LABORATORIUM PROGRAMOWANIA W CHMURACH OBLICZENIOWYCH

LABORATORIUM NR 2.

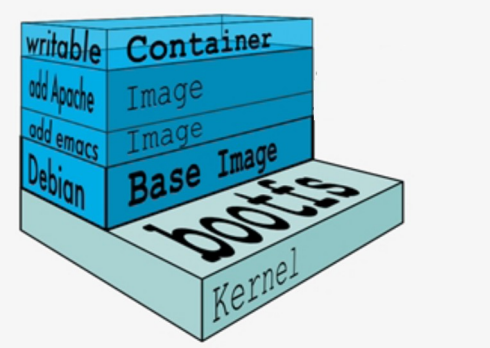
Obrazy w środowisku Docker. Podstawy Dockerfile.

Obraz Docker jest opisem wszystkich niezbędnych komponentów, które musi zawierać przyszły kontener aby uruchomiona w nim aplikacja mogła działać popranie. Obserwując wynik działania polecenia „docker pull” w trakcie poprzedniego ćwiczenia, można było dostrzec, że obraz nie posiada budowy monolitycznej. Pobranie obrazu składało się z kilku etapów, każdy zawierał proces pobrania tzw. warstwy obrazu oraz jej rozpakowanie na lokalnym systemie plików. Wynika z tego, że obrazy posiadają strukturę warstwową. Dokładniej rzecz ujmując, cały obraz składa się z tzw. manifestu oraz kolejnych warstw. Ilustruje to rysunek poniżej.

Manifest jest opisem (w formacie json) konfiguracji całego obrazu oraz poszczegolnych jego warstw.

UWAGA: Poszczególne warstwy obrazu są „nieświadome” istnienia pozostałych warstw co w praktyce oznacza, że nie ma żadnych referencji jednej warstwy do drugiej (są one całkowicie niezależne od siebie).

Warstwy powstają w wyniku kolejnych działań, które mają doprowadzić do zapewnienia dostępności niezbędnych komponentów programowych dla aplikacji uruchomianej w kontenerze opartym o ten obraz. Zatem przyporządkowania zawartości do poszczególnych warstw zazwyczaj odpowiada temu, które przedstawione jest na poniższym rysunku.

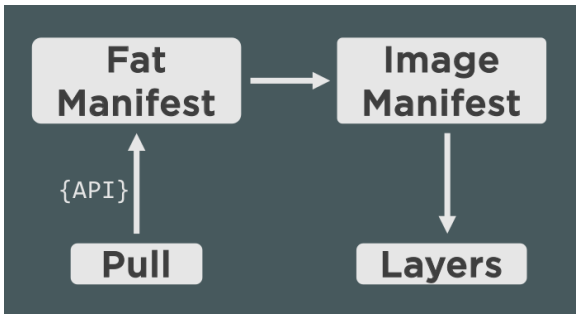


Podstawową warstwą (ang. base image) jest bazowy obraz z wybranym systemem operacyjnym (np. Debian, Ubuntu, Fedora, Redis itd.) Dodatkowe komponenty są dodawane jako kolejne warstwy. Każda modyfikacja obrazu (każde dodanie narzędzi, zasobów, bibliotek itd.) prowadzi do powstania nowej warstwy. Jednocześnie wszystkie warstwy wcześniej utworzone, nie ulegają zmianie czy też usunięciu. Przykładowo, używane na końcu poprzedniego ćwiczenia, polecenie „docker commit” prowadziło do powstania nowego obrazu. Jednocześnie jeżeli w obrazie bazowym

zostały wprowadzone zmiany, to ten nowo-powstały obraz uwzględnia je jako kolejne warstwy, nad warstwą bazową. Zostanie to szczegółowiej omówione w dalszej części laboratorium.

UWAGA: Wszystkie warstwy z wyjątkiem warstwy najwyższej (zazwyczaj bezpośrednio powiązaną z aplikacją uruchamianą w kontenerze) mają atrybut RO (ang. read-only). Warstwa najwyższa natomiast jest warstwą RW (ang. read-write).

W trakcie pobierania obrazu z danego repozytorium (np. z poznanego Docker Hub) analizowane a następnie pobierane są poszczególne składniki obrazu. W obecnej wersji środowiska Docker CE można to przedstawić następująco:



Wydanie polecenia „docker pull <nazwa obrazu> powoduje zainicjowanie zapytania (za pośrednictwem API repozytorium) czy istnieje wersja obrazu przeznaczona dla architektury sprzętowej, na której zainstalowany jest demon Docker (np. x86-64, arm itd.). Deamon Docker korzysta tu z informacji zawartych w swojej konfiguracji (dla przypomnienia, można ją wyświetlić za pomocą polecenia „Docker info”). Przykładowa deklaracja wymaganej architektury jest przedstawiona poniżej.

```
student@student-PwCh0:/var/lib/docker$ docker info |grep Architecture
WARNING: No swap limit support
Architecture: x86_64
```

UWAGA: Powyższe informacje oznaczają, że jest możliwe przygotowanie obrazu Docker dla różnych architektur sprzętowych, a polecenie „Docker pull” pobierze właściwą wersję dla danego urządzenia.

