Docker Avancée

DERNIÈRE MISE À JOUR : 21/01/2024

Evolution de l'industrie logicielle

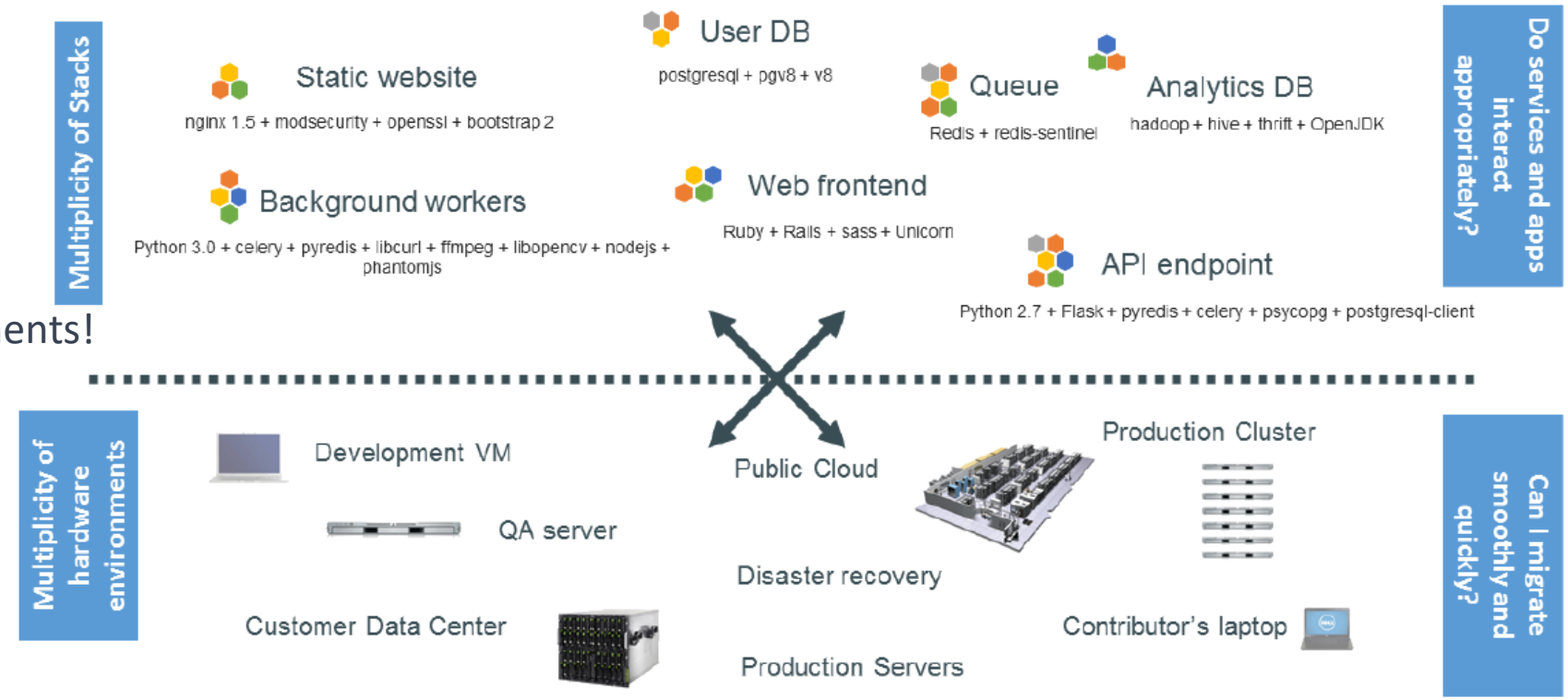
- Avant
  - Applications monolitiques
  - Cycles de dev long
  - Un seul environnement
- Après
  - Ensemble de services
  - Améliorations rapides, cycles itératifs
  - De nombreux environnements
    - Des serveurs de prod (dans le cloud?)
    - Des laptops de dev
    - Des serveurs de test (gitlab?)
    - etc.

