DOCKER
ESENCJA

Jan Michalski
https://PROGRAMATOR.blog

Spis treści

Wstęp

Cześć, tu Jan!

Bardzo się cieszę, że interesujesz się Dockerem. To technologia, która pomimo że jest obecna "pod strzechami" już od dobrych kilku lat, nadal stanowi centrum wszystkiego, co dotyczy tematów DevOpsowych: tworzenia, wdrożenia i utrzymania współczesnych aplikacji.

Jeszcze bardziej cieszy mnie, że trafiłeś na tę pozycję. Nie jest ona długa (raptem dwadzieściaparę stron), ale zawiera tylko mięso codziennych działań z Dockerem – sama praktyka.

Nie polecam wykorzystywania jej jako jedyne źródło do nauki. Do tego celu będzie po prostu zbyt gęsta :) Jak sama nazwa wskazuje, jest to raczej ściągawka i przypomnienie najpotrzebniejszych operacji.

Jeżeli właśnie uczysz się Dockera, zapraszam Cię do obejrzenia **serii filmów** na moim kanale na Youtube. Przechodzę w nich krok po kroku przez wszystko, co jest zawarte w tym zeszycie, wraz z obszernym tłumaczeniem. **Znajdziesz je tutaj:** https://www.youtube.com/playlist?list=PLkcy-k498-V5AmftzfqinpMF2LFqSHK5n

Mam nadzieję, że znajdziesz w tym zeszycie dużo przydatnych informacji. **Zaczynajmy!**

PS. Gorąco zachęcam do przepisywania kodu ręcznie, zamiast kopiowania. Przepisywanie kodu ułatwia zapamiętywanie i znacząco przyspiesza naukę w porównaniu do kopiowania gotowego kodu, nawet jeżeli "na sucho" rozumiesz co taki kod robi.

Konwencje

Tak jest oznaczana komenda wiersza poleceń (do wpisania bez początkowego znaku dolara `$`):

```
$ komenda --flagi=wartosci parametry [opcjonalne parametry]
```

Tak jest oznaczana komenda do wykonania wewnątrz kontenera (do wpisania bez początkowego `container #`):

```
container# komenda --flagi=wartosci parametry
```

Konteneryzacja

Jakie problemy rozwiązuje?

- Instalowanie bibliotek i zależności (“Dependency Hell”)
- Zarządzanie konfiguracją
- Uruchamianie różnych wersji tej samej aplikacji
- Tworzenie paczek z programami (packaging)
- Brak spójnego środowiska pomiędzy maszynami deweloperów i serwerami produkcyjnymi

Kontener

- “Wirtualny komputer” - podobne w użyciu do maszyny wirtualnej
- Wirtualizacja na poziomie pojedynczego procesu
  - vs wirtualizacja hardware i całego systemu operacyjnego w maszynie wirtualnej
- Używa wspólnego jądra systemu operacyjnego
- Kontener = proces + system plików + metadane
- Sposób pakowania i dostarczania oprogramowania
- Izolacja procesów, bezpieczeństwo

Kontener vs Obraz

- Kontener - faktycznie uruchomiony proces; instancja obrazu
- Obraz - archiwum plików i metadanych; szablon kontenera
- Z jednego obrazu możemy utworzyć wiele kontenerów
- Na podstawie gotowego kontenera możemy zapisać obraz

Czym jest Docker?

- Implementacja systemu konteneryzacji
- Platforma uruchomieniowa
- Narzędzie do budowania kontenerów
- Standard obrazów
- Dockerfile - automatyzacja budowania obrazów
- Docker Hub - centralny publiczne repozytorium obrazów
- Architektura klient-serwer: Docker Daemon (serwer) i Docker CLI (komendy w wierszu poleceń - klient)

Uruchamianie Kontenera

Docker Run

Poniższe polecenie tworzy kontener na podstawie obrazu i od razu uruchamia ten kontener z podanym poleceniem. Jeżeli nie podamy żadnego polecenia, kontener zostanie uruchomiony z poleceniem domyślnym dla danego obrazu.

```
$ docker run <nazwa_obrazu> [<polecenie do wykonania w kontenerze>]
```

Na przykład uruchomienie polecenia `ls -l` (wylistowanie zawartości katalogu) w kontenerze stworzonym na podstawie obrazu o nazwie `ubuntu` (obraz bazowej dystrybucji Ubuntu):

```
$ docker run ubuntu ls -l
```

Kontener możemy wyłączyć skrótem klawiszowym `CTRL-C`.

