Multi-stage builder

Dans ce Lab nous examinerons l'utilisation d'une méthode relativement nouvelle pour construire une image. Dans les sections précédentes, nous avons envisagé d'ajouter des fichiers binaires directement à nos images via un gestionnaire de packages, tel que l'APK d'Alpine Linux.

Mais si nous voulions compiler notre propre logiciel dans le cadre d'un build? Historiquement, nous aurions dû utiliser une image de conteneur contenant un environnement de construction complet, qui peut être très grand. Cela signifie que nous aurions probablement dû bricoler un script qui doit faire quelque chose comme le processus suivant:

1. Téléchargement de l'image du conteneur de l'environnement de build et démarrage d'un conteneur de "build"
2. Copie du code source dans le conteneur "build"
3. Compilation du code source sur le conteneur "build"
4. Copie du binaire compilé en dehors du conteneur "build"
5. Retrait du conteneur "build"
6. Utilisation d'un Dockerfile pré-écrit pour construire une image et y copier le binaire

Heureusement, la communauté Docker y a pensé, et la fonctionnalité pour y parvenir, appelée multi-stage build, a été introduite dans Docker 17.05.

Nous commencerons d'abord par voir comment construire un conteneur à partir de presque rien.

Création d'une image de conteneur à partir de zéro

Jusqu'à présent, nous avons utilisé des images préparées à partir du Docker Hub comme image de base. Il est possible d'éviter cela et de déployer votre propre image à partir de zéro, cela gardera la taille de l'image très petite, Docker a déjà effectué une partie du travail et a créé un fichier TAR vide sur Docker Hub nommé *scratch*; vous pouvez l'utiliser dans la section FROM de votre Dockerfile. Vous pouvez baser l'intégralité de votre build Docker sur cela, puis ajouter des pièces selon vos besoins.

Nous utiliserons Alpine Linux en version tar comme exemple d'application à déployer sur un conteneur scratch, il suffit de télécharger la version x86_64 de la section MINI ROOT FILESYSTEM à la page https://www.alpinelinux.org/Downloads/.

```
[root@localhost ~]# mkdir scratch
[root@localhost ~]# cd scratch/
[root@localhost scratch]# wget https://dl-
cdn.alpinelinux.org/alpine/v3.15/releases/x86_64/alpine-minirootfs-3.15.0-
x86_64.tar.gz
```

Créez un Dockerfile qui utilise scratch, puis ajouter le fichier tar.gz, comme dans l'exemple suivant:

```
[root@localhost builder]# vim Dockerfile
FROM scratch
ADD alpine-minirootfs-3.15.0-x86_64.tar.gz /
CMD ["/bin/sh"]
```

Maintenant que vous avez votre Dockerfile et votre système d'exploitation dans un fichier TAR, vous pouvez créer votre image comme suit:

```
[root@localhost scratch]# docker image build --tag local:fromscratch .
Sending build context to Docker daemon 2.726 MB
Step 1/3 : FROM scratch
 --->
Step 2/3 : ADD alpine-minirootfs-3.15.0-x86_64.tar.gz /
 ---> eaced0d8fe59
Removing intermediate container 90177c5eb7cc
Step 3/3 : CMD /bin/sh
 ---> Running in 4d2f33ce5a51
 ---> 3a09e2d8e370
Removing intermediate container 4d2f33ce5a51
Successfully built 3a09e2d8e370
```

Maintenant que notre image a été construite, nous pouvons la tester en exécutant cette commande:

```
[root@localhost scratch]# docker container run -it --name scratch-test
local:fromscratch /bin/sh

/ # cat /etc/*release
3.11.3
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.11.3
PRETTY_NAME="Alpine Linux v3.15"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
/ # exit
```

Multi-stage builds

Pour ce Lab il nous faut la dernière version de docker, donc créez une nouvelle VM dans laquelle vous allez installer Docker-ce.

```
$ sudo yum install -y yum-utils device-mapper-persistent-data lvm2
$ sudo yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo
$ sudo yum install docker-ce docker-ce-cli containerd.io
$ sudo systemctl enable docker --now
$ sudo docker run hello-world
```

Le Dockerfile suivant contient deux étapes de construction (build) différentes. La première, nommée *builder*, utilise l'image officielle du conteneur Go de Docker Hub. Ici, nous installons un prérequis, téléchargeons le code source directement depuis GitHub, puis le compilons dans un binaire statique:

```
[root@localhost builder]# mkdir builder && cd builder
```

```
[root@localhost builder]# vim app.go
package main

import (
  "fmt"
  "net/http"
)

const (
  port = ":80"
)

var calls = 0

func HelloWorld(w http.ResponseWriter, r *http.Request) {
  calls++
  fmt.Fprintf(w, "Hello, world! You have called me %d times.\n", calls)
}

func init() {
  fmt.Printf("Started server at http://localhost%v.\n", port)
  http.HandleFunc("/", HelloWorld)
  http.ListenAndServe(port, nil)
}

func main() {}
```

```
[root@localhost builder]# vim Dockerfile

FROM golang:1.9.2 AS builder
WORKDIR /go-http-hello-world/
RUN go get -d -v golang.org/x/net/html
COPY app.go    ./
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM scratch
#RUN apk --no-cache add ca-certificates
#WORKDIR /root/
COPY --from=builder /go-http-hello-world/app ./
CMD ["./app"]
```

Comme notre binaire statique a un serveur Web intégré, nous n'avons pas vraiment besoin de quelque chose d'autre du point de vue du système d'exploitation. Pour cette raison, nous sommes en mesure d'utiliser *scratch* comme image de base, ce qui signifie que tout ce que notre

image contiendra sera le binaire statique que nous avons copié à partir de l'image du *builder*, et ne contiendra rien du tout de l'environnement de build.

Pour construire l'image, il suffit d'exécuter la commande suivante:

```
[root@localhost builder]# docker image build --tag local:go-hello-world .
```

Comme vous pouvez le voir, entre les étapes 5 et 6, notre binaire a été compilé et le conteneur qui contient l'environnement de build est supprimé, nous laissant avec une image stockant notre binaire. L'étape 7 copie le binaire dans un nouveau conteneur qui a été lancé en utilisant *scratch*, nous laissant avec le contenu dont nous avons besoin.

Si vous deviez exécuter la commande suivante, vous auriez une idée de pourquoi c'est une bonne idée de ne pas expédier une application avec son environnement de construction intact:

```
[root@localhost builder]# docker images
REPOSITORY     TAG                 IMAGE ID       CREATED          SIZE
local          go-hello-world      d65129fe8b38   15 minutes ago   7.41MB
<none>         <none>              b2ba8b405ab4   15 minutes ago   862MB
golang         latest              25c4671a1478   3 weeks ago      809MB
```

Testez votre conteneur

```
[root@localhost builder]# docker container run -d -p 8000:80 --name go-
hello-world local:go-hello-world

[root@localhost builder]# curl http://localhost:8000
Hello, world! You have called me 1 times.

[root@localhost builder]# curl http://localhost:8000
Hello, world! You have called me 2 times.
```

Nettoyage

```
[root@localhost builder]# docker container stop go-hello-world

[root@localhost builder]# docker container rm go-hello-world
```