



Formation Docker & Kubernetes

SOMMAIRE

I. Section 1 : Getting Started

A. Qu'est ce que c'est que Docker

1. Définition

2. C'est quoi un container

3. Exemple

B. Pourquoi docker & containers

1. Exemple : Différentes environnement de Développement

2. Exemple : Travail d'équipe

3. Exemple : Plusieurs projet

4. Avantages

C. Virtuelles machines & Docker container

1. Fonctionnement Machines virtuelles

2. Fonctionnement Docker & container

D. Installation Docker

II. Section 2 : Docker Image & Container

A. Qu'est ce que c'est que Image et Pourquoi ?

B. Utiliser et Démarrer des images externes

C. Construire notre propre image avec Dockerfile

NB

D. Gestion des images et containers

NB

E. Entrer en mode interactive

E. Attachement et détachement d'un conteneur

NB

F. Suppression conteneur et images

a. Suppression conteneur

b. Suppression images

Note

G. Inspection d'une image

H. Copier des fichiers de ou dans un conteneur

I. Nomenclature des conteneurs et images

a. Nommer un conteneur

b. Nommer images

J. Partager d'images docker

Note

Ressources

I. Section 1 : Getting Started

A. Qu'est ce que c'est que Docker

1. Définition

Docker est une technologie de conteneur: Un outil pour la création et la gestion de containers.

2. C'est quoi un container

Un container est une unité de logiciel standardisée, ce qui signifie qu'il s'agit d'un package de code et qu'il est important d'avoir les dépendances et les outils nécessaires pour exécuter ce code.

3. Exemple

Si on crée une application nodeJs avec un container construit avec Docker, on pourrais avoir le code source de notre application dans notre container, ainsi que le runtime nodeJS et tout autre outil nécessaire pour exécuter ce code. Et les avantages que les même conteneur avec le même code nodeJS et le même outil nodeJS, avec toujours la même version, donnera toujours exactement le même comportement et le même résultat.



Docker peut être installé sur tous les systèmes d'exploitation modernes pour y travailler et Docker est finalement un outil qui simplifie le processus de création et de gestion de ces conteneurs.

B. Pourquoi docker & containers

Pour résoudre la compatibilité des versions de différents environnements, on peut utiliser les conteneurs qui nous permettront d'avoir la même version dans tous les environnements.

1. Exemple : Différents environnements de Développement

Si on a une version nodeJs 12.9.0 en local qui fonctionne correctement et qu'on la met en place dans un serveur (machine distante) qui peut avoir une autre version de nodeJs. On pourrait avoir des problèmes de version et que l'exécution ne fonctionne pas correctement. Du coup pour résoudre le problème on peut utiliser les conteneurs pour avoir la même version dans tous les environnements.

2. Exemple : Travail d'équipe

Si une équipe travaille, Une personne X "**push**" son code avec une version nodeJS 10.3.7 et une personne Y "**pull**" le code et qu'il a une version 14.2.8, Ils peuvent avoir des problèmes d'exécution de code.

3. Exemple : Plusieurs projets

Imaginons que je travaille sur des projets python A et B et que je switch entre les projets. Les outils utilisés dans le projet A ne pourraient pas être compatibles avec les outils du projet B.

4. Avantages



Avoir exactement les mêmes environnements de développement et de production → Cela garantit qu'il fonctionne exactement comme testé



Cela devrait être plus facile de partager des environnements de développement communes / installation avec (nouveau) employés et collègues



Ne pas désinstaller et réinstaller des dépendances et exécuter tous les temps

C. Virtuelles machines & Docker container

1. Fonctionnement Machines virtuelles

On pouvait utiliser une machine virtuelle mais l'un des plus gros problèmes est le système d'exploitation en générale la surcharge qu'on a sur plusieurs machines virtuelles. Chaque machine virtuelle est comme un ordinateur autonome fonctionnant au-dessus de notre machine. Et donc si on a plusieurs machines en particulier, nous avons beaucoup d'espace et de ressources gaspillés car chaque fois qu'un tout nouvel ordinateur doit être installé à l'intérieur de notre machine, cela consomme de la mémoire, du processeur et également l'espace sur notre disque dur. Et cela peut être un problème d'avoir de plus en plus de machines virtuelles.