Szczegółowa dokumentacja: https://docs.docker.com/engine/reference/run/

Automatyczne usuwanie kontenera po zakończeniu działania

Flaga `--rm` automatycznie usunie kontener po zakończeniu jego działania:

```
$ docker run --rm ubuntu ls -l
```

Umieszczanie działającego kontenera w tle

Jeżeli uruchomiliśmy kontener, możemy odłączyć się od niego bez wyłączania go z pomocą kombinacji skrótów klawiszowych:

```
CTRL-P CTRL-Q
```

Uruchamianie w tle

Możemy kontener uruchomić od razu w tle, flagą `-d` (`--detach`):

Przykład:

```
$ docker run --detach my_application
```

Do przywrócenia kontenera, działającego w tle, na pierwszy plan służy polecenie:

```
$ docker attach <kontener>
```

Czy kontenery rzeczywiście współdzielą jądro systemu operacyjnego?

Uruchamiamy dwa kontenery na podstawie różnych obrazów. Poleceniem `uname -a` sprawdzamy wersję jądra systemu operacyjnego. Oba kontenery pokażą tę samą wersję jądra (niektóre informacje zwracane przez `uname -a` mogą się różnić).

```
$ docker run ubuntu uname -a
Linux ... 4.14.79 ... x86_64 GNU/Linux

$ docker run debian uname -a
Linux ... 4.14.79 ... x86_64 GNU/Linux
```

Wiersz poleceń w kontenerze

Żeby uruchomić wiersz poleceń, albo inny interaktywny proces, musimy podać dwie dodatkowe flagi do polecenia `docker run`:

- `--interactive` będzie utrzymywała dostęp do standardowego wejścia kontenera.
- `--tty` da procesowi w kontenerze dostęp do konsoli (zaalokuje pseudo-terminal).

Na przykład uruchomienie wiersza poleceń (shella) `bash`. Następnie zapisujemy tekst do pliku i odczytujemy zawartość tego pliku. Zamykamy shell skrótem klawiszowym Ctrl-D (sygnał końca wejścia EOF) albo poleceniem `exit`:

```
$ docker run --interactive --tty ubuntu bash
container# echo "text" > test.txt
container# cat test.txt
container# ^D           \\ container# exit
```

Flagi można skrócić do `-i -t` i połączyć do `-it`:

```
$ docker run -it ubuntu bash
```

Docker run = nowy kontener

Wykonajmy powyższe polecenia, a następnie spróbujemy wykonać poniższe polecenie (wyświetl zawartość pliku test.txt):

```
$ docker run ubuntu cat test.txt
```

Okaże się, że plik test.txt nie został znaleziony. Dzieje się tak, ponieważ każde wywołanie polecenia `docker run` tworzy nowy kontener. Kontener z tego polecenia nie jest tym samym kontenerem, w którym stworzyliśmy plik test.txt.

Publikowanie (przekierowanie) portów

Przekierowanie portu z kontenera na port automatycznie przydzielony przez system operacyjny:

```
$ docker run --publish <port w kontenerze>
$ docker run --publish 5000
```

Przekierowanie portu z kontenera na określony port na hoście:

```
$ docker run --publish <port hosta>:<port w kontenerze>
$ docker run --publish 8080:5000
```

Flagę `--publish` można skrócić do `-p`.

