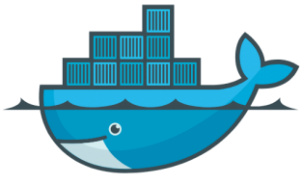
Docker

La conteneurisation de nos applications et comment s'en servir



I. Qu'est-ce que Docker ?

Docker est un outil qui permet de créer, déployer et exécuter des applications dans des environnements isolés appelés conteneurs. Il fonctionne à la fois comme un outil de déploiement, mais est aussi un atout formidable pour construire facilement des environnements de développement. Il permet effectivement de répondre à tous ces besoins :

- Éviter les problèmes de dépendances
- Faciliter le travail en équipe en uniformisant les environnements de développement
- Déployer facilement une application
- Garantir le même environnement partout

II. Un peu de jargon

1. Qu'est-ce qu'une image ?

Une image Docker est un modèle immuable qui contient tout ce dont une application a besoin :
- Le code
- Les dépendances
- La configuration

On la génère avec un fichier que l'on appelle le **Dockerfile**. C'est lui qui va contenir une **liste d'instructions** pour fabriquer un environnement de développement précis, comme par exemple démarrer un serveur Nginx Alpine puis exposer son port 80 et y importer une configuration spécifique.

☞ Comme une recette de cuisine.

2. Qu'est-ce qu'un conteneur ?

Un conteneur est tout simplement une **instance** d'une image en cours d'exécution. Docker va être capable, en lisant une **image** de générer un environnement de développement **conteneurisé**. Docker peut tout à fait générer plusieurs conteneurs à partir de la même image.

☞ Le plat préparé à partir de la recette.

L'image est un modèle d'environnement mais n'est pas exécutée, alors que le conteneur est une image en cours d'exécution.

III. Le Dockerfile

Un Dockerfile est un fichier texte contenant les instructions permettant de construire une image Docker.

Exemples d'instructions :
- **FROM** : image de base (Docker va alors aller chercher sur DockerHub l'image correspondante pour s'en servir).
- **COPY** : copier des fichiers depuis la machine locale vers le nouvel environnement
- **RUN** : exécuter des commandes (scripts dans le terminal par exemple)
- **CMD** : commande de démarrage

Prenons un exemple : Ici, je démarre un serveur Nginx Alpine

```
# On se sert de l'image nginx:alpline existante
FROM nginx:alpine
# Je lui demande de copier le contenu du répertoire /dist/browser dans le dossier /usr/share/nginx/html du serveur
COPY --from=build /app/dist/monsite/browser /usr/share/nginx/html
# Je lui demande également de copier un fichier de configuration custom contenant des informations diverses
COPY nginx/default.conf /etc/nginx/conf.d/default.conf
```

```
# Je lui dit d'ouvrir le port 80
EXPOSE 80
# Puis d'exécuter ces commandes sur le serveur nginx en question afin de démarrer nginx
CMD ["nginx", "-g", "daemon off;"]
```

IV. Commandes clés

- Une fois que mon fichier Dockerfile est créé (ne pas hésiter à aller en chercher des tout-prêts sur Internet, la plupart des environnements de dev sont déjà connus…), je dois créer un **conteneur** exécutant cette image. Pour faire ceci, j'utilise dans le répertoire qui contient mon Dockerfile :

  ```
  docker build -t leNomDeMonConteneur .
  ```

  - Le paramètre "-t" nous permet de renommer ce conteneur. C'est uniquement pour notre confort à nous, Docker utilise des ID générés automatiquement.
  - Enfin, le point **.** dit à Docker de lire le Dockerfile dans le répertoire courant

  Ici, Docker va littéralement **construire** (to build) un conteneur sur la base de l'image qu'il va trouver dans le **Dockerfile** du répertoire courant (le point .)

