Docker - Build, Ship, Run

Présentation

L'objectif de ce cours est de présenter et d'utiliser Docker afin d'automatiser le déploiement d'applications dans des conteneurs logiciels. De plus en plus populaire, cet outil permet de faire de la virtualisation dite "légère" et donc d'améliorer les performances de ses applications. Dans un premier temps, nous allons découvrir les particularités de Docker et sa prise en main. Dans un second temps, nous allons mettre en pratique

Sommaire

Historique de la virtualisation

Les premiers travaux de virtualisation (ie faire fonctionner un ou plusieurs systèmes d'exploitation/applications comme un simple logiciel, sur un ou plusieurs ordinateurs/serveurs) ont été développé dans les années 1970 par IBM à Cambridge et le MIT.

Dans les années 1980-90, des embryons de virtualisation ont été créés sur des ordinateurs personnels (les précédents travaux n'étaient destinés que pour des machines spécifiquement créés).

Dans les années 2000, ces travaux ont été popularisés sur les architectures x86 par les logiciels comme VirtualBox ou Xen.

En 2013, une nouvelle technique de virtualisation plus légère est apparu avec Docker, basée sur le système Linux de virtualisation LXC (Linux Containers).

Pour résumer, voici les quatre périodes clées à retenir sur la virtualisation :

1. 1970 : début de la virtualisation sur marchines professionnelles spécifiques
2. 1980-90 : virtualisation sur ordinateurs personnels
3. 2000 : popularisation de la virtualisation
4. 2010 : nouveau système léger : Docker

Virtualisation lourde

Chaque machine possède des ressources matérielles (CPU, RAM, disques, etc.). Toutes ces ressources sont utilisées par des applications en s'appuyant sur un système d'exploitation (utilisation indirecte).

Le principe de la virtualisation est de partager ces ressources entre différents systèmes d'exploitation. Pour cela, la virtualisation doit respecter deux principes fondamentaux :

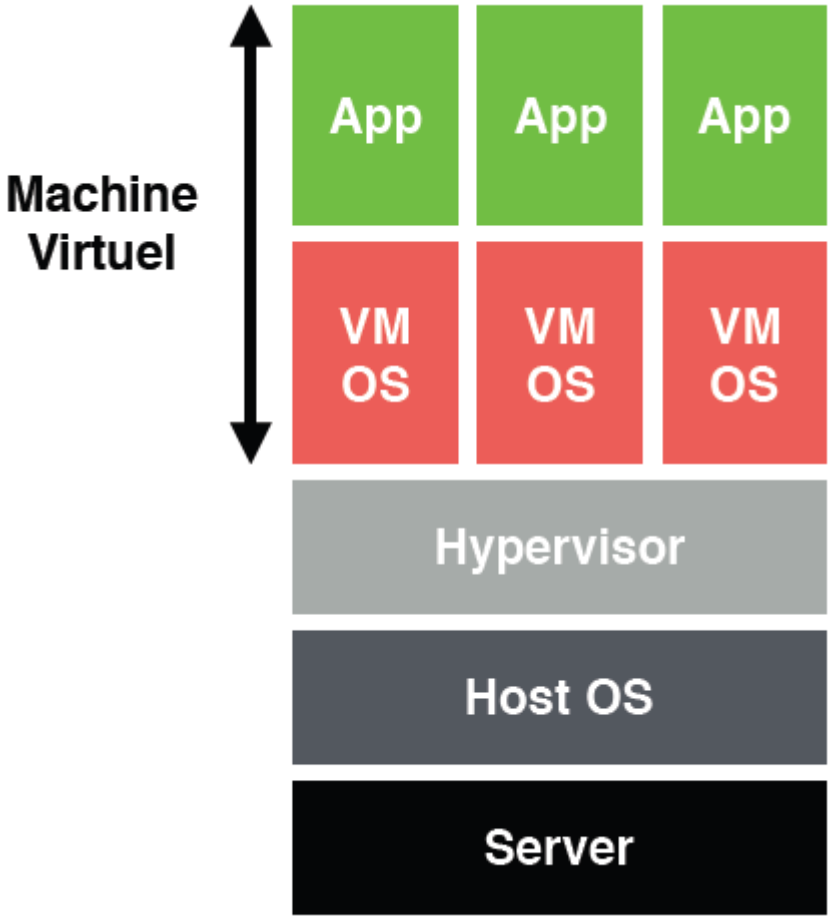
1. **le cloisonnement** : chaque système d'exploitation a un fonctionnement indépendant et ne peut interférer avec les autres ;
2. **la transparence** : fonctionner en mode virtualisé ne change rien au fonctionnement du système d'exploitation et à ses applications.

