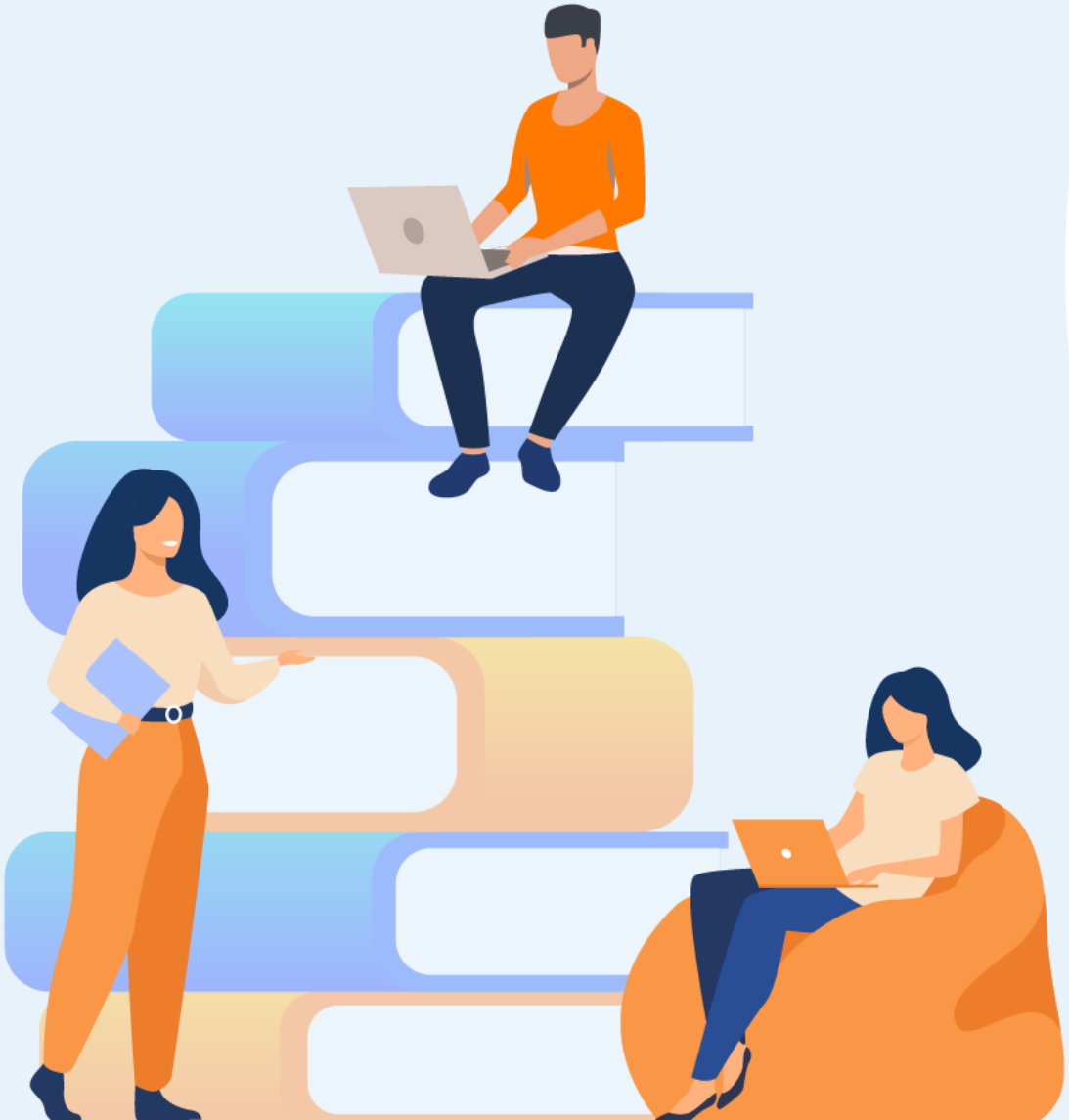


Partie 4

Manipuler les conteneurs

Dans cette partie, vous allez :

- Appréhender la notion des conteneurs
- Prendre en main Docker

CHAPITRE 1

Appréhender la notion des conteneurs

Ce que vous allez apprendre dans ce chapitre :

- Définition ;
- Différence entre machine virtuelle et conteneur ;
- Avantages

CHAPITRE 1

Appréhender la notion des conteneurs

1. **Définition ;**
2. Différence entre machine virtuelle et conteneur ;
3. Avantages

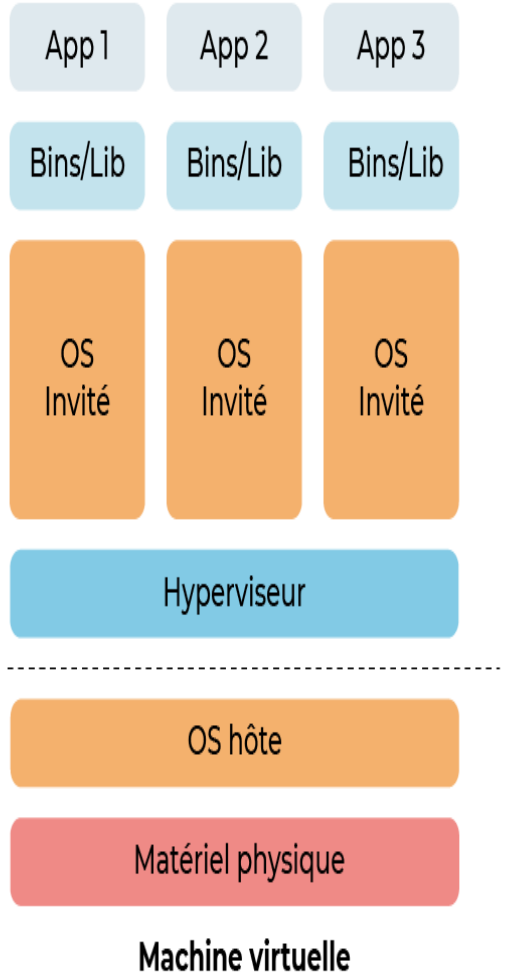
PARTIE 1

comprendre la notion de machine virtuelle

Lorsque vous utilisez une machine virtuelle (VM), vous faites ce qu'on appelle de la virtualisation lourde. En effet, vous recréez un système complet dans le système hôte, pour qu'il ait ses propres ressources.

L'isolation avec le système hôte est donc totale ; cependant, cela apporte plusieurs contraintes :

- une machine virtuelle prend du temps à démarrer ;
- une machine virtuelle réserve les ressources (CPU/RAM) sur le système hôte.



Machine virtuelle

Copyright - Tout droit réservé -

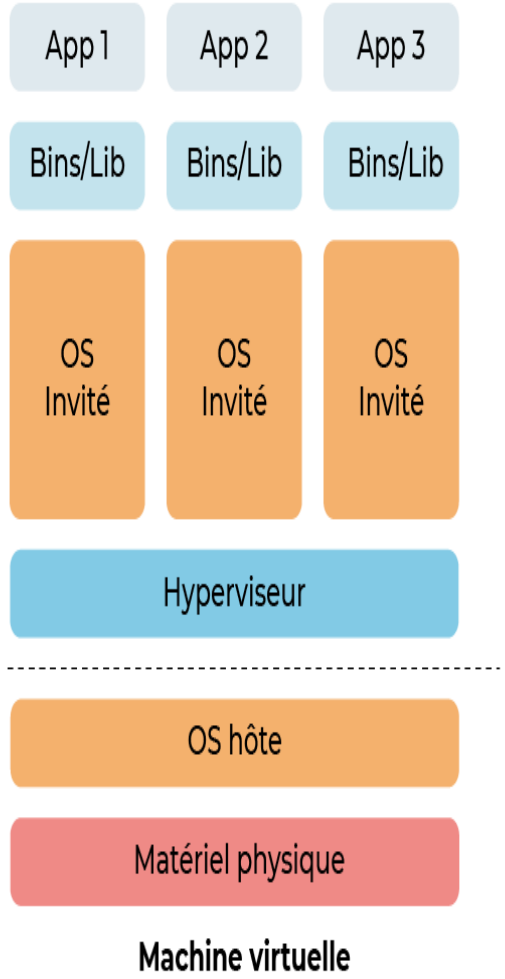
PARTIE 1

comprendre la notion de machine virtuelle

Mais cette solution présente aussi de nombreux avantages :

- une machine virtuelle est totalement isolée du système hôte ;
- les ressources attribuées à une machine virtuelle lui sont totalement réservées ;
- vous pouvez installer différents OS (Linux, Windows, BSD, etc.).

Mais il arrive très souvent que l'application qu'elle fait tourner ne consomme pas l'ensemble des ressources disponibles sur la machine virtuelle. Alors est né un nouveau système de virtualisation plus léger : les conteneurs.



