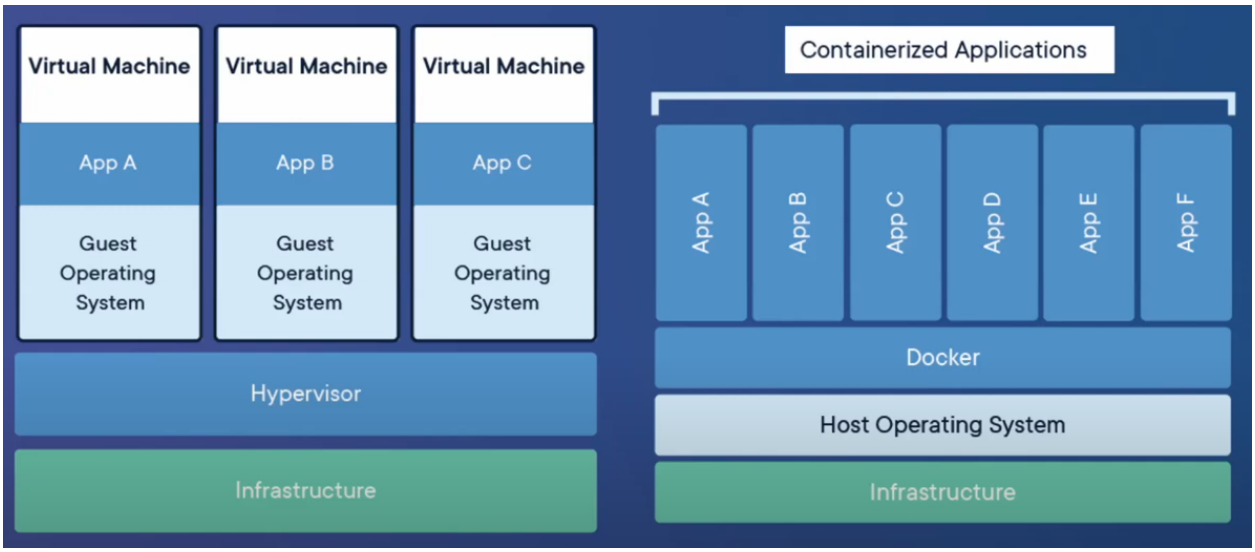
Docker - nauka

1. Docker to jedna z implementacji kontynerów
2. Kontener to przybliżenie maszyny wirtualnej, która zawiera paczkę z oprogramowaniem.

Różnice pomiędzy VM a Kontynerami



Maszyny wirtalne posiadają warstwę Hypervisor (system operacyjny służacy do zarządzania instancjami maszyn wirtualnych), gdzie pojedyńcza maszyna wirtualna składa się z systemu operacyjnego oraz aplikacji.

W przypadku kontynerów nie mamy warstwy Hypervisor, tylko mamy współdzielony system operacyjny który posiada system Docker, który zarządza wieloma instancjami niezależnych kontynerów.

Z czego składa się docker

1. Dockerfile
2. Standard obrazów
3. Narzędzia do budowania

4. Środowisko uruchomieniowe
5. Docker Hub

Docker jest bazowany na linuxie

Docker działa na linuxie. W przypatku tworzenia kontyneru w innym systemie (windows/mac OS) towrzony jest VM z linuxem na którym jest uruchomiony docker. Tak więc, docker na te systemy głónie jest do developowania, natomiast wdrażany jest docelowo na produckję w systemie linux.

W naszym przypadku będziemy się uczyć o dockerze w wersji Linux containers (uwaga, należy to przełączyć w docker desktop)

Docker Toolbox - $docker-maching ip - komenda do sprawdzenia ip kontynerów docker

