



# **FORMATION DOCKER**

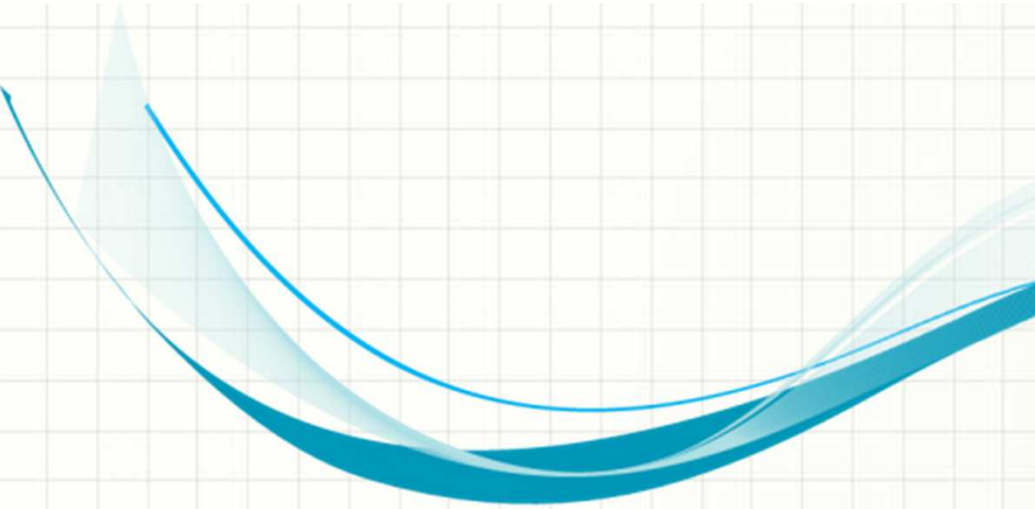
## **PARTIE 3**

Laurent YAO

Octobre 2017

# Plan du cours

- Introduction à Docker
- Premiers pas avec Docker
- Construire des images de containers personnalisées
- Mettre en œuvre une application multi container
- Intégration continue avec gitlab ci et registry
- Orchestration monitoring et cluster
- Administration des containers au quotidien



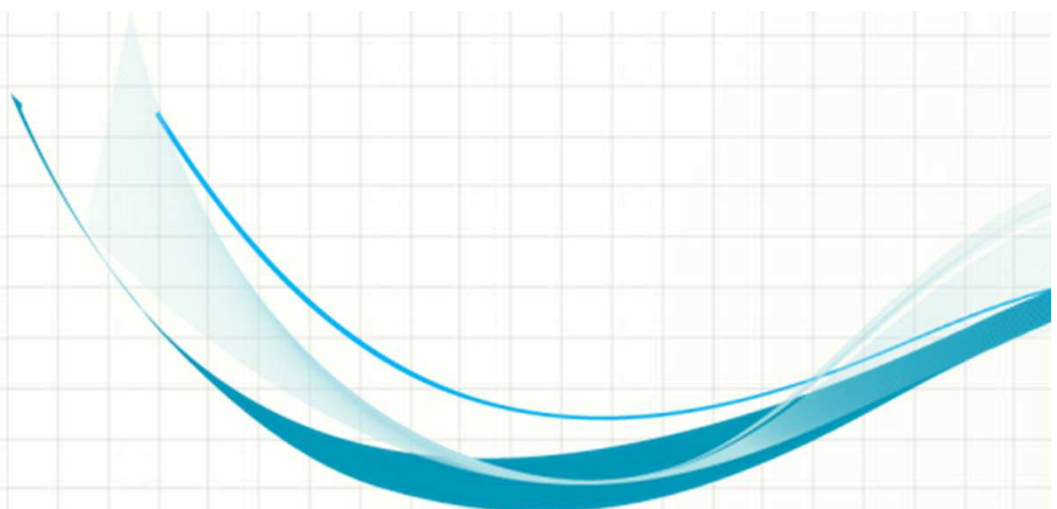
# Objectifs

# Formation Docker : objectifs

- A l'issue de ce cours, les participants disposeront des connaissances et des compétences pour maîtriser la pratique de docker et son écosystème
- Vous serez capables de:
  - décrire les concepts de base de la technologie
  - créer des conteneurs en ligne de commande
  - gérer des images personnalisées localement et à distance

