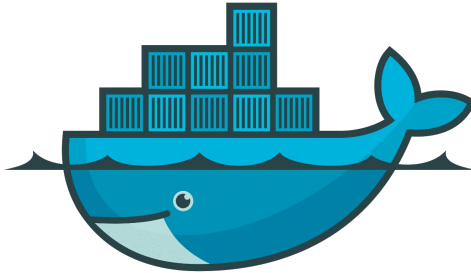


Docker



docker

Contenus techniques et objectifs de la formation

La formation aborde les contenus techniques suivants :

- Les principes de la virtualisation
- Fonctionnement de Docker
- Les conteneurs personnalisés
- Les applications multi conteneurs
- Les interfaces d'administration

Les objectifs visés par cette formation sont :

- Comprendre le fonctionnement de Docker et des conteneurs
- Utiliser l'interface en ligne de commande de Docker
- Déployer des applications dans des conteneurs
- Administrer des conteneurs

Table des matières

De la virtualisation à Docker

- Les différents types de virtualisation
- La conteneurisation : LXC, namespaces, control-groups
- Le positionnement de Docker
- Docker versus virtualisation

Présentation de Docker

- L'architecture de Docker
- Disponibilité et installation de Docker sur différentes plateformes (Windows et Linux)
- La ligne de commande et l'environnement

Mise en œuvre en ligne de commande

- Le Docker Hub : Ressources centralisées
- Mise en place d'un premier conteneur
- Mise en commun de stockage interconteneur
- Mise en commun de port TCP interconteneur
- Publication de ports réseau
- Le mode interactif

Création de conteneur personnalisé

- Produire l'image de l'état d'un conteneur
- Qu'est-ce qu'un fichier Dockerfile ?
- Automatiser la création d'une image
- Mise en œuvre d'un conteneur
- Conteneur hébergeant plusieurs services : supervisor

Mettre en œuvre une application multi-conteneur

- Utilisation Docker Compose
- Création d'un fichier YAML de configuration
- Déployer plusieurs conteneurs simultanément
- Lier tous les conteneurs de l'application

Interfaces d'administration

- Interface d'administration en mode Web
- Automatiser le démarrage des conteneurs au boot
- Gestion des logs des conteneurs

De la virtualisation à Docker



Les différents type de virtualisation

Il existe plusieurs types de virtualisation, chacun ayant ses propres avantages et inconvénients. Les principaux types sont :

- **La virtualisation complète** : Cette méthode permet d'exécuter plusieurs systèmes d'exploitation sur une même machine physique. Chaque système d'exploitation est isolé dans un environnement virtuel, ce qui lui permet d'avoir ses propres ressources et de fonctionner de manière indépendante des autres systèmes d'exploitation.
- **La virtualisation paravirtualisée** : Cette méthode permet aux systèmes d'exploitation invités de partager les ressources matérielles de la machine hôte. Les invités doivent être modifiés pour pouvoir fonctionner de manière efficace avec cette méthode de virtualisation.
- **La virtualisation basée sur les conteneurs** : Cette méthode permet d'isoler des processus ou des applications individuelles plutôt que des systèmes d'exploitation entiers. Elle utilise des fonctionnalités telles que **LXC**, **namespaces** et **control-groups** pour fournir un environnement **isolé** pour chaque conteneur.

Chacune de ces méthodes de virtualisation a ses avantages et ses inconvénients, et le choix dépendra des besoins spécifiques de l'utilisateur.

La conteneurisation : LXC, Namespace, Control-groups

- La **conteneurisation** est une méthode de **virtualisation** légère qui permet d'exécuter plusieurs instances isolées d'un système d'exploitation sur un même système hôte.
- LXC (**Linux Containers**) est une implémentation de la **conteneurisation** pour **Linux**, qui utilise des technologies comme les namespaces et les control-groups pour isoler les ressources et les processus.
- Les **namespaces** permettent de créer des environnements **isolés** pour les processus, les réseaux, les utilisateurs et autres ressources système.
- Les **control-groups** permettent de limiter et de contrôler les ressources utilisées par chaque conteneur, comme la mémoire, le CPU et le stockage.

Le positionnement de Docker

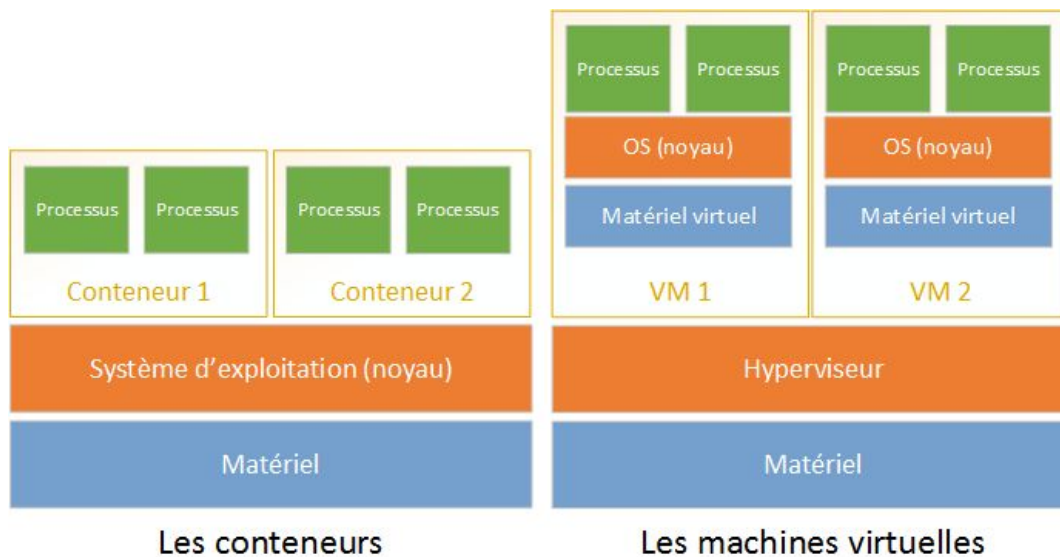
Docker est une solution de **conteneurisation** qui se situe entre la **virtualisation légère** et la **virtualisation traditionnelle**. Contrairement à la virtualisation traditionnelle, où chaque **machine virtuelle (VM)** nécessite son propre système d'exploitation, les conteneurs **Docker** partagent le même noyau **Linux** que celui de l'hôte.

Cela rend les conteneurs **Docker** plus légers et plus rapides à démarrer que les VM, tout en offrant une isolation et une sécurité similaires à celles des VM. Docker est donc une alternative efficace à la virtualisation traditionnelle, particulièrement adaptée pour les applications basées sur les microservices et les déploiements dans le cloud.

Le positionnement de Docker se situe donc entre la virtualisation légère et la virtualisation traditionnelle, offrant une solution de conteneurisation efficace et adaptée aux besoins actuels de développement et de déploiement d'applications.

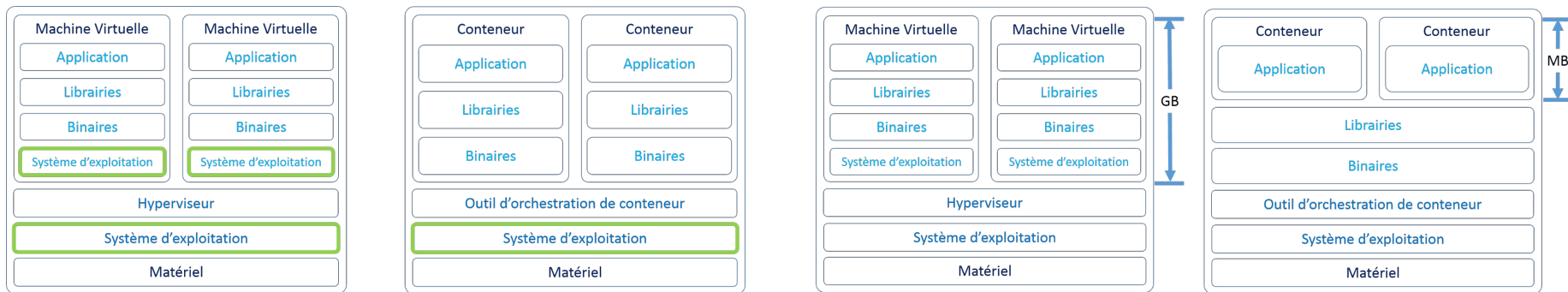
Docker VS Virtualisation

- La virtualisation consiste à émuler un environnement matériel complet pour chaque machine virtuelle, y compris le système d'exploitation.
- La conteneurisation, quant à elle, isole les processus de chaque application dans leur propre environnement, tout en partageant le noyau du système d'exploitation hôte.



Docker VS Virtualisation

- Les conteneurs sont plus légers et plus rapides à démarrer que les machines virtuelles.
- Les machines virtuelles offrent une isolation plus forte entre les applications.
- Docker peut fonctionner sur une infrastructure de virtualisation existante, offrant ainsi la possibilité de combiner les avantages des deux approches.



En fin de compte, le choix entre la virtualisation et la conteneurisation dépend des besoins spécifiques de chaque application.

Présentation de Docker

Docker est une plate-forme de virtualisation légère qui permet de créer et de gérer facilement des conteneurs d'application.

Contrairement aux machines virtuelles, les conteneurs **Docker** partagent le même système d'exploitation hôte, ce qui les rend plus légers et plus rapides à démarrer.

Docker est devenu un outil incontournable pour les développeurs et les administrateurs système car il facilite le déploiement d'applications dans des **environnements de production**, tout en réduisant les coûts et la complexité.

Dans cette partie, nous allons explorer l'architecture de **Docker**, comment l'installer sur différentes plates-formes et les commandes de base pour travailler avec **Docker**.

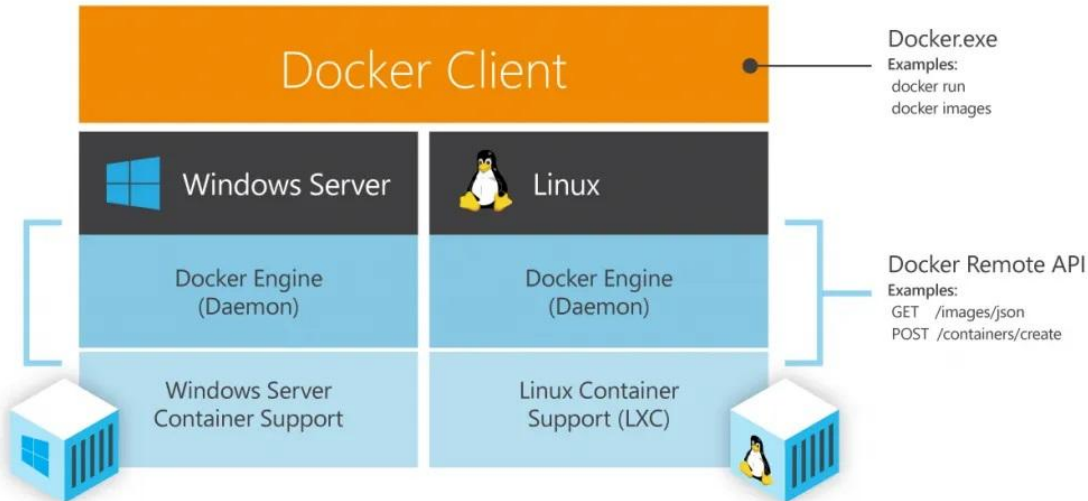
L'architecture de Docker

Docker est une plateforme de conteneurisation qui permet de créer, déployer et exécuter des applications dans des conteneurs **isolés** les uns des autres.