Le déploiement devient complexe

Des piles logicielles différentes

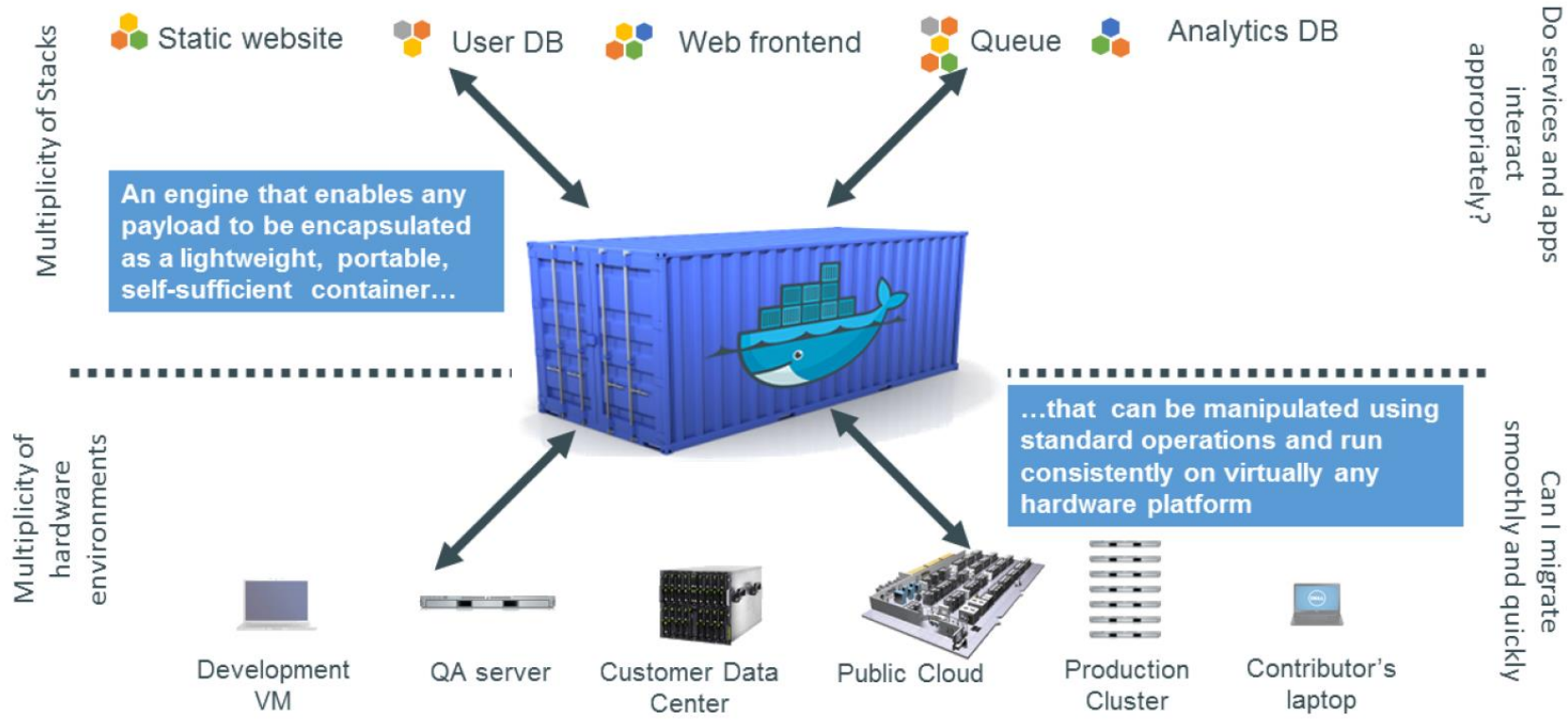
- Langages
- Frameworks
- Databases

De nombreux environnements!

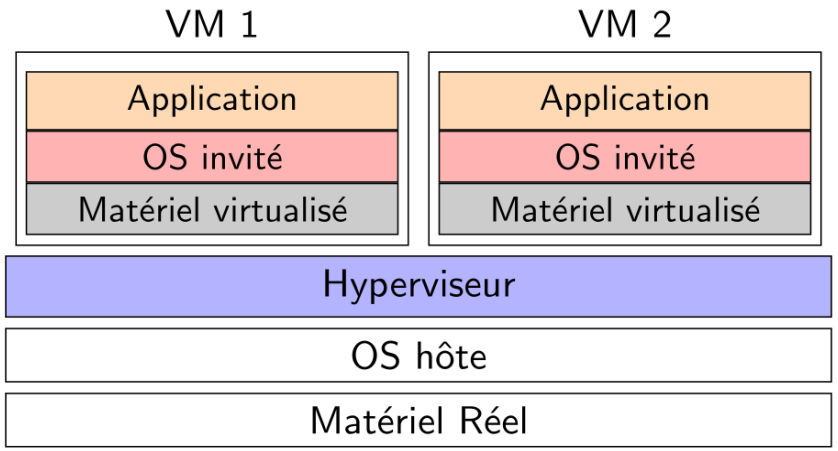


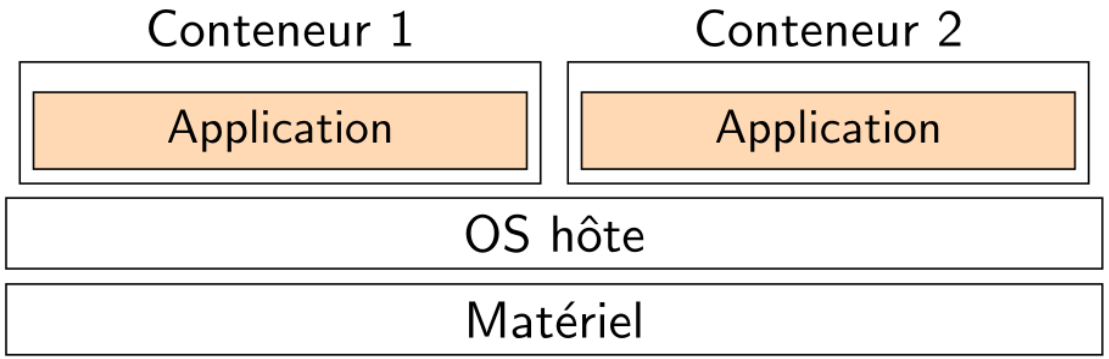
Solution : Des conteneurs pour le logiciel

Manipulation simplifiée d'un ensemble d'objets (ou d'applications) grâce à une interface standardisée.



VM vs conteneurs





- Une image de machine virtuelle contient un OS complet

- Une image de conteneur ne contient que les bibliothèques nécessaires au composant continuers
- Environnement d'exécution isola au dessus du système d'exploitation

VM

- Avantages
  - Emulation bas niveau
  - Sécurité/compartimentation forte hôte/VMs et
  - VMs/VMs
- Inconvénients
  - Usage disque important
  - Impact sur les performances

conteneur

- Avantages:
  - Espace disque optimisé
  - Impact quasi nul sur les performances CPU, réseau et I/O
- Inconvénients
  - Fortement lié au kernel Hôte
  - Ne peut émuler un OS différent que l’hôte
  - Sécurité

