

# —

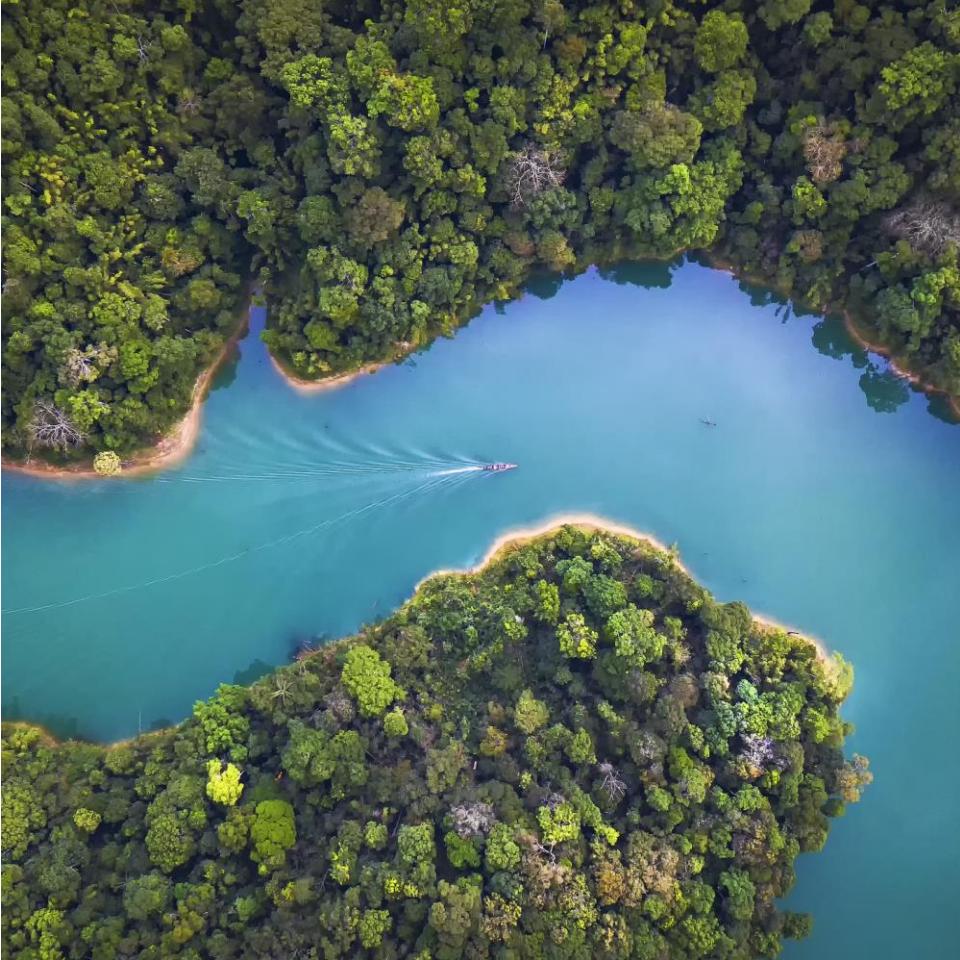
# DOCKER

Le 18/07/2023



# Sommaire

1. Présentation générale
2. TP 1 Usages de base
3. TP 2 Usages avancés
4. TP 3 Docker-compose
5. TP 4 Création d'image



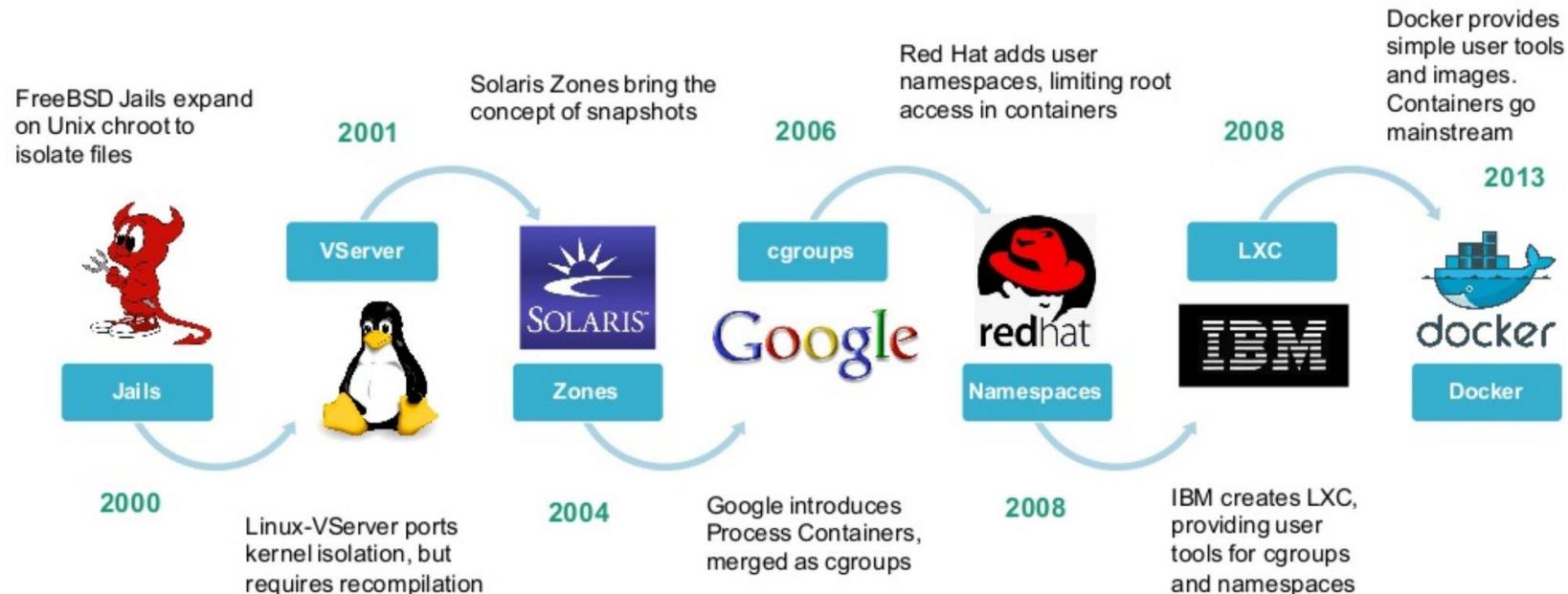
# Présentation Générale



niji

# Un peu d'histoire

La recherche de l'isolation au fil du temps



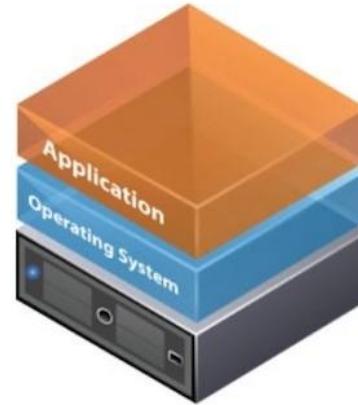
# Virtualisation VS Conteneurisation

## Virtualisation

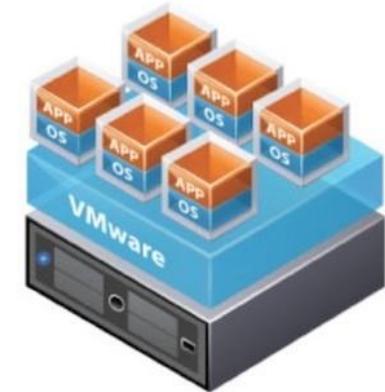
La **virtualisation** consiste à exécuter sur une machine hôte, dans un **environnement isolé**, des systèmes d'exploitation. On parle alors de **virtualisation système** (Wikipedia)



- Utilisation optimale des ressources (répartition des VM en fonction de la charge)
- Installation, déploiement et migration simple des VMs
- Economie sur le matériel par mutualisation
- Sécurisation et/ou isolation d'un réseau
- Allocation dynamique de la puissance de calcul



Traditional Architecture



Virtual Architecture



- Perte de performance de l'application
- Serveur hôte point critique de toutes les applications hôtes
- Mise en œuvre complexe et demande un investissement initial

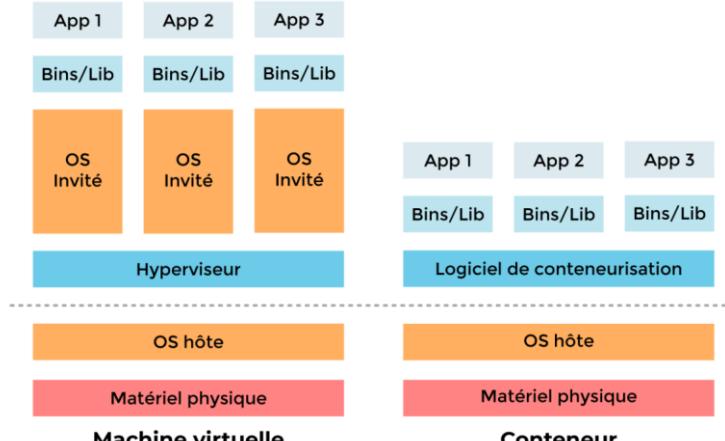
# Virtualisation VS Conteneurisation

## Conteneurisation

- Un conteneur est une **enveloppe virtuelle** qui permet de distribuer une application avec tous les éléments dont elle a besoin pour fonctionner : fichiers source, environnement d'exécution, bibliothèques, outils et fichiers.
- Contrairement à une machine virtuelle, le conteneur **n'intègre pas de noyau**, il s'appuie directement sur le noyau de l'ordinateur sur lequel il est déployé.  
(Wikipedia)



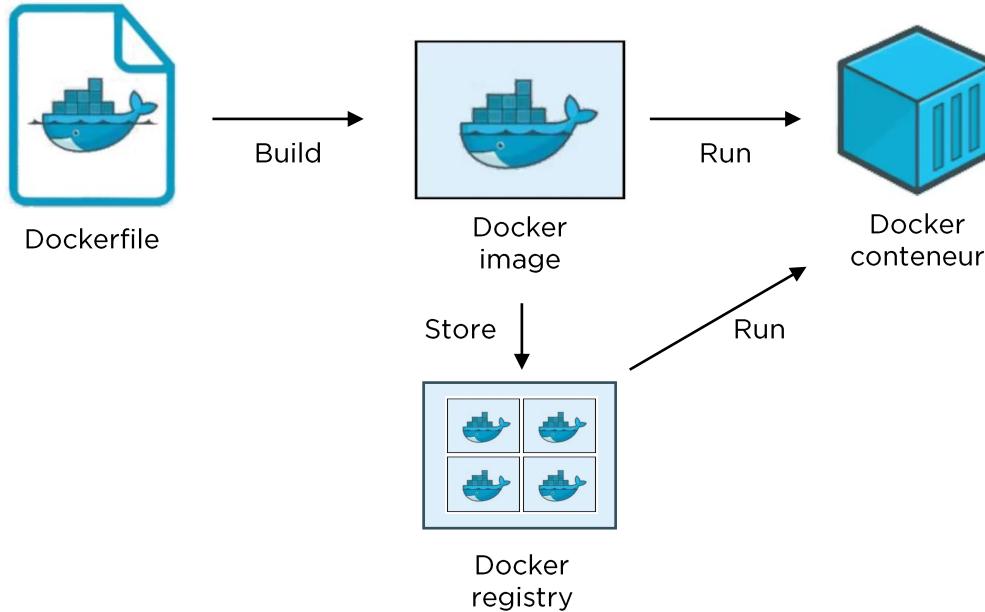
- Flexibilité : Tous les applications peuvent être transformées en des conteneurs
- Légereté : Partage le noyau du système d'exploitation de l'hôte
- Portabilité : Les images sont agnostiques de l'OS hôte du moment qu'il est sur la même architecture
- Scalabilité : Dupliquer un container est extrêmement simple, ce qui permet de réaliser de la scalabilité horizontale aisément
- Sécurité : Les conteneurs sont étanches et autonomes



- Pas nativement cross-platform une image est dépendante de l'architecture CPU (arm / x86 / ...) et du type d'OS hôte. La plupart des images sont Linux / x86 et tournent dans une couche de virtualisation sous Windows

# Cycle de vie

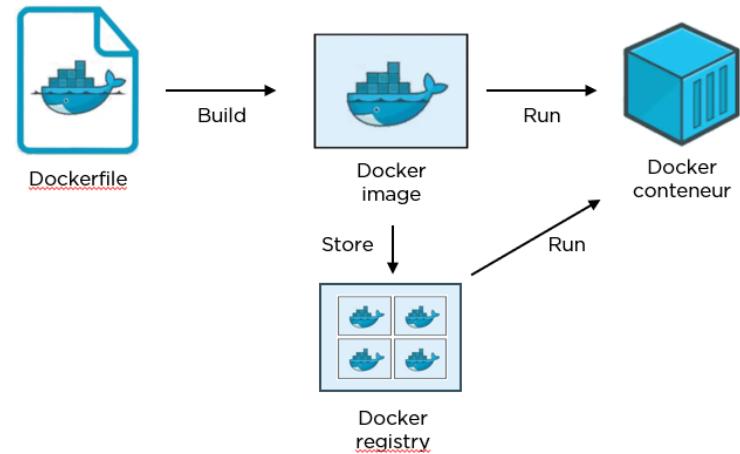
Les étapes de la définition à l'exécution



# Cycle de vie

## Jargon

- **Dockerfile** : Description de la construction d'une image
- **Image** : Fichier immuable, qui contient le code source, les librairies, dépendances et outils permettant à l'application de s'exécuter
- **Conteneur** : Un conteneur Docker est une instanciation d'image
- **Registry** : Zone de stockage d'images



# Ecosystème normatif

OCI - CNCF

L'écosystème « conteneur » est **normé et piloté** par

- L'Open Container Initiative ([OCI](https://opencontainers.org/)) <https://opencontainers.org/>
- La Cloud Native Computing Foundation ([CNCF](https://www.cncf.io/)) <https://www.cncf.io/>

