Cours BTS SIO

Version 0.1

Bauer Baptiste

avr. 06, 2022

Systèmes

Vous trouverez sur cette page des cours en ligne pour le BTS SIO.

Note : Mise à jour : 24/02/2022

Ce dépôt est en cours de rédaction.

Auteur : **Bauer Baptiste**, Enseignant en BTS SIO au lycée Paul Claudel(02)

CHAPITRE 1

La plateforme Docker

1.1 1.0 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.

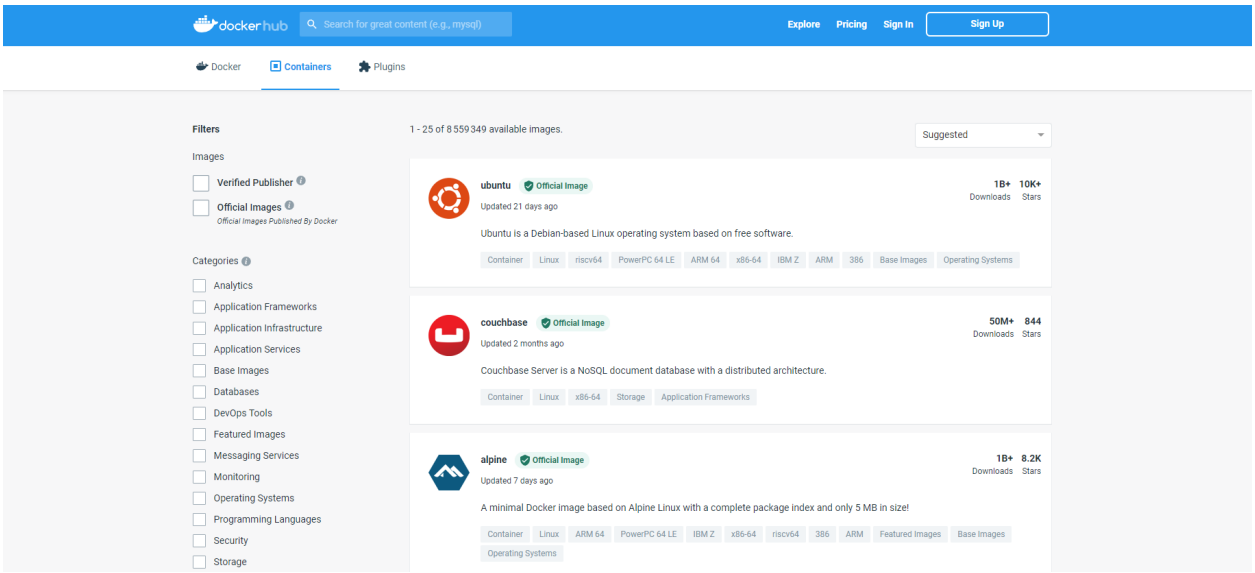
Nous aurons un chapitre sur la **plateforme Docker**, son architecture, son fonctionnement et sa mise en place. Nous verrons comment Docker rend très simple la manipulation des **containeurs**. Nous parlerons de la notion d'images qui permet de **packager** une application et ses dépendances. Dans le 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**.

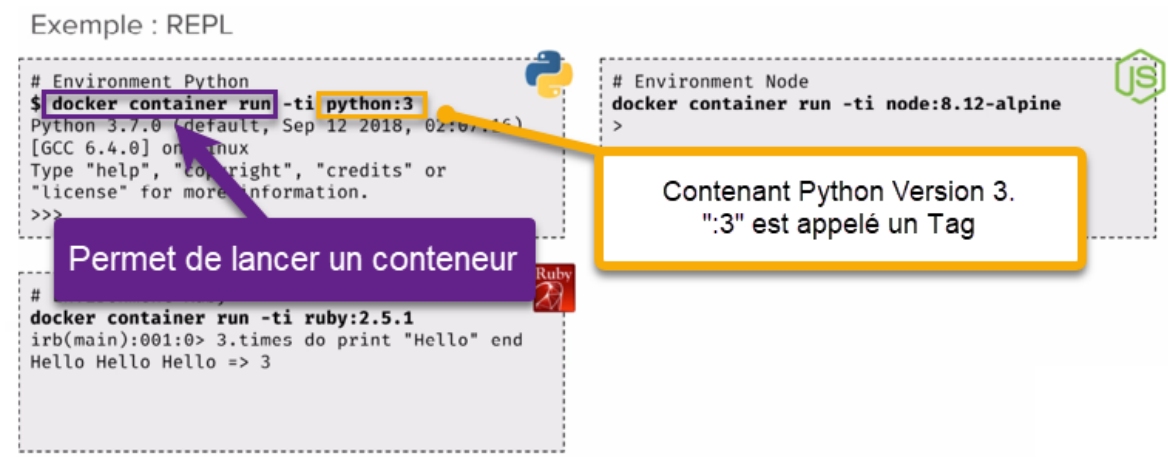
1.1.1 1.1 Pour qui ? pourquoi ?

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



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** :

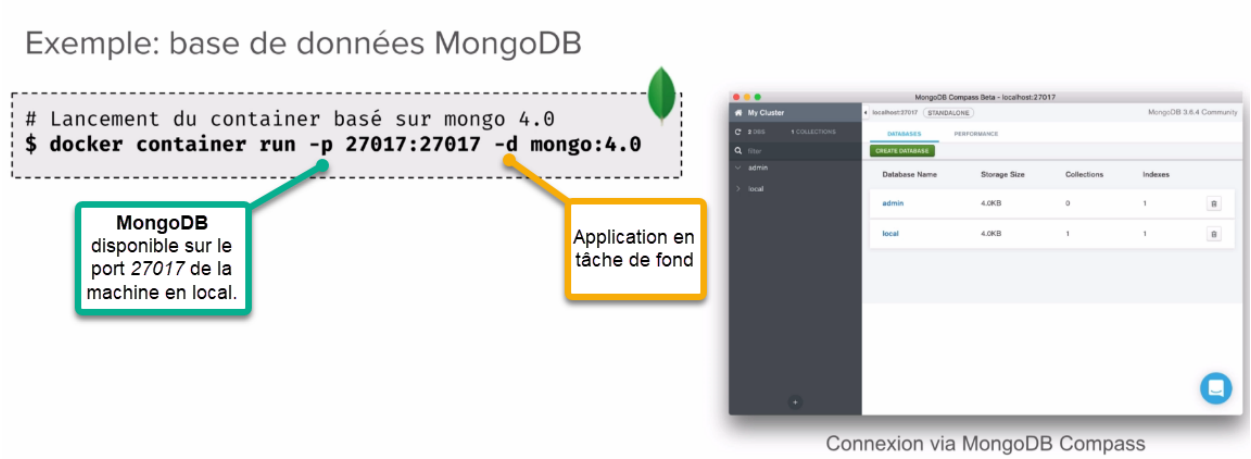
Par exemple, grâce à **Docker** nous pouvons lancer un interpréteur interactif (**REPL**) pour des langages de programmation comme le **Python**, le **Ruby On Rail** ou le **Javascript**.



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.

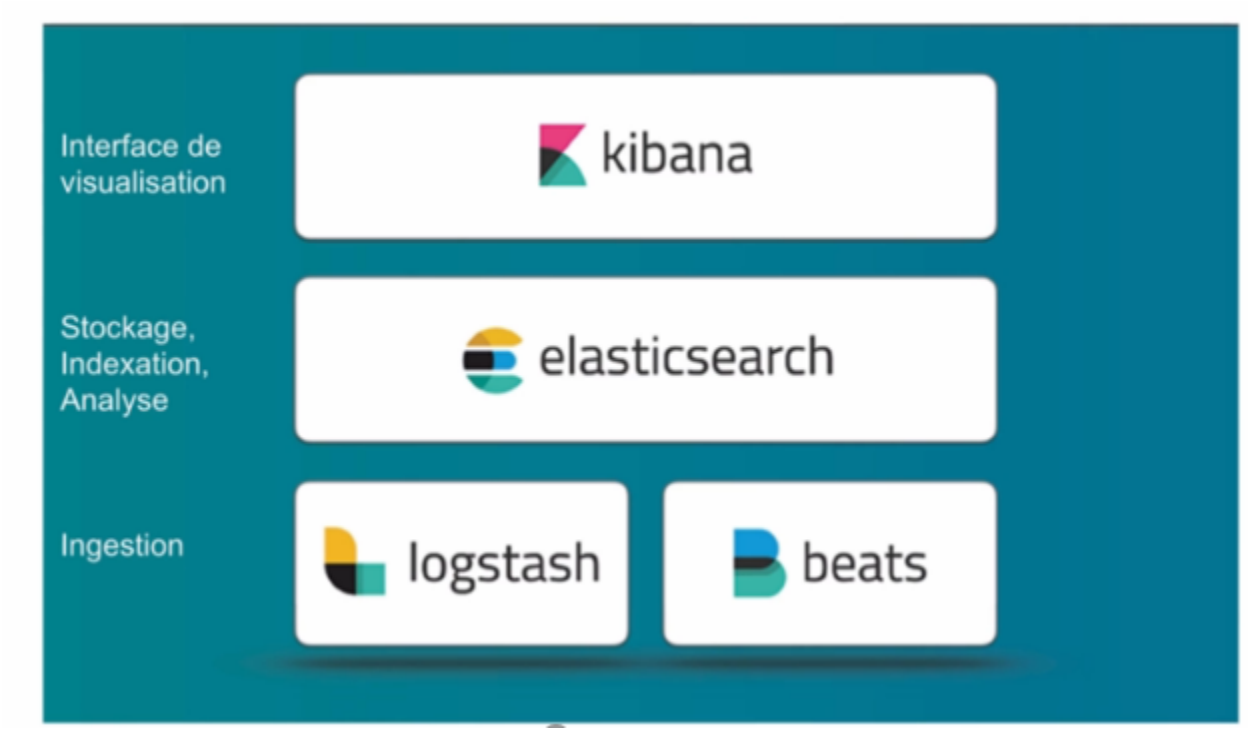
Par exemple si nous avons besoin d'une base de données **MongoDB** dans la version 4.0.Nous n'avons qu'à trouver une image disponible dans le **Docker Hub**.



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.

1.1.2 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.



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.

1.1.3 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 :

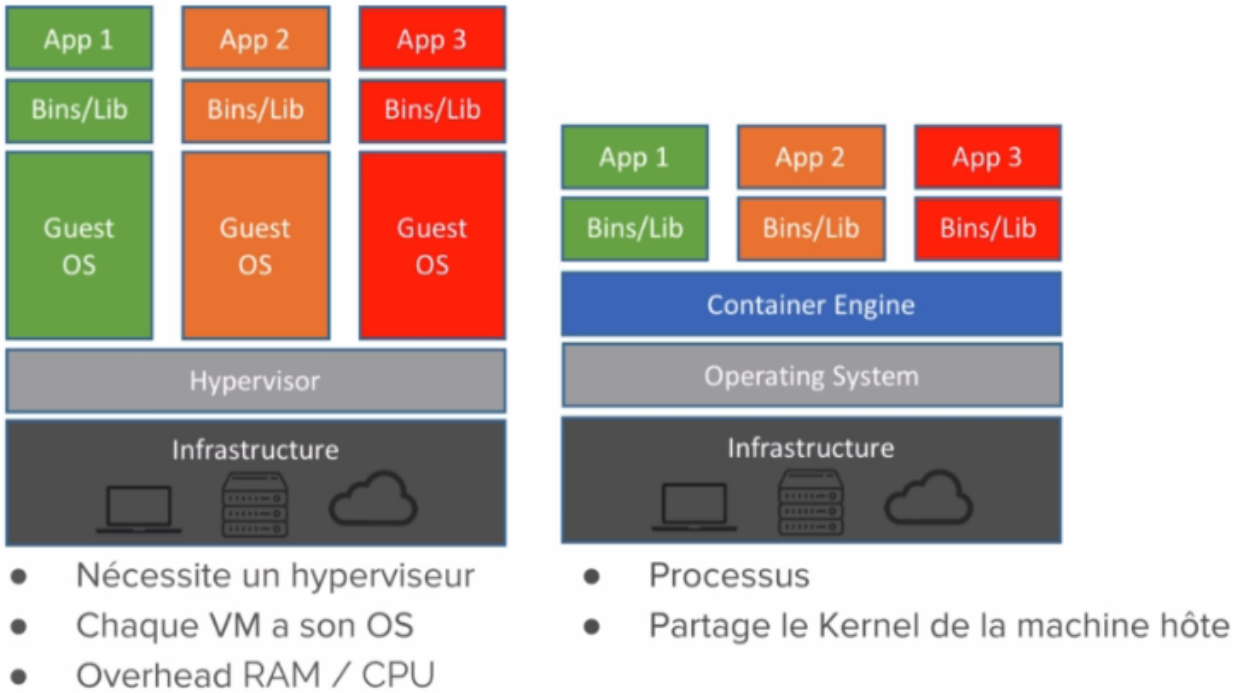
1. **Pid** : Permet de donner à un processus la vision de lui-même et de ses processus enfant.
2. **Net** : Permet de donner au processus son propre réseau privé.
3. **Mount** : Permet de donner au processus un système de fichiers privé.
4. **Uts** : Permet la gestion du nom de l'hôte.
5. **Ipc** : Isole 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



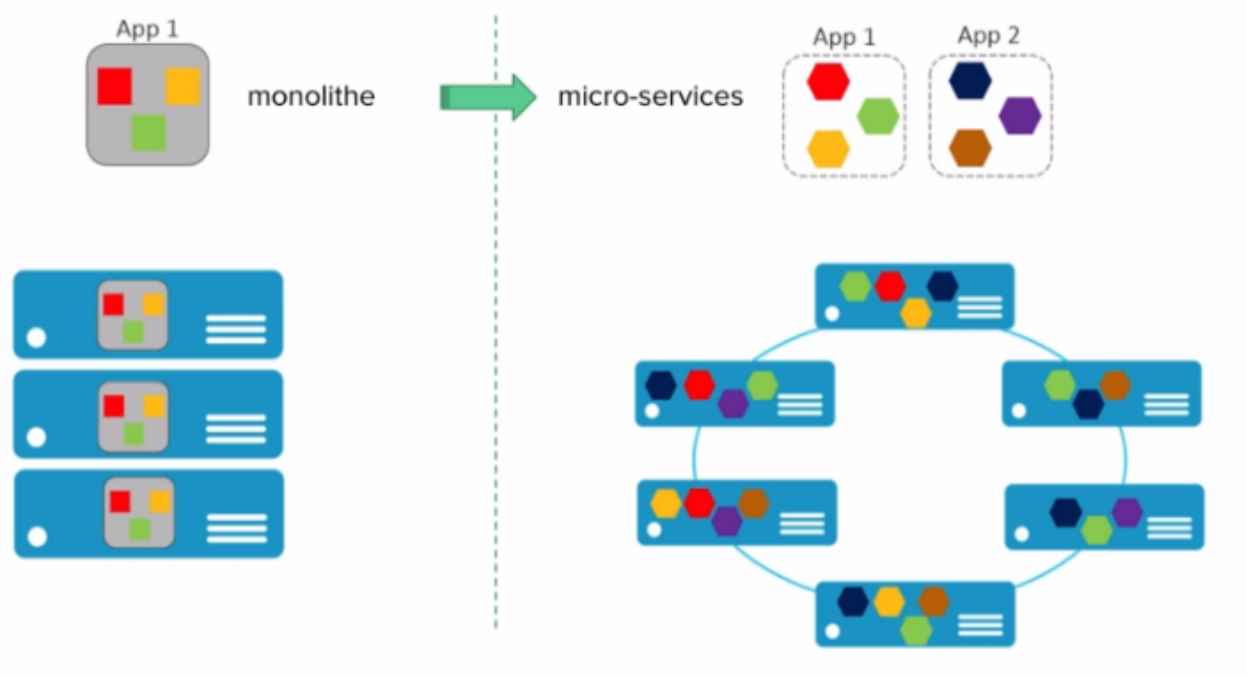
On compare souvent les containers à des machines virtuelles, car elles permettent d'exécuter des applications de manière isolée.

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.**

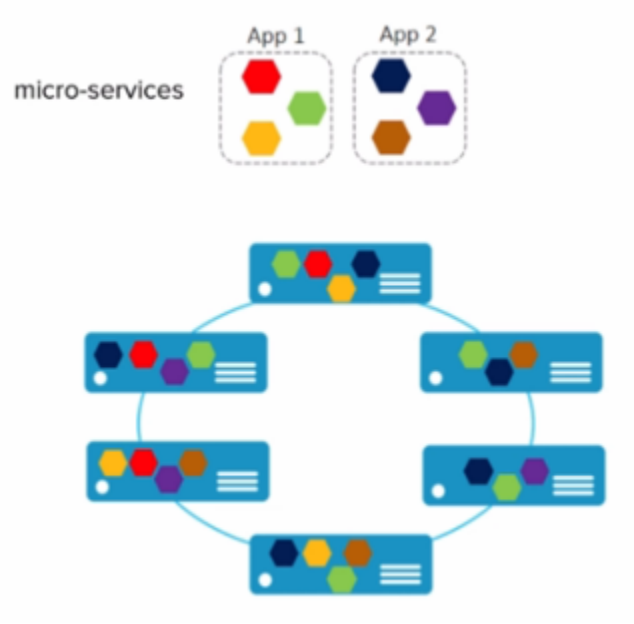
1.3.5 Architecture micro-services

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**.



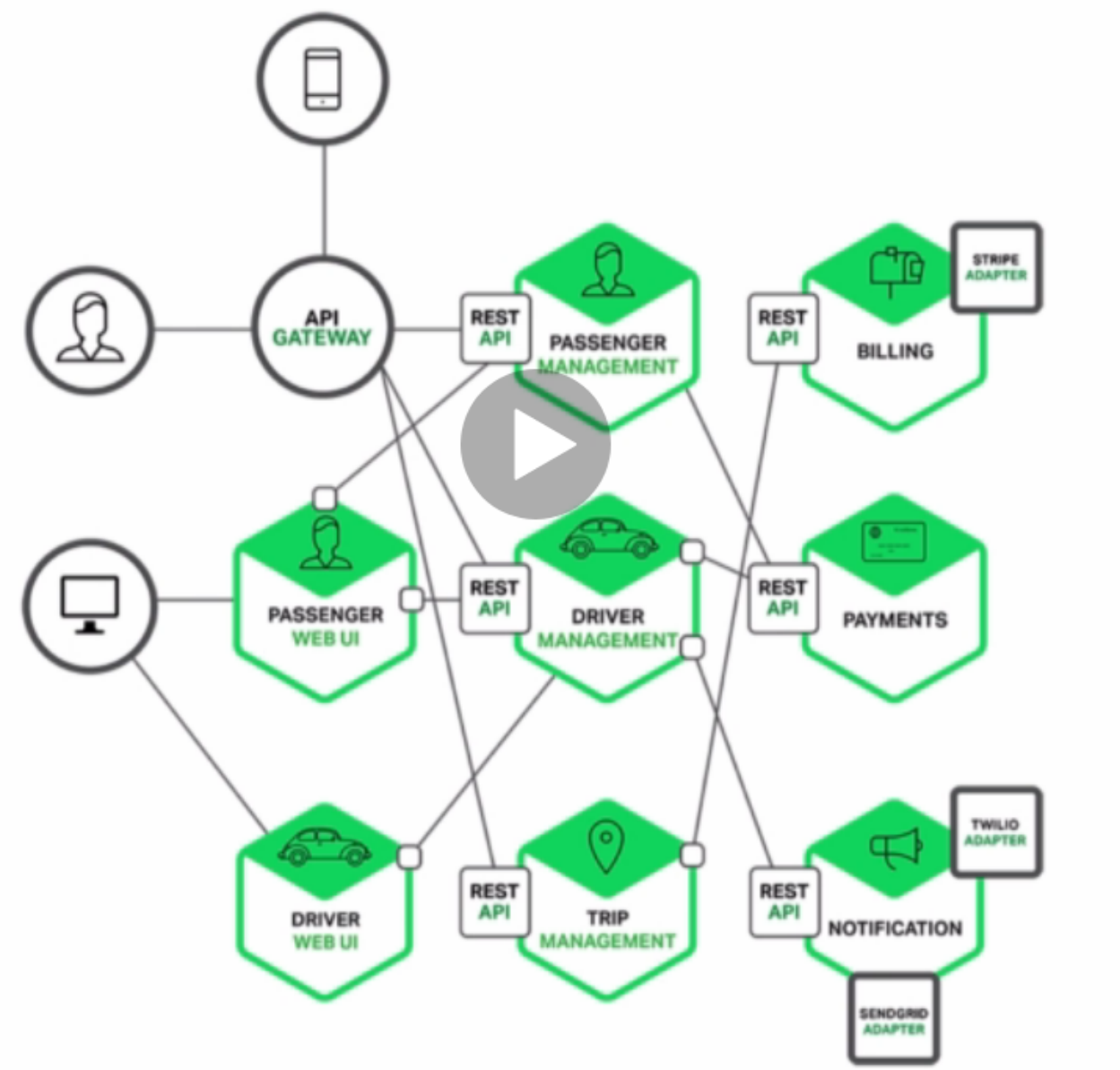
Aujourd'hui, une application est constituée de **plusieurs petits composants** qui sont des services qui ont leur propre rôle et fonctionnalité. Et c'est l'**interconnexion** de l'ensemble de ces services qui permettent de définir l'application globale.

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.



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



https://www.nginx.com/blog/introduction-to-microservices/

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

1.3.5 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

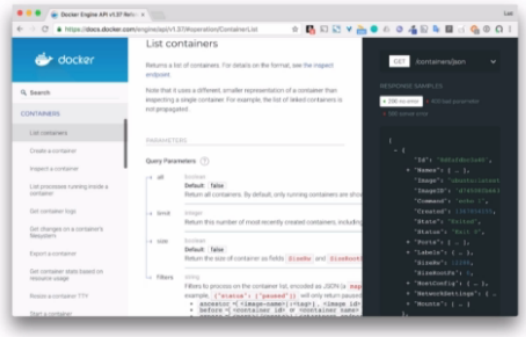
1.2 2.0 La plateforme Docker

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.

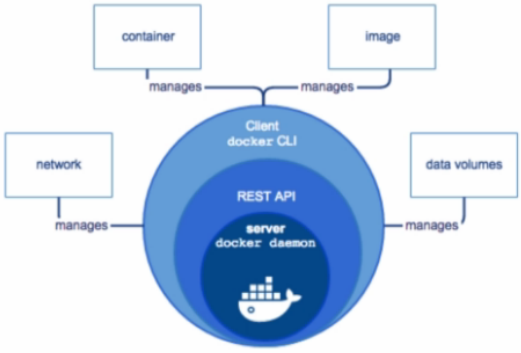
Note : Cherchez la définition du terme **scalabilité**.

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

1.2.1 2.1 Le modèle client/serveur



https://docs.docker.com/engine/api/v1.37/



https://docs.docker.com/

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.

2.1.1 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.

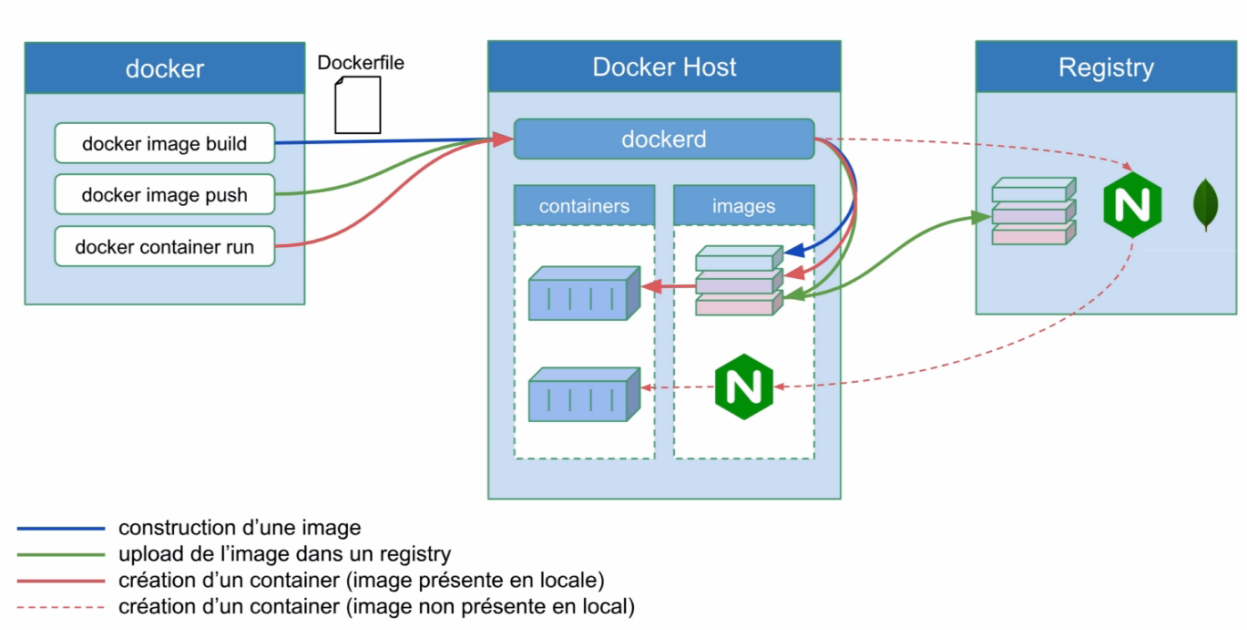
2.1.2 Le client : docker

- 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**.

2.1.3 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**.

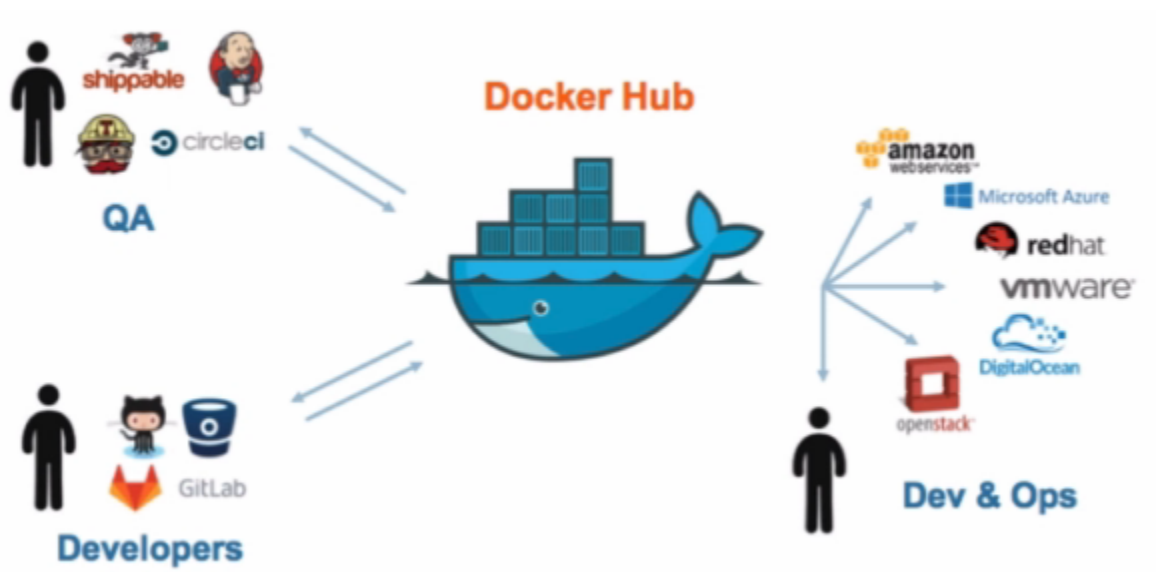


Quand on installe la plateforme Docker nous avons donc : un client et un serveur (ou daemon) qui tourne constamment et qui est responsable de la gestion des containers et des images.

2.1.4 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.

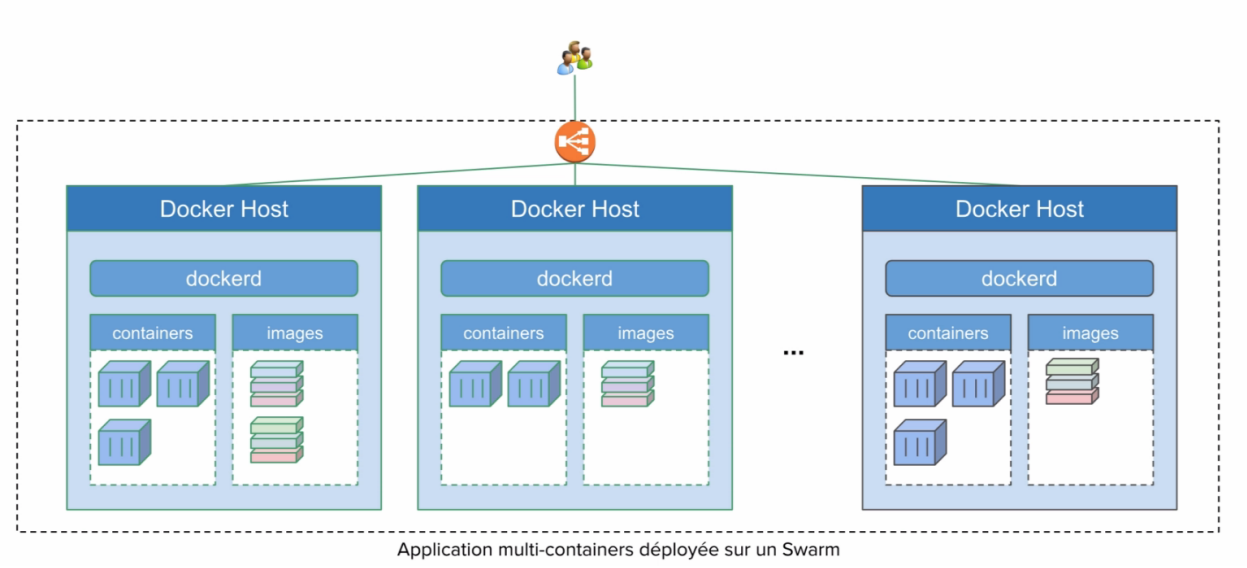


Les images du Docker Hub peuvent être classées en plusieurs catégories.

- 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).

2.1.4 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.

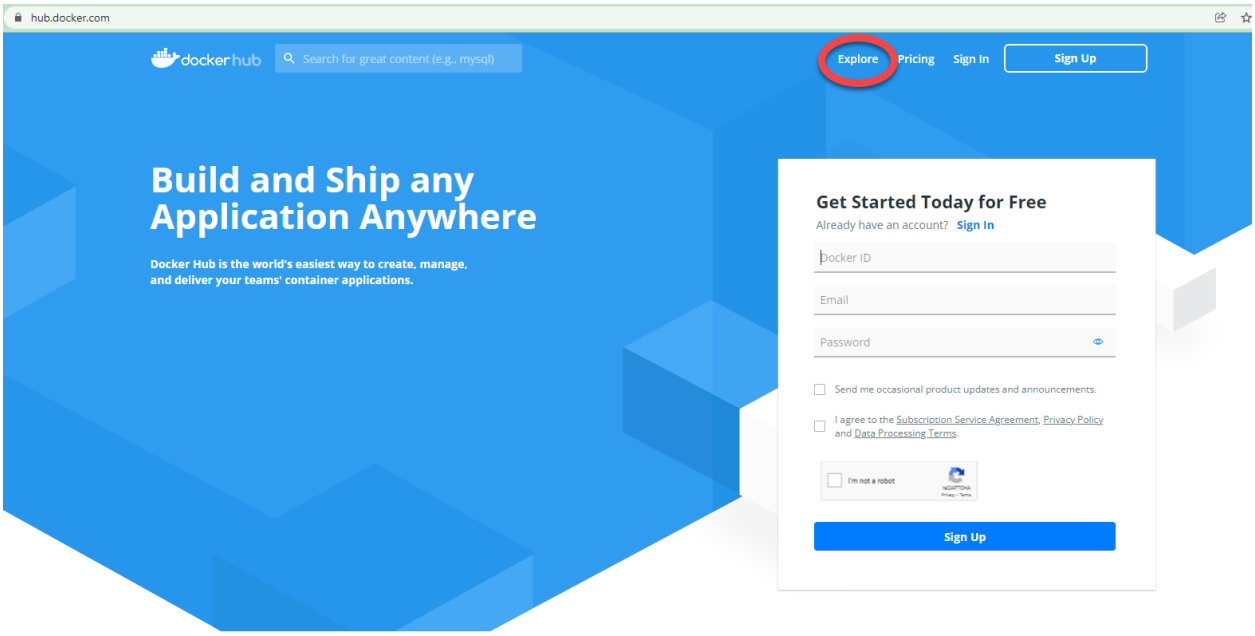


Ses machines vont communiquer entres elles afin d'orchestrer des applications et d'assurer qu'elles fonctionnent de la manière voulue.

