

# DOCKER AVANCE

# MODULE 1: GESTION DU DEMON

# Contrôle et configuration du Daemon

- La configuration du démarrage et arrêt du démon Docker dépend d'un certain nombre de facteurs
  - Le démon s'exécute comme un service ?
  - Quelle distribution Linux
    - service
    - systemctl
- Exécution en mode interactif en arrière plan

# Exécution en mode service

## Ubuntu et Debian

- Utiliser la commande **service**
  - `sudo service docker stop`
  - `sudo service docker start`
  - `sudo service docker restart`

# Exécution en mode service

## RHEL & CentOS

- Utiliser la commande `systemctl`
  - ▣ `systemctl start docker`
  - ▣ `systemctl stop docker`
  - ▣ `systemctl restart docker`
- Dans les nouvelles version la commande `service` est une redirection vers la commande `systemctl`

# Exécuter le démon de manière interactive

- S'il ne fonctionne pas en tant que service, lancez le démon Docker
  - sudo docker daemon & (avant la 1.12)
  - sudo dockerd & (depuis la 1.12)
- S'il ne fonctionne pas en tant que service, envoyez le signal SIGTERM pour arrêter le démon
  - sudo kill \$(pidof docker)

# Fichiers de configuration du démon

## **Ubuntu et Debian**

- /etc/default/docker
- La variable DOCKER\_OPTS est utilisée pour ajouter des options au démon s'il est lancé comme un service
- Relancez le service pour prendre en compte les modifications  
`sudo service docker restart`

# Fichiers de configuration du démon

## RHEL et Centos

- **Le mécanisme** `systemd` **est utilisé pour exécuter le démon**
- **Les options de lancement sont dans le fichier** `docker.service`
- **Le fichier** `docker.service` **se trouve dans**  
`/usr/lib/systemd/system` **ou** `/etc/systemd/service`
- **Exécuter la commande :** `find / -name docker.service`

```
[root@docker ~]# find / -name docker.service -print
/etc/systemd/system/multi-user.target.wants/docker.service
/usr/lib/systemd/system/docker.service
```

# Le fichier docker.service

```
[root@docker ~]# cat /etc/systemd/system/multi-user.target.wants/docker.service
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues
still
# exists and systemd currently does not support the cgroup feature set
required
# for containers run by docker
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
EnvironmentFile=-/etc/sysconfig/docker-network
ExecStart=/usr/bin/dockerd $OPTIONS \
          $OPTIONS_STORAGE \
          $OPTIONS_NETWORK
ExecReload=/bin/kill -s HUP $MAINPID
```

# Le fichier de configuration du démon

- Le fichier `docker.service` utilise les fichiers `EnvironmentFile` qui font référence aux fichiers `/etc/sysconfig/docker` ...
- Ce sont ces derniers fichiers qui seront utilisés pour modifier les options du démon

```
[root@docker ~]# cat /etc/sysconfig/docker
OPTIONS=" \
-H tcp://0.0.0.0:2375 \
-H unix:///var/run/docker.sock \
"
```

# Les options du démon

- ❑ Lancer le démon en mode réseau.
  - Le démon écoute sur une socket TCP
- ❑ Activer le mode debug
  - Valider le niveau de journalisation
- ❑ Spécifier un serveur DNS
- ❑ Ajouter des registres non sécurisés
- ❑ Valiser la sécurité du démon avec TLS

# Journalisation du démon

- Démarrez le démon avec l'option `--log-level` et spécifiez le niveau de journalisation
  - Debug
  - Info
  - Warn
  - Error
  - Fatal

# Journalisation sur RHEL et CentOS

- Sur les systèmes basés sur systemd la journalisation est gérée par le démon journald
- La commande journalctl permet de visualiser le log
- Utiliser l'option **-u** pour filtrer par service  
journalctl -u docker.service

```
[root@docker ~]# journalctl -u docker.service
```

```
-- Logs begin at lun. 2017-10-02 07:57:37 CEST, end at mar. 2017-10-03 09:26:49 CEST. --
oct. 02 11:55:04 poste508.s11.pfd systemd[1]: Starting Docker Application Container Engine...
oct. 02 11:55:04 poste508.s11.pfd dockerd[32645]: time="2017-10-02T11:55:04.797590516+02:00" level=info msg="libcon
oct. 02 11:55:05 poste508.s11.pfd dockerd[32645]: time="2017-10-02T11:55:05.800716404+02:00" level=warning msg="fai
oct. 02 11:55:06 poste508.s11.pfd dockerd[32645]: time="2017-10-02T11:55:06.198230281+02:00" level=info msg="Graph
oct. 02 11:56:23 poste508.s11.pfd dockerd[32645]: time="2017-10-02T11:56:23.479289269+02:00" level=info msg="Attemp
oct. 03 07:46:40 docker systemd[1]: Stopping Docker Application Container Engine...
oct. 03 07:46:40 docker dockerd[32645]: time="2017-10-03T07:46:40.678359275+02:00" level=info msg="Processing signa
oct. 03 07:46:40 docker dockerd[32645]: time="2017-10-03T07:46:40.731091041+02:00" level=info msg="stopping contain
```

# Le démon docker en mode réseau

- Par défaut, le client Docker et le démon sont sur le même hôte
- Pour connecter le client à un démon Docker exécuté sur un hôte différent, il faut :
  - ▣ Tout d'abord, le daemon Docker doit écouter sur une socket réseau (TCP)
  - ▣ Pour des raisons de sécurité, le démon peut utiliser une socket cryptée. Il faudra configurer TLS
  - ▣ Modifier le client pour se connecter au démon distant

# Types de socket

- unix
- tcp
- La socket par défaut est une socket unix créée sur /var/run/docker.sock
- Les permissions root sont requises

# Erreur de connexion

- Le type message d'erreur ci-dessous signifie généralement
  - Le démon Docker ne fonctionne pas
  - Problème de permission de connection au démon docker
  - Votre client Docker essaie de se connecter au démon en utilisant la socket Unix, mais le démon ne l'écoute pas
  - Vous n'utilisez pas TLS pour vous connecter au démon

```
[root@docker ~]# docker info
```

```
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the
docker daemon running?
```

# Ecoute sur une socket TCP

- Configurer le démon Docker pour écouter sur une socket TCP, en utilisant l'option `--host` ou `-H` et spécifiez l'adresse TCP et le port
- Pour l'adresse, vous pouvez spécifier une adresse IP pour écouter ou spécifier `0.0.0.0` pour écouter toutes les adresses réseau
- Le port `2375` pour une communication non chiffrée
- Le port `2376` pour une communication cryptée

# Ecoute sur une socket TCP

- La socket TCP écoute toutes les adresses réseau

```
dockerd -H tcp://0.0.0.0:2375
```

- La socket TCP écoute une adresse spécifique

```
dockerd -H tcp://10.2.1.1:2375
```

```
[root@docker ~]# cat /etc/sysconfig/docker  
OPTIONS="-H tcp://0.0.0.0:2375"
```

# Connexion du client

- Par défaut, le client Docker suppose que le démon écoute sur une socket Unix
- Il faut configurer le client pour se connecter à un démon docker distant
  - ▣ Utiliser l'option `-H` de la commande `docker`
  - ▣ Configurer la variable d'environnement `DOCKER_HOST`

# Ecoute sur plusieurs sockets

- Le démon Docker peut écouter à la fois la socket Unix et la socket TCP
- Utiliser l'option `-H` plusieurs fois
  - Pour la socket unix
    - `unix:///var/run/docker.sock`
  - Pour la socket réseau
    - `tcp://0.0.0.0:2376`

# Le fichier de configuration

## □ /etc/docker/daemon.json

```
{  
    "dns": [],  
    "storage-driver": "",  
    "labels": [],  
    "debug": true,  
    "hosts": [],  
    "log-level": "",  
    "tls": true,  
    "default-gateway": ""  
}
```