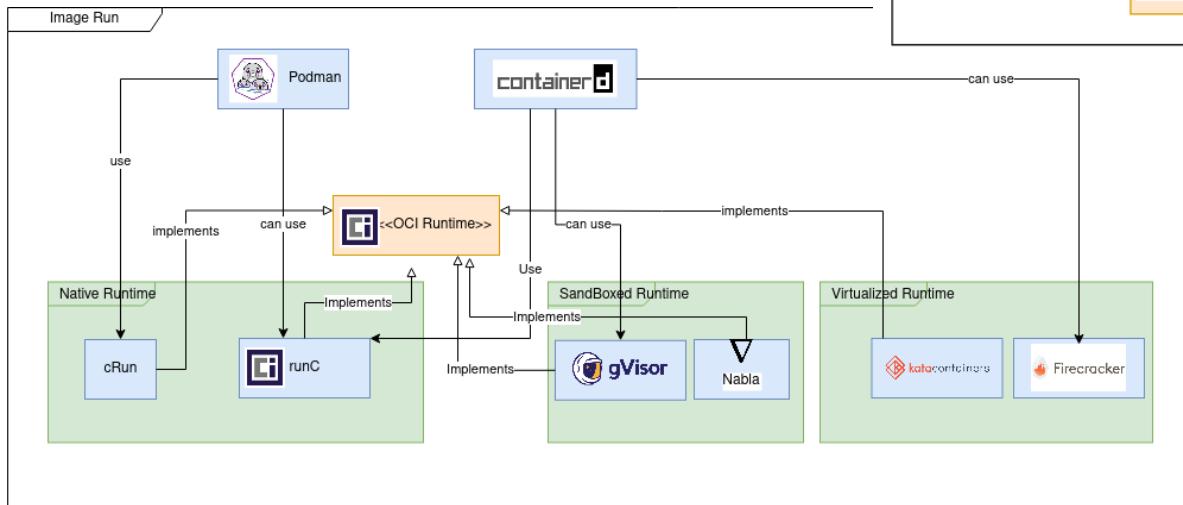
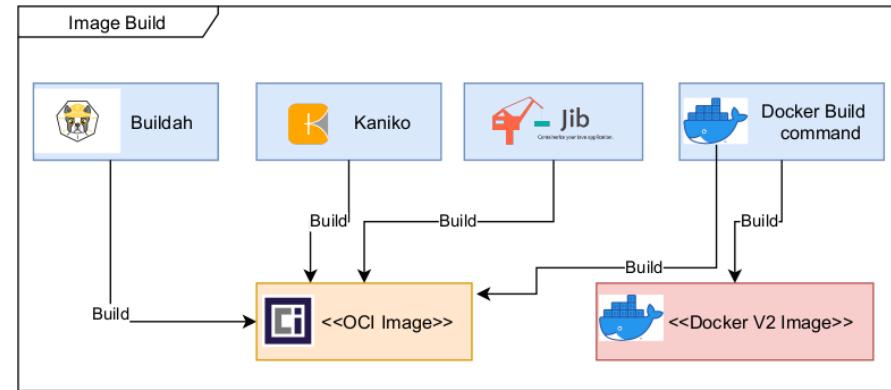
Les trois principales normes décrivent :

- OCI Image Specification : la définition des descriptions d'images
- OCI Runtime Specification : la définition des images
- OCI Distribution Specification : la définition du stockage des images (registry)

# De multiples acteurs

Sur tous les périmètres

Docker n'est pas le seul acteur sur les périmètres du build, run et store

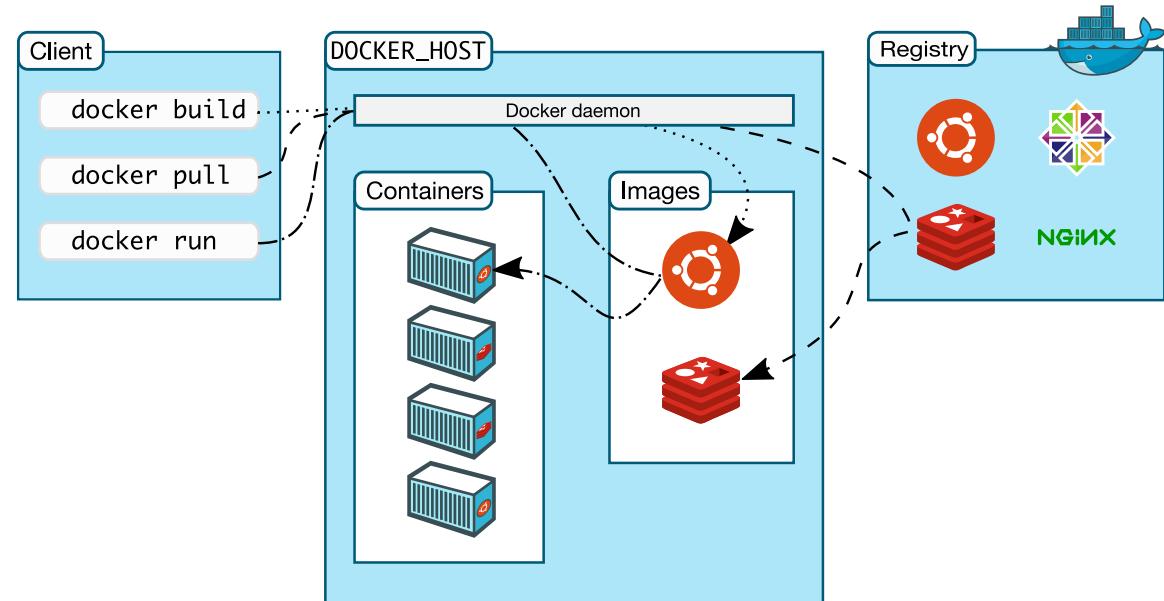


# Docker architecture

CLI, Daemon

Deux principaux composants :

- **Docker CLI** : Interface ligne de commande permettant **d'interagir avec le Docker Daemon** et notamment de construire, stocker des images, créer des conteneurs,...
- **Docker Daemon** : Composant en charge de **réaliser les actions** demandées par le Docker CLI



# Usage des conteneurs

Pour les dèvs, pour les tests, pour la prod, ...

## Développeurs

Gestion simple, rapide et sans impact sur le système hôte des dépendances applicatives (base de données, application tierce).

Facilite la gestion des environnements de développement multi projets, multi technos, multi versions

Facilité la standardisation et la distribution d'un environnement de développement au sein d'une équipe

## Tests

Facilite la mise en place d'environnements éphémères et reproductibles pour les tests

Par conséquence facilite la mise en place de tests automatisés au sein d'une chaîne d'intégration continue

Garantie sur la composition de l'application et donc la validité des tests. Une image de conteneur est immuable, son contenu est connu et non modifiable

## Intégration et déploiement

Facilite le déploiement applicatif, le conteneur étant autonome et consistant

Facilite la scalabilité horizontale, une conteneur étant très rapidement instanciable

Permet d'optimiser les ressources en utilisant un orchestrateur de conteneur

Facilité le GoToCloud

# TP 1

## Usages de base



# TP

```
git clone https://github.com/stephaneloison/formation-docker
```

# TP 1.1 pull

pull : récupération d'une image depuis un registre

```
> docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:faa03e786c97f07ef34423fccceec2398ec8a5759259f94d99078f264e9d7af
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

Ici le registre d'image utilisé est le registre par défaut : <https://hub.docker.com/>  
Ce registre est public.

L'image est récupérée et stockée dans le registre locale

```
> docker pull registry.hub.docker.com/library/hello-world
```

Commande équivalente à la précédente sauf qu'ici le registre est précisé explicitement

# TP 1.2 login & image list

## login & image list

L'usage d'un registre privé nécessite une phase d'authentification, par exemple sur Azure Cloud ou GCP

```
> az login  
> az acr login --name myregistry  
> docker login myregistry.azurecr.io
```

```
> docker login -u _json_key --password-stdin  
https://eu.gcr.io < account.json
```

Pour connaître les images présentent dans le registre local.

```
> docker image list  
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE  
hello-world     latest    feb5d9fea6a5  14 months ago  13.3kB
```

# TP 1.3 run

run : création et exécution d'une instance d'image (conteneur)

```
> docker run hello-world  
Hello from Docker!  
...
```

Ici un conteneur est créé puis exécuté.

La commande finale du conteneur termine et le conteneur est stoppé.

Pour voir l'historique d'exécution des conteneurs et notamment lorsqu'un conteneur ne s'est pas exécuté correctement

L'option -a permet de voir les conteneurs « actifs » et ceux arrêtés						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e1d74882dfad	hello-world	"/hello"	6 minutes ago	Exited (0) 6 minutes ago		nervous_turing

Annotations below the table:

- Référence du conteneur
- Nom de l'image
- Commande principale de l'image
- Durée de vie
- Etat du conteneur
- Mapping de ports
- Nom « lisible » du conteneur

# TP 1.4 logs & rm

## logs : récupération des logs

Pour voir les logs des conteneurs et notamment lorsqu'un conteneur ne s'est pas exécuté correctement

```
➤ docker logs e1d74882dfad  
Hello from Docker!
```

...

Utilisation de la référence du conteneur, le nom « lisible » est aussi utilisable

## rm : suppression d'un conteneur inactif

La suppression d'un conteneur inactif permet de nettoyer votre historique de conteneur et de libérer de la place sur votre disque

```
➤ docker rm e1d74882dfad
```

Utilisation de la référence du conteneur, le nom « lisible » est aussi utilisable

# TP 1.5 run

## run : création et exécution d'une instance d'image non éphémère

L'image hello-world exécute une commande principale éphémère (affichage d'un texte), le conteneur est donc arrêté lorsque la commande termine.

Exemple d'une image lançant une commande à durée de vie « longue »

```
➤ docker run nginx
...
2022/11/23 15:21:55 [notice] 1#1: nginx/1.21.6
...
2022/11/23 15:21:55 [notice] 1#1: start worker process 42
```

Le conteneur ne termine pas temps que la commande principale du conteneur (ici le lancement de nginx) ne termine pas

Il n'est pas possible de supprimer un conteneur actif

```
➤ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
857a59e5a779 nginx "/docker-entrypoint...." 13 seconds ago Up 12 seconds 80/tcp hungry_fermi

➤ docker rm 857a59e5a779
Error response from daemon: You cannot remove a running container
857a59e5a779d2877e7cf8b7951d25c3123cafe2a6456de03aa23b525352f31e. Stop the container before attempting removal
or force remove
```

# TP 1.6 stop

## stop : arrêt d'un conteneur actif

Pour arrêter un conteneur actif tout en conservant son historique et en pouvant le relancer :  
« stop »

```
> docker ps
CONTAINER ID  IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
857a59e5a779  nginx     "/docker-entrypoint...."  15 minutes ago  Up 15 minutes  80/tcp    hungry_fermi

> docker stop 857a59e5a779
857a59e5a779

> docker ps
CONTAINER ID  IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
```



Le conteneur est arrêté. Il est toujours présent dans l'historique des conteneurs actifs et inactifs

```
> docker ps -a
CONTAINER ID  IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
857a59e5a779  nginx     "/docker-entrypoint...."  15 minutes ago  Exited (0) 4 seconds ago  hungry_fermi
```



# TP 1.7 start

## start : lancement d'un conteneur inactif

Pour lancer un conteneur inactif : « start »

```
➤ docker ps -a
CONTAINER ID  IMAGE      COMMAND           CREATED        STATUS          PORTS     NAMES
857a59e5a779  nginx      "/docker-entrypoint...."  15 minutes ago  Exited (0) 4 seconds ago
                                                              
➤ docker start 857a59e5a779
857a59e5a779
```

```
➤ docker ps
CONTAINER ID  IMAGE      COMMAND           CREATED        STATUS          PORTS     NAMES
857a59e5a779  nginx      "/docker-entrypoint...."  24 minutes ago  Up 2 seconds  80/tcp   hungry_fermi
```

Conteneur inactif

Conteneur actif

# TP 1.8 restart

## restart : relancer un conteneur actif

Parfois il est nécessaire de relancer un conteneur actif : « restart »

Conteneur actif

```
➤ docker ps
CONTAINER ID  IMAGE      COMMAND           CREATED        STATUS          PORTS     NAMES
857a59e5a779  nginx      "/docker-entrypoint...."  41 minutes ago  Up 17 minutes  80/tcp   hungry_fermi
```

```
➤ Docker logs -f 857a59e5a779
```

```
...
2023/02/23 17:09:06 [notice] 1#1: start worker process 32
2023/02/23 17:09:06 [notice] 1#1: start worker process 33
```

Visualisation « active » des logs du conteneur

Dans un autre terminal

```
➤ docker restart 857a59e5a779
857a59e5a779
```

Dans le premier terminal

```
2023/02/23 17:09:06 [notice] 1#1: start worker process 32
2023/02/23 17:09:06 [notice] 1#1: start worker process 33
exit
```

Le conteneur a redémarré

# TP 1.8 restart

## restart : relancer un conteneur actif

Parfois il est nécessaire de relancer un conteneur actif : « restart »

```
➤ docker logs -f 857a59e5a779
...
2023/02/23 17:30:12 [notice] 1#1: worker process 23 exited with code 0
2023/02/23 17:30:12 [notice] 1#1: worker process 32 exited with code 0
2023/02/23 17:30:12 [notice] 1#1: exit
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: IPv6 listen already enabled
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/02/23 17:30:13 [notice] 1#1: using the "epoll" event method
2023/02/23 17:30:13 [notice] 1#1: nginx/1.23.2
```

Le conteneur a redémarré

Un peu de ménage...

```
➤ docker stop 857a59e5a779
857a59e5a779
➤ docker rm 857a59e5a779
857a59e5a779
```

# TP 1.9 run --rm

run --rm : suppression automatique du conteneur en cas d'arrêt

Pour instancier un conteneur qui sera automatiquement supprimé de l'historique : « --rm »

```
➤ docker run --rm nginx
...
2022/11/23 15:21:55 [notice] 1#1: nginx/1.21.6
...
2022/11/23 15:21:55 [notice] 1#1: start worker process 42
CTRL+C
```

Le conteneur est arrêté et automatiquement supprimé de l'historique, [les logs ne sont plus disponibles](#)

```
➤ docker ps -a
CONTAINER ID  IMAGE   COMMAND  CREATED  STATUS    PORTS     NAMES
```

# TP 1.10 exec

exec : exécution d'une commande dans une instance de conteneur active

Il est possible de lancer une commande dans un conteneur actif

Lancer un conteneur « long » en arrière plan

```
➤ docker run -d nginx  
c9ff1b359036489cf9651a9db6667eefe32957f065d9348ca0409f0e810d49f7
```

Lancement en  
arrière plan  
« daemon »

Par exemple visualiser quelle est la base d'OS de l'image

```
➤ docker exec ffabb more /etc/os-release  
.....  
/etc/os-release  
.....  
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"  
NAME="Debian GNU/Linux"  
...
```

# TP 1.11 exec -it

exec : exécution d'une commande interactive dans une instance de conteneur active

« exec » est utile pour se connecter à un conteneur actif afin d'explorer son contenu et éventuellement le modifier

```
➤ docker exec -it ffabb bash  
root@ffabb615ec18:/#
```

CTRL+D

Lancement d'un « bash » dans le conteneur en mode « interactif »

# TP 1.12 run -it

run -it : exécution d'une commande interactive lors de l'instanciation d'un conteneur

Il est aussi possible d'exécuter une commande interactive à la création du conteneur

```
> docker run --rm -it node:18 bash ← Lancement d'un « bash » dans le
root@ 89166ca25707:/#                                         conteneur en mode « interactif »

root@89166ca25707:/# npm -v
9.5.0

root@89166ca25707:/# node
Welcome to Node.js v18.14.2.
Type ".help" for more information.
>
```

Docker permet ici de bénéficier d'un environnement node18 sans pour autant l'avoir installé sur son poste de travail.

# TP 2

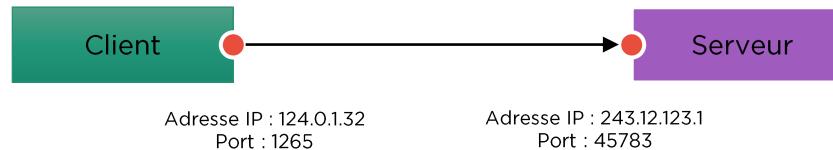
## Usages avancés



# TP 2.1 -p

## Gestion des ports interne, port externe

Certaines applications réalisent **des ouvertures de port** afin de pouvoir communiquer en TCP/IP : serveur web, serveur de base de données, etc..



Par exemple un conteneur nginx expose le port 80 en attente de connexion d'un client

```
➤ docker run -d --rm nginx
```

Par défaut le port 80 est ouvert dans le conteneur mais pas au niveau de l'hôte