# Formation Docker : objectifs

- Vous serez capables de:
  - d'administrer des conteneurs en production
  - créer et déployer des applications multi-conteneurs

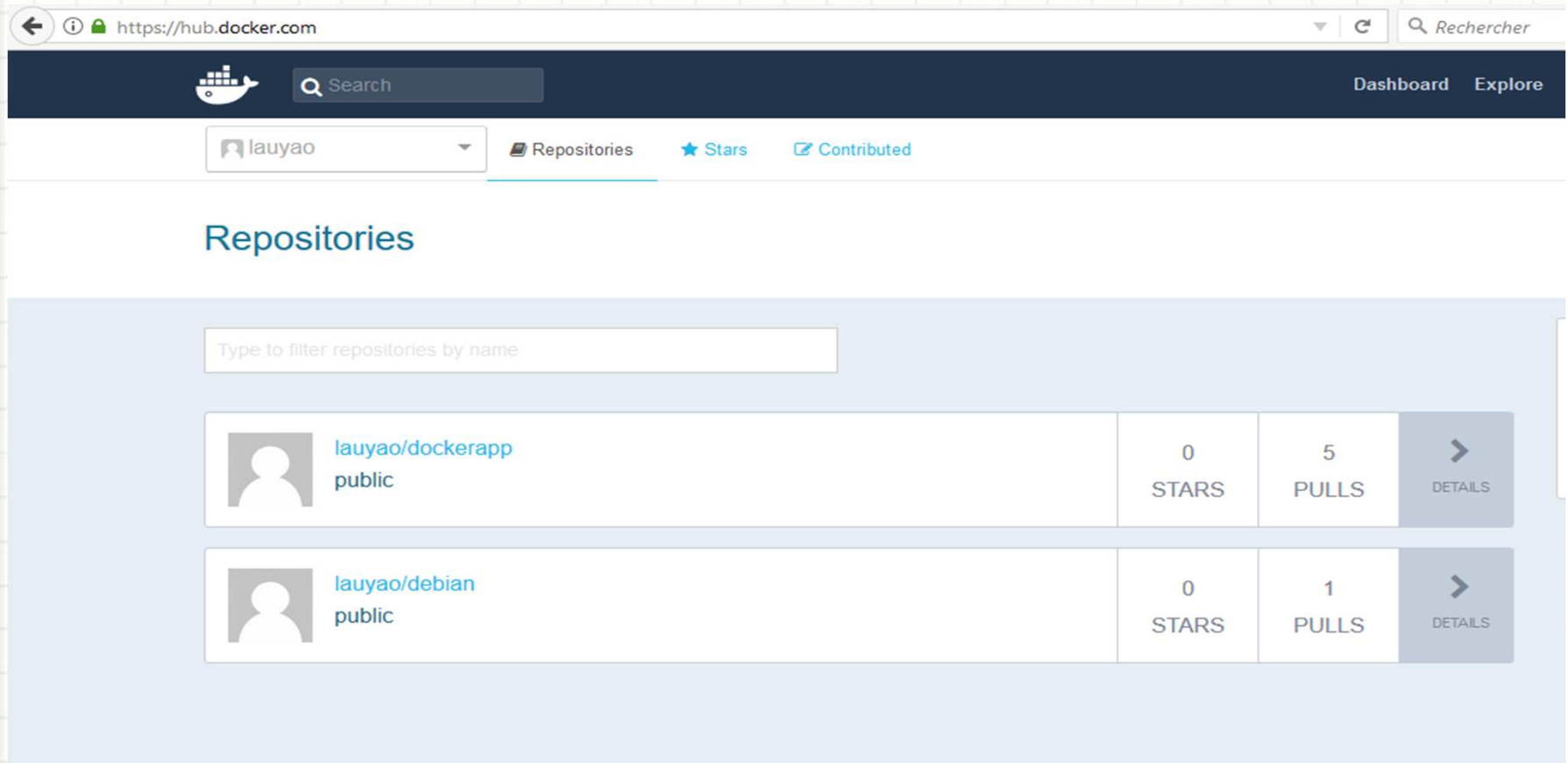


Pousser une image  
docker dans un  
registry (docker  
hub)



