windows** pro avec HyperV d'activé, il n'y'a pas besoin d'installation car Docker machine est installé par défaut.

Découverte des drivers et des commandes Docker Machine

Docker machine utilise le concept des **drivers** (en fr : pilotes). Les drivers vous permettent depuis votre Docker machine. de créer un ensemble complet de ressources sur vos machines virtuelles sur des services tiers tels qu'Azure, Amazon, VirtualBox, etc. Vous retrouverez la liste des différents drivers [ici](#).

Avant de vous décrire l'utilisation de certains drivers. Voici d'abord **la commande qui permet de créer une machine virtuelle depuis votre Docker Machine :**

```
docker-machine create --drive <DRIVER NAME> <MACHINE NAME>Copier
```

La commande `docker-machine create` **télécharge une distribution Linux légère nommée [boot2docker](#)** venant avec le moteur Docker installé et crée et démarre la machine virtuelle. Les options de cette commande peuvent différer selon le type de driver que vous utilisez.

Nous allons voir ci-dessous comment créer des hôtes Docker onpremise avec le driver virtualBox et hyperv mais aussi dans le Cloud avec le driver d'AWS (Amazon Web Service) nommé "amazonec2".

Découverte du pilote VirtualBox et utilisation des commandes Docker Machine

Je suis actuellement sous Linux avec la distribution **Fedora 30**. Si vous êtes sous une autre distribution alors la configuration risque d'être un peu différente. Dans tous les cas, voici la **configuration requise pour le driver VirtualBox :**

- Virtualbox à partir de la version 5
- Le module de noyau vboxdrv

Je vous fais confiance pour l'installation de VirtualBox, mais si jamais vous rencontrer des problèmes, alors n'hésitez pas à m'en faire part dans l'espace commentaire, il est prévu pour ça ☐. En ce qui me concerne j'ai téléchargé VirtualBox en version 6.0.10.

Pour **installer le module de noyau vboxdrv**, il faut au préalable **installer le package kernel-devel**. Pour information le package kernel-devel télécharge les fichiers d'en-têtes du noyau Linux qui vont permettre aux développeurs d'accéder aux différentes fonctionnalités du noyau. De façon plus simple, il est nécessaire au développement et à la compilation de pilotes. Et c'est exactement ce qu'il nous faut !

```
sudo dnf -y install kernel-devel
```

Attention

Pour ceux qui sont sous une autre distribution, veuillez chercher l'équivalent de ce package sur votre distribution.

Lancez ensuite la commande suivante pour **installer le module vboxdrv** :

```
sudo /sbin/vboxconfig
```

Résultat :

```
vboxdrv.sh: Stopping VirtualBox services.  
  
vboxdrv.sh: Starting VirtualBox services.  
  
vboxdrv.sh: Building VirtualBox kernel modules.g
```

Une fois les deux prérequis de configurations satisfaites, vous pouvez dès lors créer votre hôte Docker en lançant la commande **create** en utilisant le driver **virtualbox** avec les options par défaut :

```
docker-machine create --driver virtualbox vbox-test
```

Résultat :

```
Running pre-create checks...  
  
(vbox-test) Image cache directory does not exist, creating it at  
/home/hatim/.docker/machine/cache...  
  
(vbox-test) No default Boot2Docker ISO found locally, downloading  
the latest release...  
  
(vbox-test) Latest release for github.com/boot2docker/boot2docker is  
v18.09.7  
  
(vbox-test) Downloading  
/home/hatim/.docker/machine/cache/boot2docker.iso from  
https://github.com/boot2docker/boot2docker/releases/download/v18.09.  
7/boot2docker.iso...
```

```
(vbox-test)
0%....10%....20%....30%....40%....50%....60%....70%....80%....90%...
.100%

Creating machine...

...

Checking connection to Docker...

Docker is up and running!

To see how to connect your Docker Client to the Docker Engine
running on this virtual machine, run: docker-machine env vbox-test
```

Ensuite, vérifiez la liste des machines Docker disponible en exécutant la commande suivante :

```
docker-machine ls
```

Résultat :

NAME	ACTIVE	DRIVER	STATE	URL
SWARM	DOCKER	ERRORS		
vbox-test	-	virtualbox	Running	
				tcp://192.168.99.100:2376
				v18.09.7

D'après le résultat, notre hôte **vbox-test** est bien présent avec l'état **Running** et possède le moteur docker en version v18.09.7.

Si vous retournez à la fin du résultat de la commande `docker-machine create`, vous remarquerez le message suivant (traduit en français) : "Pour voir comment connecter Docker à cette machine, exécutez: `docker-machine env vbox-test`". Cette manipulation, va nous permettre de **recupérer les variables d'environnements de la nouvelle VM à exporter**. Parfait, utilisons la alors :

```
docker-machine env vbox-test
```

Résultat :

```
export DOCKER_TLS_VERIFY="1"
```

```
export DOCKER_HOST="tcp://192.168.99.100:2376"

export DOCKER_CERT_PATH="/home/hatim/.docker/machine/machines/vbox-
test"

export DOCKER_MACHINE_NAME="vbox-test"

# Run this command to configure your shell:

# eval $(docker-machine env vbox-test)
```

Le résultat nous indique clairement que si on souhaite **utiliser le moteur Docker de la machine virtuelle sur notre shell courant** il faut alors utiliser la commande suivante :

```
eval $(docker-machine env vbox-test)Copier
```

En exécutant cette commande sur votre shell courant, alors n'importe quelle commande Docker que vous exécuterez, sera dorénavant directement prise en compte par votre hôte Docker **vbox-test** et non plus par votre hôte maître.

Par ailleurs si vous souhaitez **vérifier sur quelle hôte Docker se lanceront vos prochaines commandes docker** alors soit vous vérifiez si une étoile existe dans la colonne **ACTIVE** de la commande `docker-machine ls`. Soit plus simple encore, vous lancez la commande suivante :

```
docker-machine activeCopier
```

Résultat :

```
vbox-test
```

Le résultat nous indique distinctement, que nos futurs commandes docker sur le shell courant s'exécuteront directement sur la machine Docker **vbox-test**.

Afin de vous prouver que c'est effectivement le cas, je vais télécharger et exécuter l'image [httpd](#) sur le shell courant :

```
docker run -d -p 8000:80 --name vbox-test-httpd httpdCopier
```

À présent, ouvrez un nouveau terminal et vérifiez les conteneurs disponibles, vous verrez ainsi que vous ne retrouverez pas le conteneur **vbox-test-httpd** créé précédemment :

```
docker ps
```

Résultat :

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES					

Cependant si je retourne sur mon ancien shell avec la machine docker **vbox-test** activée, dans ce cas j'obtiendrai bien un résultat avec le conteneur **vbox-test-httpd** :

```
docker ps
```

Résultat :

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
347723c8291f	httpd	"httpd-foreground"	3
minutes ago	Up 3 minutes	0.0.0.0:8000->80/tcp	vbox-
test-httpd			

Vous n'êtes pas encore convaincu ? Alors **connectez vous carrément à la machine Docker** à l'aide de la commande suivante :

```
docker-machine ssh vbox-test
```

Résultat :

```
( '>' )

/) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.

(/-_-_-\)      www.tinycorelinux.net

docker@vbox-test:~$
```

Revérifiez une nouvelle fois la liste de vos conteneurs et vous verrez que le conteneur **vbox-test-httpd** est bien dedans :

```
docker ps
```

Résultat :

CONTAINER ID STATUS	IMAGE PORTS	COMMAND NAMES	CREATED
347723c8291f minutes ago test-httpd	httpd Up 4 minutes	"httpd-foreground" 0.0.0.0:8000->80/tcp	3 vbox-

Pour vous assurer que le client Docker est automatiquement configuré au début de chaque session de shell, vous pouvez alors intégrer la commande `eval $(docker-machine env vbox-test)` votre fichier `~/.bash_profile`.