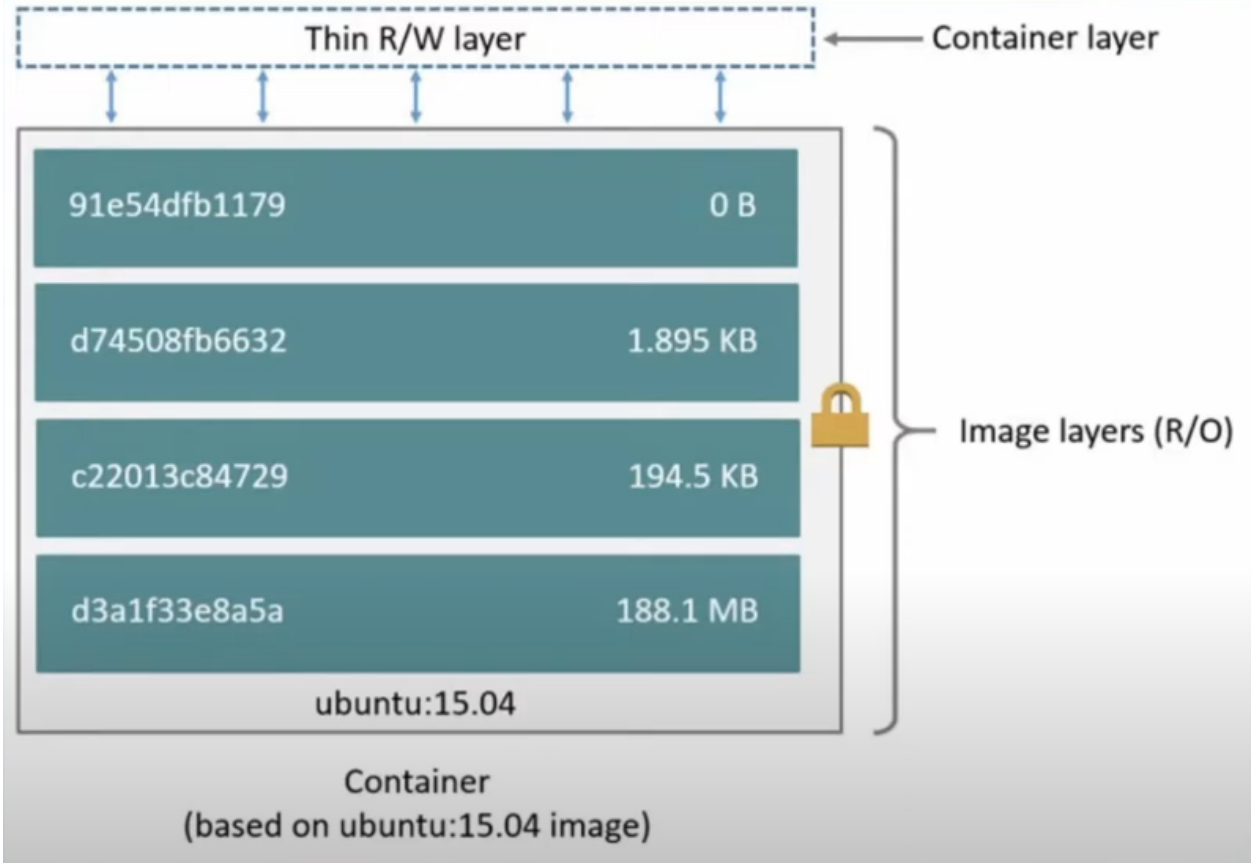
Kontyner - Jest to instancja uruchomina na systemie operacyjnym (odiziolowany proces)

Obraz - Szablon służący do stworzenia kontynera

Podstawowe komendy:

```
//Tworzenie kontenera docker i wylistowanie plików wewnątrz kont
docker run ubuntu ls -l
//Tworzenie kontynera wraz z opcjami interactive oraz tty (reako
docker run --interactive --tty ubuntu bash
//Sprawdzenie wszystkich kontynerów (również wyłączonych)
docker container ls -a
//Sprawdzenie wszystkich kontynerów (tylko aktywnych)
docker container ls / docker ps
//Uruchamianie kontynera
docker start #containerId
//Uruchamianie polecenia wewnątrz działającego kontynera
docker exec #containerId (polecenia)
```

Sturktura działana obrazu i kontynera docker



Każdy obraz docker składa się z kilku warstw. Każda warstwa przechowuje tylko zmiany, które zostały wprowadzone do obazu - co pozwala na lekkość działania.

```
//Tworzenie obrazu na podstawie kontynera
docker commit #containerID nazwa_img
//Listowanie dostępnych obrazów
docker image ls
//Wyświetelenie warstw obrazu
docker history nazwa_obrazu
```

AmigosCode - Docker and Kubernetes - Full Course for Beginners

Co to jest docker

Jest to narzędzie do uruchamiania aplikacji w odizolowanym środowisku.

