Virtualisation

Professeur : Florent Gluck Assistant : Sébastien Chassot

April 13, 2022

Dockerfiles

Objectif

Le but de ce travail pratique est de vous familiariser avec la création d'images grâce à l'utilisation de Dockerfiles.

Généralement, un container Docker devrait réaliser une fonction spécifique (évitez les container "fourre-tout" !). Lorsqu'une application gagne en complexité, il devient souhaitable de la modulariser en fonctionnalités distinctes (serveur web, DB, etc.), chacune s'exécutant dans un container dédié.

Exercice 1

Sur le git du cours, vous pouvez trouver l'arborescence de fichiers liée à ce travail pratique dans le sous-répertoire `resources`. Dans le répertoire `ex01` se trouve un Dockerfile, quelques fichiers et une archive `tar.gz`. Inspectez le contenu du Dockerfile pour vous faire une idée de ce qu'il réalise.

Créez une image à partir de ce Dockerfile, nommez la `dockerfiles_ex01` et taggez la `v1.0`. Vérifiez que cette nouvelle image est présente dans la repository de votre serveur Docker, puis instanciez un container depuis cette image.

a) Décrivez ce que réalise ce container en expliquant chaque ligne du Dockerfile.

b) Pourquoi le fichier `file04` n'est pas présent dans le répertoire `/dir_add` du container ?

c) En observant la sortie du container, que remarquez-vous au niveau des attributs des fichiers ajoutés ? D'où vient cette différence ?

d) Inspectez l'historique (non-tronquée) de l'image que vous avez créée et décrivez chaque *layer* (couche) de l'image.

e) Re-créez l'image en tant qu'utilisateur root. Est-ce que cela change quelque chose ? Justifiez.

Exercice 2

Dans le répertoire `ex02` de l'arborescence, observez les différences entre les quatre Dockerfiles.

Ecrivez un petit script `bash` afin de créer 4 images nommées `dockerfiles_ex02:v1`, `dockerfiles_ex02:v2`, `dockerfiles_ex02:v3`, `dockerfiles_ex02:v4`, une par Dockerfile. A noter que `bash` vous permet de réaliser facilement des boucles, par exemple:

```
for i in {1..4}; do echo $i; done
```

A savoir aussi que `docker build` est capable de lire le contenu d'un Dockerfile depuis l'entrée standard (`stdin`) en spécifiant le caractère "`-`" en fin de ligne (cf. `docker help build`).

Une fois ces 4 images créées, comparez-les avec `history` et `inspect`.

a) Quelles sont les tailles respectives de ces images ?

b) Comment expliquez-vous ces différences de tailles ?

c) Ecrivez un script qui affiche le nombre de couches de chaque image (indice: `grep` et `wc` sont vos amis).

d) Quel est l'intérêt de la ligne `ENV DEBIAN_FRONTEND noninteractive` dans le 4ème Dockerfile (indice: `man debconf`) ?

Exercice 3

On désire créer un container Docker permettant de convertir (non-récursivement) les images du répertoire courant dans tout autre format d'image.

Pour information, la commande `mogrify -format png *.jpg` permet de convertir au format PNG tous les fichiers (du répertoire courant) se terminant par l'extension `jpg`. A savoir que l'outil `mogrify` est disponible dans le package `imagemagick`.

Votre solution doit respecter les points suivants :

- On doit pouvoir spécifier, au moment de l'instantiation du container, en quel format les images doivent être converties, de même que les images à convertir. Par exemple, passer l'argument `a*.jpg"` doit convertir tous les fichiers du répertoire courant commençant par `a` et se terminant par `.jpg`.
- Au cas où aucun argument n'est spécifié, on désire qu'un texte d'aide soit affiché, indiquant la syntaxe à utiliser.
- On veut que le Dockerfile se base sur une image de distribution Linux spécifique à une version donnée (donc pas `latest`), ceci afin de rendre le container le plus portable possible. Le choix de la distribution Linux est libre.
- On désire une image aussi petite que possible.
- Afin d'éviter une accumulation inutile de containers, on désire qu'après exécution du container, celui-ci soit supprimé.

