M3204
Introduction Docker

Patrice Gommery – Mai 2020

PRE-REQUIS

Pour réaliser ce TP vous devez avoir en votre possession :
- L'adresse IP de votre serveur VPS hébergé chez PulseHeberg
- Votre identifiant **MMI** (mmiXXxxx)
- Le mot de votre identifiant **MMI** créé lors de l'initialisation de votre VPS.

En cas de problème :
- Un accès à votre compte PulseHeberg pour lancer la console VNC
- Le mot de passe du compte root de votre VPS (premier mail de PulseHeberg)

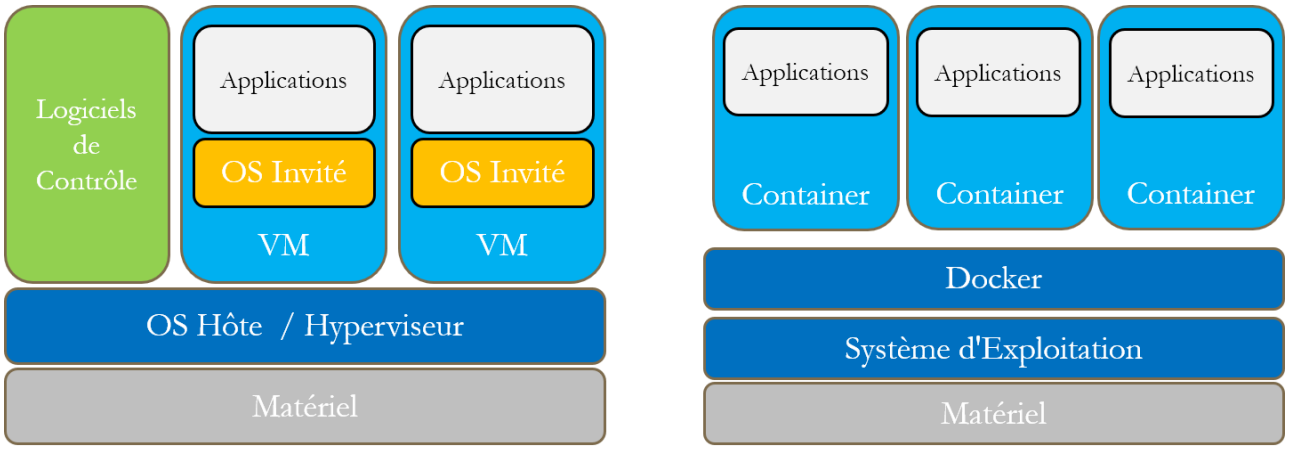
Consignes Générales pour le TP

Pendant tout le TP,
remplacez **VPS** l'adresse IP de votre VPS PulseHeberg
remplacez **MMI** par votre identifiant **MMI** (mmiXXxxx)

Préambule :

Ce TP est une introduction à la plateforme Docker. Un système de containers qui vous permettra de créer des environnements de tests ou de développement sans avoir à modifier en profondeur la configuration de votre VPS. Pour les besoins du cours, nous implémenterons Docker directement sur vos VPS, mais Docker peut aussi être installé directement sur vos postes de travail (Mac, Windows ou Linux) pour être exploité de la même façon.

Architecture Docker :



A gauche, une architecture basée sur des machines virtuelles (VPS) ou chaque VM embarque un système d'exploitation complet. Les machines sont gérées par un hyperviseur (type proxmox).Chaque machine est indépendante des autres, mais doit embarquer son propre système d'exploitation dans sa globalité.

A droite, une architecture basée sur des containers qui sont de simples processus isolés les uns des autres par les technologies présentes dans le noyau Linux comme les Cgroups et les NameSpaces . Ces technologies permettent l'isolation des ressources utilisées par chacun des containers tout en permettant à chaque processus d'accéder à l'ensemble de la machine hôte. Chaque container ne contient que le minimum requis pour faire tourner l'application embarquée. Le noyau et les librairies du système hôte sont mis en commun pour tous les containers.

Docker va donc vous permettre d'implémenter et de gérer vos propres containers avec des applications différentes sans avoir besoin de mettre à jour ou de reconfigurer votre système de base.

Les principaux composants de Docker qui seront abordés dans ce TP :

Docker Engine	: L'application Docker (Client/serveur) qui exécute et gère les containers
Docker Containers	: Les containers Docker
Docker Images	: Les images Docker qui permettent de créer les containers.

Partie 1 : INSTALLATION

Pour commencer , nous allons donc installer Docker Engine sur nos serveurs VPS .
L'installation est assez classique et reprend le même schéma que clles utilisées pour les mises à jour de PHP et MariaDB dans les TP précédents : Ajout d'un dépôt supplémentaire et installation du paquet désiré.

MISE A JOUR DE NOTRE SYSTEME :

```
apt update
apt install  apt-transport-https  ca-certificates  curl  gnupg-agent  software-properties-common
```

À la suite des mises à jour de PHP et MariaDB, la plupart des paquets sont déjà présents sur votre VPS.

