ESI - DOCKER

Baptiste Bauer

Version v0.0.0.sip-230426121932, 2023-04-26 12:13:01

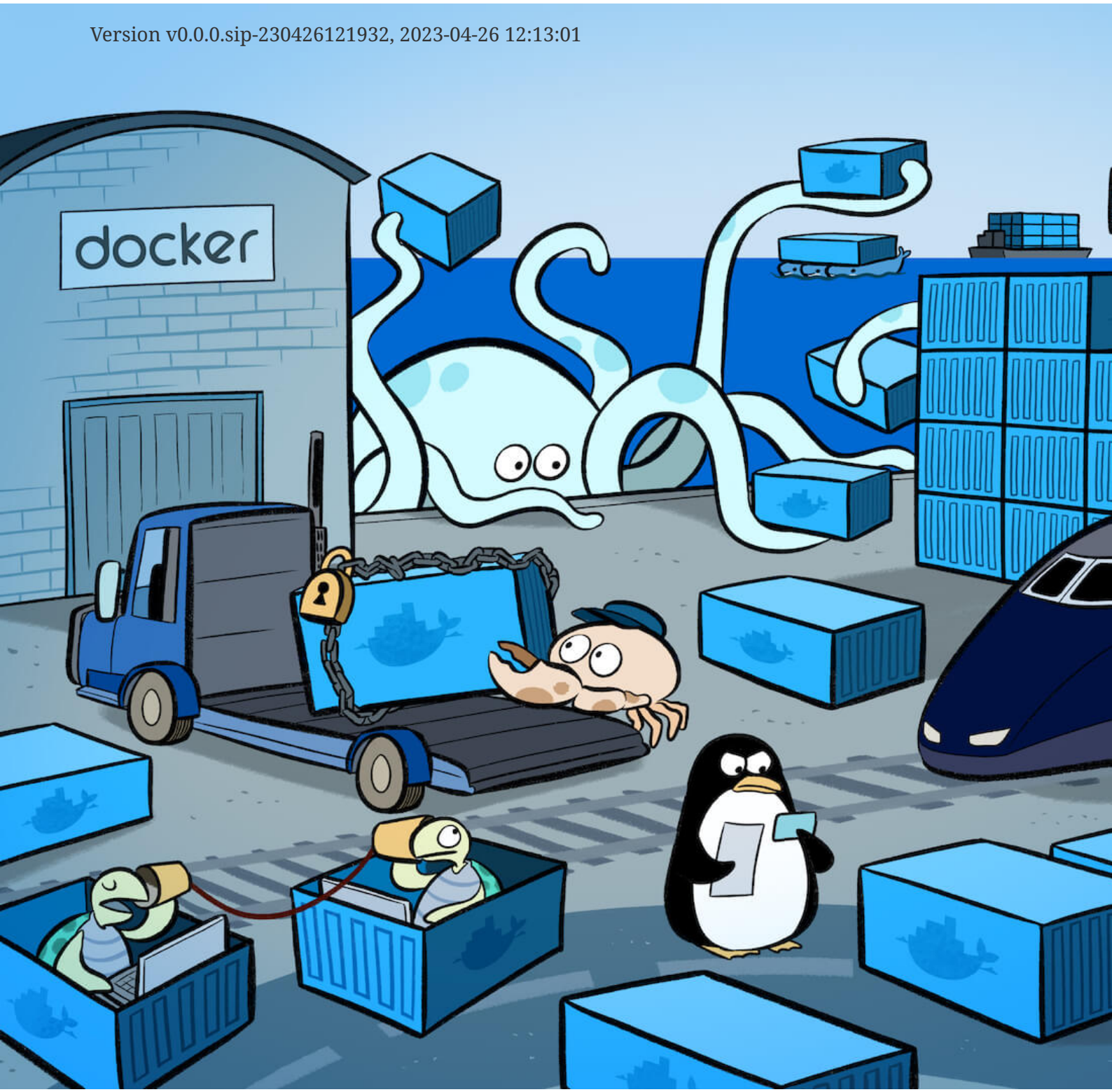

Table des matières

1. Introduction

Ce cours est découpé en différents chapitres et permet un apprentissage progressif des différents concepts Docker et de leur mise en pratique. On commencera par donner quelques exemples de ce qu'il est possible de faire avec Docker dans la section **Pour qui ? pourquoi ?**. Nous ferons références à des concepts utiles comme les **containers** Linux, les Micro services, le Dev Ops.etc.

Nous aurons un chapitre sur la **plateforme Docker**, son architecture, son fonctionnement et sa mise en place. Nous verrons comment Docker rend très simple la manipulation des **containeurs**. Nous parlerons de la notion d'images qui permet de **packager** une application et ses dépendances. Dans le chapitre sur le stockage, nous apprendrons à utiliser **Docker** pour que les données puissent persister dans les conteneurs.

Thèmes abordé dans ce cours :

- **Docker Machine** pour créer des hôtes Docker.
- **Docker compose** qui permet de créer des applications en multi container.
- **Docker Swarm**, la solution d'orchestration de **Docker** qui permet de gérer des applications qui tournent dans des containers.
- Le **réseau** dans Docker.
- La **sécurité**.

1.1. Pour qui ? pourquoi ?

Très souvent le premier contact que l'on a avec **Docker** s'effectue via le **Docker Hub** accessible sur https://hub.docker.com.

[Docker Hub]

Il s'agit d'un **registre** (ou **registry**) dans lequel nous retrouvons beaucoup d'applications packagées dans des images **Docker**. Cette notion d'image est la base de ce qu'apporte **Docker**. Voici un exemple de services qui peuvent être contenu dans une image **Docker** :

[image]

Par exemple, grâce à **Docker** nous pouvons lancer un interpréteur interactif (**REPL**) pour des langages de programmation comme le **Python**, le **Ruby On Rail** ou le **Javascript**.

[image]

Nous avons alors accès à un environnement **Python** en interactif et c'est le flag `-ti` qui permet l'interactivité avec le processus du containeur.

De la même manière, nous pouvons lancer un environnement **NodeJs**, ici contenant le Tag `8.12-alpine`. **8.12** est la version de **NodeJs** et **alpine** est le nom de la distribution Linux utilisée dans le container.

Par exemple si nous avons besoin d'une base de données **MongoDB** dans la version 4.0.Nous

n'avons qu'à trouver une image disponible dans le **Docker Hub**.

[image]

On peut imaginer avoir besoin de lancer plusieurs containers **MongoDB** avec des versions différentes. Cela peut être utile pour tester une différence de comportement entre deux versions par exemple.

1.2. Des Stacks complètes

Une application fonctionne rarement seule et est souvent constituée d'un ensemble de services. Cet ensemble constitue une **Stack applicative**. Par exemple, prenons le cas de la Stack **Elastic**, qui est souvent utilisée pour la gestion des log. Elle est constituée de **BEATS** et **LOGSTASH** qui est là pour l'ingestion des logs, de **ELASTICSEARCH** pour l'analyse et le stockage des logs et **KIBANA** qui permet de visualiser tout cela.

[image]

Il existe une multitude d'applications prêtent à être utilisée avec Docker, accessible en ligne de commande. Nous verrons rapidement comment Docker permet de créer notre propre package d'application pour faciliter : l'installation, l'utilisation et le déploiement.

1.3. Quelques concepts utiles pour les Développeurs

1.3.1. Un container Linux, c'est quoi ?

Un **container** est simplement un **processus** particulier qui tourne sur le système. Il est isolé des autres **processus**. Il possède sa **propre vision** du système sur lequel il tourne, on appelle cela les **Namespaces**. On peut limiter les ressources utilisées par ce processus en utilisant les **Controls Groups** (ou **Cgroups**). Le même système peut exécuter plusieurs containers en même temps, c'est d'ailleurs ce qui constitue l'avantage de cette technologie. Le noyau Linux de la machine hôte est **partagé** entre tous ses containeurs.

1.3.2. Containers Linux : Les Namespaces

Les **Namespaces** sont des technologies Linux qui servent à isoler un processus. Cela permet de limiter ce qu'un processus peut voir. Il existe **6** NameSpaces différents :

1. **Pid** : Permet de donner à un processus la vision de lui-même et de ses processus enfant.
2. **Net** : Permet de donner au processus son propre réseau privé.
3. **Mount** : Permet de donner au processus un système de fichiers privé.
4. **Uts** : Permet la gestion du nom de l'hôte.
5. **Ipc** : Isole les communications inter processus.
6. **User** : permet de faire un mapping entre les utilisateurs de l'hôte et les containeurs.

1.3.3. Containers Linux : Control Groups (cgroups)

Les **cgroups** sont une autre technologie Linux qui va permettre de limiter les ressources qu'un processus va utiliser. Par exemple, pour limiter l'utilisation :

- **RAM**
- **CPU**
- des **I/O** (périphériques d'entrées et de sorties)
- du **Réseau**

1.3.4. Containers Linux : VM/Container

[image]

On compare souvent les containers à des machines virtuelles, car elles permettent d'exécuter des applications de manière isolée.

Mais la virtualisation nécessite un **hyperviseur** qui s'exécute **sur le système d'exploitation de l'hôte** et nécessite également que **chaque machine virtuelle** ait son propre système d'exploitation. Alors que l'approche du container est **beaucoup plus légère** car chacun partage le **Kernel Linux de la machine hôte**.

La machine virtuelle consomme plus de disque mémoire et de ram que les containers. **Cela implique que beaucoup plus de containers peuvent fonctionner sur une même machine hôte.**

1.3.5. Architecture micro-services

Depuis quelques années, les applications sont développées autour d'une architecture appelée **micro-services**. Alors qu'avant une application était souvent un gros bloc unique **monolithique**.

[image]

Aujourd'hui, une application est constituée de **plusieurs petits composants** qui sont des services qui ont leur propre rôle et fonctionnalité. Et c'est l'**interconnexion** de l'ensemble de ces services qui permettent de définir l'application globale.

[image]

Dans une **application monolithique**, si l'on veut que plusieurs instances de l'application soient déployées il faut créer plusieurs machines virtuelles contenant l'application dans son entièreté.

[image]

Alors que dans le contexte d'une application micro-services chaque service peut être déployé indépendamment des autres services, nous avons plusieurs machines virtuelles sur lesquelles les services des différentes applications sont dispatchées.

Exemple d'architecture micro-services : l'application UBER

[image]

Chaque processus métier est isolé dans un service :

- **Paiement**
- **Notification**
- **Facturation**

Avantages de l'architecture micro-services :

- **Découpage** de l'application en **processus** (services) indépendants.
- Chacun a sa propre **responsabilité métier**.
- **Equipe dédiée** pour chaque service.
- Plus de **liberté** de choix de langage.
- **Mise à jour**.
- Containers très adaptés pour les micro-services.

Inconvénients :

- Nécessite des interfaces bien définies.
- Focus sur les tests d'intégration.
- Déplace la complexité dans l'orchestration de l'application globale. (Docker SWARM ou Kubernetes).

APPLICATION CLOUD NATIVE

On entend de plus en plus parler d'applications **Cloud Native** définies par plusieurs critères :

- Applications qui suivent une architecture **microservices**.
- Utilisant la **technologie des containers**.
- L'orchestration est faite **dynamiquement**.

Il existe une branche de la **Linux Foundation** : la **CNCF** (**C** loud **N** ative **C** omputing **F** oundation) qui porte de nombreux projets **Cloud Native** comme :

- **Kubernetes**
- **Prometheus**
- **Fluentd**

Site de la cncf

1.3.6. Questionnaire de synthèse

1. **Quels sont les éléments permettant la création d'un container sous Linux ?**
    - Le kernel Linux et le système de fichiers.
    - Les namespaces et les control groups.

- Les control groups et le système de fichiers.

2. **Les cgroups permettent :**
   - De limiter la vision d'un processus
   - De limiter les ressources que peux utiliser un processus
   - D'isoler le système de fichiers d'un processus
   - De faire un chroot
3. **Un container c'est**
   - Une mini machine virtuelle
   - Un répertoire sur le système de fichiers
   - Un processus qui tourne de manière isolée des autres processus
   - Une technologie créée par Docker = La plateforme Docker

Docker apporte une facilité de développement, de packaging et de déploiement d'applications **quelque soit le langage de programmation**. Un développeur peut **tester une application** sur sa machine en **imitant** les conditions de l'environnement de **production** tout en nécessitant une **configuration minimale**. Si l'application est soumise à un **fort stresse**, **Docker** peut orchestrer l'allocation d'autres containers. La **scalabilité** s'effectue très rapidement car un container peut être lancé en quelques secondes.

Cherchez la définition du terme **scalabilité**.

Docker permet également d'**augmenter** le rythme de **mise à jour** des logiciels.

1.4. Le modèle client/serveur

[image]

Docker utilise un modèle **client/serveur**. D'une part nous avons le client **Docker**, un fichier binaire écrit en **GO**. Et d'autre part nous avons le **Docker Daemon** (appelé **dockerd**), écrit aussi en GO, et qui expose une **API REST** consommée par le client. Le client envoie des commandes au **Docker Daemon** pour gérer les containers, les images entre autres.

1.4.1. Le serveur : Dockerd

Processus : dockerd

- Gestion des images, networks, volumes, cluster, ...
- Délègue la gestion des containers à containerd.
  - Expose une **API Rest**.
  - Ecoute sur le **socket unix** `/var/run/docker.sock` par défaut.
  - Peut-être configuré pour écouter sur un socket tcp.

1.4.2. Le client : docker

- Installé en même temps que **dockerd**.
- Communique avec le **daemon local** par défaut via `/var/run/docker.sock`.
- Peut être configuré pour communiquer avec un **daemon distant**.

1.4.3. Concepts essentiels

- **Docker** facilite la manipulation des **containers Linux**. Et cache la complexité sous-jacente.
- Introduction de la **notion d'image** : Format d'un package qui contient une application.
- Une image est un **template** qui sert pour la création d'un container.
- Pour créer une image on utilise un **Dockerfile**. Un fichier texte qui contient une liste d'instructions.
- La distribution de ces images se fait par l'intermédiaire d'un **Registry**.
- Docker permet de lancer des containers sur une machine unique ou sur un ensemble d'hôtes regroupées en un **cluster Swarm**.

Voici un schéma qui montre le **fonctionnement global des composants de base de Docker**.

[image]

Quand on installe la plateforme Docker nous avons donc : un client et un serveur (ou daemon) qui tourne constamment et qui est responsable de la gestion des containers et des images.

1.4.4. Docker Hub

Par défaut le **daemon Dockerd** communique avec le **Docker Hub**, qui est le **Registry** officiel de Docker disponible à l'adresse : https://hub.docker.com

Il existe bien entendu beaucoup d'autres Registry que l'on peut utiliser si on le souhaite.

[image]

Les images du Docker Hub peuvent être classées en plusieurs catégories.

- Les images officielles qui sont validées et que l'on peut utiliser avec confiance.
- Les images publiques à utiliser avec précaution.
- Les images privées dédiées qu'aux utilisateurs autorisés (partage d'images au sein d'une entreprise par exemple).

1.4.5. Cluster Swarm

Un Cluster Swarm est un ensemble de **Docker Host**, c'est-à-dire un ensemble de machines sur lesquel le **Docker Démon** est installé.

[image]

Ses machines vont communiquer entres elles afin d'orchestrer des applications et d'assurer qu'elles fonctionnent de la manière voulue.

1.5. Installation de Docker

Nous allons voir ici comment installer **Docker** sur votre environnement.

Rendez-vous tout d'abord dans le Docker hub puis sélectionner l'onglet **Explore**:

[image]

Sélectionnez ensuite l'onglet **Docker** dans le sous menu:

[image]

Sur la gauche vous verrez alors un menu vous permettant de sélectionner différents éléments :

- plateforme
- système d'exploitation
- architecture

Comme nous pouvons le constater, Docker peut être installé sur des systèmes divers: machine de développement, l'infrastructure d'un cloud provider, et même des devices de type Raspberry PI.

1.6. Installation sur un poste élève

Normalement, il faudrait télécharger `Docker Desktop` depuis le site officiel. Mais pour économiser la bande passante, utilisez le fichier d'installation présent dans le répertoire `\\COMMUN\BAUER\Docker\`.

Doucle cliquez sur l'installateur et laissez les options d'installation cochées par défaut. `WSL 2` est nécessaire pour faire fonctionner **DOCKER**.

Si tout se passe bien vous devriez avoir cet écran vous invitant à redémarrer la machine :

[image]

- Lancer l'application : Docker Desktop**

[image]

[image]

Si vous tentez d'exécuter l'application, il est fort probable que vous ayez un message d'erreur vous indiquant :

[image]

Pour résoudre ce problème, nous avons besoin d'ajouter les utilisateurs de la machine au groupe `docker-users` nouvellement créé par l'installation.

Ouvrez une session en `administrateur` de la machine locale : compte `INFO/INFO`. Dans **WINDOWS 10**, tapez dans le champ de recherche situé en bas à gauche :

« modifier les utilisateurs et les groupes locaux »

Cette fenêtre devrait s'ouvrir :

[image]

[image]

Double-cliquez sur le groupe `docker-users`.

Et ajoutez un nouvel utilisateur : votre compte issu du domaine `sio`

> [image]

Le système vous demandera de saisir l'identifiant et le mot de passe du compte à intégrer à ce groupe.

Redémarrer la machine et reconnectez-vous maintenant à votre compte WINDOWS standard.

Lancez L'application **Docker Desktop** et validez les conditions d'utilisation. Vous devriez avoir ce message d'erreur :

> [image]

Fermez alors la fenêtre et rendez-vous sur ce site :

Étapes d'installation manuelle pour les versions antérieures de WSL | Microsoft Docs

Suivez les étapes d'installation :

Vous allez installer `WSL2` qui est un sous-système **Linux** pour **WINDOWS**. Cela va permettre d'utiliser des commandes **Linux** dans un terminal Windows.

Tapez ensuite la commande :

```
wsl.exe --set-default-version 2
```

Nous pouvons en profiter pour installer le nouveau **Terminal de Windows**. Cela va apporter plus de confort durant la pratique de ce cours.

Lien vers la page Terminal Windows

Il faut un compte « **Microsoft** ».

Redémarrez la machine encore une fois pour que **WSL2** soit pris en compte.

Docker devrait maintenant pouvoir démarrer :

[image]

Il faut maintenant configurer le client en cliquant sur l'engrenage en haut à droite.

Cochez les options comme sur la capture d'écran :

[image]

N'oubliez pas de cliquer sur « **Apply & Restart** »

Configurez le PROXY

[image]

Si vous allez dans l'onglet **WSL** intégration :

[image]

Vous êtes maintenant prêt !

Bienvenue dans le monde de DOCKER.

Passez directement à la partie : **Vérification de l'installation**

1.7. Installation pour Windows 10 ou MacOS

Si vous êtes sur **MacOS** ou **Windows 10 (Entreprise ou Pro)** vous pouvez installer **Docker Desktop**, un environnement compatible pour chacune de ces plateformes

- Docker Desktop for Windows
- Docker Desktop for Mac

1.8. Installation pour Linux

Si vous êtes sur **Linux**, vous pouvez sélectionner la distribution que vous utilisez (**Fedora**, **CentOS**, **Ubuntu**, **Debian**) et vous obtiendrez alors un lien vers la documentation à suivre pour installer **Docker** sur la distribution en question.

Pour aller un peu plus vite, vous pouvez également lancer la commande suivante (compatible avec les principales distribution **Linux**) :

```
curl -sSL https://get.docker.com | sh
```

En quelques dizaines de secondes, cela installera **la plateforme Docker** sur votre distribution. Il sera ensuite nécessaire d'**ajouter votre utilisateur** dans le **groupe docker** afin de pouvoir interagir avec le **daemon** sans avoir à utiliser **sudo** (il faudra cependant lancer un nouveau **shell** afin que ce changement de groupe soit pris en compte.)

```
sudo usermod -aG docker <UTILISATEUR>
```

Note

Il est également possible d'installer **Docker** sur d'autres types d'**architecture infrastructure**.

1.9. Vérification de l'installation

Une fois installé, lancez la commande suivante afin de vérifier que tout est fonctionnel :

```
docker info
```

[image]

2. Les containers avec Docker

Après avoir présenté la **plateforme Docker**, nous allons voir comment créer **des conteneurs** en ligne de commande afin de lancer des services en tâche de fond et/ou rendre disponible des repertoires de la **machine hôte** dedans.

Nous verrons comment lancer un conteneur dans un mode d'**accès privilégié**, ainsi que les commandes de bases pour la gestion de son cycle de vie.

Avant la **1.13**, lancer un **conteneur** s'effectuait avec la commande : `Docker Run` sans le mot clé `container`.

Il est toujours possible de le faire. Mais maintenant les commandes ont été regroupées aux composant auquel elles se rapportent. C'est la raison pour laquelle le mot clé `container` a été rajouté pour les commandes relatives à la gestion du conteneur.

`docker container run [OPTIONS] IMAGE [COMMAND] [ARG]`

D'autres groupes de commande existent et nous les étudierons plus tard.

2.1. Hello World

Lançons notre premier container **Hello-World**.

[][] | *../images/image62.png*

Ouvrez un **terminal** et tapez :

```
docker container run hello-world
```

[][] | *../images/image63.png*

Le client demande au `daemon` (*processus*) de lancer un **conteneur** basé sur l'image `Hello-World`.

Cette image, n'étant pas disponible en local, est téléchargée et le **processus** présent dans cette image est automatiquement exécuté.

Et dans le cas de notre `Hello-world`, il s'agit seulement d'écrire du texte sur la sortie standard : **Hello from Docker** suivi d'un texte.

Cet exemple est simple, mais il met en avant le mécanisme sous-adjacent. À la fin du texte, on nous demande d'essayer un exemple plus ambitieux, c'est ce que nous allons faire par la suite.

Activité 1 : Expérimentez la commande sur votre machine :

```
docker container run hello-world
```

2.2. Ubuntu sous docker

Nous pouvons lancer un autre conteneur basé sur l'image d'**Ubuntu** et lui demander d'afficher `Hello` dans le contexte de cette image.

```
docker container run ubuntu echo hello
```

[image11] | *../images/image11.png*

Activité 2 : Analyser le contenu des cadres ci-dessus. A quelles actions correspondent-ils ?

2.3. Un containeur dans un mode intéractif

Le mode **intéractif** permet d'avoir accès à un `shell` depuis le client local qui tourne dans le **contexte du conteneur**.

Pour cela il faut rajouter **deux options** à notre commande :

`-t` qui permet d'allouer un pseudo terminal à notre container.

`-i` qui va permettre de laisser l'entrée standard du container ouverte.

Nous allons utiliser l'image **Ubuntu** qui contient les binaires et les bibliothèques du système d'exploitation **Ubuntu**.

Le processus du **conteneur** dans le **système de fichier** qui est amené par le système **Ubuntu**.

```
docker container run -t -i ubuntu bash
```

ou

```
docker container run -ti ubuntu bash
```

Résultat :

[image12] | *../images/image12.png*

Nous voyons que nous avons accès à un `shell` (*coquille* en anglais soit *interface système* dans le jargon informatique).

Nous reconnaissons sans peine le Prompt **Ubuntu/Linux** dans lequel nous pouvons écrire par exemple une commande Linux : `ls`

Activité 3 :

Tapez dans le `shell`, la commande : `cat /etc/issue`.

- Quelle information obtenez-vous ?

Pour sortir du conteneur, nous allons tuer le processus via la commande : `exit`

Nous aurions pu faire la même chose en utilisant une autre image que celle d'Ubuntu.

Lançons par exemple un conteneur basé sur la distribution **Linux Alpine** qui est distribution légère et sécurisée.

```
docker container run -t -i alpine
```

ou

```
docker container run -ti alpine
```

[image13] | *../images/image13.png*

Vous savez maintenant lancer un `shell` interactif dans un conteneur.

Quand un **conteneur** est lancé seulement avec la commande `docker container run`, il est exécuté par défaut en `foreground`, mais si l'on veut l'exécuter en `background`, c'est-à-dire, en tâche de fond, il faudra utiliser l'option `-d` et la commande retournera alors l'**identifiant** du conteneur que l'on pourra utiliser par la suite pour effectuer différentes actions.

Par exemple nous pouvons lancer un conteneur basé sur l'image `nginx`, un **serveur http**.

Example 1. Container NGINX en **foreground**

Le conteneur est lancé et occupe notre console. Nous n'avons plus la main.

```
docker container run nginx
```

[image14] | *../images/image14.png*

Example 2. Container NGINX en **background**

Le conteneur est lancé et nous récupérons la main sur la console.

```
docker container run -d nginx
```

[image15] | *../images/image15.png*

NGINX tourne en tâche de fond et nous pourrons accéder à ce conteneur par la suite grâce à son **identifiant**.

Nous pourrions aussi accéder à ce serveur web depuis un **navigateur**. Cela n'est actuellement pas possible car nous n'avons pas publié de **port**.

2.4. Publication de port.

La publication d'un port est utilisée pour qu'un **conteneur** puisse être accessible depuis l'extérieur. Afin de publier un port nous utilisons l'option `-p HOST_PORT:CONTAINER_PORT`.

