

LABORATORIUM PROGRAMOWANIA W CHMURACH OBLICZENIOWYCH

LABORATORIUM NR 5.

A. Pliki Dockerfile

Dockerfile jest plikiem tekstowym, który zawiera ciąg instrukcji utworzonych przez użytkownika/programistę. W sytuacji gdy wywołane jest polecenie *build* to zapisy w tym pliku definiują sposób tworzenia obrazu w środowisku Docker. W instrukcji do ćwiczenia nr 2 (część C) zostały przedstawione podstawowe instrukcje wykorzystywane w plikach Dockerfile. Pełna dokumentacja składni Dockerfile znajduje się pod adresem: <https://docs.docker.com/engine/reference/builder/>.

Poniżej wybrane elementy typowego pliku Dockerfile są przedstawione na przykładzie tworzenia obrazu serwera nginx.

```
FROM alpine:latest
LABEL maintainer="PwCh0 <email@domena>"
LABEL description="Przykładowy Dockerfile dla serwera NGINX."
RUN apk add --update nginx && \
    rm -rf /var/cache/apk/* && \
    mkdir -p /tmp/nginx/
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/default.conf /etc/nginx/conf.d/default.conf
ADD files/html.tar.gz /usr/share/nginx/
EXPOSE 80/tcp
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

Każdy Dockerfile musi zaczynać się od instrukcji FROM. Jedynym wyjątkiem jest instrukcja ARG, która definiowana może być wcześniej. Przykład wykorzystania obu instrukcji:

```
ARG APP_VERSION=latest
FROM base:${CODE_VERSION}
CMD /app/run-app
```

ARG

Instrukcja ARG zdefiniowana przed FROM, nie jest w zasięgu polecenia *build* i tym samym nie może być użyta w połączenie z jakąkolwiek instrukcją po FROM. Jeżeli jednak istnieje potrzeba wykorzystania wartości ARG zadeklarowanej przed FROM to należy użyć ARG raz jeszcze ale bez nadawania wartości. Ilustruje to przykład poniżej:

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

Środowisko Docker posiada również predefiniowane wartości instrukcji ARG, które można wykorzystać bez konieczności dodawania instrukcji ARG w pliku Dockerfile. Są to odpowiednio:

- HTTP_PROXY
- http_proxy
- HTTPS_PROXY
- https_proxy
- FTP_PROXY
- ftp_proxy
- NO_PROXY
- no_proxy

Aby je wykorzystać należy użyć flagi: `--build-arg <varname>=<value>` w poleceniu *build*.

UWAGA 1: Ze względów głównie bezpieczeństwa, predefiniowane wartości zmiennych nie są dostępne w poleceniu *docker history*.

UWAGA 2: Z tych samych względów nie powinno się umieszczać w ARG danych wrażliwych, np. kluczy i haseł dostępu, haseł użytkownika itp. ponieważ będą one widoczne w *docker history*.

LABEL

Instrukcja LABEL jest wykorzystywana do umieszczania dodatkowych informacji o obrazie. Należy jednak pamiętać, że zbyt duża liczba instrukcji tego typu powoduje niepotrzebne zwiększenie obrazu. Zaleca się podawanie jedynie informacji o twórcy obrazu oraz o zawartości obrazu.

RUN

Instrukcja RUN pozwala na instalację oprogramowania czy też uruchamianie skryptów. RUN może być wykorzystane według dwóch poniższych schematów:

- RUN <polecenie> (tzw. schemat *shell*, polecenie jest wykonywane w powłoce tj. domyślnie w systemie Linux `/bin/sh -c` oraz `cmd /S /C` w systemie Windows)
- RUN ["program wykonywalny", "param1", "param2"] (tzw. schemat *exec*)

UWAGA: Domyślną powłokę można zmienić za pomocą instrukcji SHELL. Natomiast, aby w pojedynczym przypadku wykorzystać inną niż domyślną powłokę należy użyć schematu *exec*, np. jak poniżej:

```
RUN ["/bin/bash", "-c", "echo hello"]
```

W przykładzie dotyczącym NGINX wykorzystano łańcuch poleceń. Jest to zalecana metoda (w stosunku do podawania kolejno poleceń z instrukcją RUN. Wykorzystanie łańcucha zmniejsza ilość warstw a tym samym czyni przyszły kontener wydajniejszym.

