

ARKADIUSZ SICZEK | KRZYSZTOF GODZISZ

FUNDAMENTY DOCKERA I KUBERNETESA



Spis treści

| | |
|--|----|
| Rozdział 1: Podstawowe informacje i wyjaśnienia | 9 |
| 1.1. Co to jest Docker?..... | 9 |
| 1.2. Zapoznanie z pomocą polecenia | 9 |
| 1.3. Co to jest obraz?..... | 13 |
| 1.4. Co to jest kontener?..... | 14 |
| 1.5. Dockerfile i compose.yaml | 14 |
| 1.6. Jak wyglądają sieci w Dockerze..... | 14 |
| 1.7. Podsumowanie | 15 |
| Rozdział 2: Instalacja Dockera w systemach Linux | 16 |
| 2.1. Dystrybucja Debian | 16 |
| 2.1.1. Instalacja z wykorzystaniem repozytorium Debiana | 16 |
| 2.1.2. Instalacja z wykorzystaniem repozytorium Dockera | 17 |
| 2.1.3. Instalacja określonej wersji Dockera | 18 |
| 2.2. Dystrybucja Ubuntu | 19 |
| 2.2.1. Instalacja z wykorzystaniem repozytorium Ubuntu | 19 |
| 2.2.2. Instalacja z wykorzystaniem repozytorium Dockera | 20 |
| 2.2.3. Instalacja określonej wersji Dockera | 21 |
| 2.3. Dystrybucja Fedora | 22 |
| 2.3.1. Instalacja z wykorzystaniem repozytorium Fedory | 22 |
| 2.3.2. Instalacja z wykorzystaniem repozytorium Dockera | 23 |
| 2.3.3. Instalacja określonej wersji Dockera | 24 |
| 2.4. Dystrybucja CentOS..... | 25 |
| 2.4.1. Instalacja z wykorzystaniem repozytorium CentOS | 25 |
| 2.4.2. Instalacja z wykorzystaniem repozytorium dockera | 26 |
| 2.4.3. Instalacja określonej wersji Dockera | 27 |
| 2.5. Instalacja z paczki | 28 |
| 2.5.1. Instalacja dla systemu Debian oraz Ubuntu | 28 |
| 2.5.2. Dla systemu Fedora..... | 29 |
| 2.5.3. Dla systemu CentOS | 30 |
| 2.6. Instalacja Dockera przy pomocy plików binarnych | 31 |
| 2.7. Sprawdzenie czy instalacja zakończyła się powodzeniem..... | 32 |
| 2.8. Błędy z jakimi możesz się spotkać..... | 32 |
| 2.8.1. Błąd podczas uruchomienia | 32 |
| 2.8.2. Zduplikowane repozytoria | 33 |

| | |
|---|----|
| 2.8.3. Brak uprawnień | 33 |
| 2.9. Podsumowanie..... | 33 |
| Rozdział 3: Docker - obrazy | 34 |
| 3.1. Pobieramy nasz pierwszy obraz..... | 34 |
| 3.2. Wyświetlanie listy obrazów | 34 |
| 3.3. Skąd pobierane są obrazy | 35 |
| 3.4. Typy obrazów | 35 |
| 3.5. Różne wersje obrazów | 36 |
| 3.6. Usuwanie zbędnych obrazów | 38 |
| 3.7. Wyszukiwanie obrazów w Dockerze..... | 39 |
| 3.8. Podsumowanie..... | 42 |
| Rozdział 4: Docker - kontenery | 43 |
| 4.1. Pierwszy kontener..... | 43 |
| 4.2. Tworzymy kontener | 43 |
| 4.3. Analiza przykładu | 44 |
| 4.4. Uruchomienie kontenera | 45 |
| 4.5. Nadanie nazwy kontenerowi | 45 |
| 4.6. Usuwanie kontenerów | 46 |
| 4.7. Pobranie obrazu oraz uruchomienie kontenera | 47 |
| 4.8. Zmiana nazwy kontenera | 49 |
| 4.9. Uruchomienie kontenera w trybie ciągłym | 50 |
| 4.10. Zamknięcie kontenera | 51 |
| 4.11. Uruchomienie powłoki kontenera | 51 |
| 4.12. Podsumowanie..... | 52 |
| Rozdział 5: Docker - Sieci..... | 53 |
| 5.1. Po co używać sieci | 53 |
| 5.2. Automatyczna sieć Dockera | 53 |
| 5.3. Dbanie o porządek | 55 |
| 5.4. Inspekcja kontenera | 56 |
| 5.5. Dlaczego tak | 59 |
| 5.6. Tworzenie własnej sieci | 59 |
| 5.7. Dodanie do sieci | 60 |
| 5.8. Inspekcja sieci..... | 60 |
| 5.9. Odłączenie od sieci..... | 62 |
| 5.10. Używamy nazwy kontenera do komunikacji | 64 |
| 5.11. Podsumowanie..... | 64 |

| | |
|---|----|
| Rozdział 6: Docker - Wolumeny | 65 |
| 6.1. Tworzenie pierwszego wolumena | 65 |
| 6.2. Wyświetlanie listy | 65 |
| 6.3. Dane z kontenera | 65 |
| 6.4. Inspekcja kontenera | 66 |
| 6.5. Weryfikacja magazynowania danych | 67 |
| 6.6. Życotność danych | 67 |
| 6.7. Współdzielenie danych | 68 |
| 6.8. Tworzenie kontenerów z wolumenami | 69 |
| 6.9. Usuwanie zbędnych wolumenów | 70 |
| 6.10. Podsumowanie | 71 |
| Rozdział 7: Docker - Dockerfile | 72 |
| 7.1. Czym jest Dockerfile? | 72 |
| 7.2. Prosty obraz | 72 |
| 7.3. Nasz pierwszy Dockerfile | 73 |
| 7.4. Budujemy nasz pierwszy obraz | 73 |
| 7.5. tag – none | 74 |
| 7.6. Wywołanie polecenia „<i>w locie</i>” | 75 |
| 7.7. Bardziej pożyteczny przykład | 76 |
| 7.8. Apache2 plus PHP | 78 |
| 7.9. Poprawna forma zapisu | 79 |
| 7.10. Pozostałe słowa klucze Dockerfile | 80 |
| 7.10.1. FROM | 81 |
| 7.10.2. RUN | 81 |
| 7.10.3. CMD i ENTRYPOINT | 82 |
| 7.10.4. COPY i ADD | 83 |
| 7.10.5. EXPOSE | 84 |
| 7.10.6. ENV | 84 |
| 7.10.7. VOLUME | 84 |
| 7.10.8. USER | 84 |
| 7.10.9. WORKDIR | 84 |
| 7.11. Podsumowanie | 85 |
| Rozdział 8: Funkcje oficjalnych obrazów na bazie MySQL | 86 |
| 8.1. Wersje oraz Dockerfile | 86 |
| 8.2. Analiza wykonywanych czynności w pliku | 87 |
| 8.2.1. RUN | 87 |

| | |
|---|-----|
| 8.2.2. ENV | 87 |
| 8.2.3. Wolumen | 88 |
| 8.2.4. COPY | 88 |
| 8.2.5. ENTRYPOINT | 88 |
| 8.2.6. EXPOSE i CMD..... | 88 |
| 8.3. Inny plik | 88 |
| 8.4. Zmienne środowiskowe | 89 |
| 8.4.1. MYSQL_ROOT_PASSWORD..... | 89 |
| 8.4.2. MYSQL_DATABASE..... | 89 |
| 8.4.3. MYSQL_USER, MYSQL_PASSWORD | 89 |
| 8.4.4. MYSQL_ALLOW_EMPTY_PASSWORD..... | 89 |
| 8.4.5. MYSQL_RANDOM_ROOT_PASSWORD | 89 |
| 8.4.6. MYSQL_ONETIME_PASSWORD..... | 90 |
| 8.4.7. MYSQL_INITDB_SKIP_TZINFO | 90 |
| 8.5. Praktyczne użycie zmiennych środowiskowych..... | 90 |
| 8.6. Podsumowanie..... | 91 |
| Rozdział 9: Docker – Docker Compose..... | 92 |
| 9.1. Tworzymy plik | 92 |
| 9.2. Notacja plików | 92 |
| 9.3. Instalacja Docker Compose | 92 |
| 9.3.1. Dwie formy compose | 93 |
| 9.3.2. Wersja pliku..... | 93 |
| 9.3.3. Nagłówek pliku..... | 93 |
| 9.3.4. Elementy składowe pliku | 93 |
| 9.3.5. Apache2..... | 94 |
| 9.3.6. PHP | 96 |
| 9.4. Baza MySQL..... | 97 |
| 9.5. PHPMyAdmin | 98 |
| 9.6. Wszystkie pliki | 99 |
| 9.6.1. Compose.yaml | 99 |
| 9.6.2. ./apache/grupaadm.apache.conf | 100 |
| 9.6.3. ./apache/Dockerfile | 101 |
| 9.6.4. ./php/Dockerfile..... | 101 |
| 9.7. Budowa oraz uruchomienie serwera | 101 |
| 9.8. Pliki | 102 |
| 9.9. Sprawdzamy co zostało stworzone | 103 |

| | |
|--|-----|
| 9.10. Sieci..... | 105 |
| 9.11. Dostęp do panelu PHPMyAdmin..... | 108 |
| 9.12. Podsumowanie..... | 109 |
| Rozdział 10: Docker Swarm..... | 110 |
| 10.1. Jak wygląda konstrukcja Docker Swarm..... | 110 |
| 10.2. Docker vs Docker Machine vs Docker Swarm..... | 112 |
| 10.2.1. Docker, a Docker Swarm | 112 |
| 10.2.2. Co to jest Docker Machine i co ma wspólnego z Docker Swarm? | 112 |
| 10.2.3. Czy użycie Docker Machine jest konieczne? | 113 |
| 10.2.4. Inne sposoby na uruchomienie Docker Swarm | 113 |
| 10.2.5. Dlaczego użyłem VirtualBox..... | 114 |
| 10.2.6. Wymagania co do zasobów sprzętowych | 114 |
| 10.3. Pierwszy sposób: instalacja i konfiguracja ręczna w VirtualBox | 115 |
| 10.4. Sposób drugi: Instalacja przy pomocy docker-machine | 121 |
| 10.5. Zapoznanie się z pomocą polecenia..... | 121 |
| 10.6. Pierwszy etap tworzenia elementów klastra..... | 123 |
| 10.7. Drugi etap tworzenie klastra..... | 127 |
| 10.7.1. Połaczenie ssh w przypadku ręcznej instalacji..... | 127 |
| 10.7.2. Połaczenie ssh z użyciem docker-machine..... | 127 |
| 10.7.3. Dalsza konfiguracja niezależnie od wersji..... | 127 |
| 10.7.4. Ustawienie menadżera | 128 |
| 10.7.5. Dodanie węzłów wykonawczych | 129 |
| 10.8. Wyświetlanie zawartości naszego klastra | 129 |
| 10.9. Tworzenie i uruchamianie serwisu | 131 |
| 10.10. Odłączenie jednej z maszyn | 137 |
| 10.11. Aktualizacja obrazu | 138 |
| 10.12. Usuwanie klastra | 139 |
| 10.13. Usuwanie pozostałości z docker-machine | 140 |
| 10.14. Podsumowanie..... | 140 |
| Rozdział 11: Instalacja Kubernetes w dystrybucjach Red Hat Enterprise Linux i Debian..... | 141 |
| 11.1. Red Hat Enterprise Linux..... | 141 |
| 11.2. Debian | 141 |
| 11.3. Instalacja Dockera | 142 |
| 11.3.1. Instalacja w systemie Red Hat Enterprise Linux | 142 |
| 11.3.2. Instalacja w systemie Debian | 143 |
| 11.3.3. Dodanie do grupy oraz sprawdzenie czy Docker działa..... | 143 |

| | |
|--|-----|
| 11.4. Instalacja kubectl | 144 |
| 11.4.1. Instalacja kubectl w Red Hat Enterprise Linux | 144 |
| 11.4.2. Instalacja kubectl w Debian | 145 |
| 11.4.3. Weryfikacja kubectl | 145 |
| 11.5. Minikube | 145 |
| 11.5.1. Instalacja w dystrybucji Red Hat Enterprise Linux | 146 |
| 11.5.2. Instalacja w dystrybucji Debian | 146 |
| 11.6. Konfiguracja minikube | 146 |
| 11.7. Podsumowanie | 147 |
| Rozdział 12: Kubernetes - Podstawy | 148 |
| 12.1. Dokumentacja | 148 |
| 12.2. Podstawowe zagadnienia | 149 |
| 12.3. Podstawowe polecenia Kuberntesa | 149 |
| 12.4. Wstęp do podów | 150 |
| 12.5. Restarts kontenerów w podach | 150 |
| 12.6. Możliwość skracania nazw | 152 |
| 12.7. Nody | 153 |
| 12.8. Usuwanie podów | 154 |
| 12.9. Podsumowanie | 154 |
| Rozdział 13: Wstęp do plików yaml | 155 |
| 13.1. Podstawowa konstrukcja pliku | 155 |
| 13.2. Pierwsze uruchomienie poda | 156 |
| 13.3. Polityka restartu typów | 157 |
| 13.4. Wywołanie polecenia w kontenerze | 159 |
| 13.5. Dwa kontenery w jednym podzie | 161 |
| 13.6. Powłoka kontenera | 162 |
| 13.7. Logi | 163 |
| 13.8. Podsumowanie | 163 |
| Rozdział 14: Deployment i wszystko co z nim związane | 164 |
| 14.1. Usunięcie podów z poprzedniego rozdziału | 164 |
| 14.2. Poznajemy typ ReplicaSet | 164 |
| 4.2.1. Usuwanie podów | 167 |
| 14.2.2. Lista ReplicaSet | 168 |
| 14.2.3. Usuwanie ReplicaSet | 168 |
| 14.3. Deployment | 169 |
| 14.3.1. Uruchomienie Deploymentu | 169 |

| | |
|---|-----|
| 14.3.2. Sprawdzenie funkcjonowania replik..... | 170 |
| 14.3.3. Lista z Deploymentami..... | 171 |
| 14.3.4. Aktualizacja wersji obrazu | 171 |
| 14.3.5. Opis do wprowadzonych zmian | 173 |
| 14.3.6. Cofanie wprowadzonych zmian | 174 |
| 14.3.7. Sprawdzanie ustawień deploymentu | 175 |
| 14.3.8. Zmiana ilości replik..... | 177 |
| 14.3.9. Inny sposób wprowadzenia ustawień..... | 178 |
| 14.3.10. Usuwanie Deploymentu..... | 181 |
| 14.4. Podsumowanie..... | 181 |
| Rozdział 15: Wykonywanie pojedynczych zadań..... | 182 |
| 15.1. Typ Job..... | 182 |
| 15.1.1. Ponownie o konstrukcji..... | 183 |
| 15.1.2. Zapisanie pliku z wynikiem | 183 |
| 15.1.3. Udostępnienie katalogu za pomocą wolumina | 184 |
| 15.1.4. Restart Policy..... | 184 |
| 5.1.5. Określenie miejsca udostępnienia | 185 |
| 15.1.6. Uruchomienie Job | 185 |
| 15.1.7. Miejsce zapisania plików..... | 186 |
| 15.1.8. Usuwanie Joba | 187 |
| 15.2. CronJob..... | 189 |
| 15.2.1. Ustawienie harmonogramu | 189 |
| 15.2.2. Automatyczne usunięcie poda po zakończeniu zadania | 190 |
| 15.2.3. Tworzymy kontener | 190 |
| 15.2.4. Czynności jakie mają być wykonane | 190 |
| 15.2.5. Restart Policy..... | 190 |
| 15.2.6. Uruchomienie CronJob'a..... | 191 |
| 15.2.7. Usunięcie CronJob | 192 |
| 15.3. Podsumowanie..... | 192 |
| Rozdział 16: Przestrzenie nazw, zmienne środowiskowe i sekrety | 193 |
| 16.1. Namespaces - przestrzenie nazw | 193 |
| 16.1.1. Co to są namespaces - przestrzenie nazw? | 193 |
| 16.1.2. Wyświetlanie dostępnych przestrzeni nazw..... | 193 |
| 16.1.3. Tworzenie własnej przestrzeni nazw | 194 |
| 16.1.4. Dodajemy namespace do deploymentu - sposób z użyciem polecenia | 195 |
| 16.1.5. Wyświetlanie listy naszej przestrzeni nazw | 196 |

| | |
|--|-----|
| 16.1.6. Usuwanie przestrzeni nazw | 196 |
| 16.1.7. Określamy namespace w pliku | 197 |
| 16.1.8. Wyświetlanie listy z wszystkich dostępnych przestrzeni nazw | 198 |
| 16.2. Zmienne środowiskowe | 198 |
| 16.3. Typ Secret..... | 200 |
| 16.4. Podsumowanie..... | 203 |
| To jeszcze nie koniec..... | 204 |
| Szkolenia i kursy | 204 |

Rozdział 1: Podstawowe informacje i wyjaśnienia

1.1. Co to jest Docker?

Docker jest platformą przeznaczoną dla **developerów** oraz **administratorów systemów**. Jest oprogramowaniem **Open Source** dostępnym na większość używanych platform. Możesz ją wykorzystywać zarówno w zakresie prywatnym, jak i komercyjnym bez ponoszenia żadnych dodatkowych kosztów, a także zainstalować niezależnie od posiadanej systemu operacyjnego. W obecnych czasach stał się narzędziem bardzo często wykorzystywany we wszelkich dziedzinach działalności w Internecie, dlatego jego poznanie stało się jednym z ważnych zagadnień dla **administratorów**.

Docker jest rozpoznawany jako narzędzie, dzięki któremu możemy **uruchamiać aplikacje w odizolowanym środowisku** takim jak **kontener**. Dzięki swojej prostocie jest **łatwy do opanowania**, dlatego korzystanie z niego **nie wymaga dużych umiejętności z zakresu IT**.

Jako **administrator** zdarzy Ci się, że będziesz musiał uruchomić aplikację z różnymi konfiguracjami w celu jej przetestowania. Możesz to zrobić bez użycia **Dockera** i na przykład skorzystać z **maszyn wirtualnych** lub też w bardziej *naturalny* sposób, stworzyć kilka folderów w których uruchomisz aplikację z różnymi konfiguracjami.

Niesie to za sobą przymus poświęcenia bardzo dużej ilości czasu niż w przypadku uruchomienia przy użyciu **Dockera**. Tworząc **gotowy obraz** z zainstalowaną aplikacją możesz uruchomić kilka kontenerów z różnymi konfiguracjami, a następnie usunąć zbędne przy pomocy jednego polecenia nie zostawiając żadnych plików po sobie. Co ważniejsze, takie **kontenery** zawierają będą jedynie **oprogramowanie niezbędne do uruchomienia aplikacji**, co oszczędza zarówno i miejsce, jak i czas poświęcony na ich usunięcie. Jest to zupełnie inne podejście niż w przypadku wspomnianych **maszyn wirtualnych**, które w większości sytuacji posiadają sporo zbędnego oprogramowania.

Podsumowując to, co do tej pory przeczytałeś...

Dockera wykorzystać możesz teoretycznie do wszystkiego - od postawienia własnego serwera mailowego po uruchomienie własnej aplikacji, czy też skryptu. Pozwala on zachować wszelkie środki bezpieczeństwa ponieważ **kontener, który uruchamiamy nie ma wpływu na nasz system główny**. Stanowi to również dodatkową zaporę bezpieczeństwa.

1.2. Zapoznanie z pomocą polecenia

Choć nie zainstalowaliśmy jeszcze Dockera, to w tym miejscu chciałbym opowiedzieć trochę o konstrukcji polecenia na bazie pomocy dostępnej wewnątrz programu. Jeżeli jesteś doświadczonym

użytkownikiem systemu **Linux** to na pewno wiesz, że pierwsze co należy zrobić, aby poznać nowe polecenie to zapoznać się z jego dokumentacją. Dlatego zróbmy to w tym miejscu:

```
docker --help
```

Wynik pomocy możemy podzielić na kilka elementów.

Pierwszą informacją jaką otrzymamy to w jaki sposób należy korzystać z Dockera:

Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Dzięki powyższemu przykładowi dowiadujemy się, że po nazwie programu czyli **Docker**, możemy wpisać **opcję** i na samym końcu **polecenie zarządzające/wykonujące**.

Wspomniane opcje powinny wyświetlić się jako następne, a są to:

Options:

```
--config string Location of client config files (default  
"/home/darki/.docker")  
-c, --context string Name of the context to use to connect to the  
daemon (overrides DOCKER_HOST env var and  
default context set with "docker context use")  
-D, --debug Enable debug mode  
-H, --host list Daemon socket(s) to connect to  
-l, --log-level string Set the logging level  
("debug"|"info"|"warn"|"error"|"fatal")  
(default "info")  
--tls Use TLS; implied by --tlsverify  
--tlscacert string Trust certs signed only by this CA (default  
"/home/darki/.docker/ca.pem")  
--tlscert string Path to TLS certificate file (default  
"/home/darki/.docker/cert.pem")  
--tlskey string Path to TLS key file (default  
"/home/darki/.docker/key.pem")  
--tlsverify Use TLS and verify the remote  
-v, --version Print version information and quit
```

Przyznam, że z wymienionych w przykładzie, głównie korzystam z ostatniej czyli z opcji dzięki której poznamy wersję **Dockera**.

Następnie są to polecenia do **konfiguracyjne**:

Management Commands:

app* Docker App (Docker Inc., v0.9.1-beta3)

builder Manage builds

buildx* Docker Buildx (Docker Inc., v0.9.1-docker)

compose* Docker Compose (Docker Inc., v2.10.2)

config Manage Docker configs

container Manage containers

context Manage contexts

image Manage images

manifest Manage Docker image manifests and manifest lists

network Manage networks

node Manage Swarm nodes

plugin Manage plugins

scan* Docker Scan (Docker Inc., v0.17.0)

secret Manage Docker secrets

service Manage services

stack Manage Docker stacks

swarm Manage Swarm

system Manage Docker

trust Manage trust on Docker images

volume Manage volumes

Kolejno **polecenia funkcjonalne**, czyli takie uruchamiające, a nie dokonujące konfiguracji jak w poprzednim przypadku:

Commands:

attach Attach local standard input, output, and error streams to a running container

build Build an image from a Dockerfile

commit Create a new image from a container's changes

cp Copy files/folders between a container and the local filesystem

create Create a new container

diff Inspect changes to files or directories on a container's filesystem
events Get real time events from the server
exec Run a command in a running container
export Export a container's filesystem as a tar archive
history Show the history of an image
images List images
import Import the contents from a tarball to create a filesystem image
info Display system-wide information
inspect Return low-level information on Docker objects
kill Kill one or more running containers
load Load an image from a tar archive or STDIN
login Log in to a Docker registry
logout Log out from a Docker registry
logs Fetch the logs of a container
pause Pause all processes within one or more containers
port List port mappings or a specific mapping for the container
ps List containers
pull Pull an image or a repository from a registry
push Push an image or a repository to a registry
rename Rename a container
restart Restart one or more containers
rm Remove one or more containers
rmi Remove one or more images
run Run a command in a new container
save Save one or more images to a tar archive (streamed to STDOUT by default)
search Search the Docker Hub for images
start Start one or more stopped containers
stats Display a live stream of container(s) resource usage statistics
stop Stop one or more running containers
tag Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
top Display the running processes of a container
unpause Unpause all processes within one or more containers
update Update configuration of one or more containers
version Show the Docker version information

wait Block until one or more containers stop, then print their exit codes

Na samym końcu wypisana jest informacja w jaki sposób można uzyskać pomoc w przypadku poszczególnych poleceń oraz adres strony, gdzie znajduje się oficjalna dokumentacja z poradnikami.

Run 'docker COMMAND --help' for more information on a command.

To get more help with docker, check out our guides at <https://docs.docker.com/go/guides/>

Nie jest to mocno szczegółowy opis. Chciałbym abyś zapamiętał, że polecenia **Dockera** możemy podzielić na dwie grupy:

- przeznaczone do tworzenia, budowania, skanowania;
- przeznaczone do operowania na obrazach, kontenerach i sieciach;

1.3. Co to jest obraz?

Pierwszym bardzo istotnym elementem **Dockera** z jakim musisz się zapoznać jest to **obraz**. To właśnie z niego powstają **kontenery**, ale tak naprawdę czym jest ten obraz?

Otóż najprościej można to wyjaśnić na przykładzie systemu operacyjnego.

Obraz jest to system operacyjny, jak na przykład Debian czy Ubuntu z zainstalowanym oprogramowaniem, które umożliwia mu uruchomienie i nic poza tym. Nie zawiera żadnego programu, który nie jest wymagany, by się uruchomił. W związku z tym to, co nazywamy obrazem zajmuje bardzo mało miejsca. Oczywiście istnieją odstępstwa od tego, ale tylko w sporadycznych przypadkach.

Istnieje również coś, co możemy nazwać podziałem obrazów dostępnych w Dockerze. Posiadamy **obrazy oficjalne**, o które dbają pracownicy samego Dockera oraz **obrazy użytkowników**, o które dba lub dbają jego twórca lub twórcy.

W związku z tym, że możemy tworzyć własne obrazy otwiera się kolejna możliwość. Nie dość, że korzystamy z **Dockera** nie ponosząc żadnych kosztów związanych z jego używaniem zarówno w sferze prywatnej, jak i komercyjnej, to jeszcze możemy tworzyć własne obrazy i udostępniać je innym. **Docker** umożliwia publikację na stronie <https://hub.docker.com/>, oczywiście po wcześniejszej rejestracji.

Z tego materiału chciałbym, żebyś zapamiętał, że **obraz w Dockerze** to nic innego jak **system operacyjny zawierający oprogramowanie niezbędne do jego uruchomienia**.

1.4. Co to jest kontener?

Jak już wspominałem z obrazu powstaje kontener. W trakcie jego tworzenia jest automatycznie dodawany do sieci utworzonej przez Dockera. W związku z tym od samego początku jest odizolowany od głównego systemu. Z jednego obrazu może powstać niezliczona liczba kontenerów, dlatego tego typu rozwiązanie zaczęło dominować od czasów, gdy je wprowadzono.

Taki kontener posiada wszystko co zawiera obraz. Jeżeli wprowadzisz w nim jakieś zmiany, jak na przykład doinstalujesz program, nie będzie to mało żadnego wpływu na sam obraz, pozostałe on nienaruszony. Pamiętaj jednak, że może istnieć tylko jeden kontener o takiej samej nazwie. Natomiast jeżeli o tym zapomnisz to nie martw się, Docker Ci przypomni.

1.5. Dockerfile i compose.yaml

Możesz teraz się zastanawiać w jaki sposób wykorzystanie Dockera może być tak praktyczne? Przecież pobieram obraz z tak zwanym *gołym* systemem, gdzie ta moja aplikacja którą chcę testować?

Jeżeli doszedłeś do takiego wniosku, to bardzo się z tego powodu cieszę. Otóż tak, nie było tam wzmianki w jaki sposób taki obraz powstaje. Teraz chciałbym wspomnieć o istnieniu czegoś takiego jak **Dockerfile**. Jest to plik, w którym umieszczamy polecenia, dzięki którym instalujemy interesujące nas oprogramowanie i w ten sposób powstaje obraz z zainstalowanym potrzebnym nam oprogramowaniem lub zainstalowaną aplikacją.

Drugim istotnym elementem jest docker compose. Za pomocą plików w formacie YAML, możesz połączyć z sobą kilka kontenerów zawierających niezbędne dla swojej funkcjonalności oprogramowanie.

Bardzo dobrym przykładem jest na przykład **Wordpress**, gdzie potrzebujemy serwer z **PHP** oraz **bazę danych**. To dzięki odpowiednio skonfigurowanemu plikowi tego typu będziemy w stanie sprawić, aby oba kontenery powstały i zaczęły z sobą współpracę.

1.6. Jak wyglądają sieci w Dockerze

Sieci w Dockerze są potocznie mówiąc *wisienką na torcie*. Musisz mi wybaczyć to stwierdzenie, ale po prostu tym są. Dzięki sieciom występuje izolacja od samego powstania.

Możesz i powinieneś tworzyć własne sieci. To podniesie bezpieczeństwo i poprawi funkcjonalność. Tworząc własne sieci możesz przydzielić do nich dostęp tylko konkretnym kontenerom.

Przykładem tego jest sytuacja, gdy chcemy stworzyć dość rozbudowany system, gdzie nie wszystkie kontenery powinny mieć do siebie dostęp, wtedy bardzo istotne jest odpowiednie przydzielenie sieci.

Natomiast innym bardzo wygodnym elementem jest **sposób komunikacji kontenerów** ze sobą w sieci stworzonej przez nas. W tym wypadku nie musimy korzystać z adresacji IP tylko z nazwy kontenera, co na pewno przy odpowiednim nazewnictwie ułatwi konfigurację.

1.7. Podsumowanie

Celem powyższego rozdziału jest, abyś zapoznał się z podstawowymi zagadnieniami związanymi z Dockerem. Nie przedstawiłem wszystkiego. Ten rozdział potraktuj jako wstęp, teorię, która wprowadzi Cię do świata kontenerów i Dockera. Natomiast w dalszych omówimy bardziej praktycznej Dockera gdzie wiedza z tego rozdziału na pewno ułatwi Ci zrozumienie wielu aspektów, które niestety mogą zostać pominięte.

Niemniej jednak pierwsze co musimy zrobić to zainstalować Dockera, czym zajmiemy się w następnym rozdziale.

Rozdział 2: Instalacja Dockera w systemach Linux

Jak sam tytuł sugeruje zajmiemy się w nim instalacją Dockera w systemach Linux. Postaram się opisać jak najwięcej możliwości tak, abyś niezależnie od dystrybucji z jakiej korzystasz mógł zainstalować Dockera.

Oprogramowanie Docker możemy zainstalować ze środowiskiem graficznym lub bez. Przyznam, że osobiste nie korzystam z **Docker Desktop**, czyli wersji **GUI** ponieważ jako zapalony użytkownik Linuksa zawsze posługuję się konsolą. Wersja z GUI nie posiada tylu możliwości co konsolowa, dlatego w niektórych momentach będziesz i tak zmuszony do korzystania z wiersza poleceń. Drugą sprawą jest to, że wersja pulpitowa nadal jest w fazie testów stąd też mogą pojawić się z nią problemy.

2.1. Dystrybucja Debian

Nim przystąpimy do instalacji pierwszym krokiem jaki należy wykonać jest **odinstalowanie starszych wersji**, o ile takie są zainstalowane. Robimy to przy pomocy polecenia:

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

Otrzymany wynik zależy od tego czy mieliśmy zainstalowanego Dockera czy też nie.

2.1.1. Instalacja z wykorzystaniem repozytorium Debiana

Jest to jedna z najprostszych instalacji jakie można wykonać ponieważ polega jedynie na wpisaniu nazwy jednego pakietu, a reszta zainstaluje się automatycznie. Prezentowany sposób **zainstaluje Dockera w najnowszej wersji dostępnej z danego repozytorium**. Aby wykonać instalację należy wprowadzić w konsoli następujące polecenia:

```
sudo apt update
```

```
sudo apt install docker.io
```

Wszelkie niezbędne paczki zostaną pobrane automatycznie, my musimy jedynie potwierdzić, że tego chcemy.

Czasami jednak potrzebujemy jak najbardziej aktualną wersję lub też w repozytorium danej dystrybucji nie znajdują się paczki z Dockerem. Wtedy należy skorzystać z opcji prezentowanych dalej.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji, możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązania.

2.1.2. Instalacja z wykorzystaniem repozytorium Dockera

W przypadku instalacji z zewnętrznego źródła musimy poczynić następujące kroki:

```
sudo apt update
```

```
sudo apt install ca-certificates curl gnupg lsb-release
```

Po zainstalowaniu niezbędnego oprogramowania dodajemy **klucz GPG dockera**:

```
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o  
/usr/share/keyrings/docker-archive-keyring.gpg
```

Oczywiście ściągnięcie klucza nie wymaga uprawnień administracyjnych, dlatego na samym początku nie korzystamy z **sudo**. Dopiero w drugim członie polecenia korzystamy z uprawnień administracyjnych gdyż musimy dodać pobrany klucz.

Jeżeli nie otrzymaliśmy, żadnej odpowiedzi w konsoli oznacza to, że wszystko zostało wykonane poprawnie i klucz został dodany. Możemy to sprawdzić przy pomocy prostego polecenia:

```
ls /usr/share/keyrings/docker-archive-keyring.gpg
```

Jako odpowiedź powinniśmy otrzymać

```
/usr/share/keyrings/docker-archive-keyring.gpg
```

Powyższy wynik oznacza, że plik został dodany i możemy przejść do następnego kroku.

