Les images Docker

Nous allons commencer avec la forme la plus simple de création d'image, dans laquelle nous allons simplement faire un commit d'une de nos instances de conteneur en tant qu'image. Ensuite, nous explorerons une méthode beaucoup plus puissante et utile pour créer des images: le Dockerfile.

Création d'images à partir d'un conteneur

Commençons par exécuter un shell interactif dans un conteneur Alpine. Pour personnaliser un peu les choses, nous allons installer un paquet appelé *figlet* dans ce conteneur. Votre conteneur doit toujours être en cours d'exécution, alors tapez les commandes suivantes sur votre ligne de commande de conteneur Alpine:

```
[admuser@localhost ~]$ docker container run -it alpine ash
/ #
/ # apk add figlet
fetch https://dl-
cdn.alpinelinux.org/alpine/v3.19/main/x86_64/APKINDEX.tar.gz
fetch https://dl-
cdn.alpinelinux.org/alpine/v3.19/community/x86_64/APKINDEX.tar.gz
(1/1) Installing figlet (2.2.5-r3)
Executing busybox-1.36.1-r15.trigger
OK: 8 MiB in 16 packages
/ #
/ # figlet hello
 _          _ _
| |__   ___| | | ___
| '_ \ / _ \ | |/ _ \
| | | |  __/ | | (_) |
|_| |_|\___|_|_|\___/

/ # exit
```

Supposons maintenant que cette nouvelle application figlet soit très utile et que vous souhaitiez la partager avec le reste de votre équipe. Vous pouvez leur dire de faire exactement ce que vous avez fait ci-dessus et d'installer figlet dans leur propre conteneur, ce qui ne serait pas assez simple s'il s'agissait d'une application réelle où vous veniez d'installer plusieurs packages et d'exécuter un certain nombre d'étapes de configuration. Au lieu de cela, il serait plus facile de créer une image que vous pouvez partager avec votre équipe.

Avant de créer notre propre image, nous pourrions vouloir inspecter toutes les modifications que nous avons apportées.

```
[root@localhost ~]# docker container diff 944107f2ed58
C /var
C /var/cache
C /var/cache/apk
A /var/cache/apk/APKINDEX.6bd8ca47.tar.gz
A /var/cache/apk/APKINDEX.728a4eb9.tar.gz
C /etc
C /etc/apk
C /etc/apk/world
...
```

Vous devriez voir une liste de tous les fichiers qui ont été ajoutés ou modifiés dans le conteneur lorsque vous avez installé figlet. Docker conserve pour nous toutes ces informations. Cela fait partie du concept de couche.

A Un fichier ou un répertoire a été ajouté

D Un fichier ou un répertoire a été supprimé

C Un fichier ou un répertoire a été modifié

Maintenant, pour créer une image, nous devons «valider» ce conteneur. Commit crée une image localement sur le système exécutant le Docker engine. Exécutez la commande suivante, en utilisant l'ID de conteneur que vous avez récupéré, afin de valider le conteneur et de créer une image à partir de celui-ci.

```
[root@localhost ~]# docker container commit 944107f2ed58
sha256:a6f74cd34651708cd41ed970db46e90b4fc613a9e951fce0ebcb6e38fdc401ca
[root@localhost ~]#
[root@localhost ~]# docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
<none>                  <none>              a6f74cd34651        14 seconds ago      289 MB
docker.io/alpine        latest              e7d92cdc71fe        7 weeks ago         5.59 MB
docker.io/centos        latest              470671670cac        7 weeks ago         237 MB
docker.io/hello-world   latest              fce289e99eb9        14 months ago       1.84 kB
```

Notre image personnalisée ne contient aucune information dans les colonnes REPOSITORY ou TAG, ce qui rendrait difficile d'identifier exactement ce qu'il y avait dans ce conteneur si nous voulions partager entre plusieurs membres de l'équipe.

