INTRODUCTION À LA CONTENEURISATION : DOCKER

Table des matières

QU'EST-CE QUE DOCKER ?

Les conteneurs ont permis de développer une nouvelle approche dans le développement logiciel : la portabilité. En effet, les lignes de code sont empaquetées en respectant un standard afin d'étendre la flexibilité dans leur intégration et offrir un mécanisme d'exécution rapide sur un système d'exploitation tiers.

Docker est une plate-forme libre sous licence Apache 2[1] écrit en *GoLang*[2], basée sur la technologie de virtualisation en « *conteneurs logiciels* », dédiée principalement aux administrateurs systèmes et développeurs pour le développement, les tests, la livraison et l'exécution d'applications distribuées.

Depuis sa version 0.9[3], *Docker* vient compléter son API, à l'origine basée sur les fonctionnalités offertes par le gestionnaire de conteneur Linux (*LXC*) telles que *CGroups*, en implémentant un moteur de conteneurisation nommé *LibContainer* (voir Schéma 1).

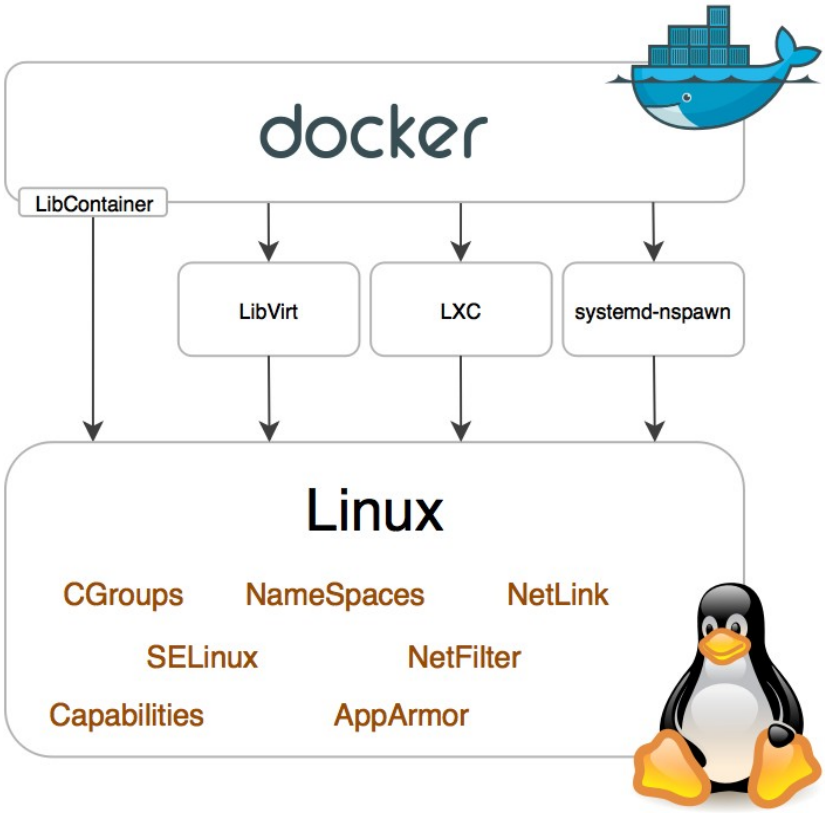
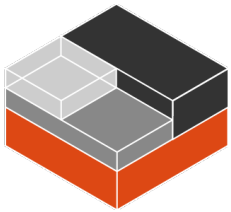


Schéma 1: Interfaces de virtualisation de Docker basée sur LibContainer[3]

QUELQUES DÉFINITIONS

LXC



LXC[4] (ou « *Linux Containers* ») est un système de virtualisation de type *OSLV* (« *Operating-System-Level Virtualization* »), c'est-à-dire que le noyau du système d'exploitation permet le partitionnement des ressources logiques (S/W ou « *Software* ») d'un système, contrairement aux systèmes de para-virtualisation basée sur un partitionnement des ressources physiques (H/W ou « *Hardware* ») d'un ordinateur (voir Schéma 2).

Un système de virtualisation *OSLV* permet ainsi de supporter l'exécution isolée et simultanée de plusieurs contextes utilisateurs (instance ou « *userspace* »), c'est-à-dire qu'ils ne possèdent aucun accès direct aux ressources matériels de l'ordinateur, gérés par le noyau du système d'exploitation hôte.

Cependant, l'exécution isolée et simultanée de plusieurs espaces-utilisateurs n'est pas sans difficulté. En effet, la relation de concurrence induite de cette exécution, de par l'indéterminisme de l'accès aux ressources partagées[1], peut générer des incohérences et provoquer une fuite de ressources partagée, provoquant des accès multiples, non-gérés et simultanées ou une saturation par débordement de cette ressource, par exemple, à cause d'une mauvaise gestion de la libération des allocations aux ressources (cf. *fork-bomb*).

Pour cela, le noyau et les jeux d'instructions processeur fournissent généralement des mécanismes de gestion d'allocations des ressources, afin de prévenir les fuites qui pourraient survenir, tels que les mécanismes d'exclusions mutuelles (ou « *mutex* »).

1– **Ressources partagées** : Désigne un élément logique ou physique accessible à un ensemble de séquence d'instruction (e.g. programme), par exemple, un espace mémoire, une interface de communication (*sockets*), un traitement de fichiers (*file handles*), un espace de nommage pour l'identification de processus (*PID namespace*).

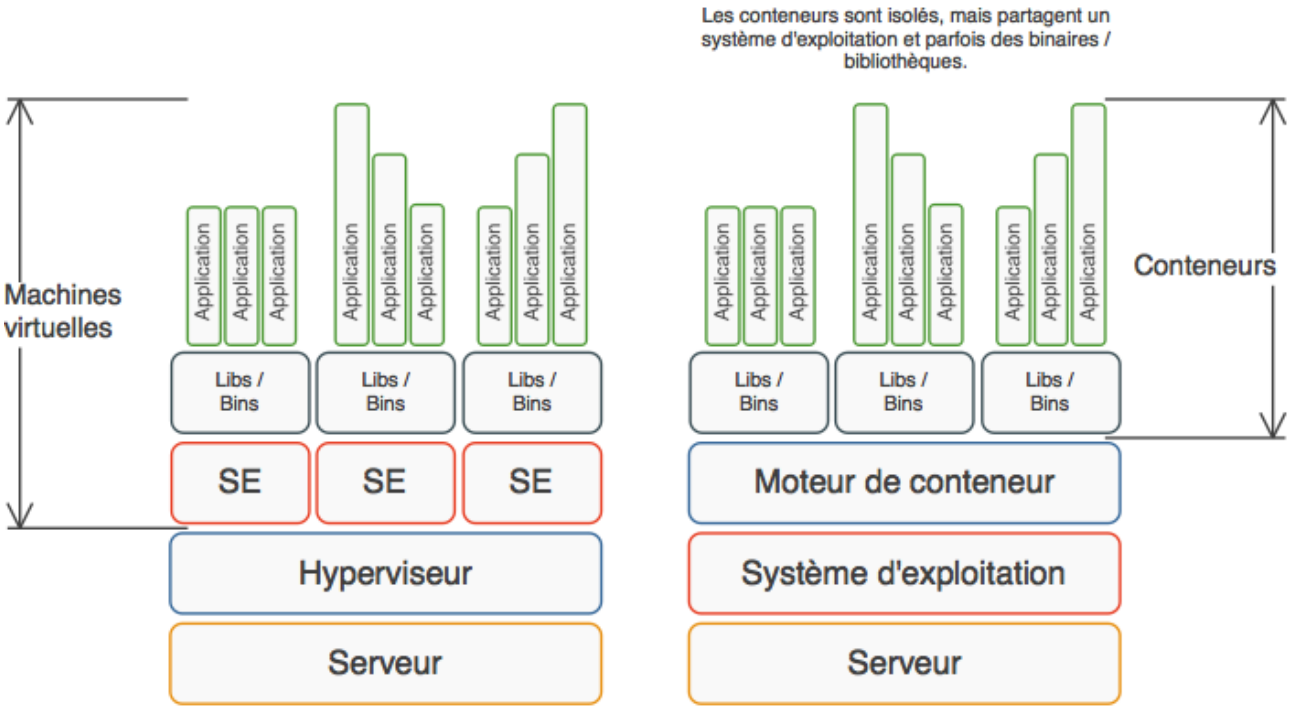


Schéma 2: Différence entre para-virtualisation et conteneurs[5]

Verrou (ou « lock ») itératif / récursif

Chacune des tâches (« *thread* »), exécuté dans un environnement concurrentiel, procède à l'acquisition de l'autorisation d'accès – unique à une ressource partagée – qu'il verrouillera avant son traitement et pour la durée de son cycle d'exécution. Une tâche concurrente à cette ressource sera alors dans l'incapacité d'allouer un accès à celle-ci tant que la tâche à l'origine du verrou ne l'aura pas expressément libéré, elle entre alors dans un cycle d'attente active (« *spinlock* »).

