Virtualisation des systèmes

Florent Glück

May 04, 2025

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 conteneur devrait réaliser une fonction bien spécifique, donc évitez les conteneurs "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 conteneur dédié.

Exercice 1

Sur le git du cours, vous pouvez trouver l'arborescence de fichiers pour cet exercice dans l'archive `lab-dockerfiles.files.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 conteneur depuis cette image.

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

b) Pourquoi le fichier `file04` n'est pas présent dans le répertoire `/dir_add` du conteneur (indice: inspectez les fichiers dans `lab-dockerfiles`) ?

c) En observant la sortie du conteneur, 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

On aimerait déployer un serveur web, basé sur `darkhttpd`, de manière conteneurisée. Pour parvenir à la solution finale, vous allez passer par trois itérations.

Étape 1

À l'aide d'un Dockerfile nommé `Dockerfile.darkhttpd.v1`, créez l'image `darkhttpd:1.0` basée sur `alpine 3.21` et le programme `darkhttpd`. On aimerait que le répertoire racine des fichiers servis par le service `darkhttpd` soit `/var/www/darkhttpd`. Faites en sorte que le fichier `index.html` ci-dessous y soit présent :

```
<!DOCTYPE html>
<html>
<body>
<h1>Welcome to darkhttpd!</h1>
<p>If you see this page, it means the darkhttpd web server is running successfully.</p>
<p>Check the project's homepage at <a href="https://unix4lyfe.org/darkhttpd/">https://unix4
lyfe.org/darkhttpd/</a></p>
</body>
</html>
```

Le point d'entrée de votre image doit être le programme `darkhttpd`. Vous pouvez lister les arguments possibles du programme en exécutant celui-ci sans argument. Assurez-vous que `darkhttpd` soit exécuté en root et écoute sur le port 2000 dans le conteneur.

Depuis votre machine client vérifiez que votre serveur web conteneurisé fonctionne comme attendu...

Vous devriez remarquer que vous n'arrivez pas à communiquer avec votre serveur web !

C'est normal, car aucun port du conteneur n'est exposé vers l'hôte. Pour cela, vous devez indiquer à Docker d'exposer tout port d'intérêt vers un port de l'hôte. Lisez ici comment rediriger le port 2000 du conteneur vers le port 8000 de l'hôte : https://docs.docker.com/get-started/docker-concepts/running-containers/publishing-ports/#publishing-ports.

Vérifiez donc que votre serveur web est atteignable sur le port 8000 (de l'hôte).

Étape 2

On désire maintenant que le programme `darkhttpd` ne soit pas exécuté par l'utilisateur root, mais qu'il soit exécuté par un utilisateur non privilégié (d'où l'intérêt à écouter sur un port non privilégié > 1024).

Pour cela, créez le fichier `Dockerfile.darkhttpd.v2` utilisé pour créer l'image `darkhttpd:2.0`. Assurez-vous de créer l'utilisateur `www` à la création de l'image. Cet utilisateur doit être non privilégié, système, sans répertoire *home* et sans mot de passe. Lisez l'aide de la commande `adduser` dans alpine pour vous aider (argument `--help`).

À nouveau, vérifiez que votre serveur web est atteignable sur le port 8000 (de l'hôte).

- Comment pouvez-vous vérifier que le processus `darkhttpd` est bien exécuté par l'utilisateur `www` dans le conteneur ?
- Quel est le user ID (UID) de `www` dans le conteneur ?
- Quels sont le user et le UID du processus `darkhttpd` sur l'OS hôte ?
- Que se passe-t-il si vous modifiez l'image afin d'exécuter `darkhttpd` sur le port 80 ?

Étape 3

On désire maintenant que le conteneur ne serve plus un fichier contenu dans l'image, mais serve des fichiers se trouvant à l'extérieur du conteneur, donc dans un volume.

Créez donc un volume pour cela et ajoutez-y une arborescence de fichiers html qui seront servis par votre service `darkhttpd`.

Comme auparavant, vérifiez que votre serveur web est atteignable sur le port 8000 (de l'hôte) et que les fichiers du volume sont servis comme attendu.

Exercice 3

On désire créer un conteneur 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 :

- Votre fichier Dockerfile doit être nommé `Dockerfile.convert`
- On doit pouvoir spécifier, au moment de l'instantiation du conteneur, 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`.
- Les fichiers générés par le conteneur doivent appartenir à l'utilisateur ayant exécuté le conteneur et non à root. Pour rappel, pour exécutez un conteneur avec un utilisateur spécifique relisez les slides de cours "06-Docker_overview" (et aussi ce que réalise la commande `id`).
- 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 comportement du conteneur plus stable. Le choix de la distribution est par contre libre.
- On désire une image aussi petite que possible.
- Afin d'éviter une accumulation inutile de conteneurs, on désire que le conteneur soit supprimé une fois son exécution terminée.