A JOUT DU DEPOT DOCKER :

```
curl  -fsSL  https://download.docker.com/linux/debian/gpg  |  sudo apt-key add  -
add-apt-repository  "deb [arch=amd64]  https://download.docker.com/linux/debian stretch stable"
apt update
```

IMPORTANT : En cas d'erreur lors de l'update, ne continuez pas le TP sans avoir corrigé les problèmes.
stretch est la version de Debian installée initialement sur vos VPS, si vous l'avez mise à jour vers buster, n'oubliez pas de changer cette valeur.

INSTALLATION DE LA VERSION COMMUNITY EDITION DE DOCKER :

```
apt install  docker-ce
apt install  docker-ce-cli
apt install  containerd.io
```

VERIFICATION DE L'INSTALLATION AVEC UN CONTAINER DE TEST :

```
docker container run hello-world
```

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:8e3114318a995a1ee497790535e7b88365222a21771ae7e53687ad76563e8e76
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Comme vous l'explique le reste du message :

1) Docker n'a pas trouvé l'image en local	(Unable to find image)
2) Il l'a donc téléchargée depuis le docker Hub	(Pulling from libray/hello-world)
3) Il a ensuite créé un container à partir de l'image	(docker container run)
4) et renvoyé le résultat de son exécution sur la sortie standard	(echo Hello from Docker!)

Si le message **Hello from Docker!** apparait, votre installation est correcte.

CONFIGURATION SUPPLEMENTAIRE :

Pour terminer, nous allons autoriser notre utilisateur **MMI**, ainsi que l'utilisateur **prof** à exécuter Docker (sans avoir à faire sudo). Ceci pour éviter de lancer Docker en tant que **root** et éviter certaines erreurs que vous pourriez commettre. Nous allons aussi désactiver Docker au reboot de votre VPS pour éviter de charger votre machine inutilement lors d'un reboot.

```
usermod -aG docker MMI
usermod -aG docker prof
systemctl disable docker
```

Pour conclure l'installation et vérifier nos derniers changements, nous allons rebooter la machine.

```
reboot
```

Attendez un peu (*2 à 3mn, peut-être plus sur certaines machines)* et reconnectez-vous en **ssh** avec votre utilisateur comme d'habitude, **mais ne faites pas sudo -i une fois connecté**. Restez sur l'invite de commandes : **MMI@MMI:~$** et saisissez la commande :

```
ps -aux | grep docker
```

```
MMI  2355  0.0  0.0  12776   904 pts/1    S+   16:30   0:00 grep docker
```

Vous ne devez voir apparaitre qu'une seule ligne, celle de la commande que vous venez de saisir.
Docker n'est donc pas démarré au démarrage . Pour lancer Docker Engine :

```
sudo systemctl start docker
ps -aux | grep docker
```

```
root     ...    16:33   0:00 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
MMI      ...    16:33   0:00 grep docker
```

On voit bien ici, le daemon dockerd qui est démarré.
Essayons maintenant de créer et d'exécuter un autre container de test (sans faire de sudo)

```
docker run hello-world
```

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Nous pouvons donc manipuler nos containers sans avoir besoin d'être connecté en tant que root. Ceci évitera les confusions lorsque nous travaillerons dans les containers et sécurisera aussi notre installation. Nous pouvons aussi arrêter et démarrer Docker en fonction de nos besoins.

Rappels :

```
Pour stopper Docker          : sudo systemctl stop docker
Pour démarrer Docker         : sudo systemctl start docker
Pour vérifier s'il est lancé : ps -aux | grep docker
```

Pour voir la version de docker installée :

```
docker version
```

Pour valider votre TP, créez un fichier **TP05.txt** dans votre dossier **/mmitrans** et répondez aux différentes questions en utilisant comme d'habitude le format QX:REPONSE (Une ligne par question)

Q1: Quelle version de Docker Engine avez-vous installée sur votre VPS ?

REMARQUE : En tant que simple utilisateur , vous n'avez pas le droit d'écrire dans le dossier /mmitrans, vous devrez donc saisir : sudo nano /mmitrans/TP05.txt pour éditer le fichier.

Partie 2 : CONTAINERS

Docker est maintenant installé et configuré sur notre VPS. Essayons maintenant de comprendre les différents commandes permettant de manipuler les containers.

La syntaxe est assez simple : ***docker container COMMAND [Options]***
Les commandes de bases sont les suivantes :

run	Création d'un container
ls	Liste des containers
inspect	Détails d'un container
logs	Visualisation des logs
exec	Lancement d'un processus dans un container existant
stop	Arrêt d'un container
rm	Suppression d'un container

CREATION ET VIE D'UN CONTAINER :

Un container est créé à partir d'une image. Des images de bases sont disponibles sur le Docker Hub qui est donc le repository officiel de Docker. Vous pouvez consultez les images disponibles à l'adresse : https://hub.docker.com/search?q=&type=image

On y retrouve des images pour les principales distributions Linux (Debian, Ubuntu, Centos, fedora, OpenSuse ...) mais aussi des versions plus légères comme Alpine que nous verrons un peu plus tard. Vous y trouverez aussi des images pour créer des plateformes applicatives complètes avec des langages (php, ruby, python, java ..) , des bases de données (mysql, mariadb, postgress, mongodb, ..) et même des CMS prêts à être exécutés comme Wordpress ou Joomla par exemple.

