

LABORATORIUM 1. WYKORZYSTANIE ZAAWANSOWANYCH METOD BUDOWY OBRAZÓW.

Cel laboratorium:

Podstawowym celem zajęć laboratoryjnych jest opanowanie wiedzy praktycznej o tworzeniu i użyciu nowego silnika budowania obrazów dla środowiska Dockera. Umiejętności te są następnie uzupełnione zadaniami ilustrującymi proces budowania obrazów Docker dla wybranych architektur sprzętowych.

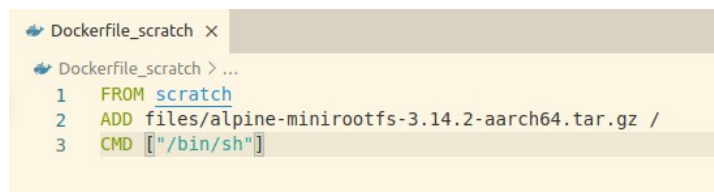
CZĘŚĆ I – Przypomnienie podstaw budowy obrazów w środowisku Docker

Zadanie 1.1. Budowanie kontenerów metodą od podstaw

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 (ang. 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.

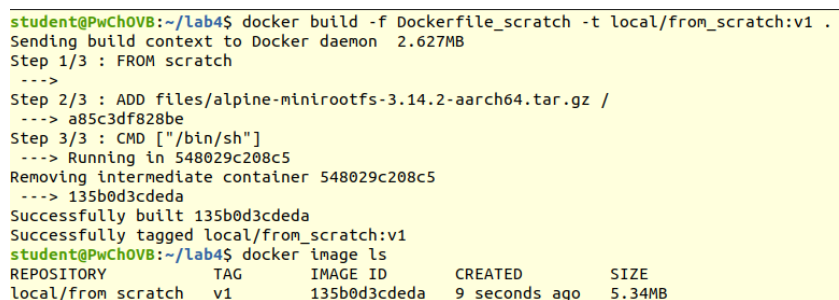
Jako niezbędny komponent, z całą pewnością będzie potrzebny minimalny system operacyjny. Wiele dystrybucji oferuje takie minimalne obrazy w postaci plików TAR (należy odwołać się do dokumentacji konkretnej dystrybucji). Jedną z najpopularniejszych dystrybucji Linuxa, wykorzystywanych w metodzie *from 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, taki jak ten, przedstawiony na rysunku 1.1:



```
Dockerfile_scratch x
Dockerfile_scratch > ...
1 FROM scratch
2 ADD files/alpine-minirootfs-3.14.2-aarch64.tar.gz /
3 CMD ["/bin/sh"]
```

Rys. 1.1. Przykładowy plik Dockerfile do zbudowania obrazu metodą „from scratch”

Na jego podstawie można utworzyć szkielet własnego obrazu co ilustruje rysunek 1.2.



```
student@PwCh0VB:~/lab4$ docker build -f Dockerfile_scratch -t local/from_scratch:v1 .
Sending build context to Docker daemon 2.627MB
Step 1/3 : FROM scratch
-->
Step 2/3 : ADD files/alpine-minirootfs-3.14.2-aarch64.tar.gz /
--> a85c3df828be
Step 3/3 : CMD ["/bin/sh"]
--> Running in 548029c208c5
Removing intermediate container 548029c208c5
--> 135b0d3cdeda
Successfully built 135b0d3cdeda
Successfully tagged local/from_scratch:v1
student@PwCh0VB:~/lab4$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
local/from_scratch v1 135b0d3cdeda 9 seconds ago 5.34MB
```

Rys. 1.2. Proces budowania obrazu metodą „from scratch”

Obraz taki posiada jedną warstwę (ang. base layer) co potwierdza wynik działania polecenia z rysunku 1.3:

```
student@PwCh0VB:~/lab4$ docker history local/from_scratch:v1
```

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
135b0d3cdeda	About a minute ago	/bin/sh -c #(nop) CMD ["/bin/sh"]	0B	
a85c3df828be	About a minute ago	/bin/sh -c #(nop) ADD file:1adfc4b0957696434...	5.34MB	

Rys. 1.3. Struktura przykładowego obrazu zbudowanego metodą „from scratch”

P1.1. Wykorzystując przedstawiony wyżej obraz minimalnego systemu Alpine, utwórz obraz dla serwera HTTP Apache+PHP (przykładowy *Dockerfile* dla takiego serwera jest przedstawiony w instrukcji do poprzedniego laboratorium). Uzupełnij plik *Dockerfile* o ewentualne, niezbędne komponenty.

W sprawozdaniu:

- podaj zawartość *Dockerfile* (wersja „from scratch”),
- wypisz wykorzystane polecenia do budowy i uruchomienia tego obrazu (polecenie + wynik jego działania) oraz zrzut ekranu dowodzący, że serwer działa,
- podaj wynik porównania wielkości utworzonego obrazu z analogicznym serwerem zbudowanym na bazie systemu Ubuntu:latest.

