

# DOCUMENTATION DOCKER **Docker**

27 février 2023

## 1 Généralités

Dans cette première partie, nous allons découvrir les **conteneurs** ainsi que Docker. Mais avant cela, nous allons revenir sur quelques notions importantes :

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

### 1.1 Machine virtuelle (VM)

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

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

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 ressources 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**.

### 1.2 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 quelques avantages des conteneurs.

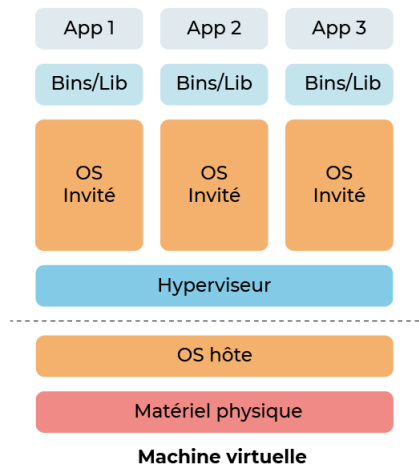


FIGURE 1 – Fonctionnement d'une machine virtuelle

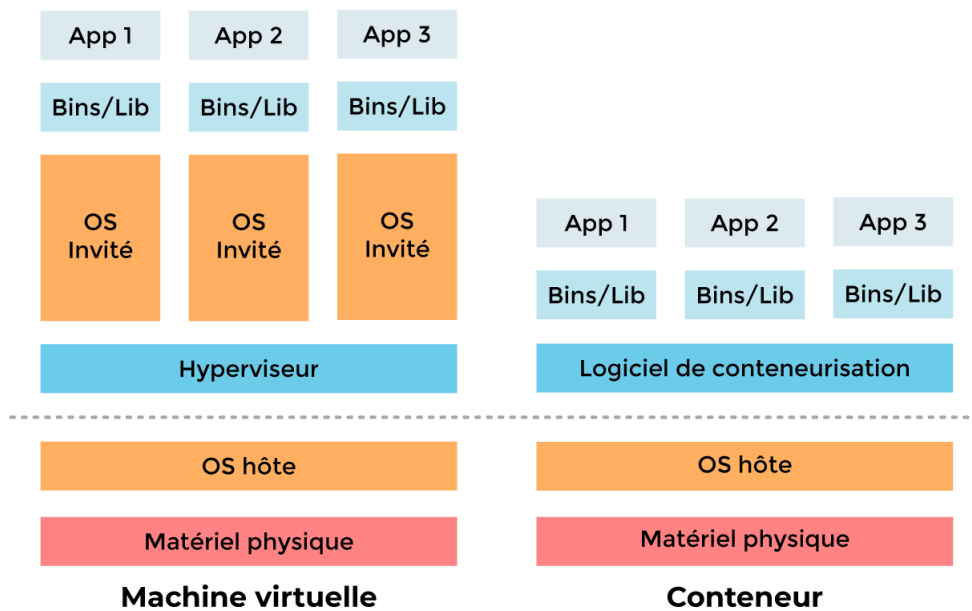


FIGURE 2 – Fonctionnement d'une machine virtuelle

- **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 des processus, ils peuvent **démarrer beaucoup plus rapidement** et plus fréquemment qu'une machine virtuelle sur nos serveurs 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.

### 1.3 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.

### 1.4 Exemple d'équation entre Administrateur système et Développeurs

L'une des équation possible entre administrateur système (admin) et développeurs (dev) pourrait être la suivante :

Administrateur système = Garant de la stabilité de la sécurité des systèmes informatique  
Développeurs = Créateurs de nouvelles fonctionnalité et applications

- **Administrateur système** est responsable de maintenir la stabilité et la sécurité des systèmes informatique en s'assurant que tous les logiciels et configurations fonctionnent correctement et que les données soit protégées contre les menaces de sécurité.
- Les **Développeurs**, quant à eux, sont responsable de créer de nouvelles fonctionnalités et applications pour répondre aux besoins de l'entreprise et des utilisateur.

Ces deux rôles sont complémentaires et interdépendants. Les développeurs ont besoin d'un environnement de développement stable et sécurisé pour travailler efficacement, tandis que l'administrateur système a besoin de comprendre les besoins des développeurs pour configurer les systèmes en conséquence. Ensemble, ils travaillent pour assurer le bon fonctionnement des systèmes informatiques de l'entreprise.

