Adopter les bonnes pratiques avec Docker

Table des matières

Objectifs

- Utilisation des fonctions avancées de Docker
- Application des bonnes pratiques relatives à la création et l'utilisation des images
- Utiliser Docker et ses composantes dans un projet concret

Introduction

Les conteneurs sont de formidables outils pour les développeurs et ont de nombreux avantages :

- Faible utilisation de ressources
- Isolation
- Portabilité des conteneurs
- ...

Docker est de plus en plus utilisé et au fil du temps de nombreuses bonnes pratiques se sont imposées.

Dans ce module, nous allons nous intéresser à la rédaction des images Docker qui sont à la base des conteneurs mais aussi à la communication entre conteneurs.

I Construire ses propres images

Objectifs

- Pourquoi construire mes propres images
- Création de mon premier Dockerfile
- Les bonnes pratiques autour des Dockerfiles

1. Pourquoi construire ses propres images

Le Docker Hub

Fondamental

Le Docker Hub est un marché ou l'on peut télécharger gratuitement tout type d'image Docker.

On y retrouve différents types d'images :

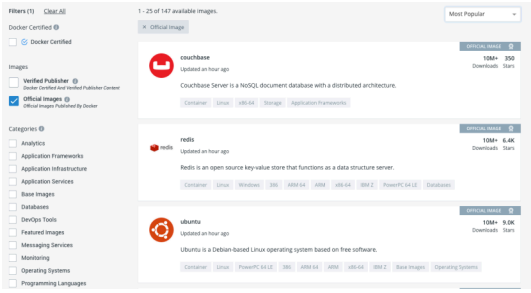
- Les images officielles
- Les images vérifiées
- Les images d'utilisateurs

Les images officielles

Exemple

Les images officielles sont des images qui sont créées par les équipes de Docker. On y retrouve un catalogue de produits ou de distributions open sources.

Ces images sont mises à jour régulièrement et sont considérées comme dignes de confiance.

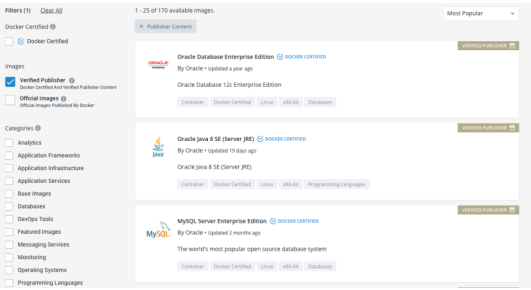


Les images vérifiées

Exemple

Les images vérifiées sont issues de personnes ou d'organisation vérifiées par Docker. Ces images sont publiées et maintenues par une organisation commerciale.

Lorsqu'un utilisateur souhaite bénéficier du support commercial de son produit, il doit utiliser ces images. Elles sont mises à jour régulièrement et peuvent être considérées comme étant dignes de confiance.



Remarque

Les images officielles et les images vérifiées sont construites automatiquement et les sources sont disponibles pour les utilisateurs. Il est donc possible de voir ce que contiennent ces images et comment elles fonctionnent et même y contribuer pour les améliorer.

Les images d'utilisateurs

Exemple

Le Docker Hub peut aussi héberger les images de n'importe quel utilisateur. Ces images sont à utiliser avec précaution. En effet, contrairement aux deux autres types d'images, les sources des images ne sont pas obligatoirement publiées. De la même manière, les images ne sont pas toujours mises à jour.

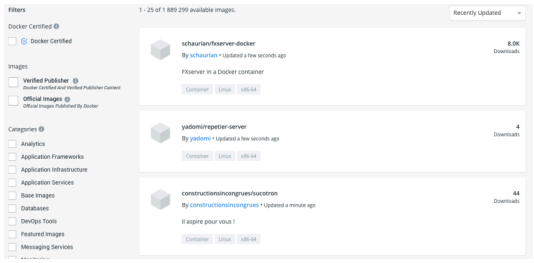
Il faut donc être très prudent lors du choix de ses images. Utiliser ce genre peut être risqué.

Un utilisateur malveillant peut par exemple cacher dans une image a priori inoffensive un programme qui va dérober les données des utilisateurs ou encore miner des cryptomonnaies en utilisant votre puissance de calcul.

Les images d'utilisateurs du Docker Hub ne sont soumises à aucun test de sécurité ou test fonctionnels. Il est par exemple possible d'envoyer sur le Docker Hub une image s'appelant "hacker/postgresql" qui contient en réalité une porte dérobée pour prendre le contrôle du serveur de l'utilisateur.

Entre 2017 et 2018, Docker a supprimé 17 images du Docker Hub[1] contenant des programmes malicieux. Ces images ont été utilisées par des millions d'utilisateurs. Il a fallu pas moins d'un an pour que les premiers chercheurs en sécurité alertent les utilisateurs sur la dangerosité de ces images.

Aujourd'hui encore certains serveurs sont toujours infectés même après avoir supprimé les conteneurs infectés.



1. https://www.bleepingcomputer.com/news/security/17-backdoored-docker-images-removed-from-docker-hub/

Remarque

Le Docker Hub n'est pas l'unique endroit où l'on peut récupérer des images. Il est tout à fait possible d'avoir son propre système d'hébergement d'image ou d'utiliser des systèmes tiers comme quay.io[2]

2. Construire ses propres images

Images personnalisées

Méthode

Nous l'avons vu, il n'est pas forcément recommandé d'utiliser des images utilisateurs issues du Docker Hub. Il arrive également que l'on souhaite créer une image avec des contraintes particulières :

- Je veux utiliser une distribution (debian, ubuntu, alpine) particulière
- Je veux utiliser des variables d'environnement pour configurer mes paramètres
- Je veux disposer mes fichiers de configuration dans le dossier /configuration

À la manière d'un langage, le Dockerfile est régi par l'utilisation de mots clés. Les plus utilisés sont FROM, RUN, CMD... Ils ont chacun une action particulière et donne en sortie un layer. Nous détaillerons cela par la suite.

Il faut garder en tête que l'ensemble de ces instructions sont ensuite assemblées pour au final représenter l'image Docker.

Méthode

Il est possible de construire une image Docker en lançant un conteneur et en exécutant manuellement dans ce conteneur des commandes comme l'installation d'un paquet ou encore l'édition d'un fichier. Une fois que toutes les modifications ont été apportées, il est possible de sauvegarder le résultat de ce conteneur dans une image en utilisant la commande docker commit

```
~ docker run --name mon-conteneur -it debian:stretch bash
root@584bdfe3d445:/# apt-get update && apt-get install -y curl
Ign:1 http://cdn-fastly.deb.debian.org/debian stretch InRelease
...
done.
root@584bdfe3d445:/# exit
~ docker commit mon-conteneur mon-image
sha256:0744ad5f1e7e816474d35d0a680e3421c332489ebf11f21c3ab0ce3779f34f54
~ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
mon-image           latest              0744ad5f1e7e        4 seconds ago
138MB
debian              stretch             de8b49d4b0b3        10 days ago
101MB
```

2. https://quay.io/

Attention

Cette manière de construire des images est difficilement reproductible et automatisable. De fait, on perd le plus grand interêt des conteneurs qui est d'avoir une resource portable mais également reproductible quelque soit l'environnement.

Dockerfile

Fondamental

Comme meilleure alternative, nous pouvons recourir à une approche automatisée de fabrication des images à l'aide de Dockerfile.

Un Dockerfile est un script de construction à base de texte contenant des instructions spéciales permettant de générer les images correctes et pertinentes à partir des images officielles de base. Les instructions séquentielles contenues dans Dockerfile peuvent inclure la sélection de l'image de base, l'installation de l'application requise, l'ajout de fichiers de configuration ou encore des fichiers de données. D'autres directives permettent à l'utilisateur de lancer l'exécution automatique d'un service ou encore l'exposition de ces services au monde externe.