```
RUN apk add --update nginx && \
rm -rf /var/cache/apk/* && \
mkdir -p /tmp/nginx/
```

UWAGA: Proszę zapoznać się z dokumentacją dotyczącą wykorzystania cache w procesie budowy obrazów Docker, szczególnie w odniesieniu do instrukcji RUN.

COPY oraz ADD

Instrukcja COPY może być użyta według poniższych, dwóch schematów:

- COPY [--chown=<user>:<group>] <src>... <dest>
- COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]

UWAGA1 (odnosi się tak do instrukcji COPY jak i ADD): Wykorzystanie opcji chown oraz group jest możliwe tylko w przypadku kontenerów opartych o systemy Linux.

UWAGA2 (odnosi się tak do instrukcji COPY jak i ADD): Wszystkie nowe pliki i/lub katalogi są tworzone z UID oraz GID o wartości 0 chyba, że użyta została flaga `--chown`

UWAGA3: (odnosi się tak do instrukcji COPY jak i ADD): Sam katalog nie jest kopiowany, kopiowana jest tylko jego zawartość.

COPY wykorzystywane jest głównie w przypadku umieszczania/zastępowania plików np. konfiguracyjnych, które znajdują się w zakresie kontekstu danego projektu (pojęcie kontekstu jest bardzo ważne i zostanie dokładniej wspomniane w dalszej części instrukcji). Instrukcja ADD działa podobnie jak COPY i też posiada dwa schematy użycia;

- ADD [--chown=<user>:<group>] <src>... <dest>
- ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]

W omawianym przykładzie, instrukcję ADD użyto w następujący sposób:

```
ADD files/html.tar.gz /usr/share/nginx/
```

Plik `html.tar.gz` zostanie skopiowany do wskazanego <dest> (jeśli miejsce docelowe nie istnieje to zostanie utworzone). Dodatkowo jednak, poza pobraniem w/w pliku, zostanie on rozpakowany i skopiowany do katalogu docelowego.

Instrukcję ADD można też użyć do dodanie zawartości ze zdalnych źródeł, np. serwerów HTTP. np.:

```
ADD http://www.myremotesource.com/files/html.tar.gz /usr/share/nginx/
```

Należy jednak pamiętać, że archiwa pochodzące ze źródeł zdalnych są traktowane jako pliki i nie są dekompresowane. W takich przypadkach należy użyć instrukcji RUN by rozpakować je w docelowym miejscu.

UWAGA: Pliki skompresowane są rozpoznawane nie po ich nazwie lub rozszerzeniu a po ich zawartości.

UWAGA: Jeżeli plik jest pod adresem URL, który jest zabezpieczony mechanizmami autoryzacji to instrukcja ADD nie może być użyta. W takim wypadku należy użyć instrukcji RUN. np. RUN wget ... lub RUN curl

EXPOSE

Instrukcja EXPOSE informuje jaki port oraz jaki protokół ma być wystawiony w celu zapewnienia komunikacji usługi/aplikacji ze światem zewnętrznym. Instrukcja ta nie jest tożsama z mapowaniem portów, które należy wykonać oddzielnie jako opcję polecenia RUN.

ENTRYPOINT oraz CMD

Obie te instrukcje są bardzo często wykorzystywane łącznie. Definiują one jakie polecenie ma być wykonane podczas uruchamiania kontenera. Zasady współpracy można podsumować w postaci poniższych punktów:

- Dockerfile powinien zawierać przynajmniej jedną instrukcję CMD lub ENTRYPOINT.
- Instrukcja ENTRYPOINT powinna być zdefiniowana w przypadku gdy przyszły kontener ma być bezpośrednio wykonywany jako element wykonywalny.
- Instrukcja CMD definiuje domyślny argument dla instrukcji ENTRYPOINT lub (w przypadku braku ENTRYPOINT) definiuje wykonania ad-hoc polecenia w kontenerze.
- Zawartość instrukcji CMD może być nadpisana przez alternatywny argument, podany podczas uruchamiania kontenera.

W przykładzie przytoczonym na początku instrukcji, obie instrukcje są użyte w następujący sposób:

```
ENTRYPOINT ["nginx"]  
CMD ["-g", "daemon off;"]
```

Oznacza to, że gdy kiedykolwiek uruchomiony zostanie kontener, uruchomiony zostanie obiekt binarny nginx a ze względu na zapisy w instrukcji, uruchomienie to wykorzysta dwa argumenty. Innymi słowami, uruchamiając kontener, uruchomione zostanie polecenie:

```
$ nginx -g daemon off;
```

Jeśli natomiast kontener zostanie uruchomiony za pomocą poniższego polecenia:

```
$ docker container run --name nginxv dockerfile-example -v
```

to kontener wywoła polecenie

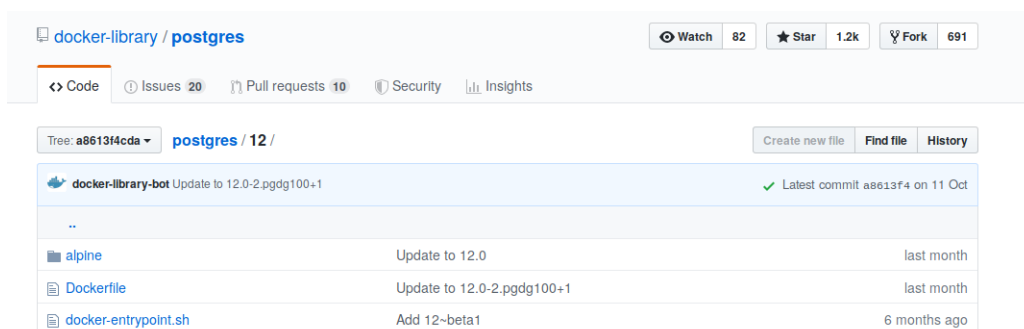
```
$ nginx -v
```

Podobnie jak poprzednio omawiane instrukcje, instrukcja ENTRYPOINT może być używana według dwóch schematów:

- ENTRYPOINT ["program wykonywalny", "param1", "param2"] (tzw. schemat *exec*, jest to schemat preferowany)
- ENTRYPOINT command param1 param2 (tzw. schemat *shell*)

Bardzo często szczegółową konfigurację kontenera przenosi się do oddzielnego skryptu, który z kolei jest wywoływany przed odpowiednio zdefiniowaną instrukcją ENTRYPOINT.

Jednym z najlepszych przykładów (i jednym z najbardziej złożonych) takiego skonfigurowania zawartości obrazu jest obraz bazy postgres, dostępny na DockerHub.



Proszę przyrzeć się i przeanalizować to rozwiązanie.

UWAGA: Jeśli w opisie danego obrazu wykorzystywany jest skrypt startowy do zdefiniowania konfiguracji określonej aplikacji to należy zapewnić by ta aplikacja (plik wykonywalny) otrzymywał sygnały Unix. Zapewnia to użycie poleceń *exec* lub *gosu*.

ENV

Instrukcja ENV ustawia wartości zmiennych środowiskowych w ramach obrazu (tak w trakcie jego budowania jak i uruchamiania wynikowego kontenera). Należy pamiętać, że wartości te można nadpisać podczas uruchamiania obrazu (flaga *-e*).

Wykorzystanie instrukcji ENV możliwe jest według dwóch schematów:

```
ENV <key> <value>
ENV <key>=<value> ...
```

W pierwszym przypadku definiowana jest pojedyncza wartość zmiennej środowiskowej. Cały ciąg po pierwszej spacji (pustym znaku) jest traktowany jako wartość zmiennej, łącznie ze znakami pustymi. W drugim schemacie możliwe jest zdeklarowanie wielu wartości jednocześnie. Tym jednak razem konieczne jest wykorzystanie cudzysłowu i/lub backslash-y do oznaczenia pustych znaków w wartości zmiennej. Przykładowe wykorzystanie instrukcji ENV, odpowiednio w pierwszym i drugim schemacie, są zilustrowane poniżej:

```
ENV pw Pierwsza Wartosc
ENV dw Druga Wartosc
```

```
ENV pw="Pierwsza Wartosc" dw=Drugą\ Wartosc \
```

UWAGA: Możliwe jest zdefiniowanie wartości zmiennej środowiska wyłącznie w obrębie jednego polecenia. W takim wypadku należy wykorzystać składnię `RUN <key>=<value> <polecenie>`.

Poza typowym wykorzystaniem instrukcji ENV, dosyć często służy ona do szybkiej zmiany wersji wykorzystywanych komponentów wchodzących w skład danego obrazu. Przykładowo:

```
FROM alpine:latest
LABEL maintainer="PwCh0 <email@domena>"
LABEL description="Przykładowy Dockerfile dla serwera Apache & PHP."
ENV PHPVERSION=7
RUN apk add --update apache2 php${PHPVERSION}-apache2 php${PHPVERSION} && \
rm -rf /var/cache/apk/* && \
mkdir /run/apache2/ && \
rm -rf /var/www/localhost/htdocs/index.html && \
echo "<?php phpinfo(); ?>" > /var/www/localhost/htdocs/index.php && \
chmod 755 /var/www/localhost/htdocs/index.php
EXPOSE 80/tcp
ENTRYPOINT ["httpd"]
CMD ["-D", "FOREGROUND"]
```

Zmieniając wersję PHP (np. z wersji 7 na wersję 5) można szybko (bez konieczności) przeglądania całego pliku Dockerfile, uzyskać nową wersję obrazu serwera Apache. W przypadku wykorzystywania GitHub-a, proste wykonanie *commit* może uruchomić budowę nowego obrazu.

Pojęcie kontekstu

Polecenie służące do budowania obrazów na podstawie plików Dockerfile ma składnie:

```
$ docker image build --file <path_to_Dockerfile> --tag <REPOSITORY>:<TAG> .
```

Wykorzystując poprzedni przykład, można wygenerować obraz na dwa sposoby. W pierwszym, wykorzystuje się pąnię składnię polecenia. <REPOSITORY> oznacza typowo nazwę prywatnego repozytorium lub nazwę użytkownika na DockerHub. Gdy chcemy zbudować obraz lokalnie należy użyć słowa *local*. Z kolei <TAG> jest unikatową wartością (np. numerem wersji lub opisem) przypisaną do nowego obrazu.

W większości przypadków pomija się opcje *-file*. Oznacza to, że Dockerfile oraz wszystkie niezbędne komponenty projektu umieszczane są w tym samym katalogu. Jest to dobra praktyka, która ułatwia budowanie obrazów np. poprzez uproszczenie instrukcji ADD czy też COPY. Wobec tego przykładowe polecenie według tego, drugiego sposobu wygląda następująco:

```
$ docker image build --tag local:dockerfile-example .
```

Proszę koniecznie zwrócić uwagę na kropkę na koncu polecenia. Opcja *-file* jak i wspomniana kropka definiuje to co w środowisku Docker jest określane mianem kontekstu. Polecenie *build* buduje obraz na podstawie wpisów w pliku Dockerfile oraz właśnie kontekstu. Kontekst jest rozumiany jako zestaw plików w określonej lokalizacji (PATH lub URL gdzie URL to typowo repozytorium Git). Kontekst jest przetwarzany rekursywnie co oznacza, że jakiegokolwiek podkatalogi czy też submoduły repozytorium są brane pod uwagę.

UWAGA: Wobec rekursywności obsługi kontekstu nie wolno wykorzystywać katalogu / jako ścieżki definiującej kontekst ponieważ spowoduje to przeniesienie całej zawartości systemu plików do daemona Docker.

Unikanie kopiowania niepotrzebnych elementów w procesie budowania obrazów może być nadzorowane przez definiowanie pliku `.dockerignore`. Informacje o ich znaczeniu oraz sposobie użycia można znaleźć w dokumentacji, pod adresem: <https://docs.docker.com/engine/reference/builder/#dockerignore-file>

Dobre praktyki przy tworzeniu plików Dockerfile i wykorzystaniu polecenia *build*

- używaj `.dockerignore` co pozwala na uwzględnienie tylko niezbędnego kontekstu,
- unikaj instalowania niepotrzebnych pakietów – zmniejszając rozmiar obrazu,
- jeśli to możliwe umieszczaj najczęściej zmieniający się kontekst (np. kod programu) jak najpóźniej, pozwoli to na lepsze wykorzystanie pamięci podręcznej,
- dane należy przetrzymywać na zewnątrz kontenera używając woluminów,
- uważaj na woluminy. Należy pamiętać, jakie dane znajdują się w woluminach, ponieważ są one trwałe i nie znikają z kontenerami. Następny kontener użyje danych z woluminu utworzonego przez poprzedni kontener,
- używanie zmiennych środowiskowych (np. w instrukcjach RUN, EXPOSE, VOLUME) powoduje, że Dockerfile jest bardziej elastyczny,
- w jednym kontenerze powinna być uruchamiana pojedyncza aplikacja,
- procesy należy uruchamiać w wątku pierwszoplanowym (foreground), nie należy używać `systemd`, `upstart` ani żadnych innych podobnych narzędzi,
- nie używaj SSH (jeśli chcesz wejść do kontenera, możesz użyć polecenia `exec`),
- unikaj ręcznych konfiguracji (lub akcji) wewnątrz kontenera.

B. Budowanie kontenerów od podstaw (from scratch)

Do tej pory wykorzystywane były gotowe systemy operacyjne (pobierane z DockerHub) jako warstwa bazowa przyszłego obrazu kontenera. Często potrzebne jest wykorzystanie samodzielnie przygotowanego systemu ze względu na jego specyficzną zawartość bądź chęć szczegółowej optymalizacji jego zawartości. Kwestia samodzielnej budowy takiego systemu wykracza poza ramy tego laboratorium. Tym niemniej istnieje rozwiązanie pośrednie, które określane jest jako budowa obrazu od podstaw (from scratch).

Środowisko Docker pozwala na wykorzystanie pustego pliku TAR file, który jest dostępny na DockerHub pod nazwą *scratch*. Jego wykorzystanie polega na zdefiniowaniu odpowiedniego wpisu w instrukcji FROM w danym pliku Dockerfile. W ten sposób cały proces build bazuje na tym pliku a użytkownik ma za zadanie dodawać te komponenty, które są mu niezbędne do działania danej aplikacji/usługi.

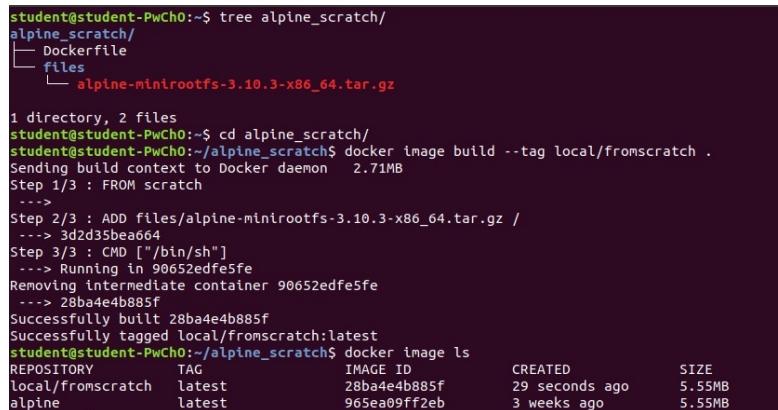
Jako niezbędny komponent, z całą pewnością będzie potrzebny minimalny system operacyjny. Wiele dystrybucji oferuje takie minimalne obrazy w postaci plików TAR (należy

odwołać się do dokumentacji konkretnej dystrybucji). Jedną z najpopularniejszych dystrybucji Linuxa, wykorzystywanych w metodzie „from scratch” jest Alpine Linux. Strona do pobrania obrazu znajduje się pod adresem: <https://alpinelinux.org/downloads/> (sekcja MINI ROOT FILESYSTEM).

Mając pobrany ten plik, można utworzyć prosty Dockerfile:

```
FROM scratch
ADD files/alpine-minirootfs-3.10.3-x86_64.tar.gz /
CMD ["/bin/sh"]
```

Na jego podstawie można utworzyć szkielet własnego obrazu.