L'ajout de ces informations à une image est appelé marquage d'une image. À partir de la commande précédente, obtenez l'ID de la nouvelle image créée et étiquetez-la pour qu'elle s'appelle *ourfiglet* :

```
[root@localhost ~]# docker image tag a6f74cd34651 ourfiglet
[root@localhost ~]# docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
ourfiglet               latest              a6f74cd34651        5 minutes ago       289 MB
docker.io/alpine        latest              e7d92cdc71fe        7 weeks ago         5.59 MB
docker.io/centos        latest              470671670cac        7 weeks ago         237 MB
docker.io/hello-world   latest              fce289e99eb9        14 months ago       1.84 kB
```

Nous allons maintenant exécuter un conteneur basé sur l'image ourfiglet nouvellement créée :

```
[root@localhost ~]# docker container run ourfiglet figlet hello
 _          _ _
| |__   ___| | | ___
| '_ \ / _ \ | |/ _ \
| | | |  __/ | | (_) |
|_| |_|\___|_|_|\___/
```

Création d'images à l'aide d'un Dockerfile

Au lieu de créer une image binaire statique, nous pouvons utiliser un fichier appelé Dockerfile pour créer une image.

Nous allons utiliser un exemple simple et créer une application «hello world» dans Node.js.

```
[root@localhost ~]# mkdir hellojs && cd hellojs/
```

Nous allons commencer 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* :

```
[root@localhost hellojs]# vim index.js
var os = require("os");
var hostname = os.hostname();
console.log("hello from " + hostname);
```

Le fichier que nous venons de créer est le code javascript de notre serveur. Nous allons Dockerizer cette application en créant un Dockerfile. Nous utiliserons alpine comme image du système d'exploitation de base, ajouterons un runtime Node.js, puis copierons 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-y le contenu suivant :

```
[root@localhost hellojs]# vim Dockerfile
FROM alpine
RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app
CMD ["node","index.js"]
```

Ce Dockerfile effectue les opérations suivantes:

- Indique depuis (**FROM**) où récupérer l'image de base - alpine.
- Ensuite, il exécute (**RUN**) les deux commandes (apk update et apk add) à l'intérieur de ce conteneur qui installent le serveur Node.js.
- Ensuite, nous lui avons dit de copier (**COPY**) les fichiers de notre répertoire de travail dans le conteneur. Le seul fichier que nous avons actuellement est index.js.
- Ensuite, nous spécifions le **WORKDIR** - le répertoire que le conteneur doit utiliser au démarrage
- Et enfin, nous avons donné à notre conteneur une commande (**CMD**) à exécuter lorsque le conteneur démarre.

Construisons notre première image à partir de ce Dockerfile et appelons-la bonjour:v0.1 :

```
[root@localhost hellojs]# docker image build -t hello:v0.1 .
Sending build context to Docker daemon 3.072 kB
Step 1/5 : FROM alpine
 ---> e7d92cdc71fe
Step 2/5 : RUN apk update && apk add nodejs
 ---> Running in d9d494d249c1

fetch http://dl-
cdn.alpinelinux.org/alpine/v3.11/main/x86_64/APKINDEX.tar.gz
fetch http://dl-
cdn.alpinelinux.org/alpine/v3.11/community/x86_64/APKINDEX.tar.gz
v3.11.3-120-g02b0001e98 [http://dl-cdn.alpinelinux.org/alpine/v3.11/main]
v3.11.3-122-g82e6adffb8 [http://dl-
cdn.alpinelinux.org/alpine/v3.11/community]
OK: 11268 distinct packages available
(1/7) Installing ca-certificates (20191127-r1)
(2/7) Installing c-ares (1.15.0-r0)
(3/7) Installing libgcc (9.2.0-r3)
(4/7) Installing nghttp2-libs (1.40.0-r0)
(5/7) Installing libstdc++ (9.2.0-r3)
(6/7) Installing libuv (1.34.0-r0)
(7/7) Installing nodejs (12.15.0-r1)
Executing busybox-1.31.1-r9.trigger
Executing ca-certificates-20191127-r1.trigger
OK: 36 MiB in 21 packages
 ---> 97a3be673d92
Removing intermediate container d9d494d249c1
Step 3/5 : COPY . /app
 ---> fe7f14902d17
Removing intermediate container 0339438341db
Step 4/5 : WORKDIR /app
 ---> c8a13dd01f15
Removing intermediate container 2bf559a1a878
Step 5/5 : CMD node index.js
 ---> Running in c42b0d3a848b
 ---> f3a8fe233a36
Removing intermediate container c42b0d3a848b
Successfully built f3a8fe233a36

[root@localhost hellojs]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello               v0.1                f3a8fe233a36        21 minutes ago      39 MB
ourfiglet           latest              a6f74cd34651        53 minutes ago      289 MB
```

Nous démarrons ensuite un conteneur pour vérifier que nos applications fonctionnent correctement:

```
[root@localhost hellojs]# docker container run hello:v0.1
hello from 47249432d78a
```

Couches d'images

Lors de l'exécution, les conteneurs se comportent comme un seul système d'exploitation et une seule application. Mais les images elles-mêmes sont en fait construites en couches.