Machine virtuelle

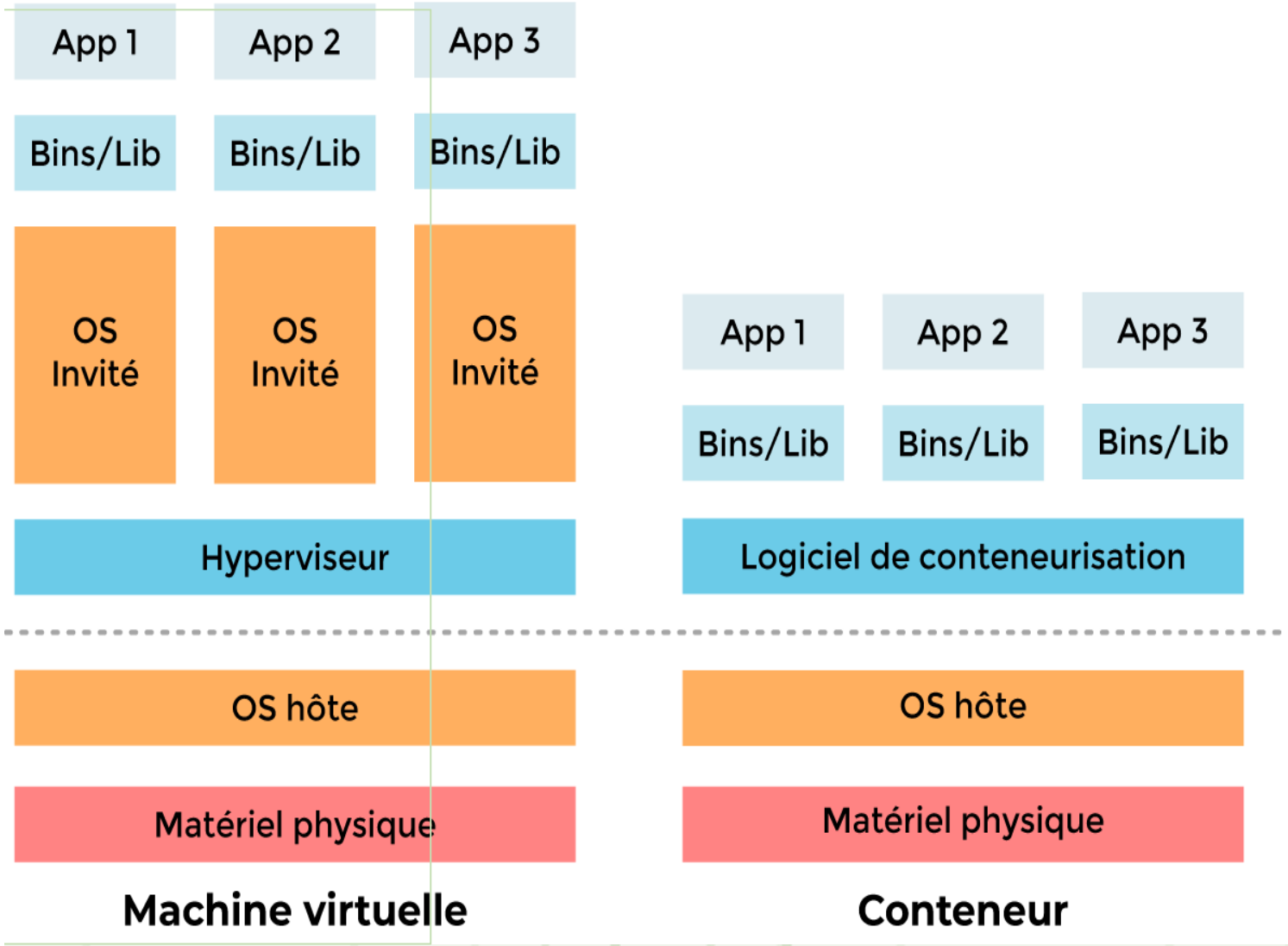
Copyright - Tout droit réservé -

PARTIE

Un conteneur ?

Un conteneur Linux est un **processus** ou un ensemble de processus isolés du reste du système, tout en étant **légers**.

Le conteneur permet de faire de la **virtualisation légère**, c'est-à-dire qu'il ne virtualise pas les ressources, il ne crée qu'une **isolation des processus**. Le conteneur partage donc les ressources avec le système hôte.



Copyright - Tout droit réservé -

avantages des conteneurs.

Ne réservez que les ressources nécessaires :

Une autre différence importante avec les machines virtuelles est qu'un conteneur ne réserve pas la quantité de CPU, RAM et disque attribuée auprès du système hôte. Ainsi, nous pouvons allouer 16 Go de RAM à notre conteneur, mais si celui-ci n'utilise que 2 Go, le reste ne sera pas verrouillé.

Copyright - Tout droit réservé -

avantages des conteneurs.

Démarrez rapidement vos conteneurs :

Les conteneurs n'ayant pas besoin d'une virtualisation des ressources mais seulement d'une isolation, ils peuvent **démarrer beaucoup plus rapidement** et plus fréquemment qu'une machine virtuelle sur nos serveurs hôtes, et ainsi réduire encore un peu les frais de l'infrastructure.

Copyright - Tout droit réservé -

avantages des conteneurs.

Donnez plus d'autonomie à vos développeurs:

En dehors de la question pécuniaire, il y a aussi la possibilité de faire tourner des conteneurs sur le poste des développeurs, et ainsi de réduire les différences entre la "sainte" production, et l'environnement local sur le poste des développeurs.

PARTIE 4

Copyright - Tout droit réservé -

Pourquoi utiliser des conteneurs ?

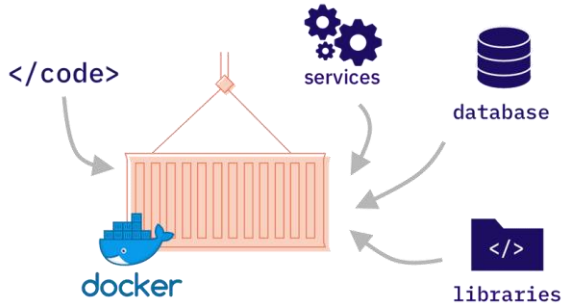
Les conteneurs permettent de **réduire les coûts**, d'augmenter la **densité de l'infrastructure**, tout en améliorant le cycle de déploiement.

Copyright - Tout droit réservé -

PARTIE 4

Définition d'un conteneur

- La **conteneurisation,** est un type de virtualisation, qui consiste à rassembler le code du logiciel et tous ses composants (bibliothèques, frameworks et autres dépendances) de manière à les isoler dans leur propre « **conteneur** » ;



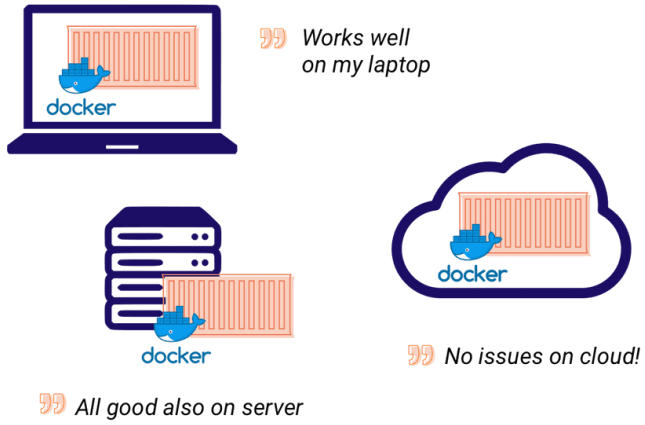
- Le logiciel ou l'application dans le conteneur peut ainsi être **déplacé** et **exécuté** de façon cohérente dans **tous les environnements** et sur **toutes les infrastructures**, indépendamment de leur système d'exploitation ;
- Aujourd'hui, il existe divers outils et plateformes de conteneurisation, à savoir : Docker, LXC, Podman … ;
- **Docker** est l'écosystème le plus populaire et le plus utilisé.

Copyright - Tout droit réservé -

PARTIE 4

Définition de Docker

- Docker est **une plateforme de conteneurs** lancée en 2013 ayant largement contribué à la démocratisation de la conteneurisation.
- Elle permet de **créer facilement des conteneurs** et des applications basées sur les conteneurs.
- C'est **une solution open source**, sécurisée et économique.
- Initialement conçue pour Linux, Docker permet aussi la prise en charge des containers **sur Windows ou Mac** grâce à une " layer " de virtualisation Linux entre le système d'exploitation Windows / macOS et l'environnement runtime Docker.
- L'outil Docker est à la fois bénéfique pour les développeurs et pour les administrateurs système. On le retrouve souvent au cœur des processus **DevOps**.
- Les développeurs peuvent se focaliser sur leur code, sans avoir à se soucier du système sur lequel il sera exécuté. En outre, ils peuvent gagner du temps en incorporant des programmes pré-conçus pour leurs applications.