Dodajemy nasze repozytorium:

```
echo \
```

```
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]  
https://download.docker.com/linux/debian \  
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Docker posiada różne wersje oprogramowania. Jedną z najczęściej instalowanych jest wersja **stable**. W naszym przypadku z takiej będziemy korzystać. Natomiast jeżeli chciałbyś zainstalować inną wersję, masz dwa wyjścia **test** lub **nightly**. W celu instalacji jednej z dwóch wymienionych wystarczy zamienić słowo **stable** w powyższym poleceniu na nazwę wersji którą wybrałeś. Dla przykładu aby zainstalować wersję **test** wystarczy, że użyjemy polecenia:

```
echo \
```

```
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]  
https://download.docker.com/linux/debian \  
$(lsb_release -cs) test" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
$(lsb_release -cs) test" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Po dodaniu klucza oraz wybraniu interesującej nas wersji i dodaniu repozytorium do naszego systemu przyszła pora na instalację. Wykonujemy ją w następujący sposób:

```
sudo apt update
```

```
sudo apt install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

Jeżeli postępowaliśmy zgodnie ze wcześniejszymi instrukcjami instalacja powinna odbyć się bez żadnego błędu.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji, możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.1.3. Instalacja określonej wersji Dockera

Może się zdarzyć, że będziesz potrzebował zainstalować konkretną wersję Dockera. Aby mieć jak największą ilość opcji do wyboru, najlepiej jest **dodać repozytorium Dockera do naszego systemu**. Czyli przed przystąpieniem do poniższych działań, jeżeli jeszcze tego nie zrobłeś, dodaj repozytorium tak jak robiłem to w podrozdziale 2.1.2., ale nie przystępuj do instalacji. Aby wyświetlić możliwe do zainstalowania wersje należy skorzystać z polecenia:

```
apt-cache madison docker-ce
```

```
docker-ce | 5:20.10.15~3-0~debian-bullseye | https://download.docker.com/linux/debian bullseye/stable  
amd64 Packages
```

```
docker-ce | 5:20.10.14~3-0~debian-bullseye | https://download.docker.com/linux/debian bullseye/stable  
amd64 Packages
```

```
docker-ce | 5:20.10.13~3-0~debian-bullseye | https://download.docker.com/linux/debian bullseye/stable  
amd64 Packages
```

```
docker-ce | 5:20.10.12~3-0~debian-bullseye | https://download.docker.com/linux/debian bullseye/stable  
amd64 Packages
```

```
docker-ce | 5:20.10.11~3-0~debian-bullseye | https://download.docker.com/linux/debian bullseye/stable  
amd64 Packages
```

```
docker-ce | 5:20.10.10~3-0~debian-bullseye | https://download.docker.com/linux/debian bullseye/stable  
amd64 Packages
```

docker-ce | 5:20.10.9~3-0~debian-bullseye | <https://download.docker.com/linux/debian> bullseye/stable
amd64 Packages

docker-ce | 5:20.10.8~3-0~debian-bullseye | <https://download.docker.com/linux/debian> bullseye/stable
amd64 Packages

docker-ce | 5:20.10.7~3-0~debian-bullseye | <https://download.docker.com/linux/debian> bullseye/stable
amd64 Packages

docker-ce | 5:20.10.6~3-0~debian-bullseye | <https://download.docker.com/linux/debian> bullseye/stable
amd64 Packages

Mamy do wyboru kilka opcji co potwierdza powyższa lista. Jeżeli decydujemy się na instalację którejś z listy wystarczy, że zmodyfikujemy poznane wcześniej polecenie:

```
sudo apt install docker-ce=<nazwa-wersji> docker-ce-cli=<nazwa-wersji> containerd.io docker-compose-plugin
```

W miejscu *nazwa wersji* wpisujemy interesującą nas komplikację, na przykład, aby zainstalować wersję **5:20.10.9~3-0~debian-bullseye** wpiszemy polecenie w następujący sposób:

```
sudo apt-get install docker-ce=5:20.10.9~3-0~debian-bullseye docker-ce-cli=5:20.10.9~3-0~debian-bullseye containerd.io docker-compose-plugin
```

W celu sprawdzenia czy wszystko działa wpisujemy:

```
docker -v
```

I sprawdzamy czy zgadza się z wersją którą chcieliśmy zainstalować.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.2. Dystrybucja Ubuntu

Nim przystąpimy do instalacji pierwszym krokiem jaki należy wykonać jest **odinstalowanie starszych wersji**, o ile takie są zainstalowane. Robimy to przy pomocy polecenia:

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

Otrzymany wynik zależy od tego czy mieliśmy zainstalowanego Dockera czy też nie.

2.2.1. Instalacja z wykorzystaniem repozytorium Ubuntu

Jest to jedna z **najprostszych instalacji** jakie można wykonać, ponieważ polega jedynie na wpisaniu nazwy jednego pakietu, a reszta zainstaluje się automatycznie. Prezentowany sposób **zainstaluje**

Dockera w najnowszej wersji dostępnej z danego repozytorium. Aby wykonać instalację należy wprowadzić w konsoli następujące polecenia:

```
sudo apt update
```

```
sudo apt install docker.io
```

Wszelkie niezbędne paczki zostaną pobrane automatycznie, my musimy jedynie potwierdzić, że tego chcemy. Czasami jednak potrzebujemy jak najbardziej aktualną wersję lub też w repozytorium danej dystrybucji nie znajdują się paczki z Dockerem. Wtedy należy skorzystać z opcji prezentowanych poniżej.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.2.2. Instalacja z wykorzystaniem repozytorium Dockera

W przypadku instalacji z zewnętrznego źródła musimy poczynić następujące kroki:

```
sudo apt update
```

```
sudo apt install ca-certificates curl gnupg lsb-release
```

Po zainstalowaniu niezbędnego oprogramowania dodajemy **klucz GPG dockera**:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o  
/usr/share/keyrings/docker-archive-keyring.gpg
```

Oczywiście ściągnięcie klucza nie wymaga uprawnień administracyjnych, dlatego na samym początku nie korzystamy z **sudo**. Dopiero w drugim członie polecenia korzystamy z uprawnień administracyjnych, gdyż musimy dodać pobrany klucz.

Jeżeli nie otrzymaliśmy, żadnej odpowiedzi w konsoli oznacza to, że wszystko zostało wykonane poprawnie i klucz został dodany. Możemy to sprawdzić przy pomocy prostego polecenia:

```
ls /usr/share/keyrings/docker-archive-keyring.gpg
```

Jako odpowiedź powinniśmy otrzymać

```
/usr/share/keyrings/docker-archive-keyring.gpg
```

Powyższy wynik oznacza, że plik został dodany i możemy przejść do następnego kroku.

Dodajemy nasze repozytorium:

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/debian \
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Docker posiada różne wersje oprogramowania. Jedną z najczęściej instalowanych jest wersja **stable**. W naszym przypadku z takiej będziemy korzystać. Natomiast jeżeli chciałbyś zainstalować inną wersję, masz dwa wyjścia **test** lub **nightly**. W celu instalacji jednej z dwóch wymienionych wystarczy zamienić słowo **stable** w powyższym poleceniu na nazwę wersji którą wybrałeś. Dla przykładu aby zainstalować wersję **test** wystarczy, że użyjemy polecenia:

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/debian \
$(lsb_release -cs) test" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Po dodaniu klucza oraz wybraniu interesującej nas wersji i dodaniu repozytorium do naszego systemu przyszła pora na instalację. Wykonujemy ją w następujący sposób:

sudo apt update

sudo apt install docker-ce docker-ce-cli containerd.io docker-compose-plugin

Jeżeli postępowaliśmy zgodnie z wcześniejszymi instrukcjami instalacja powinna odbyć się bez żadnego błędu.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązania.

2.2.3. Instalacja określonej wersji Dockera

Może się zdarzyć, że będziesz potrzebował zainstalować konkretną wersję Dockera. Aby mieć jak największą ilość opcji do wyboru najlepiej jest **dodać repozytorium Dockera do naszego systemu**. Czyli przed przystąpieniem do poniższych działań, jeżeli jeszcze tego nie zrobiłeś, dodaj repozytorium tak jak robiłem to w podrozdziale 2.2.2., ale nie przystępuj do instalacji.

Aby wyświetlić możliwe do zainstalowania wersje należy skorzystać z polecenia:

apt-cache madison docker-ce

```
docker-ce | 5:20.10.15~3-0~ubuntu-jammy | https://download.docker.com/linux/ubuntu jammy/stable  
amd64 Packages
```

```
docker-ce | 5:20.10.14~3-0~ubuntu-jammy | https://download.docker.com/linux/ubuntu jammy/stable  
amd64 Packages
```

```
docker-ce | 5:20.10.13~3-0~ubuntu-jammy | https://download.docker.com/linux/ubuntu jammy/stable  
amd64 Packages
```

Mamy do wyboru kilka opcji ,co potwierdza powyższa lista. Jeżeli decydujemy się na instalację którejś z listy wystarczy, że zmodyfikujemy poznane wcześniej polecenie:

```
sudo apt install docker-ce=<nazwa-wersji> docker-ce-cli=<nazwa-wersji> containerd.io docker-compose-plugin
```

W miejscu *nazwa wersji* wpisujemy interesującą nas komplikację, na przykład, aby zainstalować wersję **5:20.10.14~3-0~ubuntu-jammy** wpiszemy polecenie w następujący sposób:

```
sudo apt-get install docker-ce=5:20.10.14~3-0~ubuntu-jammy docker-ce-cli=5:20.10.14~3-0~ubuntu-jammy  
containerd.io docker-compose-plugin
```

W celu sprawdzenia czy wszystko działa wpisujemy:

```
docker -v
```

I sprawdzamy czy zgadza się z wersja którą chcieliśmy zainstalować.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązania.

2.3. Dystrybucja Fedora

Nim przystąpimy do instalacji pierwszym krokiem jaki należy wykonać jest **odinstalowanie starszych wersji**, o ile takie są zainstalowane. Robimy to przy pomocy polecenia:

```
sudo dnf remove docker docker-client docker-client-latest docker-common docker-latest docker-latest-logrotate docker-logrotate docker-selinux docker-engine-selinux docker-engine
```

Otrzymany wynik zależy od tego czy mieliśmy zainstalowanego Dockera czy też nie.

2.3.1. Instalacja z wykorzystaniem repozytorium Fedory

Jest to jedna z najprostszych instalacji jakie można wykonać ponieważ polega jedynie na wpisaniu nazwy jednego pakietu, a reszta zainstaluje się automatycznie. Prezentowany sposób **zainstaluje**

dockera w najnowszej wersji dostępnej z danego repozytorium. Aby wykonać instalację należy wprowadzić w konsoli następujące polecenia:

```
sudo dnf install docker
```

Wszelkie niezbędne paczki zostaną pobrane automatycznie my musimy jedynie potwierdzić, że tego chcemy. Aplikacja Docker automatycznie nie uruchamia się dlatego nie możemy z niej korzystać. Aby to zmienić należy wpisać polecenie:

```
sudo systemctl start docker
```

Czasami jednak potrzebujemy jak najbardziej aktualną wersję lub też w repozytorium danej dystrybucji nie znajdują się paczki z Dockerem. Wtedy należy skorzystać z opcji prezentowanych poniżej.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.3.2. Instalacja z wykorzystaniem repozytorium Dockera

W przypadku instalacji z zewnętrznego źródła musimy poczynić następujące kroki:

```
sudo dnf -y install dnf-plugins-core
```

W ten sposób instalujemy niezbędne oprogramowanie, aby móc dodawać do repozytorium Fedory. Następnie pozostało je tylko dodać. Robimy to tak jak w poniższym przykładzie:

```
sudo dnf config-manager \
```

```
--add-repo \
```

```
https://download.docker.com/linux/fedora/docker-ce.repo
```

Docker posiada różne wersje oprogramowania. Jedną z najczęściej instalowanych jest wersja **stable**. W naszym przypadku z takiej będziemy korzystać i takie repozytorium jest ustawione przy użyciu powyższego polecenia. Natomiast jeżeli chciałbyś zainstalować inną wersję, masz dwa wyjścia **test** lub **nightly**. Aby zainstalować jedną z wymienionych wersji należy użyć odpowiedniego polecenia do zamiany:

```
sudo dnf config-manager --set-enabled docker-ce-nightly
```

```
sudo dnf config-manager --set-enabled docker-ce-test
```

Wybrane polecenie umożliwia dostęp do odpowiedniego repozytorium. Gdybyśmy chcieli zrezygnować z korzystania z któregokolwiek, wystarczy wpisać polecenie:

```
sudo dnf config-manager --set-disabled docker-ce-test
```

Po wybraniu interesującej nas wersji i dodania jej repozytorium, nadeszła pora na instalację:

```
sudo dnf install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

Jeżeli postępowaliśmy zgodnie z wcześniejszymi instrukcjami instalacja powinna odbyć się bez żadnego błędu.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegała pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. tego i poszukaj rozwiązań.

2.3.3. Instalacja określonej wersji Dockera

Może się zdarzyć, że będziesz potrzebował zainstalować konkretną wersję Dockera. Aby mieć jak największą ilość opcji do wyboru najlepiej jest **dodać repozytorium Dockera do naszego systemu**. Czyli przed przystąpieniem do poniższych działań, jeżeli jeszcze tego nie zrobiłeś, dodaj repozytorium tak jak robiłem to w podrozdziale 2.3.2., ale nie przystępuj do instalacji.

Aby wyświetlić możliwe do zainstalowania wersje Dockera należy skorzystać z polecenia:

```
docker-ce --showduplicates | sort -r
```

```
docker-ce.x86_64 3:20.10.15-3.fc35 @docker-ce-stable
```

```
docker-ce.x86_64 3:20.10.14-3.fc35 docker-ce-stable
```

```
docker-ce.x86_64 3:20.10.13-3.fc35 docker-ce-stable
```

```
docker-ce.x86_64 3:20.10.12-3.fc35 docker-ce-stable
```

```
docker-ce.x86_64 3:20.10.11-3.fc35 docker-ce-stable
```

```
docker-ce.x86_64 3:20.10.10-3.fc35 docker-ce-stable
```

Mamy do wyboru kilka opcji co potwierdza powyższa lista. Jeżeli decydujemy się na instalację którejś z listy wystarczy, że zmodyfikujemy poznane wcześniej polecenie:

```
sudo dnf -y install docker-ce=<nazwa-wersji> docker-ce-cli=<nazwa-wersji> containerd.io docker-compose-plugin
```

W miejscu *nazwa wersji* wpisujemy interesującą nas komplikację, na przykład aby zainstalować wersję **3:20.10.13-3.fc35** wpiszemy polecenie w następujący sposób:

```
sudo dnf -y install docker-ce=20.10.13 docker-ce-cli=20.10.13 containerd.io docker-compose-plugin
```

W celu sprawdzenia czy wszystko działa wpisujemy:

```
docker -v
```

I sprawdzamy czy zgadza się wersja którą chcieliśmy zainstalować.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.4. Dystrybucja CentOS

Nim przystąpimy do instalacji pierwszym krokiem jaki należy wykonać jest **odinstalowanie starszych wersji**, o ile takie są zainstalowane. Robimy to przy pomocy polecenia:

```
sudo yum remove docker docker-client docker-client-latest docker-common docker-latest docker-latest-logrotate docker-logrotate docker-selinux docker-engine-selinux docker-engine
```

Otrzymany wynik zależy od tego czy mieliśmy zainstalowanego Dockera czy też nie.

2.4.1. Instalacja z wykorzystaniem repozytorium CentOS

Jest to jedna z najprostszych instalacji jakie można wykonać ponieważ polega jedynie na wpisaniu nazwy jednego pakietu, a reszta zainstaluje się automatycznie. Prezentowany sposób **zainstaluje dockera w najnowszej wersji dostępnej z danego repozytorium**. Aby wykonać instalację należy wprowadzić w konsoli następujące polecenia:

```
sudo yum install docker
```

Wszelkie niezbędne paczki zostaną pobrane automatycznie, my musimy jedynie potwierdzić, że tego chcemy. Dystrybucja **CentOS** jest oparta o **Red Hat Enterprise Linux**, który posiada zamiennik Dockera zwany **Podman**. W powyższy sposób zainstalujemy oprogramowanie **Podman**. Aplikacja ta jest zgodna ze specyfikacją Dockera i powinna działać identycznie jak sam Docker. Instalację polecam przeprowadzić jednak w powyższy sposób ze względu na to, że zostanie automatycznie doinstalowana paczka **podman-docker** która umożliwi Ci korzystanie z aplikacji w taki sam sposób jak ze standardowego dockera. Czyli wpisywać możesz w konsoli **Docker** zamiast **Podman** co myślę, że bardzo ułatwi korzystanie z aplikacji w tym kursie.

W przypadku jeżeli chcesz jednak korzystać tylko i wyłącznie z Podmana wystarczy, przeprowadzić instalację w poniższy sposób:

```
sudo yum install podman
```

Czasami jednak potrzebujemy jak najbardziej aktualną wersję lub też w repozytorium danej dystrybucji nie znajdują się paczki z Dockerem. Wtedy należy skorzystać z opcji prezentowanych poniżej.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.4.2. Instalacja z wykorzystaniem repozytorium dockera

W przypadku instalacji z zewnętrznego źródła musimy poczynić następujące kroki:

```
sudo yum -y install yum-utils
```

W ten sposób instalujemy niezbędne oprogramowanie, aby móc dodawać do repozytorium CentOS. Następnie pozostało je tylko dodać. Robimy to tak jak w poniższym przykładzie:

```
sudo yum config-manager \
```

```
--add-repo \
```

```
https://download.docker.com/linux/centos/docker-ce.repo
```

Docker posiada różne wersje oprogramowania. Jedną z najczęściej instalowanych jest wersja **stable**. W naszym przypadku z takiej będziemy korzystać i takie repozytorium jest ustawione przy użyciu powyższego polecenia. Natomiast jeżeli chciałbyś zainstalować inną wersję, masz dwa wyjścia **test** lub **nightly**. Aby zainstalować jedną z wymienionych wersji należy użyć odpowiedniego polecenia do zamiany:

```
sudo yum-config-manager --set-enabled docker-ce-nightly
```

```
sudo yum-config-manager --set-enabled docker-ce-test
```

Wybrane polecenie umożliwia dostęp do odpowiedniego repozytorium. Gdybyśmy chcieli zrezygnować z korzystania z któregokolwiek, wystarczy wpisać polecenie:

```
sudo yum-config-manager --set-disabled docker-ce-test
```

Po wybraniu interesującej nas wersji i dodaniu jej repozytorium, nadeszła pora na instalację:

```
sudo yum install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

Jeżeli postępowaliśmy zgodnie z wcześniejszymi instrukcjami instalacja powinna odbyć się bez żadnego błędu.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.4.3. Instalacja określonej wersji Dockera

Może się zdarzyć, że będziesz potrzebował zainstalować konkretną wersję Dockera. Aby mieć jak największą ilość opcji do wyboru najlepiej jest **dodać repozytorium Dockera do naszego systemu**. Czyli przed przystąpieniem do poniższych działań, jeżeli jeszcze tego nie zrobiłeś, dodaj repozytorium tak jak robiłem to w podrozdziale 2.4.2., ale nie przystępuj do instalacji.

Aby wyświetlić możliwe do zainstalowania wersje Dockera należy skorzystać z polecenia:

```
yum list docker-ce --showduplicates | sort -r
```

Installed Packages

```
docker-ce.x86_64 3:20.10.15-3.el9 docker-ce-stable
```

```
docker-ce.x86_64 3:20.10.15-3.el9 @docker-ce-stable
```

W przypadku **CentOS** wybór na dzień dzisiejszy jest tylko jeden, dlatego instalacja wersji z przykładu jest równoznaczna z instalacją którą poznaleś we wcześniejszym podrozdziale. Niemniej jednak może się to za jakiś czas zmienić, lub też korzystasz z innej wersji **CenOS** niż ja. Jeżeli decydujesz się na instalację innej wersji niż najnowsza wystarczy, że zmodyfikujesz poznane wcześniej polecenie:

```
sudo yum install docker-ce-<VERSION_STRING> docker-ce-cli-<VERSION_STRING> containerd.io docker-compose-plugin
```

W miejscu *nazwa wersji* wpisujemy interesującą nas komplikację, na przykład aby zainstalować wersję **3:20.10.15-3.el9** wpiszemy polecenie w następujący sposób:

```
sudo yum install docker-ce=20.10.15 docker-ce-cli=20.10.15 containerd.io docker-compose-plugin
```

W celu sprawdzenia czy wszystko działa wpisujemy:

```
docker -v
```

I sprawdzamy czy zgadza się wersja którą chcieliśmy zainstalować.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.5. Instalacja z paczki

Jednym ze sposobów instalacji jest pobranie odpowiedniej paczki oraz jej zainstalowanie. To rozwiązanie stosujemy kiedy z jakiś powodów nie udało nam się zainstalować Dockera z repozytorium. Wszystkie dostępne paczki dostępne są pod adresem <https://download.docker.com/linux/>.

2.5.1. Instalacja dla systemu Debian oraz Ubuntu

Ubuntu jest systemem opartym o dystrybucję Debian dlatego ich system pakietów jest identyczny. Niemniej jednak dla każdego z dwóch wymienionych dystrybucji zostały przygotowane oddzielne pakiety.

Dla dystrybucji Debian pobieramy paczki: <https://download.docker.com/linux/debian/dists/>

Dla dystrybucji Ubuntu pobieramy paczki: <https://download.docker.com/linux/ubuntu/dists/>

Następnie wybieramy odpowiednią **wersję systemu z jakiego korzystamy** i wchodzimy w folder **pool**. Wybieramy **wersję** jaką chcemy zainstalować oraz **architekturę**.

W poniższym przykładzie korzystam z **Ubuntu Jammy Jellyfish**, gdzie chcę zainstalować wersję stabilną dlatego ściągam pliki z adresu:

<https://download.docker.com/linux/ubuntu/dists/jammy/pool/stable/amd64/> .

Do kompletnej instalacji dockera potrzebujemy sześciu plików **docker-ce docker-ce-cli containerd.io docker-compose-plugin docker-ce-rootless-extras docker-scan-plugin**. Ściągać będę najnowsze wersje z gałęzi **stable** o numerze **20.10.15** dlatego pobieram:
https://download.docker.com/linux/ubuntu/dists/jammy/pool/stable/amd64/docker-ce-cli_20.10.15~3-0~ubuntu-jammy_amd64.deb

https://download.docker.com/linux/ubuntu/dists/jammy/pool/stable/amd64/docker-ce_20.10.15~3-0~ubuntu-jammy_amd64.deb

https://download.docker.com/linux/ubuntu/dists/jammy/pool/stable/amd64/docker-compose-plugin_2.5.0~ubuntu-jammy_amd64.deb

https://download.docker.com/linux/ubuntu/dists/jammy/pool/stable/amd64/containerd.io_1.6.4-1_amd64.deb

https://download.docker.com/linux/ubuntu/dists/jammy/pool/stable/amd64/docker-ce-rootless-extras_20.10.15~3-0~ubuntu-jammy_amd64.deb

https://download.docker.com/linux/ubuntu/dists/jammy/pool/stable/amd64/docker-scan-plugin_0.17.0~ubuntu-jammy_amd64.deb

Po pobraniu odpowiednich paczek przechodzimy do folderu, w którym się znajdują i przy pomocy poniższego polecenia instalujemy:

```
sudo dpkg -i docker-ce-cli_20.10.15~3-0~ubuntu-jammy_amd64.deb docker-ce_20.10.15~3-0~ubuntu-jammy_amd64.deb docker-compose-plugin_2.5.0~ubuntu-jammy_amd64.deb containerd.io_1.6.4-1_amd64.deb docker-ce-rootless-extras_20.10.15~3-0~ubuntu-jammy_amd64.deb docker-scan-plugin_0.17.0~ubuntu-jammy_amd64.deb
```

W taki sam sposób postępujesz, jeżeli korzystasz z dystrybucji **Debian**. W jego wypadku zmieniają się głównie **ostatnie człony nazw plików**.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.5.2. Dla systemu Fedora

Dla dystrybucji **Fedora** pobieramy paczki: <https://download.docker.com/linux/fedora/>

Po pierwsze wybieramy odpowiednią **wersję systemu z jakiego korzystamy**. Następnie określamy **architekturę naszego procesora**. Na końcu wybieramy **wersję jaką chcemy zainstalować** i wchodzimy w folder **Packages**.

W poniższym przykładzie korzystam z **Fedory 35** oraz chcę zainstalować wersję **stable** dlatego ściągam pliki z adresu https://download.docker.com/linux/fedora/35/x86_64/stable/Packages/.

Do kompletnej instalacji Dockera potrzebujemy sześciu plików **docker-ce docker-ce-cli containerd.io docker-compose-plugin docker-ce-rootless-extras docker-scan-plugin**. Ściągać będę najnowsze wersje z gałęzi **stable** o numerze **20.10.15** dlatego pobieram:

https://download.docker.com/linux/fedora/35/x86_64/stable/Packages/docker-ce-20.10.15-3.fc35.x86_64.rpm

https://download.docker.com/linux/fedora/35/x86_64/stable/Packages/docker-ce-cli-20.10.15-3.fc35.x86_64.rpm

https://download.docker.com/linux/fedora/35/x86_64/stable/Packages/docker-compose-plugin-2.5.0-3.fc35.x86_64.rpm

https://download.docker.com/linux/fedora/35/x86_64/stable/Packages/containerd.io-1.6.4-3.1.fc35.x86_64.rpm

https://download.docker.com/linux/fedora/35/x86_64/stable/Packages/docker-ce-rootless-extras-20.10.15-3.fc35.x86_64.rpm

https://download.docker.com/linux/fedora/35/x86_64/stable/Packages/docker-scan-plugin-0.17.0-3.fc35.x86_64.rpm

Po pobraniu odpowiednich paczek przechodzimy do folderu, w którym się znajdują i przy pomocy poniższego polecenia instalujemy:

```
sudo dpkg -i docker-ce-20.10.15-3.fc35.x86_64.rpm docker-ce-cli-20.10.15-3.fc35.x86_64.rpm docker-compose-plugin-2.5.0-3.fc35.x86_64.rpm containerd.io-1.6.4-3.1.fc35.x86_64.rpm docker-ce-rootless-extras-20.10.15-3.fc35.x86_64.rpm docker-scan-plugin-0.17.0-3.fc35.x86_64.rpm
```

Instalacja powinna zakończyć się powodzeniem. Oczywiście nazwy plików z biegiem czasu ulegną zmianie dlatego zwróć na to uwagę.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.5.3. Dla systemu CentOS

Dla dystrybucji CentOS pobieramy paczkę: <https://download.docker.com/linux/centos/>

Po pierwsze wybieramy odpowiednią **wersję systemu z jakiego korzystamy**. Następnie określamy **architekturę naszego procesora**. Na końcu wybieramy **wersję jaką chcemy zainstalować** i wchodzimy w folder **Packages**.

W poniższym przykładzie korzystam z **CentOS 9** oraz chcę zainstalować wersję **stable** dlatego ściągam pliki z adresu https://download.docker.com/linux/centos/9/x86_64/stable/Packages/.

Do kompletnej instalacji dockera potrzebujemy sześciu plików **docker-ce docker-ce-cli containerd.io docker-compose-plugin docker-ce-rootless-extras docker-scan-plugin**. Ściągać będę najnowsze wersje z gałęzi **stable** o numerze **20.10.15** dlatego pobieram:

https://download.docker.com/linux/centos/9/x86_64/stable/Packages/docker-ce-20.10.15-3.el9.x86_64.rpm

https://download.docker.com/linux/centos/9/x86_64/stable/Packages/docker-ce-cli-20.10.15-3.el9.x86_64.rpm

https://download.docker.com/linux/centos/9/x86_64/stable/Packages/docker-compose-plugin-2.5.0-3.el9.x86_64.rpm

https://download.docker.com/linux/centos/9/x86_64/stable/Packages/containerd.io-1.6.4-3.1.el9.x86_64.rpm

https://download.docker.com/linux/centos/9/x86_64/stable/Packages/docker-ce-rootless-extras-20.10.15-3.el9.x86_64.rpm

https://download.docker.com/linux/centos/9/x86_64/stable/Packages/docker-scan-plugin-0.17.0-3.el9.x86_64.rpm

Po pobraniu odpowiednich paczek przechodzimy do folderu, w którym się znajdują i przy pomocy poniższego polecenia instalujemy:

```
sudo dpkg -i docker-ce-20.10.15-3.el9.x86_64.rpm docker-ce-cli-20.10.15-3.el9.x86_64.rpm docker-compose-plugin-2.5.0-3.el9.x86_64.rpm containerd.io-1.6.4-3.1.el9.x86_64.rpm docker-ce-rootless-extras-20.10.15-3.el9.x86_64.rpm docker-scan-plugin-0.17.0-3.el9.x86_64.rpm
```

Instalacja powinna zakończyć się powodzeniem. Oczywiście nazwy plików z biegiem czasu ulegną zmianie dlatego zwróć na to uwagę.

Jeżeli zainstalowałeś w powyższy sposób Dockera i nie masz potrzeby zapoznania się z innymi możliwościami instalacji możesz przejść do podrozdziału 2.7., gdzie przetestujesz czy instalacja na pewno przebiegła pomyślnie. Jeżeli przy instalacji napotkałeś na jakieś problemy przejdź do podrozdziału 2.8. i poszukaj rozwiązań.

2.6. Instalacja Dockera przy pomocy plików binarnych

Jednym z ostatecznych rozwiązań jest instalacja Dockera przy pomocy plików binarnych. **Tego typu sposób powinniśmy zastosować tylko i wyłącznie gdy wszystkie pozostałe zawiodły.** Obecnie Docker jest dostępny na większość platform, dlatego instalacja przy pomocy plików binarnych powinna być ostatecznością. Ma to związek z tym, że aplikacja instalowana w ten sposób nie uwzględnia automatycznych aktualizacji co wiąże się z dość sporym zagrożeniem.

Pomimo ryzyka jakie niesie za sobą instalacja przy pomocy plików binarnych czasami się przydaje. W tym materiale przeprowadzę ją w systemie **Mint 20.3 Una**, ale wygląda tak samo na każdym innym systemie z rodziny Linux.

Rozpoczynamy od ściągnięcia pliku binarnego. Odnajdziemy go pod adresem <https://download.docker.com/linux/static/stable/>. Wybieramy architekturę z jakiej korzystamy. Następnie z wyświetlonej listy odnajdujemy najnowszą dostępną wersję. Na dzień dzisiejszy jest to

[20.10.15](#). Po pobraniu pliku przechodzimy do folderu do którego go ściągnęliśmy i rozpakowujemy przy pomocy polecenia:

```
tar xzvf docker-20.10.15.tgz
```

W kolejnym kroku przekopiujemy rozpakowane pliki do folderu **/usr/bin/** tak aby można było korzystać z poleceń bezpośrednio w konsoli:

```
sudo cp docker/* /usr/bin/
```

Na sam koniec **uruchamiamy demoną** odpowiedzialnego za Dockera:

```
sudo dockerd&
```

2.7. Sprawdzenie czy instalacja zakończyła się powodzeniem

Robimy to w bardzo prosty sposób. W wierszu polecień wpisujemy:

```
sudo docker run hello-world
```

Po krótkiej chwili powinniśmy otrzymać informację:

Hello from Docker!

This message shows that your installation appears to be working correctly.

Informacja która pojawiła się oznacza, że docker został poprawnie zainstalowany.

Co dokładnie zrobiliśmy przy użyciu tego polecenia wyjaśnimy sobie w jednym z następnych rozdziałów.

2.8. Błędy z jakimi możesz się spotkać

2.8.1. Błąd podczas uruchomienia

Pierwszym dość często spotykanym błędem jest brak uruchomionej usługi Docker. Dlatego jeżeli pojawi Ci się informacja o treści:

```
docker: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?  
See 'docker run --help'.
```

W tej sytuacji musisz włączyć usługę Docker przy pomocy polecenia:

```
sudo systemctl start docker
```

I problem powinien zostać rozwiązany.

2.8.2. Zduplikowane repozytoria

Kolejnym problemem są **zduplikowane repozytoria**. W tej sytuacji należy wybrać wersję oprogramowania jakie chcemy zainstalować. Pokazywałem jak to robić przy każdym z opisywanych systemów.

2.8.3. Brak uprawnień

Ostatnim błędem z jakim się spotkałem i czasami sam go popełniam, to brak uprawień do wykonywania niektórych czynności w Dockerze. W tym przypadku musimy pamiętać, by korzystać z **sudo**.

