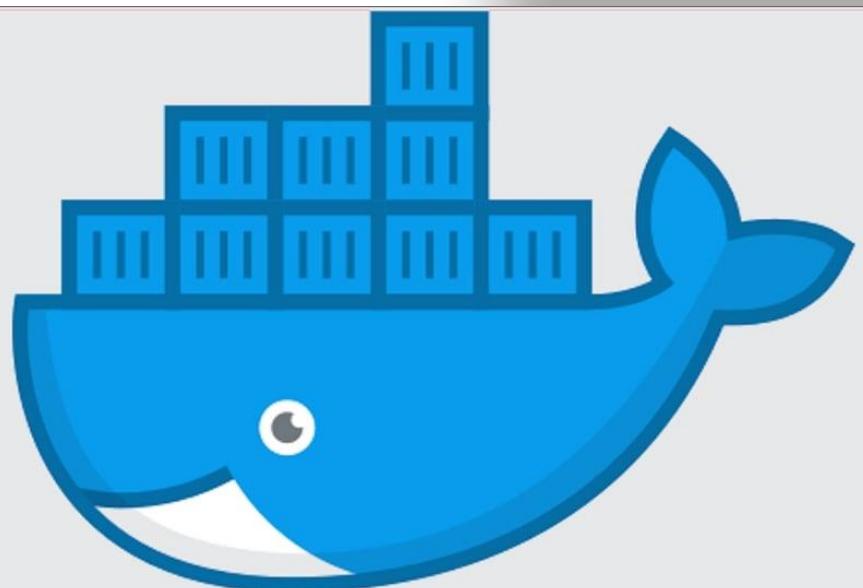


Docker - Exercices

Docker



docker

Version :	Etat :	Date Création :
1.0	production	11/04/18

AUTEURS

Contributeur	Courriel
LECOEUVRE Aymeric	Lecoeuvrepro@gmail.com
RICOU Ewen	Ewen.ricou2015@gmail.com

MISES A JOUR

Version	§	Commentaires
V1.0		Version initiale

DOCUMENTS ANNEXES

TERMINOLOGIE



Ce sigle signale une remarque importante dont il faut tenir compte.



Ce sigle signale une note, un exemple.



Ce sigle signale une remarque prioritaire.



Ce sigle signale une note technique

Les images utilisées sont libres de droit et issues des sites Microsoft Office® et OpenClipart.org.

TABLE DES MATIERES

1. PRESENTATION DOCKER	5
2. EXERCICE 1	6
2.1. PREMIER CONTENEUR	6
2.2. CONTENEUR LINUX ALPINE	8
2.2.1. Docker Container Run	8
2.3. ISOLATION D'UN CONTENEUR	10
2.4. TERMINOLOGIE	12
3. EXERCICE 2	14
3.1. CREATION D'IMAGE A PARTIR D'UN CONTENEUR	14
3.2. CREATION D'IMAGE A L'AIDE D'UN FICHIER DOCKER	17
3.3. COUCHES D'IMAGE	19
3.4. INSPECTION D'IMAGE	21
3.5. TERMINOLOGIE	22
4. EXERCICE 3	23
4.1. INITIALISEZ VOTRE ESSAIM	23
4.2. AFFICHER LES MEMBRES DE L'ESSAIM	24
4.3. DEPLOYER UNE PILE	24
4.4. MISE A L'ECHELLE D'UNE APPLICATION	27
4.5. CONCLUSION	27
5. EXERCICE 4	28
5.1. INTRODUCTION AUX PERMISSIONS	28
5.2. COMMENCEMENT	28
5.3. TEST DES PERMISSIONS DE DOCKER	29
5.4. CONCLUSION	30
6. EXERCICE 5	31

1. PRESENTATION DOCKER

Docker est un logiciel libre qui automatise le déploiement d'applications dans des conteneurs logiciels, Docker est un outil qui peut empaqueter une application et ses dépendances dans un conteneur isolé, qui pourra être exécuté sur n'importe quel serveur.

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

Docker	
	
<u>Développeur</u>	Docker, Inc. (en) 
<u>Première version</u>	13 mars 2013
<u>Dernière version</u>	18.04.0 (10 avril 2018) 
<u>Version avancée</u>	17.04.0 (6 avril 2017) 
<u>Etat du projet</u>	En développement actif
<u>Écrit en</u>	Go 
<u>Environnement</u>	Multi-plateforme
<u>Type</u>	Gestionnaire de conteneurs
<u>Politique de distribution</u>	Gratuit
<u>Licence</u>	Apache-2.0 et licence propriétaire 
<u>Site web</u>	www.docker.com [archive]

Sources : Wikipédia

2. EXERCICE 1

2.1. PREMIER CONTENEUR

Dans cet exercice on va apprendre à exécuter un conteneur, gratuit et léger, et explorer les bases du fonctionnement des conteneurs, on va voir aussi comment Docker exécute et isole les conteneurs les uns des autres.

Concepts de cet exercice :

- Moteur Docker
- Conteneurs et images
- Registres d'images et Docker Store
- Isolement du conteneur

Notre premier conteneur va être le conteneur Hello World. Pour l'exécuter il faut faire :

```
docker container run hello-world
```

Ce conteneur va nous dire ce qu'il s'est passé quand on l'exécute :

```
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://cloud.docker.com/

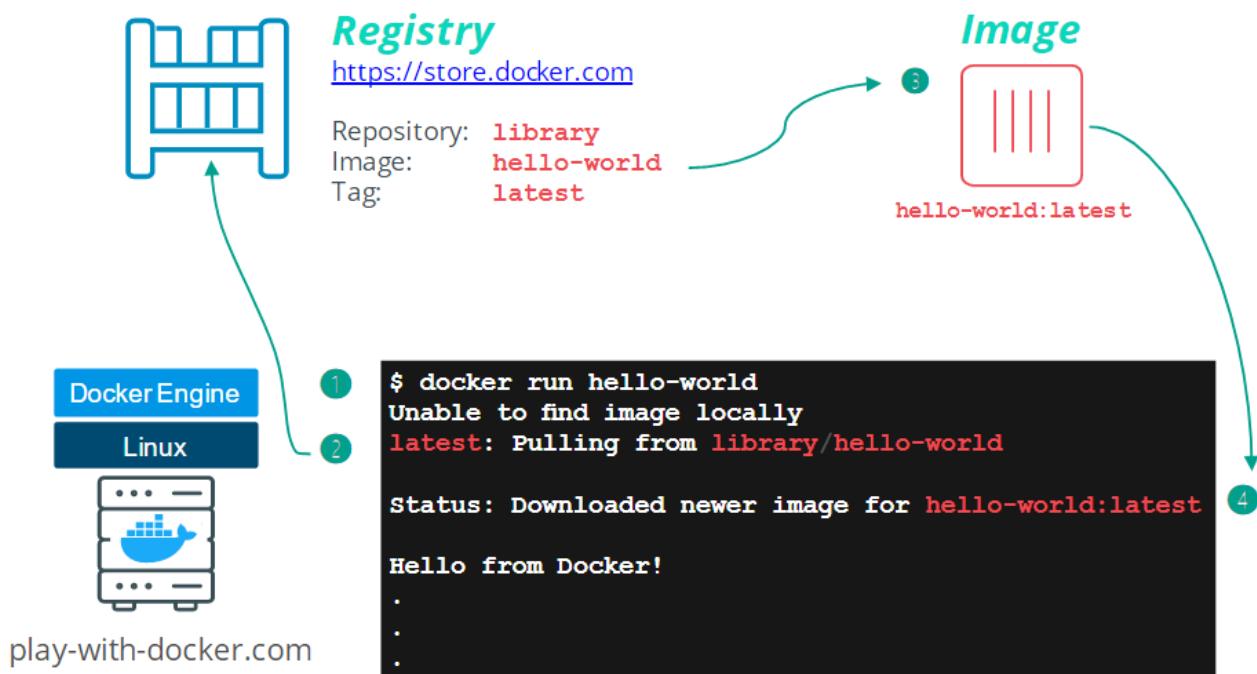
For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
```

