OPTIMIZAREA IMAGINILOR DOCKER

Florinel Daniel BANĂ*

Universitatea Politehnica Timișoara, Facultatea de Inginerie Hunedoara,
Departamentul de Inginerie Electrică și Informatică Industrială,
grupa Tehnici informatice în ingineria electrică, an I, Hunedoara, Romania

*Autorul corespondent: Bană Florinel Daniel, f.danielbana@gmail.com

Rezumat. *Una din problemele dezvoltarii aplicatiilor este problema dependintelor. Unele aplicatii depind de alte programe, frameworkuri, etc. O aplicatie care functioneaza pe computerul nostru poate sa nu functioneze pe alt computer din cauza dependintelor. Docker rezolva aceasta problema, "impachetand" aplicatiile impreuna cu dependintele lor in niste medii virtuale numite "imagini docker" si rulandu-le ca o unitate intr-un container. Pentru a rula eficient aplicatiile, vom optimiza imaginile dupa procedeele pe care le voi prezenta in urmatoarele randuri.*

Cuvinte cheie: *docker, dockerfile, imagine, optimizare, multi-stage, build*

Introducere

Docker este un software care ne permite să creem, să lansam și să rulam aplicatii prin intermediul unor *containere*. Containerele ne permit dezvoltarea unei aplicatii impreuna cu toate partile acesteia, precum librari și alte programe de care depinde rularea acestei aplicatii, în felul acesta, vom putea rula alpicatia pe orice computer, fara grija dependintelor.

Un container este o unitate software standard, în care putem rula aplicațiile impreuna cu dependitele acestora. Containerele de docker sunt create pornind de la o *imagine*, care reprezinta o unitate in care sunt definite toate instructiunile, comenzile și dependintele de care aplicația pe care o dezvoltam are nevoie. Imaginile sunt construite prin intermediul unui fisier numit *Dockerfile* care contine un set de instructiuni dupa care Docker va construi imaginea.

Aplicatia de test

In urmatorul exeplu, avem o aplicatie server de tip *REST* (Representational State Transfer). Aceasta aplicatie are un *endpoint* numit "test". Accesand acest enpoint prin metoda http *GET* primim ca raspuns, un obiect de tip *JSON* (JavaScript Object Notation) care are un singur camp numit "message" si care are valoarea "App is running", aceasta insemnand ca aplicatia ruleaza.

Crearea fisierului Dockerfile si construirea imaginii

Pentru a rula aplicatia pe computerul personal, avem nevoie de cateva dependinte, cum ar fi NodeJS, care este un runtime environment pentru JavaScirpt si mai avem nevoie de **NPM** (Node Package Manager) pentru a instala pachetele de care aplicatia depinde. Pentru a rula aplicatia pe orice computer, fara a mai instala dependintele, putem sa rulam aplicatia intr-un container de Docker.

Pentru a creea imaginea aplicatie, avem nevoie de un fisier numit Dockerfile. Acesta va avea urmatoarea structura:

```
FROM ubuntu:latest
ENV PORT=8080
WORKDIR /home/Test
COPY . .

RUN apt-get update
RUN apt-get -y install npm
RUN npm install
```

```
RUN npm i -g typescript
RUN tsc main.ts
RUN useradd -M testuser
RUN chown -R testuser ./
RUN chmod -R 754 ./

EXPOSE 8080

USER testuser
CMD ["node", "main"]
```

Construirea imaginii

Dupa ce rulam comanda *docker build –t test_image .* , obtinem imaginea dorita. Docker build ne construieste imaginea, -t test_image o denumeste test_image, iar . ne indica folderul in care este fisierul Dockerfile, in cazul nostru, . inseamna folderul curent.

Utilizand comanda *docker images,* putem vedea toate imaginile de Docker de pe computer. Aici putem vedea si imaginea creeata de noi, numita test_image.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
test_image	latest	fa4ac1593e2f	4 minutes ago	658MB

Figura 2. Versinea initiala a imaginii

Optimizarea imaginii

Observam ca imaginea are o dimnesiune foarte mare si va fi nevoie sa o optimizam. De asemenea, timpul de creeare al imaginii este si el foarte mare.