- **L'architecture** de Docker est basée sur un modèle client-serveur, avec un daemon Docker qui gère les opérations de conteneurisation sur l'hôte, et une API REST qui permet aux clients de communiquer avec le daemon.
- **Le daemon** Docker est responsable de la création, de l'exécution et de la gestion des conteneurs. Il utilise une technologie de conteneurisation de niveau système, qui permet de créer des environnements d'exécution isolés pour chaque conteneur.
- **Les images** Docker sont utilisées pour créer des conteneurs. Une image est un paquet léger qui contient tous les fichiers et les dépendances nécessaires pour exécuter une application. Elles sont écrites en Layers.
- **Les conteneurs** Docker sont des instances en cours d'exécution d'une image Docker. Chaque conteneur est isolé des autres conteneurs et de l'hôte, ce qui permet d'exécuter plusieurs instances d'une même application sur le même hôte sans risque de conflit.
- **Les volumes** Docker permettent de stocker des données persistantes en dehors des conteneurs. Cela permet de séparer les données de l'application et de les conserver même si le conteneur est supprimé ou recréé.
- **Les Networks Docker** permettent à des conteneurs de se connecter les uns aux autres, de s'isoler des autres conteneurs, ou de se connecter à des réseaux externes.

Installation Multi-plateforme : infos

- Docker est disponible sur différentes plateformes, notamment Linux, Windows et MacOS.
- Il peut être installé sur ces plateformes à l'aide de packages ou de programmes d'installation spécifiques.
- Les commandes de Docker sont les mêmes sur toutes les plateformes, ce qui facilite l'apprentissage et l'utilisation de Docker sur différentes machines.
- Docker est un outil très flexible et portable, qui peut fonctionner sur une grande variété de systèmes d'exploitation.

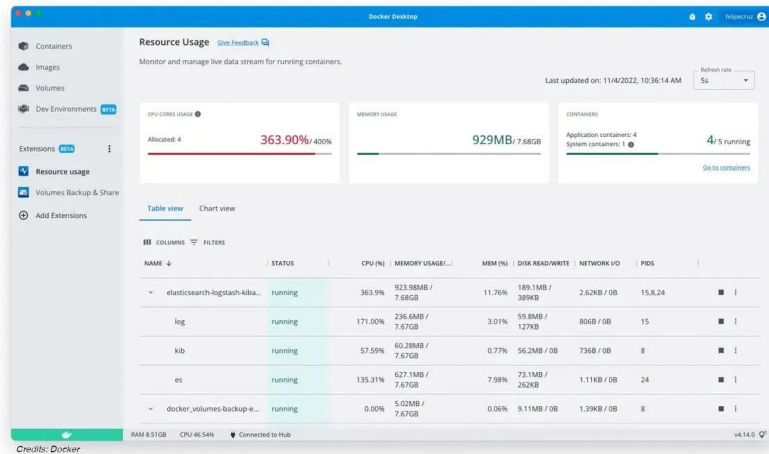


Installation Multi-plateforme : Windows

- Docker est compatible avec Windows depuis Windows 10 Pro et Enterprise, et Windows 11 en partie.
- Pour installer **Docker** sur **Windows**, téléchargez l'installateur à partir du site officiel de Docker
- L'installateur vous guidera tout au long du processus d'installation
- Pour vérifier que Docker est correctement installé, ouvrez une invite de commande et exécutez la commande "**docker --version**"

Une fois installé, Docker peut être utilisé à partir de la ligne de commande ou via une interface graphique comme Docker Desktop

Note : Il est important de vérifier que la version de Windows que vous utilisez est compatible avec Docker avant de l'installer.



Attention au WSL 2

WSL 2 (**Windows Subsystem for Linux 2**) est la version la plus récente de la plateforme de sous-système Windows pour Linux, qui permet d'exécuter un système d'exploitation Linux directement sur Windows.

Pour activer **WSL 2** sur Windows :

1. Vérifiez que vous disposez de la version 1903 ou ultérieure de Windows 10.
2. Ouvrez le menu Démarrer de Windows et recherchez "Fonctionnalités de Windows".
3. Cliquez sur "Activer ou désactiver des fonctionnalités de Windows".
4. Cochez la case "Sous-système Windows pour Linux" et cliquez sur "OK".
5. Redémarrez votre ordinateur.
6. Ouvrez Microsoft Store et recherchez "Linux" pour télécharger la distribution Linux de votre choix (par exemple, Ubuntu, Debian, etc.).
7. Lancez la distribution Linux installée et suivez les instructions pour configurer un compte utilisateur. (non obligatoire)

Une fois que vous avez installé une distribution Linux sur WSL 2, vous pouvez installer Docker pour Linux et l'utiliser normalement.

Installation Multi-plateforme : Linux

Docker est compatible avec de nombreuses distributions Linux, comme Ubuntu, Debian, CentOS, etc.

- L'installation de Docker sur Linux peut varier en fonction de la distribution choisie, mais en général, cela nécessite l'ajout du dépôt Docker à votre liste de sources de paquets.
- Ensuite, il suffit d'installer Docker à partir de ce dépôt à l'aide de la commande appropriée pour votre distribution.

Il est également possible d'installer Docker à l'aide de scripts d'installation fournis par Docker.

Une fois Docker installé, il est prêt à être utilisé pour exécuter des conteneurs sur votre système Linux.

Note : il est important de vérifier la compatibilité de votre distribution Linux avec Docker avant de l'installer, car certaines versions de noyau Linux ne sont pas compatibles avec Docker.

Les lignes de commandes et env

- Les lignes de commandes Docker permettent de gérer les images, les conteneurs et les réseaux.
 - Exemples de commandes Docker :
 - **docker build** : permet de construire une image à partir d'un fichier Dockerfile
 - **docker run** : permet de lancer un conteneur à partir d'une image
 - **docker ps** : permet de lister les conteneurs en cours d'exécution
 - **docker stop** : permet d'arrêter un conteneur en cours d'exécution
 - **docker network** : permet de gérer les réseaux Docker
 - docker rm
 - `docker container prune`

Note : Les commandes et variables d'environnement peuvent varier en fonction des versions et des plateformes Docker.

Mise en oeuvre en ligne de commande

Docker hub : ressources centralisées

- Docker Hub est un service centralisé de Docker pour partager, stocker et rechercher des images de conteneurs.
- Il est souvent comparé à GitHub, car il sert de référentiel de code source pour les images Docker.
- Il permet également de trouver des images de conteneurs officielles et communautaires pour de nombreuses applications et systèmes d'exploitation.
- Les utilisateurs peuvent également stocker leurs propres images sur Docker Hub pour les partager avec d'autres utilisateurs ou les déployer sur d'autres systèmes.
- Docker Hub offre également une fonctionnalité de gestion des versions pour les images, permettant aux utilisateurs de suivre les modifications et les mises à jour de leurs images de conteneurs.
- Attention, il vous faudra un compte Docker Hub pour toute interaction



Docker hub : ressources centralisées : Les Images

The screenshot shows the Docker Hub search results for the query 'node'. The interface includes a top navigation bar with the Docker Hub logo, a search bar containing 'node', and links to 'Explore', 'Repositories', 'Organizations', and 'Help'. A user profile for 'evengylbstorm' is visible in the top right corner. On the left side, there are filters for 'Products' (Images, Extensions, Plugins) and 'Trusted Content' (Docker Official Image, Verified Publisher, Sponsored OSS). The main content area displays two search results. The first result is for the 'node' image, which is a Docker Official Image. It has over 1 billion downloads and 10K+ stars. The description states that Node.js is a JavaScript-based platform for server-side and networking applications. The second result is for the 'mongo-express' image, also a Docker Official Image, with over 100 million downloads and 1.3K stars. It is described as a web-based MongoDB admin interface written with Node.js and express. Both results include a 'Pulls' section showing the number of pulls and a date range, along with a 'Learn more' link.

Filters

Products

- ☐ Images
- ☐ Extensions
- ☐ Plugins

Trusted Content

- ☐ Docker Official Image
- ☐ Verified Publisher
- ☐ Sponsored OSS

Operating Systems

- ☐ Linux
- ☐ Windows

1 - 25 of 10 000 results for **node**.

Best Match

node DOCKER OFFICIAL IMAGE · 1B+ · 10K+

Updated 4 days ago

Node.js is a JavaScript-based platform for server-side and networking applications.

Linux IBM Z 386 x86-64 ARM ARM 64 PowerPC 64 LE

Pulls: 10,119,687
Mar 20 to Mar 26

[Learn more](#)

mongo-express DOCKER OFFICIAL IMAGE · 100M+ · 1.3K

Updated 4 months ago

Web-based MongoDB admin interface, written with Node.js and express

Linux ARM 64 x86-64

Pulls: 260,687
Mar 20 to Mar 26

[Learn more](#)

Docker hub : ressources centralisées : Les tags



node

DOCKER OFFICIAL IMAGE

1B+ · 10K+

Node.js is a JavaScript-based platform for server-side and networking applications.

docker pull node



TAG

[gallium-bullseye-slim](#)

Last pushed 4 days ago by [dolanj](#)

DIGEST

[0b889cbf70af](#)

[2a060e67ed3e](#)

[ac2896b7018b](#)

+2 more...

OS/ARCH

linux/amd64

linux/arm/v7

linux/arm64/v8

VULNERABILITIES

0 H 0 M 23 L

0 H 0 M 23 L

0 H 0 M 23 L

COMPRESSED SIZE

66.23 MB

58.97 MB

65.17 MB

docker pull node:gallium-bullseye_



TAG

[gallium-bullseye](#)

Last pushed 4 days ago by [dolanj](#)

DIGEST

[ce2203388aec](#)

[411bc9bafed4d](#)

[f1d88674f526](#)

+2 more...

OS/ARCH

linux/amd64

linux/arm/v7

linux/arm64/v8

VULNERABILITIES

0 H 2 M 101 L

0 H 2 M 101 L

0 H 2 M 101 L

COMPRESSED SIZE

342.9 MB

302.57 MB

335.16 MB

docker pull node:gallium-bullseye_



Docker hub : ressources centralisées : Le pull de l'image

Le **Docker Hub** permet de centraliser les images de conteneurs et de les partager facilement. Pour télécharger une image depuis le Docker Hub, il suffit d'utiliser la commande "**docker pull**" suivi du nom de l'image et de sa **version**.

Cette commande va récupérer l'image depuis le Docker Hub et la stocker en local sur votre machine.

PS : il n'est pas nécessaire de savoir où l'image est stockée.

Les images sur le **Docker Hub** peuvent avoir différents tags qui correspondent à différentes versions de l'image.

Les tags sont généralement utilisés pour différencier les images en fonction de la version ou de la configuration. Par exemple, l'image "**nginx**" peut avoir des tags tels que "**latest**", "**1.23**", "**1.23-alpine**", etc.

Le tag "**latest**" correspond généralement à la version la plus récente de l'image, tandis que les autres tags correspondent à des versions spécifiques de l'image avec des configurations spécifiques.

