

TD n° 10

Conteneurisation avec Docker

Dans ce dernier « grand » TD, nous allons explorer les mécanismes de conteneurisation. Ce sera l'occasion de revenir sur plusieurs des concepts que nous avons étudiés depuis le début de ce cours : les exécutables statiques, les bibliothèques partagées, les processus, les signaux ... mais aussi des concepts vus dans d'autres cours, comme celui de réseau.

Nous allons donc installer docker sur votre machine pour réaliser ces différents exercices.

1 Installation

Commencez par récupérer l'archive disponible sur la page de ce cours. Cette archive contient un script (`install.sh`) qui, en l'exécutant dans votre machine virtuelle de travail, vous permettra d'installer automatiquement docker. Une fois l'installation faite, si vous rencontrez un problème en lançant la commande `docker ps -a`, vous vous déconnecterez et reconnecterez.

Pour vous assurer que votre installation de docker est bien fonctionnelle, vous lancerez la commande suivante et vérifierez que vous obtenez bien l'affichage d'un message du type « Hello from Docker! »

```
docker run hello-world
```

2 Construction d'images

Comme souvent en informatique, vous allez commencer par « Hello, World ! ». A l'aide de différentes images de base, vous allez créer un conteneur qui affichera ce message. Pour vous aider à démarrer plus vite, vous trouverez plusieurs dossiers dans l'archive sur le site du cours.

2.1 Ubuntu

Exercice n°1:

La première image que vous allez créer sera basée sur l'image `ubuntu:22.04`. Cette image devra avoir pour nom `hello-world` et pour tag `ubuntu-22.04`. Créez le fichier `Dockerfile` permettant de construire cette image, contenant le programme `hello.exe` que vous avez créé à l'aide du code source dans l'archive.

Vous testerez que cette image est bien fonctionnelle et que vous obtenez l'affichage du message.

2.2 Alpine

Exercice n°2:

La deuxième image à créer sera quant à elle basée sur `alpine:3.17.3`. Cette image devra avoir pour nom `hello-world` et pour tag `alpine-3.17`. Créez le fichier `Dockerfile` permettant de construire cette image et qui contiendra le fichier `hello.exe`. Testez que votre image fonctionne bien (ou pas).

Exercice n°3:

Si vous obtenez une erreur lors de l'exécution de ce conteneur, pouvez-vous en expliquer la raison ? Quel(s) correctif(s) pouvez-vous apporter ? (Il y a deux solutions possibles pour corriger le problème rencontré, mais n'en appliquez qu'une).

2.3 Scratch

Exercice n°4:

Enfin, vous ferez une dernière version de hello-world basé sur `scratch` (le système de fichier vide pour une image docker), image que vous nommerez toujours `hello-world` et avec le tag `scratch`. Vérifiez que cette image est bien

TD n° 10

Conteneurisation avec Docker

fonctionnelle. Si ce n'est pas le cas, corrigez le problème avec une des deux solutions vues avec la question précédente. Laquelle est réellement applicable dans ce cas particulier ?

2.4 Comparaison des images obtenues

Vous disposez donc maintenant de 3 images `hello-world` : `hello-world:latest` (qui a été récupérée à la fin de l'installation pour tester que `docker` fonctionne bien) et les trois images que vous venez de créer : `hello-world:ubuntu-22.04`, `hello-world:alpine-3.17` et `hello-world:scratch`.

Exercice n°5:

Commencez par comparer la taille de vos 3 images. N'est-ce pas un peu beaucoup pour afficher le message « Hello, World ! » ? A l'aide de l'outil `dive` (<https://github.com/wagoodman/dive#installation>), que vous installerez sur vos machine, comparez rapidement les couches (*layers*) qui constituent chacune des images. Comparez ces tailles d'images avec celle du `hello-world` par défaut (`hello-world:latest`). Comment pouvez-vous expliquer que cette image soit plus petite que la plus petite que vous avez créée ?

3 Un exemple plus complet : un serveur web avec Node.js

Nous allons maintenant réaliser un exemple un peu plus complet qui parte de la construction de l'image, en passant par l'optimisation de celle-ci et jusqu'à la résolution de problèmes classiques.