2.9. Podsumowanie

Mam nadzieję, że udało Ci się zainstalować Dockera bez jakichkolwiek problemów. Starałem się dość kompleksowo wytłumaczyć kilka możliwości instalacji tego oprogramowania. W przypadku dystrybucji nieopisywanych często wystarczy zerknąć czy na jakiejś dystrybucji ona się nie opiera. W dalszej części kursu przejdziemy już do praktycznych rzeczy. Mam nadzieję, że też nie możecie się już doczekać.

Rozdział 3: Docker - obrazy

Pierwszym elementem z jakim moim zdaniem należy się zapoznać są **obrazy**. Wybrałem je na sam początek, ponieważ są systemem bazowym, elementem, z którego składają się kontenery, dlatego inaczej można je nazwać początkiem wszystkiego w **Dockerze**.

Część teoretyczną przedstawiłem na samym początku. W tym rozdziale zajmiemy się bardziej praktycznym wykorzystaniem **Dockera**. Dlatego ważne jest, abyś miał już go u siebie zainstalowanego. Poniższe informacje oraz przykłady nie będą się różniły, dlatego nieważne jest z jakiego systemu korzystasz. Jedyna różnica może wystąpić, gdy używasz **Podmana**, wtedy należy zastąpić słowo **Docker**, wspomnianym.

3.1. Pobieramy nasz pierwszy obraz

Jednym z najczęściej wybieranych obrazów dla serwerów oraz innych działalności w Internecie jest **CentOS Linux** dlatego też uważam, że będzie on bardzo dobrym przykładem na sam początek:

```
docker pull centos
```

Using default tag: latest

latest: Pulling from library/centos

a1d0c7532777: Pull complete

Digest: sha256:a27fd8080b517143cbbb9dfb7c8571c40d67d534bbdee55bd6c473f432b177

Status: Downloaded newer image for centos:latest

docker.io/library/centos:latest

Jak zostało zapisane w przykładzie *Pull complete* oznacza, że poprawnie pobraliśmy określony obraz. Zwróć uwagę, na prostotę polecenia. Przetłumaczyć możemy je na język polski jako *docker pobierz centosa*, chyba prościej się nie da.

3.2. Wyświetlanie listy obrazów

Jak się zapewne domyślasz nie musisz posiadać tylko jednego obrazu dlatego też ważne jest wiedzieć w jaki sposób wyświetlić listę już pobranych. Do tego posłuży polecenie:

```
docker images
```

REPOSITORY TAG IMAGE ID CREATED SIZE

centos latest 5d0da3dc9764 12 months ago 231MB

W przykładzie wyświetliśmy listę i tak jak w przypadku pobierania obrazów nie zalicza się to do skomplikowanych czynności. Używamy **nazwy programu** i dodajemy słowo **images**.

Natomiast analizując przykład, to idąc po kolejnej kolumnie, w pierwszej znajduje się **nazwa pobranego obrazu**, w drugiej jego **wersja**, tym zajmiemy się w jednym z następnych punktów, następnie **ID obrazu**, w przedostatnim, kiedy został utworzony / aktualizowany oraz ile zajmuje miejsca.

Zapamiętaj jednak, że **data utworzenia nie określa daty, w której pobrałeś obraz tylko datę, kiedy uległ on modyfikacjom lub został przesłany**. W prezentowanym przypadku jest to dość logiczne, bo przecież właśnie pobrałeś obraz, a nie 12 miesięcy temu. Natomiast możesz spotkać się z sytuacją, że obraz będzie z tego samego dnia lub też z dnia poprzedniego. Dlatego zapamiętaj, że **data wyświetlna jest to data utworzenia obrazu, a nie jego pobrania**.

3.3. Skąd pobierane są obrazy

Wiemy już w jaki sposób są pobierane obrazy, ale teraz pytanie brzmi skąd? Otóż Docker posiada swoją platformę zwaną **DockerHub** na której umieszczone są wszystkie dostępne obrazy. Platformę tą powinniśmy traktować jako repozytorium. Zerknijmy teraz na stronę <https://hub.docker.com/>.

Obecnie z lewej strony ekranu znajduje się wyszukiwarka z jakiej możemy skorzystać by odnaleźć obraz, który nas interesuje. W poprzednim paragrafie pobraliśmy obraz systemu **CentOS**, dlatego teraz skorzystajmy z wyszukiwarki i wpiszmy nazwę tego systemu. Jednym z pierwszych wyników będzie https://hub.docker.com/_/centos.

Na stronie obrazu przesuwając się w dół odnajdziesz szczegółowe informacje, sposoby uruchomienia oraz konfiguracji natomiast z prawej strony jest pokazane polecenie, z którego korzystaliśmy, aby go pobrać.

Jeżeli jesteś początkującym użytkownikiem **Dockera**, niektóre informacje zawarte w instrukcji mogą okazać się niezrozumiałe. W tym miejscu dla mnie istotne jest, abyś wiedział, że **obrazy pobierane są z repozytorium znajdującego się na wskazanej stronie**. Każdy z nich możemy wyszukać we wskazany sposób oraz że warto zatrzymać się przy instrukcji, ponieważ tam autor dość często przekazuje informacje w jaki sposób można z niego korzystać.

3.4. Typy obrazów

Możliwe, że zwróciłeś uwagę przy wyszukiwaniu na ikonę znajdującą się przy obrazie **CentOS. Docker Official Image** oznacza, że obraz jest oficjalnie aktualizowany przez developerów samego Dockera. Najczęściej takie obrazy dotyczą konkretnego systemu lub aplikacji jak na przykład Wordpress. Jeżeli teraz wpiszesz na przykład Ubuntu to taki sam znaczek będzie widniał przy jego nazwie.

Innymi typami z jakimi możesz się spotkać są to **Verified Publisher** czyli zweryfikowany wydawca, jak i również **Sponsored OSS** czyli obrazy sponsorowane przez Dockera.

Pierwszy typ oznacza, że obrazy są bezpośrednio nadzorowane przez określoną firmę i zostały zweryfikowane przez pracowników dockera.

W przypadku **Sponsored OSS** są to obrazy nadzorowane przez projekty **Open Source** w których **Docker** bierze czynny udział i je sponsoruje.

Możesz się zastanowić, dlaczego jest to istotne. Otóż obrazy, które posiadają jeden z oznakowań opisywanych zaliczane są do tak zwanych bezpiecznych. Zostały one zweryfikowane, dlatego istnieje małe ryzyko, że będą zawierały jakieś złośliwe oprogramowanie. Dlatego też przy doborze obrazów należy zachować pewną ostrożność.

W związku z powyższym obrazy możemy zaliczyć do dwóch grup. Jedna z nich to grupa, do której wliczam trzy typy wymienione powyżej, natomiast do drugiej grupy zaliczam obrazy użytkowników które nie były weryfikowane. Przykładem tego może być **Linux Mint**, który nie posiada swojego oficjalnego obrazu. W związku z tym wpisując w wyszukiwarce strony **linuxmint** odnajdziesz tylko obrazy, które zostały stworzone przez użytkowników.

Na zakończenie chciałbym jeszcze dodać, że obrazy oficjalne są zapisywane po nazwie systemu lub aplikacji natomiast pozostałe zawierają dodatkowo nazwę użytkownika, a następnie nazwę obrazu. Zerknij na poniższy przykład:

docker pull centos

W nim korzystamy z oficjalnego obrazu Dockera, natomiast w przypadku nieoficjalnych, przed nazwą obrazu pojawi się nazwa użytkownika jak w poniższym przykładzie na bazie Red Hata, <https://hub.docker.com/r/redhat/ubi8>:

docker pull redhat/ubi8

Z powyższych informacji chciałbym, żebyś zapamiętał, że istnieją **różne typy obrazów** oraz potrafił je zweryfikować przy pomocy strony.

3.5. Różne wersje obrazów

Gdy pobieraliśmy obraz w pierwszym przykładzie otrzymaliśmy informację:

Using default tag: latest

latest: Pulling from library/centos

Jeżeli korzystamy z polecenia do pobrania obrazu w sposób prezentowany w poprzednim przykładzie to zawsze pobierany jest **najnowszy obraz**, tak zwany **latest**. Docker widząc samą nazwę niezależnie od tego czy to oficjalny czy też nie, zawsze dodaje **latest** do nazwy. Inaczej pisząc modyfikuje polecenie do pobierania w następujący sposób:

docker pull centos:latest

Jeżeli wprowadzisz polecenie w takiej formie zostanie pobrany obraz identyczny jak bez wykorzystania słowa **latest**, zresztą sprawdź sam:

docker pull centos:latest

```
latest: Pulling from library/centos
```

```
Digest: sha256:a27fd8080b517143cbbb9dfb7c8571c40d67d534bbdee55bd6c473f432b177
```

```
Status: Image is up to date for centos:latest
```

```
docker.io/library/centos:latest
```

Docker nie pobrał nic, ponieważ wskazany obraz pobraliśmy już wcześniej, co potwierdza powyższy przykład.

Jeżeli jesteśmy w stanie określić najnowszy obraz, to też możemy pobrać jego starsze wersje.

Wystarczy, że po dwukropku zamiast słowa **latest** wprowadzamy wersję, którą chcemy zainstalować. Spójrzmy na poniższy przykład:

docker pull centos:7

```
7: Pulling from library/centos
```

```
2d473b07cdd5: Pull complete
```

```
Digest: sha256:c73f515d06b0fa07bb18d8202035e739a494ce760aa73129f60f4bf2bd22b407
```

```
Status: Downloaded newer image for centos:7
```

```
docker.io/library/centos:7
```

Tym razem nie otrzymaliśmy informacji, że obraz jest już pobrany tylko Docker pobiera wersję przez nas wskazaną. Zerknijmy jeszcze na listę posiadanych obrazów:

docker images

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|-----|----------|---------|------|
|------------|-----|----------|---------|------|

| | | | | |
|--------|---|--------------|---------------|-------|
| centos | 7 | eeb6ee3f44bd | 12 months ago | 204MB |
|--------|---|--------------|---------------|-------|

| | | | | |
|--------|--------|--------------|---------------|-------|
| centos | latest | 5d0da3dc9764 | 12 months ago | 231MB |
|--------|--------|--------------|---------------|-------|

Czyli teraz mamy pobrane w naszej bazie dwie różne wersje systemu **CentOS**. Jeden najnowszy, drugi w wersji 7.

3.6. Usuwanie zbędnych obrazów

Na pewno zdarzy się sytuacja, w której będziesz zmuszony usunąć nieużywane obrazy. Dbanie o porządek w Dockerze ma duże znaczenie, szczególnie jeżeli jest to środowisko deweloperskie, w którym wykonujesz testy. W zastosowaniu serwerowym pobierasz niezbędne obrazy i poza tym nie prowadzisz żadnych testów, dlatego też problem z tak zwanym bałaganem nie dotyczy już środowiska produkcyjnego. Chyba, że nie stosujesz się do zaleceń minimalistycznych które zawsze w takich środowiskach obowiązują.

Ponownie wyświetlimy listę dostępnych obrazów:

docker images

```
REPOSITORY TAG IMAGE ID CREATED SIZE
```

```
centos 7 eeb6ee3f44bd 12 months ago 204MB
```

```
centos latest 5d0da3dc9764 12 months ago 231MB
```

Lista z przykładu nie powinna dziwić nikogo. Pobraliśmy najnowszy obraz systemu **CentOS** oraz jego wersję 7. Tę drugą nie będziemy wykorzystywać w żaden sposób, dlatego też warto jest ją usunąć by nie zabierała miejsca na dysku. Zerknijmy na poniższy przykład:

docker rmi centos

```
Untagged: centos:latest
```

```
Untagged: centos@sha256:a27fd8080b517143cbbb9dfb7c8571c40d67d534bbdee55bd6c473f432b177
```

```
Deleted: sha256:5d0da3dc976460b72c77d94c8a1ad043720b0416bfc16c52c45d4847e53fad6
```

```
Deleted: sha256:74ddd0ec08fa43d09f32636ba91a0a3053b02cb4627c35051aff89f853606b59
```

Do usuwania obrazów używamy polecenia rmi.

Jeżeli zerkniesz na przykład zauważysz, że przez przypadek usunęliśmy nie ten obraz co zamierzaliśmy. Otóż w tym wypadku funkcjonuje również zasada wersji, o której pisałem parę paragrafów wyżej. Jeżeli wpiszesz samą nazwę obrazu to Docker automatycznie doda dwukropka oraz słowo latest. Czyli tym sposobem usunęliśmy najnowszy obraz CentOS.

Chcę zwrócić szczególną uwagę na ten element, na tę funkcję dockera. Najczęściej popełnianym błędem wśród początkujących użytkowników dockera jest usunięcie nie tego obrazu czy też kontenera. Teraz nasza lista obrazów wygląda następująco:

docker images

```
REPOSITORY TAG IMAGE ID CREATED SIZE
```

```
centos 7 eeb6ee3f44bd 12 months ago 204MB
```

Pozostał tylko obraz **CentOS w wersji 7** czyli ten niechciany, aby go usunąć możemy zrobić to na dwa sposoby. Pierwszy z nich polega na wprowadzeniu jego nazwy oraz wersji, natomiast drugi na wpisaniu jego ID:

```
docker rmi centos:7 ← z użyciem nazwy oraz wersji
```

```
docker rmi eeb6ee3f44bd ← z użyciem ID obrazu
```

```
Untagged: centos:7
```

```
Untagged: centos@sha256:c73f515d06b0fa07bb18d8202035e739a494ce760aa73129f60f4bf2bd22b407
```

```
Deleted: sha256:eeb6ee3f44bd0b5103bb561b4c16bcb82328cfe5809ab675bb17ab3a16c517c9
```

```
Deleted: sha256:174f5685490326fc0a1c0f5570b8663732189b327007e47ff13d2ca59673db02
```

Niezależnie od wybranego rodzaju polecenia, zostanie usunięta jego właściwa wersja, dlatego gdy użyjemy polecenia do wyświetlania listy będzie ona pusta.

Tu warto się chwilę zastanowić jaka opcja będzie lepsza, jeżeli chodzi o usuwanie obrazów. Jeżeli używasz nazw obrazów to jest to bardziej czytelniejsze, ale istnieje szansa, że usuniesz to czego tak naprawdę nie chciałeś jak w prezentowanym przykładzie. Natomiast korzystając z ID usuniesz na pewno ten obraz, który wybierzesz, niezależnie od wersji, ale jest to mniej czytelniejsze.

Przyznam, że osobiste w przypadku obrazów posługuję się nazwami natomiast w przypadku kontenerów ID, ale o tych drugich porozmawiamy w następnym rozdziale.

3.7. Wyszukiwanie obrazów w Dockerze

W jednym z poprzednich paragrafów pokazałem w jaki sposób wyszukiwać obrazy przy pomocy strony internetowej **DockerHub**. Tym razem chcę pokazać w jaki sposób zrobić to samo z wykorzystaniem wbudowanego polecenia.

Polecenie jak wszystkie dotychczas nie należy do skomplikowanych, dlatego spójrzmy na poniższy przykład:

```
docker search centos
```

```
NAME DESCRIPTION STARS OFFICIAL AUTOMATED
```

```
centos The official build of CentOS. 7330 [OK]
```

```
kasmweb/centos-7-desktop CentOS 7 desktop for Kasm Workspaces 24
```

```
couchbase/centos7-systemd centos7-systemd images with additional debug... 4 [OK]
```

```
dokken/centos-7 CentOS 7 image for kitchen-dokken 3
```

```
continuumio/centos5_gcc5_base 3
dokken/centos-stream-9 2
dokken/centos-stream-8 2
spack/centos7 CentOS 7 with Spack preinstalled 1
spack/centos6 CentOS 6 with Spack preinstalled 1
corpusops/centos-bare https://github.com/corpusops/docker-images/ 0
dokken/centos-6 CentOS 6 image for kitchen-dokken 0
ustclug/centos Official CentOS Image with USTC Mirror 0
dokken/centos-8 CentOS 8 image for kitchen-dokken 0
bitnami/centos-extras-base 0
datadog/centos-i386 0
corpusops/centos centos corpusops baseimage 0
couchbase/centos-72-java-sdk 0
couchbase/centos-72-jenkins-core 0
bitnami/centos-base-buildpack Centos base compilation image 0 [OK]
fnndsc/centos-python3 Source for a slim Centos-based Python3 image... 0 [OK]
couchbase/centos-69-sdk-build 0
couchbase/centos-69-sdk-nodevtoolset-build 0
couchbase/centos-70-sdk-build 0
dokken/centos-5 EOL DISTRO: For use with kitchen-dokken, Bas... 0
spack/centos-stream 0
```

Jak widać korzystamy z wyrażenia **search**, po którym wprowadzamy nazwę obrazu jaki chcemy wyświetlić. Polecenie posiada filtry, dzięki którym możesz wyświetlić tylko te dane, które zechcesz. Na przykład, aby pokazała się lista obrazów oficjalnych polecenie należy wprowadzić w następujący sposób:

```
docker search -f is-official=true centos
```

```
NAME DESCRIPTION STARS OFFICIAL AUTOMATED
```

```
centos The official build of CentOS. 7330 [OK]
```

W tym wypadku korzystamy z opcji **-f** gdzie wprowadzamy filtr, dzięki któremu w tym wypadku wyświetliśmy tylko obraz oficjalny. Nie jest to jedyna możliwość wykorzystania tego filtra. Możemy jeszcze chcieć wyświetlić obrazy zawierające minimalną ilość gwiazdek:

```
docker search -f stars=1 centos
```

```
NAME DESCRIPTION STARS OFFICIAL AUTOMATED
```

```
centos The official build of CentOS. 7330 [OK]
```

```
kasmweb/centos-7-desktop CentOS 7 desktop for Kasm Workspaces 24
couchbase/centos7-systemd centos7-systemd images with additional debug... 4 [OK]
continuumio/centos5_gcc5_base 3
dokken/centos-7 CentOS 7 image for kitchen-dokken 3
dokken/centos-stream-8 2
dokken/centos-stream-9 2
spack/centos7 CentOS 7 with Spack preinstalled 1
spack/centos6 CentOS 6 with Spack preinstalled 1
```

Natomiast samo polecenia ma możliwość formatowania wyświetlanego wyniku. Dlatego też, jeżeli skonstruujemy je w następujący sposób:

```
docker search --format "table {{.Name}}\t{{.StarCount}}\t{{.IsOfficial}}" centos
```

| NAME | STARS | OFFICIAL |
|----------------------------------|-------|----------|
| centos | 7330 | [OK] |
| kasmweb/centos-7-desktop | 24 | |
| couchbase/centos7-systemd | 4 | |
| dokken/centos-7 | 3 | |
| continuumio/centos5_gcc5_base | 3 | |
| dokken/centos-stream-9 | 2 | |
| dokken/centos-stream-8 | 2 | |
| spack/centos7 | 1 | |
| spack/centos6 | 1 | |
| datadog/centos-i386 | 0 | |
| dokken/centos-6 | 0 | |
| ustclug/centos | 0 | |
| dokken/centos-8 | 0 | |
| corpusops/centos-bare | 0 | |
| bitnami/centos-extras-base | 0 | |
| corpusops/centos | 0 | |
| couchbase/centos-72-java-sdk | 0 | |
| couchbase/centos-72-jenkins-core | 0 | |
| couchbase/centos-70-sdk-build | 0 | |
| fnndsc/centos-python3 | 0 | |
| couchbase/centos-69-sdk-build | 0 | |

```
bitnami/centos-base-buildpack 0
couchbase/centos-69-sdk-nodevtoolset-build 0
spack/centos-stream 0
dokken/centos-5 0
```

Ustawiamy formatowanie tak, aby każdy z rekordów znajdował się w tabeli. Bez słowa **table** nie będzie zastosowany czytelny podział na kolumny tak jak w powyższym przykładzie. Następnie w podwójnym nawiasie { wprowadzamy nazwę kolumny jaką chcemy wyświetlić oddzielając każdą tak zwanym tabulatorem, by pomiędzy nimi był równy odstęp. Na samym końcu podajemy nazwę poszukiwanych obrazów. Obie poznane funkcje możemy połączyć ze sobą w następujący sposób:

```
docker search -f is-official=true --format "table {{.Name}}\t{{.StarCount}}\t{{.IsOfficial}}" centos
```

```
NAME STARS OFFICIAL
centos 7330 [OK]
```

Wyświetliliśmy tylko te oficjalne obrazy oraz zastosowaliśmy kolumny, które najbardziej nas interesują.

3.8. Podsumowanie

Jeżeli dokładnie przeanalizowałeś materiał z tego rozdziału, to chyba przyznasz, że nie ma w tym nic trudnego. Dzięki kilku poleceniom, które posiadają bardzo prostą konstrukcję nauczyłeś się pobierać, wyszukiwać, usuwać obrazy. Z tych najważniejszych rzeczy dotyczących tematu wiesz wszystko. Natomiast w dalszej części wróćmy jeszcze na chwilę do analizowania obrazów oraz wyświetlania bardziej zaawansowanych informacji o nich, ale o tym poświęcony będzie ostatni rozdział.

Rozdział 4: Docker - kontenery

Jak dowiedziałeś się z poprzedniego rozdziału **z obrazu powstaje kontener**. W obrazie zawarty jest tak zwany system bazowy i to na jego podstawie tworzona jest dowolna ilość kontenerów. Po stworzeniu są one identyczne do siebie.

Do tej pory bazowaliśmy jedynie na obrazach już z wgranym tylko systemem. Natomiast teraz chciałbym, abyś zrozumiał, że **w większości wypadków obraz to nie jest tylko system bazowy** taki jak poznaliśmy wcześniej na bazie **CentOS**. Najczęściej w takim obrazie już jest coś zainstalowane jak na przykład **WordPress**.

Czyli mamy **system bazowy** plus do tego wgrany **serwer apache2** lub inny, plus do tego **PHP** i wszystkie niezbędne komponenty, by to działało. Posiadając taki obraz możemy stworzyć dowolną ilość kontenerów z wgranym WordPressem przy pomocy jednego polecenia. Oczywiście do tego jeszcze jest potrzebna baza danych, ale o tym jeszcze porozmawiamy. Jak tworzyć własne obrazy poznasz w jednym z dalszych rozdziałów natomiast teraz chcę pokazać w jaki sposób tworzyć kontenery z już istniejących.

4.1. Pierwszy kontener

Swój pierwszy kontener masz już za sobą. O ile oczywiście wykonywałeś instrukcję krok po kroku przy instalacji Dockera. Niemniej jednak prezentowany sposób stworzenia jaki chcę zaprezentować jest bardziej złożony niż tamten.

W tym rozdziale omówię użytku tam metodę **run**, ale nim to nastąpi chciałbym, żebyś poznał, jak to jest skonstruowane oraz uruchamiane.

4.2. Tworzymy kontener

Istnieją dwa sposoby na stworzenie kontenera. Pierwszy z nich polega na jego utworzeniu, ale w żaden sposób nie uruchomimy. Jeżeli wykonywałeś wszystkie czynności z poprzedniego rozdziału to twoja lista obrazów jest pusta. Dlatego też musimy pobrać obraz **CenOS** ponownie:

docker pull centos

Jak pamiętasz, jeżeli nie określmy po dwukropku wersji jaką chcemy pobrać to Docker automatycznie pobierze najnowszą wersję obrazu jaką posiada w swoim repozytorium. Spójrzmy jeszcze na listę pobranych obrazów:

docker images

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|-----|----------|---------|------|
| centos | | | | |

```
centos latest 5d0da3dc9764 12 months ago 231MB
```

Na liście pojawił się wybrany przez nas system w najnowszej wersji. To wszystko powinieneś znać, powtarzam, bo wszystkie powyższe informacje przydadzą się do stworzenia kontenera oraz innych czynności związanych z tym rozdziałem.

Mamy już pobrany obraz teraz pozostało nam stworzyć kontener. Robimy to przy pomocy polecenia:

docker create centos

Jest to najprostsza forma, dzięki której powstać może kontener. Istnieje lista, w której wyświetlane są wszystkie pobrane obrazy, jak i istnieje lista z wszystkimi kontenerami. Natomiast istnieją dwie listy, gdzie jedna posiada stworzone kontenery, a druga tylko te uruchomiona. Pierwsza z nich zawiera wszystkie kontenery, te uruchomione i te tylko stworzone.

Są one oddzielone od siebie, dlatego wywołanie każdej z nich wygląda zupełnie inaczej.

Aby wyświetlić listę z stworzonymi kontenerami musimy użyć polecenia w następujący sposób:

docker container ls -a

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

```
5b7755a9eab5 centos "/bin/bash" 3 minutes ago Created vibrant_banzai
```

Polecenie jest prawie identyczne jak to z systemu **Linux**. Natomiast istnieje jeszcze trochę prostsza, ale starsza wersja polecenia do wyświetlania stworzonych kontenerów:

docker ps -a

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

```
5b7755a9eab5 centos "/bin/bash" 10 minutes ago Created vibrant_banzai
```

Otrzymaliśmy dwa identyczne wyniki. Musisz znać zarówno jedno, jak i drugie, ponieważ wielu użytkowników Dockera korzysta z rozwiązania starszego. Przyznaje, że do tej grupy należę i ja. W większości przykładów spotkasz się z zastosowaniem polecenia **ps** które samo w sobie również występuje w **Linuksie**. Która wersję wybierzesz pozostawiam Tobie. Ja natomiast, aby nie robić bałaganu i stosować ich zamiennie będę korzystał ze swoich przyzwyczajień i używałem głównie **ps**.

4.3. Analiza przykładu

Wiemy w jaki sposób wyświetlić listę utworzonych kontenerów, dlatego teraz omówmy ją:

- Pierwsza kolumna reprezentuje **numer identyfikacyjny kontenera**. O tym wspominałem przy obrazach, bo one też posiadają swój unikatowy numer ID;
- W następnej kolumnie znajduje się **nazwa obrazu z jakiego kontener powstał**;
- Następnie znajduje się główne polecenie kontenera które uruchamia się przy starcie kontenera;

- W czwartej, wypisana jest informacja, ile czasu temu kontener powstał;
- Kolejna przedstawia status kontenera;
- Przedostatnia określa numer portu, na którym kontener jest udostępniony;
- Ostatnia określa ID kontenera w formie nazwy;

4.4. Uruchomienie kontenera

Stworzyliśmy kontener, ale jak sam możesz powiedzieć nic z tego nie wynika. Pojawił się on na liście.

Dlatego teraz spróbujemy uruchomić go:

docker start vibrant_banzai

vibrant_banzai

Uruchomiliśmy nasz kontener używając jego nazwy jako odpowiedź została zwrócona jego nazwa co widać po powyższym przykładzie. Teraz zerknijmy na listę uruchomionych kontenerów:

docker container ls

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Do wyświetlania listy uruchomionych kontenerów posłużyć się możemy tymi dwoma poleceniami tylko tym razem nie korzystamy dodatkowo z opcji **-a**. Pomimo wyświetlonej listy nasz kontener nie pojawił się na niej. Teraz pytanie brzmi, dlaczego tak się stało?

Otoż uruchomiliśmy nasz kontener, ale po wykonaniu polecenia znajdującego się w kolumnie **COMMAND** nie otrzymał dalszych instrukcji, dlatego zakończył swoje działanie. W jaki sposób uruchomić kontener, aby działał w tak zwanym *trybie ciągłym* pokaże w późniejszym czasie.

4.5. Nadanie nazwy kontenerowi

Możliwe, że zwróciłem uwagę na nazwę jaka została wygenerowana przy tworzeniu kontenera. Jest ona czytelniejsza niż w przypadku **ID**, ale nadal nie na tyle by swobodnie się nią posługiwać. Zobaczmy w jaki sposób możemy stworzyć kontener nadając mu własną nazwę:

docker create --name system centos

Dzięki dyrektywie **--name** jesteśmy w stanie nadać dowolnie nasz kontener. Spójrzmy na listę stworzonych kontenerów:

docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

345a61e0b1d0 centos "/bin/bash" 7 seconds ago Created system

5b7755a9eb5 centos "/bin/bash" About an hour ago Exited (0) 14 minutes ago vibrant_banzai

Nowo utworzony nosi nazwę system, którą my nadaliśmy i rozumiemy w stu procentach. Odpowiednie nazewnictwo kontenerów jest bardzo ważnym aspektem w Dockerze, ponieważ pozwala swobodniej poruszać się pomiędzy poszczególnymi kontenerami. W chwili obecnej posiadamy tylko dwa, ale z czasem zdarzy się, że lista będzie o wiele obszerniejsza i wtedy ratuje nas jedynie odpowiednie nazewnictwo. Pamiętaj o tym, bo będziesz musiał operować tak naprawdę na dwóch listach, jednej z stworzonymi kontenerami i drugiej z już uruchomionymi, dlatego też istotne jest odpowiednie nazewnictwo.

Istnieje jeden sposób. Dla niektórych wydaje się być bardziej czytelny niż prezentowany. Wygląda on następująco:

```
docker container create --name system-centOS centos
```

Jest to dłuższe polecenie o jedno słowo, ale może okazać się szczególnie dla wzrokowców czytelniejsze. Wiemy, że opcja **create** dotyczy kontenerów, ponieważ poprzedza je słowo **container**.

Pokazuję to wszystko, ponieważ przy analizie przykładów czy też, gdy ktoś będzie tłumaczył coś związanego z dockerem, stosować będzie albo jeden albo drugi sposób. Nie chcę abyś był zaskoczony, że coś takiego występuje, a Ty o tym nie wiesz.

Natomiast którą wersję stosować? Moim zdaniem tę którą najbardziej rozumiesz, tę która jest wygodniejsza w związku z twoimi preferencjami. Ja osobiście korzystam z tych krótkich rozwiązań, ponieważ jest mi tak wygodniej.

Chciałbym zwrócić na jeszcze jedno uwagę. Jeżeli analizujesz przykłady dość wnikliwie to możliwe, że zauważysz. Kontener, który uruchomiliśmy w kolumnie **Status** posiada informację **Exited (0) 14 minutes ago**. Natomiast kontener, który był tylko stworzony posiada słowo **created**. W ten sposób można odróżnić kontener tylko stworzony od tego który był już uruchomiony.

4.6. Usuwanie kontenerów

W przypadku kontenerów wspomniałem, że bardziej korzystam z **ID** niż z nazw. Pomimo tego co przeczytałeś powyżej nadal tak jest. Może się wydawać to nielogiczne, ale nazwy kontenerów służą mi tylko do rozpoznawania ich oraz do używania we własnej sieci. O sieciach porozmawiamy w kolejnym rozdziale natomiast teraz sprawdźmy w jaki sposób jesteśmy w stanie usunąć niepotrzebne kontenery.

Polecenie niewiele różni się od poznanych wcześniej. Jednak nim przejdziemy do samego usuwania wyświetlimy naszą listę już stworzonych:

```
docker ps -a
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|--------|-------------|-------------------|------------|----------------|----------------|
| cd6b6159549d | centos | "/bin/bash" | 37 seconds ago | Created | | system-centOS |
| 345a61e0b1d0 | centos | "/bin/bash" | 14 minutes ago | Created | | system |
| 5b7755a9eb5 | centos | "/bin/bash" | About an hour ago | Exited (0) | 28 minutes ago | vibrant_banzai |

Usuniemy kontener, któremu Docker nadał automatycznie nazwę. Robimy to przy pomocy poleceń:

```
docker rm vibrant_banzai
```

```
docker container rm vibrant_banzai
```

Niezależnie od tego, z którego skorzystasz polecenia, zostanie usunięty wskazany kontener.

Istnieje jeszcze sposób na usunięcie wszystkich kontenerów:

```
docker container prune
```

WARNING! This will remove all stopped containers.

```
Are you sure you want to continue? [y/N] y
```

Deleted Containers:

```
9f51fe677bbe9d544276cdc23d1d8524c5754df24acbc7745c15732d7ae6daa3
```

```
cd6b6159549de8ed7e2c4842e314afeebfa26330c7b8435be7183425046b4eb0
```

Total reclaimed space: 0B

Sposoby prezentowane można również zastosować z wykorzystaniem ID kontenerów. Myślę, że nie ma sensu tego prezentować na przykładach, ponieważ różniły by się tylko miejscem, w którym podawałyś ID zamiast nazwy.

4.7. Pobranie obrazu oraz uruchomienie kontenera

Jeżeli korzystałeś z tej publikacji do instalacji Dockera i wykonywałeś wszystko krok po kroku to, gdy sprawdzałeś, czy instalacja przebiegła poprawnie to korzystałeś z polecenia **run**. W tym paragrafie zajmiemy się tym poleceniem. Nim jednak do tego przejdziemy usuń wszystkie obrazy oraz kontenery z jakich dotychczas korzystałeś.

Po usunięciu wszystkiego wprowadźmy ponownie polecenie, którego użyliśmy w rozdziale 2:

```
docker run hello-world
```

Omówimy sobie teraz wszystkie czynności jakie wykonał Docker, gdy użyliśmy tego polecenia.

Pierwsze to pobrał obraz:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:62af9efd515a25f84961b70f973a798d2eca956b1b2b026d0a4a63a3b0b6a3f2
Status: Downloaded newer image for hello-world:latest
```

Nie znalazł go lokalnie na dysku, dlatego tak jak w przypadku polecenia **pull** pobrał go w wersji **latest**.

