TRAVAUX PRATIQUES
DOCKER : DOCKER COMPOSE

I. introduction

Dans ce TP, nous allons voir la gestion d'un projet mettant en jeu plusieurs images docker représentant des services interdépendants

II. Premier fichier docker compose

Nous allons démarrer une simple image à l'aide de docker compose. Vous allez écrire le code suivant dans un fichier nommé docker-compose.yml. Ce ficher s'écrit en yaml, comme les fichiers Dockerfile

```
version: "3"
services :
   monservice :
    image : debian:latest
```

On peux voir dans ce document 2 sections, la version de docker-compose à utiliser, ici la version 3, la dernière en date. La seconde section porte sur la liste des services à instancier, à savoir les conteneurs à lancer. Nous allons donc créer un service nommé monservice (ça sera le nom du conteneur) et ce conteneur sera instancié à partir d'une image debian :latest.

les différentes commandes pour gérer les services sont :

- **docker compose up** : permet de démarrer les services. cette commande charge les images si elles ne sont pas dans la cache local, ou fait un build si cela est nécessaire (nous verrons cette commande dans la partie suivante)
- **docker compose down** : permet de supprimer les services
- **docker compose ps** : permet de lister les services. l'option **-a** permet de voir les services qui s'arrêtent

(1) Que constatez vous après avoir lancé **docker compose up** ?

(2) Que vous donne le résultat de **docker compose ps** ? ajoutez l'option **-a** ?

(3) faite un **docker compose down** puis à nouveau le **docker compose ps -a** ? que s'est il passé ?

(4) Vous allez remplacer l'image debian par une image nginx et refaite la même suite de commandes. Pour récupérer la main sur le terminal, dans un autre terminal dans le même répertoire, faite un **docker compose down** puis relancer la commande **docker compose up –detach** (ou **-d**)

III. construction d'une image au travers d'un Dockerfile et de docker compose

Nous allons construire à présent un service qui n'est pas une simple image. Vous allez donc créer un Dockerfile permettant à partir d'une image alpine d'exécuter la commande curl sur le site de l'uha (https ://www.uha.fr) Dans un nouveau répértoire, vous allez créer un fichier Dockerfile permettant de construire cette image, et un fichier docker-compose.yml dans lequel vous allez créer le service en remplaçant la section image, par une section build : qui contient l'item **context :** indiquant le chemin d'accès au Dockerfile permettant de construire l'image. Ensuite vous lancerez **docker compose up** afin d'exécuter le conteneur.

Ensuite vous lancerez les commandes suivantes :

(1) **docker ps -a** et vous identifierez le conteneur lancé dans le compose

(2) **docker inspect #conteneur_id** du conteneur en question

(3) **docker network ls** : que constatez vous par rapport au TP précédent

(4) **docker network inspect #id_network_compose** : pour voir le détail de ce réseaux

(5) refaite un **fdocker network ls** après avoir fait un **docker compose down**. Que constatez vous ?

IV. docker compose et les variables d'environnement

il est parfois nécessaire de passer des variables d'environnement à notre conteneur. pour cela, vous pouvez ajouter une section **environment** qui permet de définir les variables d'environnement sous la forme d'une liste

```
version: "3"
services :
   monservice :
    image : debian:latest
    environment:
     - VAR1 : mavariable1
     - VAR2 : mavariable2
    command : echo $VAR1 $VAR2
```

la partie **command** permet de passer une commande au conteneur. Nous aurions pu aussi utiliser entrypoint comme exécution.

V. docker compose et le réseaux

Nous allons reprendre l'image nginx, mais à présent nous allons ajouter la couche réseaux. En effet, sans paramètre, les ports réseaux ne sont pas exposés. Vous allez donc ajouter la partie ports dans le docker compose dans la section *monservice*

```
version: "3"
services :
   monservice :
    image : nginx:latest
    ports:
     - "80:80"
```

Nous avons aussi la possibilité de créer un sous réseaux particulier plutôt que d'utiliser celui défini par défaut. Dans ce cas, il faut ajouter une section **networks :** au même niveau que la section **services**.

```
version: "3"
services :
   monservice :
      image : nginx:latest
      ports:
       - "80:80"
      networks :
       - monreseau
networks:
   monreseau:
      driver : bridge
```

La ligne de driver est optionnelle car le driver bridge est utilisé par défaut. Vous voyez que nous précisions le réseau à utiliser dans le service. Vous pouvez définir plusieurs réseaux et en activer plusieurs dans un même service. Cela vous permet d'isoler des segments d'une application utilisant des micro-services.

Mettez en place cette solution pour votre service nginx. Vous pouvez aussi créer un réseau au préalable avec les commande **docker network**, puis le rendre disponible dans votre infra définie par **docker compose** en précisant dans la section networks juste le nom du réseaux avec l'option **name**

Dans le sous réseaux, les noms de services sont définis comme les hostnames des machines dans le dns de chaque conteneur, ce qui fait que vous pouvez dialoguer entre machine en utilisant les noms des services. Faite un exemple de 2 conteneur, un basé sur une alpine qui exécute une commande sleep(100) et un autre basé sur une alpine qui fait un ping sur le conteneur précédent.

VI. gestion du stockage dans docker compose

Nous allons voir pour rendre persistantes les données. Pour définir des volumes, nous allons ajouter comme pour les réseaux une section séparer pour définir les volumes à créer, puis une sous section dans les services pour définir quel volume utiliser et sous quel chemin il va être monté dans le conteneur. Cela donne l'exemple suivant

```
version: "3"
services :
   monservice :
    image : debian:latest
    command : echo "bonjour le monde" > /tmp/bonjour.txt
    volumes:
      - monvolume:/tmp:rw
volumes:
   monvolume :
     driver : local
```

Faite de même pour votre conteneur nginx avec le répertoire contenant les fichiers html par défaut et modifier le.

VII. interdépendance des services

Pour finir, nous pouvons avoir des services dépendant d'autre services, comme une application démarrée dans un conteneur nécessitant une base de données active pour pouvoir être lancé. Cette base de données étant dans un autre conteneur.

pour indiquer cette dépendance, vous allez ajouter dans la section du service dépendant la ligne suivante

```
version: "3"
services :
   monservice :
     image : debian:latest
     depends_on: autre_service
   autre_service :
```

l'autre service sera donc démarré avant le service qui en dépend.

VIII. Application

Vous allez donc créer le docker compose permettant de déployer une application Django, s'appuyant sur une base de données mysql déployé dans un autre conteneur, est accessible dans un serveur web nginx déployé dans un 3ème conteneur. Ces trois conteneur seront sur un même sous réseaux, seul le port 80 du serveur nginx sera exposé sur le port 80 de l'hôte. Le serveur applicatif gunicon sera installé sur le conteneur déployant Django. le volume de la base de données sera persistant. Pour la base de données, vous regarderez la façon de passer les variables d'environnement permet de configurer ce service sur la page de l'image officielle sur le hub docker.