Donc on a des avantages et des inconvénients avec des machines virtuelles.

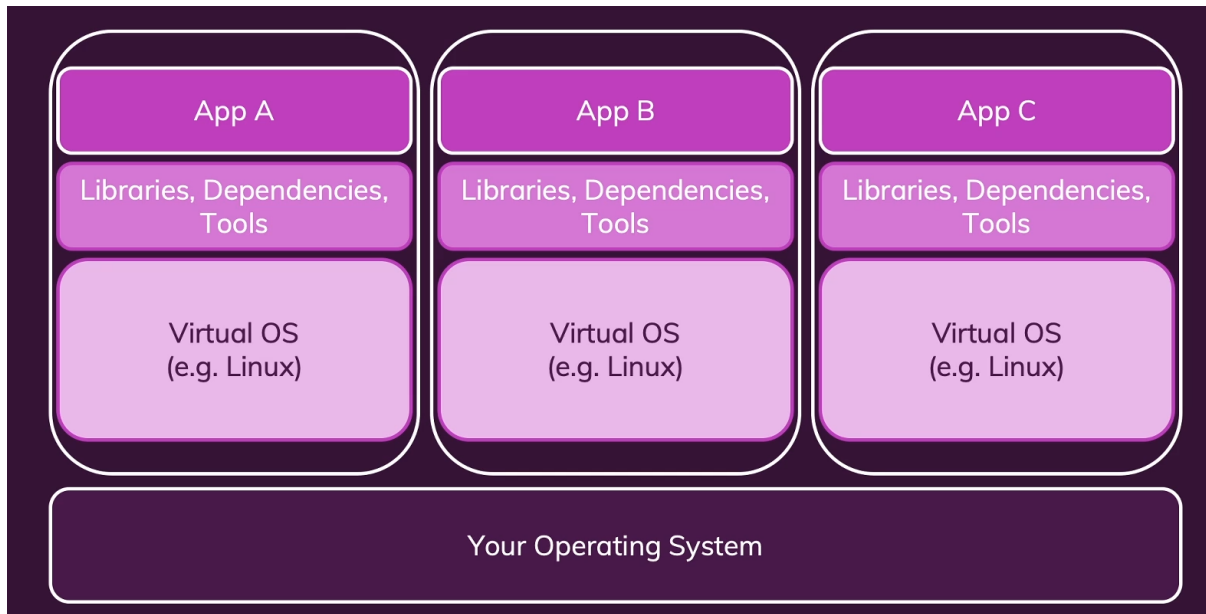
Avantages

- Créer environnement séparée
- Configuration spécifique à l'environnement
- Partager tout et reproduire de manière fiable

Inconvénients

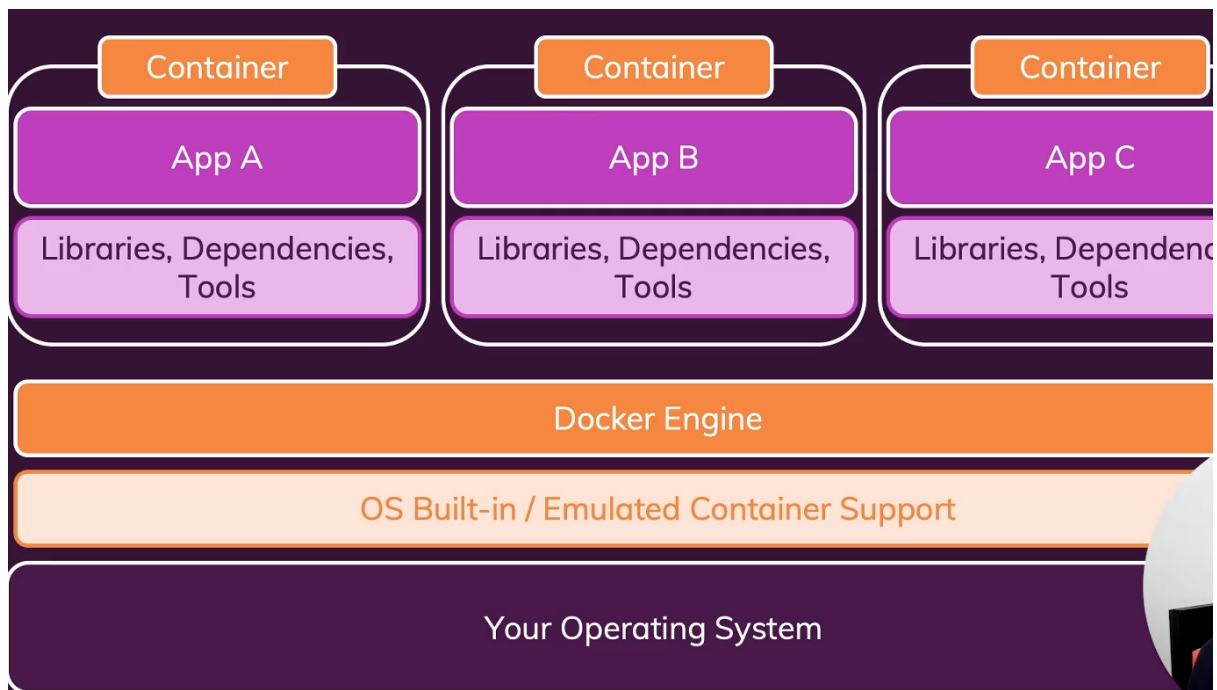
- Duplication redondante/ espace de gaspillage
- Mauvaise performance dû à plusieurs machines virtuelles
- Faire les configurations de la même manière/ Pas de fichier configurable