### 1.5 Docker préambule

Docker est une plateforme de conteneurisation qui permet d'emballer et d'exécuter des applications dans des conteneurs isolés.

### 1.5.1 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**.

### 1.5.2 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.

### 1.5.3 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 !

## 1.6 Comprendre la terminologie de Docker avant de se lancer !

Avant de se lancer dans l'utilisation de Docker, il est important de comprendre la terminologie suivante :

- Image : une image Docker est un paquet de base qui contient tout le nécessaire pour exécuter une application, y compris le code source, les dépendances, les bibliothèques et les fichiers de configuration.
- Conteneur : un conteneur Docker est une instance d'une image qui s'exécute de manière isolée dans un environnement de conteneur. Un conteneur peut être considéré comme une boîte virtuelle qui contient tout ce dont l'application a besoin pour s'exécuter.
- Dockerfile : un Dockerfile est un fichier de configuration qui permet de créer une image Docker personnalisée. Il contient les instructions nécessaires pour créer une image Docker à partir d'une base existante.
- Docker Compose : Docker Compose est un outil qui permet de définir et de gérer des applications multi-conteneurs. Il permet de définir plusieurs conteneurs et leurs dépendances dans un fichier de configuration unique.
- Registre : un registre Docker est un référentiel centralisé qui permet de stocker et de distribuer des images Docker. Docker Hub est le registre public officiel de Docker.

En comprenant ces termes de base, vous serez mieux préparé pour travailler avec Docker et comprendre comment créer, déployer et gérer des conteneurs Docker

## 2 Docker

### 2.1 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 apt-get install docker.io
install docker-ce == moteur
docker-ce-cli == pour communiquer avec le moteur
containerd.io == gérer l'exécution de conteneurs.
docker-buildx-plugin == pour builder nos images
docker-compose-plugin
```

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

## 3 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**.

### 3.1 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**.

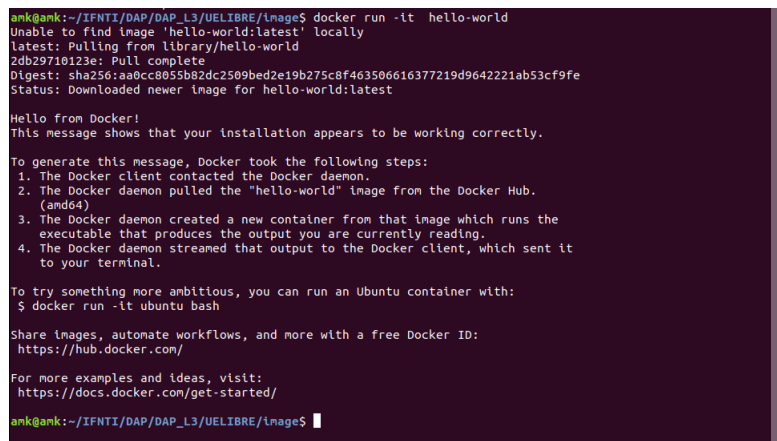
Une **registry** est un logiciel qui permet de partager des images à d'autres personnes. C'est un composant majeur dans l'écosystème Docker, car il permet :

- 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.

### 3.2 Démarrez votre premier conteneur Docker

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

```
docker run hello-world
```



```
ank@ank:~/IFNTI/DAP/DAP_L3/Uelibre/lnage$ docker run -it hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:aa0cc8055b82dc2509bed2e19b275c8f463506616377219d9642221ab53cf9fe
Status: Downloaded newer image for hello-world:latest

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

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (and4)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

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

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

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

ank@ank:~/IFNTI/DAP/DAP_L3/Uelibre/lnage$
```

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 `-d` (*-detach*). 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`.

### 3.3 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_DU_DOCKER_RUN 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`.

### 3.4 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.

### 3.5 Quelques commandes

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