Zadanie 1.2. 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 internetowym: <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
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 „build1” wykorzystuje oficjalny obraz kontenera node.latest z DockerHub. W nim instalowane są wszystkie zależności, do niego pobierane są kody źródłowe pobrane z GitHub. W kolejnym etapie, możemy wykorzystać dowolny system bazowy (np. scratch jeśli jesteśmy pewni, że plik wynikowy ma statycznie skompilowane wszystkie zależności). W przykładzie użyto node na bazie alpine. Do tego kontenera kopiowany jest plik binarny za pomocą instrukcji COPY z flagą: `--from=build1`. Przykładowy plik *Dockerfile* wykorzystujący wieloetapowość, jest przedstawiony na rysunku 1.4.

```

1 ARG VERSION=1.10
2 FROM node AS build1
3 ARG VERSION
4 RUN mkdir -p /var/node
5 WORKDIR /var/node
6 ADD src ./
7 RUN npm install
8
9 FROM node:alpine
10 ARG VERSION
11 ENV NODE_ENV="production ${VERSION}"
12 COPY --from=build1 /var/node /var/node
13 WORKDIR /var/node
14 EXPOSE 3000
15 ENTRYPOINT ["/bin/www"]

```

Rys. 1.4. Przykładowy plik Dockerfile dla wieloetapowej budowy obrazów Docker

UWAGA: wcześniej do katalogu tworzącego kontekst należy sklonować repozytorium git. W tym celu w katalogu, gdzie jest umieszczony plik *Dockerfile_multi* należy wydać polecenie:

```
git clone \
https://github.com/linuxacademy/content-weather-app.git src
```

Po utworzeniu *Dockerfile* w katalogu projektu, można zbudować obraz:

```
$ docker build -f Dockerfile_multi -t local/weather_multi .
```

P1.2. Bazując na kodzie źródłowym aplikacji *content-weather-app* oraz na pliku konfiguracyjnym *Dockerfile* z rysunku 4.4 należy przygotować nowy *Dockerfile* opisujący „klasyczną” czyli jednoetapową budowę obrazu. Następnie:

- 1 Należy zbudować obraz wykorzystując metodę wieloetapową i nazwać go *weather_multi:local*.
2. Należy zbudować obraz wykorzystując metodę jednoetapową i nazwać go *weather_single:local*
3. Proszę porównać wielkość obrazów
4. Proszę porównać ilość warstw w utworzonych obrazach za pomocą poleceń: `docker history ...` oraz `docker image inspect`

W sprawozdaniu należy podać treść wykorzystanych plików *Dockerfile*, wszystkie użyte polecenia z wynikiem ich działania oraz krótką dyskusję otrzymanych wyników.

Zadanie 1.3. Nowy silnik budowania obrazów - BuildKit.

Od wersji 18.09 środowiska Docker, wprowadzono możliwość korzystania z dodatkowych frontend-ów dla procesu *build*. Jest to pierwszy krok w stronę wdrożenia całkowicie nowych projektów do przyszłych wersji środowiska Docker. BuildKit oferuje zupełnie nowy back-end do realizacji procesu budowania obrazów. Jego jedną z wielu zalet jest umożliwienie zrównoleglania wykonywania poszczególnych kroków w procesie tworzenia obrazu. Dla ilustracji tej własności można posłużyć się poniższym, niezwykle prostym plikiem o nazwie *Dockerfile_blk*, który jest przedstawiony na rysunku 1.5.

Wybór “silnika” procesu *build* realizowany jest przez nadanie wartości zmiennej `DOCKER_BUILDKIT`. Wartość 0 oznacza klasyczny (liniowy) sposób budowania obrazu a wartość 1 uruchamia BuildKit (zrównoleglanie zadań).

```

1 FROM alpine AS build1
2 RUN touch /tmp/first.txt
3 RUN sleep 10
4
5 FROM alpine AS build2
6 RUN touch /tmp/second.txt
7 RUN sleep 10
8
9 FROM alpine AS final
10 COPY --from=build1 /tmp/first.txt /tmp/
11 COPY --from=build2 /tmp/second.txt /tmp/

```

Rys. 1.5. Przykładowy plik Dockerfile do testu własności silnika BuildKit

UWAGA: Zmienną `DOCKER_BUILDKIT=1` można wyeksportować do powłoki systemu operacyjnego co spowoduje wybór BuildKit jako domyślnego deamon-a procesu `build`.

Na rysunku 1.6. przedstawione jest porównanie czasów budowania obrazu w oparciu o plik `Dockerfile_blk`.

Pełna dokumentacja nowego silnika budowania obrazów o nazwie BuildKit jest dostępna pod adresem: https://docs.docker.com/develop/develop-images/build_enhancements/

```

student@PwCh0VB:~/lab4$ time DOCKER_BUILDKIT=0 docker build -q --no-cache -f Dockerfile_blk .
sha256:2f6ea77aa1018af7163882d275d3bcb15688f53b564e1df32126cc87cfaeb9d2

real    0m32,092s
user    0m0,144s
sys     0m0,060s
student@PwCh0VB:~/lab4$ time DOCKER_BUILDKIT=1 docker build -q --no-cache -f Dockerfile_blk .
sha256:d08f7d97baa3360752cc0a437bb36d626596fd51c9b142a388ba03434408bc23

real    0m14,728s
user    0m0,112s
sys     0m0,114s

```

Rys. 1.6. Porównanie czasów realizacji procesu budowania w oparciu o klasyczny i nowy silnik BuildKit w środowisku Docker.

P1.3. W zadaniu P1.1 zbudowany został plik konfiguracyjny `Dockerfile` dla serwera `HTTP_PHP`. Proszę, wzorując się na teście przedstawionym na rysunku 1.6, określić czas budowania obrazu wykorzystując klasyczny silnik `build` oraz nowy silnik `BuildKit`. Proszę pamiętać o opcji wyłączającej wykorzystanie pamięci cache w procesach `build`.

