

Outils et méthodes de développement

Johann Chopin

Premier semestre de la 3ème années de la licence
Informatique et ingénierie du web.



Introduction à Docker

Plan

1. Conteneurisation avec Docker

- 1.1. Définitions et conc...
- 1.1.1. Qu'est-ce que DevO...
- 1.1.2. Qu'est-ce que la virt...
- 1.1.3. Qu'est ce que Docker?
- 1.1.4. Exécution de conten...
- 1.1.5. Images et conteneurs
 - 1.1.5.1. Image
 - 1.1.5.2. Dockerfile
 - 1.1.5.3. Conteneur
- 1.1.6. CLI de Docker: les b...
 - 1.1.6.1. Exercice:

1.2. Exécution et arrêt ...

1.3. Analyse des images

- 1.3.1. D'où viennent les im...
- 1.3.2. Analyse détaillée d'u...
- 1.3.3. Build une image
- 1.3.4. Exercice
- 1.3.5. Différence entre les i...

1.4. Stockage des donn...

1.4.1. Les volumes

1.5. Les ports

1.6. Publier une image

1.7. Best practices pour...

1.7.1. Analyse des layers d'...

1.7.2. Gérer le cache des la...

1.7.3. Réduire la taille des l...

1.7.4. Minimiser le nombre ...

1.8. En résumé

1.9. Exercice:

2. Orchestration de conteneurs avec Docker

2.1. Docker Compose

2.1.1. Format YAML

2.1.2. Modèle d'application ...

2.1.2.1. Services

2.1.2.2. Networks

2.1.2.3. Volumes

2.1.2.4. Secrets

2.1.2.5. Version

2.2. Docker compose CLI

2.3. Exercice

2.4. Conteneurs et dév...

2.4.1. Spécification de build

2.4.2. Exercice

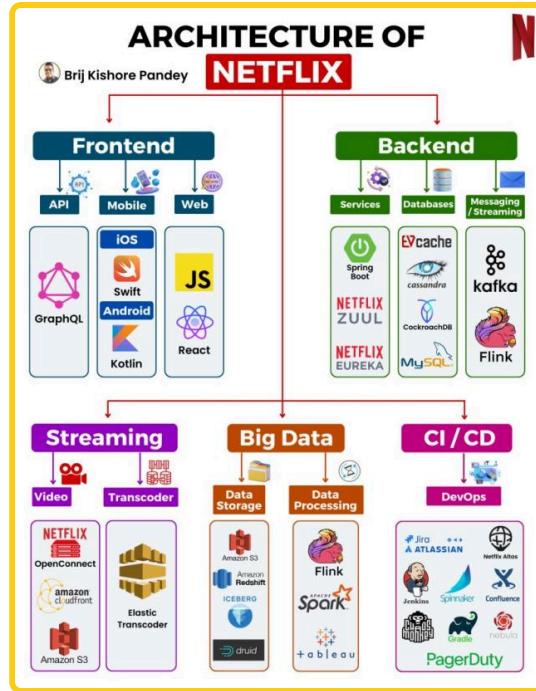
2.4.3. Utilisation de Compo...

2.5. Docker Compose e...

2.6. En résumé

3. Ressources

Conteneurisation avec Docker

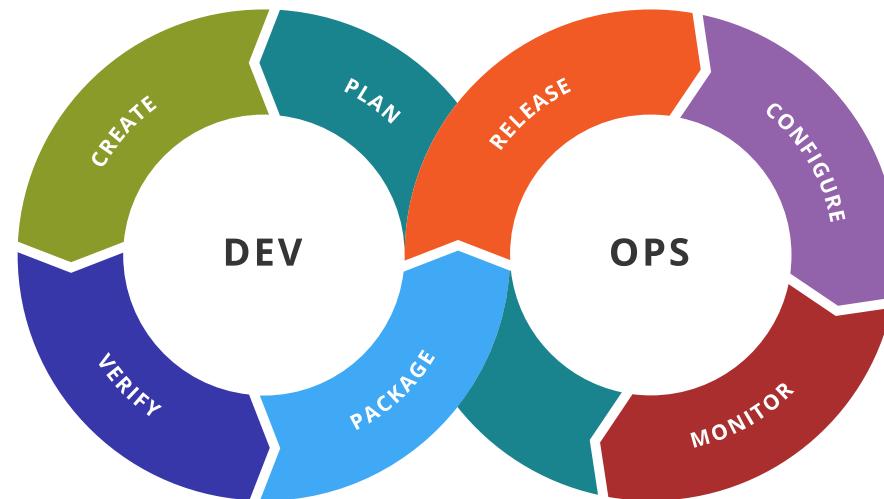


Définitions et concepts de base

Qu'est-ce que DevOps ?

"DevOps est une méthodologie de développement visant à combler le fossé entre le développement et les opérations, en mettant l'accent sur la communication et la collaboration, l'intégration continue, l'assurance qualité et la livraison avec un déploiement automatisé utilisant un ensemble de pratiques de développement."

— Jabbari et al



Ce cours se concentre principalement sur les notions:

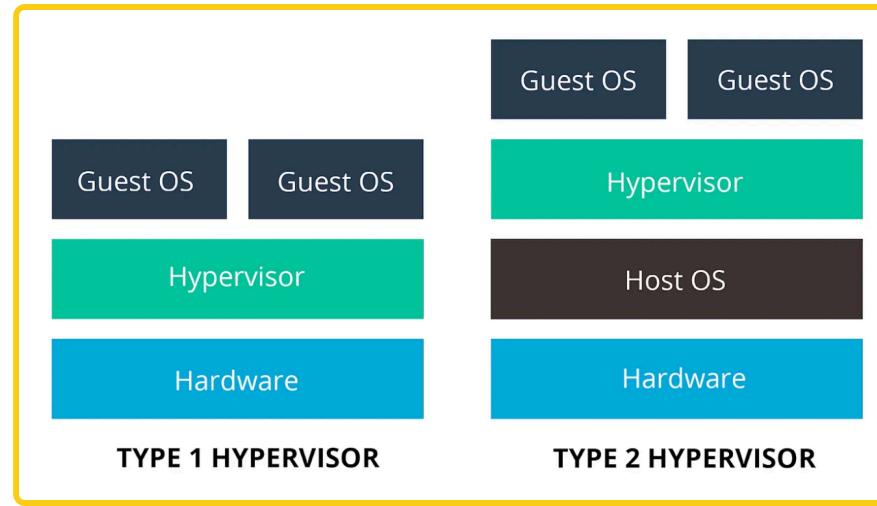
- Verify
- Packaging
- Releasing
- Configuring

Qu'est-ce que la virtualisation ?