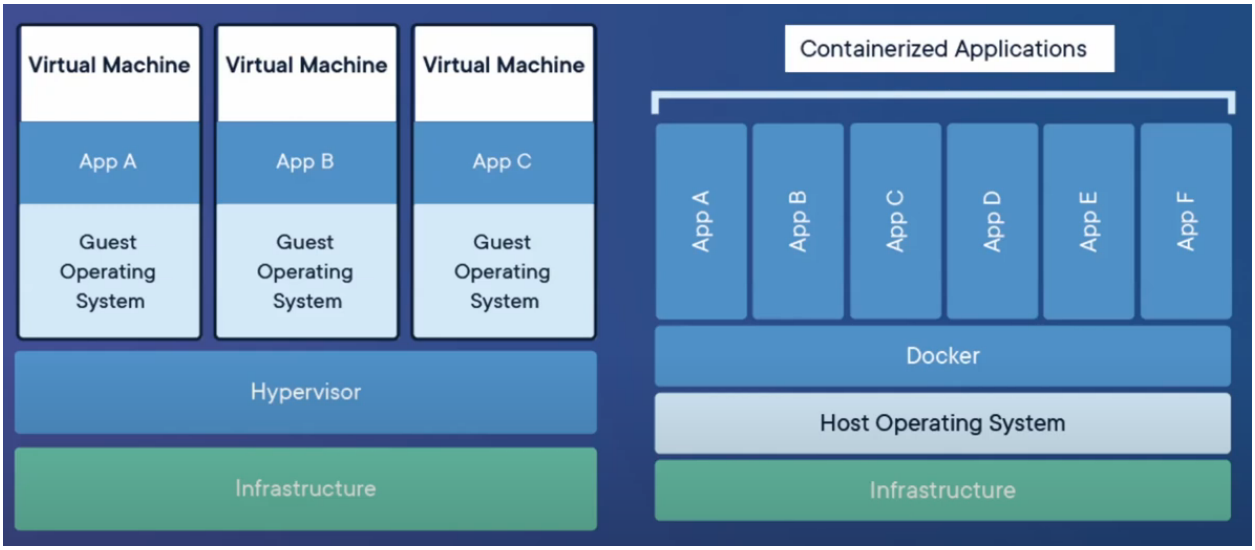
Jest to koncepcja podobna do VM'ek, ale zoptymalizowane pod względem systemowym.

Jest to standardowe narzędzie dla software deploymentu

Różnica pomiędzy VM a Kontynerem

Kontyner to abstrakcyjny element warstwy aplikacji, który składa się z kodu wraz zależnościami. Wiele instancji kontynera może działać niezależnie od siebie, korzystając z tego samego, wspólnego systemowego jądra (kernel) - każdy kontyner to osobny proces.

VM natomiast są abstrakcją na warstwie fizycznej, gdzie 1 serwer poprzez warstwę hypervisor zarządza wieloma instancjami vm, gdzie każda instancja ma swój osoby system operacyjny.



Zalety i wady:

1. Szybsze uruchomienie

2. Lepsze wykorzystanie pamieci
3. Współdzielony system

Instalancja docker desktop

https://www.docker.com/products/docker-desktop/

Docker image

Jest to wzór do tworzenia kontynerów

Jest to snapshot przechowujący wersje

Zawiera pełen zestaw danych potrzebych do działania aplikacji

Kontyner

Jest to instancja obrazu docker

DockerHub

Zawiera obrazy (również oficjalne) które zawierają obrazy narzędzi

Aby pobrać narzędzie do wykonujemy polecenie:

```
docker pull nazwaObrazy
//ex
docker pull nginx
//Sprawdzenie dostępnych obrazów
docker images
```