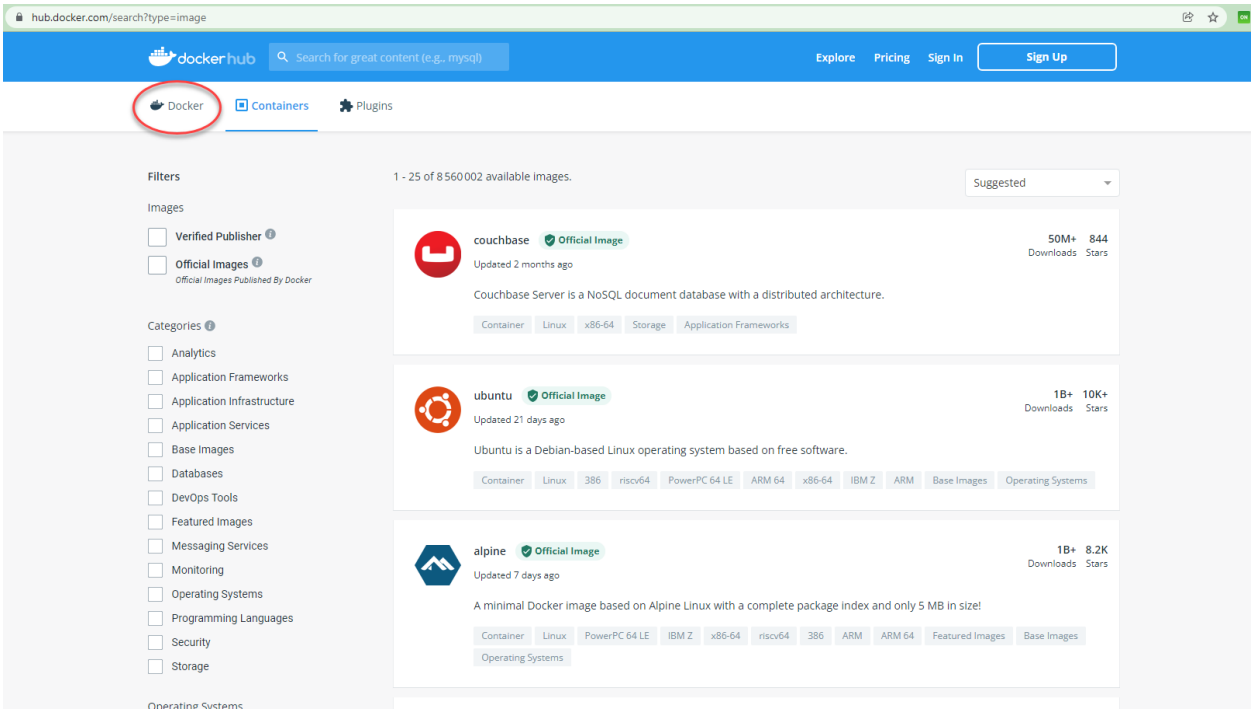
1.2.2 2.2 Installation de Docker

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

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



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



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

- plateforme
- système d'exploitation
- architecture

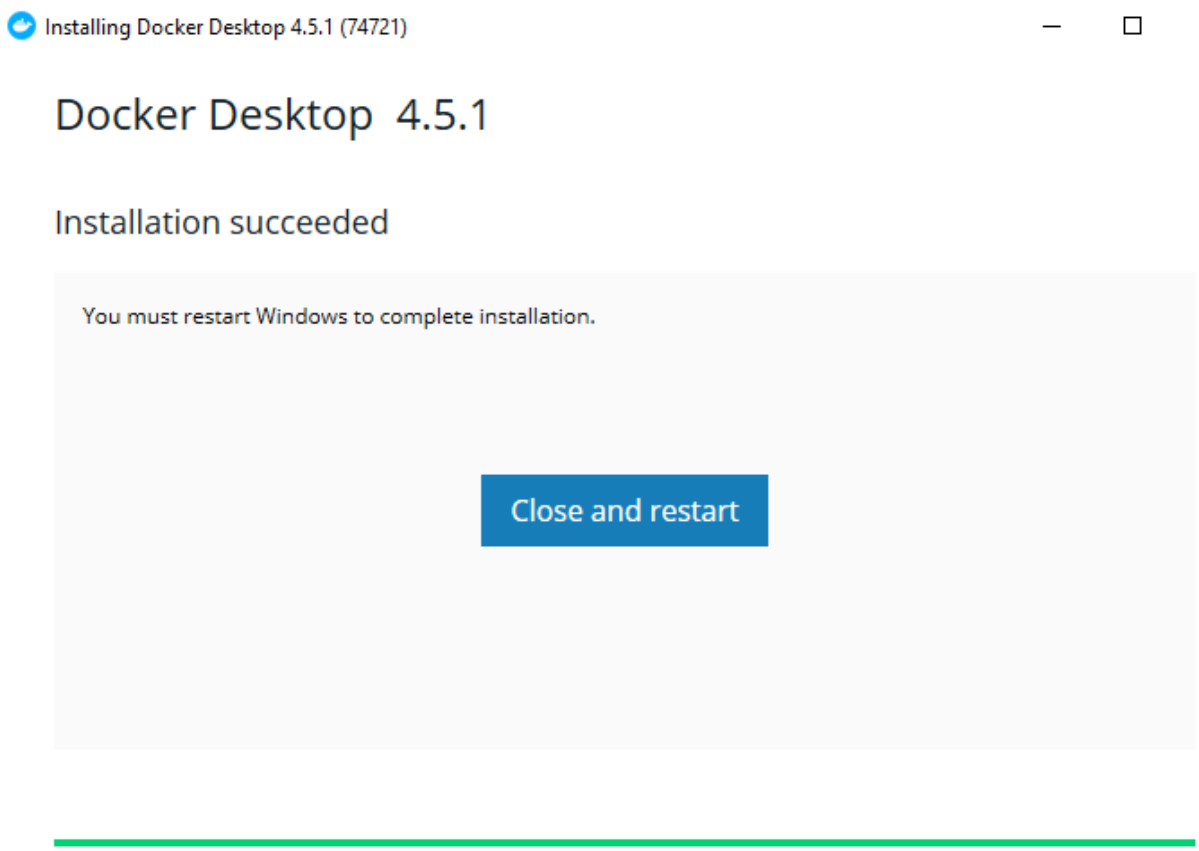
Comme nous pouvons le constater, Docker peut être installé sur des systèmes divers : machine de développement, l'infrastructure d'un cloud provider, et même des devices de type Raspberry PI.

1.2.3 Installation sur un poste élève du Lycée Paul Claudel -LAON (02)

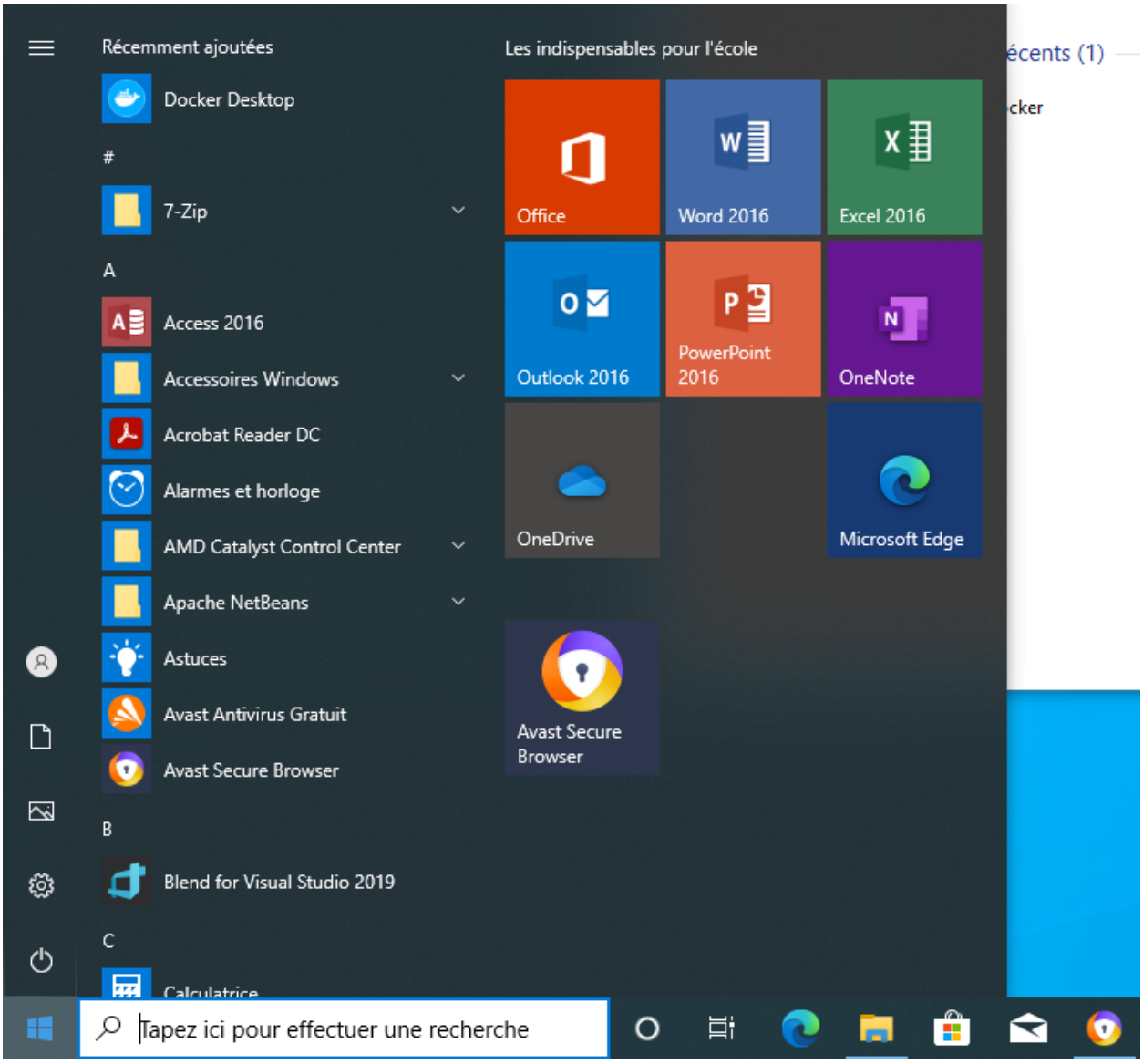
Normalement, il faudrait télécharger `Docker Desktop` depuis le site officiel. Mais pour économiser la bande passante, utilisez le fichier d'installation présent dans le répertoire `\\COMMUN\BAUER\Docker\`.

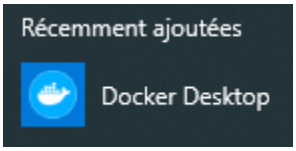
Doucle cliquez sur l'installateur et laissez les options d'installation cochées par défaut. `WSL 2` est nécessaire pour faire fonctionner **DOCKER**.

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

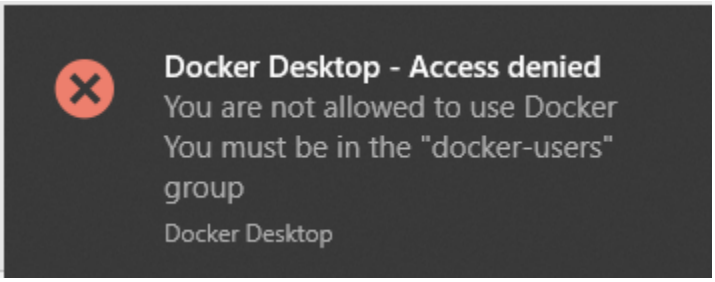


** Lancer l'application : Docker Desktop**





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



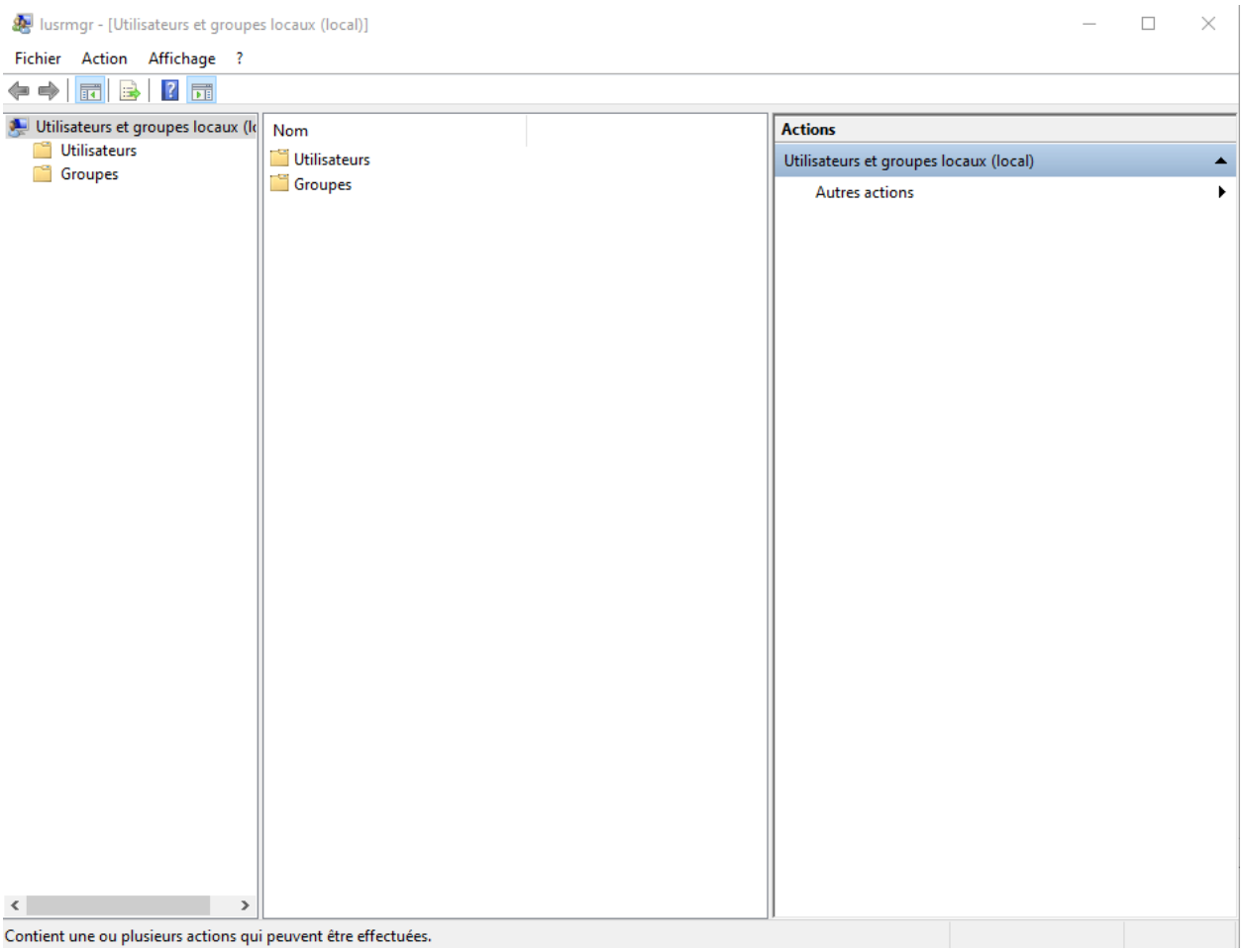
Pour résoudre ce problème, nous avons besoin d'ajouter les utilisateurs de la machine au groupe `docker-users` nou-

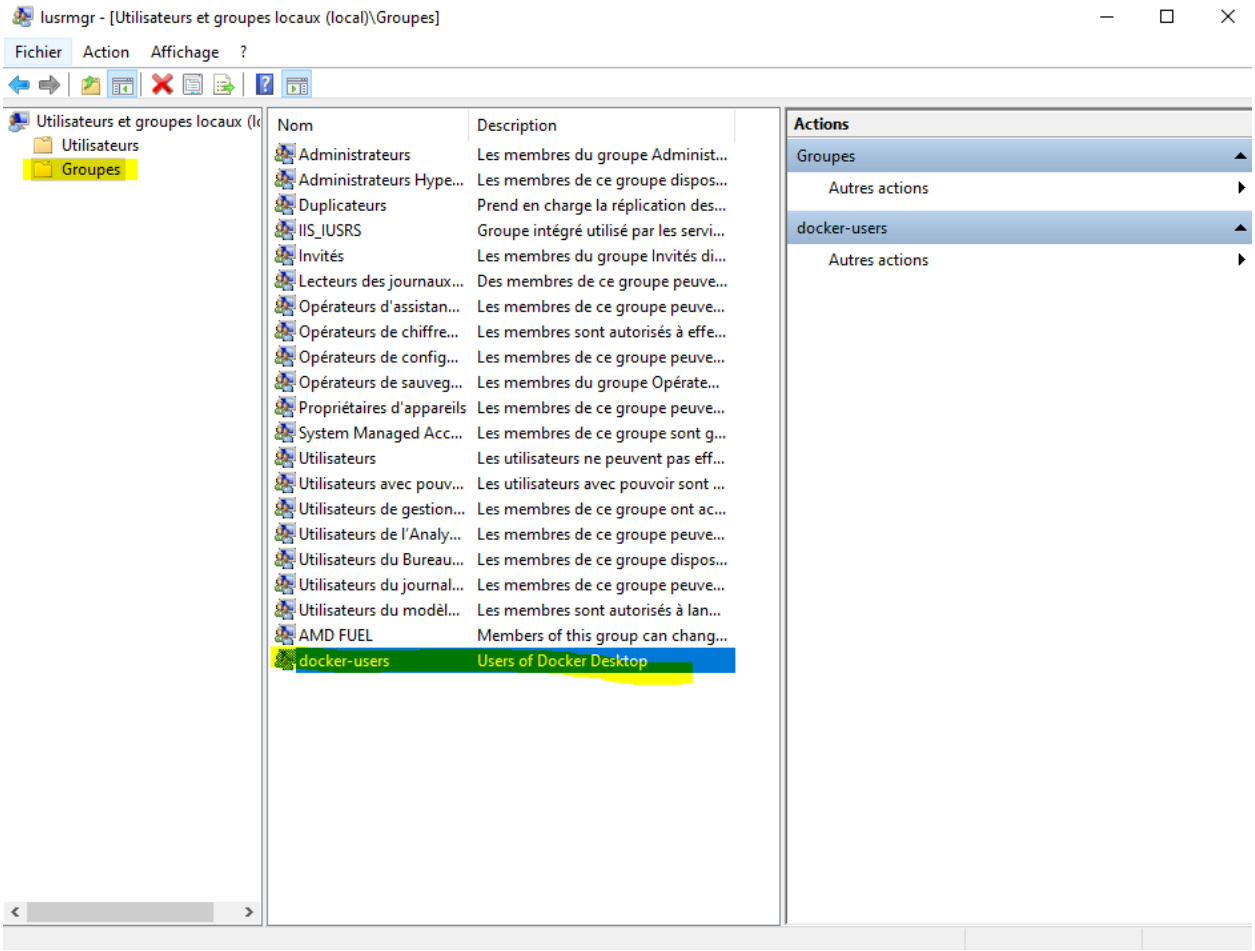
vellement créé par l'installation.

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 »

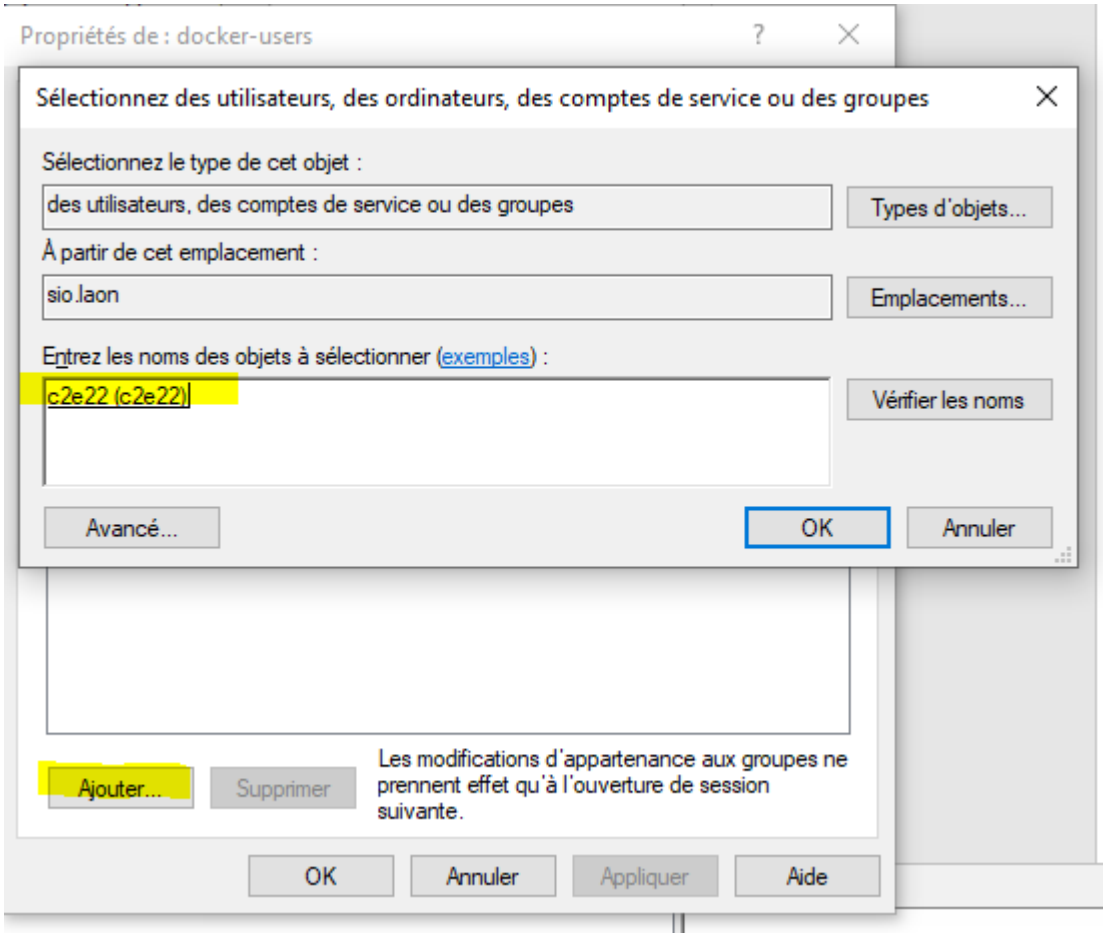
Cette fenêtre devrait s'ouvrir :





Double cliquez sur le groupe `docker-users`.

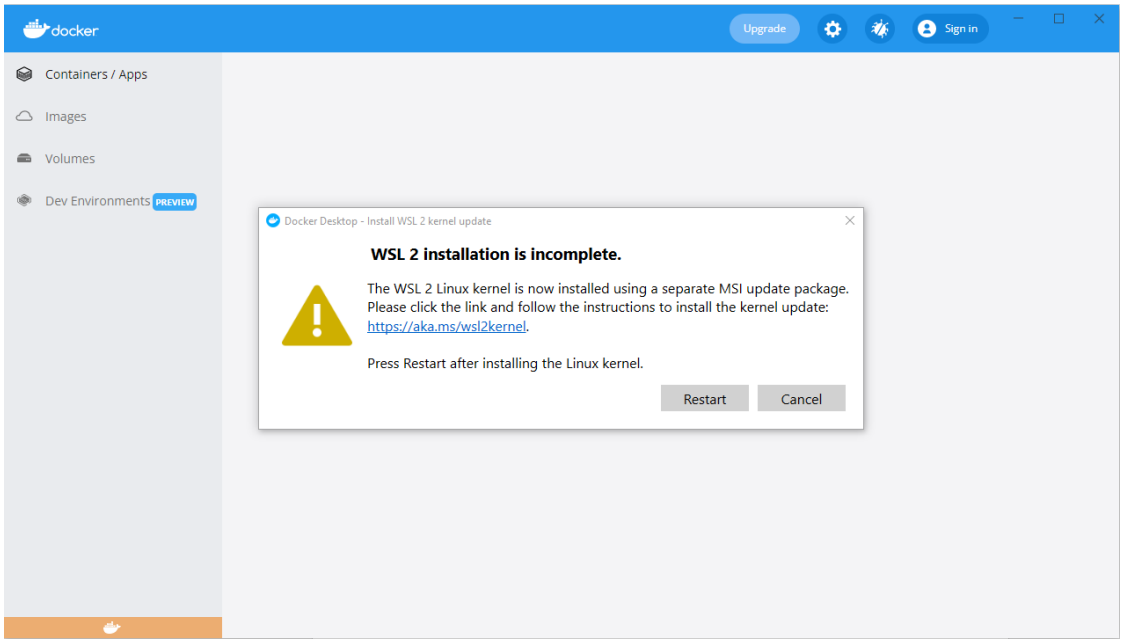
Et ajoutez un nouvel utilisateur : votre compte issu du domaine `sio`



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 :



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
```

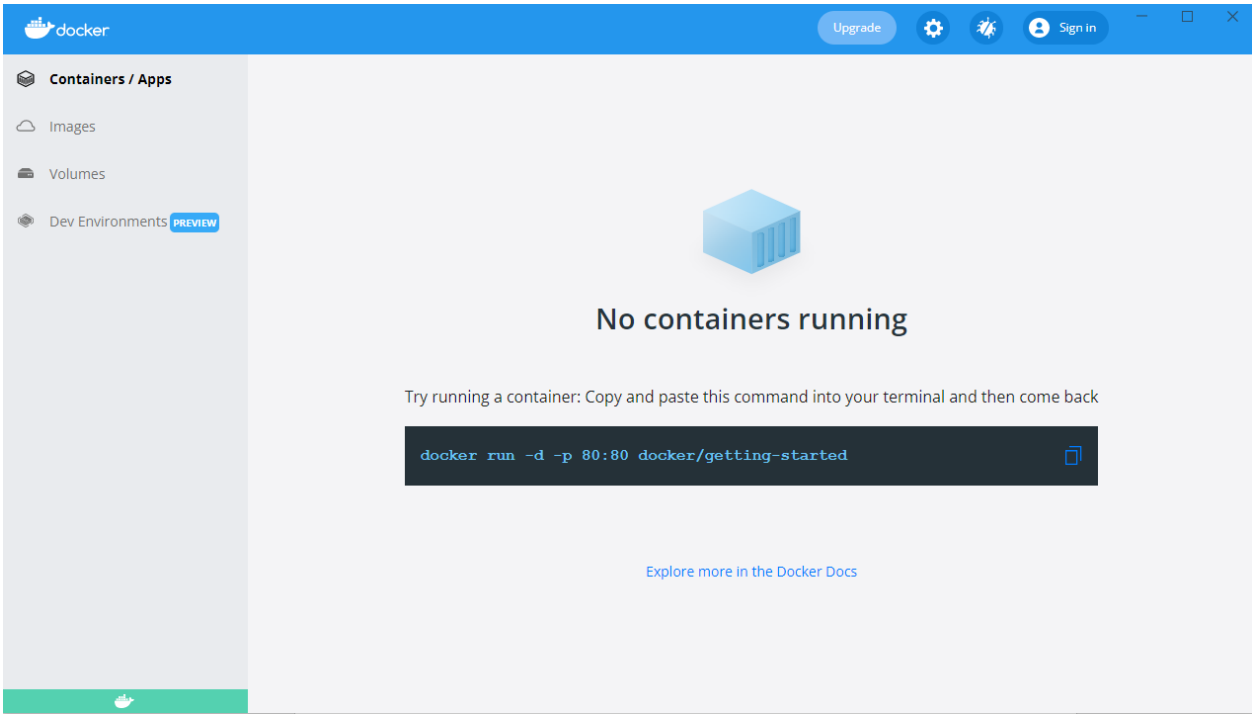
Nous pouvons en profiter pour installer le nouveau **Terminal de Windows**. Cela va apporter plus de confort durant la pratique de ce cours.

Lien vers la page Terminal Windows

Il faut un compte « **Microsoft** » .

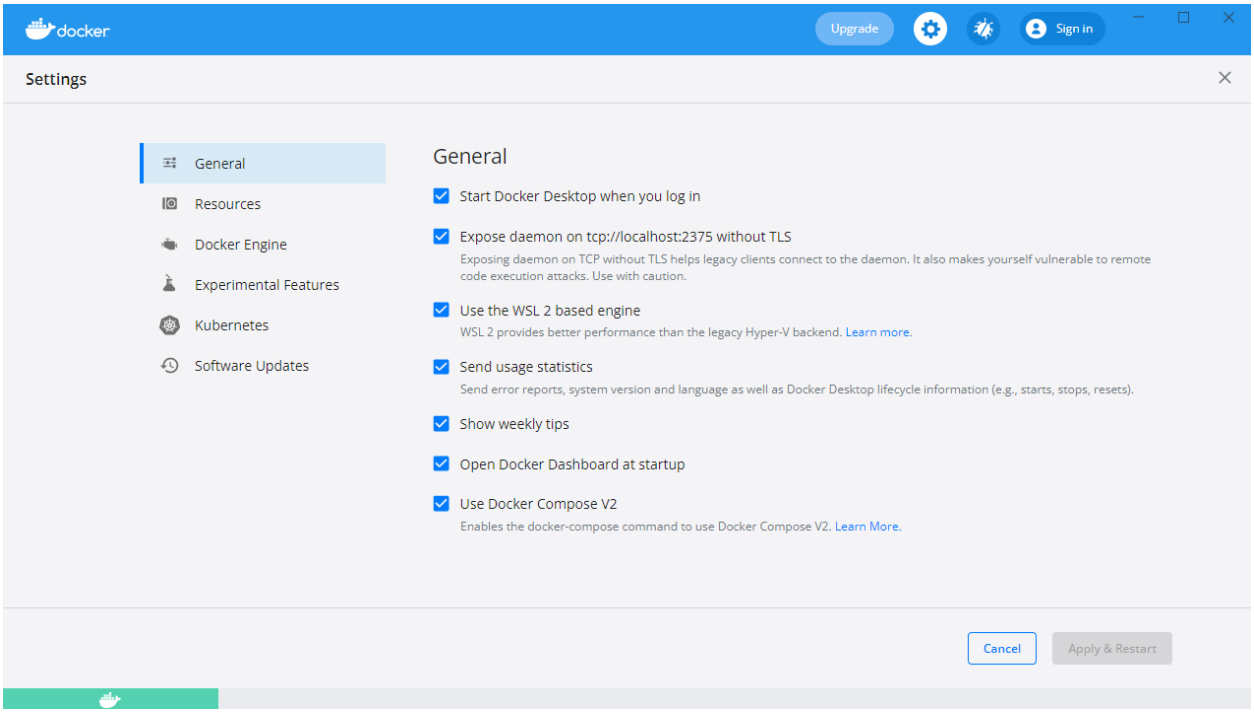
Redémarrez la machine encore une fois pour que **WSL2** soit pris en compte.

Docker devrait maintenant pouvoir démarrer :



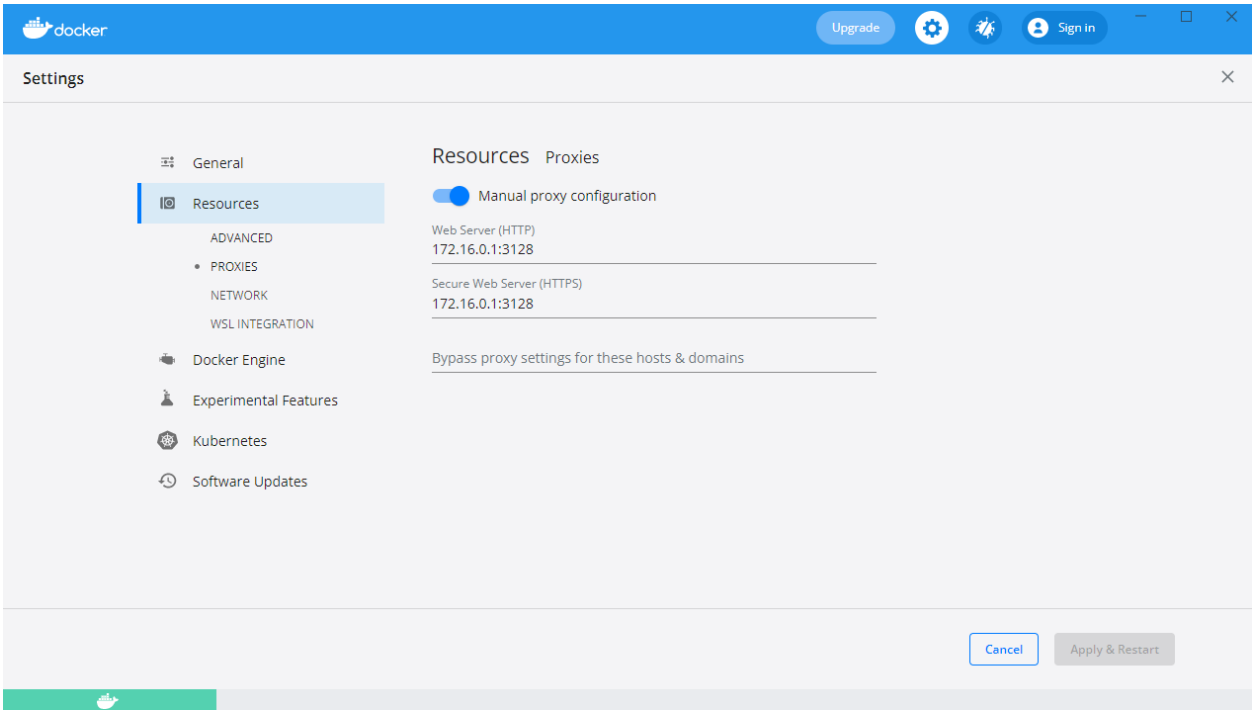
Il faut maintenant configurer le client en cliquant sur l'engrenage en haut à droite.

Cochez les options comme sur la capture d'écran :

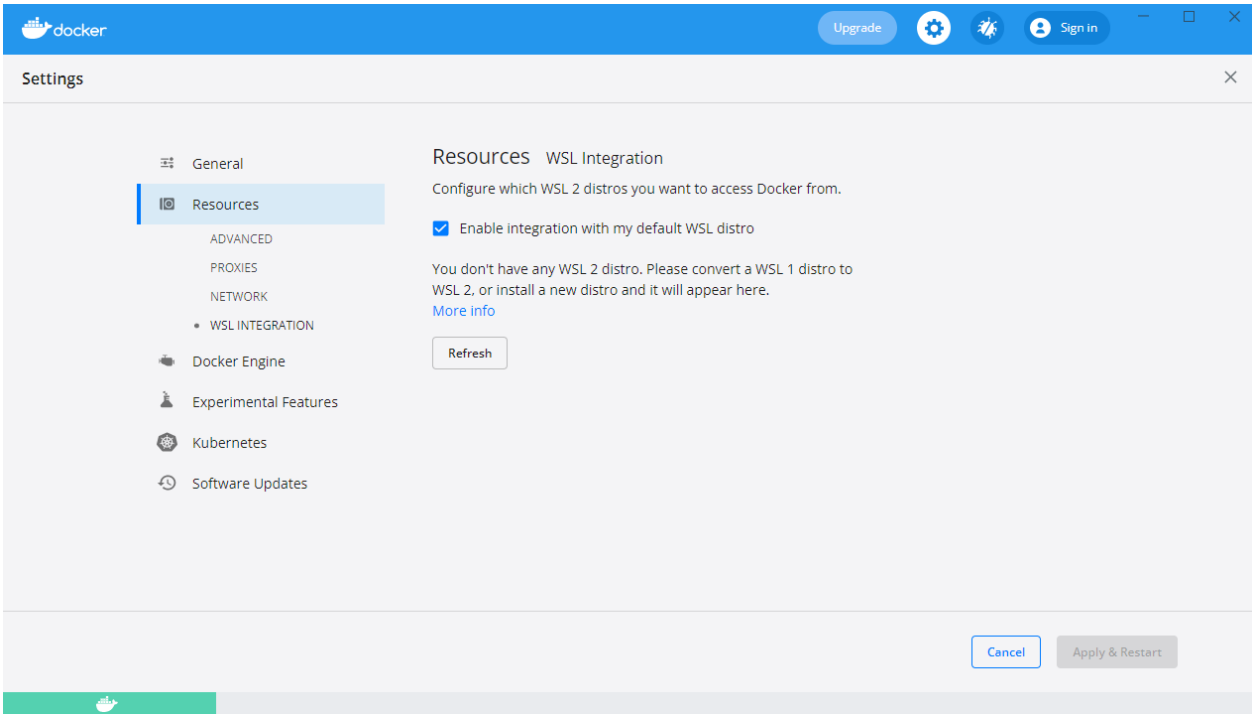


N'oubliez pas de cliquer sur « **Apply & Restart** »

Configurez le PROXY



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



Vous êtes maintenant prêt !

Bienvenue dans le monde de DOCKER.

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

1.2.4 Installation pour Windows 10 ou MacOS

Si vous êtes sur MacOS ou Windows 10 (Entreprise ou Pro) vous pouvez installer Docker Desktop, un environnement compa

— Docker Desktop for Windows
— Docker Desktop for Mac

1.2.5 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 intéragir 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**.

1.2.6 Vérification de l'installation

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

```
docker info
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tous droits réservés.