Si vous pensez avoir fini d'utiliser une machine Docker, vous pouvez l'arrêter avec la commande `docker-machine stop` et la redémarrer plus tard avec la commande `docker-machine start`, exemple :

```
docker stop vbox-testCopier  
  
docker start vbox-testCopier
```

Enfin, vous pouvez surcharger les ressources allouées automatiquement par défaut à hôte Docker en utilisant les options venant avec le driver virtualbox. Dans cet exemple je vais créer une machine Docker avec 30 Go d'espace disque (20 Go par défaut) et avec 2 Go de ram (1Go par défaut) et

```
docker-machine create -d virtualbox \  
  
--virtualbox-disk-size "30000" \  
  
--virtualbox-memory "4000" \  
  
vbox-test-biggerCopier
```

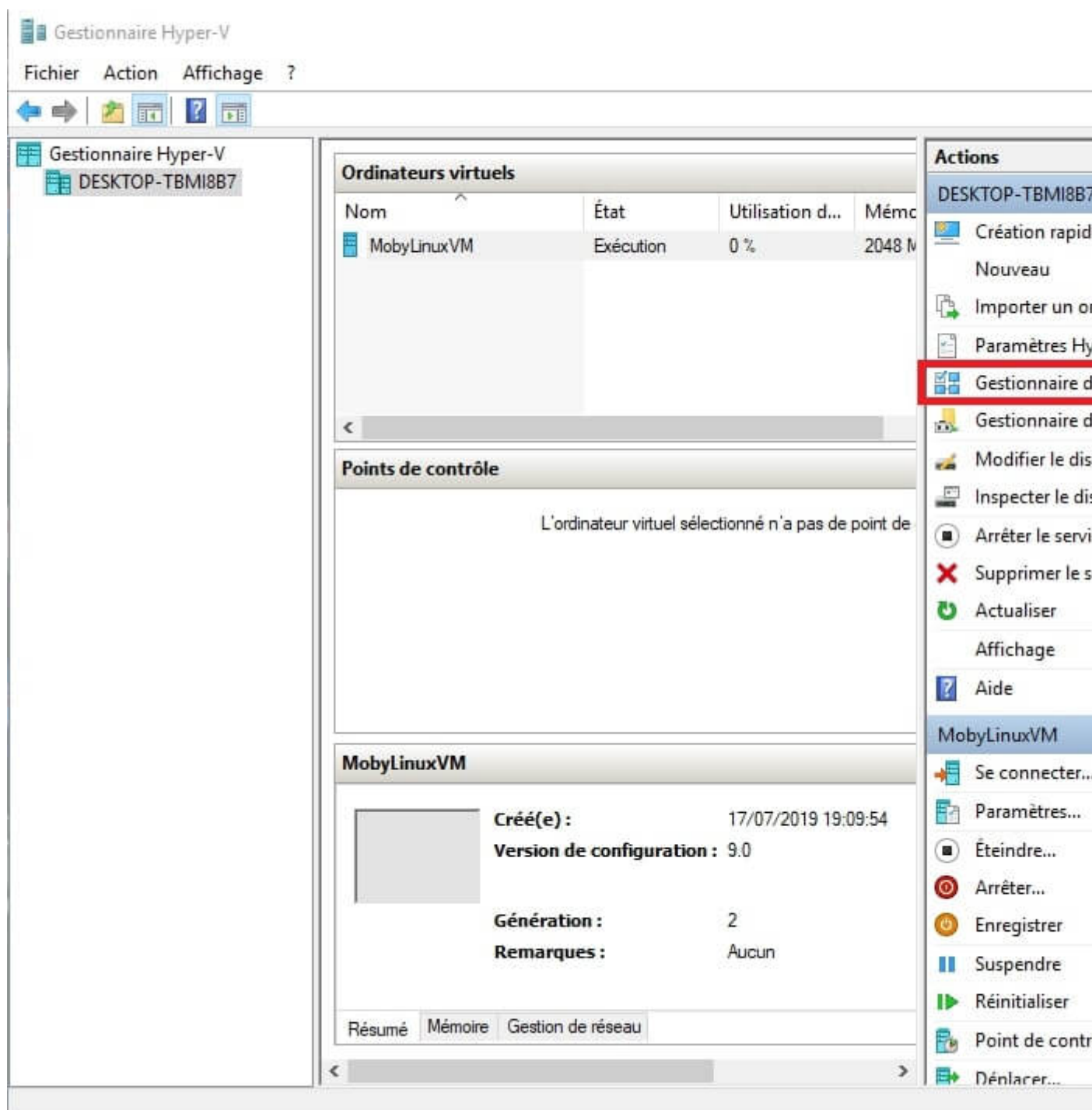
Vous retrouverez plus d'informations sur les options liées au driver virtualbox [ici](#).

Le pilote HyperV

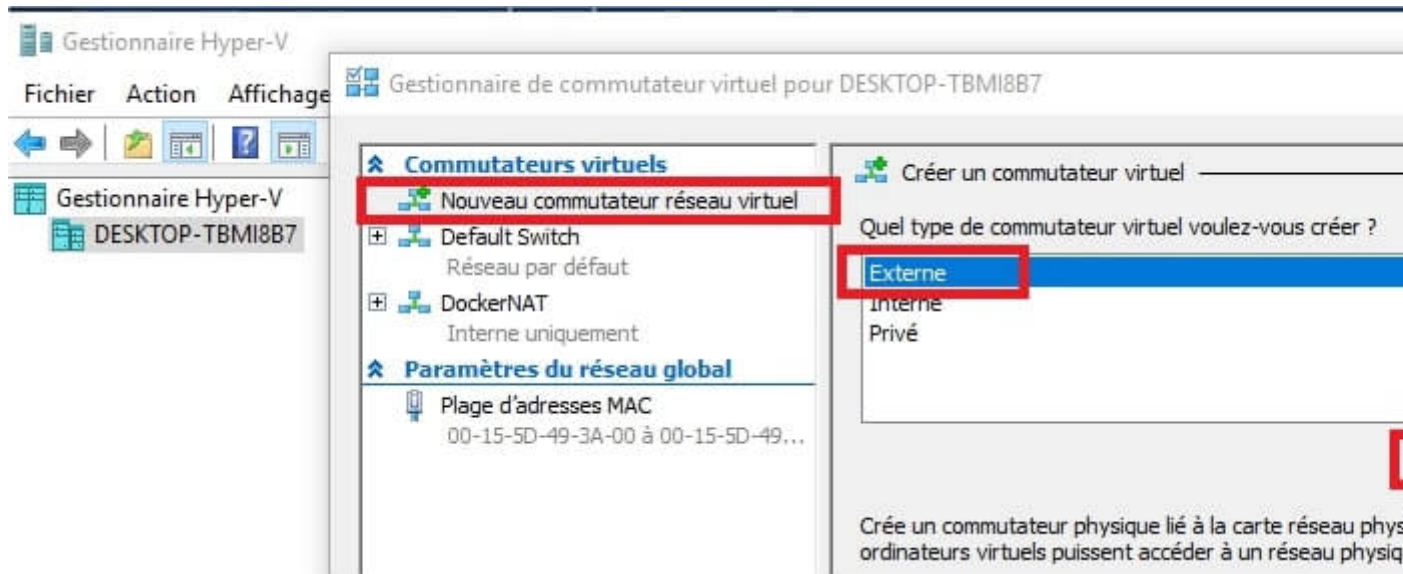
À cet instant je suis passé sur ma machine Windows afin d'utiliser le pilote hyperv et par la suite créer une machine docker basée sur ce driver.

