TRAVAUX PRATIQUES

TP 04 : *Volumes et Réseaux Docker*

Portainer

```
docker volume create portainer_data
docker run --detach --name portainer \
    -p 9000:9000 \
    -v portainer_data:/data \
    -v /var/run/docker.sock:/var/run/docker.sock \
    portainer/portainer-ce
```

PARTIE 1 : DOCKER NETWORKING

Pour expérimenter avec le réseau, nous allons lancer une petite application nodejs d'exemple (moby-counter) qui fonctionne avec une file (*queue*) redis (comme une base de données mais pour stocker des paires clé/valeur simples).

Récupérons les images depuis Docker Hub:

- `docker image pull redis:alpine`
- `docker image pull russmckendrick/moby-counter`
- Lancez la commande `ip a | tee /tmp/interfaces_avant.txt` pour lister vos interfaces réseau et les écrire dans le fichier

Pour connecter les deux applications créons un réseau manuellement:

- `docker network create moby-network`

Docker implémente ces réseaux virtuels en créant des interfaces. Lancez la commande `ip a | tee /tmp/interfaces_apres.txt` et comparez (`diff /tmp/interfaces_avant.txt /tmp/interfaces_apres.txt`). Qu'est-ce qui a changé ?

Maintenant, lançons les deux applications en utilisant notre réseau :

- `docker run -d --name redis --network <réseau> redis:alpine`
- `docker run -d --name moby-counter --network <réseau> -p 80:80 russmckendrick/moby`

- Visitez la page de notre application. Qu'en pensez vous ? Moby est le nom de la mascotte Docker 🐬 😊. Faites un motif reconnaissable en cliquant.

Comment notre application se connecte-t-elle au conteneur redis ? Elle utilise ces instructions JS dans son fichier `server.js` :

```
var port = opts.redis_port || process.env.USE_REDIS_PORT || 6379;
var host = opts.redis_host || process.env.USE_REDIS_HOST || "redis";
```

En résumé par défaut, notre application se connecte sur l'hôte `redis` avec le port `6379`

Explorons un peu notre réseau Docker.

- Exécutez (`docker exec`) la commande `ping -c 3 redis` à l'intérieur de notre conteneur applicatif (`moby-counter` donc). Quelle est l'adresse IP affichée ?

```
docker exec moby-counter ping -c3 redis
```

- De même, affichez le contenu des fichiers `/etc/hosts` du conteneur (c'est la commande `cat` couplée avec `docker exec`). Nous constatons que Docker a automatiquement configuré l'IP externe **du conteneur dans lequel on est** avec l'identifiant du conteneur. De même, affichez `/etc/resolv.conf` : le résolveur DNS a été configuré par Docker. C'est comme ça que le conteneur connaît l'adresse IP de `redis`. Pour s'en assurer, interrogeons le serveur DNS de notre réseau `moby-network` en lançant la commande `nslookup redis 127.0.0.11` toujours grâce à `docker exec` : `docker exec moby-counter nslookup redis 127.0.0.11`

- Créez un deuxième réseau `moby-network2`
- Créez une deuxième instance de l'application dans ce réseau : `docker run -d --name moby-counter2 --network moby-network2 -p 9090:80 russmckendr`
- Lorsque vous pingez `redis` depuis cette nouvelle instance `moby-counter2` , qu'obtenez-vous ? Pourquoi ?

Vous ne pouvez pas avoir deux conteneurs avec les mêmes noms, comme nous l'avons déjà découvert. Par contre, notre deuxième réseau fonctionne complètement isolé de notre premier réseau, ce qui signifie que nous pouvons toujours utiliser le nom de domaine `redis`. Pour ce faire, nous devons spécifier l'option `--network-alias` :

- Créons un deuxième redis avec le même domaine: `docker run -d --name redis2 --network moby-network2 --network-alias redis redis:a`
- Lorsque vous pingez `redis` depuis cette nouvelle instance de l'application, quelle IP obtenez-vous ?
- Récupérez comme auparavant l'adresse IP du nameserver local pour `moby-counter2` .
- Puis lancez `nslookup redis <nameserver_ip>` dans le conteneur `moby-counter2` pour tester la résolution de DNS.
- Vous pouvez retrouver la configuration du réseau et les conteneurs qui lui sont reliés avec `docker network inspect moby-network2` . Notez la section IPAM (IP Address Management).
- Arrêtons nos conteneurs : `docker stop moby-counter2 redis2` .
- Pour faire rapidement le ménage des conteneurs arrêtés lancez `docker container prune` .
- De même `docker network prune` permet de faire le ménage des réseaux qui ne sont plus utilisés par aucun conteneur.