Avantages et usages des conteneurs

- Utilisation pour de l'intégration continue (CI)
  - Environnement de test créé à partir d'une image Docker
    - Les tests peuvent être exécutés sur n'importe quelle plateforme (plateforme de CI)
  - Un nouveau conteneur créé pour chaque étape de tests
    - Pas de polution entre les étapes d'un test
    - Pas de polution entre plusieurs exécutions des tests
  - Les tests peuvent être exécutés très souvent
- Découplage de la "plomberie" et de la logique applicative
  - Utiliser des noms de services dans votre code (bd, api, etc)
  - Utiliser une composition (ou un orchestrateur si sur plusieurs serveurs) pour démarrer votre application
  - On peut redimensionner, faire de l'équilibrage de charge, répliquer sans changer lecode

Avantages et usages des conteneurs

- Passage du dev à la production (DevOps)
  - Créer une image de conteneur et une composition
  - L'image peut être directement déployé en production
    - Même environnement pour le dev, le test, et la production
    - Deploiement simplifié
    - Les devs peuvent se charger du déploiement
  - Le passage en production est fluidifié
- Infrastructure as code
  - L'infrastructure est décrite dans des fichiers texte
    - Les Dockerfile (= du code)
    - Sert aussi de documentation de l'infrastructure
  - Cette description peut etre versionnée (dépot git)
    - Suivi des modificiations
    - Possibilité de revenir en arrière
  - Mise en place automatique de l'infrastructure
    - Déterministe
    - Reproductible

Services fournis par Docker

- Construction d'images
- Gestion d'images
  - Localement sur une machine
  - Globalement (Registre -- Docker hub)
- Exécution et gestion de conteneurs

Qu'est ce qu’un conteneur

- Les technologies de conteneurs offrent une solution de packaging pour une application et ses dépendances.

- Les images de conteneurs
  - Package de l'application et de ces dépendances
  - Peut être exécutée sur différents environnements
  - Décrites par un fichier texte.
    - Infrastructure-as-code
- Un Conteneur
  - Une instance d'une image de conteneur
    - S'exécute dans un environnement isolé
- Analogie POO
  - Une image = une classe
  - Un conteneur = une instance

Docker: les briques principales

Docker engine

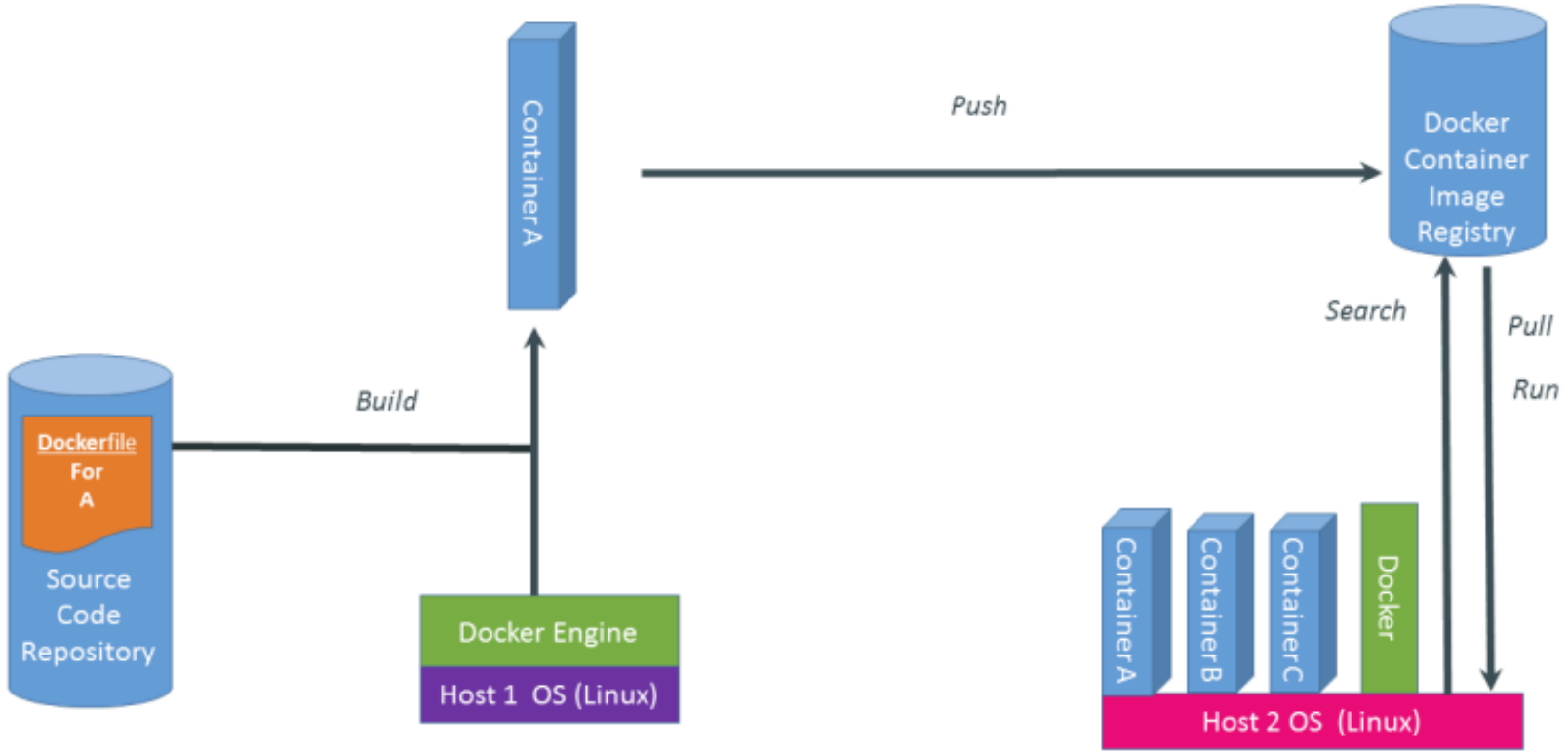
Un environnement d'exécution et un ensemble de services pour manipuler des conteneurs docker sur une machine