En premier Docker contacte le programme docker exécuté en arrière-plan, ensuite le programme docker a été chercher l'image intitulé hello-world sur le hub docker, puis il a créé un conteneur depuis cette image et nous a envoyé ce message. Avant ce message docker n'a pas pu trouver l'image localement du coup il a demandé au programme docker :

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:97ce6fa4b6cdc0790cda65fe7290b74cfecbd9fa0c9b8c38e979330d547d22ce1
Status: Downloaded newer image for hello-world:latest
```

Voici un schéma d'explication de ce qu'il s'est passé :

Hello World: What Happened?



En premier il ne trouve pas l'image localement, ensuite il demande au hub docker s'il le connaît, ensuite docker crée le conteneur associé à l'image trouvé sur le hub docker.

2.2. CONTENEUR LINUX ALPINE

Pour le reste de l'exercice on va pouvoir exécuter un conteneur linux Alpine est une distribution Linux légère, elle est donc rapide à « pulled » (récupérer).

Pour commencer exéutez cette commande :

```
docker image pull alpine
```

La commande pull récupère l'image alpine sur le registre Docker et l'enregistre dans notre système. Dans le cas présent le registre docker est le HUB docker.

On peut utiliser la commande docker image pour voir la liste des images sur notre système

```
docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
alpine	latest	c51f86c28340	4 weeks ago	1.109 MB
hello-world	latest	690ed74de00f	5 months ago	960 B

2.2.1. Docker Container Run

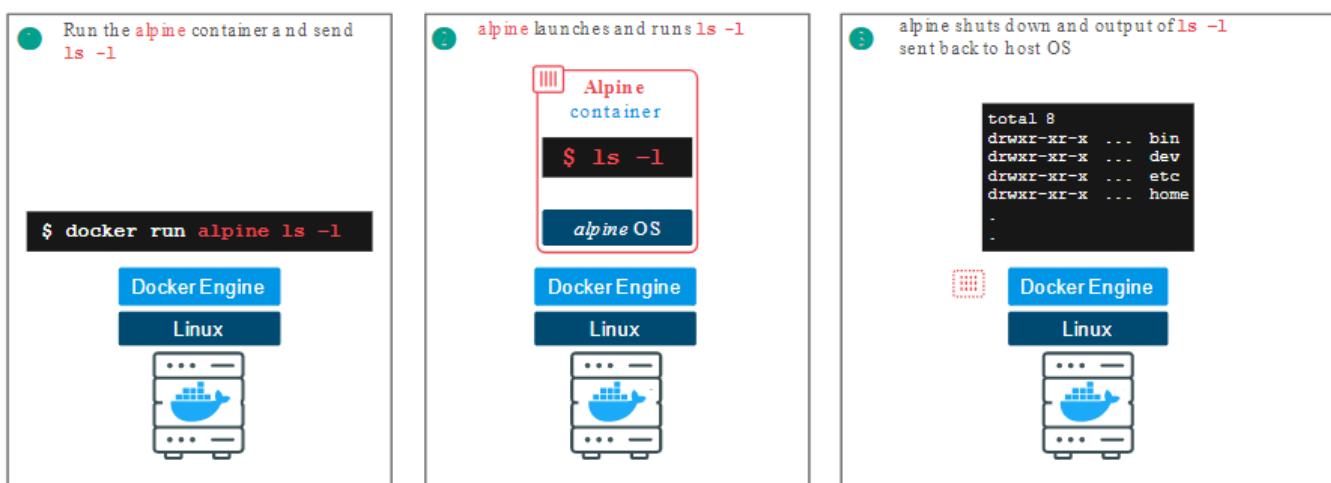
Nous pouvons dès à présent lancer un conteneur docker basé sur cette image. Pour ce faire, il faut utiliser la commande :

```
Docker container run alpine ls -l
```

```
total 48
drwxr-xr-x  2 root  root  4096 Mar  2 16:20 bin
drwxr-xr-x  5 root  root  360  Mar 18 09:47 dev
drwxr-xr-x 13 root  root  4096 Mar 18 09:47 etc
drwxr-xr-x  2 root  root  4096 Mar  2 16:20 home
drwxr-xr-x  5 root  root  4096 Mar  2 16:20 lib
.....
.....
```

Lorsqu'on appelle run, le client Docker trouve l'image, crée le conteneur, puis exécute une commande dans ce conteneur. Lorsqu'on exécute la commande docker container run alpine ls -l. Docker exécute la commande ls -l dans le système alpine. Une fois la commande ls faite, le conteneur s'arrête.

docker run Details



Comme je l'ai expliqué on exécute le conteneur alpine est un envoi la commande ls -l, puis le conteneur s'éteint et nous revois la réponse de la commande.

Essayez de rentrer la commande qui suit :

```
docker container run alpine echo "hello from alpine"
```

On obtient alors ceci :

```
hello from alpine
```

Dans ce cas, le client Docker a consciencieusement exécuté la commande echo dans notre conteneur alpine, et l'a arrêté. Tout cela se fais très rapidement, c'est la grande différence avec une machine virtuelle, si on devait faire ça avec une machine virtuelle il faudrait démarrer l'OS, puis lancer l'application, et entrer la commande, il faudrait en gros 1 à 2 minutes pour avoir la réponse du echo. Les conteneurs docker fonctionnent au niveau de la couche application, de sorte qu'ils ignorent la plupart des étapes requises par les machines virtuelles et n'exécutent que ce qui est requis.

Essayez une autre commande :

```
Docker container run alpine /bin/sh
```

Après cette commande il n'a dit rien se passé à première vue, en vérité c'est comme si vous avez démarré une 3^{ème} instance du conteneur alpine et exécuté la commande /bin/sh et qu'il s'est fermé. Il a donc simplement lancé le shell, quitté le shell et arrêté le conteneur. Docker a une fonctionnalité pour pouvoir rester dans le conteneur sans le fermer ce « flag » c'est : -it.

Par exemple tapez ce qui suit :

```
Docker container run -it alpine /bin/sh
```

Normalement vous êtes à l'intérieur du conteneur exécutant un shell linux, c'est comme si vous avez démarrée une machine virtuelle mais avec juste le shell. Pour quitter le shell faites : EXIT

Pour voir les conteneurs en cours d'exécution on peut faire la commande :

Docker container ls

Comme aucun conteneur n'est en cours d'exécution, la réponse est une ligne vide. Essayons maintenant une variante plus utile :

Docker container ls -a

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
36171a5da744 2 minutes ago	alpine	"/bin/sh" fervent_newton	5 minutes ago	Exited (0)
a6a9d46d0b2f 6 minutes ago	alpine	"echo 'hello from alp" lonely_kilby	6 minutes ago	Exited (0)
ff0a5c3750b9 8 minutes ago	alpine	"ls -l" elated_ramanujan	8 minutes ago	Exited (0)
c317d0a9e3d2 12 minutes ago	hello-world	"/hello" stupefied_mcclintock	34 seconds ago	Exited (0)

Ce que vous voyez maintenant c'est une liste de tous les conteneurs que vous avez exécutés.