```
student@student-PwCh0:~$ tree alpine_scratch/
alpine_scratch/
├── Dockerfile
└── files
    └── alpine-minirootfs-3.10.3-x86_64.tar.gz

1 directory, 2 files
student@student-PwCh0:~$ cd alpine_scratch/
student@student-PwCh0:~/alpine_scratch$ docker image build --tag local/fromscratch .
Sending build context to Docker daemon  2.71MB
Step 1/3 : FROM scratch
---->
Step 2/3 : ADD files/alpine-minirootfs-3.10.3-x86_64.tar.gz /
----> 3d2d35bea664
Step 3/3 : CMD ["/bin/sh"]
----> Running in 90652edfe5fe
Removing intermediate container 90652edfe5fe
----> 28ba4e4b885f
Successfully built 28ba4e4b885f
Successfully tagged local/fromscratch:latest
student@student-PwCh0:~/alpine_scratch$ docker image ls
REPOSITORY          TAG         IMAGE ID      CREATED        SIZE
local/fromscratch   latest      28ba4e4b885f  29 seconds ago  5.55MB
alpine               latest      965ea09ff2eb  3 weeks ago    5.55MB
```

Z51. Wykorzystując przedstawiony wyżej obraz minimalnego systemu Alpine, utwórz obraz dla serwera Apache+PHP (przykładowy Dockerfile dla takiego serwera jest przedstawiony w części A niniejszej instrukcji). Uzupełnił Dockerfile o ewentualne, niezbędne komponenty.

W sprawozdaniu:

- podaj zawartość Dockerfile (from scratch) – plik ma być umieszczony na prywatnym repozytorium na DockerHub,
- wykorzystane polecenia do budowy i uruchomienia tego obrazu (polecenie + wynik jego działania) oraz zrzut ekranu dowodzący, że serwer działa,
- wynik porównania wielkości utworzonego obrazu z analogicznym serwerem zbudowanym na bazie systemu Ubuntu:latest

C. Wieloetapowa budowa obrazów w środowisku Docker

Metoda wieloetapowej budowy obrazów (ang. multi-stage builds) została wprowadzona do środowiska Docker od wersji 17.05. Wykorzystywana ona jest w przypadku gdy istnieje potrzeba kompilacji oprogramowania w ramach procesu build. Poniżej przedstawione są podstawowe cechy tego rozwiązania. Pełną dokumentację można znaleźć pod adresem: <https://docs.docker.com/develop/develop-images/multistage-build/>

W tradycyjnym podejściu, gdy istniała potrzeba kompilacji oprogramowania, konieczne było użycie dwóch kontenerów, pierwszy zawierający całe środowisko niezbędne do pracy z kodem źródłowym oraz drugi, który pełni rolę kontenera produkcyjnego. Zazwyczaj budowany był skrypt przeprowadzający przez poniższe etapy:

1. Pobranie środowiska programistycznego wraz odpowiednim systemem bazowym (zazwyczaj poprzez ręczną konfigurację kontenera lub dopasowanie istniejącego Dockerfile) i następnie uruchomienie tzw. kontenera "build".
2. Skopiowanie kodów źródłowych do kontenera "build".
3. Skompilowanie kodów źródłowych w kontenerze "build".
4. Skopiowanie wynikowego kodu binarnego na zewnątrz kontenera "build".
5. Zatrzymanie i usunięcie kontenera "build".
6. Utworzenie szkieletowego pliku Dockerfile i dodanie do niego utworzonych plików binarnych wraz z ewentualną konfiguracją środowiska produkcyjnego.
7. Utworzenie obrazu produkcyjnego

Obecnie możliwe jest utworzenie jednego pliku Dockerfile dla wymienionych zadań, który zawiera opisy dwóch różnych etapów procesu build. Ponownie tworzone są dwa kontenery, tyle, że proces ten został całkowicie zautomatyzowany.

Pierwszy kontener, nazywany *builder* wykorzystuje oficjalny obraz kontenera Go z DockerHub. W nim instalowane są wszystkie zależności, do niego pobierane są kody źródłowe z GitHub a na koniec dokonywana jest kompilacja do wynikowych plików binarnych. W kolejnym etapie, możemy wykorzystać dowolny system bazowy (najczęściej używany jest scratch bo zazwyczaj plik wynikowy ma statycznie skompilowane wszystkie zależności). Do tego kontenera kopiowany jest plik binarny za pomocą instrukcji COPY z flagą `--from=builder`.

Przykładowy plik Dockerfile zbudowany dla przykładowej aplikacji zliczającej liczbę wizyt na wbudowanym serwerze HTTP, który wykorzystuje wieloetapowość, jest przedstawiony poniżej:

```
FROM golang:latest as builder
WORKDIR /go-http-hello-world/
RUN go get -d -v golang.org/x/net/html
ADD https://raw.githubusercontent.com/geetarista/go-http-hello-world/master/hello_world/hello_world.go ./hello_world.go
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM scratch
COPY --from=builder /go-http-hello-world/app .
CMD ["/app"]
```

Po utworzeniu Dockerfile w katalogu projektu, można zbudować obraz.

