LAB 7 – docker-compose

Installation de l'utilitaire docker-compose sur Centos 7

Afin d'obtenir la dernière version, prenez la tête des documents Docker et installez Docker Compose à partir du binaire dans le référentiel GitHub de Docker.

- Vérifiez la version actuelle et si nécessaire, mettez-la à jour dans la commande ci-dessous:

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.27.4/docker-compose-
$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

- Ensuite, définissez les autorisations pour rendre l'exécutable binaire:

```
sudo chmod +x /usr/local/bin/docker-compose
```

- Ensuite, vérifiez que l'installation a réussi en vérifiant la version:

```
[root@localhost ~]# docker-compose --version
docker-compose version 1.27.4, build 40524192
```

Exécution d'un conteneur avec Docker Compose

Exemple 1 – Hello world

- Tout d'abord, créez un répertoire pour notre fichier YAML:
- Créez maintenant le fichier YAML docker-compose.yml
- Mettez le contenu suivant dans le fichier:

```
my-test:

  image: hello-world
```

- Exécutez la commande suivante pour créer le conteneur:

```
[root@localhost ~]# docker-compose up
```

```
[root@localhost ~]# docker-compose up
Pulling my-test (hello-world:)...
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:31b9c7d48790f0d8c50ab433d9c3b7e17666d6993084c002c2ff1ca09b96391d
Status: Downloaded newer image for hello-world:latest
Creating root_my-test_1 ... done
Attaching to root_my-test_1
my-test_1  |
my-test_1  | Hello from Docker!
my-test_1  | This message shows that your installation appears to be working correctly.
my-test_1  |
my-test_1  | To generate this message, Docker took the following steps:
my-test_1  |  1. The Docker client contacted the Docker daemon.
my-test_1  |  2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
my-test_1  |     (amd64)
my-test_1  |  3. The Docker daemon created a new container from that image which runs the
my-test_1  |     executable that produces the output you are currently reading.
my-test_1  |  4. The Docker daemon streamed that output to the Docker client, which sent it
my-test_1  |     to your terminal.
my-test_1  |
my-test_1  | To try something more ambitious, you can run an Ubuntu container with:
my-test_1  |  $ docker run -it ubuntu bash
my-test_1  |
my-test_1  | Share images, automate workflows, and more with a free Docker ID:
my-test_1  |  https://hub.docker.com/
my-test_1  |
my-test_1  | For more examples and ideas, visit:
my-test_1  |  https://docs.docker.com/get-started/
my-test_1  |
root_my-test_1 exited with code 0
```

- Afficher votre groupe de conteneurs pour le projet hello-world
- Arrêter tous les conteneurs du projet hello-world

Exemple 2 : Une application Web Python simple exécutée sur Docker Compose

Présentation

L'application utilise le framework Flask et gère un compteur d'accès dans Redis. Bien que l'exemple utilise Python, les concepts présentés ici devraient être compréhensibles même si vous ne le connaissez pas.

Étape 1: Configuration

Définissez les dépendances de l'application.

- Créez un répertoire pour le projet:

```
mkdir composetest
cd composetest
```

- Créez un fichier appelé **app.py** dans le répertoire de votre projet et collez-le dans:

```
import time
import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

Dans cet exemple, **redis** est le nom d'hôte du conteneur redis sur le réseau de l'application. Nous utilisons le port par défaut pour Redis, 6379.

- Créez un autre fichier appelé **requirements.txt** dans le répertoire de votre projet

```
Flask
redis
```

Étape 2: Créez un Dockerfile

Dans cette étape, vous écrivez un **Dockerfile** qui crée une image Docker. L'image contient toutes les dépendances requises par l'application Python, y compris Python lui-même.

Dans le répertoire de votre projet, créez un fichier nommé **Dockerfile** et collez ce qui suit:

```
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

Cela indique à Docker de:

- Créez une image en commençant par l'image Python 3.7.
- Définissez le répertoire de travail sur /code.
- Définissez les variables d'environnement utilisées par la flaskcommande.
- Installez gcc et d'autres dépendances
- Copiez requirements.txtet installez les dépendances Python.
- Ajoutez des métadonnées à l'image pour décrire que le conteneur écoute sur le port 5000
- Copiez le répertoire actuel .du projet dans le répertoire .de travail de l'image.
- Définissez la commande par défaut du conteneur sur flask run.

