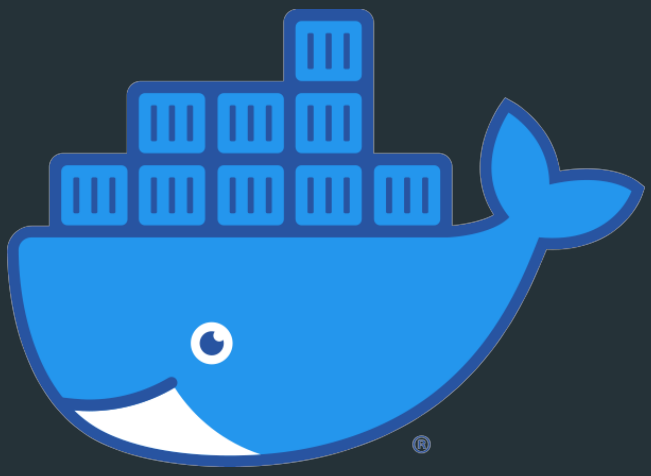
Docker Cookbook

Guide de mise en pratique
sur une architecture micro-service



NANONINJA

nanoninja

Docker Cookbook

Qu'est-ce que Docker ?

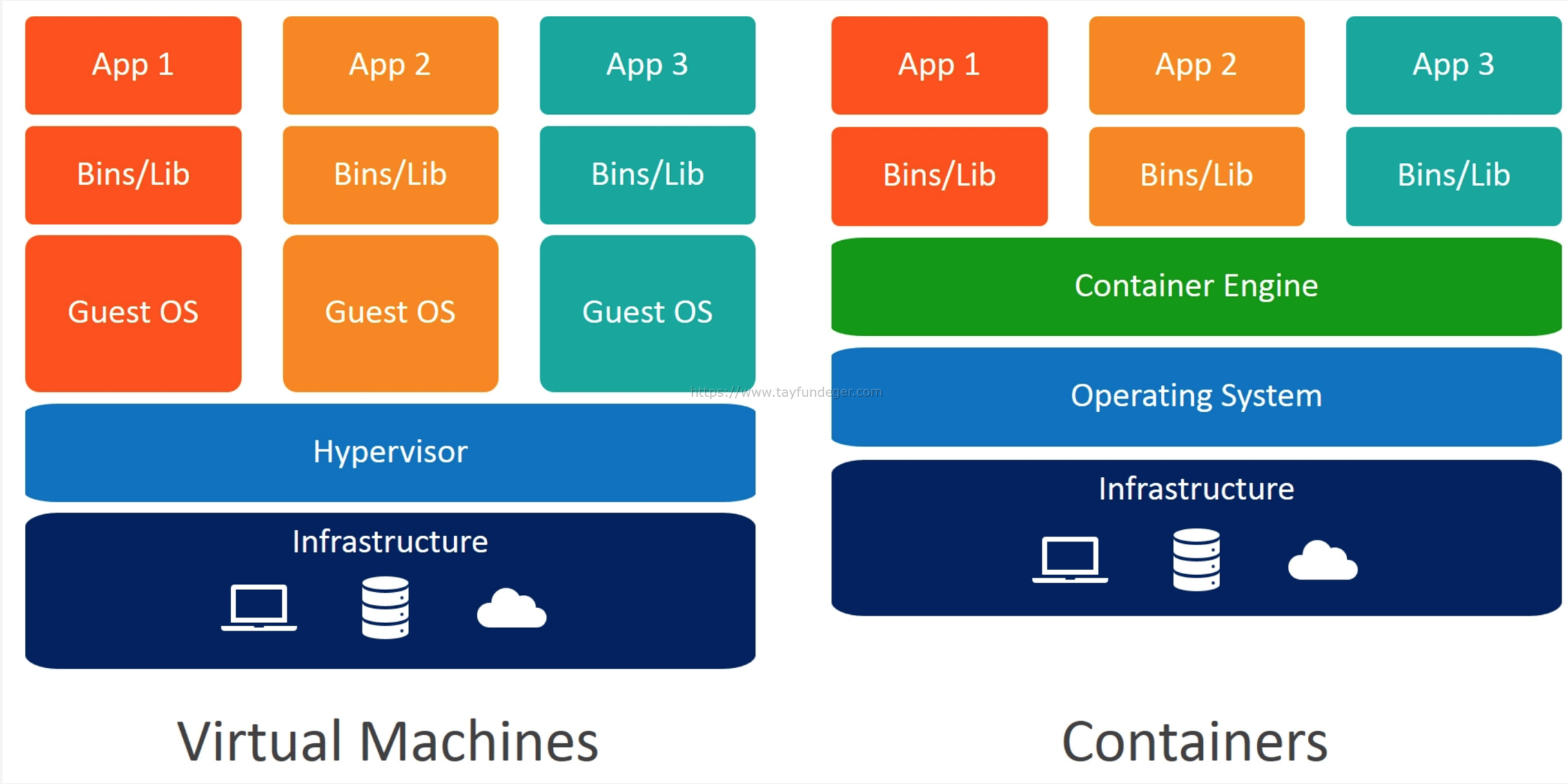
Docker est une plateforme de haut niveau permettant d'empaqueter une application et ses dépendances dans un conteneur virtuel isolé qui pourra être exécuté de la même manière sur n'importe quel serveur (Linux, Mac, Windows).