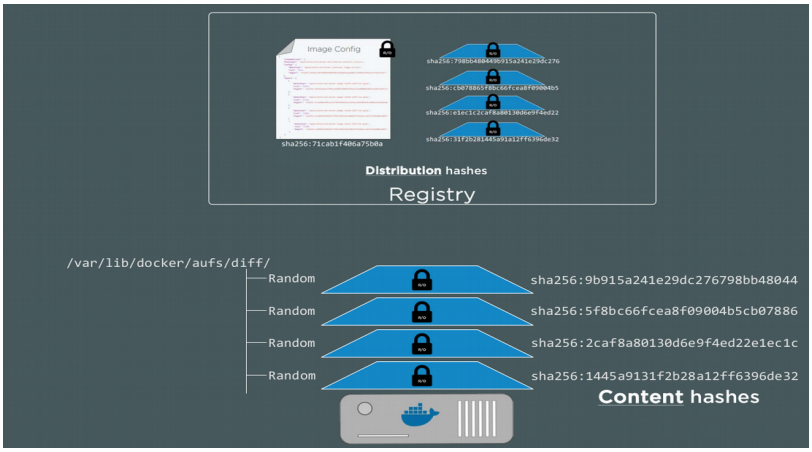
Zapytanie jest analizowane w repozytorium w odniesieniu do tzw. pełnego manifestu (ang. fat manifest). Jeżeli istnieje informacja o wsparciu dla architektury z zapytania, pobierany jest manifest obrazy (ang. image manifest). Na podstawie jego opisu, pobierane są kolejno warstwy obrazu i rozpakowywane w lokalnym systemie plików.

Ponieważ obrazy są pobierane zazwyczaj ze zdalnych lokalizacji (zdalnych repozytoriów) to kluczowym jest zapewnienie, że to do otrzymano jest tym obrazem, o który proszono wydając polecenie „Docker pull”. W skrócie, w tym celu wykorzystuje się narzędzia kryptograficzne. Ilustruje to rysunek ponizej.

Dla manifestu jak i dla poszczególnych warstw dostępna jest wartość funkcji skrótu (and. digest). Wartość ta jest porównywana z wartościami hash-y wyliczonych na podstawie pobranych i rozpakowanych komponentów obrazu.

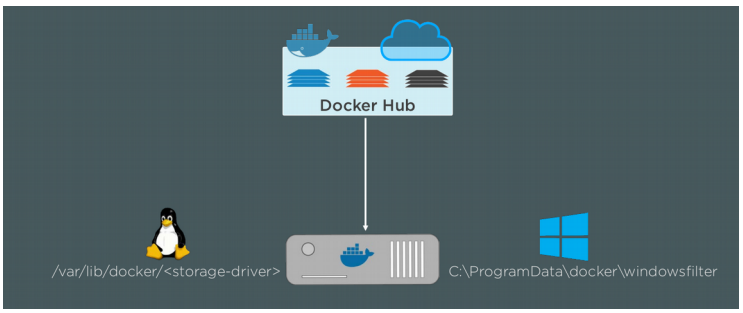
UWAGA: Digest dla obrazu jest wartością hash-a dla manifestu.

W praktyce sytuacja jest bardziej skomplikowana. Po pierwsze, wysyłając obraz do repozytorium lub pobierając go, operuję się wersjami skompresowanymi. Wobec tego, z oczywistych przyczyn, wartość hash-y dla elementów obrazu przed i po skompresowaniu są różne. Z tego powodu wyróżnia się tzw. *content hashes* (hashe dla nieskompresowanych warstw i manifestu) oraz *distribution hashes*, które odnoszą się do spakowanych składników obrazu, przechowywanych w repozytoriach. Ilustruje to rysunek poniżej.



Po drugie, pobrane i rozpakowane komponenty obrazu są zapisywane w lokalnym systemie plików. Tam również stosowane są funkcje skrótu.