Cela permet de publier un **port du conteneur** sur un **port de la machine hôte**.

L'option `-P` quant à elle laisse le choix du port au `docker démon`.

Reprenons notre conteneur **NGINX** qui est un serveur **http**. Par défaut, **NGINX** est un processus qui se lance sur le **port 80** dans le conteneur.

Si nous souhaitons accéder à notre conteneur depuis **un navigateur de la machine hôte** sur le **port 8080** de la machine hôte, nous lançerons le conteneur **nginx** avec la commande suivante :

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

[image16] | *../images/image16.png*

Maintenant, nous pouvons ouvrir notre navigateur sur l'adresse : http://localhost:8080

[image17] | *../images/image17.png*

2.5. Bind-mount

Nous allons maintenant voir comment **monter un répertoire de la machine hôte** dans un conteneur.

Cela s'effectue grâce à l'option `-v <HOST_PATH>:<CONTAINER_PATH>`

Il existe une autre notation avec l'option `--mount type=bind, src=<HOST_PATH>,dst=<CONTAINER_PATH>`

Cela permet de partager, par exemple, le code source d'un programme présent sur une **machine hôte** avec un **conteneur** ou de monter le **socket Unix** du **daemon Docker** (`/var/run/docker.sock`) pour permettre à un conteneur de dialoguer avec le **daemon**.

2.5.1. Exemple : Monter un dossier 'www'

Quand vous développez une application et que vous modifiez le code source, il peut être intéressant que cela soit pris en compte dans le conteneur.

C'est le cas lors du développement d'une **application web**. Nos **fichiers sources** sont sur une

machine locale, et dans **un conteneur,** nous avons un serveur **WEB** avec **NGINX** par exemple. Nous allons alors monter le dossier `www` local dans le **conteneur**.

```
  docker container run -v $PWD/www:/usr/share/nginx/html -d -p 80:80 nginx
```

ou

```
docker container run --mount type=bind,src=$PWD/www,dst=/usr/share/nginx/html -d -p
80:80 nginx
```

`$PWD` est une variable d'environnement qui va être créé par le **SHELL** et prendra comme valeur le **chemin du répertoire courant** dans lequel la commande a été lancé.

[image18] | *../images/image18.png*

2.5.2. Exemple: Intéragir avec le Docker Daemom

Dans cet exemple nous allons voir comment lier(*bind*) `/var/run/docker.sock`. Ce qui nous permettra d'interagir avec le `Docker Daemon` directement depuis le conteneur et cela nous donnera accès à l'**API du Daemon**.

[image19] | *../images/image19.png*

2.5.2.1. LINUX UNIQUEMENT

Créons donc un simple conteneur : avec l'image d' **Alpine**.

```
  docker container run --rm -it --name admin -v
/var/run/docker.sock:/var/run/docker.sock alpine
```

Maintenant que le conteneur est monté et branché au `Docker Daemon`, nous pouvons lui envoyer des requêtes.

Depuis le Shell:

Installons **CURL** :

`apk add curl` pour ajouter l'utilitaire `CURL`.

Nous allons lancer une requête **http POST** sur le `Docker DAEMON` :

```
    curl -X POST --unix-socket /var/run/docker.sock -d '{"Image":"nginx:1.12.2"}' -H
'Content-Type: application/json' http://localhost/containers/create
```

Cela aura pour effet de demander au **Docker Daemon** de créer un nouveau conteneur avec l'image **NGINX version 1.12.2**.

Le paramètre `-X POST` permet d'effectuer quel type de requête `http` ? Sous quel format sont envoyés les instructions de configuration de l'image **Docker** à créer ?

Pour lancer le conteneur depuis le conteneur **ADMIN** :

```
  curl -XPOST --unix-socket /var/run/docker.sock
http://localhost/containers/6b24...283b/start
```

Question 1 : Dans cette commande, à votre avis à quoi corresponde la chaine de caractère : `6b24···283b` ?

2.5.3. Ecouter les actions demandées au Docker Daemon

Nous allons lancer un autre conteneur dans lequel le **socket** est monté. Puis nous écouterons les actions demandées sur le **Docker Daemon**.

Même ceux provenant d'autre conteneur

```
  docker container run --name admin -ti -v /var/run/docker.sock:/var/run/docker.sock
alpine
```

```
  curl --unix-socket /var/run/docker.sock http://localhost/events
```

2.5.4. Limitation des ressources

Nous avons dit que le lancement d'un **conteneur** revient en fait à exécuter un **processeur**, et par défaut, il n'y a pas de limite de consommation des ressources matériels.

Par exemple, Un conteneur pourra utiliser toute la RAM et impacter tous les autres conteneurs qui tournent sur la même machine hôte.

Nous pouvons toutefois imposer des limites à un conteneur.

Lançons un conteneur avec l'image `estesp/hogit` qui a pour objectif de consommer de la ram.

```
docker container run --memory 32m estesp/hogit
```

Avec `--memory 32m`, nous avons fixé une limite : quand le processus aura atteint la limite de `32M` de **RAM** consommée, il sera tué par **Docker**.

Nous pouvons limiter l'utilisation du **CPU** également.

Lançons un conteneur avec l'image `progrium/stress` qui va se charger de stresser les cœurs du **CPU**.

```
docker container run -it rm progrium/stress -cpu 4
```

Ici les **4** cœurs du **CPU** serons utilisés car nous n'avons pas imposé de limite.

Maintenant lançons la même commande avec le flag `--cpus 0.5` pour limiter l'utilisation du **CPU à la moitié d'un cœur**. (**12%** d'utilisation)

```
docker container run -it --rm progrium/stress -cpu 4 -cpus 0.5
```

En utilisant la valeur du flag : `--cpus 2` , nous limitons l'utilisation à 2 cœurs seulement. (**50%** d'utilisation)

2.5.5. Les droits dans un conteneur

Dans un conteneur, s'il n'est pas précisé explicitement, l'utilisateur `root` sera utilisé comme propriétaire. L'utilisateur `root` du conteneur correspond à l'utilisateur `root` de la machine hôte.

Une bonne pratique est d'utiliser un autre utilisateur pour lancer le conteneur.

Il y a plusieurs façons de le définir :

- Soit à la création de l'image,
- Soit en utilisant l'option `user`,
- Soit en changeant l'utilisateur dans le processus du conteneur (`gosu`).

Lançons un conteneur basé sur l'image **Alpine** et exécutons l'instruction `sleep 10000`.

```
docker container run -d alpine sleep 10000
```

Nous allons vérifier le `owner` du processus depuis la machine hôte :

Pour LINUX :

```
ps aux | grep sleep
```

Pour WINDOWS :

Sous Windows, nous n'avons pas accès aux commandes LINUX nativement. Il faut utiliser les commandes Docker natives pour avoir accès aux informations liées aux processus des conteneurs par l'intermédiaire de leur identifiant ou nom et via la commande `top`.

Récupérez l'identifiant ou le nom du conteneurs obtenue avec la commande précédente puis :

```
docker container top <identifiant ou nom du container>
```

[image74] | *../images/image74.png*

Faisons la même manipulation, mais cette fois avec l'image officielle de **MongoDB**

a. code-block::

```
docker container run -d mongo
```

[image75] | *../images/image75.png*

On constate que le processus est la propriété d'un `owner` qui posséde un `UID` de **999**. Nous verrons par la suite comme il est possible de configurer le `owner` d'un processus lors du montage de **container**.

2.5.6. Des options utiles

- `--name` qui permet de donner un nom au container.
- `--rm` pour supprimer le container quand il est stoppé.
- `--restart=on-failure` pour relancer le container en cas d'erreur.

2.5.7. Les commandes de base avec Docker

```
docker container <command>
```

1. Résumé des commandes de base avec Docker.

Commande	Description
`run`	Création d'un conteneur
`ls`	Liste des conteneurs
`inspect`	Détails d'un conteneur
`logs`	Visualisation des logs
`exec`	Lancement d'un processus dans un conteneur existant
`stop`	Arrêt d'un conteneur
`rm`	Suppression d'un container

- La commande `ls` :

La commande `docker container ls` montre les containers qui sont en cours d'execution.

[image20] | *../images/image20.png*

Pour lister tous les conteneurs actifs et stoppés :

```
docker container ls -a
```

[image1] | *../images/image1.png*

Pour lister les identifiants des containers actifs et stoppés : `docker container ls -a -q`.

[image2] | *../images/image2.png*

A partir d'un nom ou identifiant d'un container on peut l'inspecter :

[image3] | *../images/image3.png*

La commande renvoie une multitude d'information de configuration du container. On peut utiliser des templates (**Go Template**) pour formater les données reçues et même extraire seulement des informations nécessaires : par exemple : **Obtenir l'IP**

a. code-block::

```
  docker container inspect --format '{{ .NetworkSettings.IPAddress }}' clever_kilby
* La commande `logs` :
```

Cette commande,nous permets de visualiser les logs d'un container , l'option `-f` permet de les lire en temps réel.

Créons un container sous une image **alpine** qui exécutera une commande `ping 8.8.8.8` et qui sera nommé : **ping**

```
docker container run --name ping -d alpine ping 8.8.8.8
```

Puis, écoutons en temps réel les **logs** du container nommé **ping**

```
docker container logs -f ping
```

[image]

- La commande `exec` :

Cette commande permet de lancer un processus dans un container existant pour faire du débug par exemple. Dans ce cas nous utiliserons les options `-t` et `-i` pour obtenir un `shell` intéractif.

Exemple : lançons un container qui attend 100000 secondes, et demandons ensuite d'ouvrir un shell pour lister les processus de ce container.

```
docker container run -d --name debug alpine sleep 100000
```

On lance le container avec l'option `-d` pour le mettre en tâche de fond et récupérer la main sur le

terminal et on lui donne le nom `debug` pour le manipuler facilement.

Ensuite nous utilisons la commande `exec` qui injectera dans notre container une commande, à savoir ici, la demande d'ouverture d'un `shell`.

```
docker container exec -ti debug sh
```

[image]

Sur la capture d'écran : Dans le shell, nous avons exécuté la commande `ps aux`. Qui permet de lister les processus et leur `owner`. On constact que le processus de **PID 1** , correspond à la commande `sleep`. Et le processus de PID 15 correspond à notre `ps aux`.

Warning

Si l'on `kill` le processus de **PID 1**, le container s'arrêtera, car un container n'est actif que tant que son processus de **PID 1** spécifié au lancement est en cours d'exécution.

- La commande `stop` :

Cette commande permet de stopper un ou plusieurs containers.

```
docker container stop <ID>
```

```
docker container stop <NAME>
```

Nous pouvons combiner des commandes !

Rappel : Obtenir la liste des containers en cours d'exécution :

```
docker container ls -q
```

Donc pour stopper les containers en cours d'exécution :

```
docker container stop $(docker container ls -q)
```

Les containers stoppés existent toujours :

```
docker container ls -a
```

- La commande `rm` :

Pour supprimer un container.

```
docker container rm <ID>
docker container rm <NAME>
```

Donc, par combinaison de commande, nous pouvons supprimer définitivement un ou plusieurs containers qui sont déjà stoppé.

```
docker container rm $(docker container ls -aq)
```

Avec l'option `-f` nous pouvons forcer l'arrêt d'un container et le supprimer dans la foulée.

2.6. En pratique :

Lançons quelques containers pour pratiquer, vous devez être en mesure de comprendre maintenant la finalité de ces 3 commandes :

```
docker container run -d -p 80:80 --name www nginx
```

```
docker container run -d --name ping alpine ping 8.8.8.8
```

```
docker container run hello-world
```

Listons les containers :

[image]

Nous voyons les 2 premiers containers avec le statut **UP**. Nous ne voyons pas le 3 ieme container pour la simple raison qu'une fois qu'il a effectué son action : `echo hello world` , il s'est arrêté automatiquement. Par contre avec un : `docker container ls -a` celui-ci est visible.

[image]

Son statut est **exited**, indiquant qu'il n'est pas démarré.

Nous pouvons inspecter les containers et en particulier extraire une information comme l'**adresse IP** de notre serveur web **NGINX** :

```
docker container inspect --format '{{ .NetworkSettings.IPAddress }}' www
```

[image]

Nous pouvons lancer une commande dans un container en cours : par exemple nous voulons lister la liste des processus en cours dans le container **ping** :

```
docker container exec -ti ping sh
```

Un `shell` est alors disponible, et dedans nous pouvons taper la commande : `ps aux`

[image]

Tapez : `exit` pour sortir du `shell`.

Stoppons les containers : **ping** et **www**

```
docker container stop ping www
```

faites ensuite : `docker container ls`

- Que constatez-vous ? Pourquoi ?
- Même question avec : `docker container ls -a`

Supprimons maintenant les containers créés :

`docker container rm $(docker container ls -a -q)`

2.7. Exercices :

2.7.1. Exercice 1 : Hello From Alpine

Le but de ce premier exercice est de lancer des containers basés sur l'image **alpine**.

1. Lancez un container basé sur alpine en lui fournissant la command echo hello
2. Quelles sont les étapes effectuées par le docker daemon ?
3. Lancez un container basé sur alpine sans lui spécifier de commande. Qu'observez-vous ?

2.7.2. Exercice 2 : Shell intéractif

Le but de cet exercice est lancer des containers en mode **intéractif**.

1. Lancez un container basé sur alpine en mode **interactif** sans lui spécifier de commande
2. Que s'est-il passé ?
3. Quelle est la commande par défaut d'un container basé sur **alpine** ?
4. Naviguez dans le **système de fichiers**
5. Utilisez le gestionnaire de package d'alpine (`apk`) pour ajouter un package : `apk update` et `apk add curl`.

2.7.3. Exercice 3 : foreground / background

Le but de cet exercice est de créer des containers en **foreground** et en **background**.

1. Lancez un container basé sur alpine en lui spécifiant la commande `ping 8.8.8.8`.
2. Arrêter le container avec `CTRL-C`

Le container est t-il toujours en cours d'exécution ?

Note

Vous pouvez utiliser la commande `docker ps` que nous détaillerons prochainement, et qui permet de lister les containers qui tournent sur la machine.

1. Lancez un container en mode interactif en lui spécifiant la commande `ping 8.8.8.8`.
2. Arrêter le container avec `CTRL-P CTRL-Q`

Le container est t-il toujours en cours d'exécution ?

1. Lancez un container en **background**, toujours en lui spécifiant la commande `ping 8.8.8.8`.

Le container est t-il toujours en cours d'exécution ?

2.7.4. Exercice 4 : Publication de port

Le but de cet exercice est de créer un container **en exposant un port** sur la machine **hôte**.

1. Lancez un container basé sur `nginx` et publiez le `port 80` du container sur le `port 8080` de l'hôte.
2. Vérifiez depuis votre navigateur que la page par défaut de `nginx` est servie sur `http://localhost:8080`.
3. Lancez un second container en publiant le même port.

Qu'observez-vous ?

2.7.5. Exercice 5 : Liste des containers

Le but de cet exercice est de montrer les différentes options pour lister les containers du système.

1. Listez les containers en cours d'exécution.

Est ce que tous les containers que vous avez créés sont listés ?

1. Utilisez l'option `-a` pour voir également les containers qui ont été stoppés.
2. Utilisez l'option `-q` pour ne lister que les IDs des containers (en cours d'exécution ou stoppés).

2.7.6. Exercice 6 : Inspection d'un container

Le but de cet exercice est l'inspection d'un container.

1. Lancez, en **background**, un nouveau container basé sur **nginx** en publiant le **port 80** du container sur le **port 3000** de la machine host.

Notez l'identifiant du container retourné par la commande précédente.

1. Inspectez le container en utilisant son identifiant.
2. En utilisant le **format Go template**, récupérez le nom et l'**IP** du container.
3. Manipuler les **Go template** pour récupérer d'autres information.

2.7.7. Exercice 7 : exec dans un container

Le but de cet exercice est de montrer comment lancer un processus dans un container existant.

1. Lancez un container en background, basé sur l'image alpine. Spécifiez la commande `ping 8.8.8.8` et le nom ping avec l'option `--name`.
2. Observez les logs du container en utilisant l'ID retourné par la commande précédente ou bien le nom du container.

Quittez la commande de logs avec `CTRL-C`.

1. Lancez un shell `sh`, en mode **interactif**, dans le container précédent.
2. Listez les processus du container.

Qu'observez vous par rapport aux identifiants des processus ?

2.7.8. Exercice 8 : cleanup

Le but de cet exercice est de stopper et de supprimer les containers existants.

1. Listez tous les containers (**actifs** et **inactifs**)
2. Stoppez tous les containers encore actifs en fournissant la liste des IDs à la commande `stop`.
3. Vérifiez qu'il n'y a plus de containers actifs.
4. Listez les containers arrêtés.
5. Supprimez tous les containers.
6. Vérifiez qu'il n'y a plus de containers.

2.8. En résumé

Nous avons commencé à jouer avec les containers et vu les commandes les plus utilisées pour la gestion du cycle de vie des containers (`run`, `exec`, `ls`, `rm`, `inspect`). Nous les utiliserons souvent dans la suite du cours.

C'est parfois utile d'avoir un `Shell` directement sur la machine hôte. C'est-à-dire la machine sur laquelle le `Docker Daemon` tourne. Si l'on est sur `linux`, le client et le `daemon` tournent sur la **même machine.** Par contre le `docker daemon` va tourner sur une **machine virtuelle** sous Windows alors que le client sera lui sur une machine locale.

3. Les images Docker

Nous allons parler des images **Docker**. Une image est un système de fichier qui contient une application et l'ensemble des éléments nécessaires pour la faire tourner. On peut voir une image comme étant un **template** permettant la création d'un container. L'image est portable sur n'importe quel environnement où tourne **Docker** et est composée de **couches (layers)** qui peuvent être réutilisé par d'autres images. On distribue une image via un **registry** (ex : Docker Hub)

Contenu d'une image :

[image]

La construction du fichier image, se fait dans l'ordre inverse du contenu d'une image que l'on vient de lister.

On part d'un OS de base qui va ajouter une ou plusieurs couches comme le système de fichiers. A cet OS on va ajouter une ou plusieurs couches liées à l'environnement de notre application puis de la même façon les dépendances et le code applicatifs.

Et l'ensemble de ses couches forment l'image.

[image]

3.1. Union Filesystem

Une image est donc constituée d'un ensemble de **layers** ou **couches** et chacune d'elles est en **lectures seules**. Et c'est le rôle du **storage/graph driver** de constituer le système de fichier global de l'instance du container.

Le **Graph driver** ajoute en plus une couche qui est en **lecture/écriture** pour permettre au processus de modifier le filesystem sans que les modifications ne soient persistées dans les layers de l'image. Il existe plusieurs **filesystem** et le choix du système dépend principalement du **filesystem hôte**. Par default, toutes les layers sont installées dans le répertoire `/var/lib/docker` sur la machine hôte et c'est à cette endroit que sont stockées toutes les layers des images.

Sur windows 10, docker s'exécute sur une VM. Ressources à consulter pour comprendre comment Docker fonction sous Windows :

1. https://docs.docker.com/desktop/windows/
2. https://forums.docker.com/t/the-location-of-images-in-docker-for-windows/19647

Pour accèder à ce dossier sous Windows, il faut alors créer un container et le lier avec Docker :

```
docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -i sh
```

Nous pouvons alors lister le dossier `/var/lib/docker` dans le shell.

```
ls /var/lib/docker
```

Il est possible de modifier des fichiers qui sont apportés par une layer ; cela s'appelle : **copy-On-Write**. Le fichier original est alors copier dans la layer qui est en **lecture** / **écriture** et la modification peut être **persistée**.

[image]

3.2. Exercices

3.2.1. Exercice 1 : Container's layer

La layer d'un container, est la layer **read-write** créé lorsqu'un container est lancé. C'est la layer dans laquelle tous les changements effectués dans le container sont sauvegardés. Cette layer est supprimée avec le container et ne doit donc pas être utilisée comme un stockage persistant.

Lancement d'un container

Utilisez la commande suivante pour lancer un **shell intéractif** dans un container basé sur l'image `ubuntu`.

```
docker container run -ti ubuntu
```

Installation d'un package

figlet est un package qui prend un texte en entrée et le formatte de façon amusante. Par défaut ce package n'est pas disponible dans l'image ubuntu.

Vérifiez le avec la commande suivante:

```
figlet
```

La commande devrait donner le résultat suivant :

```
bash: figlet: command not found
```

Installez le package **figlet** avec les commandes suivantes:

```
apt-get update -y
apt-get install figlet
```

Vérifiez que le binaire fonctionne :

```
figlet Hola
```

Ce qui devrait donner le résultat suivant

```
| | | | ___ | | __ _
| |_| |/ _ \| |/ _` |
|  _  | (_) | | (_| |
|_| |_|\___/|_|\__,_|
```

Sortez du container.

```
exit
```

Lancement d'un nouveau container

Lancez un nouveau container basé sur `ubuntu`.

```
docker container run -ti ubuntu
```

Vérifiez si le package figlet est présent :

```
figlet
```

Vous devriez obtenir l'erreur suivante:

```
bash: figlet: command not found
```

Comment expliquez-vous ce résultat ? Chaque container lancé à partir de l'image **ubuntu** est différent des autres. Le second container est différent de celui dans lequel **figlet** a été installé. Chacun correspond à une instance de l'image ubuntu et a sa propre **layer**, ajoutée au dessus des layers de l'image, et dans laquelle tous les changements effectués dans le container sont sauvegardés.

Sortez du container.

```
exit
```

Redémarrage du container

Listez les containers (en exécution ou non) sur la machine hôte.

```
docker container ls -a
```

Depuis cette liste, récuperez l'ID du container dans lequel le package figlet a été installé et redémarrez le avec la commande suivante.

Note: la commande `start` permet de démarrer un container se trouvant dans l'état `Exited`.

```
docker container start <CONTAINER_ID>
```

Lancez un **shell intéractif** dans ce container en utilisant la commande `exec`.

```
docker container exec -ti <CONTAINER_ID> bash
```

Vérifez que **figlet** est présent dans ce container.

```
figlet Hola
```

Résultat :

```
| | | | ___ | | __ _
| |_| |/ _ \| |/ _` |
|  _  | (_) | | (_| |
|_| |_|\___/|_|\__,_|
```

Vous pouvez maintenant sortir du container.

```
exit
```

Nettoyage

Listez les containers (en exécution ou non) sur la machine hôte

```
docker container ls -a
```

Pour supprimer tous les containers, nous pouvons utiliser les commandes `rm` et `ls -aq` conjointement. Nous ajoutons l'option `-f` afin de forcer la suppression des containers encore en exécution. Il faudrait sinon arrêter les containers et les supprimer.

```
docker container rm -f $(docker container ls -aq)
```

Tous les containers ont été supprimés, vérifiez le une nouvelle fois avec la commande suivante:

```
docker container ls -a
```

3.3. DockerFile

Le **DockerFile** est un fichier texte qui est utilisé pour la construction d'**une image DOCKER**. Il contient des instructions pour la construction du système de fichier d'une image. Nous allons partir d'un fichier de base qui sera enrichie par notre application et l'ensemble de ses dépendances.

- Exemple d'un **Docker File** dans laquelle est packagée une application **NODEJS**.

[image]

Avec l'instruction **FROM** nous définissons une image de base dans laquelle l'application **NODEJS** sera packagée.

- **COPY** qui permet d'ajouter la liste des dépendances.
- **RUN** permet de définir la commande d'installation des dépendances.
- **EXPOSE** défini le port utilisé par l'application.
- **WORKDIR** nous positionnes dans le répertoire de travail.
- **CMD** défini la commande à lancer lorsqu'un container sera lancer à partir de cette image.

Voici la liste des principales instructions à utiliser dans un **DockerFile**.

[image]

L'instruction FROM.

Il s'agit de la première instruction dans un DOckerFile. Elle permet de spécifier l'image à partir de laquelle nous allons créer une nouvelle image. On peut partir d'une image d'un OS, ou d'une image contenant déjà des applications comme un serveur web, ou un environnement d'exécution enveloppé dans une image contenant un OS de base.

Nous pouvons utiliser également une image particulière qui s'appelle **SCRATCH**, c'est une image au sens **DOCKER** même si elle est vide, et peut etre utilisé par example dans une application écrite en langage GO qui n'a pas besoin d'être packagé dans un système de fichier.

L'instruction ENV.

Cette instruction nous permet de définir des variables d'environnement. Et pourront être utilisée dans les instructions suivantes lors de la construction de l'image. On les retrouvera dans l'environnement des containers lancés à partir de cette image.

[image]

Dans cet exemple, nous construisons une image basée sur NGINX et on défini une variable `path` que l'on pourra utiliser dans les instructions suivantes : **WORKDIR** et **COPY**.

L'instruction COPY / ADD.

Permet de copier des ressources locales vers le système de fichier de l'image que l'on créé.

Et cela engendre la création d'une nouvelle layer pour l'image.

Avec l'option `chown` on peut définir les droits sur ces fichiers qu'auront les utilisateurs de l'image.

