DOCUMENTATION DOCKER **Docker**

15 février 2023

1 Plan de cours

Séance	chapitre
1	• Généralité. • Installer docker. • Lancer notre première container.
2	• Dockerfile : Créer une image. • Fonctionnement du cache.
3	• Comprendre les Layers / Couches. • Les Réseaux. • Docker Compose. • ENTRYPOINT VS CMD.
4	• Sécuriser le user namespace. • Dokerfile multi-stage. • Les images : TAGS, PULL ET PUSH. • L'API Docker.
5	• Développent des modules pour **PORTE-IFNTI**.

2 Généralités

Dans cette première partie, nous allons découvrir les **conteneurs** ainsi qu'avec Docker. Mais avant cela, nous allons revenir sur quelque notions importantes :

- comprendre la notion de **machine virtuelle** ;
- comprendre la notion de **conteneur** ;
- **pourquoi** utiliser les conteneurs ?

2.1 Machine virtuelle (VM)

Utiliser une machine virtuelle c'est faire de la **virtualisation lourde**. Pourquoi car nous recréons un système complet (avec ses propres ressource) dans notre système.

L'isolation avec le système hôte est donc totale ; cependant cela nous apporte plusieurs contraintes et avantages.

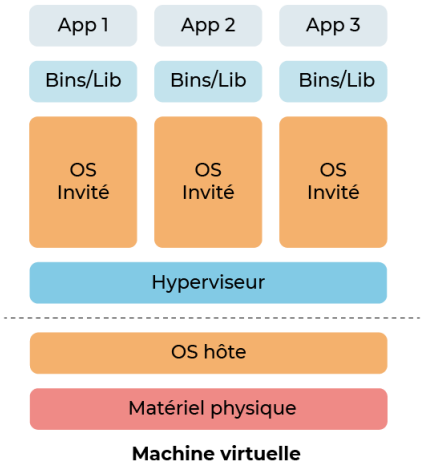


FIGURE 1 – Fonctionnement d'une machine virtuelle

Contraint.

- Une machine virtuelle prend du **temps** à démarrer ;
- Une machine virtuelle **réserve les ressources (CPU/RAM)** sur le système hôte.

Avantages.

- Une machine virtuelle est totalement **isolée** du système hôte ;
- Les ressource attribuées à une machine virtuelle lui sont totalement **réservées**.
- Vous pouvez installer **différents OS** (Linux, Windows, etc.).

Mais il arrive souvent que l'application qu'elle fait tourner ne consomme pas l'ensemble des ressources disponibles sur la machine virtuelle. Alors est né un nouveau système de virtualisation plus léger : les **conteneurs**.

2.2 Conteneur

Un conteneur Linux est un **processus** ou un ensemble de processus isolés du reste du système, tout en étant **légers**.
Le conteneur permet de faire de la **virtualisation légère**, c'est à dire qu'il ne virtualise pas les ressources, il ne crée qu'une **isolation des processus**. Le conteneur partage donc les ressources avec le système hôte.

Attention, les conteneurs existent depuis plus longtemps que **Docker**. **OpenVZ** ou **LXC** sont des technologies de conteneur qui existent depuis de nombreuses années.

Les conteneurs, au sens d'OpenVZ et LXC, apportent une **isolation importante des processus systèmes** ; cependant, les ressources CPU, RAM et disque sont totalement partagées avec l'ensemble du système. Les conteneurs partagent entre eux le kernel Linux ; ainsi, il n'est pas possible de faire fonctionner un système Windows ou BSD dans celui-ci.

Maintenant voyons quelque exemple des conteneurs.

— **Ne réservez que les ressources nécessaires** Une autre différence importante avec les machines virtuelles est qu'un conteneur **ne réserve pas** la quantité de CPU, RAM et disque attribuée auprès du système hôte. Ainsi, nous pouvons allouer 16 Go de RAM à notre conteneur, mais si celui-ci n'utilise que 2 Go, le reste ne sera pas verrouillé.

— **Démarrez rapidement vos conteneurs** Les conteneurs n'ayant pas besoin d'une virtualisation des ressources mais seulement d'une isolation, ils peuvent **démarrer beaucoup plus rapidement** et plus fréquemment qu'une machine virtuelle sur nos serveurs