- Une application client-serveur
  - Le serveur -- Un daemon (processus persistant) qui gère les conteneurs sur une
  - Machine -- Le client -- Une interface en ligne de commande

Un/des registres d'images docker

- Bibliothèque d'images disponibles
  - Un serveur stockant des images docker
  - Possibilité de récupérer des images depuis ce serveur (pull)
  - Possibilité de publier de nouvelles images (push)
- Docker Hub
  - Dépôt publique d'images Docker

Principes de fonctionnement de Docker

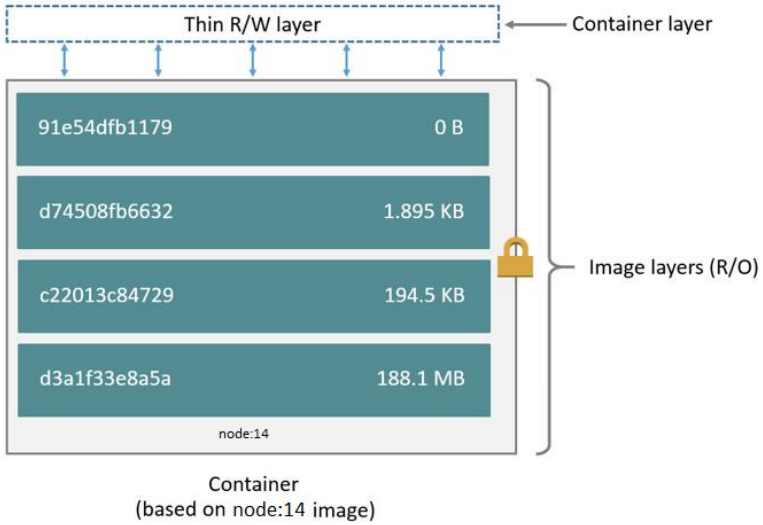


Un ensemble de concepts de composants

- Layers
- Stockage
- Volumes
- Réseau
- Publication de ports
- Service Discovery

Layers, couches

- Une image est composée d'un ensemble de couches (layers)
- Les layers peuvent être réutilisés entre différents conteneurs
- Chaque couche correspond à une instruction dans le fichier Dockerfile décrivant l'image.
- Gestion optimisée de l'espace disque.




```
# Étape 1: Définir l'image de base
FROM node:14

# Étape 2: Définir le répertoire de travail dans le conteneur
WORKDIR /usr/src/app

# Étape 3: Copier les fichiers de dépendances
COPY package*.json ./

# Étape 4: Installer les dépendances de l'application
RUN npm install

# Étape 5: Copier les fichiers source de l'application dans le conteneur
COPY . .

# Étape 6: Exposer le port sur lequel l'application va s'exécuter
EXPOSE 3000

# Étape 7: Définir la commande pour démarrer l'application
CMD ["node", "app.js"]
```

Layers, couches

- **Images de base**
  - Toute image est définie à partir d'une image de base
  - Des images de base officielles sont déjà fournies
  - Exemples: ubuntu:latest, ubuntu:14.04, opensuse:latest, alpine:latest
    - alpine: Image de base minimaliste (5MB) -- intéressant pour les performances
- **Relation avec le système d'exploitation hôte**
  - Une image n'inclut que les bibliothèques et services du système d'exploitation mentionné
  - Le conteneur utilise le noyau du système hôte

Layers, couches

- Les couches correspondent aux différentes modifications qui sont faites pour construire l'image à partir de l'image de base.
  - Pour sauvegarder une nouvelle image, il suffit de sauvegarder les nouvelles couches qui ont été créées au-dessus de l'image de base
  - Chaque couche capture les écritures faites par une commande exécutée lors de la création de l'image
  - Faible espace de stockage utilisé
- Dans un conteneur en cours d'exécution, il existe une couche supplémentaire accessible en écriture
  - Toutes les écritures vers le système de fichier faites à l'exécution du conteneur sont stockées dans cette couche.
  - Les autres couches, définies au sein de l'image utilisée pour instancier le conteneur, ne sont accessibles qu'en lecture.
  - Cette couche est supprimée à la suppression du conteneur.

Plus d'informations sur les couches

- Affichage de l'ensemble des couches d'une image

  $ docker history image_name

- Avantages liés aux mécanismes de couches
  - Chaque couche est stockée une seule fois localement
  - Si certaines couches nécessaires pour une image à télécharger sont déjà présentes, pas besoin de les télécharger à nouveau.
  - Réduction de l'espace de stockage
- Optimisations à l'exécution
  - Démarrage rapide : Démarrer un conteneur nécessite simplement de créer la couche accessible en écriture
  - Faible utilisation de l'espace disque: Si plusieurs containers sont instanciés à partir de la même image, ils partagent les couches en lecture seule.

Les namespace d’images

- 3 namespaces
  - les images officielles : Images sélectionnée par Docker Inc, accessible via Docker Hub, mais en générales produites et maintenues par les développeurs du logiciel ou l'organisation
    - Des images de distributions à utiliser comme images de base (ex: ubuntu,)
    - Des composants prêts à être utilisés (ex: Pyhton, redis, mysql)
    - Des petites images couteau suisse (ex: Alpine, busybox)
  - Les images d'utilisateurs/organisations
    - ex: tropars/myapp (nom de l'utilisateur / nom de l’image)
  - Les images hébergées (en dehors de docker hub): Elles contiennent le hostname ou l'adresse IP, et optionnellement le numéro de port du serveur.
    - ex: registry.example.com:5000/my-private/image

Liste d'images stockées localement

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
debian              latest              a6916e41aa87        10 days ago         117MB
hello-world         latest              d2c94e258dcb        8 months ago        13.3kB
```

Chercher des images