Pour simplifier la résolution de cet exercice, on part du principe que client et daemon Docker s'exécutent sur la même machine.

Notez qu'une manière relativement simple pour implémenter ce type de container est de créer un script qui sera exécuté à l'instanciation du container. Cela permet d'être flexible sur la gestion des paramètres et du comportement du programme à ensuite exécuter.

Donnez :

a) Le contenu d'un Dockerfile permettant de réaliser ce qui est demandé ainsi que toutes dépendences nécessaires (scripts, etc.)
b) La commande permettant de builder l'image Docker à partir du Dockerfile
c) La commande d'instantiation du container pour convertir toutes les images du répertoire courant matchant le nom "*a*.jpg" en images PNG
d) L'UID et GID des images (bitmap) créées par votre container ? Comprenez vous pourquoi ?

Il est possible d'indiquer à Docker d'exécuter le point d'entrée du container avec un utilisateur autre que root, grâce à l'argument `--user` passé à `docker run`.

- Modifiez donc la commande exécutant votre container afin que les fichiers créés possèdent appartiennent à l'utilisateur courant. Pour cela, inspectez l'aide de la commande `docker run`. Votre solution doit fonctionner quel que soit le UID de l'utilisateur courant (donc celui-ci ne doit pas être hardcodé ! Indice: `id`).

Exercice 4

Nous sommes intéressés à créer une image Docker pour le programme `asciicat`, dont le code source GO est disponible sur github à l'url suivante : https://github.com/alfg/asciicat.

`asciicat` prend en argument une image bitmap et produit sur la sortie standard une représentation de l'image en ASCII art. Par exemple, pour afficher l'image `tux.png` en ASCII art, il suffit d'exécuter :

```
asciicat -i tux.png
```

Pour compiler le code source du programme, `main.go`, le compilateur GO est nécessaire. Vous pouvez produire l'exécutable `asciicat` en exécutant les commandes suivantes :

```
go get github.com/alfg/asciicat
go build
```

Tout comme pour l'exercice précédent, on part du principe que client et daemon Docker s'exécutent sur la même machine.

Partie 1

Donnez le contenu d'un Dockerfile permettant de créer l'image `asciicat` en respectant les points suivants :

- Le container instancié à partir de cette image doit au minimum prendre en argument l'image à affiche en ASCII art.
- L'argument optionnel `-w` qui permet de définir la largeur de "l'image" (en caractères) affichée doit être supporté (p.ex. `-w 40`).
- Le container doit pouvoir lire toute image se trouvant dans le répertoire courant côté client.
- Lors du *build* de l'image, le code source du programme `asciicat` doit être obtenu depuis le git du projet.

Une fois les points ci-dessus réalisés :

a) Quelle est la taille de l'image générée ?
b) Quelle est la ligne de commande à exécuter pour afficher l'image `cat.png` (se trouvant dans le répertoire courant) en ASCII art sur une largeur de 40 caractères ?

Partie 2

Etant donné que l'image générée est volumineuse, nous aimerions en minimiser la taille. Utilisez le mécanisme de *multi-stage build* en combinaison avec une distribution Alpine afin de produire une image finale dont la taille est la plus petite possible.

- Donnez le contenu du Dockerfile correspondant.
- Quel est le nom de l'image pour la dernière version de la distribution Alpine disponible sur DockerHub ?
- Quelle est la taille de cette nouvelle image et quel facteur de taille avez-vous gagné par rapport à l'image précédente ?

Partie 3

On désire finalement figer notre image de sorte à ne pas dépendre d'une image externe qui pourrait potentiellement changer (même de manière minime) au cours du temps. Pour cela, vous utiliserez une archive locale comme système de fichiers racine pour votre container (couche de base).

Décrivez exactement et exhaustivement toutes les étapes réalisées pour parvenir à cet objectif, ainsi que le contenu du Dockerfile implémenté.