# MODULE 2: SECURITE TLS

# conteneurs Linux et la sécurité

- Docker permet de rendre les applications plus sûres car il fournit un ensemble réduit de privilèges
- Les espaces de noms (namespaces) fournissent une vue isolée du système. Chaque conteneur a son propre :
  - IPC, stack TCP/IP, file system / etc...
- Les processus s'exécutant dans un conteneur ne peuvent pas voir les processus d'un autre conteneur
- Les groupes de contrôle (Cgroups) isolent l'utilisation des ressources par conteneur
- S'assure qu'un conteneur compromis ne fera pas tomber l'hôte entier en épuisant les ressources

# Considérations

- Le démon Docker doit fonctionner en tant que root
- Assurez-vous que seuls les utilisateurs de confiance peuvent contrôler le démon Docker
- Qui fait partie du groupe de docker
- Si vous liez le démon à une socket TCP, sécurisez-le avec TLS

# TLS Transport Layer Security

- Evolution de SSL
- Utilise la cryptographie à clé publique pour crypter les connexions.
- Les clés sont signées avec des certificats gérés par une partie de confiance.
- Ces certificats attestent l'identité du serveur
- Chaque transaction est donc cryptée et authentifiée

# Usage de TLS pour Docker

- Docker fournit des mécanismes pour authentifier à la fois le serveur et le client.
- Fournit une authentification forte, une autorisation et un cryptage pour toute connexion API sur le réseau.
- Les clés client peuvent être distribuées aux clients autorisés
- **Pré-requis**
  - OpenSSL 1.0.1 installé
  - Créez un dossier pour stocker vos clés et assurez-vous que le dossier est protégé en mode 700

# Processus de configuration de TLS

- Créer l'autorité de certification (CA)
  - ▣ Besoin d'une clé privée et d'un certificat CA
- Configurer la clé privée du serveur
- Créer une demande de signature de certificat (CSR) pour le serveur
- Signez la clé du serveur avec le CSR par rapport à notre CA
- Créer une clé privée client et CSR
- Signez la clé du client avec le CSR par rapport à notre autorité de certification
- Exécutez le démon Docker avec TLS activé et spécifiez l'emplacement de la clé privée de l'autorité de certification, du certificat de serveur et de la clé du serveur
  - ▣ Et configurez-le pour écouter sur TCP
- Pointez le client Docker vers l'adresse TCP du démon et spécifiez l'emplacement du certificat client et la clé en tant que clé privée de l'autorité de certification.

# Create the Certificate Authority

- Nous avons besoin de l'autorité de certification pour signer nos clés de serveur et de client
- Créez la clé privée de l'autorité de certification. Vous serez invité à entrer une phrase secrète. Assurez-vous de vous en souvenir
- Créez la clé publique

```
openssl genrsa -aes256 -out ca-key.pem 2048
```

```
openssl req -new -x509 -days 365 \
    -key ca-key.pem -sha256 -out ca.pem
```

# Configurer la clé du serveur et CSR

- La demande de signature de certificat (CSR) est nécessaire pour que nous puissions signer notre clé de serveur.
- Lors de la création du fichier CSR, assurez-vous de spécifier le nom d'hôte de la machine sur laquelle votre démon Docker s'exécute dans l'attribut CN.

## Créer la clé privée du serveur

```
openssl genrsa -out server-key.pem 2048
```

## Créez le CSR.

```
openssl req -subj "/CN=<host name>" \  
           -new -key server-key.pem -out server.csr
```

## Signez la clé du serveur

- Avant de signer notre clé de serveur, nous allons définir une extension de certificat pour spécifier le subjectAltName
- subjectAltName nous permet de spécifier les adresses IP de connexion autorisées

```
echo subjectAltName =  
IP:10.10.10.20,IP:127.0.0.1 > extfile.cnf
```

# Signez la clé du serveur

- Nous signons maintenant la clé du serveur à l'aide de l'autorité de certification que nous avons créée
- Spécifiez le fichier d'extension de certificat (extfile.cnf)

```
openssl x509 -req \
    -days 365 \
    -in server.csr -CA ca.pem \
    -CAkey ca-key.pem \
    -CAcreateserial \
    -out server-cert.pem \
    -extfile extfile.cnf
```

# Créer des clés de client

**Nous créons d'abord la clé privée des clients**

```
openssl genrsa -out client-key.pem 2048
```

**Ensuite, nous créons la demande de signature de certificat client**

```
openssl req -subj '/CN=client' \  
-new \  
-key client-key.pem \  
-out client.csr \  
\\
```

# Signer les clés du client

**Nous avons besoin d'un fichier de configuration d'extension avec l'extension extendedKeyUsage afin de rendre la clé adaptée à l'authentification du client**

```
echo extendedKeyUsage = clientAuth > extfile.cnf
```

**Maintenant, nous pouvons signer notre clé publique client**

```
openssl x509 -req -days 365 \
              -in client.csr \
              -CA ca.pem \
              -CAkey ca-key.pem \
              -CAcreateserial \
              -out client-cert.pem \
              -extfile extfile.cnf
```

# Activer TLS sur le démon Docker

## Options

Exécuter le daemon avec les options

```
dockerd  
--tlsverify  
--tlscacert=<path to ca cert>  
--tlscert=<path to server  
certificate>  
--tlskey=<path to server key>  
-H=0.0.0.0:2376
```

# Protégez nos certificats et clés

- Pour la clé privée CA, la clé privée du serveur et la clé privée du client, vous voulez rendre les fichiers lisibles seulement pour vous  
chmod -v 0400 ca-key.pem client-key.pem \  
server-key.pem
- Pour les certificats, vous voudrez supprimer l'accès en écriture  
chmod -v 0444 ca.pem server-cert.pem client-  
cert.pem
- Il est également recommandé de déplacer le certificat du serveur, la clé privée du serveur et la clé privée CA dans un dossier tel que /etc/docker

# Spécification de TLS sur le client

- Maintenant que le démon Docker a TLS activé, lorsque nous utilisons le client, nous devons spécifier pour activer TLS et spécifier notre certificat client et la clé
- ```
docker
--tlsverify
--tlscacert=<path to ca cert>
--tlscert=<path to client certificate>
--tlskey=<path to client key>
-H=<server url>:2376
```

# Configuration client

- Pour configurer le client, nous pouvons placer notre clé et certificat client avec la clé CA dans le dossier caché .docker. Ce dossier réside dans notre répertoire personnel.
- Les fichiers
  - ▣ ca.pem, cert.pem, key.pem
- Exécuter la commande docker, avec l'option TLS ou les variables d'environnement
  - ▣ --tlsverify
  - ▣ -H  
docker --tlsverify -H 127.0.0.1:2376 ps -a

# Variables d'environnement

```
export DOCKER_HOST="tcp://<ipaddress>:2376"  
export DOCKER_TLS_VERIFY=1
```

# MODULE 3: RESEAU MULTI-HÔTE

# Réseau Multi-hôte

- Les conteneurs fonctionnant sur des hôtes différents ne peuvent pas communiquer entre eux sans mapper leurs ports TCP aux ports TCP de l'hôte
- Le réseau multi-hôte permet à ces conteneurs de communiquer sans nécessiter de mapping de port
- Le Docker Engine prend en charge les réseaux multi-hôtes par le biais du pilote réseau Overlay
- Pré-requis pour créer un réseau Overlay
  - ▣ Accès à un service clé-valeur
  - ▣ Un cluster d'hôtes connecté au service clé-valeur
  - ▣ Tous les hôtes doivent avoir la version kernel 3.16 ou supérieure