Copyright - Tout droit réservé -

CHAPITRE 1

Appréhender la notion des conteneurs

1. Définition ;
2. Différence entre machine virtuelle et conteneur ;
3. Avantages

CHAPITRE 2

Prendre en main Docker

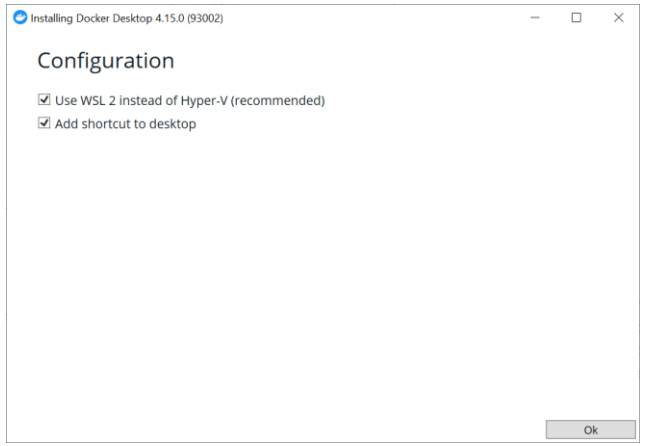
1. Installation de Docker Desktop ;
2. Terminologies Docker : images, Containers, Docker Hub ;
3. Création d’un Dockerfile pour une application simple node js ;
4. Introduction à Docker Compose ;
5. Gestion de plusieurs images Docker avec docker-compose :
6. Démarrage et gestion des conteneurs (commandes docker-compose) ;

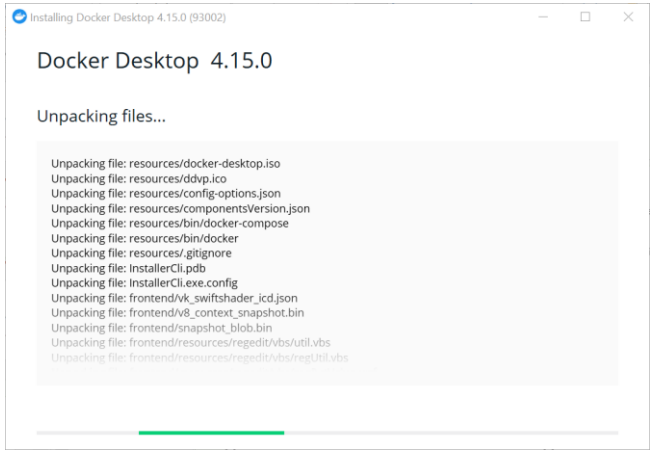
PARTIE 4

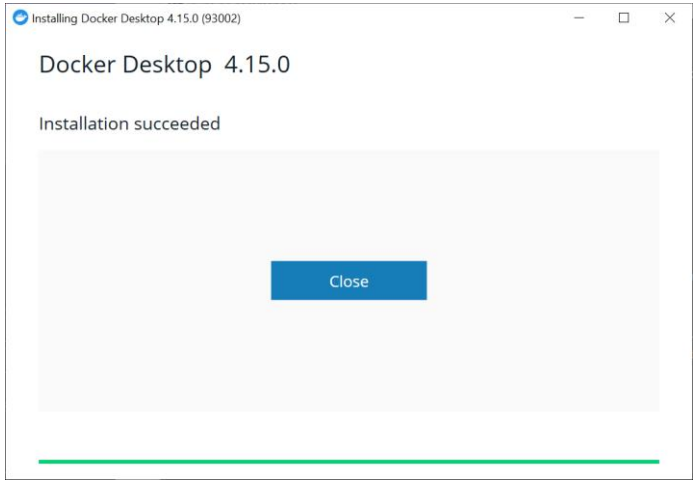
Installation de Docker Desktop :

- Docker Desktop est l'application PC native conçue par Docker pour Windows et Mac. C'est la façon la plus simple d'exécuter, de construire, de déboguer et de tester des applications Dockerisées.

- Le fichier d'installation de Docker Desktop pour windows est à télécharger depuis le site officiel de Docker : https://docs.docker.com/desktop/install/windows-install/
- Sur le même site, on peut vérifier les différents prérequis système nécessaires au bon fonctionnement de Docker Desktop.

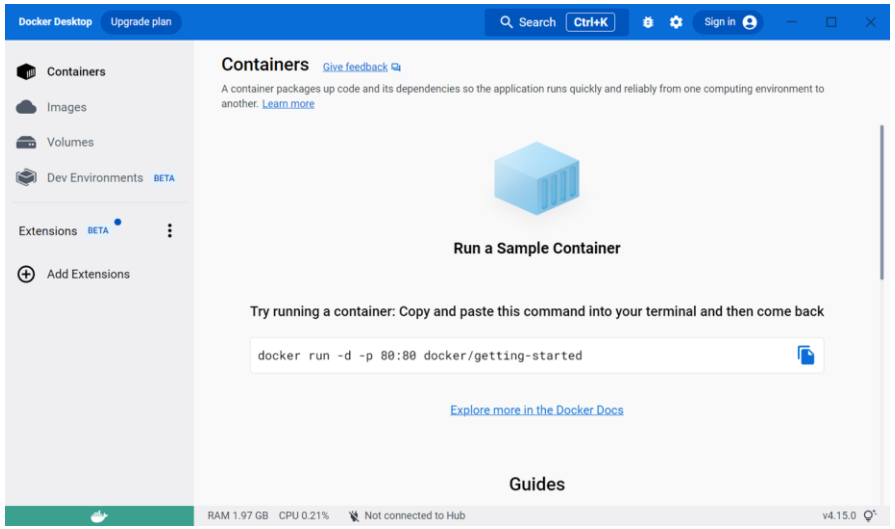






PARTIE 4

Installation de Docker Desktop :



