Premiers pas avec Docker

Table des matières

Objectifs

- Comprendre la technologie Docker, ses concepts et ses avantages
- Savoir installer Docker sur son serveur et le prendre en main
- Connaître les commandes de base pour manipuler des conteneurs

Introduction

Lors de ce module, nous allons tâcher de mettre en place Docker sur notre machine (ici une distribution GNU/Linux Debian). Nous allons tout d'abord étudier ce que Docker permet de faire, puis mettre en place l'outil sur notre machine, et enfin faire nos premiers pas avec cette technologie de plus en plus utilisée.

I Introduction à la conteneurisation

1. Présentation de Docker

Présentation de Docker

Docker permet de créer des environnements (appelés conteneurs, ou *containers* en anglais) de manière à isoler des applications. Docker repose sur le *kernel Linux* p.50 et sur une fonctionnalité : les *containers*, que vous connaissez peut-être déjà sous le nom de *LXC* p.50. L'idée est d'exécuter un processus (ou plusieurs) dans un *environnement isolé* p.50. Le conteneur qui accueille votre application aura un système de fichier qui lui est propre (pouvant contenir uniquement les fichiers nécessaires à l'application) et aura accès à certaines ressources du système hôte (mémoire, CPU, réseau). Avec Docker, on va pouvoir très simplement gérer nos différents conteneurs.

Mais en pratique, à quoi Docker sert vraiment ? Exemple

Un petit exemple d'utilisation très classique nous est donné sur le site de Red Hat

« Imaginons que vous êtes en train de développer une application. Vous travaillez sur un ordinateur portable dont l'environnement présente une configuration spécifique. D'autres développeurs peuvent travailler sur des machines qui présentent des configurations légèrement différentes. L'application que vous développez repose sur cette configuration et dépend de fichiers spécifiques. En parallèle, votre entreprise exploite des environnements de test et de production qui sont standardisés sur la base de configurations qui leur sont propres et de l'ensemble des fichiers associés. Vous souhaitez émuler ces environnements autant que possible localement, mais sans avoir à payer les coûts liés à la recréation des environnements de serveur. Comment faire pour que votre application en développement puisse fonctionner dans ces environnements, passer l'assurance qualité et être déployée sans prise de tête, sans réécriture et sans correctifs ? La réponse est simple : il vous suffit d'utiliser des conteneurs. Le conteneur qui accueille votre application contient toutes les configurations (et les fichiers) nécessaires. Vous pouvez ainsi le déplacer entre les environnements de développement, de test et de production, sans aucun effet secondaire. La crise est évitée, le travail peut continuer. »
(source *redhat.com* p.51)

Principes de Docker Az Définition

Docker permet de créer des conteneurs en se basant sur 3 caractéristiques principales : la reproductibilité, l'isolation et la portabilité. La **reproductibilité** nous évite le classique "works in my machine[1]" et nous permet d'avoir un même environnement en développement et en production, facilitant la mise en production des livrables. L'**isolation**, comme nous avons pu le lire plus haut, nous permet d'exécuter nos applications de manière indépendante les unes des autres. Enfin, la **portabilité** des conteneurs Docker nous permettra d'exécuter nos conteneurs de manière similaire dans des environnements physiques qui peuvent être très différents. Docker est une solution open-source, qui repose sur une grande communauté[2].

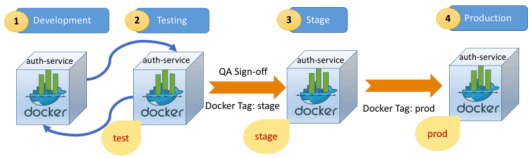
1. Works in my machine by leadingagile - https://www.leadingagile.com/2017/03/works-on-my-machine/
2. Communauté officielle Docker - https://www.docker.com/docker-community

Remarque

Le déploiement et l'orchestration font également partie du cœur de Docker, notamment avec des composantes comme *Compose* p.50 et *Swarm* p.50 . Nous étudierons Docker Compose dans une seconde partie, mais les concepts et le fonctionnement de Swarm ne sera pas abordé ici.

2. Pourquoi utiliser Docker ?

Pour les développeurs



Il est aujourd'hui facile pour un développeur d'utiliser un environnement de travail qui ne correspond pas à l'environnement de production final.

Par exemple, un développeur utilisant la version macOS de PHP n'exécutera probablement pas la même version que le serveur Linux hébergeant le code de production. Même si les versions sont identiques. Il faut alors gérer les différences de configuration et d'environnement de la version de PHP, (par exemple les permission sur les fichiers ne sont pas gérés de la même manière entre les deux systèmes d'exploitation).

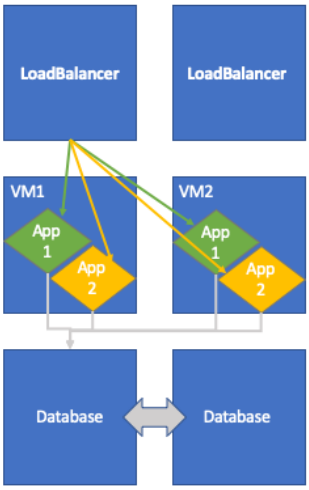
Lorsqu'un développeur doit déployer son code sur les machines de production et que cela ne fonctionne pas; L'environnement de production doit-il être configuré pour correspondre à la machine du développeur, ou les développeurs ne doivent-ils travailler que dans des environnements correspondant aux machines de productions ?

Dans un monde idéal, tout devrait être cohérent, de l'ordinateur du développeur aux serveurs de production. Cependant, cette utopie a toujours été difficile à réaliser. Tout le monde a sa propre façon de travailler et ses préférences personnelles. Il est déjà assez difficile d'appliquer la même cohérence sur plusieurs plates-

formes lorsqu'il s'agit d'un seul ingénieur travaillant sur ses propres systèmes, sans parler d'une équipe d'ingénieurs travaillant avec une équipe potentiellement composée de centaines de développeurs.

Docker est une solution à ces problèmes. En créant des "environnements" qui vont suivre le cycle de vie d'une application de son développement à sa mise en production.

Pour les administrateurs système



Prenons l'exemple d'un administrateur système possédant 6VMs pour ses différents services (base de données, serveurs web, load balancer).

Dans un premier temps, on lui demande de déployer sur les serveurs web une application qui dépend d'un logiciel quelconque de version 1. Tout se passe bien. Un jour, on lui demande de déployer un module supplémentaire de l'application qui nécessite quant à elle la version 2 du même logiciel. C'est alors qu'un problème se pose. Les applications ne peuvent pas cohabiter sur le même serveur si on les déploie de manière "classique".

Que peut-on faire ?

- Demander plus de serveurs ? C'est la solution la plus sûre, mais aussi la plus coûteuse. Il n'est pas toujours possible de déployer de nouvelles machines.

- Revoir l'architecture en déployant la seconde application sur un autre serveur existant (load balancer ou base de données). Encore une fois cette solution n'est pas optimale, en cas de panne sur un des composants, nous avons deux points de faute au lieu d'un si chaque VM avait son rôle précis.

- Essayer d'installer les deux versions du logiciel sur la même machine. C'est sûrement possible et ça semble être un bon "quick win". En revanche, cette solution est difficilement maintenable dans le temps. On peut évoquer l'application des patchs de sécurité qui vont être compliqués à effectuer.

Docker permet de faire tourner sur une même machine les deux applications et permet en plus de gérer plus facilement les mises à jour de chaque application de manière indépendante en recréant simplement de nouveaux conteneurs.

3. Différences entre conteneurs et VM

Docker isole un environnement comme une VM ?

Maintenant que l'on sait pourquoi Docker a été développé. Nous allons voir comment cela fonctionne sous le capot.

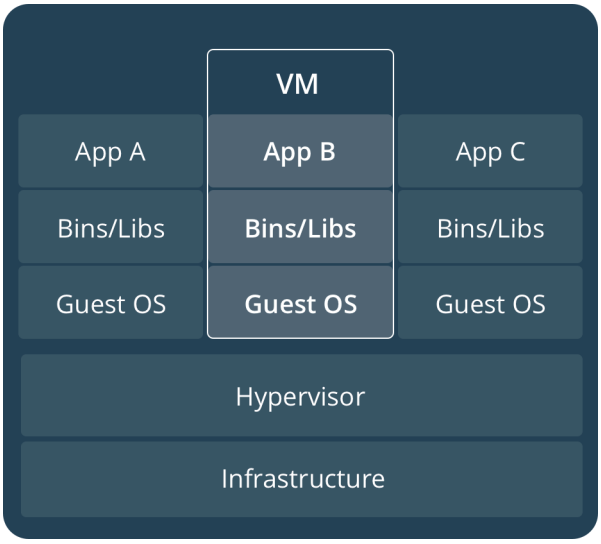
Docker n'est pas fait pour isoler les applications les unes des autres à la manière d'une VM. L'élément clé ici pour bien comprendre est la manière dont un conteneur s'exécute sur le serveur.

Une machine virtuelle, comment son nom l'indique, va simuler (virtualiser) une machine entière. C'est à dire simuler le matériel de la machine sur lequel sera exécuté un système d'exploitation complet (une distribution GNU/Linux, un Windows ou une version de Mac OS). Il n'y aura pas de réelle interaction avec le système hôte, qui se contentera de simuler un autre serveur, plus petit.

Dans le cas de Docker, l'hôte va mettre à disposition ses ressources pour le conteneur, mais celui-ci va directement utiliser des ressources de l'hôte (CPU, RAM, network, etc.). Le conteneur aura un environnement isolé et dédié (avec potentiellement des limitations en ressources), mais ne va pas simuler la totalité d'un serveur ni exécuter un système d'exploitation (et ses nombreux processus).

Avec Docker, il est commun d'imaginer un conteneur par fonctionnalité : base de donnée, service web, gestion des logs, etc. l'ensemble fonctionnant et communiquant ensemble.

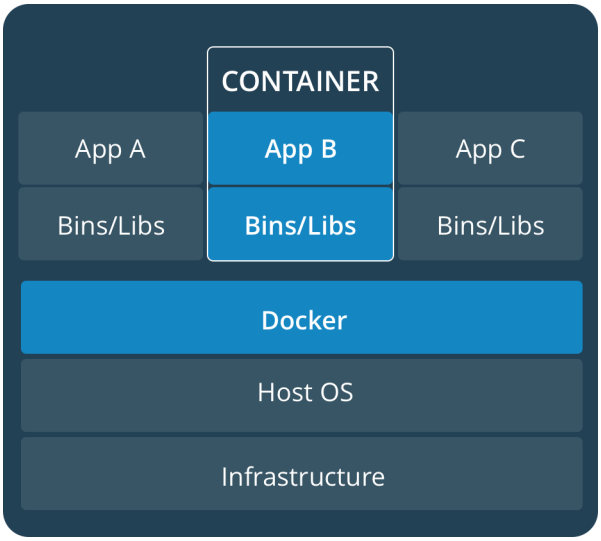
Finalement, Docker se présente comme une alternative plus légère que la mise en place de VM. Vous trouverez ci-dessous deux images illustrant leur mode de fonctionnement de base.



Virtual Machine

Comme on peut le voir sur ce schéma, dans le cadre d'une machine virtuelle classique, chaque VM possède son propre système d'exploitation (Guest OS) contenant lui-même des bibliothèques (bin/lib) qui sont répliquées pour chacune des machines virtuelles.