```
➤ curl http://127.0.0.1:80
curl: (7) Failed to connect to 127.0.0.1 port 80: Connection refused
```

```
➤ docker exec 06ca0b37de47 curl http://127.0.0.1:80
% Total  % Received % Xferd Average Speed Time  Time  Current
          Dload Upload Total Spent Left Speed
100  615 100  615  0    0  55909   0 --:--:-- --:--:-- 76875
<!DOCTYPE html>
<html>
<head>
```

Tentative de connexion depuis l'hôte : KO

Tentative de connexion dans le conteneur : OK

# TP 2.2 -p

## Gestion des ports interne, port externe

Pour pouvoir utiliser ce port sur l'hôte il faut « publier » ce port en indiquant le numéro de port correspondant sur l'hôte, on parle de mapping de port.

Exemple : mapping de port permettant d'ouvrir le port 90 sur l'hôte. Ce port est mappé sur le port 80 en interne du conteneur

```
➤ docker run -d --rm -p 90:80 nginx
```

```
➤ curl http://127.0.0.1:90
<!DOCTYPE html>
<html>
<head>
```

Tentative de connexion depuis l'hôte : OK

```
➤ docker ps
CONTAINER ID IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
d29e6392c12f  nginx      "/docker-entrypoint...."  3 minutes ago  Up 3 minutes  0.0.0.0:90->80/tcp, ::90->80/tcp
sharp_meitner
```

Mapping du port externe:interne

# TP 2.3 -p

## Gestion des ports interne, port externe

Ce gestion du mapping de port est très pratique lorsque plusieurs application écoutant naturellement sur le même port doivent être lancées en parallèle

```
➤ docker run -d --rm -p 90:80 nginx  
➤ docker run -d --rm -p 91:80 nginx  
➤ docker run -d --rm -p 92:80 nginx
```

```
➤ curl http://127.0.0.1:90  
<!DOCTYPE html>  
<html>  
<head>
```

```
➤ curl http://127.0.0.1:91  
<!DOCTYPE html>  
<html>  
<head>
```

```
➤ curl http://127.0.0.1:92  
<!DOCTYPE html>  
<html>  
<head>
```

# TP 2.4 cp

## Echange de fichiers

Il est parfois utile de récupérer ou envoyer des fichiers dans un conteneur:

- Récupération d'une configuration,
- Dépose de sources

Exemple envoi d'un fichier dans un conteneur.

Ici modification du fichier index.html d'un conteneur nginx

```
➤ docker run --rm -d -p 90:80 nginx  
186c555425baa8d27e634a4b62a6f482559dff0293f8c701ac9e235fc92f7865
```

```
➤ docker cp index.html 186c:/usr/share/nginx/html/.  
➤ curl http://127.0.0.1:90  
well done !
```

Copie d'un fichier  
dans le conteneur

Vérification de prise  
en compte

# TP 2.4 cp

## Echange de fichiers

Exemple récupération d'un fichier dans un conteneur. Ici récupération de la configuration nginx

```
➤ docker run --rm -d -p 90:80 nginx
186c555425baa8d27e634a4b62a6f482559dff0293f8c701ac9e235fc92f7865

➤ docker cp 186c:/etc/nginx/nginx.conf .
➤ ls
index.html nginx.conf
```

# TP 2.5 -v

## Gestion des volumes

« cp » est pratique pour gérer des fichiers ponctuels mais lorsqu'un ensemble de fichiers doit être échangé entre l'hôte et le conteneur cela devient vite fastidieux.

Dans ce cas Docker permet le montage d'un répertoire ou de fichiers du système de fichiers hôte vers le conteneur

```
➤ docker run --rm -d -p 90:80 -v $(pwd)/index.html:/usr/share/nginx/html/index.html nginx  
D29e6392c12fec44565e1453c41366f18ae98d840c06c1abb86af923c6c49eb6  
➤ curl http://127.0.0.1:90  
well done !
```

Prise en compte du fichier local

```
➤ echo 'well done sir !' > index.html  
➤ curl http://127.0.0.1:90  
well done sir !
```

Edition du fichier local qui est « vu » directement du conteneur

La gestion des volumes est assez pointilleux, par exemple il faut toujours avoir un chemin absolu dans les fichiers ou répertoires référencés. Une mauvaise configuration n'est pas forcément indiquée par Docker !

```
➤ docker run --rm -d -p 90:80 -v index.html:/usr/share/nginx/html/index.html nginx  
D29e6392c12fec44565e1453c41366f18ae98d840c06c1abb86af923c6c49eb6  
➤ curl http://127.0.0.1:90  
<!DOCTYPE html>  
<html>  
<head>
```

Point de montage hôte relatif « index.html »

Le fichier n'est pas pris en compte

# TP 2.6 -v

## Gestion des volumes

Exemple d'usage : utilisation d'une image « utilitaire » busybox afin de lancer une commande non disponible sur son environnement local

```
> docker run --rm -v $(pwd):/tmp busybox find /tmp -name "*e1*"  
/tmp/de12  
  
> docker run --rm -v $(pwd):/tmp busybox /bin/sh -c "cd /tmp; find . -name '*e1*'"  
.de12
```

Exécution de la commande « find »  
sur les fichiers locaux montés dans le  
conteneur

Le retour « /tmp/\* » est normal car la  
commande est lancée dans la conteneur

Exécution d'un ensemble de  
commandes dans le conteneur

Ici nous pouvons aussi lancer un conteneur en mode « interactif » pour lancer la commande

```
> docker run --rm -it -v $(pwd):/tmp busybox sh  
/ # cd /tmp  
/tmp # find . -name '*e1*'  
.de12
```

Lancement d'un shell dans  
un nouveau conteneur en  
mode « interactif »

# TP 2.7 -v

## Gestion des volumes

Exemple d'usage : utilisation d'une image « node » non présente localement

```
> docker run --rm -v $(pwd):/usr/src/app -w /usr/src/app node:18 npm install
npm WARN deprecated core-js@2.4.1:
...
added 368 packages from 254 contributors and audited 369 packages in 22.644s
...
```

Positionnement du répertoire de travail dans le conteneur

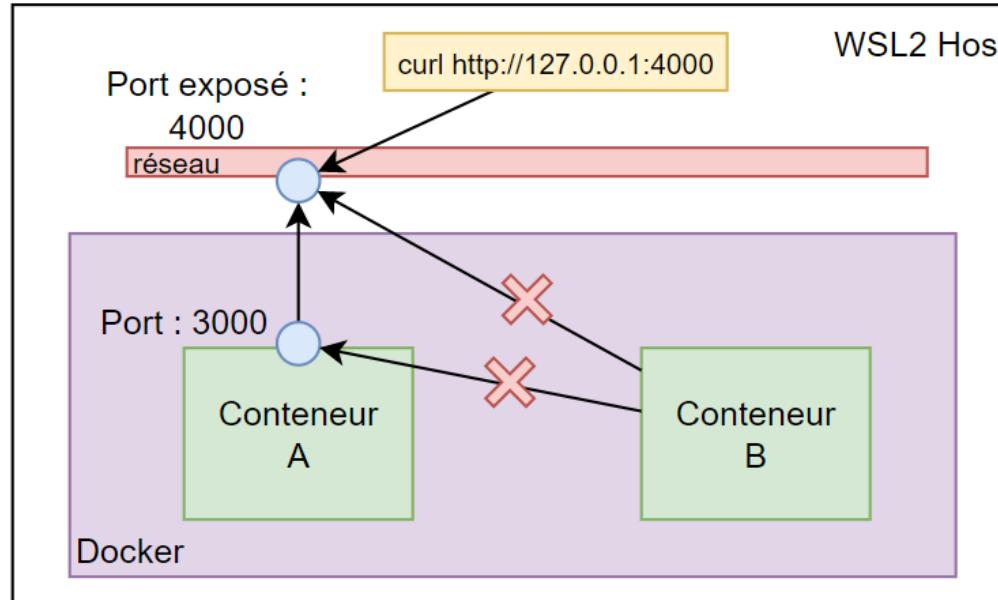
Lancement de l'application toujours sans avoir d'environnement « node » local

```
> docker run --rm -d -v $(pwd):/usr/src/app -w /usr/src/app -p 3000:3000 node:18 npm start
> curl http://127.0.0.1:3000
Hello World!
```

# TP 2.8 --network

## Gestion des networks

Par défaut un conteneur n'est pas connecté au réseau de l'hôte, il ne voit donc pas les ports exposés sur l'hôte et notamment les ports exposés par d'éventuels autres conteneurs



# TP 2.8 --network

## Gestion des networks

Lancement de l'application hello-world « node »

```
> docker run --rm -d -v $(pwd):/usr/src/app -w /usr/src/app -p 4000:3000 --name nodeapp node:18 npm start  
5505b08b15ab3ff7d03a53cc2ed1be778c750cc1cdee935cce79ede538d0777b  
> curl http://127.0.0.1:4000  
Hello World!
```

L'hôte voit le port exposé et peut l'appeler

Nom du container, peut servir de dns dans les autres conteneurs

Test d'appel dans un autre conteneur

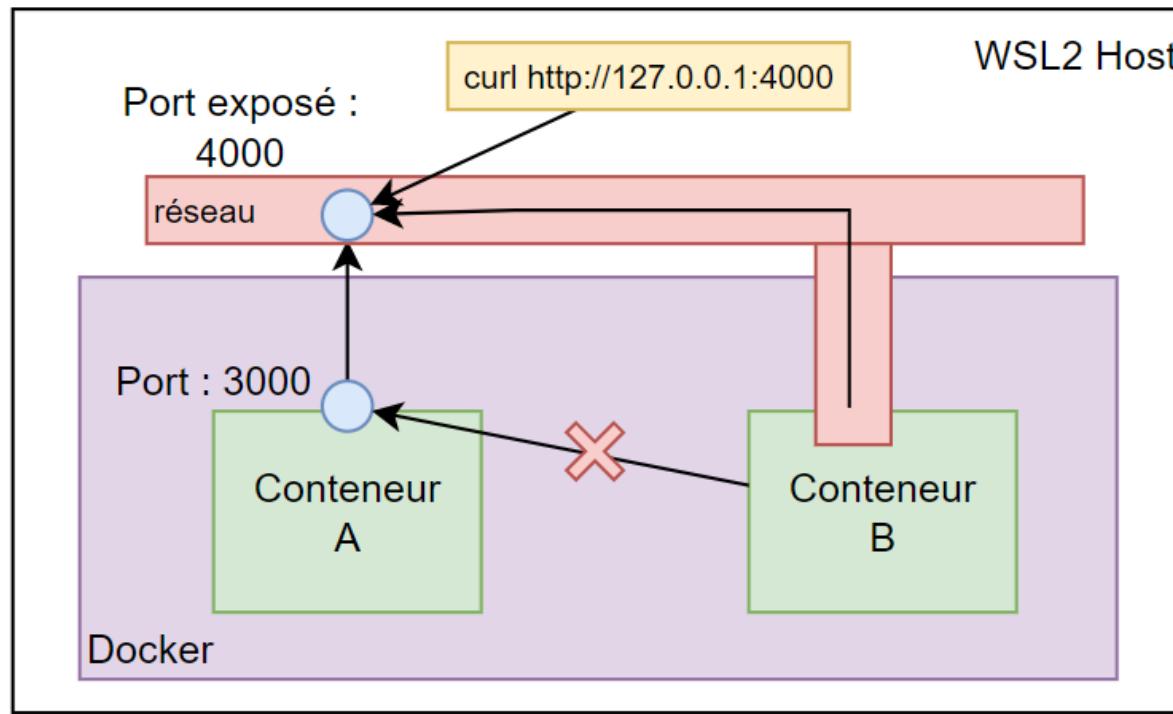
```
> docker run --rm busybox wget http://nodeapp:4000  
wget: bad address 'nodeapp:4000'
```

Echec de l'appel le conteneur n'a pas accès au réseau de l'hôte

# TP 2.8 --network

## Gestion des networks

Solution 1 : connexion du conteneur au réseau « hôte »



# TP 2.8 --network

## Gestion des networks

```
> docker run --rm --network=host busybox wget http://127.0.0.1:4000  
Connecting to 127.0.0.1:4000 (127.0.0.1:4000)  
saving to 'index.html'
```

Connexion OK en localhost !

Problème de cette solution, il n'est plus possible pour le conteneur d'exposer des ports

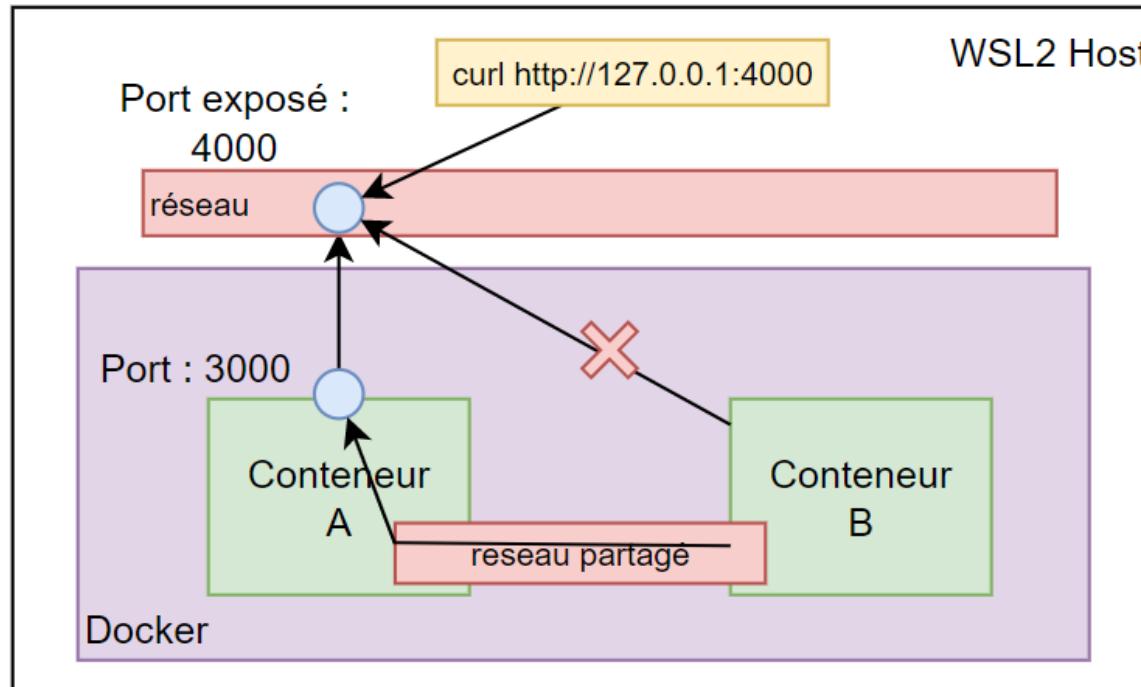
```
> docker run --rm --network=host -p 666:666 busybox wget http://127.0.0.1:4000  
WARNING: Published ports are discarded when using host network mode
```

Cela peut être une difficulté...

# TP 2.8 --network

## Gestion des networks

Solution 2 : création d'un réseau partagé entre conteneurs



# TP 2.8 --network

## Gestion des networks

Solution 2 : création d'un réseau partagé entre conteneurs