Testez le nouveau système multiplateforme PowerShell https://aka.ms/pscore6

PS C:\Users\baptiste> docker info
Client:
 Context:    default
 Debug Mode: false
 Plugins:
  buildx: Build with BuildKit (Docker Inc., v0.6.3)
  compose: Docker Compose (Docker Inc., v2.1.1)
  scan: Docker Scan (Docker Inc., 0.9.0)

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 20.10.10
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Cgroup Version: 1
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 5b46e404f6b9f661a205e28d59c982d3634148f8
 runc version: v1.0.2-0-g52b36a2
 init version: de40ad0
 Security Options:
  seccomp
   Profile: default
 Kernel Version: 5.4.72-microsoft-standard-WSL2
 Operating System: Docker Desktop
 OSType: linux
 Architecture: x86_64
 CPUs: 4
 Total Memory: 18.71GiB
 Name: docker-desktop
 ID: YLBX:Q5HX:KIWS:PRRS:FZJE:K6WA:TEEW:SBRK:XPD2:IPNB:KQWR:IZJI
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false

WARNING: No blkio throttle.read_bps_device support
WARNING: No blkio throttle.write_bps_device support
WARNING: No blkio throttle.read_iops_device support
WARNING: No blkio throttle.write_iops_device support
PS C:\Users\baptiste>
```

1.3 3.0 Les containers avec Docker

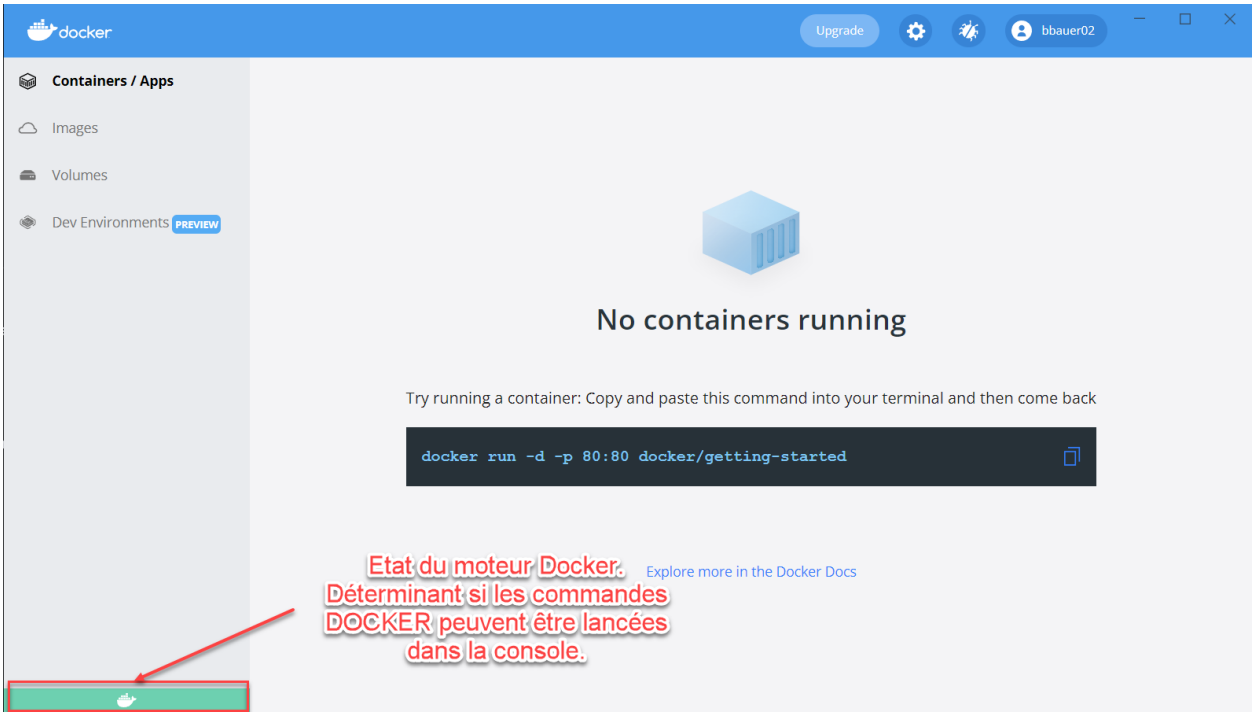
Après avoir présenté la plateforme Docker, nous allons voir comment créer des **containers** en ligne de commande pour lancer des services en tâche de fond et/ou pour rendre disponible dans un container des repertoires de la **machine hôte**.

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

Avant la 1.13, lancer un **container** 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 regroupé aux composant auquel elles se rapportent. C'est la raison pour laquelle le mot clé `container` a été rajouté pour les commandes relatifs à la gestion des containers. `docker container run [OPTIONS] IMAGE [COMMAND] [ARG]` D'autres groupes de commande existent et nous les étudierons plus tard.

1.3.1 3.1 Hello World

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



Ouvrez un **terminal** et tapez :

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

```
Invite de commandes
Microsoft Windows [version 10.0.19044.1466]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\baptiste>Docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:507ecde44b8eb741278274653120c2bf793b174c06ff4eaa672b713b3263477b
Status: Downloaded newer image for hello-world:latest

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

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

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

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

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

C:\Users\baptiste>
```

Téléchargement de l'image depuis le registry (Docker Hub)

Exécution de la commande définie dans l'image.

Le client demande au **daemon** (processus) de lancer un **container** 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-jacent. A la fin du texte on nous demande d'essayer un exemple plus ambitieux, c'est ce que nous allons faire par la suite.

Expérimentez la commande : `docker container run hello-world` sur votre machine

1.3.2 3.2 Ubuntu sous docker

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

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

```
PS C:\Users\baptiste> docker container run ubuntu echo hello
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
08c01a0ec47e: Pull complete
Digest: sha256:669e010b58baf5beb2836b253c1fd5768333f0d1dbcb834f7c07a4dc93f474be
Status: Downloaded newer image for ubuntu:latest
hello
PS C:\Users\baptiste>
```

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

1.3.3 3.2 Un container dans un mode Interactif

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

Pour cela il faut rajouter deux options à notre commande :

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

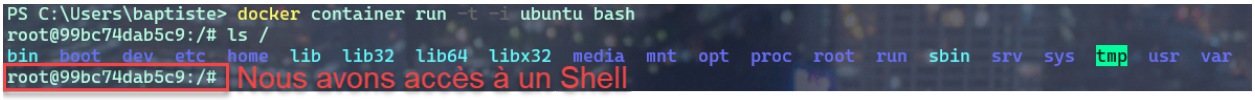
`-i` qui va permettre de laisser l'entrée standard du container ouverte.

Nous allons utiliser l'image **Ubuntu** qui contient les binaires et les bibliothèques du système d'exploitation Ubuntu. Le processus du **container** s'exécutera donc dans cette environnement, c'est-à-dire 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
```



Nous voyons que nous avons accès à un `shell` (*coquille en anglais, interface système*). Nous reconnaissons sans peine le prompt **Ubuntu/Linux** dans lequel nous pouvons écrire par exemple une commande Linux : `ls`

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

Quelle information obtenez-vous ?

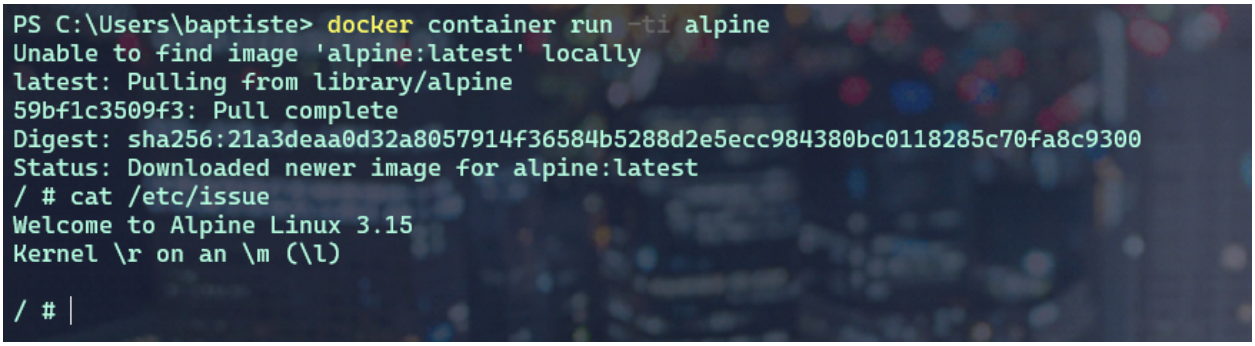
Pour sortir du container on va tuer le processus avec la commande : `exit`

Nous aurions pu faire la même chose en utilisant une autre image que celle d'Ubuntu. Par exemple : Nous souhaitons lancer un container basé sur la distribution **Linux Alpine**. C'est une distribution légère et sécurisée.

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

ou

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



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

Quand on lance un **container** avec seulement la commande `docker container run` par défaut le container est exécuté 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 container basé sur l'image `nginx`, un **serveur http**.

Container NGINX en foreground

Création du **conteneur** en **foreground**, cela signifie que l'on ne récupère pas la main

```
docker container run nginx
```

```
PS C:\Users\baptiste> docker container run nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
5eb5b503b376: Pull complete
1ae07ab881bd: Pull complete
78091884b7be: Pull complete
091c283c6a66: Pull complete
55de5851019b: Pull complete
b559bad762be: Pull complete
Digest: sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f16e78d00deb7e7767
Status: Downloaded newer image for nginx:latest
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2022/02/09 11:00:30 [notice] 1#1: using the "epoll" event method
2022/02/09 11:00:30 [notice] 1#1: nginx/1.21.6
2022/02/09 11:00:30 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2022/02/09 11:00:30 [notice] 1#1: OS: Linux 5.4.72-microsoft-standard-WSL2
2022/02/09 11:00:30 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2022/02/09 11:00:30 [notice] 1#1: start worker processes
2022/02/09 11:00:30 [notice] 1#1: start worker process 32
2022/02/09 11:00:30 [notice] 1#1: start worker process 33
2022/02/09 11:00:30 [notice] 1#1: start worker process 34
2022/02/09 11:00:30 [notice] 1#1: start worker process 35
```

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

Container NGINX en background

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

```
PS C:\Users\baptiste> docker container run -d nginx
6a54b3bae3345f6caf3aead71329e4cb40ea6b34d99910a2b3980920b8cd1081
PS C:\Users\baptiste>
```

Nous voyons ici que nous avons **NGINX** qui tourne en tâche de fond et nous pouvons accéder à ce container 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**.

1.3.4 3.3 Publication de port.

La publication d'un port est utilisée pour qu'un **container** 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**.

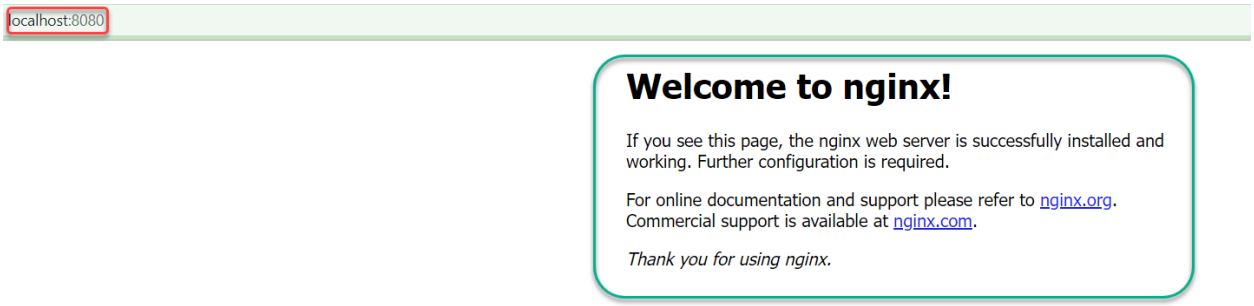
L'option `-P` quant à elle laisse le choix du port au `docker démon`.

Reprenons notre container **NGINX** qui est un serveur **http**. Par défaut, **NGINX** est un processus qui se lance sur le **port 80** dans le container. Si nous souhaitons accéder à notre container depuis **un navigateur de la machine hôte** sur le **port 8080** de la machine hôte, nous lançerons le container **nginx** avec la commande suivante :

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

```
PS C:\Users\baptiste> Docker container run -d -p 8080:80 nginx
f37a9ffd84ca3b07430c03693061a1e968381636cec27e0bf307401903713cf2
```

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



1.3.5 3.4 Bind-mount

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

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 des **containers** ou de monter la **socket Unix** du **daemon Docker** (`/var/run/docker.sock`) pour permettre à un container de dialoguer avec le **daemon**.

3.4.1 Exemple 1 : 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 **container**.

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

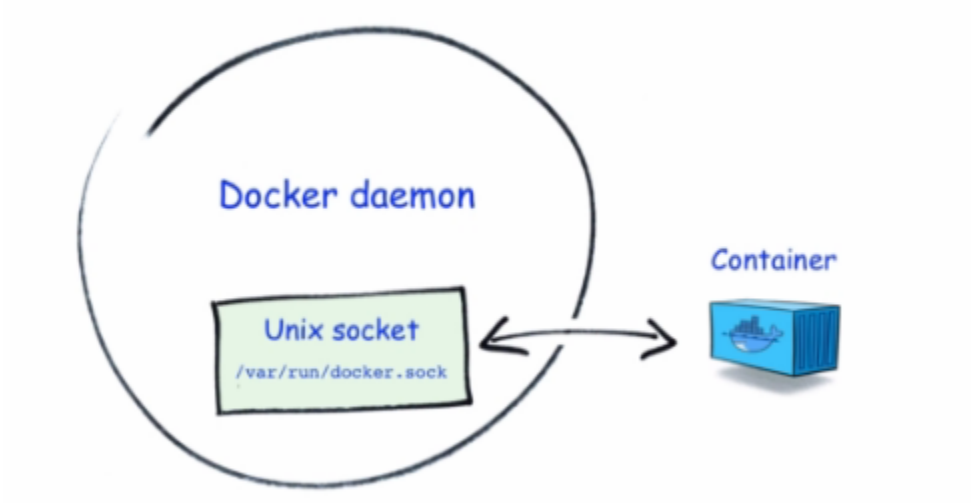
Ou

```
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.

3.4.2 Exemple 2 : 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 container et cela nous donnera accès à l' **API du Daemon**.



Pour LINUX UNIQUEMENT

Créons donc un simple container : 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 container 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 container 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 container depuis le container **ADMIN** :

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

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

Pour WINDOWS UNIQUEMENT

Bientôt disponible ....

3.4.3 Exemple 3 : Ecouter les actions demandées au Docker Daemon

Nous allons lancer un autre dans laquel le **socket** est monté. Et nous allons écouter les actions demandées sur le **Docker Daemon**. **Même ceux provenant d'autres containers**.

```
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
```

1.3.6 3.5 Limitation des ressources

Nous avons dit que le lancement d'un **containeur** 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 container pourra utiliser toute la RAM et impacter tout les autres containeurs qui tournent sur la même machine hôte.

Nous pouvons toutefois imposer des limites à un containeur.

Lançons un containeur 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** consommeecho, il sera tué par **Docker**.

Nous pouvons limiter l'utilisation du **CPU** également. Lançons un containeur 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)

1.3.7 3.6 Les droits dans un container

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

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

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 container (`gosu`).

Lançons un container 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 containeurs par l'intermédiaire de leur identifiant ou nom et via la commande `top`.

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

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

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

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

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**.

1.3.8 3.6 Des options utiles

Note :
- `--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.

1.3.9 3.7 Les commandes de base avec Docker

```
docker container <command>
```

Tableau 1 – Les commandes de base de docker container

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

- La commande `ls` :

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

```
bauer@DESKTOP-9MLO85I:~$ docker container ls
CONTAINER ID   IMAGE     COMMAND        CREATED         STATUS         PORTS     NAMES
5305cda41c82   alpine    "sleep 1000"   8 minutes ago   Up 8 minutes             clever_kilby
```

Pour lister tout les containers actifs et stoppés : `docker container ls -a`.

```
bauer@DESKTOP-9MLO85I:~$ docker container ls -a
CONTAINER ID   IMAGE       COMMAND                  CREATED         STATUS                      PORTS     NAMES
5305cda41c82   alpine      "sleep 1000"             9 minutes ago   Up 9 minutes                          clever_kilby
a37b0c664066   alpine      "sleep 1000"             23 minutes ago  Exited (0) 6 minutes ago              hopeful_blackburn
f0dab6b381f0   mongo:4.0   "docker-entrypoint.s…"   25 minutes ago  Exited (0) 24 minutes ago             blissful_poitras
c1cc9f763a86   alpine      "sleep 1000"             28 minutes ago  Exited (137) 24 minutes ago           adoring_liskov
243c4a3ad725   mongo:4.0   "docker-entrypoint.s…"   33 minutes ago  Exited (0) 24 minutes ago             wonderful_brattain
c231d84d9ea3   alpine      "sleep 1000"             36 minutes ago  Exited (137) 24 minutes ago           relaxed_moore
```

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

```
bauer@DESKTOP-9MLO85I:~$ docker container ls -a -q
5305cda41c82
a37b0c664066
f0dab6b381f0
c1cc9f763a86
243c4a3ad725
c231d84d9ea3
```

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

```
bauer@DESKTOP-9MLO85I:~$ docker container inspect clever_kilby
[
    {
        "Id": "5305cda41c82dfbea49919a2aaf8752f6b75542cba6f9273e1eec9597ac1e086",
        "Created": "2022-02-21T10:57:14.8917367Z",
        "Path": "sleep",
        "Args": [
            "1000"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 1909,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2022-02-21T10:57:15.3015127Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:c059bfaa849c4d8e4aecaeb3a10c2d9b3d85f5165c66ad3a4d937758128c4d18",
```

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**

```
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
```

```
PS C:\Users\baptiste> docker container run --name ping -d alpine ping 8.8.8.8
b23e5c4c3c4078fc1e3c30044606643539060d29fe27f80679c06e14fb69da70
PS C:\Users\baptiste> docker container logs -f ping
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=37 time=16.148 ms
64 bytes from 8.8.8.8: seq=1 ttl=37 time=17.334 ms
64 bytes from 8.8.8.8: seq=2 ttl=37 time=17.132 ms
64 bytes from 8.8.8.8: seq=3 ttl=37 time=16.003 ms
64 bytes from 8.8.8.8: seq=4 ttl=37 time=15.244 ms
64 bytes from 8.8.8.8: seq=5 ttl=37 time=15.405 ms
64 bytes from 8.8.8.8: seq=6 ttl=37 time=14.706 ms
```

— 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.

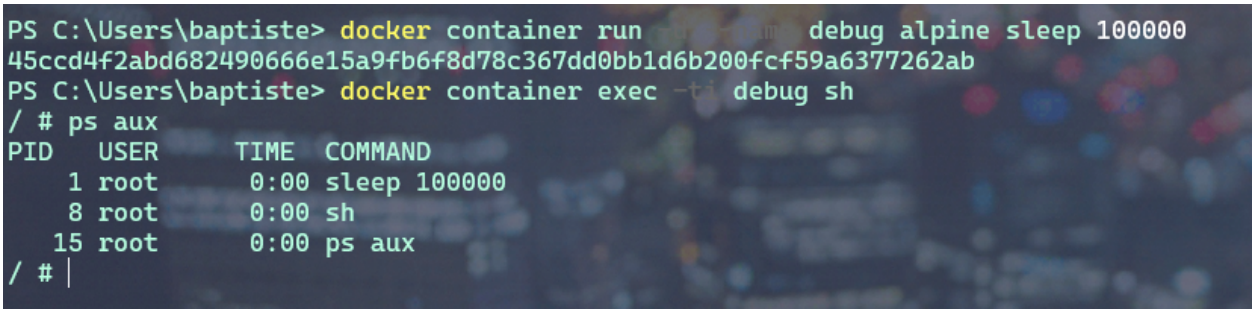
Exemple : lançons un container qui attend 100000 secondes, et demandons ensuite d'ouvrir un shell pour lister les processus de ce container.

```
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
```



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`.

> **Avertissement :** 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.

1.3.10 3.8 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 :

```
PS C:\Users\baptiste> docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS              PORTS                    NAMES
6f94e474cab6   alpine    "ping 8.8.8.8"           About a minute ago   Up About a minute                            ping
40c3e10088aa   nginx     "/docker-entrypoint.…"   3 minutes ago        Up 3 minutes        0.0.0.0:80->80/tcp       www
```

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.

```
PS C:\Users\baptiste> docker container ls
CONTAINER ID   IMAGE         COMMAND                  CREATED         STATUS                     PORTS                    NAMES
599ddf847b4b   hello-world   "/hello"                 3 minutes ago   Exited (0) 3 minutes ago                            fervent_napier
6f94e474cab6   alpine        "ping 8.8.8.8"           4 minutes ago   Up 4 minutes                                        ping
40c3e10088aa   nginx         "/docker-entrypoint.…"   5 minutes ago   Up 5 minutes               0.0.0.0:80->80/tcp       www
```

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
```

```
PS C:\Users\baptiste> docker container inspect --format '{{ .NetworkSettings.IPAddress }}' www
172.17.0.2
```

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`

```
PS C:\Users\baptiste> docker container exec     ping sh
/ # ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 ping 8.8.8.8
    7 root      0:00 sh
   14 root      0:00 ps aux
/ #
```

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)
```

1.3.11 3.9 Exercices :

3.9.1 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 ?

3.9.2 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`.

3.9.3 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 ?

3.9.4 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 ?

3.9.5 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).

3.9.6 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.

3.9.7 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 ?

3.9.8 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.

1.3.12 3.10 En résumé

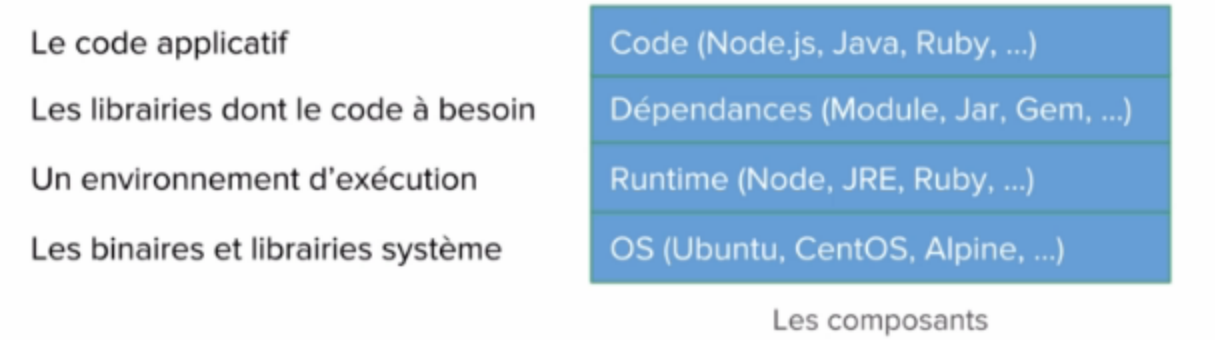
Nous avons commencé à jouer avec les containers et vu les commandes les plus utilisées pour la gestion du cycle de vie des containers (`run`, `exec`, `ls`, `rm`, `inspect`). Nous les utiliserons souvent dans la suite du cours.

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.

1.4 4.0 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)

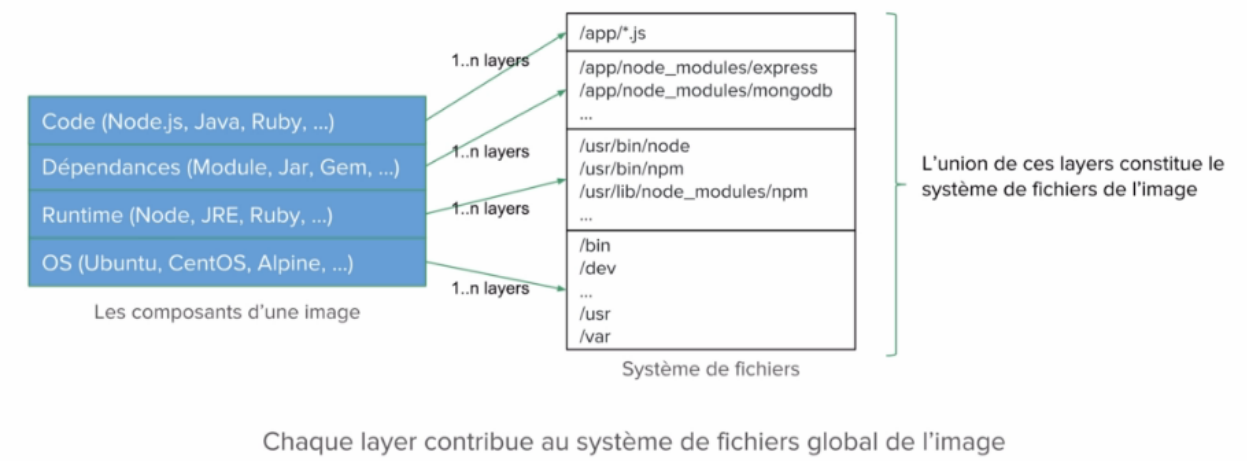
Contenu d'une image :



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.



1.4.1 4.1 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.

> **Avertissement :** 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

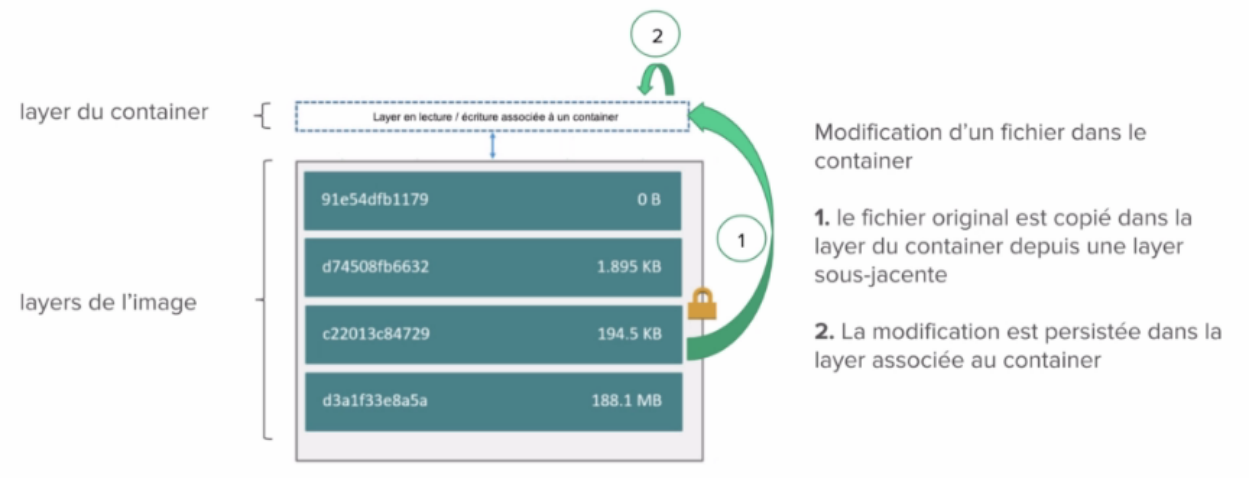
Pour accèder à ce dossier sous Windows, il faut alors créer un container et le lier avec Docker :

```
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**.



1.4.2 4.2 Exercices

4.2.1 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
```

1.4.3 4.3 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 de base
FROM node:8.11.1-alpine

# Copie de la liste des dependances
COPY package.json /app/package.json

# Installation / compilation des dependances
RUN cd /app && npm install

# Copie du code applicatif
COPY . /app/

# Exposition du port HTTP
EXPOSE 80

# Positionnement du répertoire de travail
WORKDIR /app

# Commande exécutée au lancement d'un container
CMD ["npm", "start"]
```

Exemple de Dockerfile utilisé pour une application Node.js

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**.

FROM	Image de base
ENV	Définition de variables d'environnement
RUN	Exécution d'une commande, construction du filesystem de l'image
COPY / ADD	Copie de ressources depuis la machine local dans le filesystem de l'image
EXPOSE	Expose un port de l'application
HEALTHCHECK	Vérifie l'état de santé de l'application
VOLUME	Définition d'un volume pour la gestion des données
WORKDIR	Définition du répertoire de travail
USER	Utilisateur auquel appartient le processus du container
ENTRYPOINT	Définie la commande exécutée au lancement du container
CMD	

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.

```
FROM nginx:1.14.0
ENV path /usr/share/nginx/html/
WORKDIR ${path}
COPY . $path
```

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**.

- Shell : lancé dans un shell avec "/bin/sh -c" par défaut

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

- Exec : non lancé dans un shell

```
RUN ["/bin/bash", "-c", "echo hello"]
```

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.

```
…
EXPOSE 27017
CMD ["mongod"]
```

Extrait du Dockerfile de MongoDB

```
…
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Extrait du Dockerfile de Nginx

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**.

```
…
RUN mkdir -p /data/db /data/configdb \
      && chown -R mongodb:mongodb /data/db /data/configdb
VOLUME /data/db /data/configdb

COPY docker-entrypoint.sh /usr/local/bin/
ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]
```

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.

```
FROM node:8.11-alpine
RUN apk update && apk add curl
HEALTHCHECK --interval=5s --timeout=3s --retries=3 CMD curl -f http://localhost:8000/health || exit 1
COPY package.json /app/package.json
WORKDIR /app
RUN npm install
COPY . /app
CMD ["npm", "start"]
```

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.

```
ENTRYPOINT [“curl”]
CMD [“--help”]
```

1.4.4 4.3 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.

1.4.5 4.4 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 :

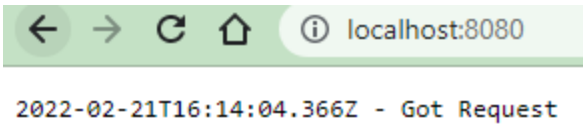
```
npm install
```

puis

```
npm start
```

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

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



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

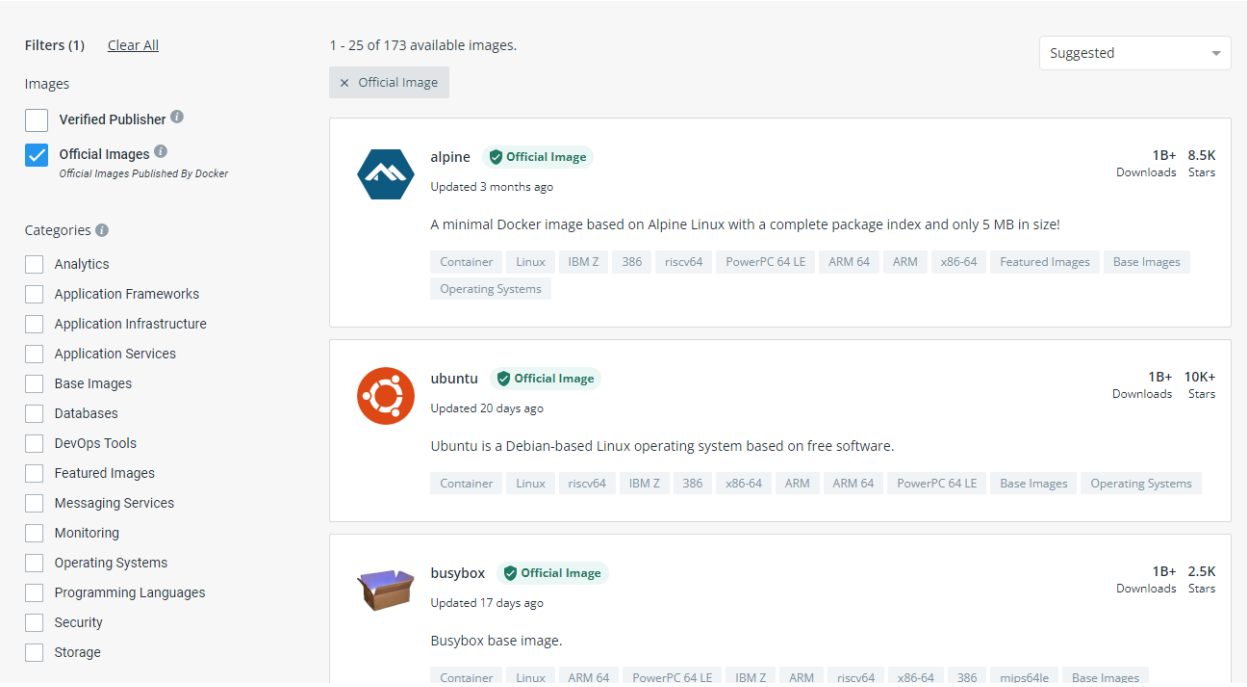
1. Il doit avoir NODEJS d'installé sur sa machine.
2. Il doit installer les dépendances du projet, ici `express`.
3. Il doit lancer le serveur Nodejs.

Il faudrait donc créer une image réalisant ces étapes ! !

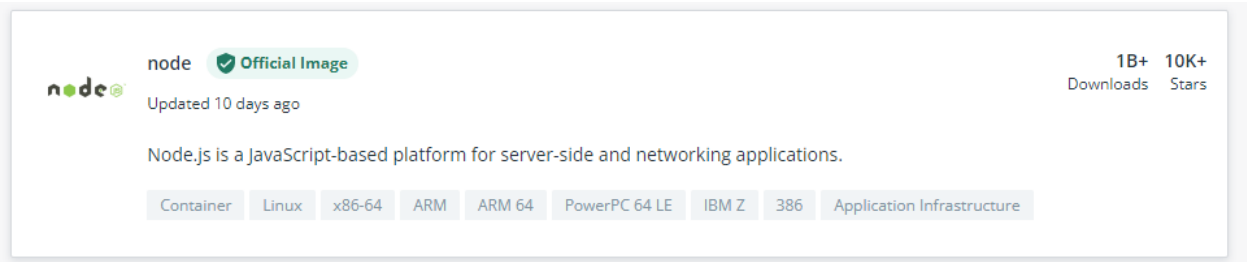
Nous allons créer un **DockerFile**.

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

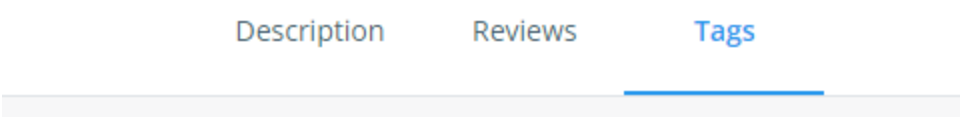
Cochez : « **Official Images** » pour n'avoir que des images officielles. Nous voyons que nous avons une multitude de possibilité pour concevoir notre image.



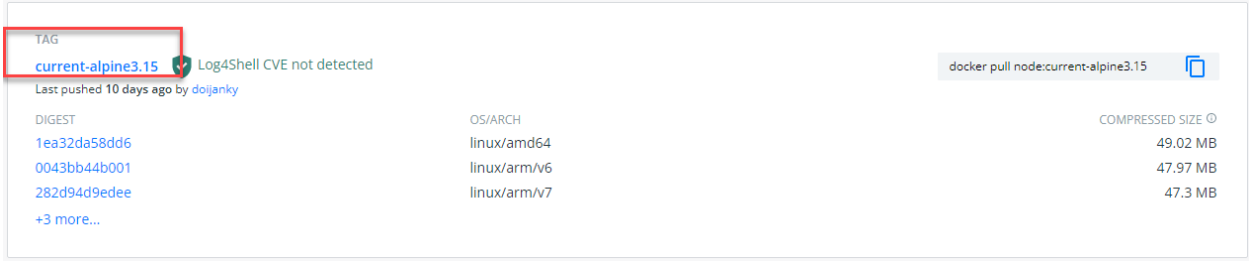
Nous pouvons partir sur une image **LINUX** : **UBUNTU**, **ALPINE** . . . Etc mais aussi une image où le runtime **NODEJS** est déjà packagé. C'est ce que nous allons choisir.



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



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



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

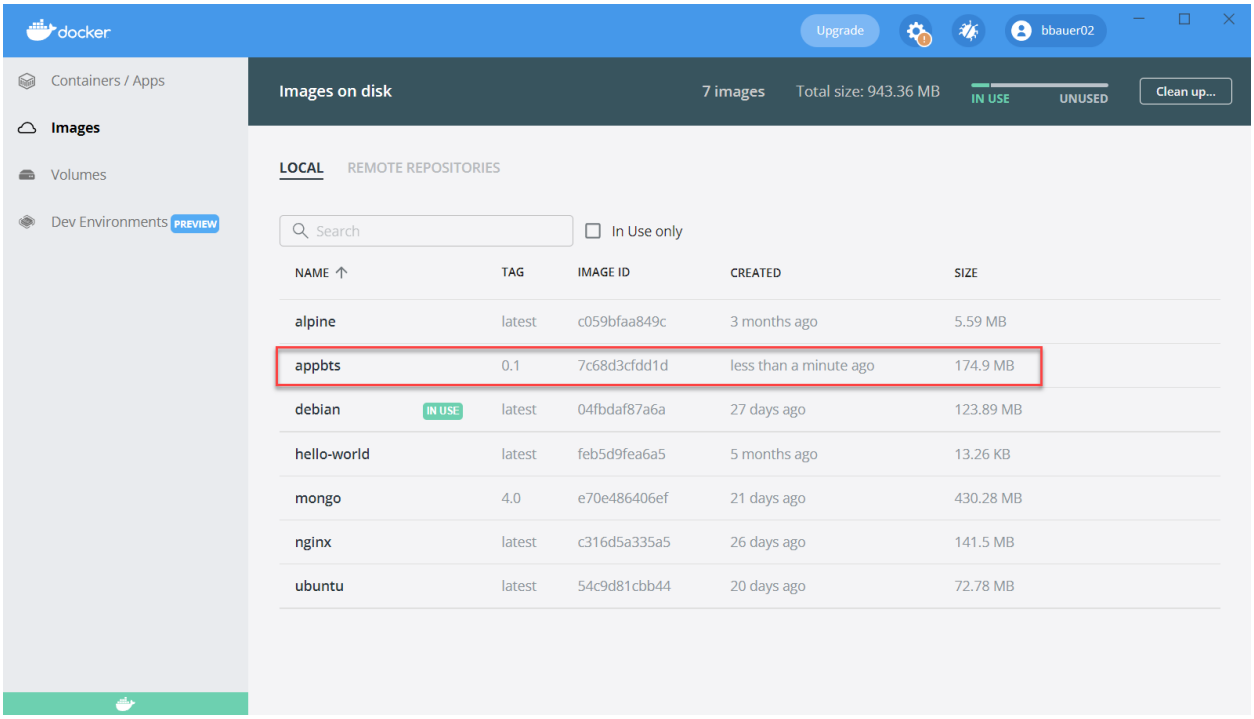
```
# Nous renseignons dans l'instruction FROM le Tag de notre image qui servira de base à␣
↪notre application
FROM node:current-alpine3.15

# 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 .
```

Nous voyons que pour chaque instruction nous avons une étape.

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



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

1.4.6 4.5 Exercices : Création d'images

4.5.1 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.

4.5.2 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.

1. Construire l'image en la taguant `pong:v1.0`.
2. Lancez un container basé sur cette image en publiant le `port 80` sur le `port 8080` de la machine hôte.
3. Tester l'application.
4. Supprimez le container.

4.5.3 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.

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.

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`.

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.

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
```

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.