Virtualisation: : faire fonctionner sur un même serveur physique, plusieurs systèmes, serveurs (virtualisation système) ou applications (virtualisation applicative) comme s'ils fonctionnaient sur des machines distinctes. La virtualisation est un composant technique clé dans le **Cloud Computing**.

Principes de Virtualisation:

- Un système d'exploitation principal ou hôte est installé sur un serveur physique unique. Ce système hôte va servir à accueillir plusieurs autres systèmes d'exploitation par le biais d'un logiciel appelé **hyperviseur**.
- **Hyperviseur:** Logiciel de virtualisation.



	Hyperviseurs de type 1	Hyperviseurs de type 2
Également connu sous le nom de	Hyperviseur matériel nu	Hyperviseur hébergé
Fonctionne sur	Matériel physique sous-jacent de la machine hôte	Système d'exploitation sous-jacent (système d'exploitation hôte)
Idéal pour	Charges de travail volumineuses, nécessitant une quantité importante de ressources ou à usage fixe	Environnements de bureau et de développement
Peut-il négocier des ressources dédiées ?	Oui	Non
Exemples	VMware ESXi, Microsoft Hyper-V, KVM	Oracle VM VirtualBox, VMware Workstation, Microsoft Virtual PC

Intérêts de la virtualisation:

- utilisation optimale des ressources d'un parc de machines
- installation, déploiement et migration facile des machines virtuelles d'une machine physique à une autre, livraison facilitée
- économie sur le matériel par mutualisation
 - consommation électrique
 - entretien physique, surveillance
 - support
 - compatibilité matérielle
- installation, tests, développements, cassage et possibilité de recommencer sans casser le système d'exploitation hôte
- sécurisation et/ou isolation d'un réseau
- isolation des différents utilisateurs simultanés d'une même machine
- allocation dynamique de la puissance de calcul

Limitations de la virtualisation:

- Redondance de la virtualisation d'un OS
- L'application client est dépendante de l'OS utilisé de base et une migration sera nécessaire pour passer sur un OS "incompatible"

💡 Si seulement il était possible d'être indépendant des différents OS

Qu'est ce que Docker?



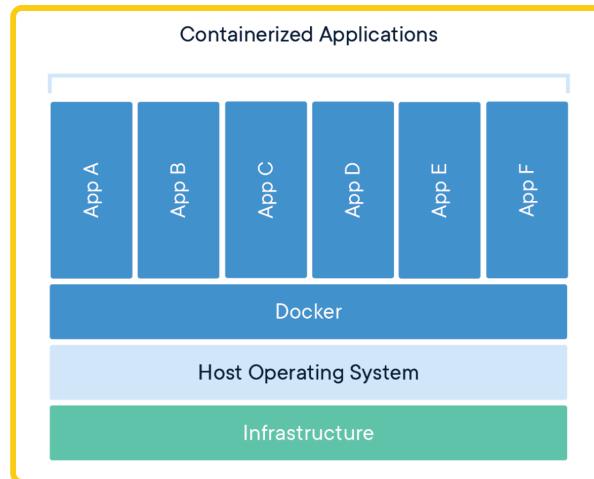
"Docker est un ensemble de produits PaaS (platform as a service) qui utilisent la virtualisation au niveau du système d'exploitation pour fournir des logiciels dans des paquets appelés conteneurs."

— O'Gara, Maureen

En éliminant le jargon, nous obtenons 2 définitions :

- Docker est un ensemble d'outils permettant de fournir des logiciels dans des conteneurs.
- Les conteneurs sont des paquets de logiciels.

Conteneurs: abstraction au niveau de la couche applicative qui regroupe le code et les dépendances. Plusieurs conteneurs peuvent fonctionner sur la même machine et partager le noyau du système d'exploitation avec d'autres conteneurs, chacun fonctionnant comme un processus isolé dans l'espace utilisateur.



Avantages des conteneurs:

■ Résolution du "It works on my machine"

- Le logiciel fonctionne sur votre ordinateur, vous l'envoyez sur le serveur et... 🚨
- → Les conteneurs résolvent ce problème en permettant au développeur d'exécuter personnellement l'application à l'intérieur d'un conteneur, qui inclut alors toutes les dépendances nécessaires au fonctionnement de l'application.

■ Environnements isolés

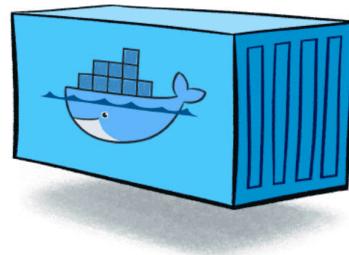
- Vous devez déployer 5 applications Python différentes qui nécessitent différentes versions de Python sur un serveur commun.
- → Les conteneurs empaquettent le logiciel avec toutes ses dépendances, vous empaquerez chacune des 5 applications avec leurs versions Python.

■ Développement

- Vous arrivez sur un projet qui nécessite l'utilisation d'autres services comme Postgres, MongoDB, Redis, ...
- → En une seule commande, vous obtenez une application isolée qui tourne sur votre machine.