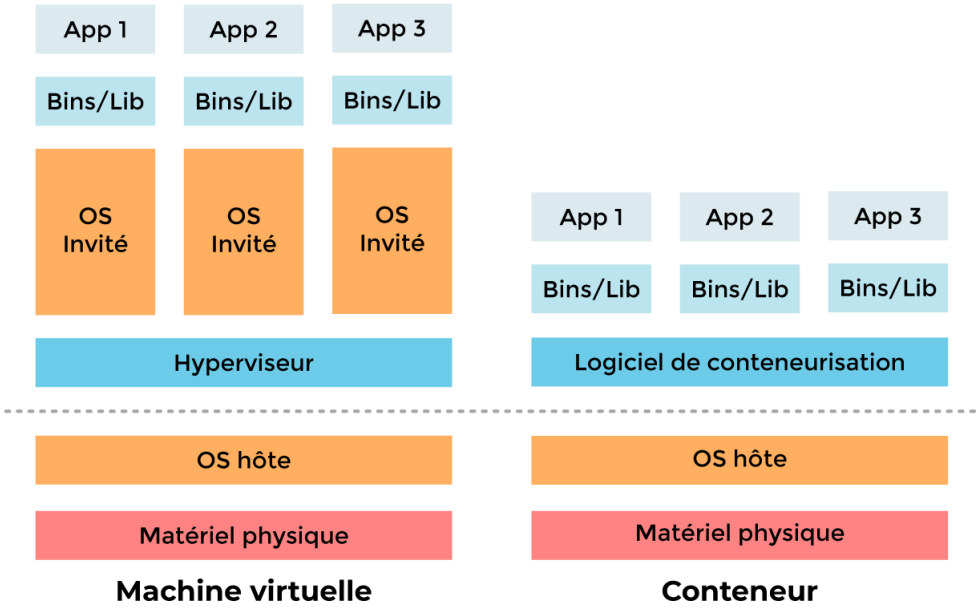


FIGURE 2 – Fonctionnement d'une machine virtuelle

hôtes, et ainsi réduire encore un peu les frais de l'infrastructure.

— **Donnez plus d'autonomie à vos développeurs** En dehors de la question pécuniaire, il y a aussi la possibilité de faire tourner des conteneurs sur le poste des développeurs, et ainsi de réduire les différences entre la "sainte" production, et l'environnement local sur le poste des développeurs.

2.3 Pourquoi utiliser des conteneurs ?

Les conteneurs permettent de **réduire les coûts**, d'augmenter la **densité de l'infrastructure**, tout en améliorant le cycle de déploiement.

Grace à leur capacité de démarrer plus rapidement, les conteneurs sont souvent utilisés en production pour ajouter des ressources disponibles, et ainsi répondre à des besoins de mise à l'échelle ou de scalabilité. Mais ils répondent aussi à des besoins de préproduction ; en étant légers et rapides au démarrage, il permettent de créer des environnements dynamique et ainsi de répondre à des besoins métier.

3 Docker

3.1 Une petite histoire

Docker a été créé pour les besoins d'une société de Platform as a Service (PaaS) appelée **DotCloud**. Finalement, en mars 2013, l'entreprise a créé une nouvelle structure nommée **Docker Inc** et a placé en open source son produit **Docker**.

3.2 Objectifs

Docker apporte une notion importante dans le monde du conteneur. Dans la vision Docker, un conteneur ne doit faire tourner qu'**un seul processus**. Ainsi, dans le cas d'une stack LAMP (Linux, Apache, MySQL, PHP), nous devons créer **3 conteneurs** différents, un pour Apache, un pour MySQL et un dernier pour PHP. Alors que dans un conteneur LXC ou OpenVZ, nous aurions fait tourner l'ensemble des 3 services dans un seul et unique conteneur.

3.3 Pourquoi utiliser Docker ?

Docker répond à une problématique forte dans le monde du développement. Prenons un exemple : vous avez développé votre projet de Twitter Lite en local. Tout fonctionne bien, mais au moment de mettre en production, vous vous rendez compte que vous ne savez pas comment **déployer votre projet**. Un autre exemple : vous êtes dans une équipe de 10 personnes et chacun utilise un OS différent (Ubuntu, macOS, Windows, CentOS, etc.). Comment faire pour avoir **un environnement unifié et fonctionnel** chez l'ensemble des développeurs ?

Docker répond à ces problématiques en créant des conteneurs. Grâce à Docker, vous n'aurez plus de problème de différence d'environnement, et votre code marchera partout !

3.4 Installer Docker

Docker Inc distribue 3 versions de Docker différentes :

- Docker Community Edition (Linux seulement) ;
- Docker Desktop (Mac ou Windows) ;
- Docker Enterprise (Linux seulement).

Docker Desktop et Docker Community Edition (CE) sont deux versions de Docker gratuites. Avec les deux solutions, vous aurez un Docker fonctionnel sur votre ordinateur.

Si vous êtes sous Windows ou macOS, utilisez Docker Desktop qui va créer pour vous l'ensemble des services nécessaires au bon fonctionnement de Docker.

Si vous êtes sous Linux, prenez la version Community Edition (CE) ; vous utiliserez aussi cette version pour vos serveurs.

La version Docker Enterprise ne ressemble pas du tout aux versions Desktop et CE. Celle-ci répond à des besoins plus poussés des entreprises, et propose une interface de gestion d'infrastructures sous Docker. Cette version est soumise à une licence fournie par Docker Inc.

NB : Pour installer docker sur Mac ou Windows rendez vous ici. Dans le cas de Linux, vous devez utiliser les commandes suivante.

```
sudo aptget install docker.io
```

Après ou avant l'installation de docker il est conseiller de créer votre compte au niveau

4 Lancer notre premier conteneur

Dans cette partie, je vous propose de **prendre en main Docker**. Nous allons commencer par découvrir l'**interface en ligne de commande**, qui nous permet de discuter avec le **daemon Docker** installé précédemment. D'ici la fin de cette partie, vous serez capable de **lancer et gérer vos conteneurs**. Mais commençons dans ce chapitre par comprendre ce qu'est le **Docker Hub**, puis nous lancerons notre **premier conteneur**.

4.1 Le Docker Hub

Avant de démarrer votre premier conteneur Docker, rappelez-vous quand vous avez créé votre compte sur le Docker Hub pour télécharger votre version de Docker. Celui-ci est aussi **la registry officielle de Docker**.

