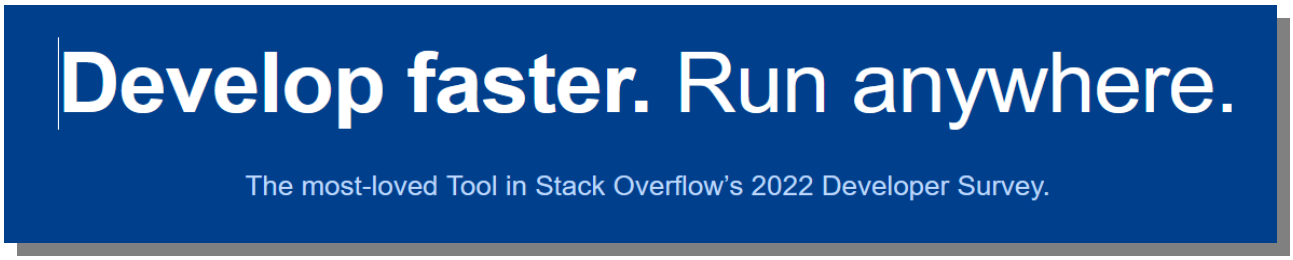
TP Docker - part1

BUT2 – IUT Limoges 2025

R4.A.08 : Virtualisation



Documentation officielle : https://docs.docker.com/

Docker est une plateforme permettant de lancer certaines applications dans des conteneurs logiciels.

Selon la firme de recherche sur l'industrie 451 Research, « Docker est un outil qui peut empaqueter une application et ses dépendances dans un conteneur isolé, qui pourra être exécuté sur n'importe quel serveur ». Il ne s'agit pas de virtualisation, mais de conteneurisation, une forme plus légère qui s'appuie sur certaines parties de la machine hôte pour son fonctionnement. Cette approche permet d'accroître la flexibilité et la portabilité d'exécution d'une application, laquelle va pouvoir tourner de façon fiable et prévisible sur une grande variété de machines hôtes, que ce soit sur la machine locale, un cloud privé ou public, une machine nue, etc.

Techniquement, Docker étend le format de conteneur Linux standard, LXC, avec une API de haut niveau fournissant une solution pratique de virtualisation qui exécute les processus de façon isolée. Pour ce faire, Docker utilise entre autres LXC, cgroups et le noyau Linux lui-même. Contrairement aux machines virtuelles traditionnelles, un conteneur Docker n'inclut pas de système d'exploitation, mais s'appuie au contraire sur les fonctionnalités du système d'exploitation fournies par la machine hôte.

La technologie de conteneur de Docker peut être utilisée pour étendre des systèmes distribués de façon à ce qu'ils s'exécutent de manière autonome depuis une seule machine physique ou une seule instance par nœud. Cela permet aux nœuds d'être déployés au fur et à mesure que les ressources sont disponibles, offrant un déploiement transparent et similaire aux PaaS pour des systèmes comme Apache Cassandra, Riak, ou d'autres systèmes distribués.

Table des matières

A. Installation de Docker

1. Installation de Docker sous Debian 11

Docker était à l'origine basé sur les conteneurs Linux (LXC) et les fonctionnalités du noyau Linux comme les namespaces et les cgroups, etc.

L'installation est possible sur un grand nombre de distributions (Debian, Ubuntu, Arch, Fedora, CentOS, RHEL, etc...)

Documentation Debian : https://docs.docker.com/engine/install/debian/

2. Installation de docker sous Windows 10

C'est une drôle d'idée, mais Docker est également disponible sur Windows

https://learn.microsoft.com/en-us/windows/wsl/install
https://docs.docker.com/desktop/install/windows-install/

3. Installation de docker sous MacOS

Docker est également disponible sur Mac (même sur Apple silicon) :

https://docs.docker.com/desktop/install/mac-install/

TAF

À partir de la documentation officielle de Docker

✔️ Installer Docker sous Debian (ou Mac, ou Windows soyons fous)

✔️ Ajouter l'utilisateur de la machine au groupe docker pour permettre à celui-ci d'utiliser le docker CLI.

Note : Ajouter un utilisateur au groupe docker sous Linux
sudo groupadd docker
sudo usermod -a -G docker std
redémarrer la session
tester avec docker ps

B. Introduction à Docker

Docker est la star incontestée (pour l'instant) des outils utilisés pour la conteneurisation.

