

Docker

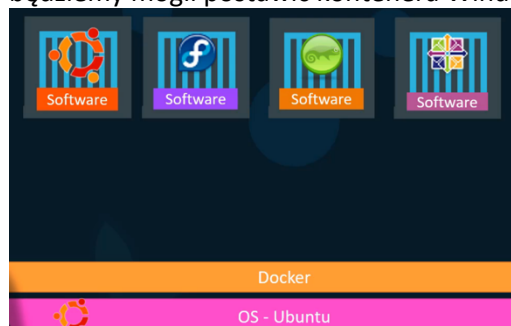
Docker Overview

W trakcie tworzenia aplikacji może wystąpić wiele trudności i przeszkód np. kompatybilność poszczególnych tooli z systemem operacyjnym, którego używamy. Dodatkowo toole muszą być kompatybilne ze sobą i z bibliotekami. Gdy stworzymy taki skomplikowany system wprowadzanie jakichkolwiek zmian zajmuje bardzo dużo czasu i jest trudne, tak samo jak nauczanie nowych inżynierów tego jak jest zbudowana infrastruktura.

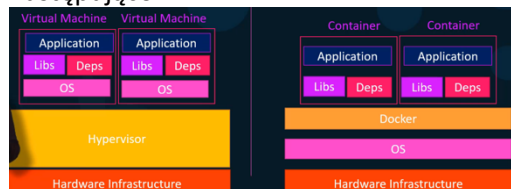
W takim przypadku użycie Dockera jest bardzo pomocne. Pozwala on na uruchomienie każdego składnika środowiska w osobnym kontenerze z własnymi bibliotekami i parametrami. Jedyne co jest potrzebne to zainstalowanie Dockera na swoim systemie operacyjnym.

Kontenery to osobne środowiska mające swoje procesy, karty sieciowe, czy mounty, czyli tak jak Virtual Machine, ale różnica jest taka, że wszystkie współdzielą ten sam OS Kernel. Jeśli spojrzymy na różne systemy operacyjne takie jak Ubuntu, Fedora, CentOS itp. To składają się one z dwóch części: Kernel (w tym przypadku wszystkie mają Linux) i Software, który jest zainstalowany na Linuxie i odróżnia te systemy od siebie (mogą to być różne drivery, kompilatory sterowników itp.).

Jak powiedziano wyżej, kontenery dzielą ten sam Kernel to znaczy, że np. jak mamy swój system Ubuntu i zainstalujemy na nim Dockera to możemy mieć w nim dowolne kontenery, które używają Linuxowego Kernela czyli np. Fedora, CentOS itp. Należy pamiętać, że na takim Docker hoście nie będziemy mogli postawić kontenera Windowsowego, bo ma on już inny Kernel.



Takie jeśli chcielibyśmy mieć np. Linuxowe kontenery na Windowsie to trzeba użyć Virtual Machines (potrzebny będzie jakiś Virtual Hypervisor). Różnica pomiędzy tymi dwoma podejściami wygląda następująco:



Jak widzimy Docker jest zainstalowany bezpośrednio na systemie operacyjnym i jego kontenery korzystają z Kernela, a mają swoje biblioteki i Software. VM natomiast jest na Hypervisorze i każda z nich ma swój system operacyjny. Przez to VM mają większe zużycie zasobów, bo jest więcej systemów operacyjnych i kerneli, dodatkowo zajmują więcej miejsca na dysku niż obrazy Dockerowe, dzięki temu obrazy są szybciej uruchamiane. W dużych firmach często te dwie technologie są używane razem, czyli mamy Dockerowe hosty instalowane na VM.

Mamy wiele wstępnie skonfigurowanych Dockerowych obrazów zapisanych w publicznym repozytorium - DockerHub. Aby stworzyć kontener musimy tylko uruchomić komendę: `docker run nazwa_obrazu`.

Można mieć kilka instancji tego samego obrazu i np. load balancera to rozdzielania ruchu. Jak coś się zepsuje, to można usunąć taką instancję i potem stworzyć szybko od nowa.

Różnica pomiędzy kontenerami i obrazami jest taka, że obraz to wstępnie skonfigurowana templatka, a kontener to uruchomiona instancja na podstawie obrazu. Jest ona izolowana i ma własne procesy.

Getting Started with Docker

Docker ma dwa wydania: Community i Enterprise. Community to wersja podstawowa i darmowa, natomiast w Enterprise mamy dodatkowo płatne toole pomagające w zarządzaniu obrazów, bezpieczeństwem itp.

Setup and install Docker on Linux

Trzeba otworzyć dokumentację Dockera i wybrać system operacyjny, na którym chce się zainstalować Docker - tu Ubuntu.

Najpierw trzeba odinstalować wszystkie stare wersje Dockera:

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

Następnie trzeba ustawić repozytorium i zainstalować software. Można do tego użyć tzw. convenience script. Automatyzuje on cały proces. Wystarczy najpierw go pobrać:

```
curl -fsSL https://get.docker.com -o get-docker.sh
```

I potem go uruchomić:

```
sudo sh get-docker.sh
```

Teraz komendą `sudo docker version` możemy sprawdzić, czy aby na pewno docker zostanie wykryty z jego poprawną wersją. Dokumentacja: [Install Docker Engine on Ubuntu](#) | [Docker Documentation](#)

Docker on Windows

Na Windowsie możemy mieć Dockera jako Docker Toolbox lub Docker Desktop for Windows.

Docker Toolbox - jest zbiór tooli, które są instalowane, aby umożliwić instalację Dockera na Windowsie. Najpierw jest instalowany Oracle VirtualBox z Linuxem i tam bezpośrednio jest instalowany sam Docker. Wymaganiem jest włączenie Virtualizacji na Windowsie.

Docker Desktop for Windows - tutaj jest używane MicroSoft Hyper-V zamiast Oracle VirtualBox. Na tej VM jest też Linux i tam na nim Docker. To rozwiązanie na nowych wersjach Windowsa pozwala też na uruchamianie kontenerów Windowsowych.

Mamy dwa rodzaje kontenerów Windowsowych:

- Windows Server - działa tak jak Linuxowy, że mamy wspólny kernel dla kontenera i systemu operacyjnego Docker hosta.

- Hyper-V Isolation - tutaj każdy kontener ma swój własny kernel i mogą one być różne. Każdy kontener jest uruchamiany w osobnej Virtual Machine.

Windowsowe kontenery są tylko wspierane na Windows Server 2016, Nano Server i Windows 10 Professional and Enterprise. Dodatkowo dla Windowsa mamy tylko dwa rodzaje podstawowych image: Windows Server Core i Nano Server.

Docker on MAC

Jest tutaj podobnie do Windowsa bo mamy Docker Toolbox (tak samo jak na Widnowsie, Docker będzie zainstalowany na Linuxowej VM od VirtualBoxa) i Docker Desktop for Mac (tutaj zasada jest taka sama, ale jest HyperKit zamiast VirtualBoxa).

Oba rozwiązania umożliwiają tylko uruchamianie kontenerów Linuxowych.