- Pour vérifier que le conteneur que nous voulons créer est bien créé, je peux vérifier son nom, son ID et son statut avec la commande

  ```
  docker ps
  ```

  Ce qui va m'afficher un tableau de ce genre :

```
nicolas@MUSS:~/wsldev/projects$ docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED          STATUS          PORTS                                       NAMES
dde9400ce0d6   angular-burger-app   "/docker-entrypoint.…"   33 minutes ago   Up 33 minutes   0.0.0.0:8080->80/tcp, [::]:8080->80/tcp   amazing_grothendieck
```

- Maintenant que mon conteneur est créé, je dois le lancer. Pour ceci je vais utiliser la commande

  ```
  docker run leNomDeMonConteneur
  ```

  Cette commande s'accompagne très souvent d'un certain nombre de paramètres que l'on va détailler ici :

  - –d ou –-detach : permet de faire tourner le conteneur en arrière plan, et affiche l'ID du conteneur dans la console.
  - –p ou –-expose permet d'ouvrir un port sur le conteneur que l'on est en train de démarrer.

V. Exemples concrets

Pour un projet Angular :

À la racine de n'importe quel projet Angular dans votre dossier wsldev/projects, créez un répertoire "nginx" et un fichier default.conf dedans. Copiez ce contenu dans default.conf :

```
# monsite/nginx/default.conf
server {
  listen 80;
  server_name localhost;
  root /usr/share/nginx/html;
  index index.html;
  # Redirect all routes to index.html for Angular's client-side routing
  location / {
    try_files $uri $uri/ /index.html;
  }
  # Configure caching for static assets
  location ~* \.(js|css|png|jpg|jpeg|gif|ico)$ {
    expires 1y;
    add_header Cache-Control "public, max-age=31536000";
  }
}
```

Ensuite, nous allons créer notre fichier Dockerfile, toujours à la racine. Collez ceci dans le fichier Dockerfile en remplaçant bien "monsite" par le nom de votre projet bien sûr :

```
# monsite/Dockerfile
# Étape 1: générer un build avec notre appli Angular
# On prend une image de Node.js alpine
FROM node:alpine AS build
# On précise dans quel répertoire on travaille
WORKDIR /app
# On récupère les deux fichiers package.json et package-lock.json
COPY package.json package-lock.json ./
# On lance la commande npm install pour générer le dossier node_modules
RUN npm install
# Puis on copie tout le répertoire courant dans la racine du projet
COPY . .
# Enfin, on génère un build
RUN npm run build

# Étape 2: Démarrer l'appli avec un serveur NGINX
```

```
# On se sert de l'image nginx:alpine existante
FROM nginx:alpine
# Je lui demande de copier le contenu du répertoire /dist/browser dans le
dossier /usr/share/nginx/html du serveur
COPY --from=build /app/dist/monsite/browser /usr/share/nginx/html
# Je lui demande également de copier un fichier de configuration custom
contenant des informations diverses
COPY nginx/default.conf /etc/nginx/conf.d/default.conf
# Je lui dit d'ouvrir le port 80
EXPOSE 80
# Puis d'exécuter ces commandes sur le serveur nginx en question afin de
démarrer nginx
CMD ["nginx", "-g", "daemon off;"]
```

Maintenant que notre Dockerfile existe et est rempli, nous allons devoir **monter** cette image sur un conteneur. Pour cela nous allons utiliser **docker build** comme ceci :

```
docker build -t mon-conteneur .
```

Vérifions que le conteneur est bien créé avec **docker ps**

```
docker ps
```

Et si on le voit bien dans la liste, alors on va le lancer en exposant so port 80 pour que nous puissions y avoir accès :

```
docker run -d -p 8080:80 mon-conteneur
```

Une fois cette commande effectuée, vous devriez avoir l'ID du conteneur dans le terminal, et surtout vous devriez avoir accès à votre site en ouvrant http://localhost:8080/

Et ce, sans avoir lancé ng serve ou démarré je ne sais quelle VM. C'est bien le serveur Nginx dans votre conteneur qui vous renvoie votre site ☺ Et pour l'éteindre vous pouvez faire la commande

```
docker stop id-de-mon-conteneur
```

Pour un projet PHP

Voici un exemple de projet PHP classique en PHP8.2, avec Xdebug et phpMyAdmin d'installés, en plus de MariaDB, de MailPit… Oula, ça commence à faire beaucoup de choses installées d'un coup là, est-ce que ça va vraiment tenir dans un seul container ?

Eh bien non ☺ C'est ici que l'on va devoir utiliser la commande spéciale "docker compose", qui a besoin d'un fichier docker-compose.yml

Premièrement, nous allons créer notre projet PHP.