```
amk@amk:~$ docker images
REPOSITORY          TAG          IMAGE ID      CREATED        SIZE
nginx               latest       a99a39d070bf 12 days ago   142MB
alpine              latest       042a816809aa 2 weeks ago   7.05MB
ubuntu              latest       6b7dfa7e8fdb 6 weeks ago   77.8MB
postgres            12           31c3beb3b896 6 weeks ago   373MB
bfirsh/reticulate-splines latest       b1666055931f 6 years ago   4.8MB
amk@amk:~$ docker image ls
REPOSITORY          TAG          IMAGE ID      CREATED        SIZE
nginx               latest       a99a39d070bf 12 days ago   142MB
alpine              latest       042a816809aa 2 weeks ago   7.05MB
ubuntu              latest       6b7dfa7e8fdb 6 weeks ago   77.8MB
postgres            12           31c3beb3b896 6 weeks ago   373MB
bfirsh/reticulate-splines latest       b1666055931f 6 years ago   4.8MB
amk@amk:~$
```

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

— **SIZE** L'espace mémoire qu'elle 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.

## 3.6 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 manuellement l'ensemble des ressources 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.

## 4 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 volumes persistant.

### 4.1 Les volumes persistant

Vous vous souvenez du serveur nginx que nous avons 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 seueur -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 a aucun éditeur installé par défaut.

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -it serveur bash
root@9e32c83133a0:/# cd /usr/share/nginx/html/
root@9e32c83133a0:/usr/share/nginx/html# ls
50x.html  index.html
root@9e32c83133a0:/usr/share/nginx/html# echo "<h1>Welcome to AMK web site :) ls</h1>" > index.html
root@9e32c83133a0:/usr/share/nginx/html# cat index.html
<h1>Welcome to AMK web site :) ls</h1>
root@9e32c83133a0:/usr/share/nginx/html#
```

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 modifications 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 **/usr/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/public_html/:/usr/share/nginx/html/ nginx
```

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$ mkdir public_html
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$ echo "<h1>Welcome to ank web site :) ls</h1>" > public_html/index.html
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$ docker run -d --name serveur -p 8001:80 -v /home/amk/IFNTI/DAP/DAP_L3/UELIBRE/Test/test/public_html/:/usr/share/nginx/html/ nginx
69cfff651d630c1526cdf5122a0132e283b6f368becc25e271bbb094c395debb
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test/test$ tree
.
├── public_html
│   └── index.html
└── 1 directory, 1 file
amk@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.

## 4.2 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'à taper **docker volume** dans votre terminal vous verrez ça.

```
amk@amk:~/IFNTII/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:~/IFNTII/DAP/DAP_L3/UELIBRE/Test/test$
```

Nous allons créer un volume nommé monvolume voici la commande.

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

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

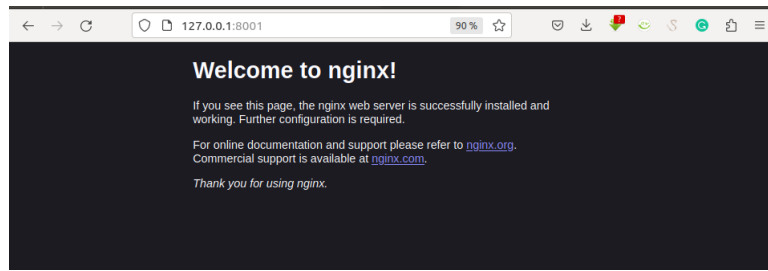
```
amk@amk:~/IFNTII/DAP/DAP_L3/UELIBRE/Test/test$ docker volume inspect monvolume
[
  {
    "CreatedAt": "2023-01-25T10:07:02Z",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/monvolume/_data",
    "Name": "monvolume",
    "Options": {},
    "Scope": "local"
  }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

```
amk@amk:~/IFNTII/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/html nginx
```

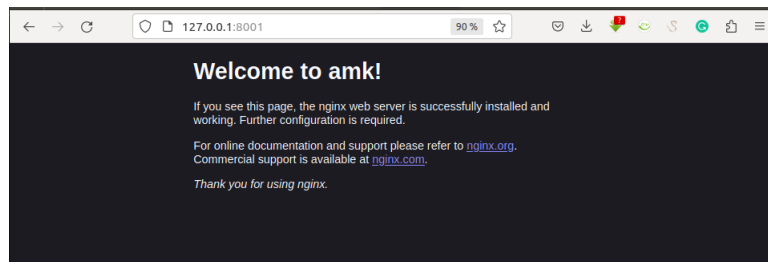
Voici le résultat :



Vous pouvez modifier le contenu 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**.

## 5 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 but est de limiter le nombre de layers, pour que l'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 contenu 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,
- on définit le répertoire de travail,
- on supprime le fichier index.html
- on copie notre fichier index.html.

