Formation Docker

Version 0.5

Julien Robert

juin 26, 2025

Table des matières

1

Tables des matières

1.1 Introduction

Support de présentation : https://soleil-docker.jrobert-orleans.fr/01-Intro_docker (et en pdf : `Intro`)

Lien vers le QCM de début et de fin de formation : TODO

Lien vers le questionnaire de satisfaction (à remplir en fin de formation) : https://forms.office.com/e/TFmN6k7uDV

Ce document est téléchargeable en PDF : `formationdocker.pdf`.

1.1.1 Présentation

Les points qui vous ont été présentés :
- La problématique de la mise en production
- Virtualisation
- Conteneurisation
- Ce qu'apporte Docker
- Démo

1.1.2 Docker, premiers pas

Installation

La documentation pour l'installation est disponible ici : https://docs.docker.com/engine/install/
- Si vous êtes sous ubuntu, choisissez la méthode décrite ici : https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository
- Si vous êtes sous debian, choisissez la méthode décrite ici : https://docs.docker.com/engine/install/debian/#install-using-the-repository
- Vous pouvez aussi installer docker-desktop en suivant la documentation décrite ici : https://docs.docker.com/desktop/ <https://docs.docker.com/desktop/>`_

Vérification de l'installation

Après l'installation, vérifiez que tout fonctionne correctement (si vous utilisez docker desktop, ne mettez pas sudo) :

```
sudo docker version

sudo docker info

sudo docker run --rm hello-world
```

Post-installation (si vous ne passez pas par docker-desktop)

Pour pouvoir lancer la commande docker sans avoir à utiliser sudo, suivez les instructions *Manage Docker as a non-root user* du lien suivant, résumées ci-dessous : https://docs.docker.com/engine/install/linux-postinstall/

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

Attention : cela donne des droits *élevés* à votre utilisateur. Un utilisateur qui appartient au groupe *docker* a les mêmes possibilités que le superutilisateur.

A l'issue de cette étape, vous pouvez lancer la commande docker sans sudo :

```
docker run --rm hello-world
```

Premier conteneur

Ouvrez deux terminaux, l'un pour lancer un conteneur, et l'autre pour observer le système.

Dans le premier terminal, lancez un conteneur ubuntu en exécutant : `docker run -it ubuntu bash`

Dans l'autre terminal, listez les conteneurs actifs avec : `docker ps`.

Vous pouvez agrémenter cette commande de l'option `--size` pour observer l'espace disque pris par les conteneurs : `docker ps --size`. Cela affiche 2 valeurs : la taille prise par l'image et la taille prise par les données que vous avez ajoutées.

Faites un tour à l'intérieur du conteneur dans votre premier terminal :

- constatez que l'utilisateur est root (avec la commande `whoami`).
- constatez que le système de fichiers n'est pas celui de la machine hote (par exemple avec `ls /home/`).
- constatez que vous pouvez installer des logiciels (`apt update` puis `apt install vim`). Remarquez l'augmentation de l'espace utilisé par votre conteneur.
- constatez que vous ne pouvez pas rebooter la machine, ni tuer de processus qui ne sont pas dans le conteneur, etc.

Interaction avec le processus du conteneur

Par défaut la sortie standard d'un conteneur est « branchée » dans votre terminal. Par contre, l'entrée standard ne l'est pas, il faut pour cela utiliser l'option *-i* de *docker run*. Pour le constater, on va utiliser la commande *cat* qui recopie sur la sortie standard ce qu'elle reçoit sur l'entrée standard.

- Essayez de lancer `docker run ubuntu cat`. Remarquez que la commande s'arrête immédiatement : cat n'arrive pas à lire quoi que ce soit sur l'entrée standard et termine.
- Essayez de lancer `docker run -i ubuntu cat`. Cette fois ci on peut envoyer du texte à cat.

Mais ce n'est pas toujours suffisant, pour avoir une interaction complète avec notre conteneur, on ajoutera l'option « -t » qui permet d'émuler un terminal. C'est ce qu'on a fait précédemment avec : `docker run -it ubuntu bash`

On peut choisir de ne pas attacher un conteneur à notre terminal avec l'option *-d* `docker run -it -d ubuntu bash` et on peut rattacher le terminal avec `docker attach` puis se détacher avec la séquence `ctrl-pq`.

Cycle de vie d'un conteneur

Un conteneur peut être dans l'un des états suivants :

- Created (docker create) : vient d'être créé mais n'est pas lancé.
- Running (docker run).
- Restarting (docker restart).
- Exited (docker kill/stop) : les processus du conteneur sont tous « morts ».
- Paused (docker pause / docker unpause) : les processus sont tous en pause, prend des ressources mémoire.
- Dead : en cours de suppression.

À vous

- Dans quel état se trouve votre conteneur actuel (pour l'observer, utilisez `docker ps`) ?
- Tuez votre conteneur avec `docker kill PREFIXE_ID_CONTENEUR` où PREFIXE_ID_CONTENEUR est le début de l'ID de votre conteneur.
- Pour constater que ça a fonctionné, vous ne devez plus le voir apparaître lorsque vous faites `docker ps`.
- Pourtant il existe encore, mais est dans l'état « exited ». Observez son existence avec `docker ps -a` ou `docker ps --all`.
- Relancez le avec `docker start`. Constatez que votre conteneur a été relancé.
- Pour terminer, essayez de supprimer votre conteneur avec `docker rm` .

Récapitulatif

---

commandes à retenir

- `docker ps -a`
- `docker run -it -d IMAGE COMMANDE`
- `docker kill`
- `docker rm`

À vous

— Lancez la commande `sleep 2d` dans un conteneur ubuntu.
— Quelle est la taille prise par l'image « alpine » ?
— L'image alpine contient-elle un exécutable pour bash ? Quel shell peut y être utilisé ?
— Que fait le code suivant ?

```
id=$(docker container create -it ubuntu sleep 2d)
docker container start $id
```

Quelques autres petites options de docker run

En vous aidant du man de *docker-run*, essayez les options suivantes :
— `--restart`
— `--rm`
— `-h,--hostname`
— `-m,--memory` (pour le tester, inspirez vous de la commande suivante `bash -c "cat /dev/zero | head -c 100M | tail` : elle crée un processus qui consomme environ 100M de mémoire)
— `--name`
— `-u,--user`

Nettoyage

Pour supprimer tous les conteneurs terminés, vous pouvez utiliser `docker container prune`

1.1.3 Dockerhub

Présentation

DockerHub est le « registry public » officiel de Docker. Il permet de :
— Stocker et partager des images Docker
— Automatiser la construction d'images
— Intégrer avec des outils CI/CD
— Collaborer en équipe

Types d'images

Sur DockerHub, on trouve différents types d'images (cf : https://docs.docker.com/docker-hub/repos/manage/trusted-content/) :
— **Images officielles** : Maintenues par Docker ou les éditeurs officiels
— **Images vérifiées** : Créées par des Docker Verified Publishers
— **Images communautaires** : Créées par la communauté
Lorsque vous avez fait `docker run ubuntu`, l'image *ubuntu :latest* a été téléchargée à partir de docker hub.

Pour en savoir plus sur le contenu de cette image, on a la documentation ici : https://hub.docker.com/_/ubuntu .

— On remarque par exemple que cette image propose le tag « 24.04 », probablement la mouture de 2024 de ubuntu. On peut la lancer en faisant : `docker run -it ubuntu:24.04` . Regardez le contenu de */etc/os-release*.