PARTIE 2 : VOLUMES DOCKER

Introduction aux volumes

- Pour comprendre ce qu'est un volume, lançons un conteneur en mode interactif et associons-y le dossier `/tmp/data` de l'hôte au dossier `/data` sur le conteneur :

```
docker run -it -v /tmp/data:/data ubuntu /bin/bash
```

- Dans le conteneur, navigons dans ce dossier et créons-y un fichier :

```
cd /data/
touch testfile
```

- Sortons ensuite de ce conteneur avec la commande `exit`

```
exit
```

- Après être sorti·e du conteneur, listons le contenu du dossier **sur l'hôte** avec la commande suivante ou avec le navigateur de fichiers d'Ubuntu :

```
ls /tmp/data/
```

Le fichier `testfile` a été crée par le conteneur au dossier que l'on avait connecté grâce à `-v /tmp/data:/data`

L'app `moby-counter` , Redis et les volumes

Pour ne pas interférer avec la deuxième partie du TP :

- Stoppez tous les conteneurs redis et moby-counter avec `docker stop` ou avec Portainer.
- Supprimez les conteneurs arrêtés avec `docker container prune`
- Lancez `docker volume prune` pour faire le ménage de volume éventuellement créés dans les TPs précédent

- Lancez aussi `docker network prune` pour nettoyer les réseaux inutilisés

Passons à l'exploration des volumes:

- Recréez le réseau `moby-network` et les conteneurs `redis` et `moby-counter` à l'intérieur :

```
docker network create moby-network
docker run -d --name redis --network moby-network redis
docker run -d --name moby-counter --network moby-network -p 8000:80
russmckendrick/moby-counter
```

- Visitez votre application dans le navigateur. **Faites un motif reconnaissable en cliquant.**

Récupérer un volume d'un conteneur supprimé

- supprimez le conteneur `redis` : `docker stop redis` puis `docker rm redis`
- Visitez votre application dans le navigateur. Elle est maintenant déconnectée de son backend.
- Avons-nous vraiment perdu les données de notre conteneur précédent ? Non ! Le Dockerfile pour l'image officielle Redis ressemble à ça :

```
FROM alpine:3.5

RUN addgroup -S redis && adduser -S -G redis redis
RUN apk add --no-cache 'su-exec>=0.2'
ENV REDIS_VERSION 3.0.7
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-3.0.7.tar.gz
ENV REDIS_DOWNLOAD_SHA e56b4b7e033ae8dbf311f9191cf6fdf3ae974d1c
RUN set -x \
    && apk add --no-cache --virtual .build-deps \
        gcc \
        linux-headers \
        make \
        musl-dev \
        tar \
    && wget -O redis.tar.gz "$REDIS_DOWNLOAD_URL" \
    && echo "$REDIS_DOWNLOAD_SHA *redis.tar.gz" | sha1sum -c - \
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -r /usr/src/redis \
    && apk del .build-deps

RUN mkdir /data && chown redis:redis /data
VOLUME /data
WORKDIR /data
COPY docker-entrypoint.sh /usr/local/bin/
RUN ln -s usr/local/bin/docker-entrypoint.sh /entrypoint.sh # backwards compat
ENTRYPOINT ["docker-entrypoint.sh"]
EXPOSE 6379
CMD [ "redis-server" ]
```

Notez que, vers la fin du fichier, il y a une instruction `VOLUME` ; cela signifie que lorque notre conteneur a été lancé, un volume “caché” a effectivement été créé par Docker.

Beaucoup de conteneurs Docker sont des applications *stateful*, c’est-à-dire qui stockent des données. Automatiquement ces conteneurs créent des volument anonymes en arrière plan qu’il faut ensuite supprimer manuellement (avec rm ou prune).

- Inspectez la liste des volumes (par exemple avec Portainer) pour retrouver l’identifiant du volume caché. Normalement il devrait y avoir un volume `portainer_data` (si vous utilisez Portainer) et un volume anonyme avec un hash.
- Créez un nouveau conteneur redis en le rattachant au volume redis “caché” que vous avez retrouvé (en copiant l’id du volume anonyme) :
  `docker container run -d --name redis -v <volume_id>:/data --network moby-network r`

- Visitez la page de l'application. Normalement un motif de logos *moby* d'une précédente session devrait s'afficher (après un délai pouvant aller jusqu'à plusieurs minutes)
- Affichez le contenu du volume avec la commande : `docker exec redis ls -lha /data`

Bind mounting

Finalement, nous allons recréer un conteneur avec un volume qui n'est pas anonyme.

En effet, la bonne façon de créer des volumes consiste à les créer manuellement (volumes nommés) : `docker volume create redis_data` .

