Kurs Dockera

Łukasz Koszyk

Gigaset, Wrocław, Kompilacja: 1 sierpnia 2019, 15:43

Spis treści

Wprowadzenie

Niniejszy podręcznik powstał na potrzeby szkolenia z dockera, od zera, do bohatera. Albo przynajmniej do sprawnego użytkownika. Jest to w zasadzie uzupełnienie warsztatów, pozwalające uporządkować sobie wiedzę, z ćwiczeniami. W razie znalezionych błędów, proszę o informację.

1. Po co docker?

Po co powstał docker? Można przecież wszystko na fizolu postawić, a jak ktoś chce odseparować procesy, to na wirtualce. Wystarczy zainstalować paczkę, skonfigurować zgodnie z krótkim poradnikiem, i działa. A jak się zna Puppeta, Salta czy Ansible'a, to jest to jeszcze prostsze. Wdrażanie bazy danych w dockerze? Aplikacji? Ale po co? Przecież nie można tu już użyć gotowych modułów do wspomnianego Puppeta czy Salta. Trzeba samemu zatroszczyć się o wszelkie inne rzeczy, które normalnie są robione w cronie lub przez inne usługi, a wszystko jest instalowane z paczki, jak rotowanie logów, cykliczne odpalanie jakiegoś sprzątania itd. Po co to w ogóle powstało? Do czego to się może przydać?

1.1. Wirtualizować czy nie?

Przed epoką konteneryzacji były dwa rozwiązania do wyboru: uruchomić procesy po prostu w systemie operacyjnym (na fizolu lub w wirtualce) lub też każdy z osobna w swojej wirtualce. Pierwsze rozwiązanie to typowa prosta instalacja wszystkich usług. Zazwyczaj działa. Problem pojawia się, gdy trzeba postawić kilka instancji takiej aplikacji. Kupować kilka komputerów? A co z zapewnieniem spójności konfiguracji? Trzeba użyć automatu typu Puppet, Chef. Skala skomplikowania rośnie, gdy trzeba zaktualizować system operacyjny do nowszej głównej wersji, albo jedną usługę: mamy dwie aplikacje, obie z zależnościami do konkretnej biblioteki w konkretnej wersji. Aktualizacja jednej z nich zmienia to, że wymaga wspomnianej biblioteki w innej wersji, i się robi konflikt. Albo inny przykład - wszędzie mamy CentOS, a jedną aplikację potrzebujemy z jakichś przyczyn postawić na Debianie, albo wszędzie mamy Centos 7, a jedna aplikacja wymaga tak nowych bibliotek, że tylko najnowsza Fedora dostarcza takich paczek. Po to wymyślono tzw. „chroot jaila", czyli w jakimś swoim katalogu, powiedzmy `/opt/app` należało zainstalować cały system operacyjny z użyciem febootstrap, debootstrap itp, podlinkować niezbędne urządzenia do `/dev`, `/sys` i `/proc`, potem `chroot /opt/app`, i można uruchamiać aplikację. Wada? Utrzymanie aktualności, kłopotliwe backupowanie i przenoszenie takiego wynalazku między maszynami, czy zapewnianie spójności tych systemów między maszynami. Rsyncem? Czy może Puppetem? A jak Puppet to gdzie go odpalać? Najlepiej w chroocie, żeby nie trzeba było pisać swoich modułów. A może pakować tarem i wersjonować? Albo walnąć wszystkie pliki w rpmkę i czekać godzinę na yum install?

Z drugiej strony mamy do wyboru wirtualki, czyli prawdziwy osobny system operacyjny, bez linkowania i udawania. Noo prawie bez udawania. Łatwiej się przenoszą między fizycznymi hostami, bo cały system plików to jeden plik na hoście, choć to czasem dziesiątki gigabajtów, backupowanie tego nadal jest mało przyjemne, a aktualizacja o oczko łatwiejsza od pierwszego przypadku. Wady? Niepotrzebne zużywanie RAMu i czasu procesora na system dookoła aplikacji. W zasadzie bardzo duże marnotrawstwo zasobów. Dla jednej jedynej aplikacji chodzi kilkanaście-kilkadziesiąt usług w maszynie wirtualnej, trzeba jeszcze dać maszynie zapas na zadania w tle. Zapewnienie spójności konfiguracji między maszynami również wymaga użycia automatu, typu Puppet, Chef, Ansible. Dostawianie instancji kosztuje trochę czasu, gdy się ma kickstart, lub sporo pracy, gdy się go nie ma.

No więc jak pogodzić jedno z drugim? Bo mimo wszystko chcemy postawić łatwo i szybko kilka instancji aplikacji, tak jak łatwo i szybko instaluje się paczkę. Ale też bez szarpania się z konfiguracją za każdym razem. Chcemy oszczędzać zasoby, ale jednocześnie mieć separację na systemie plików i między procesami. I chcemy to zrobić na jednej maszynie, ale tak, żeby łatwo się przenosiło na inne. Backupować chcemy automatycznie tylko tyle ile jest potrzebne. Na dodatek wdrożenie

ma być szybkie i łatwe, bo nie ma czasu, mamy ważne sprawy do załatwienia z kolegami przy piłkarzykach. Utopia? Nieeee... LXC!

1.2. LXC a Docker

Na powyższe potrzeby odpowiedzią jest LXC. To jest taki chroot, ale nie do końca, taka paczka, ale też nie do końca, osobny system operacyjny, ale bez swojego kernela. Czyli takie coś pomiędzy wirtualką, „chroot jailem". Jedyne, czego nie ma, to wygoda. Nadal trochę pokraczne i upierdliwe. Owe wady zauważyli deweloperzy i napisali Dockera, jako nakładkę na LXC.

LXC jest więc funkcją samego Kernela, a Docker to oprogramowanie, które jest frontendem do tego. Tu krótko i zwięźle na ten temat: `https://www.upguard.com/articles/docker-vs-lxc`.