Sur le site dockerhub : https://hub.docker.com/ dans l'onglet « Explore », on y trouvera tout un tas d'images.

— Pouvez vous trouver une image avec un interpréteur python ?
— Pouvez vous trouver une image avec un interpréteur python en version 2.7 ?
— Est-ce qu'il y a une image pour postgresql ? openjdk ? nextcloud ? nginx ? docker ? D'autres choses que vous aimeriez y trouver ?

Limites et alternatives

— **Limites de DockerHub gratuit (cf** [https://docs.docker.com/docker-hub/usage/) :]
  — 100 pulls anonymes / 6h
  — 200 pulls authentifiés / 6h
  — Un seul build concurrent
— **Alternatives à DockerHub :**
  — GitHub Container Registry
  — Google Container Registry
  — Amazon Elastic Container Registry
  — Azure Container Registry

1.1.4 Quiz

1.2 Images

Support de présentation : https://soleil-docker.jrobert-orleans.fr/02-Images-Dockerfiles (et en pdf : `02-Images-Dockerfiles-export`)

1.2.1 Gestion des images

Questions

Si ces questions n'ont pas été abordées, prenons un temps pour le discuter :

— Lancer un nouveau conteneur ubuntu après en avoir lancé un premier prend il de la place sur le disque ?
— Sur dockerhub certaines images sont « officielles », d'autres « vérifiées » et certaines, rien du tout. Comment le distingue-t-on et quels sont les risques à utiliser une image non vérifiée ?
— A quoi servent les tags des images ? Pour ou contre l'utilisation de « latest » (ça dépend ?) ?
— Qu'est-ce qu'un « registre » docker ?

Architecture Docker – Présentation

Les points qui vous ont été présentés :
- un exemple d'utilisation de docker pour un administrateur : architecture avec proxy oauth + conteneur derrière un virtualhost en quelques lignes.
- l'architecture docker : un démon avec une API http qui communique avec containerd qui lance des conteneurs via runc.

Image docker – Présentation

Les points qui vous ont été présentés :
- Une image = fichiers + meta-données. Standard « OCI » (open container initiative).
- Un conteneur = instanciation d'une image avec montage en lecture seule de l'image + une couche de persistance.
- Une image est constituée de couches.
- Dockerfile : fichier permettant de construire chaque couche d'une image.

Image docker - manipulations

Pour gérer les images la commande à utiliser sera `docker image` :
- Lister les images présentes sur la machine : `docker image ls` (avec la version raccourcie : `docker images`)
- Récupérer une image : `docker image pull` (avec la version raccourcie : `docker pull`)
- Supprimer une image : `docker image rm` (avec la version raccourcie : `docker rmi`)
- Obtenir des informations sur une image : `docker image history` et `docker image inspect` (avec les versions raccourcies : `docker history` et *docker inspect*)

Appliquez ces commandes pour :
- Déterminer le nombre d'images que vous avez sur votre machine.
- Déterminer la taille de l'image *alpine :latest*.
- Déterminer le nombre de couches de l'image *nginx :latest*.
- Supprimer les images que vous avez inutilement téléchargées.

Image docker - nettoyage

- Listez les images inutilisées : `docker image ls -f dangling=true`
- Nettoyez les images : `docker image prune`
- Vérifiez l'espace disque récupéré : `docker system df`

1.2.2 Création d'images

Avant de commencer

Dans la suite, nous allons utiliser le moteur de création d'images « buildkit », qui est plus intéressant (performances, fonctionnalités) que celui « par défaut » de docker.

Pour cela, vous devez exécuter :

```
docker buildx install
```

Note : avec les versions de docker récentes, il n'est plus nécessaire de faire `docker buildx install`.

Dockerfile

Créez un dossier « premier_dockerfile ».

Dans ce dossier écrivez un fichier *hello.sh* avec le contenu suivant :

```
#!/bin/sh
echo Bonjour tout le monde
```

Créez un second fichier, nommé *Dockerfile* avec le contenu suivant :

```
# syntax=docker/dockerfile:1
# L'instruction suivante indique qu'on souhaite se baser sur l'image ubuntu avec le␣
↪tag 24.04
FROM ubuntu:24.04
# L'instruction suivante permet de copier hello.sh dans le dossier /bin/ de l'image␣
↪en lui donnant le droit 755
COPY --chmod=755 hello.sh /bin/
```

Créez une image en exécutant :

```
docker build -t "ma_premiere_image:0.1" .
```

Remarquez que si vous lancez de nouveau cette commande, son exécution est bien plus rapide.

Lancez un conteneur à partir de cette image :

```
docker run -it ma_premiere_image:0.1 bash
```

Constatez que vous pouvez y lancer « hello.sh »

Layers

Exécutez la commande suivante :

```
docker image history ma_premiere_image:0.1
```

- Dans quel ordre les couches sont elles indiquées ?
- À quoi correspond la colonne SIZE ?
- Remarquez la colonne « CREATED ». Relancez le build et observez de nouveau cette colonne : les temps n'ont pas changé, le nouveau build n'a en fait rien fait d'autre que de constater qu'il n'avait rien à faire.
- Essayez de nouveau, mais cette fois ci en ajoutant l'option `--no-cache` à docker build. Constatez que cette fois les images ont bien été recréées.

Essais

Créez un fichier `README.txt` avec quelques lignes. Ajoutez une instruction *à la fin* de votre Dockerfile pour que ce fichier soit copié dans `/opt/`.

Lancez le build avec ce nouveau Dockerfile. Constatez avec `docker image history` que seul le cache a été utilisé pour les anciennes couches.

Ajoutez l'instruction suivante à la fin de votre Dockerfile :

```
RUN <<EOF
    apt update
    apt install --no-install-recommends -y vim-tiny
    rm -rf /var/lib/apt/lists/*
EOF
```

— Observez la taille des couches avec docker history.
— Constatez que si vous lancez à nouveau le build, docker utilise son cache et que tout est exécuté très rapidement.

Les instructions Dockerfile

Jusqu'ici nous avons utilisé trois instructions différentes : FROM, COPY et RUN :
— `FROM` permet de dire de quelle image on part. La bonne pratique voudra qu'on spécifie une version bien précise (ubuntu :22.04 plutot que ubuntu ou que ubuntu :latest)
— `COPY` permet de copier un fichier ou dossier dans le système de fichiers de l'image. L'option `--chmod` permet de préciser les droits que l'on donne aux fichiers copiés
— `RUN` permet d'exécuter une commande et si celle ci impacte les fichiers, les modifications feront partie de l'image.

Il y a en tout une quinzaine d'instructions, documentées ici : https://docs.docker.com/engine/reference/builder/ et avec une dizaine on est déjà très bien. Plus que quelques unes à connaître !

ENTRYPOINT et CMD

Dans la suite, nous nous assurerons systématiquement que notre conteneur contient les deux instructions suivantes :

```
ENTRYPOINT ["commande", "argument1", "argument2"]
CMD ["argument_supplémentaire_par_défaut1", "argument_supplémentaire_par_défaut2"]
```

Lorsqu'on lance un conteneur en faisant `docker run -it NOM_IMAGE`, le processus qui sera lancé à l'intérieur du conteneur sera : `commande argument1 argument2 argument_supplémentaire_par_défaut1 argument_supplémentaire_par_défaut2`.

Lorsqu'on lance un conteneur en faisant `docker run -it NOM_IMAGE argument_supp1 argumentsupp2`, le processus qui sera lancé à l'intérieur du conteneur sera : `commande argument1 argument2 argumentsupp1 argumentsupp2`.

