Tworzenie i uruchamianie prostych kontenerów w środowisku docker-compose

Grzegorz Koperwas

4 stycznia 2021

Streszczenie

W tej pracy zostaną pokazane absolutne podstawy pisania plików `dockerfile` oraz uruchamiania tych kontenerów w środowisku `docker-compose` na serwerze z systemem *GNU/Linux*. Autor zakłada iż czytelnik posiada zainstalowanego `docker`'a oraz `docker-compose` w miarę aktualnych wersjach.

1. Jak działają obrazy aplikacji?

By uruchomić daną aplikację w środowisku `docker`'a musimy utworzyć jej obraz. `Docker` tworzy swoje obrazy według instrukcji w pliku zwanym `dockerfile`, możemy o nich myśleć jako instrukcjach które znajdują się na wielu repozytoriach na *Github'ie* mówiących jakie biblioteki są potrzebne do kompilacji oraz jak kompilować i instalować dany program, czasami nawet zawierają one informacje dla poszczególnych dystrybucji.

Na potrzeby tej pracy będziemy chcieli stworzyć obraz z kompilatorem sieciowym *distcc*[1].

Pierwszym krokiem do stworzenia jakiegoś obrazu aplikacji musimy wybrać obraz *bazowy*. Docker posiada swoją platformę *Docker Hub* na której możemy szukać właśnie interesujących nas obrazów, które mają formę od gotowych do pracy dystrybucji linuxa (alpine, arch czy debian) do gotowych obrazów popularnych aplikacji wraz z sterownikami graficznymi[2].

My będziemy wykorzystywali obraz `archlinux:latest`, nie dlatego że jest to najlepszy wybór, ale dlatego iż możemy łatwo go skonfigurować. Warto tu jeszcze powiedzieć o *tagach* (ta część po „:"), w naszym przypadku używamy taga *latest*, czyli najnowszej wersji obrazu, jednak w wypadku innych obrazów należy sumiennie przeczytać dokumentację w celu wyboru właściwego.

Co napisać w pliku `dockerfile`?

Całość zawartości pliku dockerfile jest przedstawiona w załączniku 1, omówmy zatem co się dzieje w każdej linijce jego zawartości.

[1] https://distcc.github.io/

[2] Na przykład `jrottenberg/ffmpeg:4.1-nvidia` zawiera debian'a z programem ffmpeg skonfigurowanym tak by wykorzystywał akceleracje wideo kart firmy *nvidia*.

1. Polecnie `FROM`

Polecenie `FROM` mówi *docker*'owi z jakiego obrazu ma on stwożyć nasz obraz, czyli definiujemy obraz bazowy. Widzimy iż przekazujemy mu nazwę obrazu `archlinux:latest`, jeśli nie mamy jej jeszcze na swoim komputerze to *docker* pobierze ją z *docker hub*'a

2. Polecenie `RUN`

Polecenie `RUN` pozwala nam uruchomić dane polecenie w kontenerze (*docker* uruchomi obraz, wykona polecenie, zapisze obraz). Często używamy go do instalowania jakiś zależności lub jak tutaj, aplikacji którą chcemy „skonteneryzować".

Polecenie `pacman -Syu --noconfirm distcc`[…]`cmake` aktualizuje wszystkie pakiety oraz instaluje pakiety distcc, make, git, gcc i cmake bez pytania się użytkownika o potwierdzenie (`--noconfirm`)

> **Ważne!**
>
> 1. Docker uruchamia wszystkie polecenia w kontenerach jako `root`
> 2. Docker by zwiększyć prędkość tworzenia obrazów zapisuje je do cache po każdym kroku, zatem warto je łączyć (patrz załącznik 2)

3. Polecenia `EXPOSE`

Polecenie `EXPOSE` pozwala nam otworzyć dane porty w kontenerze dla danych protokołów. Na przykład `EXPOSE 3632/tcp` otwiera port *distcc* dla protokołu tcp, a `EXPOSE 3632/udp` dla protokołu udp.

4. Polecenie `ENTRYPOINT`

Poleceniem `ENTRYPOINT` definiujemy jaki program lub skrypt powinien zostać uruchomiony w kontenerze kiedy zaczyna on swoją pracę. Warto zaznaczyć że docker zakłada iż kontener skończył swoją pracę jeśli polecenie skończy się wykonywać, dlatego dodajemy opcję `--no-detach` do polecenia uruchamiającego daemona *distccd*.

Opcja `--allow-private` udostępnia kompilację dla wszystkich klientów w sieci lokalnej.

5. Polecenie `COPY`

Warto również wspomnieć o poleceniu `COPY`, które nie jest wykorzystywane w naszym `dockerfile`'u ale jest przydatne jeśli chcemy przekopiować jakiś nasz lokalny plik do obrazu.