Miejsce przechowywania obrazów zależy od systemu operacyjnego oraz od wykorzystywanego sterownika wirtualnego systemu plików (ang. storage driver) dla środowiska Docker. W przypadku systemów opartych o Linux, dwa obecnie wykorzystywane sterowniki to *aufs* oraz *overlay2* z tym, że ten drugi jest rozwiązaniem nowszym i zdecydowanie przyszłościowym. Ponownie, nazwę wykorzystywanego sterownika można odczytać na podstawie wyniku działania polecenia „docker info”. Domyślne miejsca przechowywania warstw obrazu przedstawia rysunek poniżej.



Mechanizmy zaimplementowane w środowisku Docker powodują, że przy uruchamianiu kontenera, wszystkie warstwy są łączone w jeden system plików, widziany z wnętrza tego kontenera.

A. Funkcjonowanie opisanych wyżej mechanizmów można zaobserwować w praktyce. Przykładowo pobierzmy obraz dla Redis. Przy okazji, uściślijmy nazewnictwo stosowane przy pobieraniu (i wysyłaniu) obrazów. Zgodnie z dokumentacją Docker, poprawny schemat jest następujący.

Obecnie korzystamy z punlicznego repozytorium Docker Hub (budowa własnego registry będzie jeszcze przedmiotem oddzielnego omówienia) wobec czego można pominąć składową nazwy <registry>. Dalej podawane jest repozytorium (repo). Dopiero trzecią częścią jest nazwa obrazu. W potocznym języku nazywana jest znacznikiem (ang. tag) a obraz (image) tym co według standardu jest skladową nazwy <repo>. Należy pamiętać o tej nieścisłości pomimo że dalej wykorzystywać będziemy potoczną nomenklaturę.

```
student@student-PwChO:~$ docker pull redis
Using default tag: latest
latest: Pulling from library/redis
f17d81b4b692: Pull complete
b32474098757: Pull complete
8980cabe8bc2: Pull complete
f2ceb2908b0a: Pull complete
4330a6e801e5: Pull complete
ac88bc721c38: Pull complete
Digest: sha256:b6977d3ce8c7ee5c1a59146cd58912b5945b8b5ecf887ef2f3d1b2e402387f4e
Status: Downloaded newer image for redis:latest
student@student-PwChO:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
redis               latest              f1897cdc2c6b        2 days ago          83.4MB
student@student-PwChO:~$
```

Pobrany obraz posiada 6 warstw. Dodatkowo, po wypisaniu dostępnych lokalnie obrazów, wyświetlony został obraz redis i jego Image ID. Aby przyjrzeć się bliżej strukturze tego obrazu należy użyć polecenia:

$ docker images –digests

```
student@student-PwChO:/var/lib/docker$ docker images --digests
REPOSITORY          TAG                 DIGEST                                                                    IMAGE ID            CREATED             SIZE
redis               latest              sha256:b6977d3ce8c7ee5c1a59146cd58912b5945b8b5ecf887ef2f3d1b2e402387f4e   f1897cdc2c6b        2 days ago          83.4MB
student@student-PwChO:/var/lib/docker$
```

W wyniku tego polecenia można odczytać tak wartość DIGEST jak i IMAGE ID. Są one różne. Wyjaśnienie można znaleźć, analizując wynik działania kolejnego polecenia:

$ docker image inspect <nazwa obrazu>

Z początkowej części wyniku działania tego polecenia można odczytać ID manifestu, którego początek odpowiada poprzednio wyświetlonemu IMAGE ID oraz RepoDigest, które jest takie samo jak wartość DIGEST w wyniku poprzedniogo polecenia.

```
student@student-PwChO:/var/lib/docker$ docker image inspect redis
[
    {
        "Id": "sha256:f1897cdc2c6b41217b73899298717e81d10dda561cdd4939dfb7ecbaf35b4b94",
        "RepoTags": [
            "redis:latest"
        ],
        "RepoDigests": [
            "redis@sha256:b6977d3ce8c7ee5c1a59146cd58912b5945b8b5ecf887ef2f3d1b2e402387f4e"
        ],
```

Przechodząc do końcowej części wyświetlonych informacji, można odczytać sposób składania warstw za pomocą wykorzystywanego sterownika wirtualnego systemu plików (sekcje DATA oraz

NAME). Niżej natomiast podane są hash-e dla warstw obrazu, przechowywanych w lokalnym systemie plików (sekcja RootFS).