Przekierowanie portów w Docker Toolbox

W systemach Linux oraz używając Docker Desktop, przekierowane porty z kontenera są dostępne jako `://localhost:<port>` albo `://127.0.0.1:<port>`.

Jeżeli korzystasz z Docker Toolbox, docker działa w maszynie wirtualnej uruchomioną na twoim komputerze. Dlatego przekierowane porty nie będą dostępne pod `localhost`. Będą dostępne pod adresem zwróconym przez poniższe polecenie:

```
$ docker-machine ip
192.168.99.100
```

W powyższym przykładzie port kontenera przekierowany na 8080 będzie dostępny pod `192.168.99.100:8080`.

Działania na kontenerach

Wyświetlanie stworzonych kontenerów

```
$ docker container ls [-a]
$ docker ps [-a]
```

Oba powyższe polecenia są równoważne. Służą do wyświetlania działających kontenerów.

Opcjonalna flaga `-a` (`--all`) powoduje wyświetlenie wszystkich utworzonych kontenerów, zarówno działających jak i wyłączonych.

Przykładowy wynik polecenia:

```
$ docker ps -a
CONTAINER ID   IMAGE    COMMAND   CREATED
685dc393ae23   ubuntu   "bash"    15 minutes ago

STATUS                       PORTS   NAMES
Exited (130) 15 minutes ago          wonderful_hawking
```

Oto jakie kolumny widzimy:

- `CONTAINER ID` unikalny identyfikator kontenera, przypisany automatycznie
- `IMAGE` obraz użyty do stworzenia kontenera
- `COMMAND` komenda uruchomiona w kontenerze
- `CREATED` kiedy kontener został stworzony
- `STATUS` stan kontenera - możliwe stany to m.in.:
  - Created - stworzony, ale nie uruchomiony.
  - Up - działający.
  - Exited - zakończył działanie. W nawiasach podany jest kod wyjścia zwrócony przez polecenie.
- `PORTS` przekierowane portów (zob. Przekierowanie portów)
- `NAMES` nazwy nadane kontenerowi
  - automatycznie generowana jest nazwa składająca się z przymiotnika i nazwiska znanego naukowca

Identyfikowanie kontenera

We wszystkich poleceniach dotyczących pojedynczego kontenera możemy zamiennie używać nazwy kontenera albo jego identyfikatora. Dodatkowo identyfikator możmy skracać tak długo jak pozostanie on unikalny.

Przykład:

- Jeżeli mamy tylko jeden kontener (jak poniżej), możemy się do niego odwołać jako `wonderful_hawking`, `685dc393ae23`, `685d`, a nawet tylko `6`.

```
$ docker ps -a
CONTAINER ID  ...   NAMES
685dc393ae23  ...   wonderful_hawking
```

- Jeżeli mamy kilka kontenerów (jak poniżej), do pierwszego z nich możemy się odwołać `wonderful_hawking`, `685dc393ae23`, `685d`, ale już nie `685`.

```
$ docker ps -a
CONTAINER ID  ...   NAMES
685dc393ae23  ...   wonderful_hawking
68511fe859cd  ...   priceless_dijkstra
```

W poleceniach poniżej, w miejsce `<kontener>` używamy dowolnego z powyższych identyfikatorów.

Uruchamianie zatrzymanego kontenera

```
$ docker start <kontener>
```

Uruchomienie (albo ponowne uruchomienie) danego kontenera.

Wykonanie polecenia w działającym kontenerze

```
$ docker exec <kontener> cat test.txt
```

Wykonanie polecenia `cat test.txt` w działającym kontenerze. Kontener musi działać. Wyłączony kontener musi najpierw zostać uruchomiony.