Pour aller plus loin : ou est stockée mon image ?

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/sha256/00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc05816
./imagedb/metadata/sha256/
↪00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc05816
./imagedb/metadata/sha256/
↪00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc05816/lastUpdated
./imagedb/metadata/sha256/
↪00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc05816/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/cache-
↪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/cache-id
49908d07e177f9b61dc273ec7089efed9223d3798ad1d86c78d4fe953e227668
```

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

```
/var/lib/docker/overlay2/
↪49908d07e177f9b61dc273ec7089efed9223d3798ad1d86c78d4fe953e227668/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.

1.4.7 4.6 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
```

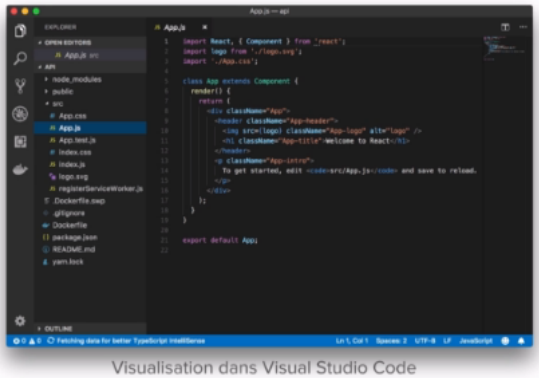
un dossier `api` est créé.

```
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.

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

Création d'une application React



Visualisation dans Visual Studio Code

```
FROM node:8.11-alpine as build
WORKDIR /app
COPY package.json ./package.json
RUN npm install
COPY . ./
RUN npm run build

FROM nginx:1.14.0
COPY --from=build /app/build/ /usr/share/nginx/html/
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Dockerfile avec multi stage build

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 :

```
$ docker image build -t app:1.0 .
Sending build context to Docker daemon  116.8MB
Step 1/10 : FROM node:8.11-alpine as build
 ---> ca148a52ea10
Step 2/10 : WORKDIR /app
 ---> ac238d1a666d
...
Step 8/10 : COPY --from=build /app/build/
/usr/share/nginx/html/
 ---> Using cache
 ---> a8028d66c9f9
Step 9/10 : EXPOSE 80
 ---> Running in 2f47dcb17484
Removing intermediate container 2f47dcb17484
 ---> a87f7038dbb3
Step 10/10 : CMD ["nginx", "-g", "daemon off;"]
 ---> Running in 5aa44388a837
Removing intermediate container 5aa44388a837
 ---> b4f80ee7f075
Successfully built b4f80ee7f075
Successfully tagged app:1.0
```

Création d'un container à partir de l'image app:1.0

```
$ docker run -p 9000:80 app:1.0
```

4.6.1 Mise en pratique

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

Rappel

Comme nous l'avons vu, le Dockerfile contient une liste d'instructions qui permet de créer une image. La première instruction est FROM, elle définit l'image de base utilisée. Cette image de base contient souvent beaucoup d'éléments (binaires et librairies) dont l'application finale n'a pas besoin (compilateur, ...). Ceci qui peut impacter de façon considérable la taille de l'image et également sa sécurité puisque cela peut considérablement augmenter sa surface d'attaque. C'est la qu'intervint le multistage build...

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)
}
```

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"]
```

Note : 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.

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.

1.4.8 4.7 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 :

- Chaque instruction utilise le cache créé lors du build précédent
- Création de l'image quasi immédiate

```
FROM node:8.11-alpine
COPY package.json /app/package.json
RUN cd /app && npm install
COPY . /app/
WORKDIR /app
EXPOSE 80
CMD ["npm", "start"]
```

```
$ docker image build -t app:1.0 .
Sending build context to Docker daemon  4.096kB
Step 1/7 : FROM node:8.11-alpine
 ---> 8adf3c3eb26c
Step 2/7 : COPY package.json /app/package.json
 ---> Using cache
 ---> 1d0614163d62
Step 3/7 : RUN cd /app && npm install
 ---> Using cache
 ---> de7ac239b6b5
Step 4/7 : COPY . /app/
 ---> Using cache
 ---> 812921ed8643
Step 5/7 : WORKDIR /app
 ---> Using cache
 ---> ea7c2d576e53
Step 6/7 : EXPOSE 80
 ---> Using cache
 ---> 202601df911a
Step 7/7 : CMD ["npm", "start"]
 ---> Using cache
 ---> 24f42e7ed5ea
Successfully built 24f42e7ed5ea
Successfully tagged app:1.0
```

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 .

```
PS C:\Users\baptiste\Documents\docker\imageNodeJs> docker image build -t appbts:0.1 .
[+] Building 1.8s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 32B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/node:current-alpine3.15
 => [auth] library/node:pull token for registry-1.docker.io
 => [internal] load build context
 => => transferring context: 20.98kB
 => [1/4] FROM docker.io/library/node:current-alpine3.15@sha256:0e83c810225bc29e614189acf3d6419e3c09881cefb9f7a170fdcfe3e15bbfd5
 => CACHED [2/4] COPY . /app/
 => CACHED [3/4] RUN cd /app && npm install
 => CACHED [4/4] WORKDIR /app
 => exporting to image
 => => exporting layers
 => => writing image sha256:1cab61d3de9b52003b3dc2950fd5a93b19f57bf74fb89bd53974b2033724f0ec
 => => naming to docker.io/library/appbts:0.1
```

Utilisation du cache

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

```
PS C:\Users\baptiste\Documents\docker\imageNodeJs> docker image build -t appbts:0.1 .
[+] Building 3.6s (9/9) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 32B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/node:current-alpine3.15
 => [internal] load build context
 => => transferring context: 21.28kB
 => CACHED [1/4] FROM docker.io/library/node:current-alpine3.15@sha256:0e83c810225bc29e614189acf3d6419e3c09881cefb9f7a170fdcfe3e15bbfd5
 => [2/4] COPY . /app/
 => [3/4] RUN cd /app && npm install
 => [4/4] WORKDIR /app
 => exporting to image
 => => exporting layers
 => => writing image sha256:7b68a4b2a8d48bde9e7b5e793db1e5c9a652577ea46b7fcf06f8d2d96a3e4fc5
 => => naming to docker.io/library/appbts:0.1
```

Docker Daemon détecte que les fichiers ont été modifiés.
Alors le cache n'est pas utilisé pour les étapes 2 à 4

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.

```
PS C:\Users\baptiste\Documents\docker\imageNodeJs> docker image build -t appbts:0.1 .
[+] Building 1.3s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 32B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/node:current-alpine3.15
 => [1/5] FROM docker.io/library/node:current-alpine3.15@sha256:0e83c810225bc29e614189acf3d6419e3c09881cefb9f7a170fdcfe3e15bbfd5
 => [internal] load build context
 => => transferring context: 21.28kB
 => CACHED [2/5] COPY package.json /app/package.json
 => CACHED [3/5] RUN cd /app && npm install
 => [4/5] COPY . /app/
 => [5/5] WORKDIR /app
 => exporting to image
 => => exporting layers
 => => writing image sha256:05cb03980b25bedbe7d0b3cbed335dc612ad8021f291b95b0e6fb8b4b3d375d8
 => => naming to docker.io/library/appbts:0.1
```

Package.json est bien utilisé en cache ici

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

4.7.1 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

1.4.9 4.8 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 :

```
PS C:\Users\baptiste\Documents\docker\imageNodeJs> docker image build -t appbts:0.1 .
[+] Building 1.3s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 32B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/node:current-alpine3.15
 => [1/5] FROM docker.io/library/node:current-alpine3.15@sha256:0e83c810225bc29e614189acf3d6419e3c09881cefb9f7a170fdcfe3e15bbfd5
 => [internal] load build context
 => => transferring context: 21.28kB
 => CACHED [2/5] COPY package.json /app/package.json
 => CACHED [3/5] RUN cd /app && npm install
 => [4/5] COPY . /app/
 => [5/5] WORKDIR /app
 => exporting to image
 => => exporting layers
 => => writing image sha256:05cb03980b25bedbe7d0b3cbed335dc612ad8021f291b95b0e6fb8b4b3d375d8
 => => naming to docker.io/library/appbts:0.1
```

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 :

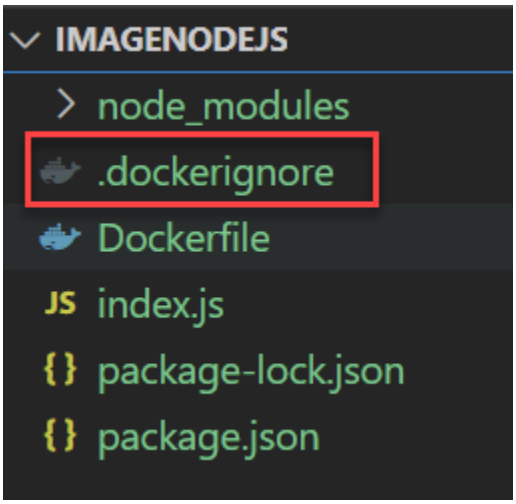
Git init

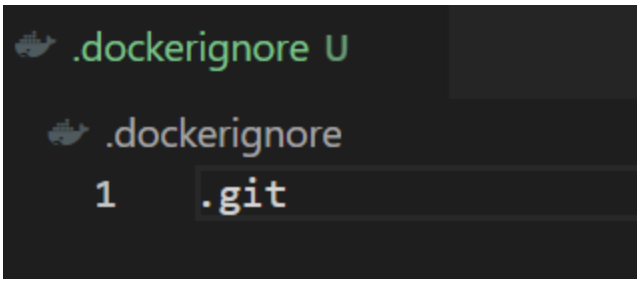
```
PS C:\Users\baptiste\Documents\docker\imageNodeJs> git init
Initialized empty Git repository in C:/Users/baptiste/Documents/docker/imageNodeJs/.git/
```

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.





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.

1.4.10 4.9 Les commandes de base avec docker image.

La commande PULL.

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

```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE

$ docker image pull alpine
Using default tag: latest
latest: Pulling from library/alpine
0a8490d0dfd3: Pull complete
Digest: sha256:dfbd4a3a8ebca874ebd2474f044a0b33600d4523d03b0df76e5c5986cb02d7e8
Status: Downloaded newer image for alpine:latest

$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              88e169ea8f46        2 weeks ago         3.98 MB
```

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.

```
$ docker image inspect alpine
[
    {
        "Id": "sha256:88e169ea8f46ff0d0df784b1b254a15ecfaf045aee1856dca1ec242fdd231ddd",
        "RepoTags": ["alpine:latest"],
        ...
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 3983615,
        "VirtualSize": 3983615,
        "GraphDriver": {
            "Name": "aufs",
            "Data": null
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:60ab55d3379d47c1ba6b6225d59d10e1f52096ee9d5c816e42c635ccc57a5a2b"
            ]
        }
    }
]

$ docker image inspect -f '{{ .Architecture }}' alpine
Amd64

$ docker inspect --format '{{ .ContainerConfig.Cmd }}' mongo:3.2
[/bin/sh -c #(nop)  CMD ["mongod"]]
```

La commande : History

Permet de voir l'historique d'une image.

La commande : ls

Permet d'énumérer les images localement.

```
$ docker image ls
REPOSITORY   TAG      IMAGE ID       CREATED            SIZE
node         6.9.4    c5667be18e4e   2 days ago         655 MB
alpine       latest   88e169ea8f46   3 weeks ago        3.98 MB
<none>       <none>   d02c4d04476c   11 hours ago       653.2 MB

$ docker image ls -a
REPOSITORY   TAG      IMAGE ID       CREATED            SIZE
<none>       <none>   dcb9d60b5a87   About a minute ago 653.2 MB
<none>       <none>   9a77011f0d01   About a minute ago 653.2 MB
node         6.9.4    c5667be18e4e   2 days ago         655 MB
alpine       latest   88e169ea8f46   3 weeks ago        3.98 MB
<none>       <none>   d02c4d04476c   11 hours ago       653.2 MB

$ docker image ls node
REPOSITORY    TAG      IMAGE ID      CREATED            SIZE
node          6.9.4    06b984afb149  9 days ago         655 MB

$ docker image ls --filter dangling=true
REPOSITORY   TAG      IMAGE ID       CREATED            SIZE
<none>       <none>   d02c4d04476c   11 hours ago       653.2 MB

$ docker image prune
Total reclaimed space: 653.2 MB
```

Images créées ou téléchargées

Liste les images temporaires créées lors du build

Filtre des images par nom

Les images "dangling" ne sont plus référencées

Suppression des images "dangling"

Les commandes Save et Load.

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

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              88e169ea8f46        3 weeks ago         3.98 MB

$ docker save -o alpine.tar alpine

$ ls
alpine.tar

$ docker image rm alpine
Untagged: alpine:latest
Untagged: alpine@sha256:dfbd4a3a8ebca874ebd2474f044a0b33600d4523d03b0df76e5c5986cb02d7e8
Deleted: sha256:88e169ea8f46ff0d0df784b1b254a15ecfaf045aee1856dca1ec242fdd231ddd
Deleted: sha256:60ab55d3379d47c1ba6b6225d59d10e1f52096ee9d5c816e42c635ccc57a5a2b

$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE

$ docker load < alpine.tar
60ab55d3379d: Loading layer [==================================================>] 4.226 MB/4.226 MB
Loaded image: alpine:latest

$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              88e169ea8f46        3 weeks ago         3.98 MB
```

La commande : rm

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

```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              f49eec89601e        16 hours ago        129 MB
mhart/alpine-node   6.9.4               d448eac1cfdb        2 weeks ago         49 MB
alpine              latest              88e169ea8f46        3 weeks ago         3.98 MB

$ docker image rm ubuntu
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:71cd81252a3563a03ad8daee81047b62ab5d892ebbfbf71cf53415f29c130950
Deleted: sha256:f49eec89601e8484026a8ed97be00f14db75339925fad17b440976cffcbfb88a
Deleted: sha256:3e2d12b23fafe176cf40429fb25be6572212807f27455b8e3e114c397324446f
Deleted: sha256:88a37465e211da3c72acbd999b158ee31c6c7239131f6308f000dcf52d622a7b
Deleted: sha256:99be185bee70b39c64096b8d39b96153d28d3caa7764961a9285ad4d189cd536
Deleted: sha256:fc7e2c65ec42780443f87ae7d9621cd6fcdb371e2127dd461b449d9e50b7ab7b
Deleted: sha256:4f03495a4d7de505ccb8b8e4cfd0a8ac201491e1f67ae54b65584e0012aaab9c

$ docker image ls -q
d448eac1cfdb
88e169ea8f46

$ docker image rm $(docker image ls -q)
```

4.9.1 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 ?

1. Supprimez l'image mongo :3.6

1.5 5.0 Registry

En cours de rédaction …

1.6 6.0 Stockage

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

1.6.1 6.1 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.

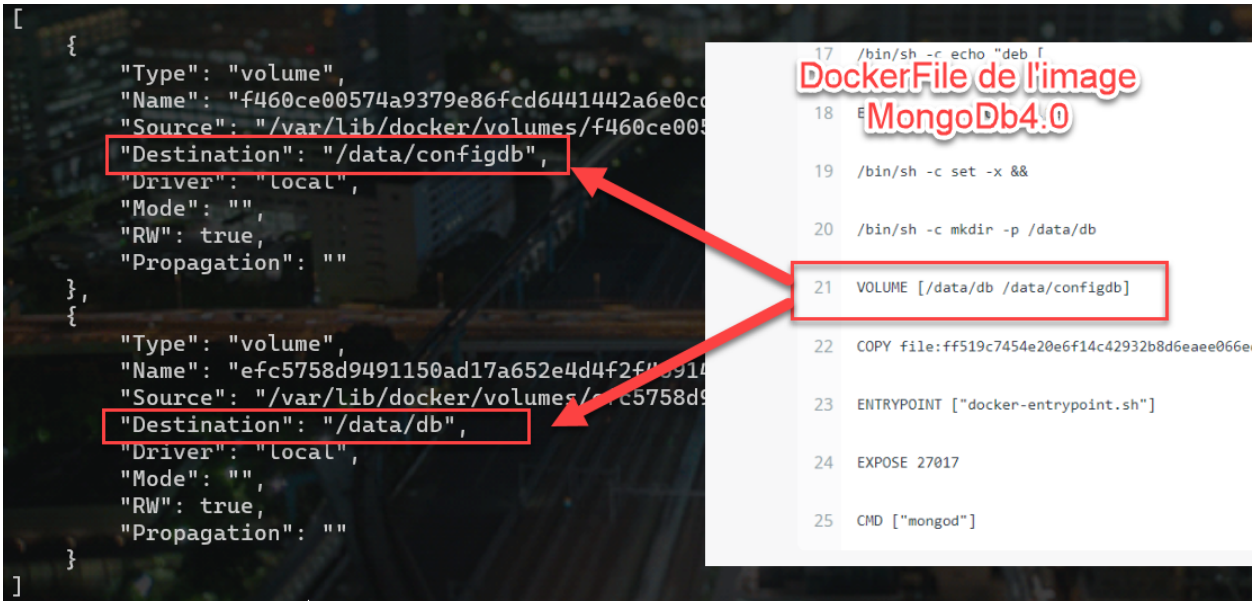
On utilise la persistance des données dans le cas de l'utilisation d'une base de données par exemple ou des fichiers de log.

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
```



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 :

```
PS C:\Users\baptiste> docker volume --help

Usage:  docker volume COMMAND

Manage volumes

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

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.

```
PS C:\Users\baptiste> docker volume ls
DRIVER    VOLUME NAME
local     db-data
local     efc5758d9491150ad17a652e4d4f2f469146ef9b397f4189ad467d0fbe944bd2
local     f460ce00574a9379e86fcd6441442a6e0cd2b6bdc08a1d0900a737c0ce978cde
```

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
```

```
PS C:\Users\baptiste> docker volume inspect db-data
[
    {
        "CreatedAt": "2022-03-10T14:50:20Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/db-data/_data",
        "Name": "db-data",
        "Options": {},
        "Scope": "local"
    }
]
```

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
```

1.7 7.0 Docker Machine

En cours de rédaction …

1.8 8.0 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.

1.9 8.1 Structure du fichier `docker.compose.yml`

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

- Les services.
- Les volumes.
- Les Networks qui permettent d'isoler les services.
- Les secrets (données 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` :

Code source 1 – Exemple d'application web

```yaml
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 fernière 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 a 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 ré-

seaux 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 besoin d'avoir accès directement. Comme le service `WEB`, qui ne doit pas pouvoir accès 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 joins, 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 déployer 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.

1.10 8.2 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).

Code source 2 – Utilisation de `docker-compose`

```
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 défini l'application. (par défault 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.

Tableau 2 – title

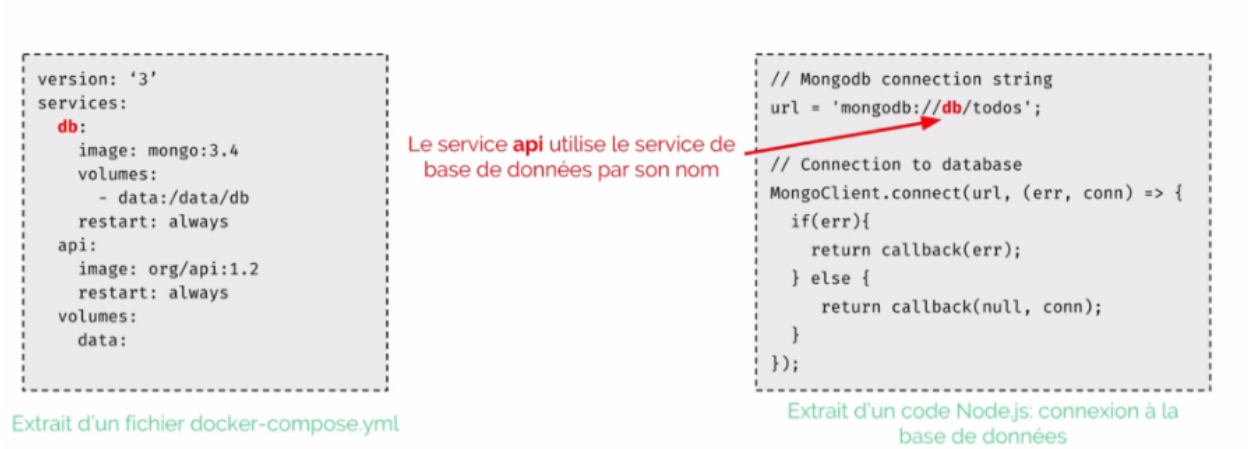
Commande	Utilisation
up / down	Création / Suppression d'une application (services, volumes, réseaux)
start / stop	Démarrage / arrête d'une application
build	Build des images des services (si instruction build utilisée)
pull	Téléchargement d'une image
logs	Visualisation des logs de l'application
scale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

1.11 8.3 Service discovery

Une application définie par docker-compose est en général constituée de plusieurs services dont certain 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 :



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

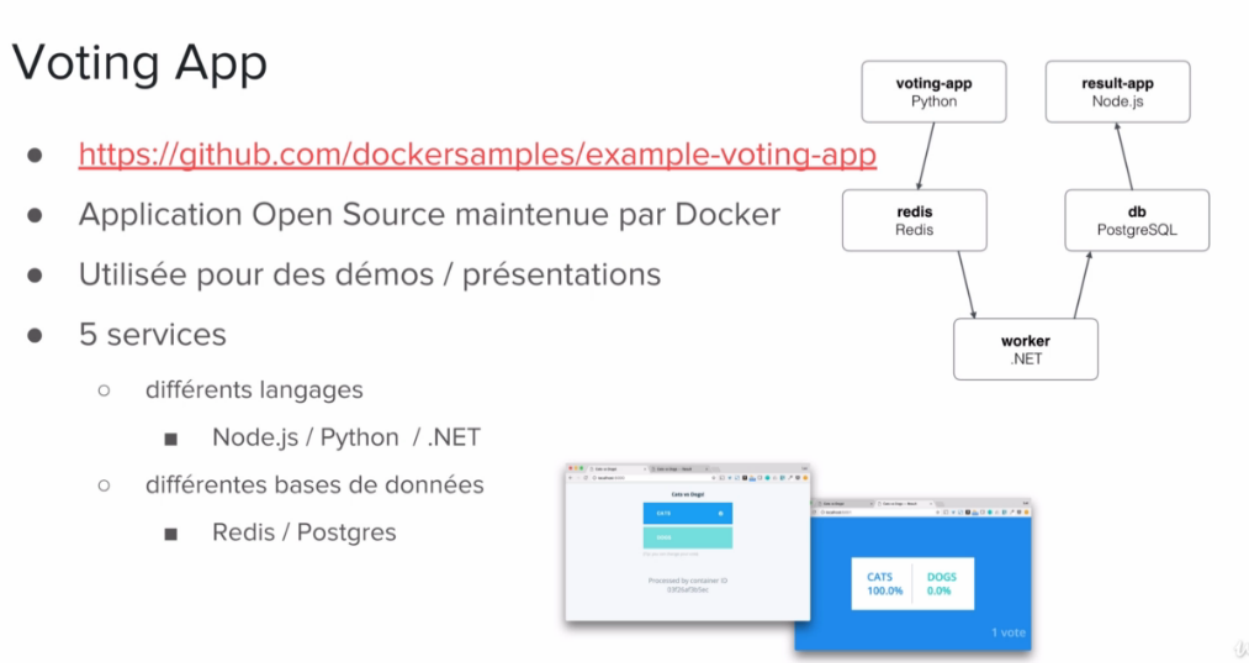
Un service est utilisé pour la base de données, `db` et un pour l'"`api` qui utilise ce service `db`.

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

A 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 renouveller 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.

1.12 8.4 Mise en oeuvre d'une application microservice : Voting App.



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 utilise 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'enregister 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 illistrent différentes 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**.

docker-compose-javaworker.yml	Set an explicit username/password for Postgres in the other Compose f...	2 years ago
docker-compose-k8s.yml	Version Was invalid, Fix to Version3 docker-compose	17 months ago
docker-compose-simple.yml	Set an explicit username/password for Postgres in the other Compose f...	2 years ago
docker-compose-windows-1809.yml	Update images for Windows 1809	3 years ago
docker-compose-windows.yml	Pin to Nano Server SAC 2016 images	4 years ago
docker-compose.seed.yml	adding data seeding	14 months ago
docker-compose.yml	healthchecks! move to compose spec version!	14 months ago
docker-stack-simple.yml	Set an explicit username/password for Postgres in the swarm and k8s f...	2 years ago
docker-stack-windows-1809.yml	Add Windows 1809 support	4 years ago
docker-stack-windows.yml	Pin to Nano Server SAC 2016 images	4 years ago
docker-stack.yml	Set an explicit username/password for Postgres in the swarm and k8s f...	2 years ago

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

Code source 3 – 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:
```

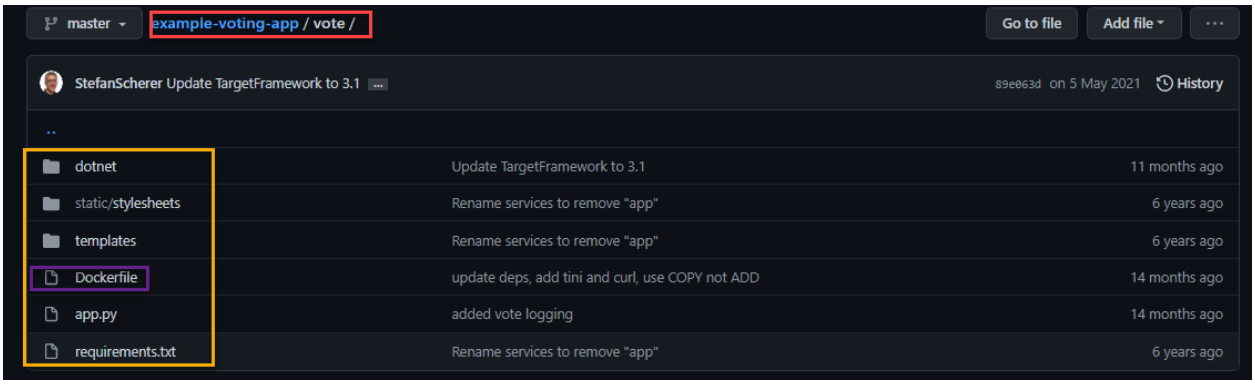
(suite sur la page suivante)

(suite de la page précédente)

```
      - "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 nécessaire 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`.



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 déployer sur un cluster **Swarm** de production.

Code source 4 – Fichier docker-stack.yml

```
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"
```

(suite sur la page suivante)

(suite de la page précédente)

```
      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
```

(suite sur la page suivante)

(suite de la page précédente)

```
          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:
```

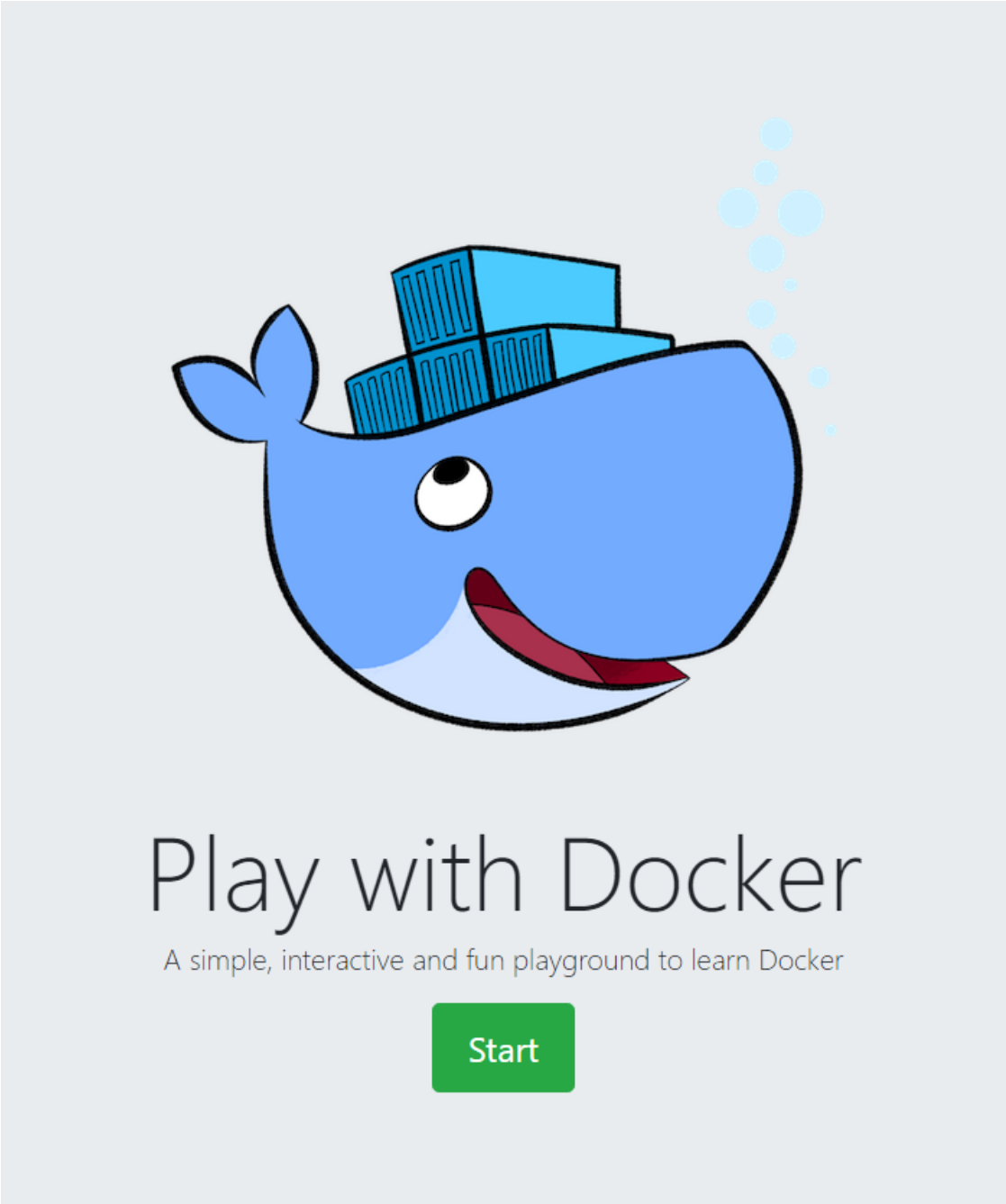
Si l'on regarde dans le service **vote** par exemple. On peut voir que contrairement au fichier `docker-compose-simple`, nous n'avons pas l'instruction `build` mais `image`. Ce qui est logique puisqu'en production nous déployons les images des services et non pas les codes applicatifs dans les containers.

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 effectué avec `update_config`. Par exemple,si un service a deux réplicas, 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`.

1.13 8.5 Voting App Installation sur `Play Docker`.