■ Scaling

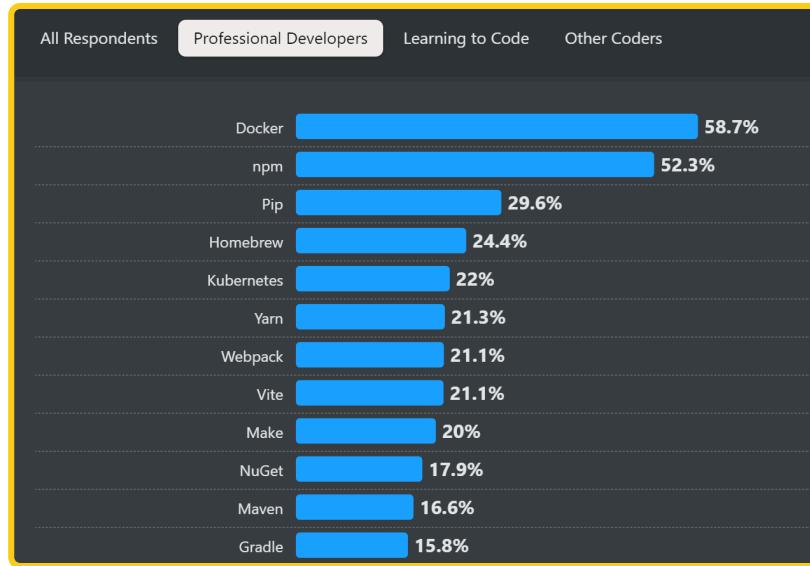
- Imaginez que vous lancer votre propre Netflix ou Facebook
- → Docker vient avec un système d'orchestration des conteneurs permettant la répartition de la charge.



Docker se base sur les 5 principes des conteneurs industriels utilisés pour le transport maritime:

- **Facilité de chargement:** n'importe quel expéditeur peut se procurer un conteneur et le charger à son rythme sachant qu'il sera accepté par le transporteur.
- **Amélioration de la manutention du conteneur** par sa standardisation (taille).
- **Augmentation de la vitesse** de transport (on transporte le conteneur et non plus les colis un par un).
- **Stockage optimisé** car les dimensions sont **normalisées**, permettant un empilement et un ajustement des espaces de stockage sans perte de place.
- **Interchangeabilité** partout dans le monde: un conteneur endommagé peut facilement être réparé ou remplacé sans retour à son expéditeur.

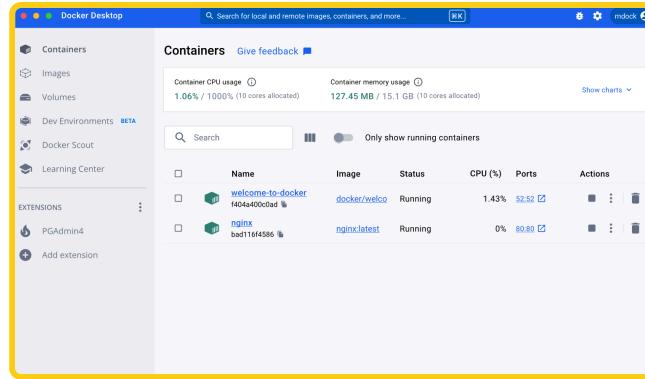
Docker est l'outil de compilation, de construction et de test le plus utilisé par les développeurs professionnels.



StackOverflow Survey 2024

Exécution de conteneurs

À vous de jouer: Installez l'application officiel Docker Desktop depuis <https://www.docker.com/products/docker-desktop> et créez un compte *Docker Personal*.



Assurez-vous que Docker CLI est bien disponible depuis votre terminal:

```
$ docker -v
Docker version 28.3.3, build 980b856
```

Créer et exécuter un nouveau conteneur à partir d'une *image* à l'aide de la commande:

```
$ docker container run [OPTIONS] IMAGE_NAME [COMMAND] [ARG ... ]  
$ docker run # Alias
```

À vous de jouer: Lancez un container à partir de l'image nommée `hello-world` et analysez l'output du terminal.

Résultat:

```
$ docker run hello-world      # docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:308866a43596e83578c7dfa15e27a73011bdd402185a84c5cd7f32a88b501a24
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

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

Maintenant exécutez à nouveau la même commande. Quelle différence voyez-vous?

Images et conteneurs

→ Les conteneurs sont des instances d'images.



Conteneur: un repas prêt à consommer, qu'il suffit de réchauffer et de manger.



Image: une recette ou les ingrédients de ce repas.

Image

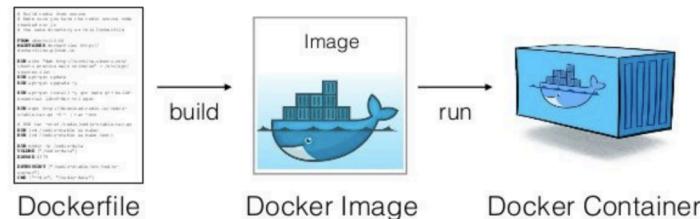
→ Une image Docker est un fichier. Une image ne change jamais; vous ne pouvez pas modifier un fichier existant. La création d'une nouvelle image se fait en partant d'une image de base et en y ajoutant de nouvelles couches (**Layers**).

À vous de jouer: Listez toutes vos images avec `docker image ls` :

```
$ docker image ls
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
hello-world    latest    d1165f221234  9 days ago   13.3kB
```

Dockerfile

Une image est construite à partir d'un fichier d'instructions nommé **Dockerfile** qui est analysé lorsque vous exécutez la commande `docker image build`.