Dans un premier temps, nous travaillerons avec des containers basés sur l'image **alpine**, une distribution légère de linux suffisante pour un environnement de travail simple et surtout d'une taille réduite.

Commençons par créer un premier container basé sur cette image :

```
docker container run alpine
```

```
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
cbdbe7a5bc2a: Pull complete
Digest: sha256:9a839e63dad54c3a6d1834e29692c8492d93f90c59c978c1ed79109ea4fb9a54
Status: Downloaded newer image for alpine:latest
```

Comme pour l'exemple hello-world fait en début de tp, vous voyez que Docker a téléchargé automatiquement l'image (pull) à partir du Docker Hub puisqu'elle n'existait pas en local (Unable to find). Comme nous n'avons pas précisé de version de l'image, il a donc pris la plus récente (latest).

Regardons ce qu'est devenu notre container :

```
docker container ls
```

Il n'apparait pas dans la liste ? Pas de panique, cela est tout à fait normal et nous allons essayer de comprendre pourquoi. Saisissez de nouveau la commande, mais avec l'option **-a** *(all)*

```
docker container ls -a
```

```
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS                      PORTS   NAMES
b61479cb30c4   alpine    "/bin/sh"   9 minutes ago    Exited (0) 2 minutes ago            pedantic_swirles
```

Notre container est bien là, mais comme vous pouvez le voir son exécution est terminée (***Exited***)

A quoi peut bien servir un container qui s'arrête immédiatement après son lancement ?
Effectivement, si nous le créons avec simplement avec **run** sans aucune option, ça ne sert à rien .
Le container se lance, mais comme aucun processus n'est exécuté dans celui-ci, il s'arrête immédiatement. C'est le principe de base d'un container, celui-ci ne reste en activité que si au moins un processus (ou un service) est actif dans le container. Et ce n'est bien sûr pas le cas avec notre image alpine, puisqu'elle ne contient aucun applicatif, seulement un système de base. La seule commande exécutée est l'appel du shell (/bin/sh) , mais sans interactivité elle est aussitôt suivi d'un exit qui met fin a l'exécution du container.

Pour que le container ne se termine pas automatiquement, il faut donc interagir avec lui.
Créons un nouveau container, mais cette fois si en utilisant l'option **-ti** *(terminal interactif)*

```
docker container run -ti alpine
```

```
/ #
```

Là, le container ne vous rend pas la main et attend que vous saisissiez une commande. Pour être plus précis c'est le shell (/bin/sh) lancer automatiquement en fin de création du container et le terminal qui attendent vos instructions. Essayez ceci :

```
ps
```

```
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    6 root      0:00 ps
```

On peut voir ici les processus actifs dans le container. Notamment le processus initial qui est donc bien /bin/sh . Tant que ce processus sera actif, le container restera en exécution et comme vous pouvez le constater ne vous rendra pas la main. Pour récupérer votre invite de commandes, saisissez :

```
exit
```

Maintenant, l'exécution du container est terminée (vous êtes revenu sur votre invite de commandes). La commande exit permet effectivement de clore la session du shell, mais a aussi mis fin au processus initial du container et l'a donc fermé. Saisissez de nouveau la commande :

```
docker container ls -a
```

```
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS                      PORTS   NAMES
1a3fb10ba4dc   alpine    "/bin/sh"   5 minutes ago    Exited (0) 2 minutes ago            elastic_goodall
b61479cb30c4   alpine    "/bin/sh"   15 minutes ago   Exited (0) 10 minutes ago           pedantic_swirles
```

Vous constaterez qu'un second container a été créé, toujours avec l'image alpine, mais qu'il est , comme son prédécesseur arrêté depuis quelques minutes.

Comment faire la différence entre nos deux containers ?

Pour cela nous disposons de deux informations :
CONTAINER ID qui est donc un identifiant unique créé lors de la création du container
NAMES qui est donc un nom unique attribué lui aussi lors de la création du container.

Remarquez que les noms sont générés de façon aléatoire en regroupant deux mots (en anglais) séparés par un underscore. Ces noms n'ayant aucun lien avec l'image et donc le contenu du container, il peut devenir rapidement difficile de les retenir. Heureusement, l'option **--name** permet de nommer vos containers plus simplement lors de la création.

Comment faire pour laisser le container actif et pouvoir interagir avec lui en même temps ?