Si vous regardez la sortie de la commande *docker image build* précédente, vous remarquerez qu'il y avait 5 étapes et chaque étape avait plusieurs tâches. Vous devriez voir plusieurs tâches «search» et «pull» où Docker récupère diverses données depuis Docker Store ou d'autres endroits. Ces données ont été utilisées pour créer une ou plusieurs couches (layers) de conteneurs.

Tout d'abord, consultez l'image que vous avez créée précédemment à l'aide de la commande history (n'oubliez pas d'utiliser la commande *docker image ls* des exercices précédents pour trouver vos ID d'image):

```
[root@localhost hellojs]# docker image history f3a8fe233a36
IMAGE          CREATED         CREATED BY                                      SIZE    COMMENT
f3a8fe233a36   47 minutes ago  /bin/sh -c #(nop)  CMD ["node" "index.js"]     0 B
c8a13dd01f15   47 minutes ago  /bin/sh -c #(nop) WORKDIR /app                  0 B
fe7f14902d17   47 minutes ago  /bin/sh -c #(nop) COPY dir:ec8ec91247a7b1a..   187 B
97a3be673d92   47 minutes ago  /bin/sh -c apk update && apk add nodejs        33.4 MB
e7d92cdc71fe   7 weeks ago     /bin/sh -c #(nop)  CMD ["/bin/sh"]             0 B
<missing>      7 weeks ago     /bin/sh -c #(nop) ADD file:e69d441d729412d..   5.59 MB
```

Ce que vous voyez est la liste des images de conteneurs intermédiaires qui ont été construites en cours de route pour créer votre image d'application Node.js finale. Certaines de ces images intermédiaires deviendront des couches dans votre image de conteneur finale. Dans la sortie de la commande history, les couches alpines d'origine sont en bas de la liste, puis chaque personnalisation que nous avons ajoutée dans notre Dockerfile est sa propre étape dans la sortie. Grace à ce concept si nous devons apporter des modifications à notre application, cela peut n'affecter qu'une seule couche! Pour voir cela, nous allons modifier un peu notre application et créer une nouvelle image.

```
[root@localhost hellojs]# echo "console.log(\"this is v0.2\");" >> index.js

[root@localhost hellojs]# cat index.js
var os = require("os");
var hostname = os.hostname();
console.log("hello from " + hostname);
console.log("this is v0.2");
```

Cela ajoutera une nouvelle ligne au bas de votre fichier index.js afin que votre application génère une ligne de texte supplémentaire. Nous allons maintenant créer une nouvelle image en utilisant notre code mis à jour. Nous marquerons également notre nouvelle image pour la marquer comme une nouvelle version afin que quiconque consomme nos images plus tard puisse identifier la bonne version à utiliser:

```
[root@localhost hellojs]# docker image build -t hello:v0.2 .
Sending build context to Docker daemon 3.072 kB
Step 1/5 : FROM alpine
 ---> e7d92cdc71fe
Step 2/5 : RUN apk update && apk add nodejs
 ---> Using cache
 ---> 97a3be673d92
Step 3/5 : COPY . /app
```

```
 ---> 76af62d930d1
Removing intermediate container 27cbe7f8c20b
Step 4/5 : WORKDIR /app
 ---> 72299e2ee2ee
Removing intermediate container 5c2fb396c37f
Step 5/5 : CMD node index.js
 ---> Running in 3385810c71e4
 ---> 759959768615
Removing intermediate container 3385810c71e4
Successfully built 759959768615
```

Remarquez quelque chose d'intéressant dans les étapes de construction cette fois. Dans la sortie, il passe par les cinq mêmes étapes, mais notez que dans certaines étapes, il indique Utilisation du cache.

