

LABORATORIUM PROGRAMOWANIA W CHMURACH OBLICZENIOWYCH

LABORATORIUM NR 1.

Wprowadzenie do środowiska Docker. Kontenery Docker.

Docker, jest nazywany kontenerem, ale właściwie powinno się myśleć o nim jako jednej z wielu technik konteneryzacji. Reprezentuje on lekkie, przenośne rozwiązanie kontenera programowego (nie tylko systemów Linux ale również Windows), które realizuje idee "build once, configure once and run anywhere". Stronę domową projektu można znaleźć pod adresem <https://www.docker.com/>. Zgodnie z definicją umieszczoną na tej stronie, Docker jest otwartą platformą do budowy, udostępniania i uruchamiania aplikacji rozproszonych. To daje programistom, zespołom programistów i inżynierom środowisk chmurowych wspólny zestaw narzędzi, niezbędnych, aby budować i uruchamiać usługi i/lub aplikacje w dowolnej sieci rozproszonej lub dowolnej chmurze komputerowej.

Docker posiada cechy, które odnaleźć można również w standardowych mechanizmach wirtualizacji czy też kontenerach LXC/LXD (<https://linuxcontainers.org/>). Jednocześnie, co jest niezwykle istotne z punktu widzenia tematyki laboratorium, Docker pozwala na:

- uniezależnienie aplikacji od zależności, które programiści typowo muszą brać pod uwagę przy budowaniu aplikacji na danej platformie systemowej,
- tworzenie obrazów aplikacji (odpowiednik obrazów systemów) i systemu zarządzania tymi obrazami,
- aplikacje w Docker są gotowe do uruchomienia w dowolnym środowisku obsługującym Docker (obecnie wszystkie popularne systemy operacyjne i środowiska chmurowe, tak komercyjne jak i open source),
- zapewnienie ciągłości rozwoju aplikacji polegającej na możliwości testowania nowych wersji aplikacji bez wpływu na prace wersji pierwotne, prosta zmiana wersji, możliwość korzystania z kilku wersji aplikacji równolegle.

Z punktu widzenia programisty, Docker można być wykorzystywane na trzy różne sposoby (najczęściej korzysta się ze wszystkich trzech choć nic nie stoi na przeszkodzie by specjalizować się w jednej czy dwóch):

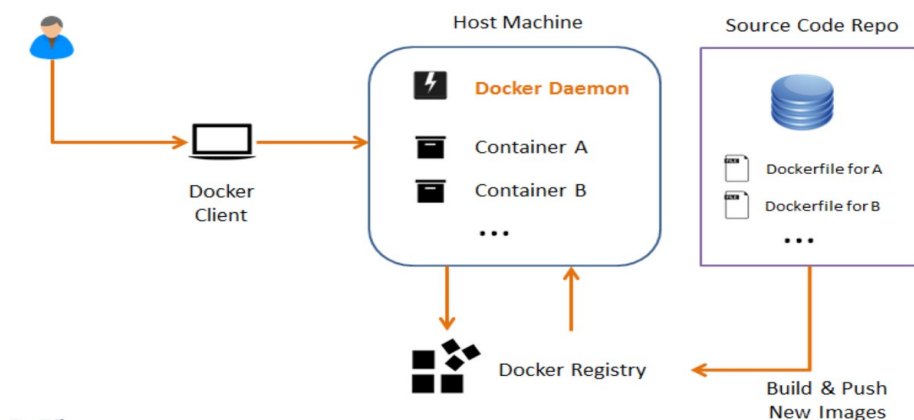
- opracowanie kodu aplikacji, utworzenie obrazu aplikacji (ang. application image), dystrybucja obrazów,
- tworzenia kontenera na podstawie obrazów, testowanie aplikacji, dystrybucja kontenerów, tworzenie kolejnych wersji oraz dystrybucja kontenerów,
- budowa końcowej usługi i/lub w oparciu o konfigurację (statyczna lub dynamiczna) zestawu programów (dostępnych w kontenerach) i przyporządkowanie im zasobów określonego środowiska uruchomieniowego (np. określonego środowiska chmurowego).

Ostatni punkt jest podstawą schematu programowania dla chmur komputerowych, które określane jest jako podejście mikro-usługowe (ang. microservicing). Schemat ten będzie wiodącą przesłanką dla realizacji kolejnych ćwiczeń ale już teraz student powinien zapoznać się z podstawowymi cechami tego paradygmatu tworzenia aplikacji i usług przeznaczonych dla środowisk chmurowych np. artykuł dostępny pod adresem <https://martinfowler.com/articles/microservices.html> a dla preferujących wykłady wideo: <https://www.youtube.com/watch?v=wgdBVIX9ifA>

Kontenery Docker wykorzystują standardowe narzędzia oferowane przez środowiska Unix, które zapewniają bezpieczeństwo i efektywność tej metody lekkiej wirtualizacji, Do najważniejszych należą:

- Namespaces – stanowią pierwszy poziom izolacji kontenerów Docker. Dzięki ich wykorzystaniu procesy uruchomione w danym kontenerze nie mogą “widzieć” efektów pracy procesów działających w innych kontenerach,
- Control Groups – kluczowy element tak kontenerów LXC jak Docker. Pozwala na zarządzanie dostępem do zasobów przypisanych danemu kontenerowi,
- FileSystem – poza standardowymi zadaniami, w przypadku kontenerów Docker pozwala on na tworzenie warstw w ramach kontenera co pozwala np. na zarządzania wersjami oprogramowania.