Kopiowanie plików z i do kontenera

Kopiowanie plików z hosta (lokalnego kontenera) do kontenera:

```
$ docker copy <plik_lokalny> <kontener>:<plik>
```

Kopiowanie plików z kontenera na hosta:

```
$ docker copy <kontener>:<plik> <plik_lokalny>
```

Przykład: kopiujemy plik test.txt z kontenera a1b2c3e4 do katalogu `foo/`:

```
$ docker copy a1b2c3e4:test.txt foo/test.txt
```

Szczegóły kontenera

Poleceniem `docker inspect <kontener>` możemy wyświetlić wszystko co Docker wie o danym kontenerze (w formacie JSON):

```
$ docker inspect a1b2c3d4
[
  {
    "Id": "224ccb310d99b54c7c271cfa402cb...
    "Created": "2020-05-17T13:42:15.196545091Z",
    "Path": "bash",
    "Args": [],
    "State": {
        "Status": "exited",
        "Running": false,
         ...
```

Widzimy m.in. stan kontenera, lokalizację plików, sieci.

Obrazy

Obrazy Dockerowe składają się z warstw. Warstwy są niezmienne. Pojedyncza warstwa składa się z odniesienia do poprzedniej warstwy i zmian w systemie plików w stosunku do niej.

Obrazy mogą współdzielić te same warstwy. Przechowujemy tylko jedną kopię warstwy, niezależnie od tego ile obrazów wykorzystuje tę warstwę. Tym samym oszczędzamy miejsce na dysku i użycie sieci, bo nie musimy przesyłać warstw, które mamy już zapisane na dysku.

Tworzenie obrazu

Mając kontener, w którym wprowadziliśmy zmiany w systemie plików np. zainstalowaliśmy paczki z oprogramowaniem, zapisujemy obraz na podstawie aktualnego stanu tego kontenera. Wykorzystujemy polecenie:

```
$ docker commit <kontener> [nazwa_obrazu]
```

Parametr `nazwa_obrazu` jest opcjonalny. Jeżeli go nie podamy, do obrazu będziemy mogli odwołać się tylko za pomocą automatycznie nadanego identyfikatora.

Lista obrazów

Dwa równoważne polecenia:

```
$ docker image ls
$ docker images
REPOSITORY   TAG      IMAGE ID       CREATED      SIZE
my_image     latest   fd50fe0f7c13   3 days ago   946kB
```

Kolumny:

- `REPOSITORY` nazwa obrazu
- `TAG` wersja obrazu (może być nadana dowolnie)
- `IMAGE ID` automatycznie nadany identyfikator

Podejrzenie warstw obrazu

```
$ docker history <nazwa_obrazu>
```

Na przykład:

```
$ docker history ubuntu
IMAGE          CREATED    CREATED BY                                      SIZE
4e5021d210f6   1 day ago  /bin/sh -c #(nop)  CMD ["/bin/...   0B
<missing>      1 day ago  /bin/sh -c mkdir -p /run/syste...   7B
<missing>      1 day ago  /bin/sh -c set -xe   && echo '...   745B
<missing>      1 day ago  /bin/sh -c [ -z "$(apt-get ind...   987kB
<missing>      1 day ago  /bin/sh -c #(nop) ADD file:594...   63.2MB
```

Przykład stworzenia kontenera i obrazu

```
$ docker run -it ubuntu bash
container# apt update
container# apt install vim
container# exit

$ docker ps -a
CONTAINER ID ...
224ccb310d99 ...

$ docker commit 224ccb vim_image
sha256:bd914da009f8ef29780173f6b2e98461f4c183625d29ab936d0ccedc03df3f7e

$ docker images ls
REPOSITORY   TAG      IMAGE ID       CREATED          SIZE
vim_image    latest   bd914da009f8   29 seconds ago   153MB

$ docker history vim_image
```

Docker Hub

Docker Hub - centralne repozytorium obrazów dockerowych.