Exemple pratique :

Pensez au traitement de texte sous LibreOffice, un fichier « .lock » est créé lors de l'ouverture en écriture d'un fichier afin d'indiquer aux utilisateurs ou processus concurrents que le traitement en parallèle est incertain, il n'est alors possible pour les autres processus que de procéder à une lecture-seule de celui-ci, on parle alors d'un partage exclusif de verrou (cette possibilité variant dépendamment de la ressource concernée).

Problème : Un phénomène d'interblocage (ou « *deadlock* ») peut avoir lieu lors de ce cycle d'attente active, par exemple :

- Une tâche A est à l'origine d'un verrou sur une ressource R1 ;
- Une tâche B est à l'origine d'un verrou sur une ressource R2 ;
- La tâche A, pour avancer dans son exécution, entre en cycle d'attente active pour la ressource R2 ;
- La tâche B, pour avancer dans son exécution, entre en cycle d'attente active pour la ressource R1 ;
- Aucune des deux tâches ne procède jamais à la libération de la ressource allouée, car elle n'a pas fini son cycle d'exécution ;
- Une boucle d'interblocage est formée.

Variable de synchronisation (ou « semaphore »)

Permet le traitement protégé, d'une ressource partagée et allouée à un processus au cours d'un cycle non-interruptible, afin de garantir que les contrôles opérés sur cette ressource, ne peuvent être altérés pendant la durée du cycle d'exécution, on parle d'atomicité.

Exemple pratique :

Le cycle de circulation d'un TGV (Train à Grande Vitesse) entre deux points d'arrêts (i.e. gare) est atomique (non-interruptible). Les voies ferrées sont conçues de telle manière à optimiser la circulation en évitant les conditions de ralentissement (e.g. virage, circulation alternées), elles peuvent ainsi être amenées à traverser le réseau routier, désigné comme la ressource partagée du processus de circulation du TGV et de la traversée de la voie ferrée par des véhicules terrestres.

Le sémaphore impose, par témoin visuel, à chacun des véhicules usagers du réseau routier, une mise à l'arrêt dans l'attente de libération de la voie bitumée, effective uniquement après la traversée complète du TGV. Les usagers du réseau routier forment alors une file d'attente du type « premier entré, premier sorti » (« *First-In-First-Out (FIFO) Queue* »), et entrent alors dans un processus d'attente active afin d'assurer la sécurité de la circulation respective des véhicules. En cas d'accident, la circulation sera bloquée et provoquera le phénomène du bouchon qui sera très long à débloquer.

Problème :

À nouveau, un phénomène d'interblocage peut avoir lieu si l'un des processus de la file d'attente ne libère pas son cycle de traitement dans l'attente d'une ressource allouée à un autre processus de la file.

CGroups

Pour procéder à l'isolation des accès aux ressources partagées, *LXC* se base sur une fonctionnalité du noyau système Linux appelé CG*roups*[6][7][8][9] (pour « C*ontrol groups* »). Celui-ci permet de fournir des mécanismes de contrôles granulaires pour :

- Provisioning et la limitation des ressources : Gestion de l'allocation des ressources pour chaque groupe, par exemple, une quantité définie de mémoire vive (i.e. RAM) et donc du cache de système de fichier (« *buffer cache* ») dans un contexte fonctionnel d'écriture asynchrone sur le support de stockage de masse ;
- Priorisation : Un groupe se voit attribuer un ordre de priorité sur les autres afin de disposer d'un accès privilégié aux ressources ;
- Comptabilité : Facturation des ressources consommées. Utilisée principalement dans les infrastructures de *Cloud Computing*, notamment pour la gestion de serveurs privés virtuels ;
- Isolation : Séparation par la création d'un espace de nommage (« *namespace* ») distinct dans les groupes (voir Schéma 3), par exemple, un espace de nommage par identifiants de processus (« *Process Identifier* » ou « *PID* ») ;
- Contrôle, manipulation des groupes : Fonctions permettant, par exemple, le démarrage, arrêt, gel, clonage, sauvegarde et restauration de contexte.

LibContainer

LibContainer est une bibliothèque libre[10] développée par la société *Docker* et écrite en *GoLang*. Elle permet de fournir une interface standard pour la gestion et la création de conteneurs en permettant, sur un système Linux, l'interfaçage direct avec les *APIs* exploités jusqu'alors par *LXC* (voir Schéma 1), on parle d'une couche d'abstraction du système d'exploitation hôte.

Cette bibliothèque présente un intérêt non négligeable dans le déploiement continue puisqu'elle permettra l'exécution de Docker sur des systèmes d'exploitations ne se basant pas sur le noyau Linux (e.g. Windows[11], Mac, BSD, Solaris).

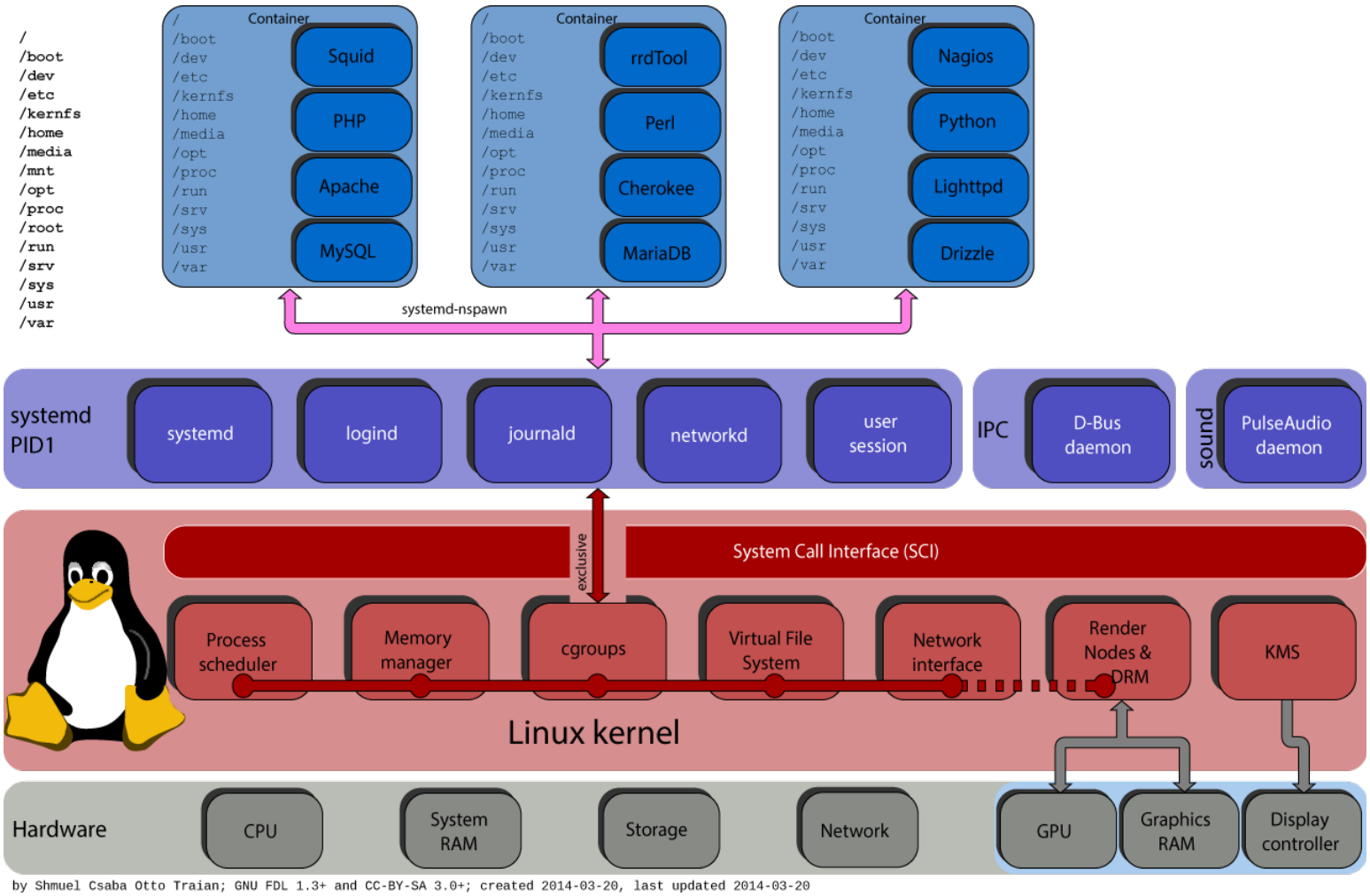


Schéma 3: Structure hiérarchique de CGroups[6]

COMPOSANTS DE DOCKER

Docker est composé essentiellement de deux composants :

- *Docker Engine* ;
- *Docker Hub*.