Après avoir installé et démarré Docker Desktop, on peut afficher les différentes informations concernant la version Docker installée en exécutant la commande (sous l'invite de commande) : **docker version**

```
C:\Users\      >docker version
Client:
 Cloud integration: v1.0.29
 Version:           20.10.21
 API version:       1.41
 Go version:        go1.18.7
 Git commit:        baeda1f
 Built:             Tue Oct 25 18:08:16 2022
 OS/Arch:           windows/amd64
 Context:           default
 Experimental:      true

Server: Docker Desktop 4.15.0 (93002)
 Engine:
  Version:          20.10.21
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.18.7
  Git commit:       3056208
  Built:            Tue Oct 25 18:00:19 2022
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.6.10
  GitCommit:        770bd0108c32f3fb5c73ae1264f7e503fe7b2661
 runc:
  Version:          1.1.4
  GitCommit:        v1.1.4-0-g5fd4c4d
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

CHAPITRE 2

Prendre en main Docker

1. Installation de Docker Desktop ;
2. Terminologies Docker : images, Containers, Docker Hub ;
3. Création d'un Dockerfile pour une application simple node js ;
4. Introduction à Docker Compose ;
5. Gestion de plusieurs images Docker avec docker-compose :
6. Démarrage et gestion des conteneurs (commandes docker-compose) ;

PARTIE 4

Terminologies Docker

Concepts clés de Docker :

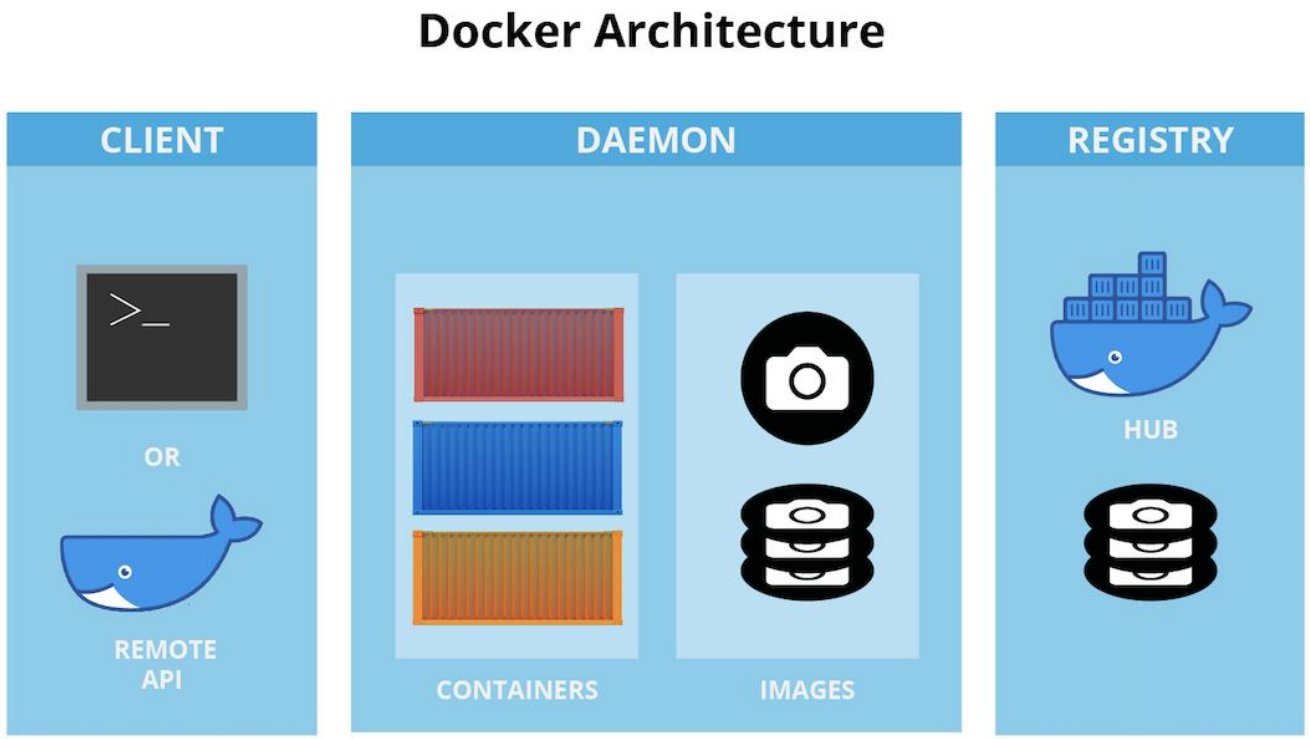
- Docker utilise une architecture **client-serveur** et se compose de :

- **Moteur Docker**

Il s'agit de l'application que vous installez sur votre ordinateur hôte pour créer, exécuter et gérer des conteneurs Docker. En tant que cœur du système Docker, il réunit tous les composants de la plate-forme en un seul endroit.

- **Docker Daemon**

Le serveur Docker (dockerd) écoute les requêtes de l'API Docker et gère les objets Docker tels que les images, les conteneurs, les réseaux et les volumes. Un serveur Docker peut également communiquer avec d'autres serveurs pour gérer les services Docker.



Terminologies Docker

- **Client Docker**

Il s'agit de l' interface utilisateur principale pour communiquer avec le système Docker.
Il accepte les commandes via l'interface de ligne de commande (CLI) et les envoie au démon Docker.

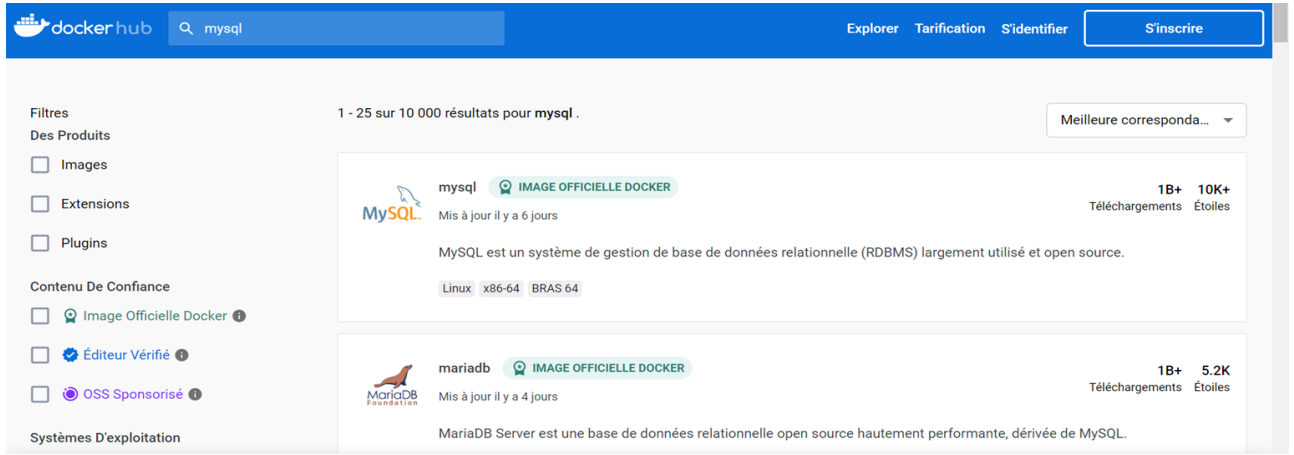
- **Registre Docker**

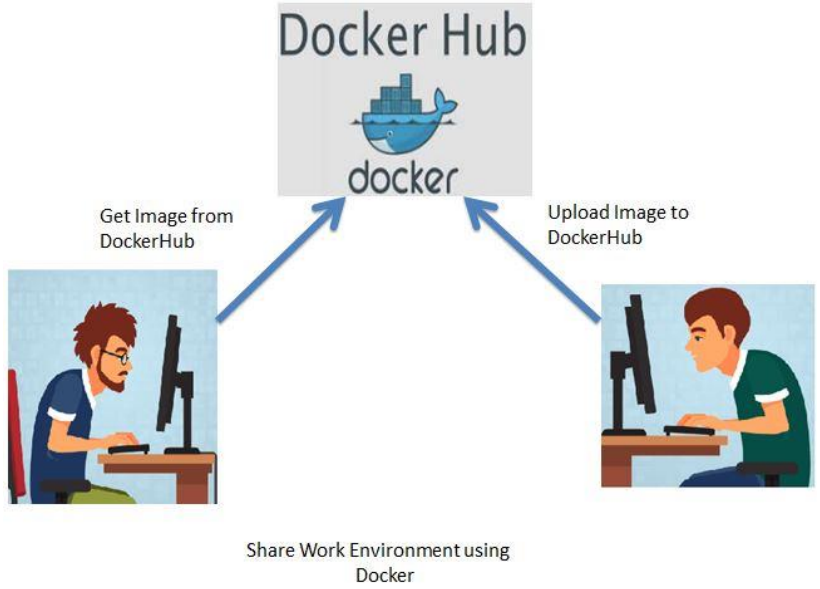
Un système de **catalogage** pour héberger, pousser et extraire des images Docker.
Vous pouvez utiliser votre propre registre local ou l'un des nombreux services de registre hébergés par des tiers (par exemple, Red Hat Quay, Amazon ECR, Google Container Registry et **Docker** Hub).
Un registre Docker organise les images dans des emplacements de stockage, appelés référentiels , où chaque référentiel contient différentes versions d'une image Docker qui partagent le même nom d'image.

PARTIE 4

Terminologies Docker

- **Docker Hub : https://hub.docker.com/**
  - C'est un registre public que n'importe qui peut utiliser
  - Docker est configuré pour rechercher des images sur Docker Hub par défaut.
  - Il est considéré comme la plus grande bibliothèque des images des conteneurs, permettant d'héberger environ 100 000 images.
  - On peut y chercher une image en tapant son nom dans la zone de recherche :

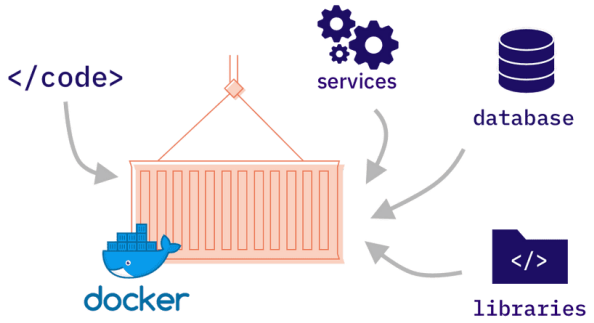




Terminologies Docker - Les Objets Docker

- **Les Objets Docker : Image Docker**

  Un modèle en lecture seule utilisé pour créer des conteneurs Docker.
  Il se compose d'une série de couches qui constituent un package tout-en-un , qui contient toutes les installations, dépendances, bibliothèques, processus et code d'application nécessaires pour créer un environnement de conteneur entièrement opérationnel.



- ✔ Souvent, une image est basée sur une autre image, avec quelques personnalisations supplémentaires.
- ✔ Vous pouvez créer vos propres images ou n'utiliser que celles créées par d'autres et publiées dans un registre.
- ✔ Pour construire votre propre image, vous pouvez créer un **Dockerfile** avec une syntaxe simple pour définir les étapes nécessaires pour créer l'image et l'exécuter.

Terminologies Docker - Les Objets Docker

- **Les Objets Docker : Image Docker**

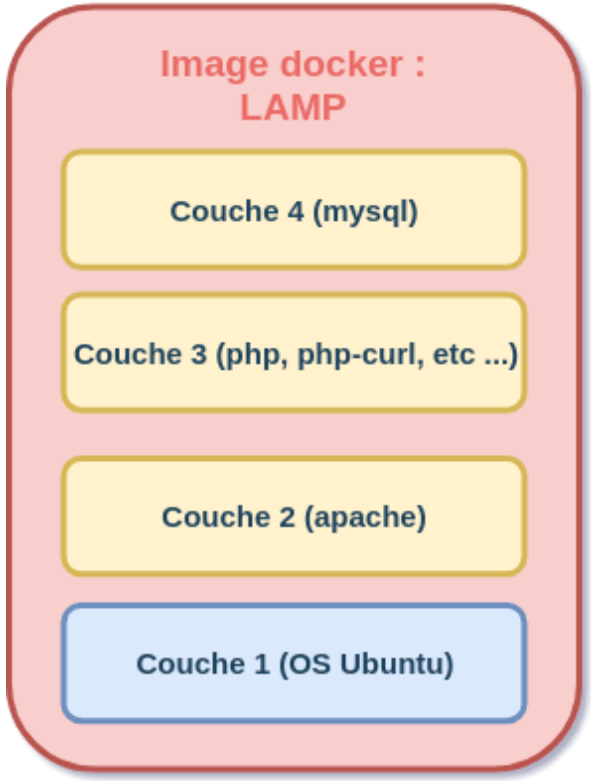
Exemple :

Imaginons par exemple qu'on souhaite déployer notre application web dans un serveur LAMP (Linux Apache MySQL PHP) au moyen de Docker.

Pour créer notre stack (pile en français), nous aurons besoin de :

- Une couche OS (système d'exploitation) pour exécuter notre Apache, MySQL et Php
- Une couche Apache pour démarrer notre serveur web et pourquoi pas la config qui va avec (.htaccess, apache2.conf, site-available/, etc ...)
- Une couche php qui contiendra un interpréteur Php mais aussi les bibliothèques qui vont avec (exemple : php-curl)
- Une couche Mysql qui contiendra notre système de gestion de bases de données Mysql

Au total, notre image docker sera composée de quatre couches, en schéma ceci nous donnerai :



Terminologies Docker - Les Objets Docker

- **Les Objets Docker : Image Docker**

Commandes de manipulation des images docker:

Commande	Signification
docker image ls / docker images	Lister les images Docker sur l'hôte local
docker pull <image>	Extraire (télécharger) l'**image Docker** depuis un registre (Docker hub par défaut)
docker push <image>	Envoyer (pousser) une **image** dans un registre
docker rmi <image>	Supprimer une **image** depuis l'hôte local

La manipulation des images docker est ainsi possible via l'application Docker Desktop.

Terminologies Docker - Les Objets Docker

- **Les Objets Docker : Conteneur Docker**

- Un conteneur est donc un espace dans lequel une application tourne avec son propre environnement.
- Il permet d'exécuter un microservice individuel ou une pile d'applications complète .
- Chaque conteneur est une instance d'**une image**. Il possède son propre environnement d'exécution et donc ses propres répertoires.
- l'API Docker ou la CLI(Command Line Interface) permettent de démarrer, arrêter ou supprimer un conteneur Docker.
- On peut connecter un conteneur à un ou plusieurs réseaux, y attacher un stockage ou même créer une nouvelle image en fonction de son état actuel.

Terminologies Docker - Les Objets Docker

- **Les Objets Docker : Conteneur Docker**

  Commandes de manipulation des conteneurs docker:

Commande	Signification
docker run --name <nom_conteneur> <nom_image>	crée et démarre un conteneur sur la base d'une image.
docker ps	liste les conteneurs actifs
docker stop <id_conteneur>	Arrête un conteneur
docker start <id_conteneur>	Démarrer un conteneur arrêté
docker rm <id_conteneur>	Supprime un conteneur
docker restart <id_conteneur>	Redémarre un conteneur

Manipuler un conteneur :

- **Les Objets Docker : Conteneur Docker**

Exemple :

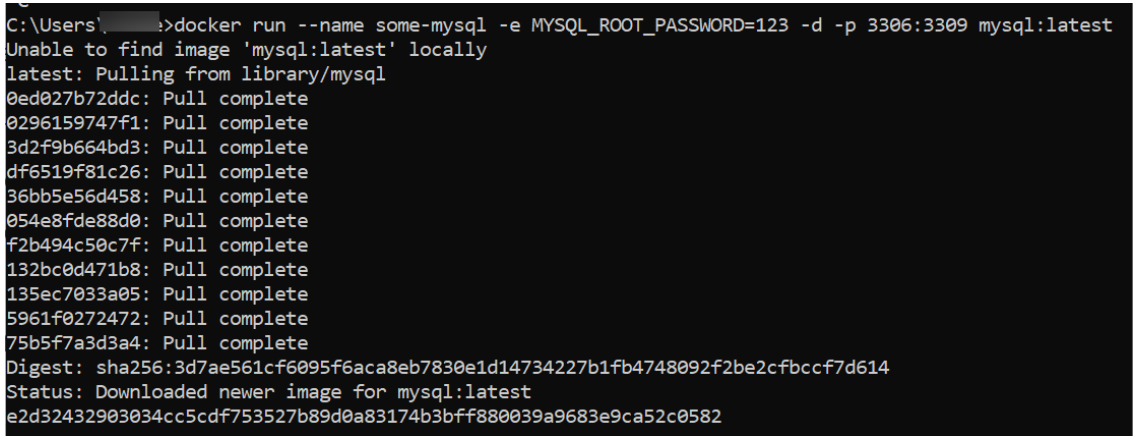
✔ Dans cet exemple, on souhaite créer et démarrer un conteneur en se basant sur une image mysql;

✔ Lançant cette commande :