Étape 3: définir les services dans un fichier de composition

Créez un fichier appelé **docker-compose.yml** dans le répertoire de votre projet et collez ce qui suit:

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

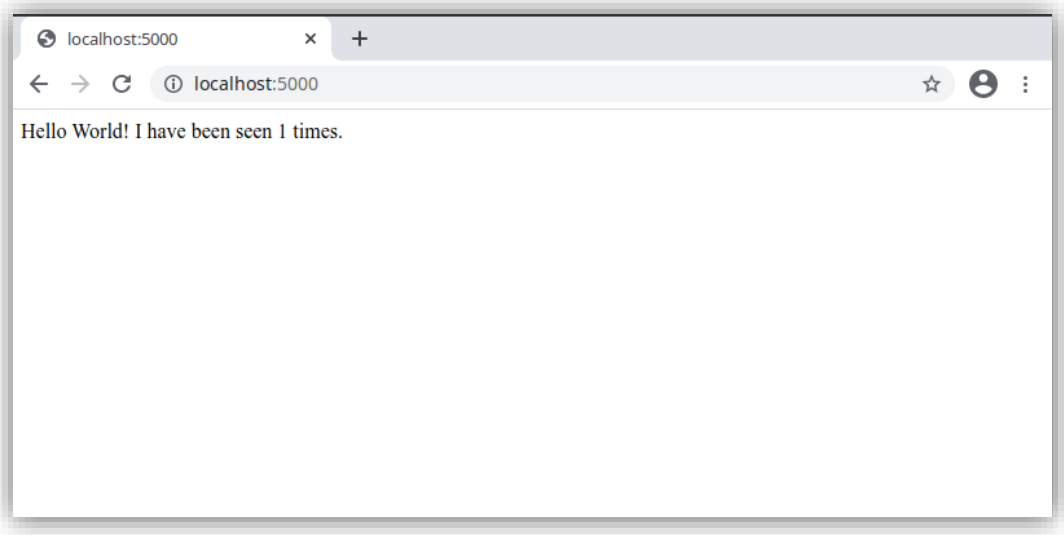
Ce fichier Compose définit deux services: **web** et **redis**.

Étape 4: Créez et exécutez votre application avec Compose

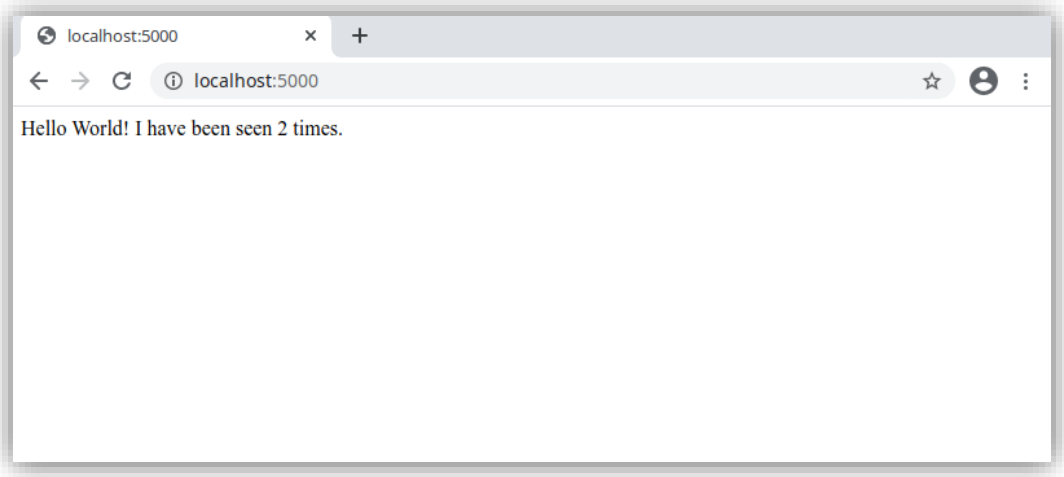
Depuis le répertoire de votre projet, démarrez votre application en exécutant docker-compose up.

```
docker-compose up
```

Entrez http: // localhost: 5000 / dans un navigateur pour voir l'application en cours d'exécution.



Actualiser la page. Le nombre doit augmenter.



Étape 5: Modifiez le fichier de composition pour ajouter un montage de liaison

Modifiez **docker-compose.yml** dans votre répertoire de projet pour ajouter un montage de liaison pour le web service:

```
version: "3.9"

services:
```

```
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

La nouvelle clé **volumes** monte le répertoire du projet (répertoire courant) sur l'hôte à l'intérieur du **/code** du conteneur, vous permettant de modifier le code à la volée, sans avoir à reconstruire l'image.

La clé **environment** définit la variable d'environnement **FLASK_ENV**, qui indique **flask run** de s'exécuter en mode développement et de recharger le code en cas de modification. Ce mode ne doit être utilisé qu'en développement.

Étape 6: et exécutez l'application avec Compose

À partir du répertoire de votre projet, tapez **docker-compose up** pour créer l'application avec le fichier Compose mis à jour et exécutez-le.

