Docker[1]
laboratorium
2023/24
T. Goluch

1. Wstęp

Docker jest oprogramowaniem open-source pozwalającym na automatyzację procesu wdrożenia aplikacji. Jest to koncepcja podobna do maszyn wirtualnych jednak dużo lżejsza. Kontener Docker'a to opakowanie pozwalające na uruchamianie pojedynczych procesów. W przeciwieństwie do VM uruchamiane są w jądrze systemu operacyjnego hosta co objawia się minimalizacją dodatkowych narzutów na czas wykonania ale też zmniejszeniem bezpieczeństwa uruchamianego kodu. Dzięki wykorzystaniu minimalnej przestrzeni dyskowej ich czas uruchamiania jest ekstremalnie szybki co powoduje, że mogą służyć do wykonywania krótkich zadań będąc wielokrotnie tworzone i usuwane (idea mikro-serwisów). Komunikują się ze sobą i z otoczeniem podobnie jak komputery w sieci lokalnej. Każdy z nich posiada unikalny adres IP przypisany do wirtualnego interfejsu.

Kontenery najlepiej sprawdzają się w przypadku aplikacji bezstanowych, pozwalając na łatwe skalowanie horyzontalne. W przeciwnym razie stan aplikacji powinien być przechowywany na zewnątrz kontenera np. w bazie danych albo w systemie plików – co jest niezalecane ponieważ wprowadza dodatkowe zależności. Bezstanowość kontenerów bierze się z faktu, że podczas każdego uruchamiania jest on odtwarzany z obrazu.

Początkowo istniały jedynie kontenery Linuxowe, Microsoft zaczął wspierać tę technologię dopiero w systemach Windows 10 (x64) oraz Server 2016. Wraz z udostepnieniem kontenerów Windows pojawiły się – uruchamiane w nich – tzw. Nano Serwery pozwalające na hostowane aplikacji napisanych w .NET, Java, JavaScript/Node.js czy Python/Django. Rosnąca popularność Dockera sprawiła, że są one wspierane przez czołowych dostawców chmur obliczeniowych.

W niniejszej instrukcji będą wykorzystywane poniższe terminy:

obraz (*ang. image*) – to z niego tworzony jest kontener, możemy spojrzeć na niego jak na klasę tworzonych instancji kontenerów.

warstwa obrazu (*ang. image layer*) – warstwy powstają w wyniku wykonywania czynności takich jak: instalacja pakietu, kopiowanie danych, ustawienie zmiennej środowiskowej. Są dostępne jedynie do odczytu i to z nich składają się obrazy.

kontener (*ang. container*) – jest to uruchomiona instancja obrazu Dokera. Zawiera aplikację wraz z wszystkimi jej zależnościami pozwalając na uruchomienie jej jako izolowany proces w przestrzeni użytkownika w systemie operacyjnym hosta. Może znajdować się w jednym z następujących stanów: *created*, *restarting*, *running*, *removing*, *paused*, *exited* i *dead*.

demon/serwer dockera (*ang. docker daemon/server*) – usługa działająca w tle na hoście. Zarządza budową, działaniem i dystrybucją kontenerów Dockera.

klient dockera (*ang. docker client*) – narzędzie typu wiersz poleceń (*ang. command-line tool*). Pozwala użytkownikowi na interakcję z demonem/serwerem Dockera

Docker Hub – publiczny rejestr obrazów Dockera, zawiera dużą liczbę gotowych do użycia obrazów.

[1] https://docs.docker.com/

2. Instalacja

Linux[2]

W celu instalacji Docker'a na Linuxie musimy sprawdzić podłączenie z Internetem. W terminalu[3] uruchamiamy polecenie:

```
ping 8.8.8.8
```

W przypadku niepowodzenia sprawdzamy czy nasz interfejs jest prawidłowo zdefiniowany w systemie:

```
vi|vim[4]|nano /etc/network/interfaces
```

Większość komend służących do instalacji oprogramowania oraz konfiguracji systemu operacyjnego wymaga uprawnień root'a. Możemy wykorzystać narzędzie suid root poprzedzając każdą taką komendę słowem kluczowym `sudo` albo raz zalogować się na konto root'a:

```
sudo -i
```

Do wylogowania służy komenda `exit`. W przypadku niektórych wersji Linuxa, np. CentOS będzie wymagane dodanie użytkownika do grupy sudo:

```
sudo usermod -G sudo <user>
```

Zawartość pliku w przypadku współpracy z usługą dhcp powinna być następująca:

```
...
auto eth0
iface eth0 inet dhcp
```

albo w przypadku konfiguracji statycznej:

```
...
auto eth0
iface eth0 inet static
address X.X.X.X
gateway X.X.X.X
netmask X.X.X.X
network X.X.X.X
broadcast X.X.X.X
dns-nameservers X.X.X.X [X.X.X.X]
```

Po zapisaniu zmian restartujemy (można wykonać kolejno metody stop i start) interfejsy sieciowe:

```
/etc/init.d/networking restart|stop|start
```

W celu sprawdzenia konfiguracji interfejsu możemy posłużyć się pakietem Iproute2:

```
ip a
```

---

[2] https://docs.docker.com/install/

[3] Okno konsoli otwieramy skrótem Ctrl+Alt+t (Ubuntu), w większości Linuxów można przełączyć się do konsoli za pomocą: Ctrl+Alt+F{X}, gdzie X⊂{1,2,3,4,5,6}, powrót do domyślnego XWindows za pomocą Ctrl+Alt+F7.

[4] **Insert** – rozpoczęcie edycji, **Esc** – zakończenie, **:w [nazwa_pliku]** – zapis [nowa nazwa plku], **:wq [nazwa_pliku]** – zapis [nowa nazwa plku] i wyjście, **:q!** – wyjście bez zapisywania.

albo starszą komendą:

```
ifconfig -a
```

W kolejnym kroku należy odświeżyć listę dostępnych pakietów (apk/apt/yum/dnf):

```
apk update                  (Apline)
apt-get update              (Ubuntu)
yum check-update            (CentOS)
dnf upgrade –refresh        (Fedora)
```

W przypadku Alpine (apk) musimy ręcznie dodać link do repozytorium:

```
vi|vim|nano /etc/apk/repositories
```

dodajemy zawartość:

```
...
http://dl-cdn.alpinelinux.org/alpine/edge/main
http://dl-cdn.alpinelinux.org/alpine/edge/community
```

Jesteśmy już gotowi do instalacji Dockera[5] (apk/apt/yum/dnf):

```
apk add docker                  (Apline)
apt-get install -y docker.io    (Ubuntu)
yum install -y docker           (CentOS)
dnf install –y docker           (Fedora)
```

Sprawdzenie statusu/uruchomienie/zatrzymanie Dockera:

```
service docker status|start|stop
```

lub

```
systemctl status|start|stop docker.service
```

Demon Docker'a podczas swojej pracy wymaga dostępu do portów, gniazd, plików co powiązane jest z posaidaniem uprawnień administratora. Nadal możemy korzystać z komendy `sudo`, jednak bardziej komfortową opcją będzie dodanie użytkownika do grupy `docker`. Najpierw należy sprawdzić czy użytkownik nie został już dodany do takiej grupy:

```
groups student
```

w przeciwnym przypadku należy dodać do niej użytkownika:

```
usermod -aG docker <user>
```

ostatecznie wymagany będzie restart systemu operacyjnego.