```
$ docker search python
NAME                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
python              Python is an interpreted, interactive, objec…   9371      [OK]
pypy                PyPy is a fast, compliant alternative implem…   385       [OK]
hylang              Hy is a Lisp dialect that translates express…   59        [OK]
bitnami/python      Bitnami Python Docker Image                     27                   [OK]
```

Télécharger des images

- docker pull: télécharge une image explicitement
- docker run: Commande servant à créer un conteneur à partir d'une image
  - Télécharge l'image si elle n'est pas présente localement
- Pas besoin de tags quand
  - On prototype/teste
  - On veut la dernière version d'une image
- Tags à utiliser quand
  - Quand on va en production
  - Pour s'assurer que la même version va être utilisée partout
  - Pour avoir de la reproductibilité

Chercher des images

```
$ $docker pull redis
Using default tag: latest
latest: Pulling from library/redis
a2abf6c4d29d: Pull complete
c7a4e4382001: Pull complete
4044b9ba67c9: Pull complete
c8388a79482f: Pull complete
413c8bb60be2: Pull complete
1abfd3011519: Pull complete
Digest: sha256:db485f2e245b5b3329fdc7eff4eb00f913e09d……
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest]
```

Créer des images

2 manières:

- Interactivement
- En utilisant un Dockerfile

Construire une image interactivement

Objectifs

Créer une image à partir d'une image de base dans laquelle nous allons installer cowsay

Les étapes

- Créer un conteneur à partir de l'image de base
- Installer le logiciel manuellement dans le conteneur et en faire une nouvelle image

Construire une image interactivement

Démarrer un conteneur Ubuntu

```
$ docker run -it ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
ea362f368469: Pull complete
Digest: sha256:b5a61709a9a44……
Status: Downloaded newer image for
ubuntu:latest
root@f461e2e7afff:/#
```

f461e2e7afff est l'id du conteneur créé

Installer le programme dans le conteneur

```
root@f461e2e7afff:/# apt-get update
root@f461e2e7afff:/# apt-get install python3
```

Quitter la session interactive:

```
root@f461e2e7afff:/# exit
```

Inspecter les changements:

```
 docker diff f461e2e7afff
C /var
C /var/log/apt
A /etc/python3.10
A /etc/python3.10/sitecustomize.py
…….
```

C: fichier ou répertoire modifié

A: fichier ou répertoire ajouté

Construire une image interactivement

Sauvegarder les changements dans une nouvelle image

```
$ docker commit <yourContainerId> <newImageId>
```

Sauvegarde les changements dans une nouvelle couche et sauvegarde l'image

Exécution de la nouvelle image

```
docker run -it <newImageId>
root@7267696dc8c6: / python3
```

Python3 est lancé

On peut tagger une image pour lui associer un nom

```
docker tag <newImageId> ubuntu_python
```

On peut aussi tagger lors de la création de l'image

```
docker commit <containerId> ubuntu_python_2
```

L'image peut maintenant être exécutée en utilisant ce nom:

```
docker run -it ubuntu_python_2
```

Processus long et source d'erreurs
Difficulté de Reproduction

Construire une image avec Dockerfile

Possibilité de construire son image à la main (long et source d'erreurs)

Suivi de version et construction d'images de manière automatisée

Utilisation de Dockerfile afin de garantir l'idempotence des images

Dockerfile :

Un *Dockerfile* est une recette décrivant comment construire une image

Contient une suite d'instructions qui définit une image

Permet de vérifier le contenu d'une image

Le commande docker build permet de créer une image à partir d'un Dockerfile

DockerFile : Éléments de syntaxe

FROM : Définit l'image de base à partir de laquelle la nouvelle image est créée.

LABEL : Associe des métadonnées à la nouvelle image (par exemple, l'auteur de l'image).

RUN : Exécute une commande dans une nouvelle couche au-dessus de l'image courante lors de la construction de l'image.

ENV : Définit des variables d'environnement qui seront disponibles dans le conteneur à son lancement.

WORKDIR : Définit le répertoire de travail dans le conteneur. Si le répertoire n'existe pas, il sera créé.

EXPOSE : Informe Docker que le conteneur écoutera sur les ports réseau spécifiés.

COPY : Copie un fichier ou un répertoire depuis le contexte de construction de l'image vers la nouvelle couche.
La destination peut être un chemin absolu ou relatif par rapport au WORKDIR.

CMD : Définit la commande par défaut exécutée au démarrage du conteneur.
Cette commande peut être remplacée en fournissant une commande différente lors de l'exécution du conteneur.

ENTRYPOINT : Définit la commande de base (ou "préfixe") lancée par le conteneur.
Contrairement à CMD, les arguments supplémentaires fournis lors de l'exécution du conteneur sont ajoutés à cette commande, plutôt que de la remplacer.

Les commandes de Dockerfile ne sont pas semsibles à la casse. On les note en majuscule par convention (facilite la lecture)

Notre premier Dockerfile

La création d'un Dockerfile doit se faire dans un nouveau répertoire

Créer le fichier Dockerfile

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install python3
```

Les commandes RUN doivent être non-interactives : L'option -y de apt évite qu'il demande si on est sur de vouloir installer

Construire l' image

```
docker build -t ubunty_python_3 .
```

-t permet de tagger avec un nom l'image qui va être créée
. indique le contexte de construction de l'image (où se trouve le Dockerfile)

```
$ docker build -t ubunty_python_3 .
Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM ubuntu
 ---> d13c942271d6
Step 2/3 : RUN apt-get update
 ---> Running in 80f5510281d9
Removing intermediate container 80f5510281d9
 ---> cb1643c4393c
Step 3/3 : RUN apt-get -y install python3
 ---> Running in 01834650511b
Removing intermediate container 01834650511b
 ---> 9ca55c5ccc54
Successfully built 9ca55c5ccc54
Successfully tagged ubunty_python_3 :latest
```

Que se passe-t-il?