De fait, les machines virtuelles sont lourdes ; ajoutons à cela le fait qu'elles doivent simuler tous les composants physiques d"une machine. Toute cette architecture de virtualisation consomme beaucoup de ressources pour fonctionner.



Container

Ce schéma nous donne une bonne idée du principal avantage de Docker. Il n'est pas nécessaire de disposer d'un système d'exploitation complet chaque fois que nous devons créer un nouveau conteneur, ce qui réduit la taille globale des conteneurs.

Docker utilise le noyau Linux du système d'exploitation hôte (car presque toutes les versions de Linux utilisent les modèles de noyau standard) pour le système d'exploitation sur lequel il a été créé, telle que Debian, Ubuntu ou CentOS. Pour cette raison, vous pouvez utiliser presque n'importe quel système d'exploitation Linux en tant que système d'exploitation. Il est ainsi possible d'avoir un conteneur CentOS sur une machine Debian.

Un autre avantage de Docker est la taille des images lors de leur création. Ils ne contiennent pas le noyau. Cela les rend incroyablement petits, compacts et faciles à déployer.

4. Exercice : Conteneurisation ou virtualisation

[solution n°1 p. 37]

Exercice

Que permet la virtualisation ?

- **A** Faire tourner plusieurs systèmes d'exploitations différents sur une même machine.
- **B** Faire tourner plusieurs applications qui se partagent un même noyau.
- **C** Isoler l'exécution de mes applications.

Exercice

Que permet la conteneurisation ?

- **A** Faire tourner plusieurs systèmes d'exploitations différents sur une même machine.
- **B** Faire tourner plusieurs applications qui se partagent un même noyau.
- **C** Isoler l'exécution de mes applications.

II Prise en main de Docker

1. Installation de Docker sur Ubuntu

La documentation officielle en tant que base Remarque

Pour l'installation de Docker, nous allons simplement suivre la documentation officielle[3][4][5] de Docker pour **Ubuntu**, qui manque parfois de clarté mais est un bonne référence.[6]

Avant de commencer l'installation

Avant de commencer l'installation de Docker (ou de n'importe quel paquet d'ailleurs), il est de bonne pratique de mettre à jour la liste des paquets existants.

```
sudo apt update
```

Il va désormais être nécessaire d'ajouter les dépôts où se trouve Docker à la liste de nos dépôts, afin que la commande *apt* puisse aller les chercher. Installons tout d'abord les dépendances :

```
sudo apt install ca-certificates curl gnupg
```

Ajouter la clé GPG officielle du *repository* de Docker, afin de signer les paquets :

```
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Dernière étape : mettre en place le *repository* stable.

```
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Installer Docker

Après ces petites manipulations (que nous n'aurons heureusement pas à refaire), on peut enfin commencer à installer Docker

```
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Et voilà ! Docker est normalement installé sur notre machine. Confirmons le au plus vite.

Lançons notre premier conteneur !

```
sudo docker run hello-world
```

Cette commande va télécharger une image de test et l'exécuter dans un conteneur. Lorsque le conteneur sera en exécution, il va afficher un message d'information et quitter.

3. https://docs.docker.com/install/linux/docker-ce/debian/
4. https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository
5. https://docs.docker.com/install/linux/docker-ce/debian/
6. https://docs.docker.com/install/

```
root@dev:/home/dev# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:97ce6fa4b6cdc0790cda65fe7290b74cfebd9fa0c9b8c38e979330d547d22ce1
Status: Downloaded newer image for hello-world:latest

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

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/

root@dev:/home/dev#
```

Conseil

Par défaut les commandes Docker ne peuvent se lancer qu'avec les droits administrateur. C'est vite contraignant de devoir préfixer ses commandes par sudo. On va donc s'ajouter au groupe *docker*, ce qui nous permettra d'omettre *sudo* :

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

Puis déconnectez - reconnectez vous au serveur pour appliquer le changement de groupe.

Attention

Ne mettez dans le groupe *docker* que les utilisateurs à qui vous faites confiance. Il est très facile de se rendre root au travers du groupe *docker* sans pour autant être dans le groupe *sudo*.

2. Exercice : Premiers pas dans un conteneur

Notre premier conteneur

Maintenant que Docker est installé, nous allons pouvoir appréhender par la pratique les conteneurs.

Pour commencer nous allons simplement lancer un conteneur Debian, sur une ancienne version (ici Bullseye)

```
docker run -it debian:bullseye bash
```

On ne s'attarde pas trop sur tout les détails de la commande ici, la plupart seront expliqués par la suite. Mais brièvement :

- `docker run` est la commande Docker pour démarrer un conteneur
- `debian:bullseye` indique que l'on souhaite démarrer un conteneur à partir de l'image Debian en version Bullseye
- `bash` est le processus que l'on souhaite lancer dans le conteneur. Ici Bash nous permet simplement d'avoir un shell dans le conteneur, pour y lancer des commandes

Une fois que la commande est lancée, on se retrouve à l'intérieur de notre conteneur. Nous allons faire quelques manipulations pour constater l'isolation apportée par le conteneur vis à vis de la machine hôte.

Système de fichiers

On peut, dans le conteneur, regarder le contenu du fichier `/etc/debian_version`, qui indique simplement la version de Debian du système. Son contenu devrait être `11.XX`.

Si on regarde le même fichier sur la machine, le résultat devrait être différent (`12.XX` si vous êtes sous Debian Bookworm). Le système de fichiers du conteneur est donc bien celui d'une distribution Debian Bullseye, différent de celui de l'hôte.

De plus, en parcourant l'arborescence dans le conteneur, on ne retrouve pas les fichiers du système hôte, et inversement. Le conteneur a son système de fichiers propre et indépendant de l'hôte.

Kernel

Si le système de fichiers est différent entre le conteneur et l'hôte, les processus de notre conteneur s'exécutent bien sur la machine hôte.

Question 1

[solution n°2 p. 37]

Qu'est ce que l'on constate lorsque l'on lance `uname -r` dans le conteneur et sur le serveur ? Que cela signifie ?

Indice :

La commande `uname -r` retourne la version du noyau (ici du noyau Linux) du système.

Processus

Une autre différence majeure avec les machines virtuelles est que l'on ne fait pas s'exécuter tout un système d'exploitation dans le conteneur.

Question 2

[solution n°3 p. 37]

Combien de processus s'exécutent dans le conteneur ? Lesquels ?

Indice :

On utilise la commande `ps aux` pour lister les processus d'une machine.

Indice :

Les images docker ayant pour philosophie d'être les plus légères possible, ps n'est pas encore installé. Il faudra donc mettre les mirroirs à jour et installer le paquet *procps* depuis l'intérieur du container.

Namespaces

Si les processus sont isolés, cela ne veut pas dire pour autant qu'il ne s'exécutent pas sur le système hôte.

Ils s'exécutent dans ce que l'on appelle un *namespace*. On ne rentrera pas dans le détail, mais un *namespace* est un des mécanismes utilisé par Docker pour mettre en place des conteneurs. Les processus d'un *namespace* ne verront pas les processus des autres *namespaces*. Par contre tous utilisent le même noyau système : celui de l'hôte.

Une seconde manipulation permet de valider cela : dans le conteneur, on va lancer un processus. Pour l'exemple on va simplement lancer `sleep 60`, un processus qui ne va rien faire pendant 60 secondes. En parallèle, on va lister les processus sur le système hôte.

Question 3

[solution n°4 p. 37]

Quelle commande utiliser sur le système hôte pour vérifier que le processus `sleep` du conteneur est bien visible dans la liste des processus du système ? Qu'est ce que l'on observe ?

Indice :

La commande `grep` permet de filtrer du texte.

Conclusion

À l'issue de cette première expérimentation avec Docker, on a vu ce que signifie l'isolation lorsque l'on utilise des conteneurs. Bien que ces expérimentations soient très basiques, il est important de les comprendre, et surtout de bien intégrer leurs conclusions (système de fichiers différent, processus isolés). Sans cela, il sera difficile de comprendre les concepts de Docker qui seront introduits dans la suite du cours.

3. Images Docker et registry

Images Docker

Dans les exercices précédents, nous avons vu que le système de fichiers dans notre conteneur était différent de celui de l'hôte. Une question se pose alors : d'où vient ce système de fichier ? Tout simplement de ce que l'on appelle une image Docker.

Une image Docker est un *template* de système de fichiers qui sera utilisé pour démarrer un conteneur. Au démarrage le conteneur va récupérer le système de fichiers de l'image Docker pour l'utiliser, un peu de la même manière qu'une image ISO qui va servir à démarrer un système d'exploitation.

Il est très important de ne pas confondre les concepts de conteneur et d'image. L'image n'est rien de plus qu'un *template*, un système de fichiers. Le conteneur, lui, utilise une image en tant que *template* pour se lancer. D'une certaine manière, le conteneur est une instance de l'image, et plusieurs conteneurs peuvent être lancés en se basant sur la même image.

Une image Docker est immutable, elle n'est pas modifiable par un conteneur, son système de fichiers est figé. Celui du conteneur, à l'inverse, est créé au démarrage du conteneur (à partir de l'image) mais peut-être modifié par le conteneur et est éphémère : il est complètement supprimé lorsque l'on arrête le conteneur.

Registry Docker

Docker permet de créer des images qui seront utilisées par nos conteneurs. Ceci sera l'objet d'un prochain cours, pour le moment nous utiliserons seulement des images existantes.

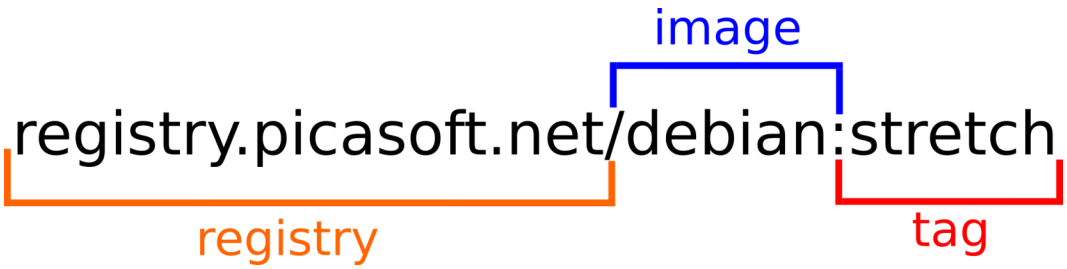
Nous pouvons trouver des images existantes dans le Docker Hub, qui est une *registry* Docker. Une *registry* est un serveur qui va tout simplement stocker des images Docker, un peu à la manière d'un serveur Git qui stocke des *repositories*.

Bien que n'importe qui puisse maintenir sa propre *registry*, l'entreprise Docker Inc. fournit une *registry*, qui s'appelle le Docker Hub[7], dans laquelle on retrouve un très grand nombre d'images déjà existantes. N'importe qui peut pousser une image dans cette *registry*, mais certaines sont officiellement maintenues par Docker Inc. et/ou par les mainteneurs de logiciel ou de distribution Linux. Parmi ces images officielles, on peut trouver :

- des images relativement basique pour les différents systèmes d'exploitations (Debian, Ubuntu, Centos, Alpine, etc.)
- des images adaptées pour lancer des programmes dans un langage particulier (Python, Go, Node.js, etc.)
- des images qui encapsulent un logiciel comme un serveur Web (Apache, Nginx) ou un SGBD (PostgreSQL, MariaDB)

Identification des images

Pour les identifier, les images Docker ont un nom complet qui a la forme suivante.



La première partie correspond à l'adresse de la *registry* Docker. Dans le cas d'images venant du Docker Hub, cette première partie est absente (car le Docker Hub est la *registry* par défaut).

La seconde partie est le véritable nom de l'image.

La dernière partie est le *tag* de l'image. Le *tag* peut-être vu comme un moyen de versionner une image Docker. Lorsque le *tag* n'est pas spécifié, c'est le *tag* `latest` qui est utilisé. *`latest`* est un *tag* par défaut, qui est normalement utilisé pour pointer vers la dernière version de l'image.

Dans nos premiers pas avec Docker, c'est une image officielle (maintenue par des développeurs Debian) de Debian que nous avons utilisé. Le conteneur avait été démarré avec le paramètre `debian:bullseye`. C'est donc l'image `debian`, venant du Docker Hub, en version `bullseye`.

4. Exercice : Images Docker et registry

[solution n°5 p. 38]

Registry ?

Qu'est-ce qu'une *registry* Docker ?

- **A** Un logiciel qui me permet de créer des images Docker
- **B** Un serveur qui héberge des images Docker

7. https://hub.docker.com

- **C** Un serveur qui sert de registre pour enregistrer tout les utilisateurs de Docker

Un peu de recherche

En cherchant sur le Docker Hub, à quel version de Debian correspond le *tag* `latest` actuel ?

Image Docker ?

Une image Docker :

- **A** est la même chose qu'un conteneur Docker.
- **B** peut-être modifiée, par exemple si je créé un fichier dans mon conteneur.
- **C** permet de créer plusieurs conteneurs identiques.

Identifier une image - 1

L'image Docker `registry.picasoft.net/pica-mattermost:7.10.0`

- **A** Vient du Docker Hub
- **B** A pour nom d'image `pica-mattermost`
- **C** A pour *tag* `7.10.0`

Identifier une image - 2

L'image `bitnami/postgresql:latest`

- **A** Vient du Docker Hub
- **B** A pour nom `bitnami/postgresql`
- **C** A pour tag `latest`

III Docker en pratique

Introduction

Il est temps de rentrer dans le vif du sujet et de voir les principales commandes et les principaux mécanismes à connaître lorsque l'on veut utiliser des conteneurs Docker.

Toutes les commandes Docker sont des sous-commandes que l'on appelle avec `docker`. Pour connaître la totalité des commandes possibles et leurs options, il est possible d'utiliser `docker help` ou `docker COMMANDE help` mais aussi de consulter la documentation très complète[8]. Au fur et à mesure que nous allons avancer, il ne faut pas hésiter à aller lire plus en détail la documentation des commandes qui seront lancées.

1. Manipuler les images

Précédemment nous avons vu que les conteneurs avaient besoin d'une image Docker pour démarrer et que ces images sont stockées dans une *registry*. Cependant Docker conserve en local une copie des images nécessaires pour faire tourner les conteneurs. Pour lister les images présentes sur la machine, on utilise la commande suivante :