Pentru optimizarea dimensiunii imaginii, puteam utiliza o alta imagine de baza, numita **Alpine**. Alpine este o imagine bazata pe sistemul de operare cu acelasi nume, aceasta avand o dimensiune redusa. Schimbam *FROM ubuntu:latest* cu *FROM alpine:latest*

Dupa schimbarea imaginii de baza, obtinem o imagine de 278 MB, aproximativ de 3 ori mai mica.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
test_image	latest	11308aad9aa8	22 seconds ago	278MB

Figura 3. Dimensiunea dupa schimbarea imaginii de baza

Pentru a optimiza timpul de creeare al imagini, putem inlantui toate instructiunile din RUN, intru-una singura, astfel:

```
RUN apk add --no-cache --update npm && \
    npm install && \
    npm i -g typescript && \
    tsc main.ts && \
    adduser testuser --no-create-home --disabled-password && \
    chown -R testuser ./ && \
    chmod -R 754 ./
```

Fiecare instructiune va creea un nou **layer**, adica un ultim stadiu de la care porneste urmatoarea instructiune. Inlantuind instructiunile RUN intr-una singura, reducem signifiant numarul de layere.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
test_image	latest	7af560929171	About a minute ago	169MB

Figura 4. Dimensiunea imaginii dupa inlantuirea instructiunior RUN

Prin acest procedeu, nu am reusit doar sa reducem numarul de layere si implicit timpul de construire al imaginii, dar si dimensiunea acesteia, acum avand 169MB.

Multi-stage build

Intr-un final, pentru a optimiza la maxim imaginea acestei aplicatii, vom folosi un procedeu numit **multi-stage build**. Adica vom porni de la o imagine de baza, vom instala toate dependintele si de asemenea, vom instala si proiectul, iar apoi, pornint din nou de la o imagine de baza, vom copia proiectul, deja pregatit din stadiul precedent.
Intr-un final, fisierul Dockerfile va arata astfelt:

```
ARG HOME="/home/test"

FROM alpine:latest as builder
ARG HOME
WORKDIR $HOME
COPY . .

RUN apk add --no-cache --update npm && \
    npm install && \
    npm i -g typescript && \
    tsc main.ts && \
    rm -rf ./node_modules && \
    find . -name "*.ts" -type f -delete && \
    npm install --production

FROM alpine:latest
ARG HOME
WORKDIR $HOME
COPY --from=builder $HOME $HOME

RUN apk add --no-cache --update nodejs && \
    adduser testuser --no-create-home --disabled-password && \
    chown -R testuser ./ && \
    chmod -R 754 ./

ENV PORT=8080
EXPOSE 8080
USER testuser

CMD ["node", "main"]
```

Dupa ce rulam comanda docker build –t test_image . , iar apoi docker images, observam ca dimensiunea imaginii a fost redusa drastic, la 61.6 MB

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test_image          latest              5653465d6bdb        9 seconds ago       61.6MB
```

Figura 5. Versiunea finala a imaginii

Iar in imaginea urmatoare, avem o comparatie cu stadiile precedente ale imaginii.

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test_image          latest              5653465d6bdb        9 seconds ago       61.6MB
<none>              <none>              9afd7cd961c9        6 minutes ago       169MB
<none>              <none>              7af560929171        10 minutes ago      169MB
<none>              <none>              11308aad9aa8        18 minutes ago      278MB
<none>              <none>              fa4ac1593e2f        37 minutes ago      658MB
```

Figura 6. Comparatie intre versiunile imaginii

Pentru a testa aplicatia, vom rulaentru a testa aplicatia, vom rula imaginea prin comanda *docker run –it –p 8080:8080 test_image.* Aplicatia ruleaza pe portul TCP 8080.

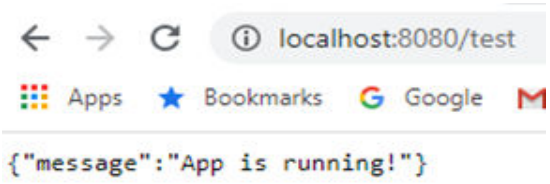

Figura 6. Raspunsul aplicatiei in browser

Concluzii

Utilizand procedeele prezentate in aceasta lucrare, putem creea imagini docker optimizate, care pot avea dimeniusi reduse, de aprox. 10 ori mai mici decat imaginile prost-optimizate, iar timpul lor de creare va fi scurt.

Mulțumiri

Ținem să mulțumim pentru ajutorul acordat în realizarea acestui articol dnei **Raluca ROB**, **Șef Lucrări Dr. Ing.**.

Referinte:

1. Docker website: https://www.docker.com/
2. Docker documentation: https://docs.docker.com/
3. Multi-stage build: https://docs.docker.com/develop/develop-images/multistage-build/
4. Alpine – DockerHub: https://hub.docker.com/_/alpine