unique



2. Fonctionnement Docker & container

Avec les conteneur, nous avons notre système d'exploitation hôte (Linux, windows, macOS, ...) d'où nous utilisons un support de conteneur intégré ou un support de conteneur émulé (Docker). Et nous exécutons un outils appelé Docker Engine et tout cela configuré par Docker lors de l'installation. **Docker Engine est un outil léger pouvant faire tourner les containers.** Et ces dernières contiennent notre code et les outils et environnements d'exécution essentiels dont notre code a besoin mais ils ne contiennent pas de système d'exploitation surchargé ou d'outils supplémentaire. Ils peuvent avoir une petite couche de système d'exploitation à l'intérieur du conteneur qui sera une version très légère d'un systèmes d'exploitation. On pourrais aussi configurer et les décrire avec un fichier de configuration et ensuite partager ce fichier avec d'autres afin qu'il puisse recréer le conteneur ou construire le container de ce qu'on appelle **image**. Et on pourra partager cette image avec d'autres.




Avantages

- Faible impact sur notre système d'exploitation et notre machine (plus rapide et moins de disque)
- Partager, reconstruire et distribuer les conteneurs très facilement (car nous avons ces images et ces configurations)
- Des applications et des environnements encapsulés avec tout ce dont notre application a besoin

D. Installation Docker

Get Docker CE for Ubuntu

Instructions for installing Docker CE on Ubuntu

 <https://docs.docker.com.zh.xxy2401.com/v17.12/install/linux/docker-ce/ubuntu/#set-up-the-repository>

Docker Hub : C'est un service qui nous permet d'héberger nos images dans le cloud sur le web.

Docker Compose : Il nous facilite la gestion de conteneurs plus complexe ou multi-conteneurs

II. Section 2 : Docker Image & Container

A. Qu'est ce que c'est que Image et Pourquoi ?

Une image est un modèle/plan d'un conteneur. C'est l'image qui contiendra le code et les outils nécessaires pour exécuter le code. Et c'est le conteneur qui s'exécute et exécute ensuite le code.

Nous créons une image avec toutes ces instructions de configuration et tout notre code une fois, mais ensuite nous pouvons utiliser cette image pour créer plusieurs conteneurs basés sur cette image.

Principales différences : des images Docker et des conteneurs Docker

Un conteneur Docker est une application ou un service logiciel intégré et exécutable. D'autre part, une image Docker est le modèle chargé sur le conteneur pour l'exécuter, comme un ensemble d'instructions.