```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
debian              bullseye            bb64860610f6        2 weeks ago         127MB
```

La commande nous retourne, sous forme de table, la liste des images. Pour chaque image nous avons les informations suivantes :

- le nom de l'image
- le tag de l'image
- un identifiant unique pour l'image Docker
- la date de création de l'image
- la taille de l'image

Dans notre cas il n'y a que l'image `debian:bullseye`. Elle a été téléchargée automatiquement lorsque nous avons démarré le conteneur dans les exercices précédents. Pour la suite, nous allons utiliser une image plus récente de Debian, l'image `debian:bookworm`. Pour télécharger cette image, on utilise simplement la commande `docker pull NOM_IMAGE`. Comme l'image de *bullseye* ne sera plus utile, il est tout simplement possible de la supprimer avec la commande `docker image rm NOM_IMAGE`.

2. Manipuler les conteneurs

Démarrer un conteneur

Dans nos premiers pas, nous avons lancé un conteneur avec la commande `docker run`. C'est celle qui nous permet de démarrer un conteneur à partir d'une image. Instinctivement on pourrait lancer :

```
docker run debian:bookworm
```

Le résultat est cependant assez décevant, globalement rien ne se passe. En effet cette commande lance simplement un conteneur à partir de l'image `debian:bookworm`, mais on ne lui a rien demandé de plus.

---

8. https://docs.docker.com/engine/reference/commandline/cli/

Lorsque l'on lance un conteneur, il convient de spécifier 2 informations supplémentaires :

- le processus que l'on souhaite démarrer dans le conteneur
- si l'on souhaite interagir avec le processus dans le conteneur ou si l'on veut le lancer en arrière-plan

Concernant le processus, nous avions précédemment lancé le processus `bash` grâce à la commande `docker run debian:bookworm bash`. Le second paramètre de la commande `docker run` est la commande que l'on souhaite lancer dans le conteneur. Cependant, la plupart des images ont une commande par défaut à lancer, et dans le cas de `debian:bookworm` c'est la commande `bash` (il est possible d'obtenir cette information avec la commande `docker image inspect debian:bookworm`, mais nous ne nous attarderons pas là dessus).

La seconde information à spécifier est le mode de lancement du conteneur. Par défaut la commande `docker run` ne connecte pas le terminal de l'utilisateur dans le conteneur, pour cela on passe généralement 2 options :

- `-i` qui permet d'attacher le terminal à l'entrée standard du processus que l'on lance

- `-t` qui permet d'allouer un TTY au conteneur

Ces 2 options sont généralement passées ensemble et elles permettent de "connecter" son terminal au conteneur (au processus lancé dans le conteneur plus précisément).

En conclusion, pour refaire les manipulations du début de module, on peut lancer notre conteneur avec la commande :

```
1 docker run -it debian:bookworm
```

Cependant il est aussi possible de lancer un conteneur en mode détaché, c'est d'ailleurs le cas le plus fréquent. Par exemple si l'on veut faire tourner un service web et une base de données en arrière-plan sur le serveur. Pour cela on utilise simplement l'option `-d`. Par exemple la commande `docker run -d debian:bookworm sleep 10` va lancer, en arrière plan, un processus `sleep 10` dans un conteneur. Une fois le processus terminé le conteneur s'arrêtera.

Gérer les conteneurs

Maintenant que l'on sait démarrer nos conteneurs, il serait intéressant de pouvoir contrôler leur cycle de vie. En effet un conteneur Docker peut-être dans plusieurs états :

- *running* indique que le conteneur est en cours d'exécution
- *exited* indique le conteneur a terminé son exécution
- *paused* indique que le conteneur est mis en pause dans son exécution
- d'autres états de transition (*created*, *restarting*, *removing*) ou d'erreur (*dead*) qui ne seront pas approfondis ici

Lorsque l'on lance la commande `docker run`, Docker va d'abord créer un conteneur à partir de l'image choisie, puis le démarrer pour le mettre dans un état *running*. Quand le processus à l'intérieur du conteneur aura terminé de s'exécuter, le conteneur va s'arrêter et deviendra *exited*.

Sur notre machine, on peut lister tout les conteneurs existants avec la commande `docker container ls -a`.

Pour chaque conteneur on obtient les informations suivantes :

- un identifiant unique pour le conteneur
- l'image Docker utilisée par ce conteneur
- la commande (ou le processus) lancée dans le conteneur
- la date de création du conteneur

- son statut et le temps depuis lequel il est dans ce statut
- le nom du conteneur

Premier constat : Docker a donné des noms aux conteneurs alors que nous ne lui avons rien demandé ! En effet Docker génère automatiquement un nom (selon un algorithme[9] plutôt rigolo) pour chaque nouveau conteneur, mais il est possible (et d'ailleurs fortement recommandé) de nommer les conteneurs autrement. Pour cela on utilisera l'option `--name NOM_CONTENEUR` au lancement de `docker run`.

Deuxième constat : tout les conteneurs que nous avons lancé jusqu'ici sont listés, même si ils sont arrêtés. En effet l'option `-a` permet de lister la totalité des conteneurs (sans cela seuls ceux qui sont démarrés seraient listés). On peut d'ailleurs supprimer rapidement les conteneurs en statut *Exited* avec la commande `docker container rm NOM_CONTENEUR`, pour éviter que la liste ne grandisse trop au fur et à mesure de nos manipulations. Il est aussi possible d'indiquer à Docker de supprimer automatiquement un conteneur lorsque celui-ci se termine, grâce à l'option `--rm` dans la commande `docker run` (nous avions utilisé cette option au tout début de la prise en main des conteneurs).

Stopper, suspendre et lancer un processus dans un conteneur

Avec Docker il est possible de changer l'état de son conteneur manuellement. Par exemple démarrer un conteneur qui est arrêté, on utilisera tout simplement `docker start NOM_CONTENEUR`. De la même manière c'est avec `docker stop` que l'on peut stopper le conteneur prématurément.

Une autre action est aussi possible : mettre en "pause" un conteneur. Avec `docker pause`, le processus dans le conteneur verra son exécution suspendue. La commande `docker unpause` permet de relancer le processus qui reprendra son exécution dans le même état que lors de son interruption. C'est une fonctionnalité qui peut s'avérer intéressante lorsque l'on veut sauvegarder des données qui sont manipulées par le conteneur (par exemple dans le cas d'une base de donnée) et donc bloquer les *I/O* très temporairement.

Enfin, il est possible de lancer des commandes directement dans un conteneur en cours d'exécution, en parallèle du processus courant. Ceci est possible grâce à la commande `docker exec -it NOM_CONTENEUR COMMANDE`, elle est très souvent utilisé lorsqu'il est nécessaire d'ouvrir un *shell* dans un conteneur. Prenons l'exemple d'un conteneur qui exécute une instance de base de donnée : PostgreSQL. On peut très simplement démarrer un conteneur à partir de l'image officielle `postgres` qui démarre une instance PostgreSQL :

Le conteneur démarre en mode détaché et l'instance PostgreSQL est démarrée. Si on souhaite accèder au SGBD afin de créer une base de données (par exemple) on peut tout simplement

lancer "docker exec -it postgresql psql -U postgres".

Ceci va simplement lancer la commande "psql -U postgres" à l'intérieur du conteneur, nous permettant de faire ensuite des manipulations (par exemple créer une BDD avec "CREATE DATABASE").