```
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=123 –p 3306:3309 -d mysql:latest
```

Cette commande permet de :

- Démarrer un conteneur basé sur l'image ***mysql*** avec le tag ***latest*** (à télécharger depuis docker hub s'il ne le trouve pas sur la machine locale) (`mysql:latest`)
- De donner à notre conteneur le nom some-mysql (`--name some-mysql`)
- D'attribuer un mot de passe à l'utilisateur root ; (`-e MYSQL_ROOT_PASSWORD=123`)
- D'exposer publiquement le port 3309 du conteneur en tant que port 3306 (`–p 3306:3309`)
- De démarrer en mode détaché : Il s'exécute en arrière-plan du terminal (`–d`)



Manipuler un conteneur :

- **Les Objets Docker : Conteneur Docker**

Exemple :

On peut afficher les conteneurs démarrés via la commande : docker ps

```
C:\Users\      >docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS                                       NAMES
e2d324329030   mysql:latest   "docker-entrypoint.s…"   7 minutes ago   Up 7 minutes   3306/tcp, 33060/tcp, 0.0.0.0:3306->3309/tcp   some-mysql
```

On peut arrêter ce conteneur en récupérant son ID et en lançant la commande : **`docker stop e2d324329030`**

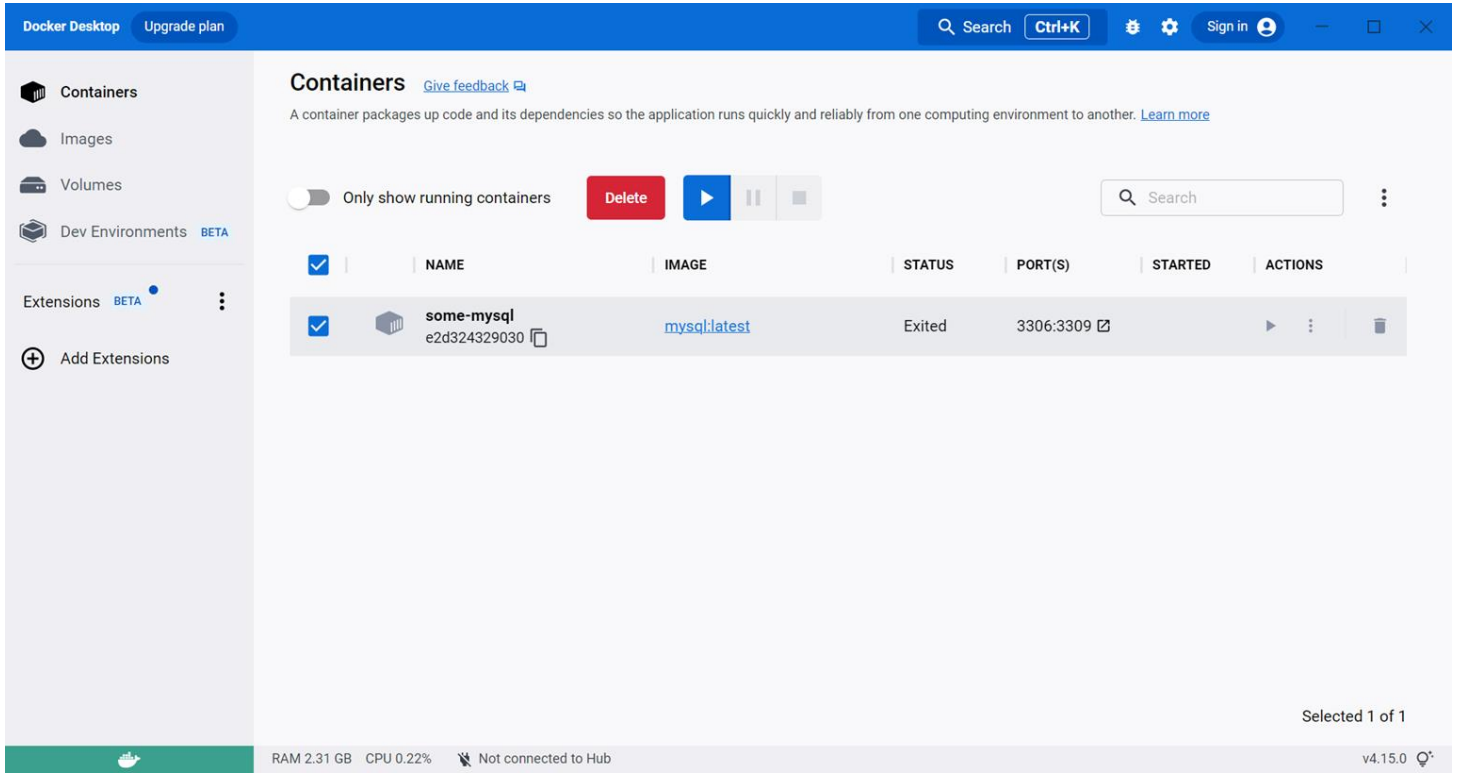
```
C:\Users\      >docker stop e2d324329030
e2d324329030
```

Manipuler un conteneur :

- **Les Objets Docker : Conteneur Docker**

Exemple :

Les conteneurs peuvent aussi être manipulés via Docker Desktop :



PARTIE 4

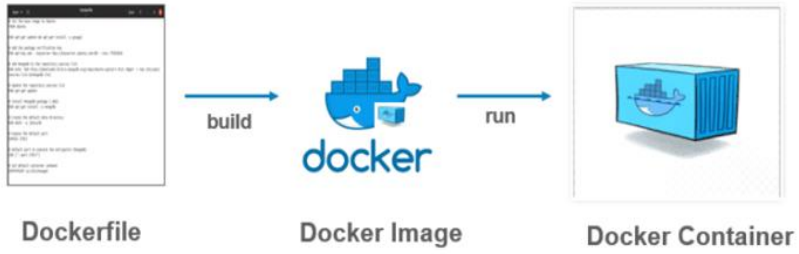
CHAPITRE 2

Prendre en main Docker

1. Installation de Docker Desktop ;
2. Terminologies Docker : images, Containers, Docker Hub ;
3. Création d’un Dockerfile pour une application simple node js ;
4. Introduction à Docker Compose ;
5. Gestion de plusieurs images Docker avec docker-compose :
6. Démarrage et gestion des conteneurs (commandes docker-compose) ;

Définition:

Un Dockerfile est un fichier qui liste les instructions à exécuter pour construire (**build**) une image.



Voici un exemple visuel de ce qu'un Dockerfile peut ressembler :

Il est lu de haut en bas au cours du processus de construction.

```
FROM node:17-alpine

WORKDIR /app

COPY package*.json .

RUN npm install --production

COPY . .