Najważniejsze cechy Dockera:

- szybki we wdrożeniu - używa się publicznego repozytorium z gotowymi obrazami tzw. docker huba, lub swojego repozytorium, czyli docker registry
- obrazy są oszczędne - każda zmiana w systemie plików to osobna skompresowana warstwa obrazu, warstwy mogą być współdzielone między różnymi obrazami, jeśli dwa różne obrazy mają ten sam rdzeń (np. goły system operacyjny) to jest on współdzielony zawsze
- tworzenie nowych obrazów od zera jest tak samo łatwe jak chroot, tworzenie własnych na bazie jakiegoś innego jest bardzo proste i przyjemne
- procesy są od siebie odseparowane (czyt. kernel namespace `https://en.wikipedia.org/wiki/Linux_namespaces`)
- każdemu kontenerowi można ściśle ustawić zasoby - docker używa tutaj mechanizmu cgroup:
  - `https://www.cloudsigma.com/manage-docker-resources-with-cgroups/`
  - `https://wiki.archlinux.org/index.php/Cgroups`
  - `http://0pointer.de/blog/projects/resources.html`
- każdemu kontenerowi można łatwo dostarczyć pliki z hosta montując katalog do środka kontenera
- kontenery mogą się łatwo ze sobą komunikować przez sieć, bardzo łatwo wystawia się z nich usługi na konkretnych portach (przez sockety również)

2. Podstawy obsługi

2.1. Przydatne polecenia

- `docker ps` - lista pracujących kontenerów
- `docker images` - lista pobranych na dysk obrazów
- `docker ps -a` - lista pracujących + padłych kontenerów
- `docker images -a` - lista pobranych na dysk obrazów + ich wszystkich warstw pośrednich
- `docker rm` - usuwa kontener
- `docker rmi` - usuwa obraz
- `docker run -it <obraz>:<tag> <polecenie>` - uruchomienie polecenia w kontenerze
- `docker run -dit <obraz>:<tag> <polecenie>` - uruchomienie polecenia w kontenerze, w tle

- `docker kill <id kontenera>` - ubija kontener
- `docker inspect <id kontenera/obrazu>` - szczegóły dot. kontenera/obrazu
- `docker logs <id kontenera>` - to, co główny proces wypisuje na standardowe wyjścia
- `docker logs -f <id kontenera>` - jak wyżej + śledzenie na żywo, odpowiednik `tail -f`

2.2. Odpalamy pierwszy kontener

W tym celu użyjemy gotowy obraz systemu Centos 7 i sprawdzimy datę:

```
docker run -it centos:7 date
```

Wypisał datę. To teraz sprawdzimy czy chodzi kontener: `docker ps` i jest pusto... `docker ps -a` pokazuje że jest jeden padły kontener. Czemu tak się stało? Jest to pierwsza ważna informacja, którą należy wiedzieć: **kontener żyje tak długo, jak długo żyje proces, z którym go uruchomiliśmy**. Teraz zróbmy tak, żeby kontener żył dłużej:

```
docker run -it centos:7 top
```

Następnie wykonując `docker ps` w sąsiedniej konsoli widać kontener. Ale jest kolejna dziwna rzecz: top to program pokazujący między innymi procesy. A tu jest tylko jeden proces - top. Popsute? Niieee... to druga ważna rzecz: **typowo w kontenerze chodzi tylko jeden proces - ten, z którym go uruchomiliśmy**. Oczywiście są od tego odstępstwa[1], ale typowo jest jeden proces. Zauważ, że nie ma tam nawet procesu init, więc SysVinit czy SystemD również nie ma, więc systemctl nie będzie działać w ogóle, a skrypty z `/etc/init.d` różnie. Kontener top nam dalej żyje, można go ubić wciskając klawisz `Q` w topie, albo przećwiczyć kolejną ważną komendę:

```
docker kill <id kontenera>
```

to ID bierzemy z pierwszej kolumny z docker ps (hash) lub z ostatniej kolumny (name). Docker dba, żeby name i hash były unikalne na danej maszynie. Teraz `docker ps -a` pokaże nam kolejny padły kontener z innym exit kodem. Sekcję zwłok można zrobić poleceniem

```
docker inspect <id kontenera>
```

wtedy dostajemy bardzo szczegółową informację. Korzysta się z tego rzadko, ale warto znać. Częściej patrzy się go logów:

```
docker logs <id kontenera>
```

które z racji użytego programu nie pokazują nic interesującego.

2.3. Zadania

Zad. 1

Usuń wszystkie padłe kontenery (Odp: 1)

Zad. 2

Usuń obraz centos:7 (Odp: 2)

Zad. 3

Uruchom w tle pętlę uruchamiającą co sekundę polecenie `date` (Odp: 3)

[1] proces główny jest forkujący np. apache czy nginx, albo jest uruchomiony inny proces z użyciem docker exec np. z crona, żeby coś posprzątać. Osobną kwestią jest budowanie wieloprocesowych kontenerów, omówione pod koniec

3. Informacje systemowe

Ile docker zajmuje miejsca? Ile zajmuje kontener? Trudno na te pytania znaleźć odpowiedzi, chyba że skorzystamy z `docker system https://docs.docker.com/engine/reference/commandline/system`. Alternatywą jest użycie docker history i ręczne sprawdzanie zajętości bezpośrednio w systemie plików.

