TP Docker_3_Images - Ynov DevOps B3

Exercice 1 : Faites parler votre container

Créer un nouveau dossier `cowsay`

Par convention, il faut créer un nouveau dossier pour chaque application. De plus il faut mettre le Dockerfile à la racine de notre projet. Cette bonne pratique s'explique par l'utilisation des repository distant (Github/Gitlab/Jenkins) pour l'intégration continue. Pour builder une application, il est nécessaire d'avoir toutes les informations, à savoir *code, dockerfile, ressources statiques*, au même endroit.

Créer un Dockerfile qui permet de :

- docker run --rm cowsay Hello_World !
- docker run --rm cowsay -f stegosaurus Ynov B3

```
{21:42}/opt/Docker3_Images/cowsay:main ✗ ➲ cat Dockerfile
FROM ubuntu
RUN apt-get update
RUN apt-get -y install cowsay

ENV PATH="$PATH:/usr/games"

ENTRYPOINT [ "cowsay"]
{21:42}/opt/Docker3_Images/cowsay:main ✗ ➲ docker build --network=host -t cowsay .
Sending build context to Docker daemon  2.048kB
Step 1/5 : FROM ubuntu
 ---> ba6acccedd29
Step 2/5 : RUN apt-get update
 ---> Using cache
 ---> f4ead90947e3
Step 3/5 : RUN apt-get -y install cowsay
 ---> Using cache
 ---> 21a4f3a61a8f
Step 4/5 : ENV PATH="$PATH:/usr/games"
 ---> Using cache
 ---> 6317465ffd81
Step 5/5 : ENTRYPOINT [ "cowsay"]
 ---> Using cache
 ---> f4c712bd1942
Successfully built f4c712bd1942
Successfully tagged cowsay:latest
{21:42}/opt/Docker3_Images/cowsay:main ✗ ➲
```

Le Dockerfile est basé sur une image Ubuntu car le paquet **Cowsay** est présent dans les dépôts de paquet officiel. Pour optimiser le build nous aurions pu utiliser une image de base plus légère.

Une image Docker est composé de layer ou de couche. Chaques instructions rajoute une couche à l'image, c'est pourquoi il faut quand c'est possible optimiser les commandes et réduire le nombre d'instructions. Cela réduira le temps de build.

Lors de l'installation de paquet pour personnaliser notre image, il faut absolument rendre les commandes non interactives car nous n'avons pas la main pour renseigner des paramètres (Par exemple : *yes* pour l'installation de paquet) lors du build.

Pour cela nous rajoutons le flag **-y** qui permet d'auto-valider les paquets lors de l'installation.

Nous aurions aussi pu rajouter une variable d'environnement ajouté avant la commandne `apt` : `DEBIAN_FRONTEND=noninteractive`.

Extrait du man `debconf` :

```
   noninteractive
      This is the anti-frontend. It never interacts with you  at  all,
      and  makes  the  default  answers  be used for all questions. It
      might mail error messages to root, but that's it;  otherwise  it
      is  completely  silent  and  unobtrusive, a perfect frontend for
      automatic installs. If you are using this front-end, and require
      non-default  answers  to questions, you will need to preseed the
      debconf database; see the section below  on  Unattended  Package
      Installation for more details.
```

Lors de la commande docker build, le Dockerfile est envoyé comme une archive au daemon docker. Dans notre cas il est présent sur notre machine, mais il existe des cas ou le daemon docker (Le service docker) est sur un serveur remote. Cela permet d'avoir une machine dédiée au build d'image et de libérer de la bande passante sur les postes des développeurs ou des Ops.