Résumé des différences : des images Docker et des conteneurs Docker

	Image Docker	Conteneur Docker
De quoi s'agit-il ?	Fichier réutilisable et partageable utilisé pour créer des conteneurs.	Une instance d'exécution ; un logiciel intégré.
Créé à partir de	Code logiciel, dépendances, bibliothèques et Dockerfile.	Une image.
Composition	Couches en lecture seule.	Couches en lecture seule avec une couche de lecture-écriture supplémentaire sur le dessus.
Mutabilité	Immuable. S'il y a des modifications, vous devez créer un nouveau fichier.	Mutable ; vous pouvez le modifier au moment de l'exécution si nécessaire.
Quand utiliser	Pour stocker les détails de configuration de l'application sous forme de modèle.	Pour exécuter l'application.

B. Utiliser et Démarrer des images externes

Les conteneurs ne sont que les instances d'images en cours d'exécution.

Pour exécuter un conteneur, on peut faire :

```
docker run < Nom de l'image >
```

On pourrait exécuter par exemple node JS pour obtenir un shell interactif dans lequel nous pouvons insérer une commande. Mais par défaut, un container est isolé de l'environnement qui l'entoure.

On peut consulter tout les conteneurs existant avec la commande :

```
docker ps -a
```



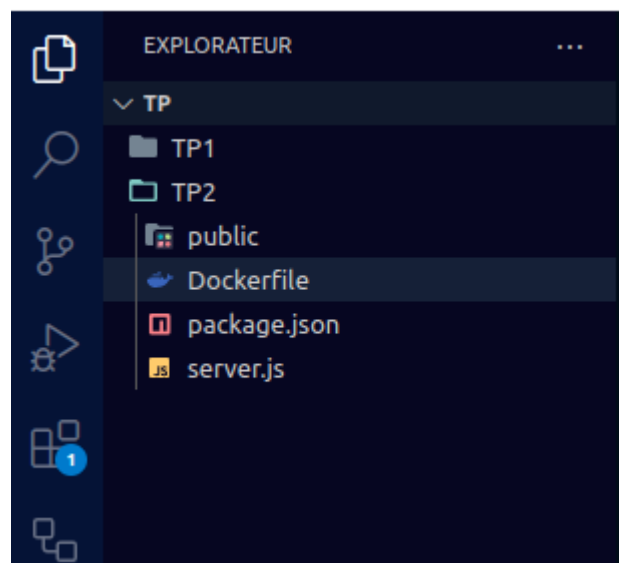

ps : processus

Pour obtenir le mode interactive, nous pouvons utiliser la commande

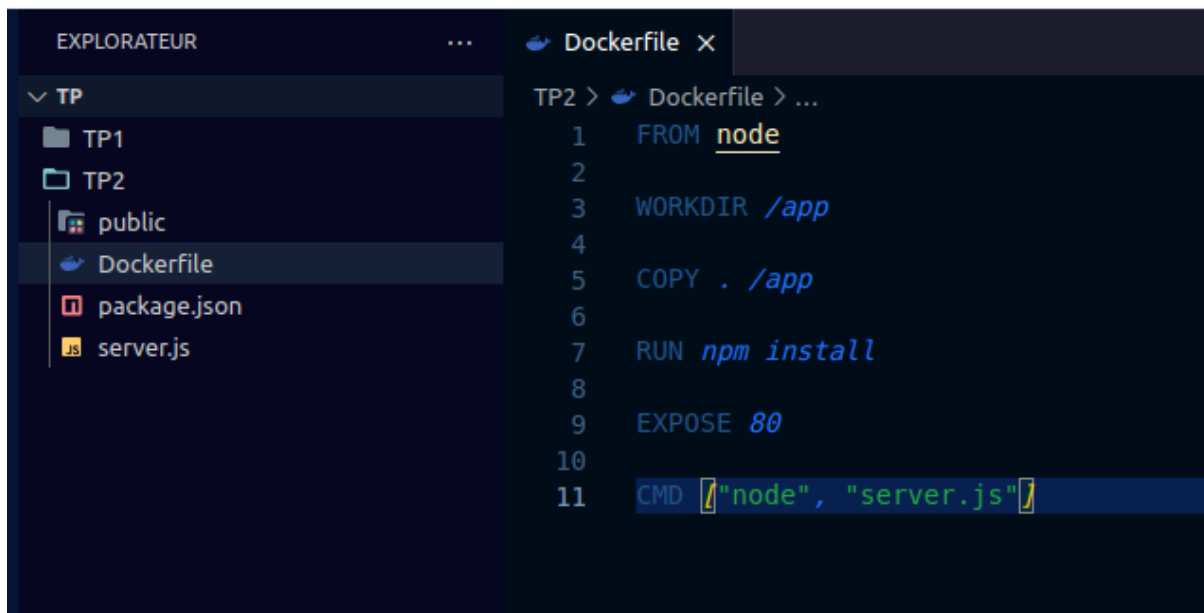
```
docker run -ti < Nom de l'image ex: node >
```

C. Construire notre propre image avec Dockerfile

Nous pouvons construire nos propres images en créant un fichier **dockerfile** dans le dossier racine où se trouve nos fichier.