```
1 docker run -d --name postgresql -e
  POSTGRES_PASSWORD=votresupermotdepassepourlabasededonnee postgres
```

Le conteneur démarre en mode détaché et l'instance PostgreSQL est démarrée. Si on souhaite accéder au SGBD afin de créer une base de données (par exemple) on peut tout simplement lancer :

```
1 docker exec -it postgresql psql -U postgres
```

Ceci va simplement lancer la commande `psql -U postgres` à l'intérieur du conteneur, nous permettant de faire ensuite des manipulations (par exemple créer une BDD avec CREATE DATABASE).

9. https://github.com/moby/moby/blob/master/pkg/namesgenerator/names-generator.go

Conclusion

Nous venons d'aborder, de manière un peu condensée, les commandes et mécanismes de bases de Docker. Il est particulièrement important de les maîtriser, et il peut-être judicieux de prendre le temps de manipuler quelques conteneurs (éventuellement tester des images présentes sur le Docker Hub) pour s'assurer que ces bases sont acquises.

3. Exercice : Exercice - Manipulation de conteneurs

[solution n°6 p. 39]

Docker run

Que fait la commande `docker run --name test -d nginx`

- **A** Elle lance un conteneur basé sur l'image `nginx` en arrière plan.
- **B** Elle lance le processus `bash` dans un conteneur.
- **C** Elle lance le processus par défaut de l'image `nginx` dans un conteneur.
- **D** Elle ouvre un *shell* interactif dans un conteneur basé sur l'image `nginx`.

État d'un conteneur

Le résultat de la commande `docker container ls -a` est le suivant :

```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
eb561ad59a86        nginx               "nginx -g 'daemon of…"   5 seconds ago       Up 4 seconds                0.0.0.0:80->80/tcp  musing_germain
7378fdaf1c88        debian:stretch      "bash"                   11 seconds ago      Exited (0) 10 seconds ago                       dazzling_goldwasser
71b33d993a20        postgres            "docker-entrypoint.s…"   36 seconds ago      Up 34 seconds               5432/tcp            sad_herschel
7a7c17757b92        debian:stretch      "bash"                   45 seconds ago      Exited (0) 44 seconds ago                       quirky_darwin
```

Dans quel état se trouve le conteneur `quirky_darwin` ?

- **A** Arrêté
- **B** En exécution
- **C** Supprimé

Image d'un conteneur

Le résultat de la commande `docker container ls -a` est le suivant :

```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
eb561ad59a86        nginx               "nginx -g 'daemon of…"   5 seconds ago       Up 4 seconds                0.0.0.0:80->80/tcp  musing_germain
7378fdaf1c88        debian:stretch      "bash"                   11 seconds ago      Exited (0) 10 seconds ago                       dazzling_goldwasser
71b33d993a20        postgres            "docker-entrypoint.s…"   36 seconds ago      Up 34 seconds               5432/tcp            sad_herschel
7a7c17757b92        debian:stretch      "bash"                   45 seconds ago      Exited (0) 44 seconds ago                       quirky_darwin
```

Quelle est l'image utilisée par le conteneur `dazzling_goldwasser` ?

Stopper un conteneur

Le résultat de la commande `docker container ls -a` est le suivant :

```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
eb561ad59a86        nginx               "nginx -g 'daemon of…"   5 seconds ago       Up 4 seconds                0.0.0.0:80->80/tcp  musing_germain
7378fdaf1c88        debian:stretch      "bash"                   11 seconds ago      Exited (0) 10 seconds ago                       dazzling_goldwasser
71b33d993a20        postgres            "docker-entrypoint.s…"   36 seconds ago      Up 34 seconds               5432/tcp            sad_herschel
7a7c17757b92        debian:stretch      "bash"                   45 seconds ago      Exited (0) 44 seconds ago                       quirky_darwin
```

Quelle commande lancer pour stopper le conteneur basé sur l'image `nginx` ?

Renommage

Je viens de lancer la commande suivante :

```
$ docker run -d postgres
2c8f63cd308def6d1ebeab8b6b2ec3a074304c4843c5521817ed43c3b46a1592
```

Docker a choisi un nom aléatoire pour mon conteneur mais je souhaite le renommer. En cherchant sur Internet et/ou dans la documentation de Docker, quelle est la commande complète que je peux utiliser pour renommer mon conteneur en `mabdd`.

4. Conteneur et système de fichiers

Jusqu'à présent notre utilisation des conteneurs a été rudimentaire. Une fonctionnalité qui va nous ouvrir de premières possibilités intéressantes est le partage de fichiers entre le conteneur et le système hôte.

Imaginons que nous voulions lancer un script Python très classique qui est donné si dessous :

```
import sys
def fibo(n):
  if n==1 or n==2: return 1
  return fibo(n-1) + fibo(n-2)
print( fibo(int(sys.argv[1])) )
```

C'est un script très basique (et mal écrit) qui va simplement calculer afficher le nombre de Fibonnaci correspondant à un entier. Pas besoin de connaître le Python, nous allons simplement vouloir l'exécuter sans installer Python sur notre machine. Pour cela 2 possibilités :

1. on copie manuellement ce script dans notre conteneur
2. on utilise un point de montage pour partager ce script avec notre conteneur

Il est évident que la solution 1 n'est pas viable. À chaque fois que l'on créé un nouveau conteneur, celui-ci démarre sur le système de fichier de l'image qu'il va utiliser. Ce qui veut donc dire que, à chaque fois que l'on va redémarrer le conteneur, il sera nécessaire de copier le script manuellement. Nous allons donc utiliser une fonctionnalité intéressante de Docker : les *bind mounts*. C'est un mécanisme très simple qui permet de partager un dossier du système hôte avec le conteneur, grâce à l'option `-v /chemin/sur/hote:/chemin/dans/conteneur` au démarrage d'un conteneur.

Ici imaginons que le script Python ci-dessus soit placé sur notre serveur dans le dossier *~/scripts*. On peut lancer la commande suivante pour exécuter notre script à l'aide de l'image Python en version 3 :

```
docker run -it -v ~/scripts:/code python:3 python /code/fibo.py 16
```

Cette commande va lancer un conteneur (en mode interactif) et monter le contenu du dossier *~/scripts* dans le dossier */code* du conteneur. Puis le conteneur va lancer la commande `python /code/fibo.py 16` qui permet simplement d'appeler le script avec le paramètre 16.

> 💬 Remarque
>
> Les bind-mounts fonctionnent aussi bien sur les dossiers que les fichiers, mais il faut alors préciser leurs noms des deux côtés.

5. Exercice : Système de fichiers

L'objectif de cet exercice sera de compiler un script basique (un *Hello World*) dans le langage de programmation Go, avec Docker. Il n'est absolument pas nécessaire de connaître le langage Go, il faut simplement savoir que c'est un langage compilé.

Nous avons le script suivant, qui affiche simplement "Hello World", à l'emplacement */root/scripts/hello.go* sur notre hôte :

```
1 package main
2
3 import "fmt"
4
5 func main() {
6   fmt.Println("Hello world")
7 }
```

Le but est de pouvoir exécuter ce script sur notre machine hôte, après avoir compilé le script, sans avoir à installer un compilateur Go sur la machine. Sachant que :

- une image `golang` existe sur le Docker Hub et embarque les utilitaires pour compiler du Go
- la commande `go build -o mon_binaire mon_source.go` permet de compiler le script *mon_source.go* en un binaire *mon_binaire*

Question

[solution n°7 p. 40]

Donnez la commande Docker qui permet de générer un binaire, à partir du script ci-dessus, à l'emplacement /root/binaries/hello_world sur la machine hôte.

6. Exercice : Environnement du conteneur

Variable d'environnements

Un autre mécanisme très intéressant avec Docker, est de pouvoir positionner des variables d'environnement dans le conteneur. Ces variables seront donc accessibles pour le processus qui est lancé, ce qui est très pratique si l'on veut passer de la configuration à notre conteneur.

L'utilisation des variables d'environnement pour configurer une application dans un conteneur est d'ailleurs une bonne pratique à adopter et fait partie des recommandation du 12factor app[10]

Pour cela on utilise simplement l'option `-e VARIABLE="valeur"` lors du démarrage du conteneur. On peut lister les variables d'environnement dans un conteneur pour vérifier que la variable qui est passée se retrouve bien dans notre conteneur :

```
1 $ docker run -it -e TEXT="Bonjour" debian env
2 PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
3 HOSTNAME=4989de4edf56
4 TERM=xterm
5 TEXT=Bonjour
6 HOME=/root
```

10. https://12factor.net/fr/

Mise en pratique

Le cas d'utilisation le plus courant est la configuration d'un processus à partir de variables d'environnements. En reprenant la documentation de l'image[11] de PostgreSQL utilisée précédemment on remarque qu'il est possible de créer automatiquement une base de donnée avec un utilisateur à l'aide des variables suivantes :

- `POSTGRES_DB` : nom de la base de donnée à créer
- `POSTGRES_USER` : nom de l'utilisateur à créer ayant les accès sur la base de données
- `POSTGRES_PASSWORD` : mot de passe de l'utilisateur

Question 1

[solution n°8 p. 40]

À l'aide des informations précédentes, donnez la commande complète permettant de démarrer un conteneur PostgreSQL en créant une base de donnée *picasoft* avec un utilisateur *picasoft* et le mot de passe *thisissimplepassword*.

Question 2

[solution n°9 p. 40]

Donnez une commande pour valider que l'on peut se connecter à la base de donnée avec l'utilisateur *picasoft* depuis le conteneur.

7. Conteneur et réseau

Dans l'exercice précédent, nous avons démarré une instance PostgreSQL dans un conteneur. Cependant cette base de données ne pouvait être utilisée que en local dans le conteneur.

C'est assez limitant, d'autant plus que la philosophie de Docker veut que l'on segmente les différents services dans des conteneurs séparés : un serveur Web dans un conteneur, une base de données dans un autre, etc.

Docker place les différents conteneurs dans des réseaux privés virtuels isolés, sur la machine hôte. Par exemple si l'on regarde l'adresse IP d'un conteneur :

```
1 docker run -it debian bash
2 root@26d068da63e7:/# apt update && apt install -y iproute2 && ip a
3 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
  qlen 1000
4     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
5     inet 127.0.0.1/8 scope host lo
6       valid_lft forever preferred_lft forever
7 33: eth0@if34: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
  group default
8     link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff link-netnsid 0
9     inet 172.17.0.4/16 brd 172.17.255.255 scope global eth0
10      valid_lft forever preferred_lft forever
```

On constate que le conteneur a une interface sur un réseau *172.17.0.0/16*. Ce réseau est un réseau par défaut que Docker a créé. C'est un réseau de type *bridge*, c'est à dire que le conteneur a accès à internet via la machine hôte, et la machine hôte peut accéder au conteneur en passant par le réseau virtuel de Docker.

Connecter le conteneur à un port de l'hôte

Cependant il est possible de connecter un (ou plusieurs) port spécifique du conteneur à un port de la machine hôte : le processus dans le conteneur aura ainsi directement accès à la couche réseau de la machine hôte pour le port en question. Pour cela on utilise l'option `-p PORT_HOTE:PORT_CONTENEUR` au démarrage du conteneur.

11. https://hub.docker.com/_/postgres

Par exemple, on lance un conteneur Nginx avec :

```
docker run -d --name nginx-1 -p 8080:80 nginx
```

Lorsque l'on ouvre son navigateur et que l'on navigue sur la machine hôte, sur le port 8080, on constate la page par défaut de Nginx. Le port 8080 de la machine hôte redirige automatiquement vers le port 80 du conteneur (sur lequel écoute le serveur Nginx).

En cherchant dans la documentation de Docker et sur internet, trouvez une commande permettant d'afficher en direct les logs d'accès sur votre serveur Nginx.

Connecter des conteneurs dans un même réseau

Cependant, il n'est pas forcément recommandé de publier tout les ports de tout les services de notre machine. Par exemple il est judicieux de ne pas exposer une base de données directement sur internet lorsque ce n'est pas nécessaire, mais plutôt de la placer dans un réseau privé auquel les clients (typiquement les serveurs web) auront accès. Docker permet cela grâce à la possibilité de créer des réseaux Docker et d'y connecter des conteneurs.

Pour commencer, on peut créer notre premier réseau avec `docker network create myfirstnet`. Il sera ensuite possible de placer des conteneurs dans ce réseau, avec l'option `--net myfirstnet`, qui pourront communiquer entre eux. D'ailleurs Docker permet à des conteneurs d'un même réseau de communiquer entre eux en se basant sur leurs noms (à l'aide d'un DNS dynamique).

Par exemple nous pouvons lancer 2 conteneurs différents:

```
docker run -it --name container1 --net myfirstnet debian bash
docker run -it --name container2 --net myfirstnet debian bash
```

Dans le premier conteneur on peut contacter le second, sur son IP privée, grâce à son nom DNS *container2*

```
$ docker run -it --name container1 --net myfirstnet debian bash
root@26d068da63e7:/# apt update && apt install -y iputils-ping
root@26d068da63e7:/# ping container2
PING container2 (172.18.0.3) 56(84) bytes of data.
64 bytes from container2.myfirstnet (172.18.0.3): icmp_seq=1 ttl=64 time=0.160 ms
64 bytes from container2.myfirstnet (172.18.0.3): icmp_seq=2 ttl=64 time=0.107 ms
64 bytes from container2.myfirstnet (172.18.0.3): icmp_seq=3 ttl=64 time=0.109 ms
64 bytes from container2.myfirstnet (172.18.0.3): icmp_seq=4 ttl=64 time=0.108 ms
64 bytes from container2.myfirstnet (172.18.0.3): icmp_seq=5 ttl=64 time=0.068 ms

--- container2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4090ms
rtt min/avg/max/mdev = 0.068/0.110/0.160/0.030 ms
```

8. Exercice : Bilan de la mise en pratique de Docker

Préparation

À ce stade des nos manipulations, il est fort probable que la commande `docker ps -a` retourne un grand nombre de conteneurs sur votre machine. Avant de passer à un petit exercice de mise en pratique, il convient de supprimer tout les conteneurs existants, qui ne servent plus. Pour cela on peut simplement lancer :

```
docker rm -vf $(docker ps -a -q)
```

Sans rentrer dans les détails de la commande, elle va simplement lister les identifiants de touts les conteneurs et les passer en argument à une commande qui va les supprimer.

Base de données dans Docker

À l'aide des connaissances acquises et de la documentation[12] de l'image `postgres` sur le Docker Hub, nous avons le nécessaire pour déployer une base de données PostgreSQL avec Docker sur la machine, tout en garantissant son isolement réseau et la persistance des données.

Question 1

[solution n°10 p. 41]

Donnez les commandes permettant de déployer une base de données PostgreSQL :

- se trouvant sur un réseau privé Docker nommé `pg-net`
- persistant les données de la base sur le disque (les données ne doivent pas être perdues si l'on re-créé le conteneur) dans un dossier */root/bdd*
- avec un utilisateur *snowden* ayant un accès à une base *prism* avec le mot de passe *nsa*

Question 2

[solution n°11 p. 41]

Donnez une commande simple à lancer pour valider que la base de donnée est bien accessible dans le réseau `pg-net`.

Indice :

On peut utiliser un autre conteneur dans le même réseau pour se connecter à la base de données.

12. https://hub.docker.com/_/postgres

IV Utilisation de Docker Compose

Objectifs

- Découvrir les fonctionnalités et cas d'usage de Docker Compose
- Comprendre et apprendre à manipuler les commandes nécessaires afin de déployer un ensemble de conteneurs

1. Présentation de Docker Compose

Problèmes du CLI Docker

La philosophie d'utilisation de Docker veut que l'on favorise la segmentation des différentes applications d'un système dans différents conteneurs, qui communiquent entre-eux, plutôt que la mise en place d'un conteneur qui ferait tourner tout les composants d'une application.

Au fur et à mesure des précédents exercices, de nombreuses options possibles sont venues s'ajouter à la commande `docker run` et on se rend vite compte que la manipulation de nombreux conteneurs de cette manière peut devenir fastidieuse pour plusieurs raisons :

- tout les conteneurs sont créés manuellement, il faut donc se souvenir de la totalité des options si l'on souhaite recréer le conteneur (par exemple si l'image a été mise à jour)
- lorsqu'il y a des dépendances entre les conteneurs (par exemple une application web qui nécessite une base de données) il est nécessaire de connaître le bon ordre de lancement de chaque conteneurs. Lorsque l'on parle d'application d'une dizaine de conteneurs cela devient compliqué relativement complexe.

Docker Compose

Docker Compose est un outil qui permet de résoudre cette problématique.

La base de Docker Compose est le fichier (généralement appelé *docker-compose.yml*) qui permet de décrire l'ensemble des conteneurs que l'on souhaite faire tourner sur le serveur. C'est un fichier écrit en YAML,[13] une format de représentation de données.

À titre d'exemple, un fichier Docker Compose ressemble à ceci (il n'est pas nécessaire de le comprendre bien entendu) :

```yaml
version: '3'
services:
  traefik:
    image: traefik:latest
    container_name: traefik
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /DOCKER/volumes/traefik/traefik.toml:/traefik.toml
      - /DOCKER/volumes/traefik/certs:/certs
  registry:
    image: registry:2
    container_name: registry
    environment:
      REGISTRY_AUTH: htpasswd
      REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd
```

13. https://fr.wikipedia.org/wiki/YAML

```
19      volumes:
20        - /DOCKER/volumes/registry/data:/var/lib/registry
21        - /DOCKER/volumes/registry/auth:/auth
22      labels:
23        - "traefik.frontend.rule=Host:registry.picasoft.net"
24        - "traefik.port=5000"
25        - "traefik.enable=true"
```

Les fichiers Docker Compose permettent ainsi de décrire un groupe de conteneurs, qui pourront être manipulés (démarrés, arrêtés, etc.) en même temps.

2. Le fichier de Docker Compose

La base du fichier Docker Compose

Le fichier de Docker Compose (généralement, et pour la suite de ce cours, nommé *docker-compose.yml*) a systématiquement 2 clefs à sa racine : `version` et `services`. En effet le format du fichier Docker Compose a changé au cours du temps (nous en sommes à la version 3) et il est donc utile de préciser le format du fichier courant. Le reste du fichier sera consacré à la description des différents services (entendre ici *conteneurs*) que nous voulons gérer.

En prenant pour exemple le fichier précédent, son squelette est :

```
1 version: '3'
2 services:
3   traefik:
4     [...]
5
6   registry:
7     [...]
```

Ce fichier utilise le format Docker Compose en version 3 et décrit 2 services :

- `traefik`
- `registry`

La configuration de chacun des services sera ensuite décrites à l'intérieur de chacun des blocs.

Correspondances entre la CLI Docker et le format Docker Compose

Pour chaque service, différentes options peuvent être configurées dans le fichier Docker Compose. La liste est disponible dans la (très complète) documentation de Docker Compose[14] que nous vous incitons à garder dans un coin. Nous n'allons pas entrer dans le détail des options, mais simplement décrire celles qui nous permettant de remplacer les options de la CLI que nous avons vu dans les parties précédentes.

- `container_name` permet de nommer le conteneur (équivalent à `--name`)
- `environment` est la liste des variables d'environnement à passer au conteneur (équivalent à `--environment`)
- `image` est le nom de l'image Docker à utiliser
- `ports` est la liste des ports que l'on souhaite connecter entre l'hôte et le conteneur
- `volumes` est la liste des volumes que l'on souhaite monter dans le conteneur. Cette notion n'a pas encore été vue, mais sachez que les *bind mounts* sont une forme de volume et trouvent leur place dans cette liste.

14. https://docs.docker.com/compose/compose-file/#service-configuration-reference

Avec ces informations, on peut désormais interpréter les informations des services du fichier *docker-compose.yml* précédent. Par exemple le service `traefik` était configuré ainsi :

```
traefik:
  image: traefik:latest
  container_name: traefik
  ports:
    - "80:80"
    - "443:443"
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    - /DOCKER/volumes/traefik/traefik.toml:/traefik.toml
    - /DOCKER/volumes/traefik/certs:/certs
```

Ici le service `traefik` va créer un conteneur nommé `traefik` à partir de l'image `traefik:latest`. Le port 80 du conteneur va être exposé sur le port 80 de l'hôte, de même pour le port 443. De plus 3 *bind mounts* permettent de monter, dans le conteneur, les fichiers */var/run/docker.sock* et */DOCKER/volumes/traefik/traefik.toml* ainsi que le dossier */DOCKER/volumes/traefik/certs*. On constate que la syntaxe de chaque option respecte celle de l'outil en ligne de commande de Docker.

De la même manière, on peut analyser le service `registry` :

```
registry:
  image: registry:2
  container_name: registry
  environment:
    REGISTRY_AUTH: htpasswd
    REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd
  volumes:
    - /DOCKER/volumes/registry/data:/var/lib/registry
    - /DOCKER/volumes/registry/auth:/auth
  labels:
    - "traefik.frontend.rule=Host:registry.picasoft.net"
    - "traefik.port=5000"
    - "traefik.enable=true"
```

Un conteneur nommé `registry` est créé à partir de l'image `registry:2`. On monte 2 dossiers de l'hôte en *bind mounts* et on passe 2 variables d'environnement au conteneur : `REGISTRY_AUTH` (valeur *htpasswd*) et `REGISTRY_AUTH_HTPASSWD_PATH` (valeur */aut/htpasswd*). On assigne aussi 3 labels au conteneur. Les labels sont une notion que nous n'avons pas abordé, il faut simplement savoir qu'il est possible de mettre des labels (un couple clef/valeur) aux conteneurs (avec le mot clef *labels*).

Gestion des dépendances

Un intérêt de Docker Compose est aussi de gérer les dépendances entre conteneur. Dans l'exemple de fichier qui nous prenons, le service `registry` nécessite que le service `traefik` soit démarré pour fonctionner correctement. Pour cela on peut utiliser la clef `depends_on` qui prend en paramètre la liste des services dont dépend un conteneur. Par exemple, dans le cas de notre service `registry`, on pourrait ajouter :

```
registry:
  depends_on:
    - traefik
  [...]
```

Gestion des réseaux

Docker Compose permet aussi de connecter différents conteneurs à un même réseau.

Reprenons l'exemple que nous avons vu plus tôt dans ce cours. Nous avions créé 2 conteneurs avec les commandes suivantes :

```
docker run -it --name container1 --net myfirstnet debian bash
docker run -it --name container2 --net myfirstnet debian bash
```

Avec Docker Compose, on peut attacher les conteneurs à un réseau à l'aide de la clef `networks` dans un service. De plus il faut rajouter une autre clef `networks` à la racine du fichier docker-compose.yml pour décrire les différents réseaux. En pratique :

```
version: '3'
services:
  firstcontainer:
    image: debian
    container_name: container1
    networks:
      - myfirstnet
  secondcontainer:
    image: debian
    container_name: container2
    networks:
      - myfirstnet

networks:
  myfirstnet:
    external: true
```

On constate que, à la fin du fichier, il est nécessaire de définir tout les réseaux qui sont utilisés dans le fichier. Ici l'ajout de `external : true` permet d'indiquer à Docker Compose que le réseau Docker nommé `myfirstnet` existe déjà sur la machine (nous l'avions créé précédemment). Si ce n'est pas spécifié, Docker Compose va créer directement un réseau mais avec un nom qui n'est pas forcément exactement celui que l'on souhaite (un mélange entre le nom du dossier du fichier et le nom du réseau).

C'est donc une bonne pratique de systématiquement créer ses réseaux manuellement (avec `docker network create`) et d'utiliser la directive `external`.

3. Commandes de Docker Compose

Démarrer les services

Nous avons vu comment créer un fichier Docker Compose, mais sans utiliser la commande associée, il ne sert pas à grand chose. Eh bien cette commande c'est `docker compose`. Elle-ci permet de faire plusieurs choses, en particulier les mêmes manipulations que la commande `docker`, mais pour un groupe de conteneur.

Pour démarrer les services d'un fichier *docker-compose.yml* (c'est le fichier par défaut qui est lu par `docker compose`) on peut simplement lancer la commande `docker compose up`. Par exemple sur un fichier Docker Compose très basique (qui démarre simplement une *registry* Docker):