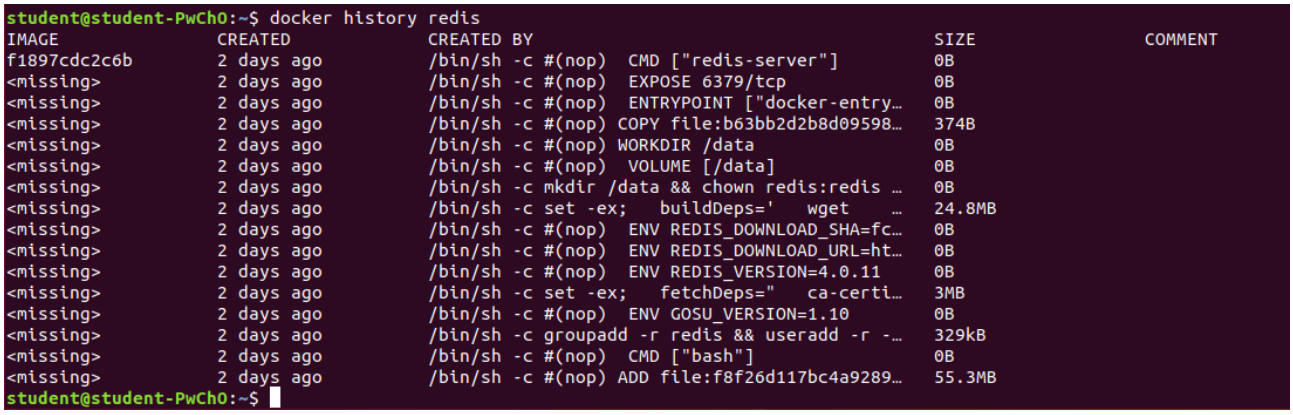
Przechodząc do katalogu (w naszym przykładzie): *var/lib/docker/overlay2* można zobaczyć podkatalogi z danymi powiązanymi z poszczególnymi warstwami (KONIECZNE prawa root-a)

W poszczególnych podkatalogach są katalogi o nazwie „diff", które zawierają dane przypisane dla danej warstwy.

UWAGA: Prezentowany przykład jest dla sterownika overlay2. Dla aufs struktura katalogów jest nieznacznie równa ale zgodna z prezentowanym schematem rozumowania.

Operowanie na lokalnym systemie plików nie jest szczególnie wygodne. Aby zapoznać się z tym co zawierają poszczególne warstwy należy poznać historię modyfikacji/uzupełnień obrazu bazowego (zgodnie z informacjami wyżej, każde dodanie do obrazu narzędzia/zasobu pociąga za sobą utworzenie nowej warstwy). Pomocne w tym zakresie jest polecenie:

$ docker history <nazwa obrazu>



Zgodnie z wcześniejszymi danymi o obrazie Redis, wiadomo, że posiada on 6 warstw. Na powyższym zrzucie ekranowym można przeanalizować, w wyniku jakiś działań zostały one utworzone. Pierwsza od dołu to system bazowy, który utworzył warstwę bazową. Kolejne warstwy

są wynikiem: modyfikacji użytkowników i grup, dalej wygenerowania certyfikatów, pobrania (prawdopodobnie bibliotek) za pomocą wget, utworzenia niezbędnej struktury katalogowej i wreszcie zainstalowania właściwej aplikacji.

B. Modyfikacja obrazów bazowych jest najprostszą formą tworzenia własnych obrazów (konteneryzowania aplikacji).

2Z1. Proszę wykonać ciąg następujących poleceń:

1. Pobrać obraz bazowy ubuntu:latest i sprawdzić czy jest dostępny lokalnie.
2. Uruchomić kontener na bazie tego obrazu z opcjami -i oraz -t oraz uruchamianiem powłoki bash.
3. Z poziomu powłoki zaktualizować bazę repozytoriów ubuntu (polecenie: apt-get update).
3. Z poziomu powłoki zainstalować pakiet wget.
4. Z poziomu powłoki sprawdzić czy wget jest w systemie (polecenie which wget)
5. Otworzyć drugi pseudo-terminal na macierzystym systemie ubuntu.
    a. Sprawdzić w pomocy środowiska Docker do czego służy polecenie

    $ docker diff <nazwa/kontener ID>

    b. W tym terminalu uruchomić to polecenie w stosunku do modyfikowanego kontenera (w sprawozdaniu umieścić tylko początek rezultatu działania tego polecenia).
6. Zamknąć drugi terminal i zatrzymać zmodyfikowany kontener. Za pomocą polecenia „docker commit" utworzyć obraz ze zmodyfikowanego kontenera i nazwać go „mutant".
7. Sprawdzić dostępność w systemie lokalnym i dane nowego kontenera „mutant".
8. Sprawdzić czy kontener posiada warstwy. Wykorzystując narzędzia z punktu A tej instrukcji, przeanalizować położenie i zawartość poszczególnych warstw.

Wszystkie punkty powinny być w sprawozdaniu zilustrowane: użytym poleceniem, wynikiem jego działania oraz (tam gdzie to celowe, komentarzem).