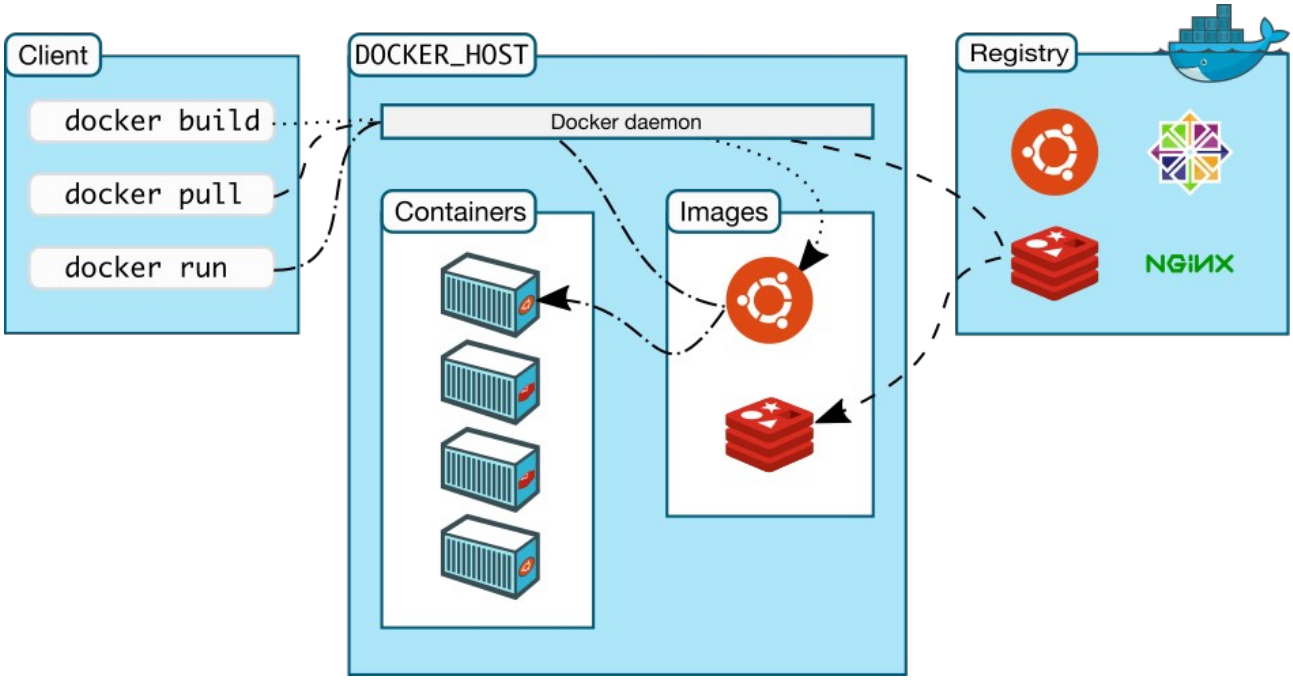


Schéma 4: Architecture de communication de Docker Engine[12]

Docker Engine

Docker Daemon

Le *daemon Docker*, en accord avec la convention de nommage du système Linux, est le service installé sur le système d'exploitation hôte du serveur (voir Schéma 4).

Docker Client

Le client *Docker*, présenté sous la forme de binaires exécutables, est l'interface de communication entre le *daemon* et l'utilisateur final. Celui-ci peut être installé sur un client physique distinct du serveur exécutant le *Docker Daemon*.

Docker Registry

Le registre *Docker* permet de télécharger, partager, indexer des images *Docker* en respectant les standards de publications des référentiels *Git. Docker* dispose d'un référentiel central public appelé *Docker Hub*[13].

INSTALLATION DE DOCKER

Debian Jessie

Tel qu'expliqué dans les précédentes parties, *Docker* est basé sur diverses fonctionnalités implémentées par le noyau Linux. Il est donc nécessaire, pour procéder à son installation, de posséder une distribution Linux basée sur un noyau 64 bits et de version 3.10 au minimum[14].

Pour vérifier la version du noyau (*kernel*), ouvrir un terminal et saisir :

```
# uname -r
4.4.0-amd64
```

Installation des outils de gestion des certificats et de gestion des dépôts utilisant HTTPS pour APT :

```
# apt-get install apt-transport-https ca-certificates
```

Ajout du dépôt à la base de sources de APT :

```
# cat <<-'EOF' >/etc/apt/sources.list.d/docker.list
# Dépôt de base de Debian Jessie
deb http://httpredir.debian.org/debian/ jessie main contrib
deb-src http://httpredir.debian.org/debian/ jessie main contrib

# Mises à jour distribution stable
deb http://httpredir.debian.org/debian/ jessie-updates main
deb-src http://httpredir.debian.org/debian/ jessie-updates main

# Mises à jour vers distribution stable
deb http://httpredir.debian.org/debian/ jessie-backports main
deb-src http://httpredir.debian.org/debian/ jessie-backports main

# Mises à jour de sécurité
deb http://security.debian.org/ jessie/updates main
deb-src http://security.debian.org/ jessie/updates main

# Docker Project
deb https://apt.dockerproject.org/repo debian-jessie main
EOF
```

Mise à jour des caches, mise à jour du système et des paquets :

```
# apt-get clean
# rm -rf /var/lib/apt/lists/*
# apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys F76221572C52609D
# apt-get update
# apt-get -fy upgrade
# apt-get -fy dist-upgrade
```

Installation de *Docker Engine* en utilisant le dépôt de *Docker* :

```
# apt-get install cgroupfs-mount
# apt-get install --target-release debian-jessie docker-engine
```

Ces commandes vont alors permettre l'installation des paquets :

- cgroupfs-mount : voir CGroups ;
- docker-engine : voir Docker Engine.

Démarrage et vérification de l'exécution de *Docker Daemon* en utilisant le client docker :

```
# systemctl start docker.service
# systemctl status docker.service
# docker info
```

Remarque : L'utilisation de *Docker* n'est possible que dans un contexte d'exécution *root*. En effet, *Docker Daemon* est implémenté en écoute sur des *UNIX socket*, dont la gestion dépend de l'utilisateur *root*, ainsi, lui-seul est dans la capacité d'agir, par défaut, sur le fonctionnement de ce dernier. Ainsi, pour utiliser un utilisateur standard du système pour la gestion des conteneurs, il est possible d'ajouter un utilisateur au groupe « *docker* » :

```
# useradd --system --uid 1337 -user-group \
--create-home --home-dir /home/Creased --skel /etc/skel \
--shell /bin/bash --comment "Utilisateur standard du système" Creased
# usermod -aG docker Creased
# getent passwd Creased
Creased:x:1337:996:Utilisateur standard du système:/home/Creased:/bin/bash
# getent group Creased docker
Creased:x:996:
docker:x:997:Creased
```

Windows

Docker ToolBox

Actuellement, il est possible d'installer *Docker* sous Windows en utilisant la *Docker ToolBox*[15]. Cette boîte à outils est entre-autre composée :

- *Docker Client ;*
- *Kitematic*[16] : Outil graphique écrit en *Electron*[17] permettant la gestion graphique et la création rapide de conteneurs *Docker* depuis un *Docker Registry*[13] ;
- *Docker Compose*[18] : Outil permettant la création et l'exécution rapide d'application basée sur plusieurs conteneurs liés en se basant sur un fichier de recette (« *recipe* ») ;
- *Docker Machine*[19] : Permet l'installation du *Docker Daemon* (via *boot2docker*), par exemple sur Windows et Mac, en utilisant un *driver* d'intégration, par exemple, *Oracle VirtualBox*[20] ;

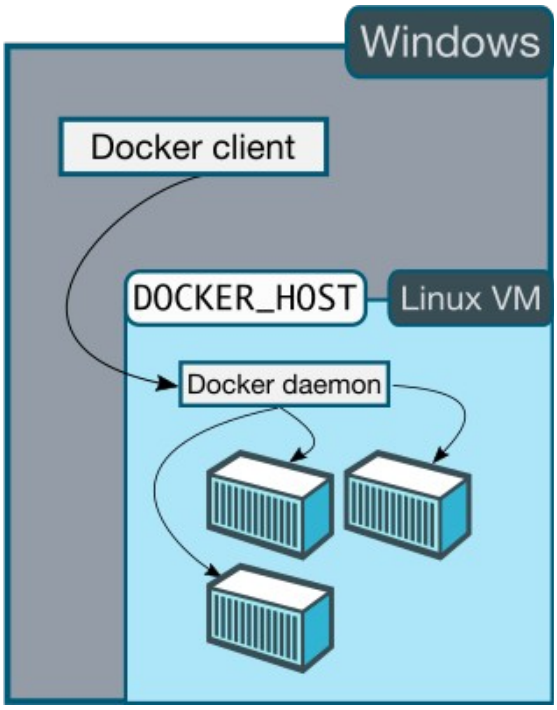


Schéma 5: Architecture de communication de Docker sous Windows[12]

L'objectif premier de la *Docker ToolBox* étant de simplifier l'installation de *Docker* en la rendant transparente, il serait contre-productif de complexifier sa procédure d'installation en la décrivant.