```
version: '3'
services:

  registry:
    image: registry:2
    container_name: registry
```

```
$ docker compose up
Creating registry ... done
Attaching to registry
registry    | time="2019-01-13T17:04:01.922076394Z" level=warning msg="No HTTP secret provided - generated random secret. This may cause problems with uploads if multiple registries are behind a load-balancer. To provide a shared secret, fill in http.secret in the configuration file or set the REGISTRY_HTTP_SECRET environment variable." go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
registry    | time="2019-01-13T17:04:01.922448703Z" level=info msg="Starting upload purge in 27m0s" go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
registry    | time="2019-01-13T17:04:01.922633856Z" level=info msg="redis not configured" go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
registry    | time="2019-01-13T17:04:01.932827813Z" level=info msg="using inmemory blob descriptor cache" go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
registry    | time="2019-01-13T17:04:01.933478222Z" level=info msg="listening on [::]:5000" go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
```

On peut constater que la commande `docker compose` nous attache par défaut à la sortie de notre conteneur (on voit ici les logs de la *registry*). Comme pour la commande `docker run`, on peut simplement rajouter l'option `-d` pour lancer les conteneurs en arrière plan.

On peut donc relancer la commande précédente avec cette option, et éventuellement accéder aux logs de notre conteneur par la suite :

```
$ docker compose up -d
Starting registry ... done
$ docker compose logs
Attaching to registry
registry    | time="2019-01-13T17:04:01.922076394Z" level=warning msg="No HTTP secret provided - generated random secret. This may cause problems with uploads if multiple registries are behind a load-balancer. To provide a shared secret, fill in http.secret in the configuration file or set the REGISTRY_HTTP_SECRET environment variable." go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
registry    | time="2019-01-13T17:04:01.922448703Z" level=info msg="Starting upload purge in 27m0s" go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
registry    | time="2019-01-13T17:04:01.922633856Z" level=info msg="redis not configured" go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
registry    | time="2019-01-13T17:04:01.932827813Z" level=info msg="using inmemory blob descriptor cache" go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
registry    | time="2019-01-13T17:04:01.933478222Z" level=info msg="listening on [::]:5000" go.version=go1.11.2 instance.id=286f7beb-a74d-47e4-9c88-625a6984bd7f service=registry version=v2.7.0
registry    | time="2019-01-13T17:08:11.030997995Z" level=warning msg="No HTTP secret provided - generated random secret. This may cause problems with uploads if multiple registries are behind a load-balancer. To provide a shared secret, fill in http.secret in the configuration file or set the REGISTRY_HTTP_SECRET environment variable." go.version=go1.11.2 instance.id=ad81362b-2e78-4039-b21c-9b8d837948c8 service=registry version=v2.7.0
registry    | time="2019-01-13T17:08:11.031106558Z" level=info msg="redis not configured" go.version=go1.11.2 instance.id=ad81362b-2e78-4039-b21c-9b8d837948c8 service=registry version=v2.7.0
registry    | time="2019-01-13T17:08:11.031174581Z" level=info msg="Starting upload purge in 52m0s" go.version=go1.11.2 instance.id=ad81362b-2e78-4039-b21c-9b8d837948c8 service=registry version=v2.7.0
registry    | time="2019-01-13T17:08:11.04644768Z" level=info msg="using inmemory blob descriptor cache" go.version=go1.11.2 instance.id=ad81362b-2e78-4039-b21c-9b8d837948c8 service=registry version=v2.7.0
registry    | time="2019-01-13T17:08:11.046715056Z" level=info msg="listening on [::]:5000" go.version=go1.11.2 instance.id=ad81362b-2e78-4039-b21c-9b8d837948c8 service=registry version=v2.7.0
```

Filtrer les actions

Avec la commande `docker compose`, il est possible de spécifier en paramètre le ou les services qui sont concernés par l'action. Par défaut tout les services du fichier *docker-compose.yml* sont pris en compte. Par exemple on ajoute un service très simple à notre *docker-compose.yml* précédent.

```yaml
version: '3'
services:
  registry:
    image: registry:2
    container_name: registry
  traefik:
    image: traefik
    container_name: traefik
```

On peut ne démarrer que ce service `traefik` avec la commande :

```
docker compose up -d traefik
```

Changer l'état des conteneurs

De la même manière que la commande `docker`, il est possible de changer l'état des conteneurs de notre fichier avec la commande `docker compose`. Sans revenir en détail sur leurs effets (les mêmes que avec la commande `docker`, mais sur un groupe de conteneurs), les actions suivantes sont possibles :

- `docker compose start` qui démarre les conteneurs (si ils avaient été arrêtés)
- `docker compose stop` qui stoppe l'execution des conteneurs
- `docker compose restart` qui permet de redémarrer les conteneurs qui sont actuellement en fonctionnement

Manipulations

Maintenant que nous savons écrire un fichier Docker Compose et que l'on connaît les commandes de bases, vous pouvez essayer de reproduire, via Docker Compose certains exercices précédemment réalisés dans ce cours.

V Exercices bilan

Objectifs

Tout au long de ce module, nous avons étudié les mécanismes de bases de Docker et Docker Compose, permettant de créer et gérer des conteneurs Docker facilement sur une machine. À travers 2 exercices, il s'agit désormais de valider, par la pratique, que les bases de Docker ont été appréhendées.

1. Exercice : Mise sous Docker Compose

Dans la première partie de ce cours, nous avions démarré une base de donnée PostgreSQL dans un conteneur, sur un réseau spécifique. Pour rappel la commande était la suivante :

```
1 docker run -d --name pg-test -e POSTGRES_DB=prism -e POSTGRES_USER=snowden -e
  POSTGRES_PASSWORD=nsa -v /root/bdd:/var/lib/postgresql/data --net pg-net postgres
```

Question

[solution n°12 p. 41]

À l'aide de cette commande et des connaissances acquises sur Docker Compose, écrivez un fichier Docker Compose qui permette de démarrer ce service. On considère que le réseau `pg-net` existe déjà sur l'hôte.

2. Exercice : "Hello world" web

Pour clôturer ce cours nous allons simplement mettre en place un serveur Web sur notre serveur. À l'aide des connaissances acquises, et en utilisant Docker Compose, il est demandé de fournir un fichier *docker-compose.yml* permettant de démarrer un serveur web avec les critères suivants :

- le serveur utilisé doit être Apache (image `httpd`[15]) ou Nginx (image `nginx`[16]). La documentation pour utiliser ces images est présente sur le Docker Hub
- le serveur doit répondre sur le port 1194 de la machine hôte
- la racine doit simplement afficher le message "`Hello world !`" au client Web
- le conteneur doit porter le nom `web`
- le ou les fichiers qui affichent le Hello World doivent être présent dans le dossier */root/web* de la machine hôte

Question

[solution n°13 p. 42]

Donnez le fichier Docker Compose à utiliser

3. Exercice : Multisite avec Docker 1.0

Nginx + PHP-FPM + Database multisite

L'objectif de cet exercice est de lancer 3 groupes de conteneurs contenant à la fois un reverse proxy Nginx, un conteneur en charge de l'interprétation des fichiers PHP et une base de données pour stocker des données.

Un reverse proxy sera également présent pour rediriger les conteneurs vers chacune des stacks.

15. https://hub.docker.com/_/httpd
16. https://hub.docker.com/_/nginx

Pour cela, nous allons utiliser un fichier docker-compose permettant de regrouper les différents sites.

Question 1

[solution n°14 p. 42]

Créez un premier fichier docker-compose composé d'un conteneur Nginx, d'une base de données PostgresSQL et d'un conteneur PHP-FPM.

On utilisera respectivement les images docker nginx, postgres et pichouk/php (qui contient les bibliothèque php postgresql pour fonctionner avec la base de données).

Vous utiliserez des volumes pour monter les fichiers de configuration dans les conteneurs et pour partager les fichiers PHP entre le conteneur Nginx et le conteneur PHP-FPM.

Les conteneurs utiliseront un network appelé backend et le conteneur nginx exposera son port 80 sur le port 80 de votre hôte docker

Indice :

Ci-dessous un example de fichier PHP appelant la base de données

`./html/index.php`

```php
<?php
    echo "<h1>Dix commandes très utiles sur Ubuntu</h1></br>";
    echo "<ol>";
    $connexion = new PDO('pgsql:host=postgresql;port=5432;dbname=prism', 'snowden', 'nsa');
    $sql = 'SELECT * FROM commande';
    $results = $connexion->prepare($sql);
    $results->execute();
    while ($row = $results->fetch(PDO::FETCH_ASSOC)){
        echo "<li><b>" . $row['com'] . "</b> : ";
        echo $row['def'] . "</li>";
    }
    echo "</ol>";
?>
```

Indice :

Vous devez créer un fichier de configuration Nginx qui va lire un fichier d'index PHP et demander son traitement à votre conteneur PHP

`./nginx/site1.conf`

```
server{
        listen 80;

        server_name _;
        root /usr/share/nginx/html;
        index index.php index.html;

        location / {
                try_files $uri $uri/ =404;
        }


        location ~ \.php$ {
          try_files $uri =404;
          fastcgi_split_path_info ^(.+\.php)(/.+)$;
          fastcgi_pass php:9000;
          fastcgi_index index.php;
          include fastcgi_params;
          fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
          fastcgi_param PATH_INFO $fastcgi_path_info;
```

```
        }

}
```

Question 2

[solution n°15 p. 43]

Nous allons maintenant passer par un conteneur nginx qui sera en charge de faire reverse proxy vers notre application. Celui-ci se trouvera dans un réseau appelé `front`

Indice :

Là encore nous allons devoir monter un fichier de configuration pour nginx afin de lui indiquer vers quel conteneur il faut rediriger les requêtes provenant d'un nom de domaine.

L'idée est de rediriger toutes les requêtes provenant du nom de domaine `monsite1.fr` vers notre conteneur appelé web.

`./nginx/proxy.conf`

```
server {
  listen 80;
  server_name monsite1.fr;
  location / {
        proxy_pass http://web:80;
    }
}
```

Indice :

Le conteneur proxy doit se trouver dans les deux networks pour être en mesure de rediriger le trafic vers les conteneurs de notre application.

Notre conteneur Nginx applicatif n'a plus besoin d'exposer son port sur l'hôte Docker, c'est le reverse proxy qui va s'en charger.

Question 3

[solution n°16 p. 44]

Nous allons maintenant ajouter une seconde application. Celle-ci sera elle aussi composée d'un conteneur Nginx, d'un conteneur PHP-FPM et d'une base de données PostgreSQL indépendante.

Créez les fichiers de configuration correspondante et mettez à jour le docker-compose.yml pour prendre en compte ce nouveau site.

Indice :

Les conteneurs de cette nouvelle application vont se trouver dans un network séparé de la première application qui sera appelé `backend2`

Indice :

Notre fichier index.php a été mis à jour pour notre nouvelle application. On notera que l'URL de la base de données a été mise à jour pour utiliser une autre base de données que celle de notre première application.

`./html2/index.php`

```
<?php
    echo "<h1>Dix commandes très utiles sur Ubuntu v2</h1></br>";
    echo "<ol>";
    $connexion = new PDO('pgsql:host=postgresql2;port=5432;dbname=prism', 'snowden',
'nsa');
    $sql = 'SELECT * FROM commande';
    $results = $connexion->prepare($sql);
    $results->execute();
    while ($row = $results->fetch(PDO::FETCH_ASSOC)){
```

```
        echo "<li><b>" . $row['com'] . "</b> : ";
        echo $row['def'] . "</li>";
    }
    echo "</ol>";
?>
```

Indice :

Vous devez créer un fichier de configuration Nginx qui va lire un fichier d'index PHP et demander son traitement à votre conteneur PHP. Cette fois-ci, nous allons changer de serveur PHP en modifiant la ligne `fastcgi_pass`

`./nginx/site2.conf`

```
server{
        listen 80;

        server_name _;
        root /usr/share/nginx/html;
        index index.php index.html;

        location / {
                try_files $uri $uri/ =404;
        }


        location ~ \.php$ {
          try_files $uri =404;
          fastcgi_split_path_info ^(.+\.php)(/.+)$;
          fastcgi_pass php2:9000;
          fastcgi_index index.php;
          include fastcgi_params;
          fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
          fastcgi_param PATH_INFO $fastcgi_path_info;
        }

}
```

Indice :

Nous devons également ajouter une nouvelle entrée à notre reverse proxy. Les clients arrivant avec l'url http://monsite2.fr seront alors redirigés vers notre nouvelle application (hébergée dans le conteneur web2) tout en laissant la première application fonctionner normalement.