Il permet (entre autre) :
- de s'affranchir des problèmes de compatibilité entre environnement de développement et de production (« Mais puisque je te dis que chez moi ça marche ! »)
- de simplifier l'intégration, la livraison et le déploiement continus
- faciliter l'approche DevOps

Philosophie Docker :
- Un conteneur est immuable (enfin devrait être)
- Un conteneur = une application

TAF

✔️ Lancer un premier conteneur hello-world pour tester l'installation de Docker

✔️ Expliquer le résultat de la commande

Note : docker run hello-world

1. Définitions et Command Line Interface

NOTIONS DOCKER

image : un modèle immuable constitué d'un système de fichiers par « couches » et de métadonnées au format JSON qui peut être utilisé pour créer des conteneurs Docker. Elle contient le code de l'application, l'environnement d'exécution de l'application, les bibliothèques nécessaires à l'application, les fichiers de configuration, etc.

conteneur : une instance d'une image en cours d'exécution.

volume : espace de stockage pour un ou plusieurs conteneurs.

registre : (En anglais registry) Serveur de stockage d'images Docker versionnées. Le plus connu étant DockerHub

docker-compose : outils pour définir et créer des applications multi-container.

dockerfile : Fichier texte. C'est un modèle de création d'une image à partir d'une image de base et d'instructions pour ajouter des couches à cette image.

Docker Hub : Le registre officiel d'images proposé par Docker Inc.

Command Line Interface

https://docs.docker.com/engine/reference/run/

Il est possible d'avoir la liste des commandes du Docker Engine en tapant :

```
std@kathara:~$ docker --help

Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers
```

```
Common Commands:
  run         Create and run a new container from an image
  exec        Execute a command in a running container
  ps          List containers
  build       Build an image from a Dockerfile
  pull        Download an image from a registry
  push        Upload an image to a registry
  images      List images
  login       Log in to a registry
  logout      Log out from a registry
  search      Search Docker Hub for images
  version     Show the Docker version information
  info        Display system-wide information
```

La commande docker run permet de créer et lancer un nouveau conteneur.

Pour connaître les options de cette commande il est encore possible de taper :

```
std@kathara:~$ docker run --help

Usage:  docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Create and run a new container from an image

...
```

Quelques exemples

```
docker run hello-world
```

Cette commande permet de lancer un conteneur basé sur l'image hello-world.

Si cette image existe localement, un conteneur est instancié à partir de cette image, sinon elle est téléchargée (par défaut sur DockerHub) puis instanciée.

Quelques options courantes de docker run (voir docker run –help pour plus d'options)

- -d : lance le conteneur en mode daemon ou détaché et libérer le terminal
- -p : permet de mapper un port réseau entre l'hôte et le conteneur, lorsque l'on souhaite accéder à l'application depuis l'hôte.
- -v monte un « volume » partagé entre l'hôte et le conteneur.
- -it lance une commande en mode **i**nteractif (sh ou bash par exemple).

```
docker stop nginx_ct1
```

Cette commande stoppe le conteneur nommé nginx_ct1

```
docker stop 0a65bf71a836
```

Cette commande stoppe le conteneur ayant pour ID 0a65bf71a836

idem pour start, pause, restart, etc..

```
docker exec -it 0a65bf71a836 /bin/bash
```

Cette commande permet d'exécuter (docker exec) une commande en mode interactif à l'intérieur du conteneur ayant pour ID 0a65bf71a836, ici /bin/bash. Concrètement, cette commande permet d'obtenir un shell bash à l'intérieur du conteneur.

```
std@kathara:~$ docker exec -it 0a65bf71a836 /bin/bash
root@0a65bf71a836:/# ls
bin  boot  dev   docker-entrypoint.d  docker-entrypoint.sh  etc    home
lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

Il est possible de visualiser les conteneurs en cours d'exécution avec la commande :

```
docker ps
```

L'option -a permet de voir les conteneurs arrêtés (voir docker ps --help)

```
std@kathara:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS                                       NAMES
0a65bf71a836   nginx          "/docker-entrypoint.…"   7 days ago     Up 2 hours
0.0.0.0:8008->80/tcp, :::8008->80/tcp       quizzical_carson
```

Ici, on voit un conteneur nommé « **quizzical_carson** » ayant pour ID « **0a65bf71a836** ». il a été créé il y a **7 jours** à partir de l'image **nginx** et tourne depuis **2h**.

Les mappings de ports permettent de pouvoir accéder depuis l'hôte aux applications qui sont exécutées dans le conteneur.
Ici, depuis l'hôte, l'url http://localhost:**8008** permet d'accéder à l'application qui écoute sur le port **80** dans le conteneur

Pour visualiser les images disponibles localement, on peut utiliser la commande :