Docker Machine

Jeśli z pewnych względów nie chcemy instalować pod Linuxem natywnego Dockera istnieje możliwość zainstalowania docker-machine:

```
base=https://github.com/docker/machine/releases/download/v0.14.0 &&
curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/tmp/docker-machine &&
sudo install /tmp/docker-machine /usr/local/bin/docker-machine
```

[5] Opcja –y pozwala automatycznie odpowiadać Yes na wszystkie monity

Do sprawdzenia wersji docker-machine ewentualnie weryfikacji poprawności instalacji służy komenda:

```
docker-machine version
```

Do poprawnej pracy docker-machine wymaga dostawcy wirtualizacji np. VirtualBox'a, którego możemy zainstalować (Ubuntu):

```
apt-get install virtualbox
```

Utworzenie nowej maszyny:

```
docker-machine create <machine-name>
```

Wyświetlenie listy aktywnych maszyn:

```
docker-machine ls
```

Windows

W przypadku systemu operacyjnego Windows 10 (x64) oraz Server 2016 należy zainstalować aktualną wersje Dockera, Numer wersji można sprawdzić poleceniem:

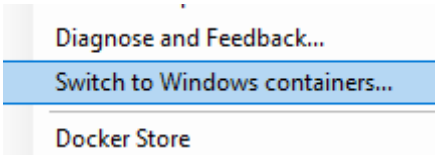
```
docker version
```

Wynik dla aktywnego kontenera w wersji Linux:

```
PS C:\Users\student> docker version
Client:
 Version:       18.03.0-ce
 API version:   1.37
 Go version:    go1.9.4
 Git commit:    0520e24
 Built: Wed Mar 21 23:06:28 2018
 OS/Arch:       windows/amd64
 Experimental:  false
 Orchestrator:  swarm

Server:
 Engine:
  Version:      18.03.0-ce
  API version:  1.37 (minimum version 1.12)
  Go version:   go1.9.4
  Git commit:   0520e24
  Built:        Wed Mar 21 23:14:32 2018
  OS/Arch:      linux/amd64
  Experimental: false
```

Po instalacji „Docker for Windows" domyślnie uruchomiony jest kontener Windowsa przełączanie się pomiędzy rodzajami obsługiwanych kontenerów jest dostępne z menu kontekstowego Docker'a (Switch to Windows/Linux containers…) dostępnego z zasobnika systemowego (za pierwszym razem może być wymagany restart systemu).

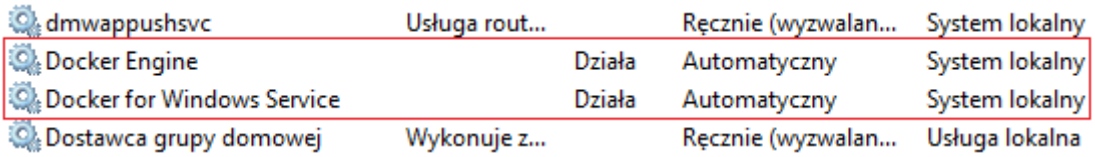


W przypadku niepowodzenia należy sprawdzić czy jest włączona wirtualizacja, np. w menedżerze zadań.

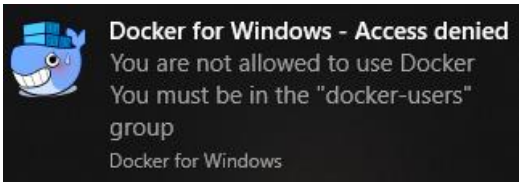
Tym razem `docker version` powinno zwrócić nast. wynik:

```
Server:
 Engine:
  Version:      18.03.0-ce
  API version:  1.37 (minimum version 1.24)
  Go version:   go1.9.4
  Git commit:   0520e24
  Built:        Wed Mar 21 23:21:06 2018
  OS/Arch:      windows/amd64
  Experimental: false
```

W przypadku problemów należy sprawdzić czy uruchomione są usługi Dockera:



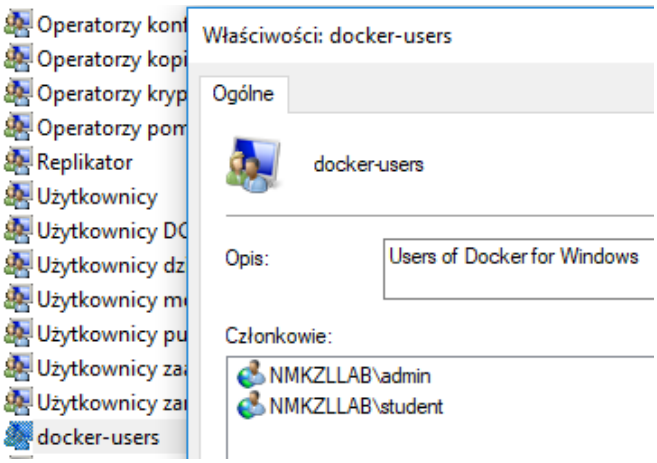
Jeśli po uruchomieniu Docker'a dostaniemy następującą informację, o braku dostępu:



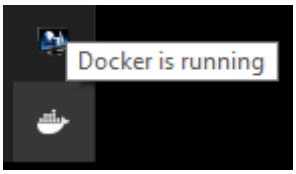
Musimy dodać naszego użytkownika do grupy *docker-users*, możemy wykorzystać konsolę zarządzania Microsoft (MMC) uruchamianą poleceniem:

```
lusrmgr.msc
```

i dodać użytkownika student (wymagane uprawnienia administratora):



Po ponownym zalogowaniu uruchomienie Docker for Windows w zasobniku systemowym powinna pojawić się ikonka informująca o działających usługach:



W obydwu trybach (kontenery Linux oraz Windows) możemy upewnić się, że instalacja przebiegła poprawnie posługując się komendą:

```
docker run hello-world
```

Powinniśmy zobaczyć informację o poprawnej instalacji oraz krokach jakie zostały wykonane przez Dockera. Mianowicie – klient Dockera skontaktował się z demonem Dockera, który ściągnął obraz „hello-world" z Docker Hub'a. Następnie utworzył nowy kontener w którym uruchomione zostały pliki wykonywalne, a ich wyjście zostało przesłane do klienta, który wyświetlił je w terminalu.

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Do wyświetlenia aktywnych maszyn służy komenda:

```
docker-machine ls
```

zaś do wyłączania/wyłączania maszyny Docker:

```
docker-machine stop/start [NAME]
```

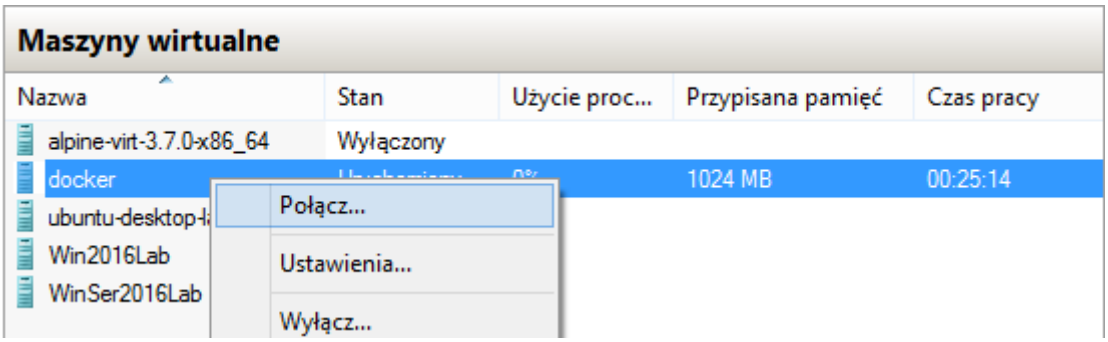
Uruchamianie Dockera pod Windows w wirtualnym systemie jest możliwe ale nie wskazane i narażone na błędy: Running Docker for Windows in nested virtualization scenarios.