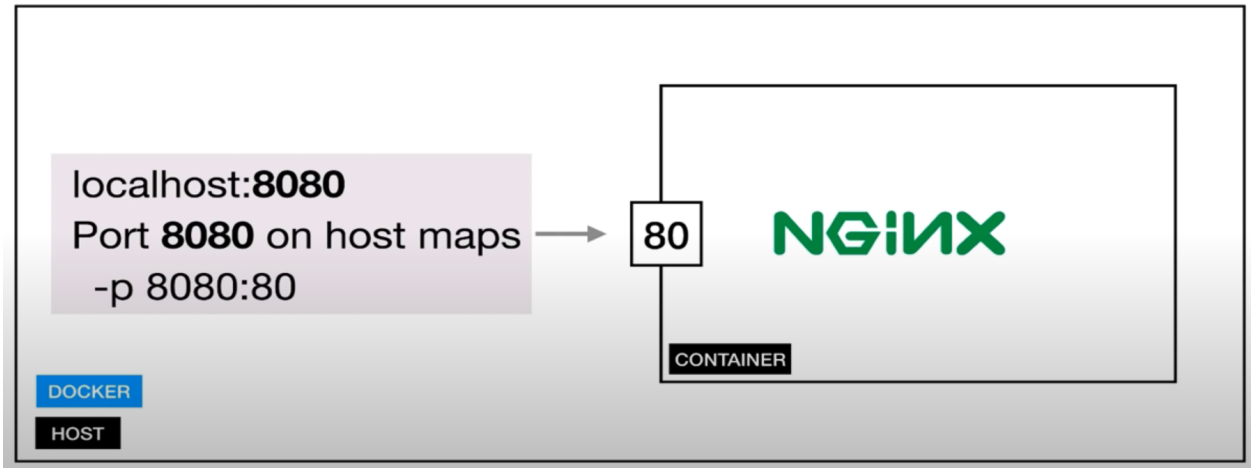
Uruchomenie kontynera z obrazu

```
//Uruchomenie kontynera z obrazu (w trybie detached)
doceker run -d #dockerImageId
//Sprawdzenie uruchomionych kontynerów
docker container ls
```

Porty docker - exposing port

Jeżeli kontyner zawiera aplikację, która udostępnia swoją usługę na kontretnym porcie np. nginx udostępnia na porcie 80, to chcemy, aby aplikacja była dostępna z poziomu portu kontynera 8080.

Należy wykonać mapowanie




```
//zatrzymanie kontyneru docker
docker stop #containerId
//Uruchomienie kontynera z mapowanem portów (host port:containe
docker run -d -p 8080:80 #dockerImageId
```

Teraz aplikacja nginx jest dostępna pod portem 8080

Można również mapować więcej portów na jeden poprzez komendę:

```
docker run -d -p 8080:80 3000:80 #dockerImageId
```

Zarządzanie kontynerem

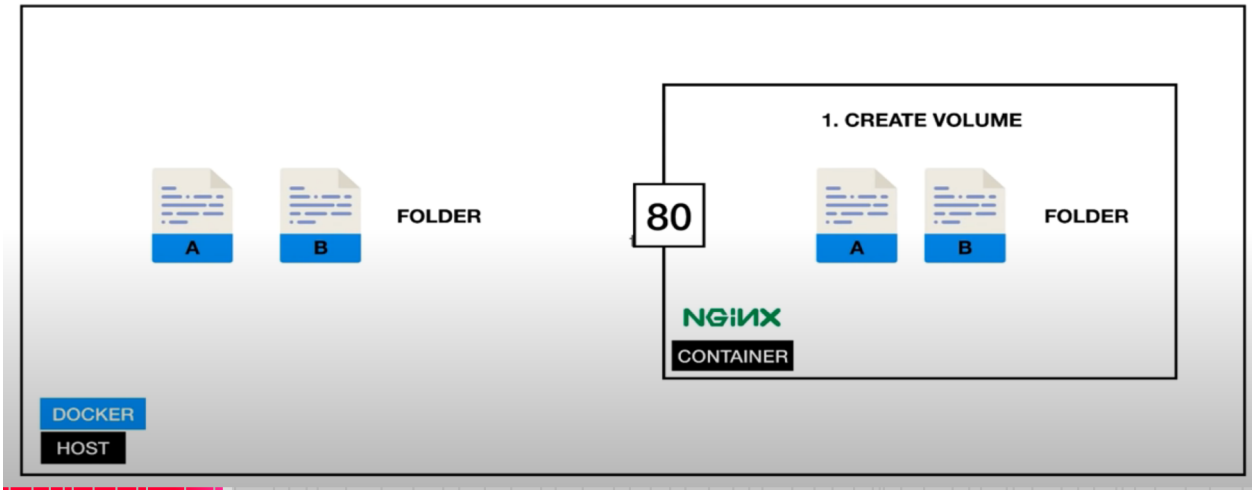
```
//listowanie kontynerów uruchomionych
docker ps
//zatrzymnie kontyneru
docker stop #dockerContainerId lub nazwa
// uruchamianie kontynera
docker start #dockerContainerID lub nazwa
// opcje listowania
docker ps --help
// wszystkie kotynery (również te nie uruchoione)
docker ps -a
// usuwanie kontynerów (nie może działać, lub trzeba dodać flage
docker rm #dockerContainerID lub nazwa
```