Toutefois, il sera nécessaire de prendre en compte certains paramètres lors de son installation, par exemple, si vous possédez un *toolkit* pour disposer des outils standards de Linux tel que *Gow*, celui-ci pourra éventuellement devenir source de conflit.

DOCKER MACHINE

À titre indicatif, voici une capture d'écran indiquant les résultats de chacune des commandes traitées dans cette partie : http://imgur.com/Wtd4aeW.

Suite à l'installation de la *Docker ToolBox*, pour vérifier que *Docker Machine* est utilisable, ouvrir un interprète de commande Windows et saisir :

```
$ docker-machine version

docker-machine.exe version 0.6.0, build e27fb87
```

Création d'une machine en utilisant *Oracle Virtualbox* :

```
$ docker-machine create --driver virtualbox --virtualbox-cpu-count "2"
--virtualbox -memory "2048" default
```

Liste des machines :

```
$ docker-machine ls
```

Affichage des informations sur la machine créée (formaté en *JSON*) :

```
$ docker-machine inspect default
```

Activation contextuelle de la machine pour l'utilisation de *Docker Client* :

```
$ eval "`docker-machine env --shell bash default`"
```

Affichage de la machine active :

```
$ docker-machine active
```

Accès en *Secure SHell* (*SSH*) à la machine *Docker* :

```
$ docker-machine ssh
```

DOCKER IMAGE

À titre indicatif, voici une capture d'écran indiquant les résultats de chacune des commandes traitées dans cette partie : https://imgur.com/hTPFQcf.

Une image Docker est un modèle (*template*) en lecture-seule. Il peut contenir, par exemple, un système de fichier dédié à un système d'exploitation/application. Ces images sont utilisées pour créer des conteneurs Docker.

En effet, les conteneurs sont basés sur une pile d'image, lorsque l'utilisateur démarre un conteneur depuis une image, une nouvelle couche avec les droits d'écritures est créée en utilisant le mécanisme d'union de système de fichier offert par *AUFS,* à chaque modification correspondra une nouvelle image (voir Schéma 6).

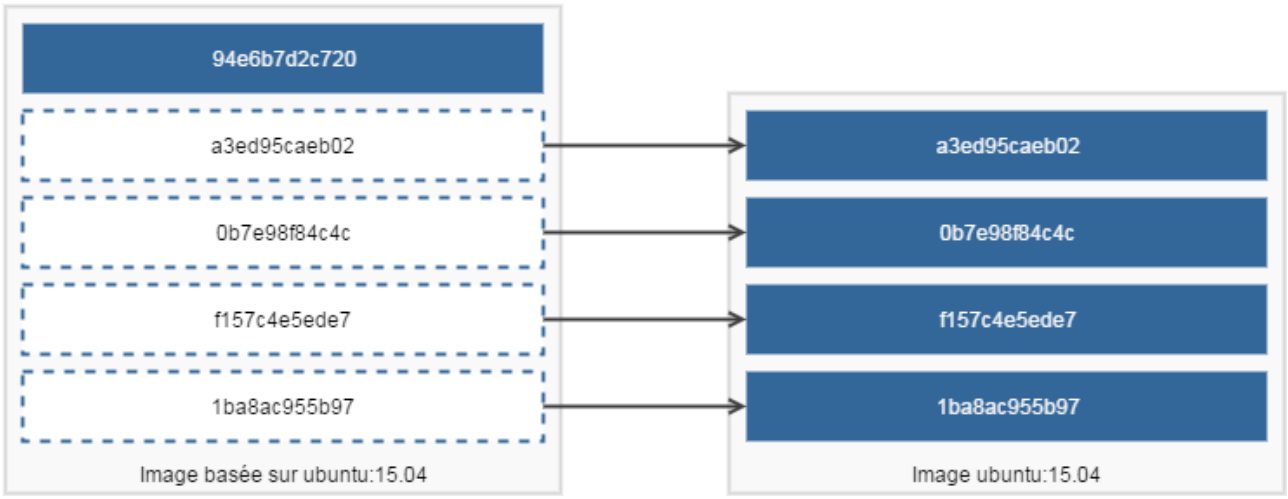


Schéma 6: Exemple d'image basée sur Ubuntu 15.04

Recherche d'une image sur le *Docker Hub* :

```
$ docker search debian
```

Téléchargement de l'image depuis le *Docker Hub* :

```
$ docker pull debian:latest
```

Note : Le *tag* (ici « latest ») correspond généralement à la version de l'image, il désigne, dans ce cas-ci, la dernière mise à jour de l'image et évolue dynamiquement à chaque modification.

Liste des images *Docker* téléchargées :

```
$ docker images -a
```

Sauvegarde de l’image :

```
$ docker save --output debian.tar debian
```

Suppression de l’image :

```
$ docker rmi debian
```

Restauration de l’image :

```
$ docker load --input debian.tar
```

DOCKER CONTAINER

À titre indicatif, voici une capture d'écran indiquant les résultats de chacune des commandes traitées dans cette partie : http://imgur.com/Hwvpt7e.

Les images sont aux conteneurs ce qu'un disque d'installation est à un système d'exploitation, ils consistent à définir une base statique sur laquelle le conteneur va fonctionner (e.g. système de fichier, fichiers de configurations).

Création d'un conteneur basé sur une image de *Debian* en lui allouant un terminal interactif :

```
$ docker create --interactive --tty --hostname debian --name debian
debian:latest
```

Note : L'instruction « *hostname* » indique le nom d'hôte interne au conteneur, là où l'instruction « *name* » indique le nom unique du conteneur.

Liste des conteneurs avec leur identifiant, image de base, nom, statut :

```
$ docker ps --all --format "table {{ .ID }}\t{{ .Image }}\t{{ .Names }}\t{{
.Status }}"
```

Démarrage du conteneur créé :

```
$ docker start debian
```

Attache au contexte d'exécution du conteneur, il sera possible de s'en détacher en pressant la combinaison de touche indiquée dans l'argument « *detach-keys* » :

```
$ docker attach --detach-keys="ctrl-w" debian
```

Dans ce conteneur, il est possible de saisir des commandes :

```
# printf '\033[4;37;42mHello, World!\033[0m\n'
# ping 127.0.0.1 -c 20
```

À présent si l'on se détache de son contexte d'exécution, il est possible d'afficher le flux de sortie standard généré pendant son cycle d'exécution :

```
$ docker logs --follow
```

Note : Le fonctionnement de docker logs est semblable à celui de *tail*.

Arrêt du conteneur :

```
$ docker stop debian
```

Suppression du conteneur :

```
$ docker rm debian
```

CAS PRATIQUE : *NGINX*

À titre indicatif, voici une capture d'écran indiquant les résultats de chacune des commandes traitées dans cette partie : http://imgur.com/tCDisLH.

Création du conteneur

Création d'un conteneur sous *Debian* :

```
$ docker create --publish-all --interactive --tty --hostname debian-nginx
--name debian-nginx debian:latest
```

Démarrage du conteneur et attache à son contexte d'exécution :

```
$ docker start debian-nginx
$ docker attach --detach-keys="ctrl-w" debian-nginx
```

Installation de *Nginx*

Ajout des sources pour le gestionnaire de paquets *APT* :

```
# apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys ABF5BD827BD9BF62
# cat <<-'EOF' >/etc/apt/sources.list
deb http://httpredir.debian.org/debian jessie main
deb http://httpredir.debian.org/debian jessie-updates main
deb http://security.debian.org jessie/updates main
deb http://nginx.org/packages/mainline/debian/ jessie nginx
EOF
```

Mise à jour des caches :

```
# apt-get update
```

Installation de *Nginx* et *PHP-FPM* sans les paquets suggérés :

```
# apt-get install -y --no-install-recommends --no-install-suggests ca-
certificates nginx="1.9.0-1~jessie" spawn-fcgi gettext-base php5-fpm
```

Suppression des caches du gestionnaire de paquets *APT* :

```
# apt-get clean
# rm -rf /var/lib/apt/lists/*
```

Démarrage du service *Nginx* en premier plan et détache du contexte d'exécution (CTRL+W) :

```
# nginx -g "daemon off;"
```

Création d'une image

Création d'une image en se basant sur l'état actuel du conteneur :

```
$ docker commit --message="initial commit" --author="Baptiste MOINE
<bap.moine.86@gmail.com>" --pause=true --change="EXPOSE 80" --change="CMD
[\"nginx\", \"-g\", \"daemon off;\"]" a1a468c6241b debian-nginx:1.0
```

Note : Le conteneur étant basé sur *Debian*, il n'était pas nécessaire, lors de sa création, de créer un mappage de port sur ce conteneur. Ainsi, pour pouvoir exploiter le service *HTTP* de *Nginx*, on indique que l'image, lorsqu'elle sera utilisée comme base pour un nouveau conteneur, exposera le port 80. De plus, on indique que la commande exécutée lors du démarrage d'un conteneur basé sur cette image sera « nginx -g daemon off ».