```
server {
  listen 80;
  server_name monsite1.fr;
  location / {
        proxy_pass http://web:80;
    }
}
server {
  listen 80;
  server_name monsite2.fr;
  location / {
        proxy_pass http://web2:80;
    }
}
```

Question 4

[solution n°17 p. 45]

De la même manière ajouter une troisième application composée d'un Nginx, d'un PHP-FPM et d'une base de données PostgreSQL.

N'oubliez pas de l'ajouter à votre reverse proxy.

Solutions des exercices

Solution n°1

[exercice p. 10]

Exercice

Que permet la virtualisation ?

A Faire tourner plusieurs systèmes d'exploitations différents sur une même machine.

B Faire tourner plusieurs applications qui se partagent un même noyau.

Dans le cas d'une machine virtuelle, le système d'exploitation dans son ensemble est virtualisé, noyau inclus.

C Isoler l'exécution de mes applications.

Exercice

Que permet la conteneurisation ?

A Faire tourner plusieurs systèmes d'exploitations différents sur une même machine.

B Faire tourner plusieurs applications qui se partagent un même noyau.

C Isoler l'exécution de mes applications.

Solution n°2

[exercice p. 13]

La valeur retournée dans le conteneur et sur le serveur est la même car les processus dans le conteneur utilisent le noyau de l'hôte.

Solution n°3

[exercice p. 13]

Dans le conteneur, seul le processus `bash` s'exécute (il est aussi possible que l'on voit le processus associé à la commande `ps` que l'on a lancé) et il a comme PID 1.

Si on le souhaite on peut lancer d'autres processus dans le conteneur, ils seront eux aussi isolés et ne verront pas les processus du système hôte.

Solution n°4

[exercice p. 14]

On utilise `ps aux | grep sleep`. C'est à dire que l'on liste tout les processus et on redirige la sortie de la commande sur une seconde commande (avec le *pipe* `|`) qui va faire le filtre sur le mot *sleep*. On constate que notre processus `sleep`, qui s'exécute dans le conteneur, est bien visible au niveau du système d'exploitation hôte.

Solution n°5

[exercice p. 15]

Registry ?

Qu'est-ce qu'une *registry* Docker ?

- **A** Un logiciel qui me permet de créer des images Docker
- **B** Un serveur qui héberge des images Docker
- **C** Un serveur qui sert de registre pour enregistrer tout les utilisateurs de Docker

Un peu de recherche

En cherchant sur le Docker Hub, à quel version de Debian correspond le *tag* `latest` actuel ?

bookworm

Sur le Docker Hub, la page d'une image comporte généralement la liste des tags qui sont utilisés pour cette image. Dans le cas de l'image Debian, une même image peut avoir plusieurs *tags* différents (par exemple ici la *tag* `latest` qui désigne aussi `bookworm`).

Image Docker ?

Une image Docker :

- **A** est la même chose qu'un conteneur Docker. — L'image Docker est le *template* que le conteneur va utiliser pour démarrer.
- **B** peut-être modifiée, par exemple si je créé un fichier dans mon conteneur. — Pour modifier une image Docker, il faut la recréer. Un conteneur ne peux pas modifier l'image sur laquelle il se base, il ne peux manipuler que son propre système de fichiers.
- **C** permet de créer plusieurs conteneurs identiques.

Identifier une image - 1

L'image Docker `registry.picasoft.net/pica-mattermost:7.10.0`

- **A** Vient du Docker Hub — L'adresse de la *registry* de cette image est `registry.picasoft.net`
- **B** A pour nom d'image `pica-mattermost`

C

A pour *tag* `7.10.0` Le *tag* de l'image est la partie qui se trouve après les deux-points. Ici c'est `7.10.0`

Identifier une image - 2

L'image `bitnami/postgresql:latest`

A Vient du Docker Hub

B A pour nom `bitnami/postgresql`

C A pour tag `latest`

Solution n°6

[exercice p. 20]

Docker run

Que fait la commande `docker run --name test -d nginx`

A Elle lance un conteneur basé sur l'image `nginx` en arrière plan.

B Elle lance le processus `bash` dans un conteneur.

C Elle lance le processus par défaut de l'image `nginx` dans un conteneur.

D Elle ouvre un *shell* interactif dans un conteneur basé sur l'image `nginx`.

État d'un conteneur

Le résultat de la commande `docker container ls -a` est le suivant :

```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
eb561ad59a86        nginx               "nginx -g 'daemon of…"   5 seconds ago       Up 4 seconds                0.0.0.0:80->80/tcp  musing_germain
7378fdaf1c88        debian:stretch      "bash"                   11 seconds ago      Exited (0) 10 seconds ago                       dazzling_goldwasser
71b33d993a20        postgres            "docker-entrypoint.s…"   36 seconds ago      Up 34 seconds               5432/tcp            sad_herschel
7a7c17757b92        debian:stretch      "bash"                   45 seconds ago      Exited (0) 44 seconds ago                       quirky_darwin
```

Dans quel état se trouve le conteneur `quirky_darwin` ?

A Arrêté

B En exécution

C Supprimé

Image d'un conteneur

Le résultat de la commande `docker container ls -a` est le suivant :


```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
eb561ad59a86        nginx               "nginx -g 'daemon of…"   5 seconds ago       Up 4 seconds                0.0.0.0:80->80/tcp  musing_germain
7378fdaf1c88        debian:stretch      "bash"                   11 seconds ago      Exited (0) 10 seconds ago                       dazzling_goldwasser
71b33d993a20        postgres            "docker-entrypoint.s…"   36 seconds ago      Up 34 seconds               5432/tcp            sad_herschel
7a7c17757b92        debian:stretch      "bash"                   45 seconds ago      Exited (0) 44 seconds ago                       quirky_darwin
```

Quelle est l'image utilisée par le conteneur `dazzling_goldwasser` ?

debian

Stopper un conteneur

Le résultat de la commande `docker container ls -a` est le suivant :

```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS                   NAMES
eb561ad59a86        nginx               "nginx -g 'daemon of…"   5 seconds ago       Up 4 seconds                0.0.0.0:80->80/tcp      musing_germain
7378fdaf1c88        debian:stretch      "bash"                   11 seconds ago      Exited (0) 10 seconds ago                           dazzling_goldwasser
71b33d993a20        postgres            "docker-entrypoint.s…"   36 seconds ago      Up 34 seconds               5432/tcp                sad_herschel
7a7c17757b92        debian:stretch      "bash"                   45 seconds ago      Exited (0) 44 seconds ago                           quirky_darwin
```

Quelle commande lancer pour stopper le conteneur basé sur l'image `nginx` ?

docker stop musing_germain

Renommage

Je viens de lancer la commande suivante :

```
$ docker run -d postgres
2c8f63cd308def6d1ebeab8b6b2ec3a074304c4843c5521817ed43c3b46a1592
```

Docker a choisi un nom aléatoire pour mon conteneur mais je souhaite le renommer. En cherchant sur Internet et/ou dans la documentation de Docker, quelle est la commande complète que je peux utiliser pour renommer mon conteneur en `mabdd`.

docker rename 2c8f63cd308def6d1ebeab8b6b2ec3a074304c4843c5521817ed43c3b46a1592 mabdd

Solution n°7

[exercice p. 22]

On utilisera la commande suivante :

```
docker run -v /root/scripts:/code -v /root/binaries:/dist golang go build -o /dist/hello_world /code/hello_world.go
```

La commande va lancer un conteneur basé sur l'image `golang`. Dans le conteneur, 2 dossiers du système hôte sont montés : */root/scripts* est monté dans */code* (pour récupérer le fichier source) et */root/binaries* est monté dans */dist* (pour récupérer le fichier compilé). Dans le conteneur on lance la commande `go build -o /dist/hello_world /code/hello_world.go` qui permet de compiler notre fichier source et de déposer le binaire dans le dossier partagé.

Si l'exercice est réussi il est désormais possible, depuis la machine hôte, de lancer le binaire */root/binaries/hello_world* qui affiche un "Hello World"

```
$ /root/binaries/hello_world
Hello world
```

Solution n°8

[exercice p. 23]

On peut utiliser la commande :

```
docker run -d --name postgres -e POSTGRES_DB=picasoft -e POSTGRES_USER=picasoft -e POSTGRES_PASSWORD=thisissimplepassword postgres
```

Solution n°9

[exercice p. 23]

```
docker exec -it postgres psql -U picasoft
```

Solution n°10

[exercice p. 25]

Pour créer un réseau dédié, on utilise :

```
docker network create pg-net
```

Pour lancer notre conteneur, on utilise une commande similaire à celle-ci :

```
docker run -d --name pg-test -e POSTGRES_DB=prism -e POSTGRES_USER=snowden -e
POSTGRES_PASSWORD=nsa -v /root/bdd:/var/lib/postgresql/data --net pg-net postgres
```

On lance un conteneur basé sur l'image `postgres` et nommé `pg-test`. On configure la base de donnée et l'utilisateur à l'aide de variables d'environnements passées via l'option `-e`.

Pour assurer la persistance des données, on monte le dossier */root/bdd* dans le dossier */var/lib/postgresql/data*, là où se trouve les données de PostgreSQL.

Enfin, on connecte le conteneur dans notre réseau à l'aide de l'option `--net`

Solution n°11

[exercice p. 25]

On lance un conteneur (basé sur `postgres` pour avoir la commande *psql*) dans le même réseau Docker et l'on tente d'accéder à la base de données.

```
$ docker run -it --net pg-net postgres psql -h pg-test -U snowden prism
Password for user snowden:
psql (11.1 (Debian 11.1-1.pgdg90+1))
Type "help" for help.
prism=#
```

Après nous avoir demandé le mot de passe, on arrive à se connecter à la base de donnée que l'on fait tourner dans un conteneur isolé.

Solution n°12

[exercice p. 32]

Voici un fichier *docker-compose.yml* qui définit un service bdd qui va démarrer le conteneur de la même manière que la commande ci-dessus :

```
version: '3'
services:
  bdd:
    image: postgres
    container_name: pg-test
    environment:
      POSTGRES_DB: prism
      POSTGRES_USER: snowden
      POSTGRES_PASSWORD: nsa
    volumes:
      - /root/bdd:/var/lib/postgresql/data
    networks:
      - pg-net

networks:
  pg-net:
    external: true
```

Solution n°13

[exercice p. 32]

Dans le cas d'utilisation d'un serveur Apache, voici le fichier qui doit être utilisé.