Ce système de construction automatisé basé sur Dockerfile simplifie considérablement le processus de construction d'image. Il offre également une grande flexibilité dans l'organisation des instructions de construction et dans la visualisation du processus de construction complet.

On peut comparer le Dockerfile à une recette de cuisine. L'étape suivante est la préparation de cette recette en utilisant le processus de construction d'image de Docker.

À l'aide de la commande docker build, le démon Docker va itérer ligne par ligne sur les instructions du Dockerfile et construire l'image petit à petit.

Remarque

De par son format texte, il est très facile d'ajouter le Dockerfile à un outil de versionnement afin d'historiser les changements qui ont pu avoir lieu sur le fichier et revenir en arrière si nécessaire.

3. Exercice : Hello World

Hello World

L'objectif de cet exercice est de construire ensemble notre première image Docker.

En informatique la coutume veut que cela soit un Hello World. L'objectif ici est donc de constuire une image très simple dont le but va être d'afficher un message prédéfini.

Nous allons nous baser sur le Dockerfile suivant :

```
1 FROM busybox:latest
2 CMD echo Hello World!!
```

Question 1

[solution n°1 p. 32]

Essayez de deviner ce que font chacune des deux lignes de ce Dockerfile

Question 2

[solution n°2 p. 32]

À l'aide de la commande `docker build`, construisez votre image

Indice :

La commande `docker build` a la syntaxe suivante :

```
1 docker build <repertoire_du_dockerfile>
```

Question 3

[solution n°3 p. 32]

Lancez un conteneur basé sur votre image

Indice :

On utilise la commande `docker run`

Indice :

L'image à utiliser est indiquée à la toute fin de la commande de `build`. Dans notre cas `70cd93b6b962`

Question 4

[solution n°4 p. 32]

Intéressons nous maintenant à la liste de nos images. Lancez la commande suivante :

```
1 docker images
```

Que remarquez vous ?

Question 5

[solution n°5 p. 33]

Recommencez le build de l'image mais cette fois ci en utilisant l'option `-t` de la commande `docker build`

Indice :

```
1 ~ docker build -t <nom-image> .
```

II Image VS Conteneur

Objectifs

- Pourquoi construire mes propres images
- Création de mon premier Dockerfile
- Les bonnes pratiques autour des Dockerfiles

1. Image VS Conteneur

Avant de rentrer dans le vif du sujet, quelques éléments de vocabulaire qui vont nous aider à nous y retrouver

Image Docker Fondamental

Une image Docker est un ensemble de fichiers constituant une application. Cette collection comprend l'application elle-même, ainsi que toutes les bibliothèques (libraries), les fichiers binaires et d'autres dépendances, telles que les fichiers de configurations, nécessaires à l'exécution de l'application.

Lorsque l'on construit une nouvelle image, on se rend compte que tous les fichiers contenus dans l'image Docker sont en lecture seule et par conséquent, le contenu de l'image ne peut pas être modifié. Si vous choisissez de modifier le contenu de votre image, la seule option permise par Docker est d'ajouter une autre couche (un layer) avec les nouvelles modifications. En d'autres termes, une image Docker est composée layers que vous pouvez réviser. La commande `docker history`, permet de lister tous ces layers et de savoir ce qu'ils contiennent.

```
~ docker history mon-conteneur
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
3e12e812ffba        3 days ago          bash
19B
de8b49d4b0b3        2 weeks ago         /bin/sh -c #(nop)  CMD ["bash"]
0B
<missing>           2 weeks ago         /bin/sh -c #(nop) ADD
file:da71baf0d22cb2ede...   101MB
```

L'architecture des images Docker tire efficacement parti de ce concept de superposition pour ajouter de manière transparente des fonctionnalités supplémentaires aux images existantes. Ce système permet également de mutualiser certains layers et ainsi de réutiliser les layers identiques entre plusieurs images.

En d'autres termes, des fonctionnalités peuvent être ajoutées aux images existantes en ajoutant des layers supplémentaires par-dessus une image existante donnant naissance à une nouvelle image.

Les images Docker ont une relation parent-enfant et l'image la plus basse est appelée image de base. L'image de base est une image spéciale sans parent.

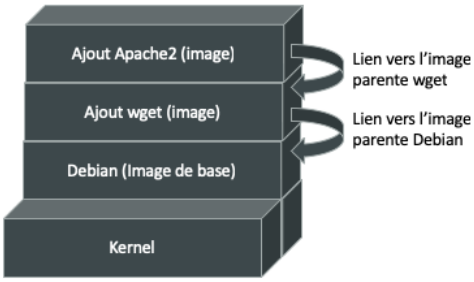
Exemple

L'image Debian Docker est un ensemble minimaliste de librairies et de fichiers binaires essentiels à l'exécution d'une application. Il n'inclut pas le noyau Linux (le *kernel* p.41), les pilotes de périphériques et divers autres services fournis par un système d'exploitation de la machine hôte.

Comme vous pouvez le voir dans la figure ci-contre, on utilise une image de base. Dans cet exemple, il s'agit de Debian. Dans un premier layer, on ajoute le paquet `wget`. Ce layer fait référence à l'image Debian en tant que parent. Dans la couche suivante, une instance du serveur d'applications `Apache 2` est ajoutée et fait référence à l'image `wget` en tant que parent.

Chaque ajout apporté à l'image de base d'origine est stocké dans un layer séparé.

La construction d'une image donne naissance à un identifiant unique qui regroupe tous les layers et qu'il est ensuite possible de partager au travers d'un registre comme le Docker Hub



Conteneur Docker

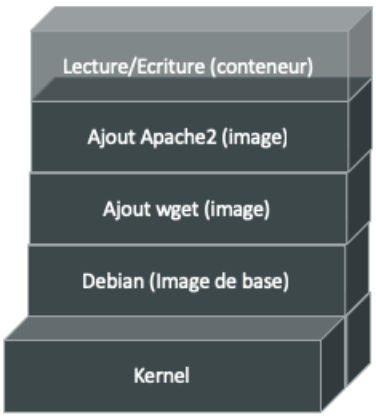
Fondamental

Les images Docker sont donc des modèles d'application composés de layer et qui sont en lecture seule.

Le conteneur Docker est dérivé de l'image Docker auquel il ajoute un layer de lecture-écriture au-dessus des layers d'image statiques. En programmation-objet, on pourrait comparer l'image a une classe alors qu'un conteneur est une instance de cette classe.

L'image Docker définit le comportement du conteneur Docker, par exemple le processus à exécuter lors du démarrage du conteneur. Ce sont les images Docker qui vont par la suite définir le comportement du conteneur une fois lancé.

Comme l'illustre la figure ci-contre, lorsque le conteneur est créé, un layer en écriture (lecture-écriture) est ajoutée au-dessus de l'image afin de conserver l'état de l'application. Il pourrait y avoir plusieurs images en lecture seule sous la couche conteneur (en écriture).



Réutilisation des layers et cache Remarque

Docker possède un système intelligent de cache qui est en mesure d'utiliser des layers déjà existant si les instructions ainsi que les images de base sont les mêmes.

Ce principe permet de limiter la taille des images sur les registy en utilisant au maximum la déduplication. Il permet également d'accélérer le build des images en ne rejouant pas systématiquement les actions des layers lors de la construction des images.