Par exemple, supposons qu'on ait une image essai_entrypoint construite à partir du Dockerfile suivant :

```
# syntax=docker/dockerfile:1
FROM alpine:3.16
ENTRYPOINT ["/bin/echo", "Bonjour"]
CMD ["tout","le","monde!"]
```

- Sans essayer : quelle serait la sortie de la commande `docker run essai_entrypoint` ?
- Sans essayer : quelle serait la sortie de la commande `docker run essai_entrypoint Hello` ?
- Écrivez un dockerfile ayant pour image de base alpine :3.16, avec comme ENTRYPOINT `["/bin/ls"]` et comme CMD `["-lah"]`. Que cela fait-il ?
- Écrivez un dockerfile ayant pour image de base alpine :3.16, avec comme ENTRYPOINT `["/bin/sh","-c"]` et comme CMD `["sh"]`. Que cela fait-il ?

On peut ne pas spécifier ENTRYPOINT ou CMD, mais le fonctionnement n'est alors pas celui décrit ci-dessus. Dans la suite nous nous assurerons de systématiquement les renseigner.

principales instuctions pour Dockerfile

Commandes	Utilisation
`FROM`	IMAGE
`LABEL`	Ajouter des métadonnées
`RUN`	Lancer une commande
`ADD` [–chown=<user> :<group>] <src> <dest>	Ajout au conteneur
`COPY` [–chown=<user> :<group>] <url> <dest>	Ajout au conteneur et aussi `COPY --from=<image>` <src><dest>
`WORKDIR`	Répertoire de travail
`EXPOSE`	Port d'écoute
`ENTRYPOINT` / `CMD`	Commande à lancer au démarrage du container
`ENV`	Définition de variables d'environnemment
`USER`	Nom d'utilisateur à utiliser
`ARG`	Passage arguments ex : *docker build –build-arg machin=bidule*

Exemple plus complet

Ecrivons une image un peu plus complète pour finir avec une petite app nodejs dans un conteneur.

Créer un répertoire de base node-serv-fic.

- Fabriquer le fichier lireFic.js et le placer à la racine :

```
let http=require('http'),fs=require('fs');
http.createServer(function (req, res) {
// Ouvre et lit servHello.js
fs.readFile('hello.txt','utf8',function(err, data) {
    res.writeHead(200,{'Content-Type': 'text/plain'});
    if (err)
        res.write('Pb ouverture fichier\n');
    else// On renvoie le fichier au client
```

(suite sur la page suivante)

(suite de la page précédente)

```
        res.write(data);
    res.end();
    });
}).listen(8200, function() {console.log('Connecte au port 8200');});
console.log('Serveur tourne sur port 8200');
```

— On utilise un fichier package.json pour les dépendances :

```
{
    "name": "nodejs-simple",
    "version": "1.0.0",
    "description": "Envoi fichier par serveur node",
    "main": "lireFic.js",
    "keywords": [
        "node",
        "serveur",
        "fichier"
    ],
    "author": "Super Geek",
    "license": "ISC",
    "dependencies": {
        "http": "^0.0.1-security"
    }
}
```

— un fichier hello.txt pour publication (contenu à votre guise !)
— Enfin le Dockerfile :

```
FROM node:23.11-alpine
WORKDIR /app
COPY package.json .
RUN npm install
COPY lireFic.js .
COPY hello.txt .
EXPOSE 8200
ENTRYPOINT ["node", "lireFic.js"]
CMD []
```

On build et on lance :
— `docker build -t node-serv-fic .`
— `docker run -d --name node-serv -p 8000:8200 node-serv-fic`
— on visite http://localhost:8000

Exemple avec votre llm favori

En partant d'une page blanche, demandez à chatGPT ou autre de :
— Créer une application python qui sert le contenu d'un fichier sur le port 3004
— Créer un Dockerfile pour faire tourner cette application
— Donner une commande pour faire tourner cette application sur votre machine de façon à ce qu'elle soit accessible sur le port 8000

Avant de lancer, discutez et validez (si vous avez des doutes, ne lancez rien !). Discutez les éventuelles erreurs / approximations du llm.

1.2.3 Quiz

1.3 Communication avec l'extérieur

Support de présentation : https://soleil-docker.jrobert-orleans.fr/03-Communication_avec_le_reste_du_monde (et en pdf : `03-Communication_avec_le_reste_du_monde-export`)

1.3.1 Réseau

La théorie

Ce qui vous a été présenté :

Un container =
- une paire de « veth » : deux interfaces virtuelles, connectées entre elle (tout ce qui rentre dans l'une ressort dans l'autre et inversement)
- une des veth est dans le namespace réseau standard de l'hôte
- l'autre est dans un namespace isolé
- La veth du namespace standard est connectée au « bridge » docker0
- Le bridge docker0 fait du DNAT : il permet l'accès à l'extérieur

(Un bon article pour aller plus loin : https://iximiuz.com/en/posts/container-networking-is-simple/)

Observer le réseau des conteneurs

Regardez les interfaces réseau et le routage d'un conteneur :

```
docker run --rm -it alpine sh
ip link
ip route
```

De la même manière, regardez ce qu'il se passe en choisissant de ne pas faire de namespace séparé pour le conteneur :

```
docker run --network host --rm -it alpine sh
ip link
ip route
```

SNAT

A la création d'un conteneur, on crée une redirection de port en ajoutant l'option `-p PORTHOTE:PORTCONTENEUR` à `docker run`. Cela ouvre le port PORTHOTE sur votre machine et redirige tout ce qui y arrive sur le port PORTCONTENEUR du conteneur.

Par exemple, en partant de l'image nginx qui lance un serveur web qui écoute sur le port 80 du conteneur, nous allons pouvoir exposer le port 8000 de notre machine : `docker run -p 8000:80 nginx`.

C'est aussi simple que ça !

1.3.2 Variables d'environnement

Créez un fichier `affiche_variable_essai.sh` contenant le code suivant :

```
#!/bin/bash

echo Le contenu de la variable ESSAI :
echo $ESSAI
```

Ce script affiche la valeur de la variable d'environnement `ESSAI`. Pour le constater, il faut faire :

```
export ESSAI="Bonjour"
./affiche_variable_essai.sh
```

- Écrivez un dockerfile sur la base de alpine dont l'entrypoint est ce fichier afficher_essai.sh (pensez à donner les droits d'exécution à `affiche_variable_essai.sh`)
- Exécutez ce conteneur après avoir exporté la variable d'environnement ESSAI. Qu'en pensez vous ?
- Pour passer une valeur à l'environnement du conteneur, deux possibilités :
  - À la création de l'image via l'instruction `ENV`. Par exemple en ajoutant la ligne `ENV ESSAI=Bonjour` au Dockerfile.
  - À l'exécution du conteneur, via l'option -e de docker run : `docker run -e "ESSAI=Bonjour" -it --rm IMAGE`

Essayez !

1.3.3 Bind mount

Un bind mount est un montage d'un dossier de l'hôte dans un dossier du conteneur. A la création d'un conteneur, on crée un bind mount en ajoutant l'option `-v CHEMINHOTE:CHEMINCONTENEUR` à `docker run`.

Essayez en lançant une image alpine avec la commande sh et montant le répertoire courant dans le dossier /opt/ .