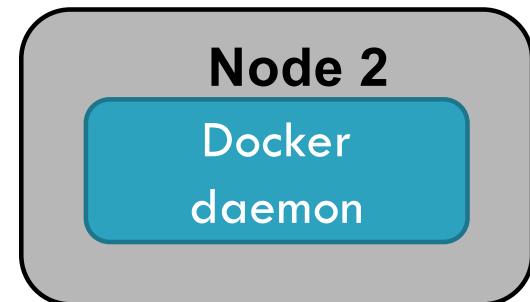
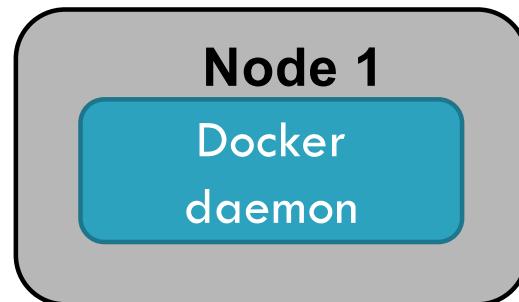
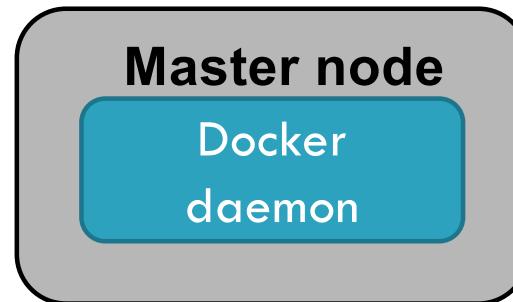
# Service clé-valeur

---

- Stocke les informations sur l'état du réseau
  - ▣ Service de découverte
  - ▣ Adresses IP
- Solutions prises en charge
  - ▣ Consul
  - ▣ Zookeeper
  - ▣ Etcd

# Configuration

- 1 noeud Master stocke la clé-valeur
- 2 noeuds



# Configuration de la clé-valeur

## **Effectuez ceci sur votre nœud Master**

- Nous utiliserons consul comme service de stockage clé-valeur
- Exécutez le consul dans un conteneur avec la commande suivante
- Vérifiez que le consul fonctionne et que le port 8500 est mappé sur l'hôte

```
docker run -p 8400:8400 -p 8500:8500 -p 8600:53/udp -h node1  
program/consul -server -bootstrap
```

# Configurer les autres noeuds

- Le démon Docker sur les autres noueds doit être configuré pour:
  - Écouter sur le port TCP 2375
  - Utiliser la clé-valeur du Consul sur notre noeud maître

# Configurer démon Docker

- Modifier la variable OPTIONS dans le fichier :

/etc/sysconfig/docker

```
OPTIONS="-H tcp://0.0.0.0:2375 \
         -H unix:///var/run/docker.sock \
         --cluster-store=consul://<Master Node IP>:8500/network \
         --cluster-advertise=eth0:2375"
```

# Configurer le réseau Overlay

## **Effectuez ceci sur le Node1 ou le Node2**

- Nous allons créer un réseau Overlay appelé multinet
- Il sera configuré avec le sous-réseau 10.10.10.0/24

```
docker network create -d overlay --subnet  
10.10.10.0/24 multinet
```

# Visualisation

---

- Une fois que vous avez créé le réseau Overlay, vérifiez qu'il est présent

```
docker network ls
```

- Vérifiez sur l'autre noeud que le réseau est également présent

# Conteneurs sur un réseau multi-hôte

- Pour exécuter un conteneur sur le réseau multi-hôte, il vous suffit de spécifier le nom du réseau sur la commande docker

```
docker run -itd --name c1 --net multinet busybox
```

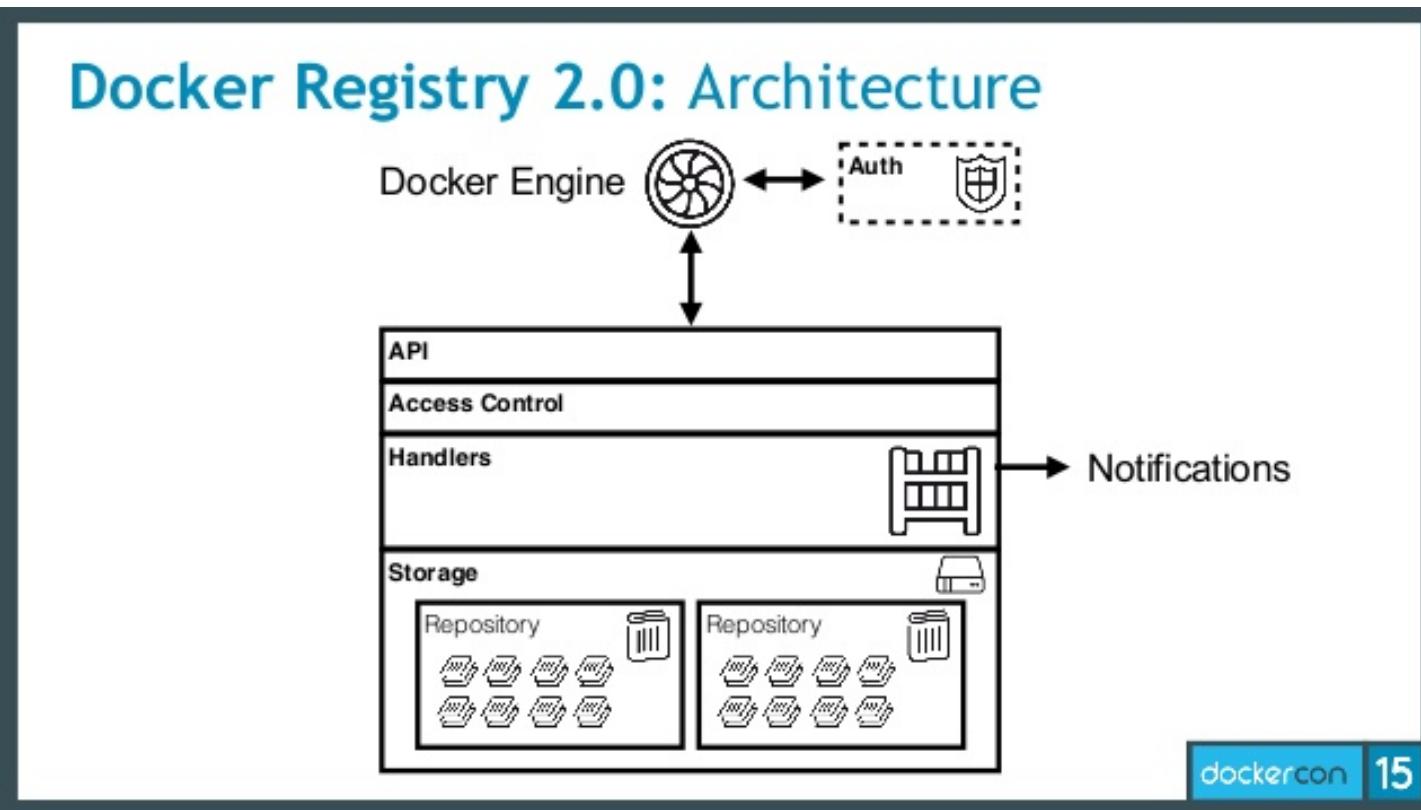
- A la création d'un réseau Overlay, Docker crée également un autre réseau appelé docker\_gwbridge .
- Le réseau docker\_gwbridge fournit un accès externe pour les conteneurs

# MODULE 4: SERVEUR DE REGISTRE PRIVE

# Serveur de registre

- Exécutez votre propre serveur de registre pour stocker et distribuer des images au lieu d'utiliser Docker Hub
- Plusieurs options
  - ▣ Exécuter le serveur de registre en utilisant le conteneur
  - ▣ Docker Trusted Registry
- Deux versions
  - ▣ Registry v1.0 jusqu'à Docker 1.5
  - ▣ Registry v2.0 depuis Docker 1.6

# Architecture



# Fonctionnalités

---

- Stockage configurable
  - ▣ Local disk
  - ▣ Amazon S3
  - ▣ Microsoft Azure
- Webhooks pour lancer une construction ou envoyer des notifications à certaines personnes lorsque des images ont été poussées
- Accès sécurisé aux images via TLS

# Public ou private

---

- Vous pouvez choisir d'exécuter un serveur de registre qui est accessible au public
- Vous pouvez le mettre derrière le pare-feu et le rendre disponible uniquement pour le personnel interne de l'entreprise

# Configuration d'un serveur de registre

- Deux méthodes
  - Utilisez l'image officielle du registre sur le hub docker
  - Téléchargez la source de distribution et créez votre propre image de registre personnalisée
- L'image officielle contient une version pré-configurée du registre v2.0
- L'image officielle est destinée à des fins d'évaluation car sa configuration par défaut ne convient pas à l'utilisation de la production
  - Pas de TLS

```
docker run -d -p 5000:5000 registry:2.0
```