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



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

Cliquez sur : **Add New Instance**

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

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

```
##################################################################
#                            WARNING!!!!                         #
# This is a sandbox environment. Using personal credentials      #
# is HIGHLY! discouraged. Any consequences of doing so are       #
# completely the user's responsibilites.                         #
#                                                                #
# The PWD team.                                                  #
##################################################################
[node1] (local) root@192.168.0.8 ~
$ git clone https://github.com/dockersamples/example-voting-app.git
Cloning into 'example-voting-app'...
remote: Enumerating objects: 975, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 975 (delta 0), reused 0 (delta 0), pack-reused 971
Receiving objects: 100% (975/975), 992.51 KiB | 14.81 MiB/s, done.
Resolving deltas: 100% (355/355), done.
[node1] (local) root@192.168.0.8 ~
$ ls
example-voting-app
[node1] (local) root@192.168.0.8 ~
$
```

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
```

```
$ docker-compose -f docker-compose-simple.yml up -d
Creating network "example-voting-app_default" with the default driver
Building vote
Step 1/8 : FROM python:3.9-slim
3.9-slim: Pulling from library/python
c229119241af: Pull complete
5a3ae98ea812: Pull complete
46d5c684ee5f: Extracting [==============================================>     ]  10.62MB/11.58MB
fdb2b484fc86: Download complete
bbf0e1f27c8a: Download complete
```

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

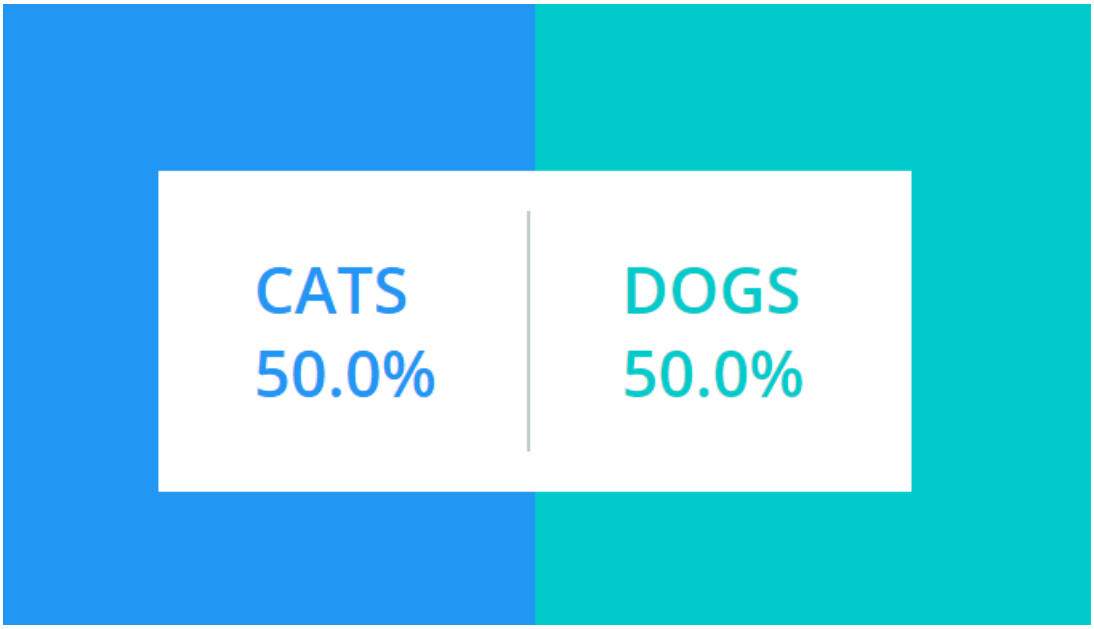
Si l'on regarde le contenu du fichier `docker-compose-simple`, nous lisons que :

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.

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

Service de vote :

Résultats des votes :



Essayez en local !

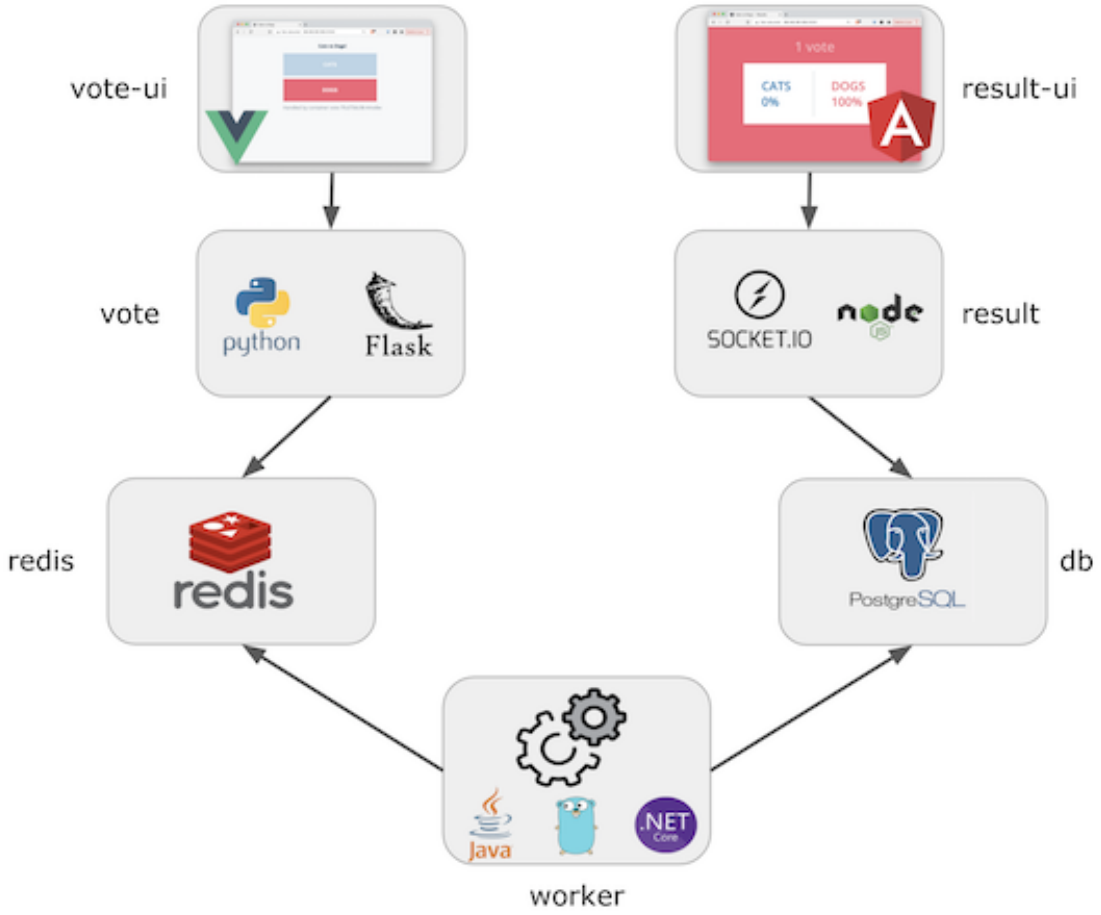
Essayez d'installer cette application en local sur votre propre machine !

1.14 8.6 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.

1.14.1 8.6.1 Vue d'ensemble

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



— **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

1.14.2 8.6.2 Récupération des repos

Code source 5 – Commandes à taper dans un dossier

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

1.14.3 8.6.3 Installation du binaire `docker-compose`

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

1.14.4 8.6.4 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 :

Code source 6 – docker-compose.yml

```
  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:
```

(suite sur la page suivante)

(suite de la page précédente)

```
      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"]
```

(suite sur la page suivante)

(suite de la page précédente)

```
      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`.

1.14.5 8.6.5 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

1.14.6 8.6.6 Les containers lancés

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

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

Code source 7 – Liste des containers lancés

```
  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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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
```

1.14.7 8.6.7 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.

1.14.8 8.6.8 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
```

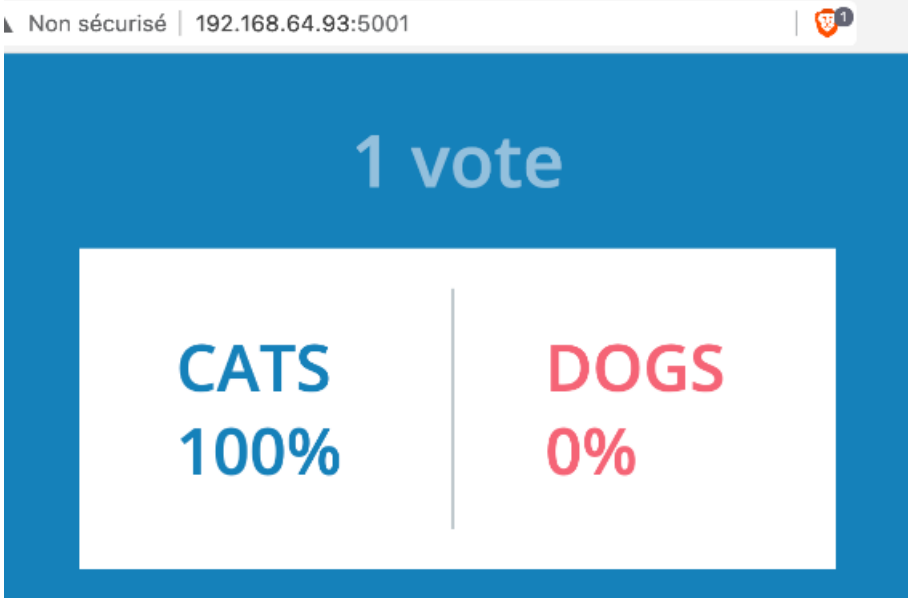
Note

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.

1.14.9 8.6.9 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

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



1.14.10 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
```

Notes : 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
```

1.14.11 8.6.11 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 supplémentaires (aggrégateur de logs, terminaison ssl, …)
- contraintes de déploiement
- …

1.15 9.0 Swarm

En cours de rédaction …

1.16 10.0 Network

En cours de rédaction …

1.17 11.0 Sécurité

En cours de rédaction …

1.18 12.0 Gestion des logs

En cours de rédaction …

1.19 13.0 Mise en pratique

Cette série de travaux reprend les concepts de base de Docker et vous guides dans leur mise en pratique.

1.19.1 13.1 Serveur HTTP avec Docker

Nous allons vous guider à chaque étape de la mise en place d'un serveur **WEB APACHE** sous **DOCKER**.

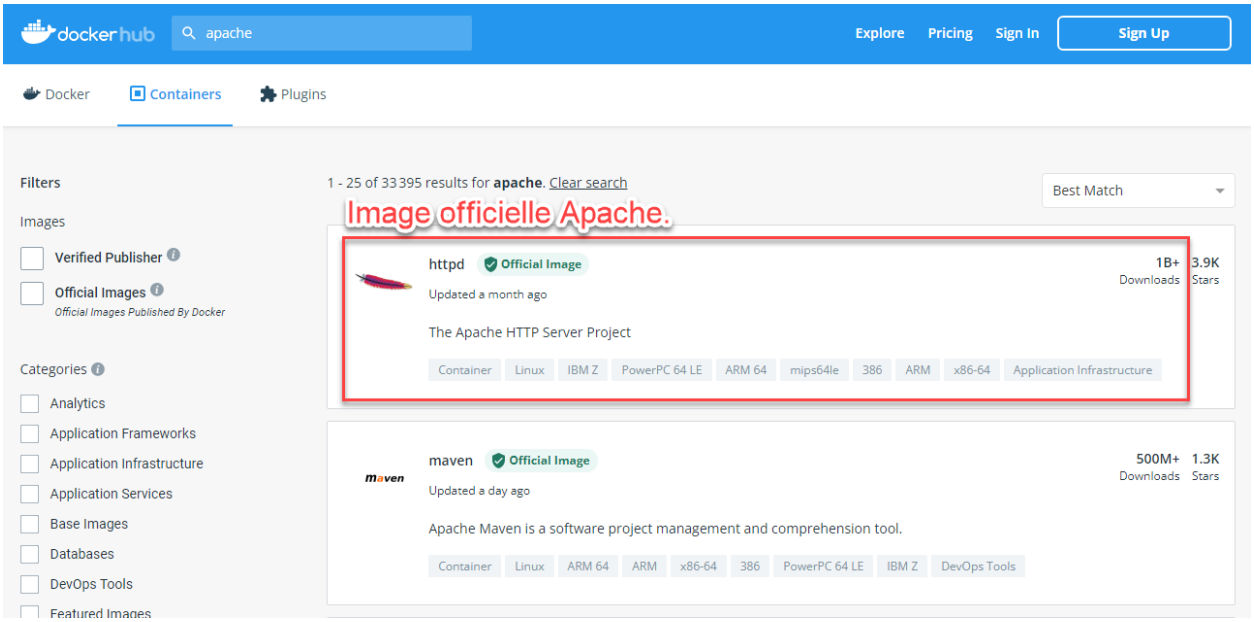
Une équipe d'**ESPORT** : **NECROMANCERS**, a confié la création de son site web à une agence de communication dont la maquette HTML est disponible dans le fichier `esport.zip`.

- **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



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**.

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
```

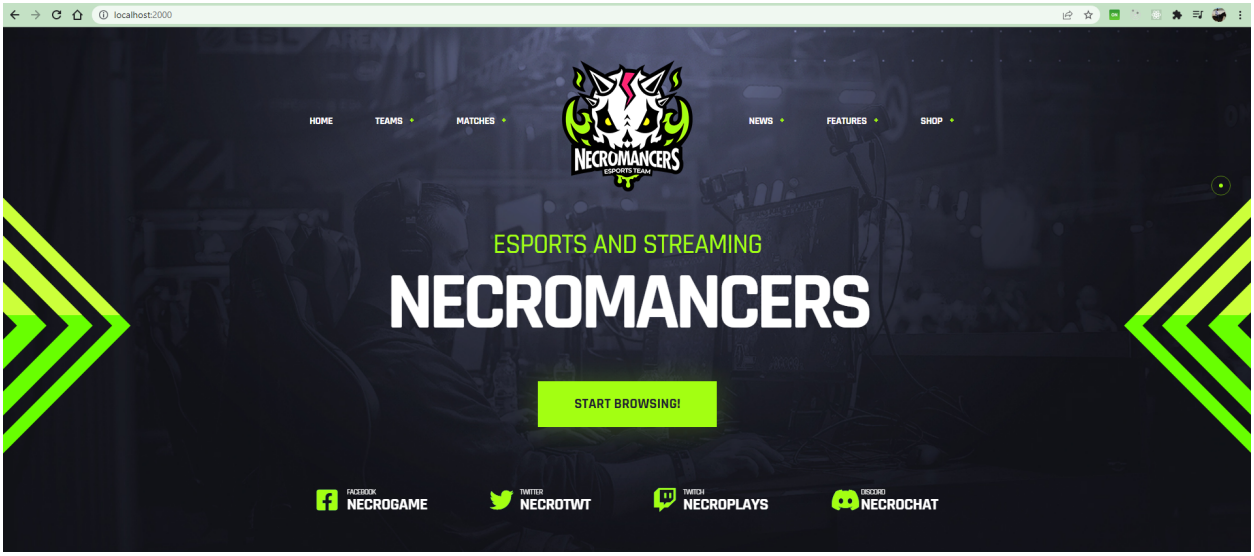
```
PS C:\Users\baptiste> docker container run -d --name serveur_http -p 2000:80 -v $PWD/Documents/docker/td/www/esport:/usr/local/apac
he2/htdocs -p 2000:80 httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
5eb5b503b376: Already exists
a43a76ccc967: Pull complete
942bd346e7f7: Pull complete
cdb155854ae6: Pull complete
10c4d45228bf: Pull complete
Digest: sha256:5cc947a200524a822883dc6ce6456d852d7c5629ab177dfbf7e38c1b4a647705
Status: Downloaded newer image for httpd:latest
137f871ec74d2cc4479b29f5d1fc1f3079e1f4b4ad48093e19613b98ad66bd8b
PS C:\Users\baptiste>
```

La variable $PWD prend la valeur du répertoire courant

Téléchargement des couches nécessaires pour faire tourner le container HTTPD nommé : serveur_http

Identifiant du container

Pour tester, ouvrez votre navigateur et saisissez l'adresse : http://localhost:2000/



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
```

```
PS C:\Users\baptiste> docker container exec -ti serveur_http sh
# ls
bin  build  cgi-bin  conf  error  htdocs  icons  include  logs  modules
# cd /usr/local/apache2/htdocs
# ls
assets                  features-bg-7.html
blog-1.html             features-contact-us.html   matches-replay.html          shop.html
blog-2.html             features-faqs.html         matches-scores.html          staff-member.html
blog-3.html             features-icons.html        matches-standings.html       streams-archive.html
blog-4.html             features-shortcodes.html   matches-stats-1.html         team-overview-2.html
blog-classic.html       features-typography.html   matches-stats-2.html         team-overview.html
blog-post.html          home.html                  matches-stats-3.html         team-player-1.html
features-404.html       index-2.html               matches-upcoming.html        team-player-2.html
features-about-us.html  index.html                 partners.html                team-player-3.html
features-bg-1.html      login-register.html        shop-2.html                  team-player-4.html
features-bg-2.html      management-and-staff.html  shop-account-billing.html    team-player-5.html
features-bg-3.html      matches-lineups-1.html     shop-account-orders.html     team-selection-1.html
features-bg-4.html      matches-lineups-2.html     shop-account-settings.html   team-selection-2.html
features-bg-5.html      matches-lineups-3.html     shop-account-shipping.html   team-selection-3.html
features-bg-6.html      matches-overview-1.html    shop-checkout.html           team-selection-4.html
#
```

FICHIERS DE NOTRE APPLICATION

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>
```

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 !

1.19.2 13.2 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.


```
-----BEGIN CERTIFICATE REQUEST-----
MIIB9TCCAWACAQAwgbgxGTAXBgNVBAoMEFF1b1ZhZGlzIExpbWl0ZWQxHDAaBgNV
BAsME0RvY3VtZW50IERlcGFydG1lbnQxOTA3BgNVBAMMMFdoeSBhcmUgeW91IGRl
Y29kaW5nIG1lPyAgVGhpcyBpcyBvbmx5IGEgdGVzdCEhITERMA8GA1UEBwwISGFt
aWx0b24xETAPBgNVBAgMCFBlbWJyb2tlMQswCQYDVQQGEwJCTTEPMA0GCSqGSIb3
DQEJARYAMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCJ9WRanG/fUvcfKiGl
EL4aRLjGt537mZ28UU9/3eiJeJznNSOuNLnF+hmabAu7H0LT4K7EdqfF+XUZW/2j
RKRYcvOUDGF9A7OjW7UfKk1In3+6QDCi7X34RE161jqoaJjrm/T18TOKcgkkhRzE
apQnIDm0Ea/HVzX/PiSOGuertwIDAQABMAsGCSqGSIb3DQEBBQOBgQBzMJdAV4QP
Awel8LzGx5uMOshezF/KfP67wJ93UW+N7zXY6AwPgoLj4Kjw+WtU684JL8Dtr9FX
ozakE+8p06BpxegR4BR3FMHf6p+0jQxUEAkAyb/mVgm66TyghDGC6/YkiKoZptXQ
98TwDIK/39WEB/V607As+KoYazQG8drorw==
-----END CERTIFICATE REQUEST-----
```


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.

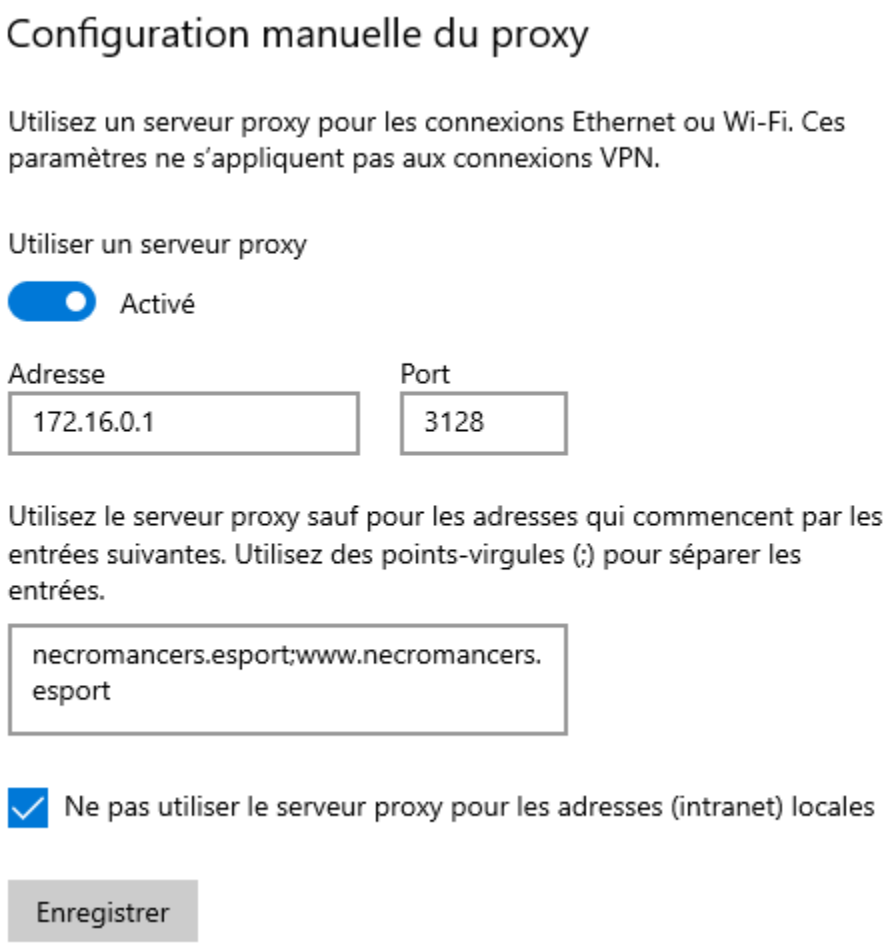
Ouvrez Visual Studio Code ou un autre éditeur comme NotePad++ avec des droits d'administrateur, et éditez le fichier :

```
C:\Windows\System32\drivers\etc\hosts
```

Ajoutez la ligne suivante :

```
127.0.0.1 necromancers.esport
```

Puis modifiez les paramètres du proxy comme suit :



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
```

```
  GNU nano 4.8                          /etc/apache2/sites-enabled/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
</VirtualHost>
```

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
```

```
  GNU nano 4.8                        /etc/apache2/sites-available/esport-ssl.conf
<IfModule mod_ssl.c>
        <VirtualHost _default_: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>
</IfModule>
```

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
```

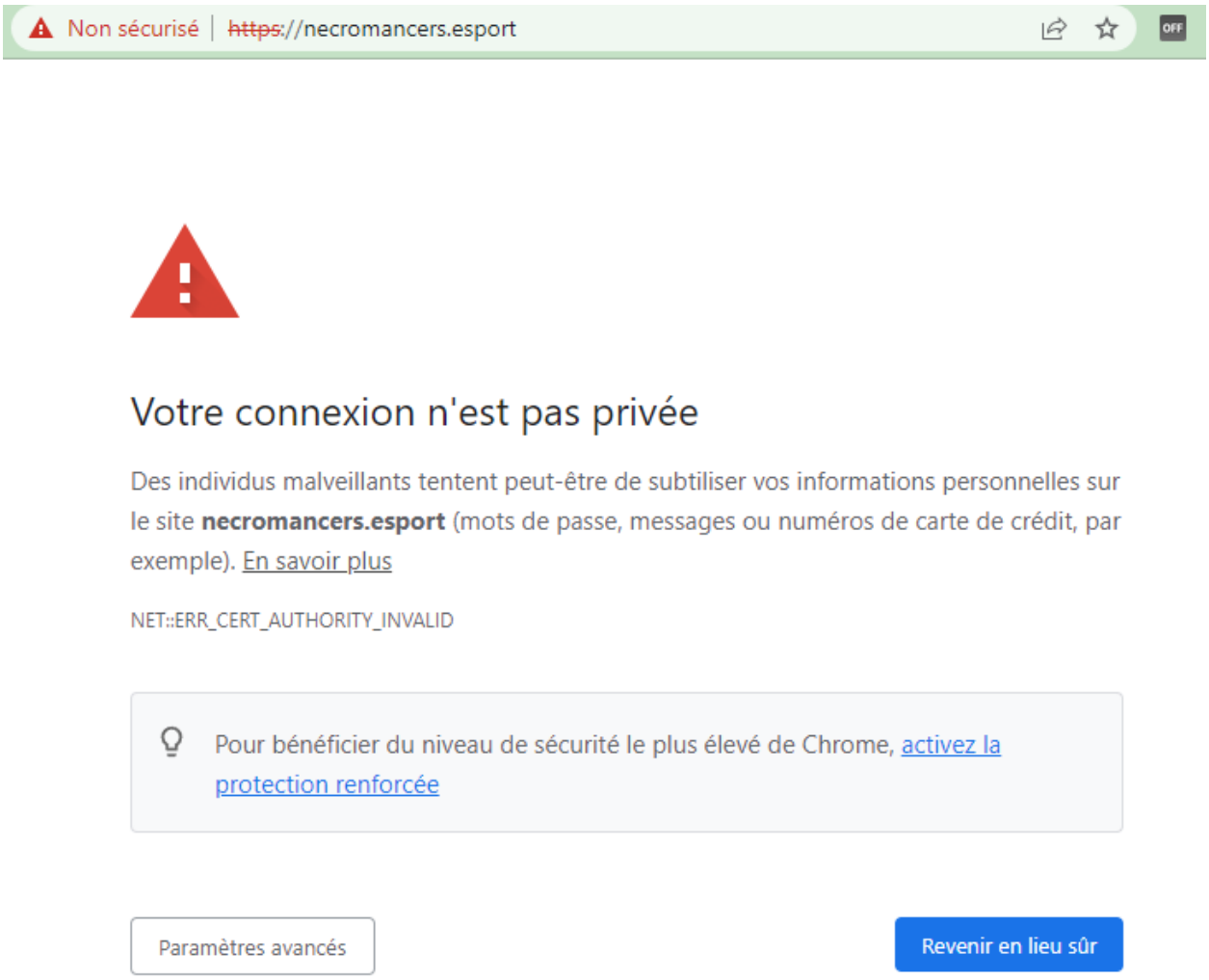
Chargeons aussi le nouveau `VirtualHost` :

```
a2ensite esport-ssl
```

En test l'adresse *https ://necromancers.esport <https ://necromancers.esport>* depuis votre navigateur, Vous devriez avoir cela :

```
# apachectl configtest
Syntax OK
```

Il faut autoriser la connexion au site :



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
```

```
PS C:\Users\baptiste\Documents\docker\td\www> docker ps -a
CONTAINER ID   IMAGE     COMMAND            CREATED       STATUS                      PORTS     NAMES
00e15c9f63ea   alpine    "/bin/sh"          2 hours ago   Exited (137) 14 minutes ago           serveur_https
137f871ec74d   httpd     "httpd-foreground" 2 days ago    Exited (0) 2 hours ago                serveur_http
```

`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 :

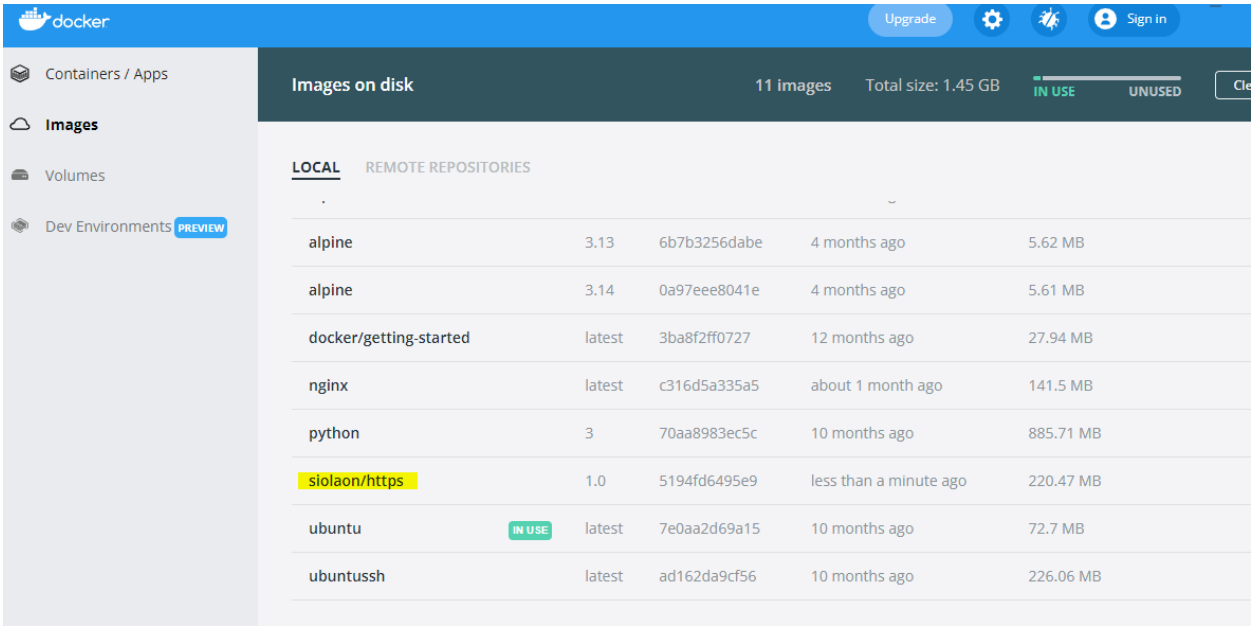
```
docker commit -a Bauer 00e15c9f63ea siolaon/https:1.0
```

Vérifions si l'image a bien été créée en listant les images disponibles sur notre machine hôte.

```
docker images
```

```
PS C:\Users\p02\Documents\Cours\cours-sio> docker images
REPOSITORY              TAG       IMAGE ID       CREATED          SIZE
siolaon/https           1.0       5194fd6495e9   45 seconds ago   220MB
```

Nous pouvons retrouver l'image également dans l'application Docker Desktop, onglet « Images ».



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
```

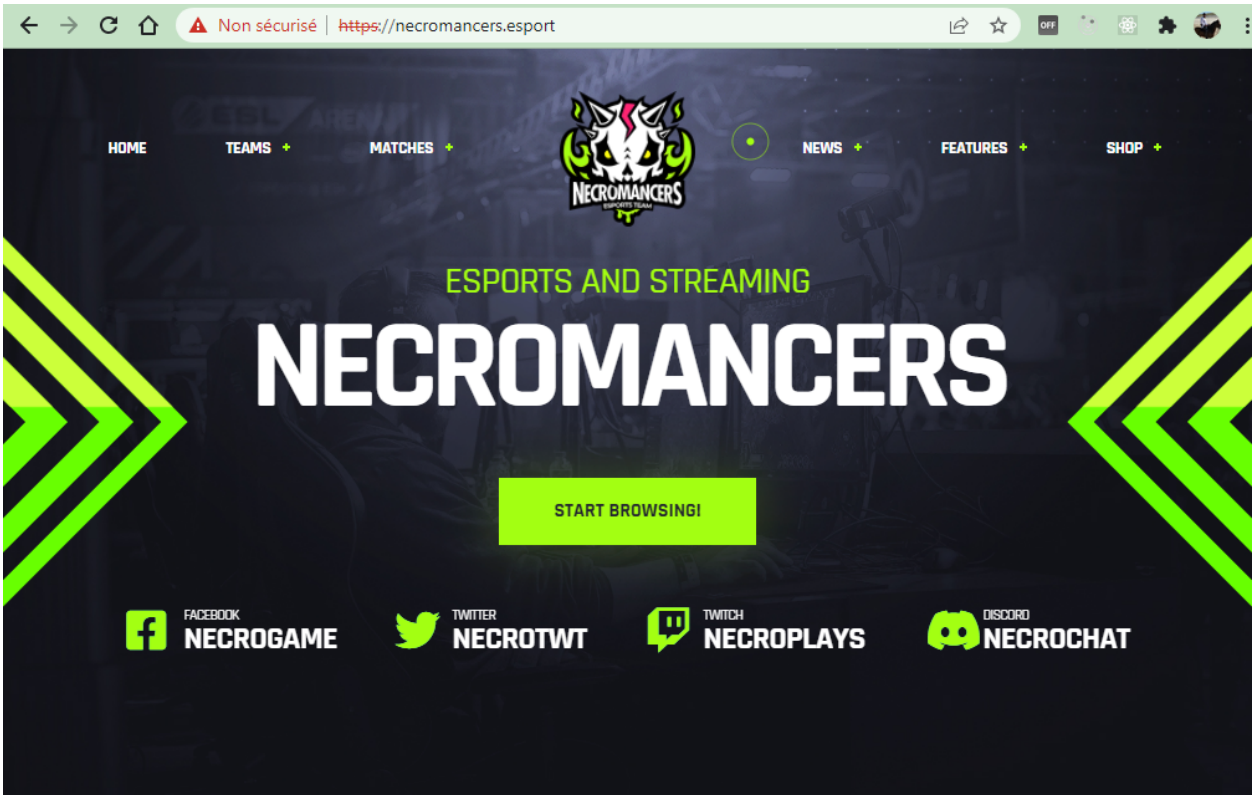
Maintenant il faut lancer le serveur apache2 manuellement depuis le serveur :

```
docker container exec -ti server_esport sh
```

et dans le `shell` lancer la commande :

```
service apache2 start
```

Ouvrez le navigateur et contemplez votre oeuvre :



1.19.3 13.3 Création d'un Dockerfile

Nous sommes satisfait 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 :

esport	03/03/2022 15:02	Dossier de fichiers	
Dockerfile	04/03/2022 08:52	Fichier	1 Ko
esport.conf	03/03/2022 15:06	Fichier CONF	1 Ko
esport-ssl.conf	03/03/2022 17:04	Fichier CONF	1 Ko

— 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.

Avertissement : 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
```

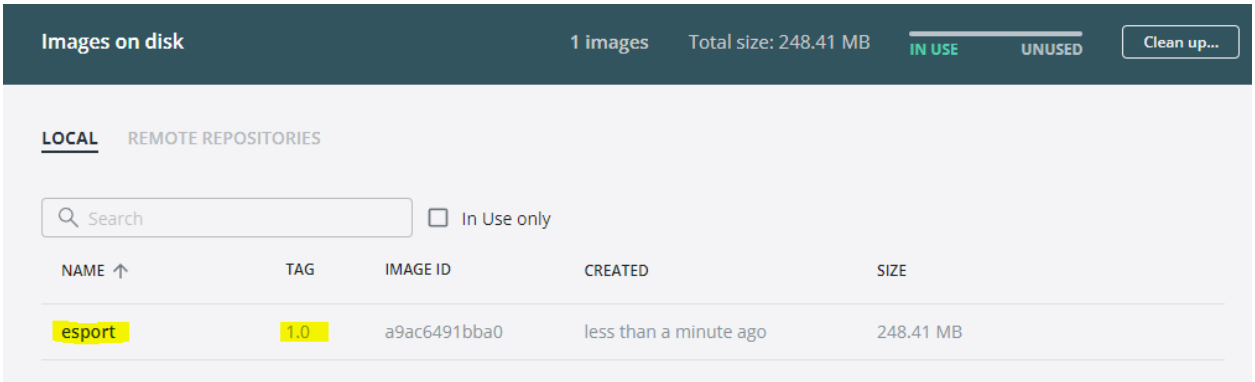
Créons maintenant notre image nommée esport dans sa version 1.0. La création peut prendre un certain temps.

Avertissement : N'oubliez pas le « . » !