Il est logique de passer du temps à se familiariser avec les commandes docker run. Pour en savoir plus sur run utilisez **docker container run –help**

2.3. ISOLATION D'UN CONTENEUR

Dans les étapes ci-dessus, nous avons exécuté plusieurs commandes via des instances de conteneur avec l'aide de **docker container run**. La commande **docker container ls -a** nous a montré qu'il y avait plusieurs conteneurs listés. Pourquoi y a-t-il autant de conteneurs s'ils proviennent tous de l'image *alpine* ?

C'est un concept de sécurité essentiel dans le monde des conteneurs Docker ! Même si chaque commande **docker container run** utilisait la même *image* *alpine*, chaque exécution est un *conteneur* séparé et isolé. Chaque conteneur possède un système de fichiers distinct et s'exécute dans un espace de noms différent ; Par défaut, un conteneur n'a aucun moyen d'interagir avec d'autres conteneurs, même ceux provenant de la même image. Essayons un autre exercice pour en savoir plus sur l'isolement.

docker container run -it alpine /bin/ash

le */bin/ash* est un autre type de shell disponible dans l'image *alpine*. Une fois que le conteneur est lancé et que vous êtes à l'intérieur de l'invite de commande du conteneur, tapez les commandes suivantes :

```
echo "hello world" > hello.txt  
ls
```

La commande echo crée un fichier appelé « hello.txt » avec les mots « hello world » à l'intérieur. La deuxième vous donne une liste des fichiers situé dans le répertoire et devrait afficher le fichier que vous avez créé. Sortez ensuite du conteneur avec la commande **exit**.

Pour montrer comment fonctionne l'isolation, exécutez la commande suivante :

```
docker container run alpine ls
```

Cette commande envoie au conteneur alpine la commande ls, mais cette fois, le fichier que vous avez créé juste avant n'est plus là, c'est l'isolement. Votre commande est exécutée dans une nouvelle instance distincte, même si elle est basée sur la même image. La 2^{ème} instance n'a aucun moyen d'interagir avec la 1^{ère} car le moteur docker les maintient séparés et nous n'avons pas configuré de paramètres supplémentaires qui permettraient à ces deux instances d'interagir.

L'isolation permet aux utilisateurs de créer rapidement des copies de test isolées et séparées d'une application ou d'un service et de les faire fonctionner côté à côté sans interférer les uns avec les autres.

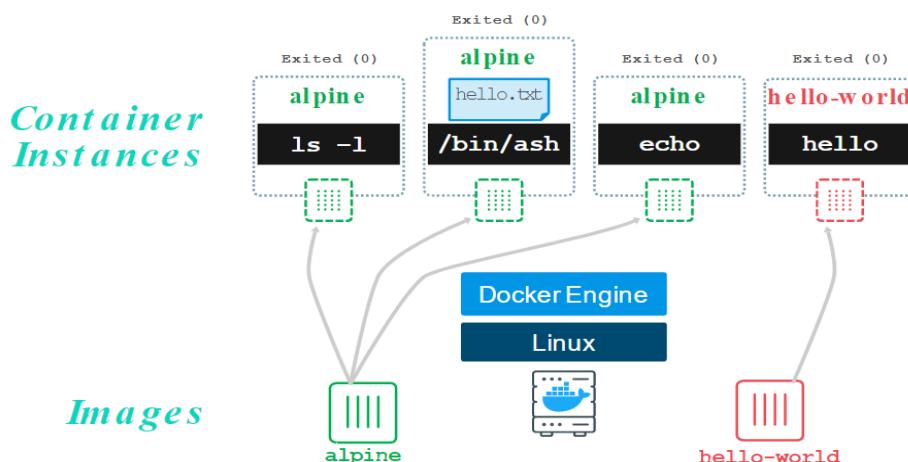
Pour pouvoir revenir sur le conteneur qui contient le fichier « hello.txt » il faut récupérer le numéro associé à l'instance du conteneur pour cela :

```
docker container ls -a
```

la commande devrait sortir ça :

CONTAINER ID	IMAGE NAMES	COMMAND	CREATED	STATUS
PORTS				
36171a5da744 2 minutes ago	alpine	"ls" distracted_bhaskara	2 minutes ago	Exited (0)
3030c9c91e12 2 minutes ago	alpine	"/bin/ash" fervent_newton	5 minutes ago	Exited (0)
a6a9d46d0b2f 6 minutes ago	alpine	"echo 'hello from alp'" lonely_kilby	6 minutes ago	Exited (0)
ff0a5c3750b9 8 minutes ago	alpine	"ls -l" elated_ramanujan	8 minutes ago	Exited (0)
c317d0a9e3d2 12 minutes ago	hello-world	"/hello" stupefied_mcclintock	34 seconds ago	Exited (0)

Docker Container Isolation



Le conteneur dans lequel nous avons créé le fichier "hello.txt" est le même que celui dans lequel nous avons utilisé le shell **/bin/ash**, que nous pouvons voir dans la colonne 'COMMAND'. Nous pouvons aussi voir un numéro d'identifiant du conteneur c'est avec ça qu'on va pouvoir retourner sur l'instance souhaité. Pour cela essayez de taper :

```
docker container start <container ID>
```

Il est possible d'utiliser qu'un petit bout de l'ID pour faire fonctionner la commande par exemple les 4 premiers caractères.

2.4. TERMINOLOGIE

Dans la dernière section, vous avez vu beaucoup de jargon spécifique à Docker qui pourrait être déroutant pour certains. Alors avant d'aller plus loin, clarifions une terminologie fréquemment utilisée dans l'écosystème Docker.

- *Images* - Le système de fichiers et la configuration de notre application qui sont utilisés pour créer des conteneurs. Pour en savoir plus sur une image Docker, exécutez **docker image inspect alpine**. Dans la démo ci-dessus, vous avez utilisé la commande **docker image pull** pour le télécharger l'image **alpine**. Lorsque vous avez

exécuté la commande `docker container run hello-world`, elle a également fait un tour `docker image pull` en coulisses pour télécharger l'image `hello-world`.

- *Conteneurs* - Exécution d'instances d'images Docker - les conteneurs exécutent les applications réelles. Un conteneur inclut une application et toutes ses dépendances. Il partage le noyau avec d'autres conteneurs et s'exécute comme un processus isolé dans l'espace utilisateur du système d'exploitation hôte. Vous avez créé un conteneur en `docker run` utilisant l'image alpine que vous avez téléchargée. Une liste des conteneurs en cours d'exécution peut être consultée à l'aide de la commande `docker container ls`.
- *Docker daemon* - Service d'arrière-plan s'exécutant sur l'hôte qui gère la construction, l'exécution et la distribution des conteneurs Docker.
- *Client Docker* - L'outil de ligne de commande qui permet à l'utilisateur d'interagir avec le démon Docker.
- *Docker Store* - est, entre autres, un [registre](#) d'images Docker. Vous pouvez considérer le registre comme un répertoire de toutes les images Docker disponibles. Vous l'utiliserez plus tard dans ce tutoriel.