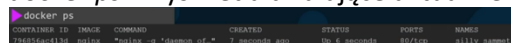
Docker Commands

Basic Docker Commands

`docker run nazwa_obrazu` - używane do uruchamiania kontenera z danego obrazu. Jeśli uruchamiamy tę komendę pierwszy raz to najpierw obraz zostanie ściągnięty z DockerHuba do nas lokalnie, a każde kolejne uruchomienie będzie już z zapisanego obrazu. Jeśli chcemy pobrać obraz z DockerHuba, ale ten obraz nie jest oficjalny np. dodany przez jakiegoś użytkownika to trzeba też sprecyzować jego nazwę w komendzie: `docker run użytkownik/obraz`.

`docker run --name nazwa nazwa_obrazu` - stworzenie kontenera o danej nazwie.

`docker ps` - wyświetla działające aktualnie kontenery i podstawowe ich informacje:



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
78f6a413d	nginx	nginx -g 'daemon off;'	7 seconds ago	Up 6 seconds	80/tcp	silly_name1

Każdy kontener po stworzeniu otrzymuje unikalne ID i nazwę od Dockera.

`docker ps -a` - wyświetla wszystkie kontenery, także te zastopowane.

`docker stop nazwa_kontenera/ID_kontenera` - zatrzymuje dany kontener.

`docker rm nazwa_kontenera/ID_kontenera` - usuwa dany kontener na stałe. Teraz `docker ps -a` już go nie pokaże.

`docker images` - pokaże wszystkie obrazy Dockerowe, które pobraliśmy lokalnie. Będą też tu podstawowe informacje o nich, takie jak rozmiar, tag, czy ID.

`docker rmi nazwa_obrazu` - usuwa dany obraz z naszej lokalnej maszyny. Należy jednak pamiętać, że zanim to uruchomimy, trzeba zastopować wszystkie kontenery, które używają tego obrazu.

`docker pull nazwa_obrazu` - pobiera dany obraz lokalnie, bez tworzenia kontenera.

`docker image prune -a` - usuwanie wszystkich obrazów lokalnych, które nie mają kontenera na sobie.

Jeśli uruchomimy np. `docker run ubuntu`. To komenda się wykona i kontener po chwili zostanie wyłączony i będziemy go widzieć jako zastopowany. Dzieje się tak dlatego, że kontenery nie mają za zadanie hostować systemu operacyjnego, ale tylko wykonać pewne zadanie. Tutaj nie sprecyzowaliśmy zadania, więc on po chwili się zatrzymał. Kontener jest włączony tak długo, jak długo działa w nim jego proces.

Aby przedłużyć jego działanie możemy np. dodać komendę `sleep`, aby została uruchomiona po starcie kontenera: `docker run ubuntu sleep 5` (tu kontener będzie uśpiony przez 5 sekund).

Generalnie

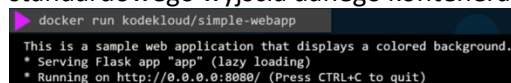
`docker run nazwa_image komenda` - uruchomi daną komendę po stworzeniu kontenera.

`docker exec nazwa/ID_kontenera komenda` - uruchamia daną komendę na działającym kontenerze np. na takim, który ma `sleep 100` na wejściu. Przykład: `docker exec nazwa cat /etc/hosts` - wyświetlenie pliku `hosts` z danego kontenera.

Przykładowo mamy obraz, który ma w sobie web server nasłuchujący na porcie 8080. Jest on zapisany w repozytorium kodekloud pod nazwą `simple-webapp`. Jeśli uruchomimy komendę:

`docker run kodekloud/simple-webapp`

To będzie to uruchomienie w trybie `attached`. To znaczy, że będziemy dołączeni do konsoli lub standardowego wyjścia danego kontenera i w tym przypadku zobaczymy output naszego web service.



```
docker run kodekloud/simple-webapp
This is a sample web application that displays a colored background.
* Serving Flask app "app" (lazy loading)
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

Nie będzie można nic robić w tej konsoli, oprócz oglądania outputu, do momentu zatrzymania kontenera. Jak klikniemy `Ctrl+C` to wyjdziemy z kontenera i zatrzymamy aplikację.

Druga opcja to uruchomienie kontenera w trybie detached:

```
docker run -d kodekloud/simple-webapp
```

Dzięki temu kontener będzie pracował w tle, a my powrócimy od razu do naszego command prompta.

Jeśli chcemy się natomiast połączyć (attach) z powrotem do pracującego w tle kontenera to uruchamiamy:

```
docker attach ID_kontenera
```

Należy pamiętać, że przy precyzowaniu ID kontenera wystarczy nam tylko kilka pierwszych jego znaków, tak aby było unikalne.

Kopiowanie plików - docker cp

Można kopiować pliki z kontenera do naszego prywatnego komputera i na odwrót. Wykorzystujemy do tego polecenie:

```
docker cp ścieżka/nazwa_pliku nazwa_kontenera:ścieżka_pliku - kopiowanie z komputera do kontenera.
```

```
docker cp nazwa_kontenera:ścieżka_pliku ścieżka_docelowa - kopiowanie z kontenera na komputer.
```

Docker Run

Docker Run

Jeśli uruchamiamy komendę `docker run redis`, to stworzymy kontener o obrazie `redis` z domyślną wartością jego taga - `latest`. Jest to najnowsza wersja tego obrazu, ale jeśli chcemy sprecyzować konkretną, którą chcemy mieć to trzeba w komendzie sprecyzować tag np:

```
docker run redis:4.0
```

Wszystkie wersje i tagi danego obrazu możemy sprawdzić w DockerHubie.

Jeśli chcemy przy tworzeniu kontenera podać coś do niego przez standardowe wejście, to domyślnie jest to zablokowane i musimy sprecyzować parametr `-i` żeby uruchomić standardowe wejście. Przykładowo mamy aplikację, która zwraca tekst "Hello name". W miejsce imienia trzeba coś podać poprzez standardowe wejście, bo jak nie to wstawi się pusty tekst:

```
docker run kodekloud/simple-prompt-docker
Hello and Welcome !
```

Dodając parametr `-i` będziemy mieć możliwość wpisania w terminalu jakiegoś tekstu, który potem będzie służył jako standardowe wejście do kontenera:

```
docker run -i kodekloud/simple-prompt-docker
Mumshad
Hello and Welcome Mumshad!
```

Widzimy jednak, że musimy wpisać imię w puste pole, nie wiedząc tak naprawdę czego chce aplikacja (mogłaby tu być np. wyświetlona jakaś instrukcja z tym, co jest oczekiwane). Dzieje się tak dlatego, że w kontenerach domyślnie nie jest montowany terminal, a to właśnie w terminalu takie instrukcje będą wyświetlane. Aby dodać terminal podajemy parametr `-t` i prompt będzie wyświetlony:

```
docker run -it kodekloud/simple-prompt-docker
Welcome! Please enter your name: Mumshad
Hello and Welcome Mumshad!
```

Parametry `-it` przydają się też jeśli chcemy uruchomić sobie kontener i wykonać w nim szybko kilka czynności testowo. Przykładowo: `docker run -it ubuntu bash` stworzy nam kontener z uruchomionym shellem, do którego będziemy mogli wpisywać komendy, jeśli chcemy coś np. przetestować. Po wyjściu z `bash`, kontener się wyłączy.

Port mapping

Jeśli mamy w kontenerze np. webserver i po uruchomieniu go nasłuchuje on na porcie 5000 to należy pamiętać, że po deploymentcie w Docker hoście, nie jest on wystawiony domyślnie na zewnątrz. Aby się do niego połączyć wewnętrznie wystarczy wziąć wewnętrzny adres IP kontenera (pokaże nam go komenda `docker inspect`) i np. z przeglądarki na Docker hoście połączyć się z <http://IP:5000>.

W przypadku łączenia się z zewnątrz możemy wziąć adres IP Docker hosta, ale domyślnie nie będzie on wiedział, co to jest ten port 5000. Musimy zrobić tzw port mapping w trakcie tworzenia kontenera - jest to mapowanie portu wewnątrz kontenera z portem zewnętrznym.



Robimy to komendą:

```
docker run -p 80:5000 image_name
```

Teraz można się łączyć z zewnątrz po linku http://IP_hosta:80.

W ten sposób można stworzyć wiele instancji naszej aplikacji i poustawiać różne porty, aby łatwo można było rozdzielić ruch.

Volume mapping

Przykładowo tworzymy kontener mysql. Po stworzeniu bazy danych, domyślnie dane są przechowywane w `/var/lib/mysql`. Jeśli wrzucimy tam nasze dane to kontener będzie z nich korzystał, jeśli natomiast usunęlibyśmy ten kontener, to wszystkie te dane będą stracone. Aby móc zapisywać dane zewnętrznie, tak by ich nie tracić, a jednocześnie móc ich używać w kontenerze trzeba zmapować folder na Docker hoście do odpowiedniego miejsca w kontenerze, w tym przypadku:

```
docker run -v /opt/datadir:/var/lib/mysql mysql
```

To zmapuje folder datadir na Docker hoście z folderem mysql w kontenerze.

Jeśli chcemy uzyskać bardziej szczegółowe dane o kontenerze to trzeba skorzystać z komendy:

```
docker inspect nazwa
```

Dane będą zwrócone w formacie JSON. Będą tu ustawienia mountów, sieci itp.

docker logs nazwa - wyświetli logi z danego kontenera. Będą to logi idące do standardowego wyjścia z kontenera np. logi aplikacji.

Docker Images

Docker Image

Gdy chcemy naszej aplikacji użyć kilka razy i traktować jako szablon to dobrze jest zrobić z niej image. Na początku trzeba dokładnie wiedzieć co chcemy skonteneryzować i jak aplikacja jest zbudowana. W naszym przypadku mamy aplikację webową zbudowaną przy pomocy Flask Python framework. Dobrze jest najpierw wypisać sobie krok po kroku, co będzie potrzebne żeby ta aplikacja działała:

1. System operacyjny np. Ubuntu
2. Aktualizacja źródłowego repozytorium przy pomocy komendy apt
3. Instalacja dependencies przy pomocy apt (parametr -y jest wymagany, bo to od razu odpowiedź yes na wszystkie prompty)
4. Instalacja dependencies Pythona przy pomocy pip
5. Kopiowanie kodu źródłowego aplikacji np. do folderu /opt
6. Uruchomienie Web Servera przy pomocy komendy flask

Teraz na podstawie tych kroków trzeba stworzyć tzw. Dockerfile, który utworzy nam image. Dla podanych wyżej komend będzie on miał postać:

```
Dockerfile
FROM Ubuntu

RUN apt-get update && apt-get -y install python

RUN pip install flask flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

Teraz aby stworzyć Docker image używamy komendy:

```
docker build . -f Dockerfile -t mmumshad/my-custom-app
```

Należy pamiętać aby dodać też tag.

W tym momencie Docker image zostanie stworzone na naszym lokalnym hoście, aby udostępnić go do DockerHub należy uruchomić komendę:

```
docker push mmumshad/my-custom-app
```

Dockerfile

Plik do tworzenia image ma określony format. Jest to instrukcja - argument. Przykładowo biorąc plik powyżej, instrukcje to FROM, RUN, COPY itp.

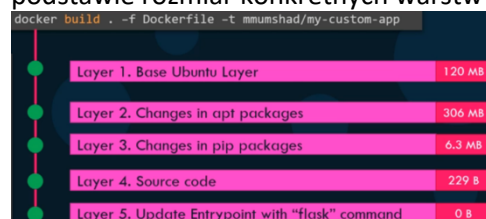
FROM - określa na podstawie jakiego obrazu budujemy nasz image (każdy Dockerfile musi się zaczynać od tej instrukcji).

RUN - odpowiada za uruchamianie odpowiedniej komendy na obrazie.

COPY - kopiowanie z danego lokalnego miejsca do określonej ścieżki w image.

ENTRYPOINT - określa komendę, która będzie uruchomiona po tym, jak zostanie zbudowany kontener na podstawie naszego image.

Gdy image jest tworzony, jest on w strukturze warstw, każda instrukcja dodaje nową warstwę. Każda warstwa przechowuje tylko i wyłącznie zmiany, które zostały zrobione w poprzedniej warstwie. Na tej podstawie rozmiar konkretnych warstw jest różny:



Jeśli uruchomimy komendę:

docker history nazwa_image

To zobaczymy tą samą informację z rozmiarem warstw.

Każda warstwa jest cachowana, dzięki temu jeśli chcemy coś dodać do naszego pliku, to warstwy już istniejące nie będą budowane, a dodane zostaną tylko te brakujące, dzięki czemu sam proces jest o wiele szybszy. W przypadku błędu budowania, ponowny build ruszy też od warstwy, która się nie udała i budowane będą jedynie kolejne po niej warstwy.

Build context

W poleceniu *docker build*nie podajemy ścieżki do Dockerfile ale do contextu, czyli do katalogu, do którego odnosi się Dockerfile. Np w Dockerfile jest polecenie *COPY skni.txt*no i ta lokalizacja, gdzie jest do skni.txt jest contextem. Dockerfile jest najczęściej też w tej lokalizacji, która jest contextem. Możemy jednak zrobić builda używając Dockerfile, z innej lokalizacji niż context. Trzeba wtedy użyć flagi *-f* (flaga ta jest też używana jak mamy kilka Dockerfile w jednej lokalizacji):

docker build -f ścieżka_do_Dockerfile . (context to kropka, czyli obecna lokalizacja). Jeśli tworzymy obraz z Dockerfile i w nim odwołujemy się do plików znajdujących się w katalogach wyżej niż context to wystąpi błąd, bo możemy używać plików z contextu lub z katalogów niżej niż context.

