Formation Docker

Version 2025 - Soleil découverte

Julien Robert

juin 19, 2025

Table des matières

1 Tables des matières

1.1 Introduction

Support de présentation : https://soleil-docker.jrobert-orleans.fr/01-Intro_docker

Lien vers le QCM de début et de fin de formation : https://forms.cloud.microsoft/e/08i0cHmL0i

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`.

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`).
- constatez que vous ne pouvez pas rebooter la machine, ni tuer de processus qui ne sont pas dans le conteneur, etc.

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`.
- 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`

Quelques autres petites options de docker run

Essayez les options suivantes :
- `--restart` : redémarre le conteneur si il se termine
- `--rm` : supprime le conteneur quand il se termine
- `--name` : nomme le conteneur

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 contenant un environnement pour votre langage de programmation préféré ?

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

1.2.1 Gestion des images

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 buid. 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 plutôt 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` ?
- Ecrivez un dockerfile ayant pour image de base alpine :3.16, avec comme ENTRYPOINT `["/bin/ls"]` et comme CMD `["-lah"]`. Que cela fait-il ?
- Ecrivez 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

1.3.1 Réseau

SNAT

Lancez un conteneur alpine et constatez que vous pouvez communique avec l'extérieur.

```
docker run -it --rm alpine sh
ping google.com
```

Par défaut vos conteneurs ont accès à l'extérieur (SNAT).

Expériementation du DNAT

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`.

Essayez ! Essayez avec d'autres ports pour vous assurer que les choses sont claires.

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. Ca ne donne rien et c'est normal, ce n'est pas de cette façon que l'on passe des variables d'environnement au conteneur.

— Pour passer une valeur à l'environnement à l'exécution du conteneur on fera : `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`.

1.4 Application multi-conteneur - Docker-compose

Support de présentation : https://soleil-docker.jrobert-orleans.fr/04-docker_compose_et_bp_decouverte

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.

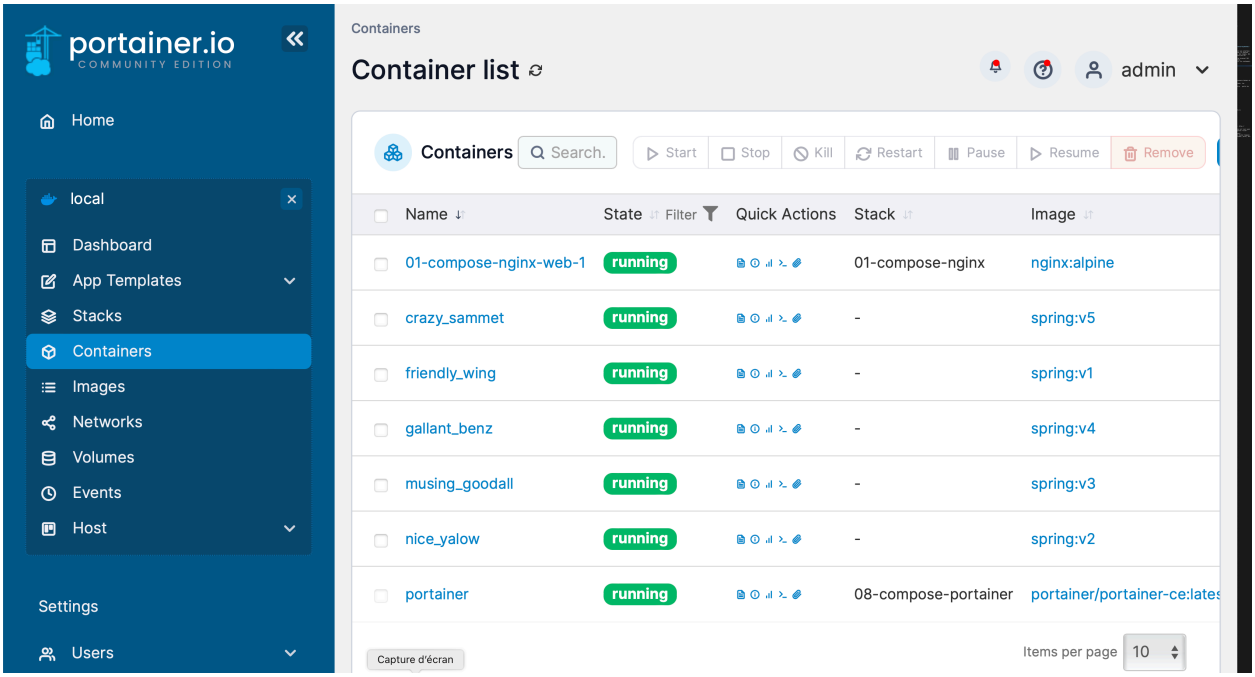
- les différents services sont décrits avec le même niveau d'indentation.
- les services peuvent se décrire avec une image préfabriquée (Dockerhub) * avec d'éventuels fichiers de configuration * avec d'éventuels paramètres
- les services peuvent également faire référence à des images fabriquées avec 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:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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)

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 compose 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 :

```
# Si yamllint est installé sur votre système:
# yamllint docker-compose.yml
# Sinon :
docker run --rm -v $(pwd):/data cytopia/yamllint /data/docker-compose.yml
```

La documentation 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.

1.4.3 Bonnes pratiques

Dans cette section, nous allons aborder quelques bonnes pratiques à suivre lors de la création de conteneurs Docker.

.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.

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/*
```

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

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