# Image docker dans docker hub

- Etape 0 : créer un compte docker hub
  - Se connecter au compte



The screenshot shows the Docker Hub web interface. The browser address bar displays 'https://hub.docker.com'. The navigation bar includes a search bar, 'Dashboard', and 'Explore' links. Below the navigation bar, the user 'lauyao' is selected, and the 'Repositories' tab is active. The main content area is titled 'Repositories' and features a search input field. A table lists the user's repositories:

Repository	Stars	Pulls	Details
lauyao/dockerapp public	0	5	> DETAILS
lauyao/debian public	0	1	> DETAILS

# Image docker dans docker hub

- Etape 1 : lier l'image au compte docker hub
  - Renommer l'image en  
« docker\_hub\_id/repository\_name » à l'aide de  
la commande docker tag
  - *docker tag SOURCE\_IMAGE[:TAG]  
TARGET\_IMAGE[:TAG]*

```
MINGW64:/c/Users/Laurent/myhttp  
Laurent@pc-laurent MINGW64 ~/myhttp  
$ docker tag imapache:1.0 lauyao/myhttpd:1.0  
Laurent@pc-laurent MINGW64 ~/myhttp  
$
```



# Image docker dans docker hub

- Etape 2 : login au registry (docker hub)
  - *docker login [OPTIONS] [SERVER]*

```
$ docker login -u lauyao
Password:
Login Succeeded

Laurent@pc-laurent MINGW64 ~/myhttp
$
```

# Image docker dans docker hub

- Etape 3 : push de l'image dans le registry
  - *docker push [OPTIONS] NAME:[TAG]*

```
$ docker push lauyao/myhttpd:1.0
The push refers to a repository [docker.io/lauyao/myhttpd]
2f14923e8441: Pushed
3936e8a13480: Pushed
f3e8d66aa97c: Mounted from library/httpd
040d4bc93d09: Mounted from library/httpd
0fe6acf04f59: Mounted from library/httpd
8de9634b27d5: Mounted from library/httpd
3729f65291bb: Mounted from library/httpd
3de6717c4697: Mounted from library/httpd
c01c63c6823d: Mounted from library/httpd
1.0: digest: sha256:80b5fba22be28f868a49f4ce4919fe12c0d45bdfa6af1a4f84e6ca35e5647e42 size: 2200

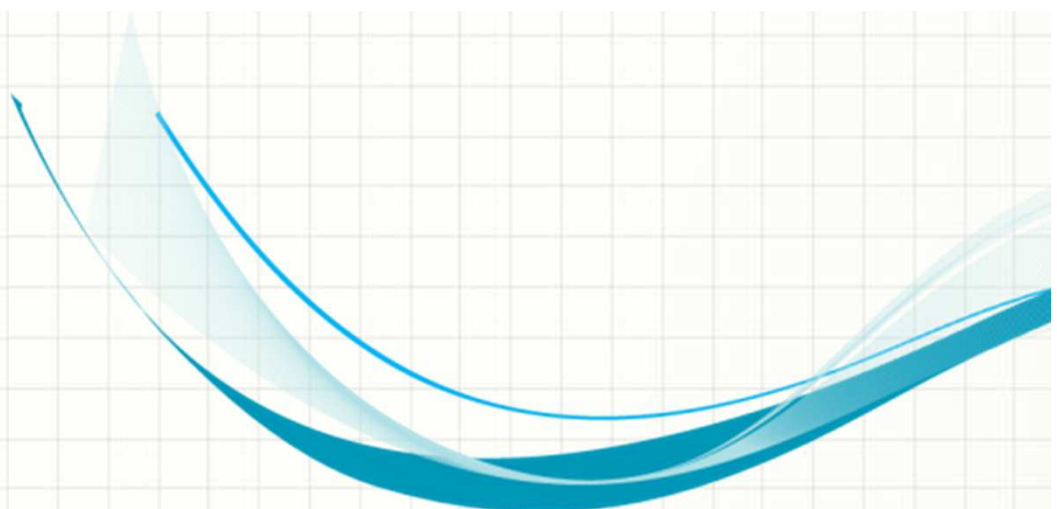
Laurent@pc-laurent MINGW64 ~/myhttp
$
```