1.3.4 Volumes

Un volume est un peu comme un bind mount, à la différence que le volume est un espace de stockage géré par docker (par exemple en utilisant un répertoire quelque part sur le disque, ou encore en utilisant un stockage réseau en NFS, ...)

Pour créer un volume on fera : `docker volume create nom_volume` . Ensuite, au lancement d'un conteneur en ajoutant l'option `-v nom_volume:CHEMINCONTENEUR` à `docker run`.

Essayez : créez un volume "essai_volume" et lancez une image alpine avec la commande sh et montant le volume dans le dossier /opt/ . Créez des fichiers dans le répertoire opt, tuez le conteneur puis lancez en un nouveau. Constatez que les fichiers sont toujours là !

A l'aide de `docker volume inspect`, retrouvez l'endroit où sont stockés vos fichiers.

Note : une chose intéressante avec les volumes et qui diffère du Bind mount : si le volume est vide, les fichiers présents dans l'image sont recopiés dans le volume. C'est utile pour récupérer ce qui est par défaut dans l'image.

1.3.5 Applications

Serveur de fichiers statiques avec nginx

Utilisez l'image nginx pour servir un fichier index.html disant « bonjour ». La documentation de l'image nginx, présente sur dockerhub (https://hub.docker.com/_/nginx) indique que les fichiers *statics* sont à mettre dans le dossier `/usr/share/nginx/html`.

Serveur postgresql

L'image postgres lance un serveur postgres qui écoute sur le port 5432, stocke les données dans le dossier /var/lib/postgresql/data et est accessible avec l'utilisateur postgres et le mot de passe défini dans la variable d'environnement `POSTGRES_PASSWORD`.

Lancez un serveur postgresql et testez qu'il fonctionne :

```
psql
# lister les base de données
\l
# Creer la base exemple
CREATE DATABASE exemple
# se connecter sur la base exemple
\c exemple
# lister les tables
\dt
# Créer une table
CREATE TABLE machin (id INTEGER PRIMARY KEY, nom VARCHAR);
```

Si vous n'avez pas fait de bind-mount de dossier, lorsque vous quittez et supprimez votre conteneur vous perdrez vos données. Modifiez la création du conteneur pour que vos données ne soient jamais perdues.

Serveur ftp

Créez/trouvez une image permettant de servir des fichiers via ftp.

Springboot

Lors d'un développement web certains paramètres du code ne doivent pas être « en dur » dans le code et seront fournis uniquement au moment du déploiement. Par exemple un serveur web connecté à une base de données sera paramétré par l'adresse du serveur de base de données, le nom de l'utilisateur et son mot de passe.

Dans cet exercice, on va voir comment envoyer une valeur à springboot depuis notre fichier compose ou depuis la ligne de commande au moment de lancer un conteneur. On choisira ici d'appeler cette variable « nom du serveur de base de données » (qui contiendra par exemple « mon_postgres.exemple.com »), même si dans notre exemple minimal notre serveur n'aura pas de base de données.

L'objectif est de créer un serveur web qui affiche à la route « / » le nom du serveur de base de données.

Ecrivez un serveur springboot minimal en suivant les instructions suivantes en vous arrêtant à « Containerize it » : https ://spring.io/guides/gs/spring-boot-docker/ (https ://spring.io/guides/gs/spring-boot-docker/)

Ajoutez à votre classe Application un attribut « nom_base_de_donnees » récupéré depuis le fichier application.properties et modifiez la route « / » pour qu'on puisse voir son contenu :

```
package com.example.demo;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {

  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }


     @Value("${test.nom_base_de_donnees}")
     private String nom_base_de_donnees;

   @RequestMapping("/")
   public String home() {
       return "Serveur de base de données : " + nom_base_de_donnees;
     }


}
```

Modifiez le fichier application.properties pour que la propriété test.nom_base_de_donnees ait la valeur de la variable d'environnement DATABASE_SERVER_NAME :

```
test.nom_base_de_donnees=${DATABASE_SERVER_NAME:default_server_name_from_springboot.
↪com}
```

Lancez votre code java pour vérifier que jusqu'ici cela fait ce que vous voulez :

```
DATABASE_SERVER_NAME=mon_postgres.exemple.com ./mvnw spring-boot:run
```

Vous pouvez aussi le packager :

```
DATABASE_SERVER_NAME='pour_les_tests' ./mvnw package
```

Créez un fichier Dockerfile pour containeriser votre application :

```
FROM openjdk:11-jdk
ENV DATABASE_SERVER_NAME default_from_docker.com
COPY target/*.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Lancez votre conteneur et observez ce que le serveur affiche à la route / .

Relancez votre conteneur en spécifiant une autre valeur pour la variable DATABASE_SERVER_NAME .

Autres technologies

Si vous utilisez d'autres technologies / frameworks, trouvez la manière d'accéder aux variables d'environnement depuis votre code et véri fiez le en construisant un exemple minimal et un dockerfile correspondant.

1.4 Application multi-conteneur - Docker-compose

Support de présentation : https://soleil-docker.jrobert-orleans.fr/04-docker_compose (et en pdf : `04-docker_compose-export`)

1.4.1 Applications multi-containers

Besoin

Pour concevoir et déployer des applications fondées sur plusieurs micro-services :
- BD
- NoSQL (Mongo etc.)
- Applications
- APIs

de nouveaux besoins apparaissent :
- nécessité de communiquer entre containers
- possibilité de créer des réseaux ad-hoc mais pas très facile à manipuler
- besoin de pouvoir décrire dans une syntaxe simplifiée un système de containers - avec des images prédéfinies - ou spécifiées dans de Dockerfiles - communicant naturellement entre elles

Solution : docker-compose

La commande docker-compose repose sur un fichier *docker-compose.yml*, écrit au format YAML.

Format YAML

- json en moins verbeux
- plus lisible, avec indentation (2 caractères décalage suffisent)
- bcp de fichiers de conf utilisent ce format
- cf exemples

docker ou docker-compose ?

Créons un conteneur nginx avec la commande Docker :

```
docker run --name web -d -p 8000:80 nginx:alpine
```

Puis allez visiter http://localhost:8000 ...

On peut vouloir monter un volume dans notre container pour publier une page de notre cru via nginx :

— créer un dossier de base nommé compose-nginx
— créer dedans dossier app contenant un fichier index.html :

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Nginx Docker</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bulma@0.9.4/css/bulma.
↪min.css">
</head>
<body>
<section class="section">
    <div class="container">
    <h1 class="Nginx via Docker">
        Hello World
    </h1>
    <p class="subtitle">
       Nginx  à l'intérieur d'un container <strong>Docker</strong>!
    </p>
    </div>
</section>
</body>
</html>
```

— Puis relançons la commande en montant app au bon endroit :

```
docker run --name web -itd -p 8000:80 -v $(pwd)/app:/usr/share/nginx/html nginx:alpine
```

Nginx avec docker-compose

Faisons plus simple avec une description en yaml :

```
services:
    web:
        image: nginx:alpine
        ports:
            - "8000:80"
        volumes:
            - ./app:/usr/share/nginx/html
```

Explications

— top level : services
— ici un seul service : web, assuré par nginx
— le volume local app sera visible dans le container, à l'emplacement /usr/share/nginx/html
— le port 8000 de l'hôte sera redirigé vers le port 80 du container
— Lancement : *docker compose up -d* en mode *detached*

Questions

— Peut-on changer en direct le contenu du fichier index.html du dossier app pendant que le conteneur tourne ?
— comment lister ce qui tourne ?
— Quels volumes sont montés ?
— Comment tout arrêter ?

