Multistage build

Dans cette mise en pratique, nous allons illustrer le multi stage build

Rappel

Comme nous l'avons vu, le Dockerfile contient une liste d'instructions qui permet de créer une image. La première instruction est FROM, elle définit l'image de base utilisée. Cette image de base contient souvent beaucoup d'éléments (binaires et librairies) dont l'application finale n'a pas besoin (compilateur, …). Ceci qui peut impacter de façon considérable la taille de l'image et également sa sécurité puisque cela peut considérablement augmenter sa surface d'attaque. C'est la qu'intervint le multistage build…

Un serveur http écrit en Go

Prenons l'exemple du programme suivant écrit en Go.

Dans un nouveau répertoire, créez le fichier *http.go* contenant le code suivant. Celui-ci définit un simple serveur http qui écoute sur le port 8080 et qui expose le endpoint /who en GET. A chaque requête, il renvoie le nom de la machine hôte sur laquelle il tourne.

```
package main

import (
        "io"
        "net/http"
        "os"
)

func handler(w http.ResponseWriter, req *http.Request) {
        host, err := os.Hostname()
        if err != nil {
          io.WriteString(w, "unknown")
        } else {
          io.WriteString(w, host)
        }
}

func main() {
        http.HandleFunc("/who", handler)
        http.ListenAndServe(":8080", nil)
}
```

Dockerfile *traditionel*

Afin de créer une image pour cette application, nous utilisons le Dockerfile suivant. A partir de l'image officielle golang, nous copions le fichier source http.go et lançons la compilation.

Dans le répertoire ou se trouve le fichier *http.go*, créez le fichier *Dockerfile* suivant:

```
FROM golang:1.9.0-alpine
WORKDIR /go/src/github.com/lucj/who
COPY http.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o http .
CMD ["./http"]
```

Dans ce *Dockerfile*, nous partons de l'image officielle *golang*. Le fichier source *http.go* est copié puis compilé.

Vous pouvez ensuite builder l'image et la nommer *who:1.0*, avec la commande suivante.

```
$ docker image build -t who:1.0 .
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM golang:1.9.0-alpine
 ---> ed119d8f7db5
Step 2/5 : WORKDIR /go/src/github.com/lucj/who
 ---> c12f059e10cd
Removing intermediate container 79f6f35b2c49
Step 3/5 : COPY http.go .
 ---> c14acda96ae7
Step 4/5 : RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o http .
 ---> Running in 86b538167121
 ---> b0319625972d
Removing intermediate container 86b538167121
Step 5/5 : CMD ./http
 ---> Running in 88ae1b957b46
 ---> 1f74e07fd4f7
Removing intermediate container 88ae1b957b46
Successfully built 1f74e07fd4f7
Successfully tagged who:1.0
```

Listez les images présentes. Quelle est la taille de l'image *who:1.0* ?

```
$ docker image ls who
```

```
REPOSITORY     TAG                 IMAGE ID            CREATED              SIZE
who            1.0                 1f74e07fd4f7        About a minute ago
276MB
```

L'image obtenue, bien que basée sur une distribution alpine a une taille relativement conséquente (276 MB) car elle contient l'ensemble de la toolchain du langage go. Or, une fois que le binaire a été compilé, nous n'avons plus besoin du compilateur dans l'image finale.

Dockerfile utilisant un build multi-stage

Le multi-stage build, introduit dans la version 17.05 de Docker permet, au sein d'un seul Dockerfile, d'effectuer le process de build en plusieurs étapes. Chacune des étapes peut réutiliser des artefacts (fichiers résultant de compilation, assets web, …) créés lors des étapes précédentes. Ce Dockerfile aura plusieurs instructions FROM mais seule la dernière sera utilisée pour la construction de l'image finale.

Si nous reprenons l'exemple du serveur http ci dessus, nous pouvons dans un premier temps compiler le code source en utilisant l'image *golang* contenant le compilateur. Une fois le binaire créé, nous pouvons utiliser une image de base vide, nommée scratch, et copier le binaire généré précédemment.

```
FROM golang:1.9.0-alpine as build
WORKDIR /go/src/github.com/lucj/who
COPY http.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o http .

FROM scratch
COPY --from=build /go/src/github.com/lucj/who/http .
CMD ["./http"]
```

> L'exemple que nous avons utilisé ici se base sur une application écrite en Go. ce langage a la particularité de pouvoir être compilé en un binaire static, c'est à dire ne nécessitant pas d'être "linké" à des librairies externes. C'est la raison pour laquelle nous pouvons partir de l'image scratch. Pour d'autres langages, l'image de base utilisée lors de la dernière étape du build pourra être différente (alpine, …)

Buildez l'image dans sa version 2 avec la commande suivante.

```
$ docker image build -t who:2.0 .
Sending build context to Docker daemon  4.096kB
Step 1/7 : FROM golang:1.9.0-alpine as build
 ---> ed119d8f7db5
Step 2/7 : WORKDIR /go/src/github.com/lucj/who
 ---> 5bbf49c30ad7
Removing intermediate container 9b3ac039f5e3
Step 3/7 : COPY http.go .
 ---> 28b995cce8c9
Step 4/7 : RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o http .
 ---> Running in ec39fa26ddb2
 ---> 94f190c3d0d9
Removing intermediate container ec39fa26ddb2
Step 5/7 : FROM scratch
 --->
Step 6/7 : COPY --from=build /go/src/github.com/lucj/who/http .
 ---> 370634625fc4
Step 7/7 : CMD ./http
 ---> Running in e2335e16038c
 ---> 21f530664efb
Removing intermediate container e2335e16038c
Successfully built 21f530664efb
Successfully tagged who:2.0
```

Listez les images et observez la différence de taille…

```
$ docker image ls who
REPOSITORY     TAG                 IMAGE ID            CREATED             SIZE
who            2.0                 21f530664efb        7 minutes ago
6.09MB
who            1.0                 1f74e07fd4f7        14 minutes ago      276MB
```

Lancez un un container basé sur l'image *who:2.0*

```
$ docker container run -p 8080:8080 who:2.0 
```

A l'aide de la commande curl, envoyez une requête GET sur le endpoint exposé. Vous devriez avoir, en retour, l'identifiant du container qui a traité la requète.

```
$ curl localhost:8080/who
7562306c6c5e
```

Pour cette simple application, le multistage build a permit de supprimer 270M de binaires et librairies dont la présence est inutile dans l'image finale. L'exemple d'une application écrite en go est extrème, mais le multistage build fait partie des bonnes pratiques à adopter quel que soit le language de développement de l'application.