# Image docker dans docker hub

## Repositories

Type to filter repositories by name

	<a href="#">lauyao/dockerapp</a> public	0 STARS	5 PULLS	<a href="#">&gt;</a> DETAILS
	<a href="#">lauyao/debian</a> public	0 STARS	1 PULLS	<a href="#">&gt;</a> DETAILS
	<a href="#">lauyao/myhttpd</a> public	0 STARS	1 PULLS	<a href="#">&gt;</a> DETAILS



# Mise en œuvre d'une application multi conteneur

# MEO application multi conteneur

- Points à aborder:
  - Application un seul conteneur plusieurs processus
  - Application multi conteneur
  - Gérer l'interconnexion de plusieurs conteneurs
  - Création d'un fichier docker-compose.yml
  - Déployer plusieurs conteneurs simultanément
  - Lier tous les conteneurs de l'application

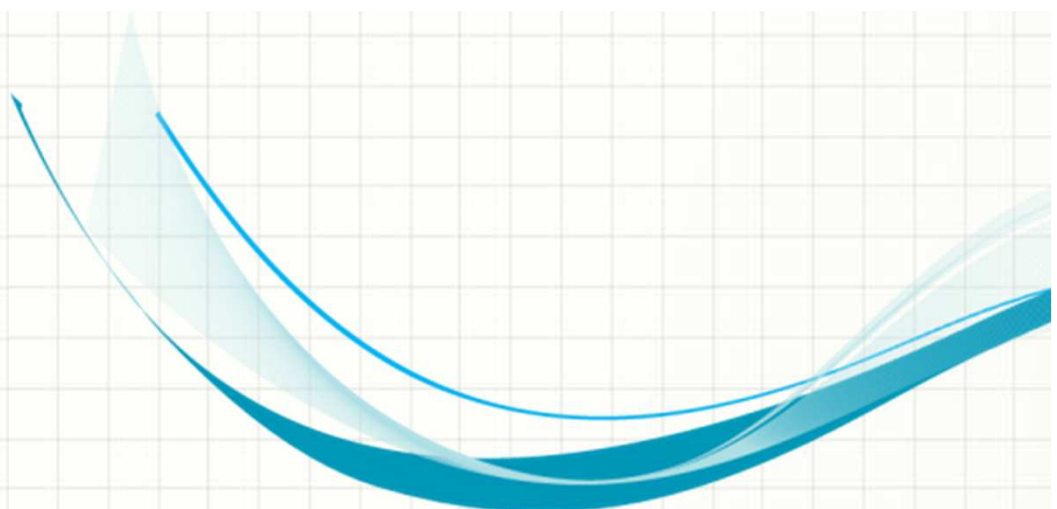
# Application un seul container

- Présentation de l'application
- Création du Dockerfile
- Construction de l'image
- Exécution du container
- Utilisation de l'application



# Application multi conteneur

- Présentation de l'application
- Création du Dockerfile
- Construction de l'image
- Exécution du container
- Utilisation de l'application



# Orchestration avec docker- compose

# Docker-compose

- Docker compose permet:
  - De définir dans un fichier de configuration toutes les dépendances d'une application multi-container
  - De démarrer tous les containers de l'application d'une seule commande

# Docker-compose

- Les principales commandes docker compose:

Commande	Résultat
docker-compose up	Crée et démarre les containers
docker-compose logs	Affiche une vue abrégée de toutes les logs de l'application
docker-compose start/stop	Démarre/arrête l'ensemble des containers de l'application
docker-compose pause/unpause	Met en pause/relance les processus des containers de l'application
docker-compose rm	Supprime tous les containers de l'application
Docker-compose down	Arrete et supprime tous les containers de l'application

# Fichier docker-compose.yml

- Le fichier compose permet de définir des services, des réseaux et des volumes
- Le chemin par défaut est ./docker-compose.yml

# Principales options

- build: pour générer une image à partir d'un Dockerfile
  - context: chemin vers le Dockerfile ou url vers un repository git
  - dockerfile: spécifie un nom de Dockerfile alternatif
  - args: argument du build, var envt accessibles seulement pendant le build
  - cache\_from : spécifie une liste d'images utilisées pour la résolution du cache
  - command