Services

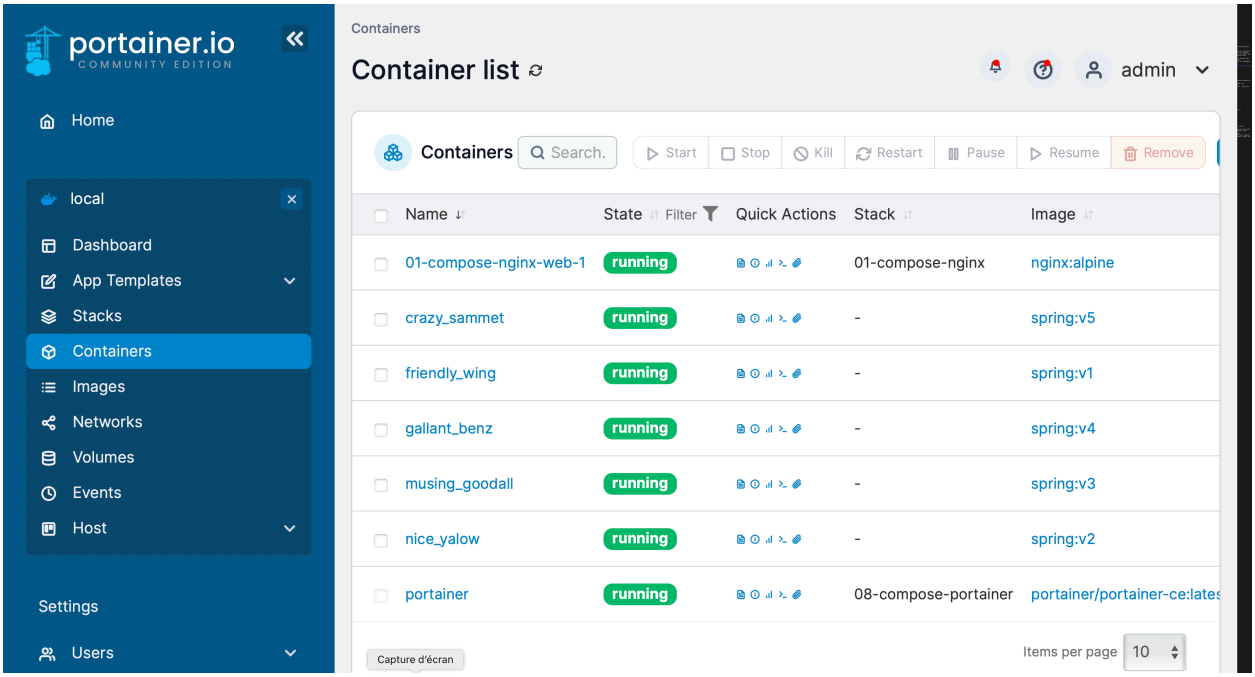
On peut évidemment placer de multiples services dans un docker-compose.
— les différents services sont décrits avec le même niveau d'indentation.
— les services peuvent se décrire avec une image préfabriquée (Dockerhub) * avec d'éventuels fichiers de configuration * avec d'éventuels paramètres
— les services peuvent également faire référence à des images fabriquées avec des Dockerfile spécifiques

principales commandes docker-compose

Commandes	Utilisation
`docker-compose build`	build
`docker-compose up`	lancer l'app
`docker-compose up -d`	lancer en arrière plan
`docker-compose ps`	lister les containers de l'app
`docker-compose logs nginx`	visualiser les logs du container nginx
`docker-compose pause`	faire une pause en gardant les containers en l'état
`docker-compose unpause`	arrêter la pause
`docker-compose stop`	arrêter l'application en gardant les données associées
`docker-compose down`	arrêter l'application en enlevant les containers, réseaux et volumes associés
`docker-compose down --rmi all`	grand ménage ! (Attention données, etc.)

Testez-les !

Portainer



Portainer est un outil web qui permet de gérer les conteneurs Docker et Docker Compose. Il permet de gérer les conteneurs, les images, les volumes, les réseaux, les stacks etc. Il est disponible sous forme d'image Docker et peut donc s'installer via Docker ! On peut par exemple le configurer et le lancer à partir du docker-compose suivant (linux ou mac) :

```
services:
portainer:
    image: portainer/portainer-ce:latest
    container_name: portainer
    restart: unless-stopped
    volumes:
        - /etc/localtime:/etc/localtime:ro
        - /var/run/docker.sock:/var/run/docker.sock:ro
        - ./portainer-data:/data
    ports:
        - 9000:9000
```

On lance la commande suivante pour lancer le service portainer :

```
$ docker compose up -d
```

L'interface web est accessible via l'adresse suivante : http://localhost:9000. Lors du premier lancement, un assistant de configuration est lancé pour créer un compte administrateur, puis choisir l'option « local » pour la connexion à Docker. Il faut nommer le container et puis cliquer sur « connect ». Vous pouvez ensuite visualiser les conteneurs Docker et Docker Compose lancés sur la machine ainsi que leurs environments, volumes, ports, etc. La documentation est disponible ici : https://docs.portainer.io

Essayez le :

— Visualisez les logs de conteneurs
— Créez une application à l'aide d'un docker-compose que vous déposerez dans la section « stacks ».

— Définissez des variables d'environnement pour votre stack.

Postgresql avec adminer

Une première application muti-container où l'on se contente d'images prédéfinies.

Le docker-compose

```
# Use postgres/example user/password credentials
services:
    db:
        image: postgres
        restart: always
        environment:
          POSTGRES_PASSWORD: example

    adminer:
        image: adminer
        restart: always
        ports:
          - 8080:8080
```

— Faites un schéma représentant ce docker compose (un simple rectangle pour chaque service, le tout dans un rectangle représentant l'hôte ; représentez aussi les ports, etc.)
— Placez le dans un répertoire séparé et testez ! (L'utilisateur dans l'interface adminer sera "postgres" et le mot de passe "example")
— Quels sont les services ?
— A quoi correspondent les directives *restart : always* et *environment* ?

Exemple multi-containers avec Dockerfile : Flask et Redis

— Exemple classique
— Redis est un système clef/valeur efficace dans le Cloud
— Flask est un micro-framework Python pour développer simplement des app Web
— Tout le code de Flask est dans un seul fichier run.py
— Créer un répertoire compose-flask-redis contenant le fichier suivant :

Le fichier run.py

```
from flask import Flask
from redis import Redis

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

@app.route('/')
def hello():
    count = redis.incr('hits')
    return 'Hello for the {} time !\n'.format(count)
```

(suite sur la page suivante)

(suite de la page précédente)

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", debug = True)
```

Conteneuriser ce service

- installer les dépendances
- lancer l'application automatiquement

Les dépendances sont gérées en Python à l'aide d'un fichier *requirements.txt* qui peut être utilisé pour créer un virtualenv ou un conteneur.

A minima :

```
flask
redis
```

(on peut préciser des versions aussi !)

On installe les requirements avec la commande : *pip install -r requirements.txt*

- Écrire le Dockerfile pour le service Flask en partant de l'image python :3.11-slim
- Ajouter le contenu du dossier courant au dossier /app du conteneur : `ADD . /app`
- puis choisir /app comme répertoire de travail
- installer les dépendances
- exposer le port 5000
- lancer run.py
- testez !

le docker compose avec le service Redis

Ajouter à présent un docker-compose.yml dans votre dossier dont voici les éléments :

- Redis étant une image standard on peut directement l'invoquer dans le docker-compose contenant deux services :