Si vous avez déjà un switch réseau externe, ignorez cette configuration et allez voir directement la commande utilisant le driver hyperv. Mais si ce n'est pas le cas alors il faut en créer un en suivant les instructions suivantes :

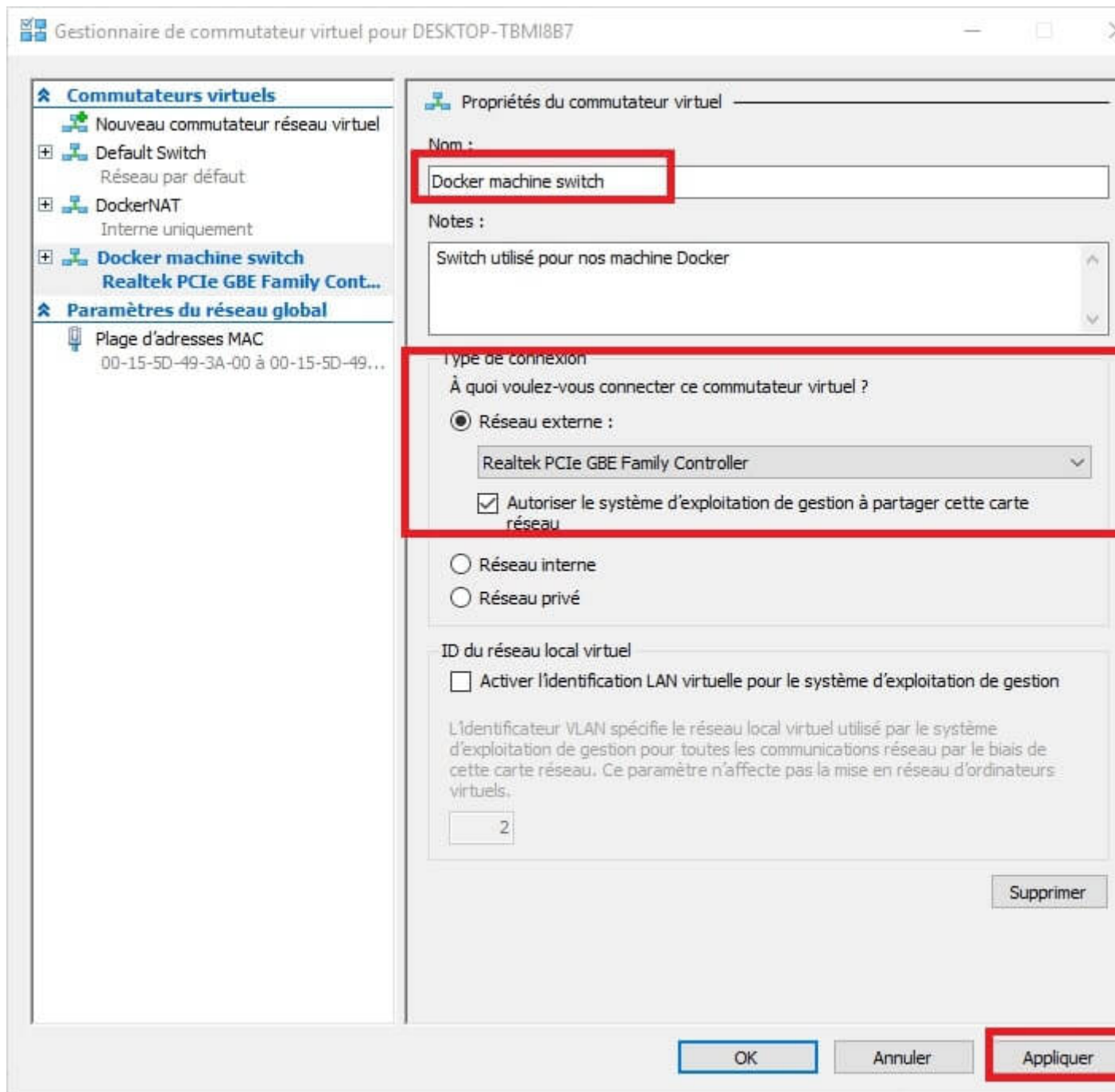
Ouvrez le gestionnaire Hyper-V et sélectionnez le "Gestionnaire de commutateur virtuel" dans le panneau d'actions de droite :



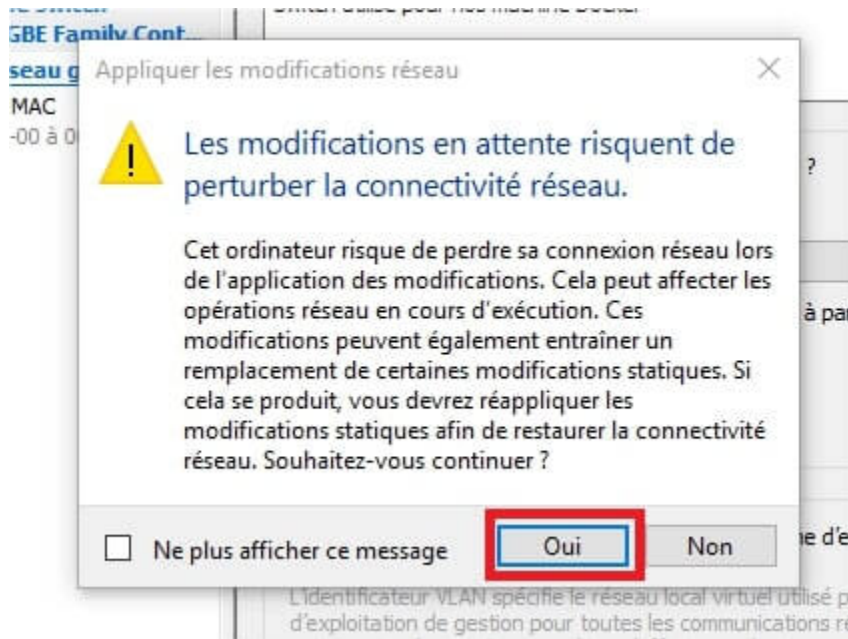
Ensuite, **configurez un nouveau switch réseau externe** à utiliser à la place du switch réseau DockerNAT :



Pour cet exemple, nous allons créer un switch virtuel appelé "Docker machine switch".



Ignorer l'avertissement en appuyant sur le bouton "oui".



Une fois le switch externe créé, on peut enfin l'utiliser pour déployer notre machine Docker depuis le driver hyperv. Lancez un powershell en mode administrateur et exécuter la commande suivante :

```
docker-machine create --driver=hyperv --hyperv-virtual-switch  
"Docker machine switch" hyperv-testCopier
```

Résultat :

```
Running pre-create checks...  
  
Creating machine...  
  
(hyperv-test) Copying  
C:\Users\hatim\.docker\machine\cache\boot2docker.iso to  
C:\Users\hatim\.docker\machine\machines\hyperv-  
test\boot2docker.iso...  
  
(hyperv-test) Creating SSH key...  
  
(hyperv-test) Creating VM...  
  
(hyperv-test) Using switch "Docker machine switch"  
  
...  
  
Checking connection to Docker...
```

```
Docker is up and running!
```

```
To see how to connect your Docker Client to the Docker Engine  
running on this virtual machine, run: C:\Program  
Files\Docker\Docker\Resources\bin\docker-machine.exe env hyperv-test
```

Vérifions ensuite la liste des hôtes Docker :

```
docker-machine lsCopier
```

Résultat :

NAME	ACTIVE	DRIVER	STATE	URL
SWARM	DOCKER	ERRORS		
hyperv-test v18.09.7	-	hyperv	Running	tcp://192.168.0.19:2376

Nous voyons bien notre nouvelle machine Docker **hyperv-test** avec le moteur Docker en version v18.09.7.

Le pilote amazonec2 (cloud)

Pour changer un peu du déploiement local, nous utiliserons cette fois-ci un service cloud, plus précisément nous utiliserons le driver amazonec2 qui va nous permettre de créer des machines sur le service cloud AWS (Amazon Web Services).

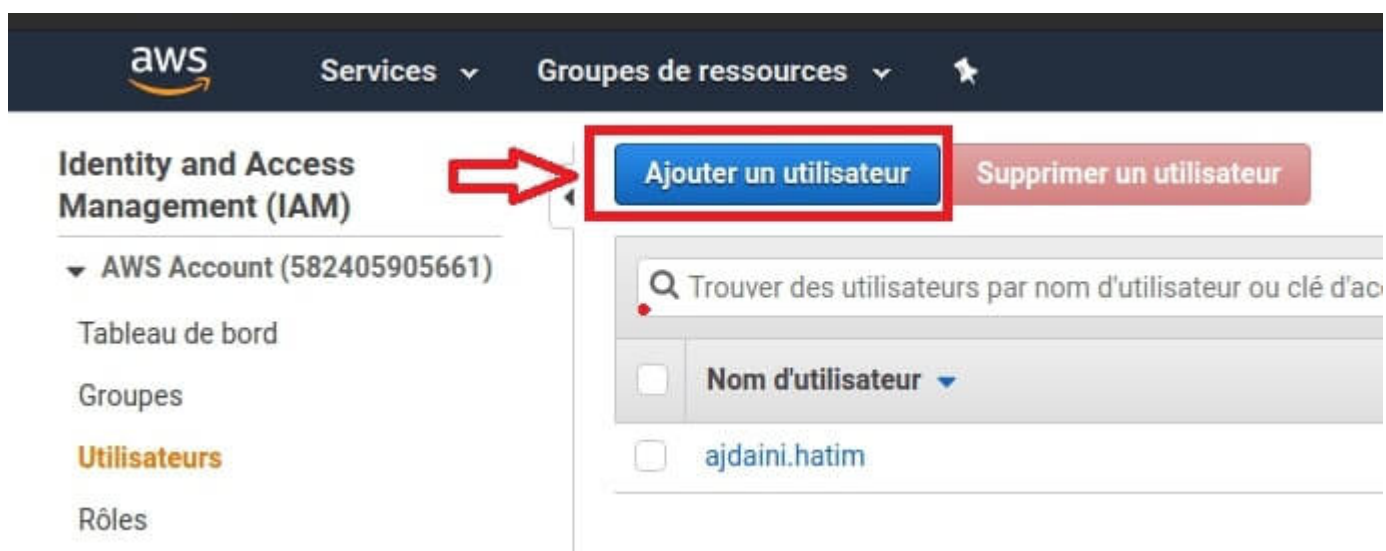
Pour créer des machines sur AWS , vous devez fournir deux paramètres :