# Push

- D'abord, créer un tag de l'image avec l'hôte IP ou le domaine du serveur de registre, puis exécuter la commande docker push

```
[root@docker ~]# docker tag 09ea64205e55 myserver.com:5000/masociete/monimage:1.0
```

```
[root@docker ~]# docker push myserver.com:5000/masociete/monimage:1.0
The push refers to a repository [myserver.com:5000/masociete/monimage]
```

# Lister les tags dans le registre

- Pour lister les tags d'une image, il faut faire une requête du type :

<registry host>:<port>/v2/<repo name>/tags/list

```
[root@docker ~]# curl http://mycompany.com:5000/v2/masociete/monimage/tags/list
{"name": "masociete/monimage", "tags": ["1.0"]}
```

# Pull

- Pour extraire une image d'un serveur de registre, il faut
  - L' URL du serveur et le port
  - Image repository
  - Image tag

# Registre non sécurisé

- Par défaut, le démon Docker suppose que tous les registres sont sécurisés et bloque la communication avec des registres non sécurisés
  - ▣ Impossible de faire un push ou pull
- Pour permettre une communication avec un registre non sécurisé, le Daemon doit être démarré avec l'option --insecure-registry et chaque registre doit être ajouté

```
--insecure-registry myregistry:5000
```

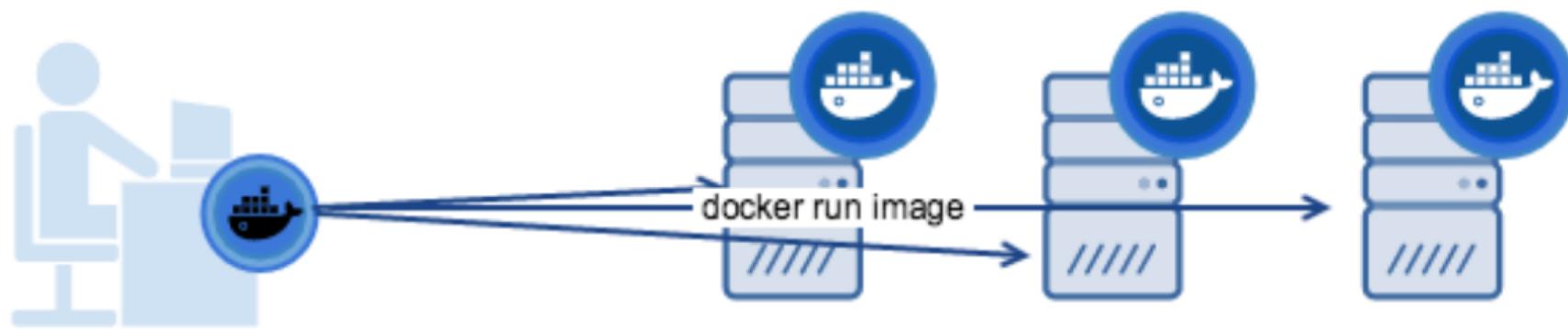
# MODULE 5: DOCKER MACHINE

# Docker Machine

---

- **Docker Machine** est un outil qui provisionne des docker hôtes et installe le démon Docker
- Crée des docker hôtes supplémentaires sur votre propre ordinateur
- Crée des docker hôtes sur le cloud (Amazon AWS, DigitalOcean etc...)
- Docker Machine crée le serveur, installe le démon Docker et configure le client Docker

# Docker Machine



# Installing Machine

- Téléchargez la version binaire spécifique au système d'exploitation à l'adresse  
<https://github.com/docker/machine/releases>
- Placez le binaire dans un dossier de votre système
  - /usr/local/bin

# Version

- Exécuter docker-machine -v pour afficher la version
- Exécuter docker-machine -help pour avoir la liste des commandes

# Docker Machine sur Windows et OSX

---

- Fait partie du Docker Toolbox
- Toolbox installe Docker Machine et le client Docker

# docker machine

- La commande `docker-machine` permet de créer et gérer les Docker hosts sur divers environnements :
  - VirtualBox
  - Amazon AWS
  - DigitalOcean
  - Azure
  - Rackspace
  - etc ...
- Chaque environnement possède son propre plugin dans le binaire `docker-machine`

# Création d'un hôte

- Utilisez la commandes **docker-machine create** et spécifiez le driver de l'environnement
- Le pilote permet à docker-machine d'interagir avec l'environnement où vous souhaitez créer l'hôte
- ```
docker-machine create --driver <driver>
<hostname>
```

# VirtualBox

- L'utilisation de VirtualBox nous permet de fournir rapidement des hôtes Docker supplémentaires sur notre Windows ou Mac
- `docker-machine create --driver virtualbox testhost`

# VirtualBox

- docker-machine va télécharger la distribution Linux boot2docker, créer et démarrer une machine virtuelle VirtualBox qui exécute Docker
- docker-machine va créer automatiquement la clé SSH pour votre hôte

```
[root@docker ~]# docker-machine create --driver virtualbox host01
Creating CA: /root/.docker/machine/certs/ca.pem
Creating client certificate: /root/.docker/machine/certs/cert.pem
Running pre-create checks...
(host01) Image cache directory does not exist, creating it at
/root/.docker/machine/cache...
(host01) No default Boot2Docker ISO found locally, downloading the latest release...
(host01) Latest release for github.com/boot2docker/boot2docker is v17.09.0-ce
(host01) Downloading /root/.docker/machine/cache/boot2docker.iso from
https://github.com/boot2docker/boot2docker/releases/download/v17.09.0-ce/boot2docker.iso...
```

# Provisionnement dans le cloud

- Chaque fournisseur de cloud offre différentes options sur la commande create machine docker et leur propre driver
- Liste des drivers
  - Amazon Web Services
  - Google Compute Engine
  - IBM Softlayer
  - Microsoft Azure
  - Microsoft Hyper-V
  - Openstack
  - Rackspace
  - Oracle VirtualBox
  - VMware Fusion
  - VMware vCloud Air
  - VMware vSphere

# DigitalOcean

- Vous aurez besoin de votre jeton d'accès au compte DigitalOcean
- Spécifier la droplet (par défaut 512mb)
- L'image à installer (par défaut ubuntu-14-04-x64)
- La région

```
docker-machine create
  --driver digitalocean \
  --digitalocean-access-token <your access token> \
  --digitalocean-size 2gb \
testhost
```

# AWS

- Pour créer des hôtes dans AWS, vous aurez besoin de votre
  - ▣ Clé d'accès AWS
  - ▣ Clé secrète AWS
  - ▣ L'ID VPC où lancer l'instance
- L'image par défaut utilisée est Ubuntu 14.04 LTS

```
docker-machine create
  --driver amazonec2 \
  --amazonec2-access-key <AWS access key> \
  --amazonec2-secret-key <AWS secret key> \
  --amazonec2-vpc-id <VPC ID> \
testhost
```

# Liste des machines

- La commande `docker-machine ls` affiche toutes les machines hôtes qui ont été provisionnées
- Des hôtes sur différents fournisseurs de cloud

```
[root@docker ~]# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS
aws1	-	amazonec2	Running	tcp://34.215.62.69:2376		v17.09.0-ce	
aws2	-	amazonec2	Running	tcp://35.160.76.216:2376		v17.09.0-ce	
node1	-	virtualbox	Running	tcp://192.168.99.100:2376		v17.06.2-ce	
node2	-	virtualbox	Running	tcp://192.168.99.101:2376		v17.06.2-ce	

# Connexion au hôte

---

- Il existe 2 méthodes pour se connecter à un hôte que le docker-machine a approvisionné
  - Utiliser la commande docker-machine ssh
  - Définissez les variables d'environnement pour pointer votre client Docker vers le démon sur l'hôte distant

# docker-machine env

- La commande `env` affiche les variables d'environnements qui doivent être configurées pour connecter votre client Docker au démon distant de l'hôte spécifié
- `docker-machine env <hostname>`