```
redis:
  image: "redis"
```

- le service Flask étant décrit par

```
web:
    build: .
    ports:
    - "4000:5000"
```

- Ecrivez le docker-compose correspondant
- puis build et lancement
- `docker compose build`
- `docker compose up -d`

Visitez : http://127.0.0.1:4000

Observation de l'app avec les outils Docker

- `docker compose ps`
- `docker logs ...`
- Quelles sont les tailles des images utilisées ?
- Lister les réseaux ? Les volumes ?
- Comment lancer un terminal interactif sur le container Flask ?
- Comment arrêter tout ?
- Faire le grand ménage ?

Améliorations du Dockerfile

- On peut définir des variables d'environnement dans FLASK : *FLASK_DEBUG* à la valeur *True* si on souhaite être en mode DEBUG et *FLASK_APP* avec le nom de l'app à lancer.
- puis lancer l'app avec la directive : *flask run*

Effectuez les petites modifs correspondantes. Tester !

Réduction des images

- essayez les images python :3.11-alpine et python :3.11
- comparez les tailles et les temps de build et de lancement

Fichier .env

De manière à ne pas versionner les données sensibles, on peut créer un fichier .env dans le dossier du docker-compose. Ce fichier a la forme suivante :

```
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
...
```

On accède dans le docker-compose.yml avec la syntaxe : `${POSTGRES_USER}` . Par exemple :

```
services:
    db:
        image: postgres
        environment:
            POSTGRES_USER: ${POSTGRES_USER}
            POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
```

Portainer

Déployez le docker compose précédent en passant par portainer.

1.4.2 Outils complémentaires pour Docker

yamllint

Vérifiez la syntaxe de votre fichier YAML avec la commande suivante :

```
$ yamllint docker-compose.yml
```

La doc est ici : https://yamllint.readthedocs.io/en/stable/

Hadolint

Hadolint vérifie la validité de votre Dockerfile et vous donne des indications pour améliorer la qualité.

```
docker run --rm -i hadolint/hadolint < Dockerfile
```

Essayez le sur les dockerfiles que vous avez. Discutez les recommendations fournies.

1.5 Mise en pratique - exemple de A à Z

Support de présentation : https://soleil-docker.jrobert-orleans.fr/06-Ecosysteme_et_Orchestration (et en pdf : `06-Ecosysteme_et_Orchestration-export`)

1.5.1 Démo complète

Nous allons créer une application avec :

- Un serveur backend à partir du code : https://gitlab.com/formation-docker_jr/backend_springboot.git
- Un serveur frontend à partir du code : https://gitlab.com/formation-docker_jr/frontend.git
- Un serveur de base de données en postgres.
- Un reverse proxy qui transferera les requêtes vers /api/ vers le backend et toutes les autres vers le frontend.

Commençons par un dessin de l'architecture visée.

Ensuite, nous allons créer des images pour chaque service. Nous allons tester ces images indépendamment du reste et noter ce qu'on a fait au format docker compose pour pouvoir ensuite le reproduire.

Je propose cet ordre (qui me semble aller du plus simple au moins simple) qui sera détaillé dans la suite :

1. Faire tourner le frontend seul en exposant sur le port 8080
2. Faire tourner une base de données seule, noter son ip,

3. Faire tourner le backend seul en exposant le port 8081 puis le brancher à la BD (en utilisant son ip)
4. Ajouter des données à la base de données

Vérifier que tout cela continue de fonctionner avec un docker compose.

On verra ensuite comment ajouter le reverse proxy pour que tout passe par un port unique puis comment sécuriser davantage les réseaux.

Frontend

Cloner le dépôt : `git clone https://gitlab.com/formation-docker_jr/frontend.git`

Créer une image (`docker build -t frontend_exemple`) et la lancer.

On peut noter tout ça :

```
services:
    frontend:
        image: fontend_exemple
        ports:
            - "8080:80"
```

Base de données

```
docker run --rm -d -e POSTGRES_PASSWORD=toto --name pgtest postgres
```

Noter l'ip du conteneur :

```
docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' pgtest
```

(ici on est un peu embêtés car il n'y a pas comme dans le docker compose la résolution de l'IP par le nom du conteneur)

Backend

D'abord sans docker :

```
export POSTGRES_SERVER=`docker inspect -f '{{range.NetworkSettings.Networks}}{{.
↪IPAddress}}{{end}}' pgtest`
export POSTGRES_DB="postgres"
export POSTGRES_USER="postgres"
export POSTGRES_PASSWORD="toto"
java -jar target/spring-boot-docker-0.0.1-SNAPSHOT.jar
```

Puis créer un Dockerfile en utilisant l'image openjdk :11-jdk :

```
FROM openjdk:11-jdk
COPY target/*.jar /opt/app.jar
ENTRYPOINT ["java","-jar","/opt/app.jar"]
```

Créez l'image en l'appelant backend_exemple

Ajout de données en BD

Il faudrait exécuter le code sql suivant dans la base de données :

```
CREATE TABLE utilisateur( nom TEXT );
INSERT INTO utilisateur VALUES ('bonjour');
INSERT INTO utilisateur VALUES ('tout');
INSERT INTO utilisateur VALUES ('le');
INSERT INTO utilisateur VALUES ('monde');
```

Pour cela, on peut par exemple exécuter postgres directement dans le conteneur pgtest :

```
docker exec -it pgtest psql -U postgres
```

Notez qu'on aurait aussi pu lancer un conteneur rien que pour cette opération. Un conteneur qui aurait psql ...

Résumé

```
 version: '3.9'
services:
     frontend:
         image: frontend_exemple
         ports:
             - "8080:80"
     backend:
         image: backend_exemple
         ports:
             - "8081:8080"
         environment: &env_db
           - POSTGRES_SERVER=database
           - POSTGRES_DB=postgres
           - POSTGRES_USER=postgres
           - POSTGRES_PASSWORD=toto
           - PGPASSWORD=toto
     database:
         image: postgres
         environment:
            -  POSTGRES_PASSWORD=toto
     init_db:
         image: postgres
         environment:
           - POSTGRES_SERVER=database
           - POSTGRES_DB=postgres
           - POSTGRES_USER=postgres
           - POSTGRES_PASSWORD=toto
           - PGPASSWORD=toto
           - |
             INIT_DB=
             CREATE TABLE utilisateur( nom TEXT );
             INSERT INTO utilisateur VALUES ('bonjour');
             INSERT INTO utilisateur VALUES ('tout');
             INSERT INTO utilisateur VALUES ('le');
             INSERT INTO utilisateur VALUES ('monde');
         command: sh -c "echo $$INIT_DB | psql -U $$POSTGRES_USER -h $$POSTGRES_
↪SERVER $$POSTGRES_DB"
```

Améliorations

A partir de là, on peut :
— segmenter mieux nos réseau
— indiquer les dépendances entre conteneurs
— documenter les ports ouverts
— choisir des versions en dur pour nos conteneurs / paquets
— relire nos dockerfile pour voir si on applique les bonnes pratiques
— ajouter un volume pour que la base de données soit persistante

Initialisation base de données

Pour automatiser la création de la base de données plusieurs options sont possibles :

1. Créer un script coté springboot (cf *https ://www.baeldung.com/spring-boot-data-sql-and-schema-sql <https ://www.baeldung.com/spring-boot-data-sql-and-schema-sql>_*)
2. Créer un script coté base de données (l'image postgres permet de lancer quelque chose au premier lancement)
3. Utiliser un outil de gestion de migration de base de données (flyway par exemple pour springboot)
4. Définir un containeur supplémentaire dans le docker compose qui sera lancé sur demande

Pour aller plus loin

Dans la suite, nous pouvons approfondir :
— l'utilisation de traefik comme reverse proxy
— l'utilisation de gitlab pour déposer nos images.
— l'automatisation de la compilation du code (ici le code du serveur devait être compilé à la main avant de pouvoir créer l'image)
— l'utilisation de gitlab pour la compilation (et le déploiement) automatique des images

1.5.2 Traefik

Avec docker compose, vous avez vu comment déclarer des services, les « scaler », .. Pour pouvoir déployer une application, vous avez croisé haproxy pour :
— l'équilibrage de charge,
— un routage qui dépend de la requête http (le conteneur questionné dépend de la route demandée),
— la gestion des certificats TLS (souvent un peu compliqué) ..

Traefik répond à *tous* ces besoins, en un peu plus simple. Vous allez avoir ici un petit aperçu de ce qui est possible et de son fonctionnement.

Ecrivez le docker-compose suivant :