Cale środowisko developerskie oparte o kontenery Docker przedstawione jest schematycznie na rysunku poniżej.



Poszczególne elementy składowe to:

- Docker Client – jest to interfejs użytkownika pozwalający na komunikację pomiędzy użytkownikiem (również programistą) a daemon-em Docker.
- Docker Daemon – podstawowy “silnik” obsługi kontenerów Docker,
- Docker Registry – rejestr służący zarządzaniu kopiami kontenerów Docker z możliwością definiowania statusu tych kopii (tj. dostęp publiczny lub prywatny czyli ograniczony),
- Docker Container – kompletne, wyizolowane środowisko działania danej aplikacji/mikrouслуги, oparte na określonym systemie operacyjnym i zdefiniowane przez pliki konfiguracyjne i zestaw meta-danych,
- Docker Image – inaczej nazywane “ready-only templates” zawiera tak aplikacje jak i niezbędne komponenty do ich uruchomienia w kontenerach Docker,
- DockerFile – plik przechowujący instrukcje automatycznego budowania obrazu kontenera.

W kolejnych laboratoriach omówione zostaną związki pomiędzy poszczególnymi elementami środowiska Docker i sposób ich wykorzystywania. Zanim to nastąpi, proszę samodzielnie zapoznać się z relacjami pomiędzy składowymi tego środowiska. Wsparciem mogą być prezentacje Docker, np.:

- wprowadzenie do Docker - <https://www.youtube.com/watch?v=Q5POuMHxW-0>
- wykład jednego z deweloperów Docker prezentujący genezę powstania i rozwoju środowiska Docker <https://www.youtube.com/watch?v=FdkNAjJO5yQ>

Zadania. Instalacja i podstawowa obsługa kontenerów Docker

A. W chwili obecnej docker jest dostępny w dwóch wersjach CE (ang. Community Edition) oraz wersjach płatnych (ang. Enterprise Edition). Ze szczegółowym opisem struktury i zasad funkcjonowania środowiska Docker jak i porównaniem jego implementacji można zapoznać się pod adresami: <https://docs.docker.com/get-started/overview/> oraz <https://docs.docker.com/get-started/> W kolejnych laboratoriach będziemy wykorzystywali wersję CE. Jest ona zainstalowana w udostępnionym obrazie maszyny wirtualnej Ubuntu 20.04, dostępnej na Moodle jak i platformie MSO365.

Uwaga1: Zainstalowane środowisko Docker CE zostało skonfigurowane tak by wszystkie podstawowe polecenia były możliwe do wykonania z uprawnieniami użytkownika „student” oraz skonfigurowano wykorzystywanie zewnętrznych serwerów DNS aby uniknąć ewentualnych konfliktów z usługami w sieci lokalnej. Inne, typowe konfiguracje środowiska można znaleźć pod adresem: <https://docs.docker.com/engine/install/linux-postinstall/>

Uwaga2: W dalszej części tej i kolejnych instrukcji ilustracje poszczególnych operacji wykonywanych w środowisku Docker pochodzą będą ze środowiska Linux. Odniesienia do środowiska Windows będą zamieszczane wyłącznie, gdy będzie to niezbędne.

Po pierwszym uruchomieniu maszyny wirtualnej bądź każdorazowo, w przypadku zauważenia problemów warto sprawdzić ustawienia środowiska Docker wykorzystując polecenia:

`$ docker version (lub $ docker --version)`

```
student@PwCh0-VB:~$ docker version
Client: Docker Engine - Community
Version: 19.03.13
API version: 1.40
Go version: go1.13.15
Git commit: 4484c46d9d
Built: Wed Sep 16 17:02:52 2020
OS/Arch: linux/amd64
Experimental: false

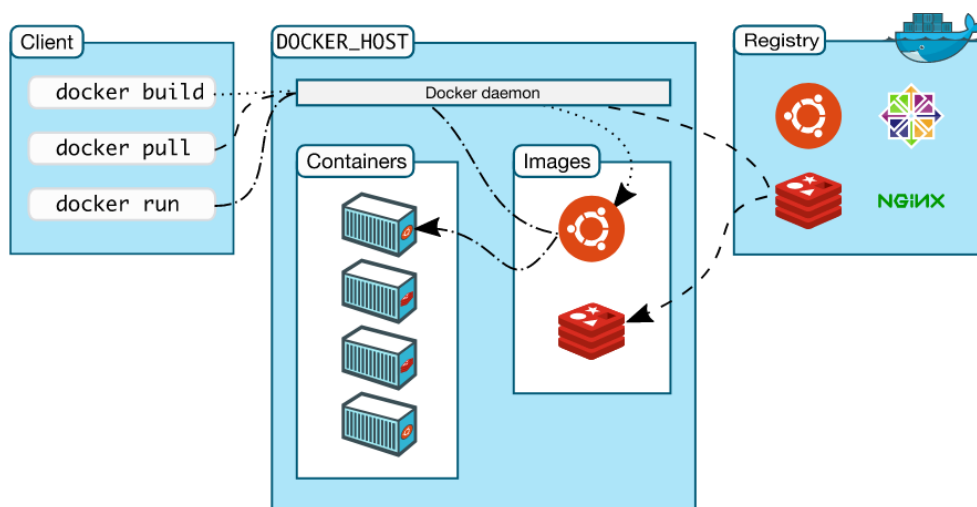
Server: Docker Engine - Community
Engine:
Version: 19.03.13
API version: 1.40 (minimum version 1.12)
Go version: go1.13.15
Git commit: 4484c46d9d
Built: Wed Sep 16 17:01:20 2020
OS/Arch: linux/amd64
Experimental: false
containerd:
Version: 1.3.7
GitCommit: 8fba4e9a7d01810a393d5d25a3621dc101981175
runc:
Version: 1.0.0-rc10
GitCommit: dc9208a3303feef5b3839f4323d9beeb36df0a9dd
docker-init:
Version: 0.18.0
GitCommit: fec3683
```

```
$ docker info
```

```
student@PwCh0-VB:~$ docker info
Client:
 Debug Mode: false

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 19.03.13
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 8fba4e9a7d01810a393d5d25a3621dc101981175
 runc version: dc9208a3303feef5b3839f4323d9beb36df0a9dd
 init version: fec3683
 Security Options:
  apparmor
  seccomp
   Profile: default
 Kernel Version: 5.4.0-48-generic
 Operating System: Ubuntu 20.04.1 LTS
 OSType: linux
 Architecture: x86_64
 CPUs: 4
 Total Memory: 3.844GiB
 Name: PwCh0-VB
 ID: 6TJQ:SKIV:AMBH:Q4W7:XDFH:NGV6:MPRG:6HEK:VH4L:B3DS:XOCU:5ZX5
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false
```

Polecenia wyświetlają wersję klienta i demona (serwera) środowiska Docker CE. Zainstalowane środowisko Docker składa się z dwóch z trzech, wzajemnie powiązanych komponentów, tak jak przedstawia to rysunek poniżej. Registry jest elementem konfigurowanym oddzielnie co zostanie omówione na jednym z kolejnych laboratoriów. Na tym laboratorium skoncentrujemy się na podstawowych poleceniach operujących na obrazach (ang. images) i kontenerach (ang. containers).



Wszystkie polecenia (narzędzia) oferowane przez Docker wraz z opisem znaczenia oraz sposobem wykorzystania można uzyskać za pomocą polecenia:

```
$ docker    oraz    $ docker containers
albo w najbardziej poprawnych formach:
$ docker --help  lub  $ docker -h
```

Uwaga: Zgodnie z informacją wyświetlaną przez powyższe polecenia, pomoc odnośnie składni poszczególnych poleceń można uzyskać, wpisując komendy o składni: `$ docker <COMMAND> --help`

B. Najprostszym sposobem utworzenia kontenera Docker jest skorzystanie z dostępnych publicznie, bazowych obrazów systemów. Dla dystrybucji Ubuntu wynik wyszukiwania obrazów przedstawiony poniżej (fragment).

```
student@pwrch0-vb:~$ docker search ubuntu
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
ubuntu	Ubuntu is a Debian-based Linux operating sys...	11398	[OK]	
dorowu/ubuntu-desktop-lxde-vnc	Docker image to provide HTML5 VNC interface ...	468		[OK]
rastasheep/ubuntu-ssh	Dockerized SSH service, built on top of offi...	248		[OK]
consol/ubuntu-xfce-vnc	Ubuntu container with "headless" VNC session...	227		[OK]
ubuntu-upstart	Upstart is an event-based replacement for th...	110	[OK]	
neurodebian	NeuroDebian provides neuroscience research s...	71	[OK]	
landiinternet/ubuntu-16-nginx-php-phpmyadmin-mysql-5	ubuntu-16-nginx-php-phpmyadmin-mysql-5	50		[OK]
ubuntu-debootstrap	debootstrap --variant=minbase --components=m...	44	[OK]	
nuagebec/ubuntu	Simple always updated Ubuntu docker images w...	24		[OK]
l386/ubuntu	Ubuntu is a Debian-based Linux operating sys...	24		
solita/ubuntu-systemd	Ubuntu + systemd	24		[OK]
landiinternet/ubuntu-16-apache-php-5.6	ubuntu-16-apache-php-5.6	14		[OK]
landiinternet/ubuntu-16-apache-php-7.0	ubuntu-16-apache-php-7.0	13		[OK]
landiinternet/ubuntu-16-nginx-php-phpmyadmin-mariadb-10	ubuntu-16-nginx-php-phpmyadmin-mariadb-10	11		[OK]

W ten sposób istnieje możliwość przeszukania zawartości (np. dostępnych dystrybucji) publicznego repozytorium Docker, które nosi nazwę: Docker Hub. Pełna listę dostępnych obrazów kontenerów, wraz z opisem, można znaleźć (przeszukując repozytorium) pod adresem <https://hub.docker.com/explore/>

Składnia polecenia do przeszukiwania:

```
$ docker search [opcje] <nazwa obrazu>
```

Uwaga: Informację na temat składni przy definiowaniu opcji można uzyskać w podręczniku polecenia: `$ man docker search` a przykłady praktyczne wykorzystania tych opcji na stronie portalu DocsDocker, pod adresem: <https://docs.docker.com/engine/reference/commandline/search/>

Z kolei pobranie zasobu (bazowych obrazów) na lokalny host możliwe jest poprzez polecenie o składni:

```
$ docker pull [opcje] <nazwa obrazu>
```

Listę pobranych (dostępnych lokalnie obrazów) można uzyskać poleceniem:

```
$ docker images    lub    $ docker image ls
```

Przy okazji, uściślijmy nazewnictwo stosowane przy pobieraniu (i wysyłaniu) obrazów. Zgodnie z dokumentacją Docker, poprawny schemat jest następujący.



W przytaczanych przykładach korzystamy z publicznego repozytorium DockerHub (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 składową nazwy <repo>. Należy pamiętać o tej nieścisłości, pomimo że dalej wykorzystywać będziemy potoczną nomenklaturę.

Uwaga1: Jeden obraz może mieć wiele tagów a domyślny tag to „latest”

1Z1. Proszę przeszukać i pobrać bazowy obraz dla dystrybucji ubuntu:16.04. Następnie sprawdzić, że jest on dostępny w lokalnym środowisku Docker. W sprawozdaniu umieścić zrzuty ekranowe użytych poleceń i wyniki ich działania.

C. Na podstawie dostępnych obrazów (sprawdzonych poleceniem docker images) można utworzyć i uruchomić kontener Docker. Istnieją dwie podstawowe metody na realizację tego zadania:

1. Stworzenie kontenera z wybranego obrazu. Uruchomienie kontenerach. Wykorzystywane polecenia to:

```
$ docker create <nazwa kontenera> <nazwa obrazu>
```

Uwaga: Tworzenie kontenera bez jego uruchamiania jest przydatne gdy chcemy zindywidualizować jego konfigurację tak by odpowiadała specyficznym wymaganiom (np. etapu pracy aplikacji). Proszę KONIECZNIE wydać polecenie:

```
$ docker create --help
```

aby zorientować się w zakresie możliwych modyfikacji konfiguracji.

Zestaw dostępnych kontenerów można wyświetlić polecenie:

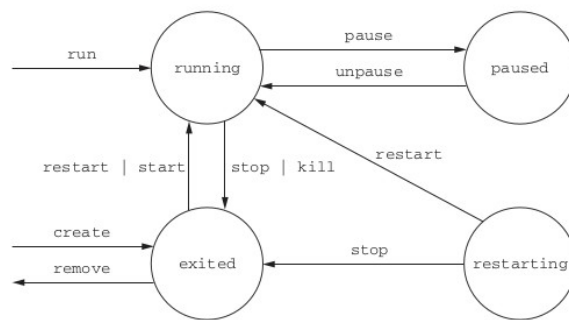
```
$ docker ps -a
```

gdzie: opcja -a oznacza wyświetlenie wszystkich kontenerów, niezależnie od ich stanu (np. running, exited, created, stopped)

Alternatywnie można użyć polecenia:

```
$ docker container ls -a
```

Uwaga: Na poniższym rysunku zilustrowane są możliwe stany kontenera jakie zwracają polecenia informujące o bieżącym statusie kontenera.



Za pomocą powyższych poleceń można poznać losową nazwę kontenera jaką mu nadał demon Dockera i jego ID (Cointainer ID)

Uruchomienie tak utworzonego kontenera jest możliwe za pomocą polecenia o składni:

```
$ docker start -i <nazwa kontenera lub jego ID>
```

Przedstawioną metodę uruchamiania kontenerów przedstawia (dla przykładowego obrazu ubuntu) rysunek poniżej:

```
student@PwCh0-VB:~$ docker create ubuntu
084ebb73529797a2a5578292268b1964f8ac5ae532e1a5f71067c89dcd5a2f20
student@PwCh0-VB:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
084ebb735297	ubuntu	"/bin/bash"	5 seconds ago	Created		funny_hoover
0a64fc1888ff	ubuntu	"/bin/bash"	About a minute ago	Created		nervous_ellis

```
student@PwCh0-VB:~$ docker start -i nervous_ellis
student@PwCh0-VB:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
084ebb735297	ubuntu	"/bin/bash"	38 seconds ago	Created		funny_hoover
0a64fc1888ff	ubuntu	"/bin/bash"	About a minute ago	Exited (0) 8 seconds ago		nervous_ellis

2. Można bezpośrednio uruchomić kontener na podstawie wybranego obrazu. W tym przypadku składnia polecenia uruchamiającego kontener Docker jest następująca:

```
$ docker run <opcje> <nazwa obrazu><polecenie/program>
```

typowe opcje (w kontekście tego laboratorium) to:

- *i* – uruchomienie kontenera i udostępnienie jego powłoki
- *d* – uruchomienie kontenera w tle (wyświetlany jest tylko jego ID)
- - *rm* - usunięcie kontenera po wykonaniu przypisanych mu zadań
- - *name* – nadanie kontenerowi własnej (a nie losowej) nazwy

Uwaga: Użycie opcji - - rm (usunięcia kontenera) jest niezwykle pożyteczne. Bez niej, może się zdarzyć, że w systemie zgromadzimy dużą liczbę kontenerów, o których już dawno zapomnieliśmy.

Działanie tej metody uruchamiania kontenera ilustruje rysunek poniżej:

```
root@06b1ac693646: /
student@PwCh0-VB:~$ docker run --rm -it --name test ubuntu /bin/bash
root@06b1ac693646: /#
```

```
student@PwCh0-VB:~$ docker ps
```

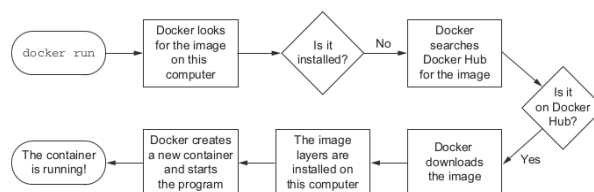
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
06b1ac693646	ubuntu	"/bin/bash"	17 seconds ago	Up 12 seconds		test

Ponieważ przykładowy kontener został uruchomiony z opcją `-rm` to „zniknie” on po wyjściu z powłoki. Dlatego (tak jak na rysunku powyżej) otworzono kolejne okno terminala. Tam sprawdzono, że kontener o nazwie „test” jest działający. Wyjście z kontenera oznacza jego bezpowrotne usunięcie (nie będzie on widoczny na liście dostępnych kontenerów).

Należy również pamiętać, że wykorzystanie tej metody uruchamiania kontenerów jest możliwe również bez wcześniejszego pobrania obrazu. Jeśli obraz nie zostanie odnaleziony lokalnie to daemon Docker pobierze go ze skonfigurowanych repozytoriów (domyślnie DockerHub) o ile oczywiście, obraz jest tam dostępny. Sytuację taką przedstawia rysunek poniżej (obraz `ubuntu:18.04` nie był wcześniej pobrany):

```
student@PwCh0-VB:~$ docker run --rm -it --name test2 ubuntu:18.04 /bin/bash
Unable to find image 'ubuntu:18.04' locally
18.04: Pulling from library/ubuntu
171857c49d0f: Pull complete
419640447d26: Pull complete
61e52f862619: Pull complete
Digest: sha256:646942475da61b4ce9cc5b3fadb42642ea90e5d0de46111458e100ff2c7031e6
Status: Downloaded newer image for ubuntu:18.04
root@f3858f85f2a3:/#
```

Uwaga: Podczas wykorzystania polecenia `docker run` w odniesieniu do obrazu, który wcześniej nie był pobrany, daemon Docker pobiera go z DockerHub. Ilustruje to rysunek poniżej:



Podczas kolejnego odwołania się do tego obrazu, nie jest on pobierany ponieważ jest już dostępny lokalnie, czyli deamo dział zgodnie z algorytmem przedstawionym poniżej:



Proszę również zapoznać się z stroną pomocy dla tego polecenia:

```
$ docker run -help
```

oraz KONIECZNIE z przykładami jego wykorzystania (niezwykle przydatne informacje tak w trakcie tego jak i kolejnych ćwiczeń), które są dostępne pod adresem: <https://docs.docker.com/engine/reference/commandline/run/>

D. Podstawowe metody monitorowania wykorzystują cztery podstawowe polecenia. Pierwsze z nich to używane już polecenie „`docker ps`”

`$ docker ps <opcje>` - wyświetla działające kontenery

typowe opcje to:

- a - all

- aq - all in quiet mode

Alternatywnie można używać polecenia o składni:

`$ docker container ls <opcje>`

Uwaga: w listingu tego polecenia widoczne są: numer i nazwa kontenera. Za ich pomocą można odwoływać się w innych poleceniach środowiska Docker do określonego kontenera.

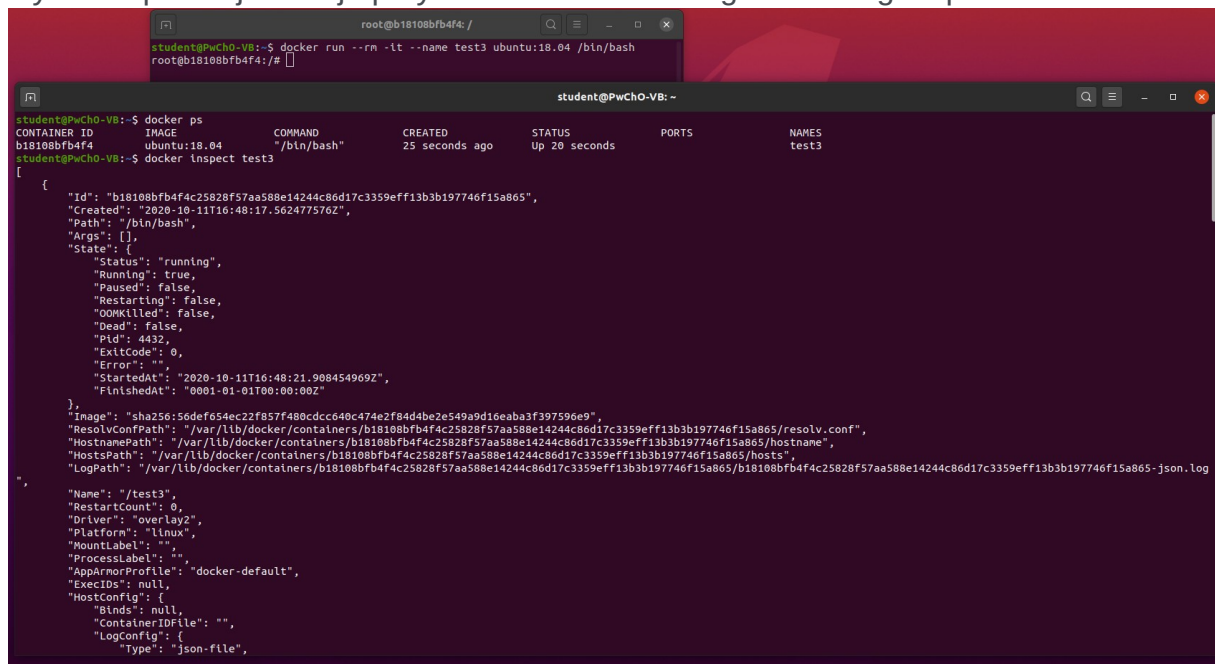
Pozostałe polecenia to:

`$ docker logs <name/container_ID>` – wyświetla działania na powłocie kontenera (szczególnie przydatne w przypadku pracy kontenera w tle).

`$ docker top <name/container ID>` - wyświetla procesy uruchomione w kontenerze

`$ docker inspect <name/container ID>` - pozwala na odczytanie szczegółowej konfiguracji i wykorzystywanych zasobów dla danego kontenera. Szczególnie jest ono przydatne przy podstawowej diagnostyce niepoprawnej pracy kontenera.

Rysunek poniżej ilustruje przykładowe działanie tego ostatniego z poleceń.



```
root@b18108bfb4f4: /
student@PwChO-VB:~$ docker run --rm -it --name test3 ubuntu:18.04 /bin/bash
root@b18108bfb4f4:/#

student@PwChO-VB:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
b18108bfb4f4        ubuntu:18.04        "/bin/bash"        25 seconds ago     Up 20 seconds                      test3