# Principales options

- Exemples pour build
  - Défini comme string avec un path vers le build context

```
version: '2'
services:
  webapp:
    build: ./dir
```

- Défini comme un objet avec context, dockerfile, args, cache\_from, etc.

```
version: '2'
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
    args:
      buildno: 1
```

# Principales options

- `image`: spécifie l'image à partir de laquelle démarrer le conteneur
- `container_name`: nom du conteneur
- `ports` : pour spécifier les ports au niveau host et container
- `depends_on`: déclarer la dépendance entre services
- `volumes`: pour monter des volumes entre le host et le conteneur

# Principales options

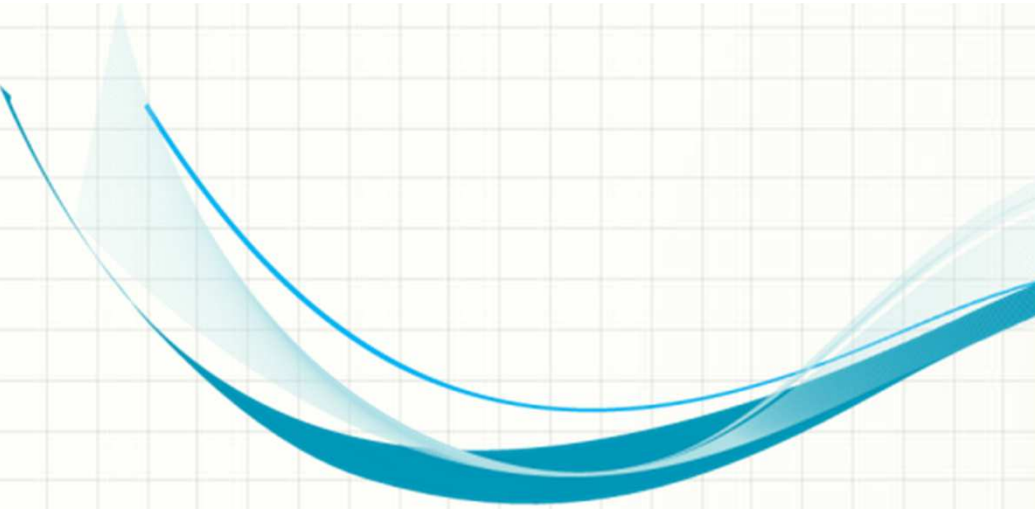
- Quelques exemples
  - *image: redis*
  - *image: ubuntu:14.04*

# Principales commandes et options

- docker-compose [options]
  - up: build, crée, démarre les conteneurs pour un service
    - -d : démarre en arrière plan
  - ps : liste les containers gérés par docker-compose
  - logs: affiche les logs des containers
    - -f: montre les ajouts dans les logs
    - *nom\_container*: affiche la log du container
  - stop: arrête les conteneurs en cours d'exécution
  - rm : supprime tous les conteneurs

# Principales commandes et options

- docker-compose [options]
  - build: recrée toutes les images
  - down: arrête et supprime les conteneurs, images, volumes, réseaux
  - run : exécute une command dans un service dans un container
  - etc.