- un ID de clé d'accès AWS
- une clé d'accès secrète AWS

Rendez-vous dans le service IAM depuis votre console AWS :



Une fois rendu dans le service IAM, si vous n'avez pas encore d'utilisateur, créez-en un en appuyant sur le bouton "Ajouter un utilisateur" :



Après cela, cliquez sur le bouton "Créer une clé d'accès" :

Autorisations

Groupes (1)

Balises

Informations d'identification de sécurité

Access Ad


Informations d'identification de connexion

Récapitulatif

Mot de passe de la console


Appareil MFA attribué

Certificats de signature

• Lien de connexion à la console : 

Activé (ne s'est jamais connecté) | [Gest](#)

Non attribué | [Gestion](#)

Aucun 

Clés d'accès

Utilisez les clés d'accès pour effectuer des demandes de protocole REST ou HTTP Query sécurisées vers l'API AWS à une rotation fréquente des clés. [En savoir plus](#)

Créer une clé d'accès

ID de clé d'accès	Créé	Dernière utilisation
-------------------	------	----------------------

Clé SSH pour AWS CodeCommit

Récupérer maintenant votre ID et clé d'accès secrète de votre compte AWS :

Créer une clé d'accès

✓

Opération réussie

C'est la **seule** fois que les clés d'accès secrètes pourront être consultées ou téléchargées. Vous n'avez pas les récupérer plus tard. Cependant, vous pouvez créer de nouvelles clés d'accès à tout moment.

Téléchargez le fichier .csv

ID de clé d'accès	Clé d'accès secrète
AKIAYPGQY3T63DHEKXKA	***** <div>Afficher</div>

Configurer les informations d'identification en utilisant le fichier d'informations d'identification standard du fichier Amazon AWS `~/.aws/credentials`, de sorte que vous n'ayez plus besoin de les saisir à chaque fois que vous exécutez la commande **create**. Voici un exemple du fichier d'identification :

```
[default]

aws_access_key_id = AKIAYPGQYET63DHEKXKA

aws_secret_access_key = VOTRE-CLE-SECRETECopier
```

On peut dès à présent créer notre instance EC2 (VM aws) depuis notre machine maître Docker. Dans cet exemple nous allons utiliser la région us-west-1 et autoriser le port 8000 dans le security groupe (firewall AWS) lié à l'instance EC2 :

```
docker-machine create --driver amazonec2 --amazonec2-open-port 8000
--amazonec2-region us-west-1 aws-testCopier
```

Résultat :

```
Running pre-create checks...
```

```
Creating machine...

(aws-test) Launching instance...

Waiting for machine to be running, this may take a few minutes...

Detecting operating system of created instance...

Waiting for SSH to be available...

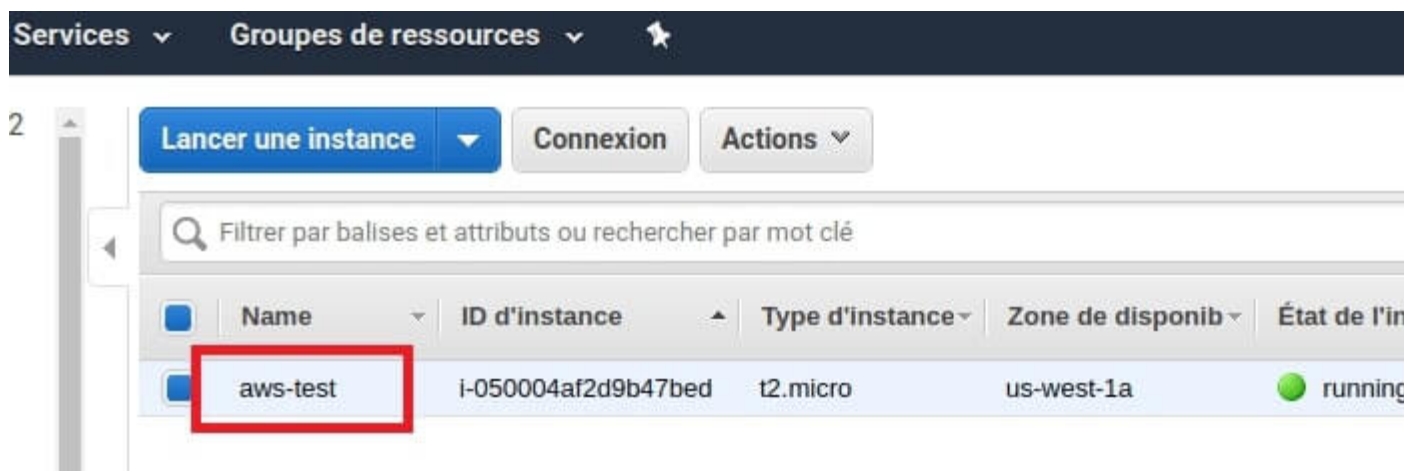
...

Checking connection to Docker...

Docker is up and running!

To see how to connect your Docker Client to the Docker Engine
running on this virtual machine, run: docker-machine env aws-test
```

Nous allons vérifier subséquemment sur notre console si notre instance EC2 est bien présente dessus. Pour cela, rendez-vous dans le service EC2 et assurez-vous d'être bien dans la même région que celle lancée dans la commande **create** :



Cool, notre instance EC2 **aws-test** est bien présente. Maintenant vérifions si le port 8000 est bien autorisé dans le security group lié à cette instance :

The screenshot shows the AWS Management Console interface for a Security Group. At the top, there are buttons for 'Create Security Group' and 'Actions'. Below is a search bar with the text 'Filter by tags and attributes or search by keyword'. A table lists security groups with columns: Name, Group ID, Group Name, and VPC ID. The first row, 'docker-machine' with Group ID 'sg-005a887ab49570325', is highlighted with a red box. Below the table, the details for 'Security Group: sg-005a887ab49570325' are shown, with tabs for 'Description', 'Inbound', 'Outbound', and 'Tags'. The 'Inbound' tab is selected. An 'Edit' button is visible. A table of inbound rules is shown with columns 'Type' and 'Protocol'. The first rule, 'Custom TCP Rule' with 'TCP' protocol, is highlighted with a red box. Other rules include 'SSH' and another 'Custom TCP Rule', all with 'TCP' protocol.

Name	Group ID	Group Name	VPC ID
	sg-005a887ab49570325	docker-machine	vpc-d6e4e0b1
	sg-7526db09	default	vpc-d6e4e0b1

Security Group: sg-005a887ab49570325

Description Inbound Outbound Tags

Edit

Type	Protocol
Custom TCP Rule	TCP
SSH	TCP
Custom TCP Rule	TCP

Le security group autorise bel et bien le port 8000, vous pouvez dès à présent lancer vos conteneurs directement sur votre instance EC2 depuis votre machine maître. Dans cet exemple nous allons instancier l'image [httpd](http://) dans notre nouvelle machine Docker **aws-test**.

Premièrement, nous allons rendre notre hôte Docker **aws-test** active :

```
eval $(docker-machine env aws-test)Copier
```

Deuxièmement, nous allons télécharger et exécuter notre image httpd

```
docker run -d -p 8000:80 --name httpdc httpdCopier
```

Si vous visitez la page http://VOTRE_IP:8000, vous observerez alors le message "It works!".