Docker fait de la « **conteneurisation** » d'application de manière standardisée car il fait abstraction de ce que y est embarqué par le conteneur.

Un conteneur est autonome et léger, il ne charge pas de système d'exploitation et embarque seulement ce qui est strictement nécessaire à l'exécution de l'application.

De multiple services sont proposés en ligne de commande ou en mode graphique pour faciliter la gestion des conteneurs, le déploiement et la scalabilité de l'infrastructure.

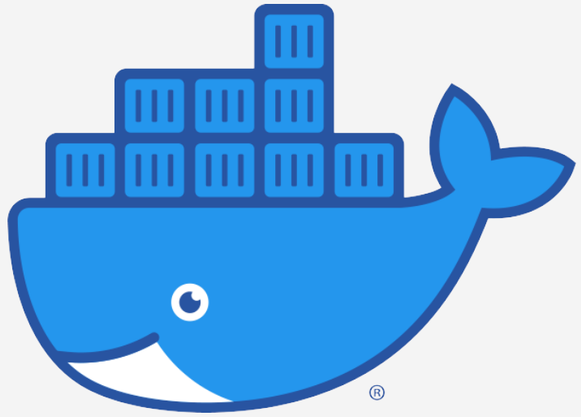
nanoninja
nanoninja
Docker Cookbook



NANONINJA

Docker Cookbook

Docker CLI



Docker CLI

Docker en ligne de commande

Les commandes docker sont organisées par groupe et sous groupe.

Le groupe principal en plus des sous groupes, propose des raccourcis pour accélérer la saisie des commandes les plus utilisées.

Groupes de commandes :

- `docker builder`
- `docker container`
- `docker compose`
- `docker context`
- `docker image`
- `docker network`
- `docker volume`
- `docker service`
- `docker swarm`
- `...`

Commandes principales - raccourcis

- `docker build (image)`
- `docker exec (container)`
- `docker images (image)`
- `docker logs (container)`
- `docker pull (image)`
- `docker ps (container)`
- `docker rm (container)`
- `docker rmi (image)`
- `docker run (container)`
- `docker start (container)`
- `docker stop (container)`
- `...`

Pour obtenir de l'aide sur les commandes :

```
docker COMMAND --help
```


Commande dans le contexte d'exécution

Les commandes docker peuvent avoir des sous commandes destinées au contexte d'exécution d'un conteneur.

Il est possible d'exécuter une commande docker pour démarrer un conteneur en lui fournissant une commande à exécuter dans le conteneur lui même (contexte d'exécution).