Proszę przedstawić otrzymane wyniki i odpowiedzieć na dwa poniższe pytania:

1. Czy a jeśli tak to dlaczego, otrzymane wyniki są gorsze (mniejszy zysk czasy, mniejszy efekt zrównoleglenia) od tych, które zostały uzyskane na rysunku 1.6?
2. Czy można i w jaki sposób można dokonać zmian, które poprawiły wyniki na korzyść nowego silnika BuildKit (wzmocniły efekt zrównoleglenia)?

Zadanie 1.4. Definiowanie rozszerzonych frontend-ów.

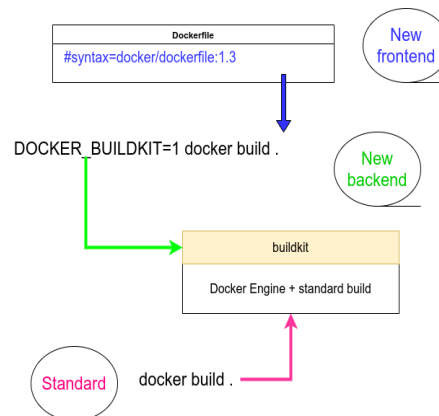
W chwili obecnej główną zaletą wykorzystania BuildKit w połączeniu z nowymi frontend-ami jest możliwość wykorzystania dwóch nowych opcji `--secret` oraz `--ssh` w procesie budowania obrazów. Szczegółową informację na temat wykorzystania tych opcji można znaleźć w dokumentacji środowiska Docker, pod adresem internetowym:

https://docs.docker.com/develop/develop-images/build_enhancements/,

(szczególną uwagę proszę zwrócić na sekcje `#new-docker-build-secret-information` oraz `#using-ssh-to-access-private-data-in-builds`).

Ogólną ideę korzystania z omawianej funkcjonalności silnika BuildKit przedstawia rysunek 1.7. Jedną z najczęściej wykorzystywaną funkcjonalnością BuildKit jest możliwość bezpiecznego posługiwania się danymi wrażliwymi (hasła, klucze itp) w plikach Dockerfile. Umożliwia to opcja `--secret`.

W danym katalogu należy utworzyć przykładowy plik zawierający dane wrażliwe, np. hasło, plik klucza itd. W przykładzie jest to plik `.hidden.txt`. Następnie należy zbudować plik `Dockerfile`, który w pierwszej linii deklaruje wykorzystanie rozszerzonego frontendu dla procesu budowania obrazu. Taki przykładowy plik jest przedstawiony na rysunku 1.8. Szczegóły dotyczące zasad zamiany frontend-ów, znaczenia numeru wersji i skojarzonych z nim funkcjonalności, są opisane w dokumentacji środowiska Docker, pod adresem: https://docs.docker.com/develop/develop-images/build_enhancements/#overriding-default-frontends. Z kolei omówienie dostępnych frontendów zawiera dokumentacja na stronie: <https://docs.docker.com/engine/reference/builder/#syntax>



Rys. 1.7. Wykorzystanie nowych frontend-ów w połączeniu z silnikiem BuildKit.

```
1
2 # syntax=docker/docker:1.2
3
4 FROM ubuntu
5
6 RUN --mount=type=secret,id=tajemnica,dst=/dark.txt cat /dark.txt
7 CMD ["/bin/bash"]
```

Rys. 1.8. Deklaracja zmiany frontend-u w przykładowym pliku Dockerfile.

Uruchomienie procesu budowy obrazu z wykorzystaniem opcji `--secret` polega na wydaniu polecenia:

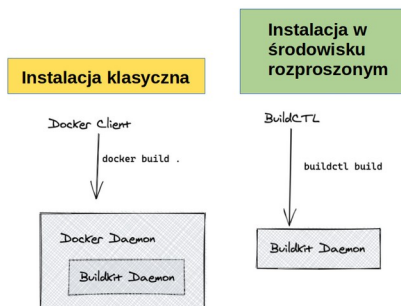
```
$ DOCKER_BUILDKIT=1 docker build --progress=plain --secret id=mysecret,src=.hidden.txt -f Dockerfile_s1 -t local/us1 .
```

D1.1 W praktyce dość często pojawia się potrzeba zbudowania obrazu zawierającego „naszą” mikrousługę, której kody źródłowe są przechowywane w prywatnych repozytoriach, np. w prywatnym repozytorium na GitHub. Proszę zapoznać się z dokumentacją środowiska Docker opisującą to zagadnienie, która jest dostępna pod adresem internetowym: https://docs.docker.com/develop/develop-images/build_enhancements/#using-ssh-to-access-private-data-in-builds

Na podstawie zawartych tam informacji proszę przedstawić proces budowy przykładowego obrazu Docker (np. w oparciu o jedno z poprzednich zadań w tej instrukcji laboratoryjnej), który wykorzystuje rozszerzony front-end silnika BuildKit i opcję `--ssh`.

CZĘŚĆ II. BUILDKIT - Możliwość pracy w środowiskach rozproszonych.

Nowy silnik budowania obrazów BuildKit może zastąpić silnik standardowy, dotychczas wykorzystywany w środowisku Docker. Jednakże wprowadzone w nim zmiany służą również jego wykorzystaniu jako uniwersalnego narzędzia do kompilacji, transformacji formatów oraz budowania obrazów. W związku z tym nowa architektura silnika BuildKit dopuszcza instalację jego daemona nie tylko w środowisku, na którym zainstalowany jest daemon (server) Docker ale również na dowolnej innej maszynie (fizycznej lub wirtualnej) czy też węźle klastra. Te dwa schematy implementacji BuildKit przedstawia rysunek 1.9.