```
[root@docker ~]# docker-machine env aws1
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://34.215.62.69:2376"
export DOCKER_CERT_PATH="/home/user1/.docker/machine/machines/aws1"
export DOCKER_MACHINE_NAME="aws1"
# Run this command to configure your shell:
# eval $(docker-machine env aws1)
```

# Connexion

- Exécuter la commande `eval $(docker-machine env <hostname>)` pour connecter le client Docker au démon du hôte
  - ▣ Fonctionne en définissant des variables d'environnement sur le client
- Exécuter la commande `eval $(docker-machine env - )` pour déconnecter le client Docker du démon du hôte

# Hôte Actif

- Sur la sortie standard de la commande docker-machine ls la colonne “ACTIVE”, indique le hôte actif.
- L'hôte actif est la machine sur laquelle le client Docker est connecté
- L'hôte actif est configuré lors de l'exécution de la commande :  
`eval $(docker-machine env <hostname>)`
- La commande docker-machine active indique aussi le hôte actif

# Réglage de l'hôte actif

```
[root@docker ~]# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS
node1	-	virtualbox	Running	tcp://192.168.99.100:2376			v17.09.0-ce
node2	*	virtualbox	Running	tcp://192.168.99.101:2376			v17.06.2-ce

```
[root@docker ~]# eval $(docker-machine env node2)
```

```
[root@docker ~]# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS
aws1	-	amazonec2	Stopped				Unknown
aws2	-	amazonec2	Stopped				Unknown
node1	-	virtualbox	Running	tcp://192.168.99.100:2376			v17.09.0-ce
node2	*	virtualbox	Running	tcp://192.168.99.101:2376			v17.06.2-ce

# Déconnexion du client

- Pour déconnecter le client Docker du démon distant, il faut désactiver les variables
  - ▣ DOCKER\_TLS\_VERIFY
  - ▣ DOCKER\_CERT\_PATH
  - ▣ DOCKER\_HOST
- Vous pouvez utiliser la commande unset ou la commande suivante

```
eval $(docker-machine env -u)
```

# Docker machine SSH

- La commande docker-machine ssh nous permet de nous connecter à un hôte approvisionné en utilisant SSH
  - Connexion en utilisant la clé SSH créée lors de la création du hôte

# Docker machine SSH

- Peut également être utilisé pour exécuter une commande sur la machine spécifiée

```
[root@docker ~]# docker-machine ssh node1 docker version
Client:
Version:      17.09.0-ce
API version:  1.32
Go version:   go1.8.3
Git commit:   afdb6d4
Built:        Tue Sep 26 22:39:28 2017
OS/Arch:      linux/amd64

Server:
Version:      17.09.0-ce
API version:  1.32 (minimum version 1.12)
Go version:   go1.8.3
Git commit:   afdb6d4
Built:        Tue Sep 26 22:45:38 2017
OS/Arch:      linux/amd64
Experimental: false
```

# Démarrer et arrêter un hôte

- Arrêter une machine hôte
  - docker-machine stop <machine name>
- Démarrer une machine hôte arrêtée
  - docker-machine start <machine name>
- Pour redémarrer une machine hôte
  - docker-machine restart <machine name>

```
[root@docker ~]# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS
node1	-	virtualbox	Running	tcp://192.168.99.100:2376		v17.09.0-ce	
node2	*	virtualbox	Running	tcp://192.168.99.101:2376		v17.06.2-ce	

```
[root@docker ~]# docker-machine stop node1
```

```
Stopping "node1"...
```

```
Machine "node1" was stopped.
```

```
[root@docker ~]# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS
node1	-	virtualbox	Stopped			Unknown	
node2	*	virtualbox	Running	tcp://192.168.99.101:2376		v17.06.2-ce	

# Inspecting host details

- docker-machine inspect permet d'obtenir des détails sur la machine hôte, les informations du driver, les chemins de certification, l'adresse IP ...

```
[root@docker ~]# docker-machine inspect node2
{
    "ConfigVersion": 3,
    "Driver": {
        "IPAddress": "192.168.99.101",
        "MachineName": "node2",
        "SSHUser": "docker",
        "SSHPort": 57355,
        "SSHKeyPath": "/Users/samir/.docker/machine/machines/node2/id_rsa",
        "StorePath": "/Users/samir/.docker/machine",
        "SwarmMaster": false,
        "SwarmHost": "tcp://0.0.0.0:3376",
        "SwarmDiscovery": "",
        "VBoxManager": {},
        "HostInterfaces": {},
        "CPU": 2,
        "Memory": 2048,
        "DiskSize": 20000,
```

# Obtenir l'adresse IP d'un hôte

- Vous pouvez voir l'adresse IP en regardant la colonne URL sur la sortie de la commande

```
docker-machine ls
```

- Par la commande

```
docker-machine ip <host name>
```

```
[root@docker ~]# docker-machine ip node2  
192.168.99.101
```

# Suppression d'hôtes

- La commande docker-machine rm supprime l'hôte
- Cela supprimera l'hôte sur l'environnement (local ou cloud) et supprime le dossier de référence local

/home/<user>/ .docker/machine/machines/<machine name>