**NB :**

Il est capitale de nommer 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'à l'exécuter. Pour lancer une image on utilise la commande 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 -d -name website site\_html :v1.0**

## 6 Les Layers

Les layers ou couches en français interviennent 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.

### 6.1 Le cache docker

But du cache :

- construire plus vite les images
- démarrer plus vite les conteneurs
- stocker des images légères
- partager des couches/layers

Démonstration :

#### 6.1.1 Lancer deux fois de suite une image (using cache)

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

```
FROM alpine:latest
```

```
LABEL maintainer="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.

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker build -t test_vim .
Sending build context to Docker daemon 14.14MB
Step 1/3 : FROM alpine:latest
latest: Pulling from library/alpine
8921db27df28: Pull complete
Digest: sha256:f271e74b17ced29b915d351685fd4644785c6d1559dd1f2d4189a5e851ef753a
Status: Downloaded newer image for alpine:latest
--> 042a816809aa
Step 2/3 : LABEL maintainer="amk"
--> Running in 9c5e7d98c6d2
Removing intermediate container 9c5e7d98c6d2
--> 50556ea2da94
Step 3/3 : RUN apk add vim
--> Running in 31f54307528d
fetch https://dl-cdn.alpinelinux.org/alpine/v3.17/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.17/community/x86_64/APKINDEX.tar.gz
(1/4) Installing xxd (9.0.0999-r0)
(2/4) Installing ncurses-terminfo-base (6.3_p20221119-r0)
(3/4) Installing ncurses-libs (6.3_p20221119-r0)
(4/4) Installing vim (9.0.0999-r0)
Executing busybox-1.35.0-r29.trigger
OK: 38 MiB in 19 packages
Removing intermediate container 31f54307528d
--> 4d3870a3abbf
Successfully built 4d3870a3abbf
Successfully tagged test_vim:latest
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

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

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker build -t test_vim .
Sending build context to Docker daemon 14.14MB
Step 1/3 : FROM alpine:latest
--> 042a816809aa
Step 2/3 : LABEL maintainer="amk"
--> Using cache
--> 50556ea2da94
Step 3/3 : RUN apk add vim
--> Using cache
--> 4d3870a3abbf
Successfully built 4d3870a3abbf
Successfully tagged test_vim:latest
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

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

```
docker build -t test_vim2
```

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker build -t test_vim2 .
Sending build context to Docker daemon 14.14MB
Step 1/3 : FROM alpine:latest
--> 042a816809aa
Step 2/3 : LABEL maintainer="amk"
--> Using cache
--> 50556ea2da94
Step 3/3 : RUN apk add vim
--> Using cache
--> 4d3870a3abbf
Successfully built 4d3870a3abbf
Successfully tagged test_vim2:latest
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

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

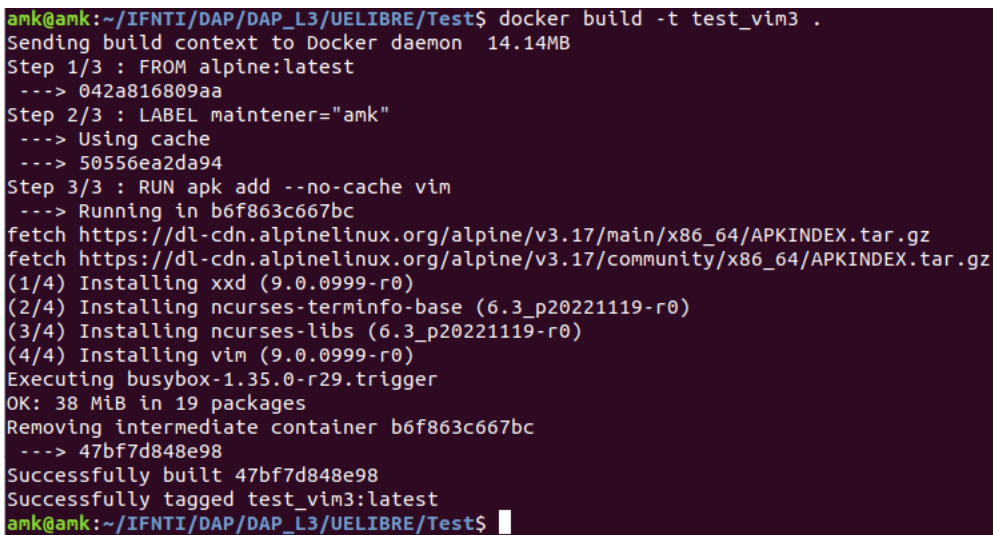
### 6.1.2 Ne pas cacher (`--no-cache`)

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
```