4. Własny obraz

4.1. Przydatne polecenia

- `docker pull <obraz>:<tag>` - pobranie obrazu na dysk
- `docker push <obraz>:<tag>` - wysłanie obrazu do rejestru dockerowego/docker huba
- `docker tag <id obrazu> <obraz>:<tag>` - otagowanie obrazu
- `docker commit <id kontenera>` - przekształcenie kontera w obraz, jako rezultat docker zwraca ID obrazu w formie hasha
- `docker build .` - buduje z pliku Dockerfile, zwraca hash
- `docker build -f plik .` - buduje z pliku `plik`, zwraca hash
- `docker build -t mojerepo/obraz:tag -f plik .` - buduje z pliku `plik`, taguje jak chcesz
- `docker history mojerepo/obraz:2` - pokazuje historię obrazu oraz rozmiary poszczególnych warstw

4.2. Własny obraz - metoda ręczna

Będzie to nie do końca własny obraz, bo użyjemy bazowego, ale reszta będzie już nasza. Uruchamiamy sobie basha w kontenerze:

```
docker run -it centos:7 bash
```

w efekcie docker zwróci nam terminal w kontenerze. Zainstalujmy sobie vima `yum install -y vim`, i załóżmy plik `/moj_plik` z jakąś zawartością. Z kontenera wychodzimy poleceniem `exit` lub kombinacją klawiszy `CTRL+D`. Używając polecenia `docker ps -a` znajdujemy ID kontenera, z którego przed chwilą wyszliśmy. Przerabiamy go na obraz:

```
docker commit <ID>
```

w efekcie docker zwróci nam ID obrazu. Poleceniem `docker images` można sprawdzić, że rzeczywiście on tam jest. I to jest nasz obraz. Możemy go użyć w następujący sposób:

```
docker run -it <ID obrazu> bash
```

i sprawdzić vimem, że rzeczywiście jest tam plik `/moj_plik` z odpowiednią zawartością. Żeby mieć taką ładną nazwę, jak centos, musimy swój obraz otagować. Wychodzimy z kontenera i tagujemy obraz poleceniem:

```
docker tag <ID obrazu> <nazwa obrazu>:<wersja>
```

w efekcie poleceniem `docker images` zobaczymy ID obrazu właściwie otagowane. Wykonując teraz polecenie:

```
docker push <nazwa obrazu>:<wersja>
```

wypchnęlibyśmy obraz do rejestru dockerowego, gdybyśmy go mieli, lub mieli uprawnienia do docker huba. Ale na to przyjdzie czas.

4.3. Własny obraz - dockerfile

Metoda ręczna jest dobra do jednorazowego tworzenia obrazu, szczególnie, gdy nie potrzebujemy mieć dokumentacji ani źródeł. Zazwyczaj ich trzeba, więc dobrze mieć zapisane gdzieś te kroki, które wykonujemy w kontenerze. Można to zrobić na dwa sposoby - napisać sobie w zeszyciku, albo stosując dockerfile. Tak jak RPMka ma SPECa, program w C źródła, tak obraz dockerowy ma swój dockerfile. Tutaj głębsze omówienie: `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`. Jest kilka podstawowych poleceń w dockerfile:

- FROM
- MAINTAINER
- RUN
- ADD
- COPY
- ENV
- CMD
- ENTRYPOINT
- WORKDIR
- USER
- VOLUME

ładnie omówione tutaj: `https://www.howtoforge.com/tutorial/how-to-create-docker-images-with-dockerfile/`. Oczywiście nie wszystkich trzeba użyć. Przepisując procedurę z poprzedniego punktu mamy:

1. `docker run -it centos:7 bash` ⇒ `FROM centos:7`
2. `yum install -y vim` ⇒ `RUN yum install -y vim`
3. wpisanie jakieś wartości do pliku ⇒ `RUN echo "jakies znaczki" > /moj_plik`

Trzeba pamiętać o jednej ważnej rzeczy: polecenia interaktywne (wymagające jakiegoś potwierdzenia, podania danych) trzeba napisać tak, żeby tego nie potrzebowały. W efekcie nasz Dockerfile wygląda tak:

```
1 FROM centos:7
2 RUN yum install -y vim
3 RUN echo "jakies znaczki" > /moj_plik
```

Listing 1: Dockerfile

Mamy dockerfile, można budować. Jest kilka podejść do tego:

- `docker build .` - szuka pliku Dockerfile, zwraca hash
- `docker build -f plik .` - szuka pliku `plik`, zwraca hash
- `docker build -t mojerepo/obraz:tag -f plik .` - szuka pliku `plik`, taguje jak chcesz

Także nie trzeba się męczyć, można sobie wejść do katalogu `listingi`, i chlasnąć polecenie:

```
docker build -t mojerepo/obraz:1 -f l1 .
```

4.3.1. A co jeśli źle otaguję?

Jeśli otagujesz i wyślesz do rejestru dockerowego to musisz doczytać dokumentację rejestru i usunąć błąd według niej. Nie zawsze jest to możliwe niestety. Jeśli jeszcze nie wysłałeś, to:

1. zweryfikuj to używając `docker images`
2. wykonaj `docker tag hash_obrazu nowy_tag`
3. zweryfikuj to używając `docker images`
4. jeśli nowy tag istnieje to `docker rmi stary_tag`

4.3.2. Dodajemy plik

Powiedzmy, że chcemy dodać do środka jakiś plik i `echo` nas nie urządza. Wówczas mamy dwa wyjścia - montowanie wolumenu, o którym później, lub kopiowanie pliku przy docker build. Utwórzmy więc plik `plik1`, a do dockerfile trzeba dodać odpowiednią linijkę. I tu problem, bo jest `COPY` i `ADD`. Różnica jest następująca: `COPY` to zwykłe kopiowanie. Tak, jak to sobie każdy wyobraża. Natomiast `ADD` to takie `COPY` na kwasie: z możliwością kopiowania z URLa i rozpakowania archiwum tar w środku. Jak człowiek weźmie nie to co trzeba i użuje `ADD` to może zobaczyć różne niespodziewane i dziwne rzeczy. Dlatego dokumentacja oficjalnie zaleca `COPY` i ewentualnie potem jakieś `RUN` które rozpakuje archiwum, jest to bardziej czytelne. Tutaj szczegóły: `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#add-or-copy`. Więc wstawiamy:

```
1 FROM centos:7
2 RUN yum install -y vim
3 RUN echo "jakies znaczki" > /moj_plik
4 COPY plik1 /tmp/plik1.txt
```