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

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

Donnez :

a) Le contenu de `Dockerfile.convert` permettant de réaliser ce qui est demandé ainsi que toutes dépendences nécessaires (scripts, etc.) a) La commande permettant de fabriquer l'image à partir de `Dockerfile.convert` a) La commande d'instantiation du conteneur pour convertir toutes les images du répertoire courant matchant le nom "**a**.jpg" en images PNG

Exercice 4

Nous sommes intéressés à créer une image contenant le jeu alienwave, dont le code source C est téléchargeable ici https://www.alessandropira.org/alienwave/alienwave-0.4.0.tar.gz. Ce jeu utilise un affichage en mode texte et nécessite la librairie de développement ncurses pour être compilé correctement.

Étape 1

Donnez le contenu d'un Dockerfile, basé sur une image Debian 12, permettant de créer l'image alienwave dans lequel se trouve l'exécutable `alienwave` fonctionnel.

Essayez de minimiser la taille de cette image tout en vous assurant qu'un conteneur basé sur celle-ci soit parfaitement fonctionnel.

a) Quelle taille d'image arrivez-vous à obtenir?

Étape 2

Afin de minimiser encore plus la taille de l'image, utilisez le mécanisme de **multi-stage build** en combinaison avec une distribution Alpine afin de produire une image finale la plus petite possible.

a) Quel est le nom de l'image pour la dernière version de la distribution Alpine disponible sur DockerHub ? a) Quelle taille d'image avez-vous ainsi réussi à obtenir?

Informations importantes

En général, dans les systèmes GNU Linux, il existe deux types de librairies :

- Les librairies nécessaires à l'**exécution** de programmes qui en dépendent, comme p.ex. :
  - libncurses6
  - libsdl2-image-2.0-0
  - libjpeg8
- Les librairies nécessaires au **build (édition des liens)** de programmes sources qui en dépendent, comme p.ex. :
  - libncurses-dev
  - libsdl2-image-dev
  - libjpeg8-dev

Les distributions GNU Linux n'utilisent pas toutes la même librairie C. La plupart des distributions desktop/serveur utilisent la GNU libc (glibc). Alpine, par exemple, utilise une libraire C beaucoup plus minimaliste : musl. Celle-ci est incompatible au niveau binaire avec la glibc. Cela signifie qu'une application liée dynamiquement avec la librairie glibc ne fonctionnera pas avec la librairie C musl. Le moyen le plus simple pour rendre une application compatible entre systèmes possédant des librairies différentes (C ou autres), est de les lier statiquement plutôt que dynamiquement. Pour cela, il suffit de passer l'argument `-static` au moment de l'édition des liens, pour autant que les librairies de développements existent au format statique (librairies au format archive → se terminant par l'extension `.a`).

Dans le cas du code source d'alienwave de cet exercice, le `Makefile` de celui-ci réalise une édition des liens dynamique avec la librairie ncurses, comme indiqué à cette ligne :

```
LIB = -lncurses
```

Pour réaliser une édition des liens statique, il est nécessaire de préciser `-static`. Ceci devrait être suffisant, mais dans ce cas particulier, il faut également explicitement spécifier la librairie aditionnelle `tinfo`. Au final cela donne donc :

```
LIB = -lncurses -ltinfo -static
```

Enfin, bien qu'alienwave soit compilé statiquement, certaines ressources liées au terminal sont nécessaires pour son bon fonctionnement. Le package `ncurses` est donc nécessaire dans l'image Alpine finale.

Exercice 5

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

Le programme `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
```

Tout comme pour l'exercice 3, on part du principe que client et daemon Docker s'exécutent sur la même machine.

Le compilateur GO est nécessaire (package `golang` dans Debian/Ubuntu) pour compiler et installer `asciicat`. Celui-ci peut être compilé et installé dans `~/go/bin/` de l'utilisateur courant via le compilateur go avec :

```
go install github.com/alfg/asciicat@latest
```

Il se peut toutefois que go ne reconnaisse pas l'autorité de certification durant le fetch du code source. Dans ce cas, assurez-vous d'installer les certificats nécessaires qui se trouvent dans le package Debian/Ubuntu `ca-certificates`.

Étape 1

Donnez le contenu d'un Dockerfile nommé `Dockerfile1` permettant de créer l'image `asciicat` en respectant les points suivants :

- Le conteneur instancié à partir de cette image doit au minimum prendre en argument l'image à afficher 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 conteneur doit pouvoir lire toute image se trouvant dans le répertoire courant.

Une fois les points ci-dessus réalisés :

a) Quelle est la taille de l'image générée ? a) 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 ?

Étape 2

Etant donné que l'image générée est extrêmement volumineuse, utilisez un **multi-stage build** en combinaison avec une distribution GNU Linux minimale afin de produire une image finale dont la taille est la plus petite possible.

Veuillez nommer ce nouveau Dockerfile `Dockerfile2`.

- Donnez le contenu de `Dockerfile2`.

- Quelle est la taille de cette nouvelle image et quel facteur de taille avez-vous gagné par rapport à l'image précédente ?

Étape 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, on désire se baser sur une archive locale comme système de fichiers racine pour le conteneur (couche de base).

Décrivez exactement et exhaustivement toutes les étapes réalisées pour parvenir à cet objectif, ainsi que le contenu du Dockerfile implémenté. Ce dernier Dockerfile sera nommé `Dockerfile3`.