Nie jest dobrze uruchamiać *docker build* w dużych katalogach, bo wtedy będzie on contextem i wszystkie pliki tam się znajdujące będą przetwarzane przez dockera. Może to spowodować duże zużycie CPU itp.

Wysyłanie to odbywa się przez Docker Client do Docker Deamona.

Tagowanie obrazów

Przy tworzeniu obrazu poleceniem build możemy dodać tag i możemy podać nazwę obrazu:

docker build --tag nazwa_obrazu ścieżka_do_Dockerfile

Tag jest to parametr dotyczący danego obrazu i jeden obraz może mieć wiele tagów. Przy dodawaniu nowych tagów ID obrazu się nie zmieni. Dodawanie tagu:

docker tag nazwa_obrazu:stary_tag nazwa_obrazu:nowy_tag

Nazwa ogólna to Repository, a wersja konkretna obrazu to Tag.

Tagi są też opisane w Docker Hubie i teraz jak ściągamy konkretny obraz to określając jego Tag możemy pobrać daną wersję np:

FROM ubuntu:19.10

...

...

Environment Variables

Czasami jest tak, że potrzebujemy dodać jakieś parametry w trakcie startu naszej aplikacji w kontenerze. Posiadanie ich w kodzie nie jest dobrą praktyką, bo zawsze, gdy coś trzeba będzie zmienić, to cały kod będzie wczytany od nowa, więc lepiej dodać te parametry jako zmienne środowiskowe. Jeśli chcemy dodać taką zmienną to przy tworzeniu kontenera możemy dodać do komendy *docker run* parametr *-e*.

docekr run -e APP_COLOR=blue simple-webapp-color - to stworzy zmienną *APP_COLOR* wewnątrz kontenera.

Jeśli chcemy zobaczyć zmienne środowiskowe w istniejącym już kontenerze to trzeba uruchomić komendę *docker inspect* i tam będą one wyodrębnione w sekcji Env.

Command vs Entrypoint

Większość obrazów dockerowych ma ustawione domyślne komendy, polecenia, które będą wykonywane w momencie startu. Gdy chcemy je nadpisać i użyć swoich, to trzeba je podać w komendzie *docker run*.

docker run sleep 5 - tutaj komenda *sleep* nadpisze inne domyślne.

Jeśli chcemy natomiast ustawić domyślną komendę dla danego image to przy budowaniu, w Dockerfile, ustawiamy pole *CMD*, które określi jakie komendy będą uruchomione w trakcie startu domyślnie. Można je precyzować w JSON lub normalnie:

CMD command parameter1

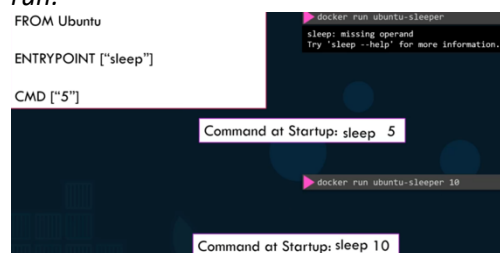
CMD ["command", "parameter1"]

Jeśli np. ustawiliśmy *CMD sleep 5*, tak aby nasz obraz spał przez 5 sekund, ale będziemy chcieli zmienić sam parametr np. do 10 sekund, to można zrobić to tak jak na początku (dodając całą komendę do *docker run*), lub użyć *ENTRYPOINT*. To co ustawimy w *ENTRYPOINT* będzie uruchomione po starcie kontenera wraz z parametrami, które ustawimy w komendzie:



Należy pamiętać, że *ENTRYPOINT* czeka zawsze na input z komendy *docker run* i jeśli w tym przypadku nie podalibyśmy żadnej cyfry to uruchomione byłoby samo *sleep* i byłby błąd z brakującym parametrem.

Jeśli chcielibyśmy w takim razie ustawić jakąś wartość domyślną to trzeba użyć *CMD* i *ENTRYPOINT* razem, wtedy to co w *CMD* to będzie domyślne i będzie to można nadpisać podając parametr w *docker run*:



Ważne żeby to zadziałało, należy sprecyzować *CMD* i *ENTRYPOINT* w formacie JSON.

Jeśli z jakiegoś powodu chcielibyśmy jednak nadpisać *ENTRYPOINT* to można to zrobić dodając odpowiednią flagę w *docker run*:

docker run --entrypoint sleep2.0 ubuntu-sleeper 10 - tutaj *sleep2.0* zastąpi to co jest w *ENTRYPOINT*.

Example of web and console app contenarization

Tworzymy sobie w głównym katalogu ('/') aplikację webową i konsolową *app.py*:

```
from flask import Flask
app=Flask(__name__)
@app.route("/")
def hello_world():
    return "Hello, Internet"
if __name__ == '__main__':
    print('Hello console')
    i=10
    while i>0:
        print(f'times{i}')
        i-=1
```

Mamy jeszcze requirements.txt i tam są potrzebne paczki. Tworzymy teraz Dockerfile.

FROM python:3.8

COPY requirements.txt .

RUN pip install -r requirements.txt - instalacja potrzebnych paczek przy użyciu pipy.

COPY app.py .

CMD python app.py - CMD - w momencie gdy kontener będzie uruchamiany to wykonaj w nim od razu dane polecenie, tu python app.py.

Budujemy kontener

docker build --tag pyapp .

Po każdej zmianie w kodzie możemy znowu uruchomić docker build i dzięki temu zostaną przesłane tylko te zmiany, rzeczy, które już istnieją nie będą kopiowane.

Uruchamianie aplikacji.

docker run pyapp - tu wykona się konsolowa część naszej aplikacji. Mając taki obraz pyapp możemy go wysłać do kogokolwiek i będzie on w stanie łatwo uruchomić tą aplikację. Jest to wygodne do testowania.

Aby uruchomić webową część aplikacji trzeba w Dockerfile zmienić w *CMD FLASK_APP=app python -m flask run --host=0.0.0.0*

Po stworzeniu kontenera i uruchomieniu go, aplikacja będzie działać wewnątrz kontenera. Aby się do niej dostać należy dodać flagę publish 5000, aby otworzyć port 5000 żebyśmy mogli się dostać do aplikacji.

docker run --publish 5000 pyapp:web

Po otworzeniu ruchu, możemy wejść do przeglądarki. Potrzebujemy IP docker machine (localhost na linuxie). W przeglądarce podajemy IP:port. Port sami ustawiamy w poleceniu docker build, przy otwieraniu ruchu na port 5000.

ADD, COPY and WORKDIR in Dockerfile

COPY - kopiuje plik z build contextu do kontenera. Kopiować można pliki też do innych lokalizacji, nie tylko do tej, w której się znajdujemy. Jeśli docelowa lokalizacja nie istnieje, to zostanie automatycznie stworzona. W przypadku katalogów należy podać slash na końcu, bo jak nie będzie slash'a to powstanie nam pojedynczy plik. Można też kopiować całe foldery. Należy pamiętać, że jeśli chcemy skopiować folder razem z zawartością to trzeba napisać:

COPY katalog/ /nowy_katalog/katalog. Można też kopiować na raz wiele plików, podajemy je wtedy po spacji, a ostatni parametr to będzie zawsze katalog, do którego będziemy kopiować pliki.