Docker a reconnu que nous avions déjà créé certaines de ces couches dans nos versions d'image précédentes et que rien n'avait changé dans ces couches, il pouvait simplement utiliser une version mise en cache de la couche, plutôt que de tirer vers le bas du code une deuxième fois et d'exécuter ces étapes.

```
[root@localhost hellojs]# docker image ls
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
hello                   v0.2                759959768615        3 minutes ago       39 MB
hello                   v0.1                f3a8fe233a36        About an hour ago   39 MB
ourfiglet               latest              a6f74cd34651        About an hour ago   289 MB
docker.io/alpine        latest              e7d92cdc71fe        7 weeks ago         5.59 MB
docker.io/centos        latest              470671670cac        7 weeks ago         237 MB
docker.io/hello-world   latest              fce289e99eb9        14 months ago       1.84 kB

[root@localhost hellojs]# docker container run hello:v0.2
hello from ab9c8d88b94f
this is v0.2
```

Inspection d'image

Maintenant, inversons un peu notre réflexion. Que se passe-t-il si nous obtenons un conteneur de Docker Store ou d'un autre registre et que nous voulons en savoir un peu plus sur le contenu du conteneur que nous consommons? Docker dispose d'une commande d'inspection des images et renvoie des détails sur l'image du conteneur, les commandes qu'il exécute, le système d'exploitation et plus encore.

```
[root@localhost hellojs]# docker image inspect alpine
[
    {
        "Id":
"sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a99776a",
        "RepoTags": [
            "docker.io/alpine:latest"
        ],
        "RepoDigests": [

"docker.io/alpine@sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3
d1172b11c4367d"
        ],
…
```

Nous pouvons utiliser certains filtres pour inspecter simplement des détails particuliers sur l'image. 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 simplement des données spécifiques de l'image.

Obtenons la liste des couches:

```
[root@localhost hellojs]# docker image inspect --format "{{ json
.RootFS.Layers }}" alpine
```

Alpine n'est qu'une petite image du système d'exploitation de base, il n'y a donc qu'une seule couche

```
["sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb10"]
```

Essayez avec l'image *ourfiglet* :

```
[root@localhost hellojs]# docker image inspect --format "{{ json
.RootFS.Layers }}" hello:v0.2
["sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb10",
"sha256:ffb2ac27e247c8bd63a94fddd708e6875cf2eb83e88117baf5c23b6dc95bda01","
sha256:0cd1b1213f8a0609bd7fca2f401e281de581d767674ee5dda7a8d1430bfb1f34"]
```

Nous avons trois couches dans notre application. Rappelons que nous avions l'image Alpine de base (la commande FROM dans notre Dockerfile), puis nous avions une commande RUN pour installer certains packages, puis nous avions une commande COPY à ajouter dans notre code javascript. Ce sont nos couches! Si vous regardez de plus près, vous pouvez même voir que alpin et hello utilisent la même couche de base, ce que nous savons car ils ont le même hachage sha256.

Remarque importante sur les couches: chaque couche est immuable. Lorsqu'une image est créée et que des couches successives sont ajoutées, les nouvelles couches gardent une trace des modifications de la couche en dessous. Lorsque vous démarrez le conteneur, une couche supplémentaire est utilisée pour suivre les modifications qui se produisent lors de l'exécution de l'application (comme le fichier «hello.txt» que nous avons créé dans les exercices précédents).

Plus avec Dockerfile

Dans ce Lab, nous couvrirons Dockerfile en profondeur, ainsi que les meilleures pratiques à utiliser.

RQ

Pour la suite du lab demandez à récupérer le fichier « html.tar.gz ».

Prenons un exemple :

```
[root@localhost ~]# mkdir -p nginx/files
[root@localhost ~]# cp html.tar.gz nginx/files

[root@localhost ~]# cd nginx/
[root@localhost nginx]# vim Dockerfile
FROM alpine:latest
LABEL maintainer="Elies Jebri <elies.jebri@gmail.com>"
LABEL description="Cet exemple installe NGINX via Dockerfile."
RUN apk add --update nginx && \
        rm -rf /var/cache/apk/* && \
        mkdir -p /tmp/nginx/
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/default.conf /etc/nginx/conf.d/default.conf
ADD files/html.tar.gz /usr/share/nginx/
EXPOSE 80/tcp
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

```
[root@localhost nginx]# vim files/nginx.conf
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
    log_format main '$remote_addr - $remote_user [$time_local] "$request"'
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for"';
    access_log  /var/log/nginx/access.log  main;
    sendfile        off;
    keepalive_timeout  65;
    include /etc/nginx/conf.d/*.conf;
}
```

```
[root@localhost nginx]# vim files/default.conf
server {
        location / {
          root /usr/share/nginx/html;
        }
}
```