```
std@kathara:~/docker/fls$ docker images

REPOSITORY                TAG           IMAGE ID       CREATED         SIZE
alpine                          latest           b2aa39c304c2    8 days ago
7.05MB
nginx                     latest        3f8a00f137a0   10 days ago     142MB
debian                           stable-slim    4ea5047878b3    10 days ago
80.5MB
portainer/portainer-ce    latest        25f956834426   13 days ago     272MB
ubuntu                    23.04         beb2152822b7   3 weeks ago     70MB
kathara/quagga            latest        c5b419d675cd   16 months ago   712MB
hello-world                    latest           feb5d9fea6a5    17 months ago
13.3kB
```

La première colonne (**Repository**) est un peu trompeuse. Il s'agit du nom (et éventuellement du nom de dépôt) de l'image

Tag correspond au versionning de l'image (latest par défaut, sinon numéro de version)

Image ID est l'identifiant de l'image. Il s'agit d'un hash de l'image.

Created : la date de création de l'image

Size : La taille de l'image.

La suite et fin de toutes les commandes Docker

```
Management Commands:
  builder     Manage builds
  buildx*     Docker Buildx (Docker Inc., v0.10.2)
  compose*    Docker Compose (Docker Inc., v2.15.1)
  container   Manage containers
  context     Manage contexts
  image       Manage images
  manifest    Manage Docker image manifests and manifest lists
  network     Manage networks
  plugin      Manage plugins
  scan*       Docker Scan (Docker Inc., v0.23.0)
  system      Manage Docker
  trust       Manage trust on Docker images
  volume      Manage volumes
```

```
Swarm Commands:
  swarm       Manage Swarm
```

```
Commands:
   attach        Attach local standard input, output, and error streams to a running
container
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
```

```
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  events      Get real time events from the server
  export      Export a container's filesystem as a tar archive
  history     Show the history of an image
  import      Import the contents from a tarball to create a filesystem image
  inspect     Return low-level information on Docker objects
  kill        Kill one or more running containers
  load        Load an image from a tar archive or STDIN
  logs        Fetch the logs of a container
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  rmi         Remove one or more images
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  wait        Block until one or more containers stop, then print their exit codes
```

```
Global Options:
      --config string      Location of client config files (default "/home/std/.docker")
   -c, --context string       Name of the context to use to connect to the daemon
(overrides DOCKER_HOST env var and default context set with "docker context use")
  -D, --debug              Enable debug mode
  -H, --host list          Daemon socket(s) to connect to
   -l, --log-level string    Set the logging level ("debug", "info", "warn", "error",
"fatal") (default "info")
      --tls                Use TLS; implied by --tlsverify
           --tlscacert  string      Trust  certs  signed  only  by  this  CA  (default
"/home/std/.docker/ca.pem")
             --tlscert  string           Path  to  TLS  certificate  file  (default
"/home/std/.docker/cert.pem")
      --tlskey string      Path to TLS key file (default "/home/std/.docker/key.pem")
      --tlsverify          Use TLS and verify the remote
  -v, --version            Print version information and quit
```

Exemple concret : Déploiement d’un site avec une image nginx.

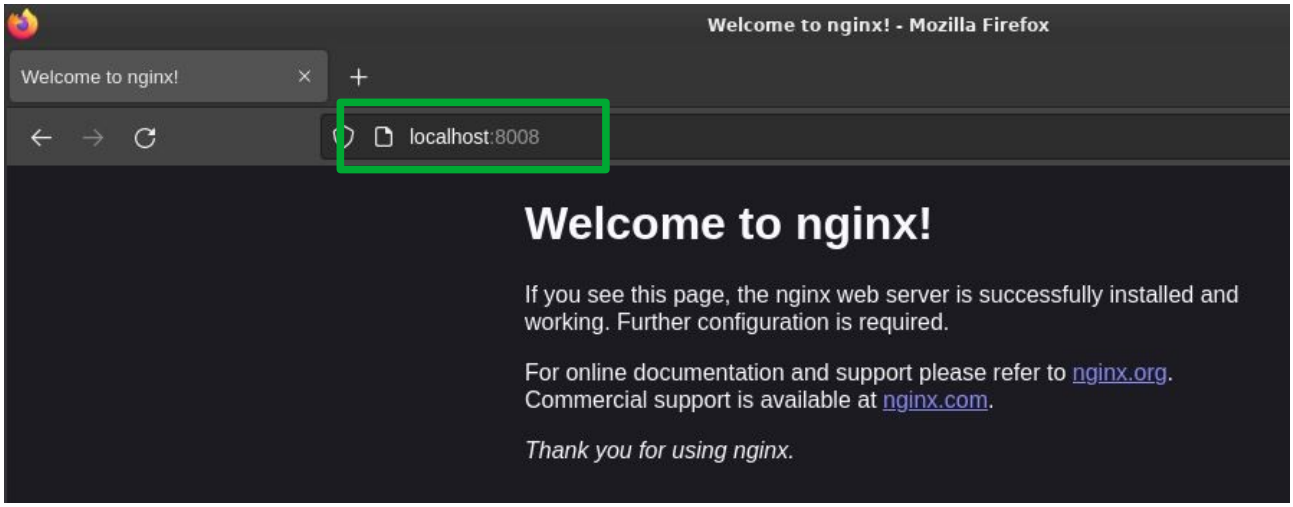
```
std@kathara:~$ docker run -d --name "mon_site" -p 8008:80 nginx
0a65bf71a8362912982f02f5911148331029c62b387d50a70f3f20ad5f3a34ed