```
services:
  reverse-proxy:
    # The official v2 Traefik docker image
    image: traefik:v2.7
```

(suite sur la page suivante)

(suite de la page précédente)

```
    # Enables the web UI and tells Traefik to listen to docker
    command: --api.insecure=true --providers.docker
    ports:
        # The HTTP port
        - "80:80"
        # The Web UI (enabled by --api.insecure=true)
        - "8080:8080"
    volumes:
        # So that Traefik can listen to the Docker events
        - /var/run/docker.sock:/var/run/docker.sock
```

Lancez le service reverse-proxy : *docker compose up -d reverse-proxy*

Ce faisant, vous avez lancé le conteneur traefik qui :
- écoute sur le port 80 de votre machine et sur le port 8080.
- a un accès à l'API docker de votre machine (via le montage de /var/run/docker.sock)

Pour le moment, rien d'intéressant sur le port 80. Et sur le port 8080, vous avez une interface de gestion de traefik (rien d'intéressant non plus).

Traefik écoute tout ce qui se passe sur l'API docker et détecte lorsque vous lancez de nouveaux conteneurs. Il regarde alors les labels associés, et se configure en conséquence.

Par exemple, ajoutez les lignes suivantes à votre docker-compose.yml :

```
whoami:
  # A container that exposes an API to show its IP address
  image: traefik/whoami
  labels:
    - "traefik.http.routers.whoami.rule=Host(`whoami.docker.localhost`)"
```

Remarquez la partie « labels », elle permettra à traefik de se configurer. Lancez le service : *docker compose up -d whoami*

Rendez-vous à l'adresse whoami.docker.localhost : le proxy traefik transmet les requêtes qui lui sont faites vers whoami. Pas mal déjà !

Modifiez le service whoami pour qu'il y ait plusieurs répliques :

```
whoami:
  # A container that exposes an API to show its IP address
  image: traefik/whoami
  labels:
    - "traefik.http.routers.whoami.rule=Host(`whoami.docker.localhost`)"
  deploy:
    replicas: 2
```

Lancez à nouveau les services. Constatez le fonctionnement « round robin » de l'équilibrage de charge !

Et ce n'est qu'une toute petite partie de ce que permet Traefik..

Pour essayer, mettez en place une application web dont le contenu dynamique est servi par un conteneur et le contenu static est servi par un autre conteneur. Pour cela, vous utiliserez les règles traefik de la forme

```
"(Host(`example.org`) && Path(`/machin`))"
```

Voir également :
- https://doc.traefik.io/traefik/getting-started/quick-start/ pour la doc.

— https://www.digitalocean.com/community/tutorials/how-to-use-traefik-v2-as-a-reverse-proxy-for-docl pour un exemple d'usage en reverse-proxy avec un Wordpress.

Traefik dans un compose séparé

Dans le cas où vous avez plusieurs applications dockerisées, vous souhaiterez avoir un reverse proxy commun à toutes les applications.

Pour cela, vous aurez un docker compose pour traefik, puis un docker compose par « application »

```
services:

  traefik:
    image: "traefik:v2.9"
    container_name: "traefik"
    command:
      #- "--log.level=DEBUG"
      - "--api.insecure=true"
      - "--providers.docker=true"
      - "--providers.docker.exposedbydefault=false"
      - "--entrypoints.web.address=:80"
    ports:
      - "8001:80"
      - "8087:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock:ro"
    networks:
      - frontends

networks:
  frontends:
    name: public
```

Puis par exemple pour une application whoami :

```
services:
  whoami:
    image: "traefik/whoami"
    container_name: "simple-service"
    networks:
      - public
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.whoami.rule=Host(`whoami.localhost`)"
      - "traefik.http.routers.whoami.entrypoints=web"

networks:
  public:
    external: true
```

Et pour une deuxième application :

```
services:
  whoami:
    image: "traefik/whoami"
```

(suite sur la page suivante)

(suite de la page précédente)

```
    container_name: "simple-service"
    networks:
      - public
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.whoami.rule=Host(`deuxieme.localhost`)"
      - "traefik.http.routers.whoami.entrypoints=web"

networks:
  public:
    external: true
```

Traefik pour le code de la démo précédente

```
services:
  reverse-proxy:
    # The official v2 Traefik docker image
    image: traefik:v2.7
    # Enables the web UI and tells Traefik to listen to docker
    command: --api.insecure=true --providers.docker
    ports:
        - "80:80"
        - "8080:8080"
    volumes:
        # So that Traefik can listen to the Docker events
        - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
        - public

  frontend: # <- le nom du conteneur
      image: image_frontend_demo
      ports:
          - "80"
      labels:
        - "traefik.http.routers.frontend.rule=Host(`demo.localhost`) && PathPrefix(`/
↪`)"
      networks:
         - public
  database:
      image: postgres:15
      ports:
          - "5432"
      environment:
          - POSTGRES_PASSWORD=toto
      networks:
         - db_network
  backend:
      image: image_backend_demo
      ports:
          - "8080"
      environment:
        - POSTGRES_SERVER=database
        - POSTGRES_DB=postgres
        - POSTGRES_USER=postgres
```

(suite sur la page suivante)

(suite de la page précédente)

```
        - POSTGRES_PASSWORD=${MACHIN}
      labels:
        - "traefik.http.routers.backend.rule=Host(`demo.localhost`) && PathPrefix(`/
↪api`)"
      networks:
         - public
         - db_network


  init_db:
        image: postgres
        environment:
          - POSTGRES_SERVER=database
          - POSTGRES_DB=postgres
          - POSTGRES_USER=postgres
          - POSTGRES_PASSWORD=toto
          - PGPASSWORD=toto
          - |
            INIT_DB=
            CREATE TABLE utilisateur( nom TEXT );
            INSERT INTO utilisateur VALUES ('bonjour');
            INSERT INTO utilisateur VALUES ('tout');
            INSERT INTO utilisateur VALUES ('le');
            INSERT INTO utilisateur VALUES ('monde');
        command: sh -c "echo $$INIT_DB | psql -U $$POSTGRES_USER -h $$POSTGRES_SERVER
↪$$POSTGRES_DB"
        profiles:
           - tache_administration
        networks:
           - public

networks:
  db_network:
  public:
```

1.5.3 Bonnes pratiques & Securité

Sécurité

Docker propose un outil de détection de vulnérabilité (Common Vulnerabilities and Exposures -CVEs) des images. C'est aussi simple que de lancer :