- Docker Hub: http://hub.docker.com
- Wyszukiwarka obrazów: https://hub.docker.com/search?q=wordpress
- Szczegóły obrazu: https://hub.docker.com/_/wordpress

Ściąganie obrazu z Docker Hub

```
$ docker pull wordpress
```

Logowanie do repozytorium z lokalnego komputera

```
$ docker login
Username: foobar
Password:
```

Wgrywanie własnych obrazów do repozytorium

```
$ docker push foobar/my_image
```

Nazywanie obrazów

Nazwa obrazu musi odpowiadać repozytorium, do którego obraz jest wgrywany. (Dlatego `docker image ls` wyświetla nazwę obrazu w kolumnie `REPOSITORY`). Domyślnie tym repozytorium będzie Docker Hub. Wgrywając na Docker Hub, musimy nazwać obraz w formacie `<nazwa_uzytkownika>/<nazwa_obrazu>` .

```
$ docker tag <obraz> <nazwa docelowa>
$ docker tag my_image foobar/my_image
```

Jeżeli chcemy wgrać obraz na inne repozytorium musimy nazwać obraz w formacie: `<url_repozytorium>/<sciezka>/<nazwa_obrazu>`

Przykład dla Google Cloud Platform registry: `gcr.io/project12345/my_image`

Dockerfile

Dockerfile umożliwia automatyzację budowania obrazów.

Dla każdego polecenie w Dockerfile, docker tworzy nowy tymczasowy kontener, wykonuje w nim polecenie i zapisuje obraz tego kontenera jako nową warstwę obrazu. Następne polecenie

- `FROM <obraz bazowy>` (np. `FROM ubuntu`) - zaczynamy budowanie naszego obrazu na bazie tego obrazu
- `COPY <sciezka zrodlowa> <sciezka docelowa>` - przekopiuj plik z build contextu do obrazu
- `ADD <sciezka zrodlowa> <sciezka docelowa>` - to samo co `COPY`, a dodatkowo pobieranie plików przez URL i rozpakowywanie archiwów tar (zob. Polecenie ADD)
- `WORKDIR <katalog>` - katalog, w którym będą wykonywane wszystkie następne polecenia; polecenie utworzy katalog jeżeli nie istniał
- `RUN <polecenie>` - wykonaj polecenie w trakcie budowania obrazu
- `CMD <polecenie>` - ustaw domyślne polecenie uruchomienia kontenera (polecenie nie będzie wykonane w trakcie budowania obrazu)
- `ENTRYPOINT <polecenie>` - ustaw domyślne polecenie uruchomienia kontenera (zob. Polecenia CMD i ENTRYPOINT)

Przykład:

```
FROM ubuntu
COPY test.txt .
RUN apt-get update
RUN apt-get install --yes vim
```

Dodatkowy przykład: zobacz Dockerfile dla Pythona

Polecenia w Dockerfile szczegółowo

Polecenie ADD

Polecenie `ADD`: * tak jak `COPY` - skopiowanie pliki z Build Contextu do obrazu * pobranie pliku z URL do obrazu - jako parametr podajemy URL do pliku, np. `ADD http://ftp.pl.debian.org/debian/pool/main/n/nano/nano_3.2-3_amd64.deb` * rozpakowanie archiwum tar do obrazu, np. `ADD archive.tar.gz`

Do pierwszego zastosowania - kopiowania plików do obrazu - polecam polecenie COPY. Będzie łatwiej zrozumieć jaki jest nasz cel.

Polecenia CMD i ENTRYPOINT

Formy argumentu: shell i exec Oba polecenia przyjmują argumenty w dwóch formach nazwanych: shell i exec.

Forma shell to zwykła komenda powłoki linuxowej . Żeby z niej skorzystać w kontenerze musimy mieć zainstalowaną powłokę (np. *sh* albo *bash*) - domyślnie dostępne w większości obrazów.

Przykład: `CMD ls -al /app/stuff | grep test.txt`

Forma exec ma postać tablicy argumentów (w formacie JSON, parametry w cudzysłowach, rozdzielane przecinkiem). Możemy jej używać zawsze, nawet gdy w kontenerze nie mamy żadnego shella. Nie możemy w niej korzystać z funkcji powłoki, takich jak przekierowanie strumieni (`cat > test.txt`) albo pipe (`cat | grep`). **Przykład:** `CMD ["ls", "-a", "-l", "/app/stuff"]`

Polecenie CMD Definiuje domyślne polecenie wykonane w kontenerze.