std@kathara:~$ docker ps
CONTAINER  ID     IMAGE                                                COMMAND
CREATED                              STATUS                                PORTS
NAMES
0a65bf71a836   nginx                                "/docker-entrypoint.…"   5
seconds  ago      Up  4  seconds      0.0.0.0:8008->80/tcp,  :::8008->80/tcp
mon_site
```

On crée un conteneur nommé « mon_site » basé sur l'image officielle nginx.

On pourra accéder, depuis l'hôte, au site par le port 8008.

Résultat :




```
std@kathara:~$ docker exec -it 0a65bf71a836 bash
```

On accède à un shell bash dans le conteneur et on regarde la configuration d'nginx.

```
root@0a65bf71a836:/# cat /etc/nginx/conf.d/default.conf
server {
    listen       80;
    listen  [::]:80;
    server_name  localhost;

    #access_log  /var/log/nginx/host.access.log  main;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
```

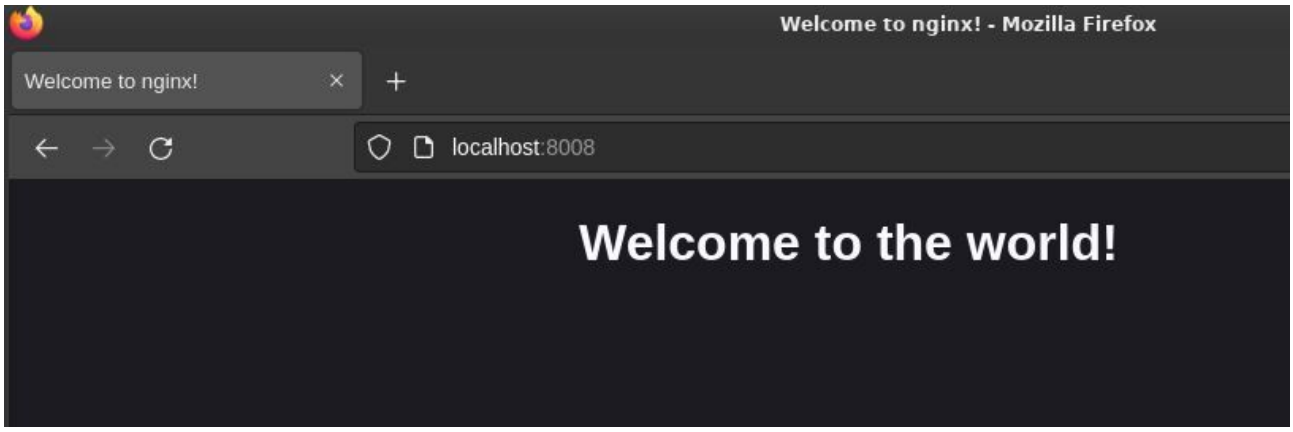
On modifie le fichier /usr/share/nginx/html/index.html

```
root@0a65bf71a836:/# apt update && apt install nano
root@0a65bf71a836:/# nano /usr/share/nginx/html/index.html
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
```

```
<body>
<h1>Welcome to the world!</h1>
</body>
</html>
```

On vérifie.



Ça fonctionne, mais on verra par la suite que cette manière de faire n'est pas du tout la bonne.

TAF

- ✔️ Créer un conteneur nginx en mappant le port 8222 de l'hôte avec le port 80 du conteneur,
- ✔️ Accéder depuis l'hôte à l'application,
- ✔️ Modifier le site,
- ✔️ Tester.

2. GUI - Portainer

Il existe plusieurs GUI pour Docker (Docker Desktop, Kitematic, Portainer, …), elles permettent de pouvoir gérer des conteneurs Docker avec une interface simple et rapide et de s'affranchir au moins en partie des lignes de commandes (ce qu'on ne fera évidemment pas).

Portainer est une interface web qui se trouve directement sous la forme d'une image Docker, donc l'installation est simple et rapide.

https://docs.portainer.io/start/install-ce/server/docker/linux

TAF

À partir de la documentation officielle de Portainer

- ✔️ Installer et tester Portainer

3. Dockerfile – Création d'une image

https://docs.docker.com/engine/reference/builder/
https://docs.docker.com/build/building/packaging/

Une image docker un modèle immuable constitué d'un système de fichiers par « couches » et de métadonnées au format JSON qui peut être utilisé pour créer des conteneurs Docker. Elle contient le code de l'application, l'environnement d'exécution de l'application, les bibliothèques nécessaires à l'application, les fichiers de configuration, etc. Elle est généralement créée à partir d'un fichier Dockerfile et chaque couche correspond (presque) à une instruction du Dockerfile.

Les images sont créées en empilant de couches en général sur une image existante (dans un registre) grâce à un système de fichiers (union file system avec overlay2)

https://www.terriblecode.com/blog/how-docker-images-work-union-file-systems-for-dummies/

Un **Dockerfile** est un fichier texte qui décrit la création d'une image à partir d'une image de base et d'instructions pour ajouter des couches à cette image.

Exemple de Dockerfile