Il suffit simplement d'ajouter l'option -d (detach) à notre commande qui va donc lancer notre container en tache de fond. Relançons notre commande et profitons-en pour donner un nom plus simple à notre container :

```
docker container run --name alpine3 -dti alpine
```

Remarquez que nous avons bien saisi **-dti** pour que le container reste actif mais en tache de fond. Si vous omettez ti, le container sera lancé en tâche de fond, mais se terminera aussitôt comme dans les premiers exemples.

```
3d8631c565cc5e91f0c33fe00b200a8f390f6c71222cc46e3cbd4649c8145078
```

Là , Docker vous a rendu la main, mais pas d'interactivité malgré le "**ti**" ? Il vous a simplement affiché l'ID complet du container qu'il vient de créer. Relançons notre commande pour lister les containers :

```
docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
3d8631c565cc	alpine	"/bin/sh"	2 minutes ago	**Up** 2 minutes		**alpine3**
1a3fb10ba4dc	alpine	"/bin/sh"	10 minutes ago	Exited (0) 7 minutes ago		elastic_goodall
b61479cb30c4	alpine	"/bin/sh"	20 minutes ago	Exited (0) 15 minutes ago		pedantic_swirles

Nous voyons bien notre container **alpine3** et nous constatons aussi qu'il est toujours actif.
Nous pouvons donc interagir avec lui en utilisons une nouvelle commande **exec**

Deux possibilités s'offrent à nous.
Soit envoyer une commande au shell qui est déjà lancer sur notre container :

```
docker container exec alpine3 ps
```

```
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
   55 root      0:00 ps
```

ici nous envoyons la commande **ps** qui nous affiche donc les processus en cours et nous redonne la main

Soit ouvrir un nouveau shell (/bin/sh) en mode interactif et saisir plusieurs commandes si nous voulons :

```
docker container exec -ti alpine3 /bin/sh
```

```
/ # ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
   60 root      0:00 /bin/sh
   65 root      0:00 ps
/ # exit
```

Si nous saisissons la commande **ps** après l'invite de commandes du container, nous voyons bien 2 processus /bin/sh de lancés. Pour quitter la session, il suffit de taper **exit** . Mais puisqu'il avait été lancé en background et que le premier shell est toujours actif, le container ne s'arrêtera pas.

Comment stopper un container tournant en tâche de fond?

Rien de plus simple : utilisez la commande **stop** suivi du nom (ou de l'ID) du container

```
docker container stop alpine3
```

```
alpine3
```

Docker vous renvoi le nom (ou l'ID) du container stoppé.

Vous pouvez vérifier immédiatement que le container est bien stoppé :

```
docker container ls -a
```

```
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS                      PORTS   NAMES
3d8631c565cc   alpine    "/bin/sh"   4 minutes ago    Exited (0) 2 minutes                alpine3
1a3fb10ba4dc   alpine    "/bin/sh"   10 minutes ago   Exited (0) 7 minutes ago            elastic_goodall
b61479cb30c4   alpine    "/bin/sh"   20 minutes ago   Exited (0) 15 minutes ago           pedantic_swirles
```

avant de continuer, répondez aux questions suivantes :

Q2:Quels sont les noms des trois containers que vous avez créés ? (séparez les réponses par une virgule)
Q3:Quels sont les ID (courts) de vos trois containers ?
Q4:Quel est le PID du processus initial d'un container ?
Q5:Quelle option de la commande run permet d'ouvrir un terminal interactif ?
Q6:Quelle option de la commande run permet d'exécuter le container en background ?

Comment supprimer un container dont nous n'avons plus besoin ?

La commande **rm** est faites pour ça. Attention par défaut , elle ne supprime que des containers inactifs. Pour forcer la suppression d'un container actif, vous pouvez utiliser l'option **-f**

```
docker container rm alpine3
```

```
alpine3
```

Docker vous renvoi le nom (ou l'ID) du container supprimé.

```
docker container ls -a
```

```
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS                      PORTS   NAMES
1a3fb10ba4dc   alpine    "/bin/sh"   15 minutes ago   Exited (0) 10 minutes ago           elastic_goodall
b61479cb30c4   alpine    "/bin/sh"   27 minutes ago   Exited (0) 20 minutes ago           pedantic_swirles
```

Et l'on constate bien que le container n'existe plus.

Une astuce (à utiliser avec modération) pour supprimer tous les containers en une seule commande :
*(L'option **-q** derrière la commande ls ne renvoie que les ID courts des containers)*

```
docker container rm $(docker container ls -qa)
```

Si certains containers ne sont pas stoppés, Docker vous renverra une erreur pour ceux-là, mais supprimera bien tous les containers inactifs. Si vous voulez vraiment supprimer tous les containers créés sur votre machine, y compris ceux en fonctionnement, ajoutez l'option **-f** ou stoppez les avant la suppression .