```
[root@localhost nginx]# docker image build --tag eliesjebri/mynginx:1.0 .
Sending build context to Docker daemon 213.5 kB
Step 1/10 : FROM alpine:latest
 ---> e7d92cdc71fe
Step 2/10 : LABEL maintainer "Elies Jebri <elies.jebri@gmail.com>"
 ---> Running in 9c599fe90e2f
 ---> d4141e19a26a
Removing intermediate container 9c599fe90e2f
Step 3/10 : LABEL description "Cet exemple de Dockerfile installe NGINX."
 ---> Running in be99e72cd9e6
 ---> e1c5ece48ee8
Removing intermediate container be99e72cd9e6
Step 4/10 : RUN apk add nginx &&      rm -rf /var/cache/apk/* &&     mkdir
-p /tmp/nginx/
 ---> Running in a7939012940d
fetch http://dl-
cdn.alpinelinux.org/alpine/v3.11/main/x86_64/APKINDEX.tar.gz
fetch http://dl-
cdn.alpinelinux.org/alpine/v3.11/community/x86_64/APKINDEX.tar.gz
(1/2) Installing pcre (8.43-r0)
(2/2) Installing nginx (1.16.1-r6)
Executing nginx-1.16.1-r6.pre-install
Executing busybox-1.31.1-r9.trigger
OK: 7 MiB in 16 packages
 ---> 4bf3fd59b197
Removing intermediate container a7939012940d
Step 5/10 : COPY files/nginx.conf /etc/nginx/nginx.conf
 ---> fb397d3a2fda
Removing intermediate container 4ce77f544fa0
Step 6/10 : COPY files/default.conf /etc/nginx/conf.d/default.conf
 ---> c12400e90a81
Removing intermediate container 285e607e95d3
Step 7/10 : ADD files/html.tar.gz /usr/share/nginx/
 ---> 5c06f7da5d36
Removing intermediate container 29b95ef43860
Step 8/10 : EXPOSE 80/tcp
 ---> Running in 806b5dd78f6c
 ---> ae3210ad95be
Removing intermediate container 806b5dd78f6c
Step 9/10 : ENTRYPOINT nginx
 ---> Running in 4a055609b2c0
 ---> 5bc8d2204c9e
Removing intermediate container 4a055609b2c0
Step 10/10 : CMD -g daemon off;
 ---> Running in 8c24eef50788
 ---> 8ff5b48c7d0d
Removing intermediate container 8c24eef50788
Successfully built 8ff5b48c7d0d

[root@localhost nginx]# docker images
REPOSITORY              TAG     IMAGE ID        CREATED             SIZE
eliesjebri/mynginx      1.0     8ff5b48c7d0d    26 seconds ago      7.21 MB

[root@localhost nginx]# docker login -u eliesjebri -p xxx docker.io
Login Succeeded
[root@localhost nginx]# docker push eliesjebri/mynginx:1.0
The push refers to a repository [docker.io/eliesjebri/mynginx]
b9637c2b060a: Pushed
d9833dbadb3f: Pushed
d41c1cf96cc7: Pushed
ce838a5b05ef: Pushing [=====================================>] 1.453 MB
```

```
5216338b40a7: Pushing [======================>
```

Exécuter le conteneur mynginx en arrière-plan

Les conteneurs d'arrière-plan permettent d'exécuter la plupart des applications en arrière plan et de vous redonner l'invite de commande.

Exécutez un nouveau conteneur mynginx avec la commande suivante.

```
[root@localhost nginx]# docker container run --detach -p 8080:80 --name
mynginx1 eliesjebri/mynginx:1.0

[root@localhost nginx]# docker container ls
CONTAINER ID   IMAGE                  COMMAND                  CREATED
STATUS     PORTS  NAMES
5d3c66beb17f   eliesjebri/mynginx:1.0  "nginx -g 'daemon ..."  2 seconds
ago       Up 1 second         0.0.0.0:8080->80/tcp   nginx1

[root@localhost nginx]# curl http://localhost:8080
<!doctype html>
<html class="no-js" lang="">
...
        <p>Hello world! This is being served from Docker.</p>

...
    </body>
</html>
```

--detach (-d) exécutera le conteneur en arrière-plan.

--name le nommera mynginx1.

Remarque :

```
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

Cela signifie que chaque fois que nous lançons un conteneur à partir de notre image, le binaire nginx est exécuté, comme nous l'avons défini comme notre `ENTRYPOINT`, puis tout ce que nous avons comme CMD est exécuté, ce qui nous donne l'équivalent de l'exécution de la commande suivante:

```
$ nginx -g daemon off;
```