```
FROM debian:stable-slim

RUN apt-get update && apt-get install apache2 -y

WORKDIR /var/www/html

RUN rm -f /var/www/html/index.html

COPY index.html /var/www/html/index.html

EXPOSE 80

ENTRYPOINT apache2ctl -D FOREGROUND
```

Explications

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

https://docs.docker.com/engine/reference/builder/#from

Une image docker est basée sur une image de base, ici **debian** dans sa version **stable-slim**

```
RUN apt-get update && apt-get install apache2 -y
```

https://docs.docker.com/engine/reference/builder/#run

Dans cette image de base, on met la liste des paquets à jour et on installe apache2 (il existe déjà des images apache2 sur DockerHub, mais c'est pour l'exemple)

*« L'instruction **RUN** exécutera donc une commande shell dans une nouvelle couche au-dessus de l'image actuelle et validera les résultats. L'image résultante sera utilisée pour l'étape suivante dans le Dockerfile. »*

```
WORKDIR /var/www/html
```

https://docs.docker.com/engine/reference/builder/#workdir

L'instruction **WORKDIR** définit le répertoire de travail pour toutes les instructions RUN, CMD, ENTRYPOINT, COPY et ADD qui la suivent dans le Dockerfile. Ici ce sera le répertoire /var/www/html (le Document root d'Apache)

```
RUN rm -f index.html
```

https://docs.docker.com/engine/reference/builder/#run

On supprime le fichier index.html (donc /var/www/html/index.html – cf. WORKDIR) qui est la page par défaut d'Apache. Cette instruction crée une nouvelle couche dans l'image.

```
COPY index.html index.html
```

https://docs.docker.com/engine/reference/builder/#copy

L'instruction COPY copie les fichiers ou répertoires depuis la source (1er argument – un fichier sur la machine hôte) et les ajoute au système de fichiers du conteneur au chemin de destination (2ème argument).

Dans cet exemple cela revient à écrire (COPY ./index.html /var/www/html/index.html – ce qui est plus clair)

Donc cela ajoute le fichier index.html situé dans le répertoire du Dockerfile dans le répertoire /var/www/html/ du conteneur.

L'instruction ADD est équivalente à quelques différences près.
https://docs.docker.com/engine/reference/builder/#add

```
EXPOSE 80
```

https://docs.docker.com/engine/reference/builder/#expose

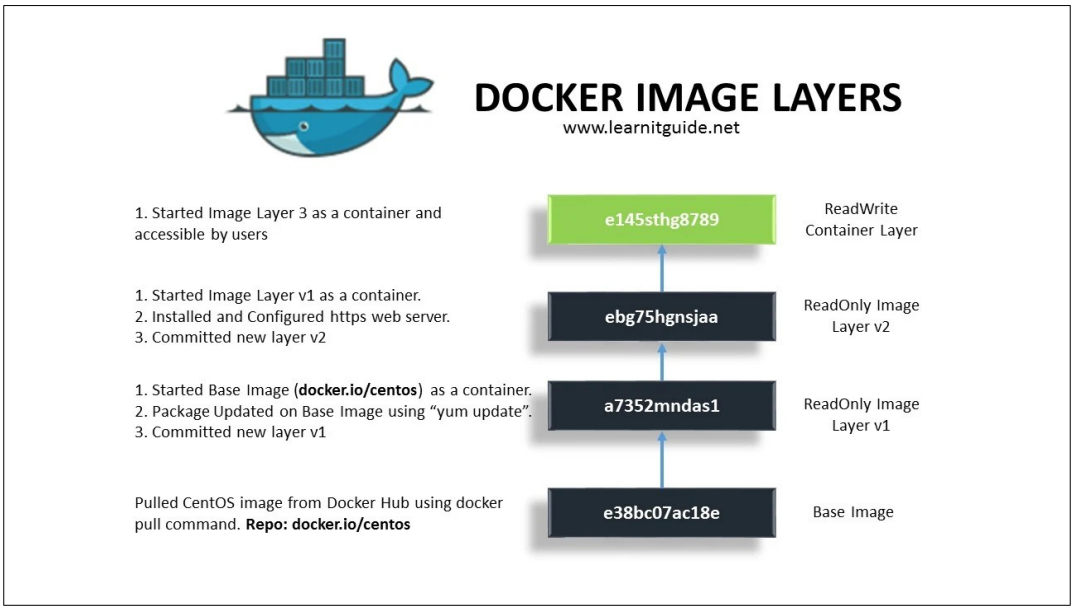
L'instruction EXPOSE informe Docker que le conteneur écoute sur les ports réseau spécifiés lors de l'exécution. Vous pouvez spécifier si le port écoute sur TCP ou UDP, et la valeur par défaut est TCP si le protocole n'est pas spécifié.