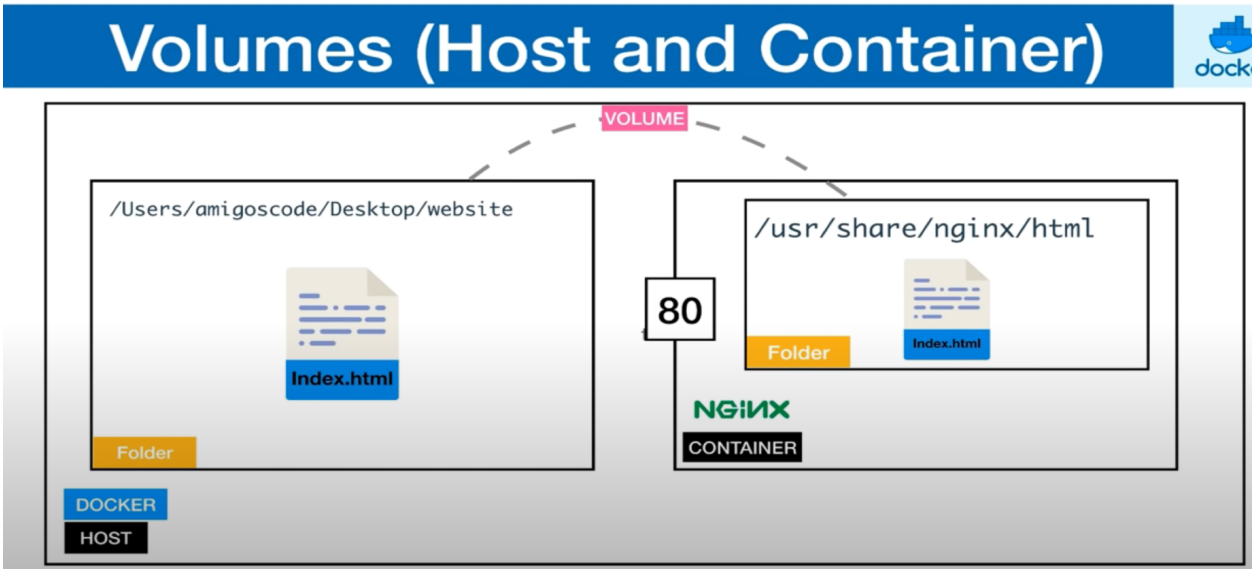
Zmiana nazw kontynerów

```
//Dodanie flagi --name, żeby nazwać kontyner
docker run --name container_name -d -p 8-80:80
//Formatowanie do human-easily readable
docker ps --format="ID\t{{.ID}}\nNAME\t{{.Names}}\nIMAGE\t{{.Ima
```

Docker volumes

Volumes umożliwiają współdzielenie plików i folderów pomiędzy hostem a kontynerem, i pomiędzy kontynerami. Możemy udostępniać dane pliki/foldery znajdujące się na hoście do kontynerów, jeżeli utworzą one volume z odpowiednim mapowaniem





Do utworzenia takiego połączenia należy podać:

```
//Należy zatrzymać kontyner
docer stop my_docker_container_name
//Przejście do folderu na hoście (polecenia cd.)
Uruchomienie z opcją -v i podanie ścierzki
docker run --name container_name -v $(pwd):scierzka_w_kontynerze
```

Takie montowanie działa dwustronnie, czyli pliki które zostały uwtorzone w współdzielonym folderze docker, zostaną również udostępnione do hosta - przykład

```
//Wejście do kontynera
docer exec -it nazwa_kontynera bash
```

Volumes between containers

Aby sprawdzić możliwości docker należy wpisać

```
docker run --help
```

Aby zamontować kontyner na podstawie innego kontynera, należy podać opcję, oraz wystawić go na nowym pocie (w naszym przypadku 8081)

```
docker run --volumes from name_of_existing_container
np.
docker run --name container_copy -d -p 8081:80 image_name
```

Dockerfile

Aktualnie wykonywaliśmy kontynery dokcer bazując na obrazach.