Il est important de faire attention au choix du tag lors du téléchargement d'une image, car cela peut affecter la stabilité et les fonctionnalités de l'application que vous souhaitez exécuter dans le conteneur **Docker**.

- 1.23.4, mainline, 1, 1.23, latest, 1.23.4-bullseye, mainline-bullseye, 1-bullseye, 1.23-bullseye, bullseye
- 1.23.4-perl, mainline-perl, 1-perl, 1.23-perl, perl, 1.23.4-bullseye-perl, mainline-bullseye-perl, 1-bullseye-perl, 1.23-bullseye-perl, bullseye-perl
- 1.23.4-alpine, mainline-alpine, 1-alpine, 1.23-alpine, alpine, 1.23.4-alpine3.17, mainline-alpine3.17, 1-alpine3.17, 1.23-alpine3.17, alpine3.17

Docker hub : ressources centralisées : Le pull de l'image

Avec la commande **docker pull**, on va pouvoir aller directement chercher une image pré-faites de ses layers sur le docker hub ou autre repo configuré dans le système.

```
Sélection C:\Windows\System32\cmd.exe - docker pull nginx:latest

C:\>docker pull nginx:latest
latest: Pulling from library/nginx
f1f26f570256: Downloading [=====>] 7.154MB/31.41MB
7f7f30930c6b: Downloading [=====>] 8.936MB/25.58MB
2836b727df80: Download complete
e1eeb0f1c06b: Waiting
86b2457cc2b0: Waiting
9862f2ee2e8c: Waiting
```

```
Sélection C:\Windows\System32\cmd.exe

C:\>docker pull nginx:latest
latest: Pulling from library/nginx
f1f26f570256: Pull complete
7f7f30930c6b: Pull complete
2836b727df80: Pull complete
e1eeb0f1c06b: Pull complete
86b2457cc2b0: Pull complete
9862f2ee2e8c: Pull complete
Digest: sha256:2ab30d6ac53580a6db8b657abf0f68d75360ff5cc1670a85acb5bd85ba1b19c0
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest

C:\>
```

Ici le pull de l'image du micro serveur web **Nginx** dans sa version **:latest** pour les mises à jour de l'image, les différentes layers de l'image sont téléchargées et appliquées, le digest confirme sa conformité et sa provenance.

Docker hub : Le pull de l'image sur Docker Desktop

Pour l'exemple, le formateur a pris le temps de télécharger la version latest et la version 1.23-alpine. On peut voir clairement la différence entre les deux images qui nous permettent différentes choses. Nous ne nous concentrerons pas sur les différences entre les versions plus que ce qu'indiqué dans les slides ci-dessus.

A partir de Docker Desktop, on peut notamment lancer directement un container sur bas de l'image, supprimer cette image etc..

<input type="checkbox"/>	nginx 8e75cbc5b25c	1.23-alpine	Unused	8 days ago	40.99 MB			
<input type="checkbox"/>	nginx 080ed0ed8312	latest	Unused	9 days ago	142.12 MB			

Mise en place d'un premier container

Pour lancer un conteneur basé sur cette image.

docker run <nom-image>

Si vous souhaitez utiliser une version spécifique de l'image, vous pouvez préciser le tag correspondant, par exemple :

docker run <nom-image>:<tag>

- **-ti** : Cette option permet de démarrer un conteneur à partir d'une image et d'ouvrir une session interactive avec le conteneur.
- **-d** : Cette option permet de démarrer un conteneur en arrière-plan (détaché) à partir d'une image.
- **--name <nom-du-conteneur>** : Cette option permet de donner un nom personnalisé au conteneur lorsqu'il est créé.
- **-p <port_externe_local_pc>:<port_interne_container>** : Cette option permet de publier un port du conteneur sur un port de l'hôte.
- **-v <répertoire_local>:<répertoire_conteneur>** : Cette option permet de monter un répertoire local en tant que volume dans le conteneur.

Mise en place d'un premier container : les options

- **—env** : permet de définir une variable d'environnement pour le conteneur, par exemple : `docker run -e VAR_NAME=value image_name`
- **—restart**: cette option spécifie le comportement de redémarrage du conteneur en cas de plantage ou d'arrêt. Les options disponibles incluent **"no"** (ne jamais redémarrer), **"on-failure"** (redémarrer uniquement en cas d'erreur) et **"always"** (redémarrer toujours).
- **—rm**: cette option supprime automatiquement le conteneur après son arrêt. Cela peut être utile si vous ne voulez pas conserver les conteneurs arrêtés.
- **—network**: cette option spécifie le réseau Docker à utiliser pour le conteneur. Vous pouvez créer des réseaux personnalisés pour les conteneurs et spécifier des options telles que **"bridge"** (le réseau par défaut), **"host"** (utiliser le réseau de l'hôte) ou **"none"** (pas de réseau).
- **—env-file**: cette option spécifie un fichier contenant des variables d'environnement à définir dans le conteneur. Les variables sont définies dans le fichier sous la forme **"VAR_NAME=value"**. (un `.env` classique)

Mise en commun et publication de Port(s) TCP/IP

Docker permet de partager des ports TCP/IP entre le système hôte et le conteneur.

Cela permet d'exposer des services à l'extérieur du conteneur et de les rendre accessibles depuis le réseau.

Pour partager un port, il suffit d'utiliser l'option **-p** lors du lancement du conteneur et de spécifier le port à partager ainsi que le port sur lequel il sera accessible depuis le réseau.

Exemple : **docker run -p 3000:80 nginx**

Dans cet exemple, le port **80** du conteneur est partagé sur le port **3000** du système hôte. Le service nginx sera donc accessible à l'adresse IP du système hôte, sur le port **3000** .
donc ici sur le **localhost:3000**

(Voir plus loin lors du lancement de notre premier container nginx pour cette utilisation des ports)

Mise en commun et publication de Port(s) TCP/IP

Pour publier plusieurs ports, vous pouvez ajouter plusieurs options `-p` à la commande.

Il est également possible de spécifier un protocole spécifique, en utilisant la syntaxe suivante : **`-p <port hôte>:<port conteneur>/<protocole>`**

Par défaut, Docker publie les ports sur toutes les interfaces réseau de l'hôte. Pour spécifier une interface réseau spécifique, utilisez la syntaxe suivante : **`-p <adresse IP>:<port hôte>:<port conteneur>`**

Enfin, il est possible de mapper un port aléatoire de l'hôte vers un port spécifique du conteneur en utilisant la syntaxe suivante : **`-p <port conteneur>`** vous trouverez le port aléatoire en listant les containers

(Voir plus loin lors du lancement de notre premier container nginx pour cette utilisation des ports)

Le mode -TI “interactif”

Le mode interactif permet de travailler directement dans un conteneur Docker en utilisant un terminal de commande.

Il est possible d'utiliser la commande "**docker run -ti**" pour lancer un conteneur en mode interactif.

En utilisant le mode interactif, l'utilisateur peut **exécuter des commandes** et interagir **directement** avec le **système d'exploitation du conteneur**.

Le mode interactif est utile pour déboguer des problèmes ou effectuer des tests dans un environnement **isolé**.

Pour sortir du mode interactif, il suffit d'utiliser la commande "**exit**" ou "**ctrl+d**".

Le mode interactif peut également être combiné avec d'autres options de Docker pour une utilisation plus avancée.

Mise en place d'un premier container : Run

En suivant les directives ci dessus nous sommes en capacités de faire rouler une image nginx simple avec la ligne de commande suivante :

docker run -ti -d --name demonginx -p 3500:80 nginx:1.23-alpine

La commande fait démarrer un conteneur nommé demonginx avec libération de la console + accès en console à ce container, le faisant tourner en local sur le port 3500 connecté au port 80 du container, port par défaut de nginx

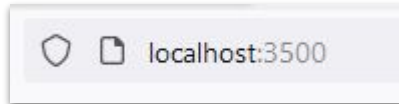
```
C:\>docker run -ti -d --name demonginx -p 3500:80 nginx:1.23-alpine  
b525048ddcf91b3f32833099f5161289cb19569d343d2dce48dff7c305eae957
```

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working.
Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

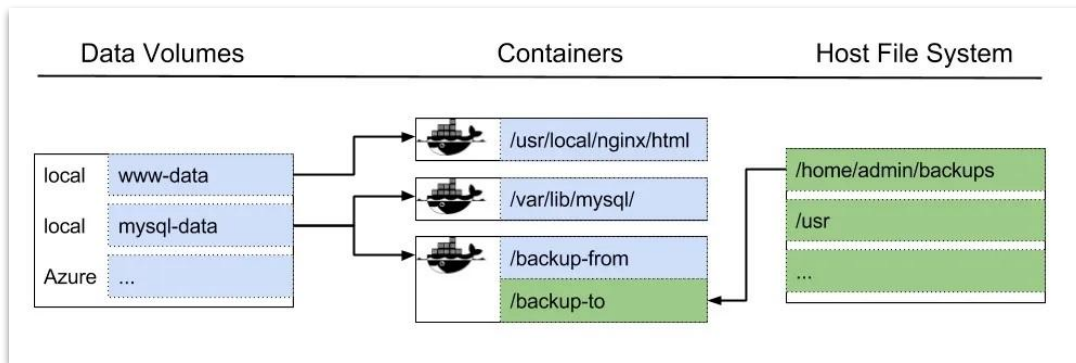


Mise en place de volume

Les **volumes Docker** sont utilisés pour persister les données d'un conteneur même après que celui-ci a été **supprimé**.

Les **volumes** sont des entités Docker **séparées** et **indépendantes** des conteneurs, ce qui signifie qu'ils peuvent être **réutilisés** entre plusieurs conteneurs.

Ils permettent également de séparer les données du conteneur de l'hôte, ce qui améliore la **portabilité** et la sécurité des données.



Mise en place de volume

Il existe deux types de volumes Docker :

- Les volumes **nommés** : ils sont créés explicitement en utilisant la commande `docker volume create` ou en les déclarant dans un fichier de configuration de **docker-compose.yml** (sera vu par la suite).
- Les volumes **anonymes** : ils sont créés automatiquement lorsqu'un conteneur en a besoin et qu'aucun volume nommé n'est disponible.

Les volumes Docker peuvent être montés sur un ou plusieurs conteneurs en même temps.

- Pour créer un volume nommé, utilisez la commande **docker volume create <nom_du_volume>**.