C. Do tej pory wszystkie obrazy były pobierane z Docker Hub. Na kolejnych laboratoriach zostanie rozwinięty temat repositories oraz registry i metod korzystania z nich. W tej części przedstawione zostanie najprostsze z rozwiązań, lokalne registry. Na początku jednak warto zdefiniować oba, wymienione wyżej pojęcia.

Registry jest miejscem gdzie obrazy mogą być rejestrowane, indeksowane i udostępniane tak publicznie jak i prywatnie. Repository jest oznaczeniem magazynu przechowywania obrazów zarejestrowanych w Registry.

W chwili obecnej, registry może być zaimplementowane przez każdego użytkownika wykorzystując dedykowany obraz Docker. Obraz ten jest dostępny na Docker Hub https://hub.docker.com/_/registry/ . Proszę zapoznać się z umieszczonym tam opisem. Instalacja registry (w przykładzie w wersji 2) polega na wydaniu poniższych poleceń:

```
student@student-PwCh0:/$ docker pull registry:2
2: Pulling from library/registry
d6a5679aa3cf: Pull complete
ad0eac849f8f: Pull complete
2261ba058a15: Pull complete
f296fda86f10: Pull complete
bcd4a541795b: Pull complete
Digest: sha256:5a156ff125e5a12ac7fdec2b90b7e2ae5120fa249cf62248337b6d04abc574c8
Status: Downloaded newer image for registry:2
student@student-PwCh0:/$ docker run -d -p 5000:5000 --name registry registry:2
ec88992a2e4b91a4e51bd4ac2aa48993ea3d081b1884b8130044b0caf00d375a
student@student-PwCh0:/$
```

W wyniku tych poleceń pobrany został obraz registry:2 z registry docker.io oraz uruchomiony kontener, który nazwano registry. Ilustruje to rysunek poniżej.

```
student@student-PwCh0:/$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mutant              latest              e4bc25ff778d        21 minutes ago      134MB
redis               latest              f1897cdc2c6b        2 days ago          83.4MB
registry            2                   2e2f252f3c88        5 weeks ago         33.3MB
ubuntu              latest              cd6d8154f1e1        6 weeks ago         84.1MB
student@student-PwCh0:/$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
ec88992a2e4b        registry:2          "/entrypoint.sh /etc…"   2 minutes ago       Up 2 minutes        0.0.0.0:5000->5000/tcp   registry
student@student-PwCh0:/$
```

Operacje przesyłania i pobierania obrazów z dowolnego registry realizowane są poprzez dwa polecenia:

$ docker push <nazwa obrazu>

$ docker pull <nazwa obrazu>

UWAGA: W środowisku Docker (w środowisko lokalnym) nie mogą występować dwa obrazy o tej samej nazwie. W związku z tym, przed przystąpieniem do przesłania obrazu z lokalnego systemu do lokalnego registry należy zmienić nazwę (nadać nowy tag) obrazowi. Analogicznie, pobierając z lokalnego registry do środowiska lokalnego Docker, należy upewnić się, czy lokalnie nie ma obrazu o takiej nazwie a jeśli tak, to usunąć go przed pobraniem innego z tą nazwą z lokalnego registry.

Zakładając, że lokalnie dostępny jest obraz redis:latest, proces przesłania go do wcześniej uruchomionego lokalnego registry (kontener registry) jest następujący:

```
student@student-PwCh0:/$ docker tag redis localhost:5000/redis.local
student@student-PwCh0:/$ docker push localhost:5000/redis.local
The push refers to repository [localhost:5000/redis.local]
d6978bcd145f: Pushed
18b1fd6dab51: Pushed
104d8d223984: Pushed
aa1a19279a9a: Pushed
197ffb073b01: Pushed
237472299760: Pushed
latest: digest: sha256:616fdab51679e064fca66537cf77292c6dfe43d3d9a26bf83973438043a2dd33 size: 1571
student@student-PwCh0:/$ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
mutant                      latest              e4bc25ff778d        27 minutes ago      134MB
redis                       latest              f1897cdc2c6b        2 days ago          83.4MB
localhost:5000/redis.local  latest              f1897cdc2c6b        2 days ago          83.4MB
registry                    2                   2e2f252f3c88        5 weeks ago         33.3MB
ubuntu                      latest              cd6d8154f1e1        6 weeks ago         84.1MB
student@student-PwCh0:/$
```

„localhost:5000” to nazwa „naszego” lokalnego registry a „redis.local” to nowa nazwa obrazu. Proszę zwrócić uwagę, że w rezultacie w systemie dostępne są dwa obrazy: „redis” i „localhost:5000/redis.local” o tym samym IMAGE ID (innymi słowy, są to te same obrazy). Obrazy są przechowywane w dedykowanym wolumenie, który jest zmapowany na lokalny system plików. Proszę przejrzeć strukturę domyślnego katalog: */var/lib/docker/volumes* . (ponownie KONIECZNE uprawnienia root-a)

Proszę zapoznać się z dokumentacją wykorzystania obrazu registry na Docker Hub https://docs.docker.com/registry/deploying/ . Szczególnie przydatne będą sekcje: „Copy an image from Docker Hub to your registry” oraz „Stop a local registry”.