Sur le WSL, dans le dossier wsldev/projects, nous allons créer un répertoire helloworld.

Puis dans ce répertoire, répertoire **src** contenant un fichier **index.php** qui contient simplement ce code :

```php
<?php

echo "Hello world";
```

Ça sera notre site très basique pour l'instant. Remontons dans le dossier **helloworld**, en sortant de **src** avec **cd ../**

Ici nous allons créer un **fichier de config Xdebug (xdebug.ini)**, qui nous servira pour construire notre config :

```
zend_extension=xdebug

xdebug.mode=debug,develop
xdebug.start_with_request=yes
xdebug.client_host=host.docker.internal
xdebug.client_port=9003
xdebug.log_level=0
```

Nous allons également créer un fichier **msmtprc**, sans extension, qui va contenir la config de **MailPit** :

msmtprc

```
defaults
auth           off
tls            off
logfile        /tmp/msmtp.log

account mailpit
host mailpit
port 1025
from dev@example.test

account default : mailpit
```

Enfin, une fois ces deux fichiers de config créés, nous allons pouvoir reprendre le fonctionnement normal de Docker en créant notre **Dockerfile**, que voici :

```
# On tire l'image d'un serveur apache avec PHP 8.2
FROM php:8.2-apache

# On exécute des commandes pour télécharger les dépendances système comme curl, pdo, mbstring, opcache etc... La plupart sont des extensions PHP
RUN apt-get update && apt-get install -y \
    libzip-dev \
    libicu-dev \
    libonig-dev \
    unzip \
    git \
    curl \
    msmtp \
    && docker-php-ext-install \
    pdo \
    pdo_mysql \
    intl \
    zip \
    opcache \
    mbstring

# On autorise le mod_rewrite sur Apache2 pour pouvoir manipuler les URLs
RUN a2enmod rewrite

# Installation de Xdebug
RUN pecl install xdebug \
    && docker-php-ext-enable xdebug

# Copie de la config locale de xdebug.ini dans la config de xdebug sur le serveur créé dans le conteneur
COPY xdebug.ini /usr/local/etc/php/conf.d/xdebug.ini
```

```
# On définit le répertoire de destination sur le serveur
WORKDIR /var/www/html

# Et on ouvre le port 80
EXPOSE 80
```

Ce Dockerfile sert à tirer depuis **DockerHub** l'image d'un serveur Apache2 avec PHP8.2 d'installé, ainsi qu'un certain nombre d'extensions PHP comme cURL, PDO, etc…

Il en manque un peu non ? Où est la config de MariaDB ? de MailPit ? de phpMyAdmin ?

C'est là que va arriver le fichier **docker-compose.yml** : il va permettre à Docker d'associer plusieurs images de différents environnements ensemble. Un peu comme on **composerait** un plat finalement, docker compose nous permet de lister les ingrédients dont notre appli aura besoin pour fonctionner. On va appeler ces ingrédients des **services**.

Analysons le fichier **docker-compose.yml** :

```
services:
  # Configuration du "service" PHP
  php:
    # On construit l'image dans le répertoire courant
    build: .
    # On nomme le container qui va en résulter helloworld
    container_name: helloworld
    # Association du port 8080 du container au port 80 du serveur apache à
l'intérieur
    ports:
      - "8080:80"
    # Définition des répertoires à créer
    volumes:
      - ./src:/var/www/html
      - ./msmtprc:/etc/msmtprc
    # Déclaration des dépendances aux autres services
    depends_on:
      - db
      - mailpit
    environment:
      PHP_IDE_CONFIG: "serverName=docker"
  # Configuration du "service" db
  db:
    # On "tire" l'image de MariaDb version 11
```