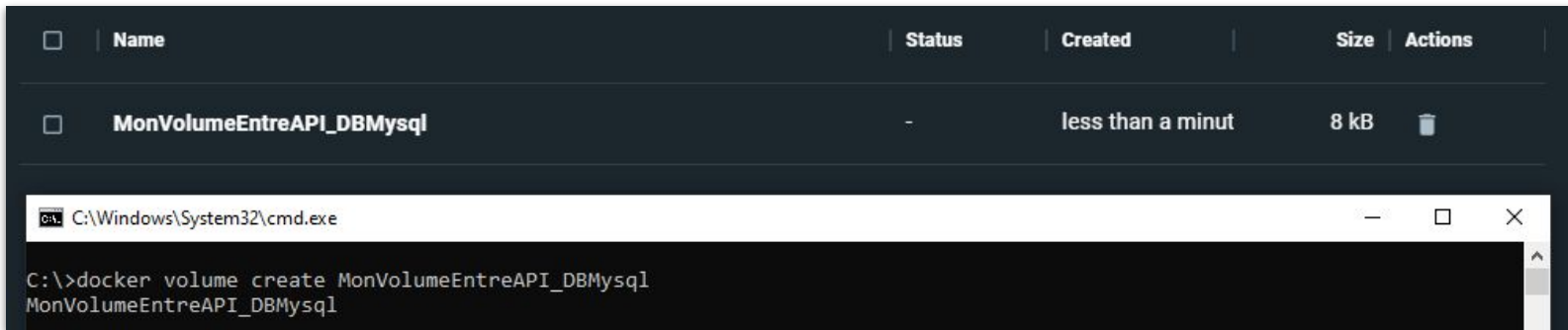
Pour monter un volume nommé sur un conteneur, utilisez l'option **-v** suivie du nom du volume et de son chemin de montage : **docker run -v <nom_du_volume>:<chemin_de_montage> <nom_image>**.

Pour monter un volume anonyme sur un conteneur, utilisez simplement l'option **-v** suivie du chemin de montage souhaité : **docker run -v <chemin_de_montage> <nom_image>**.

Mise en place de volume : Les commandes

- **docker volume create !nullable<nom_volume>**: crée un nouveau volume.
- **docker volume ls**: liste tous les volumes existants.
- **docker volume inspect !nullable<nom_volume>**: affiche les informations détaillées sur un volume.
- **docker volume rm !nullable<nom_volume>**: supprime un volume.
- **docker volume prune**: supprime tous les volumes inutilisés.

Exemples : **docker volume create MonVolumeEntreAPI_DBMysql**



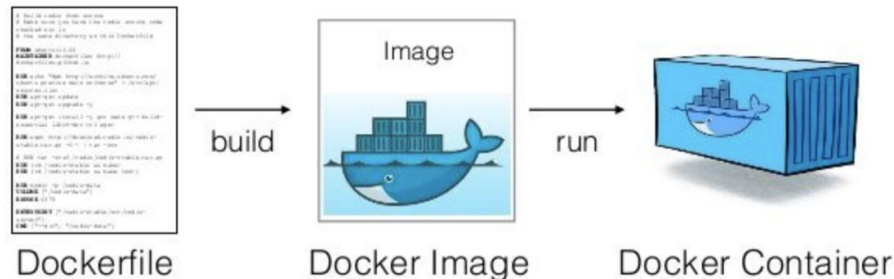
Création d'un container personnalisé

Produire l'image d'un container

Le **Dockerfile** est un fichier de configuration qui permet de définir les instructions nécessaires à la création d'une image Docker.

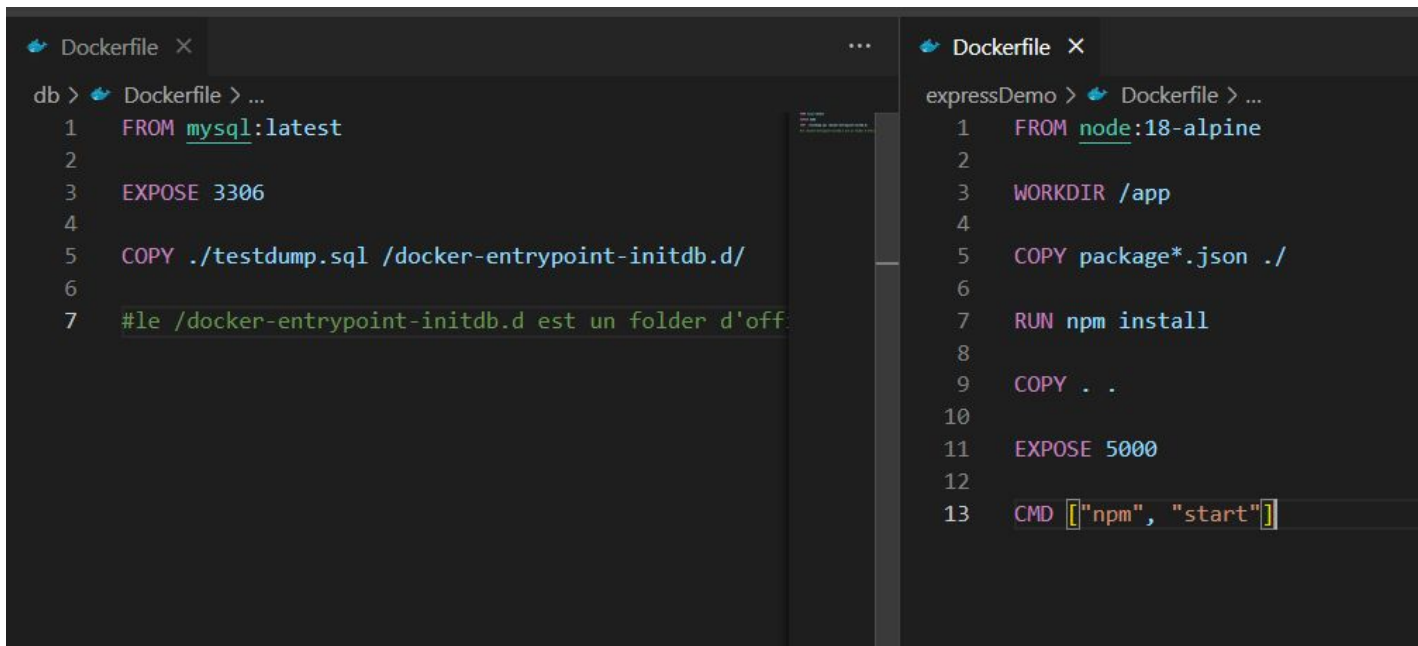
Il peut contenir des commandes d'installation de packages, de configuration de l'environnement, de copies de fichiers, etc.

Pour produire une image Docker à partir d'un **Dockerfile**, il suffit d'exécuter la commande "**docker build**" en spécifiant le chemin vers le Dockerfile et en lui donnant un nom et une version et toute les configurations que l'on souhaite apporter à sa création.



Le dockerfile

Explication au prochain slide.



```
db > Dockerfile > ...
1 FROM mysql:latest
2
3 EXPOSE 3306
4
5 COPY ./testdump.sql /docker-entrypoint-initdb.d/
6
7 #le /docker-entrypoint-initdb.d est un folder d'off

expressDemo > Dockerfile > ...
1 FROM node:18-alpine
2
3 WORKDIR /app
4
5 COPY package*.json ./
6
7 RUN npm install
8
9 COPY . .
10
11 EXPOSE 5000
12
13 CMD ["npm", "start"]
```

Le dockerfile : Les propriétés connues

Le premier dockerfile est utiliser pour mettre en place un container base sur mysql:latest

- **FROM** → permet de dire, base toi sur tel ou tel image déjà existante ou dl la.
- **EXPOSE** → permet de dire au container d'ouvrir son port sur le réseau extérieur.
- **COPY/ADD** → permet de copier un fichier ou folder vers le container, par exemple ./src ./src

Le deuxième dockerfile est utiliser pour mettre en place un container basé sur node:18-alpine

- **WORKDIR** permet de faire pointer le curseur console du container à tel endroit, ici /app, définit le répertoire de travail pour les instructions **RUN, CMD, ENTRYPOINT, COPY et ADD**
- **COPY/ADD (2ème)** → celui ci demande de copier tous les fichier avec ce pattern dans le folder courant DONC le /app
- **RUN** permet de lancer une commande shell, ici on install les dépendances à node
- **COPY/ADD . .** permet de tout copier du folder courant vers le folder courant du container
- **CMD []**, permet de renseigner la commande de démarrage lorsque le container est lancer , les paramètres de la commande sont dans un array classique
- **ENV** : définit une variable d'environnement
- **ENTRYPOINT** : spécifie la commande à exécuter lorsque le conteneur est démarré, en ajoutant des arguments possibles
- **VOLUME** : crée un point de montage pour un volume externe

Automatiser la création de l'image

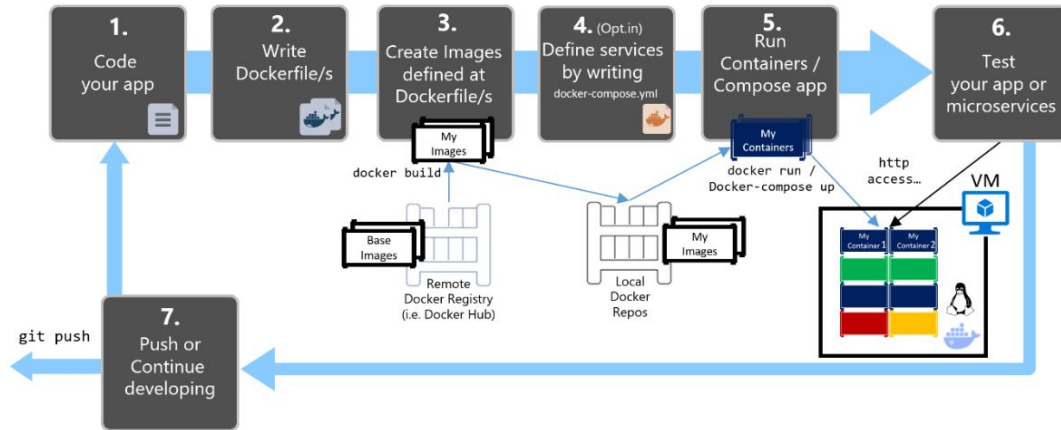
Pour créer notre image à partir de notre **Dockerfile**, nous utiliserons la commande **Docker build**. nous devons respecter certaines étapes et d'autres processus en dehors de docker pour automatiser la création des images, par exemple avec un **CRON** sous unix.

- Utilisation d'un **Dockerfile** pour décrire les étapes de création de l'image.
- Exécution de la commande **docker build** pour créer l'image à partir du **Dockerfile**.
- Utilisation de la fonctionnalité automatique de cache de Docker pour accélérer le processus de création d'image en évitant de répéter les étapes déjà effectuées.
- Utilisation de docker-compose pour automatiser la création et le déploiement de plusieurs conteneurs à la fois.

Automatiser la création de l'image

Un cycle de "dockerisation" existe, il faut au maximum suivre les étapes suivantes pour prévenir et maintenir un développement correct avec des projets dockeriser.

Inner-Loop development workflow for Docker apps



Montage d'un container (build de l'image)

La commande `docker build` permet de créer une image Docker à partir du fichier de configuration Dockerfile. Nous allons ensuite donner la syntaxe de base de la commande `docker build`, qui prend en compte plusieurs options.

Par exemple :

1. En spécifiant le chemin vers le Dockerfile :
 - a. **`docker build -t mon-image:latest /chemin/vers/Dockerfile`**
2. En utilisant une URL pour récupérer le Dockerfile :
 - a. **`docker build -t mon-image:latest https://github.com/mon-repo/mon-projet.git#chemin/vers/Dockerfile`**

Il existe de nombreuses autres options et variantes pour la commande `docker build`, en fonction des besoins spécifiques de l'utilisateur que nous détaillerons dans le slide suivant.