```
ENTRYPOINT apache2ctl -D FOREGROUND
```

https://docs.docker.com/engine/reference/builder/#entrypoint

Dans un Dockerfile, on utilise l'instruction ENTRYPOINT pour fournir des exécutables qui s'exécuteront toujours au lancement du conteneur.
Ici, on lance apache2 au premier plan afin que le conteneur reste en exécution. Un conteneur qui n'a pas de processus actif au premier plan se terminera instantanément après son lancement.

Les couches (Layers)

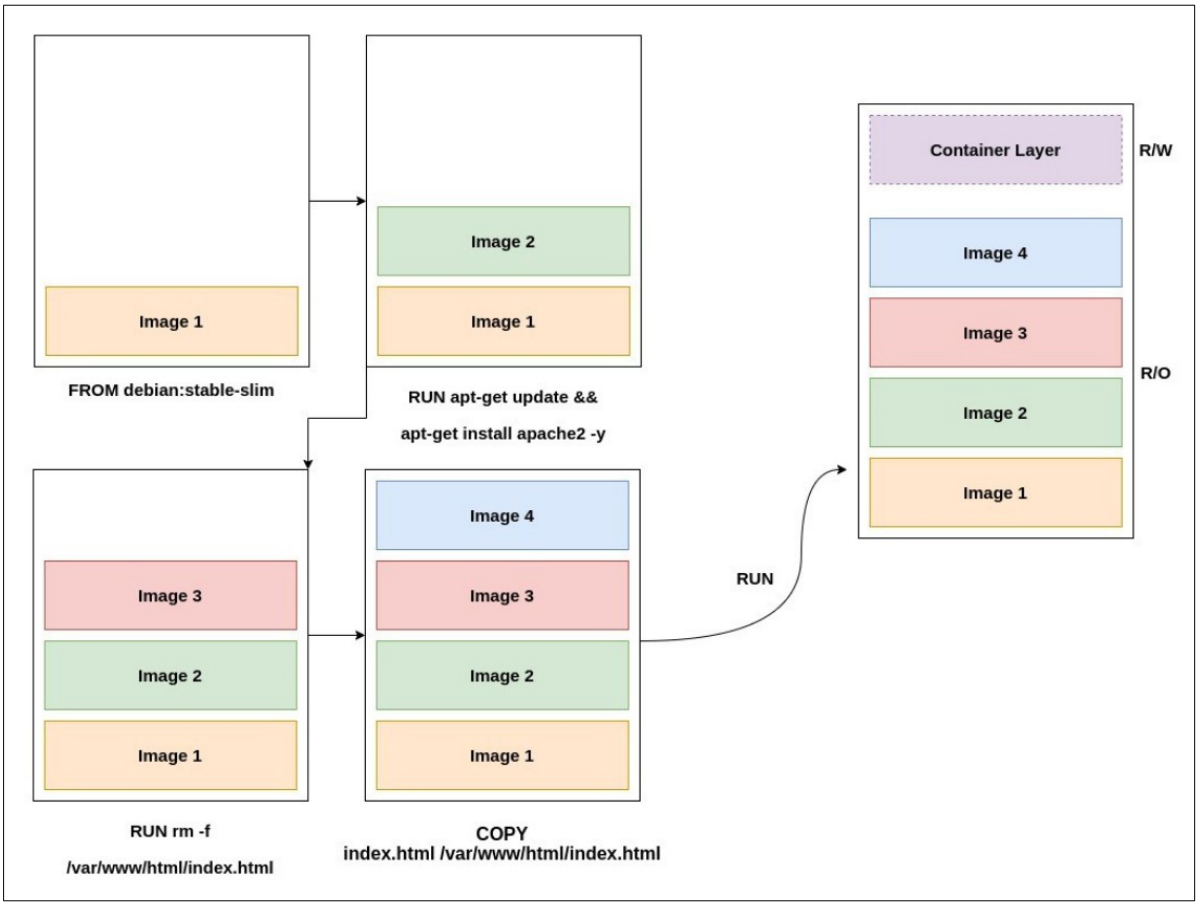


Doc : https://code.tutsplus.com/fr/tutorials/docker-from-the-ground-up-understanding-images--cms-28165

Lorsqu'un nouveau conteneur est créé à partir d'une image, toutes les couches d'image sont en lecture seule et une couche mince de lecture-écriture est ajoutée au-dessus.
Toutes les modifications apportées au conteneur spécifique sont stockées dans cette couche.

Les images sont créés en empilant de nouvelles couches sur une image existante grâce à un système de fichiers qui fait du union mount.

Pour notre exemple de Dockerfile :



Chaque commande qui modifie une image (RUN, ADD, COPY, etc) va créer une nouvelle couche. Si plusieurs images sont basées sur les mêmes couches inférieures, toutes ses couches ne seront stockées qu'une seule fois.
Seules les couches uniques à une image seront stockées pour chaque image. Cela permet des gains de place considérable.

Une couche de 50Mo x 1000 conteneurs = 50Go de stockage en moins.

Maintenant que le Dockerfile est prêt, on peut créer une image Docker à partir de ce fichier.

Maintenant que le Dockerfile est prêt, on peut créer une image Docker à partir de ce fichier.

Voici l'arborescence du répertoire