W przypadku starszych wersji systemu pozostaje nam narzędzie „Docker Toolbox for Windows" instalujące Dockera w minimalnej maszynie wirtualnej i wspierającej jedynie kontenery Linuxa. Domyślnym nadzorcą wirtualizacji jest Oracle VM VirtualBox, który musi być zainstalowany. W przypadku komputera z uruchomionym Hyper-V należy zainstalować najnowszą wersję docker-machine dostępną tutaj: https://github.com/docker/machine/releases/[6]. Instalacja polega na uruchomieniu w PowerShell 'u komendy z katalogu ze ściągniętym programem:

```
./docker-machine-Windows-<ver>.exe create --driver hyperv --hyperv-virtual-switch
<NazwaVirtualnegoSwitcha> <NazwaMaszyny>.
```

[6] aktualna wersja to 0.14.0, może mieć problem z obsługą modułu Hyper-V PowerShell, co objawia się wystąpieniem błędu: Error with pre-create check: "Hyper-V PowerShell Module is not available", rozwiązaniem jest instalacja wersji 0.13.0.

Maszyna powinna być widoczna w Menedżerze funkcji Hyper-V gdzie możemy się z nią połączyć:



W przypadku wylogowania się, domyślne dane uwierzytelniające to: login: ***docker*** hasdło: ***tcuser***.
Więcej informacji można znaleźć tutaj: https://docs.docker.com/machine/drivers/hyper-v/.

3. Praca z kontenerami[7]

Linux

Polecenie:

```
docker help
```

drukuje podstawowe komendy Docker CLI, ich lista wraz z dostępnymi opcjami oraz rozszerzonym opisem dla jest dostępna tutaj: https://docs.docker.com/engine/reference/commandline/docker/. Na początek wystarczy znajomość kilku aby móc skorzystać z podstawowych funkcjonalności Dockera.

Komenda `docker pull <nazwa_obrazu[:tag]>` pobiera obraz z repozytorium. Domyślnie pobierany jest obraz zawierający tag `latest`, który powinien być zawsze dostępny i oznaczać najnowszą wersję – niestety zależy to tylko od dobrej woli autora. Możemy również podać konkretną wersję, podając jej tag albo pobrać wszystkie dostępne obrazy (opcja -all). Przykładowo:

```
docker pull alpine –a
```

pobierze wszystkie wersje obrazów Dockera z Linuxem Alpine. Teraz możemy wyświetlić wszystkie pobrane obrazy:

```
docker images
```

Proszę sprawdzić czy rzeczywiście istnieje wersja najnowsza i czy odpowiada ona wersji o najwyższym numerze (identyczne numery „IMAGE ID"). Do usunięcia obrazu służy następująca komenda:

```
docker rmi <nazwa_obrazu[:tag]|IMAGE ID>[-f]
```

Podając nazwę obrazu bez tag'a usuniemy wersję najnowszą (`latest`). Alternatywnie jako ID wystarczy podać unikalny, początkowy ciąg znaków. W przypadku problemów skuteczna może okazać się konieczna opcja siłowa: -f. Do utworzenia kontenera z pobranego obrazu służy:

```
docker create --name <nazwa_kontenera> <nazwa_obrazu[:tag]|IMAGE ID>
```

Pozostało uruchomić nowo powstały kontener metodą:

```
docker start <nazwa_kontenera>
```

[7] https://docs.docker.com/engine/reference/run/

Komenda `run` to tak naprawdę dwie kolejne komendy `create` i `start`. Ponadto, jeśli obraz kontenera nie jest dostępny lokalnie to zostanie on automatycznie pobrany z repozytorium (metoda `pull`). Zatem wszystkie trzy kroki (pobranie obrazu, utworzenie i uruchomienie kontenera). możemy wykonać przy pomocy jednej komendy:

```
docker run <nazwa_obrazu[:tag]>
```

Polecenie:

```
docker run alpine
```

pozwala na pobranie najnowszej wersji obrazu, utworzenie kontenera o wygenerowanej nazwie oraz na jego uruchomienie. Jest adekwatna ciągowi poleceń:

```
docker pull alpine
docker create –name alpine_container alpine
docker start alpine_container
```

W obydwu przypadkach nasz kontener od razu zakończy pracę, jednak można sprawdzić, że znajduje się na liście wszystkich (aktywnych i nieaktywnych, opcja -a) kontenerów:

```
docker ps -a
```

Kontener zostaje zatrzymany, kiedy kończy pracę jego główny proces (PID=1), w przypadku kontenerów uruchamianych w tle (o czym później) do ich zatrzymania służy komenda:

```
docker stop <NAME|CONTAINER ID>
```

Usunięcie kontenera:

```
docker rm <NAME|CONTAINER ID>
```

Jeśli chcemy aby kontener był usuwany zaraz po jego zamknięciu nalezy podać podczas jego uruchamiania parameter `--rm`. Przykład uruchomienia kontenera z instancją RStudio:

```
docker run -it --rm rocker/r-base
```

Stan Docker'a (m.in. liczbę istniejących obrazów oraz kontenerów, w tym ile kontenerów jest w stanie: uruchomiony/wstrzymany/wyłączony) możemy sprawdzić poleceniem:

```
docker info
```

W poprzednim podpunkcie uruchomiliśmy nasz pierwszy kontener (hello-world). Liczba udostępnionych w ten sposób obrazów w publicznych repozytoriach jest imponująca. W celu znalezienie interesującego nas obrazu kontenera możemy przejrzeć np. repozytorium Docker'a: https://hub.docker.com/, albo skorzystać z wyszukiwania:

```
docker search <podciąg_nazwy_szukanego_obrazu>
```

Na przykład `docker search lynx` albo `docker search vim` zwróci listę kontenerów zawierających podaną frazę.

Uruchomienie kontenera uruchamiającego przeglądarkę lynx:

```
docker run -it jess/lynx
```

Wiele przykładowych obrazów Docker'a można znaleźć na: https://github.com/dockersamples i https://docs.docker.com/samples/.

Domyślnie kontenery uruchamiane są w trybie pierwszoplanowym *(ang. foreground)*, do konsoli podłączane są standardowe strumienie wejścia i wyjścia. Aby móc korzystać z procesów interaktywnych (typu shell) należy użyć

opcji `–it` (`-i` – uruchamia tryb interaktywny, `-t` – przydziela Docker'owi konsolę pseudo-tty, która jest wymagana przez bash'a).

W kolejnym przykładzie uruchomimy obraz zawierający edytor vim. Podanie opcji `--name` nadaje nazwę kontenerowi i pozwala na łatwiejsze odwoływanie się do niego. Inaczej będzie to, jak w poprzednim przykładzie losowy ciąg znaków:

```
docker run -i -t --name my-vim haron/vim
```

Po uruchomieniu kontenera jesteśmy w edytorze. Proszę zapisać plik z dowolną zawartością w domyślnej lokalizacji i zapamiętać jego nazwę. Uwaga, po opuszczeniu edytora automatycznie opuszczamy kontener, dzieje się tak ponieważ proces vim'a był głównym procesem wewnątrz kontenera (PID=1). Skoro znajdujemy się w konsoli hosta odszukajmy zapisany przed chwilą plik na dysku komendą (w tym momencie może być przydatne wyczyszczenie konsoli komendą `clear`):

```
find / -type f -name <nazwa_pliku>[8]
```

Plik powinien się znajdować w nast. lokalizacji[9]: `/var/lib/docker/<SPM>/<CONTAINER_ID>/home/dev/`, gdzie `SPM` to sterownik pamięci masowej z jakiego korzysta Docker np. (aufs, devicemapper, btrfs, vfs, overlay), a `CONTAINER_ID` to długi ID kontenera. Możemy sprawdzić, że nasz nowy kontener zawierający utworzony plik istnieje na liście ostatnio uruchamianych kontenerów (opcje: –a – wszystkie, -q – tylko numery ID):

```
docker ps [-a] [-q]
```

Jako `CONTAINER_ID` podane zostało krótkie ID zawierające 12 początkowych znaków. Nasz kontener możemy uruchomić metodą `start`:

```
docker start –a –i <unikalny_początkowy_ciąg_ID>
```

i sprawdzić, że utworzony plik nadal w nim istnieje poleceniem vim'a (nie można opuszczać edytora ponieważ spowoduje to automatyczne opuszczenie kontenera).

```
:edit <nazwa_pliku>
```

Nic nie stoi na przeszkodzie aby edytowany plik znajdował się w systemie plików hosta. W tym celu korzystamy z parametru `–v /host/path:/container/path`, który pozwala na współdzielenie katalogu (co ciekawe katalog może nie istnieć zostanie utworzony pod warunkiem, że litera dysku będzie poprawna i serwer Docker'a będzie posiadał prawa zapisu):

```
docker run -i -t --name my-vim2 -v ~/:/home/dev/src haron/vim[10]
```

Podczas zapisywania proszę podać ścieżkę do współdzielonego katalogu:

```
:w /home/dev/src/<nazwa_pliku>
```

Proszę odszukać zapisany plik w folderze domowym użytkownika.