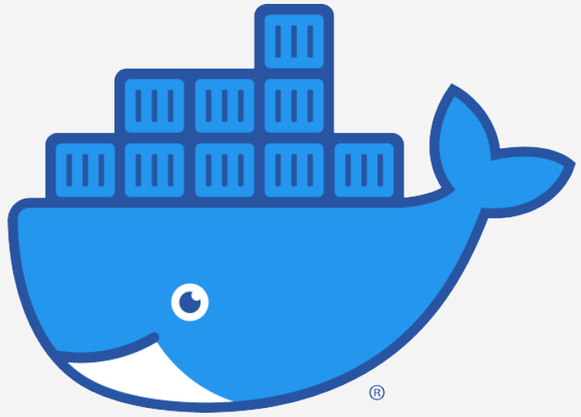
Par exemple on veut démarrer un conteneur à partir d'une image **node** et afficher la version **node** dans le terminal.

```
docker run --rm node:alpine node --version

V.17.5.0
```

La commande docker se termine après le nom de l'image **`node:alpine`** et est suivie de la commande à exécuter dans le conteneur qui fera appel à **node** avec l'option --**`version`**.

Docker Container



Docker container

Description

Exécuter une commande dans un nouveau conteneur

Usage

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG…]
```

Introduction

Un processus est un programme en cours d’exécution sur un ordinateur.

Lancer un conteneur signifie démarrer un processus dans un contexte d’isolation.

Le lancement d’un conteneur se fait à partir d’une **image** prédéfinie contenant le système de fichiers utile pour permettre au process du conteneur d’être exécuté.

L’image du conteneur est disponible sur le registre officiel **Docker Hub** ou en local.

Pour obtenir tous les commandes relatives aux conteneurs :

```
docker container
```

Ma première application

Lancement d'un serveur web via un conteneur

```
docker run --name web1 -p 8080:80 nginx
```

La commande **run** démarre le conteneur basé sur l'image **nginx** avec le nom **web1** avec un port d'écoute **8080** pour l'hôte qui redirigera vers le port **80** du serveur dans le conteneur.

Le serveur est maintenant disponible sur **http://localhost:8080**

Raccourci **CTRL+C** pour stopper le serveur.

Stoppez et supprimez le conteneur avec la commande **stop** et **rm** en utilisant le nom **web1** :

```
docker stop web1 && docker rm web1
```

L'option **-d** ou **--detach** permet de lancer un conteneur en tache de fond :

```
docker run --name web1 -p 8000:80 -d nginx
```


La commande docker **attach** fait revenir le conteneur en avant plan :

```
docker attach web1
```

L'ajout de l'option **--rm** au démarrage du conteneur signifie qu'il sera automatiquement supprimé lorsqu'il sera arrêté. C'est très pratique pour faire des testes et éviter de garder des conteneurs inutilisés :

```
docker run --rm --name web1 -p 8000:80 -d nginx
```

Lorsqu'un conteneur est lancé docker créé systématique un identifiant et lui donne un nom aléatoire si l'option **--name** n'est pas spécifiée.

Visualisation des conteneurs

Listez l'ensemble des conteneurs en cours d'exécution, en pause ou stoppés.

La commande **ps** sans arguments permet de lister uniquement les conteneurs en cours d'exécution

```
docker ps
```

La commande **ps** avec l'option **-a** liste en plus les conteneurs en pause ou stopper :

```
docker ps -a
```

Pour plus d'options utiliser l'aide :

```
docker ps --help
```


Supprimer des conteneurs

Il existe plusieurs méthodes pour supprimer un ou plusieurs conteneurs :

- Les supprimer un par un via le nom ou l'identifiant
- Supprimer seulement ceux qui sont stoppés
- Forcer la suppression même en cours d'exécution

Supprimer un conteneur via le nom ou l'ID

Pour supprimer un conteneur il faut qu'il est été stoppé au préalable :

```
docker stop web1

docker rm web1

# avec ID

docker stop 974e8bdd2a7f

docker rm 974e8bdd2a7f
```

Supprimer tous les conteneurs stoppés

La sous commande **`prune`** supprime tous les conteneurs stoppés :