2Z2. Na podstawie przedstawionego opisu i dokumentacji Docker (KONIECZNIE uwzględniając uwagę powyżej) należy:

1. Pobrać i uruchomić kontener registry w najnowszej wersji.
2. Wgrać do niego obraz o nazwie mutant (utworzony w wyniki poprzedniego zadania) i nadać mu tag „nowy”.
3. Pobrać z lokalnego registry obraz „localhost:5000/mutant.nowy” i sprawdzić jego dostępność w środowisku lokalnym.
4. Zatrzymać kontener. Usunąć go oraz usunąć obraz „registry”.

Wszystkie punkty powinny być w sprawozdaniu zilustrowane: użytym poleceniem, wynikiem jego działania oraz (tam gdzie to celowe, komentarzem).

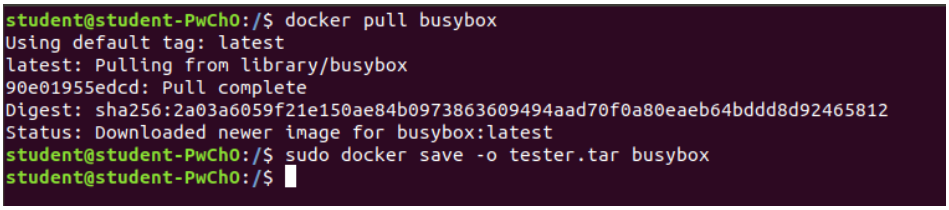
Wymiana obrazów pomiędzy np. członkami większej grupy programistów jest najprostsza również dzięki publicznym i prywatnym registry. Tym niemniej warto wiedzieć o bardzo prostej metodzie przesyłania obrazów jako skompresowanych plików. Podstawowe polecenia wykorzystywane w tej metodzie to:

$ sudo docker save <nazwa obrazu>

$ sudo docker load <nazwa pliku.tar>

Plik powstały w wyniku pierwszego z poleceń (domyślnie wykorzystywane jest narzędzie tar) można przenosić pomiędzy różnymi środowiskami Docker i wgrywać za pomocą polecenie drugiego (pod warunkiem zgodności architektur, dla których obraz jest przeznaczony).

Tworzenie pliku z obrazu można zilustrować więc poniższym schematem:



Proszę zapoznać się z dokumentacją wymienionych wyżej dwóch poleceń.

UWAGA: przydatne opcje: dla „docker save” -o <nazwa pliku.tar> pozwalająca nadać własną nazwę plikowi wynikowemu oraz dla „docker load” - i <nazwa pliku.tar> wskazująca, który plik zaimportować jako obraz.

2Z3. Proszę wykonać następujące zadania i odpowiedzieć na pytania:

1. Pobrać z Docker Hub obraz „busybox” w najnowszej wersji.
2. Zapisać ten obraz jako plik „alfa.tar”.
    a. Dlaczego przy wykonywaniu tego polecenia niezbędne były uprawnienia root-a i gdzie domyślnie utworzony plik „alfa.tar” został zapisany ?
    b. Czy ten plik można samodzielnie rozpakować a jeśli tak to co on zawiera ?
3. Usunąć obraz „busybox” z lokalnego srodowiska Docker.
3. Załadować obraz „busybox” z pliku „alfa.tar” i sprawdzić czy jest on dostępny w lokalnym środowisku Docker.

Wszystkie punkty powinny być w sprawozdaniu zilustrowane: użytym poleceniem, wynikiem jego działania oraz należy krotko odpowiedzieć na pytania 2a oraz 2b.

D. Przedstawiona w punkcie B metoda budowania własnych obrazów jest przydatna tylko i wyłącznie w najprostszych sytuacjach. Domyślnym sposobem opisu obrazu Docker jest skorzystanie z dedykowanego skryptu. Jest on zawsze reprezentowany w środowisku Docker jako plik o nazwie „Dockerfile”. Budowa i wykorzystanie tego pliku będzie tematem jednego z następnych laboratoriów. W tym miejscu przedstawione są podstawowe (wprowadzające) informacje.