Kontener można uruchomić z opcją `-d` w tzw. trybie odłączonym *(ang. detached)*. W takim przypadku należy komunikować się z naszym kontenerem poprzez sieć. Podanie parametru `-p <port_hosta>:<port_kontenera>` (przykładowo: `-p 8080:80`) mapuje port 80 z kontenera na port 8080 naszego komputera. Domyślnie możemy stosować jako adres hosta localhost albo 127.0.0.1 w przypadku stosowania Docker Toolbox'a należy podać fizyczny adres ip naszego komputera.

Przykładowo komenda:

[8] można korzystać ze znaków specjalnych * i ?.

[9] W przypadku korzystania z kontenerów Linuxowych pod Windows dostęp do pliku będzie utrudniony, ponieważ będzie się on znajdował się na maszynie wirtualnej Hyper-V o domyślnej nazwie MobyLinuxVM w której pracuje Docker.

[10] W Powershell'u znak tyldy ~ oznacza folder domowy użytkownika w przypadku wiersza poleceń wymagana będzie ścieżka dostępu: `/C/Users/><nazwa_uzytkownika>`.

```
docker run --name my_nginx -d -p 80:80 nginx
```

spowoduje uruchomienie web serwera nginx na porcie 80 hosta: http://localhost/. Możemy sprawdzić, że kontener znajduje się na liście aktywnych kontenerów:

```
docker ps
```

Proszę zatrzymać kontener (`stop`) i sprawdzić, że komenda `ps` już go nie ujmuje.

Do procesu uruchomionego w tle możemy się dołączyć, służy do tego komenda `attach`[11]. Uruchamiamy w trybie odłączonym kontener Ubuntu z uruchomionym, w trybie wsadowym (opcja -b) menedżerem procesów `top`:

```
docker run -d --name topdemo ubuntu /usr/bin/top -b
```

Może zaistnieć konieczność zdefiniowania zmiennej środowiskowej `TERM=linux` wymaganej przez `top` należy, w tym celu należy wykorzystać opcję `-e`:

```
docker run -d --name topdemo ubuntu [-e TERM=linux] -t /usr/bin/top -b
```

Aby dołączyć się do uruchomionego kontenera korzystamy ze wspomnianej komendy `attach`:

```
docker attach topdemo
```

Kombinacja klawiszy Ctrl+z spowoduje zakończenie pracy kontenera, możemy to sprawdzić wartość wyjściową (0 – brak błędów):

```
echo $?
```

oraz sprawdzić stan kontenera:

```
docker ps -a | grep topdemo
```

Możemy opuścić działający kontener (bez zatrzymywania go) kombinacją Ctrl+p+q. Aby powrócić do niego korzystamy z:

```
docker attach <NAME|CONTAINER_ID>
```

Metoda `attach` dołącza nas do procesu o PID 1 wewnątrz kontenera. Sprawdzenie procesów wewnątrz kontenera (inne numery PID):

```
ps -ef
```

Sprawdzenie logów kontenera:

```
docker logs <NAME|CONTAINER_ID>
```

Możemy zobaczyć listę aktywnych procesów w kontenerze:

```
docker top <NAME|CONTAINER_ID>
```

Możemy uzyskać szczegółowe informacje o obiekcie (kontenerze, obrazie…), np. ustawienia sieciowe (dane w tablicy JSON z pliku config.json kontenera):

```
docker inspect <NAME| ID>
```

W kolejnym kroku, jako prosty przykład spróbujmy spakować i uruchomić prostą aplikację spring na serwerze Tomcat[12]. W pierwszym kroku należy wygenerować prostą aplikację wykorzystując Maven'a:

[11] https://docs.docker.com/engine/reference/commandline/attach/
[12] http://geekyplatypus.com/packaging-and-serving-your-java-application-with-docker/

```
docker run --rm -it -v $(pwd[13]):/external -w /external maven mvn archetype:generate
-DgroupId=com.mikechernev.docker.example -DartifactId=DockerExample -
DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false
```

Kod aplikacji DockerExample powinien być dostępny w katalogu domowym użytkownika. Następnie należy zbudować war'a za pomocą Mavena (UWAGA komenda musi być uruchomiona z głównego katalogu projektu DockerExample):

```
docker run --rm -it -v $(pwd):/project -w /project maven mvn package
```

opcja `-w` zmienia domyślny katalog roboczy (working directory) lub:

```
docker run --rm -it -v $(pwd):/project maven mvn package -f /project
```

Zbudowana aplikacja (DockerExample.war) powinna być widoczna w folderze target projektu DockerExample. Na koniec pozostaje uruchomienie zbudowanej aplikacji w serwerze Tomcat (UWAGA komenda musi być uruchomiona z głównego katalogu projektu DockerExample):

```
docker run -it -p 8080:8080 -v
$(pwd)/target/DockerExample.war:/usr/local/tomcat/webapps/docker-example.war
tomcat
```

Po czym wpisujmy w przeglądarce następujący URL: http://localhost:8080/docker-example.

W kontenerach nie powinniśmy przetrzymywać danych (np. danych z bazy danych, plików konfiguracyjnych, plików użytkowników itp.). Rozważmy to w praktyce na podstawie serwera nextcloud. Można po prostu uruchomić kontener z aplikacją:

```
docker run -d --name nc -p 8080:80 nextcloud
```

Działający serwer będzie dostępny w przeglądarce pod następującym URL: http://localhost:8080/.

W powyższym przypadku pliki użytkowników, konfiguracyjne itp. znajdują się wewnątrz kontenera i zostaną utracone po jego usunięciu. Możemy łatwo temu zapobiec dodając przekierowanie tych plików na folder hosta (w tym przypadku należy uzbroić się w cierpliwośc ponieważ):

```
docker run -d --name nc -p 8080:80 -v /<dysk>/<sciezka_na_hoscie>:/var/www/html/
nextcloud
```

Domyślnie kontener używa bazy danych SQLite do przechowywania dodatkowych danych, ale kreator konfiguracji Nextcloud, który pojawia się przy pierwszym uruchomieniu kontenera umożliwia połączenie z istniejącą bazą danych MySQL/MariaDB albo PostgreSQL. Bazę danych również możemy uruchomić w kontenerze w taki sposób aby przechowywać dane na hoście a dostęp będzie się odbywał po domyślnym porcie 3306. Warto nadać jej nazwę (w poniższym przypadku mariadb) ponieważ będziemy musieli jej użyć podczas łączenia z kontenerem nexcloud:

```
docker run –d --name mariadb -p 3306:3306 --env
MARIADB_ROOT_PASSWORD=<haslo_roota> -v /<dysk>/<sciezka_na_hoscie>:/var/lib/mysql
mariadb
```

Teraz uruchomiamy kontener z nextcloud'em wykorzystując nasza bazę danych. W tym celu wykorzystamy łącze, które pozwala kontenerom na wzajemne wykrywanie się i bezpieczne przesyłanie informacji. Konfigurując łącze, tworzymy kanał między kontenerem źródłowym (nextcloud) a odbiorczym (mariadb). Odbiorca może wtedy uzyskać dostęp do wybranych danych o źródle.