```
docker container prune
```

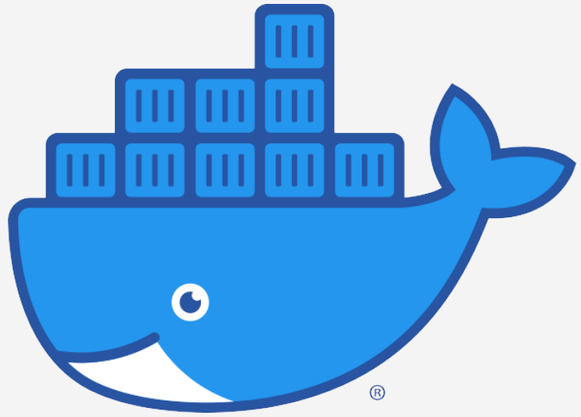
Forcer la suppression d'un conteneur

L'option **`-f`** de la commande **`rm`** force la suppression du conteneur qu'il soit en cours d'exécution, en pause ou à l'arrêt.

```
docker rm -f web1
```


NANONINJA

Docker Volume



Docker volume

Description

Gestion des volumes

Usage

```
docker volume [COMMAND]
```

Introduction

Les données manipulées par un conteneur sont éphémères. Lorsque un conteneur est détruit les données associées sont perdues.

Un volume permet de persister et partager les données d'un conteneur. Il existe de multiples façons de créer des volumes.

La plus fréquente est de déclarer un volume lors du démarrage du conteneur en spécifiant d'une part l'emplacement des fichiers que docker va utiliser depuis la machine hôte et d'autre part l'emplacement où ces fichiers seront manipulés dans le conteneur. C'est l'équivalent d'un raccourci, un lien symbolique.

Exemple avec notre serveur web **`nginx`** avec lequel il est nécessaire de créer un volume pour qu'il puisse obtenir les fichiers HTML de notre site.

Créer un volume lié à un conteneur

La création d'un volume se fait via le démarrage d'un conteneur avec l'option **`-v source:destination`** des fichiers à utiliser dans le volume :

```
docker run –name web1 \
  -p 8080:80 \
  -v /path/src/host:/path/dest/container:ro \
  -d nginx
```

L'option **`:ro`** ou **`readonly`** signifie que les fichiers du volume sont en lecture seule. Vous pouvez ajouter plusieurs volumes avec différents chemins comme les fichiers de configuration du serveur.

Exemple :

```
docker run –name web1 \
  -p 8080:80 \
  -v /path/src/html:/path/dest/html:ro \
  -v /path/src/config:/path/dest/config:ro \
  -d nginx
```

Selon l'environnement Linux, Max et Windows, il est préférable d'utiliser des chemins absolues au risque que votre volume ne soit pas pris en compte ou que cela déclenche une erreur.

Pour gagner du temps, vous pouvez utiliser une variable d'environnement **`${PWD}`** pour le chemin source et obtenir le chemin absolue du répertoire dans lequel cous exécutez la commande.

Créer et gérer des volumes

Contrairement à un montage lié, vous pouvez créer et gérer des volumes en dehors de la portée de n'importe quel conteneur.

```
docker volume create nom-du-volume
```

Dans l'exemple suivant nous allons créer un volume nommé **dbstore** pour persister une base de données MySQL via un conteneur en lui spécifiant le nom du volume à utiliser.

```
docker volume create dbstore
docker run --name db1 \
    -v dbstore:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=root \
    -e MYSQL_DATABASE=mydb \
    -e MYSQL_USER=dev \
    -e MYSQL_PASSWORD=dev \
    -d mysql
```

Supprimer des volumes

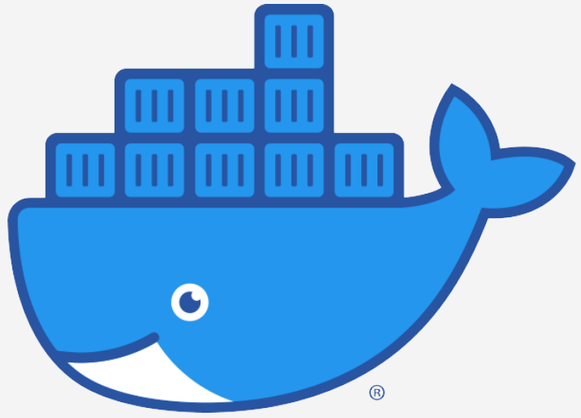
La suppression des volumes se fait avec la commande **rm** uniquement pour les volumes qui ne sont plus liés à des conteneurs.