Przykładowo w drugiej linijce załącznika 3 kopiujemy folder z aplikacją `./app`[3] do folderu `/app` w obrazie.

[3] Ścieżka jest podana relatywnie do lokalizacji pliku `dockerfile`

Istnieją również inne polecenia których możemy użyć w `dockerfile`, są one omówione w dokumentacji[incb].

Budowanie obrazu z pliku `dockerfile`

Obraz budujemy za pomocą polecenia:

```
$ docker build -t $nazwaObrazu .
```

Gdzie zamiast `$nazwaObrazu` wpisujemy nazwę dla naszego obrazu.

2. Uruchamianie obrazów w środowisku `docker-compose`

`Docker-compose` jest alternatywą dla zwykłego uruchamiania kontenerów poprzez polecenia `docker run` gdzie bardzo szybko mogą nam powstać takie cuda:

```
1 # docker run \
2     --gpus all \
3     --network "host" \
4     --device /dev/ttyUSB0:/dev/tty2 \
5     --volume /dockerStuff/rtsp/data:/rtsp/data \
6     rtsp_over_serial:latest
```

Zapamiętanie takich monstrów[4] na rzecz późniejszej edycji możemy jedynie powierzyć historii konsoli. Dlatego możemy używać środowiska `docker-compose` w celu uruchamiania kontenerów za pomocą ustawień w pliku `docker-compose.yml`. Plik ten używa formatu *YAML*.

Plik dla naszego obrazu znajduje się w załączniku 4, omówmy sobie jego zawartość.

Tworzenie plilku `docker-compose.yml`

W pliku `docker-compose.yml` pod kluczem „services” umieszczamy klucze z nazwami jakie chcemy nadać naszym kontenerom. Nasz kontener z kompilatorem *distcc* nazywamy oczywiście distcc.

1. Klucz `image`

W tym kluczu umieszczamy nazwę naszego obrazu, w naszym wypadku jest to „distcc” wraz z tagiem „latest”, który wybiera najnowszą wersję.

2. Klucz `restart`

Klucz `restart` pozwala nam przekazać środowisku `docker-compose` co ma się dziać po zakończeniu pracy kontenera lub przy restartowaniu serwera. Dostępnych mamy parę opcji:

[4] Przedstawiony przykład i tak jest rozmiaru „średniego”

- *always* - kontener jest zawsze uruchamiany ponownie. Dlatego wybieramy go dla naszego kontenera.
- *on-failure* - nie restartuje kontenera jeśli zwróci on 0.
- *no* - nie restartuje kontenera nigdy - domyślny.

3. Klucz `ports`

Pod tym kluczem możemy udostępniać porty z kontenera, w formie `<port hosta>:<port kontenera>`.

Istnieje o wiele więcej kluczy których możemy używać, ich opis możemy łatwo znaleźć w dokumentacji `docker-compose`[inca].

Uruchamianie kontenera

By uruchomić kontener wystarczy polecenie:

```
$ docker-compose up -d
```

Gdzie opcja `-d` odłącza proces od konsoli.

Bonusowy one-liner

By wyświetlać logi na bieżąco polecam polecenie:

```
$ watch --color "docker-compose logs | tail -n 20"
```

Literatura

[HBB17] K. Hightower, B. Burns, and J. Beda. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, 2017.

[inca] Docker inc. Dokumentacja docker-compse, wersja druga. https://docs.docker.com/compose/compose-file/compose-file-v2/. Dostęp: 2021-01-02.

[incb] Docker inc. Dokumentacja dockerfile. https://docs.docker.com/engine/reference/builder/. Dostęp: 2021-01-02.

```
1 FROM archlinux:latest
2 RUN pacman -Syu --noconfirm distcc make git gcc cmake
3 EXPOSE 3632/tcp
4 EXPOSE 3632/udp
5 ENTRYPOINT distccd --daemon --no-detach --verbose --allow-private
```

Załącznik 1: Dockerfile tworzący nasz obraz z programem *distcc*

Zamiast:

```
2 RUN command1
3 RUN command2
```

Lepiej:

```
2 RUN command1 && \
3     command2
```

Załącznik 2: Lepiej wiele poleceń dać do jednego *RUN'a*

```
1 FROM tiangolo/uwsgi-nginx-flask:python3.8
2 COPY ./app /app
3 RUN python3 -m pip install -r /app/requirements.txt
```

Załącznik 3: Dockerfile tworzący obraz mojej webowej aplikacji *PortLister* w `flask`'u

```
1 version: '3.3'
2
3 services:
4         distcc:
5                 image: distcc:latest
6                 restart: always
7                 ports:
8                         - "3632:3632"
```

Załącznik 4: Plik `docker-compose.yml` dla naszego obrazu z *distcc*