RUN npm run build

CMD ["node", "dist/index.js"]
```

Le **Dockerfile** permet de créer une image. Cette **image** contient la liste des instructions qu'un **conteneur** devra exécuter lorsqu'il sera créé à partir de cette même image.

PARTIE 4

Définition:

- Il faut savoir qu'une image nouvellement créée est toujours issue d'une image de base. On va juste ajouter des fonctionnalités supplémentaires, une application par exemple, afin qu'elle puisse correspondre à nos attentes.
- Une image docker est construite en exécutant la commande **docker build**. Cette dernière exécutera les lignes de commande se trouvant dans le fichier dockerfile.
- Voici la structure d'un ficher dockerfile:

```
# commentaire
INSTRUCTION arguments
```

Les instructions de base:

Instruction	Signification	Exemples
`FROM ImageName`	Sert à spécifier l'image de base que vous allez utiliser, image qui est présente sur Docker Hub	`FROM node:latest`

`RUN <command>`	- La commande **RUN** permet d'exécuter des commandes supplémentaires à l'intérieur du build du dockerfile.
	- L'argument qu'elle prend est identique à une commande Shell ordinaire.
	- On peut donc s'en servir afin de télécharger et d'installer les dépendances nécessaires à l'application ou encore à directement afficher un résultat ou un message.
		`- RUN mkdir /nvDossier`
		`RUN echo "hello world" > /nvDossier/greeting.txt`
		
		`- RUN npm install`

Définition:

Les instructions de base: https://docs.docker.com/engine/reference/builder/

Instruction	Signification	Exemples
COPY ou ADD	Permet de ***copier*** des fichiers depuis notre machine locale vers le conteneur Docker.	- ADD test.txt path/ - COPY . .
ENV	Variables d'environnements utilisables dans le Dockerfile et dans le conteneur.	ENV CONT_IMG_VER=v1.0.0
EXPOSE	Expose un port.	EXPOSE 3000
ENTRYPOINT	comme son nom l'indique, c'est **le point d'entrée** de votre conteneur, en d'autres termes, c'est la commande qui sera toujours exécutée au démarrage du conteneur.	
CMD	Spécifie les arguments qui seront envoyés au **ENTRYPOINT**	CMD ["npm", "start"]
WORKDIR	Définit le répertoire de travail qui sera utilisé pour le lancement des commandes CMD et/ou ENTRYPOINT et ça sera aussi le dossier courant lors du démarrage du conteneur.	WORKDIR /app

Définition – Exemple :

```
# Utilise l'image officielle Node.js 16 comme image parente
   FROM node:16
# Définit le répertoire de travail dans le conteneur sur /app
   WORKDIR /app
# Copie les fichiers package.json et package-lock.json dans le conteneur
   COPY package*.json ./
# Exécute npm install pour installer les dépendances dans le conteneur
   RUN npm install
# Copie le reste des fichiers de l'application dans le conteneur
   COPY . .
# Expose le port 3000, sur lequel l'application écoute
   EXPOSE 3000
# Démarre le serveur en exécutant la commande node index.js
   CMD ["node", "index.js"]
```

- On peut créer cette image en pointant sur le dossier contenant le fichier dockerfile et en exécutant la commande :

  docker build -t <image-name> .

- Pour démarrer le conteneur, il suffit de lancer :

  docker run -p 3000:3000 <image-name> -d

Docker Volume:

Le Docker Volume est l'emplacement où l'on stocke les données utilisées par les conteneurs et celles générées par Docker. Ces données persisteront sur les conteneurs qui y font appel. Cela veut dire que les volumes n'interfèrent pas dans le cycle de vie du conteneur. En plus, on peut les utiliser même si l'on exécute un conteneur Windows ou Linux.

Docker Nginx :

Nginx, que l'on prononce "engine-x", est un logiciel open source sous licence 2-clause BSD qui peut prendre plusieurs rôles. En effet, il peut être un serveur proxy inverse, un serveur web, un cache HTTP, un proxy de messagerie et un équilibreur de charge sur lequel plusieurs serveurs web s'appuient.

C'est en sa qualité de serveur web que Docker a décidé de l'intégrer dans sa plateforme, car il permet de développer des applications web et de les héberger, quel que soit le système d'exploitation utilisé.

On peut retrouver l'image Nginx sur Docker Hub si l'on souhaite l'utiliser. Il vous suffit d'effectuer un pull de cette image, de lancer le conteneur Nginx et de lancer votre application à partir de ce conteneur.

Comment nettoyer mon système:

Après avoir fait de nombreux tests sur votre ordinateur, vous pouvez avoir besoin de faire un peu de ménage. Pour cela, vous pouvez supprimer l'ensemble des ressources manuelles dans Docker.

Ou vous pouvez laisser faire Docker pour qu'il fasse lui-même le ménage. Voici la commande que vous devez utiliser pour faire le ménage :
docker system prune

Celle-ci va supprimer les données suivantes :

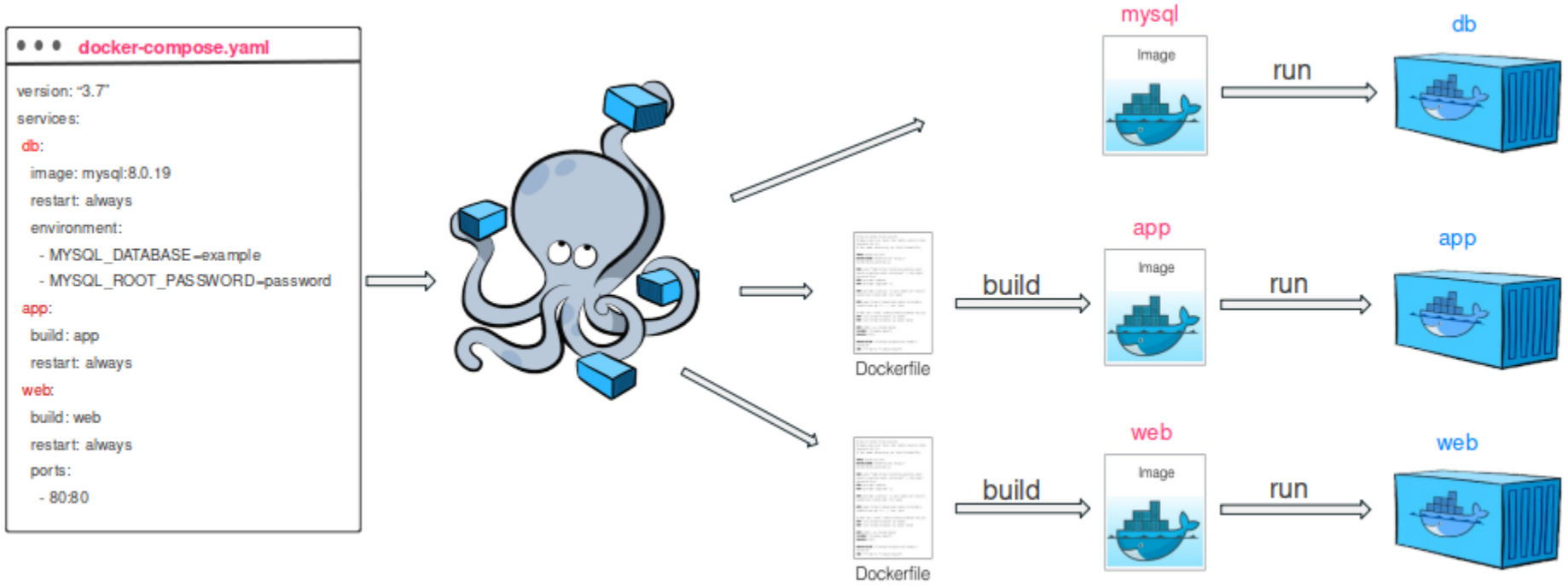
- l'ensemble des **conteneurs** Docker qui ne sont pas en status running ;
- l'ensemble des **réseaux** créés par Docker qui ne sont pas utilisés par au moins un conteneur ;
- l'ensemble des **images** Docker non utilisées ;
- l'ensemble des **caches** utilisés pour la création d'images Docker.

CHAPITRE 2

Prendre en main Docker

1. Installation de Docker Desktop ;
2. Terminologies Docker : images, Containers, Docker Hub ;
3. Création d'un Dockerfile pour une application simple node js ;
4. Introduction à Docker Compose ;
5. Gestion de plusieurs images Docker avec docker-compose :
6. Démarrage et gestion des conteneurs (commandes docker-compose) ;

- Compose est un outil permettant de définir et d'exécuter des applications Docker **multi-conteneurs**.
- Docker-compose fonctionne à partir d'un fichier **yml**, dans lequel on définit tous les services que l'on souhaite.
- Quand on lance la commande d'exécution, le Daemon Docker va lire le docker-compose.yml afin de monter chaque container avec les paramètres que l'on a choisi.



CHAPITRE 2

Prendre en main Docker

1. Installation de Docker Desktop ;
2. Terminologies Docker : images, Containers, Docker Hub ;
3. Création d'un Dockerfile pour une application simple node js ;
4. Introduction à Docker Compose ;
5. Gestion de plusieurs images Docker avec docker-compose ;
6. Démarrage et gestion des conteneurs (commandes docker-compose) ;

Le fichier de Docker Compose nommé *docker-compose.yml* a systématiquement 2 clefs à sa racine : **version** et **services**.

Exemple de squelette:

```
version: '3.9'

services:
  authentification:
    [...]
  commande:
    [...]
  produits:
    [...]
```

Dans ce fichier, on indique qu'on va utiliser la version 3.9 de docker-compose et on définit trois services: authentification, commande et produits.
La configuration de chacun des services sera ensuite décrites à l'intérieur de chacun des blocs.
Pour chaque service, différentes options peuvent être configurées dans le fichier Docker Compose.

La liste des configurations des services est fournie dans la documentation officielle, ici, on traitera quelques une:

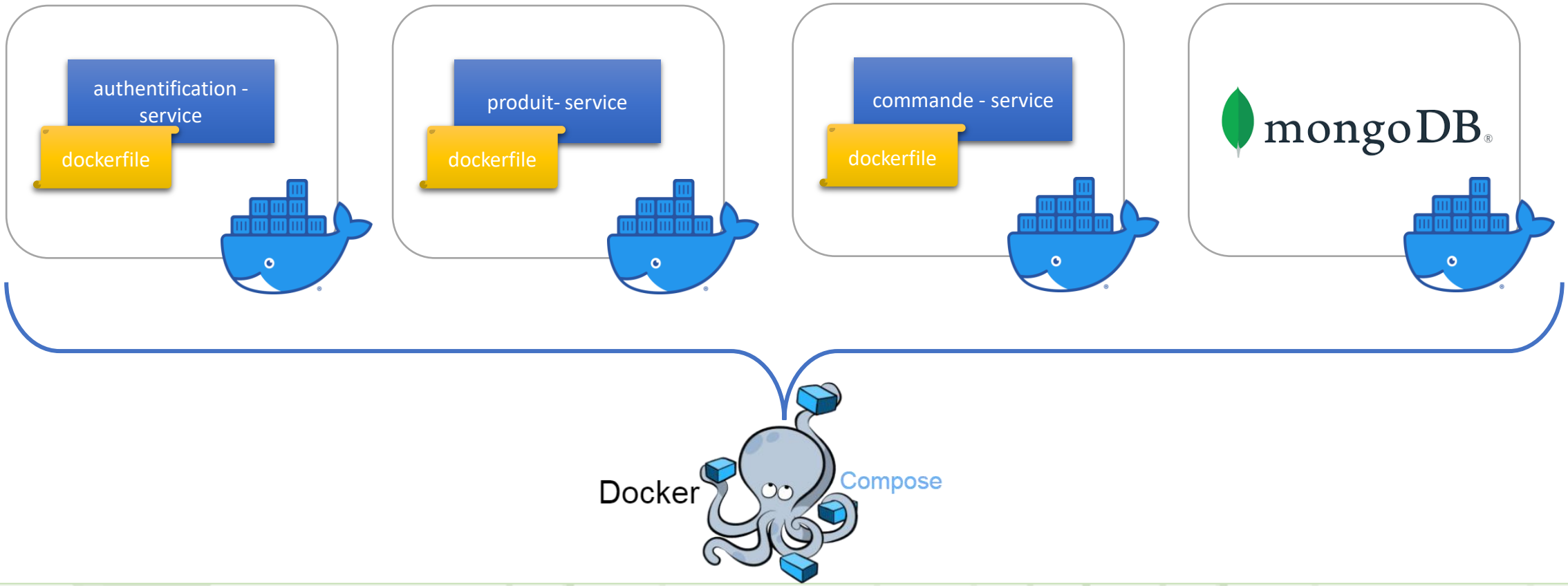
- **container_name** permet de nommer le conteneur
- **environment** est la liste des variables d'environnement à passer au conteneur
- **image** est le nom de l'image Docker à utiliser
- **build :** Permet de renseigner le Dockerfile sur lequel le conteneur se base. Au lancement du docker-compose, le Dockerfile va être build et transformé en une image qui va être utilisée par le conteneur.
- **ports** Permet de faire le mappage entre les ports à l'intérieur du conteneur et les ports qui leur correspondent à l'extérieur du conteneur, c'est-à-dire dans le système Docker.
- **volumes** est la liste des volumes que l'on souhaite monter dans le conteneur. Cette propriété permet d'utiliser un répertoire virtuel qui ne se supprime pas lorsqu'un conteneur est supprimé. C'est l'une des manières utilisées pour persister les données d'un conteneur.
- **links**: permet de lier des conteneurs sur un réseau. Lorsque nous lions des conteneurs, Docker crée des variables d'environnement et ajoute des conteneurs à la liste des hôtes connus afin qu'ils puissent se découvrir.
- **depends_on :** On y mentionne la liste des conteneurs dont un conteneur a besoin pour fonctionner. Le conteneur est démarré uniquement lorsque l'ensemble de ses dépendances le sont.

Exemple:

Retournons à notre exemple de cours où on gérait trois services: ***authentification***, ***produit*** et ***commande***.

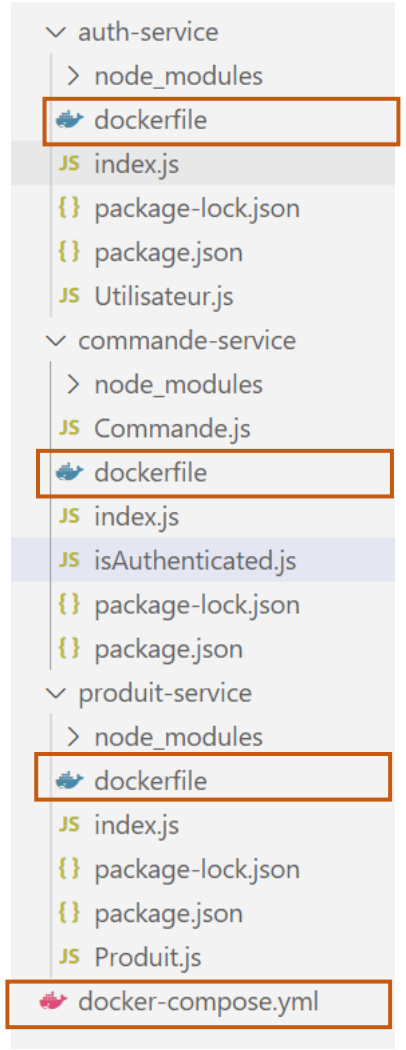
Chaque service aura son propre fichier dockerfile où on indiquera les instructions à suivre pour créer l'image docker correspondante.

En plus des trois conteneurs « service », on ajoutera un quatrième qui servira à conteneuriser le SGBD mongoDB.



PARTIE 4

Exemple:
La structure des fichiers aura l'allure suivante:



Les trois fichiers dockerfile auront le même contenu:

```
FROM node:latest

WORKDIR /app

COPY package*.json .

RUN npm install

COPY . .

CMD [ "npm", "run", "start" ]
```

Remarque : Dans quelques cas, on souhaite exclure quelques fichiers de l'image Docker à créer (fichiers volumineux, problème de sécurité …), on peut ajouter (à coté du fichier dockerfile) un autre fichier appelé **.dockerignore**.
Dans ce fichier, on cite les dossiers et/ou fichiers à exclure de l'image Docker.
Exemple du contenu du .dockerignore : node_modules

Exemple:
Le contenu du fichier docker-compose.yml est présenté ci-après :

```yaml
version: '3.9'
services:
  db:
    container_name: db
    image: mongo
    volumes:
      - ./data:/data/db
    ports:
      - "27017:27017"

  produits:
    build: ./produit-service
    container_name: produit-service
    ports:
      - "5000:4000"
    volumes:
      - /app/node_modules
    depends_on:
      - db

  authentification:
      build: ./auth-service
      container_name: auth-service
      ports:
        - "5002:4002"
      volumes:
        - /app/node_modules
      depends_on:
        - db

  commande:
    build: ./commande-service
    container_name: commande-service
    ports:
      - "5001:4001"
    volumes:
      - /app/node_modules
    depends_on:
      - authentification
      - produits
      - db
```

Exemple:

Explications

```
version: '3.9'

services:
  db:
    container_name: db
    image: mongo
    volumes:
      - ./data:/data/db
    ports:
      - "27017:27017"
```

Le nom du conteneur (un nom de votre choix)

Image « mongo » à télécharger depuis Docker Hub

Faire correspondre le dossier local **./data** au dossier **data/db** dans le conteneur
Ces dossiers sont une réplication de données de telle sorte que même si un conteneur est redémarré/supprimé à un moment donné, vous aurez toujours accès aux données générées par le conteneur.

Faire correspondre le port 27017 de la hôte au port 27017 du conteneur Docker;

Exemple:

Explications

```
produits:
  build: ./produit-service
  container_name: produit-service
  ports:
    - "5000:4000"
  depends_on:
    - db
```

L'image va être construite en se basant sur la définition fournie dans le fichier dockerfile; Ici, on indique son chemin

Le nom du conteneur (un nom de votre choix)

Faire correspondre le port 5000 de la machine hôte au port 4000 au sein du conteneur

Rappel: on a démarré ce service sur le port 4000 :