ADD - rozbudowana wersja polecenia copy. Dodatkowo pozwala ściągnąć plik z internetu (można dodać URL) oraz możemy podać do dodania plik spakowany w formacie tar.gz i podczas tworzenia obrazu, plik ten zostanie automatycznie rozpakowany w miejscu docelowym.

WORKDIR - jeśli dany katalog nie istnieje, to zostanie stworzony. Po drugie ten katalog zostanie ustawiony jako katalog, w którym się znajdujemy dla wszystkich następnych poleceń COPY, CMD, ENTRYPOINT itp.

Składnia: WORKDIR /katalog

Docker Image optimization

Zmniejszanie obrazu używając FROM scratch

Jest to przydatne w językach, które potrafią się skompilować do jednej binarki.

Mamy aplikację w C, która wyświetla tekst. Tworzymy Dockerfile, który stworzy obraz z tą binarką.

FROM gcc:10.c AS builder (obraz do C)

WORKDIR /app

COPY main.c /app/. (kod źródłowy)

RUN gcc -o my_app -static main.c (tworzenie binarki kompilatorem.)

FROM scratch - scratch to obraz, który powoduje, że zaczynamy z całkowicie pustym obrazem bez żadnych plików i bibliotek. Wszystkie biblioteki, które chcemy używać trzeba przekopiować z innych obrazów. Ale my podaliśmy wcześniej flagę static, więc wszystkie będą w tej binarce.

COPY --from=builder /app/my_app /myapp. (kopiowanie gotowej binarki)

ENTRYPOINT ["/myapp"]

Teraz możemy zbudować obraz z tego Dockerfile przez docker build. Mamy teraz obraz z samym skompilowanym plikiem binarnym, nie ma żadnych niepotrzebnych plików, ale nie ma tu też shella, bo to pusty obraz. Teraz porównując wagę obrazu binarki, a obraz z bibliotekami, to zobaczymy dużą różnicę (tu binarka-900kB).

Tego typu obrazy są najlepsze do aplikacji, które się kompilują do binarek (python tu nie będzie dobrym rozwiązaniem). Ze względu na bezpieczeństwo jest to dobre, bo nie wyciekną nam żadne pliki, bo na obrazie jest tylko binarka.

Sposoby optymalizacji obrazów Dockerowych

-instalowanie i pobieranie paczek systemowych powinno być na górze naszego Dockerfile, dalej paczki pythonowe - programistyczne, następnie kod naszej aplikacji. Chodzi o to, aby rzeczy, które się najczęściej zmieniają (kod), były jak najniżej, aby inne, niezmiennie podczas rebuilda były brane z cache.

-Łączenie poleceń: Gdy mamy np instalowanie paczek systemowych to wcześniej trzeba je najczęściej zupdate'ować, bo możemy przez przypadek zainstalować starą wersję. Jeśli raz zrobimy update i potem kiedyś dodamy kolejne paczki do instalacji to update jeśli będzie w oddzielnej linii to się nie wykona, bo będzie cache, dlatego najlepiej te dwie komendy połączyć ze sobą, żeby update był wykonany zawsze, gdy przed instalacją. Zamiast:

`RUN apt-get update`

`RUN apt-get install`

Lepiej użyć:

`RUN apt-get update && apt-get install`

Kolejną zaletą łączenia komend jest nietworzenie dodatkowych warstw. Najlepiej łączyć te polecenia, które mają być wykonywane razem (update i install np.), albo jak mamy jakieś krótkie operacje wykonywane na pliku, czyli tworzymy go, robimy coś i usuwamy, to też najlepiej te wszystkie komendy wrzucić do jednej linii. Dzięki temu niepotrzebny plik nie będzie obecny w wielu warstwach, co powiększy rozmiar obrazu.

-obrazy slim. Np debian ma edycje swoich obrazów, która się nazywa slim i ma o wiele mniejszy rozmiar niż defaultowy, bez dodatkowych programów. Czyli przy tworzeniu Dockerfile robimy:

FROM debian:slim (trzeba oczywiście w Docker Hubie sprawdzić dokładną nazwę)

Oplaca się też czasami użyć obrazu slim i doinstalować samemu potrzebne nam rzeczy.

-obrazy alpine. Jest do mało znana dystrybucja linuxa, która zajmuje bardzo mało miejsca. Jedyna różnica to taka, że manager paczek to apk, a nie apt. (Przykładowe instalowanie to apk add paczka).

FROM alpine

-plik .dockerignore. Pamiętamy, że przy użyciu docker build, cały katalog z Dockerfile jest wysłany do Docker Deamona, dlatego warto mieć Dockerfile w osobnych lokalizacjach. Jednak jest możliwość

ograniczenia tego wysyłania w pliku .dockerignore. Tu podajemy wszystkie pliki i katalogi z lokalizacji Dockerfile, których nie chcemy wysyłać do Docker Deamona.

Docker Compose

Docker Compose

Nie jest on instalowany domyślnie, więc trzeba to zrobić samemu osobno. Jeśli chcemy stworzyć jakąś bardziej skomplikowaną aplikację składającą się z kilku kontenerów to dobrze użyć Docker Compose. W nim tworzymy pliki `.yaml`, w których precyzujemy co trzeba stworzyć i następnie uruchamiamy je jedną komendą. Przykładowy plik `yaml`:

```
docker-compose.yml
services:
  web:
    image: "mmumshad/simple-webapp"
  database:
    image: "mongodb"
  messaging:
    image: "redis:alpine"
  orchestration:
    image: "ansible"
```

Przykład, którym będziemy się tu posługiwać to `voting-app`. Składa się on z aplikacji webowej, która ma interfejs do głosowania. Każdy głos jest zapisywany w in-memory DB - Redis. Tą informację odbiera tzw Worker, który jest aplikacją .NET i on zapisuje tą wartość w prawdziwej PostgreSQL db skąd jest ona brana i wyświetlana na interfejsie z wynikami. Mamy tu różne toole i platformy użyte do zbudowania tej całej aplikacji.

Gdybyśmy chcieli to wszystko stworzyć to trzeba by użyć 5 razy komendy `docker run` (porty udostępniane, aby móc otworzyć w przeglądarce aplikację i rezultaty):

```
docker run -d --name=redis redis
docker run -d --name=db postgres
docker run -d --name=vote -p 5000:80 voting-app
docker run -d --name=result -p 5001:80 result-app
docker run -d --name=worker worker
```

W tym momencie jednak, nasza aplikacja nie będzie działać, ponieważ stworzyliśmy poszczególne obiekty, ale nie połączyliśmy ich ze sobą i one nie wiedzą, gdzie mają przekazywać ruch. Trzeba tutaj stworzyć link np. dla aplikacji do głosowania:

`docker run -d --name=vote -p 5000:80 --link redis:redis voting-app` - tutaj dodaliśmy link do aplikacji redis, bo do niej właśnie będą najpierw przesyłane odpowiedzi. W praktyce jest tu tworzona nowa linijka w `/etc/hosts` w tworzonym kontenerze dodająca hosta i IP redis kontenera. Po dodaniu linków, komendy będą miały postać:

```
docker run -d --name=redis redis
docker run -d --name=db postgres
docker run -d --name=vote -p 5000:80 --link redis:redis voting-app
docker run -d --name=result -p 5001:80 --link db:db result-app
docker run -d --name=worker --link db:db --link redis:redis worker
```

Ustawienia `redis:redis` są oznaczają zmienna_ustawiona_w_kodzie:nazwa_kontenera.