W następnym kroku stworzył oraz uruchomił kontener:

```
Hello from Docker!
```

Potwierdzić to możemy wyświetlając listę kontenerów:

```
docker ps -a
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|-------------|----------|---------------|------------|---------------|--------------|
| 479079a5d226 | hello-world | "/hello" | 6 minutes ago | Exited (0) | 6 minutes ago | nifty_yallow |

Zwrć uwagę, że w kolumnie **COMMAND** kontener ma informację o wykonaniu skryptu **hello**, a następnie kończy działanie kontenera.

Jak widzisz wszystko co do tej pory zrobiliśmy przy pomocy polecień **pull**, **create** oraz **start** możemy wykonać jednym. Pokazuje je dopiero teraz, ponieważ chciałem, abyś wiedział w jaki sposób wszystko wykonywane jest od samego pobrania obrazu po uruchomienie kontenera.

Wiesz już, że dzięki poleceniu run możesz połączyć wszystkie poznane dotychczas polecenia. Przyznam, że jest ono głównie wykorzystywane do czynności poznanych wcześniej. Rzadziej spotkasz się by ktoś robił to krok po kroku tak jak ja na początku tego rozdziału.

Natomiast tak samo jak w przypadku opcji **create** jesteś w stanie stworzyć kontener ustawiając mu odpowiednią nazwę:

```
docker run --name hello hello-world
```

Zerknijmy na listę kontenerów:

```
docker ps -a
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|-------|---------|---------|--------|-------|-------|
| | | | | | | |

```
ce59b791efd2 hello-world "/hello" 3 minutes ago Exited (0) 3 minutes ago hello  
479079a5d226 hello-world "/hello" 21 minutes ago Exited (0) 21 minutes ago nifty_yallow
```

Mamy na liście dwa kontenery korzystające z tego samego obrazu. Jednemu z nich Docker nadał automatycznie wygenerowaną nazwę, drugiemu my nadaliśmy. W ten sposób powstały dwa kontenery. Piszę tutaj o tym, bo jeżeli korzystamy z polecenia **run** to zawsze powstawać będzie nowy kontener, a nie zawsze chcemy tego. Czasami potrzebujemy uruchomić już istniejący, a nie tworzyć nowy. Do tego celu posłużą nam poznane wcześniej polecenie z dodatkową opcją:

docker container start -i hello

W przypadku gdybyśmy uruchamiali kontener w trybie ciągłym, to nie musielibyśmy korzystać z opcji **-i**. Natomiast jeżeli chcemy wyświetlić odpowiedź jaką da nam kontener po uruchomieniu to musimy wprowadzić je z tą dyrektywą. W innym wypadku wyświetli się sama nazwa kontenera.

Czyli w przypadku, gdy uruchamiamy kontener tak aby działał cały czas to nie musimy korzystać z opcji **-i**, natomiast gdy chcemy tylko wyświetlić informację jaką otrzymamy po uruchomieniu to musimy skorzystać ze wspomnianej opcji.

4.8. Zmiana nazwy kontenera

Może się zdarzyć, że zapomnimy użyć opcji do nadania nazwy kontenerowi i zostanie utworzony z automatycznie nadaną przez Dockera. Nie musimy w tym przypadku usuwać już utworzonego tylko wystarczy, że zmienimy jego nazwę przy pomocy polecenia:

docker container rename 479079a5d226 hello-pierwszy

Do zmiany nazwy użyłem ID kontenera, któremu chcę zmienić nazwę, a następnie podałem nazwę, której chcę użyć. Zerknijmy teraz na listę kontenerów:

docker ps -a

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
ce59b791efd2 hello-world "/hello" 14 minutes ago Exited (0) 6 minutes ago hello  
479079a5d226 hello-world "/hello" 32 minutes ago Exited (0) 32 minutes ago hello-pierwszy
```

Została zmieniona nazwa kontenera z tej automatycznie wygenerowanej na własną. Jest to bardzo wygodne i warto zapamiętać. Mi bardzo często zdarza się zapomnieć o opcji **--name**, dlatego to polecenie jest przeze mnie bardzo często używane.

4.9. Uruchomienie kontenera w trybie ciągłym

Teraz zajmijmy się sposobem na uruchomienie kontenera w trybie ciągłym. Do tego celu posłużymy się poznanymi wcześniej poleceniami, a dodamy do nich tylko dodatkowe opcje.

Pierwszy sposób jakiego używaliśmy polegał na **pobraniu**, **stworzeniu** oraz **uruchomieniu** kontenera w trzech krokach. Teraz zróbcmy to samo w poszczególnych pozycjach, ale tym razem dodając odpowiednie opcje.

Pierwsze co robimy to pobieramy obraz **CentOS**, ten krok możesz pominąć, jak masz już to zrobione:

docker pull centos

Nic nowego, pobraliśmy obraz jak robiliśmy to dotychczas. Teraz stworzymy z niego kontener:

`docker create -it --name system-centos centos`

Lub z alpine:

`docker create -it --name system-alpinek alpine`

Tym razem przy tworzeniu użyliśmy opcji **-it**.

Pierwsza **-i** utrzymuje otwarty standardowy strumień wejścia, dzięki któremu jesteśmy w stanie w **cli** kontenera wydawać polecenia, druga **-t** uruchamia powłokę bash.

Pozostaje uruchomić kontener:

`docker start system-alpinek`

Zerknijmy teraz na listę, ale tym razem uruchomionych kontenerów:

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

d76b6cd03a1a centos "/bin/bash" 36 seconds ago Up 6 seconds system-centos

Tym razem nasz kontener jest uruchomiony w tak zwanym trybie ciągłym. Nie został zamknięty. To samo jesteśmy w stanie w trochę krótszy sposób osiągnąć korzystając z polecenia **run**:

docker run -itd --name alpinus alpine

Skorzystałem z dodatkowej opcji **-d** ponieważ jeżeli kontener zostałby uruchomiony w powyższy sposób automatycznie przeszedłbyś do powłoki systemowej, a tym zajmiemy się w następnym paragrafie. W związku z tym opcja **-d** służy do uruchomienia kontenera, ale powoduje, że nie przechodzimy automatycznie do konsoli. W ten sposób posiadamy dwa kontenery uruchomione w trybie ciągłym:

docker ps

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
fdf5a4ebef85 centos "/bin/bash" 11 minutes ago Up 11 minutes system-os-run
d76b6cd03a1a centos "/bin/bash" 24 minutes ago Up 23 minutes system-centos
```

Oczywiście, w ramach testów możesz jak najbardziej przetestować:

docker run -it --name alpinista alpine

Jak wiesz co robić to możesz się teraz pobawić. Jak nie wiesz to spokojnie, za chwilę przejdziemy do szczegółów. Wychodzimy:

exit

4.10. Zamykanie kontenera

W związku z tym, że mamy uruchomione dwa kontenery zatrzymajmy jeden z nich:

docker stop fdf5a4ebef85

Tym razem posłużyłem się ID kontenera. Zastosowanie polecenia jest dla mnie tak oczywiste, że nie ma co się rozpisywać, dlatego od razu zobaczymy w jaki sposób możemy skorzystać z powłoki systemowej.

4.11. Uruchomienie powłoki kontenera

Czynność tę możemy wykonywać przy użyciu dwóch poleceń.

Pierwszym z nich jest **attach** które, jeżeli użyte po zakończeniu pracy, zamknie również i kontener. Na tym etapie powinniśmy mieć uruchomiony jeden kontener:

docker ps

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
d76b6cd03a1a centos "/bin/bash" 39 minutes ago Up 3 minutes system-centos
```

Zalogujmy się do powłoki naszego kontenera:

docker attach system-centos

W niej jesteś w stanie wykonać wszystko co w systemie **Linux**. Powłoka niczym się nie różni od normalnie działającego systemu czy też serwera. Zawiera jednak tylko niezbędne oprogramowanie do jego funkcjonowania. Teraz jeżeli wpiszesz exit to wyjdiesz z powłoki kontenera jak i również zamkniesz kontener.

Jeżeli skorzystałeś z powyższego polecenia to teraz musisz uruchomić ponownie kontener:

docker start system-centos

Tym razem uruchomimy naszą powłokę przy pomocy polecenia **exec**:

```
docker exec -it system-centos /bin/bash
```

Polecenie różni się tylko opcjami, które opisywałem w jednym z poprzednich paragrafów jak i również na samym końcu podajemy informacje Dockerowi, że chcemy wejść do powłoki bash. Wszystko będzie funkcjonowało identycznie jak w poprzednim wypadku, jednak tutaj po wyjściu z kontenera nie zostanie on automatycznie zamknięty, ale wprowadzone zmiany zostaną zapisane. Jednak to rozwiązanie posiada jeszcze inną dodatkową możliwość. Otóż zamiast uruchomić powłokę możesz od razu wydać polecenie:

```
docker exec -it system-centos ls
```

```
bin etc lib lost+found mnt proc run srv tmp var dev home lib64 media opt root sbin sys usr
```

Zostały wypisane wszystkie foldery z głównego katalogu systemowego. Bywa to bardzo pozyteczna funkcja szczególnie gdy potrzebujemy coś na szybko wykonać w danym kontenerze.

Jeżeli chodzi o polecenie **run**, w którym użyliśmy dodatkowej opcji **-d**. Gdy decydujemy się na automatyczne logowanie do powłoki to pamiętajmy, że w tym wypadku zostanie wybrana opcja **attach** w związku z tym po wykonaniu czynności w powłoce kontener automatycznie się zamknie.

4.12. Podsumowanie

Po przeczytaniu całości tego rozdziału chyba potwierdzisz, że kontenery posiadają naprawdę ogromne możliwości. Dzięki informacjom tutaj poznanym jesteś już w stanie bezproblemowo się nimi posługiwać. Ten rozdział jest jednym z najważniejszych. Poznanie tego wszystkiego jest bardzo istotne, dlatego jeżeli czegoś nie rozumiesz, to poczytaj o tym jeszcze. Wydaje mi się, że opisałem wszystko w jak najprostszym języku tak aby każdy zrozumiał.

Rozdział 5: Docker - Sieci

Sieci są ważnym elementem Dockera, ponieważ to one scalają wszystko w całość. To właśnie dzięki sieci występuje izolacja, o której pisałem w pierwszym rozdziale. Bez dalszego wstępnu zapraszam do zapoznania się z poniższym tekstem.

5.1. Po co używać sieci

Kiedy tworzysz kontener jest on **automatycznie dodawany do sieci już wygenerowanej przez Dockera**. Niestety, od momentu stworzenia, wszystkie kontenery trafiają do tej samej sieci. I mamy tu pewne zagrożenie. Oczywiście, ryzyko jest uzależnione od tego, co dokładnie uruchamiasz w kontenerze. Czynności, które wykonywaliśmy dotychczas nie stanowiły, żadnego zagrożenia dla naszego bazowego systemu. Dlatego też czy kontenery znajdowałyby się w takiej, czy innej sieci nie miało wielkiego znaczenia.

Wyobraź sobie, że musisz uruchomić serwer z zainstalowanym **Apache2**, **PHP** i **MySQL**. Nie wykorzystując kontenera instalujesz poszczególne aplikacje na serwerze. Jeżeli odnaleziona zostanie luka w oprogramowaniu, to intruz może uzyskać pełny dostęp do twojego serwera.

W przypadku, gdy korzystasz z Dockera, to serwer z **PHP** jest jednym kontenerem, a **MySQL** drugim. W tym wypadku intruz uzyska dostęp tylko do kontenera, a nie całego serwera. Nie uzyska dostępu również do logów które znajdują się poza kontenerami. Przy odpowiednim ustawieniu kopii bezpieczeństwa łatwiej jest przywrócić dane z kontenera niż robić to w związku z całym serwerem.

5.2. Automatyczna sieć Dockera

Wiesz już, że każdy z kontenerów jest dodawany do automatycznej sieci. Jest ona utworzona wraz z instalacją Dockera i na to nie masz wpływu. Spójrzmy, jak wygląda lista sieci, którą Docker sam wygenerował:

docker network ls

```
NETWORK ID NAME DRIVER SCOPE
e478b9cd411f bridge bridge local
e2198787e635 host host local
e6137ead4366 none null local
```

Ponownie zwróć uwagę na prostotę związaną ze składnią w Dockerze. Polecenie **ls** występuje w systemie Linux i służy do wyświetlania zawartości w katalogu. W przypadku prezentowanego przykładu wyświetli listę dostępnych sieci. Sieć, z której korzystamy, jest pierwsza na liście.

Automatyczna sieć Dockera zapewnia izolację i to jest jej największym atutem. Jak zauważysz, nie istnieje tylko jedna sieć, która tworzy się wraz z instalacją. Każda z nich różni się znacznie od siebie. Zostały one stworzone w celu umożliwienia różnego rodzaju komunikacji pomiędzy kontenerami, hostem, na którym zostały uruchomione oraz innymi, zewnętrznymi urządzeniami. Te trzy podstawowe sieci podnoszą możliwości konfiguracyjne, z jakich możesz skorzystać, używając Dockera. Przez co zwiększały jego przydatność w szczególnych sytuacjach.

Pierwszą siecią z listy jest **bridge**. To do niej dodawane są automatycznie nasze kontenery przy ich powstawaniu. Natomiast poniżej pozwoliłem sobie na krótki opis każdej z nich.

Sieć bridge, czyli sieć pomostowa, zapewnia stałą łączność pomiędzy hostem a kontenerami. Dzięki niej kontenery mogą komunikować się między sobą, jak i również sam host używając przypisanych adresów IP, może komunikować się z kontenerami.

Sieć host, jest stosowana w sytuacji, gdy chcesz, aby kontener używał adresu IP hosta. Przydaje się, szczególnie, gdy musisz uzyskać dostęp do zasobów hosta. Jednak korzystając z tego typu sieci, pozbawiasz kontenery izolacji, jaką oferuje opisywana poprzednio sieć pomostowa bridge.

Sieć none używana jest, gdy chcemy odłączyć kontener od całej sieci. Przydaje się w sytuacji, gdy chcesz uruchomić kontener całkowicie odizolowany od świata zewnętrznego. Nie może komunikować się ani z hostem, ani z innymi kontenerami.

W dalszej części tego rozdziału poznasz sposoby tworzenia własnych sieci. Tutaj natomiast masz już trzy automatycznie wygenerowane, z których możesz skorzystać natychmiast. Otóż wystarczy, że przy uruchomieniu kontenera określisz mu, w jakiej ma się znaleźć. Oto przykład:

```
docker run --network none <image>
```

W powyższy sposób tworzony kontener zostanie dodany do sieci **none**. Dokładny opis pomijam, ponieważ znajduje się powyżej. Adekwatnie tak samo możesz zrobić z siecią **host**, jak i również **bridge**. Jednak w przypadku ostatniej, gdy chcemy, by kontener był dodany do niej, pomijamy opcję **--network**, ponieważ bridge jest ustalona standardowo.

Opisywane tutaj sieci nie są jedynymi. Istnieją jeszcze dwie, które w skrócie opisuję poniżej.

Sieć overlay, czyli sieć nakładkowa, umożliwia kontenerom działającym na różnych hostach Docker, komunikować się między sobą. Wykonywane jest to tak jakby, były one w tej samej sieci. Omawiana sieć jest szczególnie przydatna, gdy chcemy uruchamiać kontenery na różnych hostach jak na przykład w Docker Swarm. Pamiętaj, że aby korzystać z sieci tego typu, hosty Docker muszą należeć do klastra **Docker Swarm**.

Sieć MACVLAN umożliwia hostowi, na którym zainstalowany jest Docker na udostępnienie kontenerów bezpośrednio w fizycznej sieci hosta. Dzięki temu jesteśmy w stanie nadać im unikatowy **adres MAC**, jak i również możemy skorzystać z istniejącej sieci do nawiązania połączenia z kontenerem. W sieciach tego typu kontenery mogą posiadać własne adresy IP oraz interfejsy sieciowe. Można je konfigurować w identyczny sposób jak na hoście. Dzięki temu można traktować kontenery jako fizyczne urządzenia w sieci, a nie jako urządzenia w izolowanym środowisku.

5.3. Dbanie o porządek

Nim przejdziemy dalej musimy zrobić porządek z kontenerami, których już nie używamy:

docker container prune

WARNING! This will remove all stopped containers.

Are you sure you want to continue? [y/N] y

Deleted Containers:

```
fdf5a4eb855a2836190307b6eff07fe1ee3ec6647e727c630cf5d64e910cf0
d76b6cd03a1a34aeb500103fc076c92a71e946f72e2dcf0d41e73b174300ad2f
ce59b791efd2a62c41f7acda8281c16db862671eff089821bbf7fbbe58aedc0d
479079a5d226490041fda47f43d13f1edbf2a423ef51f6a9fb1eb8b772a8123b
```

Total reclaimed space: 41B

Następnie możemy pozbyć się obrazu *hello-world* ,ponieważ nie będziemy z niego korzystali:

docker images

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|-------------|--------|--------------|---------------|--------|
| hello-world | latest | feb5d9fea6a5 | 12 months ago | 13.3kB |
| centos | latest | 5d0da3dc9764 | 12 months ago | 231MB |

docker rmi feb5d9fea6a5

Untagged: hello-world:latest

Untagged: hello-world@sha256:62af9efd515a25f84961b70f973a798d2eca956b1b2b026d0a4a63a3b0b6a3f2

Deleted: sha256:feb5d9fea6a5e9606aa995e879d862b825965ba48de054caab5ef356dc6b3412

Deleted: sha256:e07ee1baac5fae6a26f30cabfe54a36d3402f96afda318fe0a96cec4ca393359

Wiem, że trochę odbiegłem od głównego tematu, ale wiedz, że takie czynności powinieneś dość często wykonywać. Szczególnie gdy się uczysz. Zbyt duża liczba kontenerów może doprowadzić do zawrotu głowy nawet jeżeli są one odpowiednio opisane. Dlatego warto raz na jakiś czas posprzątać.

Usuwanie sieci:

```
network rm <nazwa sieci>
```

5.4. Inspekcja kontenera

Po wykonaniu zadań porządkujących, wypróbujmy sieć. Na samym początku stworzmy dwa kontenery oparte o obraz systemu **CentOS**:

```
docker run -itd --name centos-1 centos
```

```
docker run -itd --name centos-2 centos
```

Lub z Alpine:

```
docker run -itd --name alpine-1 alpine
```

Stworzyliśmy dwa kontenery oparte o obraz **centos** oraz je uruchomiliśmy. Nie zalogowaliśmy się zarówne do jednego, jak i drugiego. Dzięki opcji **-d** pozostawiliśmy je uruchomione. Dla przypomnienia:

```
docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|--------|-------------|---------------|--------------|----------|-------|
| 24f332c44615 | centos | "/bin/bash" | 4 seconds ago | Up 3 seconds | centos-2 | |
| ed8ed192cdaf | centos | "/bin/bash" | 8 seconds ago | Up 6 seconds | centos-1 | |

Jak przedstawiono na powyższym przykładzie są one uruchomione. **Urządzenia w sieci komunikują się przy pomocy adresu IP.** W przypadku sieci Dockera nie jest inaczej. Jak wspominałem każdy kontener jest dodawany do standardowej sieci. W związku z tym otrzymuje on indywidualny adres IP, ponieważ traktowany jest jako urządzenie.

Jednak adres IP nie jest wyświetlany na żadnej z list. Aby go poznać, musimy użyć polecenia w następujący sposób:

```
docker container inspect ed8ed192cdaf
```

Tak zwanej *inspekcji* poddajemy kontener **centos-1**. W przykładzie korzystam z **ID** kontenera, a nie jego nazwy. Wynik jest bardzo obszerny, dlatego też nie umieściłem go w całości. Dla nas najbardziej istotną informacją będzie ta dotycząca sieci:

```
"NetworkSettings": {
```

```
    "Bridge": "",
```

```
    "SandboxID": "ef21161f6d44bdead254c0723add43eb1e13a0b4557e96c90766290da065cbc9",
```

```
    "HairpinMode": false,
```

```
    "LinkLocalIPv6Address": "",
```

```

"LinkLocalIPv6PrefixLen": 0,
"Ports": {},
"SandboxKey": "/var/run/docker/netns/ef21161f6d44",
"SecondaryIPAddresses": null,
"SecondaryIPv6Addresses": null,
"EndpointID": "366c4cca229091160d866063986eee1f170ed5bfa2b975e9d83f71ebb424488a",
"Gateway": "172.17.0.1",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"IPAddress": "172.17.0.2",
"IPPrefixLen": 16,
"IPv6Gateway": "",
"MacAddress": "02:42:ac:11:00:02",
"Networks": {
"bridge": {
"IPAMConfig": null,
"Links": null,
"Aliases": null,
"NetworkID": "e478b9cd411f00efba2ce8d5bea7d317e2e81ccf005519781d6d1d95fb2c9f9a",
"EndpointID": "366c4cca229091160d866063986eee1f170ed5bfa2b975e9d83f71ebb424488a",
"Gateway": "172.17.0.1",
"IPAddress": "172.17.0.2",
"IPPrefixLen": 16,
"IPv6Gateway": "",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"MacAddress": "02:42:ac:11:00:02",
"DriverOpts": null
}
}
}

```

W końcowej części przykładu powinieneś mieć podobne informacje jak te, znajdujące się w powyższym. Informacje o adresie IP znajdują się w dwóch miejscach. Zostały one pogrubione. Wykonajmy teraz to samo w stosunku do **centos-2**:

```
docker container inspect 24f332c44615
```

```
"IPAddress": "172.17.0.3",
```

W tym wypadku nie będę pokazywał całości, tylko wkleitem adres IP danego kontenera. Reasumując zebrane do tej pory informacje mamy dwa kontenery o adresach IP:

centos-1: 172.17.0.2

centos-2: 172.17.0.3

Oczywiście pamiętaj, że u Ciebie adresy mogą być zupełnie inne. Zalogujmy się teraz do pierwszego kontenera i sprawdźmy, czy jest w stanie komunikować się z drugim:

```
docker exec -it centos-1 /bin/bash
```

W Alpine działa ta komenda:

```
docker exec -it alpine-1 /bin/sh
```

W celu sprawdzenia czy oba kontenery widzą się wzajemnie, posłużymy się poleceniem **PING**. Wiesz już, że system bazowy ma zainstalowane tylko oprogramowanie umożliwiające mu uruchomienie i nic poza tym. W związku z tym, musimy zainstalować w kontenerze program obsługujący polecenie **PING**. Na samym początku jednak musimy zaktualizować repozytorium:

yum update

```
Failed to set locale, defaulting to C.UTF-8
```

```
CentOS Linux 8 - AppStream 189 B/s | 38 B 00:00
```

```
Error: Failed to download metadata for repo 'appstream': Cannot prepare internal mirrorlist: No URLs in mirrorlist
```

Zostaliśmy poinformowani przez menadżer pakietów **yum**, że nie może znaleźć danych dotyczących repozytoriów. Od początku 2022 roku zakończono wsparcie dla tej wersji systemu. Oznacza to, że nie otrzymuje już aktualizacji z oficjalnego repozytorium CentOS. Niemniej jednak nie jest to koniec samego **CentOS**. Masz teraz dwie możliwości, aby uzyskać aktualizacje. Pierwsza z nich polega na zmianie adresów w repozytoriach. Druga na zaktualizowaniu bieżącej wersji systemu do **CentOS Stream**. Ja skorzystam z tej pierwszej, dlatego pierwsze co zrobię to przejdę do miejsca, gdzie znajdują się pliki z repozytoriami:

```
cd /etc/yum.repos.d/
```

Następnie podmieniamy repozytoria:

```
sed -i 's/mirrorlist/#mirrorlist/g' /etc/yum.repos.d/CentOS-*
```

```
sed -i 's|#baseurl=http://mirror.centos.org|baseurl=http://vault.centos.org|g' /etc/yum.repos.d/CentOS-*
```

Na samym końcu aktualizujemy:

```
yum update -y
```

Razem z aktualizacją został zainstalowany pakiet **iutils** w którym znajduje się skrypt odpowiedzialny za polecenie **ping**.

Posiadając uruchomione dwa kontenery i będąc w powłoce pierwszego użyjmy polecenia **ping**:

```
ping -c 1 172.17.0.3
```

```
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.  
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.067 ms  
--- 172.17.0.3 ping statistics ---  
1 packets transmitted, 1 received, 0% packet loss, time 0ms  
rtt min/avg/max/mdev = 0.067/0.067/0.067/0.000 ms
```

Przesyłanie zakończyło się powodzeniem. W związku z tym oba kontenery są w tej samej sieci i widzą się wzajemnie. Teraz jeżeli chcesz, to możesz w drugim kontenerze wykonać to samo, co w pierwszym i sprawdzić czy będzie działał. Oczywiście robisz to dla własnej praktyki ponieważ jeżeli pierwszy odpowiada, to i drugi będzie.

5.5. Dlaczego tak

Możesz się zastanawiać dlaczego zrobiłem to w ten sposób, a nie w inny. Przecież część którą zapisałem w poprzednim paragrafie nie ma zupełnie związku z Dockerem. Wiadomo łatwiej byłoby zmienić system.

Otoż, nie!

Przy wykonywaniu jakichkolwiek czynności administracyjnych związanych bezpośrednio z Dockerem napotkasz na podobne problemy. Będziesz musiał je w jakiś sposób rozwiązać. Ponieważ nie był to serwer, podjąłem decyzję by zmienić repozytoria. Ma to na celu zobrazować, że w kontenerze opartym o system Linux wszystkie polecenia wykonuje się tak samo. Czyli wykonaliśmy podstawowe czynności w identyczny sposób jak w systemie zainstalowanym na dysku.

5.6. Tworzenie własnej sieci

Wszystkie nowe kontenery są dodawane automatycznie do domyślnej sieci Dockera. Możesz je jednak oddzielić tak, aby nie widziały się wzajemnie. W Dockerze jesteś w stanie tego dokonać tworząc własną sieć. Robi się to za pomocą:

```
docker network create siec
```

Sprawdźmy teraz jak wygląda lista wszystkich dostępnych sieci:

```
docker network ls
```

```
NETWORK ID NAME DRIVER SCOPE
e478b9cd411f bridge bridge local
e2198787e635 host host local
e6137ead4366 none null local
033ab85e4995 siec bridge local
```

Stworzona przez nas znajduje się na samym dole listy.

5.7. Dodanie do sieci

Dodajmy do niej nasze dwa kontenery:

```
docker network connect siec centos-1
```

```
docker network connect siec centos-2
```

Zastanówmy się teraz po co w ogóle dodawać kontenery do naszej sieci. Jest to bardzo ważne pytanie, ponieważ istnieją istotne powody, dla których powinniśmy stosować własną sieć.

Pierwszym z nich jest to, że własna sieć posiada o wiele większe możliwości konfiguracyjne. Otóż w takim wypadku to Ty decydujesz, które kontenery do niej przydzielasz. Nie są one w żaden sposób automatycznie dodawane. Dlatego pełną kontrolę nad tym masz Ty.

Drugim powodem jest to, że występuje inny rodzaj komunikacji pomiędzy kontenerami. W przypadku, gdy korzystasz z automatycznie generowanej sieci to, aby komunikować się z innym musisz postużyć się jego adresem IP. Natomiast, gdy kontenery dodasz do tej samej własnej sieci, to możesz posługiwać się ich nazwami. Wiadomo łatwiej jest zapamiętać nazwę niż zbiór liczb.

Kolejnym powodem jest dodatkowe bezpieczeństwo. Do twojej sieci nie jest nic automatycznie dodawane. Zatem nowo powstałe kontenery nie będą wiedziały o istnieniu pozostałych, dopóki ich sam tam nie dodasz.

5.8. Inspekcja sieci

W jednym z poprzednich paragrafów oddaliśmy inspekcji kontenery. Tym razem chcę pokazać jeszcze jedną możliwość, jaką jest **inspekcja sieci**. Dzięki temu sposobowi jesteśmy w stanie sprawdzić, które kontenery zostały do niej dodane:

```
docker network inspect siec
```

```
[
```

```
{
```

```
  "Name": "siec",
```

```

"Id": "033ab85e49951cc93174ba12ac48fa7080ceeff2802e5d3555c162b6395cc2e0",
"Created": "2022-10-05T10:36:06.052049146+02:00",
"Scope": "local",
"Driver": "bridge",
"EnableIPv6": false,
"IPAM": {
    "Driver": "default",
    "Options": {},
    "Config": [
        {
            "Subnet": "172.18.0.0/16",
            "Gateway": "172.18.0.1"
        }
    ]
},
"Internal": false,
"Attachable": false,
"Ingress": false,
"ConfigFrom": {
    "Network": ""
},
"ConfigOnly": false,
"Containers": {
    "24f332c446150327c30b80d2be7ecb5ef18e3c45bb2cb35acbf247f18df18d64": {
        "Name": "centos-2",
        "EndpointID": "8f592d02541de914196db1176edeee0d616e1fccab1c884ed976d45beb206ea2",
        "MacAddress": "02:42:ac:12:00:03",
        "IPv4Address": "172.18.0.3/16",
        "IPv6Address": ""
    },
    "ed8ed192cdaf6c436a2f0d5c5c3759c7aeec8b8d33b85bcfc6f098c9955e": {
        "Name": "centos-1",
        "EndpointID": "d330b207f15b46a589258bebc528a26df39d240619d9e79f82d57c947c641cc0",
        "MacAddress": "02:42:ac:12:00:02",
    }
}

```

```

    "IPv4Address": "172.18.0.2/16",
    "IPv6Address": ""
}
},
"Options": {},
"Labels": {}
}
]

```

Uzyskaliśmy dokładną informację, jak wygląda adresowanie naszej sieci oraz jaka ilość kontenerów się w niej znajduje. Natomiast wykonajmy jeszcze inspekcję na sieć generowaną przez Dockera:

docker network inspect bridge

```

"Containers": {
"24f332c446150327c30b80d2be7ecb5ef18e3c45bb2cb35acbf247f18df18d64": {
"Name": "centos-2",
"EndpointID": "389e57742c6e7aa6a0732b11e4979d7ac1cf547c4e7c4b84c53723fec5ea05ca",
"MacAddress": "02:42:ac:11:00:03",
"IPv4Address": "172.17.0.3/16",
"IPv6Address": ""
},
"ed8ed192cdaf6c436a2f0d5c5c3759c7aeec8b8d33b85bcfc6f098c9955e": {
"Name": "centos-1",
"EndpointID": "366c4cca229091160d866063986eee1f170ed5bfa2b975e9d83f71ebb424488a",
"MacAddress": "02:42:ac:11:00:02",
"IPv4Address": "172.17.0.2/16",
"IPv6Address": ""
}
},

```

Wkleięm tę część, która będzie nas najbardziej interesowała. Analizując dwa powyższe przykłady widać, że nasze kontenery znajdują się zarówno w sieci wygenerowanej przez Dockera, jak i również w naszej sieci.

5.9. Odłączenie od sieci

Nie zawsze chcemy, aby kontenery znajdowały się w kilku sieciach. Te dwa przykłady udowadniają, że istnieje taka możliwość. Często się to wykorzystuje, ale założymy, że my tego nie chcemy. W tym celu

musimy dwa kontenery odłączyć od *sieci dockerowej* pozostawiając je tylko w naszej sieci. Robimy to przy pomocy polecenia:

```
docker network disconnect bridge centos-1
```

```
docker network disconnect bridge centos-2
```

Zgodnie z zastosowaniem polecenia kontenery powinny zostać odłączone. My jednak sprawdzimy to na dwa poznane sposoby. Poddajemy inspekcji najpierw kontener:

```
docker container inspect centos-1
```

```
"Networks": {  
    "siec": {  
        "IPAMConfig": {},  
        "Links": null,  
        "Aliases": [  
            "ed8ed192cdaf"  
        ],  
        "NetworkID": "033ab85e49951cc93174ba12ac48fa7080ceeff2802e5d3555c162b6395cc2e0",  
        "EndpointID": "d330b207f15b46a589258bebc528a26df39d240619d9e79f82d57c947c641cc0",  
        "Gateway": "172.18.0.1",  
        "IPAddress": "172.18.0.2",  
        "IPPrefixLen": 16,  
        "IPv6Gateway": "",  
        "GlobalIPv6Address": "",  
        "GlobalIPv6PrefixLen": 0,  
        "MacAddress": "02:42:ac:12:00:02",  
        "DriverOpts": {}  
    }  
}
```

Kontener znajduje się w tylko w jednej sieci i to naszej. Natomiast, teraz sprawdź drugi kontener lub tak jak ja sieć bridge:

```
docker network inspect bridge
```

```
"Containers": {},
```

Z przykładu wynika, że w standardowej sieci nie ma dodanego żadnego kontenera. Oznacza, to że wszystko poszło zgodnie z naszym planem.