Une **registry** est un logiciel qui permet de partager des images à d'autres personnes. C'est un composant majeur dans l'écosystème Docker, car il permet :

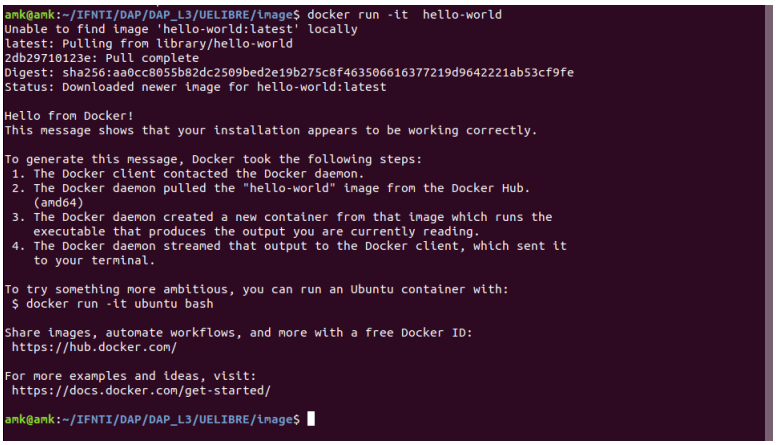
— à des développeurs de distribuer des images prêtes à l'emploi et de les versionner avec un système de tags ;

— à des outils d'intégration en continu de jouer une suite de tests, sans avoir besoin d'autre chose que de Docker ;
— à des systèmes automatisés de déployer ces applications sur vos environnements de développement et de production.

4.2 Démarrez votre premier conteneur Docker

Pour démarrer votre premier conteneur, vous devez utiliser la commande :

```
docker run hello-world
```



Quand vous utilisez cette commande, le **daemon Docker** va chercher si l'image *hello-world* est **disponible en local**. Dans le cas contraire, il va la **récupérer sur la registry Docker officielle.**

Dans notre cas, le conteneur a démarré, puis affiché du contenu, et il a fini par s'arrêter. Si vous souhaitez **que votre conteneur reste allumé jusqu'à l'arrêt du service qu'il contient**, vous devez ajouter l'argument - *-detach (-d)* . Celui-ci permet de ne pas rester attaché au conteneur, et donc de pouvoir lancer plusieurs conteneurs. Nous allons voir dans la section suivante comment utiliser l'argument -d.

4.3 Démarrez un serveur Nginx avec un conteneur Docker

vous savez lancer un conteneur, et vous avez compris les actions effectuées par le daemon Docker lors de l'utilisation de la commande docker run.

Maintenant, nous allons aller plus loin avec celui-ci. Nous allons lancer un conteneur qui démarre un serveur Nginx en utilisant deux options (**-d** et **-p**) :

```
docker run -d -p 8080:80 nginx
```

. Dans cette commande, nous avons utilisé deux options :
— -d pour détacher le conteneur du processus principal de la console. Il vous permet de continuer à utiliser la console pendant que votre conteneur tourne sur un autre processus ;
— -p pour définir l'utilisation de ports. Dans notre cas, nous lui avons demandé de transférer le trafic du port 8080 vers le port 80 du conteneur.

Ainsi, en vous rendant sur l'adresse http ://127.0.0.1 :8080, vous aurez la page par défaut de Nginx.

Vous pourriez aussi avoir besoin de "rentrer" dans votre conteneur Docker pour pouvoir y effectuer des actions. Pour cela, vous devez utiliser la commande **docker exec -ti ID_RETOURNÉ_LORS_ bash** . Dans cette commande, l'argument *-ti* permet d'avoir un shell bash pleinement opérationnel. Une fois que vous êtes dans votre conteneur, vous pouvez vous rendre, via la commande **cd /usr/share/nginx/html** , dans le répertoire où se trouve le fichier index.html , pour modifier son contenu et voir le résultat en direct à l'adresse http ://127.0.0.1 :8080.

4.4 Récupérez une image depuis le docker Hub

Vous pouvez aussi avoir besoin de récupérer des images sur le Docker Hub sans pour autant lancer de conteneur. Pour cela, vous avez besoin de lancer la commande suivante :

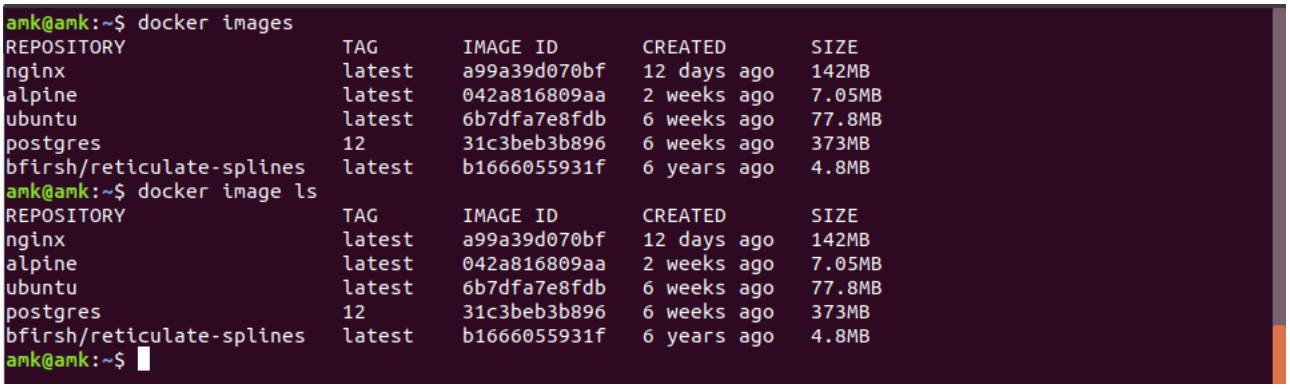
```
 docker pull hello-world

Using default tag: latest

latest: Pulling from library/hello-world

Digest: sha256:2557e3c07ed1e38f26e389462d03ed943586f744621577a99efb77324b0fe535

Status: Image is up to date for hello-world:latest
```

En lançant cette commande, vous téléchargez une image directement depuis le Docker Hub, et vous la stockez en local sur votre ordinateur.