Odpowiadający plik `yaml` do Docker Compose miałby następującą postać:

```

docker-compose.yml

redis:
  image: redis
db:
  image: postgres:9.4
vote:
  image: voting-app
  ports:
    - 5000:80
  links:
    - redis
result:
  image: result-app
  ports:
    - 5001:80
  links:
    - db
worker:
  image: worker
  links:
    - redis
    - db

```

Po stworzeniu tego pliku tworzymy wszystko komendą:
docker-compose up

Jeśli któryś z podanych obrazów nie byłby jeszcze stworzony lokalnie, ani dostępny w DockerHubie to można w pliku yml podać ścieżkę, skąd Docker Compose ma zbudować nowy obraz (musi tam znajdować się odpowiedni Dockerfile). Przykładowo dla vote aplikacji byłoby:

```

vote:
  build: ./vote
  ports:
    - 5000:80
  links:
    - redis

```

Mamy kilka różnych wersji Docker Compose. Ta której używaliśmy wyżej to version: 1. Jest to najstarsza wersja i nie ma wszystkich funkcji np. nie można w niej precyzować sieci, w której chcemy stworzyć kontener. Tutaj wszystkie kontenery będą dodane do domyślnej bridged network, a żeby umożliwić ich komunikację trzeba stworzyć linki. Dodatkowo jeśli chcieliśmy ustawić kolejność tworzenia kontenerów, też nie było to możliwe.

Te funkcjonalności doszły w 2 wersji i zmienił się też format pliku yml. Trzeba dodać na górze sekcję version, a wszystkie kontenery umieścić w sekcji services:

```

docker-compose.yml
version: 2
services:
  redis:
    image: redis
  db:
    image: postgres:9.4
  vote:
    image: voting-app
    ports:
      - 5000:80
    depends_on:
      - redis

```

W wersji 2 tworzona jest automatycznie nowa bridged network, do której zostają dodane wszystkie kontenery i mogą się one od razu ze sobą komunikować po swoich nazwach. Dodatkowo mamy tutaj opcję `depends_on`, w której możemy ustawić co jest wymagane dla danego kontenera, np. tutaj `voting-app` chce aby najpierw został stworzony `redis`.

Jest też wersja 3, która ma podobną strukturę do drugiej, ale ma dodatkowe rozszerzenia do Docker Swarm:

```

docker-compose.yml
version: 3
services:
  redis:
    image: redis
  db:
    image: postgres:9.4
  vote:
    image: voting-app
    ports:
      - 5000:80

```

Jeśli chcielibyśmy podzielić nasze kontenery na różne sieci np. tutaj wszystko do czego ma dostęp użytkownik to front-end network (`voting-app` i `result-app`), jednocześnie operacje wewnętrzne aplikacji potrzebują backend-app (dodamy też tutaj `voting-app` i `result-app`). Aby to zrealizować trzeba dodać sekcję `networks` w pliku `yml` i odnośniki do nich w poszczególnych kontenerach:

```

docker-compose.yml
version: 2
services:
  redis:
    image: redis

    networks:
      - back-end
  db:
    image: postgres:9.4
    networks:
      - back-end
  vote:
    image: voting-app
    networks:
      - front-end
      - back-end
  result:
    image: result
    networks:
      - front-end
      - back-end
networks:
  front-end:
  back-end:

```

W przypadku tworzenia obiektów przy pomocy Docker Compose, ich nazwy będą miały na początku dodaną nazwę katalogu, w którym to tworzymy.

docker-compose down - wyłączanie i usuwanie wszystkich kontenerów.

docker-compose down -v - wyłączenie kontenerów wraz z volume.

docker-compose up --build - flagę *build* dodajemy, jeśli chcemy przebudować już istniejący deployment z compose

docker-compose exec nazwa_kontenera polecenie - wykonanie konkretnych poleceń na danym kontenerze przy użyciu docker-compose.

Docker Registry

Docker Registry

Jest to miejsce gdzie przechowywane są obrazy Dockerowe. Odwołując się do nazw obrazów mamy następujący naming convention: user/image np. nginx/nginx to sprecyzowanie użytkownika nginx i obrazu nginx.

Jeśli nie sprecyzujemy sami Docker Registry, to obrazy będą domyślnie pobierane z DockerHuba, który ma link docker.io. Na tej podstawie pełne określenie image to np. docker.io/nginx/nginx.

Jeśli chcemy aby nasze obrazy były gdzieś przechowywane, ale nie udostępnione publicznie to trzeba stworzyć prywatne repozytorium, może to być w DockerHubie, ale też w każdej chmurze. Wtedy, aby pobrać obraz najpierw trzeba się zalogować do Dockera komendą:
docker login nazwa_repo.io

Jeśli chcemy stworzyć sami sobie prywatne repozytorium, to też jest taka możliwość i będzie to inny obraz o nazwie registry, który wystawia API dostępne na porcie 5000. Tworzymy je komendą:

docker run -d -p 5000:5000 --name registry registry:2

Aby dodać nowy obraz do tego registry trzeba najpierw dodać tag:

docker image tag nazwa localhost:5000/nazwa_image

I następnie można ten obraz wypchnąć do tego registry:

docker push localhost:5000/nazwa_image

Teraz można też pobrać taki obraz komendą:

docker pull localhost:5000/nazwa_obrazu

Aby sprawdzić obrazy, które są aktualnie w prywatnym repozytorium używamy komendy:

curl -X GET localhost:5000/v2/_catalog

Docker Engine, Storage and Networking

Docker Engine

Docker Engine to tak naprawdę host, na którym jest Docker zainstalowany. W trakcie instalacji tworzone są 3 narzędzia: Docker CLI (Command Line, którego używamy do wykonywania akcji. Korzysta z REST API do komunikacji z Deamonem), REST API Server (używany do komunikacji z Deamonem i przysyłania mu instrukcji) i Docker Daemon (proces działający w tle i zarządzający obiektami takimi jak volume, kontenery, obrazy czy sieci).