- Supprimez l'ancien conteneur `redis` puis créez un nouveau conteneur attaché à ce volume nommé : `docker container run -d --name redis -v redis_data:/data --network moby-network re`

Lorsqu'un répertoire hôte spécifique est utilisé dans un volume (la syntaxe `-v HOST_DIR:CONTAINER_DIR`), elle est souvent appelée **bind mounting**. C'est quelque peu trompeur, car tous les volumes sont techniquement "bind mounted". La différence, c'est que le point de montage est explicite plutôt que caché dans un répertoire géré par Docker.

- Lancez `docker volume inspect redis_data` .

Supprimer les volumes et réseaux

- Pour nettoyer tout ce travail, arrêtez d'abord les différents conteneurs `redis` et `moby-counter` .
- Lancez la fonction `prune` pour les conteneurs d'abord, puis pour les réseaux, et enfin pour les volumes.

Comme les réseaux et volumes n'étaient plus attachés à des conteneurs en fonctionnement, ils ont été supprimés.

Généralement, il faut faire beaucoup plus attention au prune de volumes (données à perdre) qu'au `prune` de conteneurs (rien à perdre car immutable et en général dans le registry).

Facultatif : Packagez votre propre app

Vous possédez tous les ingrédients pour packager l'app de votre choix désormais ! Récupérez une image de base, basez-vous sur un Dockerfile existant s'il vous inspire, et lancez-vous !

Travaux Pratiques

Docker Compose

2022/2023

Notions Abordées :

- Développement en utilisant les conteneurs Docker
- Docker network
- Docker volume
- Docker Compose

1. Envoyez votre image sur DockerHub

- Créer un compte sur le dépôt Dockerhub
- à partir de votre ligne de commande, connectez-vous à votre dépôt Dockerhub: ***docker login registry-1.docker.io***
- Envoyez votre image à votre dépôt Dockerhub ***docker push yourhubusername/imagename***. **note** : Pour envoyer votre image docker sur votre dépôt Dockerhub elle doit respecter le format suivant : votre-nom-dockerhub/imagename

2. Développer en utilisant les conteneurs Docker :

Pour réaliser cette partie du tp, on va utiliser une application CRUD implémentée en utilisant Maven et mongodb et mongo-express.

On va commencer par lier les deux conteneurs mongodb et mongo-express dans un même docker network

(a) Lister les "docker networks" que vous avez sur votre machine : *docker network ls*

(b) Créer un nouveau network nommé *maven-mongo-network*: **docker network create maven-mongo-network**

(c) Exécutez le conteneur de la base de donnée MongoDB sur le port 27017 en mode détaché :
docker run -d
-p 27017:27017
-e MONGO_INITDB_ROO_USERNAME=admin
-e MONGO_INITDB_ROOT_PASSWORD=pass
- -name mymongo
- -network mongo-network
mongo

(d) Exécutez du conteneur de mongo-express sur le port 8081 en mode détaché
docker run –d
-p 8081:8081
-e ME_CONFIG_MONGODB_ADMINUSERNAME=admin
-e ME_CONFIG_MONGODB_ADMINPASSWORD =pass

- -name mymongo-express
- -network mongo-network
-e ME_CONFIG_MONGODB_SERVER = mymongo
mongo-express

(e) Lancez mongo-express sur votre navigateur sur le port 8081

(f) Exécutez votre application.

(g) Stopper et redémarrez Mongodb et mongo-express. Qu'est-ce que vous remarquez?

(h) Utilisez Docker Volume pour la persistance de vos données docker run ... -v /home/hostpersist:/data/db

3. Docker Compose :

Dans la partie précédente, nous avons utilisé des commandes pour exécuter les différents conteneurs de notre application. Cette façon de faire est un peu compliquée, particulièrement, si on a un bon nombre de conteneurs à exécuter. Pour faire plus simple, il existe un outil qui nous permet d'exécuter plusieurs conteneur docker ainsi que toutes leurs configurations. Cet outil est **Docker Compose**

(a) Ecrivez le docker compose permettant d'exécuter les deux conteneurs concernant mongodb, mongo-express.

```
version: "3"
services:
 mymongodb:
  image: mongo
  container_name: mymongodb
  ports:
   - 27017:27017
  environment:
   - MONGO_INITDB_ROOT_USERNAME=admin
   - MONGO_INITDB_ROOT_PASSWORD=pass
 mymongodb-express:
  image: mongo-express
  container_name: mymongo-express
  ports:
   - 8081:8081
  environment:
   - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
   - ME_CONFIG_MONGODB_ADMINPASSWORD =pass
   - ME_CONFIG_MONGODB_SERVER = mymongo
```

(b) Exécutez votre fichier Docker-Compose en utilisant la commande suivante:

docker-compose -f name_Docker-Compose_file.yaml up

(c) Exécutez votre application.

(d) Si vous voulez arrêter les services de votre docker Compose
docker-compose -f name_Docker-Compose_file.yaml down

(e) Dans le même fichier Docker-compose, lancer également l'exécution du conteneur de votre propre application .