Et dans ce dockerfile, nous allons lui passer des instructions pour démarrer notre server nodeJS.



1. Construire sur l'image node JS
2. Pour les moyens pratiques on peut utiliser un dossier de travail dans notre conteneur en définissant le **WORKDIR** avec le dossier **app**
3. Ensuite on **copie** toutes les dossier et sous-dossier de notre **app** local dans le dossier **app** de notre conteneur
4. Puis on lance la commande **RUN npm install** pour installer tout les dépendance de notre projet dans le conteneur
5. Et enfin lancer commande **CMD** suivi d'un tableau avec les commandes nécessaire pour démarrer notre serveur.



Pour la commande **CMD**, nous pouvions utiliser la commande **RUN**, mais nous démarrons notre conteneur à partir d'une image. Et donc la commande **RUN node server.js** essaierait de démarrer le serveur dans l'image donc dans le template. Nous aimerions démarrer un conteneur basé sur une image aussi démarrer plusieurs conteneurs sur une seul et même image, nous démarrons également plusieurs serveur de nodeJS. La différence à exécuter est que cela ne sera plus exécuter lorsque l'image est créer mais lorsqu'un conteneur est démarré en fonction de l'image.

Pour construire l'image on peut lancer la commande :

```
docker build .  
// Lancer la commande où se situe le DockerFile
```

Ensuite démarer notre conteneur en exposant son port

```
docker run -p 3000:80 <ID de l'image>
```

Si on retape la commande docker build, on constate le build ne prends pas trop de temps car Docker met en cache chaque résultat d'instruction et lorsqu'on construit une image il utilisera ces résultats mise en cache. C'est ce qu'on appelle architecture basée des couches (Layer).

Si nous prenons notre fichier dockerfile, chaque instruction représente une couche dans Dockerfile.

NB



Dans le dockerfile, le mieux est de copier le package.json au lieu de copier le répertoire. Cela permet de faire que docker ignore certains layers comme "Run npm install" si le package.json n'est pas modifié. Et cela permet de gagner en performance.

Nous pouvons instancier et exécuter plusieurs conteneurs basés sur une image.

L'image est la chose qui contient notre code et ainsi de suite.

D. Gestion des images et containers

Pour lister tous les conteneurs en cours d'exécution ou tous les containers, nous pouvons utiliser la commande :

```
//container en cours d'exécution  
docker ps  
//tous les container  
docker ps -a
```

NB



La commande docker run nous crée un nouveau conteneur basé sur une image

S'il n'y a pas de modification dans notre image, il n'est pas nécessaire de lancer la commande docker run pour créer un nouveau conteneur. Et pour redémarrer un container on peut faire la commande :

```
docker start <Nom du conteneur ou ID>
```

E. Entrer en mode interactive

Pour entrer en mode interactive avec docker par exemple pour une application python qui demande des interaction avec le terminal, on peut faire :

```
docker run -ti <ID image>
```

E. Attachement et détachement d'un conteneur

Pour démarer avec docker start, on est en mode détaché par défaut et pour excuter avec docker run, on est en mode attaché par défaut.

Le mode attache permet d'écouter le server et d'afficher les console logs.

On peut démarer le container en mode détacher avec la commande :

```
docker run -p 3000:80 -d <ID de l'image>
```

Pour attacher un conteneur, on peut utiliser la commande :

```
docker attach <Nom du container>
```

Si un container est en mode détacher, on peut logger le sans utiliser le mode attach :

```
docker log <Nom du container>
```

NB

F. Suppression conteneur et images

a. Suppression conteneur

On ne peut pas supprimer un conteneur en cours d'exécution.