NB : Généralement le pull d'une image peut prendre, beaucoup de seconde voir minute.

4.5 Quelles que commandes

- **docker images** ou **docker image ls** : Affiche la liste de toute les images.



  On peut facilement repérer les métadonnées suivantes :
  `REPOSITORY TAG IMAGE ID CREATED SIZE`
  — **REPOSITORY** désigne souvent le nom de l'image.
  — **TAG** sa version.
  — **IMAGE ID** son identifiant (unique).
  — **CREATED** l'heur à la quelle elle à été créer et enfin
  — **SIZE** L'espace mémoire quelle occupe.
- **docker ps** ou **docker container ls** : Affiche la liste de tous les conteneurs dans l'état **RUNING**. pour afficher tous les conteneurs vous allez devoir spécifier l'option **-a** pour *all*.
- **docker stop ID_RETOURNÉ_LORS_DU_DOCKER_RUN** permet d'arrêter un conteneur.
- **docker rm ID_RETOURNÉ_LORS_DU_DOCKER_RUN** ou **docker rm NOM_CONTENEURE** permet de supprimer un conteneur.
  NB : Cela ne fonctionnera si et seulement si le conteneur est stoppé. Pour contourner ce problème vous pouvez utiliser l'option -f (pour *force*) comme ceci **docker rm -f ID_RETOURNÉ_LORS_DU_DOCKER_RUN**
- **docker image rm IMAGE_ID** ou **docker rmi IMAGE_ID** permet de supprimez une image. Vous pouvez aussi utiliser ici l'option -f.

- **docker inspect ID_RETOURNÉ_LORS_DU_DOCKER_RUN** pour avoir des détails plus détaillé sur le conteneur.

4.6 Comment nettoyer son système docker

Après avoir fait de nombreux tests sur votre ordinateur, vous pouvez avoir besoin de faire un peu de ménage. Pour cela, vous pouvez supprimer l'ensemble des ressources manuelles dans Docker.
Ou vous pouvez laisser faire Docker pour qu'il fasse lui-même le ménage. Voici la commande que vous devez utiliser pour faire le ménage : **docker system prune**.

```
 docker system prune
WARNING! This will remove:
- all stopped containers
- all networks not used by at least one container
- all dangling images
- all dangling build cache
Are you sure you want to continue? [y/N] y
Deleted Containers:
941b8955b4fd8988fefe2aa91c7eb501f2d4f8c56bf4718fea8ed50904104745
a96e73c623fb6530ab41db6a82aca7017d54a99590f0b45eb6bf934ef8e4d3ed

Deleted Images:
deleted: sha256:797a90d1aff81492851a11445989155ace5f87a05379a0fd7342da4c4516663e
deleted: sha256:c5c8911bd17751bd631ad7ed00203ba2dcb79a64316e14ea95a9edeb735ca3ea

Total reclaimed space: 21.08MB
```

Celle-ci va supprimer les données suivantes :

- l'ensemble des conteneurs Docker qui ne sont pas en status running ;
- l'ensemble des réseaux créés par Docker qui ne sont pas utilisés par au moins un conteneur ;
- l'ensemble des images Docker non utilisées ;
- l'ensemble des caches utilisés pour la création d'images Docker.

5 Les Volumes

Lorsque vous créer un conteneur, par défaut ses données ne sont pas persistant. C'est à dire que si vous supprimé un conteneur ses données disparaîtrons aussi. Pour éviter que cela ne se produise docker à mis en place la notion de volume.
Docker volume est un **mécanisme de fichiers géré par Docker permettant de sauvegardé des données générées lors de l'exécution d'un conteneur**. Il permet également de monter les données dont le conteneur a besoin à l'intérieur de ce dernier lors de son lancement. Mais avant de voir **Docker volume** nous allons voir les volume persistant.

5.1 Les volumes persistant

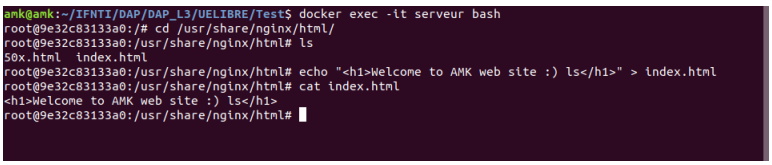
Vous vous souvenez du serveur nginx que nous avions créer ? si vous l'avez déjà supprimé, je vous pris de le recréer. Voici la commande si vous l'avez oublié

```
docker run -d --name seveur -p 8080:80 nginx
```

Une fois créer entrer dans le conteneur en utilisant la commande suivante :

```
docker exec -it serveur
```

Rendez-vous ensuite dans le fichier index.html et modifier le à votre guise. Mais souvenez vous il n'y pas de vim ou de nano par



Voici le résultat.