Cette virtualisation s'effectue avec ce que l'on appelle un hyperviseur, qui est une plateforme permettant à plusieurs systèmes d'exploitation de travailler sur une même machine en même temps.

Nous allons nous intéresser ici à un seul type de d'hyperviseur, celui de type II.

Cet hyperviseur (de type II) est en fait un "simple" logiciel (exemple : VirtualBox) qui tourne directement sur le système d'exploitation de votre machine. Il virtualise le matériel pour les OS invités qui croient alors dialoguer directement avec les ressources matérielles. Cependant le CPU est directement accessible.

Le doublet OS invité-Applications forme ce que l'on appelle une machine virtuelle, comme le résume le schéma suivant :



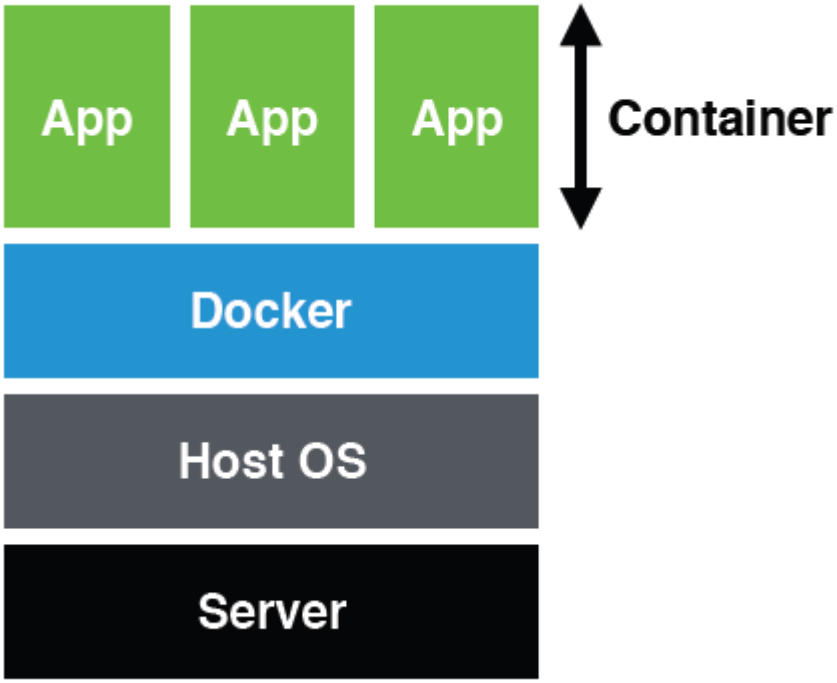
On a donc bien cloisonné les applications qui tournent sur des systèmes d'exploitation différents. Cependant, cette technique est lourde car elle fait intervenir plusieurs systèmes d'exploitation...

Pour information, d'autres types de virtualisation lourde existent (par exemple avec un hyperviseur de type I ou *Bare Metal*). Si cela vous intéresse, vous pourrez trouver des schémas et explications simples sur la page Wikipedia ou bien dans ce cours.

Virtualisation légère

Docker reprend les principes de cloisonnement et de transparence pour offrir une solution de virtualisation qui isole les applications dans ce qu'on appelle des conteneurs.