2. Exercice

Exercice 1

[solution n°6 p. 33]

De quoi est composée une image Docker

- **A** Un conteneur
- **B** Une image de base
- **C** De layers en lecture seule
- **D** Un volume

Exercice 2

[solution n°7 p. 34]

Peut-on écrire dans un layer d'une image après sa construction

- **A** oui
- **B** non

Exercice 3

[solution n°8 p. 34]

Quelle commande permet de lister les layers d'une image

- **A** docker layers ls <mon_image>
- **B** docker history <mon_image>
- **C** docker inspect <mon_image>
- **D** docker describe <mon_image>

Exercice 4

[solution n°9 p. 34]

Un conteneur est une instance d'une image

- **A** Oui
- **B** Non

Exercice 5

[solution n°10 p. 35]

Lorsque j'execute un conteneur il est possible d'écrire à l'intérieur

- **A** Oui
- **B** Non

Exercice 6

[solution n°11 p. 35]

Le processus de construction des images est optimisé avec l'utilisation de :

- **A** cache
- **B** mutualisation de layers
- **C** utilisation de conteneurs
- **D** d'images de base

III Le Dockerfile

Objectifs

- Pourquoi construire mes propres images
- Création de mon premier Dockerfile
- Les bonnes pratiques autour des Dockerfiles

1. Syntaxe des Dockerfile

Fondamental

Il existe des dizaines d'instructions pour les Dockerfiles. Nous allons ici voir quelque une des instructions principales ainsi que la syntaxe que doit avoir le Dockerfile.

Un Dockerfile est composé de deux éléments principaux : des instructions, des commentaires.

```
1 # Les commentaires commencent par un "#"
2 INSTRUCTION arguments
```

Les lignes d'instruction d'un Dockerfile sont composée de deux composants :

- L'`INSTRUCTION` elle même
- Des arguments attachés à cette `INSTRUCTION`

Bien que les instructions ne sont pas sensibles à la case, il est standard de les écrire en majuscule permettant ainsi de les différencier des arguments.

```
1 FROM busybox:latest
2 CMD echo Hello World!!
```

Dans notre exemple, `FROM` représente une instruction prenant en paramètre `busybox:latest`.

Les lignes vides ne sont pas interprétées par Docker et sont ignorées.

FROM

Méthode

L'instruction `FROM` est la plus importante et constitue la première instruction valide d'un fichier Dockerfile.

Elle définit l'image de base pour le processus de construction de notre image personnalisée. Les instructions suivantes utiliseront cette image de base et s'appuieront dessus. Le système de compilation Docker vous permet d'utiliser des images créées par n'importe qui.

Par défaut, le système de construction Docker recherche les images présentes sur la machine. Si l'image n'est pas trouvée, Docker va tenter de la télécharger depuis le Docker Hub disponible au public. Le système de construction Docker renvoie une erreur s'il ne parvient pas à trouver l'image.

COPY

Méthode

L'instruction COPY permet de copier un fichier entre la machine hôte et le système de fichier de notre nouvelle image. La syntaxe est la suivante :

```
COPY <source> <destination>
```

<source> correspond à un fichier ou un dossier présent dans le contexte de build (le dossier courant) sur la machine hôte

<destination> correspond au chemin du fichier dans notre nouvelle image

Si la destination n'est pas un chemin absolu, le démon Docker va commencer à partir de / à noter que la commande COPY est capable de construire automatiquement une arborescence dans la nouvelle image si celle-ci n'est pas existante.

ADD

Remarque

Il existe une autre commande qui a le même comportement que COPY : la commande ADD. Cette commande permet de copier des fichiers dans le conteneur aussi bien depuis votre machine ou à parti d'une URL. Elle est aussi capable de décompresser des archives automatiquement.

Cependant, son utilisation n'est pas recommandée puisqu'elle peut amener à introduire dans un conteneur des fichiers vérolés ou provenant de sources non identifiées. Avec la commande COPY, on s'assure de copier uniquement ce qui est nécessaire et dont on connaît le contenu.

RUN

Méthode

L'instruction RUN fait partie des instructions principales d'un Dockerfile. Elle peut exécuter n'importe quelle commande dans votre image. La syntaxe est la suivante :

```
RUN <commande>
```

Nous allons le voir par la suite mais chaque instruction du Dockerfile va se transformer en layer. Il est recommandé de regrouper toutes les instructions RUN dans une seule et même instruction. Pour l'installation du package wget, on va donc utiliser l'instruction suivante :

```
RUN apt-get update && \
    apt-get install -y wget && \
    apt-get clean
```

ENV

Méthode

L'instruction ENV définit une variable d'environnement dans la nouvelle image. Une variable d'environnement est une paire clé-valeur, accessible à n'importe quel script ou application. Les applications Linux utilisent beaucoup les variables d'environnement pour une configuration initiale.

```
ENV <clé> <valeur>
```

<clé>: représente la variable d'environnement qui va être définie

<value>: représenta la valeur qui va être attribuée à la variable d'environnement

VOLUME

Méthode

L'instruction VOLUME crée un répertoire dans le système de fichiers de notre nouvelle image. Il peut ensuite être utilisé pour monter des volumes à partir de l'hôte Docker ou des autres conteneurs.

```
VOLUME <point de montage>
#ou
VOLUME ["<point de montage>"]
```

Dans les lignes précédentes, `<point de montage>` est le dossier à exposer dans la nouvelle image.

EXPOSE

Méthode

L'instruction EXPOSE ouvre un port réseau de conteneur pour la communication entre le conteneur et le monde externe.

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

`<port>`: C'est le port réseau qui doit être exposé au monde extérieur.

`<protocole>`: Il s'agit d'un champ facultatif permettant de spécifier un protocole de transport spécifique, tel que TCP et UDP. Si aucun protocole de transport n'a été spécifié, alors TCP est supposé être le protocole de transport.

CMD

Méthode

L'instruction CMD peut exécuter n'importe quelle commande à la manière de l'instruction RUN. Cependant, la différence majeure entre ces deux, c'est le moment de l'exécution. La commande fournie via l'instruction RUN est exécutée pendant la construction de l'image, tandis que la commande spécifiée par l'instruction CMD est exécutée lorsque le conteneur est lancé.

L'instruction CMD fournit une exécution par défaut de l'image que l'on va construire.

```
CMD <commande>
```

Bien qu'il soit possible d'avoir plusieurs instructions CMD dans un Dockerfile, c'est la dernière instruction CMD qui va être retenue et utilisée comme commande par défaut.

Remarque

Il est possible de remplacer l'instruction CMD au lancement d'un conteneur Docker. Dans l'exemple ci-dessous, `ma_commande` va remplacer l'instruction CMD se trouvant dans le Dockerfile de `mon_image`

```
docker run mon_image ma_commande
```

2. Exercice : Conteneur Nginx

Exercice

L'objectif de cet exercice est de créer votre propre conteneur Docker `Nginx` personnalisé. Pour cela, nous allons dans un premier temps construire notre propre image `Nginx` puis dans un second temps la personnaliser en y ajoutant un le contenu d'un site web

Question 1

[solution n°12 p. 36]

Construisez une image `Nginx` en se basant sur l'image de base de `debian` en version `stretch`

Indice :

L'instruction `FROM` permet de sélectionner l'image de base

Indice :

Il est nécessaire de modifier le fichier /etc/nginx/nginx.conf pour faire fonctionner nginx dans un conteneur.

```
daemon off;
user www-data;
worker_processes  1;

error_log  stderr error;
events {
    worker_connections  1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
    access_log stdout;

    server {
        listen 80;

        root /var/www/html;
        index index.php index.html index.htm index.nginx-debian.html;

        client_max_body_size 100M;
        client_body_buffer_size 128k;

        location ~ /\.ht {
            deny all;
        }

        location ~ /(data|conf|bin|inc|config|lib)/ {
            deny all;
        }
    }
}
```

Indice :

On utilise l'instruction `COPY` pour remplacer le fichier.