```
docker-compose up
```

Vérifiez à nouveau le Hello Worldmessage dans un navigateur Web et actualisez-le pour voir l'incrémentation du nombre.

Étape 7: Mettez à jour l'application

Étant donné que le code de l'application est maintenant monté dans le conteneur à l'aide d'un volume, vous pouvez apporter des modifications à son code et voir les modifications instantanément, sans avoir à reconstruire l'image.

Modifiez le message d'accueil app.py et enregistrez-le. Par exemple, modifiez le **`Hello World!`** message en **`Hello from Docker!:`**

```
return 'Hello from Docker! I have been seen {} times.\n'.format(count)
```

Actualisez l'application dans votre navigateur. Le message d'accueil doit être mis à jour et le compteur doit toujours être incrémenté.

Exercices d'applications

Exercice 1 : voting-app

- Cloner le projet de l'application voting-app : https://github.com/dockersamples/example-voting-app
- Lancez le service voting-app à l'aide du fichier « docker-compose-simple.yml »
- Analysez le contexte d'exécution de l'application
- Arrêtez l'application
- Lancez le service voting-app à l'aide du fichier « docker-compose.yml »
- Analysez le contexte d'exécution de l'application

Exercice 2 : Application nodeJS avec Mango DB

1. Clonez le projet https://github.com/medsalahmeddeb/docker-compose-node-mango

Voici la structure du projet :

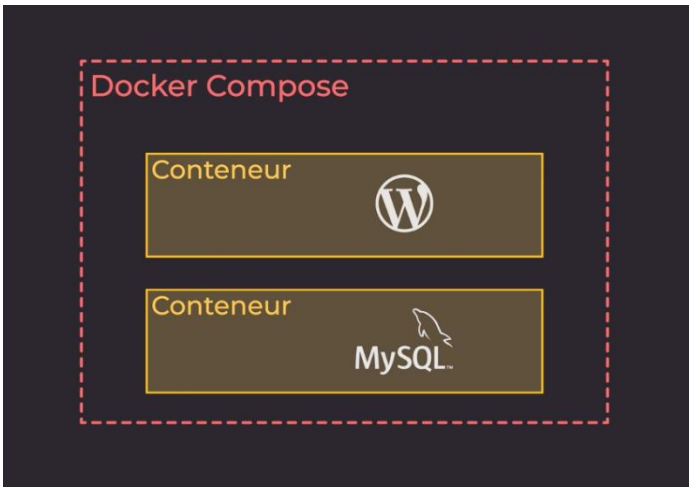
```
.
├── index.js
├── modèles
│ .. └── Item.js
├── package-lock.json
├── package.json
└── vues
    └── index.ejs
```

2. Créez un Dockerfile permettant d'empaqueter l'application nodejs selon les caractéristiques suivantes :
    - Image de base : node
    - Copiez les fichiers package.json et package-lock.json à l'intérieure du conteneur
    - Installer l'environnement d'exécution
    - Démarrer npm
3. Créez un fichier docker-compose pour créer deux services node et mango
    - ***Node*** :
        - Basé sur le Dockerfile,
        - Publier le port 3000 du conteneur
        - Mappant un volume pour partager le code de l'application
    - ***Mango*** :
        - Basé sur l'image mango

- Publier le port 27017 du conteneur
- Mappant un volume vers le dossier /data/db du conteneur

Exercice 3 : wordpress

Préparation un fichier docker-compose pour l'application wordpress



services:

wordpress:

db:

Service 1 : wordpress

- Basé sur l'image wordpress
- Utilise le port 80
- Requiert les variables d'environnements suivantes :
  - WORDPRESS_DB_HOST
  - WORDPRESS_DB_USER
  - WORDPRESS_DB_PASSWORD
  - WORDPRESS_DB_NAME
- Dépend du service DB

Service 2 : BD

- Basé sur l'image mysql
- Requiert les variables d'environnements suivantes :
  - MYSQL_ROOT_PASSWORD
  - MYSQL_DATABASE
  - MYSQL_USER
  - MYSQL_PASSWORD
- Exige un volume pour persister les données de la BD

Exemples (Bonus)

https://github.com/stefanprodan/dockprom

https://github.com/Einsteinish/Docker-Compose-Prometheus-and-Grafana