TP Docker (2)

Dans cette seconde partie du TP, nous allons approfondir notre exploration de Docker en nous concentrant sur l'utilisation de Docker Compose et la création de Dockerfiles. Nous comprendrons comment automatiser le déploiement de services multi-conteneurs avec Docker Compose et comment définir précisément l'environnement de nos conteneurs grâce aux Dockerfiles.

L'objectif de ce TP est de vous doter des compétences pratiques pour utiliser Docker Compose et Dockerfile dans vos projets de développement, afin de créer des environnements de développement cohérents et reproductibles, de simplifier le déploiement de vos applications et d'optimiser leur gestion et leur scalabilité.

Compte rendu sur 5 points à rendre sur Moodle en PDF avant la fin du TP. Merci de ne pas demander à ChatGPT (ou autre) de rédiger votre CR à votre place, faites le TP, justifiez de screenshots et écrivez vos propres phrases !

Docker Compose

Plutôt que d'utiliser des commandes Docker individuelles pour créer et configurer chaque conteneur, Docker Compose utilise un fichier YAML (docker-compose.yml) pour définir les services, les réseaux, les variables et les volumes nécessaires à votre application. Vous allez voir, ça simplifie grandement l'affaire...

Traduction de la commande de lancement de mariadb :

```
docker run -d --name db -e MARIADB_ROOT_PASSWORD=azerty -e MARIADB_DATABASE=im2ag --network lemp1 -v mariadb-data:/var/lib/mysql mariadb
```

L'option -e permet de définir des variables d'environnements.

Fichier docker-compose.yml :

```
--- # pour que l'indentation spécifique à YAML soit appliquée
services:
  db:
    image: mariadb
    container_name: db
    environment:
      MARIADB_ROOT_PASSWORD: azerty
      MARIADB_DATABASE: im2ag
    networks:
      - lemp
    volumes:
      - mariadb-data:/var/lib/mysql
    restart: always
```

Supprimez tous les conteneurs, supprimez également le dossier **LEMP** puis clonez le projet Git suivant : https://im2ag-gitlab.univ-grenoble-alpes.fr/rapacchd/lemp-tp.git

Vous venez de télécharger le dossier "lemp-tp" contenant le docker-compose.yml ci-dessus, vérifiez sa conformité :

```
docker-compose config : depuis le répertoire où se trouve le docker-compose.yml
docker-compose -f docker-compose.yml config : ou en spécifiant le docker-compose.yml
```

Corriger le docker-compose en fonction de l'erreur et déclarer aussi le réseau lemp1. Précisez bien les bons noms dans les déclarations.

Lancez la stack (toujours en mode détaché) :

```
docker compose up -d
docker compose ls : pour lister
```

Mettre dans le compte rendu le screenshot de docker-compose.yml ainsi que la commande lancée. (docker-compose up -d)

Pour arrêter une stack :

```
docker compose down : destruction seulement des conteneurs et des réseaux (non external)
docker compose down -v : avec destruction des volumes non external
```

Complétez le fichier **docker-compose.yml (1)**

Vous allez compléter le fichier YAML avec les deux conteneurs précédents (première partie du TP). Pour le container nginx, **il est nécessaire d'utiliser la clé "depends_on"** pour indiquer la liste des conteneurs dont il dépend (ici php). Pour vous aider, voici les deux commandes pour créer nginx et php, à traduire dans le docker-compose.yml :

```
docker run -d --net lemp1 --name nginx -v ./site:/usr/share/nginx/html -v ./nginx-conf:/etc/nginx/conf.d -p 8080:80 nginx
```

```
docker run -d --net lemp1 --name php -v ./site:/var/www/html:ro drapacch/php-maria
```

Une fois votre compose.yml complété, vérifiez la conformité puis lancez la stack. Tout est ok !?

Le YAML est un langage très puissant et utilisé dans un grand nombre d'outils, notamment dans les domaines du **cloud** et du **DevOps**. Il existe des notions très intéressantes comme, la mutualisation de déclarations et la définition de variable dans des fichiers spécifiques. Ce qui permet de réduire la duplication et améliorer la lisibilité des fichiers de configuration.

Il est aussi possible d'éclater la définition d'une stack sur plusieurs fichiers YAML. Il suffit ensuite de lancer la commandes par :

```
docker-compose -f fichier_1.yml -f fichier_2.yml up -d
docker-compose -f fichier_1.yml -f fichier_2.yml down
```

Mettre dans le compte rendu le screenshot du docker-compose.yml ainsi que la stack lancée. (docker-compose ps)

Complétez le fichier **docker-compose.yml (2)**

Toujours dans votre environnement lemp, ajoutez un container **phpmyadmin** afin de pouvoir gérer vos bases de données plus simplement depuis une interface web. N'oubliez pas de renseigner un port accessible (4201 par exemple).

Justifiez avec un screenshot montrant la stack lancée ainsi qu'un "**curl localhost:4201**" (juste les premières lignes)

Dockerfile

Les Dockerfiles sont des scripts qui automatisent la construction d'images Docker. Ils définissent les étapes nécessaires pour configurer un environnement, installer des dépendances et préparer une application pour le déploiement dans des conteneurs Docker.