A terminal window showing the output of a Docker build command. The user is at a prompt 'amk@amk:~/IFNTI/DAP/DAP\_L3/UELIBRE/Test\$' and runs 'docker build -t test\_vim3 .'. The output shows the build context being sent to the Docker daemon (14.14MB), followed by three steps: 1/3 FROM alpine:latest (hash 042a816809aa), 2/3 LABEL maintener="amk" (hash 50556ea2da94), and 3/3 RUN apk add --no-cache vim (hash b6f863c667bc). The RUN step shows the installation of vim and its dependencies (xxd, ncurses-terminfo-base, ncurses-libs) from the alpine repository. The build is successful, and the image is tagged 'test\_vim3:latest'.

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker build -t test_vim3 .
Sending build context to Docker daemon 14.14MB
Step 1/3 : FROM alpine:latest
--> 042a816809aa
Step 2/3 : LABEL maintener="amk"
--> Using cache
--> 50556ea2da94
Step 3/3 : RUN apk add --no-cache vim
--> Running in b6f863c667bc
fetch https://dl-cdn.alpinelinux.org/alpine/v3.17/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.17/community/x86_64/APKINDEX.tar.gz
(1/4) Installing xxd (9.0.0999-r0)
(2/4) Installing ncurses-terminfo-base (6.3_p20221119-r0)
(3/4) Installing ncurses-libs (6.3_p20221119-r0)
(4/4) Installing vim (9.0.0999-r0)
Executing busybox-1.35.0-r29.trigger
OK: 38 MiB in 19 packages
Removing intermediate container b6f863c667bc
--> 47bf7d848e98
Successfully built 47bf7d848e98
Successfully tagged test_vim3:latest
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$
```

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 compte. Il est important de mettre des instructions 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
```

## 7 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.

### 7.1 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**.

```

amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker run -tid --name conteneur1 alpine
d6b980ac9cb577b357c9a87bf8d51f767c270752f4e4e774a4728106f7ad8282
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker exec -ti conteneur1 sh
/ # 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
34: eth0@if35: <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
/ #

```

On peut remarquer que notre adresse ip commence à deux (2) au lieu de commencer à un (1). Pourquoi ? car c'est le docker 0 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
/ #
```

## 7.2 Personnalisation du bridge –net

Docker nous donne la possibilité de créer nos propres réseaux. Avant de créer notre premier réseau on va déjà regarder 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 remarquer 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 nom 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$
```

### 7.2.1 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.

### 7.2.2 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.

### 7.2.3 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.

## 7.3 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'à l'accès au host.
- `--net : container :<nomconteneur>` Ouverture réseau à un conteneur particulier.

## 7.4 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
/ #
```

## 7.5 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 même sous réseau).

```
amk@amk:~/IFNTI/DAP/DAP_L3/UELIBRE/Test$ docker run -itd --name conteneur1 --add-host myhost:192.168.60.180 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$
```

## 8 Docker compose

Docker compose est un orchestrateur de conteneur. Il permet de gérer plusieurs conteneur à la fois. Ainsi il permet de simplifier nos déploiements sur de multiple environnement. Docker compose est un outil écrit en python qui permet de décrire, dans un fichier **YAML**, plusieurs conteneurs comme un ensemble de service.

### 8.1 Installation de docker compose.

Normalement si vous avez installer docker avec notre ligne de commande, docker compose serait déjà installé si ce n'est pas le cas alors exécuter la commande suivante :

```
sudo curl -L "https://github.com/docker/compose/
releases/download/1.23.2/docker-compose-
$(uname -s)-$(uname -m)" -o /usr/bin/docker-compose &&
sudo chmod +x /usr/bin/docker-compose
```

Puis :