```
std@kathara:~/docker/img_apache$ tree .
.
├── Dockerfile
└── index.html

0 directories, 2 files
```

Le fichier index.html est présent car il est requis par l'instruction COPY dans le Dockerfile.

Build de l'image

```
docker build -t my_apache_image:1.0 .
```

```
std@kathara:~/docker/fls$ docker build --no-cache -t my_apache_image:1.0 .
[+] Building 7.0s (5/9)
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 398B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/debian:stable-slim
 => CACHED [1/5] FROM docker.io/library/debian:stable-slim
 => [internal] load build context
 => => transferring context: 31B
 => [2/5] RUN apt-get update && apt-get install apache2 -y
 => => # Selecting previously unselected package liblua5.3-0:amd64.
 => => # Preparing to unpack .../21-liblua5.3-0_5.3.3-1.1+b1_amd64.deb ...
 => => # Unpacking liblua5.3-0:amd64 (5.3.3-1.1+b1) ...
 => => # Selecting previously unselected package libicu67:amd64.
 => => # Preparing to unpack .../22-libicu67_67.1-7_amd64.deb ...
 => => # Unpacking libicu67:amd64 (67.1-7) ...
```

La commande permet de construire l'image sui sera nommée my_apache_build et qui aura pour TAG 1.0

```
std@kathara:~/docker/fls$ docker build -t my_apache_image:1.0 .
[+] Building 17.3s (10/10) FINISHED
```

Il aura fallu 17.3sec pour construire l'image.

```
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 398B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/debian:stable-slim
0.0s
 => CACHED [1/5] FROM docker.io/library/debian:stable-slim
0.0s
 => [internal] load build context
0.0s
 => => transferring context: 31B
0.0s
 => [2/5] RUN apt-get update && apt-get install apache2 -y
15.6s
 => [3/5] WORKDIR /var/www/html
0.1s
 => [4/5] RUN rm -f index.html
0.4s
 => [5/5] COPY index.html index.html
0.1s
 => exporting to image
1.0s
 => => exporting layers
0.9s
```

```
 => => writing image
sha256:5d1344cfc34dd9f1a43b8309ac53ca4acbeb4ad64f01856648de746f388b30b7
0.0s
 => => naming to docker.io/library/my_apache_image:1.0
0.0s
```

À ce stade, aucun conteneur n'a été créé/lancé, une image a été construite. Il faudra instancier cette image pour avoir un conteneur.

On vérifie les images locales.

```
std@kathara:~/docker/fls$ docker images
```

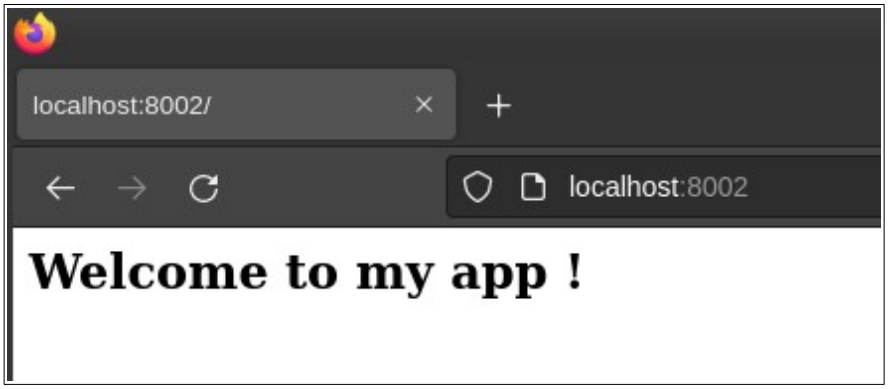
```
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
my_apache_image     1.0       5d1344cfc34d   5 minutes ago    205MB
```

On voit bien notre nouvelle image.
On instancie maintenant un conteneur basé sur cette image.