```
docker scout cves <nom_de_l_image>
```

Essayez par exemple en créant une image basée sur nginx :1.18.0 :

```
FROM nginx:1.18.0

RUN apt update
RUN apt install -y vim
```

puis en faisant un build :

```
docker build -t essai_scan .
```

puis le scan :

```
docker scout cves essai_scan
```

Suivez les recommendations pour améliorer la sécurité de votre image.

.dockerignore

Ca ne vous a peut être pas échappé, au moment de lancer un build docker affiche « Uploading build context » . Le client docker à ce moment envoie l'intégralité du répertoire courant au serveur docker. Si ce répertoire est volumineux, cela peut prendre du temps.

Une bonne pratique consiste à écrire un fichier .dockerignore contenant des expressions régulières de fichiers à exclure du contexte de build.

Essayez !

Bonnes pratiques diverses

Ces bonnes pratiques sont tirées de la documentation : https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

- Each container should have only one responsibility.
- Containers should be immutable, lightweight, and fast.
- Don't store data in your container. Use a shared data store instead.
- Containers should be easy to destroy and rebuild.
- Use a small base image (such as Linux Alpine). Smaller images are easier to distribute.
- Avoid installing unnecessary packages. This keeps the image clean and safe.
- Avoid cache hits when building.
- Auto-scan your image before deploying to avoid pushing vulnerable containers to production.
- Scan your images daily both during development and production for vulnerabilities Based on that, automate the rebuild of images if necessary.
- apt-get : privilégiez l'installation de paquets sous la forme suivante :

```
RUN apt-get update && apt-get install -y \
package-bar \
package-baz \
package-foo  \
&& rm -rf /var/lib/apt/lists/*
```

Isolation des réseaux

Dans le docker compose suivant, on crée 3 services et 2 réseaux. Lancez ce docker compose et vérifiez la communication depuis chaque conteneur : * service1 peut-il communiquer (ping) avec service2 ? avec service3 ? avec yahoo.fr ? * Idem avec service2 et service3

```
services:
service1:
    image: alpine
```

(suite sur la page suivante)

(suite de la page précédente)

```
    command: tail -f /dev/null
    networks:
    - internal_net
    - isolated_net

service2:
    image: alpine
    command: tail -f /dev/null
    networks:
    - internal_net

service3:
    image: alpine
    command: tail -f /dev/null
    networks:
    - isolated_net

networks:
internal_net:
    driver: bridge
isolated_net:
    driver: bridge
    internal: true
```

Exercice

Dans l'exemple suivant la personne qui fournit le Dockerfile a pour intention de créer une application qui qui exécute un serveur web Nginx et un serveur SSH pour qu'un utilisateur "user" puisse se connecter en SSH et modifier le contenu du serveur web. Cependant, ce qu'il propose ne respecte pas les bonnes pratiques énoncées ci-dessus.

Essayez de proposer des améliorations.

```
.
├── Dockerfile
└── index.html
```

```
FROM ubuntu:latest
MAINTAINER John Doe <ohn.doe@example.com>
RUN apt-get update
RUN apt-get install -y nginx vim git openssh-server
COPY index.html /var/www/html/index.html
RUN mkdir /run/sshd
RUN useradd -m -s /bin/bash user
RUN echo 'user:password' | chpasswd
EXPOSE 80
EXPOSE 22
ENTRYPOINT ["bash", "-c"]
CMD ["/sbin/sshd && nginx && tail -f /var/log/nginx/access.log"]
```

```
version: '3'
services:
  web:
    build: .
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ports:
      - "80:80"
      - "22:22"
    volumes:
      - ./index.html:/var/www/html/index.html
```

1.5.4 Utilisation du registry de gitlab

Nous allons ici voir comment utiliser gitlab pour partager vos images docker.

Prérequis

1. Créez un acces token pour gitlab
   — Se logguer sur gitlab
   — Choisir « edit profile » en haut à gauche
   — Choisir le menu « Access Token »
   — Donner un nom à votre token (par exemple formation_docker)
   — Choisir « read_repository » et « write_repository »
   — Générer le token et noter le code dans « new personnal access token »
2. Enregistrer ce token dans docker : docker login
3. Créez un projet « public » sur gitlab pour les besoins de cette expérimentation

Push/pull image

1. Créez un fichier Dockerfile contenant :

```
FROM hello-world
```

Ensuite :

```
docker build . -t registry.XXXX.fr/VOTRENOM/NOM_PROJET/image_hello:latest
```

Puis :

```
docker push registry.XXXX.fr/VOTRENOM/NOM_PROJET/image_hello:latest
```

On peut ensuite récupérer l'image sur un autre poste avec :

```
docker pull registry.XXXX.fr/VOTRENOM/NOM_PROJET/image_hello:latest
```

et la lancer avec :

```
docker run  registry.XXXX.fr/VOTRENOM/NOM_PROJET/image_hello:latest
```

2. (Bonus) Utilisation du CI/CD : rendez vous sur gitlab puis dans le projet et dans CI/CD -> Pipelines puis choisissez Docker puis validez.

1.5.5 Build multistage

```
FROM alpine as base
RUN echo mot_de_pass_secret > /opt/mot_de_passe_secret
RUN echo toto > /opt/code_compile
CMD ["sh", "-c", "echo Bonjour depuis base && ls /opt"]

FROM alpine as dev
COPY --from=base /opt/code_compile /opt/
CMD ["sh", "-c", "echo Bonjour depuis dev && ls /opt"]

FROM alpine as prod
COPY --from=base /opt/code_compile /opt/
CMD ["sh", "-c", "echo Bonjour depuis prod && ls /opt"]

FROM alpine as rien
CMD ["sh", "-c", "echo Bonjour depuis rien && ls /opt"]
```

```
# syntax=docker/dockerfile:1
FROM node:14-alpine AS builder
WORKDIR /opt/
COPY . .
RUN npm install
RUN npm run build -- --mode production

FROM nginx:1.23
WORKDIR /opt/
COPY --from=builder /opt/dist/ /usr/share/nginx/html/
```

Pour springboot

```
# syntax=docker/dockerfile:1
FROM openjdk:11-jdk AS builder
WORKDIR /opt/
COPY . .
RUN ./mvnw package -Dmaven.test.skip=true

FROM openjdk:11-jdk
COPY --from=builder /opt/target/*.jar /opt/app.jar
ENTRYPOINT ["java","-jar","/opt/app.jar"]
```

Intérêts

1. Images allégées, permet de ne pas stocker de données sensibles dans l'image, ..
2. Process de compilation automatisé
3. Possibilité de viser un « stage » et ainsi avec un seul Dockerfile pouvoir créer des images différentes (par exemple prod / dev).
   — Avec docker build : *docker build –target builder*
   — Avec docker compose en ajoutant une option "target" à la partie build.

2

Références

- Référence Docker (https ://docs.docker.com/reference/)
- Docker interactive Tutorial (https ://github.com/docker/getting-started/)
- Dockerfiles (https ://docs.docker.com/engine/reference/builder/)
- Images Docker (https ://docs.docker.com/engine/reference/commandline/images/)
- Applications multi-containers (https ://docs.docker.com/get-started/07_multi_container/)
- Applications multi-containers avec Java (https ://github.com/docker/labs/tree/master/developer-tools/java/)

3

Index et recherche

— genindex
— search

Index