ADD permet des actions supplémentaires comme récupérer des ressources à partir d'une URL. Ou de Dézipper des fichiers.

Il est préférable d'utiliser **COPY** par rapport à **ADD** car l'on maitrise davantage comment la copie est faite.

L'instruction RUN.

RUN est une instruction qui va engendrer la construction d'une nouvelle **layer** pour l'image.

Elle permet d'exécuter une commande dans le système de fichier de l'image comme l'installation d'un package. Il y a 2 formats pour définir la commande. Le format **SHELL** qui va lancer la commande dans le contexte d'un **shell**. Et le format **Exec** qui va définir la commande comme une liste de **string** et qui n'est pas lancée dans le contexte d'un **shell**.

[image]

L'instruction EXPOSE.

Permet de spécifier les ports sur lesquels l'application écoute au lancement du container. Mais cela peut être modifié par l'option : `-p` lors de la création du container. Nous pouvons utiliser aussi un mapping comme vu précédemment : `-p HOST_PORT:CONTAINER_PORT`.

On peut aussi utiliser l'option `P` dans ce cas le démon **DOCKER** va publier l'ensemble des ports en attribuant à chacun un port de la machine hôte.

[image]

L'instruction VOLUME.

Permet de définir un répertoire dont les données sont découplées du cycle de vie du container. Les fichiers ne seront pas stockés dans la layer **lecture/écriture** du container mais dans le système de fichier de la machine hôte. Et si le container est supprimé, les données de ce répertoire seront toujours là.

Si on reprend l'exemple du **dockerfile** de **MongoDB**.

[image]

L'instruction *VOLUME* est utilisée pour créer 2 volumes. Au lancement de cette image, deux répertoires seront créés sur la machine hôte.

L'instruction USER.

Si on ne définit pas l'utilisateur, par défaut se sera `ROOT` qui sera utilisé. Ce qui pose des problèmes de sécurité évident.

L'instruction HEALTHCHECK.

Vérifie l'état de santé du processus qui tourne dans un container. On peut définir des options comme la fréquence d'inspection.

[image]

L'instruction ENTRYPOINT / CMD.

Spécifie la commande qui sera exécuté lorsque l'on lancera un container basé sur cette image. Les instruction **CMD** et **ENTRYPOINT** sont les dernières instructions du fichier **DOCKERFILE**.

On précise souvent le binaire de l'application dans **ENTRYPOINT** et les paramètres dans CMD.

La commande alors exécuté correspondra à la concaténation de **ENTRYPOINT** et **CMD**.

On peut modifier ses paramètres au lancement du container si besoin avec l'annotation **Shell** ou **Exec** vu précédemment.

[image]

3.4. Création d'images

Il est temps maintenant de créer notre image. Dans un premier temps il faut : créer un fichier **DockerFile** qui contiendra les instructions nécessaires. Ensuite il faut utiliser la commande :

```
docker image builde [OPTIONS] PATH | URL | -
```

Des options courantes :

- `-f` : spécifie le fichier à utiliser pour la construction (**DockerFile** par défaut)
- `--tag / -t` : spécifie le nom de l'image ([registry/]user/repository :tag)
- `--label` : ajout de métadonnées à l'image.

3.5. Mise en pratique

Nous allons créer une simple application **NODEJS** qui renverra la date et l'heure. Tout l'environnement nécessaire à l'exécution de ce script sera intégré dans une image que nous allons créer.

Dans un dossier, créez le fichier `index.js`:

```
var express = require('express');
var util = require('util');
var app = express();

app.get('/', function(req, res) {
  res.setHeader('Content-Type', 'text/plain');
```

```
  res.end(util.format('%s - %s\n', new Date(), 'Got Request'));
});
app.listen(process.env.PORT || 8080);
```

Puis créez le fichier `package.js` dans le même dossier :

```
{
  "name": "testnode",
  "version": "0.0.1",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.14.0"
  }
}
```

Dans une console, placez vous dans le dossier dans lequel vous avez déposé les fichiers et tapez :

```
npm install
```

puis

```
npm start
```

Ouvrez un navigateur à l'adresse : http://localhost:8080

Si tout se passe comme prévu alors vous dévriez avoir ceci :

[image]

Notre application fonctionne, mais cela est lourd pour l'utilisateur :

1. Il doit avoir NODEJS d'installé sur sa machine.
2. Il doit installer les dépendances du projet, ici `express`.
3. Il doit lancer le serveur Nodejs.

Il faudrait donc créer une image réalisant ces étapes !!

Nous allons créer un **DockerFile**.

Il nous faut trouver une image de base sur : Docker Hub

Cochez : « **Official Images** » pour n'avoir que des images officielles. Nous voyons que nous avons une multitude de possibilité pour concevoir notre image.

[image]

Nous pouvons partir sur une image **LINUX** : **UBUNTU**, **ALPINE** ...Etc mais aussi une image où le runtime **NODEJS** est déjà packagé. C'est ce que nous allons choisir.

[image]

En cliquant dessus, sélectionnez l'onglet **TAGS**.

[image]

Et dans la liste, nous allons nous intéresser à une version de **NODEJS** sous Alpine3.15.

[image]

Et conservons en mémoire le tag de cette version de node : current-alpine3.15

Maintenant dans le dossier contenant notre application, créons un fichier : **Dockerfile**. Sans extension.

Fichier : Dockerfile

```
# Nous renseignons dans l'instruction FROM le Tag de notre image qui servira de base à
notre application
FROM node:current-alpine3.15

# Nous allons copier nos fichiers sources du répertoire courant du fichier Dockerfile
dans le repertoire /app/.
# C'est un répertoire qui sera créé dans l'image lorsque l'on va faire le build
COPY . /app/
RUN cd /app && npm install
EXPOSE 8080
WORKDIR /app
CMD ["npm", "start"]
```

A partir de ce **Dockerfile**, nous allons pouvoir créer une **image**.

```
docker image build -t appbts:0.1 .
```

[image]

Nous voyons que pour chaque instruction nous avons une étape.

Si nous allons dans **Docker Desktop** , onglet « **Images** » :

[image]

Nous voyons notre image, avec son nom et son numéro de version. Nous pouvons maintenant créer un containeur avec notre application, en précisant que nous utiliserons le `port 8080` du container sur le `port 8080` de ma machine hôte.

```
docker container run -p 8080:8080 appbts:0.1
```

Et je peux maintenant utiliser mon navigateur à l'adresse : http://localhost:8080

3.6. Exercices : Création d'images

3.6.1. Exercice 1 : Création d'une image à partir d'un container

1. Lancez une container basé sur une image **alpine**, en mode **interactif**, et en lui donnant le nom `c1`.
2. Lancez la commande `curl google.com`.

Qu'observez-vous ?

1. Installez `curl` à l'aide du gestionnaire de package `apk`.
2. Quittez le container avec `CTRL-P CTRL-Q` (pour ne pas killer le processus de **PID 1**).
3. Créez une image, nommée `curly`, à partir du container `c1`.

Utilisez pour cela la commande `commit` (`docker commit --help` pour voir le fonctionnment de cette commande).

1. Lancez un `shell` intéractif dans un container basée sur l'image `curly` et vérifiez que `curl` est présent.

3.6.2. Exercice 2 : Dockerizez un serveur web simple

1. Créer un nouveau répertoire et développez un serveur **HTTP** qui expose le endpoint `/ping` sur le **port 80** et répond par **PONG**. Inspirez vous de l'exemple du cours ci-dessus.
2. Dans le même répertoire, créez le fichier **Dockerfile** qui servira à construire l'image de l'application. Ce fichier devra décrire les actions suivantes :
   - spécification d'une image de base.
   - installation du runtime correspondant au langage choisi.
   - installation des dépendances de l'application.
   - copie du code applicatif.
   - exposition du port d'écoute de l'application.
   - spécification de la commande à exécuter pour lancer le serveur.
3. Construire l'image en la taguant `pong:v1.0`.
4. Lancez un container basé sur cette image en publiant le `port 80` sur le `port 8080` de la machine hôte.
5. Tester l'application.
6. Supprimez le container.

3.6.3. Exercice 3 : ENTRYPOINT et CMD

Nous allons illustrer sur plusieurs exemples l'utilisation des instructions **ENTRYPOINT** et **CMD**. Ces instructions sont utilisées dans un **Dockerfile** pour définir la commande qui sera lancée dans un container.

3.6.3.1. Format

Dans un **Dockerfile**, les instructions **ENTRYPOINT** et **CMD** peuvent être spécifiées selon 2 formats:

- le format `shell`, ex: `ENTRYPOINT /usr/bin/node index.js`. Une commande spécifée dans ce format sera exécutée via un shell présent dans l'image. Cela peut notamment poser des problématiques car les signaux ne sont pas forwardés aux processus forkés.
- le format `exec`, ex: `CMD ["node", "index.js"]`. Une commande spécifiée dans ce format ne nécessitera pas la présence d'un shell dans l'image. On utilisera souvent le format exec pour ne pas avoir de problème si aucun shell n'est présent.

3.6.3.2. Ré-écriture à l'exécution d'un container

ENTRYPOINT et **CMD** sont 2 instructions du Dockerfile, mais elle peuvent cependant être écrasées au lancement d'un container:

- pour spécifier une autre valeur pour l'**ENTRYPOINT**, on utilisera l'option `--entrypoint`, par exemple: `docker container run --entrypoint echo alpine hello`.
- pour spécifier une autre valeur pour CMD, on précisera celle-ci après le nom de l'image, par exemple: `docker container run alpine echo hello`.

3.6.3.3. Instruction ENTRYPOINT utilisée seule

L'utilisation de l'instruction **ENTRYPOINT** seule permet de créer un wrapper autour de l'application. Nous pouvons définir une commande de base et lui donner des paramètres suplémentaires, si nécessaire, au lancement d'un container.

Dans ce premier exemple, vous allez créer un fichier **Dockerfile-v1** contenant les instructions suivantes:

```
FROM alpine
ENTRYPOINT ["ping"]
```

Créez ensuite une image, nommée `ping:1.0`, à partir de ce fichier.

```
docker image build -f Dockerfile-v1 -t ping:1.0 .
```

Lancez maintenant un container basé sur l'image **ping:1.0**

```
docker container run ping:1.0
```

La commande `ping` est lancée dans le container (car elle est spécifiée dans **ENTRYPOINT**), ce qui produit le message suivant:

```
BusyBox v1.26.2 (2017-05-23 16:46:25 GMT) multi-call binary.
Usage: ping [OPTIONS] HOST
Send ICMP ECHO_REQUEST packets to network hosts
      -4,-6           Force IP or IPv6 name resolution
      -c CNT          Send only CNT pings
      -s SIZE         Send SIZE data bytes in packets (default:56)
      -t TTL          Set TTL
      -I IFACE/IP     Use interface or IP address as source
      -W SEC          Seconds to wait for the first response (default:10)
                      (after all -c CNT packets are sent)
      -w SEC          Seconds until ping exits (default:infinite)
                      (can exit earlier with -c CNT)
      -q              Quiet, only display output at start
                      and when finished
      -p              Pattern to use for payload
```

Par défaut, aucune machine hôte n'est ciblée, et à chaque lancement d'un container il est nécessaire de préciser un **FQDN** ou une **IP**. La commande suivante lance un nouveau container en lui donnant l'adresse IP d'un DNS Google (`8.8.8.8`), nous ajoutons également l'option `-c 3` pour limiter le nombre de ping envoyés.

```
docker container run ping:1.0 -c 3 8.8.8.8
```

Nous obtenons alors le résultat suivant :

```
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=37 time=8.731 ms
64 bytes from 8.8.8.8: seq=1 ttl=37 time=8.503 ms
64 bytes from 8.8.8.8: seq=2 ttl=37 time=8.507 ms
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 packets received, 0%
round-trip min/avg/max = 8.503/8.580/8.731 ms
```

La commande lancée dans le container est donc la concaténation de l'**ENTRYPOINT** et de la commande spécifiée lors du lancement du container (tout ce qui est situé après le nom de l'image). Comme nous pouvons le voir dans cet exemple, l'image que nous avons créée est un wrapper autour de l'utilitaire `ping` et nécessite de spécifier des paramêtres supplémentaires au lancement d'un container.

3.6.3.4. Instructions CMD utilisée seule

De la même manière, il est possible de n'utiliser que l'instruction **CMD** dans un **Dockerfile**, c'est d'ailleurs très souvent l'approche qui est utilisée car il est plus simple de manipuler les instructions **CMD** que les **ENTRYPOINT**. Créez un fichier **Dockerfile-v2** contenant les instructions suivantes:

```
FROM alpine
CMD ["ping"]
```

Créez une image, nommée **ping:2.0** , à partir de ce fichier.

```
docker image build -f Dockerfile-v2 -t ping:2.0 .
```

Si nous lançons maintenant un nouveau container, il lancera la commande ping comme c'était le cas avec l'exemple précédent dans lequel seul l'ENTRYPOINT était défini.

```
$ docker container run ping:2.0

BusyBox v1.26.2 (2017-05-23 16:46:25 GMT) multi-call binary.
Usage: ping [OPTIONS] HOST
Send ICMP ECHO_REQUEST packets to network hosts
        -4,-6           Force IP or IPv6 name resolution
        -c CNT          Send only CNT pings
        -s SIZE         Send SIZE data bytes in packets (default:56)
        -t TTL          Set TTL
        -I IFACE/IP     Use interface or IP address as source
        -W SEC          Seconds to wait for the first response (default:10)
                        (after all -c CNT packets are sent)
        -w SEC          Seconds until ping exits (default:infinite)
                        (can exit earlier with -c CNT)
        -q              Quiet, only display output at start
                        and when finished
        -p              Pattern to use for payload
```

Nous n'avons cependant pas le même comportement que précédemment, car pour spécifier la machine à cibler, il faut redéfinir la commande complète à la suite du nom de l'image.

Si nous ne spécifions que les paramètres de la commande ping, nous obtenons un message d'erreur car la commande lancée dans le container ne peut pas être interpretée.

```
docker container run ping:2.0 -c 3 8.8.8.8
```

Vous devriez alors obtenir l'erreur suivante:

```
container_linux.go:247: starting container process caused "exec: \"-c\": executable
file not found in $PATH"
docker: Error response from daemon: oci runtime error: container_linux.go:247:
starting container process ca
used "exec: \"-c\": executable file not found in $PATH".
ERRO[0000] error getting events from daemon: net/http: request canceled
```

Il faut redéfinir la commande dans sa totalité, ce qui est fait en la spécifiant à la suite du nom de l'image.

```
$ docker container run ping:2.0 ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=37 time=10.223 ms
64 bytes from 8.8.8.8: seq=1 ttl=37 time=8.523 ms
64 bytes from 8.8.8.8: seq=2 ttl=37 time=8.512 ms
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 8.512/9.086/10.223 ms
```

3.6.3.5. Instructions ENTRYPOINT et CMD

Il est également possible d'utiliser ENTRYPOINT et CMD en même temps dans un Dockerfile, ce qui permet à la fois de créer un wrapper autour d'une application et de spécifier un comportement par défaut.

Nous allons illustrer cela sur un nouvel exemple et créer un fichier Dockerfile-v3 contenant les instructions suivantes:

```
FROM alpine
ENTRYPOINT ["ping"]
CMD ["-c3", "localhost"]
```

Ici, nous définissons ENTRYPOINT et CMD, la commande lancée dans un container sera la concaténation de ces 2 instructions: ping -c3 localhost. Créez une image à partir de ce Dockerfile, nommez la ping:3.0, et lançez un nouveau container à partir de celle-ci.

```
$ docker image build -f Dockerfile-v3 -t ping:3.0 .
$ docker container run ping:3.0
```

Vous devriez alors obtenir le résultat suivant:

```
PING localhost (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: seq=0 ttl=64 time=0.062 ms
64 bytes from 127.0.0.1: seq=1 ttl=64 time=0.102 ms
64 bytes from 127.0.0.1: seq=2 ttl=64 time=0.048 ms
--- localhost ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.048/0.070/0.102 ms
```

Nous pouvons écraser la commande par défaut et spécifier une autre adresse IP

```
docker container run ping:3.0 8.8.8.8
```

Nous obtenons alors le résultat suivant:

```
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=38 time=9.235 ms
64 bytes from 8.8.8.8: seq=1 ttl=38 time=8.590 ms
64 bytes from 8.8.8.8: seq=2 ttl=38 time=8.585 ms
```

Il faut alors faire un CTRL-C pour arrêter le container car l'option -c3 limitant le nombre de ping n'a pas été spécifiée. Cela nous permet à la fois d'avoir un comportement par défaut et de pouvoir facilement le modifier en spécifiant une autre commande.

3.6.4. Pour aller plus loin : ou est stockée mon image ?

3.6.4.1. Stockage d'une image

Dans un exercice précédent, nous avons créé une image nommée ping:1.0, nous allons voir ici ou cette image est stockée.

Reprenons le Dockerfile de l'exercice :

```
FROM ubuntu:16.04
RUN apt-get update -y && apt-get install -y iputils-ping
ENTRYPOINT ["ping"]
CMD ["8.8.8.8"]
```

A partir de ce Dockerfile, l'image est buildée avec la commande suivante :

```
$ docker image build -t ping:1.0 .

Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM ubuntu:16.04
---> 5e8b97a2a082
Step 2/4 : RUN apt-get update -y && apt-get install -y iputils-ping
---> Using cache
---> 4cd5304ad0fb
Step 3/4 : ENTRYPOINT ["ping"]
---> Using cache
---> d2846bbd30e8
Step 4/4 : CMD ["8.8.8.8"]
---> Using cache
---> 00a905f2bd5a
Successfully built 00a905f2bd5a
Successfully tagged ping:1.0
```

Pour lister les images présentes localement on utilise la commande `docker image ls` (on reverra cette commande un peu plus loin). Pour ne lister que les images qui ont le nom `ping` on le précise à la suite de `ls`.

```
$ docker image ls ping

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ping                1.0                 00a905f2bd5a        4 weeks ago         159MB
```

Notre image est constituée d'un ensemble de layers, il faut voir chaque layer comme un morceau de système de fichiers. L'ID de l'image (dans sa version courte) est 00a905f2bd5a, nous allons voir à partir de cette identifiant comment l'image est stockée sur la machine hôte (la machine sur laquelle tourne le daemon Docker).

Tout se passe dans le répertoire `/var/lib/docker`, c'est le répertoire au Docker gère l'ensemble des primitives (containers, images, volumes, networks, ...). Et plus précisément dans `/var/lib/docker/image/overlay2/`, overlay2 étant le driver en charge du stockage des images.

Note: si vous utilisez **Docker for Mac** ou **Docker for Windows**, il est nécessaire d'utiliser la commande suivante pour lancer un `shell` dans la machine virtuelle dans laquelle tourne le daemon Docker. On pourra ensuite explorer le répertoire `/var/lib/docker` depuis ce shell.

```
docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sh
```

Plusieurs **fichiers** / **répertoires** ont un nom qui contient l'ID de notre image comme on peut le voir ci-dessous :

```
/var/lib/docker/image/overlay2 # find . | grep 00a905f2bd5a
./imagedb/content/sha256/00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc05
816
./imagedb/metadata/sha256/00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc0
5816
./imagedb/metadata/sha256/00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc0
5816/lastUpdated
./imagedb/metadata/sha256/00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc0
5816/parent
```

- **Content** : le premier fichier contient un ensemble d'information concernant cette image, notamment les paramètres de configuration, l'historique de création (ensemble des commandes qui ont servi à construire le système de fichiers contenu dans l'image), et également l'ensemble des layers qui la constituent. Une grande partie de ces informations peuvent également être retrouvées avec la commande :

```
docker image inspect ping:1.0
```

Parmi ces éléments, on a donc les identifiants de chaque layer :

```
"rootfs": {
  "type": "layers",
```

```
  "diff_ids": [
    "sha256:644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2",
    "sha256:d7ff1dc646ba52a02312b535446d6c9b72cd09fda0480524e4828554efb2f748",
    "sha256:686245e78935e73b737c9a82111c3c7df35f5529d06ce8c2f9a7cd32ec90b456",
    "sha256:d73dd9e652956dccbbef716de4b172cc15fff644cc92fc69d221cc3a1cb89a39",
    "sha256:2de391e51d731ba02b708038a7f98b7103061b916727bcd165e9ee6402f4cdde",
    "sha256:3045bfad4cfefecabc342600d368863445b12ed18188f5f2896c5389b0e84b66"
  ]
}
```

Si l'on considère la première layer (celle dont l'ID est 6448...), on voit dans `/var/lib/docker/image/overlay2` qu'il y a un répertoire dont le nom correspond à l'ID de cette layer, celui-ci contient plusieurs fichiers :

```
/var/lib/docker/image/overlay2 # find . | grep '644879075e24394efef8a7dddefbc133aad42'
./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2
./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/size
./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/tar-
split.json.gz
./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/diff
./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/cach
e-id
./distribution/v2metadata-by-
diffid/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d
```

Ceux-ci contiennent différentes information sur la layer en question. Parmi celles-ci, le fichier **cache-id** nous donne l'identifiant du cache qui a été généré pour cette layer.

```
/var/lib/docker/image/overlay2 # cat
./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/cach
e-id
49908d07e177f9b61dc273ec7089efed9223d3798ad1d86c78d4fe953e227668
```

Le système de fichier construit dans cette layer est alors accessible dans le répertoire :

```
/var/lib/docker/overlay2/49908d07e177f9b61dc273ec7089efed9223d3798ad1d86c78d4fe953e227
668/diff/
```

LastUpdated : ce fichier contient la date de dernière mise à jour de l'image

```
/var/lib/docker/image/overlay2 # cat
./imagedb/metadata/sha256/00a905f2bd5...459dc05816/lastUpdated
2018-07-31T07:32:04.6840553Z
```

- parent : ce fichier contient l'identifiant du container qui a servi à créer l'image.

```
/var/lib/docker/image/overlay2 # cat
./imagedb/metadata/sha256/00a905f2bd5459dc05816/parent
sha256:d2846bbd30e811ac8baaf759fc6c4f424c8df2365c42dab34d363869164881ae
```

On retrouve d'ailleurs ce container dans l'avant dernière étape de création de l'image.

```
Step 3/4 : ENTRYPOINT ["ping"]
---> Using cache
---> d2846bbd30e8
```

Ce container est celui qui a été commité pour créer l'image finale.

En résumé : il est important de garder en tête qu'une image est constituée de plusieurs layers. Chaque layer est une partie du système de fichiers de l'image finale. C'est le rôle du driver de stockage de stocker ces différentes layers et de construire le système de fichiers de chaque container lancé à partir de cette image.

3.7. Multi-Stages Build

Depuis la version `17.05` de Docker, nous pouvons découper le Build d'une image en plusieurs étapes.

Un cas d'usage courant :

Etape 1 : Avoir une image de base contenant l'ensemble des librairies et binaires nécessaires pour la création d'artéfacts.

Etape 2 : Utiliser une image de base plus light et d'y copier les artéfacts générés à l'étape précédente.

Exemple :

Considérons une application **ReactJs**. Pour créer le squelette d'un projet React nous utilisons la commande :

```
npm init react-app api
```

un dossier `api` est créé.

```
cd api
```

En utilisant le **multistage build** nous allons construire des artéfacts Web. Et nous aurons seulement besoin de copier ces artefacts dans un serveur **WEB NGINX** dans un second temps.

[image]

Dans le DockerFile : La premiere instruction **FROM** utilise une image NODE dans laquelle les dépendances de l'application seront installées. Et le code applicatif Buildé.

Et la seconde instruction **FROM** utilise utilise une image **NGINX** dans laquelle les assets web buildés précédemment sont copiés. Et au final nous avons une seule image qui contient notre application.

Cela peut être vérifié en faisant le Build de l'image :

[image]

3.8. Mise en pratique

Dans cette mise en pratique, nous allons illustrer le multi stage build.

3.8.1. Un serveur http écrit en Go

Prenons l'exemple du programme suivant écrit en Go.

Dans un nouveau répertoire, créez le fichier http.go contenant le code suivant. Celui-ci définit un simple serveur http qui écoute sur le port 8080 et qui expose le endpoint /whoami en GET. A chaque requête, il renvoie le nom de la machine hôte sur laquelle il tourne.

```
package main
import (
        "io"
        "net/http"
        "os"
)
func handler(w http.ResponseWriter, req *http.Request) {
        host, err := os.Hostname()
        if err != nil {
         io.WriteString(w, "unknown")
        } else {
          io.WriteString(w, host)
        }
}
func main() {
        http.HandleFunc("/whoami", handler)
        http.ListenAndServe(":8080", nil)
}
```

3.8.1.1. Dockerfile traditionel

Afin de créer une image pour cette application, créez tout dabord le fichier Dockerfile avec le contenu suivant (placez ce fichier dans le même répertoire que http.go):

```
FROM golang:1.17
```

```
WORKDIR /go/src/app
COPY http.go .
RUN go mod init
RUN CGO_ENABLED=0 GOOS=linux go build -o http .
CMD ["./http"]
```

Dans ce Dockerfile, l'image officielle golang est utilisée comme image de base, le fichier source http.go est copié puis compilé.