student@PwChO-VB:~$ docker inspect test3
[
  {
    "Id": "b18108bfb4f4c25828f57aa588e14244c86d17c3359eff13b3b197746f15a865",
    "Created": "2020-10-11T16:48:17.562477576Z",
    "Path": "/bin/bash",
    "Args": [],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 4432,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2020-10-11T16:48:21.908454969Z",
      "FinishedAt": "0001-01-01T00:00:00Z"
    },
    "Image": "sha256:56def654ec22f857f480cdcc640c474e2f84d4be2e549a9d16eaba3f397596e9",
    "ResolvConfPath": "/var/lib/docker/containers/b18108bfb4f4c25828f57aa588e14244c86d17c3359eff13b3b197746f15a865/resolv.conf",
    "HostnamePath": "/var/lib/docker/containers/b18108bfb4f4c25828f57aa588e14244c86d17c3359eff13b3b197746f15a865/hostname",
    "HostsPath": "/var/lib/docker/containers/b18108bfb4f4c25828f57aa588e14244c86d17c3359eff13b3b197746f15a865/hosts",
    "LogPath": "/var/lib/docker/containers/b18108bfb4f4c25828f57aa588e14244c86d17c3359eff13b3b197746f15a865/b18108bfb4f4c25828f57aa588e14244c86d17c3359eff13b3b197746f15a865-json.log"
  },
  {
    "Name": "/test3",
    "RestartCount": 0,
    "Restart": false,
    "Driver": "overlay2",
    "Platform": "linux",
    "MountLabel": "",
    "ProcessLabel": "",
    "AppArmorProfile": "docker-default",
    "ExecIDs": null,
    "HostConfig": {
      "Binds": null,
      "ContainerIDFile": "",
      "LogConfig": {
        "Type": "json-file",

```

122. Uruchom kontener na bazie obrazu Fedora w najnowszej wersji (lub innej wersji Linux-a niż Ubuntu), który wypisze na konsoli informacje o dystrybucji i tym samym potwierdzi, że jest to inna dystrybucja niż Ubuntu, na którym działa środowisko

Docker. W sprawozdaniu umieść zrzuty ekranowe zawierające wszystkie użyte polecenia wraz z wynikiem ich działania.

E. Pozostałe, podstawowe polecenia do operowania na kontenerach to:

`docker attach` – przyłącza lokalne standardowe strumienie wejścia, wyjścia i błędu do danego kontenera.

`docker exec` – wejście do kontenera pracującego w tle,

`docker stop` – zatrzymanie kontenera bez jego usuwania co pozwala na jego ponowne wystartowanie.

`docker start` – ponowne uruchomienie kontenera (po jego zatrzymaniu).

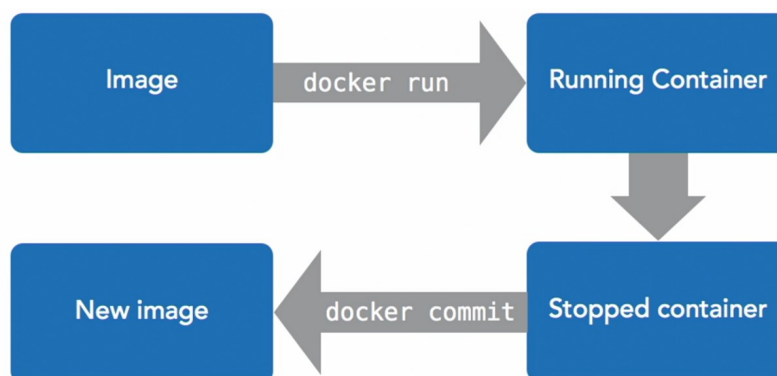
`docker rm` – usunięcie kontenera (należy wcześniej zatrzymać kontener).

`docker rmi` – usunięcie jednego lub wielu obrazów.

Proszę KONIECZNIE zapoznać się ze składnią i opcjami dostępnymi dla wyżej wymienionych poleceń

123. Stwórz a następnie uruchom w tle kontener na bazie obrazu `ubuntu:latest`. Kontener ma w pętli wyświetlać 20 razy tekst "Hello, jestem Docker". Sprawdź działanie poleceń przedstawionych powyżej (`docker ps`, `docker log` oraz `docker exec`). Zatrzymaj kontener a następnie go usuń. W sprawozdaniu umieść zrzuty ekranowe zawierające wszystkie użyte polecenia wraz z wynikiem ich działania.

Pełną obsługę obrazów i kontenerów w środowisku Docker można zilustrować jak na rysunku poniżej.



Przykłady i przytoczone polecenia zawarte w instrukcji obejmują trzy pierwsze etapy (zaznaczone kolorem niebieskim) tj. do zatrzymania kontenera. Często zdarza się, że wprowadzamy zmiany w działającym kontenerze (np. aktualizujemy lub dodajemy/usuwamy komponenty systemu) i nie chcemy tych zmian stracić (kontener po usunięciu „znika” bezpowrotnie). Rozwiązaniem w tej sytuacji jest utworzenie nowego obrazu na bazie zmodyfikowanego kontenera. Służy do tego polecenie

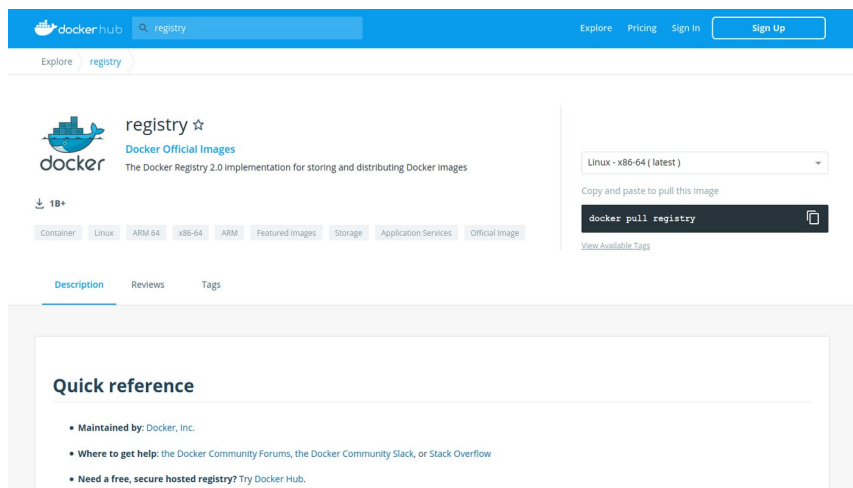
```
$ docker commit [opcje] <kontener> [repozytorium[:tag]]
```

1Z4. Na podstawie wybranego przez siebie obrazu, zilustruj kolejnymi poleceniami wszystkie etapy przedstawione na rysunku powyżej. Polecenia te przyporządkuj do konkretnego etapu i przedstaw wraz z wynikiem ich działania w sprawozdaniu.

F. 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 DockerHub https://hub.docker.com/_/registry . Ilustruje to również rysunek poniżej.



Proszę zapoznać się z umieszczonym tam opisem. Instalacja registry (w przykładzie w wersji 2) polega na wydaniu poniższych poleceń:

```
student@PwCh0-VB:~$ docker pull registry:2
2: Pulling from library/registry
cbbbe7a5bc2a: Pull complete
47112e65547d: Pull complete
46bcb632e506: Pull complete
c1cc712bcecd: Pull complete
3db6272dcbfa: Pull complete
Digest: sha256:8be26f81ffea54106bae012c6f349df70f4d5e7e2ec01b143c46e2c03b9e551d
Status: Downloaded newer image for registry:2
docker.io/library/registry:2
student@PwCh0-VB:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              9140108b62dc       2 weeks ago        72.9MB
ubuntu              18.04              56def654ec22       2 weeks ago        63.2MB
registry            2                  2d4f4b5309b1       3 months ago       26.2MB
student@PwCh0-VB:~$ docker run -d -p 5000:5000 --name myregistry registry:2
e05453baee3cc87d8ada30497a091dc925ee183e1c14cb6235db92b6cf075e7f
student@PwCh0-VB:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
e05453baee3c       registry:2         "/entrypoint.sh /etc..." 16 seconds ago     Up 11 seconds      0.0.0.0:5000->5000/tcp    myregistry
```

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

Uwaga: Ponieważ registry odgrywa ważną rolę w całym środowisku Docker to zazwyczaj zależy nam by zawsze ten element był dostępny. Wobec tego konfigurując

registry, z którego chcemy korzystać bez przerw należy do polecenia tworzącego to registry dodać opcję `--restart always`

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 `myregistry`) jest następujący:

```
student@PwCh0-VB:~$ docker tag redis localhost:5000/redis.local
student@PwCh0-VB:~$ docker push localhost:5000/redis.local
The push refers to repository [localhost:5000/redis.local]
2e9c060aef92: Pushed
ea96cbf71ac4: Pushed
47d8fad6714: Pushed
7fb1fa4d4022: Pushed
45b5e221b672: Pushed
07cab4339852: Pushed
latest: digest: sha256:02d2467210e76794c98ae14c642b88ee047911c7e2ab4aa444b0bfe019a41892 size: 1572
student@PwCh0-VB:~$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	9140108b62dc	2 weeks ago	72.9MB
ubuntu	18.04	56def654ec22	2 weeks ago	63.2MB
redis	latest	84c5f6e03bf0	4 weeks ago	104MB
localhost:5000/redis.local	latest	84c5f6e03bf0	4 weeks ago	104MB
registry	2	2d4f4b5309b1	3 months ago	26.2MB

W powyższym przykładzie, „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. Można też przejrzeć strukturę domyślnego katalogu: `/var/lib/docker/volumes` (KONIECZNE uprawnienia root-a).

Proszę zapoznać się z dokumentacją wykorzystania obrazu registry, dostępnej pod adresem: <https://docs.docker.com/registry/> .

125. Na podstawie przedstawionego opisu i dokumentacji Docker należy:

1. Pobrać i uruchomić kontener registry w najnowszej wersji.
2. Wgrać do niego obraz o nazwie mutant (utworzony w wyniku polecenia `docker commit` w poprzednim zadaniu) 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 pełniący funkcję registry. Usunąć go oraz usunąć obraz `registry:2` oraz zgromadzone przez niego dane.

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 [opcje] <nazwa obrazu>
```

```
$ sudo docker load [opcje] <nazwa pliku .tar/STDIN>
```

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 przykładowo zilustrować w następujący sposób:

```
student@PwCh0-VB:~$ docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
df8698476c65: Pull complete
Digest: sha256:d366a4665ab44f0648d7a00ae3fae139d55e32f9712c67accd604bb55df9d05a
Status: Downloaded newer image for busybox:latest
docker.io/library/busybox:latest
student@PwCh0-VB:~$ docker save -o tester.tar busybox
```

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.

1Z6. 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. 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 środowiska 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 krótko odpowiedzieć na pytania 2a oraz 2b.