Supprimer vos machines Docker

Comme je n'ai plus besoin de mes machines, je peux alors les supprimer. Pour ce faire, je vais utiliser la commande `docker-machine rm <MACHINE NAME>`. Cette commande aura pour effet de **supprimer définitivement la machine Docker** de votre plateforme de gestion de virtualisation locale mais aussi de la supprimer de votre fournisseur de cloud, si jamais vous en utilisez un.

```
docker-machine rm -f aws-testCopier  
  
docker-machine rm -f vbox-testCopier
```

Conclusion

Nous avons utilisé Docker Machine pour créer des hôtes Docker localement mais aussi dans le Cloud, cela nous montre à quel point il est facile de déployer et des machines Docker n'importe où et de centraliser la gestion de ces VMs depuis une seule machine maître. Comme à mon habitude, je partage avec vous un **aide-mémoire résumant les différentes commandes de Docker Machine**.

```
## Créer une machine Docker  
  
docker-machine create -d <DRIVER NAME> <MACHINE NAME>  
  
    -d ou --driver : choisir un driver  
  
## Rendre une machine Docker active  
  
eval $(docker-machine env <MACHINE NAME>)  
  
# Lister les machines Docker  
  
docker-machine ls  
  
# Vérifier quelle est la machine Docker active dans le shell courant
```

```
docker-machine active
```

```
## Supprimer un ou plusieurs machine(s) Docker
```

```
docker-machine rm <MACHINE NAME>
```

```
-f ou --force : forcer la suppression
```

```
## Se connecter en ssh sur une machine Docker
```

```
docker-machine ssh <MACHINE NAME>
```

```
## Stopper une machine Docker
```

```
docker-machine stop <MACHINE NAME>
```

```
## Démarrer une machine Docker
```

```
docker-machine start <MACHINE NAME>
```

```
## Redémarrer une machine Docker
```

```
docker-machine restart <MACHINE NAME>
```

```
## Récolter des informations sur une machine Docker
```

```
docker-machine inspect <MACHINE NAME>
```

```
## Récupérer les variables d'environnements d'une machine Docker

docker-machine env <MACHINE NAME>


## Mettre à niveau une machine Docker vers la dernière version de
Docker

docker-machine upgrade <MACHINE NAME>
```

Comprendre, Gérer et Manipuler un cluster Docker Swarm

Cet article vous explique les différentes notions de Docker Swarm, et vous décrit ensuite comment gérer et manipuler votre cluster Docker Swarm.

Dictionnaire Docker Swarm

Aujourd'hui nous allons nous intéresser à ma partie préférée dans Docker à savoir Docker Swarm.

C'est quoi un Docker Swarm ? Manager Swarm ? Les nœuds ? Les workers ?

Un Swarm est un groupe de machines exécutant le moteur Docker et faisant partie du même cluster. Docker swarm vous permet de lancer des commandes Docker auxquelles vous êtes habitué sur un cluster depuis une machine maître nommée **manager/leader Swarm**. Quand des machines rejoignent un Swarm, elles sont appelés **nœuds**.

Les managers Swarm sont les seules machines du Swarm qui peuvent exécuter des commandes Docker ou autoriser d'autres machines à se joindre au Swarm en tant

que **workers**. Les workers ne sont là que pour **fournir de la capacité** et n'ont pas le pouvoir d'ordonner à une autre machine ce qu'elle peut ou ne peut pas faire.

Jusqu'à présent, vous utilisiez Docker en mode hôte unique sur votre ordinateur local. Mais Docker peut également être basculé en mode swarm permettant ainsi l'utilisation des commandes liées au Swarm. L'activation du mode Swarm sur hôte Docker fait instantanément de la machine actuelle un manager Swarm. À partir de ce moment, Docker exécute les commandes que vous exécutez sur le Swarm que vous gérez, plutôt que sur la seule machine en cours.

C'est quoi un service ? une task (tâche) ?

Dans le vocabulaire Swarm nous ne parlons plus vraiment de conteneurs mais plutôt de **services**.

Un service n'est rien d'autre qu'une description de l'état souhaité pour vos conteneurs. Une fois le service lancé, une **tâche** est alors attribuée à chaque nœud afin d'effectuer le travail demandé par le service.

Nous verrons plus loin les détails de ces notions, mais histoire d'avoir une idée sur **la différence entre une tâche et un service**, nous allons alors imaginer l'exemple suivant :

Une entreprise vous demande de déployer des conteneurs d'applications web. Avant toute chose, vous allez commencer par définir les caractéristiques et les états de votre conteneur, comme par exemple :

- Trois conteneurs minimums par application afin de supporter des grandes charges de travail
- Une utilisation maximale de 100 Mo de mémoire pour chaque conteneur
- les conteneurs se baseront sur l'image httpd
- Le port 8080 sera mappé sur le port 80
- Redémarrer automatiquement les conteneurs s'ils se ferment suite à une erreur

Pour le moment vous avez défini l'état et les comportements de vos conteneurs dans votre service, par la suite quand vous exécuterez votre service, chaque nœud se verra attribuer alors une ou plusieurs tâches jusqu'à satisfaire les besoins définis par votre service.

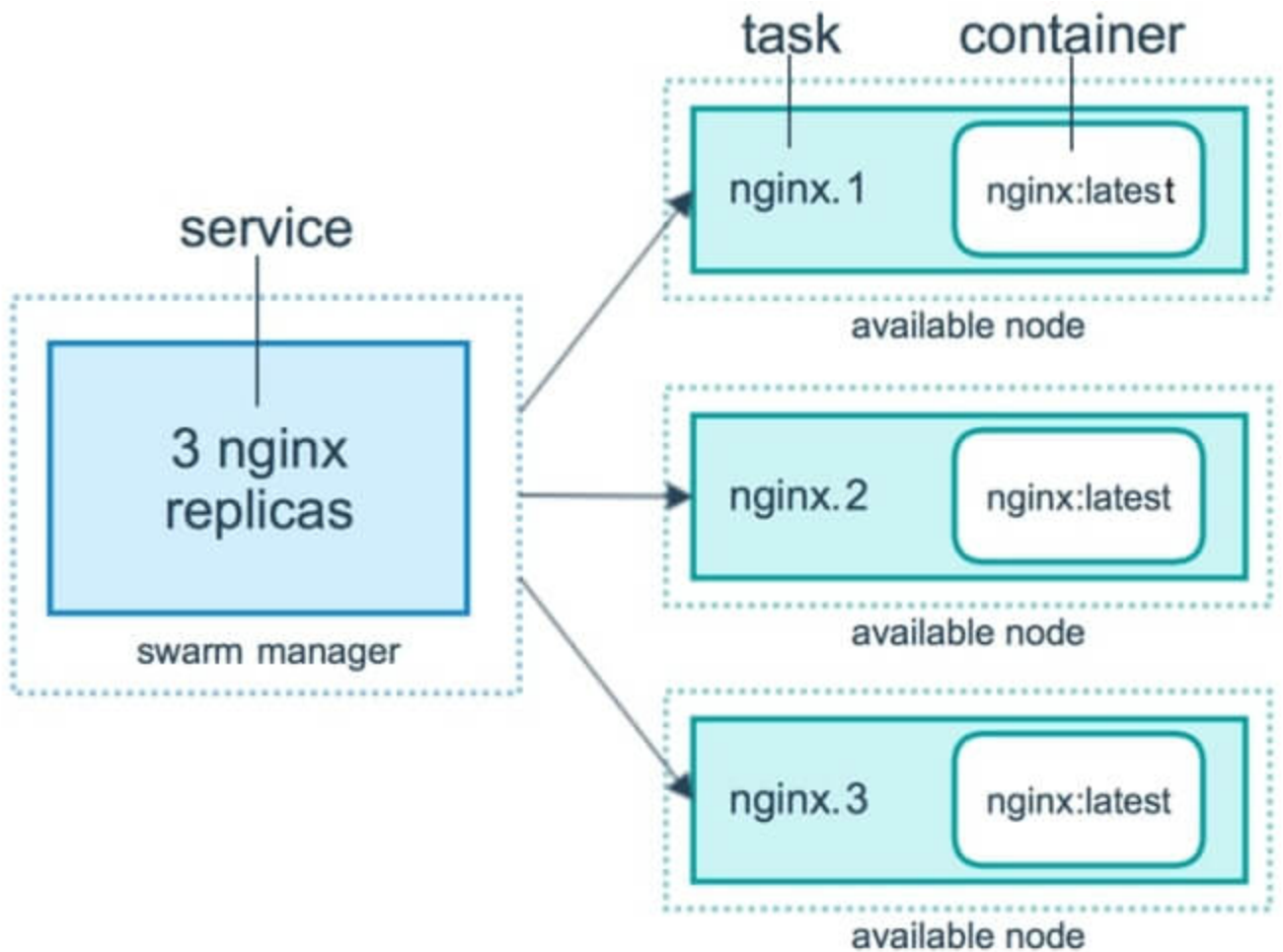
Résumons les différents concepts de Docker Swarm