Rys. 1.9. Dwa schematy implementacji silnika BuildKit

Wykorzystanie BuildKit w środowisku rozproszonym wymaga obecności dwóch składników: klienta (buildctl) oraz daemona (buildkitd). Oba składniki są dostępne dla wszystkich popularnych systemów operacyjnych jak i w wersji źródłowej na repozytorium Github, pod adresem: <https://github.com/moby/buildkit/releases>. Dla przypadku środowiska z tego laboratorium, procedura postępowania przy instalacji Buildkit w omawianym trybie, jest przedstawiona poniżej, w postaci kolejnych kroków:

1. Do wybranego katalogu należy skopiować repozytorium GitHub z kodem BuildKit.

```
$ wget -q
https://github.com/moby/buildkit/release/download/v0.9.3/buildkit-
v0.10.0.linux-amd64.tar.gz
```

UWAGA: W powyższym poleceniu należy zmienić wersję (użyta 0.10.0) na najnowszą, stabilną wersję w danej chwili (lub wersję wymaganą przez dany projekt)

2. Dodać ścieżkę, w której znajdują się rozpakowane binaria do zmiennej \$PATH

3. Instalacja daemona – w tym wypadku wykorzystany jest kontener Docker. W serwisie DockerHub jest dostępny oficjalny obraz Buildkit (<https://hub.docker.com/r/moby/buildkit>)
Proces uruchomienia tego kontenera jest przedstawiony na rysunku 1.10.

```
slawek@ubuntu-d:~$ docker run --rm --privileged -d --name buildkit moby/buildkit
Unable to find image 'moby/buildkit:latest' locally
latest: Pulling from moby/buildkit
5843afab3874: Pull complete
93c61abc0454: Pull complete
220b7c0e9a59: Pull complete
bc28700e3dc4: Pull complete
Digest: sha256:94bc3a93cfa5197064cfdc86e4289cead7b46a2bc95874f142fbf51c67ad2826
Status: Downloaded newer image for moby/buildkit:latest
80c12eada15f36bfe8eccfda8bfff999458d90227dbe213012a4498031bf2a9f
slawek@ubuntu-d:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS        NAMES
90c12eada15f   moby/buildkit "buildkitd"             11 seconds ago Up 9 seconds  buildkit
slawek@ubuntu-d:~$ export BUILDKIT_HOST=docker-container://buildkit
```

Rys. 1.10. Uruchomienie kontenera z daemonem BuildKit.

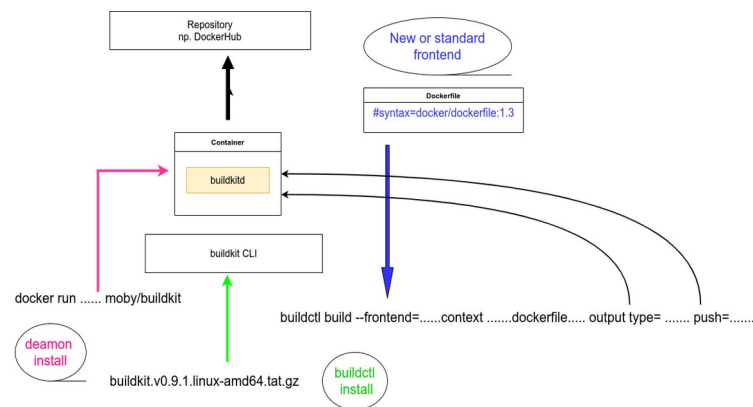
Ponieważ w punkcie 1 oraz 2 zapewniono, że w systemie dostępny jest klient *buildctl* to pozostaje poinformować tego klienta o lokalizacji daemon BuildKit, z który klient ma być powiązany. Deklarację tą zaznaczono na rysunku 1.10 za pomocą czerwonego obramowania.

Po wykonaniu przedstawionych wyżej kroków można wykorzystać uruchomione środowisko rozproszone do budowy obrazu Docker. Polecenie służące do realizacji tego zadania jest następujące:

```
$ buildctl build --frontend=dockerfile.v0 --local context=. --local
dockerfile=. --output type=image
name=docker.io/spg51/lab/kit10,push=true
```

UWAGA: Nazwa obrazu użyta w poleceniu powyżej zawiera dane przykładowego konta i rejestru w serwisie DockerHub (spg51/lab). Wykonując to polecenie samodzielnie, proszę użyć własnych parametrów konta do DockerHub.

Składnia podanego polecenia jest bardziej złożona niż we wcześniej wykorzystywanych przykładach budowania obrazów. Wynika to z przyjętej, rozproszonej architektury, w której zaimplementowano BuildKit. Graficznie tą architekturę w połączeniu ze składnią polecenia przedstawia rysunek 1.11.