```
docker volume rm nom-du-volume
```


Docker propose aussi de supprimer tous les volumes inutilisés, liés avec la commande **prune**.

```
docker volume prune
```

Docker Network



Docker network

Description

Gérer les réseaux

Usage

```
docker network COMMAND
```

Introduction

L'une des raisons pour lesquelles les conteneurs et les services Docker sont si puissants est que vous pouvez les connecter entre eux ou les connecter à des charges de travail non Docker.

Par défaut il existe 3 principaux pilotes de réseaux :

- null
- host
- bridge

Pilote réseau bridge

Le pilote réseau par défaut. Si vous ne spécifiez pas de pilote, il s'agit du type de réseau que vous créez. Les réseaux de pont sont généralement utilisés lorsque vos applications s'exécutent dans des conteneurs autonomes qui doivent communiquer.

Le pilote utilise un système de nom de domaine (DNS) basé sur le nom de conteneur pour communiquer entre eux. C'est très pratique car cela évite d'avoir à connaître l'adresse ip.

```
docker run --rm \
  --network bridge \
  --name web1 \
  -d nginx
```

Pilote réseau Host

Utile pour les conteneurs autonomes, supprime l'isolation réseau entre le conteneur et l'hôte Docker, et utilise directement la mise en réseau de l'hôte.

```
docker run --rm \
  --network host \
  --name web1 \
  -d nginx
```


Pilote réseau none

Désactive tous les réseaux. Généralement utilisé en conjonction avec un pilote réseau personnalisé. **none** n'est pas disponible pour les services Swarm.

Créer un réseau à partir d'un pilote

Docker propose créer un réseau personnalisé basé des pilotes que l'on peut paramétrer pour plus ou moins d'isolation. Cela offre la possibilité par exemple de définir une plage d'adresses ip.

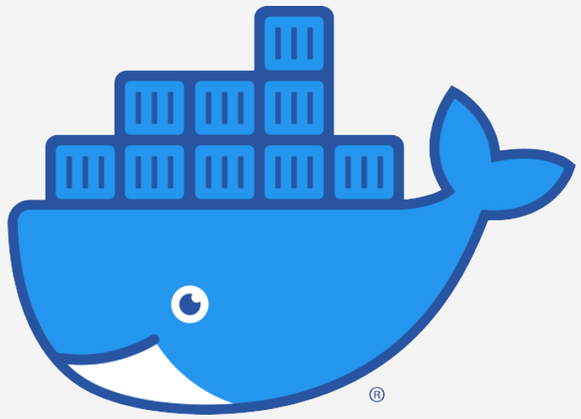
Un autre avantage c'est de diviser les réseaux par exemple pour les applications frontend et backend.

```
docker network create --driver bridge nom-du-reseau
```

NANONINJA

Docker Cookbook

Projet Nexcloud



Docker Nextcloud

Description

L'objectif de ce projet est de créer un cloud personnel à partir d'une image MySQL et Nextcloud.

Nous allons créer un réseau pour communiquer entre les conteneurs, un volume pour la base de données et le cloud, un conteneur pour exécuter MySQL et un autre pour Nextcloud.

Création du réseau et volume