Ici, pas d'hyperviseur ni de système d'exploitation supplémentaire, Docker s'appuie sur le format de conteneur Linux standard (LXC) pour offrir une solution de virtualisation qui exécute les processus de façon isolée, comme le présente le schéma suivant :



Cette solution étend les fonctionnalités du système d'exploitation fournies par l'infrastructure sous-jacente. Le véritable avantage de cette solution, outre sa légèreté, est d'automatiser le déploiement d'applications de manière flexible que ce soit sur une machine locale, dans le *cloud*, sur une machine nue (comme dans le TP), etc.

Comparaison

Pour comparer les performances des machines virtuelles et des containers Docker, voici un tableau de comparaison (Source) :

	Machine virtuelle	**Container Docker**
Taille	Quelques Go	Quelques Mo
Démarrage	~ 30-45s	~ 50ms
Arrêt	~ 5-10s	~ 50ms

Un peu de vocabulaire

Docker repose sur différents concepts bien distincts. Il est important de bien distinguer ces derniers.

On parle de *machine hôte* ou *host machine* lorsqu'il s'agit du système d'exploitation qui va faire tourner les *conteneurs* ou *containers*. Ces *conteneurs* sont générés à partir d'*images*. Docker est installé sur la *machine hôte* et permet de se provisionner en *images* à partir de *hub* comme *docker hub* ou bien en générant ses propres *images* à partir d'un *dockerfile*. C'est à partir de ces *images* que l'on peut construire et déployer des *conteneurs* qui correspondent à un instantané d'une *image*. On verra tout au long de ce cours qu'il est possible d'exposer des *ports* et des *volumes* afin de clairement isoler nos *conteneurs*.

Le Dockerfile

Docker utilise les *directives* d'un *Dockerfile* afin de construire un *conteneur*. Ces *directives* sont exécutées à la suite les unes des autres. Le *contexte* dans lequel une *image* est construite est tout aussi important. Enfin, Docker décompose chacune des étapes de la construction d'un image en *layers* ou __couches_. Ce mécanisme permet d'augmenter de manière significative des constructions ultérieures. En effet, un *layer* pré-existant pourra être utilisé afin d'augmenter la vitesse de construction.

FROM

```
FROM <image>:<tag>
```

La directive **FROM** indique à partir de quelle *image* va être construite la nouvelle. Le *tag* correspond a la version de l'image qui va être utilisée. Le *tag* est facultatif et est *latest* par défaut.

MAINTAINER

```
MAINTAINER <name>
```

La directive **MAINTAINER** permet de mentionner l'auteur de l'image.

RUN

```
RUN <command>
RUN ["executable", "param1", "param2"]
```

La directive **RUN** possède deux syntaxes différentes. Celle-ci, comme son nom l'indique, permet d'exécuter des commandes au sein de l'image lors de la construction. A noter que le répertoire d'exécution de ces commandes peut être défini à l'aide de la directive **WORKDIR**.

WORKDIR

```
WORKDIR /path/to/workdir
```

La directive **WORKDIR** permet de définir le répertoire courant pour l'ensemble des autres directives suivantes.

USER

```
USER elucterio
```

La directive **USER** permet de définir l'utilisateur qui exécutera les directives **RUN**, **CMD** and

ENTRYPOINT qui suivent dans le *Dockerfile*.

CMD

```
CMD ["executable","param1","param2"]
CMD command param1 param2
```

La directive **CMD** définie la première commande qui sera exécutée par un *conteneur* construit à partir de cette *image*. A noter que si plusieurs directives__CMD__ sont présente sans le Dockerfile, seule la dernière sera prise en compte.

EXPOSE

```
EXPOSE <port> [<port>...]
```

La directive **EXPOSE** quant à elle, permet d'exposer des ports qui seront accessibles en dehors du *conteneur* par le système d'exploitation *hôte*. Il faut noter qu'un port exposé doit tout de même être rendu accessible à l'aide du flag `-p <port>:<public_port>` lors de l'exécution du *conteneur*.