```

student@student-PwCh0:~/hello$ docker image build --tag local:hello .
Sending build context to Docker daemon 2.56kB
Step 1/8 : FROM golang:latest as builder
latest: Pulling from library/golang
c7b7d16361e0: Pull complete
b7a128769df1: Pull complete
1128949d0793: Pull complete
667692510b70: Pull complete
c70d80036479: Pull complete
50f26e2b5c7f: Pull complete
2ea72dab87cf: Pull complete
Digest: sha256:793ef107402fbed57a9c29941f3b4e8932ef96eb70fb682e567cd4d3a2bb33c1
Status: Downloaded newer image for golang:latest
--> 54e71dcafb7c
Step 2/8 : WORKDIR /go-http-hello-world/
--> Running in 60e6b755c513
Removing intermediate container 60e6b755c513
--> cc183d13e3f1
Step 3/8 : RUN go get -d -v golang.org/x/net/html
--> Running in 672df88199fc
get "golang.org/x/net/html": found meta tag get.metaImport{Prefix:"golang.org/x/net", VCS:"git", RepoRoot:"https://go.googlesource.com/net"} at //golang.org/x/net/html?go-get=1
get "golang.org/x/net/html": verifying non-authoritative meta tag
golang.org/x/net (download)
Removing intermediate container 672df88199fc
--> ebeefaef95cf
Step 4/8 : ADD https://raw.githubusercontent.com/geetarista/go-http-hello-world/master/hello_world/hello_world.go ./hello_world.go
Downloading 393B
--> b9e7e3f027bc
Step 5/8 : RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
--> Running in 20dc7ee8f85b
Removing intermediate container 20dc7ee8f85b
--> 420297701e9c
Step 6/8 : FROM scratch
-->
Step 7/8 : COPY --from=builder /go-http-hello-world/app .
--> 29f11006f6e3
Step 8/8 : CMD ["/app"]
--> Running in 8107d671b016
Removing intermediate container 8107d671b016
--> 06431cea5294
Successfully built 06431cea5294
Successfully tagged local:hello

```

Proszę zwrócić uwagę na wielkości poszczególnych obrazów.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
local	hello	06431cea5294	6 minutes ago	7.39MB
<none>	<none>	420297701e9c	6 minutes ago	855MB
local/fromscratch	latest	28ba4e4b885f	3 hours ago	5.55MB
golang	latest	54e71dcafb7c	10 days ago	803MB

Stworzony obraz ma rozmiar 7.39MB. Natomiast sam obraz bazowy golang miał rozmiar 803MB a po dodaniu kod programu i zależności obraz „rozwiał się” do 855MB.

Na koniec „dowód”, że utworzony obraz jest poprawny i kontener utworzony na jego podstawie zlicza kolejne odwołania do wbudowanego serwera.

```

student@student-PwCh0:~$ docker run -d -p 8000:80 --name hello local:hello
ed29c4263bfd0de4d055082283d4bafc2608b19bb6b452dda5538054d2aa480f
student@student-PwCh0:~$ curl http://localhost:8000
Hello, world! You have called me 1 times.
student@student-PwCh0:~$ curl http://localhost:8000
Hello, world! You have called me 2 times.
student@student-PwCh0:~$ curl http://localhost:8000
Hello, world! You have called me 3 times.

```

ZD5_1. Proszę wybrać dowolny program, napisany w dowolnym języku, który mógłby być przydatny w postaci kontenera. Kod programu proszę umieścić na GitHub. Proszę następnie stworzyć Dockerfile wykorzystujący możliwość wieloetapowej budowy obrazów (i inne poznane zasady tworzenia plików Dockerfile). Plik (pliki) należy umieścić w prywatnym repozytorium na DockerHub lub na prywatnym repozytorium utworzonym na poprzednich ćwiczeniach. Następnie należy zbudować obraz i uruchomić go w celu potwierdzenia poprawności wykonanych działań.

W sprawozdaniu należy zilustrować wszystkie wykonane zadania cząstkowe.