Lors du build nous voyons des ID apparaitre :

```bash Step 1/5 : FROM ubuntu ---> ba6acccedd29 Step 2/5 : RUN apt-get update ---> Using cache ---> f4ead90947e3 ... ---> Running in 9a2cd7624ea5
Removing intermediate container 9a2cd7624ea5
---> e02f21d509d8
Step 5/5 : ENTRYPOINT [ "cowsay"]
---> Running in cc226b9d7dfc
Removing intermediate container cc226b9d7dfc
---> 99ef5b94553a
Successfully built 99ef5b94553a
Successfully tagged cowsay:latest
```

```
Chaque ** Step** représente un layer.

A chaque step un container est créer et les instructions sont executés dans ce container. Le container est ensuite commité pour créer une nouvelle image puis supprimé.

Dans notre exemple :
- un container `ba6acccedd29` est créer à partir de l'image de base **Ubuntu**.
- L'instruction `ENTRYPOINT` est exécutée dans le container `cc226b9d7dfc` puis commitée dans `99ef5b94553a` puis supprimé.


![docker run cowsay](https://i.imgur.com/ajnJc0J.png)

### Faut-il utiliser **CMD** ou **ENTRYPOINT** ?

Dans notre DockerFile nous utilisons `ENTRYPOINT` car nous souhaitons fournir des arguments à la commande lancée en tant que PID 1.
Pour rappel : Docker utilise la commande précisé dans l'instructions CMD/ENTRYPOINT comme PID 1 au sein du container.

Pour ces 2 instructions il existe 2 façons d'écrire la commande.

Avec du shell wraping qui lance la commande spécifiée  à l'intérieur d'un shell avec /bin/sh -c:

`ENTRYPOINT cowsay`

Ou en avec la méthode exec, qui permet aux images qui n'ont pas /bin/sh de lancer les commandes :

`ENTRYPOINT ["cowsay"]`

J'ai choisi la méthode *exec* car elle permet de pousser en **PID 1** la commande cowsay et non /bin/sh -c ou bash. Ce qui permet de kill le container à l'aide d'un Ctrl-C qui envoie un `SIGTERM` au PID 1.
```

Exercice 2 : Dockerfile WordSmith

Le but de l'exercice est d'écrire les Dockerfiles pour les 3 containers.

web

words

Sans rajouter la syntaxe `exec` dans la **CMD** du Dockerfile nous ne pouvons pas quitter le container en utilisant le Ctrl-C :

En rajoutant la commande **exec** devant le lancement du programme Java nous pouvons kill le container abec Ctrl-C car le **PID 1** sera le programme java et répondra au Ctrl-C à l'inverse de *bash*.

db

Les trois containers se lancent correctement après les avoir buildé à l'aide de **docker build** :

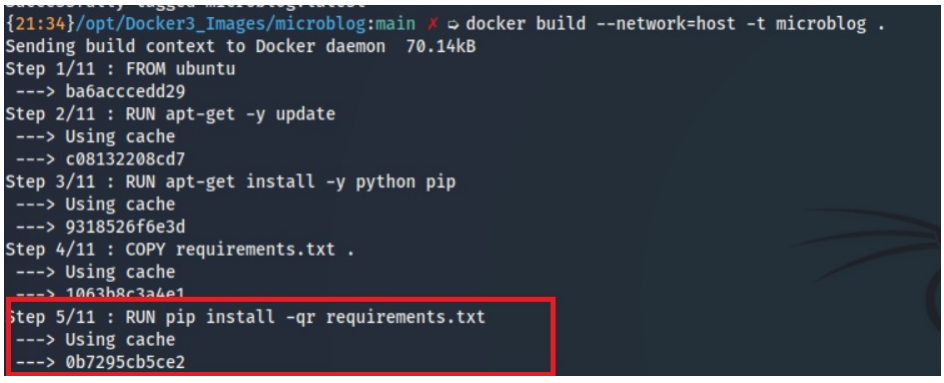
Nous pouvons naviguer sur notre navigateur et accéder au front de l'application (Qui tourne sur le container Web).

Exercice 3: Run Flask App

Créer le Dockerfile dans le dossier microblog qui va permettre de faire tourner l'application Flask :

```yaml
# Flask étant une API Python on aurait pu utiliser l'image Python
# On utilise donc Ubuntu comme image de base
# et on install Python
FROM ubuntu

RUN apt-get -y update
RUN apt-get install -y python pip

# Copie des dépendances avant le dossier pour
# ammélioner le cache de l'image
COPY requirements.txt .

# Installation des dépendances
RUN pip install -qr requirements.txt

# Ajout d'une variable d'environment
# nécessaire à l'utilisation de Flask
ENV FLASK_APP microblog.py

# Copie du projet séparé des dépendances
# pour ne pas réinstaller toutes les dépendances
# lors de la modification du code du projet
COPY ./ /microblog

# L'instruction Workdir permet de se déplacer dans un dossier
# dans le container. Si le dossier n'existe pas il sera crée
WORKDIR /microblog

ENV CONTEXT PROD

# L'instruction EXPOSE permet d'informer Docker
# que le container ecoute sur le réseau spécifié
# Par défaut le port renseigné va être en TCP
# L'instruction EXPOSE ne va pas publier le port
# Pour le publier il faut le renseigner dans le docker run avec -p ou -P
# Cette instruction est plus une forme de documentation
# On peut publier tout les ports exposés avec un -P
# Le port sera exposé en TCP et en UDP à l'aide un port éphémère
EXPOSE 5000

CMD ["./boot.sh"]
```

Construire l'image



On voit bien qu'en rebuildant l'image plusieurs fois, la fonction de cache de Docker est mise en place et on ne réinstalle pas les dépendances inutilement.