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

```
Windows PowerShell
 => => transferring dockerfile: 761B
     0.1s
 => [internal] load .dockerignore
     0.8s
 => => transferring context: 2B
     0.1s
 => [internal] load metadata for docker.io/library/ubuntu:latest
     3.7s
 => [1/5] FROM docker.io/library/ubuntu:latest@sha256:8ae9bafbb64f63a50caab98fd3a5e37b3eb837a3e0780b78e5218e63
193  6.2s
 => => resolve docker.io/library/ubuntu:latest@sha256:8ae9bafbb64f63a50caab98fd3a5e37b3eb837a3e0780b78e5218e63
193  0.3s
[+] Building 105.9s (5/9)
 => [internal] load build definition from Dockerfile                                                    0.6s
 => => transferring dockerfile: 761B                                                                    0.1s
 => [internal] load .dockerignore                                                                       0.8s
 => => transferring context: 2B                                                                         0.1s
 => [internal] load metadata for docker.io/library/ubuntu:latest                                        3.7s
 => [1/5] FROM docker.io/library/ubuntu:latest@sha256:8ae9bafbb64f63a50caab98fd3a5e37b3eb837a3e0780b78e  6.2s
 => => resolve docker.io/library/ubuntu:latest@sha256:8ae9bafbb64f63a50caab98fd3a5e37b3eb837a3e0780b78e  0.3s
 => => sha256:8ae9bafbb64f63a50caab98fd3a5e37b3eb837a3e0780b78e5218e63193961f9 1.42kB / 1.42kB          0.0s
 => => sha256:9c152418e380c6e6dd7e19567bb6762b67e22b1d0612e4f5074bda6e6040c64a 529B / 529B              0.0s
 => => sha256:2b4cba85892afc2ad8ce258a8e3d9daa4a1626ba380677cee93ef2338da442ab 1.46kB / 1.46kB          0.0s
 => => sha256:7c3b88808835aa80f1ef7f03083c5ae781d0f44e644537cd72de4ce6c5e62e00 28.57MB / 28.57MB        2.7s
 => => extracting sha256:7c3b88808835aa80f1ef7f03083c5ae781d0f44e644537cd72de4ce6c5e62e00               1.8s
 => [internal] load build context                                                                       0.8s
 => => transferring context: 36.35kB                                                                    0.1sa
 => [2/5] RUN apt update &&     apt install apache2 -y &&     echo 'ServerName localhost'  >> /etc/apa  94.4s
 => => # 86_64-linux-gnu/perl/5.30 /usr/share/perl/5.30 /usr/local/lib/site_perl /usr/lib/x86_64-linux-gnu/pe
 => => # rl-base) at (eval 19) line 2.)
 => => # debconf: falling back to frontend: Noninteractive
 => => # Setting up libbrotli1:amd64 (1.0.7-6ubuntu0.1) ...
 => => # Setting up libsqlite3-0:amd64 (3.31.1-4ubuntu0.2) ...
 => => # Setting up libsasl2-modules:amd64 (2.1.27+dfsg-2ubuntu0.1) ...
```

Notre image apparait bien dans Docker Desktop.



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
```

CHAPITRE 2

Javascript

Javascript peut être déroutant à cause de certains concepts comme `Prototypes`, les fonctions `callback` , le `Scope` et le `Hoisting` et beaucoup d'autres, surtout quand on est débutant en Javascript. Mais rassurez vous ! Javascript peut aussi jouer des tours même aux développeurs les plus expérimentés.

2.1 Le Scope ou la portée des variables

2.1.1 Définition d'un scope

Le scope est un des concepts clés de Javascript.

La **portée** ou **scope**, désigne l'accessibilité des variables, autrement dit : Où et quand une variable peut être utilisée par votre programme.

Nous désignons 3 types de `scope` :

— Le `Global Scope`

Code source 1 – **Global Scope**

```
// GLOBAL SCOPE
var animal = "Elephant";
var population = 10;
```

Les variables sont déclarées dans le script en dehors d'un bloc d'instruction ou d'une fonction.

— Le `Local Scope`

Code source 2 – **Function (local) Scope**

```
// FUNCTION SCOPE
function zoo() {
  var animal = "Tigre";
  var population = 5;
```

(suite sur la page suivante)

(suite de la page précédente)

```
  console.log(animal, population);
}
```

Les variables sont déclarées dans le corps de la fonction `zoo`.

— Le `Block Scope`, dans un bloc d'instruction(`if`, `else`, `while`, `for` ..).

Code source 3 – **Block scope**

```
// BLOCK SCOPE
for(var population = 0; population <10 ; population++) {
  console.log(animal, population);
}
```

La variable `population` est déclarée dans le block déclaratif de la boucle `for`.

2.1.2 Etude de la portée des variables

Lisez le code suivant :

```
var var1 = 10;
console.log(var1);
```

`var1` est déclaré dans le **global scope**. Nous pouvons afficher sa valeur sans problème avec un `console.log`.

Ajoutons maintenant un block conditionnel et tentons d'accèder depuis ce block à notre variable `var1`.

```
var var1 = 10;
console.log(`Global Scope 1 : ${var1}`);
if(true) {
  console.log(`Block Scope 1 : ${var1}`);
  var1++;
  console.log(`Block Scope 2 : ${var1}`);
}
console.log(`Global Scope 2 : ${var1}`);
```

La console retourne :

```
"Global Scope 1 : 10"
"Block Scope 1 : 10"
"Block Scope 2 : 11"
"Global Scope 2 : 11"
```

Observation 1

Une variable déclarée dans le **Global Scope** peut être lue et modifiée dans le **Block Scope**.

Déclarons maintenant une variable `var2` dans le **Block** et tentons d'y accèder depuis l'extérieur.

```
if(true) {
  var var1 = 10;
}
console.log(`Global Scope 1 : ${var1}`);
```

La console retourne :

```
"Global Scope 1 : 10"
```

Observation 2

Une variable déclarée dans le **Block Scope** peut être lue et modifiée dans le **Global Scope**.

Maintenant déclarons une fonction `increment` :

```
var var1 = 10;
function increment() {
  var1++;
}
increment();
console.log(`Global Scope 1 : ${var1}`);
```

La console retourne :

```
"Global Scope 1 : 11"
```

Observation 3

Une variable déclarée dans le **Global Scope** peut être lue et modifiée dans le **Function Scope**. C'est ce que l'on appelle un **effet de bord** ou **side effect**.

Avertissement : Attention aux effets de bord des fonctions que vous écrivez dans vos scripts. Ils peuvent complexifier leur lecture et leur débogage. Car une variable déclarée dans le **Global Scope** pourrait être modifiée par de nombreuses fonctions à **effets de bord** rendant difficile la prévision des valeurs durant le cycle de vie de votre script.

Déclarons une nouvelle variable dans le corps de la fonction.

```
var var1 = 10;
function increment() {
  var var2 = 20;
  var1++;
}
increment();
console.log(`Global Scope 1 : ${var2}`);
```

La console retourne :

```
"Uncaught ReferenceError: var2 is not defined"
```

Observation 4

Une variable déclarée dans le **Function Scope** ne peut être lue et modifiée dans le **Global Scope**.

Prenons un nouveau cas de figure :

Nous déclarons `var1` dans le **Global Scope** puis nous déclarons une nouvelle variable avec le même nom `var1` dans un **Block Scope**.

```
var var1 = 10;
console.log(`Global Scope 1 : ${var1}`);
if(true) {
  var var1 = 100;
  console.log(`Function Scope 1 : ${var1}`);
}
console.log(`Global Scope 2 : ${var1}`);
```

La console retourne :

```
"Global Scope 1 : 10"
"Function Scope 1 : 100"
"Global Scope 2 : 100"
```

Observation 5

Il n'existe pas de portée de Block pour les variables `var`.

La version **ECMAScript2015** a introduit dans Javascript deux nouveaux mots clés : `let` et `const`. `let` permet de pallier au problème soulevé dans l'**Observation 5**. Reprenons le code précédent et remplaçons maintenant `var` par `let`.

```
let var1 = 10;
console.log(`Global Scope 1 : ${var1}`);
if(true) {
  let var1 = 100;
  console.log(`Function Scope 1 : ${var1}`);
}
console.log(`Global Scope 2 : ${var1}`);
```

La console retourne :

```
"Global Scope 1 : 10"
"Function Scope 1 : 100"
"Global Scope 2 : 10"
```

Observation 6

Les variables créées avec le mot clé `let` appartiennent au scope dans lequel elles ont été définies. Toutefois, elles restent accessible et modifiable dans les blocks enfants.

```
let var1 = 10;
console.log(`Global Scope 1 : ${var1}`);
if(true) {
  var1 = 100;
  console.log(`Function Scope 1 : ${var1}`);
}
console.log(`Global Scope 2 : ${var1}`);
```

Note : Il faut donc privilégier l'utilisation du mot clé `let` au lieu de `var` afin d'éviter des désagréments.

Le mot clé `const` sert à déclarer une référence constante. **Attention**, une référence **constante** ne veut pas dire que la valeur derrière la référence est « **immutable** », mais que la référence elle-même est **immutable**.

```
const a = 10;
a++;
```

Nous essayons de modifier la valeur de `a` déclaré pourtant comme étant une constante `const`. Cela est interdit.

Toutefois :

```
const a = {txt: "hello"}; //Référence r0 vers l'objet
const b = a; //Référence r0 vers le même objet
a.txt += " world" //Adresse 0x0002
console.log(a); //"hello world";
console.log(b); //"hello world";
```

Ici, `a` est de type complexe, c'est un objet. Ce n'est pas sa valeur qui est stockée dans la variable mais une référence de l'objet, l'adresse mémoire qu'il occupe. Ainsi, un objet peut être déclaré avec le mot clé `const` et se voir ajouter, modifier, supprimer des éléments.

Nous n'allons pas tester toutes les scopes avec `const`.

Voici un tableau comparatif :

	var	let	const
Stockée en global	✓	✗	✗
Portée de la fonction	✓	✓	✓
Portée du block	✗	✓	✓
Ré-assignation	✓	✓	✗
Re-déclaration	✓	✗	✗
Hoisting	✓	✗	✗

2.1.3 Exercice

Code source 4 – Exercice 1 : Modifier le script pour que la console retourne la valeur 10.

```
var i = 10;
for(var i =0;i<=5;i++) {
  // Do Stuff
}
console.log(i)
```

2.2 Hoisting

Le **Hoisting** est un concept propre au moteur Javascript sur la manière d'exécuter les scripts. Cela permet à Javascript de répertorier les variables ou les fonctions qui seront utilisées **AVANT** leur utilisation dans un script.

Prenons un exemple pour comprendre :

```
1 let result = 1;
2
3 console.log(addOne(3)); // Retourne 4
4 console.log(result); // Retourne 4
5
6 function addOne(numToAdd) {
7  result = result + numToAdd;
8  return result;
9 }
```

Ici, nous appelons `addOne` à la **ligne 3** alors que nous l'avons déclaré plus bas, de la **ligne 6 à 9**, nous devrions avoir un message d'erreur. Mais grâce au mécanisme de **Hoisting**, Javascript a été capable de parcourir notre fichier avant qu'il ne s'execute et a mémorisé toutes les déclarations de fonctions.

Mais il faut garder à l'esprit que cela n'est valable que dans le cadre d'une déclaration de fonction avec le mot clé `function`.

Si nous avions une fonction anonyme stockée dans une variable comme suit :

```
1 let result = 1;
2
3 console.log(addOne(3)); // Retourne 4
4 console.log(result); // Retourne 4
5
6 const addOne = function (numToAdd) {
7  result = result + numToAdd;
8  return result;
9 }
```

La variable `addOne` sera bien entendu référencée par le mécanisme du **Hoisting** mais il ne sera pas possible d'appeler la fonction associée.

Soyez donc vigilant si vous utilisez cette syntaxe.

Pour comprendre ce que fait le **hoisting** nous allons proposer ce script simple :

```
let var1 = 100;
let var2 = increment(var1);
let var3 = decrement(var2);

function increment(value) {
  return value++;
}

const decrement = function(value) {
  return value--;
}
```

Code source 5 – Le même script réorganisé par le moteur Javascript : Hoisting

```
function increment(value) {
  return value++;
}

let var1;
let var2;
let var3;
const decrement;

var1 = 100;
var2 = increment(var1);
var3 = decrement(var2);
decrement = function(value) {
  return value--;
}
```

La déclaration de la fonction `increment` est bien déplacée, levée au haut du script, car elle a été déclarée avec le mot clé `function`.

Les variables `var1`, `var2`, `var3` viennent ensuite, mais aucune valeur ne leur a été affecté, elles sont définies pour le moment comme `undefined`.

La fonction `decrement` est considérée comme étant une variable de par sa syntaxe déclarative.

A retenir

Le moteur de **Javascript**, fait un traitement du script avant de l'**exécuter**. Il va bouger les déclarations de variables tout en haut du script, ainsi que les déclarations de fonctions.

Puis les affectations des valeurs seront faites uniquement lors de l'execution.

2.3 Comprendre les fonctions et le mot clé `This`

2.3.1 Le mot clé `This`

Javascript a beaucoup évolue depuis ses premières version, et beaucoup de concept de base ont vu leur fonctionnement modifié.

Il est donc nécessaire de faire un tour d'horizon et revoir ensemble les fondamentaux de Javascript.

Dans un premier temps, prenons une fonction simple

```
function myFunction() {
  console.log(this===window);
}
myFunction() // affiche : "true"
```

Cette fonction ne fait pas grand chose ormi comparer la stricte égalité de `this` avec `window`.

`window` est une variable globale qui represente la fenêtre du navigateur. Si l'on écrirait ce code depuis **NodeJs**, nous aurions mis à la place `global`.

En executant ce bout de code, nous voyons que la valeur de retour est `true`.

Compliquons légérement les choses, créons un objet qui contiendra une fonction qui fera la meme chose que précédement.

```
var myObject = {
  myFunction: function() {
    console.log(this === myObject);
  }
}

myObject.myFunction(); // affiche : "true"
```

L'utilisation de cet exemple permet de montrer que le pointeur `this` est placé sur l'objet `myObject` et non plus sur `window`. Et cela fonctionne comme nous pourrions l'attendre, `this` sert à appeler le contexte dans lequel une fonction est associée. Et c'est le comportement que l'on retrouve dans des langages orientés objets comme le **C#** ou **java**.

> **Avertissement :** Le terme **pointeur** employé ici n'a rien à voir avec le concept de pointeur en C.

Changeons maintenant légérement les choses dans notre code. Nous allons garder exactement la même déclaration de l'objet `myObject`, mais lieu d'appeler `myFunction` depuis l'objet directement nous allons créer une variable `myFunction` qui contiendra `myObject.myFunction`.

```
var myObject = {
  myFunction: function() {
    console.log(this === myObject);
  }
}

var myFunction = myObject.myFunction;
myFunction(); // affiche : "false"
```

Nous voyons que maintenant `this` n'est plus égale au contexte de la fonction `myFunction` mais à `window`.

Régle Javascript

Si il est appelé depuis `obj.func()` alors `this` égale `obj`. C'est à dire depuis la fonction elle-même dans son contexte.

Sinon , `this` égale `window` ou `global`.

C'est comme cela que les choses fonctionnent avec Javascript.

Prenons un autre exemple :

```
1  var myObject = {
2    myFunction: function() {
3      console.log(this===myObject); // affiche: "true"
4      setTimeout(function() {
5        console.log(this === myObject); // affiche: "false"
6        console.log(this === window); // affiche: "true"
7      },0);
8    }
9  }
10 myObject.myFunction();
```

Nous retrouvons `myObject` mais nous y avons ajouté une fonction asynchrone `setTimeout` (Nous aurions pu utiliser n'importe quelle autre fonction qui possède un `callback`).

Et nous appelons la fonction `myFunction` depuis l'objet directement. nous constatons qu'à la ligne 3, `this` est « égale » à `myObject`. Alors qu'à la ligne 5, dans la fonction **callback**, `this` égale à `window` à la place de `myObject`.

Pourquoi cela ? Pour comprendre, il faut se rapporter à la règle émise plus haut.

A la ligne 3, `this` est invoqué depuis `myFunction` par l'intermédiaire de la référence à l'objet `myObject` elle-même. Or, à la ligne 5, `this` est appelé depuis une fonction anonyme, qui n'est référencé dans aucun objet. C'est la fonction `setTimeout` qui l'appelle. Donc `this` égal à `window`.

2.3.2 Expression de fonction vs Déclaration de fonction

```
function myFunctionDeclaration() {}

var myFunctionExpression = function() {};
```

Depuis ES5, il possible de déclarer des fonctions Javascript de 2 manières différentes comme le montre le code ci-dessus.

Ces déclarations semblent différentes mais font exactement la même chose.

Toutefois comme nous l'avons expliqué dans le cours sur le `hoisting` il existe une différence lors de la déclaration et de l'invocation de la fonction.

Nous pouvons parfaitement utiliser une fonction déclarée après son appel, car le `hoisting` va se charger de remonter la déclaration tout en haut du script.

```
myfunction(); // Affiche: "Hello"

function myFunction() {
  console.log("hello");
}
```

Par contre nous aurons une erreur en utilisant la syntaxe suivante :

```
myfunction(); // Affiche: Uncaught ReferenceError: myfunction is not defined"

var myFunction = function () {
  console.log("hello");
}
```

Car le mécanisme de `hoisting` sépare les variable en deux parties : La déclaration et l'affectation. Il déplace la partie déclarative en haut du script et laisse l'affectation là où elle a été mise.

Dans notre code `var myFunction` est considéré comme une déclaration de variable et c'est ce qu'elle est : une variable auquelle est affectée une référence à une fonction anonyme. Et à la ligne 1, `myFunction` égale à `undefined`.

```
var myFunction;

myfunction();

myFunction = function () {
  console.log("hello");
}
```

2.3.3 Expressions de fonction nommée

```
var myFunction = function myOtherFunction(recurse) {
      if(recurse) {
              myFunction(false); // OK
              myOtherFunction(false); // OK
      }
};

myFunction(true); // OK
myOtherFunction(true); // ReferenceError
```

Nous avons déclaré une fonction nommée `myOtherFunction` dont la référence est assignée à la variable `myFunction`.

A l'intérieur de `myOtherFunction`, nous appelons : `myFunction` et `myOtherFunction`, et nous avons le droit de le faire.

Par contre, si à l'extérieur nous appelons `myOtherFunction` directement, nous aurons un message d'erreur de référence. Seul l'appel par `myFunction` sera valide.

2.3.4 `Call`, `apply` et `bind` : initialisation manuelle de `this`

Précédement, nous avons mis en évidence que `this` est de nouveau assigné à `global` ou `window` s'il est utilisé au sein d'une fonction asynchrone.

Etudions avec ce script comment changer la valeur de `this` dans une fonction avec les méthodes `call`, `apply` et `bind`.

call

Etudions le cas de la méthode `call` :

```
var myObject = {
  myFunction: function(a, b) {
    console.log(a + ' ' + b); // affiche : "Hello world"
    console.log(this === myObject); // False
    console.log(this === myOtherObject); // True
  }
}

var myOtherObject = {}

myObject.myFunction.call(myOtherObject, 'hello', 'world');
```

Nous créons un objet quelconque : `myOtherObject`. Nous appellons la méthode `myFunction` de l'objet `myObject`, mais nous souhaitons que la référence de `this` de `myFunction` soit celle d'un autre objet extérieur, `myOtherObject`. Cela est possible grâce à la méthode `call`, qui prend en premier argument l'objet dont vous voulons utiliser la référence et les autres arguments suivants seront ceux nécessaires à l'utilisation de la fonction `myFunction`.

apply

Il existe une autre syntaxe qui fait exactement la même chose que `call` mais avec la méthode `apply`. La seule différence réside dans la manière dont sont passées les arguments à la fonction : ils sont placés dans un tableau.

```
myObject.myFunction.apply(myOtherObject, ['hello', 'world']);
```

bind

Et finalement nous avons `bind` qui fonctionne presque pareil que `call` ormi du fait qu'il sépare la procédure d'utilisation en deux étapes séparées.

```
var myObject = {
  myFunction: function(a, b) {
    console.log(a + ' ' + b); // affiche : "Hello world"
    console.log(this === myObject); // False
    console.log(this === myOtherObject); // True
  }
}

var myOtherObject = {}

var myFunction = myObject.myFunction.bind(myOtherObject);
myFunction('hello', 'world');
```

Nous obtenons une nouvelle fonction qui possède un contexte de `this` prédéfinie, qui n'est pas celui de l'objet parent dans laquelle la fonction est déclarée, mais de l'objet `myOtherObject` passé en argument à la méthode `bind`. Nous l'avons assigné à une variable qui peut être ensuite utilisé comme une fonction classique.

`bind` est typiquement utilisé si nous avons besoin de forcer le pointeur de `this` d'une fonction **callback** par exemple.

2.3.5 Notation abrégée des objets

```
1  const myObject = {
2    myFunction() {
3      console.log(this === myObject);
4    }
5  };
6
7  myObject.myFunction(); // true
8
9  const myFunction = myObject.myFunction;
10
11 myFunction(); // false
```

Nous utilisons ici `const` pour déclarer notre objet `myObject` à la place de `var`. Et contrairement aux exemples précédents nous avons déclaré la fonction `myFunction` directement en la nommant sans utiliser le mot clé `function` et sans l'avoir assigné à une clé d'objet comme : `objectKey : function() {}`.

Cette nouvelle syntaxe introduite par **ECMA2015** permet de raccourcir les déclarations de fonction dans un objet Javascript tout en restant lisible.

A la ligne 7, nous voyons que le pointeur `this` de la fonction `myFunction` est égale à son objet parent. Toutefois lorsque nous faisons un alias à la ligne 9, `this` change de valeur. Cette syntaxe offre donc le même comportement pour la valeur de `this` qu'avec une syntaxe avec les `:` et le mot clé `function`.

2.3.6 Fonctions fléchées

Es2015 ajoute une nouvelle syntaxe pour la déclaration des fonctions en javascript : **Les fonctions fléchées**.

```
1  const myFunction = () => {
2          console.log(this === windows ); // True
3  }
4  myFunction();
```

Rappelez vous maintenant de cet exemple vu plus haut :

```
1  var myObject = {
2    myFunction: function() {
3      console.log(this===myObject); // affiche: "true"
4      setTimeout(function() {
5        console.log(this === myObject); // affiche: "false"
6        console.log(this === window); // affiche: "true"
7      },0);
8    }
9  }
10 myObject.myFunction();
```

Nous en avons conclu que le pointeur de `this` changeait dans la fonction anonyme **callback** de `setTimeout` pour prendre celui de `windows` ou `global`.

Réécrivons ce bout de code avec les nouvelles notations abordées précédement :

```
1  var myObject = {
2    myFunction() {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    console.log(this===myObject); // affiche: "true"
    setTimeout(() => {
      console.log(this === myObject); // affiche: "true"
      console.log(this === window); // affiche: "false"
    },0);
  }
}
myObject.myFunction();
```

Nous constatons que les résultats sont inversés. En utilisant les fonctions fléchés comme ci-dessus nous conservons le pointeur de `this`, qui correspond à l'objet `myObject`.

2.3.7 Fonctions fléchées et `.call()`

Vous vous rappelez de la méthode `.call()` ?

Etudions son comportement avec les fonctions fléchées.

```
const myObject = {};

const myFunction = () => {
  console.log(this === myObject);
};

myFunction(); // False

myFunction.call(myObject); // False
```

Nous avons déclaré un objet quelconque `myObject`. Nous souhaitons déplacer le pointeur de `this` vers l'objet `myObject` avec la méthode `.call()` comme nous l'avons vu avec les fonctions déclarée avec le mot clé `function`.

Et contre toute attente, nous faisons le constat que cela n'est pas possible !

Une fonction fléchée est une alternative compacte aux expressions de fonctions traditionnelles. elles ne peuvent pas être utilisé cependant dans toutes les situations.

- `this` et `super` dans leur corps ne peuvent pas se lier à leur parent, nous ne devons donc pas les utiliser comme méthode d'un objet.
- Les fonctions fléchées n'ont pas accès au mot clé : `new.target`.
- Les fonctions fléchées ne peuvent pas être utilisées par les méthodes `call`, `apply` et `bin`.
- Les fonctions fléchées ne peuvent pas être utilisée comme constructeur.
- Les fonctions fléchées ne peuvent pas utiliser `yield` dans leur corps.

2.3.8 Les propriétés d'instance dans ES2017

Avant la mise en place de ES2017, si vous vouliez ajouter une propriété à une classe, nous devions l'ajouter dans le constructeur comme suit :

```
class MyClass {
  constructor() {
    this.myProperty = 10;
  }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

const myInstance = new MyClass();
console.log(myInstance.myProperty); // 10
```

C'était extrêmement verbeux, et pouvait rendre complexe la lecture du constructeur. Maintenant, nous pouvons déclarer directement les propriétés en dehors du constructeur :

```
class MyClass {
  myProperty = 10;
}

const myInstance = new MyClass();
console.log(myInstance.myProperty); // 10
```

Mais cela entraine quelques implications particuliaire spécialement autour des fonctions. En effet grâce à cette implémentation les méthodes d'une classe peuvent être de la forme d'une fonction fléchée et être considéré comme étant membre de la classe :

```
class MyClass {

  myFunction = () => {
    console.log( this instanceof MyClass); // True
  };
}

const myInstance = new MyClass();
const myFunction = myInstance.myFunction;

myFunction();
```

Et ainsi, même en créant un alias de la méthode dans une variable, elle sera toujours considéré comme étant membre de l'instance de la classe qui l'a initié, comme cela fonctionne dans des langage comme java, C# ou C++.

CHAPITRE 3

Python

Le langage **Python** est un langage facile à apprendre avec des lignes de commande claires et concises. La mise en place de l'environnement de développement est très simple, il n'y a pas de compilateur à mettre en place car le langage est interprété. Python est utilisé pour le développement de nombreux logiciels **Open Source**. Connaitre ce langage est donc un atout.

Le langage **Python** tire son nom d'une émission humoristique britannique : « *Monty Python's flying Circus* ». Le créateur du langage, le mathématicien **Hollandais Guido van Rossum** le nomma « **Python** » en hommage à ce programme TV en 1991.

Cette séquence est découpée en chapitres qui introduisent chacun une notion. Lisez les chapitres dans l'ordre car ils sont liés entre eux. Nous allons revoir les bases de la programmation en **Python**.

Note : Pour aborder ce cours, n'hésitez pas à reproduire les lignes de code dans un éditeur de texte et à interpréter le code pour voir le résultat et comprendre. Prenez le temps de lire chacune des lignes.

3.1 1.0 Introduction

Un Programme est la description d'un algorithme dans un langage compréhensible par un humain mais aussi et surtout par une machine. C'est la machine qui va exécuter le programme afin de traiter les données et les instructions données.

Il existe une multitude de langage de programmation avec chacun leur particularité. Certain ont une syntaxe plus permissive que d'autres. D'autre sont plus proche du langage naturel humain ou à l'inverse plus proche du langage de la machine.

Le langage **Python** est de plus en plus répandu dans l'enseignement supérieur et au lycée dans le cadre de l'enseignement des mathématiques. Ce langage a été créé par *Guido Van Rossum*, un ingénieur informaticien néerlandais en 1991. Il a travaillé pour Google puis Dropbox. Nous allons utiliser la version 3 de Python.

Le langage Python est dit « multiplateforme » car il fonctionne aussi bien sur des ordinateurs sous Windows, Linux, MacOs, Android ou IOS. C'est un langage gratuit et placé sous licence libre.

Les constructions élémentaires en langage Python sont communes à de nombreux autres langages de programmation.

Un programme est composé :
- De séquences (instructions exécutées l'une après l'autre dans l'ordre où elles sont écrites).
- Des définitions de variables et de fonctions.
- D'affectations de valeurs.
- D'instructions conditionnelle.
- De boucles.
- Des appels de fonctions.

Voici ce que l'on peut faire avec du Python :
- Des petits programmes très simples, appelés **scripts**, chargés d'une mission très précise sur votre ordinateur ;
- Des programmes complets, comme des jeux, des suites bureautiques, des logiciels multimédias, des clients de messagerie…
- Des projets très complexes, comme des progiciels (ensemble de plusieurs logiciels pouvant fonctionner ensemble, principalement utilisés dans le monde professionnel).

Python est un langage de programmation **interprété**, c'est-à-dire que les instructions que vous lui envoyez sont « **transcrites** » en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont appelés « langages **compilés** » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « **compilation** ». À chaque modification du code, il faut rappeler une étape de compilation.

Les avantages d'un langage interprété sont la simplicité (on ne passe pas par une étape de compilation avant d'exécuter son programme) et la portabilité (un langage tel que Python est censé fonctionner aussi bien sous Windows que sous Linux ou Mac OS, et on ne devrait avoir à effectuer aucun changement dans le code pour le passer d'un système à l'autre). Cela ne veut pas dire que les langages compilés ne sont pas portables, loin de là ! Mais on doit utiliser des compilateurs différents et, d'un système à l'autre, certaines instructions ne sont pas compatibles, voire se comportent différemment.

En contrepartie, un langage compilé se révélera bien plus rapide qu'un langage interprété (la traduction à la volée de votre programme ralentit l'exécution), bien que cette différence tende à se faire de moins en moins sentir au fil des améliorations. De plus, il faudra installer Python sur le système d'exploitation que vous utilisez pour que l'ordinateur puisse comprendre votre code.

3.2 2.0 Eléments de base

Note : Pour afficher du texte dans la console en **Python**, on utilise la fonction : `print()`.

Pour demander à l'utilisateur de saisir des données, on utilise la fonction : `input()`. La fonction `input` renvoie toujours une chaine de caractères.

3.2.1 2.1 Variables et affectation

Dans un programme, les données utilisées sont stockées dans des variables. Une affection est le fait d'associer une donnée (valeur ou expression) avec un nom. Une variable est comme une boîte sur laquelle il y a une étiquette avec un nom et dans laquelle on y range des informations diverses. Le nom de la variable peut être n'importe quelle chaine alphanumérique (sauf certains mots clés réservés) et ne doit pas commencer par un chiffre. L'opérateur d'affectation est noté =.

Exemple 1 : Cette instruction associe la valeur `3` au nom `x`.

```
X=3
```

Exemple 2 : Cette instruction associe la valeur de l'expression à droite du signe =.

```
Y = 3 + 5  #Y vaut 8
```

Exemple 3 :

Code source 1 – this.py

```
X=3
Y=3+5
Z = X + Y
```

`Z` vaut 11. Car il est la somme des valeurs de `X` et de `Y` au nom `Z`.

En **Python**, nous pouvons faire des affectations multiples pour gagner du temps dans la rédaction et d'économiser des lignes de code. Ainsi, ces 3 lignes d'instructions :

```
X=1
Y=3
Z=5
```

Peuvent tout simplement s'écrire :

```
x,y,z = 1,2,5
```

On pourrait aussi écrire :

```
x=1; y=2; z=5
```

Vocabulaire à connaitre :

Une **variable** est composée d'un nom (ou identificateur) ; d'une adresse en mémoire où est enregistrée une valeur (ou un ensemble de valeurs), d'un type qui définit ses propriétés.

Une **expression** a une valeur qui est le résultat d'une combinaison de variables ou d'objets, de constantes et d'opérateurs.

Une **instruction** est une commande qui doit être exécutée par la machine.

Une **affectation** est une instruction qui commande à la machine de créer une variable en lui précisant son nom et sa valeur.

Attention à ne pas confondre **expression** et **instruction**. Une **expression** se calcul, elle possède une **valeur**. **L'instruction** s'exécute et n'a pas de valeur. L'écriture `x=0.5*x**2+1` est une **instruction** : affecter la valeur de **l'expression** `0.5*x**2+1`.

3.2.2 2.2 Les types simples.

Les types de base qui permettent de définir l'ensemble des valeurs qui peuvent prendre les variables sont :

— Les types numériques `int`.

Ce type représente les nombres entiers. La taille d'une variable `int` n'est limité que par la capacité de la machine et le temps nécessaire à leur utilisation.

— Les types booléens `bool`.

Ce type permet de représenter les valeurs booléennes `True` (vrai) ou `1` en binaire et `False` (faux) ou `0` en binaire.

— Les types flottants `float`.

Ce type est utilisé pour les nombres réels. La virgule est remplacée par le point.

— Les types chaines de caractères `str`.

Les chaines de caractères.

3.2.3 2.3 Opérations sur les types numériques

Tableau 1 – Liste des opérations disponible sur les types numériques.

Addition	+ Exemple : `a + b`
Soustraction	- Exemple : `a - b`
Multiplication	* Exemple : `a * b`
Exponentiation	** Exemple : `a ** b`
Division	/ Exemple : `a / b`
Division entière	// Exemple : `a // b`
Opération modulo	% Exemple : `a % b`

Pour chacune des écritures, il existe des syntaxes permettant d'aller plus vite dans la rédaction comme :

`a = a + b` peut s'écrire `a += b` par exemple.

3.2.4 2.4 Comparaison et opération booléens

Les opérateurs mathématiques de comparaisons s'écrivent ainsi :

Tableau 2 – Liste des opérateurs mathématiques de comparaisons.

Egale	==
Différent	!=
Inférieur	<
Inférieur ou égale	<=
Supérieur	>
Supérieur ou égale	>=

Valeur de retour sur les opérations

- `x==y` prend la valeur `True` si `x` et `y` sont égaux, sinon prend la valeur `False`.
- `x!=y` prend la valeur `True` si `x` et `y` sont différent, sinon prend la valeur `False`.

Valeur de retour sur les opérations logiques

- `a and b` prend la valeur `True` si `a` et `b` sont `True` et sinon prend la valeur `False`.
- `a or b` prend la valeur `False` si `a` et `b` sont `False` et sinon prend la valeur `True`.
- `not a` prend la valeur `True` si `a` est `False` et prend la valeur `False` si `a` est `True`.

3.2.5 2.5 Le type chaine de caractères

`str` est une abréviation de **string** en anglais qui veut dire **chaine de caractères**. Une chaine de caractère est par exemple tout ce que l'on saisie avec les touches du clavier. On utilise des guillemets ou des apostrophes pour les déclarer.

```
MyString = "hello world!"
MyString2 = 'hello world!'
```

Si vous écrivez `Mystring = Hello`, `Hello` sera considéré comme étant une variable. Si elle n'existe pas, il y aura une erreur. Si elle existe, `MyString` aura alors la valeur de la valeur de `Hello`.

Nous pouvons connaitre la longueur d'une chaine, qui est le nombre de caractère qui la compose, grâce à la fonction `len`.

```
len("hello") #a pour valeur un entier : 5
MyString = "Hello world"
len(MyString) #a pour valeur un entier : 11
```

Chaque caractère de la chaine possède un indice qui commence de 0 à (longueur de la chaine -1).

Indice 0	Indice 1	Indice 2	Indice 3	Indice 4	Indice 5	Indice 6	Indice 7	Indice 8	Indice 9
H	E	L	L	O		Y	O	U	!

Nous avons ici la chaine « HELLO YOU ! ». Si nous faisons un : `len("HELLO YOU !")` nous obtenons la longueur 10. Toutefois nous voyons dans le tableau que le dernier caractère se situe à l'indice 9(10 – 1).

Il est possible d'accéder à l'indice i d'une chaine grâce à cette syntaxe :

```
MyString = "Hello world"
MyString[2] #Nous accédons à l'indice 2 de la chaine. Soit au caractère l
```

Nous pouvons avoir accès également à une suite de caractères d'une chaine avec la notation : `MyString [i : j]`. L'indice `i` est inclus et `j` est exclu.