Docker CLI może być zainstalowany na innym hoście i może komunikować się zdalnie z REST API, ale trzeba sprecyzować ten zdalny Engine w każdej komendzie poprzez dodanie:

`-H=remote-docker-engine (lub jego IP):2375`

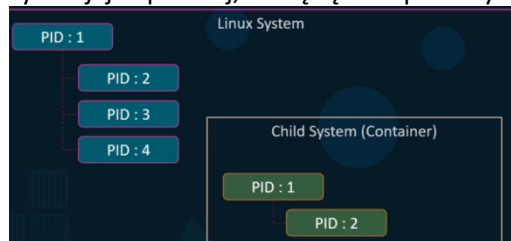
Przykładowa komenda to:

`docker -H=10.123.2.1:2375 run nginx`

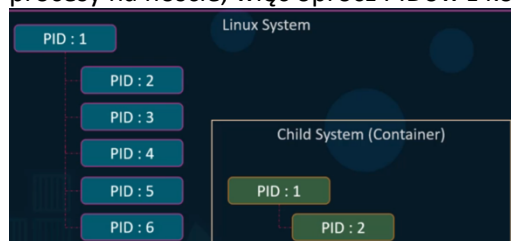
Docker używa różnych namespace, aby odizolować workloady. Proces ID, Mount, Network, InterProcess i Unix Timesharing, wszystko ma swoje namespace, żeby zapewnić izolację pomiędzy kontenerami.

PID namespace

Po uruchomieniu Linuxa, jego proces (root) dostaje PID 1, i on zaczyna uruchamiać inne procesy. Każdy z nich otrzymuje wyjątkowy PID. Jak uruchomimy kontener na hoście to w nim też będą procesy zaczynające się od 1. Wiedząc, że nie ma odizolowania pomiędzy hostem a kontenerem, może dojść do sytuacji jak poniżej, że będą dwa procesy o tym samym PIDzie:



W tym przypadku następuje użycie namespace. Dzięki nim każdy proces może mieć do siebie przypisanych wiele PIDów. Taki proces jest po prostu przypisany do różnych namespace i w nich ma inne PIDy. W praktyce jest tak, że nowe procesy stworzone na kontenerze, to tak naprawdę nowe procesy na hoście, więc oprócz PIDów z kontenera dostaną kolejne PIDy z hosta (tu 5 i 6):



PIDy 1 i 2, które mają z kontenera są widoczne tylko wewnątrz tego kontenera. Kontener ma swoje drzewko procesów. Używając komendy `docker ps` na hoście i w kontenerze możemy odnaleźć i porównać te procesy.

Wiemy, że kontenery i host używają tych samych CPU i pamięci. Domyślnie nie ma ograniczenia tego ile z tych resourców kontener może używać, jeśli jednak chcemy to ograniczyć to trzeba użyć `cgroups` (Control Groups). Możemy je dodać do komendy `docker run np`:

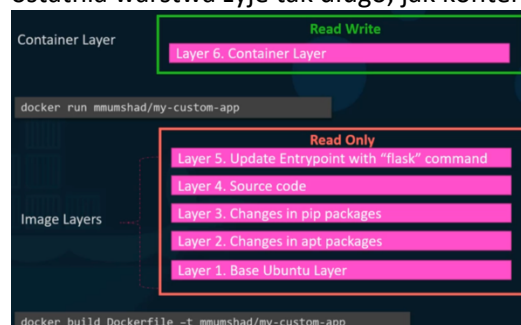
`docker run --cpus=.5 ubuntu` - w tym przypadku ustawiamy, że maksymalnie kontener może używać do 50% dostępnego CPU.

`docker run --memory=100m ubuntu` - tu analogicznie ustawienie maksymalnego zużycia pamięci na 100Mb.

Docker Storage

Domyślnie Docker przechowuje swoje pliki w `/var/lib/docker`. Tutaj mamy foldery aufs, containers, image, volumes, w których znajdziemy pliki obrazów, volume, czy kontenerów.

Jak wiemy, Docker korzysta z warstwowej architektury, więc każda nowa warstwa obrazu ma w sobie zapisane zmiany względem poprzedniej. Po skończeniu buildu, stworzone warstwy są Read Only, a gdy stworzymy kontener na bazie obrazu, to dodana zostanie dodatkowa warstwa kontenerowa, którą możemy modyfikować i zapisywać w niej zmiany. Są tam zapisywane wszystkie pliki danej aplikacji. Ta ostatnia warstwa żyje tak długo, jak kontener, więc jeśli go usuniemy to te dane zostaną stracone.



W przykładzie powyżej mamy kod aplikacji zapisany w obrazie Dockerowym. Jeśli chcielibyśmy zmodyfikować kod źródłowy, to w teorii zmienilibyśmy cały image - należy pamiętać, że ten obraz może być współdzielony pomiędzy inne kontenery. W takim przypadku Docker pomaga nam i umożliwia modyfikację plików zapisanych w obrazie bez żadnych problemów, bo, gdy zaczynamy modyfikować plik z warstwy obrazu, to jest on automatycznie kopiowany do warstwy Read Write - kontenerowej i w praktyce modyfikujemy tą kopię. Jest to nazwane Copy-On-Write.



Jeśli chcielibyśmy móc zachować nasze zmiany z kontenera to trzeba dodać volume do naszego kontenera. Najpierw trzeba go stworzyć komendą:

```
docker volume create nazwa
```

To stworzy folder w `/var/lib/docker/volumes/nazwa`

Teraz jeśli chcemy go podpiąć do kontenera to dodajemy parametr `-v` w `docker run`:

```
docker run -v nazwa_volume:ścieżka_mounta_w_kontenerze nazwa_image
```

Jeśli natomiast chcemy podpiąć inną ścieżkę z hosta do kontenera i tam zapisywać dane to wtedy podajemy ją w komendzie (jest to Bind Mounting):

```
docker run -v ścieżka_na_hoście:ścieżka_mounta_w_kontenerze nazwa_image
```

W nowszych wersjach Dockera zamiast parametru volume dobrze używać parametru mount, który da te same rezultaty, przykładowo:

```
docker run --mount type=bind,source=/data/mysql,target=/var/lib/mysql mysql
```

Wszystkie te operacje oraz tworzenie warstwowej architektury są wykonywane przez Storage Drivers. Znane to AUFS, ZFS, Device Mapper, Overlay. Są one dobierane na podstawie systemu operacyjnego Docker hosta.

docker info | grep Storage Driver - pokaże nam jaki Storage Driver mamy zainstalowany. Np. dla Ubuntu jest to AUFS. Każdy Storage Driver inaczej przechowuje dane i w jego dokumentacji znajdziemy, to jak dane o warstwach są przechowywane.

docker history image_ID - pokazuje, jak dany image powstał krok po kroku z komendami, na tej podstawie można odtworzyć Dockerfile.

docker system df - pokazuje aktualne zużycie dysku przez Dockera bez duplikatów, bo jak uruchomimy *docker images* i tam dodamy rozmiary obrazów, to będzie więcej, bo tam duplikaty są liczone oddzielnie. Jak dodamy do tego flagę *-v* to będziemy mieć rozdzielenie na poszczególne obrazy i w kolumnie SHARED SIZE będzie pokazane ile miejsca z danego obrazu to duplikat innego.

Docker Networking

Gdy zainstalujemy Dockera, tworzy on domyślnie 3 sieci: Bridge (domyślna sieć, do której kontenery są dodawane), None i Host.

