Optimisation des images Docker

Virtualisation - TD

Maxime Lambert

2023 / 2024

Modalités

- Travail en binôme en salle de machines.
- Matériel ou logiciels utilisés **:**
  - Accès au serveur Proxmox de l'université (https://proxmox-iutc3.unicaen.fr) depuis un navigateur web
  - Repartez de la VM que vous avez créée lors du TP n°3 (**r408-docker-*<<mes initiales>>***)
- Nous travaillerons de manière itérative afin d'optimiser une image docker contenant un programme écrit en C.
- Cette séance sera jalonnée de plusieurs points de synchronisation lors desquelles vous pourrez partager vos analyses et hypothèses avec le reste du groupe.

Avant-propos

- En CM, il a été indiqué que les conteneurs étaient une solution “légère” de virtualisation. Au cours de cette séance, nous nous limiterons à l’étude de l’espace disque occupé par les images Docker.

- Nous allons nous intéresser plus en détail aux couches qui composent une image. Vous allez apprendre les différences qui existent entre le contexte de construction et d’exécution d’une application dans un conteneur.

1. Mise en place de l'application témoin

- Créez un nouveau répertoire `td4` et à l'intérieur de celui-ci un répertoire que vous nommerez `1-temoin`. Le définir en tant que répertoire courant.

- En utilisant le langage C, écrivez un fichier `main.c` qui ne fait qu'afficher la chaîne de caractères "Hello World !" sur la sortie standard.

- Compilez votre code source à l'aide de `gcc`.
  - Exemple: `gcc main.c -o hello-world`

- Vérifiez la bonne exécution de votre programme.

- Affichez la taille de votre fichier binaire. Consignez la valeur, elle vous servira de référence pour la suite du TD.

2. Création de l'image hello:gcc

- Retournez dans le répertoire `td4`, créez un répertoire `2-hello-gcc`. Définissez-le en tant que votre répertoire courant et dupliquez dans ce répertoire le fichier source `main.c` précédemment créé.

- Écrivez un `Dockerfile` permettant de compiler votre code source et d'exécuter votre application.
  - Pour éviter d'utiliser le proxy, partez de l'image de base `prof:5000/gcc:latest`

- Construisez une image à partir de ce Dockerfile, taguez cette image `hello:gcc`
  - Exemple: `docker build --tag hello:gcc .`

- Instanciez un conteneur et vérifiez la bonne exécution de votre application
  - Exemple: `docker container run --rm hello:gcc`

- Comparez la taille du binaire compilé à l'étape 1 avec votre image. Quelles différences entre ces deux artéfacts ? Comment justifier cet écart de taille ?
  - Appuyez-vous sur les commandes: `docker image ls` et `docker history hello:gcc`

- Proposez une solution afin de diminuer la taille de l'image

3. Création de l'image hello:ubuntu

- Retournez dans le répertoire `td4`, créez un répertoire `3-hello-ubuntu`. Définissez-le en tant que votre répertoire courant et dupliquez dans ce répertoire le binaire `hello-world` créé à l'étape 1.

- Écrivez un `Dockerfile` permettant d'exécuter votre application.
  - Pour éviter d'utiliser le proxy, partez de l'image de base `prof:5000/ubuntu:latest`

- Construisez une image à partir de ce Dockerfile, taguez cette image `hello:ubuntu`
  - Exemple: `docker build --tag hello:ubuntu .`

- Instanciez un conteneur et vérifiez la bonne exécution de votre application
  - Exemple: `docker container run --rm hello:ubuntu`

- Comparez la taille de votre nouvelle image avec les résultats précédents. Qu'en pensez-vous ?
  - Appuyez-vous sur les commandes: `docker image ls` et `docker history hello:ubuntu`

- Pouvons-nous aller encore plus loin dans l'optimisation de la taille de l'image ?

Présentation de l'image virtuelle `scratch`

- `scratch` est un mot-clé réservé permettant de définir une image virtuelle de départ.

- L'instruction `FROM scratch`, indique au processus de construction d'images de Docker que la prochaine instruction sera la première couche du système de fichiers.