```
docker container stop $(docker container ls -q)
docker container rm $(docker container ls -qa)
```

INSPECTION ET LOGS DES CONTAINERS :

Pour en terminer avec les containers, nous allons nous intéresser aux deux dernières commandes de notre tableau : **inspect** et **logs**.

Commençons par recréer un nouveau container qui aura pour seul fonction : faire un ping vers une machine (en l'occurrence un serveur DNS libre de Google). Nous allons utiliser de nouveau notre image **alpine**, lancer le container en background (**-d**) et surtout exécuter le **ping** immédiatement après son lancement .

Contrairement aux exemples précédent, nous n'allons pas utiliser la commande **exec**, mais simplement donner la commande **ping** comme argument en fin de création. Celle-ci deviendra alors la commande de sortie de création (et remplacera le /bin/sh initial) . Comme le ping est en continu, le container ne sera pas stoppé. L'option **ti** n'est pas nécessaire non plus, car nous n'avons pas besoin d'interagir avec le container.

```
docker container run -d --name alpine4 alpine ping 8.8.8.8
docker container ls
```

```
CONTAINER ID   IMAGE     COMMAND          CREATED          STATUS          PORTS     NAMES
28522b941f0b   alpine    "ping 8.8.8.8"   7 seconds ago   Up 5 seconds              alpine4
```

On voit bien ici que notre container est toujours actif et qu'il exécute notre ping.
Nous pouvons aussi vérifier que le ping est bien notre processus initial dans le container

```
docker container exec alpine4 ps
```

```
PID   USER     TIME  COMMAND
    1 root      0:00  ping 8.8.8.8
   12 root      0:00  ps
```

Avec la commande **logs** nous pouvons vérifier le résultat de notre ping . Pour le voir en temps réel, ajoutons aussi l'option **-f** (*follow*)

```
docker container logs -f  alpine4
```

```
64 bytes from 8.8.8.8: seq=2163 ttl=57 time=1.458 ms
64 bytes from 8.8.8.8: seq=2164 ttl=57 time=1.400 ms
64 bytes from 8.8.8.8: seq=2165 ttl=57 time=1.468 ms
...
```

CTRL+C pour sortir des logs

Dernière commande de notre tableau : **inspect** . Comme son nom l'indique elle va nous permettre d'inspecter notre container et donc de connaitre les détails de sa configuration. Le résultat de la commande est au format json (https://docs.docker.com/engine/reference/commandline/inspect/)
Le contenu étant très complet, il est intéressant d'utiliser la commande avec l'option -f (format) qui permet d'extraire un élément particulier en précisant l'index json. Quelques exemples :

```
docker container inspect -f  '{{ .NetworkSettings.IPAddress }}' alpine4
docker container inspect -f  '{{ .NetworkSettings.MacAddress }}' alpine4
```

retournent respectivement l'adresse IP et l'adresse MAC du container

```
docker container inspect -f  '{{ .Config.Image }}' alpine4
```

retourne le nom de l'image ayant servi à la création du Container

avant de continuer, répondez aux questions suivantes :
Q7:Quelle est l'adresse IP du container alpine4 ?
Q8:Quelle est l'adresse MAC du container alpine4
Q9:Quelle option de la commande logs permet d'afficher le flux en temps réel ?
Q10:Quelle option de la commande rm permet de supprimer un container même s'il est actif ?

Partie 3 : Networking

Comme nous venons de le voir, un container peut effectivement posséder une ou plusieurs interfaces réseaux et donc des adresses ip . Il possède aussi en fonction des services implémentés des ports en écoute (comme par exemple 80 pour HTTP). Il peut donc communiquer avec l'extérieur comme nous l'avons vu avec le ping vers le serveur DNS de Google.

Regardons de plus près la configuration de notre container alpine4 :
La partie qui nous intéresse se trouvant à la fin de la configuration, nous n'utiliserons pas l'option -f.

```
docker container inspect  alpine4
```

```
"Networks": {
            "bridge": {
                "IPAMConfig": null,
                "Links": null,
                "Aliases": null,
                "NetworkID": "ed58ed0ce1e5a51c712b81aab8657e1714b5d7d3a4baff7e67c81c5ce6eca86d",
                "EndpointID": "d7843c724d42e31e9de3fa1a588d3b95baeb6cbbc86ac3644787b513599f66a2",
                "Gateway": "172.17.0.1",
                "IPAddress": "172.17.0.2",
                "IPPrefixLen": 16,
                "IPv6Gateway": "",
                "GlobalIPv6Address": "",
                "GlobalIPv6PrefixLen": 0,
                "MacAddress": "02:42:ac:11:00:02",
                "DriverOpts": null
            }
```

Nous retrouvons ici l'adresse IP de notre container (**172.17.0.2**) mais aussi l'adresse IP d'une passerelle (Gateway) grâce à laquelle il peut communiquer avec l'extérieur. Cette passerelle est tout simplement votre VPS sur lequel Docker a implémenté une interface particulière : **docker0** . Vous pouvez visualiser cette interface avec la commande : **ifconfig** *(faites sudo -i avant)*