```
# Dockerfile
FROM <image>:<tag>
RUN <installation de dépendances>
CMD <commande qui est exécutée lors de `docker container run`>
```

Conteneur

Listez tous vos conteneurs avec `docker container ls` :

```
$ docker container ls
CONTAINER ID    IMAGE        COMMAND      CREATED      STATUS      PORTS      NAMES
```

CLI de Docker: les bases

Commande	Description	Aliase
<code>docker image ls</code>	Liste toutes les images	<code>docker images</code>
<code>docker image rm <image></code>	Supprime une image	<code>docker rmi</code>
<code>docker image pull <image></code>	Pull une image à partir d'un <i>docker registry</i>	<code>docker pull</code>
<code>docker container ls -a</code>	Liste tous les conteneurs	<code>docker ps -a</code>
<code>docker container run <image></code>	Exécute un conteneur à partir d'une image	<code>docker run</code>
<code>docker container rm <container></code>	Supprime un conteneur	<code>docker rm</code>
<code>docker container stop <container></code>	Stop un conteneur	<code>docker stop</code>
<code>docker container exec <container></code>	Exécute une commande à l'intérieur du conteneur	<code>docker exec</code>

Docker CLI documentation

Exercice:

1. Démarrer 3 conteneurs à partir de l'image nommé `nginx` en mode détaché avec des noms définis (exemple: `nginx-1`).
2. Arrêtez 1 des conteneurs, supprimez en 1 définitivement et laissez en 1 en cours d'exécution.
3. Afficher dans votre terminal la liste des conteneurs arrêtés et en cours d'exécution de l'image `nginx` uniquement.

Réponse:

```
#/ Lancer un container  
$ docker run -d --name nginx-1 nginx  
  
#/ Stopper un conteneur  
$ docker container stop nginx-1  
  
#/ Supprimer un conteneur  
$ docker container rm -f nginx-1  
  
#/ Lister les conteneurs  
$ docker container ls -a -f ancestor=nginx
```

Exécution et arrêt des conteneurs

Lancez la commande suivante:

```
$ docker container run devopsdockeruh/simple-web-service:ubuntu
```

Inspectez les logs d'un conteneur *detached* avec la commande:

```
$ docker attach --sig-proxy=false <container>
```

Il est possible de se connecter à un conteneur de manière interactive à l'aide de la commande:

```
$ docker exec -it <container> COMMAND
```

- `-i, --interactive` : Garder STDIN ouvert
- `-t, --tty` : Allouer un pseudo-TTY

À vous de jouer: Connectez vous à votre container `devopsdockeruh/simple-web-service:ubuntu` et afficher le contenu du fichier `/usr/src/app/text.log`.

Analyse des images

Les images sont les blocs de construction de base pour les conteneurs et les autres images.
Lorsque vous "conteneurisez" une application, vous travaillez à la construction de l'image.

D'où viennent les images ?



```
$ docker search hello-world
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
hello-world	Hello World!...	1988	[OK]	
Kitematic/hello-world-nginx	A light-weig...	153		
tutum/hello-world	Image to tes...	90		[OK]
...				

<https://hub.docker.com/>

! Gardez à l'esprit que nous téléchargeons du contenu à partir d'Internet. Il est toujours bon de vérifier ce que vous utilisez.

Analyse détaillée d'une image

Chaque version d'image est associée à un tag:

```
$ docker pull ubuntu  
$ docker pull ubuntu:latest
```

Il est possible de taguer et renommer les images localement avec `docker tag`:

```
$ docker tag ubuntu:20.04 ubuntu:focal  
$ docker tag ubuntu:20.04 secret-linux-distro:focal
```

Le nom d'une image peut être composé de 3 parties plus un tag:

`registry/organisation/image:tag`

```
$ docker pull docker.io/library/ubuntu
```

Build une image

À vous de jouer: Créez un script nommé `hello.sh` dans un nouveau dossier vide:

```
#!/bin/sh
echo "Hello, docker!"
```

Ajoutez ensuite un `Dockerfile`:

```
# Utiliser l'image alpine comme base qui est une petite distribution Linux
FROM alpine:3.20
# Utiliser /usr/src/app comme répertoire de travail. Les instructions suivantes seront exécutées à cet endroit.
WORKDIR /usr/src/app
# Copier le fichier hello.sh de ce répertoire vers /usr/src/app/ en créant /usr/src/app/hello.sh
COPY hello.sh .
# Ajouter les permissions d'exécution pendant la construction.
RUN chmod +x hello.sh
# Lors de l'exécution de docker run, la commande sera ./hello.sh
CMD ./hello.sh
```

→ Nous pouvons utiliser la commande `docker build` pour transformer le Dockerfile en une image:

```
$ docker build . -t hello-docker
```

Assurons-nous que l'image existe et lancez là:

```
$ docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
hello-docker        latest   5f8f5d7445f3  4 minutes ago  7.73MB
```

Inspectez la présence du fichier dans notre image:

```
$ docker run -it hello-docker sh
```

Exercice

- Créez un nouveau fichier script.sh sur votre machine locale avec le contenu suivant :

```
while true
do
    echo "Input website:"
    read website; echo "Searching .."
    sleep 1; curl http://$website
done
```

- Créez un `Dockerfile` pour une nouvelle image à partir de `ubuntu:24.04` et ajoutez des instructions pour installer `curl` dans cette image. Configurez l'image pour que le script s'exécute au démarrage du conteneur et vous laisse entrer un site web:

```
Input website:
dfhi-isfates.eu
Searching ..
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
...
...
```

Différence entre les instructions CMD et ENTRYPOINT

- `ENTRYPOINT` doit être défini lorsque le conteneur est utilisé comme exécutable.
- `CMD` doit être utilisé pour définir les arguments par défaut d'une commande `ENTRYPOINT` ou pour exécuter une commande dans un conteneur.

```
FROM ubuntu:24.04

RUN apt-get update && apt-get install -y python3

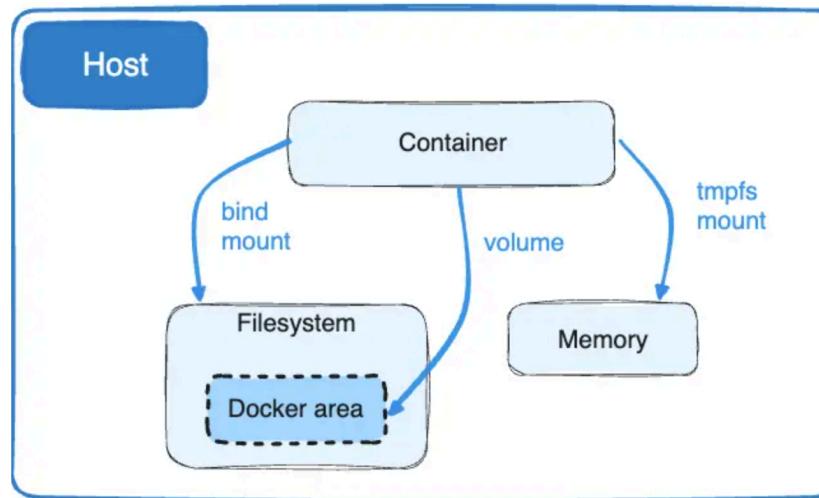
ENTRYPOINT ["python3"]
```