```
MyString = "Hello world"
MyString [2 :4] #résultat:  ll
```

3.2.6 2.6 Les types composés

Il s'agit des types `tuple`, `list`, `dict`. Nous étudierons ici les `list`. Un objet de type `list` (« une liste »), représente un ensemble ordonné d'objets éventuellement de types différents. De la même manière qu'avec les chaines de caractères, les éléments de la liste sont ordonnés en commençant à l'indice 0. Comment déclarer une liste :

```
MyList1= [] #une liste vide.
MyList2=[4] #une liste avec un seul élément, ici l'entier 4.
MyList3=[5, 'hello', 3.14, ['a', 'b']] #une liste avec des éléments de différents types.
```

La fonction `len` est aussi utilisable avec les listes et renvoie sa longueur. `len(MyList3)` a pour valeur l'entier 4.

L'accès à un élément ou une suite d'éléments se fait comme pour les chaînes :

```
MyList3[1] #est l'élément 'hello'
MyList3[0:3] #est la liste des éléments d'indices 0,1,2 soit [5,'hello',3.14]
```

Il est possible de modifier le contenu d'une liste de la sorte :

```
MyList3[1] = "AU REVOIR"
MyList3 #vaut alors [5, 'AU REVOIR', 3.14, ['a', 'b']]
```

La méthode `append` permet d'ajouter des éléments en fin de liste.

```
MyList.append("ELEMENT1")
MyList.append("ELEMENT2")
#MyList vaut: ["ELEMENT1", "ELEMENT2"]
```

3.2.7 2.7 Opération sur les types `str` et `list`

— Il est possible d'indexer à partir du dernier élément :

Exemple : Nous avons une liste de : 7 éléments, rangés de l'index 0 à 6.

```
MyList["item1","item2","item3","item4","item5","item6","item7"]
#Je peux directement aller au dernier élément :
MyList[-1]
#Et on en déduit comment accès à l'avant dernier :
MyList[-2]
```

— La concaténation.

```
Str1 = "BON"
Str2="JOUR"
```

Je peux créer une nouvelle chaine qui sera l'assemblage de `Str1` et `Str2`, on appelle cela faire une concaténation.

```
Str3 = Str1 + Str2
```

`Str3` vaut alors « BONJOUR »

— Nous pouvons aussi effectuer la concaténation de n copies :

```
strConcat = 3*Str3
```

`strConcat` vaut :`bonjourbonjourbonjour`

— Nous pouvons changer le type des variables grâce aux fonctions : `int`, `float`, `str`.

```
StrPI = "3.1415"
```

`StrPI` est de type `string`.Pour la convertir en `float` il suffira d'écrire ceci :

```
float(StrPI)
```

De ce fait :

```
NombrePI = float(StrPI)
```

`NombrePI` sera de type `float`.

— La fonction `list` permet de convertir une chaine de caractère en une liste dont les éléments sont les différents caractères de la chaine.

```
String = "3.1415"
MyList = list(String)
```

`MyList` vaut donc `['3', '.', '1', '4', '1', '5']`

3.3 3.0 Instructions conditionnelles et boucles

L'indentation est le décalage vers la droite du début de ligne. C'est un élément très important de la syntaxe en Python mais aussi dans tous les autres langages.

Cela permet de délimiter visuellement des blocs de code et aide à la lisibilité. En **Python**, quand vous créez un bloc, la ligne précédente l'indentation se termine par le signe `:`.

3.3.1 3.1 Instructions conditionnelles

```
if condition:
  Instructions
```

Condition désigne une expression et instructions désigne une instruction ou un bloc d'instructions écrites sur plusieurs lignes.

Exemple :

```
if n == 4:
  n = 4 * 2
```

Nous pouvons aussi ajouter des instructions si la condition n'est pas respectée avec le mot clé `else`.

```
if n == 4:
  n = 4 * 2
else:
    n = 4 + 1
```

Nous pouvons aussi ajouter une série de conditions dans le cas où les précédentes ne seraient pas respectées avec le mot clé `elif`.

```
if degree <= 0:
  message = "Il géle !!"
elif degree > 0 and degree < 20:
    message = "Il fait froid !!"
else:
    message ="Il faut chaud!!"
```

N'oubliez pas que c'est l'indentation qui permet de délimiter les blocs d'instructions à exécuter si la condition est vérifiée.

3.3.2 3.2 Boucles conditionnelles

Structure :

```
while condition:
  instructions
```

Tant que la condition est respectée, alors les instructions seront exécutées.

```
a = 10
while a > 0:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print("a est supérieur à zéro")
    a = a - 1
```

Si j'exécute le script, j'obtiens à l'écran :

```
a est supérieur à zéro
a est supérieur à zéro
a est supérieur à zéro
a est supérieur à zéro
a est supérieur à zéro
a est supérieur à zéro
a est supérieur à zéro
a est supérieur à zéro
a est supérieur à zéro
a est supérieur à zéro
```

Tant que `a > 10` est `True`, alors on affiche « a est supérieur à zéro » puis on soustrait 1 à la variable `a`. Au bout de 10 « tours » dans la boucle, `a` se retrouve égale à zéro, la condition `a > 0` n'est plus vrai. Les instructions cessent alors d'être exécutées.

3.3.3 3.3 Boucles non conditionnelles

Structure :

```
for i in range(n):
  instructions
```

Cette boucle permet de répéter `n` fois une instruction ou un bloc d'instructions.

```
for car in "bonjour":
  print(10*car)
```

Execution :

```
bbbbbbbbbb
oooooooooo
nnnnnnnnnn
jjjjjjjjjj
oooooooooo
uuuuuuuuuu
rrrrrrrrrr
```

Explication :

Pour chaque caractère un à un de la chaine « bonjour », on affiche 10 fois le caractère.

```
for i in range(10):
  print("hello")
```

Exécution : Pour chaque valeur de `i` en partant de 1, puis en augmentant de 1 jusqu'à 10, **afficher** : hello.

```
hello
hello
```

(suite sur la page suivante)

(suite de la page précédente)

```
hello
hello
hello
hello
hello
hello
hello
hello
```

```
for i in range(6,10):
  print("hello")
```

Exécution : Pour chaque valeur de `i` en partant de 6, puis en augmentant de 1 jusqu'à 10, **afficher** : hello.

```
hello
hello
hello
hello
```

```
for i in range(6,10,2):
  print("hello")
```

Exécution : Pour chaque valeur de `i` en partant de 6, puis en augmentant de 2 jusqu'à 10, **afficher** : hello.

```
hello
hello
```

Ajouter l'instruction `break` dans la boucle permet de sortir de celle-ci. L'instruction `continue` permet d'éviter un passage dans la boucle.

3.3.4 3.4 Mise en pratique : Modifier une liste

```
MyList = ['bijou','caillou','chou','genou','hibou','joujou','pou']
for i in range(len(MyList)):
    print(MyList[i])
```

Exécution

```
bijou
caillou
chou
genou
hibou
joujou
pou
```

Nous souhaitons ajouter un "x" à la fin de chaque mot, et les afficher de nouveau.

```
MyList = ['bijou','caillou','chou','genou','hibou','joujou','pou']
for i in range(len(MyList)):
    MyList[i] = MyList[i] + "x"
    print(MyList[i])
```

Exécution

```
bijoux
cailloux
choux
genoux
hiboux
joujoux
poux
```

3.4 3.0 Les Fonctions

3.4.1 3.1 Fonctions

Voici la syntaxe pour définir une fonction en Python :

Code source 2 – Définition d'une fonction

```
def nom_de_la_fonction(arguments):
    """ aide sur la fonction, facultatif"""
    corps de la fonction
```

Le mot clé `def` permet de déclarer une fonction. La déclaration se termine par `:` et le corps de la fonction est indenté. Il peut ne pas y avoir d'arguments. Par contre lorsqu'il y a plusieurs arguments, ils sont séparés par des virgules. La fonction peut renvoyer un résultat grâce au mot clé `return` suivi du résultat. Il peut y avoir plusieurs `return`.

Chaque `return` interrompt l'exécution de la fonction. Sans valeur de retour, une fonction renvoie `None`. On les appelle des **procédures**.

```
def premier(n):
    for d in range(2,n):
        if n % d == 0:
            return False
    return True
```

Explications

Durant l'appel de `premier(7)`, la boucle `for` est exécutée. Mais puisque les tests `n % d == 0` ont tous la valeur `False`, l'instruction `return False` n'est jamais exécutée mais `return True`.

3.2 Espace et portée des variables

Il existe différents espaces dans un programme :

- **L'espace global** : l'espace dans lequel les fonctions et les variables sont définis. Et pour chaque fonction, un espace local distinct est créé.
- **L'espace local** : l'espace réservé au corps d'une fonction.

Les variables déclarées dans l'espace local n'existent que dans l'espace local.

3.2.1 Portée d'une variable

Une fonction ne peut pas modifier par affectation la valeur d'une variable extérieur à son espace local puisqu'une affectation crée une nouvelle variable locale qui est détruite après l'utilisation de la fonction. Dans l'exemple suivant : la variable locale `x` utilisée dans la fonction est distincte de la variable globale `x` définie au début du programme par l'instruction `x=1` et n'existe plus après l'appel de la fonction.

```
x=1
def f(x):
        x = x + 1
        return x
#Affiche 2
print(f(x))
#Affiche 1
print(x)
```

Toutefois une fonction peut modifier une variable extérieure à celle-ci en la déclarant comme variable globale avec le mot-clé `global`. A éviter malgré tout.

```
x=1
def f():
        global x
        x = x + 1
        return x
#Affiche 2
print(f(x))
#Affiche 2, x a été modifié
print(x)
```

3.5 4.0 Spécification et tests

3.5.1 4.1 Spécification d'une fonction

Une spécification permet d'informer les utilisations de la tâche effectuée par la fonction, de préciser les contraintes imposées pour les paramètres et ce qui peut être attendu des résultats. Elle peut aussi préciser les messages d'erreurs en cas de mauvaise utilisation. Elle est résumée dans la `docstring`, inscrite au début du corps de la fonction entre des triples guillemets. Exemple, pour la fonction `print`, si nous souhaitons avoir des informations alors nous utiliserons la fonction `help` dans l'interpréteur.

```
>>>help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

La fonction `help` affiche la docstring inscrite dans le code de la fonction `print`. On apprend par exemple que la fonction `print` peut recevoir 4 arguments optionnels. La spécification d'une fonction est écrite, comme l'est un commentaire dans un programme, pour les utilisateurs. L'objectif est donc d'être clair, et d'aider à saisir rapidement le rôle d'une ou plusieurs instruction. Il faut donc être vigilent aux choix des noms de variables et de fonctions.

Voici comment spécifier la documentation d'une fonction :

```
def ExempleDeFonction(un_argument):
   """ Ceci est un test de documentation qui apparaitra dans le Help ! """
   print("bonjour !")
```

Le texte entre les triples guillemets fournit ici une description de l'entrée, du traitement et du résultat. La fonction est enregistrée dans un fichier et le texte s'affiche lorsqu'on écrit `help``(``ExempleDeFonction`). Les commentaires précédés par le signe # n'ont aucune importance sur l'exécution de la fonction. Ils ne sont pas lus par l'interpréteur **Python**. Testons :

```
help(ExempleDeFonction)
Help on function ExempleDeFonction in module __main__:
ExempleDeFonction(un_argument)
Ceci est un test de documentation qui apparaitra dans le Help !
```

La spécification est une sorte de contrat entre l'auteur et l'utilisateur. L'auteur garantit un résultat sous réserve d'une utilisation correcte qui est précisée.

3.5.2 4.2 Tests

Les fonctions doivent être testées avant de pouvoir être utilisées dans différents programmes. Il faut envisager tout les cas de figures. Exemple avec une fonction `permute`.

Code source 3 – Fonction « permute »

```
 def permute(liste):
      copie = liste[:]
      copie[0], copie[-1] = copie[-1], copie[0]
      return copie
```

Testons :

```
print(permute([1,2,3,4]))
>> [4, 2, 3, 1]

print(permute([ [1,2] , [3,4], [5,6] ])
>> [[5,6], [3,4], 1,2]]


Print(permute([1])
>> [1]

Print(permute([])
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print(permute([])
  File "C:/Users/Baptiste/Desktoptest.py", line 3, in permute
```

(suite sur la page suivante)

(suite de la page précédente)

```
    copie[0], copie[-1] = copie[-1], copie[0]
IndexError: list index out of range
```

Nous remarquons que la fonction effectue bien ce qui est prévu, même pour une liste ne contenant qu'un seul élément. Par contre, le cas d'une liste vide n'a pas été traité et une erreur interrompt le programme. Il faut donc modifier la définition de la fonction pour éviter qu'une erreur soit générée si l'utilisateur l'utilise avec une liste vide.

Code source 4 – Fonction « permute »

```
def permute(liste):
      if liste == []:
              return []
      copie = liste[:]
      copie[0], copie[-1] = copie[-1], copie[0]
      return copie
```

3.5.3 Assertion

Une autre manière d'anticiper une mauvaise utilisation est d'incorporer des assertions. Une assertion permet de gérer une possible erreur d'utilisation prévue à l'avance. Dans notre fonction, il suffit d'ajouter au début l'instruction : `assert liste !=[]`

Code source 5 – Fonction « permute » avec assertion.

```
  def permute(liste):
      assert liste !=[]
      copie = liste[:]
      copie[0], copie[-1] = copie[-1], copie[0]
      return copie
```

Lors de l'appel de `permute([])`, l'exécution est interrompue et un message d'erreur s'affiche en précisant que l'assertion n'a pas été vérifiée.

On utilise les assertions en phase de test. Lorsque les tests sont finis on peut les supprimer et les remplacer par des blocs `Try ...except` qui permettent au programme de réagir de manière précise. Pour tester un programme, on peut également écrire une fonction test contenant une batterie de tests.

Exemple avec une fonction de division.

Code source 6 – Fonction division

```
  def division(a,b):
      """a est un entier naturel
         b est un entier naturel non nul """
         r = a
         q = 0
         while r >= b:
                 r = r - b
                 q = q + 1
      return q,r
```

Nous allons maintenant écrire une fonction `test_division`. Cette fonction permet de vérifier que le résultat renvoyé par la fonction division est correct dans une série de cas. Cela évite d'effectuer les tests à la main, un par un .

Code source 7 – Test de la fonction division

```
def test_division():
        """ la fonction division doit renvoyer le quotient q et le reste r dans la division de a par b
        un invariant est a == b * q + r """
        for a in range(13):
                for b in range (1,13):
                        q, r = division( a, b )
                        if a != b * q + r or r >= b:
                                return False
                return True
```

Testez la fonction.

Vous constaterez qu'il n'y a aucun problème de signalé. Mais modifions la fonction division. Remplacer la ligne `while r=>b` par `while r>b`.

Code source 8 – Testons la fonction division modifiée

```
def division(a,b):
      """a est un entier naturel
         b est un entier naturel non nul """
         r = a
         q = 0
         while r > b:
                 r = r - b
                 q = q + 1
      return q,r
```

Appelons maintenant la fonction `test_division()`.

```
>>> test_division()
False
```

La fonction `test_division` retourne `False`. Nous pouvons modifier la fonction `test_division` pour connaitre les raisons de l'échec du test.

```
def test_division():
      """ la fonction division doit renvoyer le quotient q et le reste r dans la division de a par b
      un invariant est a == b * q + r """
      for a in range(13):
              for b in range (1,13):
                      q, r = division( a, b )
                      if a != b * q + r or r >= b:
                              message1 = "Echec pour a = " + str(a) + " et b = " + str(b)
                              message2 = "q = " + str(q) + " et r = " + str(r)
                              return False, message1, message2
              return True
```

Exécutons la fonction de test.

```
>>> test_division()
(False, 'Echec pour a = 1 et b = 1', 'q=0 et r=1' )
```

Nous pouvons maintenant comprendre l'origine du problème à ce test. L'objectif des tests ne doit pas être de prouver qu'il n'y a pas d'erreurs mais de les débusquer !

3.6 5.0 Modules et bibliothèques

Jusqu'ici, nous avons travaillé avec les fonctions de Python chargées au lancement de l'interpréteur. Toutefois il existe une fonctionnalité nommée : **les modules**. Un module est grossièrement un bout de code que l'on a enfermé dans un fichier. On emprisonne ainsi des fonctions et des variables ayant toutes un rapport entre elles. Ainsi, si l'on veut travailler avec les fonctionnalités prévues par le module il n'y a qu'à importer le module et utiliser ensuite toutes les fonctions et variables prévues. Il existe un grand nombre de modules disponibles avec **Python** sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires. Pour cette partie, nous prendrons l'exemple du module `math` qui contient, comme son nom l'indique, des fonctions mathématiques.

3.6.1 5.1 La méthode `import`

Lorsque vous ouvrez l'interpréteur Python, les fonctionnalités du module `math` ne sont pas incluses. Il s'agit en effet d'un module, il vous appartient de l'importer au besoin.

```
>>> import math
```

La syntaxe est facile à retenir : le mot-clé `import`, qui signifie « importer » en anglais, suivi du nom du module, ici `math`. Toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles. Pour appeler une fonction du module, il faut taper le nom du module suivi d'un point « . » puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

```
>>> math.sqrt(16)
4.0
```

La fonction `help()` pourra vous aider à trouver toutes les fonctions disponibles dans ce module.

>>>help(math)

3.6.2 5.2 Les espaces de nom

Avec un espace de nom,il s'agit de regrouper certaines fonctions et variables sous un préfixe spécifique. En vérité, quand vous tapez `import math`, cela crée un espace de noms dénommé « math », contenant les variables et fonctions du module `math`. Quand vous tapez `math.sqrt(25)`, vous précisez à **Python** que vous souhaitez exécuter la fonction `sqrt` contenue dans l'espace de noms `math`. Cela signifie que vous pouvez avoir, dans l'espace de noms principal, une autre fonction sqrt que vous avez définie vous-mêmes. Il n'y aura pas de conflit entre, d'une part, la fonction que vous avez créée et que vous appellerez grâce à l'instruction `sqrt` et, d'autre part, la fonction `sqrt` du module `math` que vous appellerez grâce à l'instruction `math.sqrt`.

Exemple :

```
import math
a = 5
b = 33.2
```

Dans l'espace de noms principal, celui qui ne nécessite pas de préfixe et que vous utilisez depuis le début du cours , on trouve :

- La variable `a`.
- La variable `b`.
- Le module `math`, qui se trouve dans un espace de noms s'appelant `math` également. Dans cet espace de noms, on trouve :
  - la fonction `sqrt` ;
  - la variable `pi` ;
  - et bien d'autres fonctions et variables…

C'est l'intérêt des modules : des variables et fonctions stockées à part dans un espace de noms, sans risque de conflit avec nos propres variables et fonctions. Mais dans certains cas, on pourra vouloir changer le nom de l'espace de noms dans lequel sera stocké le module importé.

```
import math as mathematiques
mathematiques.sqrt(25)
```

3.6.3 5.3 Une autre méthode d'importation : `from .. import`

Nous avons précédemment qu'il était possible d'importer tout un module, comme celui de `Math`. Cela veut dire que nous importons dans le programme l'ensemble des fonctions présentes dans ce module. Maintenant admettons que nous ayons besoin dans notre programme que d'une seule fonction de ce module, par exemple la fonction renvoyant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module. Cela se fait grâce à la syntaxe suivante :

```
from math import fabs
```

Et pour appeler la fonction, nous l'appelons uniquement par son nom sans utiliser le préfixe `math`. Une autre méthode pour importer toutes les variables et fonctions du module `Math` :

```
from math import *
```

En faisant ainsi vous n'aurez pas besoin de saisir le préfix `math`. devant le nom des fonctions pour les utiliser. Mais il est préférable de ne charger que les fonctions qui nous seront utiles, c'est une question de performance et de rapidité du code.

Pour résumer

- Une fonction est une portion de code contenant des instructions, que l'on va pouvoir réutiliser facilement.
- Découper son programme en fonctions permet une meilleure organisation.
- Les fonctions peuvent recevoir des informations en entrée et renvoyer une information grâce au mot-clé `return`.
- Les fonctions se définissent de la façon suivante : `def nom_fonction(parametre1, parametre2, parametreN):`

3.7 6.0 Initiation à la programmation Orientée Objet

3.7.1 6.1 Introduction

La POO est un **paradigme** de programmation, au même titre que la programmation impérative (que nous pratiquons déjà) ou la programmation fonctionnelle, ou encore d'autres paradigmes (la liste est longue). Un paradigme de programmation pourrait se définir comme une philosophie dans la manière de programmer : c'est un parti-pris revendiqué dans la manière d'aborder le problème à résoudre. Une fois cette décision prise, des outils spécifiques au paradigme choisi sont utilisés.

Métaphore : Imaginons 3 menuisiers qui ont pour mission de fabriquer chacun un meuble.

- Le premier pourra décider d'utiliser du collé-pointé : il assemblera les morceaux de bois en les collant puis utilisera des pointes. Ses outils seront le marteau et le pistolet à colle.
- Le deuxième pourra décider de visser les morceaux de bois entre eux : son outil principal sera une visseuse.
- Le troisième pourra décider de faire de l'assemblage par tenons et mortaises : son outil principal sera une défonceuse.

Pour la réalisation de sa mission, chaque menuisier utilise un paradigme différent. Qui utilise la meilleure méthode ? Cette question n'a pas vraiment de réponse : certaines méthodes sont plus rapides que d'autres, d'autres plus robustes, d'autres plus esthétiques…

Et pourquoi ne pas mélanger les paradigmes ? Rien n'interdit d'utiliser des pointes ET des vis dans la fabrication d'un meuble.

La Programmation Orientée Objet sera (surtout à notre niveau) mélangée avec de la programmation impérative, de la programmation fonctionnelle… d'ailleurs vous avez déjà manipulé des objets sans le savoir :

Des objets déjà autour de nous

Essayez le code suivant :

```
m = [4,5,2]
type(m)

Cela retourne : list
```

`m` est une liste, ou plus précisément un **objet** de type `list`. Et en tant qu'objet de type `list`, il est possible de lui appliquer certaines fonctions prédéfinies (qu'on appellera méthodes) : `m.reverse()`

La syntaxe utilisée (le `.` après le nom de l'objet) est spécifique à la POO. Chaque fois que vous voyez cela, c'est que vous êtes en train de manipuler des objets. Mais qu'a donc fait cette méthode `reverse()` ?

```
m
Cela retourne : [2, 5, 4]
```

Nous ne sommes pas surpris par ce résultat car la personne qui a programmé la méthode `reverse()` lui a donné un nom explicite. Comment a-t-elle programmé cette inversion des valeurs de la liste ? Nous n'en savons rien et cela ne nous intéresse pas. Nous sommes juste utilisateur de cette méthode.

L'objet de type `list` nous a été livré avec sa méthode `reverse()` (et bien d'autres choses) et nous n'avons pas à démonter la boîte pour en observer les engrenages : on parle de principe d'"**encapsulation**.

On peut obtenir la liste de toutes les fonctions disponibles pour un objet de type `list`, par la fonction `dir` :

```
dir(m)

Nous obtenons :
['__add__',
'__class__',
'__contains__',
'__delattr__',
'__delitem__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__getitem__',
'__gt__',
'__hash__',
'__iadd__',
'__imul__',
'__init__',
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__reversed__',
'__rmul__',
'__setattr__',
'__setitem__',
'__sizeof__',
'__str__',
```

(suite sur la page suivante)

(suite de la page précédente)

```
'__subclasshook__',
'append',
'clear',
'copy',
'count',
'extend',
'index',
'insert',
'pop',
'remove',
'reverse',
'sort']
```

Les méthodes encadrées par un double **underscore** __ sont des méthodes privées, a priori non destinées à l'utilisateur. Les méthodes publiques, utilisables pour chaque objet de type `list`, sont donc `append`, `clear`, … Comment savoir ce que font les méthodes ? Si elles ont été correctement codées (et elles l'ont été), elles possèdent une `docstring`, accessible par :

```
>>> m.append.__doc__
Retourne :'Append object to the end of the list.'
```

```
>>> m.reverse.__doc__
Retourne : 'Reverse *IN PLACE*.'
```

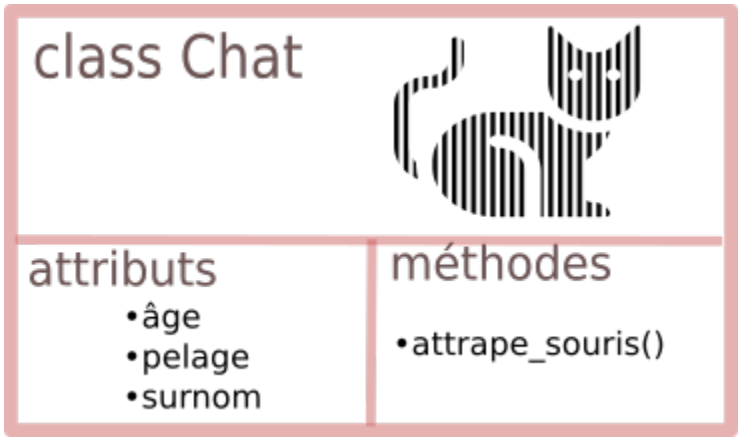
3.7.2 6.2 Créer son propre objet, sa propre classe

Vocabulaire : classe, objet, instance de classe

Jusqu'ici nous avons employé uniquement le mot «objet». Il convient maintenant d'être plus précis. On désignera par **classe** la structure de données définissant une catégorie générique d'objets. Dans le monde animal, `chat` est une **classe** (nommée en réalité `félidé`).

Chaque élément de la classe chat va se distinguer par des caractéristiques : * Un âge * Une couleur de pelage * Un surnom… (on appellera ces caractéristiques des **attributs**) * Des fonctionnalités, comme la **méthode** `attrape_souris()`.

Lorsqu'on désigne un chat en particulier, on désigne alors un **objet** (bien réel) qui est une **instance** de la **classe** (abstraite) chat. Par exemple, l'"**objet** `Larry` est une **instance** de la **classe** `chat` .



D'après Wikipedia, Exemple :

```
larry.pelage = "blanc et tabby"
larry.surnom = "Chief Mouser to the Cabinet Office"
```

Toujours d'après Wikipedia, la méthode `larry.attrape_souris()` est plutôt efficace.

Création d'une classe(Mauvaise), manière minimale

Créons une classe `voiture`. Il suffit d'écrire :

```
class Voiture :
  pass   #pass, car pour l'instant il n'y a rien dans la déclaration de la classe (et c
↪'est mal)
```

La classe `Voiture` est créée. Notez que par convention, le nom d'une classe commence toujours par une majuscule. Pour créer une instance de cette classe, on écrit : `titine = Voiture()`

`titine` est un objet, instance de la **classe** `Voiture`. Si l'on regarde le type de l'objet :

`type(titine)`

retour : `__main__.Voiture`

On peut alors donner des attributs à cette instance :

```
titine.annee = 2018
titine.couleur = "verte"
titine.vitesse_max = 162
```

Mais arrêtons-là cette mauvaise méthode. Si on désire créer une classe `voiture`, c'est pour créer un concept générique de `voiture` et d'en spécifier des caractéristiques communes : l'année, la couleur, la vitesse maximale… L'idée est donc qu'à la création (on dira plutôt à la **construction**) de chaque objet `voiture`, on va lui spécifier directement ses attributs :

(Bonne) manière : la méthode constructeur

La méthode constructeur, toujours appelée `__init__()`, est une méthode (une `def`) qui sera automatiquement appelée à la création de l'objet. Elle va donc le doter de tous les attributs de sa classe.

```
class Voiture :
  def __init__(self, annee, coul, vmax) :
      self.annee = annee
      self.couleur = coul
      self.vitesse_max = vmax
      self.age = 2020 - self.annee
```

- Le mot-clé `self`, omniprésent en POO (d'autres langages utilisent `this`), fait référence à l'objet lui-même, qui est en train d'être construit.
- Pour construire l'objet, 3 paramètres seront nécessaires : `annee`, `coul` et `vmax`. Ils donneront respectivement leur valeur aux attributs `annee`, `couleur` et `vitesse_max`.
- Dans cet exemple, les noms `coul` et `vmax` ont été utilisés pour abréger `couleur` et `vitesse_max`, mais il est recommandé de garder les mêmes noms, même si ce n'est pas du tout obligatoire.

Construisons donc notre première voiture !

```
mon_bolide = Voiture(2012, "rouge", 190)
```

`mon_bolide` possède 4 attributs :

— `annee`, `couleur` et `vitesse_max` ont été donnés par l'utilisateur lors de la création.
— `age` s'est créé «tout seul» par l'instruction `self.age = 2020 - self.annee`.

```
print(mon_bolide.annee)
print(mon_bolide.couleur)
print(mon_bolide.vitesse_max)
print(mon_bolide.age)
```

Retour :

```
2012
rouge
190
8
```

Bien sûr, on peut créer une autre voiture en suivant le même principe :

batmobile = Voiture(2036, « noire », 325)

3.7.3 6.3 TD : Jeu de cartes

La fonctionnalité la plus emblématique de la programmation orientée objet est l'héritage. L'héritage est la possibilité de définir une nouvelle classe, qui est une version modifiée d'une classe existante.

Objets carte de jeu

Il y a cinquante-deux cartes dans un paquet, dont chacune appartient à une des quatre couleurs (ou enseignes) et à l'une des treize valeurs (ou rangs). Les couleurs sont pique, cœur, carreau, et trèfle (dans l'ordre décroissant au jeu de bridge). Les valeurs sont as, 2, 3, 4, 5, 6, 7, 8, 9, 10, valet, dame (ou reine) et roi. Selon le jeu auquel vous jouez, un as peut être plus fort que le roi ou plus faible que le 2.

Si nous voulons définir un nouvel objet pour représenter une carte à jouer, il est évident que les attributs doivent être la `couleur` et la `valeur`. Le type des attributs n'est pas si évident. Une possibilité est d'utiliser des chaînes contenant des mots comme "**pique**" pour les couleurs et "**dame**" pour les valeurs. Un problème avec cette modélisation est qu'il ne serait pas facile de comparer les cartes pour voir laquelle a une valeur ou une couleur supérieure.

Une autre possibilité est d'utiliser des entiers pour **encoder** les valeurs et les couleurs. Dans ce contexte, « **encoder** » signifie que nous allons définir une correspondance entre nombres et couleurs, ou entre nombres et valeurs. Ce type d'encodage n'est pas censé être secret (ce serait du « cryptage » ou du chiffrement).

Par exemple, ce tableau montre les couleurs et les valeurs entières correspondantes :

Tableau 3 – title

Pique ->	3
Cœur ->	2
Carreau ->	1
Trèfle ->	0

Ce code facilite la comparaison des cartes ; parce que les couleurs les plus élevées correspondent aux nombres plus élevés, nous pouvons comparer les couleurs en comparant leurs codes. Le codage des valeurs est assez évident ; chacune des valeurs numériques des cartes correspond à l'entier correspondant, et pour les honneurs :

Tableau 4 – title

Valet ->	11
Dame ->	12
Roi ->	13

J'utilise le symbole pour qu'il soit clair que ces correspondances ne font pas partie du programme **Python**. Elles font partie de la conception du programme, mais elles n'apparaissent pas explicitement dans le code. La définition de la classe `Carte` ressemble à ceci :

```
class Carte:
  """Représente une carte à jouer standard."""

  def __init__(self, couleur = 0, valeur = 2):
      self.couleur = couleur
      self.valeur = valeur
```

Comme d'habitude, la méthode `init` prend un paramètre optionnel pour chaque attribut. La carte par défaut est le 2 de trèfle. Pour créer une carte, vous appelez `Carte` avec la couleur et la valeur de la carte souhaitée.

```
dame_de_carreau = Carte(1, 12)
```

Attributs de classe

Pour afficher des objets de type `Carte` d'une manière lisible facilement pour les humains, nous avons besoin d'une correspondance entre les codes nombres entiers et les couleurs et les valeurs correspondantes. Une façon naturelle de le faire est d'utiliser des listes de chaînes de caractères. Nous attribuons ces listes aux **attributs de classe** :

```
# à l'intérieur de la classe Carte :

  noms_couleurs = ['trèfle', 'carreau', 'cœur', 'pique']
  noms_valeurs = [None, 'as', '2', '3', '4', '5', '6', '7',
            '8', '9', '10', 'valet', 'dame', 'roi']

  def __str__(self):
      return '%s de %s' % (Carte.noms_valeurs[self.valeur],
                           Carte.noms_couleurs[self.couleur])
```

Les variables comme `noms_couleurs` et `noms_valeurs`, qui sont définies dans une classe, mais en dehors de toute méthode, s'appellent attributs de classe parce qu'elles sont associées à l'objet classe `Carte`.

Ce terme les distingue des variables telles que couleur et valeur, qui s'appellent **attributs** d'instance parce qu'elles sont associées à une instance particulière.

Les deux types d'attributs sont accessibles en utilisant la notation pointée. Par exemple, à l'intérieur de `__str__`, `self` est un objet `Carte` et `self.couleur` est sa couleur. De même, `Carte` est un objet classe, et `Carte.noms_valeurs` est une liste de chaînes de caractères associée à la classe.

Chaque carte a sa propre `couleur` et sa propre `valeur`, mais il n'y a qu'une seule copie de `noms_couleurs` et `noms_valeurs`.

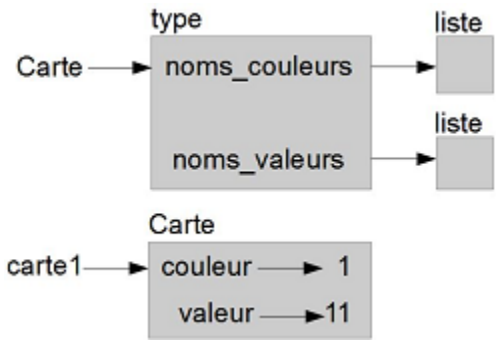
En mettant le tout ensemble, l'expression `Carte.noms_valeurs[self.valeur]` signifie « utilise l'attribut `valeur` de l'objet `self` comme un index de la liste `noms_valeurs` de la classe `Carte`, et sélectionne la chaîne de caractères appropriée. »

Le premier élément de `noms_valeurs` est `None`, car il n'y existe aucune carte de rang zéro. En incluant `None` comme un espace réservé, nous obtenons une correspondance ayant comme belle propriété le fait que l'indice 2 corresponde

à la chaîne de caractères 2, et ainsi de suite. Pour éviter de devoir faire cet ajustement, nous aurions pu utiliser un dictionnaire à la place d'une liste.

Avec les méthodes que nous avons jusqu'ici, nous pouvons créer et afficher des cartes :

```
carte1 = Carte(2, 11)
print(carte1)
>>>valet de cœur
```



La figure est un diagramme de l'objet classe `Carte` et d'une instance de `Carte`. `Carte` est un objet classe ; son type est `type`. L'objet `carte1` est une instance de `Carte`, donc son type est `Carte`. Pour économiser l'espace, je n'ai pas dessiné le contenu de `noms_couleurs` et `noms_valeurs`.

Comparer des cartes

Pour les types internes, il existe des opérateurs relationnels (<, >, == , etc.) qui comparent des valeurs et déterminent si l'un est supérieur, inférieur ou égal à un autre. Pour les types définis par le programmeur, nous pouvons remplacer le comportement des opérateurs internes en fournissant une méthode nommée `__lt__`, qui signifie **less than**, « inférieur à ». `__lt__` prend deux paramètres, `self` et `other`, et renvoie `True` si `self` est strictement inférieur à `other`.

L'ordre correct des cartes n'est pas évident. Par exemple, qu'est-ce qui est mieux, le 3 de trèfle ou le 2 de carreau ? L'une a une valeur plus élevée, mais l'autre a une couleur plus élevée. Afin de comparer les cartes, vous devez décider si la valeur ou la couleur est plus importante.