[13] %CD% w przypadku dockera dla Windows

```
docker run -d --name nc --link mariadb:mysql -e MYSQL_DATABASE=ncdb -e
MYSQL_HOST=mariadb:3306 -e MYSQL_USER=root -e MYSQL_PASSWORD=<haslo_roota> -p
8080:80 -v /e/ncdir:/var/www/html/ nextcloud
```

Ponownie wpisujmy w przeglądarce następujący URL: http://localhost:8080/.

Na zakończenie pracy z Docker'em pomocną może okazać się możliwość zatrzymania i usunięcia wszystkich kontenerów oraz obrazów (działa pod PoweShell'em).

```
docker stop $(docker ps -a -q)
docker rm $(docker ps -a -q)
docker rmi $(docker images -q) [-f]
```

Windows

Przykład uruchomienia .NET'owego Hello-World (wymaga przełączenia Dockera do obsługi kontenerów Windows):

```
docker run microsoft/dotnet-samples:dotnetapp-nanoserver
```

Uruchomienie nanoserwer'a w trybie interaktywnym:

```
docker run -it microsoft/nanoserver cmd
```

Stworzenie prostego skryptu wewnątrz kontenera:

```
powershell.exe Add-Content C:\helloworld.ps1 'Write-Host "Hello World"'
```

Do wyjścia z kontenera służy komenda `exit`. Teraz możemy zobaczyć nasz zmodyfikowany kontener wraz z jego id.

```
docker ps –a
```

W celu utrwalenia zmian utworzymy z niego nowy obraz (jako *containerid* wystarczy podać tylko unikalny początkowy ciąg ID)::

```
docker commit <NAME|CONTAINER ID> helloworld
```

Powinien być już widoczny wśród lokalnych obrazów:

```
docker images
```

Możemy zweryfikować działanie nowego obrazu uruchamiając nasz skrypt wewnątrz kontenera:

```
docker run --rm helloworld powershell c:\helloworld.ps1
```

Przykłady do samodzielnego uruchomienia:

- Django – https://github.com/atbaker/docker-django[14]
- ASP+IIS – https://github.com/friism/MusicStore

4. Tworzenie obrazu Docker'a[15]

Przykład 1:

[14] http://docker.atbaker.me/exercises/exercise_3.html
[15] https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

Następujące polecenie spowoduje pobranie najnowszej wersji obrazu oraz uruchomienia w powłoce komendy tworzącej plik w napisem `Docker is okey`:

```
docker run ubuntu bash -c  "echo 'Docker is okey' > /tmp/docker-ok.txt"
```

Aby polecenie `echo 'Docker is okey' > /tmp/docker-ok.txt` wykonało się wewnątrz kontenera należy ująć je w nawiasy, w przeciwnym przypadku wykonane zostanie w systemie hosta. Możemy sprawdzić, że kontener został utworzony i zapamiętać jego namiary:

```
docker ps –a
```

Komenda `commit` pozwoli na zapisanie utworzonych zmian do nowego obrazu, który będziemy mogli wykorzystywać w przyszłości:

```
docker commit <NAME|CONTAINER ID> ubuntu_ok
```

Na liście dostępnych obrazów powinien pojawić się `ubuntu_ok:latest`:

```
docker images
```

Polecenie `history` pozwala prześledzić nam historię tworzenia obrazu (poszczególne warstwy):

```
docker history ubuntu_ok
```

Każda warstwa jest powiązana z jakimś poleceniem, np. instalacją pakietu, stworzeniem zmiennej środowiskowej, utworzeniem pliku. Obraz możemy zapisać do archiwum oraz wczytać przy pomocy komend `save` i `load`:

```
docker save -o d:/tmp/ubuntu_ok.tar ubuntu_ok
docker load -i d:/tmp/ubuntu_ok.tar
```

Uruchamiamy nasz kontener w trybie interaktywnym:

```
docker run –it ubuntu_ok bash
```

i sprawdzamy zawartość pliku `/tmp/docker-ok.txt`:

```
cat /tmp/docker-ok.txt
```

Przykład 2:

Załóżmy, że chcemy zbudować obraz Docker'a pozwalający na uruchomienie prostej aplikacji Node.js[16]. Jako obraz bazowy wybieramy node:carbon uruchamiając w nim proces powłoki (bash) w trybie interaktywnym dodatkowo udostępniając wewnętrzny port kontenera 8080 kontenera na porcie 49160 hosta.

```
docker run -it -p 49160:8080 node:carbon bash
```

Tworzymy i przechodzimy do folderu naszej aplikacji:

```
mkdir /usr/src/app
cd /usr/src/app
```

W celu skopiowania plików aplikacji musimy opuścić kontener: Ctrl+p+q. Następnie sprawdzamy nazwę albo id naszego kontenera (powinien być jako jedyny na liście aktywnych kontenerów):

```
docker ps
```

Nasz aplikacja składa się z dwóch prostych plików. opisującego aplikację i jej zależności (`package.json`):

[16] Przykład bazujący na artykule: https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

```
{
  "name": "docker_web_app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "author": "Imie nazwisko <local-part@domain>",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.16.1"
  }
}
```

oraz aplikacji internetowej opartej o framework Express.js (`server.js`):

```
'use strict';

const express = require('express');

// Constants
const PORT = 8080;
const HOST = '0.0.0.0';

// App
const app = express();
app.get('/', (req, res) => {
  res.send('Hello world\n');
});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

Kopiujemy powyższe pliki z poziomu hosta (`package.json` i `server.js` musi znajdować się w bieżącym folderze):

```
docker cp package.json  <NAME|CONTAINER ID>:/usr/src/app
docker cp server.js  <NAME|CONTAINER ID>:/usr/src/app
```

Jesteśmy gotowi aby powrócić do kontenera:

```
docker attach <NAME|CONTAINER ID>
```

Możemy uruchomić naszą aplikację;

```
npm install
npm start
```

W przypadku wykorzystania docker-machine w celu sprawdzenia sprawdzenia wyników działania aplikacji należy zobaczyć pod jakim adresem znajduje się maszyna Dockera:

```
docker-machine ip
```

następnie możemy odpytać nasz kontener:

```
curl -i <localhost|ip>:49160
```

działa to również w przeglądarce hosta, pod adresem: <localhost|ip>:49160 powinna wyświetlić się strona z napisem Hello World. Po opuszczeniu kontenera możemy przy pomocy komendy commit utworzyć z niego nowy obraz. Znając nazwę albo id kontenera robimy to w następujący sposób:

```
docker commit <NAME|CONTAINER ID>  [uzytkownik/]nazwa_obrazu[:tag]
```

Tak zbudowany obraz możemy uruchomić przy pomocy komend run/start.

Przykład 3:

W celu zautomatyzowania procesu tworzenia obrazu w pomocą przychodzi nam plik `Dockerfile`. Oto jedna z najprostszych jego wersji:

```
FROM ubuntu
CMD ["/bin/echo", "hello world"]
```

W celu zbudowania obrazu wydajemy komendę (zakładamy, że `Dockerfile` jest w bieżącym katalogu, inaczej należy zamiast kropki podać do niego ścieżkę):

```
docker build –t <nazwa_obrazu> .
```

Sprawdzenie czy obraz istniej na liście:

```
docker images
```

Uruchomienie obrazu:

```
docker run <nazwa_obrazu>
```

Przykład 4:

Wszystkie kroki wykonane w przykładzie 2 możemy zapisać w pliku `Dockerfile`:

```
FROM node:carbon
WORKDIR /usr/src/app
COPY package.json ./
COPY server.js ./
RUN npm install
EXPOSE 8080
CMD [ "npm", "start" ]
```

opcjonalnie dodając plik `.dockerignore`:

```
node_modules
npm-debug.log
```

Zbudowanie obrazu:

```
docker build -t [uzytkownik/]nazwa_obrazu[:tag] .
```[17]

[17] Po zbudowaniu obrazu możemy otrzymać następujące ostrzeżenie:
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build context will have '-rwxr-xr-x' permissions. It is recommended to double check and reset permissions for sensitive files and directories.

Przykład 5:

W kolejnym przykładzie zbudujemy obraz uruchamiający prostą aplikację konsolową .NET w Mono. Potrzebne będą nam dodatkowe dwa pliki, kod aplikacji (*.cs):

```
using System;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World !!!");
        }
    }
}
```

oraz plik projektu (*.csproj):

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <Compile Include="<nazwa_pliku_z_kodem_aplikacji>.exe" />
  </ItemGroup>
  <Target Name="Build">
    <Csc Sources="@(Compile)"/>
  </Target>
</Project>
```

Dockerfile bazujący na obrazie Mono kopiujący powyższe pliki, oraz przeprowadzający proces kompilacji:

```
FROM mono
COPY <nazwa_pliku_projektu>.csproj ./
COPY <nazwa_pliku_z_kodem_aplikacji>.cs ./
RUN mono --version
RUN msbuild ./ <nazwa_pliku_projektu>.csproj
CMD [ "mono",  "<nazwa_pliku_z_kodem_aplikacji>.cs" ]
```

Przykład 6:

Tę samą aplikację możemy skompilować i uruchomić korzystając z obrazu microsoft/dotnet. Tym razem wykorzystamy dwa obrazy: `microsoft/dotnet:2.1-sdk` potrzebny tylko podczas budowania aplikacji oraz `dotnet:2.1-runtime` do uruchomienia naszej aplikacji. Oto nowy Dockerfile:

```
FROM microsoft/dotnet:2.1-sdk AS build
WORKDIR /app
COPY DotNetTestApp.csproj ./
COPY *.cs ./
RUN dotnet restore
RUN dotnet publish -c Release -o out
```

Oznacza to że podczas kopiowania z systemu Windows (hosta) do systemu Linux (kontener) pliki są domyślnie oznaczone jako wykonywalne. Ostrzeżenie przypomnia aby zmodyfikować `Dockerfile`, usuwając niepotrzebne atrybuty poleceniem `chmod -x`.

```
FROM microsoft/dotnet:2.1-runtime AS runtime
WORKDIR /app
COPY --from=build /app/out ./
ENTRYPOINT ["dotnet", "DotNetTestApp.dll"]
```

oraz plik projektu (*.csproj):

```
<Project Sdk="Microsoft.NET.Sdk" ToolsVersion="15.0">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>
</Project>
```

Pierwszy obraz `microsoft/dotnet:2.1-sdk` zostanie wykorzystany jedynie do kompilacji. Produkty kompilacji zostaną skopiowane do kolejnego obrazu `dotnet:2.1-runtime`, który będzie obrazem bazowym. Różnice w wielkości można zauważyć kompilując drugi obraz bez pogrubionych wierszy Dockerfile i sprawdzając komendą:

```
docker images
```

Budowa Dockerfile:

Jak wspomniano wcześniej, opis budowy obrazu umieszczony jest w pliku `Dockerfile`, można w nim znaleźć ciąg następujących komend (każda z nich będzie odpowiedzialna za utworzenie nowej warstwy obrazu):

`FROM <image>` – tą komendą musi zaczynać się każdy Dockerfile, określa ona inny obraz jako warstwę bazową nowo tworzonego. Jeśli to tylko możliwe powinniśmy korzystać z oficjalnych repozytoriów[18].

`FROM <image>:<tag>` – po dwukropku możemy określić dokładnie którą wersję chcemy pobrać. Domyślnie będzie to `latest` najnowsza.

`FROM <image>@<digest>` – dla obrazów wersji drugiej (v2) lub wyższej możemy po małpie możemy podać skrót sha256 obrazu.

Dockerfile file może zawierać kilka instrukcji `FROM`, każda definiuje osobny etap budowania obrazu.

`FROM <image> AS <nazwa>` – przypisuje nazwę aktualnemu etapowi budowanego obrazu, może być użyta w instrukcjach **`FROM`** oraz **`COPY`** aby odnieść się do obrazu zbudowanego na tym etapie, przykładowo instrukcja:

```
COPY --from=<name|index> <targ_path> <dest_path>
```

wywołana z innego etapu kopiuje pliki do katalogu `<dest_path>` bieżącego etapu z katalogu `<targ_path>` etapu o nazwie bądź indeksie `<name|index>`.

Lista wybranych obrazów bazowych:

- `scratch` – pusty obraz wykorzystywany do budowy innych przeważnie małych obrazów zawierających jedynie pojedyncze binaria oraz wymagane przez nie zasoby. Nie dodaje on warstwy do obrazu zatem nowy będzie składał się tylko z jednej warstwy,
- `nginx`, `alpine`, `python`, `node`, `Ubuntu`, `php`, `mysql`

[18] https://hub.docker.com/explore/

- `microsoft/nanoserver`[19], `microsoft/dotnet`, `microsoft/dotnet-samplesmicrosoft/windowsservercore`, `microsoft/aspnet`,`microsoft/iis`.

`LABEL`[20] – etykieta to unikalna w obrębie obiektu Dockera para stringów klucz-wartość. Obiektem może być obraz, kontener, lokalny demon itp…, i może zawierać dowolną liczbę etykiet. Służą m.in. do porządkowania obrazów, zapisu informacji licencyjnych oraz opisu powiązań między kontenerami, woluminami i sieciami.

`LABEL maintainer="email@domena"` – informacja na temat osoby lub grupy opiekującej się danym kontenerem. Wcześniej była to instrukcja `MAINTAINER <name>` (aktualnie przestarzała).

`RUN <command>` – uruchomienie wybranej komendy systemowej już w nowym systemie kontenera. Polecenie uruchamiane jest w powłoce, domyślnie `/bin/sh -c` (Linux) albo `cmd /S /C` (Windows).

`RUN ["executable", "param1", "param2"]` – uruchamiany jest (bez udziału powłoki) plik wykonywalny z podanymi argumentami.`COPY` – kopiuje lokalne pliki do kontenera.

`CMD` – definiuje polecenia, które będą uruchamiane w obrazie podczas uruchamiania. W przeciwieństwie do RUN, nie tworzy to nowej warstwy obrazu, ale po prostu uruchamia polecenie. W każdym pliku Dockerfile może występować tylko jedna komenda CMD. Aby uruchomić wiele poleceń, najlepszym sposobem na to jest uruchomienie skryptu przez CMD. CMD wymaga, aby powiedzieć, gdzie można uruchomić polecenie, w przeciwieństwie do RUN. Oto przykłady komend CMD:

- `CMD ["python", "./app.py"]`
- `CMD ["/ bin / bash", "echo", "Hello World"]`

`EXPOSE` – prosty sposób na konfigurację nasłuchiwania na wybranych portach.

`VOLUME <współdzielony system_plików>` – pozwala na mapowanie katalogów systemu operacyjnego hosta[21]

W kontenerze powinniśmy umieścić tylko wymagane pakiety. W tym celu przydatnym okazuje się plik `.dockerignore`.

```
# komentarz
    */temp*
    */*/temp*
    temp?
```

W powyższym przykładowym pliku `.dockerignore` gwiazdka zastępuje dowolny ciąg znaków a znak zapytania jeden dowolny znak.