Listing 2: Dodawanie pliku

Budujemy więc nowy obraz:

```
docker build -t mojerepo/obraz:2 -f l2 .
```

Ciekawostka: jeśli przed chwilą budowałeś poprzedni obraz, to zwróć uwagę, że tym razem docker nie odpalał yuma itd. Skorzystał z istniejących warstw i dobudował kolejną z plikiem, która zajmuje 129 bajtów. Skąd to wiem? Ano stąd:

```
docker history mojerepo/obraz:2
```

gdzie dostaję:

```
1 IMAGE               CREATED             CREATED BY
                  SIZE                COMMENT
2 8304fdae8455        7 minutes ago       /bin/sh -c #(nop) COPY file:
    eff561b96a9661...   129 B
3 77ced84e2f5e        2 hours ago         /bin/sh -c echo "jakies znaczki" >
    /moj_plik    15 B
4 635483824ead        2 hours ago         /bin/sh -c yum install -y vim
                  160 MB
5 9f38484d220f        4 months ago        /bin/sh -c #(nop)  CMD ["/bin/bash
    "]            0 B
6 <missing>           4 months ago        /bin/sh -c #(nop)  LABEL org.label-
    schema....   0 B
7 <missing>           4 months ago        /bin/sh -c #(nop) ADD file:074
    f2c974463ab3...   202 MB
```

Listing 3: Dodawanie pliku

Zad. 4

Dodaj dockerfilem archiwum tar do obrazu poleceniem COPY, a następnie rozpakuj je do /root. (Odp: 4)

5. Montowanie wolumenu

Na początek różnica między `COPY`, `ADD`, a montowaniem wolumenu. `COPY` i `ADD` tworzą kopię pliku, umieszczają go w warstwie i pakują razem z obrazem. Dzieje się to na etapie budowania obrazu i zostaje tam już na zawsze[2]. Montowanie natomiast odbywa się dopiero na etapie uruchamiania kontenera, wolumen można zamontować, ale nie trzeba. Plik (lub katalog) może dowolnie zmieniać zawartość, może być modyfikowany i od strony hosta, i od strony kontenera. Po zabiciu kontenera te modyfikacje zostają. Zestawiając te informacje mamy:

	COPY/ADD	Wolumen
Kiedy?	Budowanie obrazu	Uruchamianie kontenera
Gdzie jest plik?	Na jakiejś warstwie w obrazie	Na hoście
Persystencja	Utrata zmian po restarcie kontenera	Tak
Modyfikacje	Łatwo - nie, wymaga to sporo zachodu	Tak, przezroczyste dla aplikacji
Przypadkowe skasowanie	Nie, chyba że z całym komputerem	Tak

Typowo w formie wolumenów dostarcza się dane, które muszą być zachowanie po restarcie oraz konfigurację zarządzaną jakimś automatem. Można też w ten sposób dostarczać jakiś szybki storage na pliki tymczasowe np. ramdisk.

Wolumen dodajemy przełącznikiem `-v`:

```
docker run -it -v /tmp:/aa centos:7 bash
```

Wówczas w kontenerze mamy katalog `/aa` z podmontowanym `/tmp` z hosta. Jeśli katalog w kontenerze nie istnieje, to go utworzy. Jeśli montujemy na istniejącym, ale zawierającym dane, wtedy będą widoczne tylko te z podmontowanego wolumenu.

Inną metodą jest użycie docker volumes. Różnica między powyższą metodą (bind volume) a docker volumes, jest w sposobie zarządzania lokalnym katalogiem. Docker volumes robi to sam, w określonej ścieżce na hoście. Umożliwia też użycie innych typów przestrzeni dyskowych niż plikowa, np. NFS. Szczegóły tutaj: `https://docs.docker.com/storage/volumes/`

Zad. 5

Załóż sobie dwa katalogi i zamontuj obydwa jako wolumeny do kontenera. (Odp: 5)

6. Zmienne środowiskowe

Temat krótki i przyjemny. Konfigurację możemy dostarczać wolumenem za pośrednictwem pliku, lub zmiennymi środowiskowymi. Można to zrobić dwojako - na etapie budowania kontenera dodając w dockerfile taką linijkę:

```
ENV zmienna="wartosc"
```

lub przy uruchamianiu kontenera:

```
docker run -it -v /tmp:/aa -e zmienna="wartosc" centos:7 bash
```

Efekt jest taki sam. Zweryfikować to można w taki sposób mając shella w kontenerze:

```
echo $zmienna
```

[2] tzn. do pierwszego rm -rf w złym miejscu, i wywalenia sobie rejestru dockerowego oraz plików dockerowych na swoim kompie

Zad. 6

Napisz skrypt w bashu lub pythonie, który będzie wyświetlał na ekran co sekundę wartość zmiennej środowiskowej `zm_2`. Skrypt ma być w formie pliku wykonywalnego. Uruchom kontener w tle z tym skryptem, zmienna środowiskowa na początku ma mieć wartość 0. Następnie zmień wartość zmiennej na inną. Sprawdź w logach kontenera, czy tak się stało i wyjaśnij dlaczego nie :) Wskazówka: `docker exec -it d0c4d38a4501 bash -c "echo $zm_2"` odpalone równolegle z chodzącym kontenerem jest puste, a `docker exec -it d0c4d38a4501 bash -c 'echo $zm_2'` pokazuje 0. (Odp: 6)

Zad. 7

Napisz powyższy skrypt z drobną modyfikacją: wartość zmiennej ma odczytywać z pliku montowanego z hosta. (Odp: 7)

7. Grzebanie w chodzącym kontenerze

Czasem trzeba pogrzebać w chodzącym kontenerze, np. wprowadzić zmianę w jakimś pliku, coś sprawdzić, uruchomić jakieś polecenie. W tym celu można użyć `docker exec`. Załączmy sobie kontener w tle:
`docker run -dit centos:7 top`

załączmy sobie wyświetlanie logów w sąsiedniej konsoli:
`docker logs -f <id kontenera>`

i załączmy sobie teraz shella w kontenerze:
`docker exec -it <id kontenera> bash`