Indice :

Pour lancer nginx, on utilise la commande `/usr/sbin/nginx`

Question 2

[solution n°13 p. 36]

Nous avons développé une nouvelle application web que nous souhaitons héberger dans notre conteneur nginx. https://github.com/abarbare/api-run-app

Faites un nouveau Dockerfile utilisant en image de base l'image `Nginx` que nous venons de construire

Indice :

Vous pouvez télécharger l'application sur votre ordinateur puis la copier dans le conteneur. Il est aussi possible de télécharger l'app directement dans votre Dockerfile à l'aide de la commande git

Indice :

L'application est à ajouter dans le dossier `/var/www/html.`

Indice :

Les fichiers de l'application doivent appartenir à l'utilisateur `www-data:www-data`

IV Les volumes Docker

Objectifs

- Créer des volumes de données
- Partage des données de la machine Docker vers un conteneur

1. Volume Docker

Cycle de vie des conteneurs Rappel

Le cycle de vie d'un conteneur est différent de celui d'une machine virtuelle classique. Le conteneur vit généralement aussi longtemps que l'application qu'il contient et inversement. Cependant, cela impacte négativement les données de l'application. De nombreuses applications ont des besoins d'historiser des changements ou des modifications des données qu'elle manipule. Lors du management d'une application dans une machine virtuelle classique, l'application est régulièrement mise à jour ou subit des maintenances. Lorsque cette application meurt, les données associées à l'application sont conservées dans le système de fichier.

Dans un conteneur, le fonctionnement est différent. Les mises à jour des applications s'effectuent en construisant systématiquement un nouveau conteneur avec la version la plus récente de l'application et en supprimant simplement l'ancien. De même, lorsqu'une application fonctionne mal, un nouveau conteneur doit être lancé et l'ancien doit être jeté. En résumé, les conteneurs ont un cycle de vie très court.

Dans l'environnement des conteneurs, l'environnement d'exécution complet, y compris ses fichiers de données, est généralement regroupé et encapsulé dans le conteneur. Pour une raison quelconque, lorsqu'un conteneur est détruit, les fichiers de données de l'application périssent également avec le conteneur.

Si l'on souhaite conserver les données des applications, on doit donc stocker ces données à l'extérieur des conteneurs et transmettre ces données dans les conteneurs pour qu'elles puissent être lues et modifiées par l'application.

Pour gérer cette problématique, Docker introduit une fonctionnalité de `volume`.

Fondamental

Les volumes de données Docker permettent donc de partager des données entre la machine Docker et les conteneurs. Ces volumes font partie du système de fichier de la machine Docker et peuvent être utilisés de plusieurs manières.

- Dans la construction d'une image avec l'instruction `VOLUME`
- Utiliser l'option `-v` lors du lancement d'un conteneur Docker

2. Exercice : Comprendre les volumes

Question 1

[solution n°14 p. 36]

Créez un Dockerfile utilisant l'image de base `debian:stretch` déclarant un volume

Question 2

[solution n°15 p. 37]

Construisez l'image à l'aide de la commande `docker build` et nommez la `docker-vol`

Question 3

[solution n°16 p. 37]

Inspectez l'image construite à l'aide de la commande `docker inspect` que voyez vous ?

Question 4

[solution n°17 p. 37]

Lancez un conteneur à partir de notre image `docker-vol`

Vérifiez la présence du dossier `/mountPoint`

Question 5

[solution n°18 p. 37]

Ajoutez un nouveau fichier dans `/mountPoint`

Question 6

[solution n°19 p. 37]

Dans un autre shell, inspectez le conteneur et trouvez l'emplacement du volume de données

3. Gérer des volumes

Docker permet de gérer les volumes de données à l'aide de plusieurs commandes `docker volumes` :

- `create`: permet de créer un nouveau volume
- `inspect`: Affiche les informations d'un ou plusieurs volumes
- `ls`: Liste les volumes de l'hôte Docker
- `rm`: Supprime un volume

Création d'un volume nommé Syntaxe

```
~ docker volume create --name exemple
```

Tout comme les conteneurs, un volume peut avoir un nom ce qui permet de le manipuler plus facilement.

```
~ docker volume ls
DRIVER              VOLUME NAME
local
50957995c7304e7d398429585d36213bb87781c53550b72a6a27c755c7a99639
local               exemple
```

Informations sur les conteneurs Syntaxe

```
~ docker volume inspect example
  [
      {
          "Name": "example",
          "Driver": "local",
```

```
        "Mountpoint":
        "/var/lib/docker/volumes/exemple/_data",
        "Labels": {},
        "Scope": "local"
} ]

```

Utilisation d'un volume nommé

Syntaxe

```
~ docker run -v example:/mountPoint -it debian:stretch
```

Cette commande permet de monter un volume manuellement dans un nouveau conteneur. En cas de suppression du conteneur, il est toujours possible de connecter le volume sur un nouveau conteneur.

Nous avons décrit précédemment les étapes à suivre pour créer un volume de données dans une image Docker à l'aide de l'instruction VOLUME du fichier Docker Compse.

Cependant, Docker ne fournit aucun mécanisme permettant de partager le répertoire ou le fichier présent sur l'hôte pendant la construction afin de garantir la transférabilité des images Docker. La seule disposition fournie par Docker consiste à monter le répertoire ou le fichier de l'hôte sur le volume de données d'un conteneur lors du lancement de celui-ci.

Afin de permettre la transférabilité des données d'un conteneur à un autre, il est possible d'utiliser l'option `-v` de la sous-commande `docker run`.

- `-v <chemin dans le conteneur>`
- `-v <chemin hôte docker>:<chemin dans le conteneur>`
- `-v <chemin hôte docker>:<chemin dans le conteneur>:<read write mode>`
- `-v <nom de volume>:<chemin dans le conteneur>`
- `-v <nom de volume>:<chemin dans le conteneur>:<read write mode>`

Exemple

```
~ docker run -v example:/mountPoint -it debian:stretch
```

Cette commande permet de monter un volume manuellement dans un nouveau conteneur. En cas de suppression du conteneur, il est toujours possible de connecter le volume sur un nouveau conteneur.

```
~ docker run -v /home/myuser/docker/mon-application:/mountPoint -it
debian:stretch
```

Cette commande permet de partager des données de l'hôte Docker du chemin `/home/myuser/docker/mon-application` vers le conteneur dans le dossier `/mountPoint`

4. Exercice : Volume et PostgreSQL

L'objectif de cet exercice est de lancer un conteneur de base de données PostgreSQL qui utilise un volume de données

Question 1

[solution n°20 p. 38]

Lancer un conteneur à partir de l'image `postgres:10` et utiliser les volumes docker pour héberger le contenu du dossier `/var/lib/postgresql/data`

Indice :

Vous pouvez créer un volume docker ou partager directement un dossier de l'hôte docker

Question 2

[solution n°21 p. 38]

Entrez dans le conteneur que vous venez de créer et ajoutez une table nommée test contenant une colonne varchar data de taille 255 et ajoutez-y une entrée de votre choix

Indice :

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

Indice :

```
1 create table test(data varchar(255));
```

Indice :

```
1 insert into ma_table(ma_colonne) values('ma_valeur');
```

Question 3

[solution n°22 p. 38]

Détruisez votre conteneur de base de donnée

Question 4

[solution n°23 p. 38]

Lancez un nouveau conteneur Docker PosgreSQL réutilisant le précédent volume de données. Vérifiez la présence des données que vous avez créées précédemment.