Il est important au préalable de bien comprendre la définition de chaque concept afin d'assimiler facilement le fonctionnement global de Docker Swarm. Je vais donc vous résumer toutes ces notions à travers un seul exemple :

Imaginez que vous êtes embauché en tant qu'ingénieur système dans une start-up. Les développeurs de l'entreprise décident un jour de vous fournir les sources de leur application web tournant sous le serveur web nginx. Votre chef de projet vous demande alors de lui expliquer le process de déploiement de l'application web dans un Cluster Swarm. Comme vous êtes un très bon ingénieur ☐, vous lui expliquez les étapes suivantes :

- Un service sera créé dans lequel nous spécifierons qu'elle sera image utilisée et nous estimerons quelles seront les charges de travail suffisantes pour les conteneurs qui seront en cours d'exécution.
- La demande sera ensuite envoyée au manager Swarm (leader) qui planifie l'exécution du service sur des nœuds particuliers.
- Chaque nœud se voit assigné une ou plusieurs tâche(s) en fonction de l'échelle qui a été définie dans le service.
- Chaque tâche aura un cycle de vie, avec des états comme **NEW** , **PENDING** et **COMPLETE**.
- Un équilibreur de charge sera mis en place automatiquement par votre manager Swarm afin de supporter les grandes charges de travaux.

Voici un schéma qui reprend les étapes citées précédemment :



Créer d'un cluster Swarm

Création des nœuds

Dans le [chapitre précédent](#), nous avons vu comment déployer des machines Docker avec l'outil Docker Machine. Nous allons réutiliser cet outil de façon à créer deux VMs avec le driver virtualbox.

```
docker-machine create --driver virtualbox mymanagerCopier  
docker-machine create --driver virtualbox workerCopier
```

Vous avez maintenant créé deux machines virtuelles, nommées [mymanager](#) et [worker](#) qu'on utilisera plus tard en tant que nœuds dans notre Swarm. Nous allons exploiter la première

machine virtuelle **mymanager** en tant que manager Swarm et l'autre machine sera un nœud de travail.

Nous aurons besoin maintenant de récupérer les IPs de ces nouvelles machines à l'aide de la commande suivante :

```
docker-machine ls
```

Résultat :

NAME	ACTIVE	DRIVER	STATE	URL
SWARM	DOCKER	ERRORS		
mymanager	-	virtualbox	Running	
tcp://192.168.99.103:2376				v18.09.8
worker	-	virtualbox	Running	
tcp://192.168.99.104:2376				v18.09.8

Nous avons donc :

- **mymanager** avec l'ip **192.168.99.103**.
- **worker** avec l'ip **192.168.99.104**.

Activer le mode swarm

L'avantage de Docker Swarm est qu'il est déjà intégré dans Docker par défaut, donc pas besoin d'installation supplémentaire.

Dans cette partie, nous allons activer le mode Swarm sur la machine Docker **mymanager** de la même manière elle deviendra la leader de notre Swarm. Pour ce faire, nous utiliserons la commande `docker-machine ssh`.

Voici la **commande qui permet d'activer le mode Swarm** sur une machine d'un Swarm :

```
docker-machine ssh mymanager "docker swarm init --advertise-addr 192.168.99.103"
```

Résultat :

```
Swarm initialized: current node (seym19rjc3bo5eozlijjx7jgr) is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-1368nlbw9syzrliv44956cvp9b1g5ivmr2rmu5g238g7q1bro7-1rutjgzngabbv2jx7dlrjnqlr 192.168.99.103:2377
```

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

Votre cluster Swarm est prêt maintenant à accueillir de nouvelles machines. De plus, le résultat nous indique clairement comment rajouter une machine à notre Swarm, et c'est exactement ce que nous allons faire dans la prochaine étape.

Joindre une machine au cluster Swarm

Pour **ajouter un travailleur à un swarm**, exécutez la commande suivante :

```
docker-machine ssh worker "docker swarm join \
--token SWMTKN-1-1368nlbw9syzrliv44956cvp9b1g5ivmr2rmu5g238g7q1bro7-1rutjgzngabbv2jx7dlrjnqlr 192.168.99.103:2377" Copier
```

Résultat :

```
This node joined a swarm as a worker.
```

Voici la commande qui permet d'**afficher les différents nœuds de votre Swarm** :

```
docker-machine ssh mymanager "docker node ls" Copier
```

Résultat :

ID	HOSTNAME	STATUS
AVAILABILITY	MANAGER STATUS	ENGINE VERSION

seym19rjc3bo5eozlijjx7jgr *	mymanager	Ready
Active	Leader	18.09.8
i7f61zqyqfadzm01oj9w675ew	worker	Ready
Active		18.09.8

D'ailleurs le résultat nous indique formellement dans la colonne **MANAGER STATUS** que c'est le nœud **mymanager** le Big Boss ☐.

Rappel

Rappelez-vous que seuls les managers du Swarm comme **mymanager** exécutent les commandes Docker, les workers ne sont là juste pour fournir de la capacité.

Félicitations, vous venez de créer votre premier cluster Swarm !



Configurer le shell de notre manager Swarm

Jusqu'à présent, nous utilisons des commandes Docker en ssh afin de les exécuter sur le nœud manager. Ce n'est pas vraiment pratique comme solution, en vue de nous faciliter la vie, désormais nous chargerons les variables d'environnements de notre leader Swarm sur notre shell courant, dès lors toutes les prochaines commandes docker lancées depuis le shell courant s'exécuteront directement sur la machine **mymanager**.

Commençons par récupérer les variables d'environnements de notre nœud **mymanager** :

```
docker-machine env mymanagerCopier
```

Résultat :

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
```

```
export
DOCKER_CERT_PATH="/home/hatim/.docker/machine/machines/mymanager"

export DOCKER_MACHINE_NAME="mymanager"

# Run this command to configure your shell:

# eval $(docker-machine env mymanager)
```

La commande suivante, permet de **configurer votre shell** pour lui permettre de communiquer avec le nœud **mymanager** :

```
eval $(docker-machine env mymanager)Copier
```

Vérifions maintenant que c'est bien le nœud **mymanager** qui est actif :

```
docker-machine activeCopier
```

Résultat :

```
mymanager
```

Dorénavant, plus besoin de lancer vos commandes en ssh depuis votre shell courant pour communiquer avec le nœud **mymanager**.

Déployer une application à service individuel dans un Swarm avec Docker Service

Créer un service

La commande `docker service` est utilisée lors de la **gestion d'un service individuel dans un cluster Swarm**.

Voici une idée de ce qu'un service peut définir comme comportement et état d'un conteneur :

- Le nom de l'image et le tag que les conteneurs du nœud doivent exécuter.
- Combien de conteneurs participent au service.

- Les ports à exposer à l'extérieur du cluster Swarm.
- Comment doit agir le conteneur suite à une erreur.
- Les caractéristiques des nœuds sur lesquels le service peut s'exécuter (telles que des contraintes de ressources et ou de préférence de placement sur tel ou tel nœud).
- etc ...

C'est déjà pas mal non ? Pour l'exemple ci-dessous, on souhaite spécifier les comportements suivants :

- Un conteneur qui se base sur ma propre image **flask**.
- Trois conteneurs doivent être exécutés au minimum.
- Redémarrez le conteneur s'il se ferme suite à une erreur.
- Limiter l'utilisation de la mémoire à 100 Mo
- Mapper le porte 5001 sur le port 5000

Ci-dessous la **commande qui crée un service** avec ses options respectant les caractéristiques définis plus haut :

```
docker service create --name flaskc \
--replicas 3 \
--publish published=5001,target=5000 \
--restart-condition=on-failure \
--limit-memory      100M \
hajdaini/flask:firstCopier
```

Résultat :

```
4xp37ly8dedy0fyqn3jvocpyv

overall progress: 3 out of 3 tasks

1/3: running   [=====>]
2/3: running   [=====>]
3/3: running   [=====>]
```

Vous pouvez **connaître l'état d'avancement du service dans votre Swarm** en lançant la commande suivante :

```
docker service lsCopier
```

Résultat :

ID	NAME	MODE	REPLICAS
IMAGE	PORTS		
4xp37ly8dedy hajdaini/flask:first	flaskc *:5001->5000/tcp	replicated	3/3

Vous pouvez également **lister les différentes tâches de votre service** afin de vérifier par exemple sur quel nœud s'est exécutée votre tâche.