Montage d'un container (build de l'image)

Voici une liste des options de la commande docker build avec leurs définitions :

- **-t** : Nom et tag de l'image à construire.
- **-f** : Nom du fichier Dockerfile à utiliser.
- **--build-arg** : Définition des variables d'environnement utilisées lors de la construction de l'image.
- **--target** : Nom de la cible de la construction de l'image.
- **--cache-from** : Liste d'images à utiliser pour la mise en cache des étapes de construction.
- **--no-cache** : Ne pas utiliser le cache lors de la construction de l'image.
- **--pull** : Forcer la récupération de la dernière version de l'image de base depuis le registre.
- **--progress** : Contrôler la manière dont les informations de progression sont affichées.
- **--quiet** : Supprimer la sortie détaillée de la construction de l'image et ne montrer que les avertissements et erreurs.
- **--compress** : Compresser les couches intermédiaires lors de la construction de l'image.
- **--network** : Nom ou ID du réseau à utiliser pour la construction de l'image.
-

Il y a d'autres options plus avancées pour les cas d'utilisation spécifiques, mais celles-ci sont les plus courantes.

Montage d'un container (build de l'image) exemples

1. **`docker build -t myimage:latest .`** : Cette commande construit une image Docker nommée myimage à partir du Dockerfile situé dans le répertoire courant (.). L'image sera taguée avec la version latest.
2. **`docker build -t myimage:1.0 .`** : Cette commande construit une image Docker nommée myimage avec le tag 1.0 à partir du Dockerfile situé dans le répertoire courant (.).
3. **`docker build --build-arg MY_VAR=value -t myimage .`** : Cette commande construit une image Docker nommée myimage en utilisant la variable d'environnement MY_VAR définie dans le Dockerfile. La valeur de la variable sera définie à value.
4. **`docker build --no-cache -t myimage .`** : Cette commande construit une image Docker nommée myimage en forçant Docker à ne pas utiliser le cache de build. Cela permet de s'assurer que tous les paquets et dépendances sont téléchargés et installés à partir des sources les plus récentes.
5. **`docker build --file Dockerfile.prod -t myimage:prod .`** : Cette commande construit une image Docker nommée myimage en utilisant le fichier Dockerfile nommé Dockerfile.prod dans le répertoire courant (.). L'image sera taguée avec la version prod.

Ces exemples sont assez simples, mais ils illustrent les différentes options de la commande docker build que nous avons mentionnées précédemment.

Montage d'un container (build de l'image) exemples

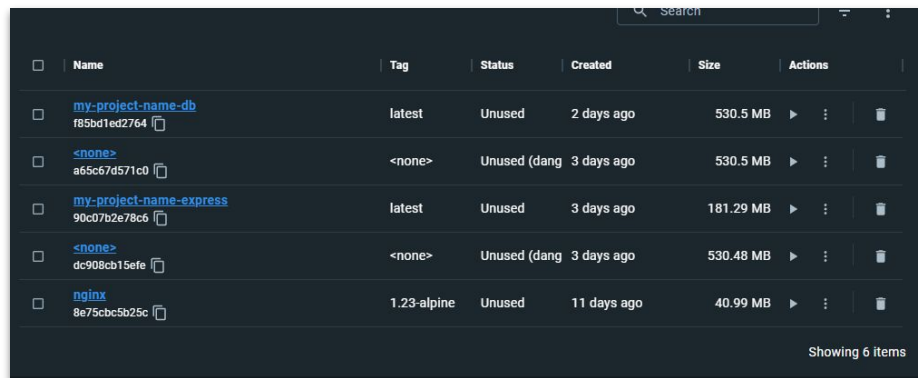
Nous avons créé notre première image personnalisée et nous avons vu les commandes liées au build de cette image.

Cette image se trouve dans les registres de Docker et n'a pas besoin d'être physiquement quelque part pour la retrouver, la commande `docker image (sans S !) ls` permet de lister les images disponibles sur le registre docker local.

Vous pouvez également les retrouver dans la partie images de docker desktop.

Différentes informations liées à l'image sont disponibles, les images avec `<none>` en nom, sont des images qui avaient un build en X layers qui a été mis à jour dans un dockerfile, et donc cette version là est présente mais non nommée et non taguée.

```
C:\>docker image ls
REPOSITORY          TAG          IMAGE ID       CREATED        SIZE
my-project-name-db   latest       f85bd1ed2764   2 days ago    531MB
<none>              <none>       a65c67d571c0   2 days ago    531MB
my-project-name-express latest       90c07b2e78c6   2 days ago    181MB
<none>              <none>       dc908cb15efe   2 days ago    530MB
nginx                1.23-alpine 8e75cbc5b25c   10 days ago   41MB
nginx                latest       080ed0ed8312   11 days ago   142MB
```



The screenshot shows the Docker Desktop interface with a search bar at the top. Below it is a table listing Docker images. Each row includes a checkbox, the image name (with a link icon), the tag, status, creation time, size, and action icons (expand, details, delete).

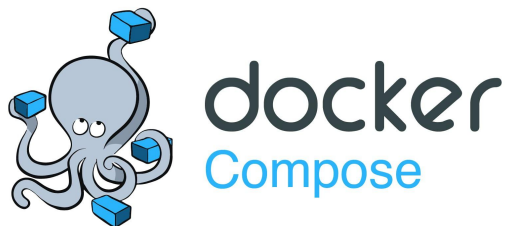
<input type="checkbox"/>	Name	Tag	Status	Created	Size	Actions
<input type="checkbox"/>	my-project-name-db f85bd1ed2764	latest	Unused	2 days ago	530.5 MB	▶ ⋮ 🗑
<input type="checkbox"/>	<none> a65c67d571c0	<none>	Unused (dang	3 days ago	530.5 MB	▶ ⋮ 🗑
<input type="checkbox"/>	my-project-name-express 90c07b2e78c6	latest	Unused	3 days ago	181.29 MB	▶ ⋮ 🗑
<input type="checkbox"/>	<none> dc908cb15efe	<none>	Unused (dang	3 days ago	530.48 MB	▶ ⋮ 🗑
<input type="checkbox"/>	nginx 8e75cbc5b25c	1.23-alpine	Unused	11 days ago	40.99 MB	▶ ⋮ 🗑

Showing 6 items

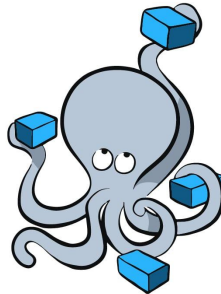
Container multi-services

- Les containers **Docker** peuvent être configurés pour exécuter **plusieurs services** simultanément.
- Cela permet de réduire le nombre de containers nécessaires et d'améliorer la modularité de l'architecture.
- Pour cela, il est possible d'utiliser des fichiers de configuration **YAML** pour définir les différents services et leurs paramètres.
- Les services peuvent communiquer entre eux à l'intérieur du container via des ports de communication définis dans la configuration.

Ce fichier s'appelle le **docker-compose.yml**, et sera vu plus en détails dans la suite de la formation.

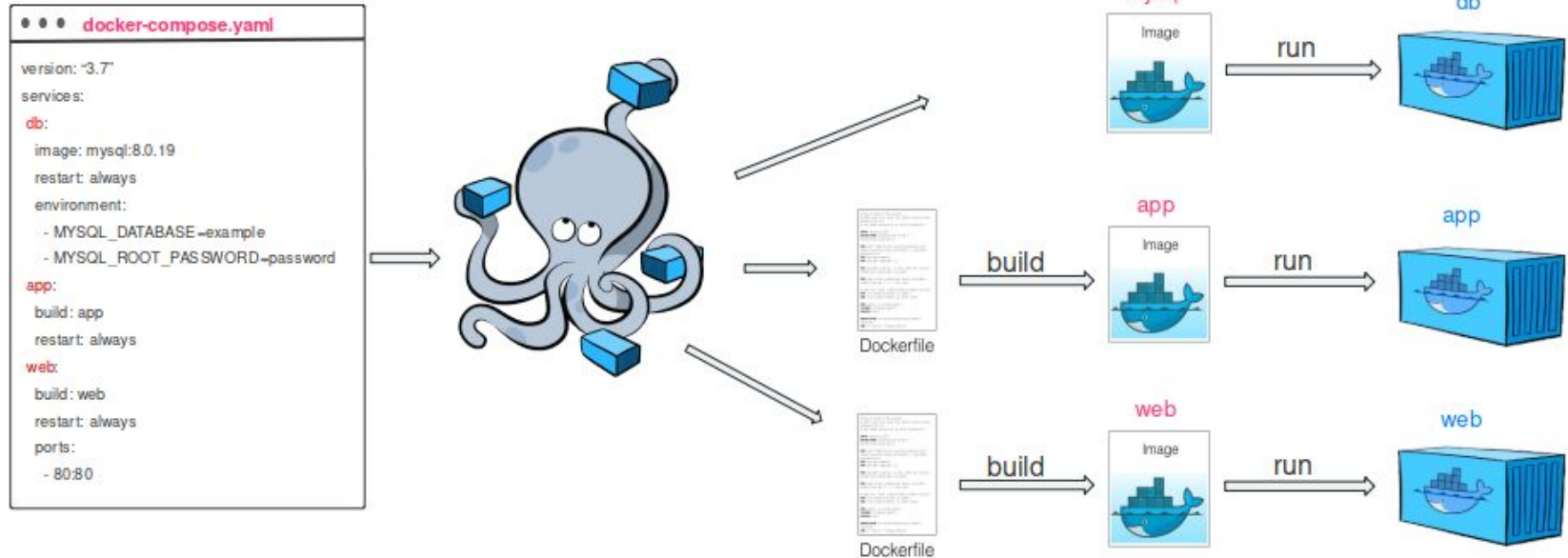


Mettre en oeuvre une appli multi containers



docker
Compose

L'orchestrateur de Multi - containers



Utilisation de Docker Compose pour orchestrer nos containers

Docker Compose est un outil magique qui permet de **décrire** et **d'orchestrer** des applications **multi-conteneurs**.

Il permet de définir les **services**, les **réseaux** et les **volumes** nécessaires pour que les conteneurs puissent communiquer et fonctionner **ensemble**.

La configuration de **Compose** se fait à travers un fichier **YAML** qui décrit les différents services et les dépendances entre eux. Une fois le fichier de configuration créé, il suffit de lancer la commande "**docker-compose up**" pour démarrer l'ensemble des conteneurs.

Docker Compose permet également de gérer les **variables d'environnement**, les **ports** exposés et les **volumes** partagés pour chaque service.

Il facilite ainsi la mise en place d'environnements de développement, de tests ou de production.

En résumé, **Docker Compose** permet de simplifier la gestion d'applications multi-conteneurs en automatisant leur déploiement et en facilitant leur configuration.

Création du fichier YAML pour Docker compose

Le fichier **YAML** de **Docker-compose** permet de décrire les différents **services** que nous souhaitons exécuter et comment les configurer.

Le fichier doit être nommé "docker-compose.yml" et doit se trouver à la racine du projet.

Le fichier est constitué de différents éléments :

- **La version de Docker compose** : cette version détermine les fonctionnalités que nous pouvons utiliser. Dans sa version actuelle, la version 3.8.
- **Les services** : chaque service est décrit dans une section avec son nom. Nous avons deux services dans cet exemple : "web" et "db". Pour chaque service, nous définissons l'image à utiliser, les ports à exposer, les variables d'environnement, etc.
- **Les réseaux** : nous pouvons créer des réseaux personnalisés pour nos services en utilisant la section "networks".
- **Les volumes** : nous pouvons également spécifier des volumes pour nos services en utilisant la section "volumes".