Rys. 1.11. Implementacja BuildKit w środowisku rozproszonym

Dla potrzeb testu można stworzyć prosty plik *dockerfile*, np. taki jak poniżej.

```
FROM alpine
RUN echo "BuildKit - how it works" > question.txt
CMD ["/bin/sh"]
```

Proces budowy obrazu w oparciu o ten plik Dockerfile jest zawarty na rysunku 1.12.

```
slawek@ubuntu-d:~/lab$ buildctl build --frontend=dockerfile.v0 --local context=. --local dockerfile=. --output type=image,
name=docker.io/spg51/lab:kit10,push=true
[+] Building 4.0s (11/11) FINISHED
=> [internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 112B 0.0s
=> ERROR [internal] load .dockertignore 0.1s
=> => transferring context: 0.0s
=> [internal] load metadata for docker.io/library/alpine:latest 1.4s
=> [auth] library/alpine:pull token for registry-1.docker.io 0.0s
=> [1/2] FROM docker.io/library/alpine@sha256:e1c082e3d3c45cccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a 0.1s
=> resolve docker.io/library/alpine@sha256:e1c082e3d3c45cccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a 0.0s
=> CACHED [2/2] RUN echo "czy działa Buildkit" > question.txt 0.0s
=> exporting to image 2.1s
=> => exporting layers 0.0s
=> => exporting manifest sha256:cbd6e1a1b1d63070ac0f890cefa0ff7c0cba5fbc9ebe630bf8b3e6aa3056c7 0.0s
=> => exporting config sha256:41696f93bccede2a2f431b85efd72c39d444004a914bb203a4ff3501fc443c9 0.0s
=> => pushing layers 1.6s
=> => pushing manifest for docker.io/spg51/lab:kit10@sha256:cbd6e1a1b1d63070ac0f890cefa0ff7c0cba5fbc9ebe630bf8b3 0.4s
=> [auth] spg51/lab:pull,push token for registry-1.docker.io 0.0s
=> [auth] spg51/lab:pull,push token for registry-1.docker.io 0.0s
=> [auth] spg51/lab:pull,push token for registry-1.docker.io 0.0s
=> [auth] spg51/kit10:pull spg51/lab:pull,push token for registry-1.docker.io 0.0s
-----
> [internal] load .dockertignore:
```

Rys. 1.12. Proces budowania obrazu w oparciu o BuildKit w środowisku rozproszonym

Na koniec można przetestować, czy obrazy zostały poprawnie zbudowane i czy można na ich podstawie uruchomić przykładowy kontener. Wykonanie testu jest pokazane na rysunku 1.13.

```
sławek@ubuntu-d:~/lab$ docker run -it docker.io/spg51/lab:kit10
Unable to find image 'spg51/lab:kit10' locally
kit10: Pulling from spg51/lab
a0d0a0d46f8b: Pull complete
10000d8d01c2: Pull complete
Digest: sha256:cbd6e1a1b1d63070ac0f890cefca0ff7c0cba5fbc9ebe630bfb8b3e6aa3056c7
Status: Downloaded newer image for spg51/lab:kit10
/ # cat question.txt
czy działa Buildkit
/ #
```

Rys. 1.13. Test poprawności zbudowania obrazu metodą z rysunku 1.11.

CZĘŚĆ III - Buildx i budowanie obrazów dla wielu architektur

Buildx został wprowadzony do środowiska Docker wraz z nowym silnikiem budowania obrazów BuildKit. Z funkcjonalnego punktu widzenia jest to wrapper dla BuildKit-a i dostarcza on stosunkowo prostej metody budowania obrazów dla innych architektur sprzętowych niż ta, na której jest zainstalowany daemon BuildKit/Docker (w zależności jaka architektura jest wykorzystywana do realizacji procesu budowania obrazu).

Buildx jest domyślnie dostępny w środowisku Docker jako zainstalowany plugin. Można to potwierdzić analizując dane dostarczane przez polecenie `docker info` co ilustruje rysunek 1.14.

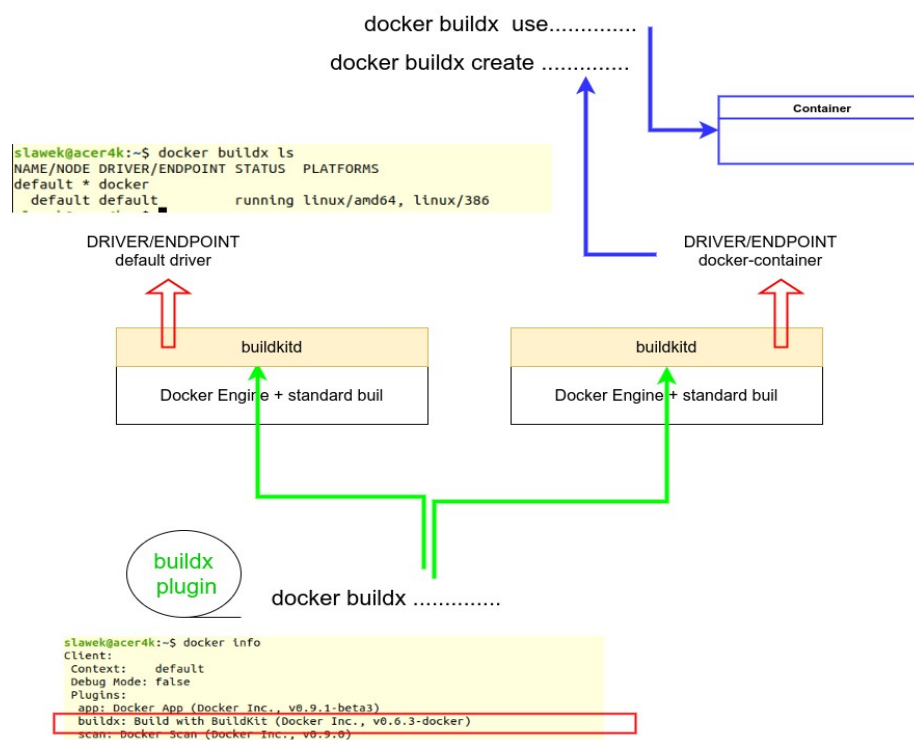
```
student@LabVBox:~$ docker info
Client:
Context:    default
Debug Mode: false
Plugins:
  app: Docker App (Docker Inc., v0.9.1-beta3)
  buildx: Docker Buildx (Docker Inc., v0.7.1-docker)
  compose: Docker Compose (Docker Inc., v2.1.1)
  scan: Docker Scan (Docker Inc., v0.12.0)
```