Stockage des données

Par défaut, tous les fichiers créés à l'intérieur d'un conteneur sont stockés sur une couche de conteneur accessible en écriture. Cela signifie que :

- Les données ne persistent pas lorsque le conteneur n'existe plus, et il peut être difficile de sortir les données du conteneur si un autre processus en a besoin.
- La couche inscriptible d'un conteneur est étroitement liée à la machine hôte sur laquelle tourne le conteneur. Il n'est pas facile de déplacer les données ailleurs.

- 2 options pour que les conteneurs stockent les fichiers sur la machine hôte de manière persistante: les **volumes** et les **bind mounts**.
- 1 option de stockage in-memory rendant le stockage non persistants: **tmpfs mount**.



Les volumes

1. Lancez un conteneur mysql :

```
$ docker run --name my-database -e MYSQL_ROOT_PASSWORD=my-secret-pwd -d mysql
```

2. Affichez les volumes:

```
$ docker volume ls
```

3. Inspectez le volume généré:

```
$ docker volume inspect <volume>
```

Les ports

L'ouverture d'une connexion entre le monde extérieur et un conteneur Docker se fait en deux étapes :

1. Exposer le port
2. Publier le port

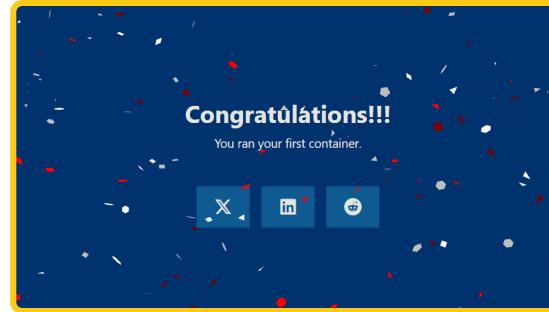
Exposer le port d'un conteneur signifie indiquer à Docker que le conteneur écoute sur un certain port. Pour exposer un port, ajoutez l'instruction `EXPOSE <port>` dans votre `Dockerfile`.

Pour publier un port, lancez le conteneur avec `-p <host-port>:<container-port>`.

À vous de jouer: Lancez un conteneur à partir de l'image `docker/welcome-to-docker` qui expose le port `80` sur votre port local `8080`.

Réponse:

```
docker run -d -p 8080:80 docker/welcome-to-docker
```



Publier une image

Publions ensemble notre image locale `hello-docker` :

1. Modifiez le fichier `hello.sh` pour le personnaliser et refaites un build de l'image.
2. Connectez-vous à <https://hub.docker.com> et cliquez sur `Create repository`.
3. Nommez le repository `hello-docker` et définir la visibilité comme **publique**.
4. Renommer votre image locale: `docker tag hello-docker <username>/hello-docker`.
5. Finalement publiez l'image: `docker push <username>/hello-docker`.
6. Lancez les conteneurs des autres: `docker run <username>/hello-docker`.

Best practices pour builder une image



Analyse des layers d'une image

En utilisant la commande `docker image history`, vous pouvez voir la commande qui a été utilisée pour créer chaque layer dans une image:

```
$ docker image history hello-docker
```

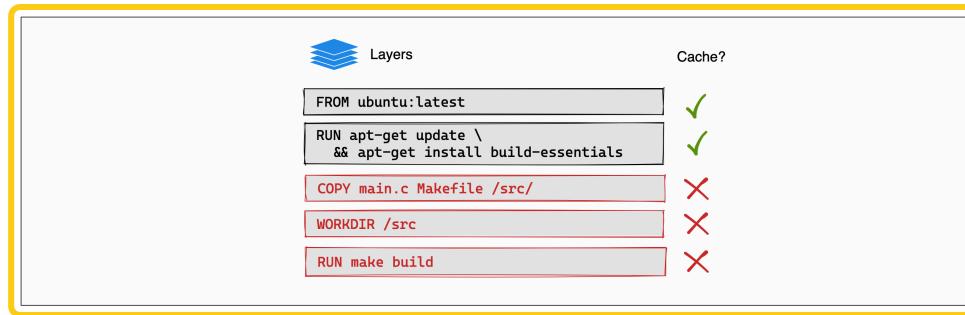
IMAGE	CREATED	CREATED BY	SIZE	COMMENT
270ca8d5e06b	4 days ago	CMD ["/bin/sh" "-c" "./hello.sh"]	0B	buildkit.dockerfile.v0
<missing>	4 days ago	RUN /bin/sh -c chmod +x hello.sh # buildkit	31B	buildkit.dockerfile.v0
<missing>	4 days ago	COPY hello.sh . # buildkit	31B	buildkit.dockerfile.v0

La taille de chaque layer est visible ce qui facilite le diagnostic des images de trop grande taille.

Ajoutez l'option `--no-trunc` pour éviter de tronquer les lignes.

Gérer le cache des layers

Lors de la construction d'une image, Docker parcourt les instructions de votre Dockerfile, exécutant chacune d'entre elles dans l'ordre spécifié. Pour chaque instruction, Docker vérifie s'il peut réutiliser l'instruction du cache de construction.



Si un layer est modifié, tous les autres layers qui le suivent doivent également être réexécutés.