```
docker service ps flaskcCopier
```

Résultat :

ID	NAME	IMAGE	NODE
DESIRED STATE	CURRENT STATE	ERROR	
PORTS			
5prkwtd89c7i mymanager	flaskc.1 Running	hajdaini/flask:first Running about a minute ago	
kmksj0834v81 mymanager	flaskc.2 Running	hajdaini/flask:first Running about a minute ago	
fee27ez96y8d worker	flaskc.3 Running	hajdaini/flask:first Running about a minute ago	

Selon le résultat nous avons deux répliques dans le nœud **mymanager** et une seule réplique dans le nœud **worker**

Accéder à votre cluster

Actuellement, nous avons trois de conteneurs tournant dans notre Swarm, tous cyclant de manière aléatoire, grâce à un **équilibreur de charge créée automatiquement par Swarm**. Oui oui, vous n'avez pas besoin de configurer un Load Balancer, Swarm s'occupe de tous.

Vous ne me croyez pas ? J'ai faits exprès sur mon image flask d'afficher le hostname du conteneur, par défaut le hostname du conteneur équivaut à son ID. Nous allons récupérer les IDs de tous nos conteneurs grâce aux commandes suivantes :

Je commence par afficher l'ID des conteneurs de mon nœud **mymanager** :

```
docker ps
```

Résultat :

CONTAINER ID CREATED	IMAGE STATUS	COMMAND PORTS	NAMES
3d04a75d3e9f minutes ago flaskc.2.kmksj0834v81edg1l8wf7ujr6	hajdaini/flask:first Up 26 minutes	"python app.py" 5000/tcp	26
f5b9a2c43de8 minutes ago flaskc.1.5prkwtd89c7ij8wegh41d17b5	hajdaini/flask:first Up 26 minutes	"python app.py" 5000/tcp	26

Je récupère ensuite l'ID des conteneurs de mon nœud **worker** :

```
docker-machine ssh worker "docker ps"
```

Résultat :

CONTAINER ID CREATED	IMAGE STATUS	COMMAND PORTS	NAMES
0f6709fbef7f minutes ago flaskc.3.fee27ez96y8dm7vei5e5hnnd7	hajdaini/flask:first Up 29 minutes	"python app.py" 5000/tcp	29

Pour résumer nous avons :

- **3d04a75d3e9f** : ID du premier conteneur du nœud **mymanager**.
- **f5b9a2c43de8** : ID du deuxième conteneur du nœud **mymanager**.
- **0f6709fbef7f** : ID du conteneur du nœud **worker**.

Nous allons accéder à notre application à partir de l'adresse IP du nœud **mymanager** (vous pouvez utiliser également l'IP du nœud **worker**) afin de vérifier le hostname récupéré par la page, voici mes résultats sous forme de Screenshots :



On peut clairement constater que tous les conteneurs du cluster sont appelés de manière aléatoire grâce à l'équilibreur de charge mis en place par Swarm.

Mise à l'échelle de votre service

Votre application revendique encore plus de puissance ? Pas de soucis, voici la commande qui permet de **scaler automatiquement vos conteneurs sans les redémarrer** :

```
docker service scale flaskc=5Copier
```

Résultat :

```
flaskc scaled to 5

overall progress: 5 out of 5 tasks

1/5: running   [=====>]
2/5: running   [=====>]
3/5: running   [=====>]
4/5: running   [=====>]
5/5: running   [=====>]

verify: Service converged
```

Vérifions maintenant le nombre de tâches de nos nœuds afin de s'assurer du nombre de répliques

```
docker service ps flaskcCopier
```

Nous avons bien cinq répliques et non plus trois :

ID	NAME	IMAGE	NODE
DESIRED STATE	CURRENT STATE	ERROR	
PORTS			
5prkwtd89c7i mymanager	flaskc.1 Running	hajdaini/flask:first Running about an hour ago	
kmksj0834v81 mymanager	flaskc.2 Running	hajdaini/flask:first Running about an hour ago	
fee27ez96y8d worker	flaskc.3 Running	hajdaini/flask:first Running about an hour ago	
zx2f23b0pfbs worker	flaskc.4 Running	hajdaini/flask:first Running 36 seconds ago	
0rk6dsp4bnqf worker	flaskc.5 Running	hajdaini/flask:first Running 36 seconds ag	

Mettre à jour votre service

Il est possible de **mettre à jour votre application depuis votre nouvelle image sans interruption de service**.

Dans cet exemple, je vais simuler une nouvelle version de notre application grâce à ma nouvelle image que j'ai poussée dans le [Docker Hub](#) :

```
docker service update --image hajdaini/flask:second flaskcCopier
flaskc

overall progress: 5 out of 5 tasks

1/5: running   [=====>]
2/5: running   [=====>]
3/5: running   [=====>]
4/5: running   [=====>]
```

```
5/5: running [=====>]
```

```
verify: Service converged
```



Version 2

- CONTAINER IP : [redacted]26
- CONTAINER NAME : 16c5440fc6f3

Sans aucune interruption de service, nous avons pu mettre à jour 5 conteneurs déployés sur différents nœuds et ceci depuis une seule commande ! Je pense que vous commencez maintenant à comprendre la puissance de Docker Swarm ☐.

Supprimer votre service

Vous n'avez plus besoin de votre service ? Aucun problème, vous pouvez utiliser la commande suivante pour **supprimer votre service** :

```
docker service rm flaskcCopier
```

Déployer une application multi-services dans un Swarm avec Docker stack

Docker Stack peut être utilisée pour **gérer une application multi-services dans votre cluster Swarm**. Pour faire simple, vous pouvez considérer que la commande `docker service` est identique à la commande `docker run` et que la commande `docker stack` est comparable à la commande `docker-compose`.

Nous allons utiliser les super pouvoirs de notre nœud **mymanager** pour déployer nos applications Docker multi-services dans notre cluster Swarm.

Comme pour la création de notre service précédemment, nous allons définir les différentes caractéristiques de nos conteneurs :

- Deux API sous forme de deux services différents.
- Trois conteneurs par service doivent être exécutés au minimum.
- Redémarrez les services s'il se ferme suite à une erreur.
- Limiter l'utilisation de la mémoire à 50 MO.
- La première API écoute sur le port 5000 mais la deuxième API utilise le port 5001 comme port cible et le port 5000 comme port source.

Pas besoin d'apprendre de nouvelles choses pour créer nos services, car nous allons réutiliser les connaissances vues dans les chapitres précédents puisqu'il suffit juste de créer un fichier `docker-compose.yml` :

```
version: "3"

services:

  api1:

    image: hajdaini/flask:api1

    deploy:

      replicas: 3

      resources:

        limits:

          memory: 50M

      restart_policy:

        condition: on-failure

    ports:

      - "5000:5000"

  api2:
```

```
image: hajdaini/flask:api2

deploy:

  replicas: 3

  resources:

    limits:

      memory: 50M

  restart_policy:

    condition: on-failure

ports:

  - "5001:5000"Copier
```

La commande suivante permet de **déployer votre application multi-services dans votre Swarm** :

```
docker stack deploy -c docker-compose.yml api-appCopier
```

Résultat :

```
Creating network api-app_default
Creating service api-app_api1
Creating service api-app_api2
```

Les commandes de Docker Stack restent très identiques aux commandes `docker services` , par exemple pour lister les différentes tâches dans votre Swarm, vous utiliserez la commande suivante :