Nadpisywane poleceniem podanym przy uruchamianiu kontenera (`docker run ubuntu <polecenie>`).

Polecenie ENTRYPOINT Definiuje program uruchomiony w kontenerze.

Nie jest nadpisywane przez polecenie podane przy uruchamianiu kontenera, ale można je nadpisać podając flagę `--entrypoint` (np. `docker run --entrypoint=bash ubuntu`).

Jeżeli podamy w Dockerfile oba polecenia z argumentem w formie exec, to polecenie CMD będzie ustawiało domyślne argumenty programu podanego w ENTRYPOINT.

Przykład:

```
Dockerfile:
ENTRYPOINT ["/myapp", "param1", "--flag1"]
CMD ["--flag2", "/some/file"]

$ docker run <obraz>
Wykona: /myapp param1 --flag1 --flag2 /some/file

$ docker run <obraz> param3 /other/file
Wykona: /myapp param1 --flag1 param3 /other/file
```

Budowanie obrazu

```
$ docker build <katalog>
```

Przykład: `$ docker build .` - kropka oznacza aktualny katalog.

Build Context

- Katalog, który podajemy w `docker build` jest istotny, bo to w nim są szukane pliki, które kopiujemy do kontenera.
- Ten katalog nazywa się Build Context.

- W momencie uruchomienia `docker build` cały katalog jest wysyłany do Dockera Deamon jako build context.
- Ponieważ cały katalog jest wysyłany (faktycznie wysyłany przez HTTP przez lokalny socket) do Docker daemona, warto nie trzymać w nim dużych plików. (see TODO)
- Flagą `-f` (`--file`) możemy podać ścieżkę do Dockerfile, który ma zostać użyty. Domyślnie jest to plik `Dockerfile` w podanym Build Contextie.

Konfigurowanie kontenerów - zmienne środowiskowe

Zazwyczaj aplikacje w kontenerach są konfigurowane za pomocą zmiennych środowiskowych.

Zmienne środowiskowe podajemy za pomocą flagi `-e` (`--environment`).

Przykład:

```
$ docker run -e USER=abc -e PASSWORD=pass -e OTHER_PARAM=true my_image
```

Volumes - wolumeny

Wolumeny

- Zarządzane przez dockera
- Tworzenie: `docker volume create <nazwa>`
- Wylistowanie: `docker volume ls`
- Usunięcie `docker volume rm <wolumen>`
- Podpięcie do kontenera: `docker run --volume <wolumen>:<katalog_kontenera> <obraz>`
- Podpięcie do kontenera tylko do odczytu:

`docker run --volume <wolumen>:<katalog_kontenera>:ro <obraz>`

Przykład:

```
$ docker volume create my-volume
$ docker run --volume my-volume:/app/data my_image
```

Dodatkowy przykład: zobacz Baza danych PostgreSQL

Wolumeny anonimowe

Tworzone automatycznie przy uruchomieniu kontenera.

```
$ docker run --volume <katalog_kontenera> <obraz>
```

Przykład:

```
$ docker run --volume /app/data my_image
```

Bind mounts

Połączenie katalogów lub plików z hosta do kontenera. Ścieżka do katalogu na hoście musi być ścieżką absolutną (typu: `/home/user/foo/bar/`; na Windowsie: `//c/Users/user/foo/bar`)

```
$ docker run --volume <katalog_hosta>:<katalog_kontenera> <obraz>
```

Na przykład możemy podpiąć katalog z kodem źródłowym z hosta do kontenera z naszym środowiskiem deweloperskim.

```
$ docker run --volume $(pwd)/src:/app my_dev_env
```

Sieci

Zapewniają komunikację pomiędzy kontenerami i między kontenerem a internetem.