```
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:f4ff:fe29:8cca  prefixlen 64  scopeid 0x20<link>
        ether 02:42:f4:29:8c:ca  txqueuelen 0  (Ethernet)
```

Ce **réseau 172.17.0.0** (Réseau privé de classe B) est donc implémenté sur votre VPS et c'est à travers lui que les containers peuvent communiquer entre eux et par extension avec l'extérieur.

Utilisez la commande **route -n** pour voir les réseaux de votre VPS :

```
Table de routage IP du noyau
Destination     Passerelle      Genmask         Indic Metric Ref    Use Iface
0.0.0.0         149.91.90.129   0.0.0.0         UG    0      0        0 eth0
149.91.90.128   0.0.0.0         255.255.255.192 U     0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
```

Si vous vous souvenez de vos cours de réseaux en S1 (ce dont je ne doute pas :-) . 172.17.0.0 est un réseau privé, donc non routable sur internet . *Comment votre container va-t-il alors pouvoir communiquer avec une machine sur Internet si personne ne peut le joindre ?* Rappelez-vous du NAT (Network Address Translation) utilisé par votre box pour faire communiquer vos postes internes avec internet.
Ici c'est la même chose. Cette translation d'adresse est faite grâce à NetFilter, le pare-feu implémenté dans le noyau de Linux. La commande **iptables** permet de voir cette fonctionnalité :

```
iptables -t nat -nL
```

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            !127.0.0.0/8         ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16       0.0.0.0/0

Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0
```

Sans rentrer dans les détails (cela pourrait faire le sujet d'un autre cours).On voit ici qu'en sortie de notre machine (**POSTROUTING**) toutes les paquets ayant comme IP source une adresse du réseau **172.17.0.0** sont donc soumis à la **cible MASQUERADE** . Cette cible (**target**) remplace simplement l'adresse IP source par l'adresse IP de votre VPS.

Les postes externes voulant communiquer avec votre container, s'adresseront donc à votre VPS.

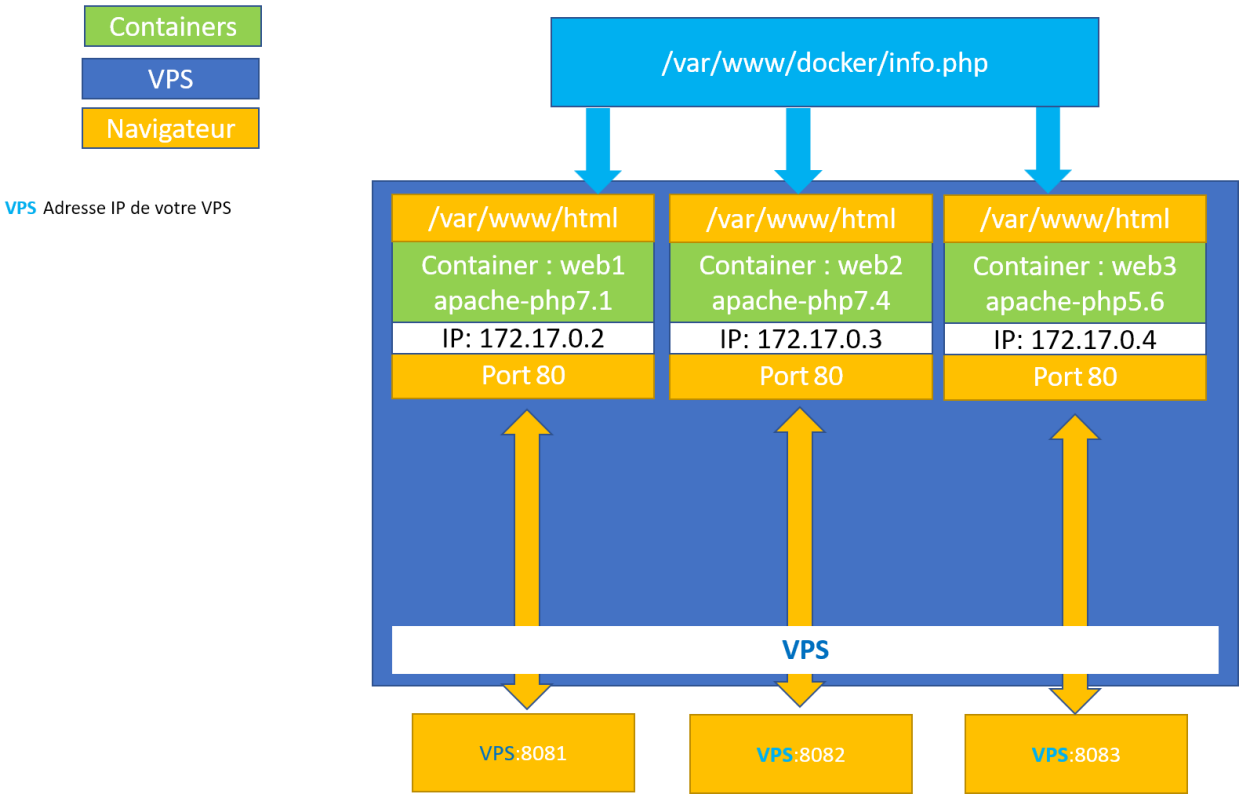
Si c'est le container qui initie la communication (comme pour le ping) pas de souci. La "**MASQUERADE**" va créer une table de translation des adresses qui permettra de router la réponse externe vers le bon container et donc d'établir une session de communication complétement transparente pour le poste situé sur Internet (Le serveur DNS de Google que vous "pinguer" ne sait pas qu'il répond à un container)