5.10. Używamy nazwy kontenera do komunikacji

Pozostało tylko sprawdzić, czy przy użyciu tylko swoich nazw kontenery są w stanie się z sobą komunikować. Dlatego logujemy się do powłoki pierwszego kontenera:

```
docker exec -it centos-1 /bin/bash
```

I wprowadzamy polecenie w poniższy sposób:

```
ping -c 5 centos-2
```

```
PING centos-2 (172.18.0.3) 56(84) bytes of data.  
64 bytes from centos-2.siec (172.18.0.3): icmp_seq=1 ttl=64 time=0.071 ms  
64 bytes from centos-2.siec (172.18.0.3): icmp_seq=2 ttl=64 time=0.077 ms  
64 bytes from centos-2.siec (172.18.0.3): icmp_seq=3 ttl=64 time=0.075 ms  
64 bytes from centos-2.siec (172.18.0.3): icmp_seq=4 ttl=64 time=0.070 ms  
64 bytes from centos-2.siec (172.18.0.3): icmp_seq=5 ttl=64 time=0.075 ms
```

```
--- centos-2 ping statistics ---
```

```
5 packets transmitted, 5 received, 0% packet loss, time 4086ms  
rtt min/avg/max/mdev = 0.070/0.073/0.077/0.009 ms
```

Zwróć uwagę na sposób w jaki polecenie się wykonuje. Nazwa jaką się posługuje to centos-2.siec. Czyli używa nazwy kontenera oraz sieci z jakiej korzysta. Automatycznie też został pobrany adres IP kontenera. Chyba przynasz, że tak jest łatwiej. Wystarczy tylko odpowiednio nazywać kontenery z którymi chcemy współpracować.

5.11. Podsumowanie

Sieci w Dockerze stanowią bardzo interesujące rozwiązanie dające nam ogrom możliwości konfiguracyjnych co myślę, że możesz potwierdzić po przeczytaniu tego materiału. Możesz tworzyć własne sieci. Dodawać do nich dowolną ilość kontenerów. Sprawiać, że niektóre z nich są dla siebie widoczne, a niektóre nie. Oto masz pełną kontrolę nad tym, co się dzieje. Dzięki temu osiągnąć mogą izolację na bardzo wysokim poziomie.

Rozdział 6: Docker - Wolumeny

Wszystkie dane jakie zostały zapisane w trakcie korzystania z kontenera są dostępne do czasu istnienia tego kontenera. Natomiast po jego usunięciu wszystko jest czyszczone razem z nim. Często może okazać się to rozwiązaniem niepożądanym, bo na przykład potrzebujemy, aby jakieś dane zostały zachowane. Oczywiście możemy wcześniej zrobić tak zwany backup. Przekopiowujemy wszystkie interesujące nas pliki i gdzieś umieszczać na zewnątrz. Jest to *jakieś* rozwiązanie, ale zalicza się do dość czasochłonnych. W tym przypadku z pomocą przychodzi funkcja dostępna w Dockerze polegająca na wolumenach. To z ich użyciem jesteśmy w stanie zapisać dane, które dostępne będą nawet po usunięciu kontenera. W ten sposób jesteśmy w stanie wykorzystać te same pliki w kilku innych kontenerach. Jednak nim przejdziemy do takich połączeń zapoznajmy się z początkowymi informacjami dotyczącymi tego polecenia.

6.1. Tworzenie pierwszego wolumenu

Sposób tworzenia takiego magazynu do przechowywania danych nie różni się znacznie od tego, co poznaliśmy do tej pory. Zarówno i w tym wypadku wszystko zostało przemyślane w taki sposób, aby było jak najłatwiejsze do zrozumienia. W związku z tym wiąże się to z prostotą użycia, jak i z funkcjonalnością. Aby stworzyć swój pierwszy wolumen wystarczy, że posłużymy się poleceniem:

docker volume create miejsce

Jak widać w przykładzie konstrukcja nie różni się znacznie od poznanych do tej pory. Wszystkie są w podobny sposób skonstruowane, dlatego też są łatwe do zapamiętania.

6.2. Wyświetlanie listy

Zerknijmy jeszcze w jaki sposób jesteśmy w stanie wyświetlić listę wszystkich dostępnych wolumenów:

docker volume ls

DRIVER VOLUME NAME

local miejsce

Jest to bardzo zbliżone do tego, co dotychczas już poznaliśmy. Wydaje mi się, że nie muszę nic w tym wypadku tłumaczyć.

6.3. Dane z kontenera

W poprzednim przypadku stworzyliśmy miejsce, w którym mogą być składowane *jakieś* dane. Nie określiliśmy jednak jakie. Można powiedzieć, że w powyższy sposób stworzyliśmy folder, w którym będziemy składowali pliki znajdujące się w kontenerze. W celu stworzenia kontenera z zapisanymi w ten sposób danymi powinniśmy posłużyć się poleceniem:

```
docker run -it --name centos-wolumen -v miejsce:/root centos
```

lub

```
docker run -it --name alpine-wolumen -v miejsce:/root alpine
```

Większą część powinieneś już znać. Stworzyliśmy kontener noszący nazwę ***centos-wolumen***. Przypisaliśmy do niego miejsce, w którym będą zapisywane dane, znajdujące się w katalogu root kontenera. Aby to sprawdzić weszyliśmy w powłokę i stworzyliśmy pusty plik:

```
cd root/
```

```
touch plik
```

```
ls
```

```
anaconda-ks.cfg anaconda-post.log original-ks.cfg plik
```

```
exit
```

Zwróć uwagę, w katalogu znajdują się już jakieś pliki. My dodaliśmy swój pusty, który znajduje się na samym końcu przykładu. Po wykonaniu wszystkich czynności wyszyliśmy z powłoki kontenera.

6.4. Inspekcja kontenera

Stworzyliśmy kontener, którego dane z katalogu root są zapisywane poza kontenerem. Teraz pytanie brzmi, gdzie on się znajduje? Otóż w tym wypadku z pomocą przychodzi opcja polecenia ***inspect***:

```
docker volume inspect miejsce
```

```
[
```

```
{
```

```
    "CreatedAt": "2022-10-14T07:43:53+02:00",
```

```
    "Driver": "local",
```

```
    "Labels": {},
```

```
    "Mountpoint": "/var/lib/docker/volumes/miejsce/_data",
```

```
    "Name": "miejsce",
```

```
    "Options": {},
```

```
    "Scope": "local"
```

```
}
```

Zgodnie z przykładem miejsce montowania katalogu znajduje się w ***/var/lib/docker/volumes/miejsce/_data***.

Warto zapamiętać, **wszystkie wolumeny jakie stworzymy znajdują się w katalogu /var/lib/docker/volumes/**.

6.5. Weryfikacja magazynowania danych

Aby móc wejść do katalogu musimy posiadać uprawnienia administratora systemu. Dlatego dopiero po uzyskaniu takich będziesz w stanie wykonać poniższe:

```
cd /var/lib/docker/volumes/miejsce/_data/  
ls  
anaconda-ks.cfg anaconda-post.log original-ks.cfg plik
```

Zawartość tego katalogu jest identyczna jak ta, która znajdowała się w kontenerze. Oznacza to, że wszystko dobrze podłączyliśmy. Zerknijmy jeszcze w jaki sposób informacja o naszym wolumenie została zapisana w kontenerze:

```
docker container inspect centos-wolumen  
lub  
docker container inspect alpine-wolumen  
"Mounts": [  
  {  
    "Type": "volume",  
    "Name": "miejsce",  
    "Source": "/var/lib/docker/volumes/miejsce/_data",  
    "Destination": "/root",  
    "Driver": "local",  
    "Mode": "z",  
    "RW": true,  
    "Propagation": ""  
  }  
,
```

Wszystkie informacje dotyczące wolumenów otrzymasz po wykonaniu polecenia z przykładu. Widać w nich nazwę oraz miejsca podpiętego *magazynu*. Magazynu, bo tak przecież możemy nazwać wolumen.

6.6. Żywotność danych

Usuńmy teraz nasz kontener:

```
docker rm centos-wolumen
```

Gdy usuwamy kontener to i wszystkie dane z nim związane. Natomiast trochę inaczej jest w przypadku wolumenów. Są one zachowane nawet w przypadku usunięcia wszystkich kontenerów do jakich zostały podpięte. Spójrz na listę:

```
docker volume ls
```

```
DRIVER VOLUME NAME
```

```
local miejsce
```

Pomimo, że usunęliśmy kontener to wolumen pozostał nienaruszony. Sprawdźmy co znajduje się w miejscu, gdzie zapisywane są dane z kontenera:

```
ls /var/lib/docker/volumes/miejsce/_data/
```

```
anaconda-ks.cfg anaconda-post.log original-ks.cfg plik
```

Czyli wszystkie pliki pozostały na swoim miejscu. A co się stanie, jak stworzymy nowy kontener i ponownie ten sam wolumen podepniemy? Sprawdźmy:

```
docker run -it --name centos-wolumen2 -v miejsce:/root centos
```

```
lub
```

```
docker run -it --name alpine-wolumen2 -v miejsce:/root alpine
```

Specjalnie zmieniłem nazwę kontenera tak, aby nie było, żadnych wątpliwości co do tego, co zobaczysz w kolejnym przykładzie:

```
ls /root
```

```
anaconda-ks.cfg anaconda-post.log original-ks.cfg plik
```

Z przykładu wynika, że wszystkie dane zostały domontowane do nowego kontenera. Teraz jeżeli coś dodamy do tego katalogu, to zostanie również zapisane w naszym magazynie.

6.7. Współdzielenie danych

Jest to bardzo przydatna funkcja, ponieważ umożliwia magazynowanie danych w tym samym miejscu bez ich duplikacji. Stwórzmy teraz następny kontener i dołączmy do niego ponownie ten sam wolumen:

```
docker run -it --name centos-wolumen3 -v miejsce:/root centos
```

```
lub
```

```
docker run -it --name alpine-wolumen3 -v miejsce:/root alpine
```

```
cd /root/
```

```
touch plik wolumen3
```

```
ls
```

```
anaconda-ks.cfg anaconda-post.log original-ks.cfg plik wolumen3
```

Wykonaliśmy wszystkie wymienione czynności. Plus do tego stworzyliśmy w kontenerze centos-wolumen3 nowy plik. Wejdźmy teraz do naszego kontenera **centos-wolumen2** i zobaczymy, jak wygląda lista plików:

```
docker start -i centos-wolumen2
```

lub

```
docker start -i alpine-wolumen2
```

```
ls /root
```

```
anaconda-ks.cfg anaconda-post.log original-ks.cfg plik wolumen3
```

Wszystkie zmiany jakie dokonamy w katalogu dodanym, jako tak zwany magazyn, będą dostępne w każdym kontenerze, do którego go domontujemy. Jest to bardzo istotna funkcja, bo dzięki niej jesteśmy w stanie magazynować wspólne dane w jednym miejscu. Dzięki takiemu rozwiązaniu oszczędzamy bardzo cenny zasób jakim jest miejsce.

6.8. Tworzenie kontenerów z wolumenami

Nie musimy oddzielać tych dwóch czynności od siebie. Możemy wolumen stworzyć przy uruchomieniu kontenera. Przyznam, że jest to częściej spotykane rozwiązanie niż to, które prezentowałem powyżej. Niesie za sobą wygodę wykonania wszystkiego w jednym poleceniu.

Możesz się zastanawiać w takim razie po co pokazywałem wszystko to tak szczegółowo. Istotne jest, abyś zrozumiał działanie takich narzędzi jak to. Nigdy nie wiesz jaka umiejętność może się przydać. Dlatego też dobrze wiedzieć jak wykonać to w sposób *krokowy*. Wydaje mi się, że takie podejście sprawia lepsze zrozumienie tematu i ułatwia dalszą pracę czy też naukę. Zobaczmy teraz w jaki sposób połączyć te dwie możliwości:

```
docker run -itd --name centos-vul-run -v /root centos
```

lub:

```
docker run -itd --name alpine-vul-run -v /root alpine
```

Przy pomocy opcji **-v** dodajemy katalog z jakiego chcemy współdzielić pliki. Czyli wszystko tak, jak w jednym z poprzednich przykładów. Zerknijmy teraz na listę dostępnych wolumenów:

docker volume ls

```
DRIVER VOLUME NAME
```

```
local f7e75c26f39a0efc336fda3b3e6d021915d60e98a48fa7f4a47c2873b0b11f92
```

local miejsce

Został utworzony nowy magazyn. Zauważ jednak, że sposób z jakiego skorzystaliśmy zapisał nasz magazyn w bardzo nieczytelny sposób. Zerknijmy teraz do katalogu przechowywania wolumenów:

```
ls /var/lib/docker/volumes/
```

```
f7e75c26f39a0efc336fda3b3e6d021915d60e98a48fa7f4a47c2873b0b11f92 miejsce
```

Chyba widzisz w czym jest problem? Ciężko będzie komukolwiek zapamiętać taką nazwę. Dlatego zawsze w przypadku tworzenia magazynu warto go odpowiednio nazwać:

```
docker run -itd --name centos-vul-run-name -v katalog_root:/root centos
```

lub

```
docker run -itd --name alpine-vul-run-name -v katalog_root:/root alpine
```

Ponownie zajrzyjmy do katalogu magazynującego wolumeny:

```
ls /var/lib/docker/volumes/
```

```
f7e75c26f39a0efc336fda3b3e6d021915d60e98a48fa7f4a47c2873b0b11f92 katalog_root miejsce
```

Pamiętaj, że do wykonania jakichkolwiek działań na tym katalogu musisz posiadać uprawnienia użytkownika root.

W tym wypadku nazwa jest już dla nas bardzo czytelna. Nasz magazyn nie jest reprezentowany przez przypadkowy zbiór znaków. Tym razem mamy czytelną nazwę. W przypadku tworzenia wolumenów uważam tę opcję za konieczną. Będzie o wiele czytelniej jak i wygodniej zarządzać *magazynami*, które z nazwy będą przez nas rozumiane. Tego chyba nie muszę nikomu udowadniać.

6.9. Usuwanie zbędnych wolumenów

Muszę przyznać, że gdy uczyłem się Dockera pominąłem temat wolumenów. Wiedziałem tylko do czego służą i tyle. Początkowo ta wiedza mi wystarczyła.

Jednak w pewnym momencie zaczęło brakować mi miejsca na dysku. Była to wina wolumenów. Kontenerów tworzyłem jak i uruchamiałem ogromną ilość. Większość z nich posiadała wolumeny. Po jakimś czasie nazbierało się prawie 100 GB danych. Choć może się wydawać ciężkie do ogarnięcia, to uwiercie mi jak testuje się sporą ilość obrazów ,to i ilość takich magazynów rośnie. W związku tym warto dbać również o usuwanie niepotrzebnych. Zerknijmy na listę posiadanych wolumenów:

```
docker volume ls
```

```
DRIVER VOLUME NAME
```

```
local f7e75c26f39a0efc336fda3b3e6d021915d60e98a48fa7f4a47c2873b0b11f92
```

```
local katalog_root
```

```
local miejsce
```

W całym tym materiale stworzyliśmy trzy magazyny. Spróbujmy usunąć ostatnio stworzony:

```
docker volume rm katalog_root
```

```
Error response from daemon: remove katalog_root: volume is in use –  
[763070ff6496280782855061de23b7bbe2da35797e09ce058bd2b2c3b9c0f3c2]
```

Jeżeli posiadasz uruchomiony kontener, w którym masz zamontowany wolumen, to nie możesz go usunąć. Otrzymasz wtedy informacje tak jak w powyższym przykładzie. W związku z powyższym pierwsze co musisz zrobić to zatrzymać kontener:

```
docker stop centos-vul-run-name
```

Następnie usunąć kontener:

```
docker rm centos-vul-run-name
```

I dopiero teraz możesz usunąć wolumen:

```
docker volume rm katalog_root
```

Gdybyś chciał wykonać ogólne porządki i usunąć wszystkie niezamontowane wolumeny wystarczy, że posłużysz się poleceniem:

```
docker volume prune
```

WARNING! This will remove all local volumes not used by at least one container.

Are you sure you want to continue? [y/N] y

Jednak pamiętaj, że jeżeli chcesz korzystać z tego polecenia, to musisz być bardzo ostrożny by nie usunąć czegoś potrzebnego. W przypadku takiego wykorzystania zasada jest podobna jak przy pojedynczym kontenerze. Usuwane są tylko te wolumeny, które nie są w jakiś sposób przypisane do kontenera.

6.10. Podsumowanie

Wolumeny są bardzo ważnym elementem w całym Dockerze. W tym materiale chciałem, abyś poznął ogólne zasady działania oraz w jaki sposób się ich używa. Natomiast bardziej pożyteczne rzeczy będąemy robili w dalszych rozdziałach z wykorzystaniem zarówno wolumenów, jak i pozostałych poznanych dotychczas rzeczy.

Rozdział 7: Docker - Dockerfile

W poprzednich materiałach wykorzystywaliśmy polecenia do pobrania oraz uruchomienia kontenerów. W przypadku prezentowanego rozdziału poznamy plik, w którym stworzymy swój własny obraz. Zawsze przed stworzeniem takiego warto sprawdzić czy już nie istnieje. Jednak najczęściej jeżeli już taki jest, to jednak nie jest dostosowany idealnie do naszych potrzeb. Dlatego najczęściej i tak jesteśmy zmuszeni tworzyć własny.

7.1. Czym jest Dockerfile?

Z samej nazwy wywnioskować można, że jest plikiem. To w nim określamy co ma się znaleźć w naszym obrazie. Inaczej pisząc, jeżeli chcemy stworzyć swój własny nieszablonowy obraz musimy użyć do tego celu pliku Dockerfile.

Używamy tej opcji, gdy obraz standardowy nie spełnia naszych wymagań. Tak naprawdę taka sytuacja będzie miała miejsce przez większą część twojej pracy z Dockerem. W związku z powyższym warto zapoznać się z zasadami ich tworzenia.

Pomimo możliwości jakie oferuje nam wspomniany plik to warto zerknąć na gotowe rozwiązania. Takie jesteś w stanie odnaleźć na wspomnianej stronie <https://hub.docker.com/>. Drugim źródłem jest GitHub <https://github.com/>. Społeczność tego drugiego wytworzyła bardzo sporą ilość plików Dockerfile, jak i również .yaml/

7.2. Prosty obraz

Przebrnęliśmy przez bardzo sporą ilość materiału, dlatego chcę w pewnym sensie od razu przejść do pliku. Dlatego stwórzmy przy pomocy swojego ulubionego edytora tekstu taki. Wspomniany plik powinien nosić nazwę Dockerfile. W ten sposób polecenie obsługujące go nie będzie wymagało określenia nazwy.

Jak wspomniałem istnieje bardzo dużo już gotowych obrazów. W związku z tym warto spojrzeć w jaki sposób możemy wykorzystać to, co już istnieje. Dobrym przykładem będzie **PHP**, który jest używany na prawie każdym serwerze. Obraz znajduje się pod adresem https://hub.docker.com/_/php. Jeżeli wczytasz się w dokumentację, to zauważysz kilka możliwości konfiguracyjnych. Warto analizować takie przypadki nawet jeżeli nie będziemy ich na razie wykorzystywać.

Proponuję zerknąć właśnie w ten obraz ponieważ konfiguracja pliku jest bardzo prosta. Dlatego nim rozpoczniesz tworzyć własne chciałbym, abyś zerknął na właśnie te, a następnie powrócił do dalszego czytania tego rozdziału.

7.3. Nasz pierwszy Dockerfile

Jako pierwszy stworzymy bardzo prosty obraz systemu **Debian**, w którym wykonamy tylko aktualizację. Stwórz plik:

nano Dockerfile

W treści:

FROM debian

RUN apt update && apt dist-upgrade -y

Zwróć uwagę na prostotę zapisu. Przełożyć to można na język polski jako „*w systemie debian uruchom aktualizację repozytorium oraz aktualizację systemu. Natomiast przy aktualizacji na każde z zadanych pytań odpowiedz yes, czyli tak*”.

W pierwszej części pliku określamy obraz systemu bazowego z jakiego chcemy skorzystać. Każdy plik **Dockerfile** zaczyna się od tego sformułowania. W ten sposób ustawiamy obraz podstawowy, nad którym rozpoczęliśmy pracę. W tym wypadku użyliśmy systemu **Debian** w jego najnowszej wersji. Jeżeli znasz **Dockera** to wiesz, że w przypadku, gdy nie określmy wersji zawsze instalowana jest najnowsza. W tym przypadku występuje taka sama zasada. Jeżeli chcemy użyć jakiejś konkretnej wersji wprowadzamy dwukropek i wpisujemy nazwę wersji z jakiej chcemy skorzystać.

Druga część pliku zawiera polecenie **RUN**, które jak się domyślasz służy do uruchamiania poleceń w powłoce systemowej. Jest to również jedna z najczęściej wykonywanych instrukcji w pliku **Dockerfile**.

7.4. Budujemy nasz pierwszy obraz

Choć napisaliśmy tylko dwie zwięzłe instrukcje, to dzięki nim jesteśmy w stanie stworzyć nasz pierwszy obraz. Robimy to przy pomocy polecenia:

docker build -t debian:1.0 .

Sending build context to Docker daemon 2.56kB

Step 1/2 : FROM debian

...

Step 2/2 : RUN apt update && apt dist-upgrade -y

---> Running in e5cbcad81558

...

Successfully built c9432af80564

Successfully tagged debian:1.0

Otrzymasz bardzo dużo informacji, których tutaj nie umieściłem. W przykładzie skopiowałem tylko te najbardziej istotne.

Do budowania obrazów w Dockerze służy polecenie **build**, co widać na powyższym przykładzie. Następnie określiliśmy **nazwę obrazu**, jak i również jego **wersję**. W przypadku wersji dobrze jest wpisać nawet samo słowo **latest** lub tak jak w przykładzie **numer**. Obraz będzie lepiej rozpoznawalny oraz czytelniejszy dla innych, jak i dla nas. Na samym końcu znajduje się kropka (.). To ona prowadzi **Dockera** do naszego pliku **Dockerfile**. W związku z tym pamiętaj by zawsze ją tam postawić.

Zerknijmy teraz na naszą listę obrazów:

docker images

```
REPOSITORY TAG IMAGE ID CREATED SIZE
debian 1.0 fc9870d8afaa 12 seconds ago 146MB
debian latest d91720f514f7 12 days ago 124MB
```

Zauważ, że został pobrany najnowszy obraz systemu, który zainstalowaliśmy jako bazowy. Natomiast z niego został stworzony nasz o numerze **wersji 1.0**.

Möżesz posiadać na swojej liście inne obrazy z poprzednich naszych ćwiczeń. U mnie się nie pojawiają, ponieważ wszystkie zostały usunięte. Obrazy, kontenery czy też wolumeny usunąłem, ponieważ nie będę z nich już korzystał. Jeżeli również Ty ich nie potrzebujesz zrób to samo. Zawsze warto dbać o porządek oraz o ilość wykorzystanego miejsca na dysku.

7.5. tag – none

Natomiast warto wspomnieć, że jeżeli nie skorzystasz z opcji **-t** i nie określisz nazwy obrazu, a wywołasz polecenie to stanie się coś w wielu przypadkach niepożdanego. Nim jednak to zrobimy usuńmy stworzony przez nas obraz oraz obraz bazowy:

```
docker rmi fc9870d8afaa d91720f514f7
```

Teraz spróbujmy zbudować obraz bez określenia jego nazwy:

```
docker build .
```

Wynik będzie identyczny z jakim się spotkałeś przy pierwszym uruchomieniu. Natomiast teraz jeżeli wypiszesz listę obrazów to zobaczysz:

docker images

```
REPOSITORY TAG IMAGE ID CREATED SIZE
<none> <none> 77bfc01ba96f 13 seconds ago 146MB
```

```
debian latest d91720f514f7 12 days ago 124MB
```

Jedyny sposób w jaki możesz korzystać z obrazu, to przy pomocy jego ID. Przyzasz, że nie jest to w żaden sposób czytelne. W związku z tym ponownie usuńmy oba obrazy:

```
docker rmi debian 77bfc01ba96f
```

Teraz wykonaj wszystkie poznane czynności, by ponownie zbudować obraz z odpowiednią nazwą i wersją. Po poprawnej instalacji powinieneś posiadać dwa obrazy:

```
docker images
```

| REPOSITORY | TAG | IMAGE | ID | CREATED | SIZE |
|------------|-----|-------|----|---------|------|
|------------|-----|-------|----|---------|------|

```
debian 1.0 216af68377a5 7 seconds ago 146MB
```

```
debian latest d91720f514f7 12 days ago 124MB
```

Dodam tylko, że nie musisz usuwać obrazu systemu **debian:latest**. Czyli z tego, z którego budujesz swój własny. Pozwoli to zaoszczędzić czas oraz zmniejszy zużycie danych połączenia internetowego.

7.6. Wywołanie polecenia „*w locie*”

Nasz obraz jedynie co robi, to pobiera repozytorium oraz aktualizuje system. Pora doinstalować do niego jakąś aplikację. Do tego przykładu wybrałem git'a. Nasz plik **Dockerfile** powinien wyglądać następująco:

```
FROM debian
RUN apt update && apt dist-upgrade -y
RUN apt install git -y
```

Polecenia RUN, FROM są opisane na dole tego rozdziału.

Następnie utwórzmy z niego obraz:

```
docker build -t debian:1.1 .
```

Teraz nasza lista powinna powiększyć się o jeden:

```
docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
debian 1.1 0b55bd58abfc 3 minutes ago 247MB
debian 1.0 216af68377a5 24 minutes ago 146MB
debian latest d91720f514f7 12 days ago 124MB
```

Jak wejść do **cli** kontenera uczyliśmy się w jednym z poprzednich materiałów. Wiesz już, że możesz skorzystać z **exec** lub **attach**. Natomiast nie musisz się logować do powłoki kontenera w celu wywołania polecenia w sposób jaki to robiłeś dotychczas. Aby wejść do powłoki na końcu polecenia wpisywaliśmy `/bin/bash`. Teraz zastąpmy to wywołanie poleceniem, dzięki któremu wypiszemy listę pomocy dla programu git:

```
docker run --rm -it debian:1.1 git help
```

Dzięki temu sposobowi zostanie stworzony kontener (**run**) na podstawie wybranego obrazu. Następnie automatycznie w powłoce kontenera (**-it**) zostanie wydane polecenie `git help`. Lista pomocy wyświetli się, a następnie kontener zakończy swoje zadanie i zostanie usunięty (**--rm**).

Choć może w przykładzie nie jest to tak widoczne, ale prezentowana możliwość wywołania polecenia jest bardzo użyteczna.

7.7. Bardziej pożyteczny przykład

Teraz wykonajmy inny przykład. Zainstalujemy oprogramowanie nmap i wykonamy skanowanie kontenera **Metasploitable2**. Zaczniemy od pobrania oraz uruchomienia **Metasploitable2**:

```
docker run -itd --name metasploitable tleemcjr/metasploitable2:latest sh -c "/bin/services.sh && bash"
```

Zgodnie z tym co tu <https://hub.docker.com/r/tleemcjr/metasploitable2>

W przypadku **Metasploitable2** pobraliśmy obraz oraz stworzyliśmy kontener:

```
docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|---------------------------------|---------------------------|---------------|--------------|-------|----------------|
| e2a1aa467314 | tleemcjr/metasploitable2:latest | "sh -c '/bin/service...'" | 3 minutes ago | Up 3 minutes | | metasploitable |

Po pobraniu oraz uruchomieniu kontenera z **Metasploitable2** pozostało stworzenie obrazu z zainstalowanym **nmap'em**:

```
FROM debian
```

```
RUN apt update && apt dist-upgrade -y
```

```
RUN apt install nmap -y
```

Oczywiście tak jak w poprzednim przypadku musimy skorzystać z polecenia budującego:

```
docker build -t debian:nmap .
```

W związku z wprowadzonymi zmianami nasza lista obrazów powinna wyglądać następująco:

```
docker images
```

| REPOSITORY | TAG | IMAGE | ID | CREATED | SIZE |
|------------|------|-------|----|---------|------|
| debian | nmap | | | | |

```
debian nmap 4935a6ac6708 5 minutes ago 176MB  
debian 1.1 0b55bd58abfc 32 minutes ago 247MB  
debian 1.0 216af68377a5 53 minutes ago 146MB  
debian latest d91720f514f7 12 days ago 124MB  
tleemcj/r/metasploitable2 latest db90cb788ea1 4 years ago 1.51GB
```

Utwórzmy jeszcze własną sieć, w której będziemy korzystali z nazw kontenerów, a nie ich adresów IP:

```
docker network create scan
```

Sprawdzamy, czy została dodana do listy:

```
docker network ls
```

```
NETWORK ID NAME DRIVER SCOPE
```

```
2f17d5a9cd7d bridge bridge local  
e2198787e635 host host local  
e6137ead4366 none null local  
7bbd15a122fa scan bridge local
```

Teraz podłączamy nasz kontener z **Metasploitable2**:

```
docker network connect scan metasploitable
```

Pozostało stworzyć obraz oraz uruchomić polecenie skanowania:

```
docker run --rm --net scan -it debian:nmap nmap metasploitable
```

```
Starting Nmap 7.80 ( https://nmap.org ) at 2022-10-17 08:59 UTC
```

```
Nmap scan report for metasploitable (172.18.0.2)
```

```
Host is up (0.000015s latency).
```

```
rDNS record for 172.18.0.2: metasploitable.scan
```

```
Not shown: 979 closed ports
```

```
PORT STATE SERVICE
```

```
21/tcp open  ftp
```

```
22/tcp open  ssh
```

```
23/tcp open  telnet
```

```
25/tcp open  smtp
```

```
80/tcp open  http
```

```
111/tcp open rpcbind
```

```
139/tcp open netbios-ssn
```

```
445/tcp open microsoft-ds  
512/tcp open exec  
513/tcp open login  
514/tcp open shell  
1099/tcp open rmiregistry  
1524/tcp open ingreslock  
2121/tcp open ccproxy-ftp  
3306/tcp open mysql  
5432/tcp open postgresql  
5900/tcp open vnc  
6000/tcp open X11  
6667/tcp open irc  
8009/tcp open ajp13  
8180/tcp open unknown
```

MAC Address: 02:42:AC:12:00:02 (Unknown)

Nmap done: 1 IP address (1 host up) scanned in 0.21 seconds

Skanowanie zakończyło się sukcesem co potwierdza powyższy przykład. Zwróć uwagę, że w tym wypadku do sieci dodaliśmy kontener w trakcie jego tworzenia. Nie ma to znaczenia, że został on za chwilę usunięty. Wykonaliśmy podstawowe skanowanie portów dostępne w **nmap**. Wynik wyświetliśmy tak jakby polecenie zostało wywołane bez użycia dockera.

7.8. Apache2 plus PHP

W prezentowany sposób można również utworzyć obraz z serwerem **apache2** i **PHP**. Pominąłem w tym wypadku bazę danych, ponieważ lepszym rozwiązaniem połączenia tych trzech opcji jest **docker compose**. Tutaj zamierzam pokazać, że możemy stworzyć w bardzo łatwy sposób obraz z serwerem.