Standardy „best practice” odnośnie tworzenia obrazów Deckera:

- używaj `.dockerignore` co pozwala na uwzględnienie tylko niezbędnego kontekstu,
- unikaj instalowania niepotrzebnych pakietów – zmniejszając rozmiar obrazu,
- jeśli to możliwe umieszczaj najczęściej zmieniający się kontekst (np. kod programu) jak najpóźniej, pozwoli to na lepsze wykorzystanie pamięci podręcznej,
- dane należy przetrzymywać na zewnątrz kontenera używając woluminów,
- uważaj na woluminy. Należy pamiętać, jakie dane znajdują się w woluminach, ponieważ są one trwałe i nie znikają z kontenerami. Następny kontener użyje danych z woluminu utworzonego przez poprzedni kontener,
- używanie zmiennych środowiskowych (np. w instrukcjach RUN, EXPOSE, VOLUME) powoduje, że `Dockerfile` jest bardziej elastyczny,
- w jednym kontenerze powinna być uruchamiana pojedyncza aplikacja,

[19] Wymaga natywnego kontenera Windows, dostępnego jedynie pod Windows 10 professional albo Windows Server 2016. Polecenie `docker version` powinno wyświetlać atrybut OS/Arch dla Servera ustawiony na `windows/amd64`.

[20] https://docs.docker.com/engine/userguide/labels-custom-metadata/

[21] Więcej informacji o woluminach można znaleźć tutaj: https://docs.docker.com/storage/volumes/

- procesy należy uruchamiać w wątku pierwszoplanowym (foreground), nie należy używać `systemd`, `upstart` ani żadnych innych podobnych narzędzi,
- nie używaj SSH (jeśli chcesz wejść do kontenera, możesz użyć polecenia `exec`),
- unikaj ręcznych konfiguracji (lub akcji) wewnątrz kontenera.

5. Docker Compose[22]

Kolejnym krokiem we współpracy z kontenerami Dockera jest Docker Compose. Pozwala on na jednocześnie uruchomienie wielu kontenerów wraz z konfiguracją między innymi takich kwestii jak: zależności oraz komunikacja pomiędzy kontenerami. Polecenia Docker Compose umieszczamy w pliku YAML: `docker-compose.yml(.yaml)`. Mogą one zawierać następujące sekcje[23]:

`version` – wersja Docker Compose.

`services` – sekcja, w której definiujemy serwisy.

`image` – nazwa obrazu z którego ma być zbudowany kontener.

`build` – ścieżka do pliku Dockerfile naszego serwisu.

`ports` – mapowanie portów z kontenera w formacie: "HOST:KONTENER". Zaleca się używać stringów.

`links` – zależność pomiędzy kontenerami, informacja z usług którego kontenera możemy korzystać.

`environment` – zmienna środowiskowa dla danego kontenera.

`command` – definiuje polecenia, które będą uruchamiane w obrazie podczas uruchamiania. Odpowiednik `CMD` z Dockerfile.

`volumes` – definiuje wolumin.

`depends_on` – wyraża zależność między serwisami, co wpływa na kolejność ich uruchamiania i zatrzymywania.

O bardziej zaawansowanych opcjach Docker Compose można uzyskać informacje na stronie: https://runnable.com/docker/advanced-docker-compose-configuration.

Przykład 1:

Konfiguracja i uruchomienie prostej aplikacji Django/PostgreSQL[24]. Należy utworzyć trzy pliki:

1. `Dockerfile:`

```
FROM python:3
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
COPY requirements.txt /code/
RUN pip install -r requirements.txt
COPY . /code/
```

2. `requirements.txt:`

```
Django>=2.0,<3.0
```

22 https://docs.docker.com/compose/overview/
23 https://docs.docker.com/compose/compose-file/
24 https://docs.docker.com/compose/django/

```
psycopg2>=2.7,<3.0
```

3. `docker-compose.yml:`

version: '3'

```
services:
  db:
    image: postgres
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    depends_on:
      - db
```

Plik `docker-compose.yml` definiuje dwa serwisy, serwer WEB (`web`) i bazę danych (`db`) oraz określa również których obrazów Dockera one używają. Dla bazy danych jest to obraz `postgres`, natomiast dla serwera WEB (web) jest to obraz który będzie zbudowany w oparciu o plik Dockerfile z bieżącego katalogu. Serwis dostępny będzie pod adresem `8000`. Sekcja `depends_on` gwarantuje, że serwis bazy danych zostanie uruchomiony przed serwerem WEB.

W celu utworzenia projektu należy wywołać poniższą komendę:

```
docker-compose run web django-admin startproject composeexample .
```

Instruuje ona Compose, aby uruchomił `django-admin startproject composeexample` wewnątrz kontenera, używając obrazu web i konfiguracji. Obraz web jeszcze nie istnieje i zostanie zbudowany. Po jego utworzeniu Compose uruchamia go i wykonuje polecenie `django-admin startproject composeexample` w kontenerze. Polecenie to nakazuje Django utworzyć zbiór plików i katalogów projektu w folderze `composeexample`.

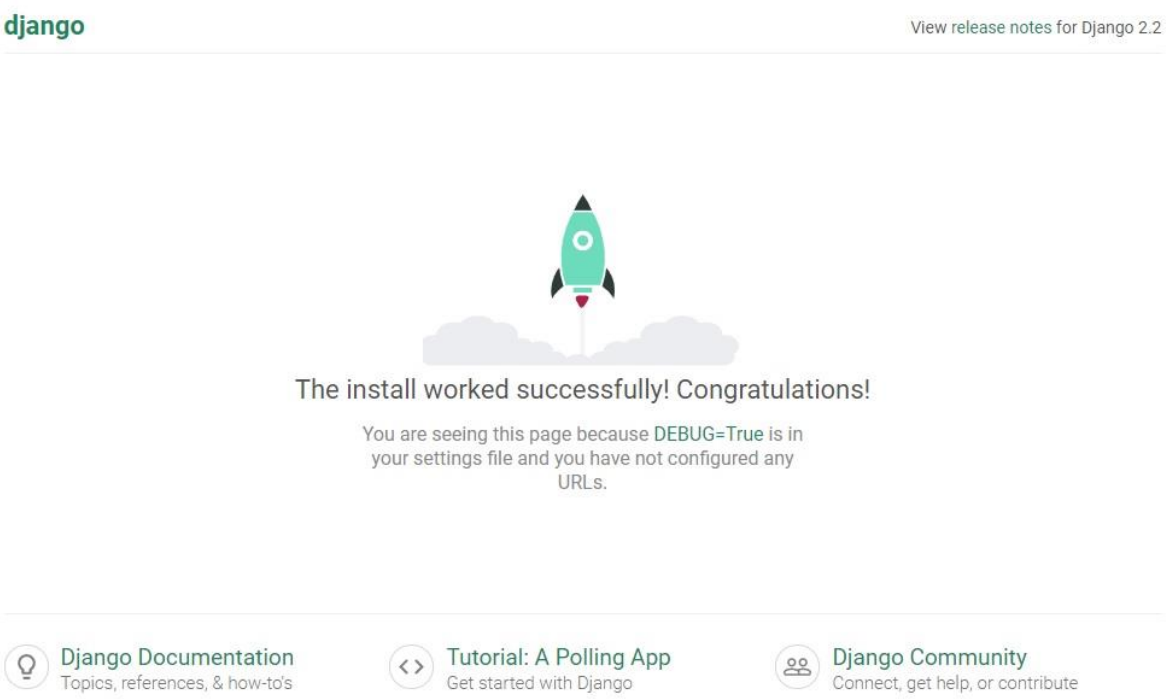
Przed uruchomieniem aplikacji należy skonfigurować połączenie bazy danych z Django. W tym celu w pliku konfiguracyjnym: `composeexample/settings.py` edytujemy sekcję `DATABASES` :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