Do tworzenia własnych obrazów służy plik dockerfile

Dokumentacja potrzebna do stworzenia dockerfile:

https://docs.docker.com/reference/dockerfile/

Dockerfile to plik określający listę kroków, które należy wykonać, w celu wykonania cokerimage

Żeby wykonać własny kontener nie jest zalecane utworzenie volume z hosta do kontynery docker, a lepiej stworzyć dockerfile, żeby stworzyć image, na którego bazie będzie można tworzyć nieograniczną ilość kontynerów.

Image powinien zawierać wszystkie informacje dostępne do stworzenia aplikacji.

```
//Sprawdzenie dostępnych obrazów
docker image ls
```

1. Stworzenie pliku dockerfile w folderze root projektu
2. Zawsze tworzymy aplikację, bazując na podanym schemcie, więc pierwszą komendą będzie FROM, który ustala bazowy image
3. ADD . - dodaje wszystkie pliki projektu z podanej ścierzki

```
FROM base_image
ADD . /user/project_root
```

Building image basic on dockerfile

Muszą być podane opcje:

1. —tag / -t - podajemy tag w formacie - nazwa:tag
2. ścieżka do pliku docker image

```
docker build --tag .
```

Uruchamianie kontynera bazując na obrazie:

```
docker run --name nazwa_kontynera -p 8080:80 -d image:tag
```

Przykładowy dockerFile dla Node

```
FROM node:lates <- podanie bazowego image ( z docker hub)
WORKDIR /app <- utworzenie/nadpisanie folderu aplikacji
ADD . . <- Dodanie wartości z akutalnego WD do nowego w kontyne
RUN npm install <- inicjalizacja
CMD node index.js <- uruchomienie aplikacji z terminala
```

Sources:

https://github.com/dsyer/gs-spring-boot-docker

https://spring.io/guides/gs/spring-boot-docker

https://www.docker.com/blog/kickstart-your-spring-boot-application-development/

Dcokerfile .dockerignore

Plik ten zaznacza pliki, które nie mają byc uwzględnione w image docker

(np. node_modules), albo inne pliki które powstają podczas budowania.

Caching and layers

Każde polecenie w dockerfile jest to warstwa, która jest cachowan'a

W przykładzie:

```
FROM node:lates <- podanie bazowego image ( z docker hub)
WORKDIR /app <- utworzenie/nadpisanie folderu aplikacji
ADD . . <- Dodanie wartości z akutalnego WD do nowego w kontyne
RUN npm install <- inicjalizacja
CMD node index.js <- uruchomienie aplikacji z terminala
```

Jeżeli mamy tylko zmianę kodu, to niektórego kodu nie trzeba powtarzać, i w tym momencie do optymalizacji można zastosować mechanizm cachowania.

Poprzez zmianę kolejności (w pierwszej kolejności instalacja paczek i zależności i następnie dodanie ich po dockerfile

```
FROM node:latest
WORKDIR /app
ADD package*.json /.
RUN npm install
```

```
ADD . .
CMD node index.js
```

Alpine dystro - redukcja wielkości obrazów

Alpine jest to jestna z dystrybucji linux'a.
Z racji, że jest ona stworzona pod minimalne wymagania dyskowe, może nam posłużyć do kompresji obrazów docker.

Pulling alpine images

Zamiast używać obrazu z tagiem lastes, można używać wersji alpine np. wersja alpine node, nginx

```
docker pull noce:lts-alpine
```

Użycie alpine

```
FROM node:alpine
WORKDIR /app
ADD package*.json /.
RUN npm install
ADD . .
CMD node index.js
```

Tags and versions

Pozwala na kontrolę wersji. W przypadku wejścia nowej wersji oprogramowania, możemy prędzej się przygotować do migracji na nowszą wersję, poprzez sprawdzenie czy aktualny kod będzie działać poprawnie na nowym środowisku,

Na dockerhub jest dostępnych kilka wersji, więc podczas tworzenia dockerfile mozna zdefinować używana wersję narzędzia. np.

```
FROM nginx:1.17.2-alpine
```

Tagging own images

Dodając cały czas wersję, jeżeli chcemy zamiast latest zmienić wersję na konkretną, możemy podać komendę:

```
docker tag amigoscode-website:lastest amigoscode-website:1
```

Powoduje to, że zostaja dodany oberaz, ale z nowym taegiem →1

W przypadku update'u funkcjonalności, najlepiej dodawać zawsze jako latest , i następnie update versjonowania np. V1.1.12 ect.