Une fois le fichier **YAML** créé, nous pouvons exécuter tous les services en utilisant la commande "**docker-compose up**".

Création du fichier YAML pour Docker compose

Pour décrire la suite des événements de la formation et spécifiquement sur la partie docker compose qui reste la partie la plus "complexe" sur docker, nous allons mettre en œuvre step by step, un cas d'app concrète.

Une **app back end avec sa db**, en micro services dédiés, donc un **composeur** pour **deux container** minimum.

le **back** sera un **expressJs** classique pour la facilité de mise en place, et la **db** sera une **mysql** classique également, tout deux étant monter sur un **dockerfile** chacun, l'app aura pour but de lister ce qu'il y a dans la db, et une url en get-post, (désolé API REST) entrera des datas dans la db de test.

Le tout étant plongé dans un **networks** qui contrôlera les **communications des deux containers** et un **volume** permettra d'isoler la **"Database"** en dehors des containers, ce qui permettra comme vous le savez maintenant, de ne pas **disparaître** lorsque le container est démonté ou fermé.

Création du fichier YAML pour Docker compose

Le fichier YAML de Docker Compose commence par la spécification de la version utilisée, qui est ici la version **3.8**. Ensuite, il définit deux services : "**db**" et "**express**".

Le service "**db**" est pour une base de données et nécessite une image définie dans le Dockerfile. L'image sera téléchargée depuis le hub Docker ou d'un registry privé si spécifié. Le port 3306 doit être ouvert et permettra de se connecter à la base de données.

Le service "**express**" correspond à une application Express.js. Ici, nous avons besoin de construire l'image à partir du Dockerfile et d'ouvrir le port 3000 pour accéder à l'application.

Le fichier **YAML de Docker Compose** se termine par la définition de **volumes** et de **réseaux**. Cela permet de partager des fichiers entre les services et de les connecter à des réseaux définis dans **Docker Compose**.

Ainsi, ce fichier **docker-compose.yaml** permet de déployer et d'orchestrer facilement deux services, une base de données et une application Express.js, en les connectant ensemble et en les reliant à des volumes et des réseaux spécifiques.

```
docker-compose-part1.yaml
1  version: "3.8"
2
3  services:
4    db:
5
6    express:
7
8  volumes:
9
10 networks:
11
```

Création du fichier YAML pour Docker compose

Pour faciliter le nommage nous prenons la décision de nommer chaque partie pour permettre de bien s'y retrouver lors des **lignes de commandes** ou sur **docker desktop**.

Le **Volumes** est déclaré, de deux manières différentes, la première, relie le service de la db a un dossier local, ici le dossier **./dumpDb** vers le container hôte de la db, dans la partie **/var/lib/mysql**.

Il n'est pas pris au hasard, c'est le folder qui contiendra nos **db** quand l'image **mysql** sera montée et que son **dockerfile** sera fait, c'est le chemin par défaut pour une installation classique **mysql** sous linux.

La deuxième façon de faire et de regrouper le **volume** dans une partie nommée **volumes** pour déclarer ensuite des nom de volumes, cela permet de pouvoir l'utiliser dans plusieurs services ou simplement de centraliser les différents volumes.

```
docker-compose-part1.yaml
1  version: "3.8"
2  name: my-project-name
3
4  services:
5    db:
6      container_name: db
7      volumes:
8        - ./dumpDb:/var/lib/mysql
9        # - db_volumes:/var/lib/mysql
10
11
12    express:
13      container_name: express
14
15
16  # volumes:
17  #   db_volumes:
18  #
19  networks:
20
```


Création du fichier YAML pour Docker compose

Ici nous appliquons des variables d'environnements, son spécifique à mysql et demande d'être créée, elles sont obligatoire pour créer un user dans la db lors de sa création et de set un password.

Les variables ici sont listées dans un .env, voir slide suivant.

Du côté express, nous aurons besoin d'une connexion à la db, donc on lui donne également des env, qui permettront de ne pas dépendre de variable statique et donc la manipulations de différents environnements de travail.

Attention que pour lire un .env file, il faudra ajouter cette ligne pour chaque services



```
23 MYSQL_PASSWORD: $APP_MYSQL_PASSWORD
24 MYSQL_HOST: $APP_MYSQL_HOST
25 ✓ env_file:
26   - ./env
27
```

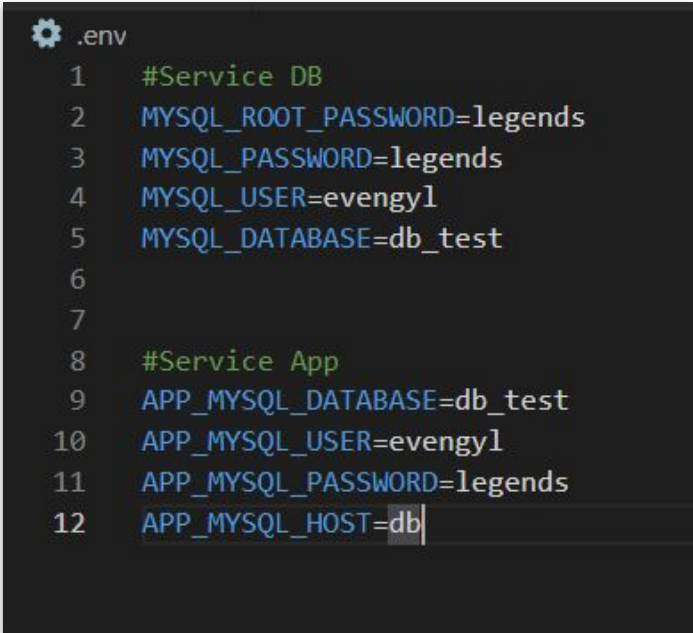
```
docker-compose-part1.yaml
1  version: "3.8"
2  name: my-project-name
3
4  services:
5    db:
6      container_name: db
7      volumes:
8        - ./dumpDb:/var/lib/mysql
9        # - db_volumes:/var/lib/mysql
10     environment:
11       MYSQL_ROOT_PASSWORD: $MYSQL_ROOT_PASSWORD
12       MYSQL_PASSWORD: $MYSQL_PASSWORD
13       MYSQL_USER: $MYSQL_USER
14       MYSQL_DATABASE: $MYSQL_DATABASE
15
16     express:
17       container_name: express
18       environment:
19         MYSQL_DATABASE: $APP_MYSQL_DATABASE
20         MYSQL_USER: $APP_MYSQL_USER
21         MYSQL_PASSWORD: $APP_MYSQL_PASSWORD
22         MYSQL_HOST: $APP_MYSQL_HOST
23
24 # volumes:
25 #   db_volumes:
26
```

Création du fichier YAML pour Docker compose

Du grand classique...

Le fichier est placé au même endroit que le docker-compose, donc en racine de notre projet.

Voyez ici la ligne 12 en surbrillance, nous lions notre **host app mysql**, qui correspondrait à un **127.0.0.1** ou **localhost** classique d'habitude, mais ici nous utilisons une fonctionnalité de **docker compose**, le service "**db**", recevra une **adresse IP** depuis **docker compose** et cette adresse sera directement utilisée comme **source** de connexion à la **db**, nous pourrons le voir dans le logs du container app express



```
.env
1  #Service DB
2  MYSQL_ROOT_PASSWORD=legends
3  MYSQL_PASSWORD=legends
4  MYSQL_USER=evengyl
5  MYSQL_DATABASE=db_test
6
7
8  #Service App
9  APP_MYSQL_DATABASE=db_test
10 APP_MYSQL_USER=evengyl
11 APP_MYSQL_PASSWORD=legends
12 APP_MYSQL_HOST=db
```

Création du fichier YAML pour Docker compose

Pour cette “dernière partie” de notre **docker compose**, nous allons omettre tout le reste pour gagner en place sur le slide, ici nous allons ajouter le lien de **construction**, comme nous allons préparer proprement deux **dockerfile** pour les **deux services**.

nous indiquons la route relative vers les deux **dockerfile** à monter, attention que l'on peut tout à fait, déjà mettre ici un nom **d'image**, **docker compose** ne saura pas faire la différence entre un lien et un nom d'image, c'est pour cela que pour une **image** déjà faite, on utilisera plutôt la commande “**image** : ”

Ensuite nous aimerions demander au container **d'automatiser** ses **plantages** occasionnels en lui demandant pour l'un : de se **restart** tout le temps peu importe ce qu'il se passe, et pour l'autre de restart uniquement lorsqu'il plante. nous dirons également que le container **express** sera lié avec le **container db**, ce qui aura pour effet que **docker** ne lancera pas le **container express** avant que le **container db** ne soit lancé, nous évitons donc les **erreurs** de **connexion** **DB** grossière et donc un plantage assuré du service pour rien.

Il est également à noter que les ports sont décrits ici, à différencier de la partie **EXPOSE** des fichiers **dockerfile** qui expose un port intrinsèquement lié à notre **applications interne**, ici les **ports** sont le **bridge** mis en place par **docker**, entre celui du **container** et celui de la machine **hôte**, **5000:5000** signifie de lié le port **5000** de **l'hôte** au port **5000** de notre **app**, celui défini dans notre application **express**.

il est à noter que nous n'exposons pas le port de la DB, car il est doit rester isolé de notre hôte, il ne doit communiquer qu'avec le container express (expliquer dans le slide juste au dessus au niveaux de l'hôte app mysql)

Ps : voyez ici les deux façon de passer à la ligne dans un fichier YAML.

```
docker-compose-part1.yaml
1  version: "3.8"
2  name: my-project-name
3
4  services:
5    db:
6      container_name:
7        - db
8      build:
9        - ./db
10     restart:
11       - always
12
13
14     express:
15       build: ./expressDemo
16       ports: "5000:5000"
17       depends_on: db
18       restart: on-failure
19
20
21  volumes:
22    db_volumes:
23
24  networks:
25
```

Création du fichier YAML pour Docker compose

```
version: "3.8"
name: my-project-name
services:
  db:
    container_name: db
    build: ./db
    env_file:
      - ./env
    environment:
      MYSQL_ROOT_PASSWORD: $MYSQL_ROOT_PASSWORD
      MYSQL_PASSWORD: $MYSQL_PASSWORD
      MYSQL_USER: $MYSQL_USER
      MYSQL_DATABASE: $MYSQL_DATABASE
    restart: always
    volumes:
      - ./dumpDb:/var/lib/mysql
      # - db_volumes:/var/lib/mysql
```

```
express:
  build: ./expressDemo
  container_name: express
  env_file:
    - ./env
  environment:
    MYSQL_DATABASE: $APP_MYSQL_DATABASE
    MYSQL_USER: $APP_MYSQL_USER
    MYSQL_PASSWORD: $APP_MYSQL_PASSWORD
    MYSQL_HOST: $APP_MYSQL_HOST
  ports:
    - "5000:5000"
  depends_on:
    - db
  restart: on-failure

# volumes:
#   db_volumes:
```

Les networks

Les **networks** de **Docker** permettent aux **containers** de communiquer entre eux et avec le **monde extérieur** en utilisant un réseau privé isolé.