Maintenant supprimé votre conteneur et recréer le.

```
docker rm -f serveur
docker run -d --name serveur -p 8080:80 nginx
```

Que se passe t-il ? Vos modification n'ont pas été pris en compte et vous voyez la page par défaut de nginx ! Pour régler ce problème nous allons utiliser un volume. créer un répertoire nommé **public_html**. Créez s'y le fichier **index.html**(mettez y ce que vous voulez). Comme la redirection de port, nous allons faire le mappage de répertoire en disant que le répertoire **/urs/share/nginx/html** doit correspondre au répertoire **public_html**. Vous comprenez donc pourquoi nous devons créer nous même le fichier **index.html**. Créez maintenant notre conteneur.

```
docker run -d --name serveur -p 8001:80 -v /home/amk/IFNTI/DAP/DAP_L3/UELIBRE/Test/test/
```

Supprimer votre conteneur et recréer le que se passe t'il ? Vos données sont devenue persistant ! C'est cool n'est pas.

Dans la suit nous allons voir docker volume.

5.2 Docker volume

Docker volume nous permet de créer des volumes à volonté. Par défaut les volumes que nous créer on leurs point de montage au niveau de la machine hôte dans le répertoire /**var**/**lib**/**docker**/**volumes**/.
Pour voir tous ce qu'on peut faire avec docker volume (et cela vaut pour toute les commande) vous n'avez qu'a taper **docker volume** dans votre terminal vous verrez ça.

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$ docker volume

Usage:  docker volume COMMAND

Manage volumes

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

Run 'docker volume COMMAND --help' for more information on a command.
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$
```

Nous allons créer un volume nommé monvolume voici la commande.

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$ docker volume create monvolume
monvolume
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$
```

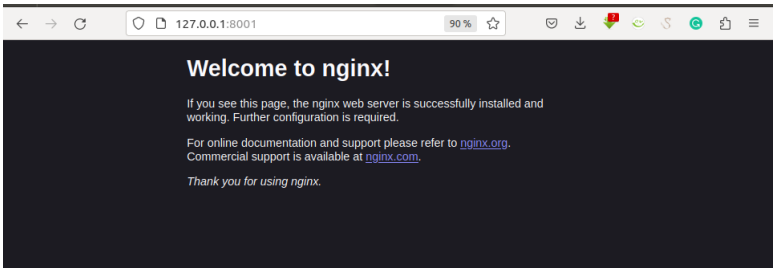
Pour avoir plus de détails sur un volume vous utiliser la commande inspect comme ceci.

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$ docker volume inspect monvolume
[
    {
        "CreatedAt": "2023-01-25T10:07:02Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/monvolume/_data",
        "Name": "monvolume",
        "Options": {},
        "Scope": "local"
    }
]
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$
```

Maintenant nous allons utiliser notre volume sur un conteneur en utilisant l'option –**mount**

```
docker run -d --name web -p 8001:80 --mount source=monvolume,target=/usr/share/nginx/htm
```

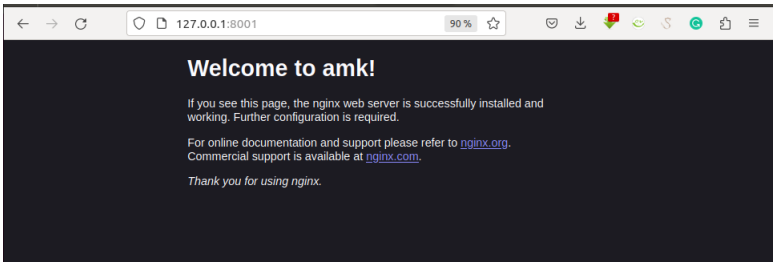
Voici le résultat :



Vous pouvez modifier le contenue du fichier index.php comme ceci :

`sudo vim /var/lib/docker/volumes/monvolume/_data/index.html`

Voici le résultat :



Maintenant que vous êtes maître amusé vous avec la commande **docker volume**.

6 Dockerfile : Créer une image.

Vous savez maintenant utiliser l'interface de commande de Docker et récupérer des images depuis le Docker Hub. Mais comment créer votre propre image ?
Dans cette partie, nous allons créer ensemble une image Docker, dans laquelle nous allons installer nginx.
Pour cela, nous allons créer un fichier nommé **"Dockerfile"** (fichier de configuration). Dans ce fichier Dockerfile, vous allez trouver l'ensemble de la recette décrivant l'image Docker dont vous avez besoin pour votre projet.
Intérêts de **Dockerfile**. *À titre de comparaison, vous pouvez voir le Dockerfile comme l'équivalent d'un fichier package.json en Node.js, ou composer.json en PHP.*
Chaque instruction que nous allons donner dans notre Dockerfile va créer une nouvelle layer correspondant à chaque étape de la construction de l'image. Notre est de limiter le nombre e layers, pour que image soit la plus légère et performante possible.

Voici quelles que closes d'un **Dockerfile** :

- **RUN** : lancement de commande (apt...).
- **ENV** : variable d'environnement.
- **EXPOSE** : exposition de port.
- **VOLUME** : définition de volumes
- **COPY** : cp entre host et conteneur
- **ENTRYPOINT** : processus maître.
- **...** : ...

Voici le contenue de notre Dockerfile.