```
FROM debian  
  
RUN apt update && apt dist-upgrade -y  
  
RUN apt install apache2 -y  
  
RUN apt install php libapache2-mod-php -y  
  
EXPOSE 80  
  
CMD apachectl -D FOREGROUND
```

W prezentowanym pliku instalujemy podstawowe paczki. Natomiast dalej przy użyciu **EXPOSE** informujemy **dockera**, że kontener powstały z obrazu będzie nasłuchiwał na określony port. Automatycznie, jeżeli tego nie określisz będzie to port typu **TCP**. Określić typ portu możesz w poniższy sposób:

```
EXPOSE 80/TCP
```

```
EXPOSE 80/UDP
```

Pamiętaj jednak, że jest to tylko wiadomość dla dockera. Port udostępnienia musi zostać uwzględniony przy tworzeniu kontenera za pomocą opcji **-p**. Na samym końcu przy pomocy **apachectl** wywołujemy polecenie do uruchomienia w tle apache. Z powstałego pliku tworzymy obraz:

```
docker build -t debian:serwer .
```

Teraz wystarczy, stworzyć kontener przy pomocy polecenia:

```
docker run -itd --name debian-serwer --net scan -p 8080:80 debian:serwer
```

Przekierowanie ustawiłem specjalnie na port **8080** gdybyś miał coś uruchomionego na 80. Po wprowadzeniu adresu **localhost:8080** pojawi się strona **apache2**.

7.9. Poprawna forma zapisu

Prezentowana forma zapisania instrukcji nie jest właściwa z formą preferowaną przez Dockera. Znajduje się ona pod adresem https://docs.docker.com/develop/develop-images/dockerfile_best-practices/. Jeżeli ponownie wpiszesz polecenie do budowania obrazu powinieneśtrzymać następujący wynik:

```
docker build -t debian:serwer .
```

```
Sending build context to Docker daemon 2.56kB
```

```
Step 1/6 : FROM debian
```

```
---> d91720f514f7
```

```
Step 2/6 : RUN apt update && apt dist-upgrade -y
```

```
---> Using cache
```

```
---> 216af68377a5
```

```
Step 3/6 : RUN apt install apache2 -y
```

```
---> Using cache
```

```
---> dc015589f965
```

```
Step 4/6 : RUN apt install php libapache2-mod-php -y
```

```
---> Using cache
```

```
---> 29ff4df73942
```

Step 5/6 : EXPOSE 80

---> Using cache

---> b6e74c21d342

Step 6/6 : CMD apachectl -D FOREGROUND

---> Using cache

---> 1f0156cc0403

Successfully built 1f0156cc0403

Successfully tagged debian:serwer

Aby obraz powstał Docker musi wykonać **aż 6 kroków**. Zadaniem naszym jest zawsze dążyć do zmniejszenia ilości kroków dlatego **Dockerfile** powinien wyglądać następująco:

```
FROM debian
```

```
RUN apt update && apt dist-upgrade -y && apt install -y \
apache2 \
php \
libapache2-mod-php
```

```
EXPOSE 80
```

```
CMD apachectl -D FOREGROUND
```

W tym wypadku ilość kroków zmniejszy się o 2. Przyspiesza to działanie Dockera oraz wydaje się być bardziej czytelne.

7.10. Pozostałe słowa klucze Dockerfile

Pozostałe elementy jesteś w stanie odnaleźć na stronie **Dockerfile**
<https://docs.docker.com/engine/reference/builder/>.

Choć cała dokumentacja Dockera jest bardzo dobrze zorganizowana, jak i również opisana pozostałe elementy opisze również i tutaj. Zawsze warto przeglądać źródła udostępniane przez Dockera. Niektóre elementy mogą z czasem ulec zmianie lub może zostać dodana jakaś nowa funkcja. Jeżeli chcesz to przed przeczytaniem dalszego materiału zapoznaj się z informacjami zawartymi w podanym linku. Następnie wróć i przeczytaj to co ja napisałem.

Dzięki słowom kluczowym sterujesz Dockerem. W poprzednim przykładzie korzystaliśmy z wyrażenia **CMD**. Przy jego pomocy po instalacji wydaliśmy polecenie uruchamiania w tle aplikacji **apache2**.

Zapoznajmy się teraz z nim dokładniej oraz z pozostałymi wbudowanymi możliwościami jakie oferuje nam docker.

7.10.1. FROM

Każdy plik Dockerfile posiada taką instrukcję. To w tym miejscu pobierany jest system bazowy, który następnie jest modyfikowany i dostosowywany do naszych potrzeb. Można go określić jako wstęp pliku z instrukcjami, a porównać do wstępu w książkach. Jego podstawowa forma to:

```
FROM <obraz>[:tag]
```

Nie jest to w jakiś sposób skomplikowane. Podajemy nazwę obrazu i następnie nazwę lub numer wersji. Określony w przykładzie tag jest opcjonalny. Oznacza to, że możesz z niego skorzystać dla własnej wygodny, ale nie musisz. W przypadku, jeżeli nie określisz wersji to docker automatycznie umieści tam słowo **latest**. W tym przypadku zostanie pobrany najnowszy obraz i na nim wykonane wszystkie pozostałe czynności.

Jedyną opcją jaka może poprzedzić wystąpienie **FROM** jest argument **ARG**. W najprostszy sposób **ARG** można określić jako zmienną, którą następnie używa się w pliku. Myślę, że poniższy przykład wszystko wyjaśni:

```
ARG VERSION=latest
```

```
FROM debian:{VERSION}
```

Jako wartość zmiennej **VERSION**, możesz określić zgodnie ze swoimi potrzebami. Po wybranym obrazie systemu **Debian** zostanie określona wersja jako najnowsza. Natomiast bardzo istotne jest to, że zmienna **VERSION** nie jest dostępna w żadnym innym miejscu poniżej **FROM**. Jeżeli chcemy, żeby była musimy ją ponownie zdeklarować w poniższy sposób:

```
ARG VERSION=latest
```

```
FROM debian:{VERSION}
```

```
ARG VERSION
```

Etap pokazany w przykładach jest określany jako etap budowania.

7.10.2. RUN

Opisać najprościej można jako instrukcje wykonującą polecenia. Jest ona najczęściej spotykana w plikach Dockerfile. Prawie tak samo jak wcześniej opisywane **FROM**. Pamiętaj jednak, że korzystanie z tej dyrektywy wiąże się również z tak zwymi dobrymi praktykami. Aby plik był czytelny musisz trzymać się kilku prostych zasad. Wspominałem o tym, gdy tworzyliśmy obraz z **apache2**. Dlatego też

nie chcę się powtarzać. Jednak pamiętaj, że polecenia służące aktualizacji oraz instalacji należy umieszczać w jednej linii. Natomiast nazwy programów w oddzielnych wierszach stosując \. Poprawi to znacznie czytelność pliku, jak i również pozwoli na szybszą analizę instalowanego oprogramowania. Natomiast inną przyczyną takiego zapisu jest buforowanie warstw. Zmniejszyliśmy ilość kroków jakie musi wykonać docker aby stworzyć obraz.

RUN posiada dwie możliwości wywołania. Pierwsza z nich wygląda następująco:

RUN <polecenie>

W powyższym przypadku wywołana zostanie automatycznie powłoka **bash** i wykonane polecenie.

Drugi przypadek wygląda następująco:

```
RUN ["powłoka","parametr 1","parametr 2"]
```

W tym przypadku nim zostanie wydane polecenie musi zostać wybrana powłoka.

7.10.3. CMD i ENTRYPPOINT

CMD jest to instrukcja, dzięki której określamy obrazowi jaki komponent ma być uruchomiony przy tworzeniu kontenera. Jego podstawową konstrukcję poznaliśmy w przykładzie dotyczącym uruchomienia serwera.

Drugim wystąpieniem było, gdy korzystaliśmy z obrazu Metasploitable2:

```
docker run -itd --name metasploitable tleemcj/metasploitable2:latest sh -c "/bin/services.sh && bash"
```

Możliwe, że zwróciłeś uwagę na końcową część tworzenia kontenera. W ten sposób jesteśmy w stanie wywołać polecenie bez użycia pliku **Dockerfile**. Można by powiedzieć, że automatycznie przed nim znajduje się **CMD**.

Wałą informacją jest to, że w pliku Dockerfile może zostać użyte tylko jedno wywołanie **CMD**. Jeżeli zapiszesz kilka, to wywołane zostanie tylko to ostatnie.

Natomiast możesz się zastanawiać po co jest **Entrypoint** czyli coś co możemy na język polski przełożyć jako **punkt wejścia**. Wyjaśnić to można bardzo prosto na przykładzie gdy korzystaliśmy z gita. Nim cokolwiek napiszę zerknij na poniższy przykład:

```
ENTRYPOINT ["git"]
```

```
CMD ["help"]
```

Oznacza to, że do polecenia git zostanie dodana opcja help. Sprawdźmy to modyfikując niedawny **Dockerfile**:

```
FROM debian
```

```
RUN apt update && apt dist-upgrade -y && apt install -y \
git
```

```
ENTRYPOINT ["git"]
```

```
CMD ["help"]
```

Po zbudowaniu obrazu oraz uruchomieniu wyświetli się lista pomocy programu **git**.

W Dockerfile instrukcja ENTRYPOINT określa polecenie, które ma zostać wykonane podczas uruchamiania kontenera. To jest główne polecenie, które zostanie wykonane podczas uruchamiania kontenera, a wszelkie dodatkowe argumenty zostaną przekazane do polecenia znajdującego się w ENTRYPOINT. Korzystanie z niego to dobra praktyka podczas tworzenia obrazów Dockera, ponieważ określa domyślne zachowania kontenera. Kiedy ktoś uruchamia kontener, może z łatwością zobaczyć domyślne polecenie, które zostanie wykonane, co ułatwia korzystanie i jego zrozumienie. Jeśli kontener jest uruchamiany na komputerze lub serwerze testowym czy też w środowisku produkcyjnym, domyślne polecenie będzie zawsze takie samo. Inną zaletą korzystania z ENTRYPOINT jest to, że określając polecenie domyślne, możesz uniemożliwić użytkownikom przypadkowe uruchomienie niezamierzonych poleceń w kontenerze. Może to pomóc w zapobieganiu naruszeniom bezpieczeństwa i zapewnieniu, że kontener jest używany zgodnie z przeznaczeniem do jakiego został stworzony.

7.10.4. COPY i ADD

Zarówno **COPY** i **ADD** służą do dodawania plików oraz katalogów do obrazów dockerowych. Składnia **COPY** wygląda następująco:

```
COPY <źródło> <cel>
```

Natomiast składnia **ADD**:

```
ADD <źródło> <cel>
```

Prościej mogłem napisać, że ich składnie wyglądają identycznie. Różnią się tylko pierwszym członem polecenia, gdzie określamy którą opcję chcemy wybrać. Jeżeli obie wykonują to samo, to po co zostały wymyślone dwie funkcje wykonujące to samo? Docker zaleca używanie opcji **COPY**, natomiast każe omijać **ADD**. Taka sytuacja ma miejsce, ponieważ istnieje pewna różnica pomiędzy nimi.

Przyczyną takiego podejścia jest dodatkowa możliwość opcji **ADD**. Dzięki niej możesz pobierać pliki przy pomocy adresów URL. W związku z tym czyni go bardziej niebezpiecznym w użyciu. Może również rozpakować skompresowane pliki, o ile rozpozna format.

Opcja **COPY** została wprowadzona w późniejszym czasie by zapewnić większe bezpieczeństwo w stosunku do podstawowych czynności jak kopiowanie. W przypadku jeżeli chcesz pobrać pliki do twojego obrazu lepiej jest skorzystać z polecenia **RUN** oraz na przykład oprogramowania **curl**.

7.10.5. EXPOSE

Instrukcja informująca o portach, na których nastuchuje aplikacja. Ważne jest, że pomimo umieszczenia tej informacji w pliku to nadal należy przy tworzeniu kontenera skorzystać z opcji **-p**. **Zastosowanie EXPOSE w pliku Dockerfile jest tylko i wyłącznie informacyjne.** Natomiast do udostępnienia portu służy opcja **-p** przy tworzeniu kontenera.

7.10.6. ENV

Dzięki tej dyrektywie definiujemy zmienne środowiskowe. Jest to dość ciekawie rozwiązane. Po pierwsze jeżeli zdefiniujesz taką zmienną środowiskową możesz użyć ją w danym pliku **Dockerfile**. Natomiast, gdy tworzysz kontener to wszystkie wskazane zmienne zostają automatycznie ustawione na takie jakie określiłeś w pliku.

7.10.7. VOLUME

O wolumenach poświęciłem oddzielny rozdział. Nie różnią się opisowo w żaden sposób od tego co już poznaliśmy. Dlatego w skrócie korzystając z wolumenów w Dockerfile określamy dockerowi które katalogi mają być przechowywane na głównym hoście. Oznacza to, że wszystkie dane z tych katalogów nie zostaną usunięte razem z kontenerem. Ale o tym wszystkim już wiesz.

7.10.8. USER

Nie musisz uruchamiać poleceń jako użytkownik **root**. Dzięki **USER** ustawisz konto użytkownika lub grupę jakie mają wykonać wszystkie polecenia wywołane w **RUN**, **CMD** i **ENDPOINT**.

USER mietek

Lub

USER grupa

Określony użytkownik lub grupa jest używany do instrukcji **RUN** oraz do wykonywania komend zawartych w **ENTRYPOINT** i **CMD**.

7.10.9. WORKDIR

Jest to sposób na zdefiniowanie katalogu roboczego.

WORKDIR /home/mietek

To on będzie wykorzystywany przy dalszych instrukcjach **RUN**, **CMD**, **ENTRYPOINT**, **COPY** i **ADD**. Jeżeli folder nie istnieje to **Docker** utworzy go sam.

7.11. Podsumowanie

Dockerfile jeżeli się dokładnie przyjrzeć nie jest aż tak bardzo rozbudowany, a co ważniejsze nie jest ciężki do zrozumienia. Po zapoznaniu się z kilkoma podstawowymi zagadnieniami jesteśmy w stanie tworzyć własne obrazy. Jednak nie zwalnia to nas z obowiązku znajomości podstawowych poleceń Linuksa. Jednak łącząc ze sobą wspomniane dwie umiejętności jesteśmy w stanie tworzyć obrazy zgodnie z naszymi upodobaniami. Co daje nam to naprawdę ogromne możliwości.

Rozdział 8: Funkcje oficjalnych obrazów na bazie MySQL

Niezależnie od tego czym się zajmujesz oraz do czego będziesz używał Dockera, po jakimś czasie spotkasz się z potrzebą wykorzystania bazy danych. Już od bardzo dawna wszelkiego rodzaju dane magazynuje się właśnie w takich bazach. Nie ma tu znaczenia czy jest to witryna internetowa czy też program do faktur. Baza danych stała się środkiem powszechnie stosowanym w każdej sferze IT. Dlatego uważam, że jest warta bardziej szczegółowego zapoznania się z nią.

Jedną z najczęściej spotykanych jest **MySQL**, dlatego w tym materiale omówimy właśnie nią. Szczegóły dotyczące obrazu znajdują się pod adresem https://hub.docker.com/_/mysql. Szczegółami zawartymi na stronie teraz się zajmiemy. Oczywiście nie istnieje jedynie jeden typ bazy. Tak jak w środowiskach, w których nie korzysta się z dockera tak samo i tutaj mamy kilka do wyboru. Jeżeli chcesz to poniżej podaję linki do tych najbardziej popularnych:

Postgres: https://hub.docker.com/_/postgres

MariaDB: https://hub.docker.com/_/mariadb

Niemniej jednak celem tego materiału jest zwrócenie twojej uwagi na informacje jakie możesz uzyskać na stronie obrazu. Oczywiście dzięki temu poznamy bardziej szczegółowo **MySQL**, ale z takimi informacjami spotkać się możesz nieraz w innych obrazach. Warto jest zawsze prześledzić co napisał autor. Szczególnie, że to właśnie tu uzyskasz informacje w jaki sposób możesz użyć dostępnych opcji by właściwie skonfigurować kontener. Czy też nawet uzyskać informacje o dostępnych wersjach.

8.1. Wersje oraz Dockerfile

Pomijając początek strony, gdzie znajdują się linki kto utrzymuje obrazy oraz gdzie znaleźć pomoc pierwsze co pojawia się z istotnych informacji to dostępne wersje obrazu. Co jeszcze bardziej ciekawe, jeżeli klikniesz w którykolwiek link zostaniesz przeniesiony do miejsca, gdzie znajduje się plik **Dockerfile**. Pamiętaj, że jeżeli chcesz nauczyć się pisać i korzystać z takich plików istotne jest, abyś przeglądał takie.

Pliki czy też może bardziej wersje plików są podzielone od najnowszych do najstarszych. Jeżeli klikniesz w którykolwiek odnośnik pojawi się plik **Dockerfile** z gotową konfiguracją do wykorzystania.

Jak wspominałem w poprzednim materiale plik rozpoczyna się od pobrania obrazu, na którym będziemy bazowali. W przypadku wersji 8.0 wykorzystywany jest system **oraclelinux:8-slim**. Dystrybucji Linuksa jest bardzo dużo, dlatego pominę opis jakiejkolwiek. Natomiast bardzo istotna jest sama końcówka. Niektóre z nich posiadają swoje wersje z określonym przeznaczeniem. Tak jak ta z

której będziemy korzystali. Skąd o tym wiem? Otóż zatrzałem do dokumentacji danego obrazu znajdującej się pod adresem https://hub.docker.com/_/oraclelinux. Znajduje się tam wzmianka o tym:

„The oraclelinux:8-slim variant is intended primarily to provide "just enough user space" for statically compiled binaries or microservices. Use of the 8-slim variant is discouraged for general purposes, due to the inclusion of microdnf in place of dnf and significantly reduced locale data.”

Czyli wykorzystuje się go do celów takich jak nasz, zainstalowania bazy danych. Jest to bardzo dobry przykład, dlaczego zawsze przed wybraniem systemu bazowego warto o nim poczytać.

8.2. Analiza wykonywanych czynności w pliku

8.2.1. RUN

Jeżeli poddasz analizie plik Dockerfile znajdujący się pod adresem <https://github.com/docker-library/mysql/blob/e0d43b2a29867c5b7d5c01a8fea30a086861df2b/8.0/Dockerfile.oracle>

to w **RUN** odnajdziesz polecenia oraz instrukcje Linuksowe.

Znajomość Dockera nie zwalnia ze znajomości Linuksa. Każdy plik Dockerfile bazy danych posiada RUN z poleceniami Linux, dla przykładu zerknij na ten z linku.

Można by powiedzieć, że jeżeli chodzi o etap konfiguracji oraz instalacji wykonujesz to wszystko co byś zrobił bez użycia Dockera. Jeżeli przeanalizujesz cały plik możesz zwrócić uwagę, że w nim znajduje się wiele instrukcji typu **RUN**. Natomiast **CMD** jest użyte tylko raz, na samym końcu. Jedną z przyczyn już znasz. **CMD może być w Dockerfile użyte tylko raz.** Natomiast drugą jest to, że **RUN działa tylko w chwili budowania obrazu.** Natomiast **CMD** po jego zbudowaniu. Dlatego w obrazie, który tworzyliśmy w poprzednim materiale na końcu skorzystałem z **CMD**.

8.2.2. ENV

Warto też zwrócić uwagę w jaki sposób są ustawiane, a następnie używane wartości **ENV**. Choć o tym wspominałem dobrze jest jeszcze zobaczyć to na własne oczy. Działają tak jak zmienne w programowaniu. W taki sposób określamy definicję:

```
ENV MYSQL_SHELL_VERSION 8.0.31-1.el8
```

A w taki z niej korzystamy:

```
RUN set -eux; \
microdnf install -y "mysql-shell-$MYSQL_SHELL_VERSION"; \
microdnf clean all; \
\
mysqlsh --version
```

Naszą zmienną wyróżniłem w przykładzie. W taki sposób jesteś w stanie na początku zdefiniować, a następnie użyć. Robi się to dla czytelności oraz aby używać jednej deklaracji w kilku miejscach. Pozwala to zaoszczędzić pamięć oraz przyspiesza działanie w tym wypadku budowania obrazu.

8.2.3. Wolumen

Do następnego elementu jest miejsce przechowywania baz danych. **MySQL** standardowo magazynuje wszystkie pliki w **/var/lib/mysql**. W związku z tym w obrazie montujemy taki wolumen na zewnątrz. W ten sposób będziemy w stanie w przypadku awarii kontenera stworzyć kolejny i dopiąć do niego pliki z danymi.

8.2.4. COPY

Następnie budowa bazy danych wymaga wykorzystania dodatkowego skryptu. Znajduje się on również w repozytorium, które przeglądamy. Musimy je dołączyć do naszego obrazu i robimy to w następujący sposób:

```
COPY docker-entrypoint.sh /usr/local/bin/
```

Pamiętaj jednak, że niezależnie czy ten, czy też inny plik, który chcesz przekopiować musi znaleźć się pod podaną ścieżką jaką wprowadzasz na początku polecenia. W tym wypadku plik znajduje się w tym samym katalogu co plik Dockerfile.

8.2.5. ENTRYPOINT

Następnie do skopowanego pliku jest tworzone dowiązanie i uruchamiany jest skrypt:

```
ENTRYPOINT ["docker-entrypoint.sh"]
```

Pewnie wiesz, ale pozwól sobie dopowiedzieć. Nie jest podana bezpośrednia ścieżka ze względu na zastosowane dowiązanie do pliku.

8.2.6. EXPOSE i CMD

W przedostatniej linii informujemy Dockera na jakich portach nasłuchiwa aplikacja, a w ostatnim uruchamiamy polecenie wykonujące konfigurację mysql'a.

8.3. Inny plik

Jeżeli wybrany system nie jest dla Ciebie znany w przypadku **MySQL** możesz wybrać jeszcze wersję opartą o dystrybucję **Debian**. Zawsze warto zerknąć na proponowane rozwiązania. Oczywiście na wzór powyższego zawsze możesz stworzyć swój własny obraz. Lub jeżeli poza dockerem instalację przeprowadzasz w inny sposób możesz spróbować ją wdrożyć. Wersja z systemem **Debian** znajduje się pod adresem

<https://github.com/docker-library/mysql/tree/80c475648969a83e52802e8eb2ad90519f882421/8.0>

Zawsze warto analizować takie pliki. Nie jesteśmy na samym początku powstawania Dockera, gdzie informacji było bardzo mało, a samych przykładów jeszcze mniej. Znając podstawowe zagadnienia w tym i poprzednich materiałach pozostała Ci tylko praktyka.

8.4. Zmienne środowiskowe

Nie jest to jeszcze jednak koniec. Chciałbym, żebyś wrócił do strony opisującej obraz **MySQL**. W ramach przypomnienia https://hub.docker.com/_/mysql. Napotkasz się na różne sposoby uruchomienia obrazu oraz na wykorzystanie go przy użyciu **docker compose**. Warto być to przejrzał. Powinieneś w większym stopniu je rozumieć.

Natomiast dla mnie teraz najważniejsze jest to, co znajduje się w **Environment Variables**, czyli w miejscu poświęconym **zmiennym środowiskowym**.

Jest tam wypisane kilkanaście ustawień. Z większości z nich możemy, ale nie musimy korzystać. Nie chcę zostawiać Ciebie z tym samego, dlatego omówmy sobie każde z nich.

8.4.1. MYSQL_ROOT_PASSWORD

Jest zmienną obowiązkową i w niej określamy hasło dla użytkownika **root**.

8.4.2. MYSQL_DATABASE

Jest to zmienna opcjonalna. Dzięki niej nadajemy nazwę naszej bazy danych. Zostanie utworzona w czasie uruchomienia obrazu. Istotne jest, to że jeżeli utworzymy użytkownika i hasło to otrzyma on dostęp do tej bazy danych.

8.4.3. MYSQL_USER, MYSQL_PASSWORD

Obie zmienne są opcjonalne. Zapisane w jednym miejscu, ponieważ aby utworzyć konto użytkownika obie wartości muszą zostać uzupełnione. Jak wspomniałem w punkcie o tworzeniu bazy. Automatycznie uzyskują uprawnienia superużytkownika dla bazy utworzonej przy pomocy **MYSQL_DATABASE**.

8.4.4. MYSQL_ALLOW_EMPTY_PASSWORD

Zmienna opcjonalna i niezalecana w większości przypadków. Ustawiasz ją jedynie w celach na przykład wykonania jakiś szybkich testów funkcjonowania. Jako wartość przyjmuje yes. Pamiętaj, że korzystając z niej każdy może uzyskać dostęp do konta nawet root.

8.4.5. MYSQL_RANDOM_ROOT_PASSWORD

Kolejna zmienna opcjonalna. Ustawiasz tak jak w poprzednim przypadku na wartość yes. Wygenerowane zostanie losowe hasło dla użytkownika root. Po wykonaniu wszystkich niezbędnych czynności zostanie wyświetcone na ekranie.

8.4.6. MYSQL_ONETIME_PASSWORD

Jest to ustawienie przeznaczone tylko dla konta root. Wymusza zmianę hasła przy pierwszym logowaniu. Każda wartość wywołuje tą funkcję. Dostępna jest od wersji MySQL 5.6 +.

8.4.7. MYSQL_INITDB_SKIP_TZINFO

Wyłącza standardowe ładowanie sfery czasowej. Korzystamy z niej wprowadzając jakąkolwiek wartość.

8.5. Praktyczne użycie zmiennych środowiskowych

Teorię mamy za sobą teraz chciałbym podejść do tego bardziej praktycznie. Dlatego zaczniemy od pobrania obrazu i następnie stworzymy z niego kontener z odpowiednimi ustawieniami. Obraz pobierzemy oraz uruchomimy w znany nam sposób prezentowany poniżej:

```
docker run --name mysql -e MYSQL_ROOT_PASSWORD=haslo -d mysql
```

Jest to jedno z najprostszych wywołań, jakie można wykonać. Tworzymy hasło tylko dla konta root. Każde z ustawień zmiennych środowiskowych musi być poprzedzone opcją **-e** po której występuje zmienna, którą ustawiamy. Opcja **-d** przeze mnie najczęściej stosowana jest na samym początku. W wielu miejscach jednak możesz spotkać się z nią na samym końcu. Oprócz wymienionych możliwości nie ma tutaj nic nowego. Możesz logować się do powłoki tak jak to robiliśmy dotychczas oraz wykonywać wszystkie pozostałe czynności jakie poznaleś. Natomiast ja osobiste preferuję w prezentowanym przypadku tworzyć bazę danych w taki sposób:

```
docker run -d --name mysql-moja --net serwer -e MYSQL_ROOT_PASSWORD=haslo -e  
MYSQL_DATABASE=nazwa_bazy -e MYSQL_USER=uzytkownik -e MYSQL_PASSWORD=haslo_uzytkownika  
mysql
```

W tym wypadku dodajemy kontener od razu do naszej sieci. Jak zapewne wiesz, jest on wtedy wykluczony z sieci generowanej przez Dockera. Według mnie konfiguracja, którą tutaj zaprezentowałem jest to minimum z jakiego powinieneś zawsze korzystać. Nigdy nie twórz baz na koncie użytkownika root. Innym, dobrym pomysłem byłoby jeszcze:

```
docker run -it --name mysql-random-pss --net serwer -e MYSQL_RANDOM_ROOT_PASSWORD=y -e  
MYSQL_DATABASE=nazwa_bazy -e MYSQL_USER=uzytkownik -e MYSQL_PASSWORD=haslo_uzytkownika  
mysql
```

Natomiast, jeżeli chcesz stworzyć środowisko testowe to wystarczy rozwiążanie prezentowane na samym początku lub ustawiasz opcję pozwalającą na konto root bez hasła.

8.6. Podsumowanie

Celem tego materiału było wskazanie w jaki sposób powstają obrazy z wykorzystaniem Dockerfile. Następnie z nich korzystać. Istnieje bardzo dużo obrazów posiadających moc zmiennych środowiskowych. **MySQL** nie należy do takich. Jeżeli zerkniesz na przykład na obraz systemu monitorowania **Zabbix** znajdujący się pod adresem <https://hub.docker.com/r/zabbix/zabbix-web-nginx-mysql>, to w opisie odnajdziesz bardzo dużo możliwości konfiguracyjnych.

Rozdział 9: Docker – Docker Compose

Bardzo pożytecznym elementem w Dockerze jest możliwość sprawienia, że kilka kontenerów Dockera współpracuje razem z sobą. Najlepszym tego przykładem jest serwer stron internetowych. W tym wypadku potrzebujemy czterech elementów **apache2**, **PHP**, **MySQL** i **phpmyadmin**. Choć ostatnia pozycja jest opcjonalna to jednak wygodne jest aby zarządzać naszymi bazami danych w sposób wygodny, jaki oferuje nam instalowane oprogramowanie. Bez dalszego słowa wstępnie przejdźmy do stworzenia takich plików.

9.1. Tworzymy plik

Pierwszym co należy zrobić, to stworzyć odpowiedni folder i w nim umieścić pliki. Dlatego tworzymy folder naszego projektu. Możemy nazwać go dowolnie, ja swój nazwałem mój. Następnie w nim tworzymy przy pomocy naszego ulubionego edytora tekstu plik o nazwie **compose.yaml** lub **compose.yml**. Obie nazwy są akceptowalne i mogą być stosowane zamiennie.

Przy analizowaniu innych projektów na pewno spotkasz się z sytuacją, w której pliki będą nosić nazwę **docker-compose.yaml** lub **docker-compose.yml**. Jest to również właściwe, ale od jakiegoś czasu zalecanymi nazwami są te którymi ja się posługuję. W związku z tym będziemy korzystali z nowego nazewnictwa.

9.2. Notacja plików

Jeżeli chodzi o system zapisu są to pliki z rozszerzeniem yaml. Notacja w takich plikach polega na stosowaniu wcięć pomiędzy poszczególnymi liniami kodu. Można to porównać do notacji stosowanej w języku **Python**. Bardzo prosto można zobrazować to na przykładzie:

Akapit:

 przynależy do akapitu

Nowy Akapit

 przynależy do nowego akapitu

 przynależy do przynależności do nowego akapitu :)

Po prostu stosujemy wcięcia. Ja pomiędzy etykietami używam najczęściej po dwie spacje, tak jak w powyższym przykładzie. Istotne jest, aby stosować taką samą zasadę w całym pliku. Po tym **docker compose** będzie wiedział, który element, do którego należy.

9.3. Instalacja Docker Compose

Instalacja:

```
apt install docker-compose
```

9.3.1. Dwie formy compose

W zależności w jaki sposób zainstalowaliśmy **docker compose**, możemy stosować dwa sposoby uruchomienia programu. Otóż, jeżeli zainstalowaliśmy z repozytorium systemowym **Dockera**, to najprawdopodobniej, aby wywołać uruchomienie musimy wpisać w wierszu polecen **docker-compose**. W przypadku jeżeli pobraliśmy repozytorium Dockera, to korzystać będziemy z **docker compose**, bez użycia myślnika. Pomiędzy tymi dwoma sposobami różnica jest jedynie w nazwie. Wszelkie funkcje są identyczne. W tym materiale będę stosował nazwę drugą, czyli **docker compose**. Mam pobrane repozytorium dockera ponieważ chcę korzystać z najnowszej wersji.

```
docker-compose version
```

```
Docker Compose version v2.12.0
```

Jeżeli masz jakieś wątpliwości lub problemy z docker compose, pełną instalację zarówno samego Dockera, jak i compose przeprowadziłem w materiale o instalacji. Dlatego zajrzyj tam.

9.3.2. Wersja pliku

Pierwszym elementem jaki należy określić, to rodzaj wersji z jakiej korzystamy. I tu przyznam, że na samym początku możemy mieć pewien problem. Nie chodzi tutaj o rodzaj wersji samego Dockera. Jak i również o docker compose. Tylko o wersję samego silnika Dockera na jakim plik ma się opierać. Czyli prościej pisząc musimy określić wersję pliku yml dostosowaną do wersji dockera jakiego użyjemy do wykonania instrukcji w pliku. Pełna lista znajduje się pod adresem:

<https://docs.docker.com/compose/compose-file/compose-versioning/>

Jeżeli zatrzałeś na powyższą stronę, znajduje się na niej tabela z rozpisanymi wersjami plików do konkretnych wersji dockera. Moja wersja dockera to:

```
docker -v
```

```
Docker version 20.10.20, build 9fdeb9c
```

W związku z tabelą do mojej wersji dockera powiniem używać wersji pliku 3.8. Pamiętaj jednak, że tutaj chodzi o wersję silnika **dockera**, a nie **compose**.

9.3.3. Nagłówek pliku

W tej sytuacji nasz nagłówek pliku powinien wyglądać w następujący sposób:

```
version: '3.8'
```

Co bardzo ważne, sama wersja musi znajdować się w cudzysłówie pojedynczym. Tak jak prezentuje przykład. Jeżeli zapomnisz o tym i użyjesz podwójnego możesz otrzymać błąd.

9.3.4. Elementy składowe pliku

Wszystkie elementy z których chcemy stworzyć nasz serwer będą znajdowały się pod:

```
version: '3.8'
```

```
services:
```

To pod nazwą **services** będziemy określali wszystkie obrazy z jakich zechcemy stworzyć nasz serwer. Ważne jest, aby zarówno wersja, jak i **services** były na tym samym poziomie. Czyli nie używamy w tym przypadku żadnych odstępów.

9.3.5. Apache2

Pierwszym obrazem jaki będziemy chcieli skonfigurować jest to **apache2**. To on jest tak naprawdę podwaliną całego serwera. To do niego dodajemy takie elementy składowe jak obsługa plików PHP. W związku z tym zajmiemy się nim jako pierwszym.