Sieć typu bridge - umożliwia podłączenie do internetu przez hosta oraz umożliwia połączenie do innych kontenerów, które są w tej samej sieci.

Sieć domyślna - bridge

Docker domyślnie tworzy domyślną sieć typu bridge o nazwie `bridge`. Każdy kontener jest do niej podłączony domyślnie, jeżeli nie podamy innej sieci.

Zapewnia połączenie pomiędzy znajdującymi się w niej kontenerami i internetem, ale nie umożliwia rozwiązywania adresu IP kontenerów po ich nazwie.

Nie można w trakcie działania kontenera odłączyć go od tej sieci, jeżeli została ona ustawiona domyślnie.

Przykład:

```
$ docker run --name containerA ubuntu
$ docker network inspect bridge
```

Wyświetlenie listy sieci

```
$ docker network ls
```

Własna sieć typu bridge

Możemy ręcznie stworzyć dowolną liczbę własnych sieci typu bridge.

W odróżnieniu od domyślnej sieci, te tworzone ręcznei umożliwiają rozwiązywanie adresu IP kontenerów po ich nazwie.

Możemy też podałczać i odłączać kontenery do/z sieci w trakcie ich działania.

Tworzenie sieci

```
$ docker network create [--driver bridge] <nazwa>
```

Podanie opcji `--driver bridge` jest opcjonalne. Bridge jest domyślnym ustawieniem.

Podłączenie kontenera do sieci przy tworzeniu

```
$ docker run --network <siec> <obraz>
$ docker run --network my-network ubuntu
```

Jeżeli nie podamy opcji `--network`, kontener zostanie podłączony do domyślnej sieci `bridge`.

Podłaczenie i odłączanie kontenera do sieci w trakcie działania

Podłączanie:

```
$ docker network connect <siec> <kontener>
$ docker network connect my-network my-container
```

Odłączanie:

```
$ docker network disconnect <siec> <kontener>
$ docker network disconnect my-network my-container
```

Łączność pomiędzy kontenerami

Przykład:

```
$ docker network create my-network
$ docker run -dit --name containerA --network my-network busybox
$ docker run -dit --name containerB --network my-network busybox
$ docker attach containerA
containerA# ping containerB
```

Docker Compose

Plik `docker-compose.yml`:

```
version: “3”
services:
  baza:
    image: postgres
    networks:
      - db-net
    volumes:
      - db-data:/var/lib/postgresql/data
    environment:
      POSTGRES_USER: db-user
      POSTGRES_DB: my-db
      POSTGRES_PASSWORD: secretpassword

  web-ui:
    image: adminer
    networks:
      - baza-net
    ports:
      - 8080:8080

networks:
  db-net:
volumes:
  db-data:
```

Uruchomienie serwisów

```
$ docker-compose up
```

Możemy podejrzeć efekty działania `docker-compose up` poleceniami:

```
$ docker ps
$ docker network ls
$ docker volume ls
```

Przykłady

Podstawowy Dockerfile dla aplikacji w Pythonie

```
FROM python:3.8
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY app.py .
CMD python app.py
```

Baza danych PostgreSQL w kontenerze

```
$ docker create db_data
$ docker run --name database --volume db_data:/var/lib/postgresql/data \
    -e POSTGRES_USER=my_user -e POSTGRES_PASSWORD=pass -e POSTGRES_DB=my_base \
    --publish 5432:5432 --detach postgres
$ docker exec -it database psql my_base my_user
$ psql --hostname=localhost --port=5432 my_base my_user
```

Powodzenia!

Mam wielką nadzieję, że to opracowanie przyda Ci się w pracy i nie tylko ;)

Życzę Ci powodzenia w nauce Dockera i dalszym rozwoju!

Moją działalność znajdziesz tutaj:

- Strona i Blog: https://programator.blog
- Youtube: https://www.youtube.com/channel/UCTR3ihcAiLude0fZjWSAHNg
- Instagram: https://www.instagram.com/jan.programator/

Dzięki i do zobaczenia,

Jan