```
[root@docker ~]# docker-machine rm node1
About to remove node1
WARNING: This action will delete both local reference and remote instance.
Are you sure? (y/n) : y
Successfully removed node1
```

# MODULE 6: DOCKER SWARM



# Swarm

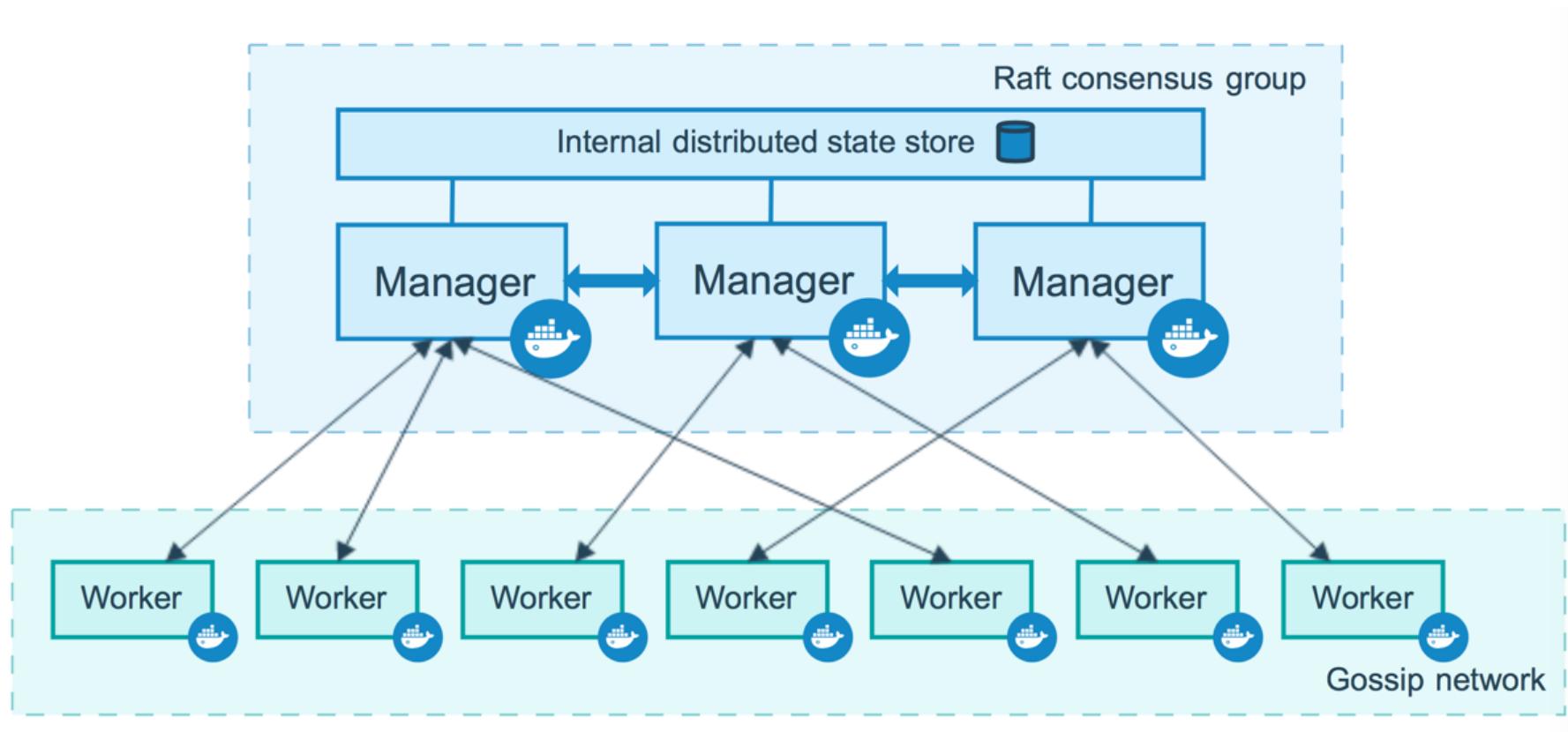


- **Docker Swarm** est un cluster de hôtes Docker en cluster sur lequel on déploie des conteneurs
- Permet de distribuer les charges de travail des conteneurs sur plusieurs machines s'exécutant dans un cluster
- L'API de Docker inclue des commandes pour gérer les noeuds du cluster (ajout, suppression des nœuds), et déployer et organiser des services à travers le cluster

# Modèle

- Pour déployer votre application dans un cluster, vous soumettez un service à un nœud de Manager. Le nœud Manager distribue les unités de travail appelées tasks aux noeuds Worker.
- Les nœuds de gestion effectuent également l'orchestration et les fonctions de gestion de cluster requises pour maintenir l'état souhaité du cluster. Les nœuds Manager élisent un seul Maître pour mener des tâches d'orchestration
- Les nœuds Worker reçoivent et exécutent les tâches envoyées depuis les nœuds Manager. Par défaut, les nœuds Manager exécutent également des services en tant que nœuds Worker, mais vous pouvez les configurer pour exécuter des tâches de gestion

# Architecture



# Agent

---

- Un agent s'exécute sur chaque nœud de travail et rapporte les tâches qui lui sont affectées. Le nœud Worker notifie au nœud du gestionnaire l'état actuel de ses tâches assignées afin que le gestionnaire puisse maintenir l'état désiré de chaque Worker.

# Service



- Un service est la définition des tâches à exécuter sur le noeud Manager ou les nœuds Worker. C'est la structure centrale du système swarm.
- Lorsque vous créez un service, vous spécifiez l'image du conteneur à utiliser et les commandes à exécuter à l'intérieur des conteneurs en cours d'exécution.

# Modèle



- Dans le modèle de services **Replicas**, le Manager distribue un nombre de copie spécifié de tâches aux Workers
- Dans le modèle de services **Global**, swarm exécute une tâche pour le service sur chaque noeud disponible dans le cluster.

# Modèle



- Une tâche comporte un conteneur Docker et les commandes à exécuter à l'intérieur du conteneur. C'est l'unité de planification atomique de swarm.
- Les noeuds Manager attribuent des tâches aux nœuds du travail en fonction du nombre de réplicas définies dans l'option scale du service.
- Une fois qu'une tâche est attribuée à un noeud, elle ne peut pas se déplacer vers un autre nœud. Il ne peut fonctionner que sur le noeud assigné ou échouer.

# Création du cluster swarm

- La commande suivante permet de créer un nouveau cluster swarm :

```
docker swarm init --advertise-addr <MANAGER-IP>
```

```
docker@node1:~$ docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (f9js2mda7uf70bqydetmcgo6p) is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-3716t7lma6yeexj1mirjhnm2yh5ydrft5 192.168.99.100:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

# Etat du cluster swarm

- La commande suivante permet de visualiser l'état du cluster swarm :

docker info

```
docker@node1:~$ docker info
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
Volume: local
Network: bridge host macvlan null overlay
Log: awslogs fluentd gcplogs gelf journalctl json-file logentries splunk syslog
Swarm: active
  NodeID: f9js2mda7uf70bqydetmcgo6p
  Is Manager: true
  ClusterID: xh1d7tk4ja3qgk5nf5n1jvjp1
  Managers: 1
  Nodes: 1
  Orchestration:
    Task History Retention Limit: 5
```

# Liste des noeuds

- La commande suivante permet de lister les noeuds du cluster swarm :

```
docker node ls
```

```
docker@node1:~$ docker node ls
ID                  HOSTNAME        STATUS        AVAILABILITY   MANAGER
STATUS
f9js2mda7uf70bqydetmcg06p *  node1          Ready        Active        Leader
```

# Ajout des Workers

La commande suivante permet d'ajouter un Worker dans le cluster swarm :

```
docker swarm join --token <token> IP-Manager
```

```
docker@node2:~$ docker swarm join --token SWMTKN-1-3716t7lm29 192.168.99.100:2377
This node joined a swarm as a worker.
```

```
docker@node1:~$ docker node ls
ID                  HOSTNAME        STATUS        AVAILABILITY   MANAGER
STATUS
f9js2mda7uf70bqydetmcgo6p *  node1          Ready        Active        Leader
2bkuf00chye6uy4cebr2728jr  node2          Ready        Active
```

# Déployer un service

La commande suivante permet de déployer un service dans le cluster swarm avec le nombre de répliques désiré :

```
docker service create --replicas <#> Image
```

```
docker@node1:~$ docker service create --replicas 2 --name ping alpine ping 127.0.0.1  
loemijbv1kne7wo2srkf04kiv
```

Since `--detach=false` was not specified, tasks will be created in the background.

In a future release, `--detach=false` will become the default.

```
docker@node1:~$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE
PORTS				
loemijbv1kne	ping	replicated	2/2	
alpine:latest				

# Inspecter un service

```
docker service inspect --pretty <service>
```

```
docker@node1:~$ docker service inspect --pretty ping
```

```
ID:          loemi7bv1kne7wo2srkf04kiv
Name:        ping
Service Mode: Replicated
Replicas:    2
Placement:
UpdateConfig:
  Parallelism: 1
  On failure:  pause
  Monitoring Period: 5s
  Max failure ratio: 0
```

# Distribution des services

```
docker service ps <service>
```

```
docker@node1:~$ docker service ps ping
ID                  NAME                IMAGE              NODE              DESIRED
STATE      CURRENT STATE          ERROR             PORTS
nd9ntu992wex    ping.1        alpine:latest    node1            Running
Running 10 minutes ago
acixdloq7zrq    ping.2        alpine:latest    node2            Running
Running 10 minutes ago
```

# Modifier le nombre de replicas

```
docker service scale    service=#
```

```
docker@node1:~$ docker service scale ping=4
ping scaled to 4
Since --detach=false was not specified, tasks will be scaled in the background.
In a future release, --detach=false will become the default.
```

```
docker@node1:~$ docker service ps ping
```

ID	NAME	IMAGE	NODE	DESIRED
STATE	CURRENT STATE	ERROR	PORTS	
nd9ntu992wex	ping.1	alpine:latest	node1	Running
Running 14 minutes ago				
acixdloq7zrq	ping.2	alpine:latest	node2	Running
Running 14 minutes ago				
js2ubbkubb8a	ping.3	alpine:latest	node3	Running
Preparing 3 seconds ago				
64gkhoa3invol	ping.4	alpine:latest	node3	Running
100Preparing 3 seconds ago				

# Noeud en état drain

```
docker node update --availability drain <node>
```

```
docker@node1:~$ docker node update --availability drain node2
Node2
```

```
docker@node1:~$ docker service ps ping
ID                  NAME                IMAGE              NODE          DESIRED
STATE      CURRENT STATE    ERROR             PORTS
nd9ntu992wex      ping.1        alpine:latest    node1        Running
Running 20 minutes ago
jbr3gzlmywgc       ping.2        alpine:latest    node1        Running
Running 10 seconds ago
acixdloq7zrq       \_ ping.2     alpine:latest    node2        Shutdown
Shutdown 10 seconds ago
js2ubbkubb8a       ping.3        alpine:latest    node3        Running
Running 5 minutes ago
64gkao3invol       ping.4        alpine:latest    node3        Running
Running 5 minutes ago
```

# Noeud en état active