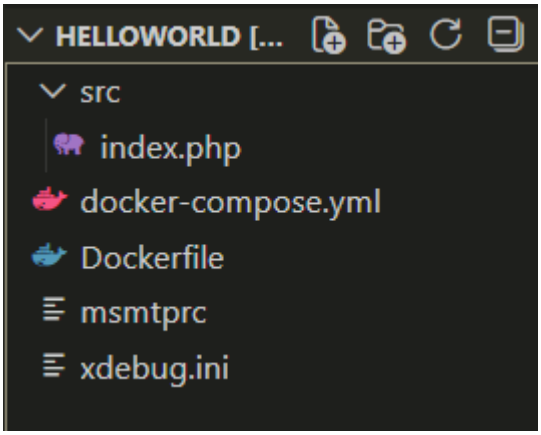
```
    image: mariadb:11
    # L"image de ce service s'appellera mariadb
    container_name: mariadb
    # On redémarre ce service à chaque démarrage du container principal
    restart: always
    # Variables d'environnement : les mots de passes et noms de base !!!
Ne pas perdre ces infos
    environment:
      MARIADB_ROOT_PASSWORD: root
      MARIADB_DATABASE: helloworld
      MARIADB_USER: app
      MARIADB_PASSWORD: app
    # Ouverture des ports 3306 de chaque côté du conteneur
    ports:
      - "3306:3306"
    # Emplacement dans le conteneur où seront contenues les données
    volumes:
      - db_data:/var/lib/mysql
  # Configuration du "service" phpMyAdmin
  phpmyadmin:
    # On "tire" l'image phpMyAdmin
    image: phpmyadmin/phpmyadmin
    # On nomme le conteneur phpmyadmin
    container_name: phpmyadmin
    # On redémarre ce service à chaque démarrage du container principal
    restart: always
    # On fait pointer phpmyadmin sur le port 8081 côté client
    ports:
      - "8081:80"
    environment:
      PMA_HOST: db
      PMA_USER: root
      PMA_PASSWORD: root
    depends_on:
      - db

  # Configuration du "service" mailpit
  mailpit:
    # On "tire" l'image de mailpit
    image: axllent/mailpit
    container_name: mailpit
    restart: always
    # Ouverture des ports
    ports:
      - "8025:8025" # UI
```

```
      - "1025:1025" # SMTP

volumes:
  # On crée un volume "db_data" qui va contenir les données de la base de
données. C'est vers ce volume qu'on fait pointer le service "db"
  db_data:
```

Arborescence finale :



Et maintenant ?

Eh bien nous pouvons lancer la commande suivante

```
docker compose up --build -d
```

Docker va alors lire le fichier docker-compose.yml et trouver toutes les informations dont il a besoin pour **construire** notre conteneur, et le faire en mode “détaché” g^race au paramètre **-d**, en lisant les informations du Dockerfile également.

Une fois la commande terminée, vous devriez avoir accès à la panoplie d’URL suivantes ☺

- Votre site : http://localhost:8080/
- Votre phpMyAdmin : http://localhost:8081/
- Votre MailPit : http://localhost:8025/
- Votre Db :
    - Host : db
    - Port : 3306
    - User : app
    - Password : app
    - Base de données : helloworld

Pour éteindre votre conteneur, vous pouvez maintenant faire la commande suivante :

```
docker compose down
```

Et si vous voulez redémarrer votre conteneur plus tard, vous n'aurez plus besoin de lui préciser –build car il aura déjà été fait. Vous pourrez alors simplement appeler

```
docker compose up -d
```

Et le meilleur dans tous ça : Vos données resteront intactes dans la BDD, même le conteneur éteint…

VI. Partager des images

Docker possède un système assez similaire à celui de Git, auquel d'ailleurs il a emprunté un peu d'inspiration puisqu'il s'appelle DockerHub.

Pour utiliser les images des autres, vous pouvez utiliser **docker pull**, comme par exemple :

```
docker pull ubuntu:24.04
```

Une fois l'image "tirée", vous pouvez vérifier qu'elle est bien connue de docker en utilisant la commande **docker images** pour lister toutes les images connues. Puis vous pourrez la "monter" dans un conteneur en faisant

```
docker run monImage
```

Ce n'est pas tout : vous pouvez aussi vous-même **pousser des images** !

Mais pour cela vous devrez d'abord créer un compte puis vous authentifier sur DockerHub avec :

```
docker login
```

Un code apparaîtra qui sera à saisir dans le navigateur avant de s'y connecter. Une fois que c'est fait, on pourra alors pousser l'image correspondante avec la commande :

```
docker tag angular-burger-app sweepacake/angular-burger-app
```

Pour renommer l'image en accordance avec le nom d'utilsateur attendu sur le dépôt, puis :

```
docker push sweepacake/angular-burger-app
```