VOLUME

```
VOLUME ["/data"]
```

La directive **VOLUME** permet d'exposer des volumes qui seront accessibles en dehors du *conteneur* par le système d'exploitation *hôte*. Il faut noter qu'un volume exposé doit tout de même être rendu accessible à l'aide du flag `-v <volume>:<public_volume>` lors de l'exécution du *conteneur*.

ENV

```
ENV <key> <value>
ENV <key>=<value> ...
```

La directive **ENV** permet de définir des variables d'environnement pour la construction mais qui persisteront aussi dans les *conteneurs* construits à partir du *Dockerfile*.

COPY

```
COPY <src>... <dest>
COPY ["<src>",... "<dest>"]
```

La directive **COPY** permet d'ajouter des fichiers ou dossiers dans *l'image*. Le chemin *src* correspond à un chemin d'accès absolu ou relatif de la *machine hôte* et le chemin *dest* correspond à chemin d'accès absolu ou bien relatif au chemin de la directive **WORKDIR**.

ADD

```
ADD <src>... <dest>
ADD ["<src>",... "<dest>"]
```

La directive **ADD** ressemble beaucoup à la directive **COPY**. Cependant, la directive **ADD** autorise les url pour *src* et si c'est un archive, elle sera automatiquement décompressée.

ENTRYPOINT

```
ENTRYPOINT ["executable", "param1", "param2"]
ENTRYPOINT command param1 param2
```

L'objectif de la directive **ENTRYPOINT** est de rendre exécutable un *conteneur*. Les arguments passés après `docker run <image>` seront ajoutés à la suite des éléments de la directive **ENTRYPOINT** et écraseront les arguments de la directive **CMD**. Tout comme la directive **CMD**, seule la dernière directive **ENTRYPOINT** sera prise en compte.

Exemple de Dockerfile

```
# Firefox over VNC
#
# VERSION               0.3

FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir ~/.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the trick)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD    ["x11vnc", "-forever", "-usepw", "-create"]
```

Vous pouvez retrouver l'intégralité des directives et d'autres exemples dans la documentation officielle de Docker : https://docs.docker.com/engine/reference/builder/

Docker-compose

`docker-compose` est un autre outil que `docker`. Il permet de définir tout une architecture de service et les liens entre eux. Un service est un container avec des paramètres de lancement.

docker-compose.yml

Pour utiliser `docker-compose` il suffit de créer un fichier `docker-compose.yml`. Dedans, on y définit

les services et leurs liens. Dans un `Dockerfile`, on définit le container mais on ne définit pas les paramètres de "lancement" de ce container. C'est le rôle du fichier `docker-compose.yml`.

Voici la structure d'un fichier `docker-compose.yml` :

```
version: '2'
services:               # list of services
  service_1:            # name of the service
    build:              # how to build the container
        context: ./path/to/dockerfile
        dockerfile: name_of_dockerfile
    ports:              # which port to bind
      - "<port_host>:<port_container>"
    volume:             # which volume to share
      - "<host_dir>:<container_dir>"
    depends_on:         # which services it depends on
      - service_2
  service_2:
                        # how to build the container
    image: docker_image_name
```

La liste de toutes les instructions est disponible ici.

BUILD

L'instruction `build` permet de spécifier comment créer le container qui caractérise le service X. Cette instruction peut avoir plusieurs paramètres. Par défaut, il suffit juste de lui donner le chemin vers le répertoire où se trouve le `Dockerfile`.

```
...
    service_1:
        build: ./path/to/Dockerfile
...
```

Si le nom du `Dockerfile` diffère, on peut le spécifier :

```
...
    service_1:
        build:
            context: ./path/to/Dockerfile
            dockerfile: name_of_dockerfile
...
```

D'autres options existent mais ne sont pas présentées dans ce cours.