```
docker node update --availability active <node>
```

```
docker@node1:~$ docker node update --availability active node2
node2
```

```
docker@node1:~$ docker service ps ping
```

ID	NAME	IMAGE	NODE	DESIRED
STATE	CURRENT STATE	ERROR	PORTS	
adj283f3acaq	ping.1	alpine:latest	node1	Running
	Running 48 seconds ago			
sed6bo01e5o5	\_			
ping.1	alpine:latest	node2	Shutdown	Shutdown 50
	seconds ago			
rqcjsjz9qaxn	ping.2	alpine:latest	node1	Running
	Running 2 minutes ago			
4jt9khyb4exe	ping.3	alpine:latest	node3	Running
	Running about a minute ago			
1522wwvt3o18	ping.4	alpine:latest	node3	Running
	Running about a minute ago			

# Noeud en état pause

```
docker node update --availability pause <node>
```

```
docker@node1:~$ docker node update --availability pause node2
node2
```

```
docker@node1:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MA
NAGER STATUS	ENGINE VERSION			
upa8ru65qmpdu4vpe03qqr0u1				
* node1	Ready	Active	Leader	18.05.0-ce
oz3yp2dmri87hb8gnvpsqvi0h	node2	Ready	Pause	18.05.0-ce
t9rhmo12lne238hngrq3j95vj	node3	Ready	Active	18.05.0-ce

# Noeud promote

```
docker node update --availability pause <node>
```

```
docker@node1:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER
STATUS	ENGINE VERSION			
upa8ru65qmpdu4vpe03qqr0u1				
* node1	Ready	Active	Leader	18.05.0-ce
oz3yp2dmri87hb8gnvpsqvi0h	node2	Ready	Active	18.
05.0-ce				
t9rhmo12lne238hngrq3j95vj	node3	Ready	Active	18.
05.0-ce				

```
docker@node1:~$ docker node promote node2
```

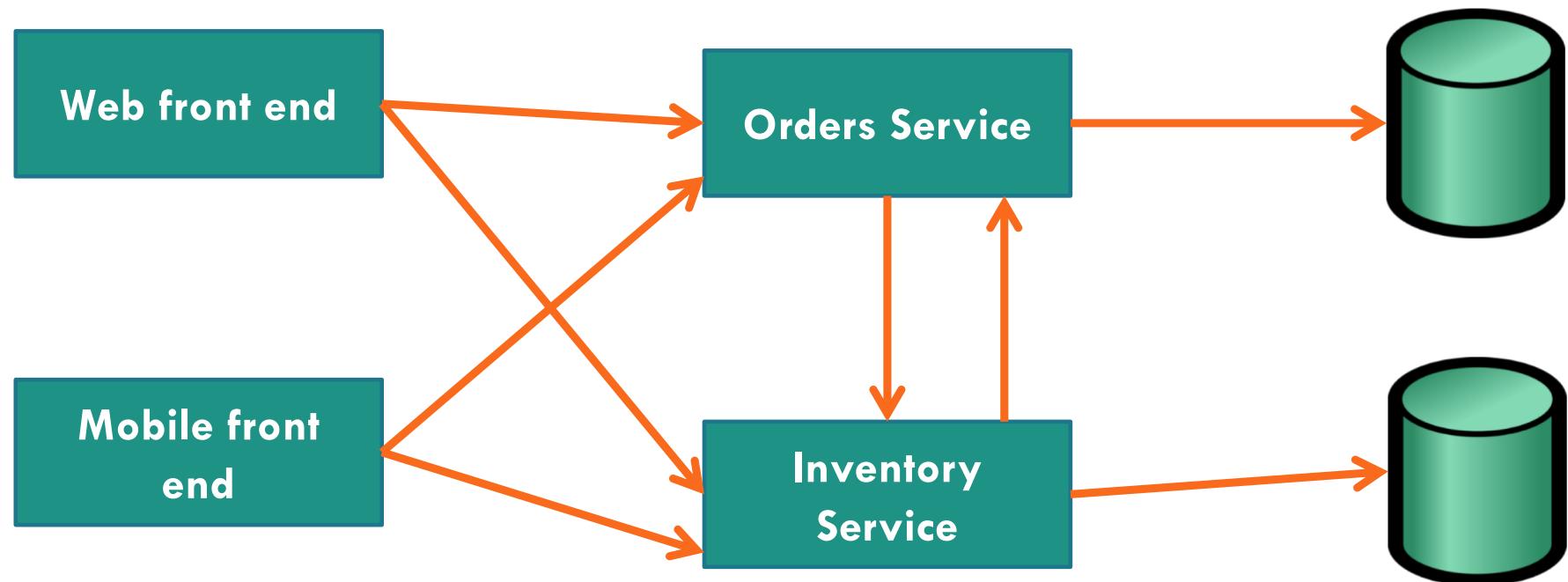
Node node2 promoted to a manager in the swarm.

```
docker@node1:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER
STATUS	ENGINE VERSION			
upa8ru65qmpdu4vpe03qqr0u1				
* node1	Ready	Active	Leader	18.05.0-ce
oz3yp2dmri87hb8gnvpsqvi0h	node2	Ready	Active	Reachable
05.0-ce				
t9rhmo12lne238hngrq3j95vj	node3	Ready	Active	18.
05.0-ce				

# MODULE 7: DOCKER COMPOSE

# Exemple d'application



# Les micro-service de l'application



# Compose

- Docker Compose est un outil pour créer et gérer des applications multi-conteneurs
- Les conteneurs sont tous définis dans un seul fichier appelé docker-compose.yml
- Chaque conteneur gère un composant / service particulier de votre application
  - Web front end
  - User authentication
  - Payments
  - Database
- Les liens entre conteneurs sont définis
- Compose lancera tous vos conteneurs en une seule commande

# Installer Compose

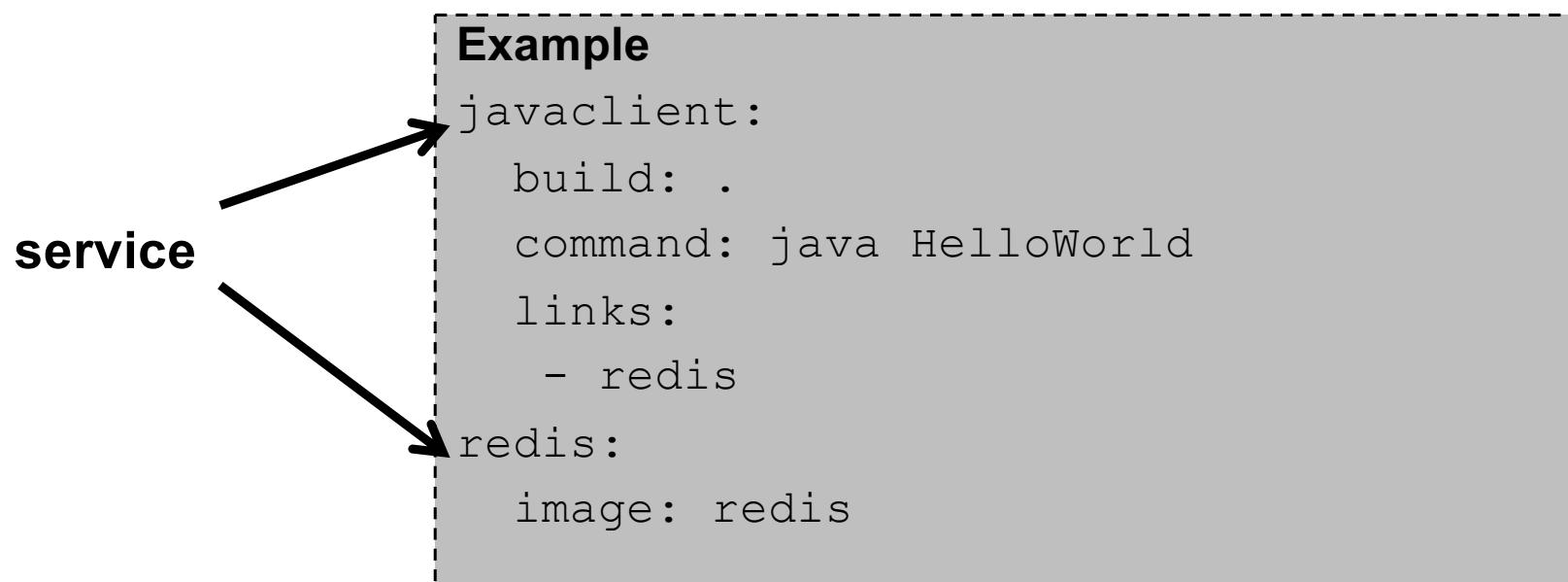
- Vérifiez la dernière version sur

<https://github.com/docker/compose/releases>

**MAC et Windows : Docker Compose est inclus dans Docker Toolbox**

# Compose fichier yml

- Définit les services qui composent votre application
- Chaque service contient des instructions pour construire et gérer un conteneur