Mais si c'est le poste externe qui initie la communication et cherche donc à atteindre un service tournant sur un container que sa passe-t-il ? Prenons un exemple simple, vous créez un container qui embarque un site web et donc expose son port 80 sur le réseau 172.17.0.0. Le client (le navigateur) va donc émettre une requête sur le port 80 (http) vers l'adresse de votre VPS puisqu'il ne peut pas utiliser l'adresse privée du container comme cible (il ne la connait pas). D'après vous, quel service va répondre ?

Evidemment, c'est votre service Apache habituel qui va répondre , et pas le service web du container.
Une première solution serait de rediriger le port 80 de votre VPS vers celui du container, mais dans ce cas il serait maintenant impossible d'accéder à votre site de travail. Il va donc falloir associer d'autres ports aux services tournant sur les containers pour les différencier de vos sites web locaux. Rassurez-vous, encore une fois rien de très compliqué avec Docker, tout est prévu.

Pour illustrer cela, je vous propose de mettre en place 3 containers avec trois services web distincts qui afficheront la même page **info.php** . Cette page contiendra juste la fonction **phpinfo()**. Pour ne pas avoir à écrire 3 pages info.php , nous utiliserons aussi la notion de Volume de Docker qui permet de relier un répertoire local à celui d'un container.

Schéma fonctionnel :



Nos Objectifs :

Le port **8081** du VPS sera redirigé vers le port 80 du conteneur **web1** à l'IP 172.17.0.2
Le port **8082** du VPS sera redirigé vers le port 80 du conteneur **web2** à l'IP 172.17.0.3
Le port **8081** du VPS sera redirigé vers le port 80 du conteneur **web3** à l'IP 172.17.0.4
Les trois services web (web1, web2, web3) utiliserons des versions différentes de php mais interrogeront tous la même page **info.php** que nous créerons dans le dossier **/var/www/docker** de notre VPS.

IMPORTANT : En fonction de ce que vous avez fait (ou pas) dans les exercices précédents les IP peuvent être différentes. Cela n'a aucune importance, puisque votre navigateur n'a pas besoin de les connaitre. Il ne connait que l'IP (ou le nom DNS) de votre VPS. C'est le numéro des ports qui permet de distinguer les containers entre eux.

Pour créer les conteneurs nous utiliserons respectivement les images :

php:7.1-apache pour le container **web1**
php:7.4-apache pour le container **web2**
php:5.6-apache pour le container **web3**

Pour rediriger les ports nous exporterons respectivement **le port 80** de chaque conteneur vers :

le port **8081** de notre VPS pour le container **web1**
le port **8082** de notre VPS pour le container **web2**
le port **8083** de notre VPS pour le container **web3**

Pour lire la page **info.php** nous associerons le dossier **/var/www/docker** de notre VPS avec le dossier **/var/www/html** présent sur chaque container (*configuration apache par défaut*)

Pour commencer , créez-le dossier **/var/www/docke**r ainsi que la page **info.php** avec comme contenu :

```
<?php
phpinfo();
?>
```

Ensuite créons, notre premier container :

```
docker container run -d --name web1 -p 8081:80  -v /var/www/docker:/var/www/html  php:7.1-apache
```

-q : Pour lancer le container en Background
--name web1 : Pour nommer notre container
-p 8081:80 : Pour exporter le port 80 du container vers le port 8081 du VPS
-v /var/www/docker:/var/www/html :
Pour monter le dossier **/var/www/docker** de notre VPS dans le dossier **/var/www/html** du container.
php:7.1-apache est le nom de l'image utilisée pour générer le container

Procédez de même pour créer les deux autres containers en changeant le **nom du container** (**web1**,**web2**,**web3**), le **numéro du port** exporter (**8081**,**8082**,**8083**) et la **version de php** (**7.1**,**7.4**,**5.6**)

Vérifiez ensuite que vos trois containers sont bien actifs :

```
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                    NAMES
cad1233895e3   php:5.6-apache "docker-php-entrypoi..." 30 seconds ago  Up 29 seconds   0.0.0.0:8083->80/tcp     web3
af9a40db4704   php:7.4-apache "docker-php-entrypoi..." 12 minutes ago  Up 12 minutes   0.0.0.0:8082->80/tcp     web2
102ce41c2098   php:7.1-apache "docker-php-entrypoi..." 24 minutes ago  Up 24 minutes   0.0.0.0:8081->80/tcp     web1
```

et pour finir, vérifier avec votre navigateur que la page **info.php** est bien accessible par chacun des containers. Utilisez les URLs suivantes pour ouvrir la page :