→ Ordonnez les instructions en allant de celles qui sont modifiées le moins souvent à celles qui sont modifiées le plus souvent lorsque c'est possible.

✗ Installation des dépendances à chaque build:

```
FROM node
WORKDIR /app
COPY . .          # Copie des fichiers
RUN npm install   # Installer les dépendances
RUN npm build     # Lancer le build
```

✓ Installation des dépendances si nécessaire:

```
FROM node
WORKDIR /app
COPY package.json yarn.lock . # Copie des packages
RUN npm install               # Installer les dépendances
COPY . .                      # Copie des fichiers
RUN npm build                 # Lancer le build
```

Réduire la taille des layers

1. Ne pas inclure de fichiers inutiles

```
COPY . /src          # ❌ Copie de tout le context local (logs, artifacts et ancien builds)
COPY ./Makefile ./src /src # ✅ Copie explicite de fichiers ou dossiers
```

Utilisez un fichier `.dockerignore` pour exclure des fichiers ou des dossiers du contexte du build.

```
target/  
  
*.class  
*.log  
*.jar
```

2. Utilisez votre package manager efficacement

Veillez à n'installer que les paquets dont vous avez besoin. Si vous ne les utilisez pas, ne les installez pas.

Minimiser le nombre de layers

Moins de layers signifie moins de choses à reconstruire lorsque quelque chose change dans votre Dockerfile.

1. Utilisez une image de base appropriée

- postgres , python , maven , kibana , ...

2. Utilisez des multi-stage builds

- Diviser votre Dockerfile en plusieurs étapes distinctes
- Permet de séparer les dépendances de compilation des dépendances d'exécution
- Docker établit les dépendances entre les étapes et les exécute en utilisant la stratégie la plus efficace

Exemple de multi-stage build:

```
FROM maven AS build
WORKDIR /app
COPY . .
RUN mvn package

FROM tomcat
COPY --from=build /app/target/file.war /usr/local/tomcat/webapps
```

3. Combinez les commandes dans la mesure du possible

2 layers

```
RUN echo "la première commande"  
RUN echo "la deuxième commande"
```

1 seule layer

```
RUN echo "la première commande" && echo "la deuxième commande"  
#/ ou pour diviser en plusieurs lignes  
RUN echo "la première commande" && \  
    echo "la deuxième commande"
```

En résumé

- Définition et usage de la méthodologie DevOps
 - Virtualisation
 - Docker
- Images et conteneurs
 - Gestion et création d'une images
 - Lancement et développement de conteneurs
- Interaction avec les conteneurs
 - Volumes
 - Ports
- Publishing d'une image
 - Docker Hub

→ Après avoir créé une image, n'importe qui peut prendre et exécuter une application sans problème de dépendance ou de version.

Exercice:

À vous de jouer: Certains de nos coéquipiers développeurs ont créé une application avec un README qui indique ce qu'il faut installer et comment exécuter une application Ruby on Rails.

Ouvrez ce projet <https://github.com/isfates-l3-outils-et-methodes-de-dev/exercices/tree/main/rails-example-project>, lisez le README et créez un fichier Dockerfile permettant de lancer l'application web dans un conteneur.

Correction:

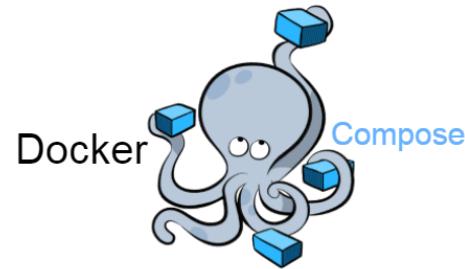
```
FROM ...
```

```
#
```

Orchestration de conteneurs avec Docker

Le démarrage et l'arrêt des conteneurs sont un peu pénibles, sans parler de l'exécution de deux applications en même temps. Si seulement il existait un moyen, un outil, pour simplifier... la **composition**.

Docker Compose



Docker Compose est un outil permettant de définir et d'exécuter des applications multi-conteneurs.

Docker Compose simplifie la gestion des services , des networks et des volumes dans un seul fichier de configuration YAML compréhensible communément appelé compose.yaml (vous pourrez également voir le nom docker-compose.yaml et docker-compose.yml).

Docker Compose fonctionne dans tous les environnements: production, staging, développement, tests et CI. Il dispose également de commandes permettant de gérer l'ensemble du cycle de vie de votre application:

- Démarrer, arrêter et reconstruire des services
- Afficher l'état des services en cours d'exécution
- Lire le journal des services en cours d'exécution
- Exécuter une commande unique sur un service

Format YAML

Yet Another Markup Language (YAML)

```
# An employee record
name: Martin D'vloper
job: Developer
employed: True
foods:
  - Apple
  - Orange
  - Strawberry
  - Mango
languages:
  javascript: Elite
  python: Elite
  java: Lame
education: |
  4 GCSEs
  3 A-Levels
```

Modèle d'application Compose

```
1  services:
2    frontend:
3      image: example/webapp
4      ports:
5        - "443:8043"
6      networks:
7        - front-tier
8        - back-tier
9      secrets:
10     - server-certificate
11
12    backend:
13      image: example/database
14      volumes:
15        - db-data:/etc/data
16      networks:
17        - back-tier
```

```
18    volumes:
19      db-data:
20        driver: flocker
21        driver_opts:
22          size: "10GiB"
23
24    secrets:
25      server-certificate:
26        external: true
27
28    networks:
29      front-tier: {}
30      back-tier: {}
```

 Documentation officielle

Services

```
1 services:
2   frontend:
3     image: example/webapp
4     ports:
5       - "443:8043"
6     networks:
7       - front-tier
8       - back-tier
9     secrets:
10      - server-certificate
11
12   backend:
13     image: example/database
14     volumes:
15       - db-data:/etc/data
16     networks:
17       - back-tier
```

```
18   volumes:
19     db-data:
20       driver: flocker
21       driver_opts:
22         size: "10GiB"
23
24   secrets:
25     server-certificate:
26       external: true
27
28   networks:
29     front-tier: {}
30     back-tier: {}
```

Un **service** est un concept abstrait mis en œuvre sur des plateformes en exécutant une ou plusieurs fois la même image de conteneur et la même configuration.