V Les bonnes pratiques

Objectifs

- Pourquoi construire mes propres images
- Création de mon premier Dockerfile
- Les bonnes pratiques autour des Dockerfiles

1. Bonnes pratiques Dockerfile

Conseil

Il n'est pas toujours facile d'écrire un Dockerfile. Vous pouvez visiter le github docker-library[3] qui regroupe tous les dockerfiles des images officielles

Dockerfile[4] du conteneur officiel hello-world

```
1 FROM scratch
2 COPY hello /
3 CMD ["/hello"]
```

Conseil

L'image de base d'un Dockerfile est très importante et nécessite d'être taguée avec une version bien précise. En effet, on doit être capable de reconstruire une image à partir de ces sources à tout moment. Les images notamment les images officielles sont mises à jour très régulièrement et il est possible qu'elle ne soit pas compatible entre elles ce qui pourrait mener à casser la construction de notre image personnalisée.

Par exemple, on construit une image `FROM debian:latest` qui correspond aujourd'hui à la version `stretch`. D'ici quelques mois, la version `buster` de Debian va remplacer la version `stretch`. Il y a de grandes chances pour que le build de notre image ne fonctionne plus à cause de ce changement.

Pour éviter ces problèmes, il est donc primordial d'indiquer une version précise dans nos Dockerfiles au niveau de l'image de base.

Fondamental

Comme nous l'avons vu, une image Docker est composée d'un ensemble de layers correspondant chacun à une instruction dans un Dockerfile. Malgré leur faible taille, l'accumulation de ces layers peut donner lieu à des images pouvant faire jusqu'à 1Go.

Avec de telles images, on est loin de la philosophie originelle des conteneurs avec une faible utilisation de ressources.

3. https://github.com/docker-library
4. https://github.com/docker-library/hello-world/blob/master/amd64/hello-world/Dockerfile

Lorsque l'on écrit un Dockerfile, il est donc recommandé de réfléchir à ce qui est vraiment nécessaire dans le conteneur pour le faire fonctionner et ne garder que les éléments essentiels. Par exemple, il n'est pas utile de garder un éditeur de texte ou encore les paquets nécessaires à la compilation de notre programme dans l'image finale.

Complément

En réduisant le nombre de paquets installés dans notre image, nous réduisons la surface d'attaque des conteneurs qui vont en découler. En d'autres termes, les conteneurs ne contenant pas de paquets superflus sont moins vulnérables aux attaques informatiques. Il faut garder en tête que chaque paquet contient des bug ou des failles de sécurité. Limiter le nombre de paquets limite donc le risque.

Fondamental

Comme nous l'avons vu à chaque fois que l'on souhaite faire une modification dans une image, on doit ajouter un nouveau layer à cette image. Dans ce nouveau layers seront stockés les modifications apportées aux fichiers. Lorsqu'un conteneur fonctionne et qu'il nécessite l'accès à un fichier contenu dans le second layer, il va devoir parcourir chacun des layers de notre image pour parvenir au fichier qui l'intéresse. Limiter le nombre de layers de notre image permet donc d'accélérer le fonctionnement de nos conteneurs en limitant le passage d'une couche à une autre.

Pour cela, on essaie le plus possible de regrouper des commandes dans nos instructions. Vous l'avez peut-être remarqué, depuis le début, nous utilisons l'instruction && dans chacune de nos instructions RUN. L'utilisation du && permet de concaténer plusieurs commandes entre elles, mais aussi de passer à la commande suivante uniquement si la commande précédente a réussie (code de sortie 0). Ainsi, on peut enchaîner plusieurs actions sur un seul et même layer.

Remarque

Pour plus de lisibilité, il est possible d'utiliser les \ associés aux && ils permettent de faire un retour à la ligne non interprété par le Docker lors de la construction du Dockerfile et qui facilite grandement la lecture des Dockerfile.

```
RUN apt-get update && \
    apt-get install -y \
      unzip   \
      wget && \
    wget https://github.com/ether/etherpad-lite/archive/1.7.0.zip && \
    unzip 1.7.0.zip && \
    bash etherpad-lite-1.7.0/bin/run.sh
```

Lors de l'installation de paquets, il est également recommandé de les installer en suivant l'ordre alphabétique, cela permet d'un coup d'oeil de voir si un paquet est installé sans avoir à parcourir tout le Dockerfile.

Fondamental

Par défaut, c'est l'utilisateur root qui est utilisé dans les images de base. L'utilisation de cet utilisateur n'est pas recommandé lors de l'utilisation d'un conteneur.

Ce super utilisateur est en effet en mesure de tout faire dans une conteneur. Dans le cas d'une faille de sécurité, l'attaquant peut ainsi modifier notre conteneur pour injecter du code malicieux à notre insu. C'est pourquoi il est recommandé d'ajouter dans nos images un utilisateur non privilégié ayant un nombre de droits limités. Ainsi en cas de piratage, l'attaquant n'aura que très peu de possibilités dans le conteneur.

La directive USER du Dockerfile permet de changer d'utilisateur lors des instructions qui suivent sont ajout. Cela permet donc d'avoir des conteneur utilisant une utilisateur non privilégié.

```
1 FROM debian:stretch
2 #utilisateur root
3 RUN apt-get update && \
4     apt-get install -y figlet
5 USER www-data
6 #utilisateur www-data
7 CMD ["figlet", "APIUTC"]
```

Syntaxe

Il peut être nécessaire de connaître le créateur d'une image afin de lui indiquer un problème ou encore lui demander d'ajouter une nouvelle fonctionnalité. Pour cela un instruction LABEL a été ajoutée dans la syntaxe du Dockerfile.

Un LABEL est une indication de la forme clé valeur qui permet de stocker des information comme la date de création de l'image ou encore son propriétaire. On l'utilise de la manière suivante :

```
1 LABEL maintainer="mon@email.com" \
2       creation_date="2019-01-13"
```

Il est ensuite possible de récupérer les labels d'une image à l'aide de la commande `docker inspect`

```
1 docker inspect mon-app | jq .[0].Config.Labels
2 {
3   "creation_date": "2019-01-13",
4   "maintainer": "mon@email.com"
5 }
```

VI Service web avancé

Objectifs

- Héberger plusieurs services web dans des conteneurs Docker
- Utiliser un reverse proxy dynamique

1. Exercice : Nginx PHP-FPM

Nginx php

Nous allons créer un conteneur nginx capable de servir à nos utilisateurs des fichiers PHP à l'aide de PHP FPM

Question

[solution n°24 p. 39]

En vous basant sur ce que vous avez appris dans le module serveur web PHP créez un conteneur Nginx et PHP FPM à partir de l'image `debian:buster`

Indice :

Vous devez installer les packages suivants :

- `nginx`
- `php7.3`
- `php7.3-fpm`
- `php7.3-gd`
- `php7.3-pgsql`
- `php7.3-xml`

Indice :

Pour configurer PHP-FPM, les commandes suivantes doivent être lancées

```
1 echo "cgi.fix_pathinfo = 0;" >> /etc/php/7.3/fpm/php.ini
2 sed -i -e "s|;daemonize\s*=\s*yes|daemonize = no|g" /etc/php/7.3/fpm/php-fpm.conf
3 sed -i -e "s|listen\s*=\s*127\.0\.0\.1:9000|listen = /var/run/php-fpm7.sock|g"
  /etc/php/7.3/fpm/pool.d/www.conf
4 sed -i -e "s|;listen\.owner\s*=\s*|listen.owner = |g"
  /etc/php/7.3/fpm/pool.d/www.conf
5 sed -i -e "s|;listen\.group\s*=\s*|listen.group = |g"
  /etc/php/7.3/fpm/pool.d/www.conf
6 sed -i -e "s|;listen\.mode\s*=\s*|listen.mode = |g" /etc/php/7.3/fpm/pool.d/www.conf
7
```

Indice :