```
➤ docker network create test-network  
Dc50639f41ba37b4b28a5b79038054180e04b6013972b00982a044ff9b1a81ad  
➤ docker network ls
```

Lié les conteneurs au réseau

Mapping de port en pratique  
inutile pour la connexion entre  
les deux conteneurs

```
➤ docker run --rm -d -v $(pwd):/usr/src/app -w /usr/src/app -p 4000:3000 --network=test-network --name nodeapp  
node:18 npm start  
2d7400b6e3b7df8038309c69feb6468010103bb3d91570ba3c8e839c01688b22
```

```
➤ docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
2d7400b6e3b7 node:18 "docker-entrypoint.s..." About a minute ago Up About a minute 0.0.0.0:4000->3000/tcp,  
:::4000->3000/tcp nodeapp  
➤ docker run --rm --network=test-network -p 666:666 busybox wget http://nodeapp:3000  
Connecting to 2d7400b6e3b7:3000 (172.24.0.2:3000)  
saving to 'index.html'
```

Connexion OK sur le port interne

# TP 2.8 --network

## Gestion des networks

```
> docker run --rm -d -v $(pwd):/usr/src/app -w /usr/src/app --network=test-network --name nodeapp node:18 npm start  
2d7400b6e3b7df8038309c69feb6468010103bb3d91570ba3c8e839c01688b22
```

```
> docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
2d7400b6e3b7 node:18 "docker-entrypoint.s..." About a minute ago Up About a minute 0.0.0.0:4000->3000/tcp,  
:::4000->3000/tcp nodeapp  
> docker run --rm --network=test-network -p 666:666 busybox wget http://nodeapp:3000  
Connecting to 2d7400b6e3b7:3000 (172.24.0.2:3000)  
saving to 'index.html'
```

Plus de mapping de port

Connexion OK sur le port interne

# TP 2.9 Digression

## Gestion des ports 127.0.0.1 vs 0.0.0.0

La gestion des ports et du réseau au sens large est une source intarissable de problèmes. Un des problèmes récurrents vient de la confusion entre les adresses 127.0.0.1 et 0.0.0.0.

- 127.0.0.1 correspond à l'adresse IP de « loopback » « boucle locale » elle permet de cibler le réseau local à machine.
- 0.0.0.0 est une adresse particulière servant notamment à autoriser l'accès depuis les autres réseaux que le réseau local

Dans l'exemple précédent

```
> docker ps
CONTAINER ID IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
d29e6392c12f  nginx      "/dock...erpoint...."  3 minutes ago  Up 3 minutes  0.0.0.0:90->80/tcp, :::90->80/tcp
sharp_meitner
```

Ouverture aux réseaux externes

```
> docker run -d --rm -p 127.0.0.1:90:80 nginx
> curl http://127.0.0.1:90
curl: (56) Recv failure: Connection reset by peer
```

Le port n'est plus ouvert sur l'hôte mais uniquement dans le conteneur

# TP 2.9 Digression

## Gestion des ports 127.0.0.1 vs 0.0.0.0

Le cas fréquent correspond à un exécutable qui ouvre un port uniquement sur la boucle locale « 127.0.0.1 » et qui donc n'est pas accessible à l'extérieur.

```
> docker run --rm -v $(pwd):/usr/src/app -w /usr/src/app node:18 npm install
> docker run --rm -d -v $(pwd):/usr/src/app -w /usr/src/app -p 3000:3000 node:18 npm start
> docker ps
CONTAINER ID  IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
20ed606322cd  node:18    "docker-entrypoint.s..."  3 seconds ago  Up 2 seconds  0.0.0.0:3000->3000/tcp, :::3000->3000/tcp  happy_elbakyan
```

Ouverture aux réseaux externes

```
> curl http://127.0.0.1:3000
curl: (56) Recv failure: Connection reset by peer
```

Pourtant l'application est inaccessible !

```
> code index.js
...
//Launch listening server on port 3000
app.listen(3000, "127.0.0.1", function () {
  console.log('app listening on port http://localhost:3000!')
})
```

L'application est configuré pour n'être accessible que sur la boucle locale

```
> docker exec 20ed606322cd curl http://127.0.0.1:3000
Hello World!
```

Dans le conteneur « localement » le port est bien disponible et l'application répond

# TP 2.9 Digression

## Gestion des ports 127.0.0.1 vs 0.0.0.0

Ici il faut changer la configuration applicative pour que le port soit ouvert sur l'extérieur

```
...
//Launch listening server on port 3000
app.listen(3000, "0.0.0.0", function () {
  console.log('app listening on port http://localhost:3000!')
})
```

Ouverture sur tous les réseaux

```
> docker run --rm -v $(pwd):/usr/src/app -w /usr/src/app node:18 npm install
> docker run --rm -d -v $(pwd):/usr/src/app -w /usr/src/app -p 3000:3000 node:18 npm start
> docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
20ed606322cd node:18 "docker-entrypoint.s..." 3 seconds ago Up 2 seconds 0.0.0.0:3000->3000/tcp, ::3000->3000/tcp happy_elbakyan
```

Aucun changement côté Docker  
Ouverture aux réseaux externes

```
> curl http://127.0.0.1:3000
> Hello World!
```

L'application est inaccessible

# TP 2.10 ps / stop / rm / rmi

## Un peu de ménage

docker ps : visualisation des conteneurs actifs

docker ps -a : visualisation des conteneurs actifs et non actifs

docker stop : arrêt d'un conteneur actif

docker rm : suppression d'un conteneur inactif

docker image list : visualisation des images du registre local

docker rmi : suppression d'une image

docker network rm : suppression d'un réseau

# TP 3

## Docker- compose



niji

# docker-compose

## Infrastructure locale

Lorsque le lancement d'une image devient complexe, nombreux ports et volumes, ou que plusieurs applications doivent être lancées et communiquer entre elles « docker-compose » simplifie la description et l'exécution de plusieurs commandes « docker » séparées.

```
version: "3"
services:
  sqlserver:
    container_name: ${COMPOSE_PROJECT_NAME}_sqlserver_portailsis
    image: mcr.microsoft.com/mssql/server:2017-latest
    environment:
      ACCEPT_EULA: "Y"
      SA_PASSWORD: "${ENV_BDD_PORTAILSIS_ROOT_PASSWORD}"
    ports:
      - ${ENV_BDD_PORTAILSIS_PORT}:1433
  mariadb_glpi10:
    container_name: ${COMPOSE_PROJECT_NAME}_bdd_glpi
    image: mariadb:10.7
    hostname: mariadb_glpi10
    environment:
      - MARIADB_ROOT_PASSWORD=${ENV_BDD_GLPI_ROOT_PASSWORD}
      - MARIADB_DATABASE=${ENV_BDD_GLPI_NAME}
      - MARIADB_USER=${ENV_BDD_GLPI_USER}
      - MARIADB_PASSWORD=${ENV_BDD_GLPI_PASSWORD}
    ports:
      - ${ENV_BDD_GLPI_PORT}:3306
```

Conteneur SqlServer

```
#BASE HELIOS
postgresql-helios:
  container_name: ${COMPOSE_PROJECT_NAME}_postgresql_helios
  image: postgres:13
  environment:
    POSTGRES_PASSWORD: ${ENV_BDD_HELIOS_PASSWORD}
    POSTGRES_USER: ${ENV_BDD_HELIOS_USER}
    POSTGRES_DB: ${ENV_BDD_HELIOS_NAME}
  ports:
    - ${ENV_BDD_HELIOS_PORT}:5432

#GLPI Container
glpi:
  container_name: ${COMPOSE_PROJECT_NAME}_glpi
  image: diouxx/glpi
  environment:
    - VERSION_GLPI=10.0.3
  ports:
    - "808:80"
```

Conteneur PostgreSQL

Conteneur GLPI

Conteneur MariaDB

# docker-compose

## Infrastructure locale

```
# Use root/example as user/password credentials  
version: '3.1'
```

```
services:
```

```
mongo:
```

```
  image: mongo  
  restart: always  
  environment:
```

```
    MONGO_INITDB_ROOT_USERNAME: root  
    MONGO_INITDB_ROOT_PASSWORD: example
```

```
mongo-express:
```

```
  image: mongo-express
```

```
  restart: always
```

```
  ports:
```

```
    - 8081:8081
```

```
  environment:
```

```
    ME_CONFIG_MONGODB_ADMINUSERNAME: root  
    ME_CONFIG_MONGODB_ADMINPASSWORD: example  
    ME_CONFIG_MONGODB_URL: mongodb://root@example@mongo:27017/
```

Premier conteneur mongo

Relance automatique

Variables d'environnement positionnées dans le conteneur

Second conteneur mongo express

Exposition de port

docker compose introduit la notion de **services**. Les services sont les conteneurs devant être démarrés par docker compose.

Les conteneurs déployés sont dans un réseau créé par défaut par docker compose

Le nom du service peut être utilisé comme nom de machine. Les deux conteneurs sont dans un réseau par défaut.

# TP 3.1 docker-compose

## Usage de base

docker-compose up -d : lancement des conteneurs en arrière plan

```
➤ docker-compose up -d
[+] Running 3/3
  #: Network tp31_default      Created
  #: Container tp31-mongo-express-1 Started
  #: Container tp31-mongo-1     Started
```

➤ docker network list

NETWORK ID	NAME	DRIVER	SCOPE
4b87d13cef58	tp31_default	bridge	local

Démarrage des conteneurs

Réseau par défaut créé

Tester mongo express : <http://localhost:8081/>

docker-compose down : suppression des conteneurs et du réseau par défaut

```
➤ docker-compose down
[+] Running 3/3
  #: Container tp31-mongo-express-1 Removed
  #: Container tp31-mongo-1     Removed
  #: Network tp31_default       Removed
```

Suppression des conteneurs

Suppression du réseau par défaut

# TP 3.2 docker-compose

## Usage des volumes et réseau

docker-compose up -d : lancement des conteneurs en arrière plan

```
➤ docker-compose up -d
[+] Running 4/4
:: Network tp32_net2    Created
:: Container demo_app2   Started
:: Container demo_app1   Running
:: Container demo_traefik Started
```

Mettre en place une résolution locale des dns :

127.0.0.1 traefik.demo.docker

127.0.0.1 app1.demo.docker

127.0.0.1 app2.demo.docker

Tester :

Traefik : <http://traefik.demo.docker/>

App1 : <http://app1.demo.docker/>

App2 : <http://app2.demo.docker/>

```
version: '3.1'

services:
  traefik:
    image: traefik:2.6
    networks:
      - net1
      - net2
    ports:
      - 80:80
    ...
  app1:
    container_name: demo_app1
    image: node:19
    ...
    networks:
      - net1
    ...
  app2:
    container_name: demo_app2
    image: node:19
    ...
    networks:
      - net2
    ...
networks:
  net1:
  net2:
```

```
➤ docker-compose down
```

# TP 3.2 docker-compose

## Usage des volumes et réseau

Pour le besoin de la démo deux réseaux sont créés. Traefik voit les deux réseaux, chaque application voit son réseau. Les applications app1 et app2 ne se voient pas.

```
version: '3.1'

services:
  traefik:
    image: traefik:2.6
    networks:
      - net1
      - net2
    ports:
      - 80:80
    ...
  app1:
    container_name: demo_app1
    image: node:19
    ...
    networks:
      - net1
    ...
  app2:
    container_name: demo_app2
    image: node:19
    ...
    networks:
      - net2
  ...
networks:
  net1:
  net2:
```

```
> docker exec demo_traefik ping demo_app1
PING demo_app1 (172.24.0.2): 56 data bytes
64 bytes from 172.24.0.2: seq=0 ttl=64 time=0.053 ms
<-- Le conteneur traefik
  « voit » le conteneur
  demo_app1

> docker exec demo_traefik ping demo_app2
PING demo_app2 (172.26.0.3): 56 data bytes
64 bytes from 172.26.0.3: seq=0 ttl=64 time=0.083 ms
<-- Le conteneur traefik
  « voit » le conteneur
  demo_app2

> docker exec demo_app1 bash -c "apt-get update; apt-get install -y iputils-ping; ping demo_app2"
...
ping: demo_app2: Name or service not known
```

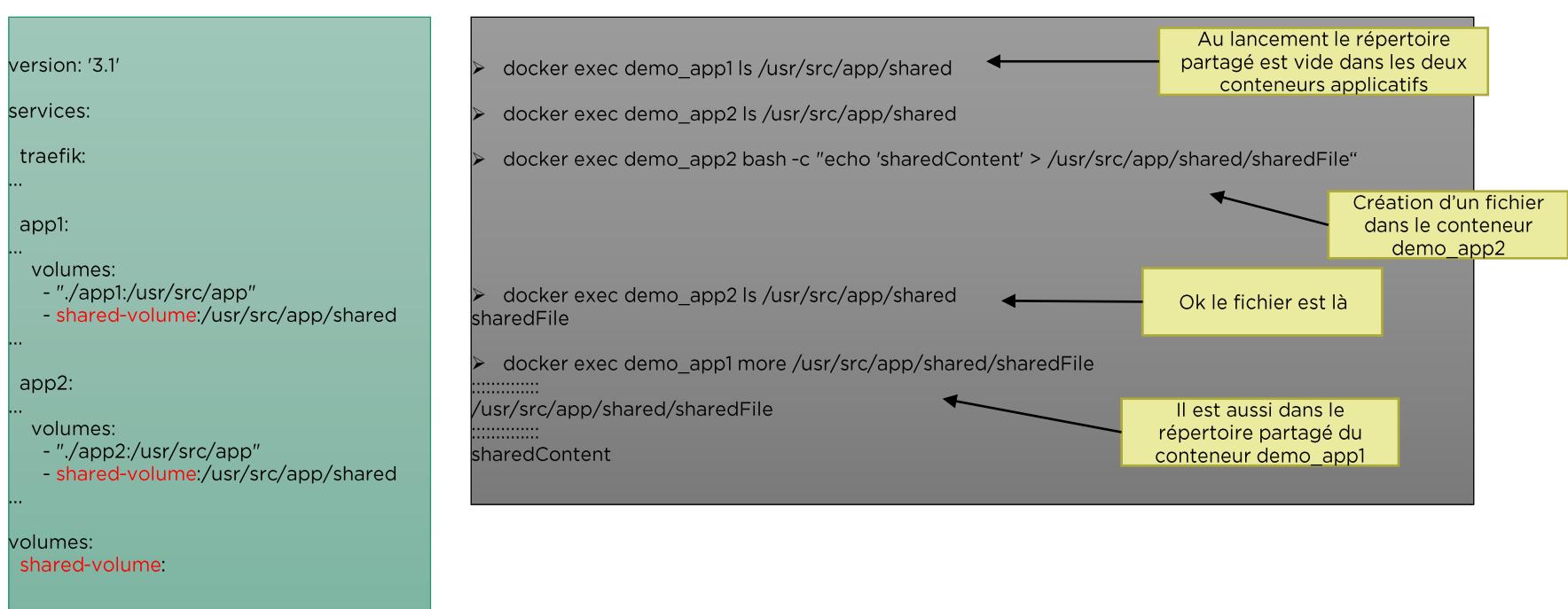
Le conteneur demo\_app1  
ne « voit » pas le  
conteneur demo\_app2

Ici nativement le conteneur demo\_app1  
ne contient pas la commande « ping ».  
Il faut l'installer avant de l'utiliser

# TP 3.2 docker-compose

## Usage des volumes et réseau

Pour le besoin de la démo une volume partagé entre les deux conteneurs applicatifs est créé.



# TP 3.3 docker-compose

## Dépendances

Il est possible de créer un arbre de dépendance entre les services. Docker-compose démarrera les services dans l'ordre choisi. ./!\\ docker-compose n'attend pas que le service en dépendance soit « prêt » mais juste qu'il soit « démarré » pour enchaîner les démarrages de services