Nous allons donc faire un petit serveur Web à l'aide de Node.js et de la bibliothèque Express.js. Pour éviter de perdre du temps inutilement avec la mise en place du code, j'ai demandé à ChatGPT (voir Annexe) de fournir un code prêt à l'emploi. Celui-ci vous est donné dans l'archive que vous avez récupérée sur le site du cours. Tout se trouve dans le dossier `node`.

3.1 Vérification que le code est bien fonctionnel sur votre machine

Avant de faire un conteneur avec notre serveur, nous allons vérifier que le code fonctionne bien sur notre machine. Si vous ne disposez pas encore de Node.js dans l'environnement de travail de ce cours (la VM de travail), vous pouvez simplement l'installer grâce aux commandes suivantes :

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -  
sudo apt-get install nodejs
```

Pour vérifier que ce code fonctionne bien, vous lancerez les commandes suivantes dans le dossier `node` :

```
npm install  
npm start
```

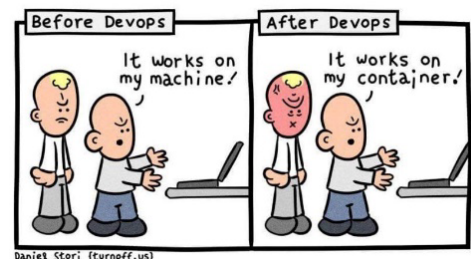
puis vous vous connecterez avec votre navigateur à l'adresse <http://localhost:8080/>

Mais...

- « Cela marche sur ma machine. »
- « Oui, mais tu ne vas pas fournir ta machine au client ! »

3.2 Conteneurisation de notre serveur

Maintenant que nous avons un code fonctionnel sur notre machine, nous allons conteneuriser cette application/ce service afin de pouvoir facilement le déployer sur n'importe quelle machine qui disposera de Docker et sans avoir à installer Node.js sur la machine hôte ainsi que tous les paquetages nécessaires au fonctionnement de l'application.



TD n° 10

Conteneurisation avec Docker

Exercice n°6:

Modifier le fichier `Dockerfile` qui vous a été fourni pour fixer les numéros de version de l'image de base de votre conteneur à la version `18.16.0-alpine3.17`. En effet, afin d'assurer la répliquabilité dans le temps de votre système, il est nécessaire de fixer très précisément les numéros de version avec lesquels votre code fonctionne. Souvent les logiciels libres ne garantissent pas à 100% la « *backward compatibility* » car c'est un effort énorme. D'expérience, cela peut même être vrai sur le numéro de patch utilisé uniquement pour les mises à jour de sécurité...

3.3 Exposition du port

Exercice n°7:

Pour tester votre conteneur, ajouter une cible `run` au `Makefile` qui vous a été donné. Vous penserez à partager le port entre votre conteneur et de l'exposer sur votre machine hôte. Vous veillerez à l'exposer sur un autre port que le 8080, par exemple le port 8888. En effet, ceci vous permettra de faire tourner en même temps votre serveur directement sur votre machine (sur le port 8080) et votre conteneur avec le même serveur en parallèle mais sur un autre port (sur le port 8888, par exemple).

3.4 Inspection de l'image produite

Votre image doit maintenant être fonctionnelle. Mais contient-elle uniquement ce qu'il faut pour l'exécuter ?

Exercice n°8:

A l'aide l'outil `dive`, que vous avez installé précédemment, inspectez les différentes couches (*layers*) de votre image. A quoi correspondent les premiers (*layers*) ? A partir de quel layer commencent vos modifications ? Dans les fichiers que vous avez ajoutés avec la commande `COPY`, n'y a-t-il pas un ou plusieurs problèmes ? Utilisez un fichier `.dockerignore` pour corriger le(s) problème(s) (utilisation identique à `.gitignore`).

Et quel(s) pourrai(en)t être le(s) problème(s) spécifique(s) avec le dossier `node_modules` ?

4 Isolation des conteneurs du système de fichiers et des variables d'environnement

Le fait d'isoler l'exécution d'une application/service est très intéressante. Cela permet de redéployer cette application sur d'autres machines très facilement par exemple. Mais il faut tenir compte du fait que l'on doit refournir un environnement d'exécution complet à notre application. Donc cela ne fonctionne pas exactement comme sur votre machine, enfin pas obligatoirement. Il faut donc maîtriser tous les aspects systèmes.

4.1 Mais quelle heure est-il ? Un problème classique avec l'isolation des conteneurs

Vous avez pu constater que notre serveur affiche « Hello, World ! » et affiche la date et l'heure. Ne voyez-vous pas une différence entre ce que vous affiche votre serveur fonctionnant « nativement » et ce qui est affiché par le serveur s'exécutant dans le conteneur ? Nous allons nous pencher sur ce problème.