- C'est une image très minimaliste, car elle est totalement vide !
  - pas de shell
  - pas de commandes permettant le débogage: ls, ping…
  - **pas de librairie dynamique**

- Elle offre quelques avantages:
  - surface d'attaque réduite
  - maintien la taille d'un conteneur au strict minimum
  - temps de démarrage très court

- *N.B: c'est une image très utile dans le monde de l'IoT où l'espace disque est généralement limité.*

4. Création de l'image hello:scratch

- Retournez dans le répertoire `td4`, créez un répertoire `4-hello-scratch`. Définissez-le en tant que votre répertoire courant et dupliquez dans ce répertoire le binaire `hello-world` créé à l'étape 1.

- Écrivez un `Dockerfile` permettant d'exécuter votre application.
  - Partez de l'image de base `scratch`

- Construisez une image à partir de ce Dockerfile, taguez cette image `hello:scratch`
  - Exemple: `docker build --tag hello:scratch .`

- Instanciez un conteneur et vérifiez la bonne exécution de votre application
  - Exemple: `docker container run --rm hello:scratch`

- Vous obtenez une erreur "`no such file or directory`". Identifiez la cause de l'erreur, corrigez votre image et instanciez un conteneur afin de valider votre correction
  - Indice: la commande `ldd` devrait vous aider dans votre analyse

- Avez-vous des critiques sur cette manière de procéder (copie d'un binaire dans une image) ?

Présentation du Dockerfile multi-stage

```
FROM golang:1.16 AS build-ctx

WORKDIR /go/src/github.com/alexellis/href-counter/

RUN go get -d -v golang.org/x/net/html

COPY app.go ./

RUN CGO_ENABLED=0 go build -a -installsuffix cgo -o app .
```

Présentation du Dockerfile multi-stage

```
FROM golang:1.16 AS build-ctx
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go ./
RUN CGO_ENABLED=0 go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=build-ctx /go/src/github.com/alexellis/href-counter/app ./
CMD ["./app"]
```

Présentation du Dockerfile multi-stage

```
FROM golang:1.16 AS build-ctx
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go ./
RUN CGO_ENABLED=0 go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=build-ctx /go/src/github.com/alexellis/href-counter/app ./
CMD ["./app"]
```

5. Création de l'image hello:multi-scratch

- Retournez dans le répertoire `td4`, créez un répertoire `5-hello-multi`. Définissez-le comme votre répertoire courant et dupliquez dedans le fichier `main.c` créé à l'étape 1.
- Écrivez un `Dockerfile` multi-stage permettant de construire le binaire puis d'exécuter votre application.
- Construisez une image à partir de ce Dockerfile, taguez cette image `hello:multi-scratch`
- Instanciez un conteneur et vérifiez la bonne exécution de votre application.
- Comparez la taille de votre nouvelle image avec les résultats précédents. Qu'en pensez-vous ?
- Selon vous, quels sont les avantages et les inconvénients d'utiliser l'image virtuelle `scratch` ?

6. Quantité d'espace disque utilisée par le démon

- Pendant cette séance, nous avons noté différentes tailles pour nos images Docker. Toutefois ce n'est pas parce qu'une image fait 1 Go que 10 versions différentes de cette image occupent 10 Go.

- Identifiez la relation entre *`SIZE`*, *`UNIQUE SIZE`* et *`SHARED SIZE`*
  - Aidez-vous de la commande `docker system df --verbose`
  - Vous pouvez également créer de nouvelles images légèrement différentes de celles créées précédemment.

- Au vu de ces résultats, que penser de l'image `scratch` ? Est-ce la solution parfaite à utiliser systématiquement ?

7. Pour aller plus loin

- Pour les plus curieux et motivés, le code source java du service utilisé dans le TP1 est mis à disposition ici:
  - https://github.com/geomatiq/r408-td3/tree/master

- Les étudiants réalisant un Dockerfile multi-stage permettant la compilation du jar et son exécution, se verront attribuer des points bonus sur le module.

- Merci d'envoyer vos travaux sous la forme d'une archive (zip ou tar.gz) au plus tard le 25 février à l'adresse mail indiquée à la fin de ce document.

Référence

Ce TD a été inspiré par cette série d'articles :

Jérôme Petazzoni. *Chérie, j'ai rétréci Docker* [en ligne]. Publié le 19 février 2020. [Consulté le 22 janvier 2023]. Disponible à l'adresse : https://enix.io/fr/blog/cherie-j-ai-retreci-docker-part1/

Merci de votre attention

Maxime Lambert

maxime.lambert@unicaen.fr