Vous pouvez ensuite builder l'image et la nommer whoami:1.0:.

```
docker image build -t whoami:1.0 .
```

Listez les images présentes et notez la taille de l'image whoami:1.0

```
$ docker image ls whoami
  REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
  whoami       1.0       16795cf36deb   2 seconds ago   962MB
```

L'image obtenue a une taille très conséquente car elle contient l'ensemble de la toolchain du langage Go. Or, une fois que le binaire a été compilé, nous n'avons plus besoin du compilateur dans l'image finale.

3.8.1.2. Dockerfile utilisant un build multi-stage

Le multi-stage build, introduit dans la version 17.05 de Docker permet, au sein d'un seul Dockerfile, d'effectuer le process de build en plusieurs étapes. Chacune des étapes peut réutiliser des artefacts (fichiers résultant de compilation, assets web, ...) créés lors des étapes précédentes. Ce Dockerfile aura plusieurs instructions FROM mais seule la dernière sera utilisée pour la construction de l'image finale.

Si nous reprenons l'exemple du serveur http ci dessus, nous pouvons dans un premier temps compiler le code source en utilisant l'image golang contenant le compilateur. Une fois le binaire créé, nous pouvons utiliser une image de base vide, nommée scratch, et copier le binaire généré précédemment.

Remplacer le contenu du fichier Dockerfile avec les instructions suivantes:

```
FROM golang:1.17 as build
WORKDIR /go/src/app
COPY http.go .
RUN go mod init
RUN CGO_ENABLED=0 GOOS=linux go build -o http .

FROM scratch
COPY --from=build /go/src/app .
```

```
CMD ["./http"]
```

L'exemple que nous avons utilisé ici se base sur une application écrite en Go. ce langage a la particularité de pouvoir être compilé en un binaire static, c'est à dire ne nécessitant pas d'être "linké" à des librairies externes. C'est la raison pour laquelle nous pouvons partir de l'image scratch. Pour d'autres langages, l'image de base utilisée lors de la dernière étape du build pourra être différente (alpine, ...)

Buildez l'image dans sa version 2 avec la commande suivante.

```
docker image build -t whoami:2.0 .
```

Listez les images et observez la différence de taille entre celles-ci:

```
$ docker image ls whoami
  REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
  whoami       2.0       0a97315aeaaa   6 seconds ago   6.07MB
  whoami       1.0       16795cf36deb   2 minutes ago   962MB
```

Lancez un container basé sur l'image whoami:2.0

```
docker container run -p 8080:8080 whoami:2.0
```

A l'aide de la commande curl, envoyez une requête GET sur le endpoint exposé. Vous devriez avoir, en retour, l'identifiant du container qui a traité la requète.

```
$ curl localhost:8080/whoami
  7562306c6c5e
```

Pour cette simple application, le multistage build a permit de supprimer les binaires et librairies dont la présence est inutile dans l'image finale. L'exemple d'une application écrite en go est extrème, mais le multistage build fait partie des bonnes pratiques à adopter pour de nombreux languages de développement.

3.9. Prise en compte du cache

Quand on écrit un Dockerfile, on doit prendre en compte le mécanisme de cache.

Pour optimiser le temps nécessaire pour construire l'image.

Quand une image est créée chaque instruction créée une layer et en fonction de la complexité du Dockerfile, le premier build peut prendre un peu de temps mais les suivant seront très rapide parce que les layers existantes seront réutilisés.

Un Dockerfile qui est créé doit s'assurer que le cache est bien utilisé.

On peut l'utiliser pour reconstruire une image après qu'un changement ait été effectué, dans un fichier de configuration par exemple de sorte qu'il empêche le code source d'être compilé à nouveau si cela n'est pas nécessaire.

Il y a plusieurs façons de forcer la recréation des layers d'une image si besoin. Notamment par la modification de la valeur d'une variable d'environnement ou si on modifie le code source qui est pris en compte dans les instructions ADD ou COPY.

Si une instruction invalide le cache, alors toutes les instructions après ne l'utiliseront pas.

A partir de l'exemple de l'application NODEJS vue précédemment :

[image]

Si on lance une nouvelle fois le build de l'image on voit que pour chaque instruction le cache est utilisé. Cela signifie que pour chaque instruction la layer qui a déjà été créé, la première version, est réutilisé. Lorsque que l'image est créée pour la première fois, cela prend un peu de temps car il faut récupérer les dépendances et l'image. Mais à l'aide du cache cela prend quelque seconde.

Faites l'expérience : Dans le dossier contenant l'application NODEJS, tapez la commande :

Docker image build -t app :0.1 .

[image]

Nous allons maintenant modifier le code de l'application. Ouvrons : index.js et modifions le label

```
var express = require('express');
var util = require('util');
var app = express();
app.get('/', function(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.end(util.format('%s - %s\n', new Date(), '==> Test Modification'));
});
app.listen(process.env.PORT || 8080);
```

et rebuildons l'image

[image]

Lorsque l'on a changé le code source, cela a entrainé la reconstruction des dépendances de package.json. Ici ce n'est pas très long car nous n'avons que le package Express mais dans des applications plus lourdes cela peut impacter les performances.

Pour éviter ce problème nous allons modifier le DockerFile.

Nous allons faire en sorte de séparer le COPY en deux.

Dans le premier nous ne copierons que le fichier PACKAGE.JSON. Puis nous déplacerons l'instruction RUN de façon à récupérer les dépendances.

Ensuite nous copierons le code applicatif.

```
FROM node:current-alpine3.15
COPY package.json /app/package.json
RUN cd /app && npm install
COPY . /app/
EXPOSE 8080
WORKDIR /app
CMD ["npm", "start"]
```

Nous rebuildons ensuite notre image. Le cache n'est pas utilisé car le DOCKERFILE a été modifié alors tout est reconstruit.

Remodifions le code source.

[image]

Le code source est rechargé sans avoir à reconstruire les dépendances.

3.9.1. Exercice : Prise en compte du cache

1. Modifiez le code du serveur pong de l'exercice précédent. Vous pouvez par exemple ajouter une instruction qui loggue une chaine de caractère.
2. Construisez une nouvelle image en la taguant pong:1.1
3. Qu'observez-vous dans la sortie de la commande de build ?
4. Modifiez le Dockerfile pour faire en sorte que les dépendances ne soient pas rebuildées si un changement est effectué dans le code. Créez l'image pong:1.2 à partir de ce nouveau Dockerfile.
5. Modifiez une nouvelle fois le code de l'application et créez l'image pong:1.3. Observez la prise en compte du cache

3.10. Le contexte de Build

Quand on construit une image Docker avec la commande Docker image build. La première chose que le client Docker fait, c'est d'envoyer au Daemon, sous forme d'une archive Tar, l'ensemble des fichiers nécessaire pour construire le système de fichier de l'image. Cet ensemble constitue le Build Context. Par défaut, c'est tout les fichiers qui sont envoyés. Cela peut être dangereux si l'on a des informations sensibles. D'où l'intérêt d'utiliser un fichier .DOCKERIGNORE pour filtrer les fichier et les répertoires qui ne doivent pas être répertorié par le contexte de build.

C'est le même principe que le fichier .gitignore sur GIT par exemple.

Reprenons l'exemple de notre application NODEJS.

Refaisons un build :

[image]

Durant le build nous constatons qu'avant de transférer le context de build au Daemon Docker, on essaie de charger le fichier .dockerignore.

Le contexte ici, correspond au répertoire courant symbolisé par le « . » à la fin de la commande docker image build.

On ne veut pas forcement que certains fichiers arrivent au Docker Daemon, comme un historique GIT ou de données sensibles comme des mots de passe stockés dans un fichier ENV ..ETC

Testons cela, en créant un dépôt git :

Dans le répertoire du projet NODEJS :

Git init

[image]

Et relançons le build :

Et constatons que le context transféré passe de : 21.28 Kb à 46.15kb. Cela signifie que l'ensemble des répertoires de git ont été transféré dans le Docker Daemon.

Créons donc un fichier .dockerignore et ajoutons le dossier .git.

[image]

[image]

Relançons le build et constatons la taille du context :

⇒ ⇒ transferring context: 21.02kB

Le .GIT n'est plus envoyé dans le context.

Dans une application NODEJS, nous pourrions aussi ajouter le répertoire node_module qui contient les dépendances de l'application dans le .dockerignore.

3.11. Les commandes de base avec docker image.

La commande PULL.

Permet de télécharger une image à partir d'un registry, par défault : Docker Hub.

[image]

Format de nommage : USER/IMAGE :VERSION

Si l'on ne précise pas de numéro de version, par défaut c'est « latest » qui est retenu.

La commande : push

La commande Push permet d'uploader une image dans un registry. Pour cela il faut avoir les droits

sur ses images. Mais avant il faut avoir précisé ses identifiants de connexion au registry avec docker login.

La commande : Inspect

Permet de voir la liste des layer qui composent une image. On peut utiliser ici aussi le formaliste Go Template.

[image]

La commande : History

Permet de voir l'historique d'une image.

La commande : ls

Permet d'énumérer les images localement.

[image]

Les commandes Save et Load.

Save permet de sauvegarder une image et Load permet de charger une image à partir d'une sauvegarde.

[image]

La commande : rm

Supprime une image avec l'ensemble de ses layers. Plusieurs images peuvent être supprimées en même temps.

[image]

3.12. Exercice : Analyse du contenu d'une image

1. Télécharger l'image **mongo:3.6** en local
2. Quelles sont les différentes étapes de constructions de l'image

Comparez ces étapes avec le contenu du Dockerfile utilsé pour builder cette image.

1. Inspectez l'image
2. En utilisant la notation Go template, listez les ports exposés
3. Exportez l'image mongo:3.6 dans un tar
    - Extrayez le contenu de cette archive avec la commande `tar -xvf`, qu'observez-vous ?
    - Extrayez le contenu d'une des layers, qu'observez-vous ?
4. Supprimez l'image mongo:3.6

4. Registry

En cours de rédaction ...

5. Stockage

Dans ce chapitre, nous verrons comment une application peut persister ses données au sein d'un container.

5.1. Volume

Si un processus modifie ou créé un fichier , cette modification sera enregistrée dans le layer du container. Pour rappel, cette layer est créé au lancement du container et est en lecture/écriture. Elle est superposée aux layers qui sont en lecture de l'image.

Quand le container est supprimé, cette layer et tout les fichiers qu'elle contient sont également supprimés.

Donc pour persister des données, il faut les stocker à l'extérieur de la layer du container de manière à ne pas dépendre de son cycle de vie.

Pour cela il va falloir monter des **volumes** pour persister des données grâce à :

- L'instruction **VOLUME** dans le `Dockerfile`.
- L'option `-v` ou `--mount` à la création d'un container.
- La commande `docker create volume` de la CLI.

On utilise la persistance des données dans le cas de l'utilisation d'une base de données par exemple ou des fichiers de log.

Par exemple, si nous montons une image basée sur MongoDB :

```
docker container run -d --name mongo mongo:4.0
```

Et que nous inspectons le container :

```
docker container inspect -f '{{json .Mounts}}' mongo | python -m json.tool
```

[image]

Nous voyons que pour chacun des volumes montés, il y a un répertoire qui a été créé sur la machine hôte.

Si l'on supprime le container, tout les fichiers créés dans ce volume persisteront dans la machine hôte.

Nous avons vu comment utiliser l'option `-v CONTAINER_PATH` dans les chapitre précédent. Elle permet de créer un lien symbolique entre un dossier de la machine hôte vers un container.

Le Docker Daemon fournit une API pour manipuler les volumes. Voici les commandes de base :

[image]

Il est possible de créer, inspecter, lister, supprimer des volumes. Grâce à la commande `docker volume prune`, il est possible de supprimer les volumes qui ne sont plus utilisés afin ne libérer de la place sur la machine hôte.

Créons un volume nommé : db-data :

```
docker volume create --name db-data
```

En listant nos volumes, nous retrouvons `db-data`, et nous voyons que c'est le driver local de Docker par défaut qui a été utilisé pour la création de ce volume.

[image]

Lorsque nous consultons la liste des volumes disponibles sur notre machine hôte, nous comprenons facilement pourquoi il est interessant de leur donner un nom. Sinon le nom sera généré automatiquement et sera difficilement exploitable.

Et si nous inspectons ce volume :

```
docker volume inspect db-data
```

[image]

Nous constatons que le volume est bien monté, et que son emplacement est autogéré par le driver `local`.

Maintenant que notre volume est créé. Nous allons pouvoir facilement le monter dans un container grâce à son nom.

```
docker container run -d --name db -v db-data:/data/db mongo
```

5.2. Démo : Volume avec MongoDb

Nous allons illustrer la notion de volume dans cette partie.

Examinons les commandes à disposition pour gérer le cycle de vie des volumes :

[image119] | *../images/image119.png*

Listons les volumes existant avec la commande :

```
docker volume ls
```

Lançons un conteneur basé sur MongoDb

```
docker container run -d mongo
```

Puis listons de nouveaux la liste des volumes :

```
docker volume ls
```

[image120] | *../images/image120.png*

Récupérons un identifiant de volume et inspectons le :

```
docker inspect  b0483657c30cdacaf742264ccdebcb8385a2aac3ffd7059e32303c87f46435ff
```

La clé `Mountpoint` nous indique le répertoire dans lequel est stocké le volume sur la machine hôte, c'est-à-dire l'ensemble des bases, des collections de MongoDb par exemple.

[image121] | *../images/image121.png*

Les données sont toujours présentes dans la machine hôte même si le conteneur est supprimé. Car le volume est stocké dans la machine hôte.

Si l'on regarde le contenu du `Dockerfile` de MongoDb (voir Docker Hub), nous constatons que deux volumes sont créés.

[image122] | *../images/image122.png*

- **Comment créer nos propres volumes ?**

```
docker container run -d --name mongo -v <data>:/data/db mongo
```

On peut alors monter un conteneur Mongo avec des données existantes. Ainsi, les deux volumes par default ne seront pas créé lors du montage du conteneur.

L'intérêt de créer des volumes est de pouvoir s'interfacer avec des solutions de stockage qui existent.

Par défaut, on utilise un driver "local" sur la machine hôte. Mais l'on peut utiliser un autre driver qui pourrait nous permettre de créer un volume sur des stockages distribués ou différentes solutions.

5.3. Exercice : Utilisation des volumes

Nous allons illustrer la notion de volume. Nous verrons notamment comment définir un volume:

- Dans un Dockerfile

- Au lancement d'un conteneur en utilisant l'option `-v`
- En utilisant la ligne de commande

5.3.1. Prérequis

- Installation de `jq`

Pour certaines commandes vous aurez besoin de l'utilitaire `jq` qui permet de manipuler des structures **json** très facilement. Vous pouvez l'installer depuis https://stedolan.github.io/jq/download/

- Si vous utilisez Docker for Mac ou Docker for Windows

la plateforme Docker est installée dans une machine virtuelle tournant sur un hyperviseur léger (*xhyve* pour macOS, *Hyper-V* pour Windows). Pour effectuer cet exercice il vous faudra utiliser la commande suivante pour accéder à un `shell` dans cette machine virtuelle.

```
$ docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sh
```

Comme nous l'avons évoqué dans le cours, cette commande permet de lancer un shell dans un conteneur basé sur debian, et faire en sorte d'utiliser les namespaces de la machine hôte (la machine virtuelle) sur laquelle tourne le daemon Docker.

Une fois que vous avez lancé ce conteneur, vous pourrez naviguer dans le *filesystem* de la machine sur laquelle tourne le daemon Docker, c'est-à-dire l'endroit où les images sont stockées.

5.3.2. Persistance des données dans un conteneur

Nous allons illustrer pourquoi, par défaut, un conteneur ne doit pas être utilisé pour persister des données.

En utilisant la commande suivante, lancez un shell intéractif dans un conteneur basé sur l'image `alpine:3.8`, et nommé le `c1`.

```
docker container run --name c1 -ti alpine:3.8 sh
```

Dans ce conteneur, créez le répertoire `/data` et dans celui-ci le fichier `hello.txt`.

```
mkdir /data && touch /data/hello.txt
```

Sortez ensuite du conteneur.

```
exit
```

Lors de la création du container, une **layer** `read-write` est ajoutée au-dessus des layers `read-only` de

l'image sous-adjacente.

C'est dans cette **layer** que les changements que nous avons apportés dans le container ont étés persistés (création du fichier `/data/hello.txt`).

Nous allons voir comment cette **layer** est accessible depuis la machine hôte (celle sur laquelle tourne le daemon Docker) et vérifier que nos modifications sont bien présentes. Utilisez la command `inspect` pour obtenir le path de la layer du container `c1`.

La clé qui nous intéresse est `GraphDriver`.

```
docker container inspect c1
```

Nous pouvons scroller dans l'output de la commande suivante jusqu'à la clé `GraphDriver` ou bien nous pouvons utiliser le format **Go templates** et obtenir directement le contenu de la clé.

Lancez la commande suivante pour n'obtenir que le contenu de `GraphDriver`.

```
docker container inspect -f "{{ json .GraphDriver }}" c1 | jq .
```

Vous devriez obtenir un résultat proche de celui ci-dessous.

```
{
  "Data": {
    "LowerDir": "/var/lib/docker/overlay2/d0ffe7...1d66-
init/diff:/var/lib/docker/overlay2/acba19...b584/diff",
    "MergedDir": "/var/lib/docker/overlay2/d0ffe7...1d66/merged",
    "UpperDir": "/var/lib/docker/overlay2/d0ffe7...1d66/diff",
    "WorkDir": "/var/lib/docker/overlay2/d0ffe7...1d66/work"
  },
  "Name": "overlay2"
}
```

Avec la commande suivante, vérifiez que le fichier **hello.txt** se trouve dans le répertoire référencé par `UpperDir`.

```
CONTAINER_LAYER_PATH=$(docker container inspect -f "{{ json .GraphDriver.Data.UpperDir
}}" c1 | tr -d '"')
```

```
find $CONTAINER_LAYER_PATH -name hello.txt
/var/lib/docker/overlay2/d0ffe7...1d66/diff/data/hello.txt
```

Supprimez le container `c1` et vérifiez que le répertoire spécifié par `UpperDir` n'existe plus.

```
docker container rm c1
```

```
ls $CONTAINER_LAYER_PATH

ls: cannot access '/var/lib/docker/overlay2/d0ffe7...1d66/diff': No such file or
directory
```

Cela montre que les données créées dans le conteneur ne sont pas persistées et sont supprimées avec lui.

5.3.3. Définition d'un volume dans un Dockerfile

Nous allons maintenant voir comment les volumes sont utilisés pour permetttre de remédier à ce problème et permettre de persister des données en dehors d'un container.

Nous allons commencer par créer un Dockerfile basé sur l'image alpine et définir /data en tant que volume. Tous les élements créés dans /data seront persistés en dehors de l'union filesystem comme nous allons le voir.

Créez le fichier *Dockerfile* contenant les 2 instructions suivantes:

```
FROM alpine:3.8
VOLUME ["/data"]
```

Construisez l'image *imgvol* à partir de ce Dockerfile.

```
$ docker image build -t imgvol .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM alpine:3.8
 ---> 3f53bb00af94
Step 2/2 : VOLUME ["/data"]
 ---> Running in 7ee2310fca60
Removing intermediate container 7ee2310fca60
 ---> d8f6d5332181
Successfully built d8f6d5332181
Successfully tagged imgvol:latest
```

Avec la commande suivante, lancez un shell intéractif dans un container, nommé *c2*, basé sur l'image imgvol.

```
$ docker container run --name c2 -ti imgvol
```

Depuis le container, créez le fichier */data/hello.txt*

```
# touch /data/hello.txt
```

Sortons ensuite du container avec la commande **CTRL-P** suivie de **CTRL-Q**, cette commande permet de passer le process en tache de fond, elle n'arrête pas le container. Pour être sur que *c2* tourne, il doit apparaitre dans la liste des containers en execution. Vérifiez le avec la commande suivante:

```
$ docker container ls
```

Utilisez la commande *inspect* pour récuperer la clé **Mounts** afin d'avoir le chemin d'accès du volume sur la machine hôte.

```
$ docker container inspect -f "{{ json .Mounts }}"  c2 | jq .
```

Vous devriez obtenir un résultat similaire à celui ci-dessous (aux ID prêts).

```
[
  {
    "Type": "volume",
    "Name": "d071337...3896",
    "Source": "/var/lib/docker/volumes/d07133...3896/_data",
    "Destination": "/data",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
]
```

Le volume */data* est accessible, sur la machine hôte, dans le path spécifié par la clé **Source**.

Avec la commande suivante, vérifiez que le fichier *hello.txt* est bien présent sur la machine hôte.

```
VOLUME_PATH=$(docker container inspect -f "{{ (index .Mounts 0).Source }}" c2)
```

```
find $VOLUME_PATH -name hello.txt
/var/lib/docker/volumes/cb5...f49/_data/hello.txt
```

Supprimez maintenant le container *c2*.

```
docker container stop c2 && docker container rm c2
```

Vérifier que le fichier *hello.txt* existe toujours sur le filesystem de l'hôte.

```
$ find $VOLUME_PATH -name hello.txt
/var/lib/docker/volumes/cb5...f49/_data/hello.txt
```

Cet exemple nous montre qu'un volume permet de persister les données en dehors de l'union filesystem et ceci indépendemment du cycle de vie d'un container.

5.3.4. Définition d'un volume au lancement d'un container

Précédemment nous avons défini un volume dans le *Dockerfile*, nous allons maintenant voir comment définir des volumes à l'aide de l'option `-v` au lancement d'un container.

Lancez un container avec les caractéristiques suivantes:

- basé sur l'image *alpine:3.8* - nommé *c3* - exécution en background (option `-d`)
- définition de */data* en tant que volume (option -v) - spécification d'une commande qui écrit dans le volume ci-dessus.

Pour ce faire, lancez la commande suivante:

```
docker container run --name c3 -d -v /data alpine sh -c 'ping 8.8.8.8 >
/data/ping.txt'
```

Inspectez le container et repérez notamment le chemin d'accès du volume sur la machine hôte.

```
docker inspect -f "{{ json .Mounts }}" c3 | jq .
[
  {
    "Type": "volume",
    "Name": "2ba36b...3ef2",
    "Source": "/var/lib/docker/volumes/2ba36b...3ef2/_data",
    "Destination": "/data",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
]
```

Le volume est accessible via le filesystem de la machine hôte dans le path spécifié par la clé **Source**.

En utilisant la commande suivante, vérifiez ce que ce répertoire contient.

```
VOLUME_PATH=$(docker container inspect -f "{{ (index .Mounts 0).Source }}" c3)
```

```
tail -f $VOLUME_PATH/ping.txt
```

```
64 bytes from 8.8.8.8: seq=34 ttl=37 time=0.462 ms
64 bytes from 8.8.8.8: seq=35 ttl=37 time=0.436 ms
64 bytes from 8.8.8.8: seq=36 ttl=37 time=0.512 ms
64 bytes from 8.8.8.8: seq=37 ttl=37 time=0.487 ms
64 bytes from 8.8.8.8: seq=38 ttl=37 time=0.409 ms
64 bytes from 8.8.8.8: seq=39 ttl=37 time=0.438 ms
64 bytes from 8.8.8.8: seq=40 ttl=37 time=0.477 ms
```

Le fichier *ping.txt* est mis à jour régulièrement par la commande ping qui tourne dans le container.

Si nous stoppons et supprimons le container, le fichier *ping.txt* sera toujours disponible via le volume, cependant il ne sera plus mis à jour.

Supprimez le container *c3* avec la commande suivante.

```
$ docker container rm -f c3
```

5.3.5. Utilisation des volumes via la CLI

Les commandes relatives aux volumes ont été introduites dans Docker 1.9.

Elles permettent de manager le cycle de vie des volumes de manière très simple.

La commande suivante liste l'ensemble des sous-commandes disponibles.

```
docker volume --help
```

La commande create permet de créer un nouveau volume. Créez un volume nommé *html* avec la commande suivante.

```
docker volume create --name html
```

Lister les volumes existants et vérifiez que le volume *html* est présent.

```
docker volume ls
```

```
DRIVER              VOLUME NAME
local               html
```

Comme pour les containers et les images (et d'autres primitives Docker), la commande inspect

permet d'avoir la vue détaillée d'un volume. Inspectez le volume *html*

```
$ docker volume inspect html
[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/html/_data",
        "Name": "html",
        "Options": {},
        "Scope": "local"
    }
]
```

La clé `Mountpoint` définie ici correspond au chemin d'accès de ce volume sur la machine hôte. Lorsque l'on crée un volume via la CLI, le path contient le nom du volume et non pas un identifiant comme nous l'avons vu plus haut.

Utilisez la commande suivante pour lancer un container basé sur *nginx* et monter le volume *html* sur le point de montage */usr/share/nginx/html* du container. Cette commande publie également le port **80** du container sur le port **8080** de l'hôte.