Tworzenie kontynerów na podstawie różnych obrazów

```
docker run --name image_latest -p8080:80 -d website:latest
docker run --name image_v2 -p8081:80 -d website:2
docker run --name image_v1 -p8082:80 -d website:1
```

Docker registry (taki Git dla Docker image)

jest to serwer umożliwiający przechowywanie oraz udsotępnianie różnych wersji docker

jest on uywany w pipeline'ach CI/CD , udostępniać kod do rozruchu na usługach typu AWS Elastic Beanstalk

Rozróżniamy prywatne oraz publiczne obrazy.

Mamy dostępnych następujących, większych providerów:

1. Docker Hub
2. quay.io

3. Amazon ECR

Creating DockerHub repository

1. Register in dockerhub
2. Należy stworzyć repozytorium (prywatne, lub publiczne)
3. Następnie trzeba dopasować, lub utworzyć kopię obrazów, aby zawierały nazwę użytkownika, nazwę repozytorium oraz interesujący nas tagname → komenda

```
docker tag actual_image_name:version username/repository_name:ta
```

Należy się upewnić, że jest się zalogowany na odpowiednie konto docker hub w aplikacji dokerhub desktop

I należy wypchać do repozytorium zdalnego

```
docker push username/repository_name:tagname
```

Docker Compose

Ponieważ każdy kontyner powinien mieć 1 odpowiedzialność, czyli powinien odpowiadać za 1 aplikację, to do prawidłowego działania aplikacji, jest potrzebne kilka kontynerów.

Docker compose umożlia stworzenie 1 pliku yaml, który definuje konfigurację wielu kontynerów w celu uruchomienia aplikacji,.

Tworzenie dockercompose.yaml

```
version: '3.1'
services:
  backend:
  build: . <- oznacza, że obraz ma być zbudowany na dockerfile w
  ports:
  - 8190:8190
```

```
    environment: <- ustawienia spring boot
    - SPRING_PROFILES_ACTIVE=local
    - SPRING_DATASOURCE_URL=jdbc:postgresql://PostgreSQL:5432/car_
    - SPRING_DATASOURCE_USERNAME=postgres
    - SPRING_DATASOURCE_PASSWORD=postgres
    networks:
    - spring-boot-postgres
    depends_on:
    PostgreSQL:
    condition: service_healthy
    PostgreSQL:
    image: postgres:15.0
    ports:
    - 5432:5432
    environment:
    - POSTGRES_DB=car_dealership
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
    volumes:
    - db-data:/var/lib/postgresql/data
    networks:
    - spring-boot-postgres
    healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 10s
    timeout: 5s
    retries: 5
  volumes:
    db-data:
  networks:
    spring-boot-postgres:
```

1. DockerCompose w wersji 3.1
2. Definujemy kontynery (services)
    a. backend - kontyner spring boot

b. Określa, że obraz Dockera dla tej usługi zostanie zbudowany na podstawie Dockerfile znajdującego się w bieżącym katalogu (.).
c. Mapowanie portów 1:1
d. enviroment - definuje zmienne środowiskowe Spring Boot
e. network - definuje wspólną sieć dla spring boot i postgres
f. depends_on - zależność, w której definujemy, że Spring Boot wstanie, gdzy dopiero postgres db wstanie

3. Postgres - tak samo z różnicami;
   a. Udostępnia usługi na porcie 5432
   b. inne zmienne środowiskowe
   c. Ma volumes

Powtórka docker

1. Co to jest docker

   Docker jest to jeden z systemów służących do kontyneryzacji, czyli uruchamiania odizolowanych aplikacji wraz z wszystkimi potrzebnymi dependencjami, jako odizolowany proces w systemie. Umożlwia w ramach 1 systemu operacyjnego na tworzenie wielu odizolowanych aplikacji-kontynerów.