Il faut savoir que les informations sur l'heure d'une machine sont accessibles via deux fichiers du système d'exploitation : `/etc/localtime` et `/etc/timezone`. Il est aussi possible d'affecter des variables d'environnement pour spécifier le fuseau horaire (`TZ`) dans lequel on se trouve. Nous allons exploiter ces deux pistes pour corriger le problème constaté.

TD n° 10

Conteneurisation avec Docker

4.1.1 Partage de volume (bind mount)

Exercice n°9:

Commençons donc par résoudre ce problème d'heure en partageant les fichiers `/etc/localtime` et `/etc/timezone` avec votre conteneur. Vous veillerez à faire ce partage en lecture seule pour éviter que, par erreur, votre conteneur modifie les fichiers de la machine hôte.

Mais l'image alpine de base ne prend pas en charge la gestion des fuseaux horaires. Il faut ajouter les paquetages `musl-locales` et `tzdata`.

Exercice n°10:

Modifiez votre fichier `Dockerfile` pour y ajouter ces deux paquetages. Vous veillerez à ajouter des paquetages au bon endroit dans la construction de l'image. Vous testerez que cela corrige bien le problème d'affichage de l'heure.

Exercice n°11:

Pouvez-vous maintenant justifier pourquoi le fichier `Dockerfile` contient tout d'abord une première commande qui copie les fichiers `package*.json` puis une deuxième commande qui copie tout ce qui se trouve dans le répertoire courant (or les fichiers `package*.json` font partie des fichiers du répertoire courant) ? En d'autres termes, pourquoi ne pas directement faire `COPY . .` et donc copier d'un coup tous les fichiers `package*.json` et les sources de l'application ?

Pour information, quand on demande à ChatGPT pourquoi il a proposé de faire deux commandes `COPY`, il dit qu'il s'est trompé et vous propose une version de `Dockerfile` avec un seul `COPY` (qui ne copie que les `package*.json` et pas le code de l'application...). Alors que dans ce cas précis, c'est mieux avec deux `COPY` dans le cas du développement de l'image (comme il l'avait proposé au départ, sans le savoir). Comme quoi, un générateur de texte n'est pas non plus une arme absolue, cela reste une intelligence **artificielle**.

4.1.2 Passage de variables d'environnement au conteneur

Nous allons tenter de résoudre ce problème d'affichage de l'heure par un autre moyen : l'utilisation de variable d'environnement. Vous commencerez par supprimer le partage des fichiers mis en place précédemment lors du lancement du conteneur.

Exercice n°12:

Passez la variable d'environnement `TZ=Europe/Paris` au lancement de votre conteneur et vérifiez que l'heure s'affiche bien correctement.

Exercice n°13:

Quels sont les avantages et inconvénients de chacune des deux solutions pour un affichage correct de l'heure ?

Pour une petite synthèse sur le sujet, vous pouvez lire la page suivante : « [Guide : corriger les problèmes d'heure dans Docker](#) ».

5 Utilisation des Volumes

Nous avons utilisé précédemment la notion de volume pour partager deux fichiers entre la machine hôte et le conteneur grâce à l'option `-v`. Nous allons essayer de creuser un peu plus cette notion de volume. Cette notion permet la persistance des données. En effet, lorsqu'un conteneur est détruit, on peut toutes les modifications faites sur les fichiers du conteneur (cf. le slide du cours sur le layer du conteneur en lecture-écriture (« au-dessus » des layers de l'image en lecture seule) qui est détruit à la destruction du conteneur).

TD n° 10

Conteneurisation avec Docker

5.1 Volume de type bind mount (ou « non nommé »)

Ce premier type de volume, que nous avons déjà expérimenté dans un exercice précédent, permet de partager un fichier ou un dossier entre le système de fichiers de la machine hôte et le conteneur. Le terme utilisé dans ce cas est « *bind mount* » (même si ce terme est trompeur car en fait tous les volumes sont techniquement « *bind mounted* ». La différence, c'est que le point de montage est explicite sur le système de fichiers hôte plutôt que caché dans un répertoire géré par Docker, derrière un nom donné pour le volume (d'où le nom « volume nommé », comme nous pourrions le voir dans la section suivante).

Pour mieux comprendre ce qu'est un volume « *bind mount* », nous allons lancer un conteneur en mode interactif et l'associer à un dossier `data` qui existe sur votre système de fichier hôte.