```
std@kathara:~/docker/fls$ docker run -d --name "my_first_apache_container"
-p 8002:80 my_apache_image:1.0
767f79e6f02144d571326076a921f2aacb1c14c6671ddf5c451679cc22237fbb
```

```
std@kathara:~/docker/fls$ docker ps
CONTAINER ID   IMAGE                                                COMMAND
CREATED                                STATUS                                PORTS
NAMES
767f79e6f021   my_apache_image:1.0                  "/bin/sh -c 'apache2…"   3
seconds ago       Up 2 seconds       0.0.0.0:8002->80/tcp, :::8002->80/tcp
my_first_apache_container
```

On teste



TAF

En vous aidant du document et de la documentation officielle Docker :

- ✔️ Écrire un Dockerfile pour déployer une application php dans un conteneur
- ✔️ Écrire un Dockerfile pour déployer une application java* dans un conteneur
- ✔️ Écrire un Dockerfile pour déployer une application C dans un conteneur

* https://www.linode.com/docs/guides/how-to-install-openjdk-on-debian-10/ **(ou pas)**

TAF

À l'aide des commandes **docker inspect** et **docker history** et de l'outil **dive**

- ✔️ Explorer les couches des différentes images créées ci-dessus
- ✔️ Décrire les modifications apportées à chaque couche

Dive : https://www.padok.fr/blog/image-docker-dive

Lien vers le paquet Debian :
https://github.com/wagoodman/dive/releases/download/v0.9.2/dive_0.9.2_linux_amd64.deb

TAF

En vous aidant du document et de la documentation officielle Docker

- ✔️ Écrire un Dockerfile pour déployer une application nginx ou apache2, php et MariaDB dans un conteneur (par exemple, l'application du TP Proxmox)

ANNEXE

Legacy builder (avant buildkit) pour appréhender le concept de couches

```
std@kathara:~/apache$ DOCKER_BUILDKIT=0 docker build -t my_apache_app .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            BuildKit is currently disabled; enable it by removing the DOCKER_BUILDKIT=0
            environment-variable.

Sending build context to Docker daemon  3.072kB
Step 1/7 : FROM debian:stable-slim
stable-slim: Pulling from library/debian
de661c304c1d: Pull complete
Digest: sha256:f711bda490b4e5803ee7f634483c4e6fa7dae54102654f2c231ca58eb233a2f1
Status: Downloaded newer image for debian:stable-slim
 ---> 4ea5047878b3
Step 2/7 : RUN apt-get update && apt-get install apache2 -y
 ---> Running in e213460c2b20
Get:1 http://deb.debian.org/debian stable InRelease [116 kB]
Get:2 http://deb.debian.org/debian-security stable-security InRelease [48.4 kB]
[…] ## le apt update
Need to get 24.7 MB of archives.
After this operation, 111 MB of additional disk space will be used.
Get:1  http://deb.debian.org/debian  stable/main  amd64  perl-modules-5.32  all  5.32.1-
4+deb11u2 [2823 kB]
Get:2 http://deb.debian.org/debian stable/main amd64 libgdbm6 amd64 1.19-2 [64.9 kB]
[…]  ## le apt install apache2 et les dépendances
done.
Removing intermediate container e213460c2b20
 ---> cb4cc6746ed9
Step 3/7 : WORKDIR /var/www/html
 ---> Running in 2c0a18e59166
Removing intermediate container 2c0a18e59166
 ---> 77d719f81ff9
Step 4/7 : RUN rm -f /var/www/html/index.html
 ---> Running in 1b02c367a3a3
Removing intermediate container 1b02c367a3a3
 ---> 1fc045b30015
Step 5/7 : COPY index.html /var/www/html/index.html
 ---> 43d3cdf3f9f1
Step 6/7 : EXPOSE 80
 ---> Running in af57b05d1437
Removing intermediate container af57b05d1437
 ---> 2dad5733de0c
Step 7/7 : ENTRYPOINT apache2ctl -D FOREGROUND
 ---> Running in 52e4c1f57613
Removing intermediate container 52e4c1f57613
 ---> 165ff352d245
Successfully built 165ff352d245
Successfully tagged my_apache_app:latest
```