PORT

L'instruction `port` permet de spécifier le binding avec la machine host. C'est l'équivalent du `-p` lors

d'un `docker run ...`. Par exemple :

```
...
    service_1:
        port:
            - "9000":"80"
            - "9001":"81"
...
```

VOLUME

L'instruction `volume` permet de spécifier le partage de volume avec la machine host. C'est l'équivalent du `-v` lors d'un `docker run ...`. Par exemple :

```
...
    service_1:
        volume:
            - .:/code   # share current dir in /code
            - /tmp      # share /tmp in /tmp
            - /tmp:/foo # share /tmp in /foo
...
```

IMAGE

Lorsque nous n'avons pas besoin de `Dockerfile` mais que nous souhaitons définir notre container à partir d'une image de base `Docker`, on peut utiliser la directive `image` afin de spécifier à partir de quelle image, le container (associé au service) doit être défini. Ici, on définit une instance de `mongodb` à partir de son image "officielle" Docker :

```
...
    mongo_db_service:
        image: mongo
...
```

DEPENDS_ON

Cette directive est l'une des plus importantes. Elle permet de lier plusieurs containers entre eux. Cela permet aussi d'injecter dans le fichier `/etc/host` du container, le nom de domaine du container dont dépend le service définit. On présente par la suite un exemple concret.

L'application X a été developpée en `go lang`. Elle a besoin d'une instance de `redis` (une base de données en mémoire centrale) pour tourner. Dans le code de l'applicaiton `go lang`, on retrouve un fichier de configuration pour définir les constantes de l'application, par exemple, l'adresse du serveur `redis`.

On souhaite dockeriser cette architecture. Mais avant de lancer à l'avance les containers, on ne sait pas quelle adresse aura le container faisant tourner `redis`. On utilise alors un nom de domaine,

celui du service.

Voici le fichier `docker-compose.yml` associé :

```
version: '2'
services:               # list of services
  app_go:               # go lang service
    build: ./my_golang_app
    ports:
      - "80:9000"       # the app is running on port 9000
    depends_on:         # the app depends on redis
      - redis_service
  redis_service:
    image: redis        # building the redis container from the official image
```

Au lancement de `docker-compose`, ce dernier va injecter dans le fichier `/etc/host` du container `app_go`, le couple (nom de service, adresse IP) des containers listés dans les dépendences. Dans notre cas, l'entrée suivante sera ajoutée :

```
redis_service 172.17.0.3
```

Ainsi, depuis le container `app_go`, on pourra contacter l'instance de `redis` grâce au nom de domaine `redis_service`. Il faut donc penser à changer les constantes dans son code pour joindre les différents services que l'on dockerise.

Exécuter docker-compose

Rien de plus simple. Placez-vous dans le répertoire qui contient le fichier `docker-compose.yml` et exécutez :

```
$ docker-compose up
```

Pour forcer à builder les images, utilisez :

```
$ docker-compose up --build
```

Les bonnes pratiques

Pourquoi utiliser Docker ? A quoi sert Docker ?

Pour résumer, Docker permet de construire et partager très facilement des images disques. Il permet aussi de lancer facilement différents OS. De plus, contrairement à une VM, Docker est très léger en termes de configurations minimales.

Quand utiliser Docker ?

Docker est utile dans de nombreuses situations et notamment quand : - on veut un système de contrôle de version pour un OS complet, - on veut distribuer/collaborer sur une OS complet avec une équipe, - on veut exécuter son code sur son PC dans le même environnement que celui du serveur, - l'application dispose de plusieurs phases de développement en même temps dev/test/qa/prod

Cas d'utilisation de Docker

Prenons un cas d'utilisation simple et concret. Une entreprise rennaise, souhaite développer une application comme ils savent le faire. Leur équipe est composée de deux personnes. Valentin est un vieux de la vieille. Lui, Debian 7, c'est parfait.

Florent lui est un vrai fanboy, il ne jure que par Apple.