```
mkdir data
docker run --rm -it -v ./data:/data alpine:3.17 /bin/sh
```

Exercice n°14:

Dans le conteneur, déplacez-vous dans le dossier `/data`. Créez un fichier vide s'appelant `todel` (commande `touch`). Vérifiez que ce fichier existe maintenant dans le dossier `./data` sur la machine hôte. Editez ce fichier depuis votre machine hôte pour ajouter une contenu quelconque. Retournez dans votre conteneur pour afficher le contenu du fichier `todel`.

5.2 Volume nommé

Nous allons créer un conteneur mais cette fois avec un volume nommé. Pour ce faire, il faut manuellement créer un volume nommé à l'aide de la commande : `docker volume create voldata`. Vous venez de créer un volume avec le nom `voldata`. Pour consulter les informations sur ce volume, vous pourrez utiliser la commande suivante : `docker inspect voldata`.

Exercice n°15:

Sur le même modèle que l'exercice précédent, créez un conteneur, mais cette fois-ci utilisant le volume `voldata` que vous venez de créer. Vous vérifierez en particulier la persistance des données : Une fois un premier conteneur créé qui vous aura permis de créer un fichier `todel` dans `voldata`, vous arrêterez et détruirez celui-ci avant de relancer un nouveau conteneur qui réutilisera le même volume `voldata`. Vous vérifierez que vous avez bien toujours accès aux fichiers que vous avez créés avec le premier conteneur. Vous tenterez aussi de consulter depuis la machine hôte le dossier qui correspond à l'endroit où sont stockées les données de `voldata`. Vous expliquerez pourquoi seul le super-utilisateur peut consulter le contenu de ce dossier depuis la machine hôte.

6 Introduction à la sécurité d'un conteneur

Nous allons maintenant introduire quelques aspects sécurité concernant la conteneurisation.

Sans aucune précision de votre part lors du lancement du conteneur ou bien dans le `Dockerfile`, les fichiers de l'image et l'exécution des processus dans le conteneur se feront en tant que `root`, donc en tant que super-utilisateur. Bien entendu, d'un point de vue sécurité, ce n'est pas souhaitable. Si le conteneur est attaqué, l'attaquant pourrait gagner des privilèges de super-utilisateur sur la machine hôte elle-même.

6.1 Utilisateur exécutant l'application

Pour nous rendre compte de ce phénomène, vous commencerez par constater avec l'outil `dive` appliqué à l'image `node-app` que tous les fichiers dans `/app` appartiennent au super utilisateur (`uid=0` et `gid=0`). Donc quand on copie

TD n° 10

Conteneurisation avec Docker

les fichiers dans le conteneur, tous appartiennent au super-utilisateur, et donc avec des droits/permissions en conséquence.

Exercice n°16:

Connectez-vous dans un Shell s'exécutant dans le conteneur depuis votre machine hôte. Pour rappel la commande à utiliser est la suivante : `docker exec -it node-app /bin/sh`

Quel est le nom de l'utilisateur avec lequel vous êtes connecté dans le conteneur (commande `whoami`) ? Quels sont les identifiants et groupes auxquels vous appartenez (commande `id`) ?

6.2 Changer l'utilisateur par défaut

Pour rendre moins vulnérable votre conteneur qui sera exposé via la redirection de port mise en place précédemment, nous allons exécuter celui-ci sous une identité moins vulnérable que `root`.

Exercice n°17:

Nous allons démarrer notre conteneur avec l'utilisateur `node` grâce à l'option `--user` de `docker run`. Modifiez pour cela le `Makefile`.

Mais au lancement du conteneur, vous constatez une erreur vous indiquant que les permissions sur le fichier `package.json` ne sont pas correctes (`Error: EACCES: permission denied, open '/app/package.json'`) et ne permettent pas l'exécution du service.

Exercice n°18:

Corrigez le fichier `Dockerfile` afin de rendre votre conteneur exécutable par un autre utilisateur que `root` pour mieux sécuriser votre application. Afin de vous guider dans cette tâche, vous devez réfléchir au bon moment où introduire le basculement vers l'utilisateur `node` à l'aide de la commande `USER` dans le `Dockerfile`. Vous utiliserez aussi l'option `--chown` de la commande `COPY` pour copier les fichiers avec les bonnes permissions dans le conteneur (pour que les fichiers appartiennent à l'utilisateur `node` et au groupe `node`).

Exercice n°19:

Vérifiez maintenant que quand vous vous connectez à votre conteneur, vous n'êtes plus l'utilisateur `root`.

Si vous souhaitez quelques explications complémentaires à ce sujet, vous pouvez lire la page suivante : « [Running a Docker container as a non-root user](#) ».

7 Introduction au réseau de conteneurs

Maintenant que notre conteneur `node-app` est un peu plus sécurisé, nous pouvons aller un peu plus loin dans notre réflexion. Est-il toujours nécessaire de « mapper » le(s) port(s) d'un conteneur sur la machine hôte. Il arrive fréquemment qu'une application soit composée de plusieurs conteneurs (micro-services). Il est alors intéressant de ne pas obligatoirement faire passer toutes les communications via la machine hôte, mais via un réseau où seuls les conteneurs ont accès pour parler entre eux et ainsi n'exposer que les ports nécessaires vers l'extérieur.

7.1 Création d'un réseau

La création d'un réseau est réalisée grâce à la commande suivante :