Un autre exemple de la façon dont ENTRYPOINT peut être utilisé est le suivant:

```
$ docker container run --name mynginx1 mynginx -v
```

Ce serait l'équivalent d'exécuter la commande suivante sur notre hôte:

```
$ nginx -v;
```

Comme nous avons le binaire nginx comme ENTRYPOINT, toute commande que nous transmettons remplace le CMD qui avait été défini dans Dockerfile.

```
[root@localhost nginx]# docker container run  eliesjebri/mynginx:1.0 -v
nginx version: nginx/1.16.1
```

Limites des ressources

Par défaut, lors de son lancement, un conteneur sera autorisé à consommer toutes les ressources disponibles sur la machine hôte s'il en a besoin.

```
[root@localhost ~]# docker container run -d --name nginx-test -p 8080:80
nginx
2e77e4f302096a1ca16d00490ae2a135e901409593584121282a418569c24664

[root@localhost ~]# docker container inspect nginx-test | grep -i memory
            "Memory": 0,
            "KernelMemory": 0,
            "MemoryReservation": 0,
            "MemorySwap": 0,
            "MemorySwappiness": -1,
```

"Memory": 0 La valeur 0 veut dire illimité.

Nous pouvons limiter les ressources que nos conteneurs peuvent consommer;

En général, nous pouvons les limiter lors du lancement du conteneur à l'aide de la commande run; Cependant, nous n'avons pas lancé notre conteneur *nginx-test* avec des limites de ressources, ce qui signifie que nous devons mettre à jour notre conteneur déjà en cours d'exécution; pour ce faire, nous pouvons utiliser la commande update:

```
[root@localhost ~]# docker container update --cpu-shares 512 --memory 128M
--memory-swap 256M nginx-test
nginx-test
[root@localhost ~]# docker container inspect nginx-test | grep -i memory
            "Memory": 134217728,
            "KernelMemory": 0,
            "MemoryReservation": 0,
            "MemorySwap": 268435456,
            "MemorySwappiness": -1,
```

pour réduire de moitié la priorité du processeur, définir une limite de mémoire de 128 Mo et du swap 2x la ram.

Pour le faire lors du lancement du conteneur :

```
[root@localhost ~]# docker container run -d --name nginx-test-restrict --
cpu-shares 512 --memory 128M -p 8081:80 nginx
421326e79db71a1bb8a4c8593d6d516e14bc20df63870361ddb94f68b53e3d63
[root@localhost ~]#
[root@localhost ~]# docker container inspect nginx-test-restrict | grep -i
memory
            "Memory": 134217728,
            "KernelMemory": 0,
            "MemoryReservation": 0,
            "MemorySwap": 268435456,
            "MemorySwappiness": -1,
```

Arrêtez et retirez tous les conteneurs

Vous pouvez obtenir une liste de tous les conteneurs Docker sur votre système à l'aide de la commande :

```
[root@localhost nginx]# docker container ls -aq
b4f3584fd8d9
caa519b23545
a7754851d96a
ab9c8d88b94f
47249432d78a
8d4fd48f661b
944107f2ed58
```

Pour arrêter tous les conteneurs en cours d'exécution, utilisez la commande :

```
[root@localhost nginx]# docker container stop $(docker container ls -aq)
```

Une fois que tous les conteneurs sont arrêtés, vous pouvez les supprimer à l'aide de la commande :

```
[root@localhost nginx]# docker container rm $(docker container ls -aq)
```

Vous pouvez aussi utiliser la commande de suppression, docker prune.

Avant d'exécuter la commande de suppression, vous pouvez obtenir une liste de tous les conteneurs non exécutés (arrêtés) qui seront supprimés à l'aide de la commande suivante:

```
$ docker container ls -a --filter status=exited --filter status=created
```

Puis

```
$ docker container prune --filter status=exited --filter status=created
```

Sinon tout simplement:

```
[root@localhost nginx]# docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
b4f3584fd8d9d456881f3fcec3e2ed271e973b8752cbbf4430dec514c3b71436
…
[root@localhost nginx]# docker container ls
CONTAINER ID  IMAGE          COMMAND    CREATED        STATUS         NAMES
7d374cac4d5b  local:mynginx  "nginx -g" 6 minutes ago  Up 6 minutes
```

Il ne restera que le conteneur en cours d’exécution.