MMI.h205.online:**8081**/info.php pour interroger le container **web1**
MMI.h205.online:**8082**/info.php pour interroger le container **web2**
MMI.h205.online:**8083**/info.php pour interroger le container **web3**

Vous devriez constater que chacun des sites utilisent bien une version de php différente comme le montre les copies d'écran ci-dessous :

Container web1 (port 8081)

Container web2 (port 8082)

Container web3 (port 8083)

Voilà un bon exemple d'utilisation de Docker : Tester une application sur plusieurs versions de PHP sans avoir besoin d'installer toutes les versions sur votre VPS.

avant de continuer, répondez aux questions suivantes :
Q11: Quelle version de Linux est installée dans le container web1 ?
Q12: Quelle version d'Apache est installée dans le container web2 ?
Q13: Quelle version d'Apache est installée dans le container web3 ?
Q14: Quelle est le nom de l'**interface réseau** créée par Docker sur votre VPS ?
Q15: Quelle est **l'adresse du réseau** IP créée par Docker sur votre VPS ?
Q16: Quelle **cible** est utilisée par **netfilter** pour faire du **NAT** sur l'IP des containers ?

Après avoir répondu aux questions, arrêtez et supprimez tous les containers que nous avons créés.

Avant de passer au dernier exercice, nous allons aussi faire un peu de ménage dans les images.
Commençons, par les lister :

```
docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
php	7.4-apache	80c24455118f	5 days ago	415MB
alpine	latest	f70734b6a266	2 weeks ago	5.61MB
hello-world	latest	bf756fb1ae65	4 months ago	13.3kB
php	7.1-apache	b9858ffdd4d2	5 months ago	401MB
php	5.6-apache	24c791995c1e	15 months ago	355MB

Vous remarquerez que la place prise par les images est assez conséquente, nous avons ici utilisé plus de 1Go de données sur notre disque. Remarquez aussi la différence de taille entre une distribution alpine (certes sans aucun service) et une distribution complète comme celles utilisées par les images de php.

Comme nous n'avons plus aucun container utilisant ces images, nous pouvons donc aussi les supprimer.

```
docker image rm $(docker image ls -q)
```

En résumé pour faire le ménage complet

```
docker container stop $(docker container ls -q)
docker container rm $(docker container ls -qa)
docker image rm $(docker image ls -q)
```

et pour arrêter docker :

```
sudo systemctl stop docker
```

Partie 4 : Exercice

En reprenant les concepts vus précédemment , installez un container nommé **cms** qui sera accessible sur le port **8085** de votre VPS et qui comme son nom l'indique affichera un site sous Wordpress. Ce site affichera simplement votre identifiant MMI, ainsi que vos nom et prénom sur la page d'accueil.

Votre démarche doit être la suivante :
Création du container : Le container doit :
- se nommer **cms** (et pas autrement)
- être construit sur l'image **wordpress:5.4.1-php7.4-apache**
- s'exécuter en tache de fond
- exposer son port **80** sur le port **8085** de votre VPS.

L'image Wordpress utilisée ne contient que les services apache et php, aucun moteur de base de données n'est installé. Nous verrons la prochaine fois comment connecter deux containers entre eux et donc comment utiliser un moteur de base de données présent dans une autre image.

Pour finaliser l'installation de Wordpress, **vous devez donc accéder au container en mode interactif**
- Installer mariadb-server (la distribution est une Debian 10)
- Démarrer le service avec service mysql start
- Créer une base de données et un utilisateur ayant les droits sur celle-ci .
  Utilisez les noms et mots de passe de votre choix, cela n'a aucune importance pour la correction

Reprenez les cours de S2 si nécessaire. Je ne répondrais à aucune question sur les points ci-dessus.
Une fois la base de données créée, vous devriez pouvoir vous connecter avec l'URL :
MMI.h205.online:8085 et terminer l'installation de wordpress.

ATTENTION : L'adresse de la base de données doit être 127.0.0.1 (pas localhost), le container ne sait pas résoudre le nom.

Pour finir l'exercice, créez votre site Wordpress (Titre et identifiant de votre choix)
Modifiez la page d'Exemple pour afficher votre Nom et Votre Prénom, ainsi que votre identifiant MMI.
Faites de cette page Exemple, la page d'accueil de votre site.

Après avoir vérifié que votre site fonctionne, vous pouvez stopper le container et docker avec les commandes :

```
docker container stop cms
sudo systemctl stop docker
```

IMPORTANT : Surtout ne supprimez pas le container, ni l'image après avoir terminé. Pour corriger, je réactiverai temporairement Docker et relancerai votre container dans son état avec les commandes :

```
sudo systemctl start docker
docker container start cms
docker container exec cms service mysql start
```

Et j'irai vérifier l'URL : **MMI.h205.online:8085** qui devra donc m'afficher votre site wordpress.