```
docker-compose --version
```

### 8.2 Le CLI de Docker Compose

Pour utiliser le **CLI (Command Line Interface)** de Docker Compose, nous avons besoin d'un fichier **docker-compose.yml** que nous allons créer dans l'exemple ci dessous ! Mais avant de le créer, nous allons commencer par découvrir ensemble l'interface en ligne de commande (CLI) qui nous permet d'utiliser le fichier docker-compose.yml.

**Le CLI de Docker Compose et celui de Docker sont très proches.** Par exemple, si vous souhaitez récupérer l'ensemble des images décrites dans votre fichier `docker-compose.yml` et les télécharger depuis le Docker Hub, vous devez faire un `docker-compose pull`. Du côté de Docker, la commande serait un `docker pull`.

*Le fait que les deux interfaces en ligne de commande soient très similaires nous évite d'apprendre un nouveau CLI. Si vous connaissez celui de Docker, vous savez globalement utiliser celui de Docker Compose !*

Cependant, nous allons quand même voir ensemble les principales commandes que vous pourriez avoir besoin d'utiliser et de connaître pour votre utilisation de Docker Compose.

### 8.3 Démarrer une stack Docker Compose

Si vous souhaitez lancer la création de l'ensemble des conteneurs, vous devez lancer la commande **`docker-compose up`** (pour rappel, vous faites un **`docker run`** pour lancer un seul conteneur). Vous pouvez ajouter l'argument **`-d`** pour faire tourner les conteneurs en tâche de fond.

Nous appelons **stack** un ensemble de conteneurs Docker lancés via un seul et unique fichier Docker Compose.

### 8.4 Voir le statut d'une stack Docker Compose

Après avoir démarré une **stack** Docker Compose, vous aurez certainement besoin de voir si l'ensemble des conteneurs sont bien dans un état fonctionnel, et prêts à rendre un service. Pour cela, vous allez utiliser la commande **`docker-compose ps`** qui vous affichera le retour suivant :

NAME	COMMAND	SERVICE	STATUS	PORTS
ou				
no configuration file provided: not found				

### 8.5 Voir les logs d'une stack Docker Compose

Votre stack Docker Compose est maintenant fonctionnelle, et l'ensemble des services répondent bien ; mais vous pourriez avoir besoin de voir les logs de vos conteneurs. Pour cela, vous devez utiliser la commande **`docker-compose logs -f -tail 5`**.

Celle-ci permet de voir l'ensemble des logs sur les différents conteneurs de façon continue, tout en limitant l'affichage aux 5 premières lignes.

Ainsi, si nos conteneurs fonctionnent depuis longtemps, nous n'aurons pas à attendre plusieurs secondes, ni à voir de nombreux logs qui ne nous intéressent pas.

### 8.6 Arrêter une stack Docker Compose

Si vous souhaitez arrêter une stack Docker Compose, vous devez utiliser la commande **`docker-compose stop`**. Cependant, celle-ci ne supprimera pas les différentes ressources créées

par votre stack.

Ainsi, si vous lancez à nouveau un **docker-compose up -d**, l'ensemble de votre stack sera tout de suite à nouveau fonctionnel.

Si vous souhaitez supprimer l'ensemble de la stack Docker Compose, vous devez utiliser la commande **docker-compose down** qui détruira l'ensemble des ressources créées.

## 8.7 Valider une stack Docker Compose

Lors de l'écriture d'un fichier docker-compose, nous ne sommes pas à l'abri d'une erreur. Pour éviter au maximum cela, vous devez utiliser la commande **docker-compose config** qui vous permettra de valider la syntaxe de votre fichier, et ainsi d'être certain de son bon fonctionnement.

Si nous créons une erreur dans notre stack, en remplaçant "image" par "images", par exemple, nous aurons le résultat suivant :

```
$ docker-compose config
```

```
ERROR: The Compose file './docker-compose.yml' is invalid because:
```

```
Unsupported config option for services.db: 'images'
```

## 8.8 En résumé

Vous connaissez maintenant les commandes principales pour utiliser une **stack Docker Compose**. Voici les commandes les plus importantes :

- **docker-compose up -d** vous permettra de démarrer l'ensemble des conteneurs en arrière-plan ;
- **docker-compose ps** vous permettra de voir le statut de l'ensemble de votre stack ;
- **docker-compose logs -f --tail 5** vous permettra d'afficher les logs de votre stack ;
- **docker-compose stop** vous permettra d'arrêter l'ensemble des services d'une stack ;
- **docker-compose down** vous permettra de détruire l'ensemble des ressources d'une stack ;
- **docker-compose config** vous permettra de valider la syntaxe de votre fichier docker-compose.yml.

## 9 Créez un fichier docker-compose pour orchestrer vos conteneurs

Nous avons vu précédent comment utiliser l'interface en ligne de commande de Docker Compose ; vous allez maintenant apprendre à créer un fichier docker-compose.yml.

### Sujet :

Vous avez un nouveau projet de site avec WordPress, et vous souhaitez simplifier la gestion de l'infrastructure. Vous devez maintenant réaliser un déploiement en production, où l'ensemble des composants sont dans des conteneurs Docker. Pour cela, vous allez avoir besoin de plusieurs composants :

- une base de données MySQL ;

— le système de fichiers WordPress.

Voici le diagramme de déploiement correspondant.