Vous devez configurer Nginx pour qu'il utilise PHP-FPM pour rende le PHP à l'aide du fichier de configuration suivant

```
1 daemon off;
2
3 user www-data;
4 worker_processes  1;
5
6 error_log stderr error;
7
```

```
events {
    worker_connections  1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;

    keepalive_timeout  65;

    server {
        listen 80;

        root /var/www/html;
        index index.php index.html index.htm;

        client_max_body_size 128M;
        client_body_buffer_size 128k;

    location ~ \.php$ {
            try_files $uri =404;
            fastcgi_pass unix:/run/php/php7.3-fpm.sock;
            fastcgi_index index.php;
            fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
            fastcgi_param PHP_VALUE "memory_limit = 256M
            post_max_size = 128M
            upload_max_filesize = 128M
            ";
            include fastcgi_params;
        }

        location ~ /\.ht {
            deny all;
        }

        location ~ /(data|conf|bin|inc)/ {
            deny all;
        }
    }
}

```

Indice :

Le conteneur doit lancer à la fois PHP-FPM mais aussi Nginx à l'aide des commandes suivantes :

- php-fpm7.3
- nginx

On fera donc un script appelé ici `start.sh` qui se chargera de démarer `php-fpm` en arrière plan à l'aide de l'instruction `&` et qui démarre ensuite `nginx` en premier plan

2. Exercice : Nginx + Base de données

Nous allons maintenant utiliser notre conteneur Nginx PHP-FPM avec une base de données PostgreSQL

Question 1

[solution n°25 p. 39]

Lancez à l'aide d'un fichier docker-compose un conteneur de base de données PostgreSQL et un conteneur Nginx PHP-FPM dans un même réseau Docker et utilisez des volumes pour stocker les données des applications

Indice :

PostgreSQL stocke ses données dans le répertoire `/var/lib/postgresql/data`

Indice :

Ajoutez les fichiers de votre application PHP dans le répertoire `/var/www/html`

Question 2

[solution n°26 p. 40]

Assurez vous que vous pouvez contacter le conteneur PostgreSQL depuis le conteneur Nginx

Indice :

Vous devez rentrer dans le conteneur

Indice :

La commande `ping` peut être utilisée

Question 3

[solution n°27 p. 40]

Vérifiez que vous avez accès à votre application

3. Exercice : Reverse proxy Traefik

Nous allons maintenant utiliser `Traefik` qui est un reverse proxy capable de se connecter à Docker pour rediriger nos clients vers notre application en se basant sur un nom de domaine.

`Traefik` est un reverse proxy capable de se connecter au démon Docker et de lire des informations sur les conteneurs qui sont lancés. Cela permet de générer la configuration du reverse proxy de manière automatique en se basant sur des labels ajoutés sur les conteneurs.

Pour fonctionner, `Traefik` nécessite un accès à la socket du démon Docker mais aussi nécessite un fichier de configuration personnalisé qui peut être monté par le biais d'un volume.

Ci-dessous un exemple de fichier docker-compose permettant de lancer `Traefik`

```
services:
  ...
  traefik:
      image: traefik:maroilles
      container_name: traefik
      networks:
        - application
      ports:
        - "80:80"
      volumes:
        - /var/run/docker.sock:/var/run/docker.sock
        - /DATA/traefik/traefik.toml:/traefik.toml
      restart: always
    ....
```

Ci-dessous le fichier de configuration `traefik.toml`

```
[docker]
endpoint = "unix:///var/run/docker.sock"
domain = "docker.localhost"
watch = true
exposedByDefault = false
usebindportip = true
network = "application"
```

Les conteneurs applicatifs nécessitent également l'ajout d'un label personnalisé afin de permettre à `Traefik` de générer sa configuration. Par exemple, pour notre conteneur Nginx PHP-FPM, on peut utiliser la configuration suivante.

Elle permet de rediriger les clients vers le conteneur Nginx lorsque ceux-ci arrivent sur le serveur web en ayant appelés `mon-application.mondomaine.fr`

```
...
services:
   nginx:
      container_name: nginx
      image: mon-nginx
      networks:
        - application
      volumes:
          - /DATA/mon-application/:/var/www/html/
      labels:
          - "traefik.frontend.rule=Host:mon-application.mondomaine.fr"
          - "traefik.port=80"
          - "traefik.enable=true"
...
```

Question 1

[solution n°28 p. 40]

Utilisez `Traefik` en tant que reverse proxy pour votre application PHP à partir d'un fichier docker-compose

Question 2

[solution n°29 p. 40]

Ajoutez un second site web de votre choix et configurez Traefik pour permettre l'accès aux deux sites en parallèle via deux noms de domaines distincts.

Vous pouvez par exemple héberger un blog Ghost[5], ou encore un instance Nextcloud[6] si vous le souhaitez

5. https://hub.docker.com/_/ghost/
6. https://hub.docker.com/_/nextcloud

VII Complément : Tips, liens et commandes utiles

J'ai l'erreur suivante quand je veux utiliser docker Conseil

```
Error response from daemon: Bad response from Docker engine
```

Vérifiez que votre démon Docker est bien lancé

```
sudo systemctl start docker.service
```

Démarrer Docker au boot de la machine

```
sudo systemctl enable docker
```

Exécuter mes commandes Docker sans sudo ou utilisateur root Conseil

La solution est très simple. Il suffit de créer un groupe nommé `docker`

```
sudo groupadd docker
```

Ajoutez désormais l'utilisateur dont vous vous servez (qui sera retourné par la variable $USER) à ce groupe.

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

Fermez votre session et reconnectez vous afin que votre adhésion au groupe soit pris en compte. Vous pouvez dès à présent tester une commande docker (`docker image ls` par exemple).

Note : si cette manipulation n'a pas abouti ou pour toute information supplémentaire, veuillez vous référer à cette section[7] de la documentation officielle.

Quelques commandes à garder sous la main

L'utilisation du `$(<commande>)` va nous permettre d'exécuter en premier lieu la commande à l'intérieur de ces parenthèses. La commande va ensuite être remplacée par sa réponse (si c'est une liste de plusieurs éléments, chaque élément sera pris en considération à la suite). Ceci va nous permettre, en quelque sorte, d'exécuter plusieurs commandes à la fois.

Nettoyage des images et conteneurs non utilisés

```
docker system prune -a
```

Stopper tous les conteneurs en cours (quand on veut par exemple les supprimer). Cette commande va renvoyer une erreur si aucun conteneur n'est en exécution.

```
docker stop $(docker container ps -q)
```

Supprimer toutes les instances de conteneurs précédemment lancés, une fois les conteneurs stoppés.

```
docker rm $(docker container ps -aq)
```

Supprimer toutes les images de notre machine

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

7. https://docs.docker.com/install/linux/linux-postinstall/#manage-docker-as-a-non-root-user

Entrer dans un conteneur en exécution avec un shell bash

```
1 docker exec -it <nom_ou_id_du_conteneur> /bin/bash
```

Solutions des exercices

Solution n°1

[exercice p. 8]

```
1 FROM busybox:latest
```

Cette première instruction permet de choisir une image de base sur laquelle va reposer notre image personnalisée.

Dans notre cas, nous avons choisi d'utiliser la distribution `busybox` connue pour sa légèreté

```
1 CMD echo Hello World!!
```

Cette seconde instruction est en charge de la commande à lancer au démarrage de notre conteneur. Ici, nous allons afficher dans la console `Hello World`

Solution n°2

[exercice p. 9]

```
1  ~ docker build .
2  Sending build context to Docker daemon  2.048kB
3  Step 1/2 : FROM busybox:latest
4  latest: Pulling from library/busybox
5  57c14dd66db0: Pull complete
6  Digest: sha256:7964ad52e396a6e045c39b5a44438424ac52e12e4d5a25d94895f2058cb863a0
7  Status: Downloaded newer image for busybox:latest
8   ---> 3a093384ac30
9  Step 2/2 : CMD echo Hello World!!
10  ---> Running in a6284278314d
11 Removing intermediate container a6284278314d
12  ---> 70cd93b6b962
13 Successfully built 70cd93b6b962
```

Solution n°3

[exercice p. 9]

```
1  ~ docker run 70cd93b6b962
2  Hello World!!
```

> **Remarque**
>
> Nous avons pu créer notre propre image personnalisée basée sur la distribution `busybox` et étendre cette image pour construire notre Hello World !!.
>
> C'est une application simple, mais les images réalisées dans des projets de plus large envergure utilisent la même technologie.