Le « build context » est envoyé vers le démon docker (contenu du répertoire .)

A chaque étape:

- Un conteneur est créé pour exécuter l'étape (Running in ...)
- Les modifications sont committées dans une nouvelle image (---> ...)
- Le conteneur est supprimé
- La nouvelle image est utilisée pour la prochaine étape

Optimisation du Dockerfile : Éviter le Bad Layering

- Combinez les Commandes RUN :
  - Utilisez une seule instruction RUN pour les mises à jour et les installations.
  - Exemple : RUN apt-get update && apt-get install -y python3.
- Réduisez les Couches :
  - Chaque instruction RUN crée une nouvelle couche.
  - Moins d'instructions RUN signifie moins de couches et donc une image plus petite.
- Nettoyage du Cache :
  - Supprimez les fichiers de cache après l'installation des paquets.
  - Ajoutez rm -rf /var/lib/apt/lists/* à la fin de l'instruction RUN.

```
FROM ubuntu
RUN apt-get update && \
    apt-get install -y python3 && \
    rm -rf /var/lib/apt/lists/*
```

Syntaxe shell vs exec

- Les commandes telles que RUN ou CMD ont 2 syntaxes possibles

- **La syntaxe shell**

  ```
  RUN apt-get install python3
  ```

- Le commande est exécutée dans un shell
  /bin/sh -c est utilisé par défaut
  - →Plus facile à lire
  - →Interprète les expressions shell (ex: $HOME)
- **La syntaxe exec**

  ```
  RUN ["apt-get", "install", "cowsay"]
  ```

- Commande parsée en JSON et exécutée directement
  Nécessite " pour chaque chaine de caractères
  - → N'essaye pas d'interpréter les arguments
  - →Ne nécessite pas que /bin/sh soit présent dans l'image

CMD et ENTRYPOINT : cas d'utilisation

- A propos de CMD
  - Défini la commande par défaut à exécuter dans le conteneur quand aucune commande n’est fournie à l'exécution de *docker run*
  - Peut être insérée à n'importe quel endroit dans le Dockerfile mais seule la dernière est conservée
- A propos de ENTRYPOINT
  - Permet de définir une commande toujours exécutée par le conteneur
  - Il est recommandé d'utiliser la syntaxe exec
  - Avec la syntaxe shell, le passage de parametres à /bin/sh risque d'être incorrect

CMD et ENTRYPOINT : cas d'utilisation

- CMD

```
FFROM ubuntu
RUN apt-get update
RUN apt-get -y install python3
CMD ["python3"]
```

```
docker build -t ubuntu_python3 .
docker run -it ubuntu_python3
docker run -it ubuntu_python3 echo
Salut
```

- Entrypoint

```
FFROM ubuntu
RUN apt-get update
RUN apt-get -y install python3
ENTRYPOINT  ["python3"]
```

```
docker build -t ubuntu_python3 .
docker run -it ubuntu_python3 -c
"print(1+1)"
```

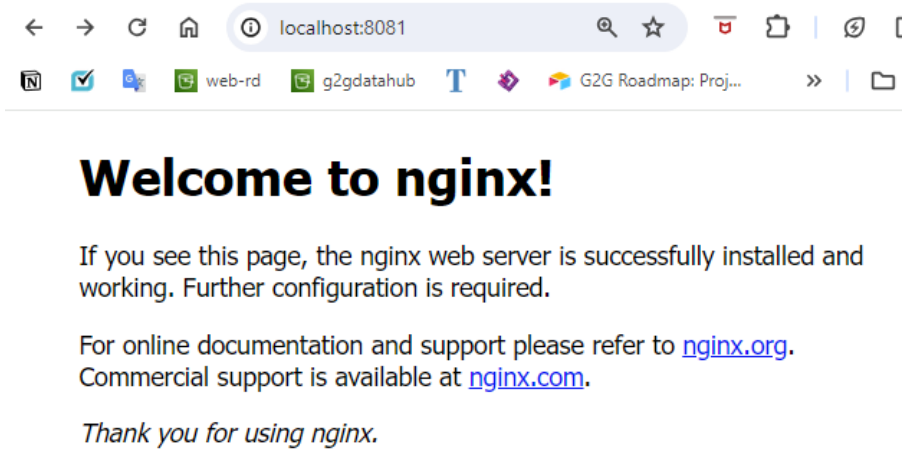
Exposer les ports d’un conteneur

- Tous les ports sont privés par défaut
  - Un port privé n’est pas accessible de l’extérieur
- C’est au client à rendre publics ou non les ports exposés
  - Public: accessible par d’autres conteneurs et en dehors de l’hôte.

```
docker run -d -p 8081:80 nginx
```

Port de l’hôte

Port interne du container



Exposer les ports d’un conteneur

```
FROM debian
RUN apt-get update
RUN apt-get install -y nginx
EXPOSE 80 443
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

- Build et run

```
docker build -t debian_ubuntu .
docker run -it -P -d debian_ubuntu
```

```
docker run -it –P 8181:80 -d debian_ubuntu
```

- Verifier les ports du conteneur