Chaque container connecté à un **network Docker** obtient une **adresse IP unique** et peut communiquer avec d'autres containers connectés au même **network**, sans avoir besoin de publier des ports ou d'exposer des **endpoints** à l'extérieur.

Dans **Docker Compose**, la création et la gestion des **networks** sont également facilitées.

Dans le fichier **YAML** de configuration, les réseaux sont définis dans la section "**networks**".

Les containers peuvent alors être attachés à ces réseaux en utilisant la clé "**networks**" dans la définition de chaque service.

Les **networks Docker** sont également personnalisables en termes de configuration **d'adressage IP**, de **masquage réseau**, de **routing**, etc.

Cela permet de créer des architectures plus complexes et de séparer les différentes parties d'une application en différents réseaux pour une meilleure isolation et une meilleure sécurité.

Les networks

De base **docker compose** va construire un **networks** pour nos **container** qui permettra le lien entre la db et l'api, vu l'option **host:db** que nous avons écrits.

il va créer un **network** qui ressemblera à celui ci : (avec la commande **docker network ls**)

```
fca47d1f2553    my-project-name_default    bridge    local
```

Avec un coup de **docker inspect my-project-name_default** (voir le slide suivant pour les infos), nous pourrons voir les différentes configuration qu'il aura créé pour nous, donc le **driver**, les **ip** configurées dedans avec les **sous-réseaux** et **passerelle** ainsi que tous les **container** lié à ce réseau.

Par défaut **docker-compose** le crée automatiquement, c'est pour ça que nous n'avons pas dû le créer nous même, et c'est également pour ça que la **db** et l'**api** sont "**magiquement**" liée entre elles, justement, avec ce **Network** automatique. En fait, il attribue à chaque services une Ip, vue sur le slide suivant, et docker-compose sait que ce nom de service sera en fait aliased avec une Ip, et que donc ça transformera, lors de la lecture des ENV ceci :

```
APP_MYSQL_HOST=172.19.0.2/16
```

```
APP_MYSQL_HOST=db
```

En cela

Les networks

On peut voir ici les différents **containers** lancés et les adresses **Mac** et **IP** liées à chaque container. c'est à partir de là que compose va pouvoir lié et isolé donc, les différents **services** créés.

On peut le voir par exemple en lançant une commande d'exécution dans un container comme l'api :

docker exec -ti apiRest sh et que de la on lance un ping sur juste "db" il connaîtra d'office l'ip liée !

ici la **0.2** comme décrit dans le **docker inspect my-project-name_default**

```
"Containers": {  
  "1d295b579bb67c700d035885a533f5dfc9c29c9afb1b049d4836208dc0aba566": {  
    "Name": "apiRest",  
    "EndpointID": "c3a023df68aab3569165ac062a825081a23935dfbb0e578c3230c687e38b57e2",  
    "MacAddress": "02:42:ac:13:00:03",  
    "IPv4Address": "172.19.0.3/16",  
    "IPv6Address": ""  
  },  
  "62b617a9ad7a6b2743459ff2c9c177854108d5cf7d8fdc944523cb68d2453e08": {  
    "Name": "dbMySQL",  
    "EndpointID": "df38270140d34b6b3cfb906b21ca15773d368a7ec88ed01746b0a8f0d4a4cacd",  
    "MacAddress": "02:42:ac:13:00:02",  
    "IPv4Address": "172.19.0.2/16",  
    "IPv6Address": ""  
  },  
  "f66fccb91dd8d24424a102aa27ba059c24700a04b5869bcf4d543cef1fe67eab": {  
    "Name": "anguDockerFile",  
    "EndpointID": "50555c5172f9905923f0a33cc8d0bf815125a09bd8e56cbe516c95cfb6eda0aa",  
    "MacAddress": "02:42:ac:13:00:04",  
    "IPv4Address": "172.19.0.4/16",  
    "IPv6Address": ""  
  }  
},
```

```
PS C:\Users\Evengyl\Desktop\Docke & k8s> docker exec -ti apiRest sh  
/app # ping db  
PING db (172.19.0.2): 56 data bytes  
64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.089 ms  
64 bytes from 172.19.0.2: seq=1 ttl=64 time=0.076 ms  
64 bytes from 172.19.0.2: seq=2 ttl=64 time=0.105 ms
```

Les networks - Isolation

Pour finir, nous avons laissé le soin à notre **engine docker** de créer un **réseau isolé**, en effet, il n'est pas possible d'accéder à ces **réseaux** en dehors des différents containers !

```
C:\Windows\System32>ping 172.19.0.2

Envoi d'une requête 'Ping' 172.19.0.2 avec 32 octets de données :
Délai d'attente de la demande dépassé.

Statistiques Ping pour 172.19.0.2:
    Paquets : envoyés = 1, reçus = 0, perdus = 1 (perte 100%),
Ctrl+C
```

Les networks - Personnalisés

Ici nous avons laissé le soin à **compose** de créer le **network** lui même, le nom qu'il lui avait attribué n'était pas très pro...

Mais nous pouvons simplement le spécifier dans le **docker-compose** :

Du côté des **services** on va donner le nom du **network** que l'on veut utiliser :

```
networks:  
  - backend
```

Et du en bas du **docker-compose** on spécifie la création de ce même **réseau**, cela veut également dire que l'on peut en créer plusieurs !

Pour le back le front et bien d'autres applications, l'isolation reste la même mais nous pouvons différencier les réseaux !

```
networks:  
  backend:
```

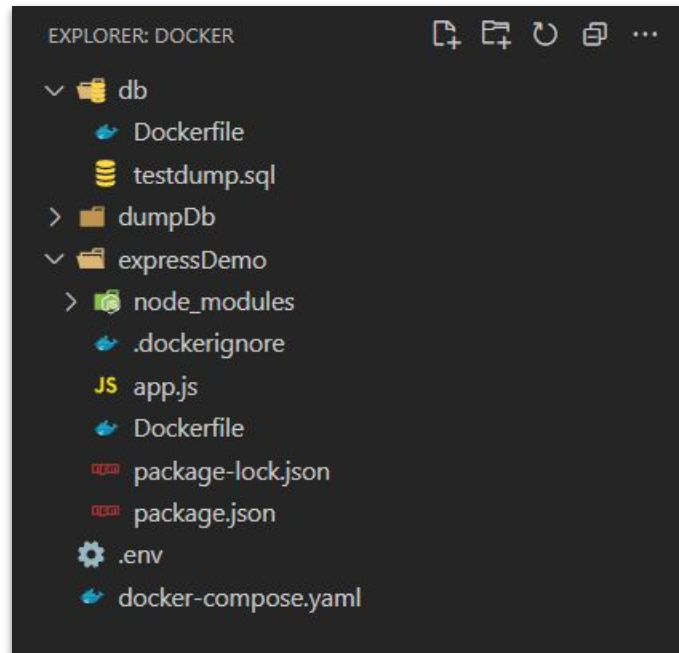
Notre architecture expliquée et détaillée
(basique)

Notre architecture

3 folders, un pour les fichiers de la db, un pour le volume partagé, et le dernier pour l'app express.

Vous remarquerez les deux fichiers dockerfile, le env, le docker-compose, et également dans la db, un petit script de post insert très utilisé pour faire des test surtout avec mysql et ses privilèges par défaut...

(Attention que les slides montre une architecture de base, le formateur aura pris soins de complexifier la structure avec d'autre volumes et containers → donc ces slides ne sont que informatifs)



L'intérieur de notre app Express


Une simple App Express pour les besoins de la démo.

Noter qu'ici les variable d'environnements seront celle que nous avons déjà dans notre fichier .env lors de la création de l'image dans le docker compose, l'avantage ici est la centralisation des données env, nous les définition à tout l'environnement docker des nos container.

```
expressDemo > JS app.js > ...
1  const express = require("express")
2  const mysql = require("mysql")
3  const app = express()
4
5  const connection = mysql.createPool({
6    connectionLimit: 10,
7    host: process.env.MYSQL_HOST,
8    user: process.env.MYSQL_USER,
9    password: process.env.MYSQL_PASSWORD,
10   database: process.env.MYSQL_DATABASE
11 })
12
13 app.get("/", (req, res) => {
14   connection.query("SELECT * FROM Student", (err, rows) =>
15     {
16       if (err)
17         res.json({ success: false, err })
18       else
19         res.json({ success: true, rows })
20     })
21 })
22
23
24 app.get("/post", (req, res) => {
25   connection.query("INSERT INTO Student (student_name, student_age) VALUES ('random student', 99)", (err, rows) =>
26     {
27       if (err)
28         res.json({ success: false, err })
29       else
30         res.json({ success: true, rows })
31     })
32 })
33
34 app.listen(5000, () => console.log("listening on port 5000"))
```

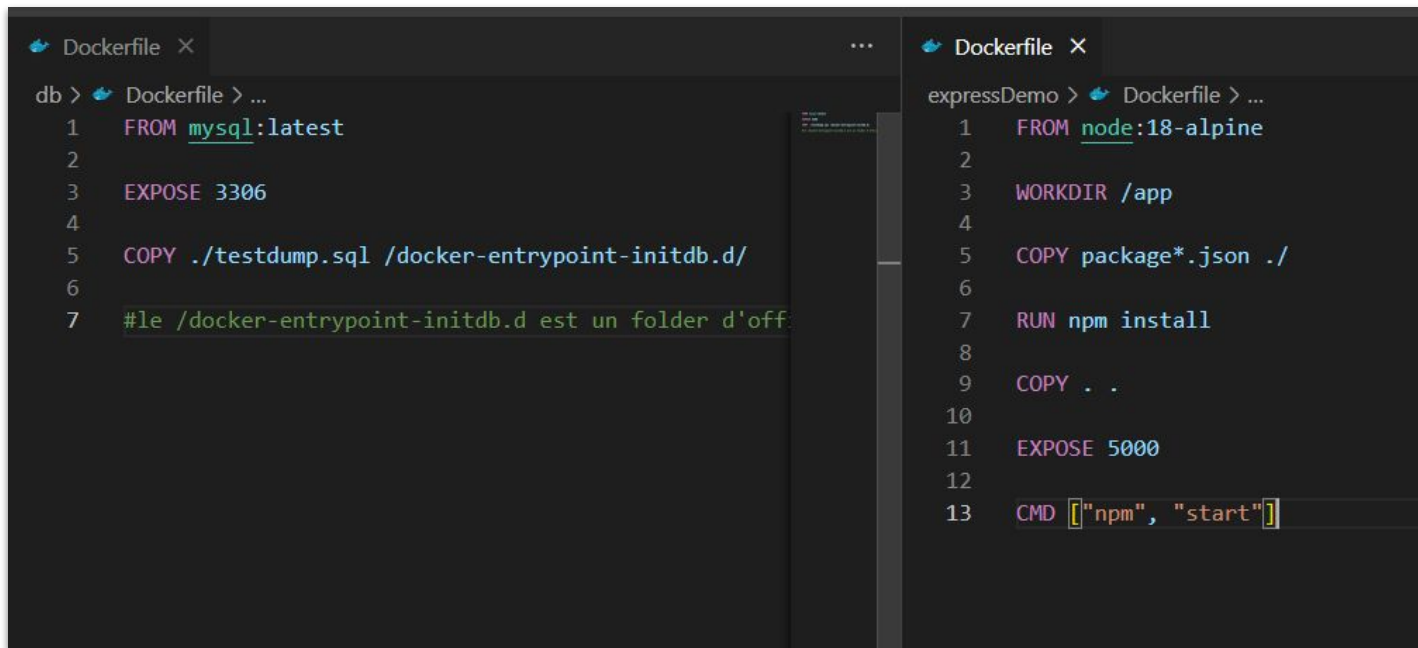
L'intérieur de notre app Express – Le .dockerignore