```
docker network create --driver bridge cloud_network
docker volume create cloud_store
docker volume create cloud_data
```

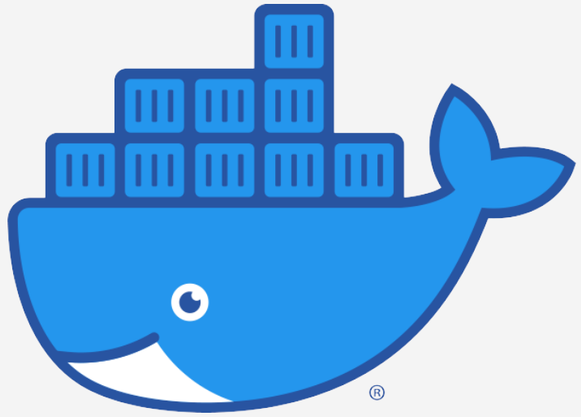
Création du conteneur Mysql

```
docker run --name cloud_db \
    -v cloud_store:/var/lib/mysql \
    --network cloud_network \
    -e MYSQL_ROOT_PASSWORD=root \
    -e MYSQL_DATABASE=mydb \
    -e MYSQL_USER=dev \
    -e MYSQL_PASSWORD=dev \
    -d mysql
```

Création du conteneur Nextcloud

```
docker run --name my_own_cloud \
    --network cloud_network \
    -v cloud_data:/var/www/html \
    -p 8080:80 \
    -d nextcloud
```


Docker Build



Docker build

Description

Construire une image à partir d'un Dockerfile

Usage

```
docker build [OPTIONS] PATH | URL | -
```

Introduction

La commande docker **`build`** crée des images Docker à partir d'un Dockerfile.

C’est un peu comme une recette de cuisine dans laquelle on défini tous les ingrédients nécessaires d’une application.

Souvent on part d’une image déjà existante comme node, nginx et on y ajoute nos fichiers d’application avec les différentes dépendances. On construit l’image et on peut la conteneuriser.

Bien qu’il soit possible de partir de rien (**`FROM scratch`**) cela peut s’avérer plus ou moins difficile dans certain cas.

Créer une image Dockerfile

À la racine de votre répertoire de travail créez un fichier nommé **`Dockerfile`** et **`app.js`**.

Dans cet example nous allons créer un serveur web avec Express qui écoutera sur le port **`3000`**.

```
# app.js
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.json({ message: 'ok' });
});

app.listen(3000);
```

Par défaut docker utilisera automatiquement votre fichier **`Dockerfile`** mais il es possible d'en avoir plusieurs avec des noms différents mais dans ce cas il faudra spécifier quel fichier Docker devra utiliser avec l'option **`-f`**.

```
docker build -f Dockerfile-2 -t nom-mon-image .
```

L'option **`-t`** signifie **`tag`** pour taguer le nom de votre image par exemple avec un numéro de version comme **`nom-mon-image:1.0.0`**.

Pour le Dockerfile nous partirons de l'image de base **`node:alpine`** car elle est légère.

```
# Nom de l'image de base
FROM node:alpine

# Répertoire pour notre application.
WORKDIR /home/node/app

# On installe express avec npm.
RUN npm install express

# On copie le fichier app.js en local
# pour le placer dans le répertoire courant
# qui correspond au WORKDIR ci-dessus.
COPY app.js .

# On expose le port 3000 pour le serveur.
EXPOSE 3000

# Commande qui sera exécutée au démarrage du
# container qui lancera notre serveur.
CMD [ "node", "app.js" ]
```

On lance la construction de l'image avec la commande build :

```
docker build -t node-app .
```

Une fois l'image créée nous démarrons notre conteneur

```
docker run --rm --name app1 \
    -p 8080:3000 \
    -d node-app
```

On ajoute l'option `--rm` qui supprimera automatiquement le conteneur lorsqu'il sera stoppé.

```
docker stop app1
```


Construire une image en plusieurs étapes

Il arrive parfois que nous ayons besoin d'utiliser des images temporaires, intermédiaires pour aboutir à la construction finale de notre image.

Par exemple une image avec des dépendances de développement indispensables à la compilation de notre application mais qui une fois compilée ne sont plus utiles pour la production.

Cela arrive par exemple avec des applications comme Angular, React, Go, C, Rust, ...

Pour réaliser ce type de construction il faut séparer les différentes étapes dans le fichier `Dockerfile` avec l'instruction `FROM`.

Chaque étape doit être nommée pour que la suivante puisse accéder au résultat de la construction précédente.