```
docker port id_conteneur
```

- Test de exposition du port

```
curl http://localhost:8181
```

COPY

- L’instruction COPY permet de copier fichiers et dossiers depuis le contexte de génération, dans le conteneur.
- Ex `COPY index.html /var/www/html/index.html`

ADD

- L’instruction ADD fonctionne comme COPY, avec des fonctionnalités en plus:
  - Peut récupérer des fichiers en ligne (URL http://)
  - Peut automatiquement décompresser des zip locaux

Gestion des Modifications : Défis et Solutions

- Si je veux modifier index.html, je dois regénérer une image,
  - Une image est en lecture seule → reconstruire une nouvelle image avec les modifications
  - laborieux surtout en phase de test
- Nginx génère des logs
  - Ces modifications engendrent des données dans une couche
  - Je voudrais les partager avec un autre serveur (ex ELK)

Volume

- Les volumes peuvent être partagés:
  - entre conteneurs
  - entre hôte et un conteneur
- Les accès au système de fichiers via un volume outre passent le Copy-on-Write des layers :
  - Meilleures performances
  - Ne sont pas enregistrés dans une couche pour ne pas être enregistrés par un docker commit

Volume

```
docker run -d -v $(pwd):/var/www/html –p 8181:80 debian_ubuntu # { }
```
powershell

```
docker exec –it id_conteneur ls /var/www/htlm
docker port id_conteneur 80
```

```
curl http://localhost:8181
echo "Mise à jour du fichier index.html" > index.html
curl http://localhost:8181
```

Création de Volume nommé

```
docker volume create --name=logs l
docker run -d -v logs:/var/log/nginx  -v ${PWD}:/var/www/html  -p 8282:80 debian_ubuntu

docker exec –it id_conteneur ls /var/www/htlm
docker port id_conteneur 80

curl http://localhost:8181
echo "Mise à jour du fichier index.html" > index.html
curl http://localhost:8181
```

Persistence des volumes

- Les volumes existent indépendamment des conteneurs
- Si un conteneur est stoppé, ses volumes sont encore disponibles
- Vous êtes responsable de la gestion, de la sauvegarde des volumes

```
docker volume ls
DRIVER VOLUME NAME
local 57a0848c5e5f2924.....
local logs
```

- On peut monter ces volumes depuis un autre conteneur

```
docker run -it --volumes-from id_conteneur_source debian
```

- Ménage des Volumes

```
docker rm 055ac104acf1d734 #supprime le conteneur Docker
docker volume ls -f dangling=true | grep logs # liste les volumes non attachés à un conteneur
docker volume rm logs
```

Network

- Nous avons déjà vu que les conteneurs pouvaient exposer leur port
- Les réseaux Docker est un autre moyen pour interconnecter des conteneurs

Un réseau docker est un switch virtuel

- Crée un sous réseau avec une plage d'addresse IP privée (172.17.0.0/16 par défaut)
  - Une addresse IP est associée à chaque conteneur
  - Chaque conteneur a son espace de nommage privé pour ces numéros de ports
- Un service DNS se charge de la résolution de noms
- Des mécanismes de NAT (Network Address Translation) permettent de router le traffic entrant sur la machine vers le bon conteneur
- Un réseau docker est implémenté par un driver

Possibilité de créer plusieurs réseaux docker sur une machine

- Des réseaux différents peuvent permettre d'isoler des conteneurs s'exécutant sur la même machine
- Les conteneurs peuvent être connectés à plusieurs réseaux

Network

- Quand vous installez Docker, 3 réseaux sont créés automatiquement, bridge, none, et host, suivant 3 pilotes bridge, null et host.

```
$ docker network ls
NETWORK ID NAME DRIVER
7fca4eb8c647 bridge bridge
9f904ee27bf5 none null
cf03ee007fb4 host hostocal
```

- none:
  - type null: auun réseau pour un conteneur sur un réseau de ce type
  - Sert à isoler le conteneur du réseau
- host
  - type host: Utilise le réseau de la machine hôte directement

Vous n'aurez sûrement jamais à utiliser ces réseaux, et créer des réseaux de ces types.

Network

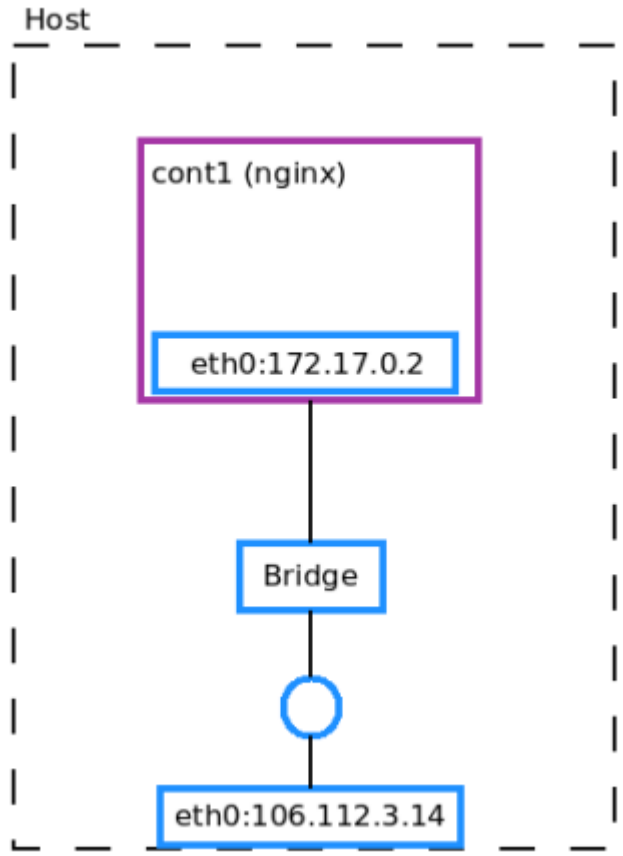
- Bridge :
  - Le pilote bridge interconnecte les conteneurs qui se trouve sur un réseau de ce type de pilote.
  - Vous pouvez exposer des ports sur ce type de réseau
  - Les conteneurs doivent tous s'exécuter sur l'hôte du réseau (mono-hôte).
  - Par défaut, le démon Docker connecte vos conteneurs dans le réseau bridge
- Overlay:
  - équivalent à bridge mais multi-host
  - Sert pour interconnecter les conteneurs s'exécutant sur différentes machines