La réponse pourrait dépendre du jeu auquel vous jouez, mais pour ne pas compliquer les choses, nous faisons le choix arbitraire que c'est la couleur qui primera, donc tous les piques surclassent tous les carreaux, et ainsi de suite.

Une fois cette décision prise, nous pouvons écrire `__lt__` :

```
# à l'intérieur de la classe Carte :

def __lt__(self, other):
    # vérifier les couleurs
    if self.couleur < other.couleur: return True
    if self.couleur > other.couleur: return False

    # les couleurs sont identiques... vérifier les valeurs
    return self.valeur < other.valeur
```

Vous pouvez réécrire cela d'une façon plus concise, en utilisant la comparaison de tuple :

```
# à l'intérieur de la classe Carte :

def __lt__(self, other):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    t1 = self.couleur, self.valeur
    t2 = other.couleur, other.valeur
    return t1 < t2
```

À titre d'exercice, écrivez une méthode `__lt__` pour des objets de type `Temps`. Vous pouvez utiliser la comparaison de tuple, mais vous pourriez aussi envisager la comparaison des entiers.

Paquets de cartes

Maintenant que nous avons les cartes, la prochaine étape est de définir les Paquets de cartes. Comme un paquet est composé de cartes, il est naturel que chaque Paquet contienne comme attribut une liste de cartes. Ce qui suit est une définition de classe pour Paquet. La méthode `init` crée l'attribut `cartes` et génère l'ensemble standard de cinquante-deux cartes :

```
class Paquet:
  def __init__(self):
    self.cartes = []
    for couleur in range(4):
      for valeur in range(1, 14):
        carte = Card(couleur, valeur)
        self.cartes.append(carte)
```

La meilleure façon de constituer le paquet est avec une boucle imbriquée. La boucle externe énumère les couleurs de 0 à 3. La boucle interne énumère les valeurs de 1 à 13. Chaque itération crée une nouvelle carte ayant la couleur et la valeur courantes, et l'ajoute à `self.cartes`.

Afficher le paquet

Voici une méthode `__str__` pour Paquet :

```
# à l'intérieur de la classe Paquet :

  def __str__(self):
    res = []
    for carte in self.cartes:
      res.append(str(carte))
      return '\n'.join(res)
```

Cette méthode montre un moyen efficace d'accumuler une longue chaîne de caractères : en construisant une liste de chaînes de caractères, puis en utilisant la méthode de chaîne de caractères `join`. La fonction interne `str` invoque la méthode `__str__` sur chaque carte et renvoie sa représentation sous forme de chaîne de caractères. Comme nous invoquons `join` sur un caractère de fin de ligne, les cartes sont séparées par des caractères de fin de ligne. Voici à quoi ressemble le résultat :

```
>>> paquet = Paquet()
>>> print(paquet)
as de trèfle
2 de trèfle
3 de trèfle
...
10 de pique
```

(suite sur la page suivante)

(suite de la page précédente)

```
valet de pique
dame de pique
roi de pique
```

Même si le résultat apparaît sur 52 lignes, c'est une longue chaîne qui contient des caractères de fin de ligne.

Ajouter, enlever, mélanger et trier

Pour distribuer des cartes, nous voudrions une méthode qui enlève une carte du paquet et la renvoie. La méthode de liste pop offre un moyen pratique de le faire :

```
  # à l'intérieur de la classe Paquet :

      def pop_carte(self):
          return self.cartes.pop()
```

Comme `pop` retire la `dernière` carte dans la liste, nous distribuons les cartes à partir de la fin du paquet. Pour ajouter une carte, nous pouvons utiliser la méthode de liste `append` :

```
# à l'intérieur de la classe Paquet :

  def ajouter_carte(self, carte):
      self.cartes.append(carte)
```

Une méthode comme celle-ci, qui utilise une autre méthode sans faire beaucoup de travail s'appelle parfois un placage. La métaphore vient du travail en bois, où un **placage** est une mince couche de bois d'essence noble collée à la surface d'une pièce en bois moins cher, pour améliorer l'apparence.

Dans ce cas, `ajouter_carte` est une méthode « mince » qui exprime une opération de liste en termes appropriés pour les paquets. Elle améliore l'apparence, ou l'interface, de la mise en œuvre.

Nous pouvons également écrire une méthode de `Paquet` nommée `battre` en utilisant la fonction `shuffle` du module `random` :

```
# à l'intérieur de la classe Paquet :

  def battre(self):
      random.shuffle(self.cartes)
```

N'oubliez pas d'importer `random`.

À titre d'exercice, écrivez une méthode de `Paquet` appelée `trier`, qui utilise la méthode de liste `sort` pour trier les cartes d'un Paquet. La méthode trier utilise la méthode `__lt__` que nous avons définie pour déterminer l'ordre.

Héritage

L'héritage est la capacité de définir une nouvelle classe qui est une version modifiée d'une classe existante. À titre d'exemple, disons que nous voulons une classe pour représenter une « main », c'est-à-dire les cartes détenues par un seul joueur. Une main est semblable à un paquet : les deux sont constitués d'une collection de cartes, et les deux nécessitent des opérations comme l'ajout et le retrait de cartes.

En même temps, une main est différente d'un paquet ; il existe des opérations que nous voulons pour les « mains » qui n'ont pas de sens pour un paquet. Par exemple, au poker, nous pourrions comparer deux mains pour voir qui gagne. Au bridge, nous pourrions calculer le nombre de points d'une main afin de faire une enchère.

Cette relation entre classes - similaires, mais différentes - se prête bien à l'héritage. Pour définir une nouvelle classe qui hérite d'une classe existante en Python, vous mettez le nom de la classe existante entre parenthèses :

```
class Main(Paquet):
    """Représente une main au jeu de cartes."""
```

Cette définition indique que `Main` hérite de `Paquet` ; cela signifie que nous pouvons utiliser des méthodes comme `pop_carte` et `ajouter_carte` tant pour les Mains que pour les Paquets.

Lorsqu'une nouvelle classe hérite d'une classe existante, la classe existante est appelée **classe mère** ou **classe parente** et la nouvelle classe est appelée **classe fille** ou **classe enfant**.

Dans cet exemple, `Main` hérite `__init__` de `Paquet`, mais celle-ci ne fait pas vraiment ce que nous voulons : au lieu d'alimenter la main avec 52 nouvelles cartes, la méthode `init` pour Mains doit initialiser `cartes` à une liste vide.

Si nous fournissons une méthode d'initialisation à la classe `Main`, elle remplace celle de la classe `Paquet` :

```
# à l'intérieur de la classe Main :

    def __init__(self, etiquette = ''):
        self.cartes = []
        self.etiquette = etiquette
```

Lorsque vous créez une Main, Python appelle cette méthode init, pas celle de Paquet.

```
>>> main = Main('nouvelle main')
>>> main.cartes
[]
>>> main.etiquette
'nouvelle main'
```

Les autres méthodes sont héritées de `Paquet`, donc nous pouvons utiliser `pop_carte` et `ajouter_carte` pour distribuer une carte :

```
>>> paquet = Paquet()
>>> carte = paquet.pop_carte()
>>> main.add_carte(carte)
>>> print(main)
roi de pique
```

Une prochaine étape naturelle consiste à encapsuler ce code dans une méthode appelée `deplacer_cartes` :

```
# à l'intérieur de la classe Paquet :

    def deplacer_cartes(self, main, nombre):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        for i in range(nombre):
            main.ajouter_carte(self.pop_carte())
```

deplacer_cartes prend deux arguments, un objet Main et le nombre de cartes à distribuer. Elle modifie tant self (le paquet) que main, et renvoie None.

Dans certains jeux, les cartes sont déplacées d'une main à l'autre, ou remises d'une main vers le paquet. Vous pouvez utiliser deplacer_cartes pour les deux opérations : self peut être soit un Paquet, soit une Main, et main, malgré le nom, peut aussi être un Paquet.

L'héritage est une fonctionnalité utile. Certains programmes qui seraient répétitifs sans héritage peuvent être écrits plus élégamment en l'utilisant. L'héritage peut faciliter la réutilisation du code, puisque vous pouvez personnaliser le comportement des classes parentes sans devoir les modifier. Dans certains cas, la structure de l'héritage reflète la structure naturelle du problème, ce qui rend la conception plus facile à comprendre.

D'un autre côté, l'héritage peut rendre les programmes difficiles à lire. Quand une méthode est invoquée, parfois on ne sait pas trop où trouver sa définition. Le code en question peut être réparti sur plusieurs modules. De plus, beaucoup de choses qui peuvent être faites en utilisant l'héritage peuvent être faites aussi bien ou mieux sans lui.

CHAPITRE 4

Algorithmes

Objectifs :

- Étudier le coût d'un algorithme et prouver sa validité sur des cas simples :
  - évaluer le nombre d'opérations effectuées ;
  - prouver la correction d'un algorithme ;
  - prouver la terminaison d'un algorithme.
- Connaître quelques algorithmes classiques de calcul et de recherche utilisant parcours séquentiel dans un ensemble de valeurs :
  - calculer une moyenne ;
  - rechercher un extremum ;
  - rechercher une valeur particulière ;
- Comprendre l'intérêt de la recherche par dichotomie :
  - prouver la validité ;
  - évaluer le coût.

4.1 1.0 Origines

L'algorithmique joue un rôle de plus en plus important dans notre société. Derrière tous les sites Web que nous utilisons se cachent des programmes sophistiqués conçus par des ingénieurs de haut niveau. Comprendre et découvrir la structure des langages joue un double rôle : d'une part connaître les qualités, mais aussi les limites de l'outil informatique et d'autre part développer l'esprit logique.

Abu Abdallah Muhammad Ibn Musa Al-Khwârizmi(780-880), le « père de l'algèbre » était un savant de Bagdad, originaire du Khwârizm, une région d'Asie Centrale, actuel Ouzbékistan. Ses écrits en langue arabe ont permis la diffusion jusqu'en Europe des chiffres arabes et de l'algèbre, mot qui a pour origine le titre d'un de ses ouvrages.

Ses écrits seront traduits en latin vers le XIIe siècle. Il a classifié les algorithmes existant à son époque et son nom est à l'origine du mot algorithme. Dans l'un de ses ouvrages, il présente les équations canoniques de degré inférieur ou égal à 2, avec des exemples. Puis il propose des algorithmes de résolution pour ces équations.

L'algorithme d'Euclide est peut-être le plus ancien algorithme non trivial. On le trouve dans le livre VII des Eléments, premier traité écrit de mathématiques, datant d'environ 2300 ans.

Mais des algorithmes étaient déjà utilisés dans le calcul à Babylone, au sud de Bagdad dans l'actuel Irak.

Donald Knuth a eu l'occasion d'étudier des tablettes datant d'environ 2000 ans avant notre ère, avec des calculs effectués en base 60 et des nombres écrits en virgule flottante !

4.1.1 Algorithme d'Euclide

Il est utilisé pour le calcul du **PGCD** de deux entiers `m` et `n`. **Étape 1** : on divise `m` par `n` et on note `r` le reste (`0 r < n`). **Étape 2** : si `r = 0`, c'est terminé, le **pgcd** est `n`. **Étape 3** : sinon, on remplace `m` et `n` par `n` et `r` et on recommence à l'étape 1.

En **python cela donne** :

Code source 1 – Fonction pgcd

```
def pdcd(m, n):
  r = m % n
  while r != 0:
    m, n = n, r
    r = m % n
  return n
```

Donald Knuth énonce quelques règles dans un ouvrage monumental, *The Art of Computer Programming*, dont l'écriture a commencé en 1962 et la publication du premier volume date de 1968.

L'ouvrage commence par un algorithme décrivant la manière de lire le premier volume de cet ensemble de livres puis de lire les différents volumes ! Après quelques pages, il présente cinq caractéristiques importantes d'un algorithme :

- Un algorithme doit toujours se terminer après un nombre fini d'étapes.
- Chaque étape d'un algorithme doit être définie précisément, les actions à mener doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas.
- Un algorithme a des entrées, zéro ou plus, quantités qui lui sont données avant ou pendant son exécution.
- Un algorithme a une ou plusieurs sorties, quantités qui oui une relation spécifiée avec les entrées.
- Les instructions doivent être suffisamment basiques pour pouvoir être en principe exécutées de manière exacte et en un temps fini par une personne utilisant un papier et un crayon.

4.2 2.0 Les outils

4.2.1 2.1 Compteurs et accumulateurs

Un **compteur**, comme son nom l'indique, sert à **compter**. Par exemple, on peut **compter** le nombre d'essais dans un jeu. De manière générale, c'est une variable initialisée à **0** qui est incrémentée d'une unité à chaque passage dans une boucle, éventuellement suite à un test.

Nous allons voir quelques exemples.

2.1.1 Compteur et boucle conditionnelle

Code source 2 – Sans test

```
def taille(n) :
 cpt=0
 while n > 0:
   cpt = cpt + 1
   n = n // 2
 return cpt
```

On compte le nombre de divisions euclidiennes successives de `n` par `2`, jusqu'à arriver à un quotient nul. On obtient donc le nombre de chiffres dans l'écriture binaire de `n`.

Code source 3 – Avec test

```
def nombre_de_1(n) :
cpt = 0
while n > 0 :
  if n % 2 == 1
    cpt = cpt + 1
  n = n // 2
return cpt
```

Le programme est identique un précédent, mais on incrémente le compteur seulement quand un reste vaut `1`. On compte donc le nombre de `1` dans l'écriture binaire de `n`.

4.2.2 2.2 Compteur et boucle inconditionnelle

Dans une boucle inconditionnelle sans test, un compteur compterait le nombre de passages dans la boucle. Or, ce nombre est connu à l'avance et donc un compteur n'apporterait rien.

2.2.1 Permutation de valeurs

On est souvent amené à devoir permuter des valeurs entre des variables, en particulier échanger les valeurs de deux variables. Cet échange est présent dans les algorithmes de tri exposés au chapitre suivant qui reposent sur la comparaison de deux valeurs et en fonction du résultat leur éventuelle permutation. Dans la construction d'une suite de nombres définie par une relation du type $u_n + 2 = f(u_n + 1, u_n)$, on calcule un terme en fonction des deux précédents. Ceci est possible en utilisant seulement deux variables puisque cela revient à passer du couple $(u_n + 1, u_n)$ au couple $(u_n + 2, u_n + 1)$. Le principe général est simple. Avant tout, que penser du code suivant ?

À la troisième affectation, $var1$ prend la valeur courante de $var2$ donc 23. La quatrième affectation utilise la valeur courante de $var1$, qui est 23 à ce moment, donc $var2$ prend la valeur 23. Les deux variables ont finalement pour valeur 23, soit la valeur initiale de $var2$. On comprend qu'il faut modifier la valeur d'une variable sans perdre sa valeur initiale qu'il faut donc stocker dans une troisième variable.

La valeur de $var1$, 17 est gardée dans `temp`. Ce nom est choisi car cette variable est temporaire, elle n'est utilisée que le temps de rechange. On peut alors modifier la valeur de $var1$ en lui affectant la valeur de $var2$ et finalement affecter à $var2$ la valeur initiale de $var1$ qui n'a pas été perdue.

Ce procédé est utilisé dans de nombreux langages. Certains langages possèdent une fonction `swap` pour permuter deux valeurs. En Python, c'est l'existence du type `tuple` qui nous permet d'avoir une instruction simplifiée : `var1, var2 = var2, var1`. Cette instruction signifie que le couple `(var1, var2)` prend la valeur du couple `(var2, var1)`

c'est-à-dire la valeur `(23, 17)`. Donc `var1` prend la valeur 23 et var2 prend la valeur 17 L'échange des valeurs est réalisé.

2.2.2 Tests et boucles

Dans la plupart de nos programmes, nous trouvons des tests qui utilisent les structures `if .. -` , ou `if ... else ...`, ou `if ... elif .`,``. else ....`` Première remarque : `else` n'est jamais suivi d'une expression. Une expression après `else` signifierait « *sinon, si l'expression a la valeur True* » et nous sommes alors dans le cadre d'une structure `elif ----"Sinon"` signifie : si ce qui est avant est faux ».

Par exemple : Nous avons trois cas distincts : le premier cas est `x > 0`, le deuxième cas est `x < 0`, le troisième cas regroupe tout ce qui n'est ni dans le premier cas ni dans le deuxième, donc si x est nul. L'écriture `else x == 0` provoquerait une erreur.

Par exemple : Si la valeur initiale de x est 5, alors elle est actualisée à 2. Si la valeur initiale de x est -2, elle est actualisée à 3. Si la valeur initiale de x est 0, elle est actualisée à 2.

Une deuxième remarque : la structure `if ... if ... else ...` n'est pas équivalente à la structure `if ... elif .. . else ....` En parlant, nous pouvons énoncer : si x est strictement positif, puis si x est strictement négatif, sinon, ... Et certains comprendront peut être que sinon correspond à x nul. Ce n'est évidemment pas le cas.

2.2.3 Examinons le code qui suit :

Code source 4 – Quelle est la valeur finale de x si la valeur initiale est 5 ?

```
if x > 0 :
  x = x -3
if x < 0 :
  x = x + 5
else :
  x = x + 2
```

Nous avons ici deux blocs distincts : le bloc `if x > 0 ... ``et le bloc ``if x > 0 ... else ....` Donc puisque la valeur de `x` est strictement positive, elle est actualisée à 2. Ensuite, puisque la valeur courante de `x` est strictement positive, elle est actualisée à 4. Si la valeur initiale de `x` est 2, elle est strictement positive donc elle est actualisée à -1. Puis elle est strictement négative donc elle est actualisée à 4.

Troisième remarque : l'expression qui suit `if` a une valeur `True` ou `False` ou une valeur qui peut être interprétée comme `True` ou `False`.

Prenons par exemple `if x > 0`. Dans cet exemple, nous n'écrirons pas `if (x > 0) == True`. Donc il en est de même si l'expression est composée d'une seule variable booléenne ou d'un appel de fonction renvoyant un booléen. Nous écrirons `if b` et `if f (x)` et non pas `if b = True` ou `if f (x) True`. Ces deux dernières écritures reviendraient à tester `True == True` ou `False == True`.

2.2.4 Boucles

Nos programmes sont constitués de boucles et même de boucles imbriquées. Nous pouvons avoir une ou plusieurs boucles `while` ou `for` a l'intérieur d'une boucle `while` ou `for`. Par exemple :

```
for i in range(4) :
  for j in range(3) :
    print(i + j)
```

Les valeurs successives des variables `i` et `j` sont : `i = 0` et `j = 0`, puis `j = 1`, puis `j = 2`, ensuite `i = 1` et `j = 0`, puis `j = 1`, puis `j = 2`, ensuite `i = 3` et `j = 0`, puis `j = 1`, puis `j = 2`.

Pour chacune des quatre valeurs de `i`, `(0, 1, 2, 3)`, `j` prend trois valeurs, `(0,1, 2)`, et nous aurons donc douze affichages avec la fonction `print`.

```
0
1
2
1
2
3
2
3
4
3
4
5
```

4.3 3.0 Validité et coût

Lorsqu'on écrit un algorithme, il est impératif de vérifier que cet algorithme va produire un résultat en un temps fini et que ce résultat sera correct dans le sens où il sera conforme à une spécification précise. Nous dirons alors que l'algorithme est valide.

4.3.1 3.1 Validité d'un algorithme itératif

Correction

Un algorithme itératif est construit avec des boucles. Pour prouver qu'il est correct, nous disposons de la notion d'invariant de boucle.

Définition

Un invariant d'une boucle est une propriété qui est vérifiée avant l'entrée dans une boucle, à chaque passage dans cette boucle et à la sortie de cette boucle. On peut faire le lien avec les suites définies par récurrence du programme de mathématiques. Pour démontrer qu'une propriété est un invariant d'une boucle, on utilise un raisonnement semblable au raisonnement par récurrence. On commence par vérifier que la propriété est vraie avant l'entrée dans la boucle. Cette étape s'appelle l'initialisation. On prouve ensuite que si la propriété est vraie avant un passage dans la boucle, alors elle est vraie après ce passage. Cette étape s'appelle l'hérédité. On peut alors conclure que la propriété est vraie à la sortie de la boucle.

Exemple Voici un algorithme de calcul avec une boucle conditionnelle et deux variables a et b, a ayant pour valeur un entier naturel :

```
m = 0
p = 0
tant que m < a
   m = m + 1
   p = p + b
fin du tant que
```

Notons m et p les valeurs des variables m et p.

Nous allons montrer que la propriété $p = m * b$ est un invariant de la boucle « **tant que** ».

Avant le premier passage dans la boucle, $m = 0$ et $p = 0$, donc l'égalité $p = m * b$ est vraie. Supposons que $p = m * b$ avant un passage dans la boucle. Les nouvelles valeurs de m et p après le passage, notées m' et p' vérifient : $m' = m+1$ et $p' = p + b$. Alors $p' = m * b + b = (m + 1) * b = m' * b$. Donc la propriété est vraie après ce passage dans la boucle. Nous pouvons conclure qu'à la sortie de la boucle, $p = mxb$. Et puisqu'à la sortie de la boucle, la variable m a pour valeur celle de a, nous avons finalement obtenu le produit $p = a * b$.

Terminaison

Un algorithme ne doit toujours comporter qu'un nombre fini d'étapes. Afin de prouver la terminaison d'un algorithme itératif, (qui contient une boucle), nous utilisons la notion de variant. Nous parlons ici de boucles conditionnelles. Dans le cas de boucles non conditionnelles, le nombre d'étapes est déterminé.

Méthode

On choisit un variant, c'est-à-dire une expression, la plus simple étant une variable, telle que la suite formée par les valeurs de cette expression au cours des itérations converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt. Considérons par exemple le code suivant où la valeur de la variable a est un nombre quelconque :

```
x = 0
while x ** 2 < a :
  x = x + 1
```

Si la valeur de a est négative ou nulle, il n'y a aucun passage dans la boucle. Sinon, la suite des valeurs de la variable x, le variant choisi, est 0, 1, 2, n et n'est certainement la première valeur supérieure ou égale à la racine carrée de la valeur de a. Le nombre de passages dans la boucle est donc fini.

Revenons sur l'exemple du produit de deux nombres étudié plus haut. Nous avons prouvé qu'en sortie de boucle, la valeur de p était le produit $a * x$. Mais nous n'avons pas prouvé la terminaison, c'est-à-dire que la sortie de boucle était effective après un nombre fini de passages. Pour cela, dans cet exemple, nous choisissons comme variant la variable m. Cette variable prend pour valeurs successives 0, 1, …, m et il y a donc exactement a passages dans la boucle, ce qui prouve la terminaison.

4.3.2 3.2 Coût

Un programme doit traiter une liste de 10^7 éléments puis une liste de 10^8 éléments. Est-ce que le temps d'exécution du programme sera multiplié par 10 ? Quel est le rapport entre le temps d'exécution est la taille de la liste ?

Ce sont des questions auxquelles il faut réfléchir quand on écrit un algorithme.

Les réponses sont variées et dépendent de l'algorithme et de la liste. Pour une liste donnée, un programme peut être plus rapide qu'un autre, mais avec une autre liste, ce peut être le contraire.

Le même programme peut être plus rapide avec la liste la plus longue.

De plus, pour traiter un même problème, non seulement nous pouvons disposer de plusieurs algorithmes mais un même algorithme peut avoir un temps d'exécution différent selon le langage de programmation utilisé et suivant la machine sur laquelle le programme est exécuté. L'étude n'est pas simple à réaliser et pour comparer deux algorithmes nous allons

ici nous concentrer sur le nombre d'opérations à effectuer en essayant d'évaluer un ordre de grandeur de ce nombre en fonction de la taille des données.

Nous parlerons du coût d'un algorithme ou de sa complexité. Ce coût pouvant être très différent pour une même taille de données, nous nous placerons dans le pire des cas, celui où le coût est le plus important.

Etudions quelques exemples simples qui sont à la base de ce type d'étude. Nous commençons par un algorithme rencontré précédemment.

```
m = 0
p = 0
tant que m < a
   m = m + 1
   p = p + b
fin du tant que
```

Les passages dans la boucle ont lieu pour les valeurs de m égales à 0, 1, 2, ..., $a - 1$ si la valeur de la variable a est un entier naturel a. Nous avons donc exactement a passages dans la boucle.

A chaque passage, nous comptons deux additions ainsi que deux affectations. Nous pouvons donc dire que le nombre d'additions est $2a$ ou que le nombre d'opérations, au sens large en comptant les affectations, est $4a$. Nous dirons alors que le coût est proportionnel à a ou qu'il est linéaire. Avec une boucle « tant que », le calcul peut être plus compliqué puisque le nombre de passages dans la boucle varie avec les cas pour une même taille de données. Nous devons alors identifier le pire des cas, c'est-à-dire celui où le nombre de passages est maximal.

Considérons une boucle « pour » où le nombre de passages dans la boucle est bien déterminé.

```
somme = 0
for i in range(1, n+1) :
   somme = somme + i
```

Cet algorithme, ou ce programme, permet de calculer la somme des entiers de 1 à n. Il y a clairement n passages dans la boucle. A chaque passage nous avons une addition et une affectation, donc un total de n additions et n affectations. Nous pouvons affirmer que le coût est linéaire.

4.3.3 3.3 Complexité linéaire

Soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $n+$, avec et réels et > 0,nous disons que l'algorithme a un coût linéaire ou une complexité linéaire. Dans le cas de deux boucles `for` imbriquées, nous avons trois cas typiques. Dans un cas, le coût est **linéaire**. Dans les deux autres cas, nous disons que le coût est **quadratique**.

4.3.4 3.4 Complexité quadratique

Soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $n2 + n+$, avec , et réels, > 0, nous disons que l'algorithme a un coût quadratique ou une complexité quadratique. Dans les codes qui suivent, les pointillés sous-entendent un nombre fixe d'opérations.

Premier cas : n est la taille de la donnée, k est un nombre fixé.

```
for i in range(n) :
    ....
    for j in range(k) :
      ....
```

Deuxième cas : est la taille de la donnée.

```
for i in range(n) :
    ....
    for j in range(n) :
      ....
```

Nous avons n passages dans la boucle externe. À chaque passage, nous avons un nombre fixe d'opérations q puis n passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de i, nous avons $q + n * r$ opérations. Le nombre total d'opérations est donc $n(q + nxr) = rn_2 + qn$ et le coût est quadratique.

Troisième cas : n est la taille de la donnée.

```
for i in range(n) :
    ....
    for j in range(i) :
      ....
```

Nous avons n passages dans la boucle externe. À chaque passage, pour chaque valeur de i, nous avons un nombre fixe d'opérations q puis i passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de i, nous avons $q + i * r$ opérations. Les valeurs de i sont successivement $0, 1, 2, ..., n - 1$.

Le nombre total d'opérations est donc $q+(q+1*r)+(q+2*r)+...+(q+(n-1)*r)$, soit $nq+r(1+2+...+(n-1))$.

Le calcul de la somme des entiers est connu. Le résultat est $nq + r(n(n - 1))/2$.

Finalement le nombre d'opérations est $r/2 * n2 + (q\breve{}r/2)n$ de la forme $n2 + n$ et le coût quadratique.

4.4 4.0 Algorithmes de tri

Les algorithmes de tri sont fondamentaux dans la gestion des données et permettent l'accès à des informations dans des délais très brefs. Aussi l'ingéniosité des informaticiens a été mise en œuvre pour élaborer ceux qui nécessitent le moins d'opérations.

Trier des données, c'est les ranger suivant un ordre défini au préalable. Par exemple avec des données numériques, nous pouvons trier ces données en utilisant l'ordre défini en mathématiques : a est avant b si $b - a$ est positif ($a < b$ si $b - a > 0$).

Si nous trions l'ensemble de nombres $3, 8, 5, 2$, nous obtenons l'ensemble $2, 3, 5, 8$ et nous disons que les nombres sont rangés suivant l'ordre croissant. Si nous les rangeons dans l'ordre inverse, nous parlons d'ordre décroissant.

Les lettres de notre alphabet sont rangées suivant l'ordre alphabétique. Nous pouvons alors trier un ensemble de mots, par exemple l'ensemble de quatre mots $bonjour, bon, table, assiette$, en suivant l'ordre lexicographique qui utilise lui-même l'ordre alphabétique. Nous obtenons l'ensemble $assiette, bon, bonjour, table$.

Des suites d'instructions permettant d'effectuer un tri ont été étudiées dès les années 1940 sur les premiers ordinateurs. Durant cette période, l'américaine **Betty Holberton** était l'une des six programmatrices de l'ENIAC, le premier ordinateur entièrement électronique.

Elle travaillait sur différents codes et applications et c'est à ce moment qu'elle développa sans doute **le premier programme de tri**. Elle travaillera plus tard avec **Grâce Hopper** sur les langages de programmation **COBOL** et **Fortran**.

Dans les algorithmes présentés plus loin, nous procédons par des comparaisons successives entre deux éléments et effectuons éventuellement une permutation des deux éléments comparés.

Le nombre de **permutations** est donc toujours **inférieur au nombre de comparaisons**. Le **coût** ou la **complexité** d'un algorithme est dans ce cas un *ordre de grandeur du nombre de comparaisons* effectuées par cet algorithme.

Pour un algorithme donné, le coût peut être différent suivant les cas à traiter, il y a des cas favorables, et d'autres non. Ce coût doit absolument être évalué avant d'écrire un programme et l'exécuter car suivant la taille de l'ensemble de données, **le temps de calcul avec une machine peut devenir rédhibitoire**.

Prenons trois nombres tout distincts a, b et c.

Il y a six manières de ranger ces trois nombres :
- (a, b, c)
- (a, c, b)
- (b, a, c)
- (b, c, a)
- (c, a, b)
- (c, b, a)

Si en comparant a et b, nous obtenons $a < b$, alors le nombre de rangements possibles est **divisé par deux**, il n'en reste plus que trois :
- (a, b, c)
- (a, c, b)
- (c, a, b)

La comparaison de a et c permet de conclure si $c < a$, sinon il faut une troisième comparaison entre b et c.

Pour quatre nombres (a, b, c, d), il y a **quatre** fois plus de rangements possibles.

En effet, pour chacun des six rangements de a, b et c, il y a quatre places possibles pour d : * en premier, * soit entre le premier et le deuxième, * soit entre le deuxième et le troisième, * soit en dernier.

Après une première comparaison, il reste **douze** rangements possibles, et après une seconde comparaison il en reste **six**. Au total, nous avons donc cinq comparaisons au maximum.

Le nombre total r de rangements possibles de n données vaut $n * (n - 1) * 2 * 1$ noté $n!$, n factoriel.

Après chaque comparaison, ce nombre de rangements possibles est divisé par 2. Donc après k comparaison, il est divisé par $2k$. Le tri est terminé lorsqu'il ne reste plus qu'un seul rangement possible, c'est-à-dire dès que $2k > r$.

Remarque : en binaire, 2 s'écrit avec $k + 1$ bits. Donc le nombre de comparaisons est de l'ordre du nombre de chiffres dans l'écriture binaire de r.

Il est intéressant de faire un test avec un jeu de 32 cartes, un ordre étant défini sur l'ensemble des cartes. On utilise un ordre sur les valeurs, 7, 8, 9, 10, valet, dame, roi, as, et pour ordonner deux cartes de même valeur, un ordre sur les couleurs. Comme pour l'ordre lexicographique avec des mots de deux lettres, il s'agit d'un ordre sur les couples $(valeur, couleur)$.

On distribue une dizaine de cartes à une personne et on lui demande d'ordonner les cartes. On recommence avec d'autres personnes. Avec chaque personne, on essaie de dégager une stratégie, **un algorithme de tri**. On peut alors remarquer que des algorithmes comme le **tri sélection**, le **tri insertion**, ainsi que d'autres plus élaborés sont utilisés et souvent à tour de rôle pour un tri donné.

L'idée « **diviser pour régner** » est présente.

4.4.1 4.1 Tri par sélection

En anglais, cet algorithme est nommé « *selection sort* ».

Le principe

On dispose de n données. On cherche la plus petite donnée et on la place en première position, puis on cherche la plus petite donnée parmi les données restantes et on la place en deuxième position, et ainsi de suite.

Si les données sont les éléments d'une liste $liste$, l'algorithme consiste donc à faire varier un indice i de 0 à $n-2$. Pour chaque valeur de i, on cherche dans la tranche liste $[i:n]$ le plus petit élément et on l'échange avec $liste[i]$.

L'algorithme de **tri par sélection** est souvent utilisé pour **trier à la main des objets**, comme des cartes à jouer, des livres, etc.

Code source 5 – **Algorithme du minimum**

```
i_mini ← i (indice du plus petit élèment)
mini ← liste[i]
pour j variant de i+1 à n-1
    si liste[j] < mini
        i_mini ← j
        mini ← liste[j]
```

Pour obtenir un algorithme du tri sélection, il ne reste qu'à insérer l'algorithme du minimum dans une boucle où i varie de 0 à $n-2$ et pour chaque valeur de i à faire échange de `liste[i]` avec `liste[i_mini]`.

La donnée en entrée est une liste de n éléments. Il n'y a pas de résultat renvoyé en sortie, la liste est modifiée en place.

Code source 6 – **Algorithme du tri**

```
POUR i VARIANT de i à n-2
    i_mini ← i
    mini ← liste[i]
    POUR j VARIANT de i+1 à n-1
        SI liste[j] < mini
            i_mini ← j
            mini ← liste[j]
    ECHANGER liste[i] et liste[i_mini]
```

```python
def tri_selection(liste) :
  for i in range(len(liste) -1 ):
      i_mini = i
      mini = liste[i]
      for j in range (i+1, len(liste)) :
          if(liste[j] < mini ) :
              i_mini = j
              mini = liste[j]
      liste[i], liste[i_mini] = liste[i_mini], liste[i]
```

Exemple avec la liste $[7,4,3,2,9,5]$ et les éléments échangés.

$[2,4,3,7,9,5]$ après échange de 2 et 7.

Pour `i` égal à 0 Pour `i` égal à 1 Pour `i` égal à 2 Pour `i` égal à 3 Pour `i` égal à 4

$[2,3,4,7,9,5]$ après échange de 3 et 4. $[2,3,4,7,9,5]$ après aucun échange. $[2,3,4,5,9,7]$ après échange de 5 et 7. $[2,3,4,5,7,9]$ après échange de 7 et 9.

La liste passée en paramètre est modifiée en place. Donc pour utiliser cette fonction, il suffit d'écrire l'instruction `tri_selection(liste)`. Si nous ne voulons pas modifier la liste passée en paramètre il faut en faire une copie et ensuite renvoyer une nouvelle liste qui est triée. On obtient alors le programme suivant :

```
def tri_selection(liste) :
liste = list(liste)
for i in range(len(liste) -1 ):
    i_mini = i
    mini = liste[i]
    for j in range (i+1, len(liste)) :
    if(liste[j] < mini ) :
        i_mini = j
        mini = liste[j]
        liste[i], liste[i_mini] = liste[i_mini], liste[i]
return liste

# testons le programme :

tri = tri_selection([7, 4, 3, 2, 9, 5])
print(tri)
```

La console retourne :

```
>>> tri = tri_selection([7, 4, 3, 2, 9, 5])
>>> print(tri)
```