Spójrz teraz do logów - przybył jeden proces: bash. I teraz możemy zrobić swoje ważne rzeczy, a jak skończymy, wystarczy `exit` lub `CTRL+D`. Uruchommy w tym kontenerze jakiś inny proces w tle:
`docker exec -dit <id kontenera> sleep 10000`

przybył kolejny proces - widać go w wyjściu z topa. Zabijmy teraz ten proces:
`docker exec -dit <id kontenera> kill -9 <id procesu sleep>`

tak też można pracować, nie trzeba odpalać basha. Proces sleep zniknął, załączmy go jeszcze raz:
`docker exec -dit <id kontenera> sleep 10000`

a następnie zabijmy proces top: `docker exec -dit <id kontenera> kill -9 1`

i dupa :), nic się nie stało. To dlatego, że proces o id 1 nie przyjmuje sygnałów `SIGKILL`. Taka ciekawostka. Można oczywiście zabić kontener używając `docker kill`, ale my byśmy chcieli tylko topa ubić, a sleepa zostawić. Trzeba to inaczej zrobić. Na hoście potrzebujemy konsoli roota. Szukamy procesu top:
`root 6377 0.0 0.0 56172 3776 pts/3 Ss+ 17:06 0:00 top`

Teraz mogę wykonać: `kill -9 6377`, czego efektem będzie zabicie kontenera, proces sleep też zostanie zabity. Dlaczego to było tym razem możliwe? Ponieważ root na hoście pracuje w innym kontekście, dla niego PID 1 ma systemd, a ten top z kontenera to był PID 6377. Dlatego to było możliwe. Stąd też płynie ważna lekcja - wszystko zależy od kontekstu, a procesy pracujące w kontenerach nie pracują w próżni, są widoczne na hoście jako zwykłe procesy.

Zad. 8

Dlaczego powyższy kontener został zabity? Przecież tam chodził proces sleep. Docker nie powinien poczekać aż ten sleep też się zakończy? (Odp: 8)

8. Sieć i komunikacja z usługą w kontenerze po porcie

Sieć w dockerze jest już odrobinę bardziej skomplikowanym tematem. Przy czym wiedza ta nie jest potrzebna do uruchomienia kontenera, ale konieczna jest do debugowania. Więc warto conieco wiedzieć. Są różne sterowniki sieci dla dockera: `https://docs.docker.com/network/#network-drivers`, jednak zazwyczaj używa się bridge'a, i jest też to domyślny sterownik. Po krótce: docker rozpina własną sieć na hoście, domyślnie 172.17.0.0/16, jest to widoczne jako osobny interfejs sieciowy (na hoście oczywiście, typowo docker0). Każdy z kontenerów ma automatycznie przydzielany IP z tej sieci. Komunikacja świat-kontener i host-kontener jest możliwa dzięki iptables i utworzonym tam regułom SNAT oraz DNAT i forwardowania, czyli komunikacja sieciowa obganiana jest firewallem i jest obustronnie maskowana. Coś jak forwardowanie portów na routerze do maszyny w podsieci. Patrząc na hosta od zewnątrz, nie do odróżnienia jest, czy na jakimś porcie odpowiedziała usługa chodząca w kontenerze, czy na hoście[3], jest to kompletnie przezroczyste. Różnica bywa natomiast na hoście. Standardowo łącząc się na jakiś port do samego siebie używamy localhosta, czyli sieci 127.0.0.0/16 (najczęściej jest to IP 127.0.0.1), natomiast chcąc podłączyć się do kontenera chodzącego lokalnie, może być konieczne wybranie IP hosta w sieci dockerowej. Można go odczytać poleceniem `ip a`, typowo jest to pierwszy IP w danej sieci, czyli w typowej sieci to będzie 172.17.0.1. Od pewnej wersji docker ma dodane regułki w firewallu na słuchanie na localhoście. Można to wyłączyć globalnie w dockerze, albo przybindować do konkretnego IP, więc nie zdziw się, gdy połączenie na localhost nie zadziała. Tyle wiedzy wystarczy w 99.9% przypadków z jakimi się spotkacie.

8.1. Typowe błędy popełniane w kontekście sieci

- łączenie się nie na ten IP, co trzeba
- restartowanie usługi iptables przy chodzącym dockerze - zazwyczaj powoduje to wyczyszczenie stanu firewalla, komunikacja sieciowa z kontenerem jest wtedy niemożliwa. Trzeba wtedy zatrzymać kontenery, zrestartować dockera i uruchomić jeszcze raz kontenery
- stawianie kontenera na bramie z zamiarem limitowania ruchu do kontenera
- oczekiwanie braku opóźnień po komunikacji sieciowej (np. między nginxem, a php-fpm). W aplikacjach rozproszonych należy uwzględniać opóźnienia na sieci. Czasem 1-2ms może być już zbyt dużym opóźnieniem. Chcąc je zminimalizować, należy zrezygnować z łączenia między np. nginxem, a php-fpm po sieci, usługi powinny wówczas łączyć się socketem.

8.2. Zabezpieczanie

Po pierwsze, i najważniejsze - nie stawiamy usług w dockerze na bramie. Nigdy, przenigdy, choćby skały srały, a madka Grażyna z horom curkom Dżesiką płakały. Niemożliwe jest sensowne, automatyczne zabezpieczenie firewallem tak postawionego kontenera. Oczywiście da się to zrobić bieda-skryptami chodzącymi w cronie co minutę, ale utrzymanie takiego wynalazku to koszmar. Pamiętaj - kontenery typowo mają IP losowane, a docker sam sobie uzupełnia firewalla. Na bramie możemy postawić cokolwiek robiące za reverse-proxy, dockera natomiast stawiamy zawsze na hoście dostępnym wyłącznie z wnętrza infrastruktury. Reverse proxy dostarcza API na zewnątrz, i w miejscu backendu ma wpisany host i port, na którym kontener jest osiągalny. Między tymi dwiema warstwami można dodać trzecią - warstwę HA, czyli jakieś haproxy, które lepiej będzie zarządzać połączeniami do backendów, i zapewni większą dostępność.