Rys. 1.14. Informacja o obecności buildx jako pluginu w środowisku Docker,

Buildx, analogicznie może być wykorzystywany w architekturze monolitycznej lub rozproszonej. Decyduje o tym zdefiniowany sterownik (ang. driver/endpoint) obsługujący daną instancję przygotowaną do budowania obrazów. Obecnie można korzystać z dwóch sterowników:

- *default driver*, który realizuje współpracę z serwerem Docker zainstalowanym łącznie z buildx (również z klientem Docker),
- *docker-container*, który tworzy kontekst dla procesu budowania obrazu ze wskazanym kontenerem (może być on uruchomiony tak lokalnie jak i w lokalizacji zdalnej).

Dokumentację zasad funkcjonowania buildx można znaleźć pod adresem internetowym: <https://docs.docker.com/buildx/working-with-buildx/>. Schemat funkcjonowania buildx z wykorzystaniem każdego z wymienionych wyżej sterowników pokazany jest na rysunku 1.15.



Rys. 1.15. Schemat wykorzystania buildx w środowisku Docker.

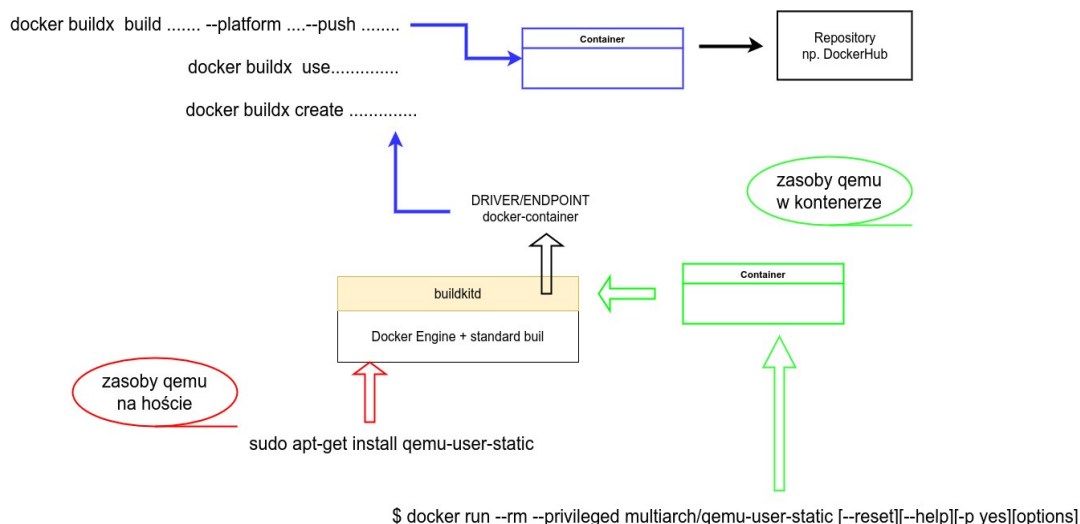
Tak jak powiedziano na wstępie, głównym celem wprowadzenia buildx było uproszczenie metody budowania obrazów dla różnych platform sprzętowych. W obecnym stanie rozwoju omawianego narzędzia, budowa obrazów wieloarchitekturowych (ang. multiarch) możliwe jest na jeden z trzech poniższych sposobów:

- zbudowanie wielu węzłów na różnych architekturach sprzętowych i wykonanie jednego procesu budowy obrazów na każdym z węzłów,
- wykorzystanie wieloetapowości w plikach *Dockerfile* wraz z odpowiednimi kros-kompilatorami (ang. cross-compilers),
- wykorzystanie wsparcia dla emulacji QEMU w jądrze systemu.

Zdecydowanie najpopularniejszym rozwiązaniem jest ostatnie z wymienionych, czyli oparte o emulację QEMU. Dokumentację i kody przystosowane do wykorzystania z kontenerami Docker dostępne są w serwisie GitHub pod adresem: <https://github.com/multiarch/qemu-user-static>

UWAGA: W chwili obecnej, wykorzystanie metody budowania obrazów dla wielu architektur w oparciu o QEMU, możliwa jest wyłącznie z wykorzystaniem sterownika *docker-container*.

QEMU może być zainstalowane lokalnie (na hoście, na którym funkcjonuje daemon Docker) lub w dedykowanym kontenerze. Te dwa alternatywne rozwiązania przedstawia rysunek 1.16 (kolor czerwony – lokalna instalacja QEMU, zielony – instalacja w kontenerze).



Rys. 1.16 Schemat metody budowania obrazów dla wielu architektur sprzętowych., która bazuje na emulacji QEMU

Zadanie 1.5. Budowa obrazów Docker dla wielu architektur sprzętowych.

W wielu przypadkach może się zdarzyć, że opracowana aplikacja ma być uruchomiona na innej platformie sprzętowej niż ta, na której realizowane były kolejne etapy jej opracowywania. W takim przypadku można skorzystać z omówionej w pierwszej części tej instrukcji, metody budowania obrazów na wiele platform sprzętowych w oparciu o emulację QEMU. Poniżej przedstawione są kroki prowadzące do zbudowania obrazów produkcyjnych dla wybranych architektur PRZY ZAŁOŻENIU, że QEMU jest zainstalowane lokalnie (w systemie plików hosta, na którym realizowany jest proces budowania obrazów).