Solution n°4

[exercice p. 9]

```
1 docker images
2 REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
3 <none>              <none>              70cd93b6b962        1  minutes ago
  1.2MB
4 mon-image           latest              3e12e812ffba        37 minutes ago
  101MB
```

```
busybox             latest              3a093384ac30        7 days ago
1.2MB
debian              stretch             de8b49d4b0b3        10 days ago
101MB
```

La nouvelle image que l'on vient de construire n'a pas de nom ni de tag.

Solution n°5

[exercice p. 9]

```
~ docker build -t helloworld .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM busybox:latest
 ---> 3a093384ac30
Step 2/2 : CMD echo Hello World!!
 ---> Using cache
 ---> 70cd93b6b962
Successfully built 70cd93b6b962
Successfully tagged helloworld:latest
```

```
~ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
helloworld          latest              70cd93b6b962        19 minutes ago
1.2MB
mon-image           latest              3e12e812ffba        41 minutes ago
101MB
busybox             latest              3a093384ac30        7 days ago
1.2MB
debian              stretch             de8b49d4b0b3        10 days ago
101MB
```

On remarque maintenant que notre image à l'id `70cd93b6b962` est maintenant référencé avec un nom et un tag. Par défaut, si aucun tag n'est spécifié, c'est `latest` qui va être utilisé par docker. Vous pouvez spécifier un tag en utilisant les "`:`" après votre nom d'image. Il est recommandé de toujours construire une image en lui spécifiant un nom.

```
docker build -t helloworld:v1 .
```

Par défaut lors de la construction d'une image, Docker va chercher dans le dossier courant un fichier nommé Dockerfile (attention à la case). Il est possible d'utiliser l'option `-f` pour spécifier un autre nom de fichier pour notre Dockerfile.

Solution n°6

[exercice p. 12]

De quoi est composée une image Docker

A Un conteneur — Un conteneur est une instance d'une image

B ☐ Une image de base — Chaque image Docker se repose sur une image de base qui est ensuite enrichie

C ☐ De layers en lecture seule — Chaque instruction d'une image va être stockée dans un layer qu'il ne sera pas possible d'éditer

D Un volume — Un volume représente un emplacement de stockage partagé entre un conteneur et la machine docker

Solution n°7

[exercice p. 12]

Peut-on écrire dans un layer d'une image après sa construction

A oui

B non

Une fois construit, il n'est plus possible de modifier le contenu d'un layer d'une image. Si l'on souhaite éditer son contenu, on doit ajouter un layer supplémentaire à l'image et effectuer les modifications dans ce nouveau layer.

Solution n°8

[exercice p. 12]

Quelle commande permet de lister les layers d'une image

A docker layers ls <mon_image>

B docker history <mon_image>

C docker inspect <mon_image>

D docker describe <mon_image>

Solution n°9

[exercice p. 12]

Un conteneur est une instance d'une image

A Oui

B Non

Un conteneur dérive d'une image. Lorsqu'il est exécuté, il va lancer les instructions contenues au sein de l'image.

Solution n°10

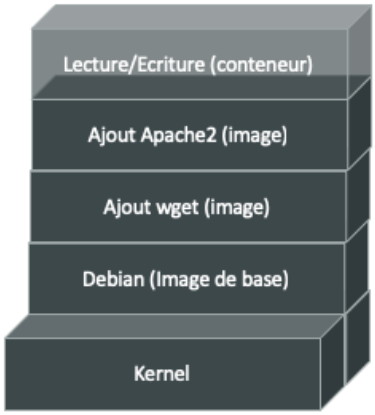
[exercice p. 13]

Lorsque j'execute un conteneur il est possible d'écrire à l'intérieur

A Oui

B Non

Lorsqu'un conteneur est lancé, un nouveau layer est ajouté à celui de l'image de laquelle il découle. Ce layer a la particularité d'être en lecture et en écriture ce qui permet à l'utilisateur de pouvoir ajouter ou modifier des fichiers dans le conteneur.



Solution n°11

[exercice p. 13]

Le processus de construction des images est optimisé avec l'utilisation de :

A

☐ cache — Lorsque des layers ont déjà été construits, Docker ne va pas systématiquement les reconstruire et va réutiliser les layer existant pour la construction de nouvelles images

B

☐ mutualisation de layers — Si deux images ayant la même image parente effectuent les mêmes actions, le layer reprenant ces actions sera le même au sein des deux images

C

utilisation de conteneurs — Les conteneurs ne sont pas utilisés dans le processus de build d'une image

D

d'images de base — L'utilisation de l'image de base n'optimise pas la construction de l'image

Solution n°12

[exercice p. 17]

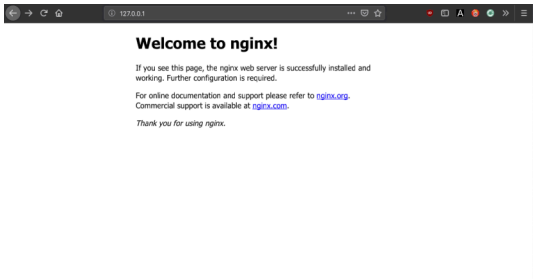
```
FROM debian:stretch

RUN apt-get update && \
    apt-get install -y nginx

COPY nginx.conf /etc/nginx/nginx.conf

EXPOSE 80

CMD ["/usr/sbin/nginx"]
```

```
docker build -t nginx .
```

On peut maintenant lancer notre nouvelle image et vérifier son bon fonctionnement

```
docker run --rm -p 80 :80 nginx
```

On peut maintenant ouvrir son navigateur à l'adresse http://127.0.0.1:80 et vérifier que l'on a bien accès à la page d'accueil d'nginx



Solution n°13

[exercice p. 18]

```
FROM nginx
COPY --chown=www-data:www-data api-run-app/ /var/www/html/
```

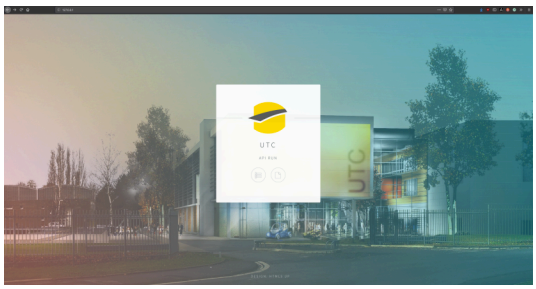
```
FROM nginx
ARG APP=https://github.com/abarbare/api-run-app.git
RUN apt-get update && \
    apt-get install -y git && \
    rm -Rf /var/www/html/ && \
    git clone $APP /var/www/html/ && \
    chown -R www-data:www-data /var/www/html/
```

L'utilisation de la directive ARG permet de construire une image différente sans avoir à modifier le Dockerfile. Si l'on souhaite construire une image pour une application similaire https://github.com/abarbare/ansible-meetup-app il suffit de construire l'image avec la commande suivante

```
docker build -t mon-app --build-arg APP=https://github.com/abarbare/ansible-meetup-app.git .
```

```
docker run --rm -p 80 :80 app
```

On peut maintenant ouvrir son navigateur à l'adresse http://127.0.0.1:80 et vérifier que l'on a bien accès à notre application