UWAGA: Pełna dokumentacja składni i sposobu wykorzystania Dockerfile jest dostępna pod adresem: https://docs.docker.com/engine/reference/builder/

Opis budowy obrazu umieszczony jest w pliku Dockerfile i można w nim znaleźć
ciąg następujących komend (zgodnia z punktem A tej istrukcji, każda z nich będzie odpowiedzialna za utworzenie nowej warstwy obrazu):

FROM <image> – tą komendą musi zaczynać się każdy Dockerfile, określa ona inny obraz jako warstwę bazową nowo tworzonego. Dobrą praktyką jest korzystanie (w miarę możliwości) tylko z oficjalnych repozytoriów .

FROM <image>:<tag> – po dwukropku możemy określić dokładnie którą wersję chcemy pobrać. Pominięcie <tag> skutkuje użyciem wersje najnowszej (latest).

FROM <image>@<digest> – dla obrazów wersji drugiej (v2) lub wyższej możemy po małpie możemy podać skrót sha256 obrazu.

Dockerfile file może zawierać kilka instrukcji FROM, każda definiuje osobny etap budowania obrazu.

FROM <image> AS <nazwa> – przypisuje nazwę aktualnemu etapowi budowanego obrazu, może być użyta w instrukcjach FROM oraz COPY aby odnieść się do obrazu zbudowanego na tym etapie, przykładowo instrukcja:

COPY --from=<name|index> <targ_path> <dest_path>

wywołana z innego etapu kopiuje pliki do katalogu <dest_path> bieżącego etapu z katalogu <targ_path> etapu o nazwie bądź indeksie <name|index>.

LABEL – etykieta to unikalna w obrębie obiektu Dockera para stringów klucz-wartość. Obiektem może być obraz, kontener, lokalny demon itp..., i może zawierać dowolną liczbę etykiet. Służą m.in. do porządkowania obrazów, zapisu informacji licencyjnych oraz opisu powiązań między kontenerami, woluminami i sieciami. Pełna dokumentacja wykorzystania pod adresem: https://docs.docker.com/config/labels-custom-metadata/

LABEL maintainer="email@domena" – informacja na temat osoby lub grupy opiekującej się danym kontenerem.
UWAGA: Wcześniej była to instrukcja MAINTAINER <name> (aktualnie przestarzała).

RUN <command> – uruchomienie wybranej komendy systemowej już w nowym systemie kontenera. Polecenie uruchamiane jest w powłoce, domyślnie /bin/sh –c (Linux) albo cmd /S /C (Windows).

RUN ["executable", "param1", "param2"] – uruchamiany jest (bez udziału powłoki) plik wykonywalny z podanymi argumentami.

COPY – kopiuje lokalne pliki do kontenera.

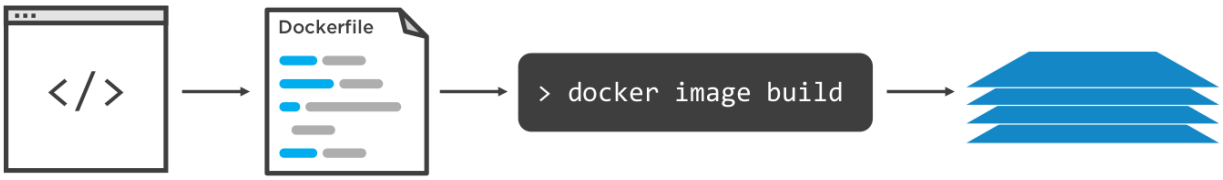
CMD – definiuje polecenia, które będą uruchamiane w obrazie podczas uruchamiania. W przeciwieństwie do RUN, nie tworzy to nowej warstwy obrazu, ale po prostu uruchamia polecenie. W każdym pliku Dockerfile może występować tylko jedna komenda CMD. Aby uruchomić wiele poleceń, najlepszym sposobem jest uruchomienie skryptu przez CMD. CMD wymaga podania informacji, gdzie można uruchomić polecenie, w przeciwieństwie do RUN. Przykłady komend CMD:

CMD ["python", "./app.py"]
CMD ["/ bin / bash", "echo", "Hello World"]

EXPOSE – konfiguracja nasłuchiwania na wybranych portach.

VOLUME <współdzielony system_plików> – pozwala na mapowanie katalogów systemu operacyjnego hosta.

Idee wykorzystania pliku Dockerfile można zilustrować rysunkiem jak poniżej.



Posiadając kod aplikacji i wiedząc jak ma ona współpracować z otoczeniem, należy określić bazowy system oraz niezbędne komponenty programowe. Następnie opisać sposób budowania kontenera za pomocą pliku Dockerfile. Ten plik jest argumentem dla narzędzia automatycznego budowania obrazów w postaci polecenia:

$ docker build [options] PATH | URL | -

UWAGA: W większości wypadków zakłada się, że plik Dockerfile jest w tym samym miejscu drzewa katalogowego (w tym katalogu), z którego wydawane jest polecenie „docker build”. Jeżeli tak nie jest, należy podać scieżkę do tego pliku lub URL i ściezkę w przypadku korzystania ze zdalnego systemu plików.