/usr/share/nginx/html est le répertoire servi par défaut par nginx, il contient les fichiers *index.html* et *50x.html*.

```
docker run --name www -d -p 8080:80 -v html:/usr/share/nginx/html nginx:1.16
```

Depuis l'hôte, regardez le contenu du volume *html*.

```
$ ls -al /var/lib/docker/volumes/html/_data
total 16
drwxr-xr-x 2 root root 4096 Jan 10 17:07 .
drwxr-xr-x 3 root root 4096 Jan 10 17:07 ..
-rw-r--r-- 1 root root  494 Dec 25 09:56 50x.html
-rw-r--r-- 1 root root  612 Dec 25 09:56 index.html
```

Le contenu du répertoire */usr/share/nginx/html* du container a été copié dans le répertoire **/var/lib/docker/volumes/html/_data** de l'hôte.

Accédez à la page d'accueil en lançant un *curl* sur http://localhost:8080

```
$ curl localhost:8080
```

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Depuis l'hôte, nous pouvons modifier le fichier *index.html*.

```
cat<<END >/var/lib/docker/volumes/html/_data/index.html
HELLO !!!
END
```

Utilisez une nouvelle fois la commande *curl* pour vérifier que le container sert le fichier *index.html* modifié.

```
curl localhost:8080
HELLO !!!
```

5.4. Drivers de volumes

Un driver de volume permet de persister des données au travers différentes solutions de stockage.

Par défaut, le driver est `local` et si on monte un volume avec ce driver, les données seront stockées dans un répertoire dédié sur la machine hôte :

`/var/lib/docker/volume/<ID DU VOLUME>`

Mais l'on peut ajouter des drivers de volume supplémentaire par l'intermédiaire de plugin pour utiliser AWS, AZURE ...Etc

Ce schéma illustre 3 conteneurs utilisant des drivers de stockage différent sur la même machine hôte.

[image123] | *../images/image123.png*

Exemple : Plugin `sshfs`

- Stockage sur un système de fichier via ssh.

https://github.com/vieux/docker-volume-sshfs

[image124] | *../images/image124.png*

Une fois que le plugin est installé :

- Création du répertoire sur le serveur `ssh`

```
ssh USER@HOST mkdir /tmp/data
```

- Création du volume avec le driver `vieux/sshfs`

```
docker volume create -d vieux/sshfs -o sshcmd=USER@HOST:/tmp/data -o password=PASSWORD
data
```

- Liste des volumes

```
docker volume ls
```

- Utilisation du volume dans un conteneur.

```
docker run -it -v data:/data alpine
# touch /data/test
```

- Vérification sur le serveur SSH

```
ssh USER@HOST ls /tmp/data
```

5.4.1. Démo : Plugin de volume

Nous allons installer un plugin de volume et l'utiliser dans un conteneur pour faire de la persistance de données.

Dans DockerHub, nous allons rechercher "un volume".

Le plugin `vieux/sshfs` permet de créer un volume dans un dossier distant à travers une connexion

`ssh`.

https://hub.docker.com/r/vieux/sshfs

Préparation d'une machine distante avec une connexion SSH

5.4.1.1. Installation et configuration de OPENSSH SERVER sur Windows 10.

5.4.1.1.1. Installation

[image125] | *../images/image125.png*

Les deux composants **OpenSSH** peuvent être installés à l'aide des Paramètres Windows sur les appareils Windows Server 2019 et Windows 10.

Pour installer les composants OpenSSH :

Ouvrez Paramètres, sélectionnez **Applications** > **Applications** et **fonctionnalités**, puis **Fonctionnalités facultatives**.

Parcourez la liste pour voir si OpenSSH est déjà installé. Si ce n'est pas le cas, sélectionnez Ajouter une fonctionnalité en haut de la page, puis :

- Recherchez **OpenSSH Client** et cliquez sur Installer.
- Recherchez **OpenSSH Server** et cliquez sur Installer.

Une fois l'installation terminée, revenez à **Applications** > **Applications et fonctionnalités** et **Fonctionnalités facultatives**.

Vous devriez voir **OpenSSH** dans la liste.

5.4.1.1.2. Démarrer et configurer OpenSSH Server

Pour démarrer et configurer **OpenSSH Server** pour une première utilisation, ouvrez **PowerShell en tant qu'administrateur**, puis exécutez les commandes suivantes pour démarrer `sshd service` :

```
# Start the sshd service
Start-Service sshd

# OPTIONAL but recommended:
Set-Service -Name sshd -StartupType 'Automatic'

# Confirm the Firewall rule is configured. It should be created automatically by
setup. Run the following to verify
if (!(Get-NetFirewallRule -Name "OpenSSH-Server-In-TCP" -ErrorAction SilentlyContinue
| Select-Object Name, Enabled)) {
    Write-Output "Firewall Rule 'OpenSSH-Server-In-TCP' does not exist, creating
it..."
    New-NetFirewallRule -Name 'OpenSSH-Server-In-TCP' -DisplayName 'OpenSSH Server
(sshd)' -Enabled True -Direction Inbound -Protocol TCP -Action Allow -LocalPort 22
} else {
```

```
    Write-Output "Firewall rule 'OpenSSH-Server-In-TCP' has been created and exists."
}
```

5.4.1.1.3. Se connecter à OpenSSH Server

Une fois l'installation terminée, vous pouvez vous connecter à OpenSSH Server à partir d'un appareil Windows 10 ou Windows Server 2019 avec OpenSSH Client installé à l'aide de PowerShell, comme suit. Veillez à exécuter PowerShell en tant qu'administrateur :

```
ssh <login windows>@<server name>
```

Une fois connecté, vous recevez un message semblable à celui-ci :

```
The authenticity of host 'servername (10.00.00.001)' can't be established.
ECDSA key fingerprint is SHA256:(<a large string>).
Are you sure you want to continue connecting (yes/no)?
```

Si vous sélectionnez Oui, ce serveur est ajouté à la liste des hôtes SSH connus sur votre client Windows.

Vous êtes alors invité à entrer le mot de passe. Par mesure de sécurité, votre mot de passe n'est pas affiché lorsque vous l'entrez.

Une fois connecté, vous voyez l'invite de l'interpréteur de commandes Windows :

```
domain\username@SERVERNAME C:\Users\username>
```

Créez un dossier `data` sur votre serveur SSH/

5.4.1.1.4. Installation du plugin et du volume sur la machine hôte:

- Installation du plugin

```
docker plugin install vieux/sshfs # or docker plugin install vieux/sshfs DEBUG=1
```

Pour lister les plugins installés sur la machine :

```
docker plugin ls
```

- Création du volume

```
docker volume create -d vieux/sshfs -o sshcmd=<user>@<host>:<path du volume> \ -o
password=<password> data
```

```
docker container run -ti -v data:/data alpine

# touch /data/hello.txt
```

Nous constatons que le fichier a bien été créé coté serveur SSH.

Il existe d'autres drivers à disposition !

6. Docker Machine

En cours de rédaction …

7. Docker Compose

Docker Compose permet de gérer des applications complexes, c'est-à-dire par exemple des applications qui dialoguent les unes avec les autres. Très simplement nous pouvons utiliser **Docker Compose** : Grâce à une configuration sous forme de fichier YAML dont le nom par défaut est `docker-compose.yml` par défaut.

7.1. Structure du fichier `docker.compose.yml`

Dans ce fichier nous définirons l'ensemble des éléments d'une application :

- Les services.
- Les volumes.
- Les Networks qui permettent d'isoler les services.
- Les secrets (donnés sensibles nécessaires au fonctionnement de l'application, pris en compte seulement dans un cluster Swarm).
- Les configs (configuration sensibles nécessaires au fonctionnement de l'application, pris en compte seulement dans un cluster Swarm).

Examinons maintenant une application web branchée à une API configurée dans le fichier `docker-compose.yml` :

```
version: '3.9'
volumes:
  data:
networks:
  frontend:
  backend:
services:
  web:
    images: org/web:2.3
    networks:
      - frontend
    ports:
      - 80:80
  api:
    image: org/api
    networks:
      - backend
      - frontend
  db:
    image: mongo
    volumes:
      - data:/data/db
    networks:
      - backend
```

Analysons les principales clées de notre fichier :

- `Version` correspond à la version du format **Compose** à mettre en relation avec la version du **Docker Daemon** de la machine hôte qui va être utilisée pour déployer notre application. Si on utilise la dernière version de **compose** avec un Daemon plus ancien, il y a certaines options écrites dans le `docker-compose.yml` qui ne seront pas prises en compte.
- `Volumes` permet de définir un volume, que l'on appelle ici, **data** et qui sera utilisé dans un service par la suite. Par défaut, ce volume utilise le driver local, qui va donc créer un répertoire sur la machine hôte.
- `Networks` permet de créer des réseaux qui serviront à isoler des groupes de services.
- `Services` contient la définition des services nécessaires au fonctionnement de notre application. Ici, nous avons nos 3 services : `WEB`, `API`, `DB`.

Pour chaque service on spécifie l'image utilisée, les volumes pour la persistance des données.Le service `DB` est le seul à persister les données et montera le volume `data` dans le répertoire `/data/db` du container qui sera lancé.

Pour chaque service, on définit aussi les réseaux attachés avec la clé `Networks`. Dans notre exemple : le service `API` doit pouvoir communiquer avec le service WEB et DB. Donc, il faut lui donner accès aux deux réseaux attachés à ces services : `backend` et `frontend`.

En isolant ainsi les services, on s'assure qu'ils ne puissent pas avoir accès à des services dont ils n'ont pas l'utilité directement. Comme le service `WEB`, qui ne doit pas pouvoir accéder au service `DB` directement. Cela ajoute un niveau de sécurité au cas où l'un des services serait compromis.

La clé `ports` publie les ports nommés vers l'extérieur pour le service qui a besoin d'être joint, comme le serveur `Web` et son port 80.

De nombreuses options sont encore disponibles pour la définition d'un service dans le format `compose`.

Voici une liste des plus utilisées :

- Image utilisée par le container du service.
- Nombre de réplicas, c'est à dire le nombre de container identique qui sera lancé pour le service. En augmentant le nombre de container, on pourra traiter les piques de charge par exemple.
- Les ports publiés à l'extérieur par les services qui ont besoin d'être accessible par l'extérieur.
- La définition d'un Health check pour vérifier l'état de santé d'un service.
- Les stratégies de redémarrage de façon à définir que faire si un container a planté par exemple.
- Contraintes de déploiement (dans un contexte de SWARM uniquement), par exemple pour imposer qu'un container doit tourner sur une machine contenant un disque SSD.
- Contraintes des mises à jour (dans un contexte de SWARM uniquement).

Un des avantages qu'il y a à déployer une application à travers le fichier `docker-compose.yml`, c'est qu'elle peut être déployé sur n'importe quel environnement. En utilisant le binaire `compose`, un développeur peut installer sur une machine son application, avec son environnement de

développement complet.

7.2. Le binaire `docker-compose`

Le binaire `docker-compose` est utilisé pour gérer une application qui est gérée selon le format `docker-compose.yml`. Cet outil est indépendant du **docker daemon** qui est souvent livré en même temps (Docker for Mac, Docker for Windows).

```
docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
```

Il y a plusieurs éléments qui peuvent être fourni au binaire :

- Le chemin d'accès aux fichiers dans lequel est définie l'application. (Par default il s'agit du fichier : `docker-compose.yml` du répertoire courant)
- Des options, comme des chemins d'accès à des certificats et clés TLS ou l'adresse de l'hôte à contacter.
- Une commande pour gérer l'application.
- Des arguments pour cette commande.

On peut avoir plusieurs fichiers pour spécifier une configuration différente par environnement de développement.

Commande	Utilisation
up / down	Création / Suppression d'une application (services, volumes, réseaux)
start / stop	Démarrage / arrête d'une application
build	Build des images des services (si instruction build utilisée)
pull	Téléchargement d'une image
logs	Visualisation des logs de l'application
scale	Modification du nombre de container pour un service
ps	Liste les containers de l'application

7.3. Service discovery

Une application définie par `docker-compose` est en général constituée de plusieurs services dont certains communique avec d'autres. Nous sommes souvent dans un environnement **microservice**.

Pour permettre la résolution du service, le **dns** intégré dans le **docker daemon** est utilisé. Ainsi nous pouvons résoudre l'IP d'un service à partir de son nom.

Voyons un exemple :

[image]

Sur la gauche, nous avons un extrait d'une application `docker-compose` composée de deux services.

Un service est utilisé pour la base de données, `db` et un pour l'`api` qui utilise ce service `db`.

Nous voyons aussi qu'il y a un volume qui se nomme `data` et qui est monté dans le service `db`.

À droite, nous avons une partie du code **nodeJs** de l'**api** qui montre comment la connexion à la base de données est réalisée. Il suffit juste de donner le nom du service de base de données dans la chaine de connexion.

C'est quelque chose de très pratique. Toutefois, il faudra ajouter un mécanisme qui permette d'attendre que la `db` soit disponible ou éventuellement renouveler la tentative de connexion.

`Docker-compose` permet d'indiquer les dépendances entre les services, mais il ne permet pas de savoir qu'un service est disponible avant de lancer un service qui en dépend.

7.4. Mise en œuvre d'une application microservice : Voting App.

[image]

L'application **Voting App** est développée et maintenue par **Docker**. Elle est beaucoup utilisée pour des présentations ou des démos. Nous pouvons la récupérer en local en clonant le répository GitHub .

C'est une application très pratique pour illustrer le concept de microservices.

Elle est composée de :

- 5 services :
  - 2 bases de données : **redis** et **postgres**
  - 3 services développés chacun dans un environnement différent : **Python**, **NodeJs** et **.NET**

Un utilisé `vote` depuis l'interface web, par défaut l'utilisateur doit choisir entre "cat" et "dog". Le vote est stocké dans la base de données **Redis**.

Le service **Worker**, va récupérer le vote depuis **Redis** et va l'enregistrer dans la base de données **PostGres** et les utilisateurs pourront consulter les résultats via l'interface **Web** fournie par le service **Result**.

Si nous visitons le dépôt **GitHub** de l'application, nous constatons qu'il existe plusieurs fichiers **docker-compose** qui illustrent différente utilisation de l'application :

Pour la production, on aura le fichier `docker-stack` alors que pour le développement nous aurons plutôt `docker-compose`. Il est possible de choisir différents langages comme `java` ou `.NET` pour le **worker**. Ainsi que différents `OS` : **Linux** ou **Windows**.

[image]

Ouvrons le fichier `docker-compose-simple.yml`

```
version: "3"

services:
  vote:
    build: ./vote
    command: python app.py
    volumes:
    - ./vote:/app
    ports:
      - "5000:80"

  redis:
    image: redis:alpine
    ports: ["6379"]

  worker:
    build: ./worker

  db:
    image: postgres:9.4
    environment:
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "postgres"

  result:
    build: ./result
    command: nodemon server.js
    volumes:
      - ./result:/app
    ports:
      - "5001:80"
      - "5858:5858"
```

L'instruction `build` sert à définir l'emplacement du contexte de construction du service : le `dockerfile` ainsi que les autres fichiers nécessaires à la construction de l'image.

Pour le service **vote**, nous voyons que nous avons bien les fichiers de l'application et le `Dockerfile` dans le dossier `vote`.

[image]

Pour les services **vote** et **result**, nous définissons dans l'instruction `volume` le `bindmount` du code applicatif depuis la machine hôte vers le répertoire `/app` dans le container. Cela permet de rendre le code source présent sur la machine de développement directement accessible dans le container.

Et une approche qui est souvent utilisée avec `Docker-compose` en développement est de redéfinir la commande qui est normalement lancée dans le container.

On utilise pour cela le mot clé `command` comme nous pouvons le voir dans les services **vote** et **result**.

Par contre si nous ouvrons le fichier `docker-stack.yml`, nous avons une définition de l'application prête à être déployé sur un cluster **Swarm** de production.

```
version: "3"
  services:

    redis:
      image: redis:alpine
      networks:
        - frontend
      deploy:
        replicas: 1
        update_config:
          parallelism: 2
          delay: 10s
        restart_policy:
          condition: on-failure
    db:
      image: postgres:9.4
      environment:
        POSTGRES_USER: "postgres"
        POSTGRES_PASSWORD: "postgres"
      volumes:
        - db-data:/var/lib/postgresql/data
      networks:
        - backend
      deploy:
        placement:
          constraints: [node.role == manager]
    vote:
      image: dockersamples/examplevotingapp_vote:before
      ports:
        - 5000:80
      networks:
        - frontend
      depends_on:
        - redis
      deploy:
        replicas: 2
        update_config:
          parallelism: 2
        restart_policy:
          condition: on-failure
    result:
      image: dockersamples/examplevotingapp_result:before
      ports:
        - 5001:80
      networks:
```

```
        - backend
      depends_on:
        - db
      deploy:
        replicas: 1
        update_config:
          parallelism: 2
          delay: 10s
        restart_policy:
          condition: on-failure

    worker:
      image: dockersamples/examplevotingapp_worker
      networks:
        - frontend
        - backend
      depends_on:
        - db
        - redis
      deploy:
        mode: replicated
        replicas: 1
        labels: [APP=VOTING]
        restart_policy:
          condition: on-failure
          delay: 10s
          max_attempts: 3
          window: 120s
        placement:
          constraints: [node.role == manager]

    visualizer:
      image: dockersamples/visualizer:stable
      ports:
        - "8080:8080"
      stop_grace_period: 1m30s
      volumes:
        - "/var/run/docker.sock:/var/run/docker.sock"
      deploy:
        placement:
          constraints: [node.role == manager]

  networks:
    frontend:
    backend:

  volumes:
    db-data:
```

Si l'on regarde dans le service **vote** par exemple. On peut voir que contrairement au fichier `docker-`

`compose-simple`, nous n'avons pas l'instruction `build` mais `image`. Ce qui est logique puisqu'en production nous déployons les images des services et non pas les codes applicatifs dans les containers.

Nous trouvons également l'instruction `deploy` qui permet de spécifier un ensemble de propriétés dans le cadre d'un déploiement sur un cluster **Swarm** comme le nombre de `replicas`, c'est-à-dire le nombre de container identique qui seront lancés par le service.

Des contraintes de `placement`, qui indique le type de machine du cluster sur lequel le service sera déployé. On peut également spécifier des conditions de redémarrage, `restart_policy` ou la façon dont la mise à jour d'un service sera effectuée avec `update_config`. Par exemple, si un service a deux replicas, on peut vouloir mettre à jour le premier, se donner quelques secondes pour être certain qu'il fonctionne correctement avant de faire la mise à jour du second. C'est le mécanisme de **rolling update** que l'on verra dans le chapitre sur **Swarm**.

Dans un contexte de production, on s'assurera également d'isoler des groupes de services par l'intermédiaire de `networks`. Ici, tout en bas du fichier nous voyons que deux `networks` sont définis : `frontend` et `backend`.

7.5. Voting App Installation sur `Play Docker`.

Nous allons installer l'application dans un environnement temporaire dans un premier temps : Play With Docker

[image]

Cela permet de créer une session Docker dans un environment de test en ligne.

Cliquez sur : **Add New Instance**

[image]

Clonez le dépôt `git` : https://github.com/dockersamples/example-voting-app.git

```
git clone https://github.com/dockersamples/example-voting-app.git
```

[image]

Naviguez dans le dossier `example-voting-app`.

```
cd example-voting-app
```

Et lançons maintenant l'application avec `Docker-compose`. Nous lui indiquons le nom du fichier avec l'option `-f`. Le paramètre `up` indique qu'il faut monter l'application et `-d` qu'il faut rendre la main de la console une fois monté.

```
docker-compose -f docker-compose-simple.yml up -d
```

[image]

Après le déploiement de l'application, il apparait dans **Play with Docker** des boutons portant les numéros des ports des applications.

Si l'on regarde le contenu du fichier `docker-compose-simple`, nous lisons que :

Le service `vote` publie son port `80` sur le port `5000` de la machine hôte. Et que le service `result` publie son port `80` sur le port `5001` de la machine hôte.

[image]

Si l'on clique dessus, nous pourrons avoir accès aux applications ciblées :

Service de vote :

[image]

Résultats des votes :

[image]

Essayez en local !

Essayez d'installer cette application en local sur votre propre machine!

7.6. Voting App Installation en `local`.

Nous allons illustrer l'utilisation de `Docker Compose` et lancer l'application **Voting App**. Cette application est très utilisée pour des présentations et démos, c'est un bon exemple d'application micro-services simple.

7.6.1. Vue d'ensemble

L'application `Voting App` est composée de plusieurs micro-services, ceux utilisés pour la version 2 sont les suivants :

[image]

- **vote-ui**: front-end permettant à un utilisateur de voter entre 2 options
- **vote**: back-end réceptionnant les votes
- **result-ui**: front-end permettant de visualiser les résultats
- **result**: back-end mettant à disposition les résultats
- **redis**: database redis dans laquelle sont stockés les votes
- **worker**: service qui récupère les votes depuis redis et consolide les résultats dans une database postgres

- **db**: database postgres dans laquelle sont stockés les résultats

7.6.2. Récupération des repos

```
mkdir VotingApp && cd VotingApp
git clone https://gitlab.com/voting-application/$project
```

7.6.3. Installation du binaire `docker-compose`

- Si vous utilisez **Docker for Mac** ou **Docker for Windows**, le binaire `docker-compose` est déjà installé.

7.6.4. Le format de fichier `docker-compose.yml`

Plusieurs fichiers, au format **Docker Compose**, sont disponibles dans `config/compose`. Ils décrivent l'application pour différents environnements. Le fichier qui sera utilisé par défaut est le fichier `docker-compose.yml` dont le contenu est le suivant:

```
services:
 vote:
   build: ../../vote
   # use python rather than gunicorn for local dev
   command: python app.py
   depends_on:
     redis:
       condition: service_healthy
   ports:
     - "5002:80"
   volumes:
     - ../../vote:/app
   networks:
     - front-tier
     - back-tier

 vote-ui:
   build: ../../vote-ui
   depends_on:
     vote:
       condition: service_started
   volumes:
     - ../../vote-ui:/usr/share/nginx/html
   ports:
     - "5000:80"
   networks:
     - front-tier
   restart: unless-stopped

 result:
```

```
    build: ../../result
    # use nodemon rather than node for local dev
    command: nodemon server.js
    depends_on:
      db:
        condition: service_healthy
    volumes:
      - ../../result:/app
    ports:
      - "5858:5858"
    networks:
      - front-tier
      - back-tier

  result-ui:
    build: ../../result-ui
    depends_on:
      result:
        condition: service_started
    ports:
      - "5001:80"
    networks:
      - front-tier
    restart: unless-stopped

  worker:
    build:
      context: ../../worker
      dockerfile: Dockerfile.${LANGUAGE:-dotnet}
    depends_on:
      redis:
        condition: service_healthy
      db:
        condition: service_healthy
    networks:
      - back-tier

  redis:
    image: redis:6.2-alpine3.13
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: "5s"
    ports:
      - 6379:6379
    networks:
      - back-tier

  db:
    image: postgres:13.2-alpine
    environment:
      POSTGRES_USER: "postgres"
```

```
      POSTGRES_PASSWORD: "postgres"
    volumes:
      - "db-data:/var/lib/postgresql/data"
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "postgres"]
      interval: "5s"
    ports:
      - 5432:5432
    networks:
      - back-tier

volumes:
  db-data:

networks:
  front-tier:
  back-tier:
```

Ce fichier est très intéressant, car il définit également des `volumes` et `networks` en plus des **services**. Ce n'est cependant pas un fichier destiné à être lancé en production notamment **parce qu'il utilise le code local et ne fait pas référence à des images existantes pour les services** `vote-ui`, `vote`, `result-ui`, `result` et `worker`.

7.6.5. Lancement de l'application

Depuis le répertoire `config/compose`, lancez l'application à l'aide de la commande suivante (le fichier `docker-compose.yml` sera utilisé par défaut):

```
>>> docker-compose up -d
```

Les étapes réalisées lors du lancement de l'application sont les suivantes:

- création des networks front-tier et back-tier
- création du volume db-data
- construction des images pour les services *vote-ui*, *vote*, *result-ui*, *result*, *worker* et récupération des images *redis* et *postgres*
- lancement des containers pour chaque service

7.6.6. Containers lancés

Avec la commande suivante, listez les containers qui ont été lancés.

```
>>> docker-compose ps
```

```
 Name                    Command               State
```

```
Ports
------------------------------------------------------------------------------------------
--------------------------------
compose_db_1          docker-entrypoint.sh postgres    Up (healthy)   0.0.0.0:5432-
>5432/tcp,:::5432->5432/tcp
compose_redis_1       docker-entrypoint.sh redis ...   Up (healthy)   0.0.0.0:6379-
>6379/tcp,:::6379->6379/tcp
compose_result-ui_1   /docker-entrypoint.sh ngin ...   Up             0.0.0.0:5001-
>80/tcp,:::5001->80/tcp
compose_result_1      docker-entrypoint.sh nodem ...   Up             0.0.0.0:5858-
>5858/tcp,:::5858->5858/tcp, 80/tcp
compose_vote-ui_1     /docker-entrypoint.sh ngin ...   Up             0.0.0.0:5000-
>80/tcp,:::5000->80/tcp
compose_vote_1        python app.py                    Up             0.0.0.0:5002-
>80/tcp,:::5002->80/tcp
compose_worker_1      dotnet Worker.dll                Up
```

7.6.7. Les volumes créés

Listez les volumes avec la CLI, et vérifiez que le volume défini dans le fichier docker-compose.yml est présent.

```
docker volume ls
```

Le nom du volume est prefixé par le nom du répertoire dans lequel l'application a été lancée.

```
DRIVER    VOLUME NAME
local     compose_db-data
```

Par défaut ce volume correspond à un répertoire créé sur la machine hôte.