8.3. Praktyka

Wiedząc to wszystko, postawimy swój pierwszy kontener z usługą dostępną sieciowo. Będzie to nginx. Idziemy do docker huba po oficjalny obraz `https://hub.docker.com/_/nginx`:

[3] o ile usługa nie dostarczy informacji systemowych - oczywiście ona sama jest w stanie zweryfikować czy chodzi w kontenerze, czy normalnie w systemie operacyjnym, choćby po liście procesów, albo po numerze jej procesu

```
docker run -dit -p 32080:80 nginx:latest
```

dochodzi nam `-p 32080:80`, czyli `<port na hoscie>:<port w kontenerze>`. Brakuje nam tylko polecenia. Jak się wczytać okazuje się, że go nie trzeba. Jak to? Ano w dockerfile zdefiniowali już `CMD https://github.com/nginxinc/docker-nginx/blob/1.17.1/stable/stretch/Dockerfile#L103`. Tak, dobrze widzisz, skrypt zaczyna się w dziewiątej linijce i jest tzw. jednolinijkowcem. Pozostaje się cieszyć, że nie my to utrzymujemy :) Wracając - uruchamiamy kontener w takiej postaci, nie podajemy mu katalogu z plikami, które ma serwować. Niech serwuje domyślne. Jak już uruchomimy kontener, można w przeglądarce wpisać:
`http://localhost:32080`
lub
`http://172.17.0.1:32080`

zabijmy ten kontener i postawmy jeszcze raz z drobną modyfikacją:
`docker run -dit -p 172.17.0.1:32080:80 nginx:latest`

czyli z IP, na którym ma słuchać (domyślnie słucha na 0.0.0.0, czyli wszystkich) i ponownie spróbuj
`http://localhost:32080`

nie działa, prawda? Ale na
`http://172.17.0.1:32080`
już działa. Jak rozpoznać do jakiego IP kontener jest przybindowany? Używając `docker ps`

```
1 CONTAINER ID        IMAGE               COMMAND                  CREATED
              STATUS              PORTS                          NAMES
2 28123dc50fdc        nginx:latest        "nginx -g 'daemon ..."   3 seconds
    ago       Up 2 seconds        172.17.0.1:32080->80/tcp
    stupefied_wright
```

Listing 4: IP bind

Zad. 9

Załącz kontener z nginxem słuchającym na takich dwóch IP: 172.17.0.1 oraz na IP hosta które dostajesz z DHCP(Odp: 9)

9. Zasoby

W dockerze w linuksie można ciąć zasoby dwiema metodami: cgrupami

- `https://www.cloudsigma.com/manage-docker-resources-with-cgroups/`
- `https://wiki.archlinux.org/index.php/Cgroups`
- `http://0pointer.de/blog/projects/resources.html`

oraz podczas uruchamiania kontenera. Zajmiemy się tym drugim, a w zasadzie zakończymy linkiem: `https://docs.docker.com/config/containers/resource_constraints/`. Zasoby można przydzielać bardzo dokładnie - pamiętaj o tym. A jak będziesz potrzebować to najlepiej spojrzeć do dokumentacji najnowszej wersji.

Zad. 10

Przytnij pamięć do 16MB, uruchom obraz dockerowy z centos 7 i spróbuj zjeść więcej pamięci (np. poleceniem mbuffer albo otwierając w vimie jakiś bardzo duży plik)(Odp: 10)

10. Tworzenie swojego base image

Każdy (noo prawie każdy) na początku przechodzi przez takie coś: „nie wezmę obrazu z docker huba bo nie wiadomo co tam jest, jakie wirusy, i może się pokłócą z moimi wirusami, albo mnie tam co potrąci". Sam se zrobię, będzie lepsze. Jest pewna nieufność co do jakości tych obrazów. Poniekąd słusznie, dlatego w docker hubie jest opcja „oficjalne obrazy". Są one tworzone z reguły przez społeczność systemów operacyjnych. Jeśli nadal nie ufasz to zostaje ostatni argument: nawet robiąc samemu taki obraz, koniec końców musisz użyć paczek deb lub rpm, które pobierasz z internetu, a które są często budowane przez te same zespoły, które robią obrazy dockerowe. Stąd można śmiało założyć, że bezpieczeństwo obrazu dockerowego jest zbliżone lub identyczne z tym w paczkach. Wadą własnych obrazów jest fakt, że trzeba je samemu utrzymywać i aktualizować. Stąd własne obrazy dockerowe są paradoksalnie mniej bezpieczne. Dlatego jeśli nie jesteśmy bankiem, nie ma sensu robić samemu obrazów dockerowych. Jest jeden wyjątek, kiedy warto samemu zrobić obraz - gdy obrazu nie ma w repo. Ze 2-3 lata temu było jeszcze tak, że oficjalne obrazy były głównie na architektury x86, posiadacze komputerów na ARMie nie mieli więc wyjścia. Obecnie, ku mojemu zdumieniu, każda szanująca się dystrybucja ma inne popularne architektury.

Załóżmy więc, że musimy utworzyć własny obraz. Sprowadza się to do trzech kroków:

1. zainstaluj bazowy system w jakimś katalogu
2. zapakuj te pliki do tara
3. zaimportuj archiwum do dockera

10.1. Przygotowanie plików bazowego systemu

Debian i Fedora mają do tego narzędzia, które się nazywają odpowiednio: debootstrap i febootstrap. Przy czym to drugie jest takiej-se jakości. Podaje się im architekturę, wersję, katalog, wciska enter, i samo się pobiera, instaluje.

Do dystrybucji redhatowych lepiej użyć yuma lub dnfa, którymi instaluje się paczki. Ma to jedną sporą wadę - bardzo utrudnione lub wręcz niemożliwe jest zainstalowanie paczek z innej architektury np. ARM na x86. Do centosa jest gotowy skrypcik: `https://github.com/moby/moby/blob/master/contrib/mkimage-yum.sh`, fedorę przyjemnie buduje się samemu dnfem. Tutaj jest ładny rozdzialik o tym: `https://docs.docker.com/develop/develop-images/baseimages/#create-a-full-image-using-tar`.

Od biedy można zainstalować system na wirtualce, uruchomić jakieś distro liveCD (byle nie ten sam system który chcemy pakować) a potem spakować pliki z dysku twardego. Też zadziała

Której metody nie wybierzesz, to rezultatem tej części jest katalog, w którym jest zainstalowany cały system operacyjny, a więc zawierający /bin /usr /var /etc itd.