Sauvegarde de l'image :

```
$ docker save --output debian-nginx.tar debian-nginx
```

Suppression du conteneur :

```
$ docker stop debian-nginx
$ docker rm debian-nginx
```

Création d'un nouveau conteneur basé sur cette nouvelle image :

```
$ docker create --publish-all --interactive --tty --hostname debian-nginx
--name debian-nginx debian-nginx:1.0
$ docker start debian-nginx
```

Affichage des paramètres du conteneur pour déterminer les ports de communication et l'adresse *IP* utilisée :

```
$ docker ps --all --format "table {{ .ID }}\t{{ .Image }}\t{{ .Ports }}"
--filter "status=running" --filter "ancestor=debian-nginx:1.0"
$ docker-machine inspect --format='{{ .Driver.IPAddress }}' default
```

Test du service HTTP (à adapter en fonction des résultats précédents) :

```
$ curl --head 192.168.99.100:32768
```

DOCKERFILE

Notre image contenant *Nginx* est créée, cependant, en utilisant une telle méthode pour sa création, on écarte une possibilité non négligeable offerte par *Docker*, l'intégration continue. En effet en se basant sur un fichier de *provisioning*, *Docker* est ainsi capable de créer lui-même une image en se basant sur des instructions minimalistes.

Cas appliqué à Nginx

Création d'une structure de fichier simple pour contenir l'ensemble des données utilisées pour la création de notre conteneur :

```
$ mkdir -p debian-nginx/files/{apt,fpm,nginx}
$ mkdir -p debian-nginx/files/fpm/conf/pool.d/
$ mkdir -p debian-nginx/files/nginx/{conf,webroot}
$ mkdir -p debian-nginx/files/nginx/conf/conf.d/
$ mkdir -p debian-nginx/files/nginx/webroot/{static,webroot}
```

Création des fichiers de configurations de *Nginx*, *APT*, *PHP-FPM* et du script de démarrage (voir le référentiel Git).

Création d'une base de fichiers pour le service HTTP (voir référentiel Git).

Création d'un *Dockerfile* (voir référentiel Git). Pour des raisons de stabilité, il est déconseillé d'exploiter une image de base en utilisant le tag « *later* ».

Création de l'image à partir du *Dockerfile* :

```
$ docker build --tag creased/debian-nginx:1.0 .
```

Création d'un conteneur à partir de cette nouvelle image :

```
$ docker create --publish 80:80/tcp --publish 443:443/tcp --interactive --tty
--hostname debian-nginx --name debian-nginx creased/debian-nginx:1.0
$ docker start debian-nginx
$ docker attach --detach-keys="ctrl-w" debian-nginx
```

DOCKER HUB

À présent que nous avons crée une structure de fichier permettant la création automatique d'une image, nous pouvons la soumettre à la communauté. Cette structure de fichier, si elle est stockée sous la forme d'un référentiel *Git* (*Github*) ou *Mercurial* (*BitBucket*), est utilisable par le *Docker Hub* pour créer une nouvelle image à chaque *commit*. En effet, en utilisant un système de crochets (« *hooks* » ou « *trigger* »), *Docker Hub* va procéder à la création d'une nouvelle image à chaque fois que le dépôt lui signalera une mise à jour de sa base de fichiers, on parle alors d'intégration continue (*CI*).

Créer un nouveau référentiel sur *Github* en renseignant une description et un titre explicite (voir Illustration 1).

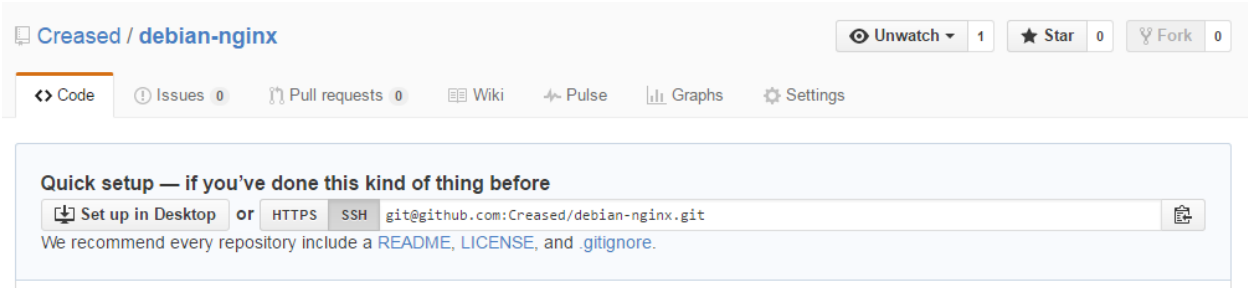


Illustration 1: Nouveau référentiel Git

Depuis le *Docker Hub*, procéder à une liaison entre le compte *Github* et *Docker Hub* (voir Illustration 2).

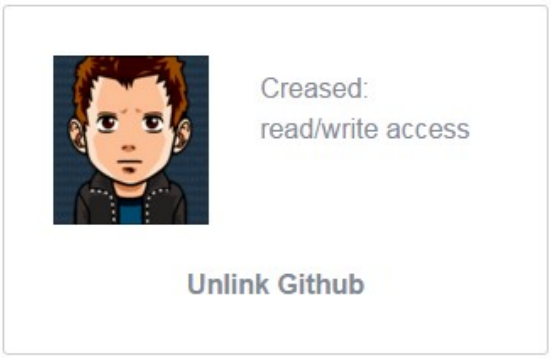


Illustration 2: Liaison Docker Hub – Github[21]

Envoyer les données sur *Github* :

```
$ cd debian-nginx
$ git init
$ git remote add origin git@github.com:Creased/debian-nginx.git
$ git add .
$ git commit --message "Initial commit"
$ git push --set-upstream origin master
```

Depuis le Docker Hub, créer une image « *Automated Build* » (voir Illustration 3)

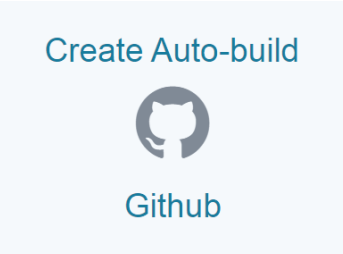


Illustration 3: Créer une image automatiquement à partir du référentiel Git

Spécifier à nouveau une courte description pertinente et explicite sur le contenu de votre image puis valider la création. Le processus de fabrication de l'image est alors lancé et ne prendra que très peu de temps, il ne reste qu'à actualiser pour admirer le résultat (voir Illustration 4).