```
app.listen(4000, () => {
    console.log(`Product-Service at 4000`);
});
```

On indique que le conteneur « produits » a besoin du conteneur « db » pour qu'il fonctionne;

Pour se connecter à la base de données du service « db », on doit modifier les chaines de connexion de tous les services de :

"mongodb://localhost/produit-service" à "mongodb://**db:27017**/produit-service »

C'est le cas même pour le reste des appels: Pour qu'un service puisse appeler un autre, il faut remplacer, dans le code,

« **localhost:port** » par son « **nom_de_service : port** »

Exemple : URL = "http://**produits:4000**/produit/acheter"

CHAPITRE 2

Prendre en main Docker

1. Installation de Docker Desktop ;
2. Terminologies Docker : images, Containers, Docker Hub ;
3. Création d'un Dockerfile pour une application simple node js ;
4. Introduction à Docker Compose ;
5. Gestion de plusieurs images Docker avec docker-compose :
6. **Démarrage et gestion des conteneurs (commandes docker-compose) ;**

Voici une liste non exhaustive des commandes permettant de manipuler de docker compose:
(il suffit de se rendre dans le répertoire où se trouve le fichier docker-compose.yml et d'exécuter l'une des commandes suivantes)

Commande	Signification
docker-compose build	**Construit** les images définies dans docker-compose.yml
docker-compose up	**Construit** les images si elles ne le sont pas déjà, et **démarre** les conteneurs.
docker-compose up (-d) (--build)	-d : démarre en mode détaché (commande exécutée en arrière-plan du terminal) --build : construit les images avant le démarrage des conteneurs
docker-compose down	**Arrête** et **supprime** l'ensemble des conteneurs qui ont été instanciés par le docker-compose.
docker-compose stop	**Stop** (mais ne supprime pas) les conteneurs
docker-compose ps	**Affiche** tous les containers qui ont été lancés par docker-compose (qu'ils tournent actuellement ou non)
docker-compose rm	**Supprime** tous les conteneurs démarrés avec une commande docker-compose.
docker-compose config	**Valide** la syntaxe du fichier docker-compose.yml

Exemple:

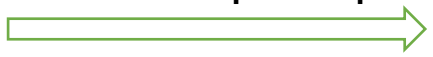
- Dans l ’exemple précédent, on a crée un fichier « docker-compose.yml » à la racine de notre projet;
- A l’aide des commandes : *docker-compose build* et *docker-compose up*, on peut construire les images et démarrer les conteneurs :

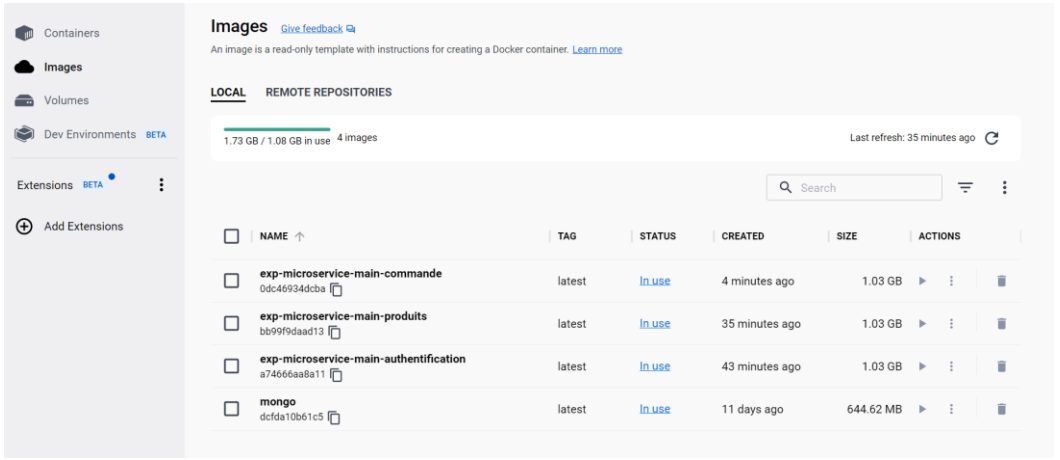
docker-compose.yml

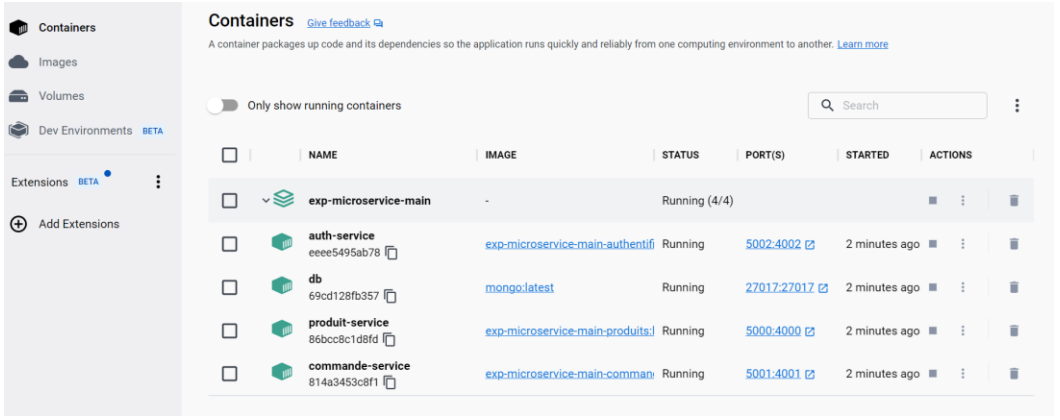
```yaml
version: '3.9'
services:
  db:
    container_name: db
    image: mongo
    volumes:
      - ./data:/data/db
    ports:
      - "27017:27017"

  produits:
    build: ./produit-service
    container_name: produit-service
    ports:
      - "5000:4000"
    volumes:
      - /app/node_modules
    depends_on:
      - db

  authentification:
      build: ./auth-service
      container_name: auth-service
      ports:
        - "5002:4002"
      volumes:
        - /app/node_modules
      depends_on:
        - db

  commande:
    build: ./commande-service
    container_name: commande-service
    ports:
      - "5001:4001"
    volumes:
      - /app/node_modules # Inside the container,
    depends_on:
      - authentification
      - produits
      - db
```

`docker-compose build`

`docker-compose up`



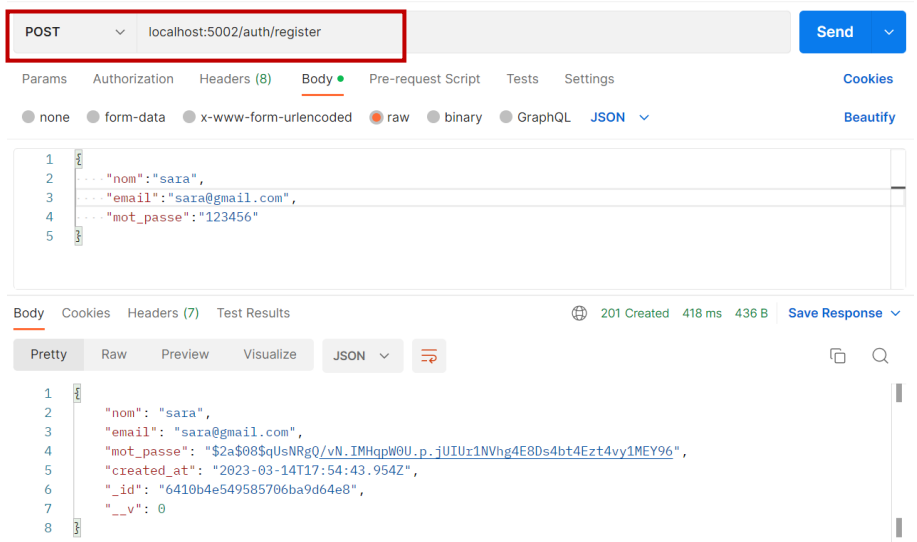




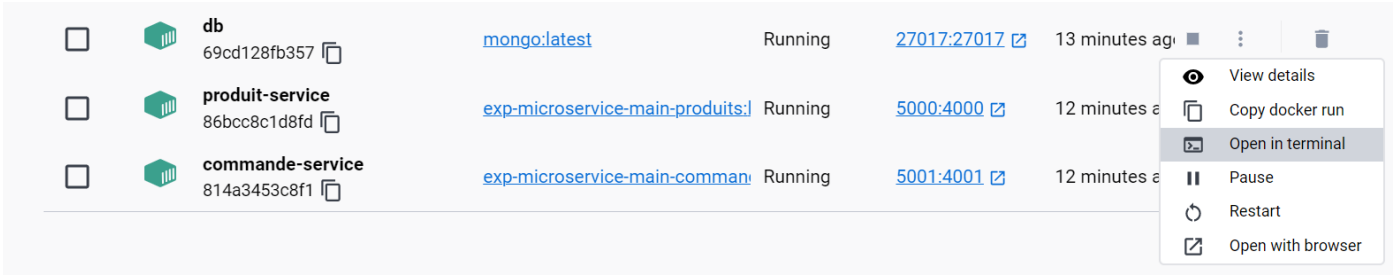
PARTIE 4

Exemple:

Pour tester nos conteneurs, on peut accéder au service « ***auth-service*** » via le port **5002** (comme indiqué dans le fichier docker-compose.yml)
Dans la capture ci-après, on teste l'enregistrement d'un nouvel utilisateur « sara ».



On peut consulter les données insérées dans le conteneur « db » en ouvrant son terminal (*sous Docker Desktop*) :



PARTIE 4

Exemple:

Tapons la commande ***mongosh*** (Shell permettant la manipulation du SGBD MongoDB)

- La commande « ***show databases*** » permet d'afficher la liste des collections disponible:

```
test> show databases
admin            40.00 KiB
auth-service      8.00 KiB
config           60.00 KiB
local            72.00 KiB
produit-service   8.00 KiB
```

- En pointant sur la collection « auth-service » et en affichant tous les utilisateurs ajoutés, on remarque bien que l'utilisateur « sara » a été bien enregistré:

```
test> use auth-service
switched to db auth-service
auth-service> db.utilisateurs.find()
[
  {
    _id: ObjectId("6410b4e549585706ba9d64e8"),
    nom: 'sara',
    email: 'sara@gmail.com',
    mot_passe: '$2a$08$qUsNRgQ/vN.IMHqpW0U.p.jUIUr1NVhg4E8Ds4bt4Ezt4vy1MEY96',
    created_at: ISODate("2023-03-14T17:54:43.954Z"),
    __v: 0
  }
]
```

PARTIE 4