```
version: '3.9'  
services:  
  app:  
    build: .  
    depends_on:  
      - db  
    ports:  
      - 4000:4000  
  
  db:  
    image: postgres  
    ports:  
      - '5432:5432'  
    environment:  
      POSTGRES_USER: postgres  
      POSTGRES_PASSWORD: postgres  
      POSTGRES_DB: dev
```

```
> docker-compose up -d  
...  
[+] Running 3/3  
  #: Network tp33_default Created 0.1s  
  #: Container tp33-db-1 Started 0.8s  
  #: Container tp33-app-1 Started 2.5s  
  
> docker logs tp33-app-1  
Error: connect ECONNREFUSED 192.168.0.3:5432  
  at TCPConnectWrap.afterConnect [as oncomplete] (node:net:1300:16) {  
  errno: -111,  
  code: 'ECONNREFUSED',  
  syscall: 'connect',  
  address: '192.168.0.3',  
  port: 5432  
} undefined
```

Lancement

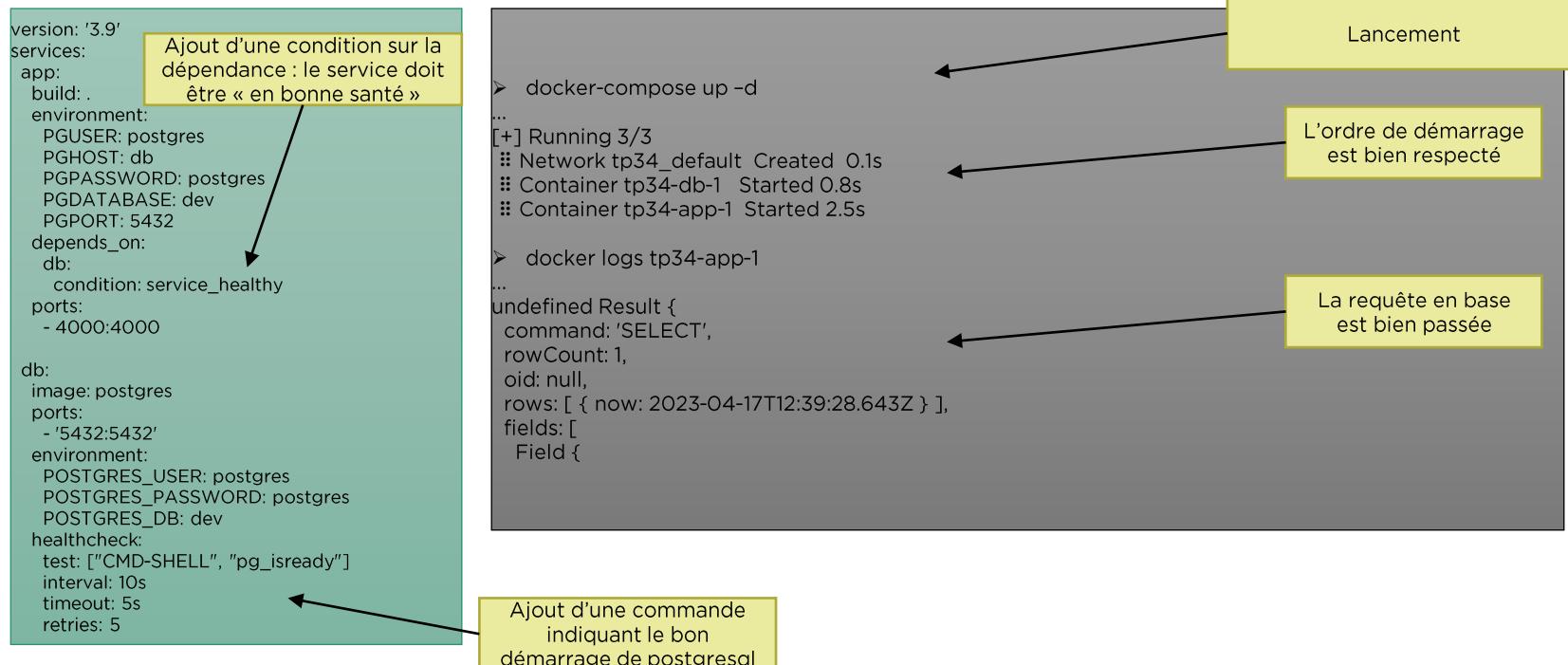
L'ordre de démarrage est bien respecté

La connexion à la base est KO

# TP 3.4 docker-compose

## Dépendances et health check

Pour gérer des dépendances il faut souvent ajouter des sondes « de bonne santé » afin que docker-compose attende le bon lancement d'un service avant de lancer le prochain



# TP 3.5 down, kill, top

## Un peu de ménage et divers

docker-compose kill : arrêt des conteneurs

docker-compose down : arrêt et suppression des conteneurs et des réseaux

docker-compose down -v : arrêt et suppression des conteneurs, des réseaux et suppression des volumes

docker-compose top : affiche les process lancés dans les conteneurs

docker-compose logs -f : affiche les logs issus de tous les conteneurs

# Stack Niji Docker Traefik Portainer



niji

# Projet Niji officiel

**git clone git@gitlab.niji.fr:dsf/docker-dev-host.git**

README.MD	Documentation chapeau
GITLAB-SSH.md	Création et installation de sa clef SSH dans gitlab niji
WSL2-Docker.MD	Installation de WSL2 et Docker
ZSH.md	Installation de ZSH dans WSL2
self-signed-ssl\README.MD	Création et prise en compte d'une clef ssh autosignée « docker.devhost »
docker-traefik-portainer\README.MD	Description de la stack docker-traefik-portainer-dnsmasq

# Lancement de la stack

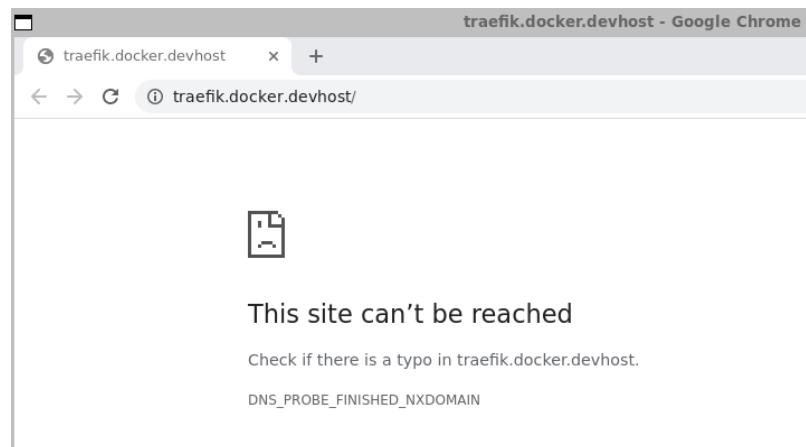
## Docker-compose

La création de la clef ssh autosignée n'est pas nécessaire. Celle en place est valide jusqu'en 2027

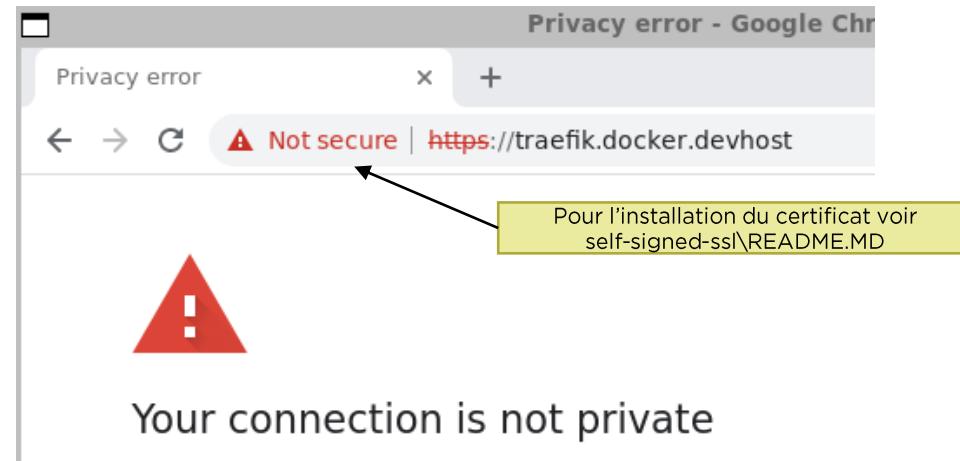
```
➤ cd docker-dev-host/docker-traefik-portainer  
➤ docker network create --driver bridge traefik-global-proxy  
➤ docker-compose up -d  
➤ google-chrome&
```

Installation de google-chrome : <https://learn.microsoft.com/fr-fr/windows/wsl/tutorials/gui-apps#install-google-chrome-for-linux>

Problème de DNS



Problème de certificat



Pour l'installation du certificat voir self-signed-ssl\README.MD

# Lancement de la stack

DNS + WSL2 + Windows = 😞

La gestion des DNS avec WSL2 et Windows est très pénible.

Normalement la configuration dnsmasq du docker-compose officiel permet de résoudre toutes les adresses en \*.docker.devhost vers 127.0.0.1

Malheureusement ça ne fonctionne pas toujours.

=> Si ça ne fonctionne pas chez vous, le plus simple est d'éditer le fichier /etc/hosts et d'ajouter la résolution dns manuellement !

Indique à WSL qu'on va gérer le fichier hosts nous même

➤ sudo vi /etc/wsl.conf

```
[network]
generateResolvConf = false
```

Ajout des résolutions DNS local

➤ sudo vi /etc/hosts

```
# This file was automatically generated by WSL. To stop automatic generation of this file,
# add the following entry to /etc/wsl.conf:
# [network]
# generateHosts = false
127.0.0.1    localhost
127.0.0.1    DFYB3X2.niji.fr DFYB3X2
127.0.0.1    traefik.docker.devhost
127.0.0.1    portainer.docker.devhost
127.0.0.1    nginx.docker.devhost
```

# Lancement de la stack

## Composition