```
version: '3.8'
```

```
services:
```

```
apache:
```

```
  container_name: apache
```

```
  build: ./apache
```

Nim przejdziemy do szczegółów chciałbym ponownie zwrócić szczególną uwagę na wcięcia zastosowane w pliku. Bez nich Docker nie zadziała i nie zbuduje naszego serwera.

Pod services stworzyliśmy nazwę usługi z jakiej skorzystamy. Lepiej można tę nazwę określić jako etykietę. Jeżeli chciałbyś mógłbyś napisać w tym miejscu *zupa_z_bananów*, a następnie przystąpić do konfiguracji **apache2**. Nie jest to jednak zalecane ponieważ to co nazywam etykietą powinno naprowadzić na funkcję jaką ma pełnić. W następnej linii określamy nazwę kontenera:

```
  container_name: apache
```

W ostatniej z przykładu miejsce do pliku który zbuduje obraz naszego serwera. Pliki do budowy obrazu poznaliśmy w poprzednich materiałach. Są to oczywiście **Dockerfile**. My stworzymy za chwilę taki oraz dodatkowy plik konfiguracyjny apache. Jednak nim to nastąpi chciałbym powiedzieć jeszcze o innym sposobie. Obraz, który stosowany jest do instalacji wspomnianego **apache2** to **httpd** znajdujący się pod adresem https://hub.docker.com/_/httpd. W tym wypadku tworzymy swój własny obraz. Natomiast nie zawsze będziemy potrzebowali tego dokonać. Możemy pobrać obraz bez jego konfiguracji. Wystarczy, że zamiast opcji **build** zastosowałbyś **image: httpd**. Takie rozwiązanie zobaczymy przy obrazie **mysql**, za jakiś czas.

W związku z tym, że zdecydowaliśmy się na budowanie konfiguracji musimy stworzyć dwa pliki. Zgodnie z instrukcjami zapisanymi w pliku **compose** muszą się one znaleźć w folderze **apache**. W tym samym miejscu co nasz plik **compose.yaml**.

Po stworzeniu katalogu przy pomocy swojego ulubionego edytora tekstu tworzymy plik o nazwie **grupaadm.apache.conf** i umieszczamy w nim treść:

```
LoadModule deflate_module /usr/local/apache2/modules/mod_deflate.so  
LoadModule proxy_module /usr/local/apache2/modules/mod_proxy.so  
LoadModule proxy_fcgi_module /usr/local/apache2/modules/mod_proxy_fcgi.so
```

```
<VirtualHost *:80>  
    ProxyPassMatch ^/(.*\.php(/.*))$ fcgi://php:9000/usr/local/apache2/htdocs/$1  
  
    DocumentRoot /usr/local/apache2/htdocs  
  
    <Directory /usr/local/apache2/htdocs>  
        Options -Indexes +FollowSymLinks  
        DirectoryIndex index.php  
        AllowOverride All  
        Require all granted  
    </Directory>  
</VirtualHost>
```

To dzięki temu plikowy ładowane są odpowiednie moduły oraz wykonuje się podstawowa konfiguracja serwera. Opis zawartości wykracza poza zakres szkolenia. Jednak myślę, że wiesz co w nim się dzieje lub chociaż domyślasz się.

Nazwa powyższego pliku tak naprawdę nie ma znaczenia. Jeżeli nazwiesz go w inny sposób to w kolejnym kroku musisz wprowadzić odpowiednią nazwę. Natomiast sam plik nic nam nie daje. Tak naprawdę **Docker** z niego jeszcze nie korzysta. Aby jednak użył go musimy stworzyć plik **Dockerfile** i w nim określić jak ma zostać użyty. Dlatego teraz tworzymy wspomniany plik i w nim umieszczamy zawartość:

```
FROM httpd  
  
COPY grupaadm.apache.conf /usr/local/apache2/conf/grupaadm.apache.conf  
  
RUN echo "Include /usr/local/apache2/conf/grupaadm.apache.conf" \  
    >> /usr/local/apache2/conf/httpd.conf
```

Tak jak pisałem przed chwilą, jeżeli zastosowałeś inną nazwę pliku, to tutaj musisz ją wprowadzić. Nasz **Dockerfile** pobiera obraz **httpd** odpowiedzialny za **apache2**, a następnie kopiuje do niego naszą konfigurację. Na samym końcu dodaje jej zawartość do głównego pliku konfiguracyjnego serwera. Pliki, które pobiorą obraz oraz wykonają podstawową konfigurację serwera mamy za sobą. Pozostałe wartości jakie musimy ustawić w naszym pliku **compose.yaml** są to:

```
version: '3.8'

services:
  apache:
    container_name: apache
    build: ./apache
    links:
      - php
    ports:
      - "80:80"
    volumes:
      - ./src:/usr/local/apache2/htdocs
```

O większości z tych rzeczy pisaliśmy w poprzednich materiałach. Teraz są one trochę inaczej zapisane. W przypadku etykiety **links**. Przyznam, że tutaj wyszedłem trochę do przodu. Służy ona do połączenia właściwości z innego obrazu. Innym słowem nasz serwer będzie posiadał możliwość obsługi PHP.

Następnie określamy port, na którym serwer będzie nadawał. Dzięki czemu po wprowadzeniu adresu **localhost** otrzymamy dostęp do niego. Jeżeli port 80 jest u Ciebie zajęty pierwsze wystąpienie portu 80 zmień na taki, na jakim ma apache2 być dostępne.

Na samym końcu montujemy wolumen. Zwróć uwagę, że przy nazwie znajduje się kropka i następnie podana jest nazwa. W ten sposób montujemy katalog w którym jeżeli umieścimy pliki **html** i **php** zostaną one obsłużone przez serwer. Ten punkt montowania pozwala na umieszczanie plików z zewnątrz. Zajmiemy się jeszcze tym w dalszej części materiału. **Apache2** mamy ustalony. Teraz zajmijmy się **PHP**.

9.3.6. PHP

Bardzo szczegółowo opisałem jak wykonać wszystko przy **apache2**. Nie chcę Cię zanudzać dlatego teraz pokazać chcę cały kod pliku **compose.yaml** dotyczący **PHP**:

```
php:
  container_name: php
  build: ./php
```

```
links:  
- mysql  
ports:  
- "9000:9000"  
volumes:  
- ./src:/usr/local/apache2/htdocs  
working_dir: /usr/local/apache2/htdocs
```

Różnice są niewielkie w porównaniu do poprzedniego opisu. Tak samo będziemy budowali nasz obraz **PHP**, do którego za chwilę przejdziemy. W przypadku **PHP** będziemy korzystali z **MySQL** dlatego w tym wypadku w **links** informujemy od tym **Dockera**. Udostępnienie występuje na porcie **9000**. Na końcu podpięty jest ten sam katalog, co w przypadku **apache2**. Dzięki temu obsługiwane pliki są również w języku **PHP**. Na samym końcu określamy folder roboczy. Stwórzmy teraz katalog **PHP** i w nim plik **Dockerfile**. Zawartość pliku wyglądać powinna następująco:

```
FROM php:fpm  
  
RUN apt-get update  
RUN docker-php-ext-install pdo pdo_mysql mysqli
```

Choć sama konstrukcja nie powinna zaskoczyć to jednak jest coś ciekawego w tym, co znajduje się w tym pliku. Otóż nie korzystamy z typowej instalacji z użyciem apt. Korzystamy z menadżera pakietów **docker-php-ext**, który jest dostępny w obrazach PHP. Wiedza ta nie wzięła się znikąd. Jak już wspomniałem zawsze warto zajrzeć do opisu znajdującego się na docker hub. Tak jak w tym wypadku https://hub.docker.com/_/php znajdują się bardzo ciekawe informacje. Natomiast korzystam z wersji **fpm** ponieważ posiada szybsze komunikację pomiędzy serwerem a **PHP**. W pliku instalujemy niezbędne oprogramowanie do obsługi **MySQL**.

9.4. Baza MySQL

W przypadku bazy dość szczegółowo omówiliśmy ją w poprzednim materiale. Nim przystąpię do opisu chcę, abyś zerknął na poniższy przykład:

```
mysql:  
image: mysql  
container_name: mysql  
environment:  
MYSQL_ROOT_PASSWORD: '<haslo-root>'  
MYSQL_DATABASE: grupaadm
```

```
MYSQL_USER: grupaadm  
MYSQL_PASSWORD: '<db-haslo>'  
  
ports:  
- "3306:3306"  
  
volumes:  
- ./database/mysql:/var/lib/mysql
```

Tym razem nie będziemy budowali obrazu, tylko skorzystamy z istniejącego. Nowością związana z **docker compose** jest **enviroment**. To tu określasz wszystkie dyrektywy dotyczące **MySQL**. Opisem wszystkich tutaj wymienionych zajęliśmy się w poprzednim materiale, dlatego pozwolę sobie je pominąć. Pamiętaj, że w miejsca oznaczone nawiasem < wprowadź odpowiadające Ci wartości. Jeżeli tego nie zmienisz i rozpocznesz budowanie oraz uruchomienie serwera to pamiętaj, że hasło dla użytkownika **root** i **użytkownika** będzie tak jak tutaj wpisałem. Abyś sobie nie pomyślał, w przykładach nigdy nie stosuję znaków polskich. Dlatego zamiast hasło masz hasło. Standardowy port na którym dostępne jest **MySQL** to 3306. Natomiast montujemy jeszcze folder, w którym znajdować się będą pliki z naszymi bazami danych.

9.5. PHPMyAdmin

Jest to bardzo użyteczny pakiet w przypadku serwerów. Pozwala na zarządzanie bazami danych w sposób bardziej obrazowy i łatwiejszy dla osoby nie zajmującą się administracją. W związku z tym zainstalujemy taki:

```
phpmyadmin:  
image: phpmyadmin/phpmyadmin  
container_name: phpmyadmin  
  
links:  
- mysql  
  
environment:  
PMA_HOST: mysql  
PMA_PORT: 3306  
PMA_ARBITRARY: 1  
  
restart: always  
  
ports:  
- 1000:80
```

Jest to kolejny przykład dla którego warto czytać informacje o obrazie. Pod adresem https://hub.docker.com/_/phpmyadmin znajdziesz komplet informacji z nim związanych. Nie ma tutaj wiele do wyjaśnienia. Oprócz tego, co znajduje się w **environment**. Dwóch pierwszych się domyślasz

ponieważ jest to o tyle logiczne, że nie wymaga komentarza. Natomiast ostatnie ustawienie środowiskowe po nazwie nic nam nie mówi.

1 można było dostać w szkole :)

Dlatego zerkamy do opisu obrazu i odnajdujemy:

PMA_ARBITRARY - when set to 1 connection to the arbitrary server will be allowed

Czyli tłumacząc na język polski stosując 1 umożliwiamy połączenie z dowolnym serwerem.

9.6. Wszystkie pliki

Mamy już w pełni napisaną naszą konfigurację, pozostało tylko uruchomić. W związku z tym, że podzieliłem całość na kilka elementów, w tym miejscu chcę pokazać każdy z plików jako całość.

9.6.1. Compose.yaml

```
version: '3.8'
```

```
services:
```

```
apache:
```

```
    container_name: apache
```

```
    build: ./apache
```

```
    links:
```

```
        - php
```

```
    ports:
```

```
        - "80:80"
```

```
    volumes:
```

```
        - ./src:/usr/local/apache2/htdocs
```

```
php:
```

```
    container_name: php
```

```
    build: ./php
```

```
    links:
```

```
        - mysql
```

```
    ports:
```

```
        - "9000:9000"
```

```
    volumes:
```

```
        - ./src:/usr/local/apache2/htdocs
```

```
working_dir: /usr/local/apache2/htdocs
```

```
mysql:
```

```
    image: mysql
```

```

container_name: mysql

environment:
  MYSQL_ROOT_PASSWORD: '<haslo-root>'
  MYSQL_DATABASE: grupaadm
  MYSQL_USER: grupaadm
  MYSQL_PASSWORD: '<db-haslo>'

ports:
  - "3306:3306"

volumes:
  - ./database/mysql:/var/lib/mysql

phpmyadmin:
  image: phpmyadmin/phpmyadmin
  container_name: phpmyadmin
  links:
    - mysql
  environment:
    PMA_HOST: mysql
    PMA_PORT: 3306
    PMA_ARBITRARY: 1
  restart: always
  ports:
    - 1000:80

```

9.6.2. ./apache/grupaadm.apache.conf

```

LoadModule deflate_module /usr/local/apache2/modules/mod_deflate.so

LoadModule proxy_module /usr/local/apache2/modules/mod_proxy.so

LoadModule proxy_fcgi_module /usr/local/apache2/modules/mod_proxy_fcgi.so

<VirtualHost *:80>
  ProxyPassMatch ^/(.*\.\php(.*))\$ fcgi://php:9000/usr/local/apache2/htdocs/\$1

  DocumentRoot /usr/local/apache2/htdocs

  <Directory /usr/local/apache2/htdocs>
    Options -Indexes +FollowSymLinks

```

```
        DirectoryIndex index.php  
        AllowOverride All  
        Require all granted  
</Directory>  
</VirtualHost>
```

9.6.3. ./apache/Dockerfile

```
FROM httpd  
  
COPY grupaadm.apache.conf /usr/local/apache2/conf/grupaadm.apache.conf  
  
RUN echo "Include /usr/local/apache2/conf/grupaadm.apache.conf" \  
    >> /usr/local/apache2/conf/httpd.conf
```

9.6.4. ./php/Dockerfile

```
FROM php:fpm  
  
RUN apt-get update  
RUN docker-php-ext-install pdo pdo_mysql mysqli
```

9.7. Budowa oraz uruchomienie serwera

Napracowaliśmy się trochę nad konfiguracją wszystkiego teraz nadeszła pora zobaczyć tego wyniki.
Pierwsze co robimy, to budujemy niezbędne paczki.

```
docker compose build  
[+] Building 1.4s (16/16) FINISHED
```

Budowanie powinno wykonać się bardzo szybko. Pozostało stworzyć i uruchomić niezbędne kontenery które uruchomią nasz serwer.

```
docker compose up  
Uruchomienie tego polecenia może zająć chwilę. Po pobraniu oraz stworzeniu kontenerów powinny pojawić się informacje:
```

```
php      | [21-Oct-2022 08:25:56] NOTICE: ready to handle connections  
phpmyadmin | [Fri Oct 21 08:25:56.437802 2022] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.53  
(Debian) PHP/8.0.19 configured -- resuming normal operations
```

```
apache    | [Fri Oct 21 08:25:56.674016 2022] [mpm_event:notice] [pid 1:tid 140147845070144] AH00489:  
Apache/2.4.54 (Unix) configured -- resuming normal operations
```

```
mysql    | 2022-10-21T08:26:03.996969Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for  
connections. Version: '8.0.31' socket: '/var/run/mysqld/mysqld.sock' port: 3306 MySQL Community Server –  
GPL.
```

Oczywiście powyższe informacje będą znajdowały się w różnych miejscach. Uzyskanych z nich, że wszystko działa poprawnie. Teraz pozostaje wpisać localhost w naszej przeglądarce. Niestety otrzymamy informację: *File not found*. Pokaż też opcję detach:

```
docker compose up --detach
```

9.8. Pliki

Nie odnaleziono plików ponieważ żadnych nie dodaliśmy. Spójrzmy w nasz katalog gdzie mamy plik compose.yaml:

```
ls
```

```
apache compose.yaml database php src
```

Zwróć uwagę, że zostały stworzone dwa dodatkowe katalogi. Pierwszy z nich to **database**. Wspominałem o nim przy opisywaniu kodu odpowiedzialnego za **MySQL**. To w nim będą znajdować się wszystkie pliki związane z bazami. Drugi to **src**. O nim też wspominałem. Wejdźmy do niego i wyświetlimy zawartość. Jest on pusty. Jak dobrze pamiętasz został domontowany do naszego głównego systemu plików. W nim mają się znajdować pliki które wyświetlać będzie serwer. W związku z tym stwórzmy dokument **html** i wpiszmy w nim jakąś zawartość:

```
<h1>Plik HTML</h1>
```

Pamiętaj jednak, że aby zapisywać do katalogu musisz posiadać odpowiednie uprawnienia. Nie zmieniam w tym wypadku tego ponieważ folder służy tylko do testowania czy serwer, PHP i baza danych będzie działać.

Po stworzeniu takiego katalogu wejdźmy pod adres localhost/index.html. Wyświetli się strona z informacją jaką wprowadziliśmy.

Sprawdźmy teraz czy działa PHP. Dlatego tym razem tworzymy plik o nazwie index.php, po czym umieszczamy w nim:

```
<?php  
echo phpinfo();  
?>
```

Następnie przechodzimy pod adres **localhost**. Automatycznie zostanie wyświetlona informacja o zainstalowanej przez nas wersji **PHP**. Pozostało sprawdzić połączenie **PHP** z **MySQL**. W tym celu tworzymy kolejny plik o nazwie **mysql.php** i umieszczamy w nim:

```
<h1> Połączenie z bazą danych </h1>
```

```
<?php  
$hostname      = "mysql";  
$username      = "grupaadm";  
$password      = "<db-haslo>";  
  
$connection = new mysqli($hostname, $username, $password);  
  
if ($connection->connect_error) {  
    die("Połączenie z bazą danych się nie powiodło " . $connection->connect_error);  
}  
  
echo "Połączenie z MySQL zakończono z sukcesem!";  
  
?>
```

Po zapisaniu pliku przechodzimy pod adres **localhost/mysql.php** i powinniśmy otrzymać informację *Połączenie z MySQL zakończono z sukcesem!*

Wszystko działa tak jak należy. Jeżeli to ma być twoje środowisko testowania aplikacji, to wygodniej będzie zmienić uprawnienia do tego katalogu.

9.9. Sprawdzamy co zostało stworzone

Uczymy się, dlatego warto zawsze posprawdzać co dokładnie się stało. Poza tym będzie to dobra powtórka poznanych wcześniej poleceń. Wyświetlmy obrazy jakie obecnie posiadamy:

docker images

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|-----------------------|--------|--------------|--------------|-------|
| moj-apache | latest | 41f1f583066c | 24 hours ago | 145MB |
| moj-php | latest | 84feffbe3655 | 27 hours ago | 468MB |
| mysql | latest | 6cc1a43ad84d | 37 hours ago | 535MB |
| phpmyadmin/phpmyadmin | latest | 4a4023c7e22a | 5 months ago | 510MB |

Dwa pierwsze, posiadające nazwę `moj` są to obrazy, które my zbudowaliśmy przy pomocy **Dockerfile**. Natomiast pozostałe dwa są to obrazy pobrane bez modyfikacji. Który do czego służy można wywnioskować po nazwach dle tego tą część pominę. Zerknijmy teraz na kontenery:

```
docker ps -a
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS |
|--------------|-----------------------|--------------------------|----------------|---------------|--|
| NAMES | | | | | |
| 9ea9483a9577 | moj-apache | "httpd-foreground" | 31 minutes ago | Up 31 minutes | 0.0.0.0:80->80/tcp, ::80->80/tcp |
| 14693de5679c | phpmyadmin/phpmyadmin | "/docker-entrypoint...." | 31 minutes ago | Up 31 minutes | 0.0.0.0:1000->80/tcp, ::1000->80/tcp |
| b5832444f41e | moj-php | "docker-php-entrypoi..." | 31 minutes ago | Up 31 minutes | 0.0.0.0:9000->9000/tcp, ::9000->9000/tcp |
| 9d03e11cad5a | mysql | "docker-entrypoint.s..." | 31 minutes ago | Up 31 minutes | 0.0.0.0:3306->3306/tcp, 33060/tcp |

Jesteśmy w posiadaniu czterech kontenerów, które ze sobą współpracują. Czyli wszystko tak jak zakładaliśmy w pliku `compose.yaml`. Jest jeszcze jedna rzecz o której nie wspomniałem.

Aha, jeżeli wygląda to u Ciebie w ten sposób:

```
root@duplicate:~# docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
dockercompose-apache    latest   29435e0586dd  3 days ago   145MB
dockercompose-php      latest   ae2c34a7c711  3 days ago   468MB
alpine               latest   042a816809aa  2 weeks ago  7.05MB
debian               serwer   51a9a1c5341f  2 weeks ago  274MB
debian               1.1      b0e70a059374  2 weeks ago  243MB
debian               1.0      ce876deecb66  3 weeks ago  142MB
debian               latest   446440c01886  5 weeks ago  124MB
mysql                latest   7484689f290f  7 weeks ago  538MB
alpine               <none>   49176f190c7e  2 months ago  7.05MB
phpmyadmin/phpmyadmin  latest   4a4023c7e22a  8 months ago  510MB
hello-world          latest   feb5d9fea6a5  16 months ago 13.3kB
root@duplicate:~#
```

To tak jest, ponieważ katalog, w którym umieściłeś wszystkie pliki nazywa się nie `moj`, tak jak w przykładzie, tylko **dockercompose**. Standardowo przy budowaniu jest pobierana nazwa katalogu jako nazwa projektu. Przy budowaniu możesz użyć opcji `--project-name NAME`. W miejsce NAME wprowadzasz odpowiednią nazwę.

9.10. Sieci

No właśnie, nie stworzyliśmy żadnych własnych sieci. Jeżeli korzystasz z **docker compose**, to automatycznie jest tworzona taka sieć. Sprawdźmy czy tak jest, wykonując inspekcje kontenera:

```
docker inspect apache
```

```
"Networks": {  
    "moj_default": {  
        "IPAMConfig": null,  
        "Links": [  
            "php:php",  
            "php:php-1",  
            "php:moj-php-1"  
        ],  
    },
```

Wkleiłem tylko interesująca nas część. Sieć, która została stworzona nosi nazwę **moj_default**. Ponieważ katalog, w którym znajdują się pliki nazywa się moj. Dokonajmy teraz inspekcji tej sieci:

```
docker network inspect moj_default
```

```
"Containers": {  
    "14693de5679c49efe6e080b7cc3e20e23986cab9a7cd65d555ac48387729de07": {  
        "Name": "phpmyadmin",  
        "EndpointID": "169fe28dd8f4fdfa23eab9b2c83d93ccd0eae0618e7ff13f09c8b95f18dcc989",  
        "MacAddress": "02:42:ac:15:00:03",  
        "IPv4Address": "172.21.0.3/16",  
        "IPv6Address": ""  
    },  
    "9d03e11cad5ac784b2b5818effd957a7c1f5099d325a91e9810eb126abbc9850": {  
        "Name": "mysql",  
        "EndpointID": "62f6d2de481a5432431e8471c3995e51e861011d244aaaf47ff0b89679b38c8f5",  
        "MacAddress": "02:42:ac:15:00:02",  
        "IPv4Address": "172.21.0.2/16",  
        "IPv6Address": ""  
    },  
    "9ea9483a9577deef426eec8e5dc12269a42029495e262a8c2e6f49748db88fb8": {  
        "Name": "apache",  
        "EndpointID": "3b68466acd584305c638d1cb1ee3a91724005dd37de3baed8f1cde150004cd0a",  
        "MacAddress": "02:42:ac:15:00:05",  
    }
```

```

    "IPv4Address": "172.21.0.5/16",
    "IPv6Address": ""

},
"b5832444f41e60a8a5035206f204d2a3575bc7a826d5138515cb56930904c78c": {
    "Name": "php",
    "EndpointID": "34fd7274eee61639ff23b153720ad2f52d7cbc4c2122f6fa435a48c3dcdf2185",
    "MacAddress": "02:42:ac:15:00:04",
    "IPv4Address": "172.21.0.4/16",
    "IPv6Address": ""

}
},

```

Z powyższego przykładu wynika, że wszystkie cztery kontenery znajdują się w jednej sieci. Dlatego też występowała komunikacja pomiędzy nimi. Natomiast takie rozwiązanie nie zawsze jest przez nas chciane. Możemy mieć potrzebę stworzenia dwóch oddzielnych sieci i posegregowania kontenerów między nimi.

Pierwsza z nich odpowiadająca za **backend** druga za **frontend**. Robimy to w bardzo prosty sposób. Ale nim to nastąpi zatrzymajmy działający serwer przy pomocy kombinacji CTRL+C. Teraz edytujmy nasz plik **compose.yaml**. Niech wygląda w następujący sposób:

```

version: '3.8'

services:
  apache:
    container_name: apache
    build: ./apache
    links:
      - php
  networks:
    - front
    - back
  ports:
    - "80:80"
  volumes:
    - ./src:/usr/local/apache2/htdocs
  php:
    container_name: php

```

```
build: ./php

links:
  - mysql

networks:
  - back

ports:
  - "9000:9000"

volumes:
  - ./src:/usr/local/apache2/htdocs

working_dir: /usr/local/apache2/htdocs

mysql:
  image: mysql
  container_name: mysql
  environment:
    MYSQL_ROOT_PASSWORD: '<haslo-root>'
    MYSQL_DATABASE: grupaadm
    MYSQL_USER: grupaadm
    MYSQL_PASSWORD: '<db-haslo>'

networks:
  - back

ports:
  - "3306:3306"

volumes:
  - ./database/mysql:/var/lib/mysql

phpmyadmin:
  image: phpmyadmin/phpmyadmin
  container_name: phpmyadmin
  links:
    - mysql
  environment:
    PMA_HOST: mysql
    PMA_PORT: 3306
    PMA_ARBITRARY: 1
  restart: always
```

networks:

- **back**

ports:

- 1000:80

networks:

front:

back:

Zmiany jakie wykonałem wyróżniłem. Z użyciem pliku **yaml** jesteśmy w stanie również tworzyć własne sieci w prezentowany sposób. Natomiast do dwóch sieci dodałem tylko apache, bo to jest serwer wyjściowy. Do pozostałych trzech tylko back. Nie ma sensu dodawania do sieci front na przykład **phpmyadmin** ponieważ on nie współpracuje z apachem tylko z PHP i MySQL.

Na samym końcu, gdy tworzymy sieci musimy je wymienić w prezentowany sposób. Aby wprowadzić zmiany i uruchomić serwer wystarczy użyć dwóch poznanych przez nas poleceń:

docker compose build

docker compose up

W ten sposób zmiany zostały wprowadzone. Możesz to sprawdzić dokonując inspekcji.

9.11. Dostęp do panelu PHPMyAdmin

Na sam koniec pozostał tylko dostęp do wymienionego panelu. W pliku **compose.yaml** ustaliliśmy port **1000**, dlatego teraz wpisujemy w przeglądarkę **localhost:1000**. Podajemy **login** i **hasło**, które ustaliśmy w wymienionym pliku i mamy dostęp do naszych baz. Serwer z jakim się łączysz to nie localhost. Musisz wpisać nazwę kontenera zamiast localhost.

The screenshot shows the phpMyAdmin login interface. At the top, it says "Witamy w phpMyAdmin". Below that is a language selector labeled "Język (Language)" with "Polski - Polish" selected. The main part is a login form with three fields: "Serwer" (server), "Użytkownik" (username), and "Hasło" (password). The "Użytkownik" field contains "grupaadm". At the bottom of the form is a "Login" button.

9.12. Podsumowanie

Compose jest bardzo ważnym i pożytecznym narzędziem, które warto zainstalować razem z Dockerem. Jak sam widzisz po przeczytaniu materiału zbudowaliśmy dwa obrazy. Następne dwa pobraliśmy, rozłożyliśmy odpowiednio po sieciach i uruchomiliśmy. Mamy już gotową konfigurację, którą przy pomocy dwóch poleceń możemy użyć w wielokrotnie.

Rozdział 10: Docker Swarm

Poznaliśmy bardzo dużo tematów związanych z samym Dockerem. Teraz zajmiemy się elementem Dockera, który sprawi, że nasz serwer czy też aplikacja stanie się bardziej odporna na awarie. Tematem będzie **Docker Swarm**. Narzędzie to przydaje się szczególnie w sytuacji gdy to co stworzymy lub administrujemy zacznie rozrastać się do tak ogromnych rozmiarów, że jeden serwer nie jest w stanie operować samoistnie takim zasobem. Najlepszym przykładem jest firma **Google** która stosuje tego typu rozwiązanie w swoich aplikacjach. W ich wypadku jeden serwer nie jest w stanie obsłużyć tylu zadań. Jeżeli w twojej firmie istnieje narzędzie które musi przetworzyć ogromną ilość zadań **Docker Swarm** może okazać się idealnym rozwiązaniem.

W przypadku gdy narzędzie o ogromnym potencjale zainstalujemy na urządzeniu może okazać się, że zasoby jakim to urządzenie dysponuje szybko się skończą.

Dzięki **Docker Swarm** rozłożymy zadania, jakie musi wykonać narzędzie, na kilka maszyn. Pomimo tego podziału będzie ono funkcjonowało jako jedno. W związku z tym zasada synchroniczności pozostanie w całości zachowana. Jak zatem widzisz, działa to na zasadzie klastra.

Wszystko jest zarządzane z jednego miejsca. W związku z tym obsługa jest łatwiejsza. **Docker Swarm** posiada menadżera, który rozdziela poszczególne zadania po urządzeniach wykonawczych. Wszystkie te urządzenia tworzą jeden.

Inną bardzo cenną usługą dostępną w narzędziu jest wykonywanie replik. Dzięki nim, gdy jedno z urządzeń przestanie działać, jego zadania przejmą pozostałe lub jedno z nich.

10.1. Jak wygląda konstrukcja Docker Swarm

Docker Swarm inaczej nazywany jest **container orchestrator (orkiestrator)** ponieważ **jest on narzędziem służącym do udostępniania, planowania i zarządzania kontenerami na dużą skalę lub dużą ilością klastrów**.

Konstruowanie takiego klastra polega na stworzeniu kilku hostów opartych o dockera i dzięki **Docker Swarm** łączymy je w jeden klaster. Jeden z tych **hostów** staje się **menedżerem** i to on zarządza całością. To on udostępnia tak zwany klucz, który umożliwia **węzłom** przyłączenie się do klastra. Po dołączeniu stają się **węzłami do wykonywania zadań**. W związku z tym wspomniane węzły możemy określić jako **urządzenia** które będą wykonywały wyznaczone przez menedżera zadania.

Komunikację pomiędzy wszystkimi wspomnianymi elementami możemy wytlumaczyć na zasadzie hierarchii w pracy. Powiedzmy, że pracujemy w firmie w której mamy kilka szczebli zarządzania. **Dyrektor** określa zadania **Kierownikowi**. Następnie **Kierownik** rozdziela te zadania pomiędzy swoich **podwładnych**. Tak samo jest w przypadku **Docker Swarm**. Otóż użytkownik aplikacji, serwera czy też narzędzia, wykonuje jakąś czynność. Czynność która ma być wykonana przekazywana jest menadżerowi. Następnie on rozdziela jej wykonanie pomiędzy **węzły**, czyli *urządzenia do wykonywania zadań*.

Jednak menedżer nie jest zupełnie pozostawiony samemu sobie. Posiada kilka wbudowanych narzędzi:

- **API** – dzięki tej funkcji obsługuje nasze zgłoszenia i tworzy z nich obiekty;
- **Orchestrator** – dzieli przesłane polecenie na mniejsze zadania i przesyła do węzłów;
- **Allocator** – przydziela adresy wewnętrzne IP klastrów wykonawczym oraz menadżerowi;
- **Dispatcher** - decyduje, który węzeł będzie obsługiwał dane zadanie i przekazuje te informacje *Orchestratorowi*;
- **Scheduler** – uruchamia przydzielone zadania/zadanie oraz decyduje które będzie uruchomione jako pierwsze;

W przypadku urządzeń wykonawczych ich zadania są o wiele prostsze w porównaniu do tego, co przeczytałeś powyżej. W pewnym sensie ograniczone są do dwóch czynności. Po pierwsze sprawdzane jest czy jakieś zadanie jest przydzielone. Po drugie je wykonuje.

Natomiast jeżeli chodzi o same zadania, to węzeł odpowiada za **tworzenie kontenerów, wolumenów** czy też **sieci, automatycznie uruchamiając je**.

O tym już wspomniałem, ale jak już funkcjonowanie rozłożyliśmy na czynniki pierwsze pozwoleć sobie wspomnieć ponownie. W przypadku, gdy jedno z urządzeń zostanie odłączone, ulegnie awarii lub coś się z nim stanie i będzie nieaktywne, to zadania jakie były do niego przydzielone zostaną przeniesione na inne węzły. Co interesujące jest to, że jeżeli urządzenie które uległo awarii wróci ponownie do działania, to zadanie jakie było mu przydzielone przed awarią może do niego wrócić, ale nie musi. Jeżeli nie obciąża mocno zasobów może pozostać w obsłudze przez urządzenie zastępcze.