Solution n°14

[exercice p. 20]

```
FROM debian:stretch
VOLUME /mountPoint
```

Solution n°15

[exercice p. 20]

```
docker build -t docker-vol .
```

Solution n°16

[exercice p. 20]

On voit une partie déclarant un volume

```
docker inspect docker-vol
          [
              {
                  "Id": "sha256:<64 bit hex id>",
                  "RepoTags": [
                      "docker-vol:latest"
                  ],
          ... TRUNCATED OUTPUT ...
                  "Volumes": {
                      "/mountPoint": {}
                  },
          ... TRUNCATED OUTPUT ...
```

Solution n°17

[exercice p. 20]

```
docker run --rm -it docker-vol
```

```
root@8d22f73b5b46:/# ls -ld /mountPoint
      drwxr-xr-x 2 root root 4096 Jan 18 20:22
      /mountPoint
```

Solution n°18

[exercice p. 20]

root@8d22f73b5b46:/# echo "Hello API Run" > /mountPoint/hello

Solution n°19

[exercice p. 20]

```
docker inspect -f '{{json .Mounts}}' <mon-conteneur>
[
{
  "Propagation": "",
  "RW": true,
  "Mode": "",
  "Driver": "local",
  "Destination": "/mountPoint",
  "Source":
"/var/lib/docker/volumes/720e2a2478e70a7cb49ab7385b8be627d4b6ec52e6bb33063e4144355d59592a/_da
  "Name": "720e2a2478e70a7cb49ab7385b8be627d4b6ec52e6bb33063e4144355d59592a"
} ]
```

Ici, le volume de données est mappé sur un répertoire de l'hôte Docker, et le répertoire est monté en mode lecture-écriture. Ce répertoire, également appelé volume, est créé automatiquement par le moteur Docker lors du lancement du conteneur.

Lorsque l'on se déplace à l'emplacement du volume ici `/var/lib/docker/volumes/720e2a2478e70a7cb49ab7385b8be627d4b6ec52e6bb33063e4144` on se rend compte que le dossier contient bien notre fichier hello

```
ls -l
/var/lib/docker/volumes/720e2a2478e70a7cb49ab7385b8be627d4b6ec52e6bb33063e4144355d59592a/_dat
```

Remarque

Le résultat aurait été le même si l'on avait utilisé l'option `-v` lors du lancement d'une image ne contenant pas l'instruction `VOLUME`

```
docker run -v /mountPoint mon-image-sans-volume
```

À chaque fois que l'on créé un nouveau conteneur basé sur une image contenant un volume, un nouveau volume va être créé au niveau de l'hôte Docker. Lorsque le conteneur est supprimé, le volume est tout de même conservé.

i l'on souhaite supprimer les volumes associés, il faut utiliser l'option `-v` lors de l'utilisation de la commande `docker rm`

```
docker rm -v 8d22f73b5b46
```

Solution n°20

[exercice p. 22]

Dans l'exemple ci-après, j'ai décidé de partager un volume de mon hôte docker

```
~ docker run --name postgresql -d -v /data/postgresql:/var/lib/postgresql/data
postgres:10
```

Solution n°21

[exercice p. 22]

```
~ docker exec -it postgresql psql -U postgres
psql (10.6 (Debian 10.6-1.pgdg90+1))
Type "help" for help.

postgres=# create table test(data varchar(255));
CREATE TABLE
postgres=# insert into test(data) values('api run');
INSERT 0 1
postgres=# select * from test;
  data
---------
 api run
(1 row)
```

Solution n°22

[exercice p. 22]

```
docker stop postgresql
docker rm postgresl
```

Solution n°23

[exercice p. 22]

Lancement d'un nouveau conteneur

```
~ docker run --name postgresql2 -d -v /data/postgresql:/var/lib/postgresql/data
postgres:10
```

Vérification que la donnée est toujours présente

```
~ docker exec -it postgresql2 psql -U postgres
psql (10.6 (Debian 10.6-1.pgdg90+1))
Type "help" for help.

postgres=# select * from test;
```

```
 data
---------
 api run
(1 row)
```

Solution n°24

[exercice p. 26]

```
FROM debian:buster

RUN apt-get update && \
    apt-get install -y \
      php7.3 \
      php7.3-fpm \
      php7.3-gd \
      php7.3-xml \
      php7.3-pgsql \
      nginx && \
    mkdir -p /run/nginx /var/www /run/php

COPY nginx.conf /etc/nginx/nginx.conf
COPY start.sh /start.sh

RUN echo "cgi.fix_pathinfo = 0;" >> /etc/php/7.3/fpm/php.ini && \
    sed -i -e "s|;daemonize\s*=\s*yes|daemonize = no|g" /etc/php/7.3/fpm/php-fpm.conf && \
    sed -i -e "s|listen\s*=\s*127\.0\.0\.1:9000|listen = /var/run/php-fpm7.sock|g" /etc/php/7.3/fpm/pool.d/www.conf && \
    sed -i -e "s|;listen\.owner\s*=\s*|listen.owner = |g" /etc/php/7.3/fpm/pool.d/www.conf && \
    sed -i -e "s|;listen\.group\s*=\s*|listen.group = |g" /etc/php/7.3/fpm/pool.d/www.conf && \
    sed -i -e "s|;listen\.mode\s*=\s*|listen.mode = |g" /etc/php/7.3/fpm/pool.d/www.conf && \
    chmod +x /start.sh

EXPOSE 80

CMD ["/start.sh"]

```

Solution n°25

[exercice p. 28]

```
version: "3"
networks:
  application:
services:
  postgresql:
      image: postgres:12
      container_name: postgresql
      networks:
        - application
      volumes:
          - /DATA/postgresql:/var/lib/postgresql/data

  nginx:
      container_name: nginx
      image: mon-nginx
      networks:
        - application
      ports:
        - 80:80
```

```
20      volumes:
21          - /DATA/mon-application/:/var/www/html/
22
```

Solution n°26

[exercice p. 28]

```
1 docker exec -it nginx bash
2 root@fslb133aLb ping postgresql
```

Solution n°27

[exercice p. 28]

Ouvrez votre navigateur ou utilisez la commande `curl` et ouvrez la page `127.0.0.1`

Solution n°28

[exercice p. 29]

```
1 version: "3"
2 networks:
3   application:
4 services:
5   traefik:
6       image: traefik:maroilles
7       container_name: traefik
8       networks:
9         - application
10      ports:
11        - "80:80"
12      volumes:
13        - /var/run/docker.sock:/var/run/docker.sock
14        - /DATA/traefik/traefik.toml:/traefik.toml
15
16  postgresql:
17      image: postgres:12
18      container_name: postgresql
19      networks:
20        - application
21      volumes:
22          - /DATA/postgresql:/var/lib/postgresql/data
23
24  nginx:
25      container_name: nginx
26      image: mon-nginx
27      networks:
28        - application
29      volumes:
30          - /DATA/mon-application/:/var/www/html/
31      labels:
32          - "traefik.frontend.rule=Host:mon-application.mondomaine.fr"
33          - "traefik.port=80"
34          - "traefik.enable=true"
```

Solution n°29

[exercice p. 29]

Glossaire

Kernel ou Noyau Linux

Le noyau Linux est un noyau de système d'exploitation de type UNIX[8]. Ce noyau est évidemment utilisé par le système d'exploitation Linux (enfin, plus exactement GNU/Linux[9]) mais aussi par Android. C'est un logiciel libre développé essentiellement en langage C. En savoir plus[10].

8. Unix selon Wikipedia - https://fr.wikipedia.org/wiki/Unix
9. GNU/Linux selon Wikipedia - https://fr.wikipedia.org/wiki/Linux
10. https://fr.wikipedia.org/wiki/Noyau_Linux