Pour supprimer un conteneur, nous pouvons utiliser la commande :

```
docker rm <Nom du conteneur>
```

Pour supprimer plusieurs conteneur, on peut utiliser la même commande **docker rm** suivi des noms des conteneurs séparés par des espaces. Nous avons aussi une alternative :

```
docker container prune
```

Cette commande supprime toutes les conteneur en stop.

b. Suppression images

Pour supprimer une image, on peut utiliser la commande :

```
docker rmi <ID image>
```

Note

On ne peut supprimer d'image que si elles ne sont plus utilisées par un conteneur et cela inclu les conteneurs arrêtés. Si nous avons un conteneur qui est arrêté, nous ne pouvons pas supprimer les images utilisées par ce conteneur. Nous devons d'abord supprimer ce conteneur. Les conteneurs doivent être supprimés en premier.

Pour supprimer tous les images non utilisées par les conteneurs, nous pouvons utiliser la commande :

```
docker image prune  
// ou  
docker rmi <ID image1> <ID image2>
```

Nous pouvons aussi supprimer un conteneur automatiquement quand il est arrêté. Pour cela lors de l'exécution du conteneur, nous pouvons rajouter le flag `--rm`

```
docker run -p 3000:80 -d --rm <<ID image>
```

G. Inspection d'une image

Pour en savoir plus sur une image, nous pouvons utiliser la commande :

```
docker image inspect <ID image>
```

H. Copier des fichiers de ou dans un conteneur

Pour ajouter des fichiers ou dossier dans notre conteneur (en cours d'exécution), nous pouvons utiliser la commande suivante :

```
docker cp dossier <Nom du conteneur>:/<dossier dans docker>  
docker cp dummy/. nom_du_conteneur:/test
```

On pourrait aussi extraire des fichiers ou dossier depuis docker:

```
docker cp <Nom du conteneur>:/test dummy
```

I. Nomenclature des conteneurs et images

a. Nommer un conteneur

Pour nommer un conteneur lors de l'exécution, nous pouvons ajouter le flag `—name` :

```
docker run -p 3000:80 -d --rm --name test <ID images>
```

b. Nommer images

On peut donner un nom aux images mais ici on les appelle tag(balise). Les tags d'une image se compose en 2 parties :

- **Name** : est le nom réel également appelé référentiel (repository) de notre image.
- Tag : séparé par 2 point avec le name

Ce concept d'avoir c'est 2 parties existe pour une raison simple. Le **name** pour le nom général de notre image et plus précisément pour créer un groupe d'images et le **tag** qui est facultatif mais on peut définir comme notre version d'image.

Pour ajouter un tag dans notre image :

```
docker build -t test:latest .
```



La partie tag peut un mot ou nombre



Pour supprimer tous les images avec les tags, nous pouvons utiliser la commande : `docker image prune -a`

J. Partager d'images docker

Nous créons nos conteneurs à partir de notre images docker raison pour laquelle nous ne partageons pas les conteneurs mais en réalité les images. Car si nous avons une image nous créer un conteneur et l'exécuter à partir de cette image.

Et pour partager notre image, nous avons 2 options :

- Dockerfile : avec le dockerfile et le code source qui appartient à l'application, nous pouvons construire notre image et exécuter notre conteneur.



Juste faire un `docker build` et exécuter l'image

- Partager une image construit dans docker hub

Pour partager notre image dans docker hub, nous devons créer un compte dans docker hub. Ensuite nous devons nous connecter avec notre docker local avec la commande :

```
//login
docker login // entrer nos identifiants
//déconnexion
docker logout
```


Ensuite nous pouvons pusher notre images dans docker hub et pour cela, nous devons d'abord créer notre repos et ensuite on renomme notre image avec le meme nom que notre repos avec la commande :

```
docker tag <nom de l'image à renommée> <nouveau nom>
```

Note

Cette dernière commande va une clone de l'image de ce fait on aura toujours l'ancienne image avec le même nom et un clone avec le nouveau nom.

Et pour push l'image on peut faire :

```
docker push <nom du repos>
```

- On pull une image publique sans se connecter

```
docker pull <nom de l'image dans docker hub>
```

Ressources

https://drive.google.com/file/d/1gdqNA_5nPKKm6GCysWOhyVeISYpkLBRL/view?usp=sharing