3. EXERCICE 2

Dans cet exercice on va apprendre à créer une image à partir d'un conteneur, à créer une image à l'aide d'un fichier docker, on va voir ce qu'est la couche d'images ainsi que les inspections d'images.

Nous allons commencer par la forme la plus simple de création d'image, dans laquelle nous avons simplement une de nos instances en tant qu'image.

Nous verrons ensuite comment obtenir les détails d'une image à travers l'inspection et explorer le système de fichiers pour avoir une meilleure compréhension de ce qui se passe derrière

3.1. CREATION D'IMAGE A PARTIR D'UN CONTENEUR

Commençons alors par ouvrir un shell à partir d'un conteneur ubuntu :

```
Docker container run -ti ubuntu bash
```

Comme vous le savez, nous avons avec cette commande été chercher l'image appelé ubuntu depuis le docker store et on a ouvert un shell dans ce conteneur.

Pour customiser un petit peu les choses on va installer un paquet appelé figlet dans ce conteneur. Pour cela tapez les commandes suivantes :

```
Apt-get update
```

```
Apt-get install -y figlet
```

```
Figlet "Hello Docker"
```

Vous devriez voir les mots « hello docker » imprimés en gros caractères sur l'écran.

Maintenant sortez de ce conteneur :

```
exit
```

Supposons maintenant que cette nouvelle application est très utile et que vous voulez la partager avec le reste de votre équipe. Vous pourriez leur dire de faire exactement ce que vous avez fait ci-dessus et installer figlet dans leur propre conteneur, mais s'il s'agissait d'une application du monde réel où vous venez d'installer plusieurs paquets et de passer par plein d'étapes de configuration, le processus peut devenir long et sujet à de nombreuses erreurs. Au lieu de cela, il serait plus simple de créer une image que vous pouvez partager.

Pour commencer, nous devons obtenir l'ID de ce conteneur en utilisant la commande ls (ne pas oublier l'option -a, sinon il va juste vous afficher les conteneurs en route, pas ceux qui sont finis).

```
Docker container ls -a
```

Avant de créer notre propre image, nous pourrions vouloir inspecter tous les changements que nous avons faits. Essayez de taper la commande **docker container diff <container ID>** pour le conteneur que vous venez de créer. Vous devriez voir une liste de tous les fichiers qui ont été ajoutés ou modifiés dans le conteneur lorsque vous avez installé

figlet. Docker garde une trace de toutes ces informations pour nous. Cela fait partie du concept de couche qu'on verra plus tard.

Maintenant, pour créer une image, nous devons « valider » ce conteneur. Pour cela il faut utiliser l'option commit qui va créer une image localement sur le système exécutant le moteur docker. Exécutez la commande suivante, en utilisant l'ID du conteneur ubuntu, afin de valider le conteneur et créer une image à partir de celui-ci.

```
Docker container commit <container id>
```

C'est tout, vous avez créé votre première image ! Une fois qu'elle a été créée, nous pouvons voir l'image dans la liste des images disponibles

```
Docker image ls
```

Vous devriez voir quelque chose comme ceci :

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	a104f9ae9c37	46 seconds ago	160MB
ubuntu	latest	14f60031763d	4 days ago	120MB

Notez que l'image que nous avons pris du store docker dans la première étape (ubuntu) est listée ici avec notre propre image personnalisée. Sauf que notre image personnalisée n'a aucune information dans les colonnes REPOSITORY ou TAG, ce qui rendrait difficile d'identifier exactement ce qui se trouvait dans ce conteneur si nous voulions le partager

L'ajout de cette information à une image est connu comme le marquage d'une image. A partir de la commande précédente, récupérez l'ID de l'image nouvellement créée et marquez-la pour qu'elle s'appelle **ourfiglet** :

```
Docker image tag <image_id> ourfiglet
```

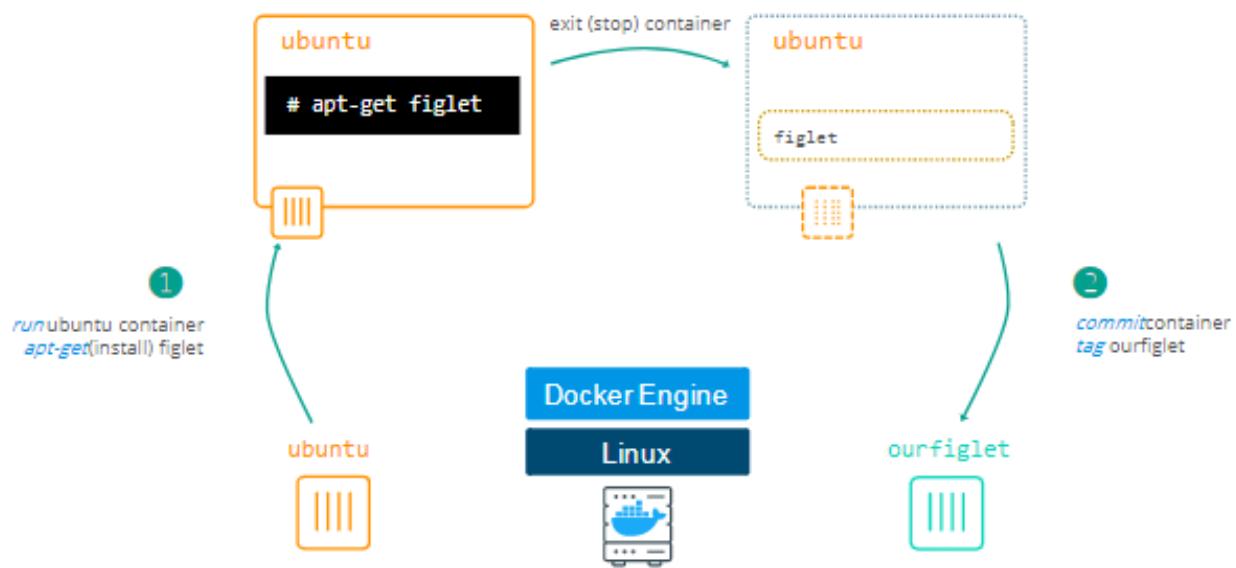
```
Docker image ls
```

Maintenant nous avons le nom plus convivial « ourfiglet » que nous pouvons utiliser pour identifier notre image.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ourfiglet	latest	a104f9ae9c37	5 minutes ago	160MB
ubuntu	latest	14f60031763d	4 days ago	120MB

Voici une vue graphique de ce qu'on a fait :

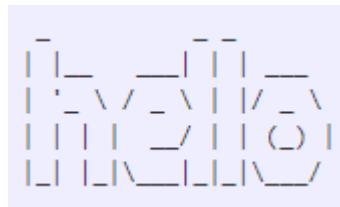
Image Creation: Instance Promotion



Maintenant, nous allons exécuter un conteneur basé sur l'image *ourfiglet* nouvellement créée :

```
Docker container run ourfiglet figlet hello
```

Comme le paquet *figlet* est présent dans notre image *ourfiglet*, la commande renvoie la sortie suivante :



Cet exemple montre que nous pouvons créer un conteneur, y ajouter toutes les bibliothèques et les applications qu'on souhaite, et il suffit de le valider pour créer une image. Nous pouvons ensuite utiliser cette image comme nous le ferions pour les images tirées du Docker Store. Nous avons encore un léger problème parce que notre image est stockée uniquement localement. Pour partager l'image, nous voulons *pousser* l'image vers un registre quelque part.