```
$ docker network create -d bridge my-bridge-network
$ docker run -d --network=my-bridge-network --name db training/postgres
```

bridge : illustration

- $ docker run -d --name cont1 nginx

- Ce conteneur est lancé en utilisant le réseau bridge par défaut de Docker, car aucun réseau spécifique n'est mentionné.



bridge : illustration

- Créer un bridge
  $ docker network create -d bridge my_bridge

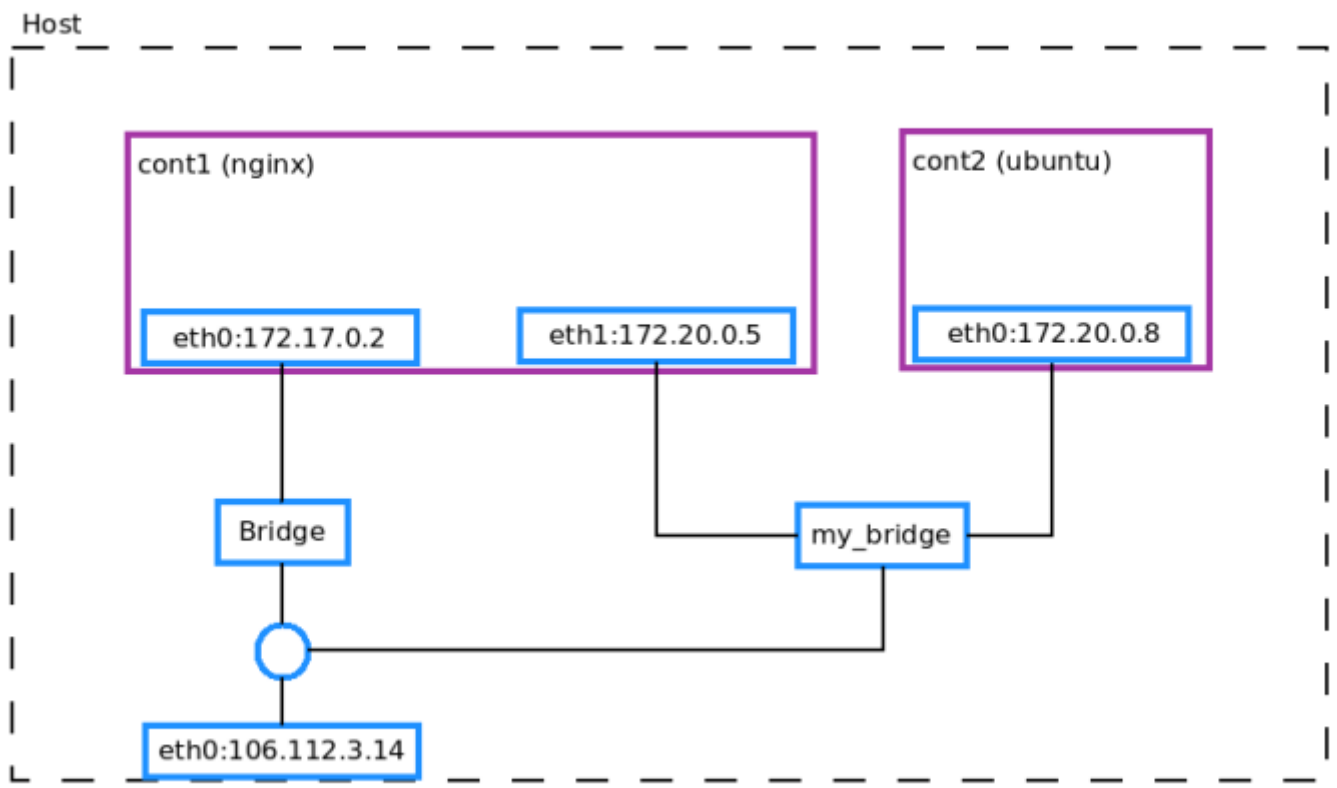
- Attacher un conteneur à un bridge
  $ docker run -d --network my_bridge --name cont2 ubuntu

- Connecter un contenur existent au bridge my_bridge :
  $ docker network connect my_bridge cont1

bridge : illustration

- cont1 peut communiquer avec cont2 au travers du bridge my_bridge Si un conteneur n'est associé qu'au bridge par défaut, il ne pourra pas communiquer avec cont2

- Le service DNS associé au bridge utilisateur permet à cont1 d'utiliser le nom du conteneur pour contacter cont2



CONTENEURS: LESBASES

Containeur basique

$ docker run debian /bin/echo "Salut"
Salut
$ docker run debian /bin/echo "Coucou"

Coucou

Lister les containers

$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

$ docker ps –a

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
d0683f6462a5	debian	"/bin/echo Salut“	About a minute ago	Exited (0)	About a minute ago
e1794g7573b6	debian	"/bin/echo Coucou“	About a minute ago	Exited (0)	About a minute ago

Ré- exécuter un container

$ docker start -i vibrant_swanson
Salut

Logs d'un container

$ docker start -i vibrant_swanson
Salut

Ménage : suppression d'un container

$ docker rm vibrant_swanson

container en cours d’exécution

$ docker run -it debian /bin/bash
root@2c666d3ae783:/#
ps -a
PID TTY TIME CMD
6 ? 00:00:00 ps

Interrompre le container

$ docker stop $id_conteneur
$ docker kill $id_conteneur
$ docker pause $id_ conteneur

Docker Exec

$ docker exec -it vibrant_swanson /bin/bash
root@2c666d3ae783:/#

root@2c666d3ae783:/# exit