```
FROM node as etape1

FROM nginx
ici on pourrait récupérer le résultat
de la construction node via le nom etape1
```

Réalisons une construction à deux étapes en partant d'une application de type API réalisée avec le langage Go.

Créez un répertoire dans lequel vous ajoutez un fichier **`main.go`** et **`Dockerfile`**. Vous trouverez ci-après le code source à copier coller dans les fichiers respectifs.

```
// main.go
package main

import (
    "encoding/json"
    "log"
    "net/http"
)

func ping(w http.ResponseWriter, r *http.Request) {
    w.Header().
        Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(map[string]string{
        "message": "ok",
    })
}

func main() {
    log.Println("Server is running")
    http.HandleFunc("/", ping)
    http.ListenAndServe(":8000", nil)
}
```



```
# Dockerfile

# Étape 1 que l'on nomme builder
FROM golang:alpine as builder

WORKDIR /usr/src/app
COPY main.go .

# Compilation de l'exécutable de notre api
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 \
    go build -o api main.go

# Étape 2
FROM scratch

# On copie le résultat de l'étape 1 nommé builder
COPY --from=builder /usr/src/app/api /app/api
EXPOSE 8000

# Le point d'entré de notre conteneur est
# l'exécutable que nous avons compilé avec Go.
ENTRYPOINT [ "/app/api" ]
```

Dans le **`Dockerfile`** nous partons d'un image dite **`from scratch`** parceque l'exécutable go est complètement autonome et ne nécessite aucune dépendances ce qui en fera une image très légère.

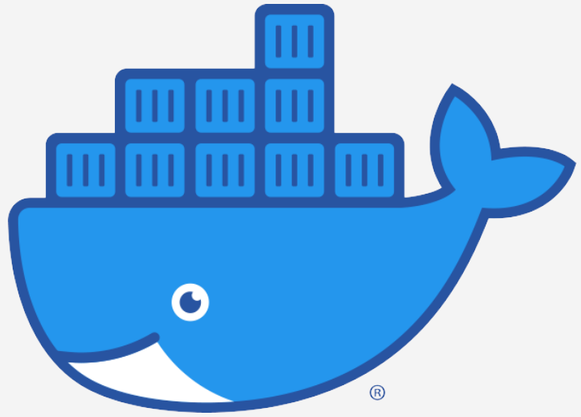
Création de l'image

```
docker build -t go-app .
```

Démarrage du container

```
docker run --rm --name goapp \
    -p 8080:8000 \
    -d go-app
```


Docker Compose



NANONINJA Docker Cookbook

Docker compose

Description

Docker Compose

Usage

```
docker compose [OPTIONS] COMMAND
```

Introduction

Compose est un outil permettant de définir et d'exécuter des applications Docker multi-conteneurs. Avec Compose, vous utilisez un fichier YAML pour configurer les services de votre application. Ensuite, avec une seule commande, vous créez et démarrez tous les services de votre configuration.

Nextcloud avec Docker compose

Reprenons l'application Nextcloud réalisée précédemment en récupérant les commandes que nous formaterons dans un fichier **`docker-compose.yml`** avec lequel nous pourrons démarrer l'ensemble des conteneurs avec une seule commande.

```
version: "3.7"
services:
  db:
    image: mysql:8.0.28
    container_name: cloud_db
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=root
          - MYSQL_DATABASE=mydb
    networks:
      - cloud_network
    volumes:
      - cloud_store:/var/lib/mysql
  web:
    image: nextcloud:21.0.9
    container_name: my-own-cloud
    depends_on:
      - db
    restart: always
    ports:
      - 8080:80
    networks:
      - cloud_network
    volumes:
      - cloud_data:/var/www/html
networks:
  cloud_network:
    driver: bridge
volumes:
  cloud_store:
  cloud_data:
```