10.2. Pakowanie do tara

Kluczowa w tej części jest jedna rzecz - należy pliki tak zapakować, żeby bin, usr, etc itd. nie były w podkatalogu. Czyli przeglądając less'em (tak, da się) archiwum tar, lub dowolną inną metodą wypisując sobie listę plików, bin, usr, etc itd. nie mogą być w podkatalogu.

10.3. Zaczytanie do dockera

`cat arch.tar | docker import -`
czyli czytamy archiwum przekierowując dane do docker import (ten myślnik jest ważny na końcu, oznacza odbieranie danych z pipe'a, do którego wpadają z STDOUT od cat'a). Rezultatem będzie hash id obrazu, z którym wiecie już co zrobić. Można też bezpośrednio pakować tarem i wrzucać do dockera, choć nie zalecam tej metody, bo się trudno potem debuguje:
`cd katalog_z_systemem && tar cf - * | docker import -`

Zad. 11

Zbuduj swój własny bazowy obraz Debiana.(Odp: 11)

11. Pierdu pierdu

Zanim przejdziemy do praktyki, kilka słów do czego dockera się używa. No już wiecie że można sobie aplikacje odpalić. Oprócz tego hosty z dockerem można klastrować, i kiedyś obiecującą technologią był Docker Swarm. Wydawało się polecenie klastrowi co ma wdrożyć, i w jakiej ilości, i to robił. Od jakiegoś czasu projekt Kubernetes wyprzedził możliwościami Swarma. Budową dość mocno przypomina Swarma - są master nody i workery, całość spięta w klaster, a stan trzymany nie w Consulu, tylko w Etcd. Różnica jest w możliwościach - Kubernetes ma o wiele większe możliwości w zakresie planowania, zarządzania zasobami. Jest bardziej zwinne i dynamiczne. No i obecnie dużo lepiej rozwijane. Tak jak mieliśmy epokę dockera i klastrów obganianych Puppetem/Saltem itp, tak teraz świat przechodzi na tzw. Cloud Native, którego fundamentem jest bardzo zwinna i elastyczna platforma. Chcąc poczytać o Kubernetes warto spojrzeć do materiałów które przygotowywał Tomek Tarczyński na różne konferencje. Dokumentacja jest zbyt przerażająca na początek :D

A tak poza tym, to zwykły Kowalski, nawet i ten nasz, niezwykły, z IT w naszej firmie, może użyć dockera do różnych ciekawych rzeczy:

- w dockerze można używać dowolnego systemu, byle się binarka odpalała, więc stare rupiecie można opakować w kontener, i już łatwiej nimi zarządzać, nie blokują nam aktualizacji
- na kompie biurkowym jest to o tyle użyteczne, że jakieś zabawki, których używamy można spakować, i potem aktualizacja systemu lub reinstalacja nam niestraszne
- jeśli nie ufamy jakiemuś oprogramowaniu np. Skype na linuxa, to można go zapakować w dockera, zrobić małe czary-mary żeby interfejs graficzny działał, i gotowe. Generalnie można w ten sposób separować każne oprogramowanie, a do kontenera przekazywać tylko te pliki (urządzenia) które są potrzebne
- mamy coś przygotować, jakiś program, pokompilować, przetestować - po co robić syf na swoim biurkowym komputerze? Lepiej w dockerze, przetestować, sprawdzić, napisać sobie już czystą procedurę, a jak coś się nie uda, to docker kill i już porządek. Nie wszystko tak się da zrobić, ale bardzo dużo. Resztę można w wirtualce
- potrzebujemy pythona w wersji 2.7, 3.2 i 3.7 naraz? Żaden problem - trzy kontenerki i po problemie

12. Wiele procesów w jednym kontenerze

Metoda niezalecana, należy się trzymać zasady: jeden kontener, jeden proces. Ale jak ktoś koniecznie potrzebuje, są trzy metody:

1. jeden proces uruchomić standardowo, a pozostałe używając docker exec w tle
2. skrypt uruchamiający wszystkie potrzebne procesy: wszystkie poza ostatnim w tle, a ostatni normalnie (w „foreground”); jako polecenie w docker run podajemy skrypt
3. użycie oprogramowania zarządzającego procesami w kontenerze: swatch, supervisord, lub podobne - do kontenera dostarczamy konfigurację, a jako polecenie w docker run podajemy supervisord owym konfigiem

13. Dodatek - rejestr dockerowy

Wszystko, co tu jest napisane, dotyczy Docker Registry w wersji 2.x z API v2

Sam rejestr dockerowy nie dostarcza wygodnych narzędzi do przeszukiwania, jednak jest to możliwe na endpoincie:

```
GET /v2/_catalog
```

Rezultatem jest lista „repozytoriów", co w naszym przypadku oznaczać będzie listę obrazów. Każdy z obrazów może mieć nadane różne tagi, które listuje się tak:

```
GET /v2/<nazwa obrazu>/tags/list
```

Nie jest to najwygodniejsza na świecie metoda. Czy da się zatem wygodniej? Da się, ale trzeba sobie postawić nakładkę graficzną, albo inną usługę:

- `http://port.us.org/`
- `https://hub.docker.com/r/parabuzzle/docker-registry-ui/`
- `https://goharbor.io/`

Oprócz tego można korzystać z docker huba (darmowe jest konto z publicznie wystawionymi obrazami, prywatne trzeba kupić), lub płatnych wersji rejestru np. w Google lub AWS.