```
▶ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
480b9d43bc4a traefik:2.9.10 "/entrypoint.sh trae..." About a minute ago Up About a minute 0.0.0.0:80->80/tcp, ::80->80/tcp, 0.0.0.0:443->443/tcp, ::443->443/tcp traefik
88b4b7dab68a portainer/portainer-ce:2.17.1-alpine "/portainer" About a minute ago Up About a minute 8000/tcp, 9000/tcp, 9443/tcp portainer
628ab5484968 4km3/dnsmasq "/usr/sbin/dnsmasq -..." About a minute ago Up About a minute 0.0.0.0:53->53/tcp, 0.0.0.0:53->53/udp dnsmasq
```

La stack contient trois applications:

- [Traefik](#) : routeur de trafic qui permet notamment d'accéder aux services via des noms de domaine
- [Portainer](#) : utilitaire de gestion des images, container, réseau, volume docker
- [DNSMasq](#) : utilitaire permettant de résoudre localement les noms de domaine \*.docker.devhost

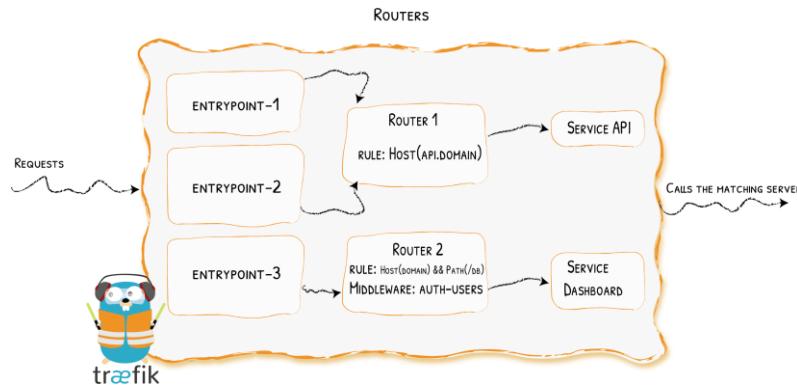
Seuls les ports 80 et 443 sont mappés l'accès aux interfaces se fait via les noms de domaine:

- traefik.docker.devhost
- portainer.docker.devhost

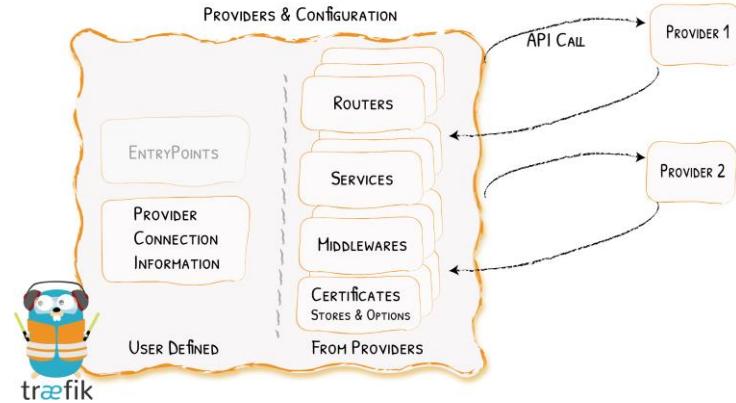
# Traefik

## Reverse-proxy

Traefik est un [router HTTP, TCP et UDP](#) permettant, une fois le fournisseur de services configurer, de [paramétrier automatiquement de nouvelles routes](#) lorsque de nouveaux services publient leur besoin de routage.



Le routage permet de renvoyer une [demande](#) vers le bon [service](#). Par exemple de gérer plusieurs noms de domaine pointant vers plusieurs site web sur la même machine.

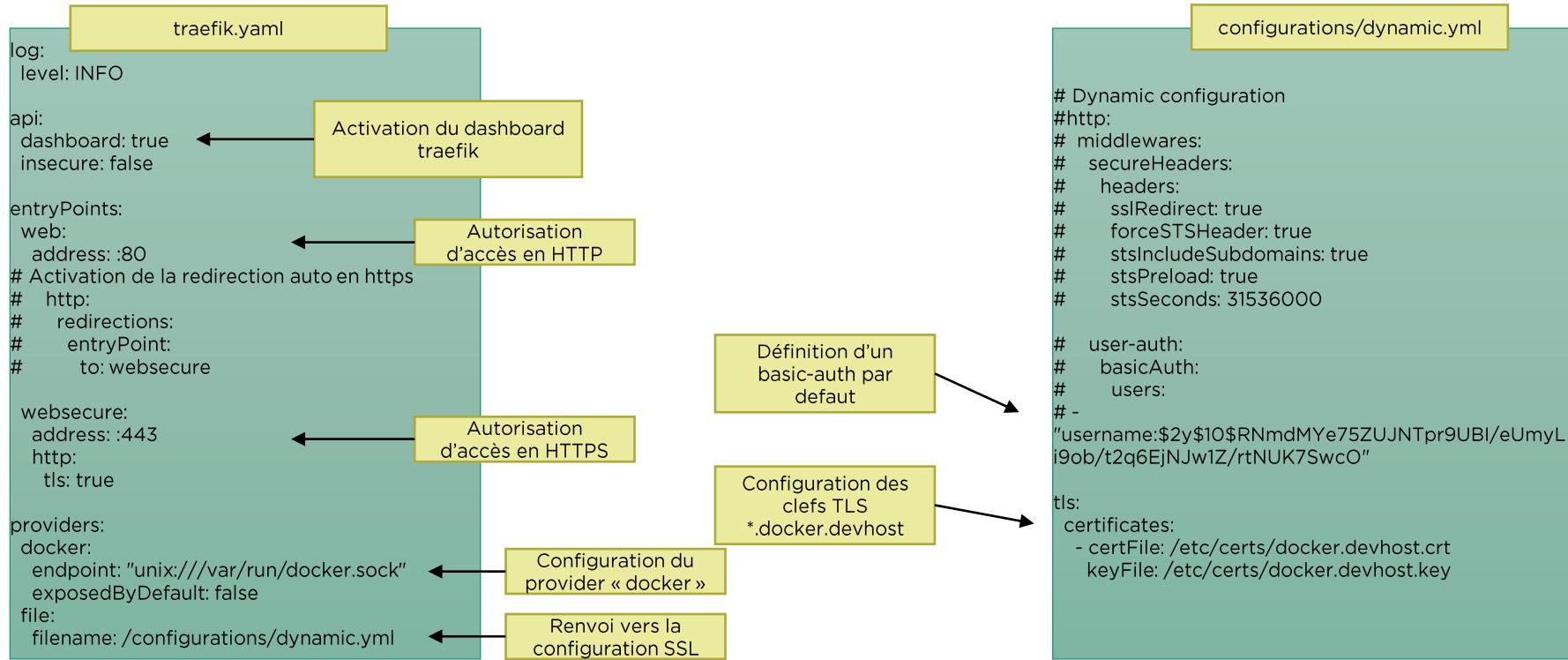


Traefik gère différents fournisseurs de services (docker, kubernetes, etc...) et découvre, quand le fournisseur est paramétré, automatiquement les nouvelles routes.

# Traefik

## Configuration : docker-traefik-portainer/traefik-data

La configuration faite par défaut est basique mais permet de gérer l'accès à des services « docker » en HTTP et HTTPS

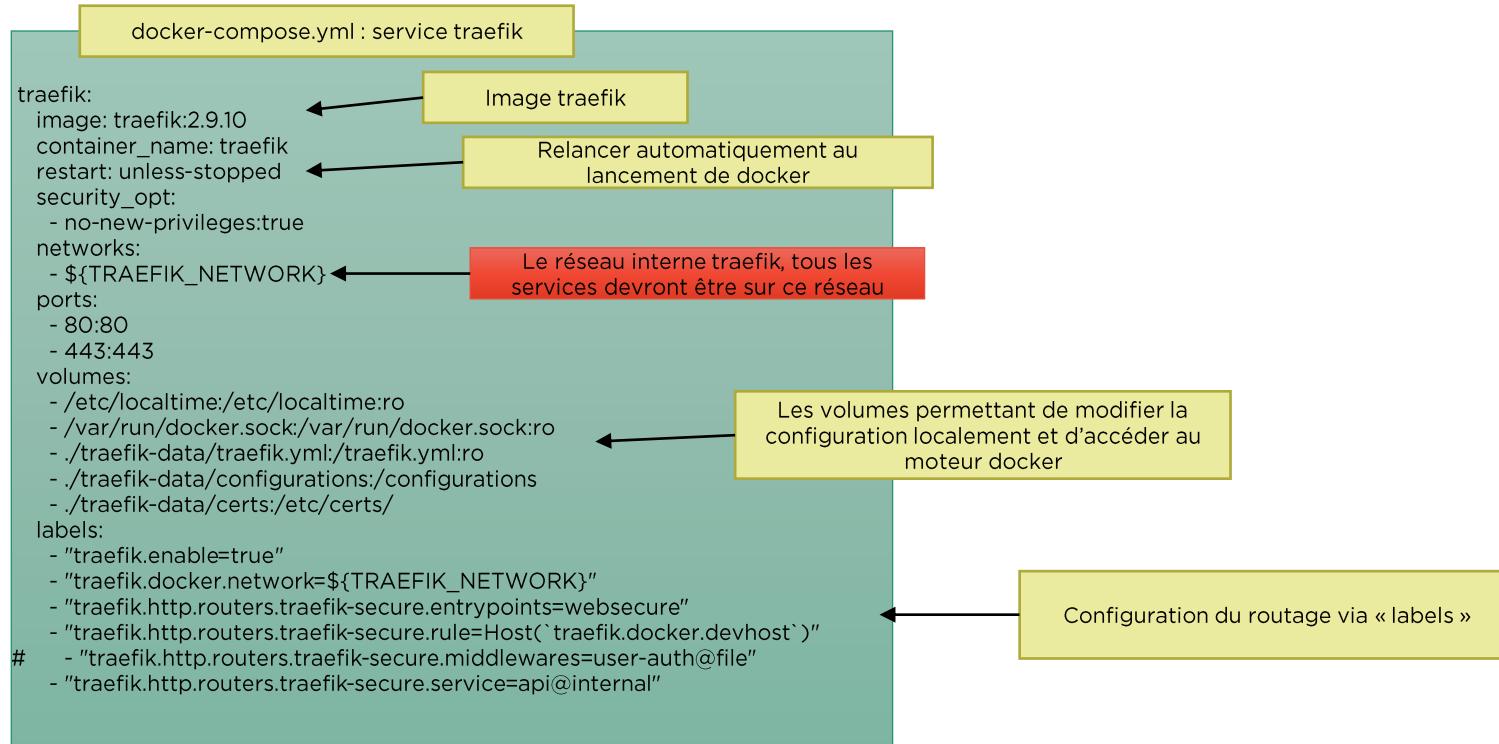


# Traefik

## Auto routage

La configuration de routage est réalisé côté fournisseur de service: docker ici.

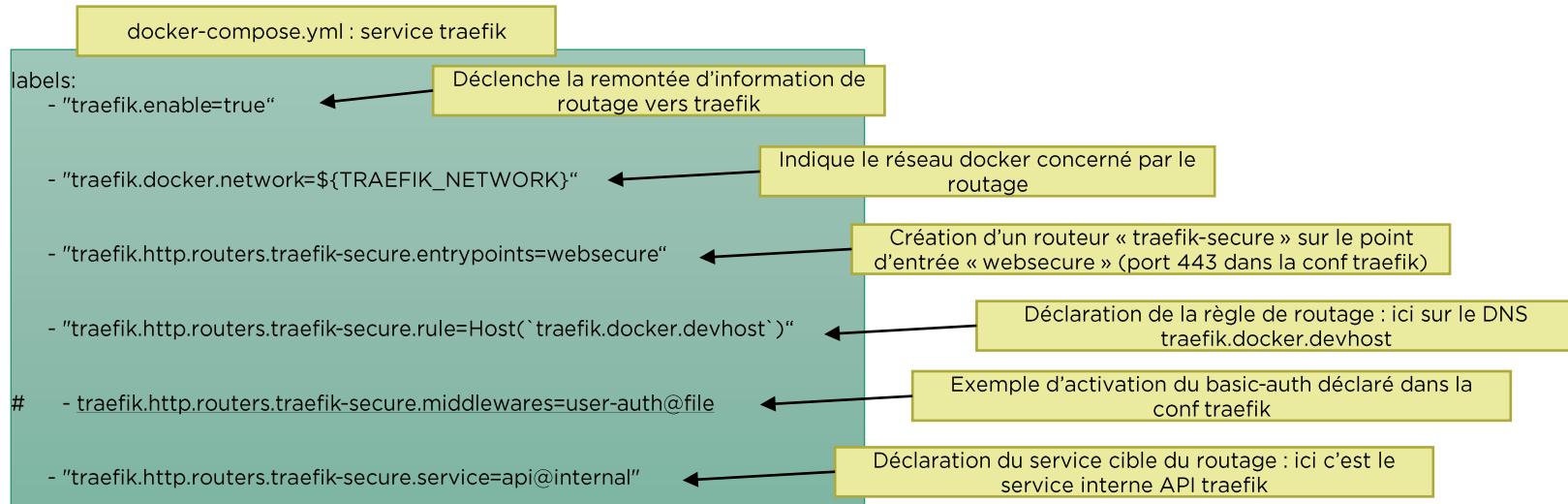
C'est dans le docker-compose qu'on va donc trouver les informations permettant l'accès aux services



# Traefik

## Auto routage : focus labels

Ici le routage permet à Traefik d'auto router sa propre interface web

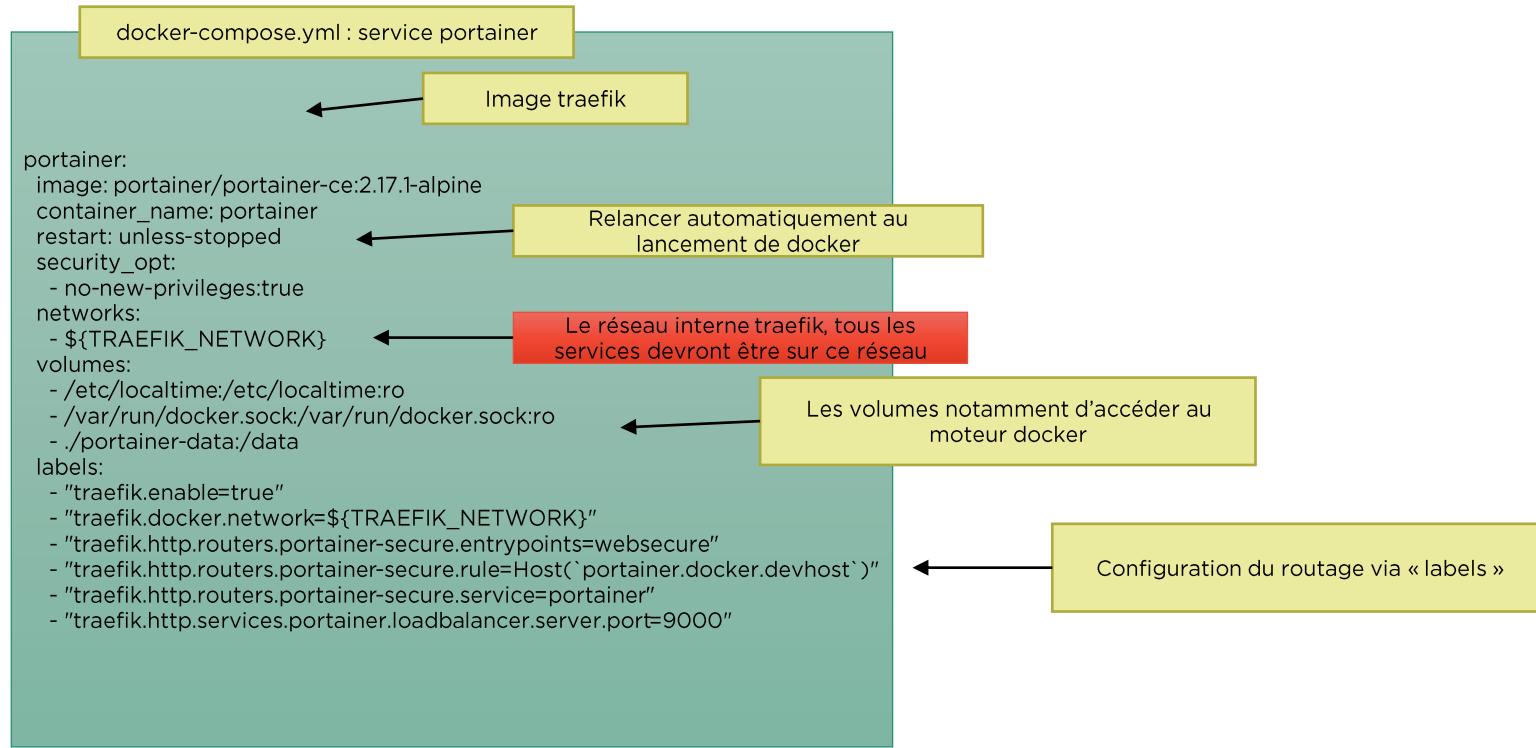


# Traefik

## Routage Portainer

La configuration de routage est réalisé côté fournisseur de service: docker ici.

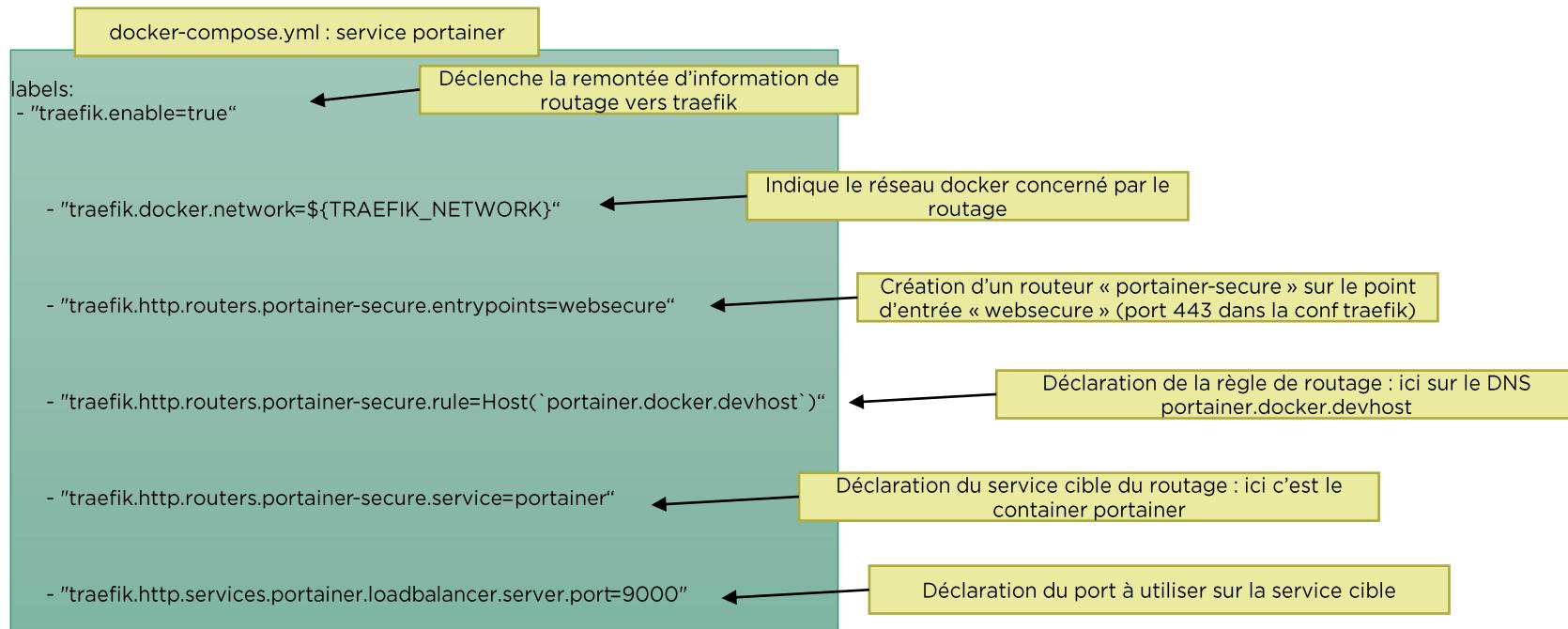
C'est dans le docker-compose qu'on va donc trouver les informations permettant l'accès aux services



# Traefik

## Routage Portainer : focus labels

Ici le routage permet à Traefik de router le DNS portainer vers le service docker lancé en interne sur le port 9000



# Traefik

## Routage applicatif

Une fois la stack lancée il est possible de lancer d'autres docker-compose et de profiter du routage traefik. Deux exemples sont disponibles sous : docker-dev-host/sample-app et docker-dev-host/sample-app-complex

docker-compose.yml : docker-dev-host/sample-app