# Instruction Build

- Build définit le chemin d'accès au fichier Docker qui sera utilisé pour créer l'image
- Le conteneur sera exécuté à l'aide de l'image construite
- Le chemin d'accès à la construction peut être un chemin relatif. Par rapport à l'emplacement du fichier yml

```
javaclient:  
  build: .  
orderservice:  
  build:  
  /src/com/company/service
```



# Instruction Image

- L'image définit l'image qui sera utilisée pour exécuter le conteneur
- L'image peut être locale ou distante
- Peut spécifier une étiquette ou un ID d'image
- Tous les services doivent comporter une instruction build ou image

Use the latest redis  
Image from Docker  
Hub

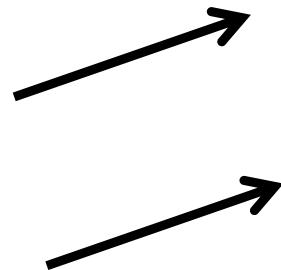
```
javaclient:  
  image:  
    johnnytu/myclient:1.0  
redis:  
  image: redis
```

# Compose fichier yml

- Version
- Référence top-level

**Utilisation**

**Déclaration**



```
version: "3.2"
services:
  web:
    image: nginx:alpine
    volumes:
      - dbdata:/var/lib/postgresql/data
volumes:
  mydata:
    dbdata:
```

# Exemple

- Deux services
  - Java client
  - Redis
- Le client java est une classe Java simple qui se connecte à notre serveur Redis et obtient la valeur d'une clé
- Code disponible sur <https://github.com/johnny-tu/HelloRedis.git>

# Code Java

```
import redis.clients.jedis.Jedis;

public class HelloRedis
{
    public static void main (String args [])
    {
        Jedis jedis = new Jedis("redisdb");

        while (true) {
            try {
                Thread.sleep(5000);
                System.out.println("Server is running: "+jedis.ping());
                String bookCount = jedis.get("books_count");
                System.out.println("books_count = " + bookCount);
            }
            catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

# Dockerfile

```
FROM java:7
COPY /src /HelloRedis/src
COPY /lib /HelloRedis/lib

WORKDIR /HelloRedis
RUN javac -cp lib/jedis-2.1.0-
sources.jar -d . \
    src/HelloRedis.java
```

# Links

- Crée une entrée pour l'alias à l'intérieur des conteneurs dans le fichier /etc/hosts

```
javaclient:  
  build: .  
  command: java HelloWorld  
  links:  
    - redis  
redis:  
  image: redis
```

# Executer l'application

- **docker-compose up**
- La commande Up
  - ▣ Créez l'image pour chaque service
  - ▣ Créer et démarrer les conteneurs
  - ▣ Les conteneurs source sont démarrés avant les destinataires
- Les conteneurs peuvent tous fonctionner au premier plan ou en mode détaché

# Exécuter l'application

```
docker@node1:~/HelloRedis-master$ docker-compose up
Pulling redis (redis:latest)...
latest: Pulling from library/redis
065132d9f705: Pull complete
Status: Downloaded newer image for redis:latest
Building javaclient
Step 1/6 : FROM java:7
7: Pulling from library/java
6263baad4f89: Pull complete
Creating helloredismaster_redis_1 ...
Creating helloredismaster_javaclient_1 ...
Attaching to helloredismaster_redis_1, helloredismaster_javaclient_1
redis_1      | 1:C 05 Oct 04:39:55.519 # o00Oo00Oo00Oo Redis is starting o00Oo00Oo00Oo
redis_1      | 1:M 05 Oct 04:39:55.521 * Ready to accept connections

javaclient_1 | Server is running: PONG
javaclient_1 | books_count = null
```

# Visualiser containers

- docker-compose ps permet de lister les services lancés par Compose
- Cela affichera uniquement les services qui ont été lancés depuis Compose tel que définis dans le fichier docker-compose.yml
- La commande doit être exécutée dans le dossier avec le fichier yml

```
docker@node1:~/HelloRedis-master$ docker-compose ps
      Name           Command           State    Ports
-----
helloredismaster_javaclient_1   java HelloRedis
helloredismaster_redis_1        docker-entrypoint.sh redis ...
                                         ...                                Up
                                         ...                                Up      6379/tcp
```

# Start et stop

- **Arrêt d'un service**

```
docker-compose stop <service name>
```

- **Arrêt de tous les services**

```
docker-compose stop
```

- **Redémarrage d'un service**

```
docker-compose start <service name>
```

- **Redémarrage de tous les services**

```
docker-compose start
```

# Suppression

- Vous pouvez supprimer manuellement chaque conteneur de service avec la commande docker rm
- Ou docker-compose rm pour supprimer tous les conteneurs de service qui ont été arrêtés
- Spécifier un service à supprimer  
docker-compose rm <service name>
- Utilisez l'option -v pour supprimer les volumes associés  
docker-compose rm -v <service name>

# logs

`docker-compose logs`

- Si un service n'est pas spécifié, le journal agrégé de tous les conteneurs sera affiché

# Scale

- Dans une architecture de micro-service, nous avons la flexibilité d'étendre un service particulier pour gérer une charge plus grande

```
docker-compose scale <service name>=<instances>
```

```
docker@node1:~/HelloRedis-master$ docker-compose scale javaclient=2
Starting helloredismaster_javaclient_1 ... done
Creating helloredismaster_javaclient_2 ...
Creating helloredismaster_javaclient_2 ... Done
```

```
docker@node1:~/HelloRedis-master$ docker-compose ps
```

Name	Command	State	Ports
-----			
helloredismaster_javaclient_1	java HelloRedis	Up	
helloredismaster_javaclient_2	java HelloRedis	Up	
helloredismaster_redis_1	docker-entrypoint.sh redis ...	Up	6379/tcp

## Compose et Dockerfile

- La plupart des paramètres dans un fichier `docker-compose.yml` ont une instruction équivalente dans Dockerfile
- Les options déjà spécifiées dans Dockerfile sont respectées par Docker Compose et n'ont pas besoin d'être spécifiées à nouveau

# Docker Compose build

- docker-compose build command construira les images pour vos services définis sans démarrer les services
- Tags des images <project name>\_<service name>:latest
- --no-cache Ne pas utiliser le cache lors de la construction de l'image

```
docker@node1:~/HelloRedis-master$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
helloredismaster_javaclient	latest	0179fc6723b3	5 minutes ago	585MB

# Variable substitution

- Les variables d'environnement peuvent être utilisées dans votre fichier de configuration YML
- **Syntaxe** \${VARIABLE} ou \${ VARIABLE }

```
webapp:  
    image: jtu/mywebapp:${MYAPP_VERSION}  
...  
...
```

# Specifier autre fichier yml

- Le fichier de configuration par défaut est docker-compose.yml
- Nous pouvons spécifier un autre fichier de configuration en utilisant l'option -f  
docker-compose -f docker-ver3.yml

# Multiples fichiers yml

- Lorsque plusieurs fichiers de configuration Compose sont spécifiés à l'aide de l'option -f, la configuration de ces fichiers est combinée.

```
docker-compose -f docker-compose.yml \
               -f docker-compose.debug.yml
```

- La configuration est construite dans l'ordre des fichiers listés