```
docker network create mynetwork
```

TD n° 10

Conteneurisation avec Docker

Exercice n°20:

Vous commencerez par lancer le conteneur `node-app` en utilisant le réseau que nous venons de créer. Vous veillerez que le port 8080 exposé par votre conteneur ne soit pas accessible depuis la machine hôte (que ce soit par le port 8080 ou le port 8888 configuré précédemment).

7.2 Utilisation d'un réseau pour communiquer entre conteneurs

Exercice n°21:

Maintenant pour nous assurer que la communication entre deux conteneurs est possible au travers du réseau créé, vous construirez une nouvelle image s'appelant `test` qui contiendra le client `curl`. Cette image ne lancera pas de commande car nous nous contenterons de l'utiliser interactivement pour nos tests. Vous lancerez cette image de test interactivement et tenterez de vous connecter à l'aide de la commande `curl` au conteneur `node-app` pour récupérer le contenu de la page web fourni par celui-ci.

Cet exemple « jouet » met en évidence l'intérêt de limiter la visibilité des conteneurs au strict nécessaire. Toutefois, celui-ci est « inversé », le serveur Web étant celui qui est masqué de l'extérieur.

Dans un vrai cas d'utilisation plus réaliste, nous pourrions avoir une application composée de deux services ; un serveur web qui serait dans un conteneur et visible depuis la machine hôte, une base de donnée dans un autre conteneur. Pour des raisons évidentes de sécurité, on éviterait d'exposer le port de communication avec la base de données sur la machine hôte. On établirait alors la communication entre le conteneur du serveur web et le conteneur de la base de données dans un réseau spécifique de docker comme nous venons de le mettre en pratique. Mais mettez concrètement cela en pratique dans le cadre du projet PS6.

TD n° 10

Conteneurisation avec Docker

Annexe

Pour démarrer plus vite le projet de la section 3, j'ai demandé à ChatGPT de me fournir un code de base pour un serveur en Node.js.



Je souhaite faire un serveur web à l'aide de Node.js et Express qui affiche "hello world" et la date qui écoute sur le port 8080. Peux-tu me donner le code ?



Bien sûr ! Voici le code que vous pouvez utiliser pour créer un serveur web à l'aide de Node.js et Express qui affiche "Hello World" et la date :