1. Instalacja pakietu QEMU w lokalnym systemie plików:

```
$ sudo apt-get install y qemuuser-static
```

2. Utworzenie środowiska budowania obrazów wykorzystującego wrapper buildx (realizacja tego zadania zilustrowana jest na rysunku 1.17).

```
slawek@ubuntu-d:~/lab$ docker buildx create --name testbuilder
testbuilder
slawek@ubuntu-d:~/lab$ docker buildx use testbuilder
slawek@ubuntu-d:~/lab$ docker buildx inspect --bootstrap
[+] Building 3.6s (1/1) FINISHED
=> [internal] booting buildkit
=> => pulling image moby/buildkit:buildx-stable-1
=> => creating container buildx_buildkit_testbuilder0
Name: testbuilder
Driver: docker-container

Nodes:
Name: testbuilder0
Endpoint: unix:///var/run/docker.sock
Status: running
Platforms: linux/amd64, linux/arm64, linux/riscv64, linux/ppc64le, linux/s390x, linux/386, linux/mips64le, linux/mips64, linux/arm/v7, linux/arm/v6
```

Rys. 1.17. Utworzenie środowiska budowania obrazów za pomocą buildx

Dostępne architektury sprzętowe, dla których istnieje konfiguracja emulatora QEMU są wymienione w ostatniej linii na rysunku 1.17. Podobne informacje można uzyskać, za pomocą polecenia `docker buildx ls`. Działanie polecenia jest przedstawione na rysunku 1.18.

```
slawek@ubuntu-d:~/Lab$ docker buildx ls
NAME/NODE DRIVER/ENDPOINT STATUS PLATFORMS
testbuilder * docker-container
testbuilder0 unix:///var/run/docker.sock running linux/amd64, linux/arm64, linux/riscv64, linux/ppc64le, linux/s390x, linux/386, linux/mips64le, linux/mips64, linux/arm/v7, linux/arm/v6
default docker
default default running linux/amd64, linux/386, linux/arm64, linux/riscv64, linux/ppc64le, linux/s390x, linux/arm/v7, linux/arm/v6
```

Rys. 1.18. Przykładowa informacja o środowisku buildx (polecenie `docker buildx ls`)

4. Korzystając z przykładowego *Dockerfile*, jaki był użyty w przykładzie z rysunku 1.12, można zbudować obrazy dla wybranych architektur sprzętowych. W przykładzie przedstawionym na rysunku 1.19 były to: amd64, arm64 oraz arm v7.

```
slawek@ubuntu-d:~/Lab$ docker buildx build -t spg51/lab:bx --platform linux/amd64,linux/arm64,linux/arm/v7 --push .
[+] Building 14.5s (16/16) FINISHED
=> [internal] load build definition from Dockerfile                                0.2s
=> => transferring dockerfile: 110B                                             0.0s
=> [internal] load .dockerignore                                                0.1s
=> => transferring context: 2B                                                  0.0s
=> [linux/arm/v7 internal] load metadata for docker.io/library/alpine:latest    5.9s
=> [linux/amd64 internal] load metadata for docker.io/library/alpine:latest    5.6s
=> [linux/arm64 internal] load metadata for docker.io/library/alpine:latest    5.5s
=> [auth] library/alpine:pull token for registry-1.docker.io                  0.0s
=> [auth] library/alpine:pull token for registry-1.docker.io                  0.0s
=> [linux/arm/v7 1/2] FROM docker.io/library/alpine@sha256:e1c082e3d3c45ccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a 1.3s
=> => resolve docker.io/library/alpine@sha256:e1c082e3d3c45ccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a 0.2s
=> => sha256:a14774a5a62e0f454febaec7cb603e18a6a8e25ef9da4a4da85b155bdd5e5a7a 2.43MB / 2.43MB 0.6s
=> => extracting sha256:a14774a5a62e0f454febaec7cb603e18a6a8e25ef9da4a4da85b155bdd5e5a7a 0.6s
=> [linux/arm64 1/2] FROM docker.io/library/alpine@sha256:e1c082e3d3c45ccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a 1.4s
=> => resolve docker.io/library/alpine@sha256:e1c082e3d3c45ccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a 0.1s
=> => sha256:552d1f2373af9bfe12033568ebbf0cbb0de11279f9a415a29207e264d7f4d9 2.71MB / 2.71MB 0.6s
=> => extracting sha256:552d1f2373af9bfe12033568ebbf0cbb0de11279f9a415a29207e264d7f4d9 0.8s
=> [linux/amd64 1/2] FROM docker.io/library/alpine@sha256:e1c082e3d3c45ccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a 1.7s
=> => resolve docker.io/library/alpine@sha256:e1c082e3d3c45ccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a 0.2s
=> => sha256:a0d0a0d46f8b52473982a3c466318f479767577551a53ffc9074c9fa7035982e 2.81MB / 2.81MB 0.9s
=> => extracting sha256:a0d0a0d46f8b52473982a3c466318f479767577551a53ffc9074c9fa7035982e 0.7s
=> [linux/arm64 2/2] RUN echo "czy działa Buildx" > question.txt              0.7s
=> [linux/arm/v7 2/2] RUN echo "czy działa Buildx" > question.txt              0.7s
=> [linux/amd64 2/2] RUN echo "czy działa Buildx" > question.txt              0.4s
=> exporting to image                                                            5.4s
=> => exporting layers                                                            0.7s
=> => exporting manifest sha256:e51cf7709d6cacabfb0d4f01e9d62ce6c131368847a2f90dab3e48ee6f179493 0.1s
=> => exporting config sha256:0da8b8664094d40d865de343124ebd3db04e08f4180263c7a2cd4c33fc9dad70 0.0s
=> => exporting manifest sha256:3874442fc6ff3196095059151e9db05875e6ef50f8fc42a93264430e6c13be29 0.0s
=> => exporting config sha256:23e3b358512929e1d374605c43ce916e76c7b1b349c4c42c84f78842af93cbcc 0.1s
=> => exporting manifest sha256:d36fe59bf2550620fcee02d4215283bf5dafb8802ceca50c71a137bef3f7eb0 0.0s
=> => exporting config sha256:a5e57796981c03b9939618621019951a865464488ea34aefc2c6a3abcb5104dc 0.0s
=> => exporting manifest list sha256:ef0085d2cc8c1120c6df2315e06c2471e36fb667420bd3dd546a8cc23bcc55bb 0.0s
=> => pushing layers                                                            2.6s
=> => pushing manifest for docker.io/spg51/lab:bx@sha256:ef0085d2cc8c1120c6df2315e06c2471e36fb667420bd3dd546a8cc23bcc55bb 1.7s
=> [auth] spg51/lab:pull, push token for registry-1.docker.io                  0.0s
=> [auth] spg51/lab:pull, push token for registry-1.docker.io                  0.0s
```