Lancer le container en publiant un port de votre hôte.

Accéder à http://localhost:5000

```
{21:40}/opt/Docker3_Images/microblog:main ✗ ➲ docker run  --rm -p 5000:5000 microblog
Running Production Server
[2021-11-11 20:40:49 +0000] [1] [INFO] Starting gunicorn 19.7.1
[2021-11-11 20:40:49 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)
[2021-11-11 20:40:49 +0000] [1] [INFO] Using worker: sync
/usr/lib/python3.8/os.py:1023: RuntimeWarning: line buffering (buffering=1) isn't supported in binary mode, the default buffer size will be use
  return io.open(fd, *args, **kwargs)
[2021-11-11 20:40:49 +0000] [9] [INFO] Booting worker with pid: 9
[2021-11-11 20:40:49,993] INFO in __init__: Microblog startup
172.17.0.1 - - [11/Nov/2021:20:40:53 +0000] "GET / HTTP/1.1" 302 223 "-" "curl/7.74.0"
```

```
{21:40}/etc/apt ➲ curl localhost:5000
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>Redirecting...</title>
<h1>Redirecting...</h1>
<p>You should be redirected automatically to target URL: <a href="/explore">/explore</a>.  If not click the link.#
{21:40}/etc/apt ➲
```

Exercice 4 : Healthcheck

- Créer un nouveau dossier `Healthcheck`
  - On créer un sous dossier pour chaque application. Comme vu précédemment il s'agit d'une bonne pratique pour permettre aux outils CI de gérer les applications/
- Créer un Dockerfile pour déployer l'application `app.py`
  - Lancez l'application avec `python app.py`
- Rajouter un `HEALTHCHECK` dans le Dockerfile pour monitorer l'état de santé du container
  - Voici la commande du HEALTHCHECK `curl --fail http://localhost:5000/health || exit 1`
- Expliquer le code Python ainsi que le lien avec l'instruction HEALTHCHECK de votre Dockerfile

```dockerfile
FROM python:alpine

RUN apk add curl
RUN pip install flask==0.10.1

ADD /app.py /app/app.py
WORKDIR /app

# L'instruction healthcheck permet de définir
# une commande qui va checker l'état de notre container
# pour le renvoyer au daemon docker qui le spécifiera
# dans les commande docker ("docker ps" par exemple)
HEALTHCHECK CMD curl --fail http://localhost:5000/health || exit 1

CMD python app.py
```

Le code python est une simple application FlasK.

Chaque route permet de créer une page web. L'application défini une variable globale `healthy` par défaut à `True`.

Sur la page `/health` l'application vérifie la variable booléenne. Si elle est à `True` le serveur renvoie une réponse HTTP avec un code **200** qui signifie Success. Si la variable est `False` alors le service va renvoyer une réponse **500** qui signifie une erreur côté serveur. La page `/kill` permet de définir la variable `healthy` à `False`.

L'instruction `HEALTHCHECK` va requêter notre application sur la page `/health` toutes les 5 secondes et si l'application renvoie un code **500** (option --fail de curl) alors la commande va faire un `exit 1`.

L'instruction `HEALTHCHECK` indique à Docker comment tester votre container pour vérifier qu'il fonctionne toujours correctement.

L'instruction HEALTHCHECK peut renvoyer 3 codes de retour :

```
0: success - the container is healthy and ready for use
1: unhealthy - the container is not working correctly
2: reserved - do not use this exit code
```

Notre instruction a un code retour 1 lorsqu'on lui envoie `/kill`.

Vous allez constaster visuellement de votre côté, 3 états possibles :

```
Starting: Votre container est en cours de démarrage.
Healthy: La commande de check renvoie un success. Votre container est fonctionnel.
Unhealthy: Votre container ne fonctionne pas correctement !
```

Source : https://www.grottedubarbu.fr/docker-healthcheck/