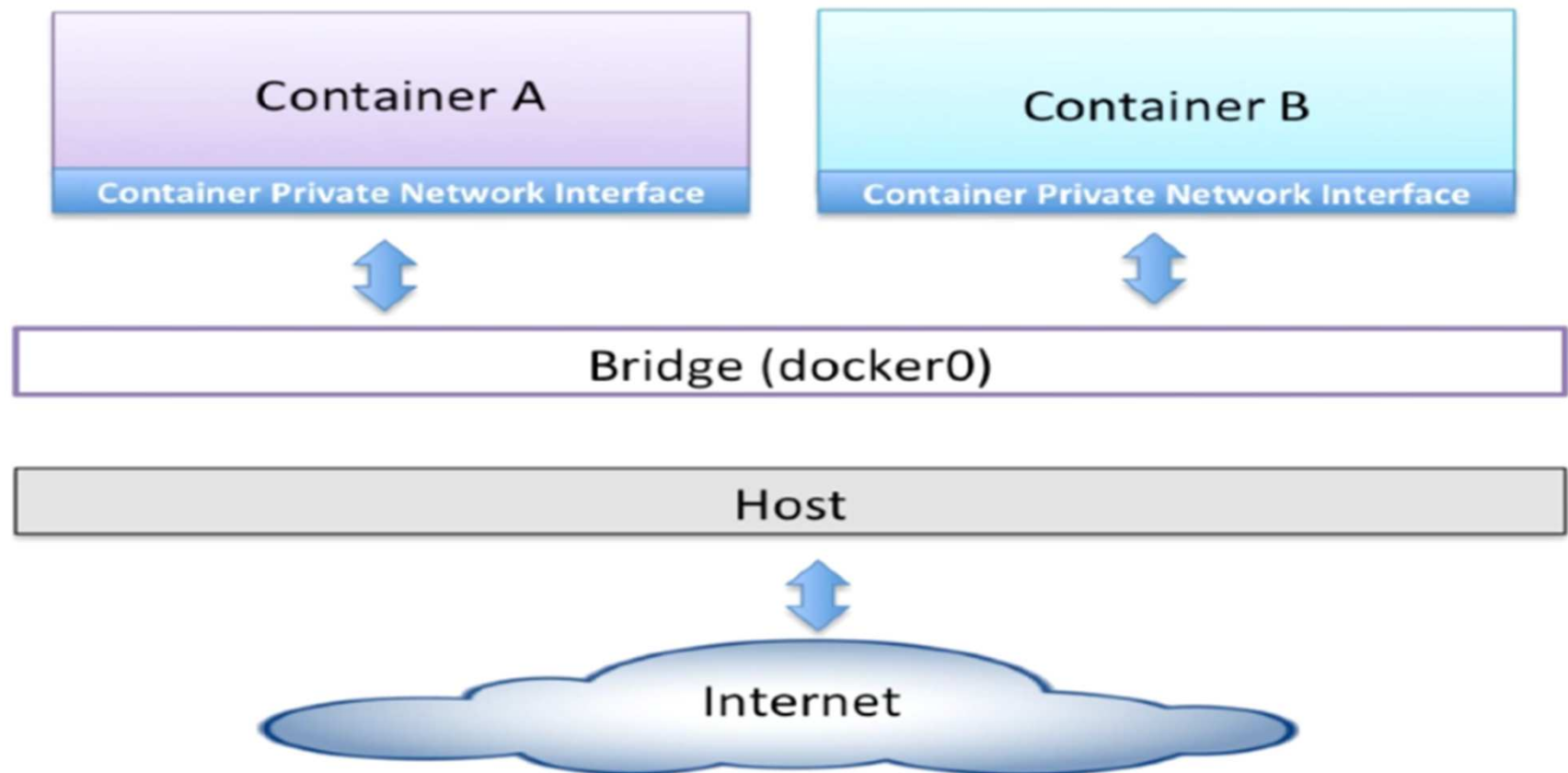


# Docker networking



# Docker networking

- Le modèle de réseau docker par défaut est:  
**Default Docker Network Model**



# Docker networking

- Docker utilise les capacités réseau de la machine host
- Quand le daemon docker est installé sur la machine host une interface de type bridge (docker 0) est installée
- Cette interface va gérer le trafic entre l'extérieur et les différents container
- Chaque container se connecte au bridge network à travers son interface réseau

# Docker networking

- Les container peuvent se connecter entre eux et se connecter au monde extérieur à travers cette interface réseau de type bridge

# Docker networking : les commandes

- Les principales commandes de gestion des réseaux docker sont les suivantes:

Commande	Objet
<code>docker network create</code>	Permet de créer un réseau Docker. Cette commande prend notamment en paramètre <code>--driver</code> qui permet de spécifier le type de réseau souhaité (par défaut bridge).
<code>docker network connect</code>	Cette commande permet de connecter un conteneur à un réseau.
<code>docker network disconnect</code>	Cette commande permet de déconnecter un conteneur d'un réseau.
<code>docker network inspect</code>	Une fonction d'inspection du réseau dont nous verrons l'utilité dans le chapitre 9.
<code>docker network ls</code>	Une commande qui liste les réseaux disponibles. Par défaut, trois réseaux sont systématiquement définis : none, host et bridge.
<code>docker network rm</code>	La commande qui permet de détruire des réseaux existants (sauf évidemment nos trois réseaux prédéfinis).