14. Zadania

Zad. 12

Uruchom w dockerze rejestr dockerowy `https://hub.docker.com/_/registry`, dopisz do /etc/hosts nazwę „mojerepo". Wrzuć tam jakiś obraz.(Odp: 12)

Zad. 13

Na bazie oficjalnego, zbuduj obraz nginxa w taki sposób, żeby serwował index.html z twoim imieniem. Bez montowania wolumenu(Odp: 14)

Zad. 14

Na bazie oficjalnego, zbuduj obraz nginxa w taki sposób, żeby serwował index.html z /tmp/ngx1 na hoście. Po uruchomieniu kontenera zmień zawartość index.html i sprawdź czy nginx serwuje nową.(Odp: 14)

Zad. 15

Mamy dwa skrypty:

```
1  #!/usr/bin/env python3
2  from http.server import HTTPServer, BaseHTTPRequestHandler
3  import requests
4
5  class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):
6    def do_GET(self):
7      args = self.path.split('/')
8      result = int(args[1]) + int(args[2])
9      try:
10       out = requests.get('http://zapisz:81/' + str(args[1]) + '/' + str(
       args[2]) + '/' + str(result))
```

```
    except:
      pass
    if out.status_code == 200:
      self.send_response(200)
      self.end_headers()
      self.wfile.write(b'ok')
    else:
      self.send_response(500)
      self.end_headers()
      self.wfile.write(b'dupa')


httpd = HTTPServer(('0.0.0.0', 80), SimpleHTTPRequestHandler)
httpd.serve_forever()
```

Listing 5: plik l5

```
#!/usr/bin/env python3
from http.server import HTTPServer, BaseHTTPRequestHandler

class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):
  def do_GET(self):
    args = self.path.split('/')
    try:
      with open('/var/log/skrypcik/' + str(args[3]), 'a') as out:
        out.write(str(args[1]) + ' + ' + str(args[2]) + ' = ' + str(args
    [3]) + '\n')
      self.send_response(200)
      self.end_headers()
      self.wfile.write(b'ok')
    except:
      self.send_response(500)
      self.end_headers()
      self.wfile.write(b'dupa')

httpd = HTTPServer(('0.0.0.0', 81), SimpleHTTPRequestHandler)
httpd.serve_forever()
```

Listing 6: plik l6

Jeden skrypt słucha na porcie 80, przyjmuje GET /liczba1/liczba2, dodaje je i wykonuje GET /liczba1/liczba2/suma na host „zapis", na port 81 do drugiego skryptu. Ten z kolei wpisuje te trzy liczby do odpowiedniego pliku w /var/log/skrypcik/. Utwórz dwa skrypty z tymi obrazami w taki sposób, że pierwszy będzie wiedział co to jest host „zapis", a drugi będzie miał wystawiony wolumen na swoje logi. Skryptów nie można modyfikować. (Odp: 15)

Zad. 16

Zadanie jak wyżej z jedną zmianą: te skrypty pythonowe mają logować do pliku, a nie na standardowe wyjście. Bez modyfikacji skryptów(Odp: 16)

Zad. 17

Uruchom kontener z centos 7 i zainstaluj gcc. Wyjdź z kontenera i wypisz jakie pliki się zmieniły. (Odp: 17)

Odpowiedź do zad. 1

```
docker rm <id kontenerow rozdzielone spacja>
```

Odpowiedź do zad. 2

```
docker rmi centos:7
```

Odpowiedź do zad. 3

```
docker run -dit centos:7 bash -c "while sleep 1; do date; done"
```

Odpowiedź do zad. 4

Można to zrobić na dwa sposoby:

WORKDIR /root
COPY arch.tar /arch.tar
RUN tar xf /arch.tar

lub tak:

COPY arch.tar /arch.tar
RUN cd /root && tar xf /arch.tar

Obie dobre, ale pierwsza bardziej elegancka

Odpowiedź do zad. 5

Kluczowa jest tu składnia, wpisujemy jak gdyby nigdy nic dwa razy przełącznik -v, i działa:
`docker run -it -v kat1:/kat1 -v kat2:/kat2 centos:7 bash`

Odpowiedź do zad. 6

Skrypt bashowy:

```
while sleep 1; do echo zm_2; done
```

można go wrzucić przez COPY albo wolumenem. Budujemy obraz: można przez dockerfile, można ręcznie i potem przez docker commit przerabiamy padły kontener na obraz. Odpalamy kontener:
`docker run -dit -e zm_2=0 centos:7 sciezka_do_skryptu`
zmienną środowiskową można też ustawić w dockerfilu jeśli tą metodą budowałeś. Zmiana wartości `zm_2`:
`docker exec -it <id kontenera> bash -c "export zm_2=1"` zmienia wartość zmiennej, ale pętla tego nie widzi, bo pracuje w innym procesie basha. Takie-se obejście jest następujące: dodać to do bashrc, a pętla powinna uruchamiać to echo przez bash -c

Odpowiedź do zad. 7

Metoda jest w zasadzie dowolna, możesz napisać skrypt, który po prostu zrobi cat na pliku, ewentualnie możesz dorobić parsowanie, żeby zmienną odróżniał od wartości. Łatwiej niż zmiennymi środowiskowymi, c'nie? Noo nie są one takie cudowne, jakby się mogło wydawać. Mają swoje wady.

Odpowiedź do zad. 8

Bo kontener chodzi tak długo, jak długo chodzi proces o PID=1, u nas to był top

Odpowiedź do zad. 9

Tak jak z wolumenem - dwa razy się wstawia:
`docker run -dit -p 172.17.0.1:32080:80 -p 192.168.1.2:32080:80 nginx:latest`

Odpowiedź do zad. 10

Odpowiedź do zad. 11

Użyj debootstrap, a reszta według instrukcji w rozdziale.

Odpowiedź do zad. 12

```
docker run -d -p 5000:5000 --restart always --name registry registry:2
vim /etc/hosts
docker tag <id> mojerepo/test:1
docker push mojerepo/obraz:cos
```

Odpowiedź do zad. 14

Tworzysz dockerfile z poleceniem COPY index.html w odpowiednie miejsce

Odpowiedź do zad. 14

Montujesz wolumen w miejsce w kontenerze zgodnie z dokumentacją do obrazu nginxa.

Odpowiedź do zad. 15

todo

Odpowiedź do zad. 16

Opakować skrypt w jakiegoś basha np:
`python3 skrypt 2>&1 >> /var/log/s1.log`
wrzucić używając COPY ten skrypt gdzieś do kontenera, i go uruchamiać przy starcie kontenera.

Odpowiedź do zad. 17

Użyj docker diff: `https://docs.docker.com/engine/reference/commandline/diff/`