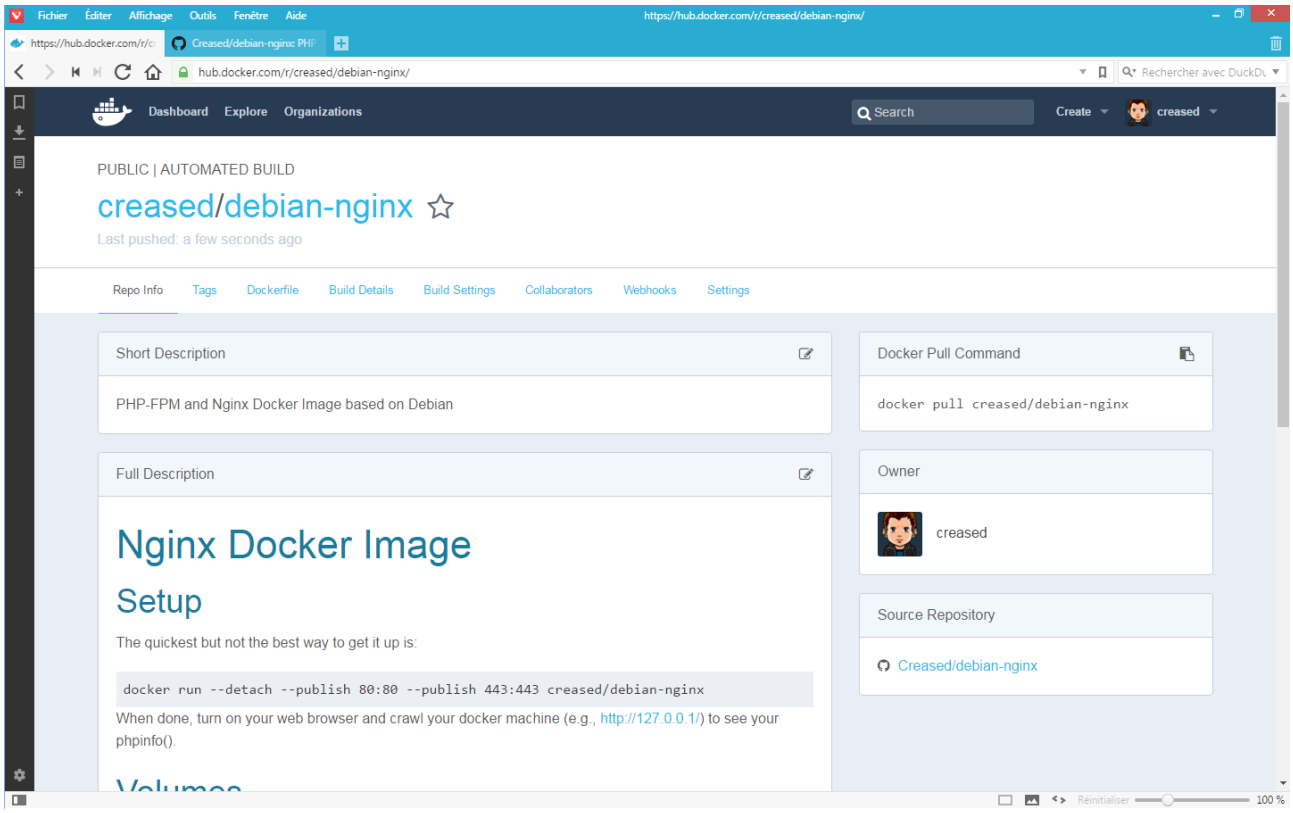


Illustration 4: Image sur le Docker Hub générée à partir d'un référentiel Git

Note : Une description détaillée est affichée suite à la création de l'image, celle-ci correspond au fichier « README.md » du dépôt. Cependant, le *Docker Hub* ne supporte pas toute la syntaxe utilisée par *Github* dans le traitement de ses fichiers MarkDown, il est ainsi nécessaire d'utiliser la syntaxe classique de *MarkDown* dans le fichier *README* au risque de générer des troubles visuels.

Suite à chaque *commit* sur le référentiel *Git*, il sera possible d'observer un nouveau processus de génération de l'image sur le *Docker Hub* (voir Illustration 5).

Status	Tag	Created	Last Updated
✔ Success	latest	2 minutes ago	a few seconds ago
✔ Success	latest	4 minutes ago	2 minutes ago
✔ Success	latest	7 minutes ago	5 minutes ago
! Error	latest	11 minutes ago	11 minutes ago

Illustration 5: Auto-Build à chaque commit sur la branche master

Note : Le premier processus a échoué suite à la création d'un fichier « *DockerFile* », la casse étant très importante dans ce processus, le fichier a dû être renommé « *Dockerfile* ».

Création d'un conteneur depuis cette image (voir Illustration 6) :

```
$ docker pull creased/debian-nginx:latest
$ docker create --publish 80:80/tcp --publish 443:443/tcp --interactive --tty
--hostname debian-nginx --name debian-nginx creased/debian-nginx:latest
$ docker start debian-nginx
$ docker attach --detach-keys="ctrl-w" debian-nginx
```

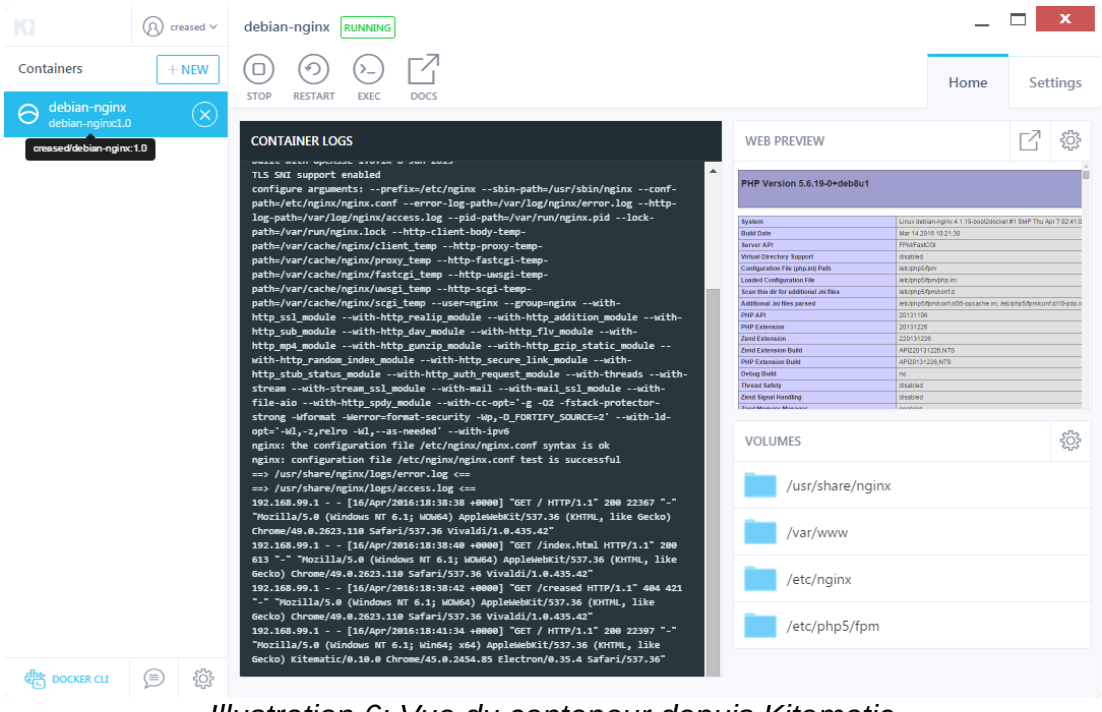


Illustration 6: Vue du conteneur depuis Kitematic

DOCKER VOLUME

À titre indicatif, voici une capture d'écran indiquant les résultats de chacune des commandes traitées dans cette partie : http://imgur.com/3Vzt1fS.

Un *Docker Volume*[22] permet de désigner un répertoire contenu dans un conteneur, dans lequel les données seront persistantes. En d'autres termes, il s'agit d'un point de montage établissant un lien direct entre un répertoire sur l'hôte et un répertoire sur le conteneur (voir Schéma 7). Il est à noter également, qu'il est ainsi possible de partager ce volume entre différents conteneurs.

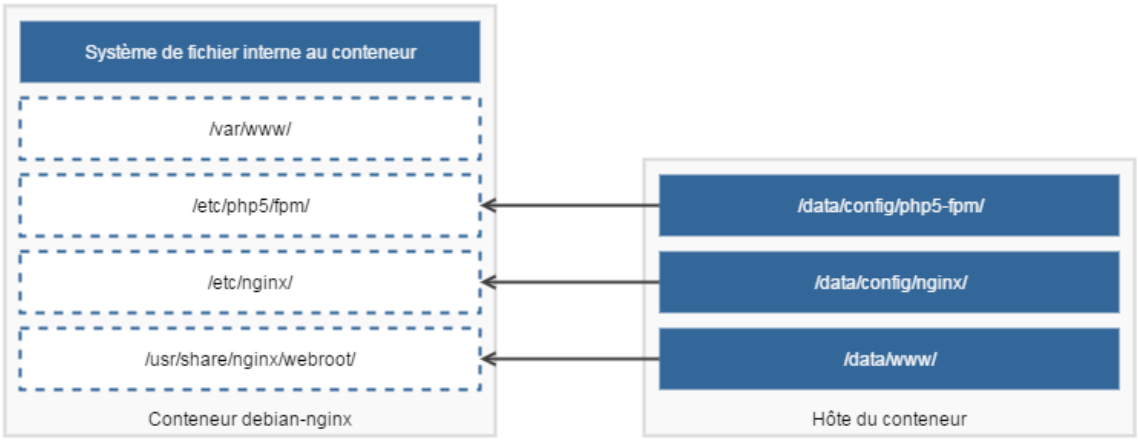


Schéma 7: Volume Docker de creased/debian-nginx

Création d'un nouveau répertoire sur l'hôte et ajout de données :

```
$ mkdir -p /data/www/
$ cat <<-'EOF' >./index.php
<?php echo("Hello, World!"); ?>
EOF
```

Arrêt du conteneur :

```
$ docker stop debian-nginx
```

Suppression du conteneur :

```
$ docker rm debian-nginx
```

Liste des volumes associés à l'image :

```
$ docker inspect -f '{{ .Config.Volumes }}' creased/debian-nginx
```

Création d'un nouveau conteneur :

```
$ docker create --publish 80:80/tcp --publish 443:443/tcp --interactive --tty
--hostname debian-nginx --name debian-nginx --volume
/`pwd`:/usr/share/nginx/webroot/ creased/debian-nginx:latest

$ docker start debian-nginx
```

Note : L'utilisation des volumes semble actuellement instable sous *Windows*[23][24], l'utilisation d'un interprète de commande basé sur *Mingw* (e.g. *Cygwin*) semble empêcher l'utilisation de la syntaxe standard pour le mappage des volumes, aussi, il semble nécessaire de préfixer le chemin d'accès au répertoire de l'hôte d'un *slash* afin d'éviter toute interprétation erronée par l'interprète de commande.