Pozostało uruchomienie tak skonfigurowanej aplikacji komendą:

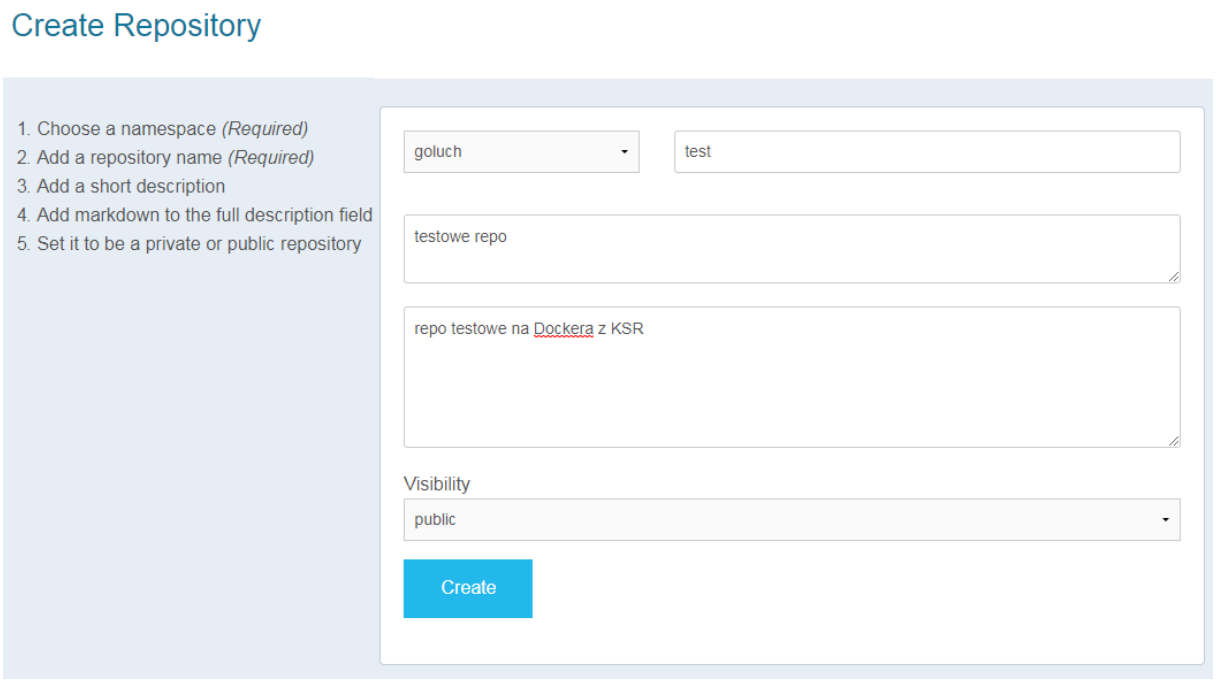
```
docker-compose up
```

Jeśli wszystko zadziała poprawnie otrzymamy pod adresem http://localhost:8000 następujący wynik:



6. Publikowanie obrazu Docker'a

Najbardziej popularnym i jednocześnie najbogatszym repozytorium obrazów Dockera jest Dockerhub: https://hub.docker.com/. W darmowym pakiecie otrzymujemy darmowe prywatne repozytorium oraz nielimitowaną liczbę repozytoriów publicznych. Po założeniu konta oraz zalogowaniu się klikamy przycisk Create Repository+. Pojawi się okno Create Repository w którym podajemy: nazwę obrazu, krótki i pełny opis oraz ustalamy widoczność (publiczne/prywatne), a następnie wybieramy Create.



Powinno być widoczne w zakładce „Repo Info" menu. W celu przesłania obrazu do repozytorium należy zalogować się w konsoli do naszego konta Dockera:

```
docker login
```

i wysłać wcześniej zbudowany obraz:

```
docker push [uzytkownik/]nazwa_obrazu[:tag]
```

Uwaga, wymagany jest prefix użytkownika jeśli tego nie zrobiliśmy to nalezy udostępnić obraz pod nową nazwą:

```
docker image tag nazwa_obrazu[:tag] uzytkownik/nazwa_obrazu[:tag]
```

Od teraz możemy normalnie korzystać z obrazu przy pomocy komend `PULL`, `RUN` oraz możemy go wykorzystywać jako obraz bazowy Dockrfile:

```
docker pull/run [uzytkownik/]nazwa_obrazu:tag
```

```
FROM [uzytkownik/]nazwa_obrazu
```

7. Lokalny rejestr Dockera

Repozytorium Deckera tzw. Docker Registry dostępne jest w postaci obrazu i nic nie stoi na przeszkodzie aby uruchomić je lokalnie. Pobranie i uruchomienie rejestru Docker Registry:
```
docker run -d -p 5000:5000 --name registry registry
```

Pobieramy obraz z huba:
```
docker pull microsoft/dotnet
```

Oznaczenie obrazu aby wskazywał na nowy rejestr:
```
docker image tag microsoft/dotnet localhost:5000/newimagename
```

Przesłanie obrazu do nowego rejestru:
```
docker push localhost:5000/newimagename
```

Teraz można ściągać obraz z lokalnego rejestru:
```
docker pull localhost:5000/newimagename
```

Zatrzymanie rejestru i usunięcie całej zawartości:
```
docker container stop registry && docker container rm -v registry
```

W przypadku korzystania z naszego repozytorium z innego komputera musimy dodać je do listy niezabezpieczonych w pliku /etc/docker/daemon.json:

```
{
...
  "insecure-registries": ["<adres_ip>:<port>"]
}
```

Innym sposobem jest dodanie swojego rejestru do polecenia start:

```
dockerd --unregister-service
dockerd --register-service -G docker -H npipe:// --insecure-registry
<adres_ip>:<port>
```

Dzieje się tak ponieważ nie używaliśmy certyfikatów do zabezpieczenia rejestru.

Niestety Docker Registry nie obsługuje kontenerów Windows. Jednak w takim przypadku możemy wykorzystać prywatny obraz np. https://hub.docker.com/r/stefanscherer/registry-windows/:

```
docker run -d -p 5000:5000 --restart=always --name registry -v C:\registry:C:\registry stefanscherer/registry-windows:2.6.2
```

Należy pamiętać o wcześniejszym utworzeniu katalogu registry na dysku C oraz o oznaczeniu repozytorium jako niezabezpieczonego (plik C:\ProgramData\docker\config\daemon.json albo).

8. Debugowanie

Polecenie `exec` pozwala na zdalne uruchomienie powłoki w działającym kontenerze:

```
docker exec –i –t <NAME|CONTAINER_ID>
```

Narzędzie nsenter wymaga pobrania i zapamiętania w zmiennej środowiskowej numeru PID kontenera do którego chcemy uzyskać dostęp:

```
PID=$(docker inspect –format \{{.State.Pid\}} <NAME|CONTAINER_ID>)
```

Uruchomienie nsenter wymaga jawnego podania elementów kontenera do których chcemy mieć dostęp:

```
nsenter –target $PID –mount –uts –ipc –net –pid
```

9. Błędy

- **no matching manifest for unknown in the manifest list entries** – Należy zmienić tryb z kontenerów Windows na Linux.

- **docker: invalid reference format** – Sprawdzić czy podczas kopiowania instrukcji nie ma dziwnych znaków (np. myślnik zamiast łącznika/minusa). Jako rozwiązanie można przekopiować komendę do notatnika, a następnie ponownie z niego skopiować do wiersza poleceń lub PowerShell'a. Jeśli to nie pomoże najlepiej przepisać ręcznie komendę.

10. Literatura

- **https://www.docker.com/blog/9-tips-for-containerizing-your-net-application/** - (9 Tips for Containerizing Your .NET Application)