Networks

```
1  services:
2    frontend:
3      image: example/webapp
4      ports:
5        - "443:8043"
6      networks:
7        - front-tier
8        - back-tier
9      secrets:
10     - server-certificate
11
12    backend:
13      image: example/database
14      volumes:
15        - db-data:/etc/data
16      networks:
17        - back-tier
```

```
18    volumes:
19      db-data:
20        driver: flocker
21        driver_opts:
22          size: "10GiB"
23
24    secrets:
25      server-certificate:
26        external: true
27
28    networks:
29      front-tier: {}
30      back-tier: {}
```

Les services communiquent entre eux par l'intermédiaire de **networks**. Dans la spécification Compose, un network est une abstraction de capacité de plateforme permettant d'établir une route IP entre des conteneurs au sein de services connectés entre eux.

Volumes

```
1  services:
2    frontend:
3      image: example/webapp
4      ports:
5        - "443:8043"
6      networks:
7        - front-tier
8        - back-tier
9      secrets:
10     - server-certificate
11
12    backend:
13      image: example/database
14      volumes:
15        - db-data:/etc/data
16      networks:
17        - back-tier
```

```
18    volumes:
19      db-data:
20        driver: flocker
21        driver_opts:
22          size: "10GiB"
23
24    secrets:
25      server-certificate:
26        external: true
27
28    networks:
29      front-tier: {}
30      back-tier: {}
```

Les services stockent et partagent des données persistantes dans des **volumes**.

Secrets

```
1  services:
2    frontend:
3      image: example/webapp
4      ports:
5        - "443:8043"
6      networks:
7        - front-tier
8        - back-tier
9      secrets:
10     - server-certificate
11
12    backend:
13      image: example/database
14      volumes:
15        - db-data:/etc/data
16      networks:
17        - back-tier
```

```
18    volumes:
19      db-data:
20        driver: flocker
21        driver_opts:
22          size: "10GiB"
23
24    secrets:
25      server-certificate:
26        external: true
27
28    networks:
29      front-tier: {}
30      back-tier: {}
```

Un **secret** est une forme spécifique de données de configuration pour les données sensibles qui ne doivent pas être exposées sans considérations de sécurité. Les secrets sont mis à la disposition des services sous forme de fichiers montés dans leurs conteneurs.

Version

La propriété `version` est définie par la spécification Compose à des fins de rétrocompatibilité mais est **obsolète**.

```
version: '3.8'  
  
services:  
  ...
```

Ce format fusionne les formats de versions `2.x` et `3.x` du modèle Compose qui ne sont plus activement maintenus.

Docker compose CLI

- `docker compose up` - build les images si elles n'existent pas et démarre tous les services définis dans des conteneurs.
- `docker compose down` - arrête et supprime tous les conteneurs, réseaux et volumes créés par `docker compose up`.
 - `docker compose down --volumes` - supprime les volumes nommés déclarés dans la section des volumes
- `docker compose build` - build les images de tous les services.
- `docker compose ps` - liste tous les conteneurs en cours d'exécution pour les services définis.
- `docker compose start <services>` - démarre les conteneurs pour les services définis.
- `docker compose stop <services>` - arrête les conteneurs pour les services définis.
- `docker compose restart <services>` - redémarre les conteneurs pour les services définis.
- `docker compose exec <service> <commande>` - exécute une commande à l'intérieur d'un conteneur en cours d'exécution pour un service défini.
- `docker compose --help` - et bien plus encore...

Exercice



Nous allons créer un environnement de développement en local permettant de lancer un site Wordpress sur une base de donnée MariaDB. Il devra être possible d'administrer la base de donnée à l'aide de phpMyAdmin.

1. Créez dans un nouveau dossier un fichier `compose.yaml` un service nommé `db` qui:

- se base sur l'image `mariadb:latest`
- stocke ses données de manière persistante dans un `volume` sous le dossier `/var/lib/mysql`
- définit ses variables d'environnements (`environment`):
 - `MYSQL_RANDOM_ROOT_PASSWORD: 1`
 - le nom de la database
 - un nom d'utilisateur
 - un mot de passe pour l'utilisateur
- définit son `hostname`
- doit toujours redémarrer (`restart`) s'il s'arrête de manière involontaire

```
#/ Correction
```

```
#
```

2. Déplacez `MYSQL_DATABASE` , `MYSQL_USER` et `MYSQL_PASSWORD` dans un fichier `.env`

3. Ajoutez un service nommé `phpmyadmin` qui:

- se base sur l'image `phpmyadmin/phpmyadmin`
- se connecte à l'host du service `db`
- est accessible en local sur le port `8080`
- doit toujours redémarrer (`restart`) s'il s'arrête de manière involontaire

Conteneurisation avec Docker

```
#/ Correction
```

```
#
```

4. Ajoutez un dernier service nommé `wordpress` qui:

- se base sur l'image `wordpress`
- stocke ses données de manière persistante dans un `volume` sous le dossier `/var/www/html/wp-content`
- se connecte à l'host du service `db` sur la database spécifié avec le bon utilisateur et mot de passe
- est accessible en local sur le port `80`
- doit toujours redémarrer (`restart`) s'il s'arrête de manière involontaire