```
version: "3"

services:
  nginx:
    image: nginx
    container_name: nginx
  networks:
    - traefik-global-proxy
  labels:
    - "traefik.enable=true"
    - "traefik.docker.network=traefik-global-proxy"
    - "traefik.http.routers.nginx-secure.entrypoints=websecure"
    - "traefik.http.routers.nginx-secure.rule=Host(`nginx.docker.devhost`)"
    - "traefik.http.routers.nginx-secure.service=nginx"
    - "traefik.http.services.nginx.loadbalancer.server.port=80"

networks:
  traefik-global-proxy:
    external: true
```

1. Déclenche la remontée d'information de routage vers traefik
2. Indique le réseau docker concerné par le routage
3. Création d'un routeur « nginx-secure » sur le point d'entrée « websecure » (port 443 dans la conf traefik)
4. Déclaration de la règle de routage : ici sur le DNS nginx.docker.devhost
5. Déclaration du service cible du routage : ici c'est le container nginx
6. Déclaration du port à utiliser sur le service cible

# Traefik

## Routage applicatif

docker-compose.yml : docker-dev-host/sample-app-complex

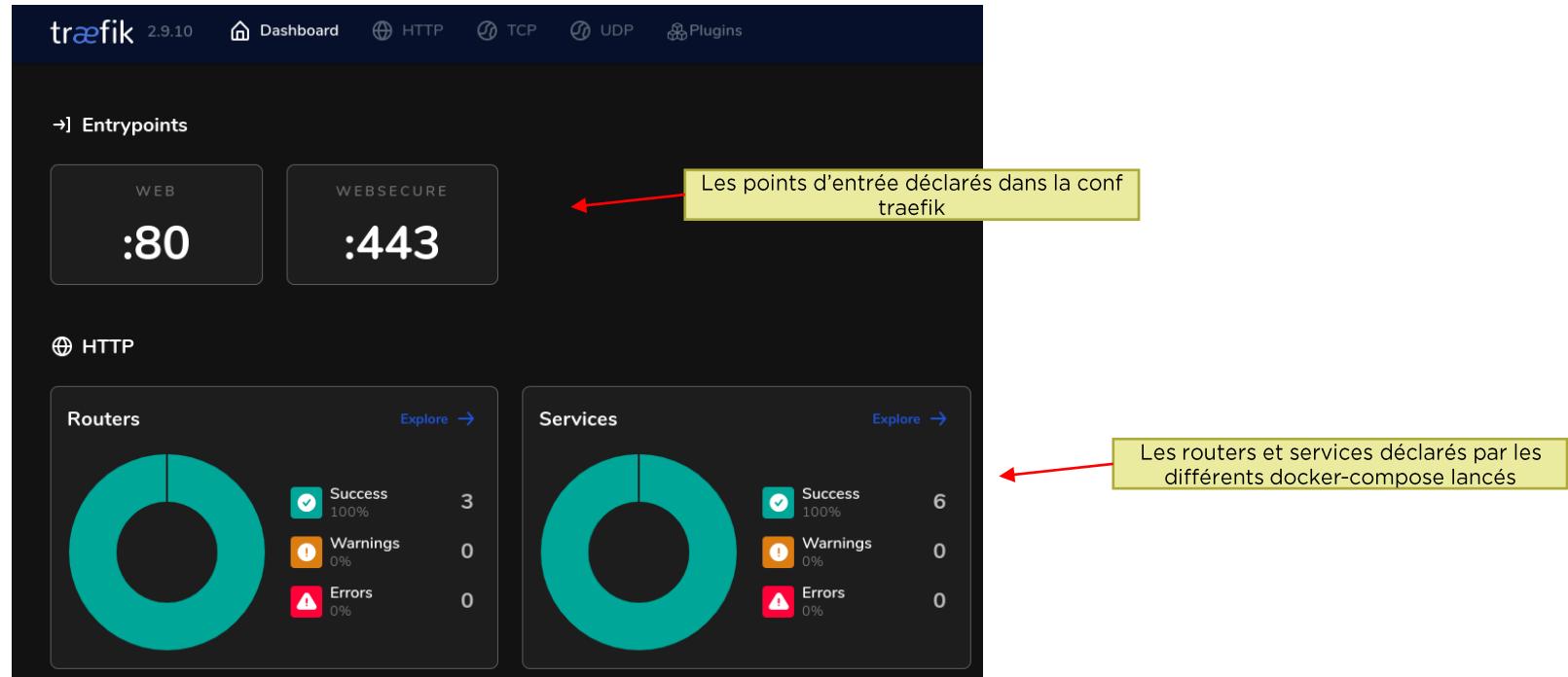
```
services:  
  
# Drupal  
cms:  
  image: drupal:8.3.7-apache  
  restart: always  
  container_name: cms  
  ports:  
    - 8080:80  
  depends_on:  
    - db  
  networks:  
    - ${TRAEFIK_NETWORK}  
    - back  
  labels:  
    - "traefik.enable=true"  
    - "traefik.docker.network=${TRAEFIK_NETWORK}"  
    - "traefik.http.routers.drupal-secure.entrypoints=websecure"  
    - "traefik.http.routers.drupal-secure.rule=Host(`${APP_DOMAIN}`)"  
    - "traefik.http.routers.drupal-secure.service=cms"  
    - "traefik.http.services.cms.loadbalancer.server.port=80"
```

```
# Postgres  
db:  
  image: postgres:9.6.5-alpine  
  restart: always  
  environment:  
    POSTGRES_DB: drupaldb  
    POSTGRES_USER: drupaluser  
    POSTGRES_PASSWORD: drupalpassword  
  ports:  
    - 5432:5432  
  networks:  
    - back  
networks:  
  traefik-global-proxy:  
    external: true  
back:
```

1. Déclenche la remontée d'information de routage vers traefik
2. Indique le réseau docker concerné par le routage
3. Création d'un routeur « apache-secure » sur le point d'entrée « websecure » (port 443 dans la conf traefik)
4. Déclaration de la règle de routage : ici sur le DNS sample-complex.docker.devhost (défini dans le .env)
5. Déclaration du service cible du routage : ici c'est le container apache
6. Déclaration du port à utiliser sur le service cible

# Traefik

Petit tour dans la dashboard : <https://traefik.docker.devhost/dashboard>



# Portainer

Petit tour dans la dashboard : <https://portainer.docker.devhost/>

The screenshot shows the Portainer Community Edition dashboard with the following details:

- Environment info:**
  - Environment: local (12 nodes, 5.2 GB - Standalone 20.10.16)
  - URL: /var/run/docker.sock
  - GPU: none
  - Tags: -
- Metrics:**
  - 13 Stacks
  - 231 Images (114.6 GB)
  - 19 Networks
  - 75 Containers
  - 52 Volumes
  - 0 GPUs

Annotations in French provide additional context:

- Yellow box over Environment info: "Les différents docker-compose lancés"
- Yellow box over Stacks: "Les images dans le registry local"
- Yellow box over Networks: "Les réseaux (utilisés ou non)"
- Yellow box over Containers: "Les conteneurs lancés (actifs ou inactifs)"
- Yellow box over Volumes: "Les volumes (utilisés ou non)"

# TP 4

## Création d'image



niji

# tp4.1 Dockerfile

## Couches/layers

Une image Docker est une **collection ordonnée de changements d'un système de fichiers et de paramètres d'exécution**.

Un ensemble atomique de changements sur le système de fichiers est **une couche**. Une image Docker peut être vue comme une **pile de couches** dont chaque couche dépend de toutes les précédentes.

Chaque couche représente donc un **ensemble de changement que l'on fait au système de fichiers de base**. Ces changements sont commis par les **instructions présentes dans le Dockerfile** de l'image. Lorsque l'on récupère une image Docker, elle se télécharge couche par couche.

```
> docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
a603fa5e3b41: Already exists
c39e1cda007e: Pull complete
90cefefba34d7: Pull complete
a38226fb7aba: Pull complete
62583498bae6: Pull complete
9802a2cfdb8d: Pull complete
Digest: sha256:e209ac2f37c70c1e0e9873a5f7231e91dcd83fdf1178d8ed36c2ec09974210ba
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

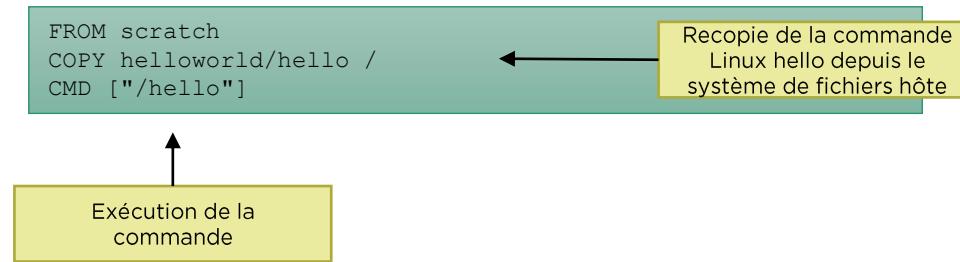
Ensemble des  
couches de l'image  
nginx



# tp4.2 Dockerfile

## Création d'une image from SCRATCH

Toutefois il existe une image SCRATCH permettant de partir « de rien ». Cette image est utilisée par les images de base de type « OS ». En partant de SCRATCH il n'y a aucune application ni librairie système présente. La seule option est donc de copier des binaires Linux compilés statiquement et donc parfaitement autonomes.



# Dockerfile

## Création d'une image

Un Dockerfile possède nécessairement:

- Une instruction FROM permettant de définir l'image de base de notre nouvelle image
- Une instruction CMD ou ENTRYPOINT permettant de définir la commande principale de l'image, commande qui sera lancée à l'exécution du conteneur

**Une image de base est une image apportant un ensemble de librairies et d'applications « standards » permettant d'avoir une souche applicative prête à l'emploi.**

Image de base de type « OS »

- Ubuntu : FROM ubuntu:22.04
- Alpine : FROM alpine:latest

....

Image de base de type « Application » elle-même dérivée d'une image de base de type « OS »

- Nginx : FROM nginx:latest
- Apache : FROM apache:latest
- Mysql : FROM mysql:latest

....

# tp4.2 Dockerfile

## Création d'une image from SCRATCH

La construction de l'image se fait via la commande « build »

```
> docker build . -t hello-demo
Sending build context to Docker daemon 9.216kB
Step 1/3 : FROM scratch
-->
Step 2/3 : COPY helloworld/hello /
--> 31eeb6e0d85b
Step 3/3 : CMD ["/hello"]
--> Running in 450589d9e741
Removing intermediate container 450589d9e741
--> 996a421d2030
Successfully built 996a421d2030
Successfully tagged hello-demo:latest
```

-t permet de tagger l'image créée

Création d'une couche « layer » docker

Lancement

```
> docker run hello-demo
Hi World
```

Le lancement de la commande principale est OK

```
> docker run hello-demo ls
docker: Error response from daemon: failed to create shim task: OCI runtime create failed: runc create failed: unable to start container process: exec: "ls": executable file not found in $PATH: unknown.
ERRO[0000] error waiting for container: context canceled
```

Par contre aucune autre commande n'est présente

# tp4.3 Dockerfile

## Création d'une image standard

La plupart du temps le FROM est une image existante, on parle d'image de base.

Lorsque le FROM possède une commande principale celle-ci peut être héritée, il n'est donc forcément nécessaire d'avoir une commande CMD/ENTRYPOINT dans son Dockerfile

Exemple d'un Dockerfile exposant un site statique, la commande principale est héritée de l'image « nginx »

```
FROM nginx
COPY html /usr/share/nginx/html
```

```
➤ docker build . -t tp4.3
...
➤ docker run -rm -d -p 80:80 tp4.3
...
2022/11/27 07:17:09 [notice] 1#1: nginx/1.23.2
...
2022/11/27 07:17:09 [notice] 1#1: start worker processes
```

```
➤ curl http://127.0.0.1
<!DOCTYPE html>
<html>
  <head>
    <title>Basic Web Page</title>
  </head>
  <body>
    Hello World!
  </body>
</html>
```

# tp4.4 Dockerfile

## Exploration d'une image

Deux outils permettent de d'inspecter une image Docker :

- docker inspect : donner une description du paramétrage d'une image
  - Variables d'environnement déclarées
  - Commande principale
  - Volume
  - Couches
  - ...
- Dive : qui permet de voir les couches de construction de l'image et les étapes de création

# tp4.4 Dockerfile

## Exploration d'une image

➤ docker inspect tp4.3

```
[  
 {  
   "Id": "sha256:27307c655f14157ff82281b1a8223c1530ae8c61b66cba3b2ba9456f8dd2f7c2",  
   "RepoTags": [  
     "tp4.3:latest"  
   ],  
   "RepoDigests": [],  
   "Parent":  
     "sha256:88736fe827391462a4db99252117f136b2b25d1d31719006326a437bb40cb12d",  
   "Comment": "",  
   "Created": "2022-11-27T07:16:15.116593Z",  
   "Container": "",  
   "ContainerConfig": {  
     "Hostname": "",  
     "Domainname": "",  
     "User": "",  
     "AttachStdin": false,  
     "AttachStdout": false,  
     "AttachStderr": false,  
     "ExposedPorts": {  
       "80/tcp": {}  
     },  
     ...  
   },  
 }
```

```
➤ docker run --rm -it \  
➤ -v /var/run/docker.sock:/var/run/docker.sock \  
➤ wagoodman/dive:latest tp4.3
```