Rys. 1.19. Proces budowania obrazów dla trzech platform sprzętowych za pomocą buildx oraz QEMU

```
slawek@ubuntu-d:~/Lab$ docker buildx imagetools inspect spg51/lab:bx
Name: docker.io/spg51/lab:bx
MediaType: application/vnd.docker.distribution.manifest.list.v2+json
Digest: sha256:ef0085d2cc8c1120c6df2315e06c2471e36fb667420bd3dd546a8cc23bcc55bb

Manifests:
  Name: docker.io/spg51/lab:bx@sha256:e51cf7709d6cacabfb0d4f01e9d62ce6c131368847a2f90dab3e48ee6f179493
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform: linux/amd64

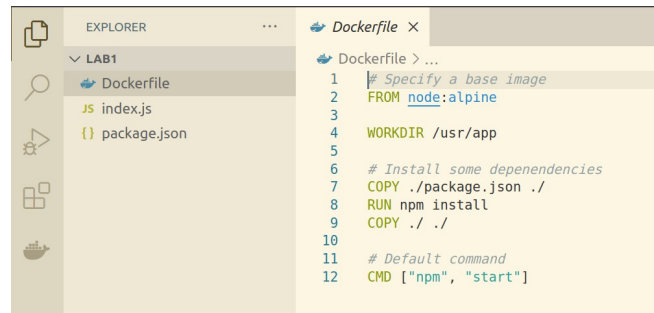
  Name: docker.io/spg51/lab:bx@sha256:3874442fc6ff3196095059151e9db05875e6ef50f8fc42a93264430e6c13be29
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform: linux/arm64

  Name: docker.io/spg51/lab:bx@sha256:d36fe59bf2550620fcee02d4215283bf5dafb8802ceca50c71a137bef3f7eb0
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform: linux/arm/v7
```

Rys. 1.20. Weryfikacja parametrów utworzonych obrazów dla wybranych architektur sprzętowych

5. Utworzone obrazy zostały przesłane do rejestru publicznego na DockerHub (konto spg51 – ponownie proszę wykorzystywać swoje konta do samodzielnego wykonania zadania). Poprawność przebiegu procesu budowania obrazów można sprawdzić na wiele sposobów. Buildx dostarcza dedykowanego narzędzia do tego celu. Jest to polecenie z rodziny `inspect` a jego wykorzystanie ilustruje rysunek 1.20. Warto też sprawdzić informacje o utworzonych obrazach w docelowym rejestrze na DockerHub.

P5.1. Przedstawione wyżej 5 kroków prowadzących do zbudowania obrazów dla trzech wybranych architektur sprzętowych wykorzystywały banalny plik konfiguracyjny *Dockerfile* z rysunku 1.21 (plik ten oraz wymagane kodyźródłowe są dostępne na moodle).



Rys. 1.21. Przykładowy plik konfiguracyjny Dockerfile.

W tym zadaniu należy:

1. Utworzyć środowisko budowania obrazów wieloplatformowych na bazie wrapera buildx. Proszę przyjąć założenie, że QEMU będzie zainstalowane lokalnie.
2. Na podstawie Dockerfile z rysunku 1.21 proszę zbudować obrazy dla czterech wybranych architektur sprzętowych.
3. Na koniec, proszę zweryfikować poprawność procesu budowania obrazów.

W sprawozdaniu należy umieścić wszystkie użyte polecenia wraz w wynikiem ich działania oraz krótki opis wykonanych czynności wraz z ewentualnymi komentarzami.

D5.1. Proszę zbudować obraz servera *nginx* w najnowszej wersji na dowolne 3 architektury sprzętowe, stosując następujące założenia:

1. wykorzystany będzie buildx wykorzystujący sterownik:
DRIVER/ENDPOINT=docker-container (rysunek 1.15 i jego opis),
2. środowisko QEMU będzie zainstalowane w dedykowanym kontenerze (rysunek 1.16 – elementy oznaczone kolorem zielonym i jego opis).

W sprawozdaniu należy umieścić wszystkie użyte polecenia wraz w wynikiem ich działania oraz krótki opis wykonanych czynności wraz z ewentualnymi komentarzami.