Il existe également un fichier dans docker qui fonctionne exactement de la même façon que le fichier **.gitignore**, le **.dockerignore**, il permet de ne pas prendre en considération ces **files/folders**, lors de la création de l'image, pratique pour ne pas devoir transférer les fichiers de git et les **node_modules** à notre app. Il est également placé à la racine de notre app express.

```
expressDemo >  .dockerignore
1  node_modules
2  .git
```

Création des fichiers DockerFile

Nous retrouvons les deux dockerfile que nous avons créé précédemment et ou les explications avec déjà été données



```
db > Dockerfile > ...
1 FROM mysql:latest
2
3 EXPOSE 3306
4
5 COPY ./testdump.sql /docker-entrypoint-initdb.d/
6
7 #le /docker-entrypoint-initdb.d est un folder d'off

expressDemo > Dockerfile > ...
1 FROM node:18-alpine
2
3 WORKDIR /app
4
5 COPY package*.json ./
6
7 RUN npm install
8
9 COPY . .
10
11 EXPOSE 5000
12
13 CMD ["npm", "start"]
```


Déployer plusieurs container en même temps

Lier des containers de l'application

Cette partie à été vue, le docker-compose sert à lier les différents containers.

il ne nous reste qu'à lancer le compositeur de nos containers.

La commande docker compose up permet de faire rouler les différents service mis en place dans notre compose.

Exemple pour notre app → **docker compose -f "docker-compose.yaml" up -d --build**

Ici, nous **nommons** le **fichier docker compose yaml** avec le **-f**, c'est un souci de lisibilité et de lecture.

le **-d** permet le mode détaché, et donc de récupérer l'invite de commande par la suite, le **--build** permet de dire à compose qu'il doit également **builder** les **images** qu'il trouvera dans le compose même si elles existent, il va revérifier les layers du dockerfiles et mettre à jour les images au besoin.

Interfaces d'administration

Interface d'administration en web

L'interface d'administration en web permet d'avoir un aperçu graphique de l'état des containers et de leur consommation de ressources.

Elle permet également d'interagir avec les containers en temps réel, par exemple en arrêtant ou en démarrant un container, en surveillant les logs, en affichant les statistiques de performance, etc.

Plusieurs outils sont disponibles pour mettre en place une interface d'administration en web pour **Docker**, tels que **Portainer**, **Docker UI**, ou encore **Kitematic**. Ces outils peuvent être facilement intégrés à un environnement **Docker** existant et offrent une interface utilisateur intuitive pour gérer les **containers**.

The Kitematic logo is displayed in white, uppercase, sans-serif font on a solid blue rectangular background. The text "KITEMATIC" is followed by a small trademark symbol (TM).

Portainer

L'interface d'administration en web permet d'avoir un aperçu graphique de l'état des containers et de leur consommation de ressources.

Elle permet également d'interagir avec les containers en temps réel, par exemple en arrêtant ou en démarrant un container, en surveillant les logs, en affichant les statistiques de performance, etc.

Plusieurs outils sont disponibles pour mettre en place une interface d'administration en web pour Docker, tels que Portainer, Docker UI, ou encore Kitematic.

Ces outils peuvent être facilement intégrés à un environnement Docker existant et offrent une interface utilisateur intuitive pour gérer les containers.

Stack details
Stacks > tutor_local

Stack

Information

This stack was created outside of Portainer. Control over this stack is limited.

Stack details

tutor_local

Containers

Columns Settings

Start Stop Kill Restart Pause Resume Remove

Search...

Name	State	Quick actions	Stack	Image	Created	Published Ports	Ownership
tutor_local_cms-worker_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/overhangio/openedx:12.0.4	2021-09-14 19:13:57	-	administrators
tutor_local_nginx_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/nginx:1.19.9	2021-09-14 19:13:57	-	administrators
tutor_local_lms-worker_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/overhangio/openedx:12.0.4	2021-09-14 19:13:55	-	administrators
tutor_local_cms_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/overhangio/openedx:12.0.4	2021-09-14 19:13:55	-	administrators
tutor_local_lms_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/overhangio/openedx:12.0.4	2021-09-14 19:13:54	-	administrators
tutor_local_forum_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/overhangio/openedx-forum:12.0.4	2021-09-14 19:13:52	-	administrators
tutor_local_mysql_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/mysql:5.7.33	2021-09-14 19:13:50	-	administrators
tutor_local_caddy_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/caddy:2.3.0	2021-09-14 19:13:50	80:80 80:80	administrators
tutor_local_redis_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/redis:6.2.1	2021-09-14 19:13:50	-	administrators
tutor_local_mongodb_1	running	Start Stop Kill Restart Pause Resume Remove	tutor_local	docker.io/mongo:4.0.25	2021-09-14 19:13:50	-	administrators

Items per page 10 1 2

Pour des contraintes de temps, les interfaces ne seront pas vues dans cette formation, vous pouvez également voir l'interface de docker Desktop au slide suivant.

L'interface d'administration de Docker Desktop

The screenshot shows the Docker Desktop interface for a container named 'my-project-name'. The path is 'C:\Users\Evengyl\Desktop\Bruxelles Formation - FS JS - 2023\Docker'. Two containers are listed: 'db' (my-project-name-db) and 'express' (my-project-name-express), both in a 'Running' state. The 'db' container has a port mapping of '5000:5000'. The logs for the 'db' container are displayed, showing a series of timestamps and messages related to the MySQL server startup and configuration.

```
2023-04-09 17:54:17 db | 2023-04-09 15:54:17+00:00 [Not
2023-04-09 17:54:18 express | 
2023-04-09 17:54:18 db | 2023-04-09 15:54:18+00:00 [Not
2023-04-09 17:54:18 db | 2023-04-09 15:54:18+00:00 [Not
2023-04-09 17:54:19 express | > demo_fs_js@0.0.1 start
2023-04-09 17:54:19 express | > node app.js
2023-04-09 17:54:19 express | 
2023-04-09 17:54:20 express | listening on port 5000
2023-04-09 17:54:18 db | '/var/lib/mysql/mysql.sock' -s
2023-04-09 17:54:19 db | 2023-04-09T15:54:19.437584Z 0
SET GLOBAL host_cache_size=0 instead.
2023-04-09 17:54:19 db | 2023-04-09T15:54:19.446230Z 0
2023-04-09 17:54:19 db | 2023-04-09T15:54:19.453665Z 0
2023-04-09 17:54:19 db | 2023-04-09T15:54:19.510022Z 1
2023-04-09 17:54:21 db | 2023-04-09T15:54:21.205834Z 1
2023-04-09 17:54:22 db | 2023-04-09T15:54:22.155346Z 0
2023-04-09 17:54:22 db | 2023-04-09T15:54:22.155844Z 0
nnel.
2023-04-09 17:54:22 db | 2023-04-09T15:54:22.187842Z 0
l OS users. Consider choosing a different directory.
2023-04-09 17:54:22 db | 2023-04-09T15:54:22.277297Z 0
k
2023-04-09 17:54:22 db | 2023-04-09T15:54:22.277858Z 0
port: 3306 MySQL Community Server - GPL.
```

The screenshot shows the Docker Desktop interface for a container named 'my-project-name-db'. The path is 'e37023ca6142'. The container is in a 'Running' state. The 'Stats' tab is selected, showing various metrics: CPU Usage (0.58%), Memory Usage (357 MB), Disk Read/Write (0 Bytes / 0 Bytes), and Network I/O (1.4 kB / 0 Bytes).

Metric	Value
CPU Usage	0.58%
Memory Usage	357 MB
Disk Read/Write	0 Bytes / 0 Bytes
Network I/O	1.4 kB / 0 Bytes

The screenshot shows the Docker Desktop interface for a container named 'my-project-name'. The path is '-'. The container is in a 'Running' state. The overview shows two containers: 'db' (my-project-name-db) and 'express' (my-project-name-express), both in a 'Running' state. The 'db' container has a port mapping of '5000:5000'.

Container Name	ID	Image	Status	Port Mapping	Time
db	e37023ca6142	my-project-name-db	Running		5 minutes ago
express	d78092148f26	my-project-name-express	Running	5000:5000	5 minutes ago

Administrer des containers en production

Automatiser le démarrage des containers au boot sys

Pour automatiser le démarrage des containers au boot système afin de s'assurer que les containers redémarrent automatiquement après un reboot du système.

Il est possible d'utiliser différentes méthodes :

- Utiliser un **script** de **démarrage** qui exécute la commande docker run pour chaque container à démarrer.
(pratique pour les linuxien - tâche cron par exemple)
- Utiliser **Docker Compose** avec l'option `—always-recreate-deps` pour s'assurer que les containers soit créés au démarrage - voir la doc de docker compose up → [docker compose up](#)
- Configurer les containers pour qu'ils redémarrent automatiquement en utilisant les options **restart** de la commande docker run
- Directement dans le fichier **docker-compose.yaml** (ce que nous allons voir dans le slide suivant)

Avec le Docker compose - Restart : always

Pour dire à **Docker Compose** de redémarrer les containers au démarrage du système hôte, il est nécessaire d'utiliser la commande `restart: always` dans le fichier **docker-compose.yaml**, pour chaque service qui doit être redémarré. Cette commande permet de définir la politique de redémarrage du service en cas d'erreur ou de redémarrage du **système hôte**.

ps * : Cette opération prend un peu de temps mais on constate que les containers ont démarré directement au lancement de **docker desktop** ! donc de **docker**.

ps formateur : prendre le temps de ou, redémarrer la machine, ou couper desktop et relancer (attention c'est long...).

```
docker-compose.yaml
1  version: "3.8"
2
3  services:
4    db:
5      restart: always
6      image: mysql:latest
7      container_name: db
8      volumes:
9        - ./data:/var/lib/mysql
10     environment:
11       MYSQL_ROOT_PASSWORD: root
12
13     express:
14       restart: always
15       build: .
16       container_name: express
17       ports:
18         - "3000:3000"
19       depends_on:
20         - db
```


Gestion des logs des containers

Les logs des **containers Docker** sont des informations **importantes** pour la maintenance et le débogage des applications en production.

Pour récupérer les logs d'un container, vous pouvez utiliser la commande **docker logs** suivi du nom ou de l'ID du container.
→ **docker logs db** **OU** **docker logs express**

Vous pouvez également configurer Docker pour rediriger les logs vers des fichiers sur le système hôte.
Pour ce faire, vous pouvez ajouter l'option `--log-driver` suivi d'un nom de pilote de journalisation (par exemple, **json-file** ou **syslog**) et des options supplémentaires telles que le chemin d'accès au fichier journal dans le fichier **docker-compose.yml**.

Voici la doc au besoin → [docker compose logs](#)

[docker logs | Docker Documentation](#)

```
logging:
  driver: "json-file"
  options:
    max-size: "50m"
    max-file: "3"
```