7.6.8. Les networks créés

Listez les networks avec la CLI. Les deux networks définis dans le fichier `docker-compose.yml` sont présents.

```
docker network ls
```

De même que pour le volume, leur nom est préfixé par le nom du répertoire.

```
NETWORK ID     NAME                 DRIVER    SCOPE
71d0f64882d5   bridge               bridge    local
409bc6998857   compose_back-tier    bridge    local
b3858656638b   compose_front-tier   bridge    local
2f00536eb085   host                 host      local
```

```
54dee0283ab4   none                null      local
```

Comme nous sommes dans le contexte d'un hôte unique (et non dans le contexte d'un cluster Swarm), le driver utilisé pour la création de ces networks est du type bridge. Il permet la communication entre les containers tournant sur une même machine.

7.6.9. Utilisation de l'application

Nous pouvons maintenant accéder à l'application : nous effectuons un choix entre les 2 options depuis l'interface de vote à l'adresse `http://localhost:5000`. Si vous avez lancé cette application sur un autre hôte que votre machine, vous aurez accès à cette interface à l'adresse http://HOST:5000

[image117]

Nous visualisons le résultat depuis l'interface de résultats à l'adresse http://localhost:5001 Si vous avez lancé cette application sur un autre hôte que votre machine, vous aurez accès à cette interface à l'adresse http://HOST:5001

[image118]

7.6.10. 8.6.10 Scaling du service worker

Par défaut, un container est lancé pour chaque service. Il est possible, avec l'option `--scale`, de changer ce comportement et de scaler un service une fois qu'il est lancé. Avec la commande suivante, augmenter le nombre de worker à 2.

```
$ docker-compose up -d --scale worker=2
compose_db_1 is up-to-date
compose_redis_1 is up-to-date
compose_result_1 is up-to-date
compose_vote_1 is up-to-date
compose_result-ui_1 is up-to-date
compose_vote-ui_1 is up-to-date
Creating compose_worker_2 ... done
```

Les 2 containers relatifs au service worker sont présents:

```
$ docker-compose ps
  Name                     Command                State
Ports

-------------------------------------------------------------------------------
-------------------------------
  compose_db_1          docker-entrypoint.sh postgres    Up (healthy)   0.0.0.0:5432-
>5432/tcp,:::5432->5432/tcp
  compose_redis_1       docker-entrypoint.sh redis ...   Up (healthy)   0.0.0.0:6379-
```

```
>6379/tcp,:::6379->6379/tcp
  compose_result-ui_1   /docker-entrypoint.sh ngin ...   Up             0.0.0.0:5001-
>80/tcp,:::5001->80/tcp
  compose_result_1      docker-entrypoint.sh nodem ...   Up             0.0.0.0:5858-
>5858/tcp,:::5858->5858/tcp, 80/tcp
  compose_vote-ui_1     /docker-entrypoint.sh ngin ...   Up             0.0.0.0:5000-
>80/tcp,:::5000->80/tcp
  compose_vote_1        python app.py                    Up             0.0.0.0:5002-
>80/tcp,:::5002->80/tcp
  compose_worker_1      dotnet Worker.dll                Up
  compose_worker_2      dotnet Worker.dll                Up
```

il n'est pas possible de scaler les services `vote-ui` et `result-ui` car ils spécifient tous les 2 un port, plusieurs containers ne peuvent pas utiliser le même port de la machine hôte

```
$ docker-compose up -d --scale vote-ui=3
...
ERROR: for vote-ui  Cannot start service vote-ui: driver failed programming external
connectivity on endpoint compose_vote-ui_2
(6274094570a329e3a4d9bdcdf4d31b7e3a8e3e7e78d3cc362ad56e14341913da): Bind for
0.0.0.0:5000 failed: port is already allocated
```

7.6.11. Suppression de l'application

Avec la commande suivante, stoppez l'application. Cette commande supprime l'ensemble des éléments créés précédemment à l'exception des volumes (afin de ne pas perdre de données)

```
$ docker-compose down
Stopping compose_result-ui_1 ... done
Stopping compose_vote-ui_1   ... done
Stopping compose_result_1    ... done
Stopping compose_vote_1      ... done
Stopping compose_worker_1    ... done
Stopping compose_redis_1     ... done
Stopping compose_db_1        ... done
Removing compose_vote-ui_3   ... done
Removing compose_vote-ui_2   ... done
Removing compose_result-ui_1 ... done
Removing compose_vote-ui_1   ... done
Removing compose_result_1    ... done
Removing compose_vote_1      ... done
Removing compose_worker_1    ... done
Removing compose_redis_1     ... done
Removing compose_db_1        ... done
Removing network compose_back-tier
Removing network compose_front-tier
```

Afin de supprimer également les volumes utilisés, il faut ajouter le flag `-v`:

>>> docker-compose down -v

Cet exemple illustre l'utilisation de **Docker Compose** sur l'exemple bien connu de la **Voting App** dans le cadre d'un hôte unique. Pour déployer cette application sur un environnement de production, il faudrait effectuer des modifications dans le fichier **docker-compose**, par exemple:

- Utilisation d'images pour les services
- Ajout de service supplémentaire (aggrégateur de logs, terminaison ssl, ...)
- Contraintes de déploiement
- ...

8. Docker Swarm

Nous allons aborder la solution d'orchestration développée par Docker : **SWARM**.

8.1. Swarm mode

8.1.1. Présentation

Lorsque l'on initialise un **Cluster Swarm**, on met donc à disposition les fonctionnalités d'orchestration du Docker Daemon.

Swarm est un **orchestrateur**, comme Kubernetes. Cela permet de gérer et déployer des applications qui tournent dans des conteneurs.

On définit une application dans le Docker Compose que l'on déploie dans le Cluster.

Dans Swarm, il y a une boucle de réconciliation qui compare l'état désiré de chaque service avec l'état actuel, et s'assure qu'il soit tel que défini dans le fichier **Docker Compose**.

Il y a une gestion des mises à jour de l'application, appelé **rolling update**.

Le **Swarm Mode** utilise une notion de `vIP`, une adresse **IP virtuelle** pour chaque service. Ainsi, quand un service est appelé par son nom, c'est le serveur DNS embarqué dans le Docker Daemon qui va retourner cette `vIP`. Et quand une requête arrive sur cette vIP, on a un mécanisme de `load balancing` qui entre en action et qui va rediriger le traffic vers un des conteneurs du service en question.

Swarm, permet de sécuriser les échanges au sein du cluster et d'encrypter les données.

8.1.2. Les primitives

Il existe plusieurs primitives qui ont été introduite dans le contexte d'un cluster Swarm.

- `Node` : Une machine membre d'un cluster Swarm.
- `Service` : Qui permet de spécifier la façon dont on veut lancer les conteneurs d'une application.
- `Stack` : Un groupe de service. Par exemple, quand on déploie une application définie dans un format `Docker Compose`, on crée une Stack.
- `Secret` : Un objet secret permet de conserver de façon sécurisée les données sensibles d'une application : Clé privée, mots de passe..Etc
- `Config` : Cet objet permet de gérer la configuration des applications.

8.1.3. Vue d'ensemble

[image126] | *../images/image126.png*

Nous constatons qu'il y a ici plusieurs `Dockers Host`, appelée : `Nodes`. Les 2 Nodes de droites sont les managers du cluster, ils participent à l'algorithme de consensus.

Les Nodes, Workers, sont destinés à déployer les applications sur le cluster.

8.1.4. Les commandes de base

[image127] | *../images/image127.png*

Pour créer un Swarm :

```
docker swarm init
```

Pour ajouter un Node :

```
docker swarm join
```

L'API permet également d'ajouter une clé de cryptage offrant une couche de sécurité en plus dans le cluster.

8.2. Raft : Algorithme de consensus distribué

Animation en anglais permettant d'illustrer le fonctionnement de cet algorithme.

http://thesecretlivesofdata.com/raft/

8.3. Node

Dans le contexte de Swarm, nous avons des Nodes, qui sont simplement des machines sur laquelle tourne un Docker Daemon et qui font partis du Swarm.

Un Node peut être une machine virtuelle ou physique et peut être soit Manager, soit Worker.

[image128] | *../images/image128.png*

S'il est Manager, il peut être Leader, c'est-à-dire responsable de la gestion du Swarm et des décisions d'orchestration. Ou bien, il peut être un manager non-leader et peut participer au consensus Raft.

Si le leader devient non disponible, il pourra être élu comme leader.

Un Node de type Worker, ne va pas participer au consensus Raft, mais va recevoir des instructions du leader et exécute des tâches.

8.3.1. Les états d'un Node (availability)

- **Active** : Peut recevoir des tâches.
- **Pause** : Ne peut pas recevoir des tâches. Les tâches en cours ne peuvent pas être modifiées.
- **Drain** : Ne peut pas recevoir des tâches. Les tâches en cours sont stoppées et relancées sur

d'autres Nodes.

Les commandes de base pour la primitive Node :

[image129] | *../images/image129.png*

8.3.2. Node : Initialisation du Swarm

Le Node sur lequel le Swarm est initialisé devient Leader.

[image130] | *../images/image130.png*

8.3.3. Node : Ajout d'un worker

- Machine pouvant communiquer avec le manager du Swarm.
- Les commandes Docker doivent être lancées sur un manager.

[image131] | *../images/image131.png*

8.3.4. Node : Ajout d'un manager

- Récupération d'un token depuis le manager
- Un manager non Leader a le statut Reachable

[image132] | *../images/image132.png*

8.3.5. Node : promotion / destitution

- Commandes lancées depuis un manager

[image133] | *../images/image133.png*

8.3.6. Node : Availability

- Une valeur parmi : Active / Pause / Drain
- Contrôle le déploiement des tâches sur un node

[image134] | *../images/image134.png*

8.3.7. Node : Label

- Permet l'organisation des nodes
- Peut-être utilisé dans les contraintes de déploiement d'un service

[image135] | *../images/image135.png*

8.4. [DEMO] Création d'un Swarm

Avec Virtual Box nous avons créé 3 machines virtuelles sous Ubuntu avec Docker Daemon d'installé dessus.

[image137] | *../images/image137.png*

Dans un premier temps nous allons initialiser notre `swarm` avec la commande :

```
docker swarm init
```

[image136] | *../images/image136.png*

Nous pouvons lister la liste des Node présent :

```
docker node ls
```

[image138] | *../images/image138.png*

Le `node` qui a initialisé le `swarm` devient automatiquement le `Leader`. À partir de lui nous pourrons ajouter un `Worker` grâce à la commande qui est fournie lors de l'initialisation.

- Allons maintenant sur le `Node2` afin d'exécuter cette commande dessus et en faire un `Worker` pour notre `Swarm`.

```
 docker swarm join --token SWMTKN-1-
4mu3p23ebeu30ghysb0bqaynse691bcjz0rqr60jsrshta27ld-76lagmzkiy07ltf0cqjk5w6r9
192.168.1.18:2377
```

Maintenant, sur `Node1` regardons la liste des Nodes présent sur le Swarm :

```
docker node ls
```

[image139] | *../images/image139.png*

Nous avons deux nœuds, un `Worker` et un `Leader Manager`.

- Ajoutons donc `Node3` dans notre `Swarm` avec un status de `Manager` :

Dans un premier temps, il faut exécuter la commande suivante sur le `Node1` afin de récupérer un `token` qui permettra à `Node3` de joindre le `Swarm`.

```
docker swarm join-token manager
```

Nous avons obtenu en réponse la commande à saisir dans le `Node3` :

```
 docker swarm join --token SWMTKN-1-
4mu3p23ebeu30ghysb0bqaynse691bcjz0rqr60jsrshta27ld-1fxsub3o4e0cxec0ovf2s3vw0
192.168.1.18:2377
```

[image140] | *../images/image140.png*

À partir du Node1, listons la liste des Nodes.

```
docker node ls
```

[imgae141] | *../images/imgae141.png*

Dans un node Manager, nous pouvons aussi inspecter un autre noeud par exemple et obtenir des informations précises.

```
docker node inspect node1
```

Nous pouvons rétrograder un `Node`, par exemple, le `Node3`, nommons le maintenant comme étant un simple `Worker`.

```
docker node demote node3
```

Et inversement, pour promouvoir :

```
docker node promote node2
```

8.5. [Exercice] Création en local

8.5.1. Quelques rappels sur Docker Swarm

Swarm est la solution de clustering de Docker et permet entre autres:

- le management d'un cluster d'hôtes Docker
- l'orchestration de services sur ce cluster

Un cluster Swarm est composé de 2 types de nodes:

- Les managers dont le rôle est de maintenir l'état du cluster. En interne, une implémentation de l'algorithme de consensus Raft est utilisée pour assurer cette fonction.
- Les workers dont le rôle est d'exécuter des tasks (= lancer des containers)

Par défaut, un manager est également un worker.

Le schéma suivant montre la vision haut niveau d'un cluster Swarm.

[image142] | *../images/image142.png*

Par défaut, un Swarm est sécurisé:

- encryption des logs nécessaires à la gestion du cluster
- communication TLS entre les différents nodes
- auto-rotation des certificats

[image143] | *../images/image143.png*

Dans la suite, nous allons créer un cluster swarm en local composé d'un manager et de 2 workers. Nous allons envisager différents cas de figure en fonction de la version de Docker:

- Docker Desktop (Mac), anciennement Docker for Mac
- Docker Desktop (Windows), anciennement Docker for Windows
- Docker for Ubuntu (ou pour d'autres distribution Linux)

Pour effectuer cela, nous utiliserons Docker Machine afin de créer 3 hôtes

8.5.2. Création des hôtes avec Docker Machine

8.5.2.1. Docker machine

Docker Machine est un utilitaire qui permet de créer une machine virtuelle et d'installer Docker, le client et le daemon, sur celle-ci. Il permet également de gérer le cycle de vie de ces hôtes Docker.

La commande suivante permet d'obtenir la liste des opérations qu'il est possible de faire avec docker-machine.

```
$ docker-machine --help
```

Si vous utilisez Docker Desktop ou Docker Toolbox, le binaire docker-machine est déjà disponible sur votre environnement. Dans les autres cas, il faudra l'installer, c'est un process très simple dont les instructions sont détaillées à l'adresse https://docs.docker.com/machine/install-machine/)

En fonction de l'environnement sur lequel vous êtes, il faudra utiliser le driver approprié ci-dessous.

8.5.2.2. Docker Desktop (Windows)

L'édition Windows de Docker Desktop utilise l'hyperviseur **Hyper-V**, il faudra dans un premier temps créer un switch virtuel depuis l'interface de Hyper-V, puis ensuite utiliser le driver "hyperv" de docker-machine en précisant le nom de ce switch.

Dans l'exemple ci dessous on utilise un switch virtuel nommé **DMSwitch**.

```
PS C:\WINDOWS\system32> docker-machine create --driver hyperv --hyperv-virtual-switch
"DMSwitch" node1
Running pre-create checks...
Creating machine...
(node1) Copying C:\Users\luc\.docker\machine\cache\boot2docker.iso to
C:\Users\luc\.docker\machine\machines\node1\boot2docker.iso...
(node1) Creating SSH key...
(node1) Creating VM...
(node1) Using switch "DMSwitch"
(node1) Creating VHD
(node1) Starting VM...
(node1) Waiting for host to start...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual
machine, run: C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe env
node1
```

De la même façon nous créeons les 2 autres hôtes, node2 et node3.

```
PS C:\WINDOWS\system32> docker-machine create --driver hyperv --hyperv-virtual-switch
"DMSwitch" node2
PS C:\WINDOWS\system32> docker-machine create --driver hyperv --hyperv-virtual-switch
"DMSwitch" node3
```

La commande suivante permet de lister les hôtes que nous venons de créer

```
PS C:\WINDOWS\system32> docker-machine ls
NAME    ACTIVE   DRIVER   STATE     URL                       SWARM   DOCKER
ERRORS
node1   -        hyperv   Running   tcp://192.168.1.25:2376           v18.09.1
node2   -        hyperv   Running   tcp://192.168.1.26:2376           v18.09.1
node3   -        hyperv   Running   tcp://192.168.1.27:2376           v18.09.1
```

8.5.2.3. Docker Desktop (Mac)

Pour Docker for Mac, nous utilisons [Virtualbox](https://www.virtualbox.org/) afin d'instantier 3 machines virtuelles.

La commande suivante permet de créer un hôte nommé **node1**

```
$ docker-machine create --driver virtualbox node1
Running pre-create checks...
Creating machine...
(node1) Copying /Users/luc/.docker/machine/cache/boot2docker.iso to
/Users/luc/.docker/machine/machines/node1/boot2docker.iso...
(node1) Creating VirtualBox VM...
(node1) Creating SSH key...
(node1) Starting the VM...
(node1) Check network to re-create if needed...
(node1) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual
machine, run: docker-machine env node1
```

De la même façon nous créeons les 2 autres hôtes, node2 et node3.

```
$ docker-machine create --driver virtualbox node2
$ docker-machine create --driver virtualbox node3
```

Nous pouvons alors nous assurer que nos 3 hôtes sont en état de marche en lançant la commande suivante:

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER
ERRORS
node1   -        virtualbox   Running   tcp://192.168.99.100:2376           v18.09.1
node2   -        virtualbox   Running   tcp://192.168.99.101:2376           v18.09.1
node3   -        virtualbox   Running   tcp://192.168.99.102:2376           v18.09.1
```

8.5.2.4. Docker for Ubuntu

De la même façon que pour Docker Desktop, dans un environnement Linux nous utiliserons [Virtualbox](https://www.virtualbox.org/) afin d'instantier 3 machines virtuelles.

```
$ docker-machine create --driver virtualbox node1
Running pre-create checks...
```

```
(node1) Default Boot2Docker ISO is out-of-date, downloading the latest release...
(node1) Latest release for github.com/boot2docker/boot2docker is v18.09.1
(node1) Downloading /home/luc/.docker/machine/cache/boot2docker.iso from
https://github.com/boot2docker/boot2docker/releases/download/v18.09.1/boot2docker.iso.
..
(node1) 0%....10%....20%....30%....40%....50%....60%....70%....80%....90%....100%
Creating machine...
(node1) Copying /home/luc/.docker/machine/cache/boot2docker.iso to
/home/luc/.docker/machine/machines/node1/boot2docker.iso...
(node1) Creating VirtualBox VM...
(node1) Creating SSH key...
(node1) Starting the VM...
(node1) Check network to re-create if needed...
(node1) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual
machine, run: docker-machine env node1
```

Nous répétons alors la procédure pour créer 2 autres hôtes Docker

```
$ docker-machine create --driver virtualbox node2
$ docker-machine create --driver virtualbox node3
```

De la même façon que précédemment, la commande suivante permet de s'assurer que les 3 hôtes sont en état de marche:

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER
ERRORS
node1   -        virtualbox   Running   tcp://192.168.99.100:2376           v18.09.1
node2   -        virtualbox   Running   tcp://192.168.99.101:2376           v18.09.1
node3   -        virtualbox   Running   tcp://192.168.99.102:2376           v18.09.1
```

8.5.3. Création du swarm

Maintenant que nous avons créé les 3 hôtes Docker, nous allons les mettre en cluster en créant un swarm.

8.5.3.1. Initialisation à partir de node1

La commande suivante permet de nous connecter en ssh sur node1

```
$ docker-machine ssh node1
```

Nous pouvons ensuite initialiser le swarm

```
$ docker swarm init
Swarm initialized: current node (preife0qe9vjyum4rv13qm33l) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-0jo31iectxf4uo4airmn1cepphe9mbg4j8j6276as56i6gi82c-
8ggnqa3165gb0x8idf8tqs68p \
    10.0.107.3:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the
instructions.
```

```
Attention, lors de l'initialisation il est possible que vous obteniez une erreur
semblable au message suivant
```

```
 ```
Error response from daemon: could not choose an IP address to advertise since this
system has multiple addresses on different > interfaces (10.0.2.15 on eth0 and
192.168.99.100 on eth1) - specify one with --advertise-addr
 ```
```

Il faudra dans ce cas préciser l'adresse IP sur laquelle le manager sera accessible. On pourra initialiser le swarm avec la commande suivante:

```
$ docker swarm init --advertise-addr 192.168.99.100
```

Le daemon Docker du node1 est maintenant en Swarm mode, nous pouvons lister les nodes de notre cluster.

```
$ docker node ls
ID                            HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS  ENGINE
VERSION
tntzagkqg9uchatr8hzh8alio *   node1      Ready    Active         Leader
18.09.1
```

8.5.3.2. Ajout des workers

Depuis les 2 autres terminaux, nous lançons la commande **docker swarm join...** obtenue lors de l'initialisation du Swarm.

```
$ docker swarm join \
    --token SWMTKN-1-0jo31iectxf4uo4airmn1cepphe9mbg4j8j6276as56i6gi82c-
8ggnqa3165gb0x8idf8tqs68p \
    192.168.99.100:2377
```

Nous obtenons alors la réponse suivante

```
This node joined a swarm as a worker.
```

Depuis le node1 (seul manager de notre swarm), nous pouvons alors lister les nodes présents dans notre cluster.

```
$ docker node ls
ID                            HOSTNAME    STATUS  AVAILABILITY  MANAGER STATUS   ENGINE
VERSION
tntzagkqg9uchatr8hzh8alio *   node1       Ready   Active        Leader
18.09.1
7avuhygyf15ia91kyotee6zxf     node2       Ready   Active
18.09.1
ua03a3x0rtgrbnbnq2nifdtm2     node3       Ready   Active
18.09.1
```

Le node1 est l'unique manager du cluster, node2 et node3 sont des workers.