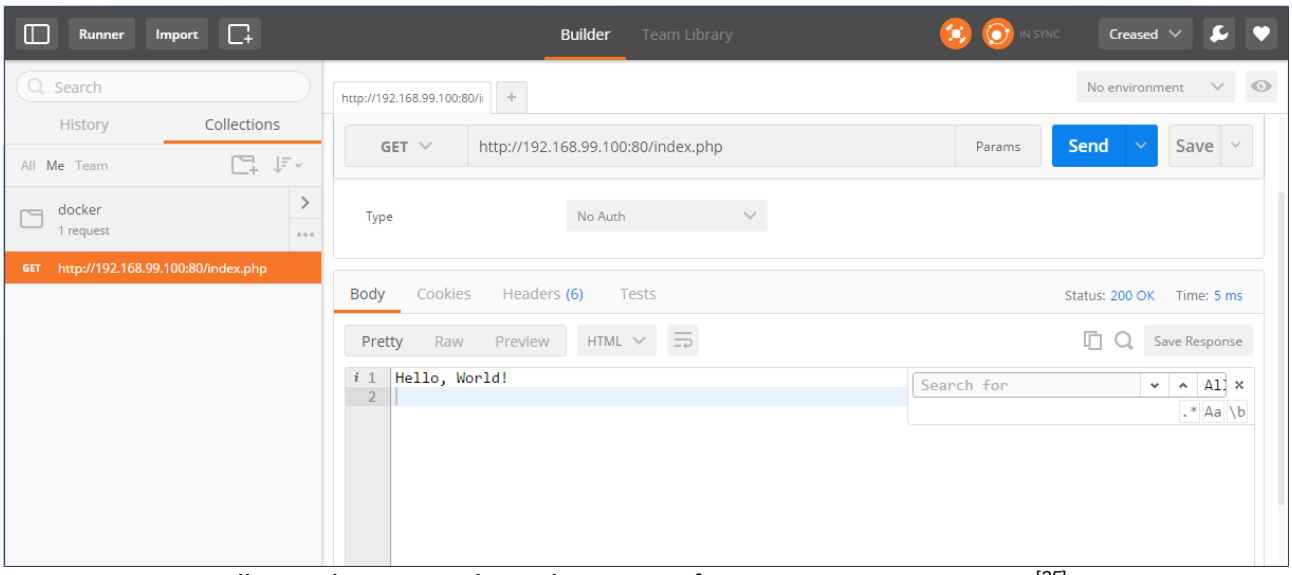


Illustration 7: Analyse d'une requête HTTP avec Postman[25]

Une alternative plus viable est celle de créer un point de montage de notre répertoire, directement dans la *Docker Machine* en utilisant l'API de *VirtualBox*, en lui donnant un nom explicite tel que « *host* ».

Suite à l'activation d'un Dossier partagé dans *VirtualBox*, il sera nécessaire, afin de préserver les modifications, d'écrire un script de démarrage sur le volume non-volatile de notre *Docker Machine*. En effet, l'implémentation de *Docker* sous *VirtualBox* se base sur un « *Live-CD* » de *Boot2Docker*, ainsi seules une partie du système de fichier possède une persistance, à savoir « */mnt/sda1/var/lib/docker* » et « */mnt/sda1/var/lib/boot2docker* ».

Création du script de démarrage pour la *Docker Machine :*

```
$ docker-machine ssh default
£ cd /mnt/sda1/var/lib/boot2docker/
£ cat <<-'EOF' >./bootlocal.sh
#!/bin/sh

mkdir /var/host/
mount.vboxsf -o rw,fmode=777 host /var/host/

EOF
£ chmod u+x ./bootlocal.sh
£ exit
```

Note : Le script de démarrage doit respecter le nom « *bootlocal.sh* », sans quoi, il ne sera pas démarré.

Il ne reste plus qu'à redémarrer la Docker Machine et vérifier le résultat :

```
$ docker-machine stop default
$ docker-machine start default
```

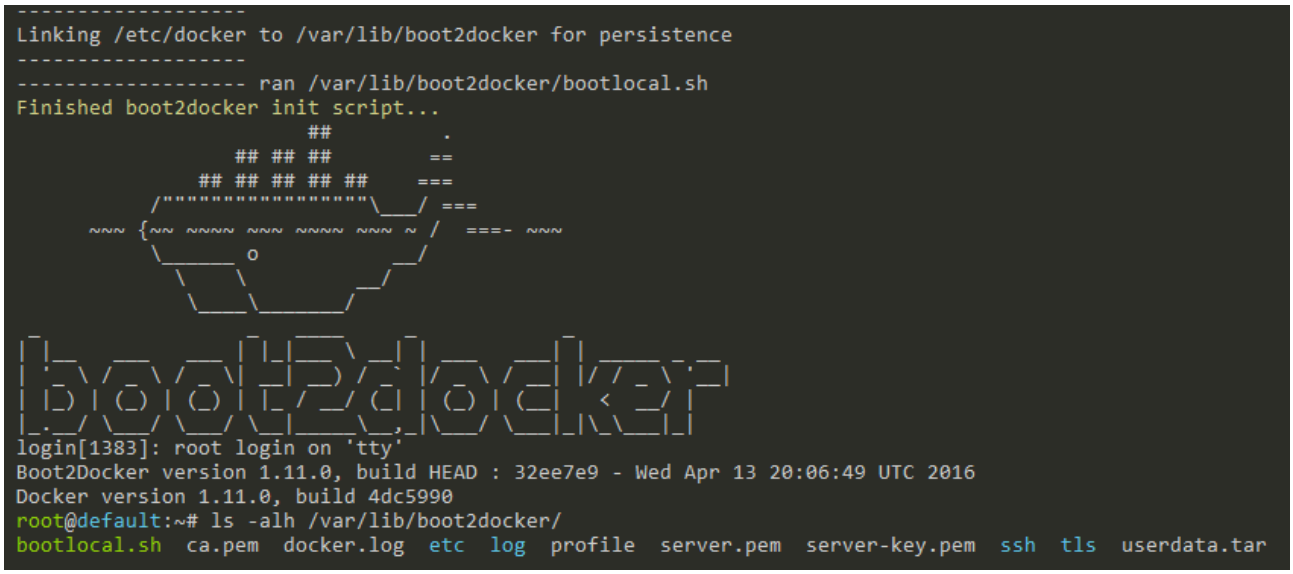


Illustration 8: Démarrage de Boot2Docker et exécution d'un script de démarrage

Création d'un nouveau conteneur en utilisant le nouveau point de montage :

```
$ docker create --publish 80:80/tcp --publish 443:443/tcp --interactive --tty
--hostname debian-nginx --name debian-nginx --volume
//var/host/www/:/usr/share/nginx/webroot/ creased/debian-nginx:latest
```

DOCKER COMPOSE

Docker Compose permet la création et l'exécution rapide d'application basée sur un ou plusieurs conteneurs liés (« *links* »). Pour cela, *Docker Compose* se base sur une recette de composition (« *recipe* »), décrite dans une syntaxe simple (*YAML*), et procède à l'installation, fabrication (voir DockerFile), liaison et démarrage de conteneurs.

Précédemment, nous avons procédé à la création d'une image contenant un environnement minimaliste composé de *Nginx* et *PHP-FPM*. Cependant, celui-ci n'est pas aisément modulable, en effet, si l'on envisage l'utilisation d'un environnement de développement pouvant s'adapter aux besoins directs du projet (e.g. utilisation de *MySQL*, *Redis*) ou tout simplement pour supporter les différentes mises à jour des composants de l'image, il sera certainement nécessaire de réécrire complètement notre fichier *Dockerfile* ce qui tend à rendre l'utilisation de cette technologie relativement complexe et laborieuse.