Layers	Size	Command	Current Layer Contents
1	0 B	FROM sha256:kff1933fbac740	bin bash cat chgrp chmod chown cp dash date dd df dir du domainname = hostname echo egrep false fgrep fmt grep gunzip gtrue gzip hostname ln ls lsblk mkdir mknode mktemp more mount mountpoint mv nisdomainname = hostname pidof = /sbin/killall ps ps aux rm rmdir run-parts
2	1.6 kB	SET -e && addgroup --system --gid 101 nginx && addu -nrxr-xr-x	-nrxr-xr-x 0:0 1.2 kB
2.1 kB	#(nop) COPY file:7b30b762e02255f04dc9012421a3009bf9abf3685	-nrxr-xr-x 0:0 44 kB	
2.1 kB	#(nop) COPY file:5c1827233434988bd0e94ec0d3d82c872c1a0a435c	-nrxr-xr-x 0:0 73 kB	
2.1 kB	#(nop) COPY file:abcbcf4d4c17ee4454b0b2e3cf914be8e002f48d4	-nrxr-xr-x 0:0 64 kB	
2.1 kB	#(nop) COPY file:e57ef017a414ca793499729d80a7b975796c9a80	-nrxr-xr-x 0:0 73 kB	
2.1 kB	#(nop) COPY dir:c3e98637bc70bb1dc37b8dd9b1dd6b65f90df1bd228	-nrxr-xr-x 0:0 151 kB	
2.1 kB	#(nop) COPY file:7b30b762e02255f04dc9012421a3009bf9abf3685	-nrxr-xr-x 0:0 126 kB	
2.1 kB	#(nop) COPY file:5c1827233434988bd0e94ec0d3d82c872c1a0a435c	-nrxr-xr-x 0:0 114 kB	
2.1 kB	#(nop) COPY file:abcbcf4d4c17ee4454b0b2e3cf914be8e002f48d4	-nrxr-xr-x 0:0 81 kB	
2.1 kB	#(nop) COPY file:e57ef017a414ca793499729d80a7b975796c9a80	-nrxr-xr-x 0:0 94 kB	
2.1 kB	#(nop) COPY dir:c3e98637bc70bb1dc37b8dd9b1dd6b65f90df1bd228	-nrxr-xr-x 0:0 147 kB	
2.1 kB	#(nop) COPY file:7b30b762e02255f04dc9012421a3009bf9abf3685	-nrxr-xr-x 0:0 85 kB	
2.1 kB	#(nop) COPY file:5c1827233434988bd0e94ec0d3d82c872c1a0a435c	-nrxr-xr-x 0:0 0 B	
2.1 kB	#(nop) COPY file:abcbcf4d4c17ee4454b0b2e3cf914be8e002f48d4	-nrxr-xr-x 0:0 40 kB	
2.1 kB	#(nop) COPY file:e57ef017a414ca793499729d80a7b975796c9a80	-nrxr-xr-x 0:0 28 kB	
2.1 kB	#(nop) COPY dir:c3e98637bc70bb1dc37b8dd9b1dd6b65f90df1bd228	-nrxr-xr-x 0:0 40 kB	
2.1 kB	#(nop) COPY file:7b30b762e02255f04dc9012421a3009bf9abf3685	-nrxr-xr-x 0:0 28 kB	
2.1 kB	#(nop) COPY file:5c1827233434988bd0e94ec0d3d82c872c1a0a435c	-nrxr-xr-x 0:0 60 kB	
2.1 kB	#(nop) COPY file:abcbcf4d4c17ee4454b0b2e3cf914be8e002f48d4	-nrxr-xr-x 0:0 203 kB	
2.1 kB	#(nop) COPY file:e57ef017a414ca793499729d80a7b975796c9a80	-nrxr-xr-x 0:0 2.3 kB	
2.1 kB	#(nop) COPY dir:c3e98637bc70bb1dc37b8dd9b1dd6b65f90df1bd228	-nrxr-xr-x 0:0 6.4 kB	
2.1 kB	#(nop) COPY file:7b30b762e02255f04dc9012421a3009bf9abf3685	-nrxr-xr-x 0:0 98 kB	
2.1 kB	#(nop) COPY file:5c1827233434988bd0e94ec0d3d82c872c1a0a435c	-nrxr-xr-x 0:0 23 kB	
2.1 kB	#(nop) COPY file:abcbcf4d4c17ee4454b0b2e3cf914be8e002f48d4	-nrxr-xr-x 0:0 73 kB	
2.1 kB	#(nop) COPY file:e57ef017a414ca793499729d80a7b975796c9a80	-nrxr-xr-x 0:0 57 kB	
2.1 kB	#(nop) COPY dir:c3e98637bc70bb1dc37b8dd9b1dd6b65f90df1bd228	-nrxr-xr-x 0:0 147 kB	
2.1 kB	#(nop) COPY file:7b30b762e02255f04dc9012421a3009bf9abf3685	-nrxr-xr-x 0:0 150 kB	
2.1 kB	#(nop) COPY file:5c1827233434988bd0e94ec0d3d82c872c1a0a435c	-nrxr-xr-x 0:0 85 kB	
2.1 kB	#(nop) COPY file:abcbcf4d4c17ee4454b0b2e3cf914be8e002f48d4	-nrxr-xr-x 0:0 77 kB	
2.1 kB	#(nop) COPY file:e57ef017a414ca793499729d80a7b975796c9a80	-nrxr-xr-x 0:0 40 kB	
2.1 kB	#(nop) COPY dir:c3e98637bc70bb1dc37b8dd9b1dd6b65f90df1bd228	-nrxr-xr-x 0:0 60 kB	
2.1 kB	#(nop) COPY file:7b30b762e02255f04dc9012421a3009bf9abf3685	-nrxr-xr-x 0:0 56 kB	
2.1 kB	#(nop) COPY file:5c1827233434988bd0e94ec0d3d82c872c1a0a435c	-nrxr-xr-x 0:0 19 kB	
2.1 kB	#(nop) COPY file:abcbcf4d4c17ee4454b0b2e3cf914be8e002f48d4	-nrxr-xr-x 0:0 147 kB	
2.1 kB	#(nop) COPY file:e57ef017a414ca793499729d80a7b975796c9a80	-nrxr-xr-x 0:0 0 B	
2.1 kB	#(nop) COPY dir:c3e98637bc70bb1dc37b8dd9b1dd6b65f90df1bd228	-nrxr-xr-x 0:0 0 B	
2.1 kB	#(nop) COPY file:7b30b762e02255f04dc9012421a3009bf9abf3685	-nrxr-xr-x 0:0 40 kB	
2.1 kB	#(nop) COPY file:5c1827233434988bd0e94ec0d3d82c872c1a0a435c	-nrxr-xr-x 0:0 0 B	
2.1 kB	#(nop) COPY file:abcbcf4d4c17ee4454b0b2e3cf914be8e002f48d4	-nrxr-xr-x 0:0 52 kB	
2.1 kB	#(nop) COPY file:e57ef017a414ca793499729d80a7b975796c9a80	-nrxr-xr-x 0:0 28 kB	

# Dockerfile

## Création d'une image

Les principales commandes permettant la construction d'une image sont :

- **FROM** : Donne l'image de base
- **COPY** et **ADD** : recopie un ou plusieurs fichiers dans système de fichiers de l'image
- **ENV** : déclare une variable d'environnement utilisée pour les instructions suivantes et dans les conteneurs instanciés
- **RUN** : exécute une commande lors de la création de l'image
- **VOLUME** : définit un volume qui sera activé à l'instanciation de l'image en conteneur
- **EXPOSE** : définit un port exposé par le conteneur lors de son instanciation
- **USER** : déclare l'utilisateur qui sera utilisé à l'instanciation du conteneur
- **WORKDIR** : déclare le répertoire de travail
- **CMD** et **ENTRYPOINT** : définit la commande principale du conteneur

Il doit toujours y a voir un FROM et un CMD ou ENTRYPOINT (éventuellement hérité)

# tp4.5 Dockerfile

## Création d'une image

Chacune de ces commandes

- Crètent une couche lors de la construction d'une image si le système de fichiers est modifié (COPY, ADD, RUN)
- Ajoute une metadata à l'image afin de définir son contexte d'exécution (ENV, WORKDIR, VOLUME, EXPOSE)

Il est recommandé de limiter les couches afin:

- D'avoir une image moins lourde
- De bénéficier des caches docker lors de la construction d'image

```
FROM alpine  
  
RUN apk update  
  
RUN apk add wget  
  
RUN apk add apache2  
  
RUN apk add php  
  
CMD echo "ras"
```

```
➤ docker build . -f ./DokerfileXXX  
➤ docker inspect XXXXXX | grep Size
```

**25 177 432**  
**4 layers**

```
FROM alpine  
  
RUN apk update && apk add wget \  
 && apk add apache2 && apk add php  
  
CMD echo "ras"
```

**25 110 907**  
**1 layer**

# tp4.6 Dockerfile

## Création d'une image : optimisation

L'ordre des instructions dans le Dockerfile détermine l'empilement des couches (layers).

Pour accélérer la construction des images, Docker implémente un mécanisme de gestion de cache des différentes couches. Il calcule un checksum des fichiers et metadonnées impliquées par l'instruction en cours. Si ce checksum correspond à un layer déjà présent dans le cache local Docker il est ajouté à la construction sinon le layer est reconstruit en exécutant les instructions.

L'utilisation de layer en cache n'est plus possible à partir du moment où une instruction conduit à la construction effective d'un layer.

L'ordre et la constitution des instructions dans le Dockerfile conditionnent donc fortement son temps d'exécution lors l'environnement dispose de cache.

# tp4.6 Dockerfile

## Création d'une image : sans optimisation

Création d'une image sans cache: création des toutes les couches : 40 s

```
> docker build . -f Dockerfile.basic
Sending build context to Docker daemon 126.5kB
Step 1/5 : FROM maven:3.9.3-eclipse-temurin
--> 583511ed23c0
Step 2/5 : ADD demo .
--> cb489f6e7994
Step 3/5 : RUN mvn clean package -DskipTests -Dcheckstyle.skip=true -Dmnd.skip=true
--> Running in 2cd86601d049
[INFO] Scanning for projects...
Downloading from central...
Removing intermediate container 2cd86601d049
--> 3e0d99800313
Step 4/5 : RUN mv target/*.jar /app.jar
--> Running in e30286fe3a4b
Removing intermediate container e30286fe3a4b
--> 8f41eacdea33
Step 5/5 : ENTRYPOINT ["sh","-c","java -jar /app.jar"]
--> Running in 76e9d95d4880
Removing intermediate container 76e9d95d4880
--> 1e82fa0af993
Successfully built 1e82fa0af993
```

Ajout des fichiers projets et création d'une couche « layer » docker

Exécution d'une commande

Pas de cache disponible, lancement d'un conteneur à partir du dernier layer et exécution de la commande dans le conteneur

Une fois la commande exécutée, suppression du conteneur temporaire et création du nouveau layer à faire du système de fichiers obtenu

Même chose que précédemment

Même chose que précédemment

# tp4.6 Dockerfile

## Création d'une image : sans optimisation

Création d'une image avec cache, temps d'exécution : 7s

```
> docker build . -f Dockerfile.basic
Sending build context to Docker daemon 127kB
Step 1/5 : FROM maven:3.9.3-eclipse-temurin-17
--> 583511ed23c0
Step 2/5 : ADD demo .
--> Using cache
--> cb489f6e7994
Step 3/5 : RUN mvn clean package -DskipTests -Dcheckstyle.skip=true -Dpmd.skip=true
--> Using cache
--> 3e0d99800313
Step 4/5 : RUN mv target/*.jar /app.jar
--> Using cache
--> 8f41eacdea33
Step 5/5 : ENTRYPOINT ["sh","-c","java -jar /app.jar"]
--> Using cache
--> 1e82fa0af993
Successfully built 1e82fa0af993
```

Ajout des fichiers projets,  
ils n'ont pas  
changé, utilisation du  
cache

Exécution d'une commande déjà  
exécutée sur le layer précédent,  
utilisation du cache

Même chose que  
précédemment

Même chose que  
précédemment

# tp4.6 Dockerfile

## Création d'une image : avec optimisation

### Version non optimale

```
FROM maven:3.9.3-eclipse-temurin-17
ADD demo .
RUN mvn clean package -DskipTests -Dcheckstyle.skip=true -
Dpmd.skip=true
RUN mv target/*.jar /app.jar
ENTRYPOINT ["sh","-c","java -jar /app.jar"]
```

Recopie de tous les fichiers dans le conteneur, le moindre changement, même sur un fichier inutile pour le packaging, entraînera une invalidation du cache

Packaging applicatif comprenant la récupération des dépendances et la compilation. La modification d'un fichier source entraînera l'invalidation des caches et une nouvelle récupération des dépendances.

### Version optimisée

```
FROM maven:3.9.3-eclipse-temurin-17
ENV HOME=/usr/app
RUN mkdir -p $HOME
WORKDIR $HOME
ADD demo/pom.xml .
RUN mvn dependency:copy-dependencies
ADD demo/src ./src
RUN mvn clean package -DskipTests -Dcheckstyle.skip=true -Dpmd.skip=true
RUN mv target/*.jar /app.jar
ENTRYPOINT ["sh","-c","java -jar /app.jar"]
```

Recopie du fichier de gestion des dépendances

Récupération des dépendances dans une couche dédiée, les modifications de code n'auront pas d'impact sur cette couche

Recopie des sources uniquement nécessaires au packaging

Construction applicative s'appuyant sur les dépendances déjà présentes

# tp4.7 Dockerfile

## Création d'une image en plusieurs étapes : multistage

Les images docker sont particulièrement utiles dans les **pipeline d'intégration continue** qui doivent exécuter des **tâches éphémères avec des outils très différents**. Si le projet déploie son applicatif sous la forme d'une ou plusieurs images docker leur construction se déroule en général en deux étapes :

- La phase de **construction applicative** qui nécessite souvent des outils de compilation, des bibliothèques additionnelles. Cette phase conduit à la création d'un **artefact applicatif** (binaire, tar.gz, .jar, ...)
- La phase de **construction de l'image de « run »** qui sera concrètement utilisée lors du déploiement applicatif. Cette image doit être la plus petite possible afin de limiter les vulnérabilités, idéalement elle ne contient que **l'artefact applicatif** et son **environnement d'exécution**.

Ces deux phases de construction peuvent être portées par **un unique Dockerfile ayant plusieurs FROM**.

Ce Dockerfile « **multistage** » produit des images éphémères et une image finale.

L'avantage est de pouvoir recopier facilement les fichiers d'une image précédente vers l'image en cours.

Notamment cela **facilite la recopie de l'artefact applicatif** dans l'image de « run ».

# tp4.7 Dockerfile

Création d'une image en plusieurs étapes : multistage

BUILD Phase

```
# Stage 1: Build an Angular Docker Image  
FROM node as build
```

Utilisation d'une image node afin d'avoir accès aux commandes de packaging applicatif. L'image courante est nommée « build »

```
WORKDIR /app  
COPY package*.json /app/  
RUN npm install  
COPY . /app  
ARG configuration=production  
RUN npm run build -- --outputPath=/dist/out --configuration $configuration
```

Recopie du package\*.json pour pouvoir charger les dépendances

Récupération des dépendances dans un couche spécifique. Elle pourra être mise en cache et ne pas être rejouée lors d'un prochain lancement

Création de l'artefact applicatif

RUN Phase

```
# Stage 2, use the compiled app, ready for production with Nginx  
FROM nginx
```

Seconde étape avec une image nginx qui suffisante pour servir l'application « statique » angular

```
COPY --from=build /app/dist/out/ /usr/share/nginx/html  
COPY /nginx-custom.conf /etc/nginx/conf.d/default.conf
```

Recopie de l'artefact applicatif de l'image « build » vers l'image en cours de construction

Chargement d'un fichier de configuration spécifique. Ici pas de CMD, il est hérité de l'image « nginx »

# tp4.7 Dockerfile

Création d'une image en plusieurs étapes : multistage

RUN Phase      BUILD Phase

```
> docker build ./angularSample -f ./Dockerfile -t tp47
Sending build context to Docker daemon 302.7MB
Step 1/10 : FROM node as build
--> eae085782a
Step 2/10 : WORKDIR /app
...
Step 7/10 : RUN npm run build -- --configuration $configuration
--> Using cache
--> 04a43e4ef7da
Step 8/10 : FROM nginx
--> 88736fe82739
Step 9/10 : COPY --from=build /app/dist/* /usr/share/nginx/html
--> b77b51433357
Step 10/10 : COPY nginx-custom.conf /etc/nginx/conf.d/default.conf
--> 3e1b54335fb0
Successfully built 3e1b54335fb0
Successfully tagged tp47:latest

> docker run --rm -p 80:80 tp47
```

# Merci.



niji