Comme mentionné ci-dessus, cette approche consistant à installer manuellement un logiciel dans un conteneur puis à l'appliquer à une image personnalisée n'est qu'une façon de créer une image. Cela fonctionne bien et est assez commun. Cependant, il existe un moyen plus puissant de créer des images. Dans le chapitre suivant, nous verrons comment les images sont créées en utilisant un *Dockerfile*, qui est un fichier texte contenant toutes les instructions pour construire une image.

3.2. CREATION D'IMAGE A L'AIDE D'UN FICHIER DOCKER

Au lieu de créer une image binaire statique, nous pouvons utiliser un fichier appelé *Dockerfile* pour créer une image. Le résultat final est essentiellement le même, mais avec un Dockerfile nous fournissons les instructions pour construire l'image, plutôt que seulement les fichiers binaires bruts. C'est utile car il devient beaucoup plus facile de gérer les changements, d'autant plus que vos images deviennent plus grandes et plus complexes.

Par exemple, si une nouvelle version de figlet est publiée, nous devrons soit recréer notre image à partir de rien, soit lancer notre image et mettre à jour la version installée de figlet. En revanche, un Dockerfile inclurait les commandes **apt-get** que nous avons utilisées pour installer figlet afin que nous - ou quiconque utilisant le Dockerfile - puissions simplement recomposer l'image en utilisant ces instructions.

En pratique, les Dockerfiles peuvent être gérés de la même manière que vous pouvez gérer un code source : ce sont simplement des fichiers texte, donc presque n'importe quel système de contrôle de version peut être utilisé pour gérer Dockerfiles au fil du temps.

Nous allons utiliser un exemple simple dans cette section et construire une application "hello world" dans Node.js. (Pas besoin d'être familier avec Node.js pour comprendre comment ça fonctionne.)

Nous commencerons par créer un fichier dans lequel nous récupérerons le nom d'hôte et l'afficherons.

Tapez le contenu suivant dans un fichier nommé index.js. Vous pouvez utiliser vi, vim ou plusieurs autres éditeurs linux dans cet exercice.

```
var os = require("os");
var hostname = os.hostname();
console.log("hello from " + hostname);
```

Le fichier que nous venons de créer est le code javascript pour notre serveur. Comme vous pouvez probablement le deviner, Node.js affichera simplement un message « hello ». Nous allons utiliser alpine comme image de base du système d'exploitation, ajouter un runtime node.js, puis copier notre code source dans le conteneur.

Nous spécifierons également la commande par défaut à exécuter lors de la création du conteneur.

Créez un fichier nommé Dockerfile et copiez le contenu suivant dans celui-ci.

```
FROM alpine
RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app
CMD ["node", "index.js"]
```

Construisons notre première image avec ce Dockerfile et nommez-le hello: v0.1 :

```
docker image build -t hello:v0.1 .
```

Voilà une image qui récapitule ce que vous avez fait :

Dockerfiles

Dockerfile:

```
FROM alpine
RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app
CMD ["node", "index.js"]
```

FROM alpine



RUN apk

```
apk update
apk add nodejs
```

COPY . /app

```
host/index.js -->
container/app
```

WORKDIR /app

CMD "node" "index.js"

```
$ cd /app
$ node index.js
```

hello:v0.1

=



Ensuite démarrez un conteneur pour vérifier que nos applications s'exécutent correctement :

```
Docker container run hello :v0.1
```

Vous devriez alors avoir une sortie similaire à la suivante (l'id sera différent pour vous):

```
hello from 92d79b6de29f
```

Qu'est-ce qui vient de se passer ? Nous avons créé deux fichiers : notre code d'application (index.js) est un simple code javascript qui imprime un message. Et le Dockerfile est les instructions pour le moteur Docker pour créer notre conteneur personnalisé. Ce Dockerfile fait ce qui suit :

1. Spécifie une image de base à tirer depuis l'image *alpine* que nous avons utilisée dans les labos précédents.
2. Ensuite, il exécute deux commandes (*apk update* et *apk add*) à l'intérieur de ce conteneur qui installe le serveur Node.js.

3. Ensuite, nous lui avons dit de copier les fichiers de notre répertoire de travail dans le conteneur. Le seul fichier que nous avons en ce moment est notre *index.js*.
4. Ensuite, nous spécifions WORKDIR - le répertoire que le conteneur doit utiliser lorsqu'il démarre
5. Enfin, nous avons donné à notre conteneur une commande (CMD) à exécuter lorsque le conteneur démarre.

3.3. COUCHES D'IMAGE

Il y a autre chose d'intéressant dans les images que nous construisons avec Docker. En cours d'exécution, ils semblent être un système d'exploitation unique et une application. Mais les images elles-mêmes sont en réalité construites en **couches**. Si vous revenez en arrière et regardez la sortie de votre commande **docker image build**, vous remarquerez qu'il y avait 5 étapes et chaque étape avait plusieurs tâches.

Les couches sont un concept important. Pour explorer cela, nous allons passer par un autre ensemble d'exercices.

Tout d'abord, vérifiez l'image que vous avez créée précédemment en utilisant la commande **history** :

```
Docker image history <image id>
```

Ce que vous voyez est la liste des images de conteneur intermédiaires qui ont été construites en même temps que la création de votre image finale de l'application Node.js. Certaines de ces images intermédiaires deviendront des **couches** dans votre image finale du conteneur.

Dans la sortie de la commande **history**, les calques Alpine d'origine sont en bas de la liste, puis chaque personnalisation que nous avons ajoutée dans notre **Dockerfile** est son propre pas dans la sortie. C'est un concept puissant car cela signifie que si nous devons apporter une modification à notre application, cela ne peut affecter qu'une seule couche ! Pour voir cela, nous allons modifier un peu notre application et créer une nouvelle image.

Tapez le texte suivant dans votre fenêtre de console :

```
echo "console.log(\"this is v0.2\");" >> index.js
```

Cela ajoutera une nouvelle ligne au bas de votre fichier **index.js**. Maintenant, nous allons construire une nouvelle image en utilisant notre code mis à jour. Nous allons également marquer notre nouvelle image pour la marquer comme une nouvelle version afin que toute personne consommant nos images plus tard peut identifier la bonne version à utiliser :

```
docker image build -t hello:v0.2
```

Vous devriez voir une sortie similaire à ceci :

```
Sending build context to Docker daemon 86.15MB

Step 1/5 : FROM alpine
---> 7328f6f8b418

Step 2/5 : RUN apk update && apk add nodejs
---> Using cache
---> 2707762fca63

Step 3/5 : COPY . /app
---> 07b2e2127db4

Removing intermediate container 84eb9c31320d

Step 4/5 : WORKDIR /app
---> 6630eb76312c

Removing intermediate container ee6c9e7a5337

Step 5/5 : CMD node index.js
---> Running in e079fb6000a3
---> e536b9dadd2f

Removing intermediate container e079fb6000a3

Successfully built e536b9dadd2f

Successfully tagged hello:v0.2
```

3.4. INSPECTION D'IMAGE

Maintenant, nous allons voir l'inspection d'image, l'inspection d'image peut être pratique dans des cas où on souhaite savoir le contenu du conteneur ainsi que ces détails, les commandes qu'il peut exécuter, le système d'exploitation et plus encore.