Toutes les commandes du client Docker doivent être envoyées sur un manager, si nous lançons la commande précédente sur node2 ou node3, nous obtenons l'erreur suivante.

```
$ docker node ls
Error response from daemon: This node is not a swarm manager. Worker nodes can't be
used to view or modify cluster state. Please run this command on a manager node or
promote the current node to a manager.
Inspection d'un node
```

8.5.4. Inspection d'un hôte

Comme pour les autres primitives de la plateforme (containers, images, volumes, networks), la commande inspect permet d'obtenir la vue détaillée d'un node.

```
$ docker node inspect node1
```

```
[
    {
        "ID": "jrqb09bsa47xb23pyrw0fgah4",
        "Version": {
            "Index": 9
        },
        "CreatedAt": "2017-11-03T14:00:17.670877138Z",
        "UpdatedAt": "2017-11-03T14:00:18.278047861Z",
        "Spec": {
            "Labels": {},
            "Role": "manager",
            "Availability": "active"
        },
        "Description": {
            "Hostname": "node1",
            "Platform": {
                "Architecture": "x86_64",
                "OS": "linux"
            },
            "Resources": {
                "NanoCPUs": 1000000000,
                "MemoryBytes": 1044123648
            },
            "Engine": {
                "EngineVersion": "17.10.0-ce",
                "Labels": {
                    "provider": "virtualbox"
                },
                "Plugins": [
                    {
                        "Type": "Log",
                        "Name": "awslogs"
                    },
                    {
                        "Type": "Log",
                        "Name": "fluentd"
                    },
                    {
                        "Type": "Log",
                        "Name": "gcplogs"
                    },
                    {
                        "Type": "Log",
                        "Name": "gelf"
                    },
                    {
                        "Type": "Log",
                        "Name": "journald"
                    },
                    {
                        "Type": "Log",
```

```
                        "Name": "json-file"
                    },
                    {
                        "Type": "Log",
                        "Name": "logentries"
                    },
                    {
                        "Type": "Log",
                        "Name": "splunk"
                    },
                    {
                        "Type": "Log",
                        "Name": "syslog"
                    },
                    {
                        "Type": "Network",
                        "Name": "bridge"
                    },
                    {
                        "Type": "Network",
                        "Name": "host"
                    },
                    {
                        "Type": "Network",
                        "Name": "macvlan"
                    },
                    {
                        "Type": "Network",
                        "Name": "null"
                    },
                    {
                        "Type": "Network",
                        "Name": "overlay"
                    },
                    {
                        "Type": "Volume",
                        "Name": "local"
                    }
                ]
            },
            "TLSInfo": {
                "TrustRoot": "-----BEGIN CERTIFICATE-----
\nMIIBaTCCARCgAwIBAgIUIDgi01oBB7DgxX0wYvAaHBj8z14wCgYIKoZIzj0EAwIw\nEzERMA8GA1UEAxMIc3
dhcm0tY2EwHhcNMTcxMTAzMTM1NTAwWhcNMzcxMDI5MTM1\nNTAwWjATMREwDwYDVQQDEwhzd2FybS1jYTBZMB
MGByqGSM49AgEGCCqGSM49AwEH\nA0IABPOOT4AHEM48JwGIp0aiFDHRtXSVQm9yJoaM3awlok7zwVJfVcLWES
VT7B9u\n4LQtjT5S1+ZUDa1UjQELWQtGaHajQjBAMA4GA1UdDwEB/wQEAwIBBjAPBgNVHRMB\nAf8EBTADAQH/
MB0GA1UdDgQWBBTdFvlG+hSDdP+AFSL0ZiSEjetnWDAKBggqhkjO\nPQQDAgNHADBEAiB9atr7uP0eclaq06Z9
1kyOH8IYxXW9jxInIfNu37BtXgIgfqWu\nQ51iyG4tNLodPgyxefMEpQrFS/pfDGDlu2Timi4=\n-----END
CERTIFICATE-----\n",
                "CertIssuerSubject": "MBMxETAPBgNVBAMTCHN3YXJtLWNh",
                "CertIssuerPublicKey":
```

```
"MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE845PgAcQzjwnAYinRqIUMdG1dJVCb3ImhozdrCWiTvPBUl9Vw
tYRJVPsH27gtC2NPlLX5lQNrVSNAQtZC0Zodg=="
            }
        },
        "Status": {
            "State": "ready",
            "Addr": "192.168.99.100"
        },
        "ManagerStatus": {
            "Leader": true,
            "Reachability": "reachable",
            "Addr": "192.168.99.100:2377"
        }
    }
]
```

Cette commande retourne entre autres: - le status du node - son rôle (manager vs worker) - les plugins disponibles - le certificat TLS

Nous pouvons utiliser les Go template afin d'obtenir une information précise contenue dans cette structure json. Par exemple, la commande suivante permet de récupérer l'adresse IP du node directement.

```
$ docker node inspect -f "{{ .Status.Addr }}" node1
```

8.5.5. Mise à jour d'un node

Une fois les nodes crées, il est possible de changer leur rôle et de: - promouvoir un worker en manager - destituer un manager en worker

8.5.5.1. Promotion de node2

La commande suivante permet de changer le node2 et d'en faire un manager

```
$ docker node promote node2
Node node2 promoted to a manager in the swarm.
```

Les commandes Docker peuvent maintenant être lancées depuis node1 ou node2 puisque tous 2 sont des managers.

```
$ docker node ls
ID                            HOSTNAME    STATUS  AVAILABILITY  MANAGER STATUS   ENGINE
VERSION
tntzagkqg9uchatr8hzh8alio *  node1       Ready   Active        Leader
18.09.1
7avuhygyf15ia91kyotee6zxf    node2       Ready   Active        Reachable
18.09.1
```

```
ua03a3x0rtgrbnbnq2nifdtm2    node3       Ready   Active
18.09.1
```

Le node2 a l'entrée Reachable dans la colonne MANAGER STATUS, ce qui signifie qu'il est du type manager, mais pas Leader.

8.5.5.2. Destitution du node2

Depuis node1 ou node2, nous pouvons destituer le node2 et le repasser en worker.

```
$ docker node demote node2
Manager node2 demoted in the swarm.
```

Si nous listons une nouvelle fois les nodes du cluster, nous pouvons voir que le node2 n'a plus d'entrée dans la colonne MANAGER STATUS.

```
$ docker node ls
ID                           HOSTNAME    STATUS  AVAILABILITY  MANAGER STATUS   ENGINE
VERSION
tntzagkqg9uchatr8hzh8alio *  node1       Ready   Active        Leader
18.09.1
7avuhygyf15ia91kyotee6zxf    node2       Ready   Active
18.09.1
ua03a3x0rtgrbnbnq2nifdtm2    node3       Ready   Active
18.09.1
```

8.5.6. Résumé

Nous avons vu dans cet exercice la façon de créer un swarm en local, soit en utilisant VirtualBox, soit en utilisant l'hyperviseur Hyper-V.

8.6. [DEMO] Création d'un Service

Nous allons créer un nouveau service basé sur l'image `nginx`.

```
docker service create --name www -p 80:80 --replicas 3 nginx
```

Listons les services de notre Swarm :

```
docker service ls
```

[image144] | *../images/image144.png*

Nous pouvons rajouter des réplicats

```
docker service scale www=5
```

Listons les services `www` :

```
docker service ps www
```

[image145] | *../images/image145.png*

Nous constatons que les services se répartissent sur les nodes du Swarm.

8.7. Création d'un service

Dans cette partie, nous allons créer un service simple, le déployer et le scaler (c'est à dire modifier le nombre de containers instanciés pour celui-ci). Nous changerons également la disponibilité d'un node pour voir l'impact sur les tâches en cours.

8.7.1. Pré-requis

Vous pouvez utiliser un Swarm que vous avez créé lors des exercices précédent, ou bien en créer un nouveau. N'hésitez pas à vous reporter aux cours ou bien aux exercices précédents si vous avez besoin de revoir le process de création.

8.7.2. Création d'un service

Utilisez la commande suivante pour créer le service nommé **vote** ayant les spécifications ci-dessous:

- basé sur l'image **instavote/vote:latest**
- publication du 80 sur le port 8080 des nodes du Swarm
- 6 réplicas

Le nombre de réplicas correspond au nombre de tâches qui seront instanciées pour ce service. Chaque tâche lancera un container basée sur l'image définie dans le service.

Lancez cette commande dpuis un node qui a le status de manager.

```
$ docker service create \
    --name vote \
    --publish 8080:80 \
    --replicas 6 \
    instavote/vote
```

Après quelques secondes, le temps que l'image **instavote/vote:latest** soit téléchargée sur chaque node, les 6 tâches du services sont lancées.

```
iloiyvba6uensc6bzvhvbxv93
overall progress: 6 out of 6 tasks
1/6: running   [==================================================>]
2/6: running   [==================================================>]
3/6: running   [==================================================>]
4/6: running   [==================================================>]
5/6: running   [==================================================>]
6/6: running   [==================================================>]
verify: Service converged
```

Utilisez la commande suivante pour lister les tâches du service **vote**.

```
$ docker service ps vote
```

Vous devriez obtenir un résultat proche du résultat suivant (aux identifiants près), indiquant qu'il y a 2 tâches par node.

```
ID            NAME    IMAGE                  NODE   DESIRED STATE  CURRENT STATE
ERROR PORTS
xnon20jonsfd  vote.1  instavote/vote:latest  node2  Running        Running 13 minutes
ago
rganh2g8y8b7  vote.2  instavote/vote:latest  node3  Running        Running 13 minutes
ago
on8oog1833yq  vote.3  instavote/vote:latest  node1  Running        Running 13 minutes
ago
hmp2wtvojxro  vote.4  instavote/vote:latest  node2  Running        Running 13 minutes
ago
vdizjy291q4t  vote.5  instavote/vote:latest  node3  Running        Running 13 minutes
ago
mjpn0ybsg6pj  vote.6  instavote/vote:latest  node1  Running        Running 13 minutes
ago
```

Le service publie le port 80 sur le port 8080 du swarm via le mécanisme de routing mesh. Cela signifie que le service sera accessible depuis le port 8080 de chaque node du cluster. Nous pouvons vérifier le routing mesh en envoyant une requête sur le port 8080 de node1, node2 ou node3. Quelque soit le node sur lequel la requête est envoyée, nous aurons accès à l'interface de l'application. Nous reviendrons en détails sur le routing mesh un peu plus tard dans le cours.

[image146] | *../images/image146.png*

Seule la partie front-end est disponible sur cet exemple, il n'y a pas le backend permettant de prendre en compte la sélection.

Les requêtes envoyées sur le port 8080 d'un node du Swarm sont traitées selon un algorithme de round-robin entre les différents containers intanciés pour le service. Cela signifie que chacun des containers recevra une requète à tour de rôle.

Nous pouvons l'observer en lançant plusieurs requêtes à la suite et observer l'identifiant ID du container depuis l'interface web

8.7.3. Ajout du service de visualisation

Utilisez la commande suivante pour lancer un service qui servira à la visualisation des containers sur le cluster.

```
$ docker service create \
  --name visualizer \
  --mount type=bind,source=/var/run/docker.sock,destination=/var/run/docker.sock \
  --constraint 'node.role == manager' \
  --publish "8000:8080" dockersamples/visualizer:stable
```

Ce service a la spécification suivante: - il est nommé **visualizer** - il fait un bind-mount de la socket **/var/run/docker.sock/** afin de permettre au container du visualizer de dialoguer avec le daemon Docker sur lequel il tourne - il utilise une contrainte de déploiement pour que le replica du service tourne sur le node qui a le rôle manager - il publie le port **8080** sur le port **8000** sur chaque hôte du Swarm

Nous pouvons alors voir la répartition des containers sur les nodes via l'interface du visualiseur. Cette interface donne le même résultat que lorsque l'on liste les service en ligne de commande, mais c'est une façon assez ludique de visualiser les containers.

[image147] | *../images/image147.png*

Note: le visualizer tourne sur le node master mais il est possible d'y accéder depuis n'importe quel node du Swarm grace au mécanisme de routing mesh

8.7.4. Passage du node2 en drain

Un node est dans l'un des états suivants: - active, il peut recevoir des nouvelles tâches - pause, il ne peut pas recevoir de nouvelles tâches mais les tâches en cours restent inchangées - drain, il ne peut plus recevoir de nouvelles tâches et les tâches en cours sont re-schédulées sur d'autres nodes

Avec la commande suivante, changez l'availability du **node2** en lui donnant la valeur **drain**.

```
$ docker node update --availability drain node2
```

Depuis le visualizer, regardez comment la répartition des tâches précédentes a été modifiée.

Nous pouvons voir qu'il n'y a plus de tâche sur le node2. Elles ont été stoppées et reschédulées sur les autres nodes.

La commande suivante change l'availability du node2 en le repositionnant dans l'état active. Les tâches existantes sur node1 et node3 restent inchangées mais de nouvelles tâches pourront de nouveau être schédulées sur node2.

```
$ docker node update --availability active node2
```

8.7.5. En résumé

Nous avons vu dans cette mise en pratique comment créer des services très simples. Nous avons également vu l'influence de la propriété **availability** d'un node par rapport à la répartition des tâches des services.

9. Network

En cours de rédaction …

10. Sécurité

En cours de rédaction …

11. Gestion des logs

En cours de rédaction …

12. Mise en Pratique

12.1. Serveur HTTP avec Docker

[image]

Nous allons vous guider à chaque étape de la mise en place d'un serveur **WEB APACHE** sous **DOCKER**.

Une équipe d'**ESPORT** : **NECROMANCERS**, a confié la création de son site web à une agence de communication dont la maquette HTML est disponible dans le fichier `esport.zip`.

- **Créez** un dossier « **docker** » dans « **mes documents** » et créez un autre dossier à l'intérieur nommé « **www** ».
- **Téléchargez** ce fichier sur votre disque dur, et **dézippez** l'archive. Vous obtenez un dossier qui se nomme : « **esport** ».
- **Déplacez** le dossier « **esport** » dans le dossier.

Votre première mission sera de mettre en place un serveur **WEB** sous **Apache** avec **Docker**.

- Nous avons plusieurs possibilités qui s'offrent à nous :
  - Soit nous téléchargeons une image d'une distribution Linux puis nous installons et configurons nous-même Apache.
  - Soit nous trouvons une image contenant déjà Apache.

La force de Docker est de posséder une sorte de « **AppStore** », un « **Hub** » appelé le « **dockerHub** » qui regroupe des images officielles et non officielles utilisables. Recherchons donc une image correspondant à notre besoin : Docker Hub

[image]

Examinons les distributions **Linux** qui accompagnent **Apache** en cliquant dessus. Nous constatons qu'**Alpine** est la distribution par défaut. C'est un bon choix, car c'est une distribution **Linux Légère**.

[image]

En parcourant la page dédiée de l'image Apache nous trouvons le sous-titre « **How to use this image** » qui nous permettra de comprendre comment utiliser ce container.

Nous l'installerons grâce au nom de l'image : `httpd`.

Nous allons avoir besoin de dire au container **Apache** où trouver les fichiers de la maquette pour les interpréter. Nous savons qu'ils sont stockés sur notre machine ,appelée la **machine hôte**, au chemin suivant dans mon cas : `C:\Users\baptiste\Documents\docker\td\www\esport`. Le container Apache, (*lire la documentation*) est configuré pour aller lire les fichiers Web dans son dossier interne : `/usr/local/apache2/htdocs/`.

Il faut donc **monter** notre répertoire de la **machine hôte** dans le dossier **htdocs** de Apache2. Cela

est possible grâce à l'option : `-v <HOST_PATH>:<CONTAINER_PATH>`

Apache, le serveur Web, écoute par défaut sur le **port 80** du container. Il faut donc lier un port de la machine hôte avec le port 80 du container. Nous décidons arbitrairement de publier le **port 80 du container** sur le **port 2000 de notre machine hôte**. Grâce à l'option `-p HOST_PORT:CONTAINER_PORT`.

Une fois que notre container est lancé, nous voulons continuer à avoir la main sur notre terminal. Par conséquent il faudra donc utiliser l'option `-d` pour lancer le container en **background** (tâche de fond).

Nous nommerons ce container avec l'option : `--name serveur_http`.

Au final, la commande pour installer et lancer notre container sera :

```
docker container run -d --name serveur_http -v
$PWD/Documents/docker/td/www/esport:/usr/local/apache2/htdocs -p 2000:80 httpd
```

[image]

Pour tester, ouvrez votre navigateur et saisissez l'adresse : http://localhost:2000/

[image]

Nous allons ouvrir un **shell** dans le container pour consulter le contenu du dossier : `/usr/local/apache2/htdocs`

Tapez la commande :

```
docker container exec -ti serveur_http sh
```

`docker container exec` permet de donner l'ordre à notre container de lancer une commande et l'option `-ti` permet de garder la main sur le **shell**.

À partir du shell, plaçons-nous donc dans le répertoire `htdocs`. .. code-block:

```
cd /usr/local/apache2/htdocs
```

et listons les fichiers le contenant :

```
ls
```

[image]

Nous voyons que le dossier **HTDOCS** de notre container contient les fichiers de notre application provenant de notre dossier **esport**.

Avec un éditeur de code comme **Visual Studio Code**, ouvrez le fichier `/www/esport/index.html` à

partir de la machine hôte.

Ajoutez le code suivant entre les lignes 250 et 251 et actualisé le navigateur :

```
<h4 class="text-white landing-title">mode dev</h4>
```

[image]

Note

Nous constatons que les fichiers de notre application ne sont pas réellement dans le container. Nous l'avons prouvé en modifiant le fichier à partir de la machine hôte et en observant que la modification a été prise en compte par le navigateur Nous n'avons créé qu'un **lien symbolique** de nos fichiers locaux dans le container.

Notre site, est accessible !!! Nous avons rempli notre première mission !

12.2. Serveur HTTPS avec Docker

Dans la partie 13.1, nous avons mis en place un container Apache permettant d'accéder à notre site web par l'intermédiaire de l'adresse : http://localhost:2000 Mais la connexion `http` n'est pas sécurisée. Pour cela, il faut que le protocole soit `https`.

Rappel : Création des certificats SSL

Les applications Web utilisent le protocole `HTTPS` pour s'assurer que les communications entre les clients et le serveur soient cryptées et ne puissent pas être interceptées. De plus, **Google** pénalise le contenu des sites web qui utilisent le protocole `HTTP` seul dans le référencement. Il est donc obligatoire de configurer notre serveur pour lui permettre d'être accessible via le protocole `HTTPS`.

Pendant le développement local, les développeurs utilisent :

- Soit le protocole **HTTP**.

  Cela signifie alors que les versions du projet en local ou en production sont développées dans un environnement différent. Cela peut être plus difficile pour repérer les problèmes.

- Soit un (faux) certificat **SSL Autosigné**.

  L'environnement de développement est alors proche de l'environnement de production, mais le navigateur continue de traiter les requêtes différemment. Par exemple, les fausses requêtes SSL ne sont pas mis en cache.

Toutes les communications clients/serveurs ont besoin d'être sécurisés avec un protocole. Nous utiliserons SSL (Secure Socket Layer).

Les communications sécurisées entre des applications se font grâce à des certificats (CERT) distribués par une autorité certifiante (CA) qui est mutuellement agréé par le client et le serveur.

Le format CERT

La plupart des certificats ont pour extension `.pem`, `.cer`, `.crt`, `.key`. Les clients (navigateurs) communiquant avec le serveur vont garder le fichier **`.pem`** `(PRIVACY ENHANCED MAIL) ou.CER` (extension pour les certificats SSL) pour établir une connexion sécurisée.

[image]

L'algorithme RSA (*Rivest Shamir Adleman*) est utilisé pour crypter et décrypter les messages dans une communication où une clé est gardée publique et l'autre clé est privée. C'est le concept de chiffrage asymétrique.

1. Le client demande une ressource protégée au serveur.
2. Le client présente les informations cryptées avec sa clé publique au serveur.
3. Le serveur évalue la requête avec sa clé privée (disponible seulement coté serveur) et répond en retour en rapport avec la ressource demandée par le client.

Cela fonctionnerait de la même manière pour l'authentification mutuelle où le client et le serveur fournissent tous deux leurs clés publiques et déchiffrent leurs messages avec leurs propres clés privées disponibles de leur côté.

Note

Nous avons déjà configuré `HTTPS` sur un serveur `apache` sous `Ubuntu` lors de d'exercices précédents avec des machines virtuelles. Aujourd'hui nous allons donc travailler sur cette distribution, revoir les étapes de création d'un certificat SSL auto-signé et l'intégration dans une image Docker. Bien entendu, il existe déjà des images toutes prêtes sur **Docker Hub** … mais nous n'apprendrons rien de nouveau aujourd'hui si nous nous contenterions d'utiliser un existant.

Maintenant, construisons notre container.

Comme nous sommes en local, il nous faudra **autosigner** nos certificats **SSL**.

Stoppons d'abord le container `serveur_http` précédent qui utilise le port `2000` :

```
docker container stop serveur_http
```

Nous allons installer un container avec une image `Ubuntu` et mapper les `ports 80` et `443` de la machine hôte avec les même ports du container.

```
docker run -it -p 80:80 -p 443:443 --name serveur_https ubuntu /bin/sh;
```

Installons un éditeur de texte `nano` , `apache2` notre serveur http et `opennss` qui permettra de générer des certificats.

```
apt update
apt install nano apache2 openssl y
```

Il faut maintenant démarrer le serveur **Apache**:

```
service apache2 start
```

Testons dans le navigateur : http://localhost

Nous voulons un site pour notre équipe de **Esport** : Les *Necromancers* !

Créons donc un dossier spécialement pour eux!

```
mkdir /var/www/html/esport
```

Et créons dedans un fichier `index.html` qui contiendra le code suivant :

```
<h1>Page de test des NECROMANCERS !!</h1>
```

Pour cela nous utiliserons notre éditeur de texte `nano` :

```
nano  /var/www/html/esport/index.html
```

Rappel : Pour sauvegarder, tapez au clavier sur les touches `CTRL` + `O` et `Entrée` et pour quitter `CTRL` + `X`

Nous allons maintenant modifier notre fichier HOSTS sur la machine hôte afin de forcer la redirection du domaine necromancers.esport sur notre serveur local en cours de conception.

Ouvrez Visual Studio Code ou un autre éditeur comme NotePad++ avec des droits d'administrateur, et éditez le fichier :

```
C:\Windows\System32\drivers\etc\hosts
```

Ajoutez la ligne suivante :

```
127.0.0.1 necromancers.esport
```

Puis modifiez les paramètres du proxy comme suit :

[image]

Ainsi, nous ne passerons pas par le proxy, ni par le DNS pour accéder à notre site avec l'url necromancers.esport , mais sur le serveur local d'addresse IP directement: `127.0.0.1`.

Maintenant, il faut configurer **Apache** dans notre container pour que notre URL pointe vers le dossiers WEB du serveur.

Apache permet de faire des redirections de connexions entrantes sur un de ses ports vers un dossier de notre choix. Cela se fait grâce aux VirtualHost. Copions le fichier VirtualHost de base nommé 000-default.conf et appelons cette copie esport.conf.

```
cp /etc/apache2/sites-available/000-default.conf /etc/apache2/sites-
available/esport.conf
```

Modifions maintenant ce nouveau fichier :

```
nano /etc/apache2/sites-available/esport.conf
```

[image]

Profitons-en aussi pour modifier le fichier /etc/apache2/apache2.conf. Et lui rajouter une ligne : ServerName localhost. Cela va permettre de nommer notre serveur local, et d'éviter d'avoir des avertissements au redémarrage.

Le fichier esport.conf est prêt ! Il faut le charger dans la configuration du serveur **Apache2**.

```
a2ensite esport
```

Pour que les modifications soient prise en compte, redémarrons le serveur.

```
service apache2 restart
```

Maintenant que notre serveur **Apache** est configuré pour que l'adresse : necromancers.esport pointe vers notre dossier web. (Testez !)

Il nous faut installer un certificat pour obtenir une connexion sécurisée en HTTPS.

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
/etc/ssl/private/esport.key -out /etc/ssl/certs/esport.crt
```

Cette commande va créer 2 certificats dans les emplacements : /etc/ssl/private/esport.key et /etc/ssl/certs/esport.crt.

Il faut maintenant installer les certificats sur le serveur et les associés à notre domaine.

Copions le fichier de base default-ssl.conf et renommons le en esport-ssl.conf.

```
cp /etc/apache2/sites-available/default-ssl.conf /etc/apache2/sites-available/esport-
ssl.conf
```

Il s'agit simplement d'un VirtualHost qui est chargé de rediriger les connexions entrantes

provenant du port 443, le port dédié au protocole `HTTPS`.

Editons ce fichier :

```
nano /etc/apache2/sites-available/esport-ssl.conf
```

[image]

Pour tester notre configuration, il faut executer la commande :

```
apachectl configtest
```

Et si tout ce passe bien, la réponse devrait être :

```
# apachectl configtest
Syntax OK
```

Chargeons le module SSL dans apache pour qu'il puisse prendre en compte les connexions HTTPS et les certificats.

```
a2enmod ssl
```

Chargeons aussi le nouveau `VirtualHost` :

```
a2ensite esport-ssl
```

En test l'adresse https://necromancers.esport https://necromancers.esport depuis votre navigateur, Vous devriez avoir cela :

[image]

Il faut autoriser la connexion au site :

[image]

Note

Pourquoi nous avons ce message d'alerte ?

Tout simplement parce que le navigateur a détecté que nous sommes connecté avec le protocole `HTTPS`. Notre serveur lui a fourni un certificat … qui est … **autosigné** ! **Cela alerte donc le navigateur**.

Nous voulons que si l'utilisateur tape `HTTP` dans l'adresse au lieu de `HTTPS` le serveur puisse le rediriger automatiquement.

Activons le mode `rewrite` de Apache qui permet à Apache de réécrire/reformater les URL captées :

```
a2enmod rewrite
```

Et éditons le fichier

```
nano /etc/apache2/sites-available/esport.conf
```

Ajoutons cette régle de réécriture d'url :

```
RewriteEngine On
RewriteRule ^(.*)$ https://%{HTTP_HOST}$1 [R=301,L]
```

Redémarrons Apache :

```
service apache2 restart
```

Notre serveur est maintenant correctement configuré !

Note

Vous pouvez être fier du travail accompli jusqu'alors ! Et pourquoi ne pas créer une image basée sur cette configuration ? Afin de pouvoir créer une infinité de container avec les même caractéristiques. Cela évitera de recommencer toutes les étapes que nous avons suivies jusqu'alors.

Création d'une image Docker

Nous avons jusqu'alors créé des containers à partir d'images de bases que nous avons modifié. Il temps de créer notre propre image qui servira de "moule" pour des containers ayant besoin des caractéristiques que nous avons paramétrées.

Mais avant faisons un peu de ménage dans notre container. Supprimons le fichier `index.html` du dossier `/var/www/html/esport`

```
rm /var/www/html/esport/index.html
```

La commande pour créer une nouvelle image à partir d'un container est :

`docker commit <CONTAINER_ID> <NOM_DE_L_IMAGE>`

Il nous faut donc récupérer l'identifiant de notre container dans un premier temps :

```
docker ps -a
```

[image]

`Serveur_https` possède bien l'identifiant : `00e15c9f63ea`

Maintenant, nous pouvons créer une nouvelle image à partir de cet identifiant. Nous respecterons les conventions de nommage : `<Nom du constructeur>` / `<Nom de l'image>` : `<Numéro de version>`.

Notre image s'appelera alors : `siolaon/https:1.0`.

Lançons la création de l'image avec l'option `-a` pour définir le nom de l'auteur, mettez le votre car vous l'avez bien mérité:

```
docker commit -a Bauer 00e15c9f63ea siolaon/https:1.0
```

Vérifions si l'image a bien été créée en listant les images disponibles sur notre machine hôte.