```
:: :: :Image :: :: :: :
```

Vous devez commencer par créer un fichier **docker-compose.yml** à la racine de votre projet. Dans celui-ci, nous allons décrire l'ensemble des ressources et services nécessaires à la réalisation de votre POC (Proof Of Concept).

## 9.1 Décrivez votre premier service : db

### 9.1.1 Définissez la version de Docker Compose

Un fichier `docker-compose.yml` commence toujours par les informations suivantes :

```
version: '3'
```

L'argument `version` permet de spécifier à Docker Compose quelle version on souhaite utiliser, et donc d'utiliser ou pas certaines versions. Dans notre cas, nous utiliserons la version 3, qui est actuellement la version la plus utilisée.

### 9.1.2 Déclarez le premier service et son image

Nous allons maintenant déclarer notre premier service, et donc créer notre **stack WordPress** !

L'ensemble des conteneurs qui doivent être créés doivent être définis sous l'argument **services**. Chaque conteneur commence avec un nom qui lui est propre ; dans notre cas, notre premier conteneur se nommera **db**.

```
services:
```

```
  db:
```

```
    image: mysql:5.7
```

Puis, vous devez **décrire votre conteneur** ; dans notre cas, nous utilisons l'argument **image** qui nous permet de définir l'image Docker que nous souhaitons utiliser.

Nous aurions pu aussi utiliser l'argument **build** en lui spécifiant le chemin vers notre fichier **Dockerfile** ; ainsi, lors de l'exécution de Docker Compose, il aurait construit le conteneur via le Dockerfile avant de l'exécuter.

### 9.1.3 Définissez le volume pour faire persister vos données

```
services:
```

```
  db:
```

```
    image: mysql:5.7
```

```
    volumes:
```

```
      - db_data:/var/lib/mysql
```

Pour rappel, nous avons vu précédemment que les conteneurs Docker ne sont pas faits pour faire fonctionner des services **stateful**, et une base de données est par définition un service **stateful**. Alors vous allez utiliser l'argument `volumes` qui vous permet de stocker l'ensemble du contenu du dossier `/var/lib/mysql` dans un disque persistant. Et donc, de pouvoir garder les données en local sur notre host comme nous l'avons vue dans la section `VOLUME`.

Cette description est présente grâce à la ligne **`db_data :/var/lib/mysql`**. `db_data` est un volume créé par Docker directement, qui permet d'écrire les données sur le disque hôte sans spécifier l'emplacement exact. Vous auriez pu aussi faire un `/data/mysql :/var/lib/mysql` qui serait aussi fonctionnel.

#### 9.1.4 Définissez la politique de redémarrage du conteneur

`services:`

`db:`

`image: mysql:5.7`

`volumes:`

`- db_data:/var/lib/mysql`

`restart: always`

Un conteneur étant par définition **monoprocessus**, s'il rencontre une erreur fatale, il peut être amené à s'arrêter. Dans notre cas, si le serveur MySQL s'arrête, celui-ci redémarrera automatiquement grâce à l'argument **`restart : always`**.

#### 9.1.5 Définissez les variables d'environnement

`services:`

`db:`

`image: mysql:5.7`

`volumes:`

`- db_data:/var/lib/mysql`

`restart: always`

`environment:`

`MYSQL_ROOT_PASSWORD: somewordpress`

`MYSQL_DATABASE: wordpress`

`MYSQL_USER: wordpress`

`MYSQL_PASSWORD: wordpress`

L'image MySQL fournie dispose de plusieurs variables d'environnement que vous pouvez utiliser ; dans notre cas, nous allons donner au conteneur les valeurs des différents mots de passe et utilisateurs qui doivent exister sur cette base. Quand vous souhaitez donner des variables d'environnement à un conteneur, vous devez utiliser l'argument **environment**, comme nous l'avons utilisé dans le fichier `docker-compose.yml` ci-dessus.

## 9.2 Décrivez votre second service : WordPress

Dans le second service, nous créons un conteneur qui contiendra le nécessaire pour faire fonctionner votre site avec WordPress. Cela nous permet d'introduire deux arguments supplémentaires.