L'image alpine doit déjà être présente localement sinon, exécutez la commande suivante :

```
docker image pull alpine
```

Une fois que nous sommes sûrs qu'il est là, inspectons-le

```
Docker image inspect alpine
```

Il y a beaucoup d'informations là-dedans :

- Les couches de l'image est composée de
- Le pilote utilisé pour stocker les calques
- L'architecture / le système d'exploitation pour lequel il a été créé
- Métadonnées de l'image
- ...

Nous n'entrerons pas dans tous les détails ici mais nous pouvons utiliser des filtres pour inspecter des détails particuliers sur l'image. Vous avez peut-être remarqué que les informations sur l'image sont au format JSON. Nous pouvons en profiter pour utiliser la commande inspect avec quelques informations de filtrage pour obtenir des données spécifiques de l'image.

Obtenons la liste des calques :

```
docker image inspect --format "{{ json .RootFS.Layers }}" alpine
```

Alpine est juste une petite image de base de l'OS donc il n'y a qu'une seule couche :

```
["sha256:60ab55d3379d47c1ba6b6225d59d10e1f52096ee9d5c816e42c635ccc57a5a2b"]
```

Nouveau regardons notre image Hello personnalisée. Vous aurez besoin de l'identifiant de l'image :

```
docker image inspect --format "{{ json .RootFS.Layers }}" <image ID>
```

Notre image est un peu plus intéressante :

```
["sha256:5bef08742407efd622d243692b79ba0055383bbce12900324f75e56f589aedb0", "sha256:5ac283aaea742f843c869d28bbeaf5000c08685b5f7ba01431094a207b8a1df9", "sha256:2ecb254be0603a2c76880be45a5c2b028f6208714aec770d49c9eff4cbc3cf25"]
```

Nous avons trois couches dans notre application.

3.5. TERMINOLOGIE

- *Calques* - Une image Docker est construite à partir d'une série de calques. Chaque couche représente une instruction dans le fichier Docker de l'image. Chaque couche sauf la dernière est en lecture seule.
- *Dockerfile* - Un fichier texte qui contient toutes les commandes, dans l'ordre, nécessaires pour construire une image donnée. La page de [référence Dockerfile](#) répertorie les différentes commandes et les détails de format pour Dockerfiles.
- *Volumes* - Une couche de conteneur Docker spéciale qui permet aux données de persister et d'être partagées séparément du conteneur lui-même. Considérez les volumes comme un moyen d'abstraire et de gérer vos données persistantes séparément de l'application elle-même.

4. EXERCICE 3

Jusqu'à présent, nous avons exploré l'utilisation d'instances uniques de conteneurs s'exécutant sur un seul hôte, un peu comme un développeur peut le faire en travaillant sur une seule application de service ou comme un administrateur informatique peut le faire sur un banc d'essai. Les applications de production sont généralement beaucoup plus complexes et ce modèle de serveur unique ne fonctionnera pas pour coordonner 10 ou 100 conteneurs et les connexions réseau entre eux, sans parler de la nécessité d'assurer la disponibilité et la capacité de mise à l'échelle.

Pour les applications réelles, les utilisateurs informatiques et les équipes d'applications ont besoin d'outils plus sophistiqués. Docker fournit deux de ces outils: **Docker Compose** et **Docker Swarm Mode**. Les deux outils ont quelques similitudes mais quelques différences importantes:

Compose est utilisé pour contrôler plusieurs conteneurs sur un seul système. Tout comme le Dockerfile que nous avons examiné pour construire une image, il y a un fichier texte qui décrit l'application : quelles images utiliser, combien d'instances, les connexions réseau, etc. Mais Compose ne fonctionne que sur un seul système.

Docker Swarm indique à Docker que vous exécuterez de nombreux moteurs Docker et que vous souhaitez coordonner les opérations entre eux. Le mode Swarm combine la possibilité de définir non seulement l'architecture de l'application, comme Composer, mais aussi de définir et de maintenir des niveaux de haute disponibilité, la mise à l'échelle, l'équilibrage de charge, etc. Avec toutes ces fonctionnalités, le mode Swarm est plus souvent utilisé dans les environnements de production que son cousin simpliste, Compose.

4.1. INITIALISEZ VOTRE ESSAIM

La première chose que nous devons faire est de dire à nos hôtes Docker que nous voulons utiliser le mode Docker Swarm. Les essaims peuvent être simplement un seul nœud, mais cela est inhabituel car vous ne disposez pas de capacités de haute disponibilité et vous limitez sévèrement votre évolutivité. La plupart des essaims de production ont au moins trois nœuds de gestion dans eux et de nombreux nœuds de travail. Trois gestionnaires est le minimum pour avoir un véritable cluster à haute disponibilité avec quorum. Notez que les nœuds de gestionnaire peuvent exécuter vos tâches de conteneur de la même manière qu'un nœud de travail, mais cette fonctionnalité peut également être séparée afin que les gestionnaires n'effectuent que les tâches de gestion.

L'initialisation du mode Docker Swarm est facile. Dans votre première fenêtre de terminal que vous utilisez entrez :

```
docker swarm init --advertise-addr $(hostname -i)
```

Vous devriez avoir ceci :

```
Swarm initialized: current node (tjocs7ul557phkmp6mkpjmu3f) is now a manager.
```

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

```
Docker swarm join --token SWMTKN-1-3b33jjwsqpkcy2c8og73aorjf2ao9sjm4crvbwg3xpd1ome459-ckfdcxqqahb9gy9s2t9n5mi78 10.0.25.3:2377
```

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

Dans la sortie de votre swarm init, on vous donne une commande au milieu qui ressemble à celle **docker swarm join -token SWMTKN-X-abcdef...** que vous utilisez pour joindre des nœuds de travail à l'essaim. Vous recevez également une deuxième commande **docker swarm join-token manager** pour l'ajout de gestionnaires supplémentaires.

Nous allons ajouter un travailleur. Copiez la commande "docker swarm join ..." de la sortie de votre manager et collez-la dans une 2ème fenêtre d'un terminal.

Votre docker swarm devrait fonctionner maintenant, vous aurez donc un gestionnaire et un travailleur. Comme indiqué ci-dessus, vous auriez presque toujours 3 nœuds de gestionnaire ou plus et plusieurs nœuds de travail afin de maintenir la haute disponibilité et l'évolutivité, mais un de chaque est suffisant pour commencer.

4.2. AFFICHER LES MEMBRES DE L'ESSAIM

À partir de la première fenêtre de terminal, vérifiez le nombre de nœuds dans l'essaim :

```
docker node ls
```

4.3. DEPLOYER UNE PILE

Une pile est un groupe de services déployés ensemble : plusieurs composants conteneurisés d'une application qui s'exécutent dans des instances distinctes. Chaque service individuel peut en fait être constitué d'un ou de plusieurs conteneurs, appelés tâches, puis toutes les tâches et tous les services forment ensemble une pile.

Comme avec Dockerfiles et les fichiers Compose, le fichier qui définit une pile est un fichier texte brut facile à éditer et à suivre. Dans notre exercice, il y a un fichier appelé **docker-stack.yml** dans le dossier en cours qui sera utilisé pour déployer l'application de vote en tant que pile.