```
docker stack ps api-appCopier
```

Résultat :

ID	NAME	IMAGE	NODE
DESIRED STATE	CURRENT STATE	ERROR	
PORTS			
t6gxznwnyfh mymanager	api-app_api2.1 Running	hajdaini/flask:api2 Running 43 seconds ago	
dbfee0mr8zgx Running	api-app_api1.1 Running 47 seconds ago	hajdaini/flask:api1	worker
qbof56ucped8 Running	api-app_api2.2 Running 43 seconds ago	hajdaini/flask:api2	worker
keggmag2rfvb Running	api-app_api1.2 Running 47 seconds ago	hajdaini/flask:api1	worker
ok1prslxmq60 mymanager	api-app_api2.3 Running	hajdaini/flask:api2 Running 43 seconds ago	
w342dvn3q934 mymanager	api-app_api1.3 Running	hajdaini/flask:api1 Running 47 seconds ago	

D'après le résultat, on peut s'apercevoir que nous avons bien deux services distincts avec trois répliques pour chaque service.

Même chose, pour accéder à nos services on peut utiliser soit l'adresse IP du nœud **mymanager** ou l'IP du nœud **worker** :

```
curl http://192.168.99.103:5000/api1Copier
```

Résultat :

```
Bonjour, je suis l'api 1
```

```
curl http://192.168.99.103:5001/api2Copier
```

Résultat :

```
Bonjour, je suis l'api 2
```

À partir de là, vous pouvez faire tout ce que vous avez appris dans les anciens chapitres et mettre à l'échelle votre application en modifiant simplement le fichier `docker-compose.yml` et d'exécuter simplement à nouveau la commande `docker stack deploy` pour prendre en compte vos nouvelles modifications.

Autre chose, si jamais vous rajoutez une nouvelle machine dans votre Swarm, il suffit juste de relancer la commande `docker stack deploy` pour que votre application tire parti des nouvelles ressources de votre nouveau nœud.

Si vous n'avez plus besoin de votre pile, vous pouvez alors la supprimer grâce à la commande suivante :

```
docker stack rm api-appCopier
```

Conclusion

Je pense que comme moi vous êtes vraiment impressionné par la simplicité d'utilisation du Docker Swarm. L'un des plus gros avantages de Docker Swarm, c'est qu'il vous permet de vous concentrer sur le développement votre application et de ne pas vous soucier de l'endroit où cette dernière va s'exécuter. De plus les services sont constamment monitorés par Swarm. En plus du monitoring, Docker Swarm se charge de la **réparation automatique**, vous aidant ainsi à garder vos services de votre cluster en bon état de fonctionnement en comparant en permanence l'état désiré avec l'état actuel.

Docker Swarm offre d'autres fonctionnalités et je vous suggère de creuser davantage dans la documentation officielle. Pour information ce n'est pas le seul orchestrateur de conteneurs qui existe dans le marché, vous pouvez par exemple aussi vous intéresser à [Kubernetes](#), je compte d'ailleurs faire un cours dessus prochainement.

Comme à mon habitude, voici un **récapitulatif des commandes Docker Swarm** vu ensemble :

```
# Gestion du cluster Swarm

docker swarm
```

```
# Gestion des conteneurs uni-service
```

```
docker service
```

```
# Gestion des conteneurs multi-services
```

```
docker stack
```

```
# Gestion des nœuds
```

```
docker node
```

```
# Activer le mode Swarm
```

```
docker swarm init
```

```
# Joindre une machine au cluster Swarm
```

```
docker swarm join --token <token> <myvm ip>:<port>
```

```
# -----
```

```
# Lister les différents nœuds de votre Swarm
```

```
docker node ls
```

Inspecter un nœud

`docker node inspect <NODE NAME>`

`--pretty` : meilleur effet visuel

Retirer un nœud de votre Swarm (ne supprime pas la VM)

`docker node rm <NODE NAME>`

`-f` ou `--force` : forcer la suppression

Créer un service

`docker service create <IMAGE NAME>`

`--name` : nom du service

`--replicas <number>` : nombre de tâches

`--publish published=<cible>,target=<source>` : mapper un port

`--restart-condition=<condition>` : condition de redémarrage en cas d'erreur

`--limit-memory <number>` : limiter l'utilisation de la mémoire

`--limit-cpu <number>` : limiter l'utilisation du CPU

```
# Visualiser l'état d'avancement de vos services Swarm
```

```
docker service ls
```

```
# lister les différentes tâches de votre service
```

```
docker service ps <SERVICE NAME>
```

```
# Mise en échelle des répliques de votre service
```

```
docker service scale <SERVICE NAME>=<NUMBER>
```

```
# Mise à jour de des conteneurs de votre service
```

```
docker service update --image <IMAGE NAME>:<TAG> <SERVICE NAME>
```

```
# Supprimer un service
```

```
docker service rm flaskc
```

```
# -----
```

```
# Déployer une nouvelle pile ou met à jour une pile existante
```

```
docker stack deploy -c <Docker Compose File> <STACK NAME>
```

```
# Lister tous les services de votre pile

docker stack services <STACK NAME>


# Répertorier les tâches de la pile

docker stack ps <STACK NAME>


# Supprimer tous les services de votre pile

docker stack rm <STACK NAME>


# Lister le nombre de services de votre pile

docker stack ls
```

Conclusion du cours complet sur la technologie Docker

Clap de fin ! Vous connaissez dès à présent tous les concepts dans Docker. Je publie cet article pour vous présenter mon message de conclusion, mes futurs projets/articles sur Docker et son écosystème.

Conclusion

Vous voici arrivé(e) au terme de ce cours et je tiens vraiment à vous **remercier** d'avoir pris le temps de suivre ce cours et à vous **féliciter** pour avoir tenu jusqu'au bout ☐.

À travers ce cours, nous avons vu **tous les aspects majeurs de Docker**, rendant ainsi son utilisation très simple, intéressante et très puissante.

Par rapport à tous ce que nous avons étudié, si je devais **décrire les différents avantages de Docker** à une entreprise, je dirai que :

Docker apporte une valeur immédiate dans les entreprises, **augmentant ainsi rapidement sa productivité**. Il vous permet de **diffuser vos applications en production plus rapidement** tout en **réduisant les coûts d'infrastructure et de maintenance**, accélérant ainsi la mise sur le marché de **nouvelles solutions**, de ce fait il fournit de **nouvelles expériences client** allant des **applications monolithiques** traditionnelles aux **applications cloud natives**.

J'espère néanmoins, que la lecture de ce cours vous aura été utile et agréable et que ça vous aura permis d'y voir un peu plus clair et que les différentes notions de Docker vues, vous auront je l'espère, donné envie d'aller encore plus loin. Car comme le dirait un certain Buzz l'Éclair "Docker t'amènera vers l'infini et au-delà" (bon ok ok, j'ai modifié un peu la phrase ☐).

Si vous souhaitez déployer vos applications en production, n'hésitez pas alors à faire un tour dans la section "Run your app in production" dans la [documentation](#). Cependant, avec la plupart des **concepts de Docker** vues dans ce cours, vous êtes d'ores et déjà capable de déployer vos premières applications dans un environnement de production.

La suite ?

Autant vous dire tout de suite, qu'on n'en aura pas encore fini avec Docker, puisque **la plateforme repose sur des technologies open source** standard, notamment le fameux [Kubernetes](#). On le retrouve vraiment partout, pas étonnant d'ailleurs qu'on aperçoit de nos jours sur la plupart des pages d'installation des projets opensource, une partie d'installation Linux, Windows et Docker.



Pour information Docker est **utilisé** par **des millions de professionnels de l'informatique** dans le monde entier, et comprend la plus grande bibliothèque de contenu de conteneurs et de son écosystème, avec plus de 100 000 images de conteneurs provenant de grands fournisseurs de logiciels, de projets open source et de la communauté.

Je continuerai à faire d'autres tutoriels concernant Docker. Il peut s'agir d'un article où je partage avec vous des images réutilisables. Mais je compte aussi prochainement prévoir des cours sur [Kubernetes](#) qui est un orchestrateur de conteneurs, qui gère par défaut sans surprise les conteneurs Docker.

Sur ce, je vous souhaite une très bonne continuation !



« Good Luck from Docker and Golang mascots »