Vous allez créer une image Docker pour une application web simple en Python utilisant le framework Flask. L'application sera un serveur web qui affiche un message de bienvenue.

- Créez un répertoire pour votre projet.
- A l'intérieur créez un fichier Python appelé app.py qui contiendra votre application Flask.

```
# app.py
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Bienvenue dans votre application Flask sur Docker!"

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

- Créez un fichier requirements.txt : ce fichier liste toutes les dépendances de votre projet, dans ce cas, juste « Flask » (sans les guillemets).
- Créez maintenant votre Dockerfile (la majuscule est importante) :

```
FROM python

# Définir le répertoire de travail dans le container puis copier requirements.txt
WORKDIR /app
COPY requirements.txt .

# Installer les dépendances Python
RUN pip install --no-cache-dir -r requirements.txt

# Copier le code de l'application dans le conteneur
COPY app.py .

# Exposer le port 5000 pour l'application Flask
EXPOSE 5000

# Définir la commande pour exécuter l'application
CMD ["python", "app.py"]
```

Il va maintenant falloir que vous construisiez votre image docker :

```
docker build -t flask-docker-app .
```

L'image est maintenant construite, vous pouvez vérifier avec un `**docker images**`, vous pouvez donc lancer un container avec cette image (en redirigeant le port 4205 vers le 5000) et vérifier que tout est ok !

Ajoutez une nouvelle instruction RUN à votre Dockerfile :

```
RUN apt-get update && apt-get upgrade -y
```

Construisez de nouveau votre image.

Justifiez avec un screenshot montrant la sortie de la commande "docker build...."

En bref, vous pouvez ajouter de nouvelles instructions qui seront prises en compte dans votre image. Pensez à limiter au maximum le nombre d'instructions RUN, afin d'alléger au mieux votre image.

Ci-dessous les instructions principales pour les Dockerfile, il en existe bien d'autres, Internet vous en donnera les définitions :

FROM : permet de définir l'image source
RUN : permet d'exécuter des commandes dans votre conteneur
ADD : permet d'ajouter des fichiers dans votre conteneur
WORKDIR : permet de définir votre répertoire de travail
EXPOSE : permet de définir les ports d'écoute par défaut
VOLUME : permet de définir les volumes utilisables
CMD : permet de définir la commande par défaut lors de l'exécution de vos conteneurs Docker

Un peu d'exercice

Il existe sur Docker Hub une image **drapacch/tp-docker-im2ag**. Cette image ne fonctionne qu'avec une base de données Redis. Écrivez un fichier docker-compose.yml avec un service **app** basé sur cette image et un service Redis qui aura pour base l'image de la base de données Redis.

- Quel est le log que le container app renvoie ?

Justifiez avec un screenshot montrant les logs.

A partir de votre docker-compose précédemment écrit, ajoutez la variable d'environnement **IM2AG=1** dans votre service app. Supprimez les containers puis relancez la stack.

- De nouveau, quel est le log que le container app renvoie ?

Justifiez avec un screenshot montrant les logs.

Après toutes ces calomnies, veuillez immédiatement corriger ces deux aberrations et les intégrer à une nouvelle image que vous allez construire via un **Dockerfile**. Puis corrigez votre docker-compose et afficher ces corrections via les logs.

PS : au lieu de construire une image avec le `docker build`, vous pouvez directement l'intégrer dans docker-compose.yml avec **"build: ."** à la place de **"image:"**.

Justifiez avec un screenshot montrant les nouveaux logs et la sortie du « docker-compose up -d »

Conclusion

Vous avez maintenant de bonnes notions en Docker. Ceux sont des notions de base qui vous permettront un grand nombre de manipulation et d'être plus à l'aise avec cet outil. J'espère surtout que vous avez compris que le grand intérêt de Docker, est d'avoir en un rien de temps, un environnement de test (ou même de production) complet sur lequel vous pouvez coder et tester vos applications, et surtout, peu importe l'OS de votre machine !

Vous pouvez retrouver sur Internet un grand nombre de tutoriels et même directement des docker-compose.yml pour déployer rapidement des environnements de développement et d'autres applications...

Dernier point, intéressant pour les futurs admins système, sachez que Docker permet également de gérer des clusters de conteneurs grâce à Docker **Swarm** ou **Kubernetes**. Docker Swarm est l'orchestrateur natif de Docker, qui facilite la gestion de clusters de conteneurs. Kubernetes, quant à lui, est une solution d'orchestration de conteneurs plus puissante et bien plus largement adoptée. Elle est idéale pour des déploiements à grande échelle avec des besoins complexes. Enfin, des outils comme **Traefik** peuvent être utilisés pour la gestion des proxys, la répartition de charge et la gestion des certificats SSL au sein de vos clusters de conteneurs. Il existe plein d'autres outils utiles pour la gestion des containers, à vous d'explorer l'univers Docker.....