Entrez les informations suivantes pour étudier le fichier **docker-stack.yml** :

```
Cat docker-stack.yml
```

Ce fichier YAML définit l'ensemble de notre pile : l'architecture des services, le nombre d'instances, la manière dont tout est câblé, comment gérer les mises à jour de chaque service. C'est le code source de notre conception d'application. Quelques éléments de note particulière :

- Près du haut du fichier, vous verrez la ligne "services :". Ce sont les composants d'application individuels. Dans l'application de vote, nous avons redis, db, vote, résultat, travailleur et visualiser comme nos services.
- Sous chaque service se trouvent des lignes qui spécifient comment ce service doit s'exécuter :
 - Notez l'image familière de terme des laboratoires plus tôt ? Même idée ici : c'est l'image du conteneur à utiliser pour un service particulier.

- *Les ports et les réseaux* s'expliquent d'eux-mêmes, bien qu'il soit utile de souligner que ces réseaux et ports peuvent être utilisés de manière privée dans la pile ou qu'ils peuvent permettre une communication externe vers et depuis une pile.²
- Notez que certains services ont une *réplique* étiquetée en ligne : cela indique le nombre d'instances, ou de tâches, de ce service que les gestionnaires Swarm doivent démarrer lorsque la pile est mise en place. Le moteur Docker est assez intelligent pour charger automatiquement l'équilibre entre plusieurs réplicas en utilisant des équilibrEURS de charge intégrés. (L'équilibrEUR de charge intégré peut, bien sûr, être remplacé par autre chose.)

Vérifiez que vous êtes dans le terminal du gestionnaire [node1] et procédez comme suit :

```
docker stack deploy --compose-file=docker-stack.yml voting_stack
```

Vous pouvez voir si la pile déployée depuis le terminal du gestionnaire [node1]

```
docker stack ls
```

La sortie devrait être la suivante. Il indique que les 6 services de la pile de l'application de vote (nommée voting_stack) ont été déployés.

NAME	SERVICES
voting_stack	6

Nous pouvons obtenir des détails sur chaque service dans la pile avec ce qui suit :

```
docker stack services voting_stack
```

La sortie devrait être similaire à la suivante, bien que naturellement vos ID seront uniques :

ID	NAME	MODE	REPLICAS	IMAGE
10rt1wczotze	voting_stack_visualizer	replicated	1/1	dockersample s/visualizer:stable
8lqj31k3q5ek	voting_stack_redis	replicated	1/1	redis:alpine
nhb4igkkyg4y	voting_stack_result	replicated	1/1	dockersample s/examplevotingapp_result:before
nv8d2z2qhlx4	voting_stack_db	replicated	1/1	postgres:9.4
ou47zdyf6cd0	voting_stack_vote	replicated	2/2	dockersample s/examplevotingapp_vote:before
rpxnwmoipagq	voting_stack_worker	replicated	1/1	dockersample s/examplevotingapp_worker:latest

Si vous voyez qu'il y a 0 répliques, attendez quelques secondes et entrez à nouveau la commande. L'Essaim finira par avoir toutes les répliques en cours d'exécution pour vous. Tout comme notre fichier **docker-stack** spécifié, il y a deux répliques du service *voting_stack_vote* et une de chacune des autres.

Lister les tâches du service de vote.

```
docker service ps voting_stack_vote
```

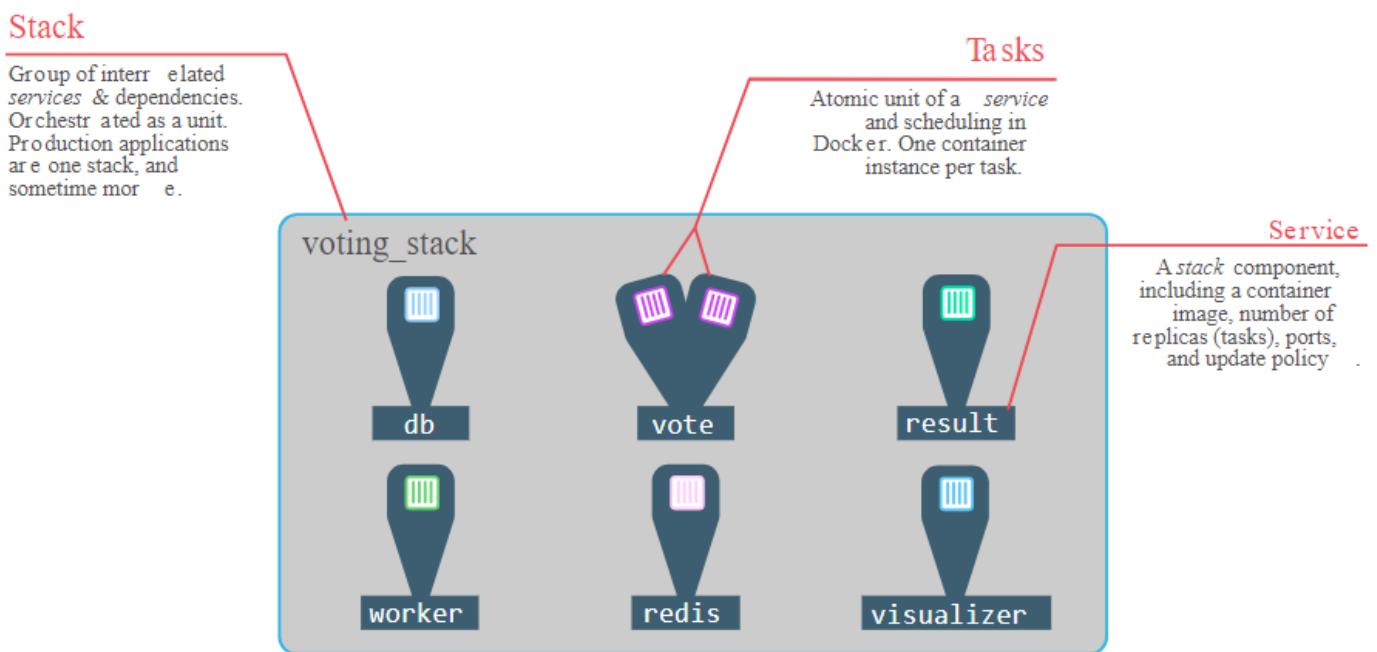
Vous devriez obtenir une sortie comme la suivante où les 2 tâches (répliques) du service sont listées.

ID	NAME	IMAGE		
NODE	DESIRED STATE	CURRENT STATE	ERROR	PORTS
my7jqgze7pgg	voting_stack_vote.1	dockersamples/examplevotingapp_vote:be fore node1 Running	Running	56 seconds ago
3jzgk39dyr6d	voting_stack_vote.2	dockersamples/examplevotingapp_vote:be fore node2 Running	Running	58 seconds ago

À partir de la colonne NODE, nous pouvons voir qu'une tâche est en cours d'exécution sur chaque nœud. Cette application est dotée d'un VISUALIZER SWARM intégré pour vous montrer comment l'application est configurée et en cours d'exécution. Vous pouvez également accéder à l' interface utilisateur Web frontale de l'application.