```
const express = require('express');
const app = express();

const PORT = 8080 ;

app.get('/', (req, res) => {
  const date = new Date().toLocaleString();
  res.send(`Hello World! Today is ${date}`);
});

app.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}`);
});
```

Dans ce code, nous importons le module express et créons une instance de l'application Express en utilisant la fonction `express()`. Nous définissons ensuite la constante `PORT` pour stocker le numéro de port sur lequel notre serveur écoutera.

Ensuite, nous définissons une route pour la racine de notre application (`/`) en utilisant la méthode `app.get()`. Dans cette route, nous créons une instance de l'objet `Date` pour obtenir la date actuelle, puis nous utilisons la méthode `res.send()` pour envoyer une réponse avec le texte "Hello World! Today is" suivi de la date formatée.

Enfin, nous appelons la méthode `app.listen()` pour démarrer notre serveur web et lui faire écouter les connexions entrantes sur le port spécifié. Lorsque le serveur démarre, nous utilisons la méthode `console.log()` pour afficher un message indiquant que le serveur est en cours d'écoute sur le port spécifié.

J'espère que cela vous aidera à démarrer avec Node.js et Express !



Peux-tu me fournir le `Dockerfile` pour une image compacte qui inclus ce serveur web ?



Absolument ! Voici un exemple de `Dockerfile` qui vous permet de créer une image Docker compacte pour votre serveur web Node.js / Express :

```
FROM node:alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD ["npm", "start"]
```




TD n° 10

Conteneurisation avec Docker

Ce `Dockerfile` utilise l'image `node:alpine` comme image de base, ce qui crée une image Docker légère basée sur la distribution Alpine Linux. Il définit le répertoire de travail de l'image à `/app`, puis copie le fichier `package.json` et `package-lock.json` à l'intérieur.

Ensuite, il exécute la commande `npm install` pour installer les dépendances de l'application. Il copie tout le contenu du répertoire courant (qui doit contenir votre fichier JavaScript de serveur web) à l'intérieur de l'image, puis expose le port 8080 pour les connexions entrantes.

Enfin, il définit la commande `npm start` comme la commande par défaut pour lancer l'application lorsque le conteneur est démarré.

Pour construire votre image Docker, vous pouvez utiliser la commande `docker build` à partir du même répertoire que votre `Dockerfile`. Par exemple :

```
docker build -t node-app .
```

Cela construira une nouvelle image Docker avec le nom `node-app` en utilisant le `Dockerfile` situé dans le répertoire courant. Vous pouvez ensuite utiliser la commande `docker run` pour démarrer un conteneur à partir de cette image.



La commande `docker run` que tu me demandes de lancer ne fonctionne pas. Peux-tu me donner la commande pour lancer ce conteneur ?



Bien sûr, voici la commande que vous pouvez utiliser pour lancer un conteneur à partir de votre image Docker `node-app` :

```
docker run -p 8080:8080 node-app
```

Cette commande utilise la commande `docker run` pour démarrer un nouveau conteneur à partir de votre image `node-app`. L'option `-p` est utilisée pour mapper le port 8080 du conteneur vers le port 8080 de votre machine hôte, ce qui permettra de se connecter à votre serveur web. La commande spécifie également le nom de l'image `node-app`.

Après avoir exécuté cette commande, vous devriez voir le message "Server listening on port 8080" s'afficher dans la console. Vous pouvez ensuite ouvrir un navigateur web et accéder à l'adresse `http://localhost:8080` pour voir votre application en action et afficher "Hello World" et la date.

J'espère que cela vous aidera à lancer votre conteneur Docker et à voir votre application Node.js en action !