# Docker networking

- Il y a 4 type de modèles réseaux dans les containers docker:
  - Closed network/none network
  - Bridge network
  - Host network
  - Overlay network

# Closed/None network model

- Ce type de réseau n'a pas accès au monde extérieur.
- Le container associé est totalement isolé
- Ex de création d'un container avec réseau de type non network

```
$ docker run -d --net none busybox:1.27
```

```
202dba023fd922690c5db2fd9803329cf3ac067fdf05586bcb946fe6a0251840
```

```
Laurent@pc-laurent MINGW64 ~/application-multi-conteneurs/supervisor (master)
```

```
$
```



# Closed/None network model

```
Laurent@pc-laurent MINGW64 ~/application-multi-conteneurs/supervisor (master)
$ docker run -d --name myctrnet --net none busybox:1.27 sleep 60
b452d579fa2055cbf5e0bc800e2ed1290bae4132877a4d61d3d467780fb4a44e

Laurent@pc-laurent MINGW64 ~/application-multi-conteneurs/supervisor (master)
$ docker exec -it myctrnet /bin/ash
/ # ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
ping: sendto: Network is unreachable
/ #
```

- Points forts:
  - Niveau de protection maximal

# Bridge network model

- C'est le modèle de réseau par défaut
- Tous les containers, au sein du même « bridge network » sont connectés entre eux et aussi connectés au monde extérieur via l'interface du bridge network
- La commande « docker network connect ... » permet de connecter un container à un autre même s'ils sont de deux « bridge network » différents
- Voir les exemples

# Bridge network model

- Par rapport au « none network », « bridge network »:
  - Réduit le niveau d'isolation du container pour une meilleure connectivité extérieure

# Host network model

- C'est le moins protégé des network model, il ajoute un container sur la pile réseau du host
- Les container déployé sur la pile réseau du host ont un accès full à l'interface réseau du host
- On les appelle « Open container »
- Niveau minimum de sécurité
- Meilleure performance

# Overlay network model

- Supporte un modèle de réseau au travers de plusieurs hosts
- Pré conditions:
  - Docker engine est exécuté en mode swarn
  - Un store key-value comme consul
- Est utilisé en production

# CNM dans docker-compose

- Docker-compose crée et utilise un network par défaut auquel tous les containers créés sont rattachés
- On peut aussi créer d'autres network qui seront partagés entre containers



# CNM dans docker-compose

```
version: '2'

services:
  proxy:
    build: ./proxy
    networks:
      - front
  app:
    build: ./app
    networks:
      - front
      - back
  db:
    image: postgres
    networks:
      - back

networks:
  front:
    # Use a custom driver
    driver: custom-driver-1
  back:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

A decorative blue wavy line with a gradient, starting from the top left and curving towards the right, positioned above the title.

# Docker volumes

# Docker volumes

- Docker volume c'est le mécanisme préféré pour persister les données
- Les volumes sont totalement gérés par Docker
- Les volumes donne les avantages suivants:
  - Fonctionne sur Linux et Windows
  - Utilisable par docker cli et docker api
  - Partagés entre plusieurs containers

# Docker volumes

- Les principales commandes:

Commande	Objet
<code>docker volume create</code>	Une commande qui permet de créer un volume qu'il sera ensuite possible d'associer à un ou plusieurs conteneurs. Cette commande prend en paramètre <code>--driver</code> qui permet de spécifier le driver utilisé pour ce volume. Il existe aujourd'hui des plugins Docker permettant de s'appuyer sur des systèmes de stockage tiers (en lieu et place d'une simple persistance sur l'hôte).
<code>docker volume inspect</code>	Une commande pour visualiser des métadonnées relatives à un volume.
<code>docker volume ls</code>	La commande qui permet de lister les volumes disponibles.
<code>docker volume rm</code>	La commande qui permet d'effacer des volumes. Attention, une fois détruit, les données associées à un volume sont perdues définitivement. Il n'est cependant pas possible d'effacer un volume utilisé par un conteneur.