Najprostszą prezentację tej metody budowania obrazów jest utworzenie w katalogu domowym pliku Dockerfile o następującej treści:

```
student@student-PwCh0:~$ cat Dockerfile
FROM busybox:latest
# komentarze sa zawsze mile widziane
CMD echo Docker is a nice and easy
```

Następnym krokiem jest wydanie polecenia:

$ docker build .

```
student@student-PwCh0:~$ docker build .
Sending build context to Docker daemon  50.04MB
Step 1/2 : FROM busybox:latest
 ---> 59788edf1f3e
Step 2/2 : CMD echo Docker is a nice and easy
 ---> Running in 46bf52bd829d
Removing intermediate container 46bf52bd829d
 ---> cd04e75d331d
Successfully built cd04e75d331d
student@student-PwCh0:~$
```

Przebieg budowania tego najprostszego obrazu informuje nie tylko o wykonywanych etapach ale również jak tworzone są poszczególne warstwy obrazu. Na początku opis zawarty w Dockerfile jest kierowany do lokalnego środowiska Docker (możliwe jest też budowanie z wykorzystaniem środowisk zdalnych). W kroku 1 budowana jest warstwa bazowa z obrazu „busybox:latest”. Następnie tworzony jest tymczasowy kontener o ID 46bf52bd829d a w nim realizowana jest komenda CMD. Jeśli brak jest błędów, tymczasowy kontener jest usuwany a ponieważ CMD nie tworzy nowej warstwy to wynik jest łączony z warstwą bazową. Można też sprawdzić dostępność utworzonego obrazu.

```
student@student-PwCh0:~$ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
<none>                      <none>              cd04e75d331d        9 minutes ago       1.15MB
mutant                      latest              e4bc25ff778d        3 hours ago         134MB
redis                       latest              f1897cdc2c6b        2 days ago          83.4MB
localhost:5000/redis.local  latest              f1897cdc2c6b        2 days ago          83.4MB
busybox                     latest              59788edf1f3e        2 weeks ago         1.15MB
registry                    2                   2e2f252f3c88        5 weeks ago         33.3MB
ubuntu                      latest              cd6d8154f1e1        6 weeks ago         84.1MB
student@student-PwCh0:~$ docker tag cd04e75d331d winner
student@student-PwCh0:~$ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
winner                      latest              cd04e75d331d        11 minutes ago      1.15MB
mutant                      latest              e4bc25ff778d        3 hours ago         134MB
redis                       latest              f1897cdc2c6b        2 days ago          83.4MB
localhost:5000/redis.local  latest              f1897cdc2c6b        2 days ago          83.4MB
busybox                     latest              59788edf1f3e        2 weeks ago         1.15MB
registry                    2                   2e2f252f3c88        5 weeks ago         33.3MB
ubuntu                      latest              cd6d8154f1e1        6 weeks ago         84.1MB
student@student-PwCh0:~$
```

Utworzony obraz posiada IMAGE ID ale nie posiada nazwy. Polecenie:

$ docker tag <IMAGE ID> <nowa nazwa>

pozwala nadać nową nazwę temu obrazowi. Można też (i jest to prostsze rozwiązanie) nadać nazwę obrazowi w trakcie jego budowania:

$ docker build -t <nowa nazwa> .

Na koniec można sprawdzić, że obraz jest rzeczywiście jednowarstwowy poprzez wydanie polecenia:

$ docker inspect winner

W końcowej części wyniku można znaleźć informacje jak ilustruje rysunek poniżej.

2Z4. Proszę utworzyć plik Dockerfile a następnie zbudować na jego podstawie obraz o nazwie „web100” przy spełnieniu następujących założeń:

a. Obraz bazowy to ubuntu w najnowszej wersji.
b. Jako autora podać swoje nazwisko i email.
c. System ubuntu ma byż zauktualizowany
d. Zainstalowany ma być serwer HTTP Apache w najnowszej wersji.
e. Serwer ma nasłuchiwać na porcie 8080.

1. Na podstawie przebiegu procesu budowy proszę opisać ile posiada on warstw i w wyniku jakich działań one powstały.

2. Proszę potwierdzić spostrzeżenia z punktu 1 za pomocą polecenia
$ docker inspect ……..
$ docker history ……

3. (zadanie dodatkowe). Prosze uruchomić kontener „web100” w taki sposób aby możliwe było uruchomienie przeglądarki internetowej (w systemie ubuntu) i połączenie się z serwerem Apache umieszczonym w działającym kontenerze.