```
FROM nginx:latest
WORKDIR /usr/share/nginx/html/
RUN rm index.html
COPY . .
```

Dans cette image on :

— part de la dernière version du serveur nginx,

— définit le répertoire de travail,

— supprime le fichier index.html

— copie de notre fichier index.html.

NB :

Il est capitale de nommé votre fichier **"Dockerfile"** tel quel. Il est très important de toujours **partir d'une base** (FROM ...). un fichier Dockerfile ne peut qu'avoir qu'une base.

Maintenant que vous savez pourquoi et comment créer un dockerfile il nous reste plus qu'a l'exécuté. Pour lancer une image on utilise la commende suivante.

docker build -t site_html :v1.0 .

Cette commande signifie que nous souhaitons créer une image dont le nom vaut *site_html* et la version *v1.0* en se servant du Dockerfile se trouvant dans notre répertoire courant (si vous ne spécifiez pas de version elle sera à latest pour dernière version). si vous faite *docker images* vous pouvez comme moi voir votre image .

: : : : : :img

On peut aussi faire *docker history site_ html :v1.0* pour voir en détails les étapes de la création de notre image.

On peut donc créer un conteneur à partir de notre image en utilisant la commande suivante : **docker run -tid –name website site_html :v1.0**

7 Les Layers

Les layers ou couches en français intervienne dans la création d'une image et d'un conteneur. Avec docker on peut distinguer deux types de couches :

— ceux en lecture seul (images).

— et ceux en lecture-écriture (conteneurs).

les images comme les conteneurs peuvent se partager au moins une couche. : : : : :Expérience des couches xavki + : : : : :Expérience de couche partagé.

Les couche peuvent se partage entre les images

Les images en lecture unique sont :

Si on passe en mode écriture, nous avons les couches en lecture, les couches sont amplilé et la dernier couche est appelé **couche de travail**. Quand on passe en mode conteneur, ce sont ces couches qui sont en état d'écriture et lecture.

Docker history <mon_image> permet de visualisé le contenu de l'image. Le résultat nous montre les différente couche de cette image. À l'intérieur il y a des couches ceux en lecture et ceux en lecteur et écriture. Les layers nous permettent la possibilité d'utiliser un fonctionnalité de cache.

Exemple :

```
FROM docker.io/debian:bullseye-slim
RUN apt upadate -qq
RUN apt install -qq -y wget
RUN apt clean
RUN rm -rf /var/lib/apt/lists/*
RUN wget http://xcal1.vodafone.co.uk/10MB.zip
RUN r -f 10MB.zip
```

chaque ligne de ce code représente une couche.

docker diff <mon_conteneur> nous fait ressortir tous les dossiers qui interagisse avec le conteneur.

7.1 Le cache docker

But du cache :
— construire plus vite les images
— démarrer plus vite les conteneurs
— stocker des images légères
— partage de couche/cache
Démonstration :

7.1.1 Lancer deux fois de suite une image (using cache)

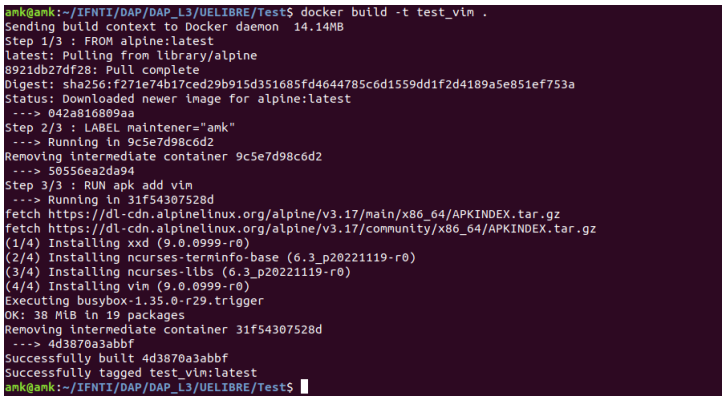
Ici nous allons essayer de lancer deux fois de suite une même image. Voici le contenue su docker file contenue dans cette image.

```
FROM alpine:latest

LABEL maintener="amk"

RUN apk add vim
```

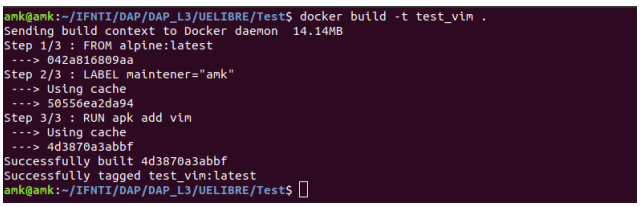
lorsqu'on lance pour la première fois cette image avec la commande :

```
docker build -t test_vim .
```

On peut remarqué qu'il n'a pas utilisation du cache.

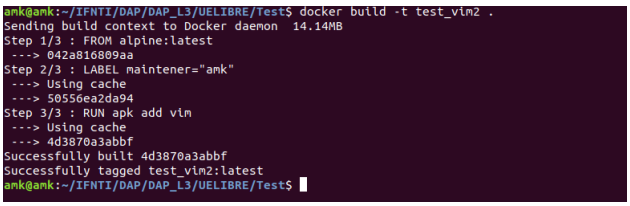


Par contre si on relance une deuxième fois l'image là il y-a utilisation du cache.



L'utilisation du cache se fait même si on change le nom de l'image comme ceci :

```
docker build -t test_vim2
```



Conclusion : Une couche n'est télécharger que si elle n'existe pas.

7.1.2 Ne pas cacher (–no-cache)

On peut vouloir ne pas utiliser le cache lors du build d'une image. Dans ce cas notre image sera comme ceci.