Le SWARM VISUALIZER vous donne la disposition physique de la pile, mais voici une interprétation logique de la façon dont les piles, les services et les tâches sont liés :

Swarm: Stacks, Services & Tasks



4.4. MISE A L'ECHELLE D'UNE APPLICATION

Comment pouvons-nous dire à notre application d'ajouter plus de répliques de notre service de *vote* ? En production, vous pouvez l'automatiser via les API de Docker mais pour l'instant nous le ferons manuellement. Vous pouvez également modifier le fichier `docker-stack.yml` et modifier les spécifications si vous souhaitez que la taille de l'échelle soit plus permanente. Tapez ce qui suit sur le terminal [node1] :

```
docker service scale voting_stack_vote=5
```

Vous devriez voir le nombre de répliques pour le service de vote augmenter à 5 et dans quelques secondes, Swarm les fera toutes fonctionner.

4.5. CONCLUSION

L'utilisation de seulement quelques commandes vous permet de déployer une pile de services en utilisant Docker Swarm Mode pour orchestrer la pile entière, toutes conservées dans le format de fichier Docker Compose simple et lisible par l'homme.

5. EXERCICE 4

Cet exercice est un exercice sécurité sur les permissions, dans cet exercice vous apprendrez les bases des fonctionnalités du noyau Linux. Vous apprendrez comment ils fonctionnent avec Docker, certaines commandes de base pour les afficher et les gérer, ainsi que comment ajouter et supprimer des fonctionnalités dans de nouveaux conteneurs.

5.1. INTRODUCTION AUX PERMISSIONS

Le noyau Linux est capable de décomposer les privilèges de l'utilisateur **root** en unités distinctes appelées **permissions**. Par exemple, la fonctionnalité **CAP_CHOWN** permet à l'utilisateur root d'apporter des modifications arbitraires aux UID et aux GID des fichiers. La fonction **CAP_DAC_OVERRIDE** permet à l'utilisateur root de contourner les contrôles d'autorisation du noyau sur les opérations de lecture, d'écriture et d'exécution de fichiers. Presque toutes les puissances spéciales associées à l'utilisateur root de Linux sont décomposées en permissions individuelles.

Cette répartition des privilèges root en petite fonctionnalités vous permet de :

- Supprimez les permissions individuelles du compte root d'utilisateur, ce qui le rend moins puissant/dangereux.
- Ajoutez des privilèges aux utilisateurs non root à un niveau très petit.

Les permissions s'appliquent aux fichiers et aux threads. Les capacités de fichiers permettent aux utilisateurs d'exécuter des programmes avec des privilèges plus élevés. Ceci est similaire à la façon dont le bit **setuid** fonctionne. Les fonctionnalités Thread permettent de suivre l'état actuel des capacités des programmes en cours d'exécution.

Docker impose certaines limitations qui rendent le travail avec des permissions beaucoup plus simple. Par exemple, les permissions de fichiers sont stockées dans les attributs étendus d'un fichier et les attributs étendus sont supprimés lorsque les images Docker sont créées. Cela signifie que vous n'aurez normalement pas trop à vous soucier des permissions de fichiers dans les conteneurs.

5.2. COMMENCEMENT

Dans cette étape, vous apprendrez l'approche de base pour gérer les permissions avec docker. Vous apprendrez également les commandes Docker utilisées pour gérer les fonctionnalités du compte racine d'un conteneur.

A partir de Docker vous avez 3 options de haut niveau pour utiliser les permissions :

- Exédez des conteneurs en tant que root avec un grand nombre de fonctionnalités et essayez de gérer manuellement les fonctionnalités de votre conteneur.
- Exédez des conteneurs en tant que root avec des permissions limitées et ne jamais les modifiez dans un conteneur
- Exédez des conteneurs en tant qu'utilisateur non privilégié sans fonctionnalités.

L'option 2 est la plus réaliste, l'option 3 serait idéale mais pas réaliste et l'option 1 devrait être évitée autant que possible.

Dans les commandes suivantes, \$cap sera utilisé pour indiquer une ou plusieurs permissions individuelles.

Pour supprimer des fonctionnalités du compte **root** d'un conteneur :

```
docker run --rm -it --cap-drop $CAP alpine sh
```

Pour ajouter des fonctionnalités au compte **root** d'un conteneur :

```
docker run --rm -it --cap-add $CAP alpine sh
```

Pour supprimer toutes les fonctionnalités, puis ajouter explicitement des fonctionnalités individuelles au compte **root** d'un conteneur :

```
docker run --rm -it --cap-drop ALL --cap-add $CAP alpine sh
```

Le noyau Linux préfixe toutes les constantes de permissions avec "CAP_". Par exemple, CAP_CHOWN, CAP_NET_ADMIN, CAP_SETUID, CAP_SYSADMIN etc. Les constantes de permissions Docker ne sont pas préfixées avec "CAP_" mais correspondent aux constantes du noyau.

5.3. TEST DES PERMISSIONS DE DOCKER

Dans cette étape, vous allez commencer divers nouveaux conteneurs. Chaque fois que vous utiliserez les commandes apprises à l'étape précédente pour modifier les fonctionnalités associées au compte utilisé pour exécuter le conteneur.

Démarrer un nouveau conteneur et prouver que le compte root du conteneur peut changer la propriété des fichiers :

```
docker run --rm -it alpine chown nobody /
```

Si la commande ne donne aucun code de retour c'est normal c'est que l'opération a réussi. La commande fonctionne par ce que le comportement par défaut est que les nouveaux conteneurs doivent être démarrés avec un utilisateur root. Cet utilisateur a la capacité CAP_CHOWN par défaut.

Démarrez un autre nouveau conteneur et supprimez toutes les fonctionnalités du compte racine des conteneurs autres que CAP_CHOWN. N'oubliez pas que Docker n'utilise pas le préfixe "CAP_" lors de l'adressage des constantes des permissions.

```
docker run --rm -it --cap-drop ALL --cap-add CHOWN alpine chown nobody /
```

Cette commande ne donne pas non plus de code retour, indiquant une exécution réussie. L'opération réussit car, bien que vous ayez supprimé toutes les fonctionnalités du compte **root** du conteneur, vous avez ajouté la fonctionnalité **chown**. La fonctionnalité **chown** est tout ce qui est nécessaire pour changer la propriété d'un fichier.

Démarrez un autre nouveau conteneur et supprimez uniquement la fonctionnalité **chown** de son compte racine.

```
docker run --rm -it --cap-drop CHOWN alpine chown nobody /
```

```
chown: /: Operation not permitted
```

Cette fois, la commande renvoie un code d'erreur indiquant qu'elle a échoué. Cela est dû au fait que le compte racine du conteneur n'a pas la capacité **CHOWN** et ne peut donc pas modifier la propriété d'un fichier ou d'un répertoire.

Créez un autre nouveau conteneur et essayez d'ajouter la fonctionnalité **CHOWN** à l'utilisateur non root appelé **nobody**. Dans le cadre de la même commande, essayez de modifier la propriété d'un fichier ou d'un dossier.

```
docker run --rm -it --cap-add chown -u nobody alpine chown nobody /
```

```
chown: /: Operation not permitted
```

La commande ci-dessus échoue car Docker ne prend pas encore en charge l'ajout de fonctionnalités aux utilisateurs non root.

5.4. CONCLUSION

Vous devriez maintenant savoir comment fonctionne les permissions avec docker, vous avez pu ajouter et supprimer des fonctionnalités à une série de nouveaux conteneurs, vous avez vu que les permissions peuvent être ajoutées et supprimées de l'utilisateur racine d'un conteneur à un niveau très petit. Vous avez également appris que Docker ne prend actuellement pas en charge l'ajout de fonctionnalités aux utilisateurs non root.

6. EXERCICE 5