```
docker images
```

[image]

Nous pouvons retrouver l'image également dans l'application Docker Desktop, onglet "Images".

[image]

Stoppons maintenant notre container `serveur_https` :

```
docker container stop serveur_https
```

Maintenant, voici venu le grand moment tant attendu ! Celui de monter notre image, dans un nouveau container avec le dossier web esport !

Positionnons nous dans le répertoire contenant notre dossier `www`, pour ma part:

```
cd C:\Users\baptiste\Documents\docker\td\www
```

```
docker container run -itd --name server_esport -v $PWD/esport:/var/www/html/esport -p
80:80 -p 443:443  siolaon/https:1.0
```

Maintenant il faut lancer le serveur apache2 manuellement depuis le serveur :

```
docker container exec -ti server_esport sh
```

et dans le `shell` lancer la commande :

```
service apache2 start
```

Ouvrez le navigateur et contemplez votre oeuvre :

[image]

12.3. Création d'un Dockerfile

Nous sommes satisfaits du résultat, mais il reste un goût d'inachevé, n'est-ce pas ?

Créer un container à partir de notre image, et devoir lancer la commande `service apache2 start` à partir de son `shell`, demande une manipulation dont on aimerait pouvoir se passer ...

Cela va être possible en créant un fichier `Dockerfile`. Ce fichier contient une liste de commande à exécuter pour concevoir notre propre image.

Listons les actions effectuées dans la partie **13.2**

- Création d'un container avec une image `Ubuntu`.
- Nous avons mis à jour les dépôts `Ubuntu`.
- Nous avons installé `Apache2`.
- Nous avons installé `Nano`.
- Nous avons installé `OpenSSL` et récupéré 2 fichiers : `esport.key` et `esport.crt`.
- Nous avons créé 2 fichiers **VirtualHost** `esport` et `vesport-ssl` pour le site en **http** et **https**.
- Nous avons activé les modules `ssl` et `rewrite` dans **Apache**.
- Nous avons chargé les **VirtualHost** `esport` et `esport-ssl` dans **Apache**.
- Nous avons redémarré **Apache** pour que les modifications soient prises en compte.
- Nous avons lancé **Apache**.

Il va falloir créer un dossier nommé par exemple : `esport_image`, qui contiendra :

[image]

- Notre dossier **esport**, avec dedans les pages html.
- Nos fichiers **VirtualHost** déjà rédigés qui seront ensuite copiés dans **Apache** automatiquement : `esport.conf` et `esport-ssl.conf`.
- Un fichier `Dockerfile`, fichier spécial composé des commandes à envoyer au **Daemon Docker** afin de générer une nouvelle image **Docker** conforme à nos objectifs.

Warning

Le fichier `Dockerfile` n'a pas d'extension.

Créez 2 fichiers : `esport.conf` et `esport-ssl.conf`. Dont le contenu est :

Fichier : `esport.conf`

```
<VirtualHost *:80>
    ServerName necromancers.esport
    ServerAlias www.necromancers.esport
    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html/esport
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
    RewriteEngine On
    RewriteRule ^(.*)$ https://%{HTTP_HOST}$1 [R=301,L]
</VirtualHost>
```

Fichier : `esport-ssl.conf`

```
<VirtualHost *:443>
    ServerAdmin webmaster@localhost
    ServerName necromancers.esport
    ServerAlias www.necromancers.esport
    DocumentRoot /var/www/html/esport
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
    SSLEngine on
    SSLCertificateFile  /etc/ssl/certs/esport.crt
    SSLCertificateKeyFile /etc/ssl/private/esport.key
    <FilesMatch "\.(cgi|shtml|phtml|php)$">
        SSLOptions +StdEnvVars
    </FilesMatch>
    <Directory /usr/lib/cgi-bin>
        SSLOptions +StdEnvVars
    </Directory>
</VirtualHost>
```

Maintenant nous allons pouvoir rédiger notre fichier **Dockerfile** :

La première ligne doit contenir l'instruction `FROM` qui définie l'image qui servira de référence. Nous allons construire notre projet autour de la distribution linux **Ubuntu** dans sa dernière version.

```
FROM ubuntu:latest
```

La dernière ligne contiendra l'instruction `CMD`. Il s'agit de la commande à exécuter dès que notre container sera lancé. Nous voulons lancer apache par la commande : `service apache2 start`.

```
CMD ["service", "apache2", "start"]
```

Entre les deux, il faut maintenant programmer la mise en place de notre serveur WEB avec un

certificat SSL autosigné et les fichiers de notre projet dedans.

L'instruction `RUN` permet d'établir une liste de commandes à exécuter. Chaque instruction `RUN` créé une couche (layer) dans notre container. Donc au lieu de lancer une instruction `RUN` par commandes, nous allons les chaîner, grâce à l'opérateur logique `&&`.

Note

Chaîner 2 ou 3 ou 4 commandes peut vite créer une ligne extrêmement longue. Par soucis de lisibilité, il est bien de pouvoir sauter une ligne entre chaque commande. Mais le compilateur qui va se charger de créer l'image ne va pas comprendre, pour l'aider, il faut ajouter un `\` après notre opérateur logique.

Exemple : .. code-block:

```
RUN apt install apache2 -y && apt install openssl -y
```

deviendra sur 2 lignes :

```
RUN apt install apache2 -y && \
apt install openssl -y
```

Donc nous aurons une instruction `RUN` qui contiendra toutes les commandes que nous avons saisi.

```
ENV DEBIAN_FRONTEND=nonintercative
RUN apt update && \
    apt install apache2 -y && \
    echo 'ServerName localhost'  >> /etc/apache2/apache2.conf && \
    apt install openssl -y && \
    openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
/etc/ssl/private/esport.key -out /etc/ssl/certs/esport.crt -subj
"/C=FR/ST=AISNE/L=LAON/O=BTS SIO/OU=IT Department/CN=necromancers.esport" && \
    mkdir /var/www/html/esport
```

Si vous vous rappelez, lorsque nous avons créé nos certificats SSL, il y a eu une série de questions qui nous a été posée. Lors de la création de notre image, nous ne pourrons pas y répondre avec notre clavier, mais seulement grâce au paramètre saisie directement dans la commande : `-subj "/C=FR/ST=AISNE/L=LAON/O=BTS SIO/OU=IT Department/CN=necromancers.esport"`.

De même, Apache demande aussi durant son installation de lui donner des informations comme le continent et le pays dans lequel nous sommes. Pour éviter cette question, et nous bloquer durant la création de l'image, nous utiliserons la variable d'environnement `ENV DEBIAN_FRONTEND=nonintercative`. Grâce à elle, notre système d'exploitation Ubuntu cessera de nous poser des questions, et nous aurons la configuration par défaut des applications que nous installerons.

L'instruction `echo 'ServerName localhost' >> /etc/apache2/apache2.conf` ajoute au fichier de

configuration d'Apache la ligne `ServerName localhost` afin de nommer le serveur par défaut.

L'instruction `COPY` va se charger de copier : les fichiers de configuration Apache et HTML dans les bons emplacements du futur container.

```
COPY esport/ ${path}/esport
COPY esport.conf esport-ssl.conf /etc/apache2/sites-available/
```

Il faut maintenant activer les modes `Rewrite` et `SSL` d'Apache, et lui injecter nos fichiers `VirtualHost`.

```
RUN a2enmod ssl && \
    a2enmod rewrite && \
    a2ensite esport &&\
    a2ensite esport-ssl
```

L'instruction `EXPOSE` nous permettra de définir les ports utilisés par défaut par le container.

```
EXPOSE 80 443
```

Ainsi, notre fichier `Dockerfile` complet sera ainsi :

```
FROM ubuntu:latest
ENV DEBIAN_FRONTEND=nonintercative
ENV path /var/www/html/
RUN apt update && \
    apt install apache2 -y && \
    echo 'ServerName localhost'  >> /etc/apache2/apache2.conf && \
    apt install openssl -y && \
    openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
/etc/ssl/private/esport.key -out /etc/ssl/certs/esport.crt -subj
"/C=FR/ST=AISNE/L=LAON/O=BTS SIO/OU=IT Department/CN=necromancers.esport" && \
    mkdir ${path}/esport

COPY esport/ ${path}/esport
COPY esport.conf esport-ssl.conf /etc/apache2/sites-available/

RUN a2enmod ssl && \
    a2enmod rewrite && \
    a2ensite esport &&\
    a2ensite esport-ssl

EXPOSE 80
CMD ["service", "apache2", "start"]
```

Nous avons rajouté une variable `ENV` nommée `path` qui nous permet de définir un chemin qui est utilisé plusieurs fois. Cette variable est utilisée grâce à cette notation `${path}`.

Il est temps maintenant, de créer notre image à partir de notre fichier `Dockerfile`.

Placez vous dans le dossier contenant ce fichier :

Pour ma part mon fichier `Dockerfile`, se trouve dans le dossier : `C:\Users\p02\Documents\Cours\docker`

```
cd C:\Users\p02\Documents\Cours\docker
```

Créons maintenant notre image nommée esport dans sa version 1.0. La création peut prendre un certain temps.

Warning

N'oubliez pas le "." !

```
docker image build -t esport:1.0 .
```

[image]

Notre image apparait bien dans Docker Desktop.

[image]

Maintenant, montons un container basée sur cette image.

Stopez tout les containers en cours d'exécution afin d'éviter que le port 80 soit déjà utilisé.

```
docker container stop $(docker container ls -q)
```

Puis :

```
docker container run -tid --name site_necroteam -p 80:80 esport:1.0 sh
```

12.4. CICD dans un Swarm, avec GitLab et Portainer

Dans cette activité, vous allez mettre en place un pipeline d'intégration continue / déploiement continu en utilisant différents composants : Docker Swarm, GitLab, Portainer

Pour faire cet exercice dans sa totalité, il sera nécessaire de créer une VM accessible depuis internet (étape 4). Pour l'ensemble des cloud providers (Google Compute Engine, Amazon AWS, Packet, Rackspace, DigitalOcean, Scaleway, ...) l'instantiation de VMs est payante (peu cher pour un test de quelques heures cependant). Si vous ne souhaitez pas réaliser la manipulation jiusqu'au bout, n'hésitez pas à suivre cet exercice sans l'appliquer, l'essentiel étant de comprendre le flow global.

12.4.1. Les étapes

Vous allez suivre les différentes étapes ci-dessous :

1. Création, dans le langage de votre choix, d'un serveur web très simple
2. Ajouter un Dockerfile et création d'une image de l'application
3. Création d'un répository sur GitLab
4. Mise en place d'un cluster Swarm
5. Mise en place d'un pipeline d'intégration continue et déploiement continu

---

12.4.2. 1ère étape: Création d'un serveur web

En utilisant le langage de votre choix, développez un serveur web simple ayant les caractéristiques suivantes: - écoute sur le port 8000 - expose le endpoint / en GET - retourne la chaine 'Hi!' pour chaque requète reçue.

vous pouvez utiliser l'un des templates ci-dessous, réalisé dans différents langages:

- NodeJs
- Python
- Ruby
- Go
- Java
- .net core

12.4.2.1. NodeJs

12.4.2.1.1. Installation

Suivez les instructions suivantes pour installer l'interpréteur NodeJS: https://nodejs.org/

12.4.2.1.2. Code source

- index.js

```
var express = require('express');
var app = express();
app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.end("Hi!");
});
app.listen(8000);
```

- package.json

```
{
  "name": "www",
  "version": "0.0.1",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.14.0"
  }
}
```

12.4.2.1.3. Installation de expressjs

```
$ npm install
```

12.4.3. Lancement du serveur

```
$ npm start
```

12.4.4. Test

```
$ curl localhost:8000
Hi!
```

12.4.4.1. Python

12.4.4.1.1. Installation

Suivez les instructions suivantes pour installer l'interpréteur Python: https://www.python.org/downloads/

12.4.4.1.2. Code source

- app.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
```

```
    return "Hi!"

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8000)
```

- requirements.txt

```
Flask==2.0.2
```

12.4.4.1.3. Installation des dépendances

```
$ pip install -r requirements.txt
```

12.4.4.1.4. Lancement du serveur

```
$ python app.py
```

12.4.4.1.5. Test

```
$ curl localhost:8000
Hi!
```

12.4.4.2. Ruby

12.4.4.2.1. Installation

Suivez les instructions suivantes pour installer l'interpréteur Ruby:

https://www.ruby-lang.org/fr/documentation/installation/

12.4.4.2.2. Code source

- app.rb

```
require 'sinatra'
set :port, 8000
get '/' do
  'Hi!'
end
```

- Gemfile

```
source :rubygems
gem "sinatra"
```

12.4.4.2.3. Installation des dépendances

```
$ bundle install
```

12.4.4.2.4. Lancement du serveur

```
$ ruby app.rb -s Puma
```

12.4.4.2.5. Test

```
$ curl localhost:8000
Hi!
```

12.4.4.3. Go

12.4.4.3.1. Installation

Suivez les instructions suivantes pour installer le compilateur GO: https://golang.org/doc/install

12.4.4.3.2. Code source

- main.go

```
package main

import (
        "io"
        "net/http"
)

func handler(w http.ResponseWriter, req *http.Request) {
    io.WriteString(w, "Hi!")
}

func main() {
        http.HandleFunc("/", handler)
        http.ListenAndServe(":8000", nil)
}
```

12.4.4.3.3. Lancement du serveur

```
$ go run main.go
```

12.4.4.3.4. Test

```
$ curl localhost:8000
Hi!
```

12.4.4.4. Java / Spring

12.4.4.4.1. Installation

Suivez les instructions suivantes pour installer le compilateur Java: https://java.com](https://java.com

L'application de test est générée depuis [](https://start.spring.io/) en utilisant les options suivantes:

![Spring generator](./images/spring-generator.png)

12.4.4.4.2. Packaging de l'application

```
$ ./mvnw package
```

12.4.4.4.3. Lancement du serveur

```
$ java -jar target/demo-0.1.jar
```

12.4.4.4.4. Test

```
$ curl http://localhost:8080/
hello World
```

12.4.4.5. DotNetCore

12.4.4.5.1. Création du projet

Générez un quelette de projet DotNetCore avec la commande suivante:

```
$ dotnet new webapi -o webapi
```

12.4.4.5.2. Compilation

```
$ cd webapi
$ dotnet restore
$ dotnet publish -c Release -o out
```

12.4.4.5.3. Lancement du serveur

```
$ dotnet out/webapi.dll
```

12.4.4.5.4. Test

```
$ curl https://localhost:5000/api/values
["value1","value2"]
```

Dans la partie suivante, vous allez packager l'application dans une image Docker.

12.4.5. 2ème étape : Création d'une image Docker

12.4.5.1. Ajout d'un fichier Dockerfile

Ajoutez un Dockerfile à la racine du répertoire contenant le code source du serveur web. Vous vous servirez ensuite de ce Dockerfile pour créer une image contenant le code de votre application et l'ensemble des dépendances qui sont nécessaires pour la faire tourner.

Vous trouverez ci-dessous des exemples de Dockerfile pour un serveur web écrit dans différents langages.

Important: essayez tout d'abord d'écrire votre propre version du Dockerfile avant d'utiliser l'un de ceux ci-dessous :)

12.4.5.1.1. Exemple de Dockerfile pour un serveur web écrit en Java

```
FROM openjdk:8-jdk
COPY . /
RUN ./mvnw --batch-mode package

FROM openjdk:8-jdk
COPY --from=0 target/*.jar app.jar
ENV SERVER_PORT=8000
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
EXPOSE 8000
```

12.4.5.1.2. Exemple de Dockerfile pour un serveur web écrit en Python

```
FROM python:3
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 8000
CMD python /app/app.py
```

12.4.5.1.3. Exemple de Dockerfile pour un serveur web écrit ern Node.Js

```
FROM node:12-alpine
COPY . /app
WORKDIR /app
RUN npm i
EXPOSE 8000
CMD ["npm", "start"]
```

12.4.5.1.4. Exemple de Dockerfile pour un serveur web écrit en Go

```
FROM golang:1.12-alpine as build
WORKDIR /app
COPY main.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

FROM scratch
COPY --from=build /app/main .
CMD ["./main"]
```

12.4.5.2. Construction de l'image

Une fois que vous avez créé votre Dockerfile, construisez une image, nommé **api** en utilisant la commande suivante:

```
$ docker build -t api .
```

12.4.5.3. Lancement d'un container

Vérifiez à présent que l'application tourne correctement lorsqu'elle est lancée dans un container (en publiant le port 8000).

```
$ docker run -ti -p 8000:8000 api
```

Vérifiez ensuite que le serveur web est accessible sur le port **8000** depuis la machine locale.

```
$ curl http://localhost:8000
```

Vous pouvez ensuite stopper le container et passer à l'étape suivante.

12.4.6. 3ème étape: repository GitLab

12.4.6.1. GitLab.com

Créez un compte depuis https://gitlab.com, ou bien utilisez votre propre compte si vous en avez déjà un.

[image148] | *../images/image148.png*

Important: assurez-vous de copier votre clé publique ssh dans la configuration de votre compte GitLab. Dans la suite, cela permettra d'uploader le code de votre serveur sans avoir à renseigner votre login et mot de passe à chaque fois.

12.4.6.2. Création d'un projet

Une fois que vous êtes connecté sur https://gitlab.com, créez un nouveau projet en lui donnant le nom que vous souhaitez.

Assurez vous de sélectionner **Public** dans la rubrique **Visibility Level**.

[image149] | *../images/image149.png*

12.4.6.3. Envoi du code de l'application

Suivez les instructions détaillées dans la section **Push an existing folder** afin d'envoyer le code de votre application dans le projet que vous venez de créer.

Dans l'étape suivante, vous allez mettre en place un cluster Docker Swarm dans lequel vous déploierez votre serveur.

12.4.7. 4ème étape: Cluster Swarm

12.4.7.1. Création d'un hôte Docker

Vous aurez besoin dans cette partie d'une machine virtuelle accessible depuis Internet. Pour quelques euros vous pourrez par exemple en créer une sur un cloud provider tel que DigitalOcean, OVH, Scaleway, Google Compute Plateform, Amazon AWS, ...

Une fois la VM mise en place, connectez vous sur celle-ci en ssh et installer Docker avec la commande suivante:

```
$ curl https://get.docker.com | sh
```

12.4.7.2. Initialisation d'un cluster Swarm

Initialisez ensuite un Docker Swarm avec la commande suivante:

```
$ docker swarm init
```

> Note: si vous obtenez une erreur similaire à: "Error response from daemon: could not choose an IP address to advertise since this system has multiple addresses on interface eth0 (178.62.15.73 and 10.16.0.7) - specify one with --advertise-addr", relancez la commande d'Initialisation en donnant l'adresse IP externe de la VM

12.4.7.3. Installation de Portainer

Toujours depuis le shell précédent, installez l'outils Portainer en tant que Docker Stack sur le Swarm. Portainer vous permettra de gérer votre Swarm avec une interface web très intuitive.

```
$ curl -L https://downloads.portainer.io/portainer-agent-stack.yml -o portainer-agent-stack.yml
$ docker stack deploy --compose-file=portainer-agent-stack.yml portainer
```

Vérifiez ensuite que Portainer est bien disponible en lançant un navigateur sur l'adresse suivante:

http://VM_IP_ADDRESS:9000

[image150] | *../images/image150.png*

Suivez les instructions et définissez un mot de passe pour l'utilisateur **admin**. Vous aurez ensuite accès au dashboard de Portainer.

[image151] | *../images/image151.png*

Cliquez sur le cluster qui est présenté, vous verrez alors l'ensemble des éléments qui tournent dans le Swarm (seul Portainer est présent pour le moment)

[151] | *../images/151.png*

Dans l'étape suivante, vous allez mettre en place un pipeline de CICD

12.4.8. 5ème (et dernière) étape: CICD

12.4.8.1. Utilisation de GitLab CI

A la racine du répertoire contenant les sources de votre projet, créez un fichier `.gitlab-ci.yml` contenant les instructions suivantes:

```
stages:
  - package

push image docker:
  image: docker:stable
  stage: package
  services:
    - docker:18-dind
```

```
  script:
    - docker build -t $CI_REGISTRY_IMAGE .
    - docker login -u gitlab-ci-token -p $CI_BUILD_TOKEN $CI_REGISTRY
    - docker push $CI_REGISTRY_IMAGE
```

Ce fichier définit une étape nommée **package** contenant un ensemble d'instructions permettant de builder une image de l'application et de la pusher dans le registry disponible dans GitLab.

Committez votre code et envoyez le dans le repository GitLab:

```
$ git add .gitlab-ci.yml
$ git commit -m 'Add GitLab pipeline'
$ git push origin master
```

Rendez-vous ensuite dans l'interface de GitLab et vérifiez, dans le menu **CI / CD**, qu'un premier pipeline est en train de tourner.

[152] | *../images/152.png*

Une fois celui-ci terminé, allez dans le menu **Packages → Container Registry** et vérifiez que la première image de votre application a bien été construite et qu'elle est disponible dans le registry.

[153] | *../images/153.png*

![GitLab pipeline](./images/gitlab_registry_1.png)

12.4.8.2. Ajout d'un test d'intégration

Dans le fichier `.gitlab-ci.yml`, ajouter une entrée **integration** sous la clé **stage**.

```
stages:
  - package
  - integration
```

A la fin du fichier, ajoutez également une nouvelle étape **integration test**:

```
integration test:
  image: docker:stable
  stage: integration
  services:
    - docker:18-dind
  script:
    - docker run -d --name myapp $CI_REGISTRY_IMAGE
    - sleep 10s
    - TEST_RESULT=$(docker run --link myapp lucj/curl -s http://myapp:8000)
    - echo $TEST_RESULT
    - $([ "$TEST_RESULT" == "Hello World!" ])
```

Cette étape définit un test qui sera effectué sur l'image créée dans l'étape précédente. Ce test très simple s'assure que la chaine de caractères retournée est "Hello World!".

Committez une nouvelle fois le code et envoyez le dans le repository GitLab.

```
$ git add .gitlab-ci.yml
$ git commit -m 'Add integration step'
$ git push origin master
```

Depuis l'interface de GitLab, vérifiez qu'un nouveau pipeline a été déclenché. Au bout de quelques dizaines de secondes, vous devriez voir que l'étape **integration** a échoué.

[154] | *../images/154.png*

Vérifiez le contenu des logs de l'étape d'intégration, cela vous permettra de corrigez le code. Vous pourrez ensuite commiter ce changement puis l'envoyer dans GitLab. Assurez-vous que le nouveau pipeline déclenché se déroule cette fois-ci sans erreur.

12.4.8.3. Creation d'un service

Vous allez à présent déployer votre application en tant que Service Docker. Utilisez pour cela la commande suivante en remplaçant: - GITLAB_USER par votre identifiant GitLab - GITLAB_REPO par le nom de votre projet GitLab

```
$ docker service create \
  --name hi \
  --publish 8080:8000 \
  registry.gitlab.com/GITLAB_USER/GITLAB_REPO:latest
```

Note: le port 8000 du service est publié sur le port 8080 sur le node du cluster Swarm car le port 8000 est déjà utilisé par Portainer.

12.4.8.4. Webhook Portainer

Depuis l'interface de Portainer, assurez-vous que le Service est actif:

[155] | *../images/155.png*

Depuis la page de détails du Service, activez le radio button **Service webhook**. Cela va générer une URL sur laquelle pourra être envoyé une requête HTTP POST afin de mettre à jour le Service.

[156] | *../images/156.png*

Copiez cette URL et utilisez cette valeur pour la variable **PORTAINER_WEBHOOK** que vous pourrez créer depuis le menu **Settings** > **CI / CD**.

[157] | *../images/157.png*

12.4.8.5. Déploiement automatique

L'étape suivante est d'ajouter une phase de déploiement, nommée **deploy** dans le fichier **.gitlab-ci.yml**. Ajoutez cette nouvelle entrée dans la liste **stages**:

```
stages:
  - package
  - integration
  - deploy
```

A la fin du fichier, ajoutez également l'étape **deploy-swarm** qui sera exécutée dans le contexte de cette phase **deploy**:

```
deploy-swarm:
  stage: deploy
  image: alpine:3.9
  script:
    - apk add --update curl
    - curl -XPOST $PORTAINER_WEBHOOK
```

La seule action de cette étape est d'appeler le webhook définit par Portainer en envoyant une requête POST sur l'URL associée au service.

Faite ensuite un changement dans le code de votre application, modifiez par exemple la chaine de caratères retournée (assurez-vous de modifier la description du test en même temps). Commitez ces changements et envoyer le code sur GitLab.

Vérifiez enfin que la nouvelle version de votre application a été correctement déployée.

12.5. En résumé

Cet exercice détaille les étapes que l'on peut suivre pour mettre en place un pipeline simple d'intégration continue / déploiement continu (CI/CD). De nombreux éléments pourraient être ajoutés afin d'améliorer ce pipeline…