```yaml
version: '3'
services:
  web:
    image: httpd
    container_name: web
    ports:
      - "1194:80"
    volumes:
      - /root/web:/usr/local/apache2/htdocs/
```

Dans le cas d'utilisation d'un serveur Nginx, voici le fichier qui doit être utilisé.

```yaml
version: '3'
services:
  web:
    image: nginx
    container_name: web
    ports:
      - "1194:80"
    volumes:
      - /root/web:/usr/share/nginx/html
```

Dans les deux cas on constate que les fichiers sont très similaires. Seuls les images et les points de montage des fichiers web dans le conteneur sont différents.

Solution n°14

[exercice p. 33]

```yaml
version: '3'
services:
  web:
    image: nginx:latest
    container_name: web
    ports:
        - "80:80"
    volumes:
        - ./html:/usr/share/nginx/html
        - ./nginx/site1.conf:/etc/nginx/conf.d/default.conf
    networks:
      - backend

  php:
    image: pichouk/php
    container_name: php
    volumes:
        - ./html:/usr/share/nginx/html
    networks:
      - backend

  postgresql:
    image: postgres:10
    container_name: postgresql
    environment:
      POSTGRES_DB: prism
      POSTGRES_USER: snowden
      POSTGRES_PASSWORD: nsa
    volumes:
      - ./bdd/data1:/var/lib/postgresql/data
```

```
    networks:
      - backend

networks:
  backend:
    external: true
```

Solution n°15

[exercice p. 34]

```
version: '3'
services:
  proxy:
    image: nginx:latest
    container_name: proxy
    ports:
        - "80:80"
    volumes:
        - ./nginx/proxy.conf:/etc/nginx/conf.d/default.conf
    networks:
      - front

  web:
    image: nginx:latest
    container_name: web
    volumes:
        - ./html:/usr/share/nginx/html
        - ./nginx/site1.conf:/etc/nginx/conf.d/default.conf
    networks:
      - front
      - backend

  php:
    image: pichouk/php
    container_name: php
    volumes:
        - ./html:/usr/share/nginx/html
    networks:
      - backend

  postgresql:
    image: postgres:10
    container_name: postgresql
    environment:
      POSTGRES_DB: prism
      POSTGRES_USER: snowden
      POSTGRES_PASSWORD: nsa
    volumes:
      - ./bdd/data1:/var/lib/postgresql/data
    networks:
      - backend

networks:
  front:
    external: true
  backend:
    external: true
```

Vous pouvez maintenant visiter le site correspondant à votre nom de domaine. Dans notre cas monsite1.fr et visualiser la même page web que précédemment

Solution n°16

[exercice p. 34]

```yaml
version: '3'
services:
  proxy:
    image: nginx:latest
    container_name: proxy
    ports:
        - "80:80"
    volumes:
        - ./nginx/proxy.conf:/etc/nginx/conf.d/default.conf
    networks:
      - front

  web:
    image: nginx:latest
    container_name: web
    volumes:
        - ./html:/usr/share/nginx/html
        - ./nginx/site1.conf:/etc/nginx/conf.d/default.conf
    networks:
      - front
      - backend

  php:
    image: pichouk/php
    container_name: php
    volumes:
        - ./html:/usr/share/nginx/html
    networks:
      - backend

  postgresql:
    image: postgres:10
    container_name: postgresql
    environment:
      POSTGRES_DB: prism
      POSTGRES_USER: snowden
      POSTGRES_PASSWORD: nsa
    volumes:
      - ./bdd/data1:/var/lib/postgresql/data
    networks:
      - backend

  web2:
    image: nginx:latest
    container_name: web2
    volumes:
        - ./html2:/usr/share/nginx/html
        - ./nginx/site2.conf:/etc/nginx/conf.d/default.conf
    networks:
      - front
      - backend2

  php2:
    image: pichouk/php
    container_name: php2
    volumes:
        - ./html2:/usr/share/nginx/html
    networks:
      - backend2

```

```
  postgresql2:
    image: postgres:10
    container_name: postgresql2
    environment:
      POSTGRES_DB: prism
      POSTGRES_USER: snowden
      POSTGRES_PASSWORD: nsa
    volumes:
      - ./bdd/data2:/var/lib/postgresql/data
    networks:
      - backend2

networks:
  front:
    external: true
  backend:
    external: true
  backend2:
    external: true
```

Vous pouvez maintenant visiter votre nouvelle application. Dans notre cas, monsite2.fr

Solution n°17

[exercice p. 36]

Ci-dessous l'arborescence finale de notre projet

```
root@docker:~# tree -L 2
.
├── bdd
│   ├── data1
│   ├── data2
│   └── data3
├── docker-compose.yml
├── html
│   └── index.php
├── html2
│   └── index.php
├── html3
│   └── index.php
└── nginx
    ├── proxy.conf
    ├── site1.conf
    ├── site2.conf
    └── site3.conf

8 directories, 8 files
```

Les fichiers de configurations nginx

```
# proxy.conf
server {
    listen 80;
    server_name monsite1.fr;
    location / {
        proxy_pass http://web:80;
    }
}
server {
    listen 80;
    server_name monsite2.fr;
    location / {
        proxy_pass http://web2:80;
    }
```

```
}
server {
    listen 80;
    server_name monsite2.fr;
    location / {
        proxy_pass http://web3:80;
    }
}
```

```
# site1.conf
server{
        listen 80;

        server_name _;
        root /usr/share/nginx/html;
        index index.php index.html;

        location / {
                try_files $uri $uri/ =404;
        }


        location ~ \.php$ {
          try_files $uri =404;
          fastcgi_split_path_info ^(.+\.php)(/.+)$;
          fastcgi_pass php:9000;
          fastcgi_index index.php;
          include fastcgi_params;
          fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
          fastcgi_param PATH_INFO $fastcgi_path_info;
        }

}
```

```
# site2.conf
server{
        listen 80;

        server_name _;
        root /usr/share/nginx/html;
        index index.php index.html;

        location / {
                try_files $uri $uri/ =404;
        }


        location ~ \.php$ {
          try_files $uri =404;
          fastcgi_split_path_info ^(.+\.php)(/.+)$;
          fastcgi_pass php2:9000;
          fastcgi_index index.php;
          include fastcgi_params;
          fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
          fastcgi_param PATH_INFO $fastcgi_path_info;
        }

}
```

```
# site3.conf
server{
        listen 80;

        server_name _;
```

```
        root /usr/share/nginx/html;
        index index.php index.html;

        location / {
                try_files $uri $uri/ =404;
        }


        location ~ \.php$ {
          try_files $uri =404;
          fastcgi_split_path_info ^(.+\.php)(/.+)$;
          fastcgi_pass php3:9000;
          fastcgi_index index.php;
          include fastcgi_params;
          fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
          fastcgi_param PATH_INFO $fastcgi_path_info;
        }

}
```

Ci-dessous le fichier docker-compose final :

```
version: '3'
services:
  proxy:
    image: nginx:latest
    container_name: proxy
    ports:
        - "80:80"
    volumes:
        - ./nginx/proxy.conf:/etc/nginx/conf.d/default.conf
    networks:
      - front

  web:
    image: nginx:latest
    container_name: web
    volumes:
        - ./html:/usr/share/nginx/html
        - ./nginx/site1.conf:/etc/nginx/conf.d/default.conf
    networks:
      - front
      - backend

  php:
    image: pichouk/php
    container_name: php
    volumes:
        - ./html:/usr/share/nginx/html
    networks:
      - backend

  postgresql:
    image: postgres:10
    container_name: postgresql
    environment:
      POSTGRES_DB: prism
      POSTGRES_USER: snowden
      POSTGRES_PASSWORD: nsa
    volumes:
      - ./bdd/data1:/var/lib/postgresql/data
    networks:
      - backend

```

```
  web2:
    image: nginx:latest
    container_name: web2
    volumes:
        - ./html2:/usr/share/nginx/html
        - ./nginx/site2.conf:/etc/nginx/conf.d/default.conf
    networks:
      - front
      - backend2

  php2:
    image: pichouk/php
    container_name: php2
    volumes:
        - ./html2:/usr/share/nginx/html
    networks:
      - backend2

  postgresql2:
    image: postgres:10
    container_name: postgresql2
    environment:
      POSTGRES_DB: prism
      POSTGRES_USER: snowden
      POSTGRES_PASSWORD: nsa
    volumes:
      - ./bdd/data2:/var/lib/postgresql/data
    networks:
      - backend2

  web3:
    image: nginx:latest
    container_name: web3
    volumes:
        - ./html3:/usr/share/nginx/html
        - ./nginx/site3.conf:/etc/nginx/conf.d/default.conf
    networks:
      - front
      - backend3

  php3:
    image: pichouk/php
    container_name: php3
    volumes:
        - ./html3:/usr/share/nginx/html
    networks:
      - backend3

  postgresql3:
    image: postgres:10
    container_name: postgresql3
    environment:
      POSTGRES_DB: prism
      POSTGRES_USER: snowden
      POSTGRES_PASSWORD: nsa
    volumes:
      - ./bdd/data3:/var/lib/postgresql/data
    networks:
      - backend3

networks:
  front:
    external: true
```

```
106  backend:
107    external: true
108  backend2:
109    external: true
110  backend3:
111    external: true
```

Glossaire

Docker Compose

Gestion de la composition et de l'orchestration de containers. En savoir plus[17].

Docker Swarm

Notamment utilisé pour le groupage de services en grappe (ou en anglais cluster). En savoir plus[18].

Environnement Isolé

Dans un environnement isolé, chaque appel aux ressources natives du système sont virtualisées (ici, par Docker). En d'autres termes, Docker va **créer des contextes pour chaque conteneurs dans lesquels les processus tourneront**. De ce fait, si un piratage est effectué sur une instance de l'application ou si une instance plante, le système reste sauf et isolé. Il suffit ainsi de terminer l'instance et en remonter une nouvelle.

Kernel ou Noyau Linux

Le noyau Linux est un noyau de système d'exploitation de type UNIX[19]. Ce noyau est évidemment utilisé par le système d'exploitation Linux (enfin, plus exactement GNU/Linux[20]) mais aussi par Android. C'est un logiciel libre développé essentiellement en langage C. En savoir plus[21].

LXC

LXC, contraction de l'anglais Linux Containers est un système de virtualisation, en savoir plus.[22][23]

17. Docker Compose - https://docs.docker.com/compose/overview/
18. Docker Swarm - https://docs.docker.com/engine/swarm/#feature-highlights
19. Unix selon Wikipedia - https://fr.wikipedia.org/wiki/Unix
20. GNU/Linux selon Wikipedia - https://fr.wikipedia.org/wiki/Linux
21. https://fr.wikipedia.org/wiki/Noyau_Linux
22. https://fr.wikipedia.org/wiki/LXC
23. Lien wikipedia - https://fr.wikipedia.org/wiki/LXC

Références

redhat.com https://www.redhat.com/fr/topics/containers/whats-a-linux-container

Crédits des ressources

Virtual Machine p. 9
https://www.docker.com/what-container#/package_software

Container p. 10
https://www.docker.com/what-container#/package_software