```
FROM alpine:latest

LABEL maintener="amk"

RUN apk add --no-cache vim
```

Donc on peut remarquer l'utilisation du cache au niveau de toute les couches sauf à la couche 3 c'est normale car nous n'avons spécifier l'utilisation du cache. On peut aussi utiliser le –no-cache comme ceci :

```
docker build --no-cache -t test_vim2
```

Mais là c'est l'intégralité du Dockerfile sur le quelle on n'appliquera pas de cache.

NB : l'ordre des éléments comptes. Il est important de mettre des instruction caché en haut et ceux non caché tous en bas. Exemple : Lancer successivement les Dockerfile suivant.
Dockerfile 1

```
FROM alpine:latest

LABEL maintener="amk"

ENV myvariable toto

RUN apk add  vim

docker build -t test

docker build -t test1
```

Dockerfile 2

```
FROM alpine:latest

LABEL maintener="amk"

ENV myvariable titi

RUN apk add  vim

docker build -t test
```

Dockerfile 3

```
FROM alpine:latest

LABEL maintener="amk"

RUN apk add  vim

ENV myvariable toto

docker build -t test
```

8 Les Réseaux

Le principale réseaux c'est le **Bridge** qu'on appelle le **Docker 0**. Il à une adresse par défaut qui est le **172.17.0.0/16**. Il permet la communication entre différent conteneurs et couvre la plus part des besoins en matière de réseau. Mais attention dans la manipulation des réseaux avec docker il est recommandé d'utiliser les nom des conteneur car les adresse ip ne sont pas fixe.

8.1 ping sur le docker 0

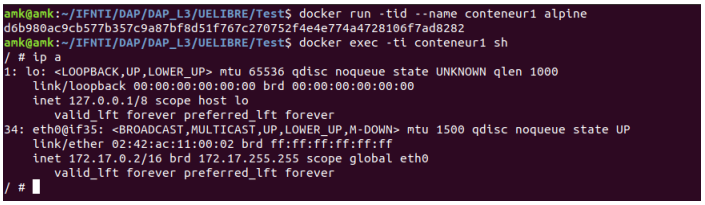
Avant de lancer un ping sur le docker 0 nous allons devoir d'abord créer un conteneur comme ceci :

```
docker run -tid --name conteneur1 alpine
```

Ensuit on rentre dans le conteneur en utilisant la commande suivante :

```
docker exec -ti conteneur1 sh
```

Une fois dans le conteneur on peut consulter son adresse ip avec la commande **ip a**.



On peut remarqué que notre adresse ip commence à deux (2) au lieu de commencer à un (1). Pourquoi ? car c'est le docker 0 qui est la première machine dans le **bridge**. On peut lancer un ping vers ce docker 0 et ça marche car le conteneur est dans le même réseau que le docker 0.

```
/ #
/ # ping 172.17.0.1
PING 172.17.0.1 (172.17.0.1): 56 data bytes
64 bytes from 172.17.0.1: seq=0 ttl=64 time=0.289 ms
64 bytes from 172.17.0.1: seq=1 ttl=64 time=0.204 ms
64 bytes from 172.17.0.1: seq=2 ttl=64 time=0.218 ms
64 bytes from 172.17.0.1: seq=3 ttl=64 time=0.178 ms
64 bytes from 172.17.0.1: seq=4 ttl=64 time=0.202 ms
64 bytes from 172.17.0.1: seq=5 ttl=64 time=0.272 ms
64 bytes from 172.17.0.1: seq=6 ttl=64 time=0.143 ms
64 bytes from 172.17.0.1: seq=7 ttl=64 time=0.160 ms
64 bytes from 172.17.0.1: seq=8 ttl=64 time=0.130 ms
64 bytes from 172.17.0.1: seq=9 ttl=64 time=0.178 ms
^C
--- 172.17.0.1 ping statistics ---
10 packets transmitted, 10 packets received, 0% packet loss
round-trip min/avg/max = 0.130/0.197/0.289 ms
/ #
```

8.2 Personnalisation du bridge –net

Docker nous donne la possibilité de créer nos propres réseaux. Avant de créer notre premier réseau on va déjà regardé quelle sont les réseaux existant pour le moment. Pour le faire lancé la commande : **docker network ls** vous devriez avoir quelle que chose de similaire sans le **domalik_default**.

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
6807044f0414   bridge            bridge    local
cbc210a9327a   domalik_default   bridge    local
f8d6e19e0842   host              host      local
ce5e4a284727   none              null      local
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

Vous pouvez remarqué que les types de réseaux sont le **bridge**, le **host** et le **none**. Nous verrons ici comment créer des réseaux bridge. Voici un exemple de réseau(de type bridge) dont le non est **mon_reseau**et le sous réseau est **172.30.0.0/16** on utilise la commande suivante :

```
docker network create -d bridge --subnet 172.30.0.0/16 mon_reseau
```

Si vous fait un **docker network ls** vous verrez que le **mon_reseau** à bien été créé. En voici la preuve :

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker network create -d bridge --subnet 172.30.0.0/16 mon_reseau
85e9312f5ef801c4edc89bb98abf4658088c5d3beda56bce8743450fdfdf3add
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker network ls
NETWORK ID     NAME              DRIVER    SCOPE
6807044f0414   bridge            bridge    local
cbc210a9327a   domalik_default   bridge    local
f8d6e19e0842   host              host      local
85e9312f5ef8   mon_reseau        bridge    local
ce5e4a284727   none              null      local
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

Pour faire un petit test de réseau on peut essayé de créer trois conteneur :

- deux première conteneur (conteneur1 et conteneur2) qui seront dans le même réseau et
- un troisième conteneur qui sera dans le bridge par défaut.

Nous essayerons de lancer des ping de la manière suivante :