```
services:

  wordpress:

    depends_on:

      - db

    image: wordpress:latest

    ports:

      - "8000:80"

    restart: always

    environment:

      WORDPRESS_DB_HOST: db:3306

      WORDPRESS_DB_USER: wordpress

      WORDPRESS_DB_PASSWORD: wordpress

      WORDPRESS_DB_NAME: wordpress
```

Le premier argument, **depends\_on**, nous permet de créer une dépendance entre deux conteneurs. Ainsi, Docker démarrera le service `db` avant de démarrer le service `WordPress`. Ce qui est un comportement souhaitable, car WordPress dépend de la base de données pour fonctionner correctement.

Le second argument, **ports**, permet de dire à Docker Compose qu'on veut exposer un port de notre machine hôte vers notre conteneur, et ainsi le rendre accessible depuis l'extérieur.

Voici le fichier **docker-compose.yml** dans sa version finale :

```
version: '3'

services:

  db:
```



```
image: mysql:5.7

volumes:

  - db_data:/var/lib/mysql

restart: always

environment:

  MYSQL_ROOT_PASSWORD: somewordpress

  MYSQL_DATABASE: wordpress

  MYSQL_USER: wordpress

  MYSQL_PASSWORD: wordpress
```

```
wordpress:

  depends_on:

    - db

  image: wordpress:latest

  ports:

    - "8000:80"

  restart: always

  environment:

    WORDPRESS_DB_HOST: db:3306

    WORDPRESS_DB_USER: wordpress

    WORDPRESS_DB_PASSWORD: wordpress

    WORDPRESS_DB_NAME: wordpress
```

```
volumes:

  db_data: {}
```

### 9.3 Lancez votre stack Docker Compose

Quand vous lancerez vos conteneurs avec la commande **docker-compose up -d** , vous devriez avoir le résultat suivant :

```
$ docker-compose up -d

Creating network "my_wordpress_default" with the default driver

Pulling db (mysql:5.7)...

5.7: Pulling from library/mysql

efd26ecc9548: Pull complete

a3ed95caeb02: Pull complete

...

Digest: sha256:34a0aca88e85f2efa5edff1cea77cf5d3147ad93545dbec99cfe705b03c520de

Status: Downloaded newer image for mysql:5.7

Pulling wordpress (wordpress:latest)...

latest: Pulling from library/wordpress

efd26ecc9548: Already exists

a3ed95caeb02: Pull complete

589a9d9a7c64: Pull complete

...

Digest: sha256:ed28506ae44d5def89075fd5c01456610cd6c64006addfe5210b8c675881aff6

Status: Downloaded newer image for wordpress:latest

Creating my_wordpress_db_1

Creating my_wordpress_wordpress_1
```

Lors de l'exécution de cette commande, Docker Compose commence par vérifier si nous disposons bien en local des images nécessaires au lancement des stacks. Dans le cas contraire, il les télécharge depuis une registry, ou les build via un **docker build** .

Puis celui-ci lance les deux conteneurs sur votre système ; dans notre cas, vous pourrez voir le résultat en vous ouvrant l'URL suivante dans votre navigateur : **http ://127.0.0.1 :8000. : : : :IMAGE : : : : :**

## 9.4 En résumé

Vous savez maintenant utiliser les commandes de base de Docker Compose, et créer un fichier `docker-compose.yml` pour orchestrer vos conteneurs Docker.

Pour rappel, voici les arguments que nous avons pu voir dans ce chapitre :

- **image** qui permet de spécifier l'image source pour le conteneur ;
- **build** qui permet de spécifier le Dockerfile source pour créer l'image du conteneur ;
- **volume** qui permet de spécifier les points de montage entre le système hôte et les conteneurs ;
- **restart** qui permet de définir le comportement du conteneur en cas d'arrêt du processus ;
- **environment** qui permet de définir les variables d'environnement ;
- **depends\_on** qui permet de dire que le conteneur dépend d'un autre conteneur ;
- **ports** qui permet de définir les ports disponibles entre la machine host et le conteneur.