Conteneurisation avec Docker

```
#/ Correction
```

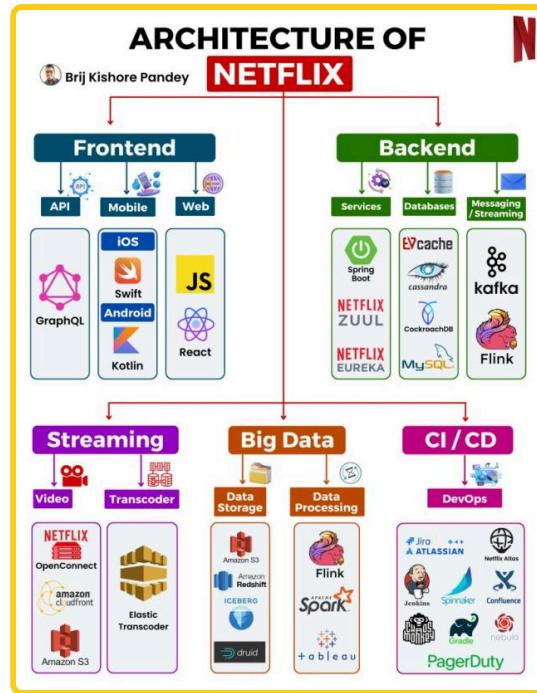
```
#
```

Conteneurs et développement

Les conteneurs ne sont pas seulement utiles en production, ils peuvent également être utilisés dans des environnements de développement.



Imaginez juste devoir installer toutes les dépendances d'un tel projet lors de votre premier jours:



Spécification de build

Un service dans un fichier `compose.yaml` peut démarrer un conteneur à partir d'une image:

```
services:  
  frontend:  
    image: org/webapp
```

mais peut aussi démarrer un conteneur défini par un Dockerfile localisé dans `build`:

```
services:  
  frontend:  
    build: ./path-to-dockerfile-folder  
    command: ls
```

Exercice

À vous de jouer: Dans votre repo local de <https://github.com/isfates-l3-outils-et-methodes-de-dev/exercices>, lisez le `README` du projet `node-dev-env` et créez un fichier `Dockerfile` qui installe les dépendances nécessaires.

```
#/ Correction
```

```
#
```

À vous de jouer: Ajoutez maintenant un fichier `compose.yaml` composé d'un service `node-dev-env` qui:

- se base sur l'image locale défini par le Dockerfile
- stocke ses données de manière persistante dans un `volume` sous le même dossier que défini dans l'image
- démarre le projet qui sera rendu accessible depuis le port `3000`

```
#/ Correction
```

```
#
```

Utilisation de Compose Watch

Fraîchement sorti fin 2023, Docker Compose Watch permet de synchroniser automatiquement votre code source local avec le code de votre conteneur Docker sans avoir besoin d'utiliser des volumes!

```
services:  
  web:  
    build: .  
    develop:  
      watch:  
        - action: sync  
          path: ./apps/web  
          target: /app/apps/web  
        ignore:  
          - node_modules  
        - action: rebuild  
          path: package.json  
        - action: sync+restart  
          path: ./apps/web/next.config.js  
          target: /app/apps/web/next.config.js
```

Conteneurisation avec Docker

L'action `sync` spécifie un chemin à surveiller pour les changements dans le système de fichiers hôte, et un chemin cible correspondant dans le conteneur pour **synchroniser les changements**.

L'action `build` spécifie un chemin d'accès pour surveiller les changements dans le système de fichiers de l'hôte et déclenche une **reconstruction du conteneur** lorsque des changements sont détectés.

L'action `sync+restart` synchronisera d'abord les répertoires, puis redémarrera immédiatement votre conteneur sans le build.

Pour build et lancer un projet Compose et démarrer le mode watch:

- `docker compose up --watch`
- `docker compose watch` - si vous ne souhaitez pas mélanger les logs de l'application avec les logs de build et les événements de synchronisation du système de fichiers.

À vous de jouer: Modifiez le fichier `compose.yaml` pour utiliser Compose Watch à la place des volumes:

```
#/ Correction
```

```
#
```

Docker Compose en production

Docker compose est réputé pour son utilisation en mode développement, mais une telle orchestration de conteneurs est tout autant utile dans différents environnements tels que CI, staging et production.

C'est pourquoi vous pouvez envisager de définir un fichier Compose supplémentaire, par exemple `production.yml`, qui spécifie la configuration adaptée à la production:

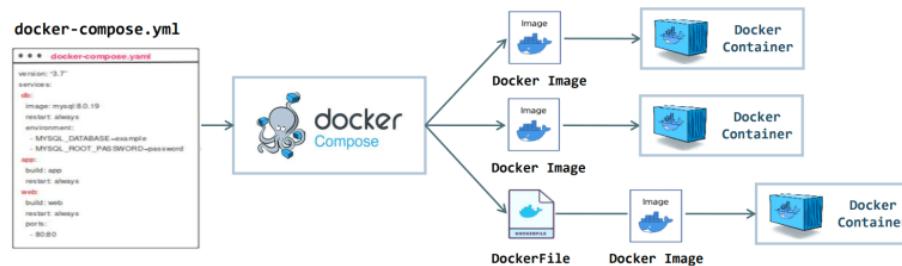
- Suppression des liaisons de `volume` non nécessaire
- Se lier à différents ports sur l'hôte
- Définir les variables d'environnement de manières plus sécurisé ([Docker Compose secrets](#))
- Spécifier une politique de redémarrage comme `restart: always` pour éviter les temps d'arrêt
- Ajouter des services supplémentaires tels qu'un agrégateur de logs.

Une fois que vous avez un deuxième fichier de configuration, vous pouvez le merger avec l'original avec l'option `-f` :

```
$ docker compose -f compose.yml -f production.yml up
```

En résumé

- Définition manuelle de conteneurs vers une composition
 - Docker Compose
 - Gestion de services dans un Dockerfile
- Docker et développement
 - Build d'images locales avec un Dockerfile
 - Partage de fichiers avec les volumes
 - Docker Watch
- Docker Compose en production
 - Merge de fichier de configuration



Ressources

Ce cours a été élaboré à partir des ressources suivantes :

- Cours précédents de [in Gary Zingraff](#)
- <https://devopswithdocker.com/getting-started>
- <https://aws.amazon.com/fr/compare/the-difference-between-docker-vm/>
- <https://www.geeksforgeeks.org/what-is-docker-daemon/>