```
conteneur1   ------> conteneur2
conteneur2   ------> conteneur1
conteneur3   ------> conteneur1
```

Sans plus tarder, créons les trois conteneurs :

```
docker run -itd --name conteneur1 --network mon_reseau alpine
docker run -itd --name conteneur2 --network mon_reseau alpine
docker run -itd --name conteneur3 alpine
```

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -it conteneur1 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> 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
43: eth0@if44: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:1e:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.30.0.2/16 brd 172.30.255.255 scope global eth0
       valid_lft forever preferred_lft forever
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -it conteneur2 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> 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
45: eth0@if46: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:1e:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.30.0.3/16 brd 172.30.255.255 scope global eth0
       valid_lft forever preferred_lft forever
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -it conteneur3 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> 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
47: eth0@if48: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

8.2.1 ping conteneur1 vs conteneur2

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -it conteneur1 ping 172.30.0.3
PING 172.30.0.3 (172.30.0.3): 56 data bytes
64 bytes from 172.30.0.3: seq=0 ttl=64 time=0.161 ms
64 bytes from 172.30.0.3: seq=1 ttl=64 time=0.179 ms
64 bytes from 172.30.0.3: seq=2 ttl=64 time=0.208 ms
64 bytes from 172.30.0.3: seq=3 ttl=64 time=0.246 ms
64 bytes from 172.30.0.3: seq=4 ttl=64 time=0.190 ms
64 bytes from 172.30.0.3: seq=5 ttl=64 time=0.163 ms
64 bytes from 172.30.0.3: seq=6 ttl=64 time=0.180 ms
^C
--- 172.30.0.3 ping statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max = 0.161/0.189/0.246 ms
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

Le ping marche car le conteneur1 et le conteneur2 sont dans le même sous réseaux.

8.2.2 ping conteneur2 vs conteneur1

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -it conteneur2 ping 172.30.0.2
PING 172.30.0.2 (172.30.0.2): 56 data bytes
64 bytes from 172.30.0.2: seq=0 ttl=64 time=0.159 ms
64 bytes from 172.30.0.2: seq=1 ttl=64 time=0.178 ms
64 bytes from 172.30.0.2: seq=2 ttl=64 time=0.226 ms
64 bytes from 172.30.0.2: seq=3 ttl=64 time=0.218 ms
64 bytes from 172.30.0.2: seq=4 ttl=64 time=0.206 ms
64 bytes from 172.30.0.2: seq=5 ttl=64 time=0.183 ms
64 bytes from 172.30.0.2: seq=6 ttl=64 time=0.208 ms
^C
--- 172.30.0.2 ping statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max = 0.159/0.196/0.226 ms
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

Le ping marche car le conteneur1 et le conteneur2 sont dans le même sous réseaux.

8.2.3 ping conteneur3 vs conteneur1

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -it conteneur3 ping 172.30.0.2
PING 172.30.0.2 (172.30.0.2): 56 data bytes
```

Le ping ne marche pas car le conteneur1 et le conteneur3 ne sont pas dans le même sous réseaux.

8.3 Autre cas –net

— **–net : none** Pour spécifier que le conteneur n'a aucune ouverture réseau (cas particulier).
— **–net : host** Son ouverture réseau ne correspond qu'a l'accès au host.
— **–net : container :<nomconteneur>** Ouverture réseau à un conteneur particulier.

8.4 Autre cas –link

–link : container :<nomconteneur> Ouverture réseau à un conteneur particulier. Mais vas aller compléter le **/etc/hosts** du conteneur.
Pour tester ça : on vas créer deux conteneurs. Le conteneur1 et le conteneur2 qui utilise le réseau du conteneur2

```
docker run -itd --name conteneur1 alpine
docker run -itd --name conteneur2 --link conteneur1 alpine
```

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -ti conteneur2 sh
/ # cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      conteneur1 408f68841408
172.17.0.3      da04f7c27a33
/ #
```

8.5 Et en plus ...

— **–add-host <nomhost> :ip** : complète le /etc/hosts.
— **–dns** : ajoute les ip de serveurs dns.
Test : ici nous avons créer le conteneur1 en précisant l'option –add-host pour cela demander à un de vos camarade de vous donner son adresse ip ou si vous aviez une deuxième machine prenez son adresse ip. Dans mon cas l'adresse ip de mon camarade est le *192.168.60.180*(Il est important que votre host soit aussi dans le sous réseau).

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker run -itd --name conteneur1 --add-host myhost:192.168.60.1
80 alpine
b99869a95db55d78a5033f8c216d34c4eb33dc2f7477fd36661057da1b621c46
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -it conteneur1 ping 192.168.60.180
PING 192.168.60.180 (192.168.60.180): 56 data bytes
64 bytes from 192.168.60.180: seq=0 ttl=63 time=0.405 ms
64 bytes from 192.168.60.180: seq=1 ttl=63 time=0.343 ms
64 bytes from 192.168.60.180: seq=2 ttl=63 time=0.626 ms
64 bytes from 192.168.60.180: seq=3 ttl=63 time=0.618 ms
64 bytes from 192.168.60.180: seq=4 ttl=63 time=0.581 ms
^C
--- 192.168.60.180 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.343/0.514/0.626 ms
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -it conteneur1 cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
192.168.60.180  myhost
172.17.0.2      b99869a95db5
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

9 Docker compose