En effet, ça pose problème que deux développeurs ne travaillent pas sur les mêmes environnements... Mais on ne leur a pas laissé le choix, ils ont un serveur de production qui tourne sur ubuntu et ils utilisent ruby pour leur appli.

Ils pourraient se compliquer la vie avec des montages obscurs mais ils ont suivi avec attention le cours sur Docker. Ils décident donc de définir un environnement commun grâce à Dockerfile. Dedans, ils peuvent définir tout ce dont ils ont besoin pour que leur application ruby tourne n'importe où, sous Debian, macOS ou même Ubuntu. Aussi, lors de l'ajout de fonctionnalités, ils peuvent modifier ce Dockerfile pour ajouter de nouvelles dépendences ruby par exemple. Ainsi leur application tourne indépendemment de l'OS sur lequel ils ont l'habitude de travailler et ça fonctionne correctement (évidemment tant qu'on ne push pas n'importe quoi sur le repo) !

Quels sont les vrais avantages de Docker ?

Comme le container n'embarque pas d'OS, à la différence de la machine virtuelle, il est par conséquent beaucoup plus léger que cette dernière. Il n'a pas besoin d'activer un second système pour exécuter ses applications. Cela se traduit par un lancement beaucoup plus rapide, mais aussi par la capacité à migrer plus facilement un container d'une machine physique à l'autre, du fait de son faible poids. Typiquement, une machine virtuelle pourra peser plusieurs Go, alors qu'un container nu représentera, lui, quelques Mo. Grâce à leur légèreté, les containers Docker sont portables de cloud en cloud.

Côté développement

Docker peut apporter une plus-value en accélérant les déploiements parce que les containers Docker sont légers. Les basculer d'un environnement de développement ou de test à un environnement de production peut donc se faire presque en un clic, ce qui n'est pas le cas pour la VM, plus lourde. Du fait de la disparition de l'OS intermédiaire des VM, les développeurs bénéficient aussi mécaniquement d'une pile applicative plus proche de celle de l'environnement de production.

Docker permet dans le même temps de concevoir une architecture de test plus agile, chaque container de test pouvant intégrer une brique de l'application (base de données, langages, composants...). Pour tester une nouvelle version d'une brique, il suffit d'inter changer le container correspondant. Côté déploiement continu, Docker présente par ailleurs un intérêt car il permet de limiter les mises à jour au container nécessitant de l'être.

Côté production

Grâce à Docker, il est possible de containériser une application, avec pour chaque couche des containers isolant ses composants. C'est le concept d'architecture de micro services. Ces containers de composant, du fait de leur légèreté, peuvent eux-mêmes, chacun, reposer sur les ressources machines voulues.

Sources:
http://www.journaldunet.com/solutions/cloud-computing/1146290-cloud-pourquoi-docker-peut-tout-changer/ https://www.wanadev.fr/23-tuto-docker-comprendre-docker-partie1/

Docker Hub: Comment automatiser son workflow

La construction d'une image docker peut parfois être longue et répétitive lorsqu'il s'agit de mettre à jour un ou plusieurs *layers* de son image. Il peut aussi être nécessaire d'intégrer à son *workflow* la construction d'*images* en fin de chaîne pour ensuite les distribuer facilement. La plateforme Docker Hub propose gratuitement pour des projets open source, de se lier à votre compte Github afin de récréer une image à chaque modification sur votre *repository*. Pour cela, il suffit simplement de lier votre Github à Docker Hub. Ensuite, en créant un simple *repository* contenant votre Dockerfile, il est possible de faire en sorte qu'à chaque *commit*, *l'image* soit recréée automatiquement et ensuite proposée sur la plateforme Docker Hub. Dès lors, un simple `docker pull <login>/<image>` suffira à télécharger l'image depuis n'importe qu'elle instance Docker. Pour allez plus loin, sachez qu'il est possible d'héberger soi-même son propre registre docker afin de garder privé ses images.

Sources: https://docs.docker.com/registry/deploying/ https://hub.docker.com/