En utilisant plusieurs conteneurs pour concevoir notre environnement de développement, ces problèmes sont minimisés.

Préparation du système

Pour commencer, afin de repartir sur une base saine, nous allons supprimer l'ensemble des conteneurs présents sur notre machine :

```
$ docker stop $(docker ps --all -quiet)
$ docker rm $(docker ps --all --quiet)
```

Ensuite, nous allons ré-employer le répertoire précédemment lié à notre machine virtuelle via l'*API* de *VirtualBox* afin d'y stocker l'ensemble des données auxquelles nous souhaiterions avoir accès depuis nos conteneurs (e.g. configurations, journaux) :

```
$ rm -rf /data/*
$ git clone https://github.com/Creased/docker-compose-nginx-fpm-mariadb.git
/data
$ cd /data/
$ git fetch origin 'refs/tags/*:refs/tags/*'
$ git checkout tags/template
$ less README.md
```

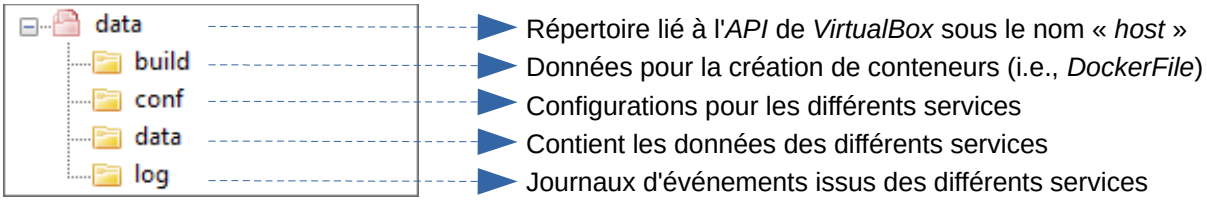


Illustration 9: Structure de fichiers pour Docker Compose

Installation de Docker Compose

Pour des raisons de compatibilité, il est actuellement nécessaire de procéder à l'installation de *Docker Compose* directement sur la *Docker Machine*. Cependant, comme expliqué précédemment toute modifications apportées au système sont non-persistantes. Ainsi, *Docker* fournis un binaire pré-compilé de *Docker Compose* qu'il suffira de copier à chaque démarrage de la *Docker Machine*.

Création d'un nouveau répertoire pour contenir les données de la *Docker Machine* :

```
$ mkdir /data/data/docker/
```

Modification du script de démarrage pour la *Docker Machine* par réécriture pour la copie automatique de *Docker Compose* à chaque démarrage *:*

```
$ docker-machine ssh default
£ curl -L https://github.com/docker/compose/releases/download/1.7.0/docker-compose-`uname -s`-`uname -m` > /var/host/data/docker/docker-compose
£ cd /mnt/sda1/var/lib/boot2docker/
£ cat <<-'EOF' >./bootlocal.sh
#!/bin/sh


mkdir /var/host/
mount.vboxsf -o rw,fmode=777 host /var/host/
mount.vboxsf -o ro,fmode=700 conf /var/host/conf
cp /var/host/data/docker/docker-compose /usr/local/bin/docker-compose
chmod 755 /usr/local/bin/docker-compose


EOF
£ exit
```

Redémarrage de la *Docker Machine* :

```
$ docker-machine stop default
$ docker-machine start default
```

Vérification de l'installation de *Docker Compose* :

```
$ docker-machine ssh default
$ docker-compose -v
docker-compose version 1.7.0, build 0d7bf73
```

Création d'un environnement de développement WEB

Ajout de *Composer*[26] à notre *Docker Machine* :

```
$ docker-machine ssh default

£ docker pull composer/composer:latest

£ cd /mnt/sda1/var/lib/boot2docker/

£ cat <<-'EOF' >>/var/host/data/docker/composer

#!/bin/sh
export
PATH=/usr/local/sbin:/usr/local/bin:/apps/bin:/usr/sbin:/usr/bin:/sbin:/bin
docker run --rm --tty --interactive --volume $(pwd):/app/ --volume
~/.ssh/:/root/.ssh/ composer/composer $@

EOF

£ cat <<-'EOF' >>./bootlocal.sh

cp /var/host/data/docker/composer /usr/local/bin/composer

chmod 755 /usr/local/bin/composer

EOF

£ composer --version

Composer version 1.0.0 2016-04-05 13:27:25

£ exit
```

Pour le reste, sans trop entrer dans les détails, il suffit de compléter le fichier de composition en remplissant tous les champs nécessaires pour la création d'un environnement de développement aussi complet que possible, composé par exemple de :

- PHP-FPM ;
- Nginx ;
- MariaDB.

Analyse du modèle de composition[18] :

```
$ less docker-compose.yml
#
# Written by:
#   Baptiste MOINE <bap.moine.86@gmail.com>
#


version: '2.0'

networks: {}

services: {}

volumes: {}
```

Pour observer directement un résultat possible (pas nécessairement le meilleur !), saisir :

```
$ git checkout tags/v1.0.0
$ less docker-compose.yml
```

Note : L'instruction « *build* » permet ici d'indiquer que l'on souhaite utiliser une image, mais qu'il faudra la construire en utilisant sa recette de composition (*Dockerfile*).

Fabrication de l'environnement de développement

Une fois la recette de composition renseignée, il peut s'avérer nécessaire de vérifier cette recette :

```
$ docker-machine ssh
£ cd /var/host/
£ docker-compose config
```

Si la recette est validée, la prochaine étape consiste à récupérer et fabriquer les images utilisées par la composition :

```
£ docker-compose pull
£ docker-compose build
```

Lancement de l'environnement de développement

Une fois préparés, les images vont pouvoir être exploitées afin de composer notre environnement de développement :

```
£ docker-compose create
```

À présent il ne reste qu'à démarrer la composition en mode détaché :

```
£ docker-compose start
£ docker-compose up -d
```

Pour suivre journaux d'événements, saisir :

```
£ docker-compose logs --follow
```

Pour lancer un terminal interactif dans un des conteneurs, saisir :

```
£ docker-compose exec db bash
```

CONCLUSION

Comme vous avez sans doute pu le constater, il est très simple de prendre en main cet outil, d'autant plus que la société à l'origine de cette solution est très active dans le développement de sa documentation et de sa solution.

Si vous rencontrez des difficultés, constatez une erreur, estimez une partie trop complexe et qui nécessiterait de plus amples explications, n'hésitez pas à me contacter afin de me décrire précisément votre problème, je ferais de mon possible pour vous aider.

Pour me contacter :

- Adresse e-mail : Baptiste MOINE <bap.moine.86@gmail.com> | PGP : 0x79FD0883
- Twitter : @Creased_

SOURCES

1. Docker – Components Licences : https://www.docker.com/components-licenses
2. GoLang : https://golang.org
3. Docker Blog – Introducing execution drivers and libcontainer : https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/
4. What's LXC ? : https://linuxcontainers.org/lxc/introduction/
5. Docker – What's Docker? : https://www.docker.com/what-docker
6. Wikipedia – Cgroups : https://en.wikipedia.org/wiki/Cgroups
7. RedHat – Resource Management Guide : https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Resource_Management_Guide/index.html
8. RDO – Hands on Linux sandbox with namespaces and cgroups : https://blogs.rdoproject.org/7761/hands-on-linux-sandbox-with-namespaces-and-cgroups
9. Libvirt Virtualization API – Control Groups Resource Management : https://libvirt.org/cgroups.html
10. GitHub – LibContainer Official Project : https://github.com/docker/libcontainer
11. GitHub – LibContainer Microsoft : https://github.com/Microsoft/libcontainer
12. Docker – Docker Engine : https://docs.docker.com/engine/understanding-docker/
13. Docker – Docker Hub Registry : https://registry.hub.docker.com/
14. Docker – Docker Installation : https://docs.docker.com/engine/installation/linux/debian
15. Docker – Docker Toolbox : https://blog.docker.com/2015/08/docker-toolbox/
16. Kitematic : https://kitematic.com/
17. Electron : https://github.com/electron/electron
18. Docker – Docker Compose : https://docs.docker.com/compose/overview/
19. Docker – Docker Machine : https://docs.docker.com/machine/overview/
20. Docker – Docker Machine Drivers : https://docs.docker.com/machine/drivers/
21. Docker – Docker Hub Linked Accounts : https://hub.docker.com/account/authorized-services/
22. Docker – Docker Volume : https://docs.docker.com/engine/userguide/containers/dockervolumes/
23. Github – Issue #12590 : https://github.com/docker/docker/issues/12590
24. Github – Issue #12751 : https://github.com/docker/docker/issues/12751

25. Postman : http://www.getpostman.com/apps
26. Docker Hub – Composer : https://hub.docker.com/r/composer/composer/