Jeśli chcielibyśmy dodać kontener do danej sieci to precyzujemy to w poleceniu:

docker run ubuntu --network=host

Bridge network to prywatna i wewnętrzna sieć tworzona przez Dockera na hoście. Wszystkie kontenery zostają do niej włączone i dostają wewnętrzny adres IP najczęściej z przedziału 172.17.0.0. Kontenery mogą się ze sobą łączyć po tych wewnętrznych adresach IP. Aby dostać się do nich z zewnątrz, trzeba zmapować porty kontenerów z portami na hoście.

Innym sposobem na udostępnienie kontenera na zewnątrz jest dodanie go do sieci Host. Zniknie wtedy izolacja sieciowa pomiędzy hostem i kontenerem, więc jeśli mamy np. web app kontener z portem 5000, to będąc w sieci Host, będzie on dostępny na zewnątrz bez robienia port mappingu. Należy pamiętać, że w tym przypadku nie robiąc port mappingu, nie będzie już można stworzyć innego kontenera z aplikacją webową na porcie 5000.

Jeśli dodamy kontener do sieci None, to znaczy, że nie będzie on w rzeczywistości dodany do żadnej sieci i nie będzie mógł się łączyć z innymi kontenerami i z internetem.

Jeśli chcielibyśmy stworzyć samemu drugą Bridge network, aby rozdzielić w nie pody to tworzymy to komendą:

docker network create --driver bridge --subnet 182.18.0.0/16 --gateway IP_gatewaya nazwa_sieci

docker network ls - wyświetla wszystkie sieci na hoście.

docker inspect nazwa_sieci - pokaże nam szczegółowe informacje o sieci, między innymi jej typ, adres IP itp.

Wszystkie kontenery w sieci Bridge mogą się ze sobą łączyć po swoich nazwach (mogą też po adresach IP, ale te się mogą zmienić). Przykładowo do połączenia się do bazy danych można użyć komendy: *mysql.connect(nazwa_kontenera)*. Dzieje się tak dlatego, że Docker ma wbudowany swój DNS server, który przechowuje te informacje. Domyślnie działa on pod adresem IP: 127.0.0.11.

Aby umożliwić izolację poszczególnych kontenerów, Docker używa Network Namespaces. Każdy kontener otrzymuje swój namespace i potem Docker używa Virtual Ethernet Pairs do łączenia ich ze sobą.

docker network connect(disconnect - analogicznie to rozłączanie) nazwa_sieci nazwa_kontenera - łączenie danego kontenera do danej sieci.

Jak połączymy tym poleceniem kontener, który jest połączony do innych sieci, to nie straci on tych połączeń, tylko będzie miał ich więcej. Teraz po połączeniu dwóch kontenerów do naszych sieci możemy się z nimi komunikować poleceniem ping po nazwie.

Jeśli dwa kontenery nie są w tej samej sieci to nie można się pomiędzy nimi komunikować i łączyć. Po połączeniu do sieci w kontenerze automatycznie będzie stworzony interfejs sieciowy.

Container Orchestration

Container Orchestration

Container orchestration pomaga nam w automatyzacji naszego działania z Dockerem. Działając z samym Dockerem, wiele rzeczy musimy robić samemu manualnie np. jak mamy pojedynczą aplikację i ona już nie daje rady, bo ma za duży workload, to musimy sami stworzyć nowe instancje żeby rozdzielić ruch. Dodatkowo sami musimy monitorować zdrowie tych instancji i jak coś się wydarzy to tworzyć nowe. To samo tyczy się samego Docker hosta, a jeśli dodatkowo mielibyśmy setki kontenerów i kilka hostów, to byłoby bardzo ciężko to ręcznie zarządzać.

Container orchestration pomaga nam w tych wszystkich problemach. Ma ona najczęściej kilka wspólnie pracujących Docker hostów, tak, aby jeśli jeden padnie, to praca dalej była kontynuowana. Pozwala to także tworzyć setki instancji naszej aplikacji jedną komendą. Nowe hosty, czy nowe instancje aplikacji są tworzone, automatycznie np. na podstawie zapotrzebowania i ilości przychodzącego ruchu.

Container orchestration ma także zaawansowane rozwiązania sieciowe dla kontenerów, dodatkowo zapewnia load balancing pomiędzy nimi i storage sharing.

Najpopularniejszym tooliem do orchestration jest Kubernetes, jest też Docker Swarm, ale nie ma on wszystkich tych możliwości, które daje Kubernetes.

Docker Swarm

Pozwala on na stworzenie clustra Dockerowego z wielu hostów. Będzie on rozdzielał aplikacje na różne hosty, aby zapewnić high availability i load balancing.

Aby go zainstalować, trzeba mieć już kilka hostów z zainstalowanym Dockerem i wybrać jednego z nich na Swarm Managera, a inne będą Workerami. Potem trzeba na masterze uruchomić komendę:

docker swarm init - inicjuje ona Docker Swarm

W outpucie będzie pokazana inna komenda, którą trzeba uruchomić na workerach. Będzie ona postaci:

docker swarm join --token token

Teraz jak już mamy cluster to można tworzyć instancje naszej aplikacji. Jeśli chcemy stworzyć wiele z nich to oczywiście można robić *docker run* na każdym z nodów, ale to dalej jest manualne rozwiązanie. Zamiast tego użyjemy komendy:

docker service create --replicas=3 nazwa_image

Service to podstawowa jednostka w Docker Swarm i może ona się odnosić do jednej lub wielu instancji naszej aplikacji. Należy pamiętać, że tak komenda musi być uruchomiona na masterze. Komenda *docker service create* jest identyczna z komendą *docker run* pod względem parametrów, które możemy przekazać.

Kubernetes Introduction

Kubernetes też zapewnia możliwości tworzenia setek instancji aplikacji na raz i automatycznie je rekreuje, gdy się zepsują. Dodatkowo może dodać nody, gdy użycie jest zbyt duże. Jeśli chcemy robić update aplikacji to pozwala automatycznie robić update np. kilku instancji, tak aby inne cały czas działały i użytkownicy nie odczuli zmiany. Dodatkowo, jak coś pójdzie nie tak, to można te zmiany łatwo cofnąć.

Kubernetes jest wspierany przez wiele różnych sposobów autentykacji, przez wiele Network Pluginów, czy rodzajów Storage. Dodatkowo jest on dostępny w publicznej chmurze takiej jak AWS, Azure czy GCP.

Kubernetes używa Dockerowych hostów do przechowywania aplikacji w formie kontenerów Dockerowych. Jego typowa infrastruktura jest podobna do Docker Swarma: jest master node i kilka worker nodów, na których są instancje aplikacji i master nimi zarządza. Na masterze mamy zainstalowane wiele tooli, które pomagają w orkestracji kontenerów.

Aby wejść w interakcję z Kubernetesem i tworzyć aplikacje używamy Kubernetes CLI - kubectl, przykładowo *kubectl run nazwa_image*.