W przypadku jeżeli **menedżer** ulegnie awarii, to automatycznie jedno z urządzeń w klastrze staje się menedżerem i działa na pełnych przedstawionych wcześniej zasadach. Na końcu chciałbym dodać, że aby **Swarm** działał efektywnie, to funkcjonować musi ponad połowa urządzeń, a przynajmniej jedno więcej..

Dość dużo teorii, ale niestety **Docker Swarm** jest to bardzo złożony proces. Pomimo swojej rozwiążności jest warty poznania.

10.2. Docker vs Docker Machine vs Docker Swarm

W następnej części materiału zajmiemy się tworzeniem klastra na dwa sposoby. Pierwszy z nich polega na zainstalowaniu serwera **Ubuntu w VirtualBox** i konfiguracji sieciowej oraz instalacji ręcznej samego **Dockera**. W przypadku drugim automatycznie wykonamy niektóre czynności przy pomocy narzędzia **docker-machine**. Nim przejdziemy do wspomnianych czynności, chciałbym jednak, abyś zapoznał się z podstawowymi informacjami oraz wymaganiami związanymi z samym **Dockerem**, jak i również z **Docker Machine** i **Docker Swarm**.

10.2.1. Docker, a Docker Swarm

Możliwie, że wiesz, iż **Dockera** możemy zainstalować na dwa sposoby. Pierwszy sposób polega na instalacji **Docker Engine**, z którego korzystamy przy pomocy wiersza poleceń. W ten sposób w całym kursie wykonywaliśmy wszystkie czynności do tej pory. Natomiast drugim rozwiązaniem jest **Docker Desktop**, który został pominięty w tym materiale. Jeżeli nauczysz się posługiwać **Dockerem**, przy użyciu poleceń, to obsługa **GUI** nie będzie dla Ciebie żadną barierą. Z nazwą **Docker Engine** możesz się jeszcze nie spotkać lub gdzieś raz czy dwa razy o tym słyszałeś. Otóż na co dzień nie korzysta się z nazwy **Docker Engine** tylko z pojedynczego słowa **Docker**. Jest tak łatwiej i przyznam, że przyjęto się wśród jego użytkowników.

Razem z instalacją **Dockera**, instalujemy takie narzędzie jak **Docker Swarm**, które służy do **orkiestracji kontenerów**. Można by prościej powiedzieć, że **Docker Swarm** jest pewnego rodzaju narzędziem służącym do zarządzania kontenerami tworzonymi w nodach.

Jednak to jeszcze nie wszystko. **Docker Swarm** nie jest to jedyne narzędzie, z którego będziemy korzystali. Otóż dysponujemy jeszcze **Docker Machine**, o którym chcę porozmawiać w następnym paragrafie.

10.2.2. Co to jest Docker Machine i co ma wspólnego z Docker Swarm?

Docker Machine jest to narzędzie, dzięki któremu jesteś w stanie automatycznie zainstalować wersję Dockera na wirtualnym i lokalnym hoście i nim zarządzać przy pomocy poleceń. Możesz korzystać z rozwiązań chmurowych dostawców takich jak **Azure**, **AWS** czy też **Digital Ocean**. Przy jego pomocy zautomatyzujesz wstępную konfigurację oraz niezbędną instalację oprogramowania.

Jeżeli uważnie przeczytałeś dotychczasowy materiał to możliwe, że zwróciłeś uwagę na pewną zbieżność wszystkich wymienionych narzędzi. Otóż **Docker Swarm** jest pewnego rodzaju organizatorem kontenerów, natomiast **Docker Machine** jest narzędziem, dzięki któremu zarządzać możesz **Dockerem** i **Dockerem**.

Używając **Docker Machine**, nie będziesz zmuszony wykonywać konfiguracji ręcznie. Stworzy ono samo maszyny przy użyciu określonego sterownika. W naszym przykładzie korzystamy z **VirtualBox**. Jednak jeśli dysponujesz innym narzędziem, to zajrzyj pod adres <https://docs.docker.com/xy2401.com/machine/drivers/>. We wskazanym odnośniku znajduje się lista obsługiwanych narzędzi oraz chmur przez **Docker Machine**. Jest ona bardzo obszerna, dlatego warto przejrzeć dostępne możliwości.

Wiesz już, że **Docker Machine** jest to narzędzie instalujące oraz zarządzające **Docker Swarmem** i **Dockerem** w chmurze, jak i również w systemach wirtualnych. Podstawowe zadanie **Docker Machine** już znasz, natomiast teraz bardziej szczegółowo chciałbym opisać **Docker Swarm**.

Otoż **Docker Swarm** jest narzędziem służącym do orkiestracji kontenerów, o czym już wspominałem. Dzięki niemu grupujemy maszyny fizyczne lub wirtualne w klaster. Tej grupie maszyn rozdzielane są odpowiednio kontenery, z których chcemy skorzystać. Ze względu na łatwy proces instalacji i nieskomplikowaną obsługę, często jest wykorzystywany do mniejszych projektów. Innym narzędziem służącym orkiestracji, ale bardziej złożonym jest **Kubernetes**, którego podstawy poznasz w dalszej części tego szkolenia. Jednak w odróżnieniu od Kuberntesa jak już wiesz, Docker Swarm jest automatycznie instalowany razem z Dockerem. Dlatego, aby z niego korzystać, nie musisz nic ponadto robić.

10.2.3. Czy użycie Docker Machine jest konieczne?

W związku z tym, co przeczytałeś do tej pory, możesz zadać sobie pytanie, czy koniecznie musimy korzystać z **Docker Machine**? W pewnym sensie odpowiedziałem już na nie w poprzednim akapicie, jednak chciałbym jeszcze chwilę poświęcić temu zagadnieniu.

Otoż w jeszcze inny sposób **Docker Machine** możemy określić jako oprogramowanie wspomagające. Dzięki niemu możemy, zarządzać maszynami wirtualnymi, jak i również automatyzujemy większość początkowych czynności, jakie musimy wykonać, by stworzyć taki klaster. Jak pokażemy w dalszej części tego materiału, mamy możliwość wykonania wszystkiego bez tej automatyzacji. Jednak to wszystko zależy od naszych potrzeb. Dlatego korzystanie z **Docker Machine** nie jest wymagane. Możemy korzystać tylko i wyłącznie z **Docker Swarma** i zainstalować wszystko ręcznie. Niesie to za sobą więcej pracy, jednak wszelkie ustawienia dostosowujemy sami, a nie korzystamy z szablonu, który wykorzystuje polecenie `docker-machine`.

10.2.4. Inne sposoby na uruchomienie Docker Swarm

Jeżeli chodzi o samo korzystanie z Docker Swarm bez użycia VirtualBox, powinieneś skonfigurować serwer w sposób, jaki zaprezentuję w przypadku instalacji i konfiguracji ręcznej. Wszystko funkcjonuje

na identycznych zasadach, jednak nie uwzględniałem w nim konieczności odblokowania portów. Porty, jakie należy odblokować wraz z krótkim opisem, przedstawiam poniżej:

- **port TCP 2377** do zarządzania komunikacją między klastrami;
- **port TCP i UDP 7946** do komunikacji między węzłami;
- **port UDP 4789** dla przepływu ruchu w nakładkowej sieci.

Na niektórych serwerach wymienione powyżej porty mogą być już odblokowane, jednak zawsze warto sprawdzić i w razie potrzeby je odblokować.

10.2.5. Dlaczego użyłem VirtualBox

Virtualboxa używam ze względu na jego dostępność. Zarówno jeden, jak i drugi przykład jesteś w stanie wykonać na swoim domowym komputerze bez ponoszenia dodatkowych kosztów, jak na przykład kilku serwerów w chmurze. Virtualbox jest udostępniony za darmo, dlatego nie musisz wykupić licencji tak jak w przypadku VMware. **Dlatego uważam tę platformę do obsługi wirtualnych maszyn za najlepszą do nauki.**

10.2.6. Wymagania co do zasobów sprzętowych

Jeżeli uczysz się obsługiwanego omawianych poniżej narzędzi, wystarczy po **2 GB pamięci RAM** oraz po **jednym procesorze** na node. Czyli przy **jednym menadżerze i 3 nodach** potrzebujemy **8GB ramów i 4 procesory**. Natomiast jak wiadomo, są to wymagania minimalne dla samego **Docker Swarm**, dlatego komputer czy też serwer powinien posiadać więcej do swojego funkcjonowania.

Jeżeli dysponujesz mniejszymi zasobami, to stwórz przynajmniej **1 menadżera i 1 noda**. Ilość potrzebna zmniejszy się 2x, czyli będziesz przy takiej konfiguracji potrzebował **4 GB pamięci RAM i 2 procesory**, które poświęcisz na Docker Swarm. W kursie chcę pokazać w jak najobszerniejszy sposób jego działanie, dlatego zdecydowałem się na większą ilość nodów. Jednak przy opisanym minimum powinieneś być również w stanie wykonać wszystkie opisane czynności.

W przypadku wykorzystania serwerowego wszystko zależy od aplikacji lub też funkcji, jaka miałaby być obsługiwana przez **Docker Swarm**. Jednak w środowiskach produkcyjnych minimum to **4 procesory / 8 GB RAM / 200 GB miejsca**, czyli tak jak zaproponowałem w tym kursie. Jednak optymalna konfiguracja powinna wyglądać następująco **8 procesorów / 16 GB RAM / 200 GB miejsca**. To jest założenie przy 1 menadżerze i 3 nodach.

W dalszej części wszystko to, co przeczytałeś do tej pory pokażę na przykładach. Dlatego, jeżeli jest coś czego nie rozumiesz mam nadzieję, że lepiej uda mi się wytłumaczyć pokazując to na przykładzie.

10.3. Pierwszy sposób: instalacja i konfiguracja ręczna w VirtualBox

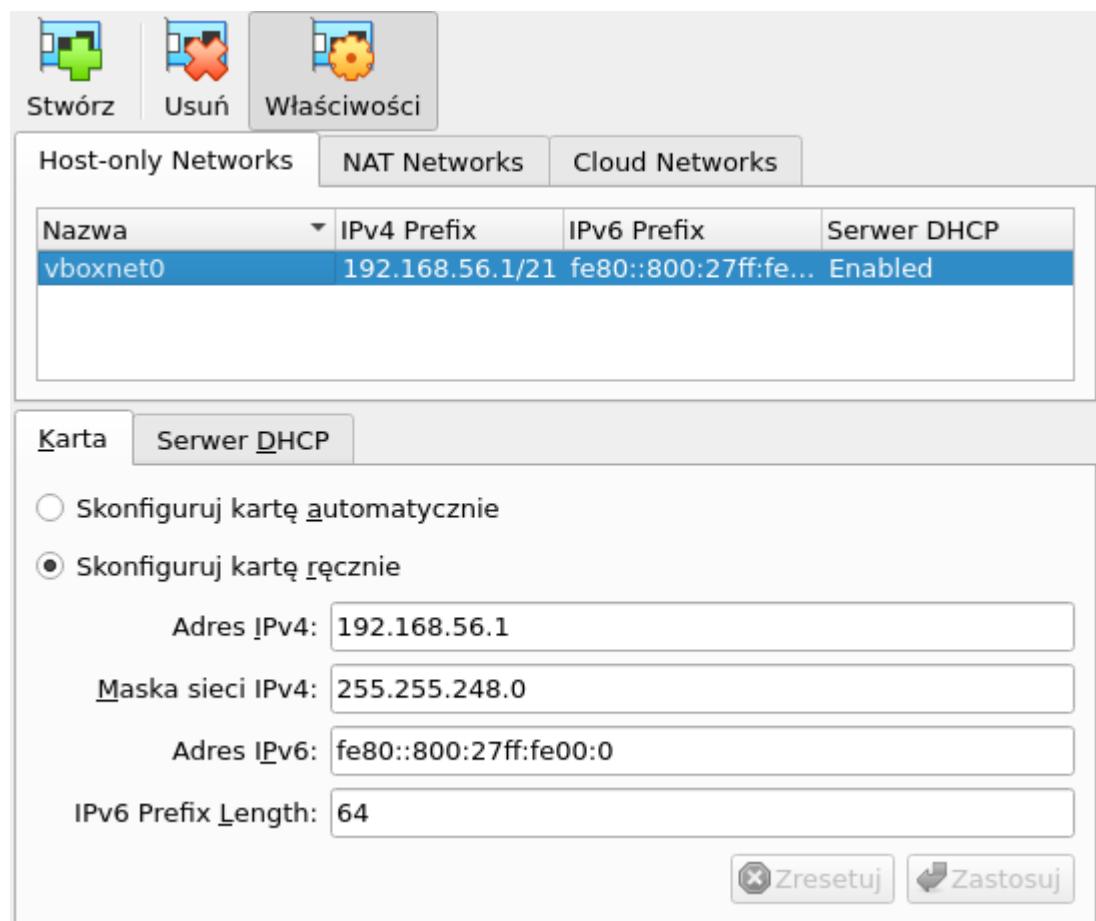
Tworzymy 3 maszyny wirtualne. 1 master, 2 nody.

W przypadku stworzenia **klastra** możemy użyć do tego polecenia, które automatycznie wykona za nas większość czynności lub możemy pokusić się o własną konfigurację. W tym przypadku wykonamy własną konfigurację. Do tego będziesz potrzebował, zainstalowanego VirtualBoxa jednak podobnie wykonasz wszystkie czynności na serwerze zewnętrznym. Zmianom ulec będą musiały jedynie adresy IP oraz zbędna będzie konfiguracja samego VirtualBoxa. Wszystko, co pokażę, wykonywałem na wersji **VirtualBox 7.0.6**.

Drugie co będzie niezbędne to serwer. Ja korzystam z **Ubuntu Serwer 22.04.1 LTS**, możesz go pobrać ze strony <https://ubuntu.com/download/server>. Następnie zainstalować przy pomocy Virtualbox. Przy tworzeniu miejsca na instalację i przydzielaniu zasobów maszynie wirtualnej, ustaw jej **2GB RAM** i **1 CPU**. Gdy rozpocznie się instalacja i staniesz przed wyborem typu instalacji, najlepiej wybierz **minimalną**. Gdy zostaniesz zapytany o **instalację SSH**, potwierdź. Przyda się do dalszych czynności.

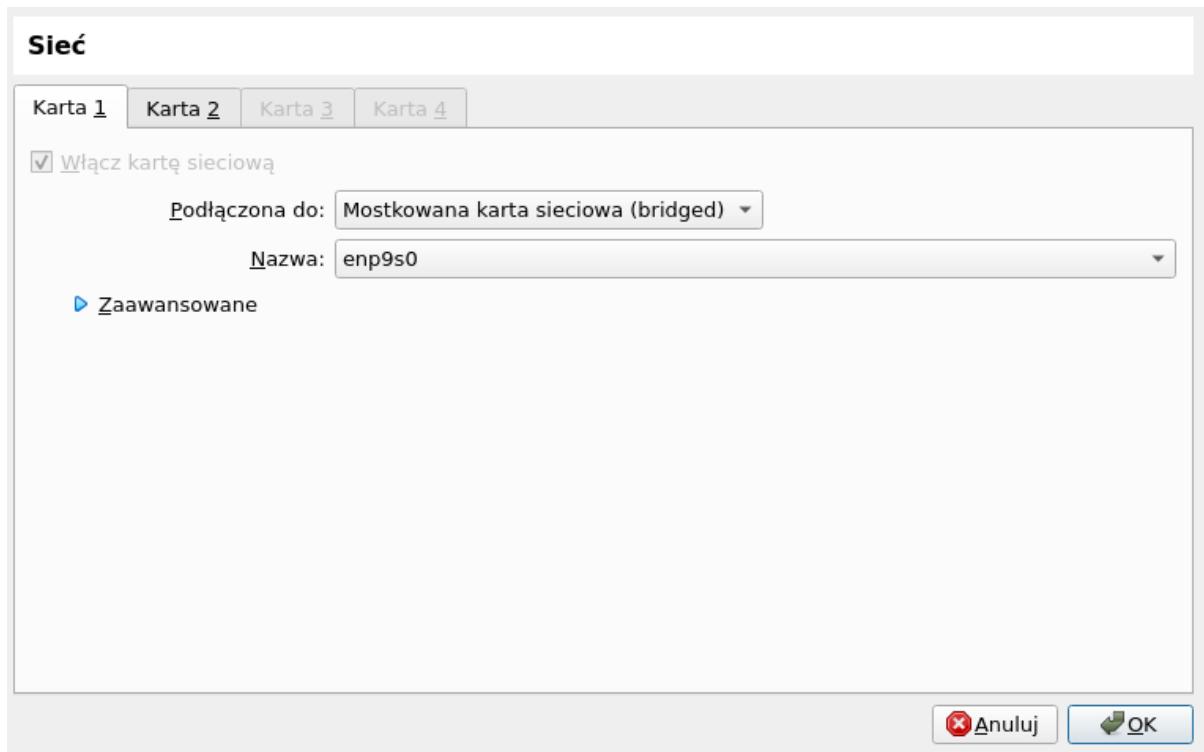
Po zainstalowaniu serwera z menu virtualboxa wybieramy **Plik > Narzędzia > Network Manager**.

Powinno pojawić się okno jak poniżej:

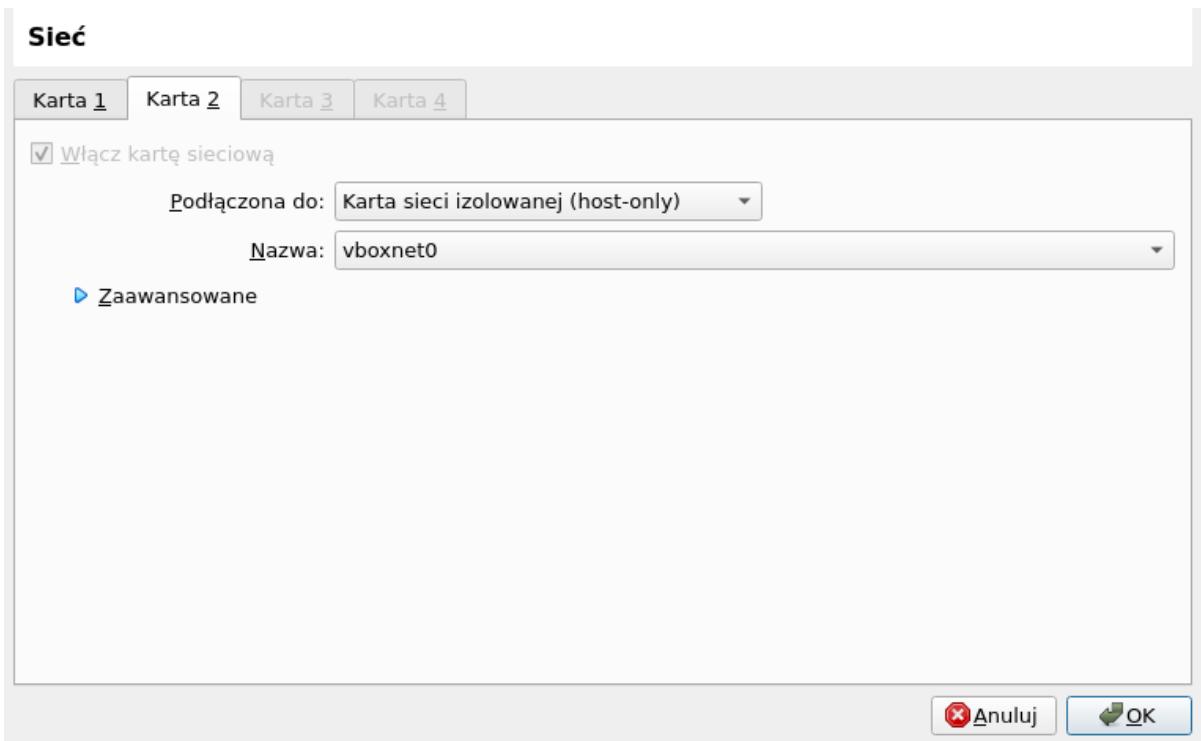


Na prezentowanym rysunku wypisana została przykładowa konfiguracja, z której ja będę korzystał. Natomiast najprawdopodobniej twoja lista będzie pusta. W górnym lewym rogu wciskasz przycisk **Stwórz** i następnie uzupełniasz. W przypadku drugiej zakładki **Serwer DHCP** możesz go wyłączyć. Nie będziemy korzystali z automatycznego przydzielania adresów IP. Po wykonaniu konfiguracji wciskasz przycisk **Zastosuj**.

Następnie konfigurujemy nasz serwer. Najlepiej na liście z lewej strony zaznaczyć obraz i gdy kurSOR będzie się na nim znajdował, kliknąć prawym przyciskiem myszki. Z listy wybrać opcję **Ustawienia**. Jeżeli wcześniej popełniłeś jakiś błąd w przydzielaniu zasobów, to możesz ponownie tego tutaj dokonać. Natomiast nas będzie interesowała zakładka **Sieć**. Po jej wybraniu pojawić się powinno okno jak na poniższym rysunku:



W pierwszej karcie ustawiasz **interfejs sieciowy**, z którego korzystasz w swoim systemie hosta. Oczywiście nazwa może być u Ciebie inna.



Następnie przechodzisz do **Karty 2** i **włączasz kartę sieciową**. Z listy *połączona do* wybierasz tak, jak wskazałem na rysunku i w polu *nazwa* wybierasz **interfejs sieciowy**, który stworzyliśmy przed chwilą. Po tych kilku krokach klikasz ok i wszystko związane z Virtualbox masz już skonfigurowane.

Uruchamiasz maszynę wirtualną ze stworzonym serwerem i logujesz się do konsoli. Teraz musisz zainstalować **Dockera**. Stworzyliśmy bardzo obszerny rozdział dotyczący instalacji Dockera.

Po wykonaniu instalacji dodam jeszcze, że dobrze jest aktywować Dockera, aby uruchamiał się automatycznie. Przy dystrybucjach Desktopowych ta czynność w większości z nich jest wykonywana automatycznie. Natomiast w wypadku serwerów, rzadko. Dlatego po zainstalowaniu wprowadź polecenie:

```
sudo systemctl enable docker
```

Przy każdym restarcie serwera Docker automatycznie będzie się uruchamiał. Przyszła kolej na ustawienie sieci. Sprawdzamy interfejs, jaki został przydzielony przy pomocy polecenia:

```
ip a
```

```
enp0s8...
```

U mnie jest to **enp0s8** i nie jest on uruchomiony. Dlatego teraz przeprowadzimy konfigurację tego interfejsu sieciowego. W wersji, z której korzystam, nie jest zainstalowany żaden edytor tekstowy. Dla ułatwienia początkującym osobom zainstalujemy prosty edytor nano:

```
sudo apt update && sudo apt install nano
```

Oczywiście, jeżeli znasz edytor **vi/vim** lub inny skorzystaj z niego. Edytor **nano** jest najprostszy, dlatego go tutaj użyłem.

Edytujemy plik **konfiguracyjny interfejsów sieciowych**:

```
sudo nano /etc/netplan/00-installer-config.yaml
```

W nim wprowadzamy następującą konfigurację:

```
# This is the network config written by 'subiquity'  
network:  
  ethernets:  
    enp0s3:  
      dhcp4: true  
    enp0s8:  
      addresses: [192.168.56.100/21]  
      dhcp4: false  
  version: 2
```

Pierwsza część była dodana automatycznie, jest to sieć naszego hosta. Natomiast drugą my musimy wprowadzić. W miejsce interfejsu wpisz nazwę interfejsu taką, jaką pojawiła się u Ciebie. Jeżeli nic nie modyfikowałeś pod względem moich ustawień, gdy tworzyliśmy sieć **vboxnet0**, to możesz wprowadzić identyczne wartości. Pamiętaj jednak, że to 100 na końcu adresu IP zmieniasz w następnych węzłach. Naszemu menedżerowi przypisujemy w tym wypadku adres z końcówką 100. Następnie węzłom będziemy przypisywać 101, 102, 103. Zapisujemy nasze ustawienia za pomocą skrótu klawiszowego **CTRL+O**, wciskamy **ENTER** i wychodzimy z edytora za pomocą klawiszy **CTRL + X**.

Pozostało aktywować nasze ustawienia za pomocą polecenia:

```
sudo netplan apply
```

Nim zrestartujesz serwer lub go zamkniesz ustaw jeszcze odpowiedni **hostname**, by łatwiej rozpoznać maszynę:

sudo hostnamectl set-hostname manager

Po ustawieniu ustawień restartujemy nasz serwer:

sudo reboot

Po jego ponownym uruchomieniu sprawdzamy, czy został właściwie przypisany adres IP:

```
enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default  
qlen 1000  
      link/ether 08:00:27:a1:e0:a7 brd ff:ff:ff:ff:ff:ff  
      inet 192.168.56.100/21 brd 192.168.63.255 scope global enp0s8  
        valid_lft forever preferred_lft forever  
      inet6 fe80::a00:27ff:fea1:e0a7/64 scope link  
        valid_lft forever preferred_lft forever
```

I sprawdzamy, czy hostname również został ustawiony, chociaż bez tego to już powinno być widoczne:

sudo hostnamectl

Static hostname: manager

Icon name: computer-vm

Chassis: vm

Machine ID: da7eebd321b14349807388b89163a70d

Boot ID: dab088064cd445c19226846f58b8a907

Virtualization: oracle

Operating System: Ubuntu 22.04.1 LTS

Kernel: Linux 5.15.0-58-generic

Architecture: x86-64

Hardware Vendor: innotek GmbH

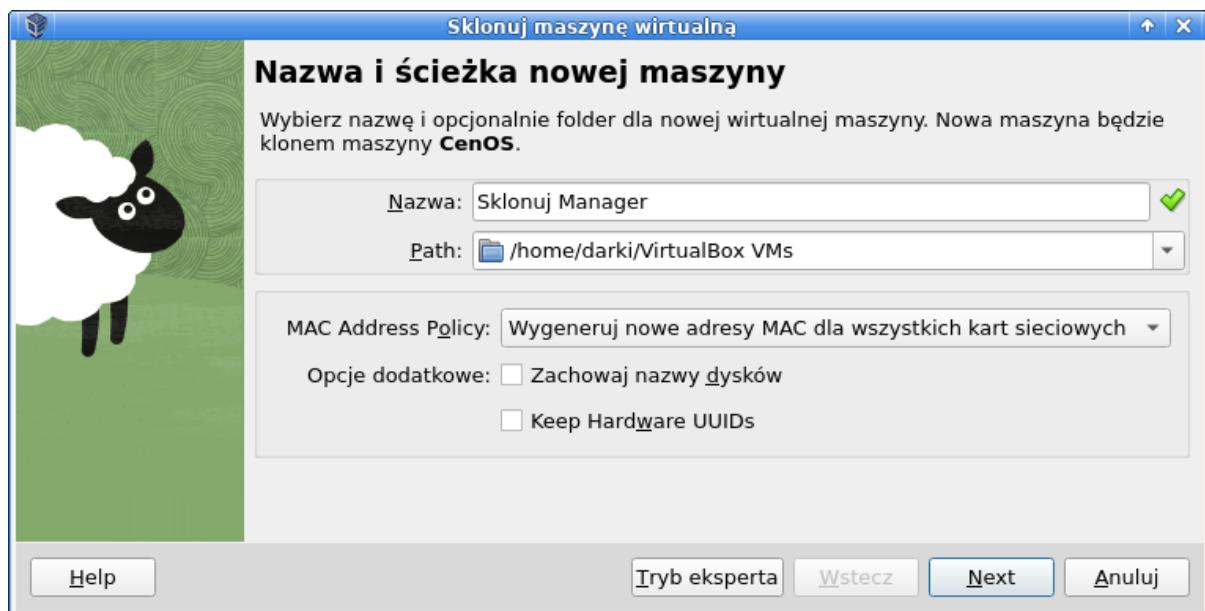
Hardware Model: VirtualBox

Zgodnie z otrzymanymi informacjami, konfiguracja przebiegła poprawnie. Teraz wyłączamy nasz serwer:

sudo shutdown -h now

Zgodnie z instrukcjami w dalszej części kursu potrzebujemy jeszcze 3 węzłów. Możemy to zrobić na dwa sposoby, zależy to od Ciebie. Albo postępujesz 3x identycznie, jak robiłem to przed chwilą, z pominięciem stworzenia sieci, bo dodajesz do niej swoje węzły. Albo klonujesz maszyny i jedyne co, w nich robisz, to zmieniasz IP i przydzielasz nazwę.

W przypadku klonowania kursorem myszki najeżdzasz na maszynę wirtualną. Następnie klikasz, sklonuj i powinno pojawić się okno:



Proponuję zmienić *nazwę* na *Wezel1* oraz *zasady adresacji MAC*, na wskazaną opcję. Po tych konfiguracjach klikamy **Next** i w następnym oknie również **Next**.

Powinien pojawić się nowy obraz z określona nazwą. Uruchamiamy go i wprowadzamy zmiany w pliku:

```
sudo nano /etc/netplan/00-installer-config.yaml
```

Zmieniamy końówkę adresu IP na 101. Zatwierdzamy zmiany:

```
sudo netplan apply
```

Zmieniamy hostname:

```
sudo hostnamectl set-hostname wezel1
```

I restartujemy serwer lub go zamykamy. Wspomniane czynności wykonujemy tyle razy, ilu węzłów potrzebujemy.

10.4. Sposób drugi: Instalacja przy pomocy docker-machine

Na początek warto zrobić aktualizację systemu:

```
sudo apt-get upgrade
```

```
sudo apt-get upgrade
```

Następnie zainstaluj Virtualbox:

```
sudo apt-get install virtualbox
```

Po instalacji VirtualBox zainstalujemy **Docker Machine**. Robimy to przy pomocy polecenia:

```
sudo curl -L https://github.com/docker/machine/releases/download/v0.16.2/docker-machine-`uname -s`-`uname -m` >/tmp/docker-machine && chmod +x /tmp/docker-machine && sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```

Gdybyś nie miał zainstalowanego Curl:

```
sudo apt-get install curl
```

W celu sprawdzenia czy **Docker Machine** został zainstalowany, wprowadzamy w konsoli polecenie:

```
docker-machine -v
```

```
docker-machine version 0.16.2, build bd45ab13
```

Zwróć uwagę, że nie posługujemy się poleceniem **docker-swarm** czy też podobnym sformułowaniem, tylko **docker-machine**. Dzięki temu narzędziu w dalszej części stworzymy nasz klaster i będziemy korzystali z Docker Swarm. Natomiast z samego określenia *docker swarm* skorzystamy wewnątrz naszych węzłów.

W prezentowanym przykładzie wskazana została wersja z jakiej będę korzystał, jak i również potwierdziłem w ten sposób instalację tego oprogramowania. Teraz sprawdźmy co dzięki niemu możemy zrobić.

10.5. Zapoznanie się z pomocą polecenia

Docker Machine jest narzędziem, które pozwala łatwo instalować i zarządzać hostami Docker na różnych platformach. Umożliwia on automatyzację procesu instalacji i konfiguracji Docker Engine na różnych systemach operacyjnych, w tym Windows, MacOS i popularnych dystrybucjach Linuksa, takich jak Ubuntu, CentOS i Fedora.

Za pomocą Docker Machine można tworzyć i zarządzać wirtualnymi maszynami wirtualnymi lub fizycznymi, które są hostami Docker. Pozwala to na łatwe rozpoczęwanie i zatrzymywanie kontenerów Docker, a także umożliwia korzystanie z narzędzi do zarządzania siecią i magazynem plików.

W skrócie, Docker Machine to narzędzie, które umożliwia łatwą instalację i zarządzanie środowiskiem Docker na różnych platformach i systemach operacyjnych. Najnowszą wersję i instrukcję instalacji znajdziesz w tym miejscu: <https://github.com/docker/machine/releases/>

W moim przypadku zainstalowałem oprogramowanie za pomocą poniższego polecenia:

```
curl -L https://github.com/docker/machine/releases/download/v0.16.2/docker-machine-`uname -s`-`uname -m` >/tmp/docker-machine &&
chmod +x /tmp/docker-machine &&
cp /tmp/docker-machine /usr/local/bin/docker-machine
```

Docker machine nie różni się znacznie pod względem dostępnych opcji od tych, które omawialiśmy w podstawowym dockerze. Jednak, jego zastosowanie jest zupełnie inne. Żeby to potwierdzić wystarczy wyświetlić dostępne opcje przy pomocy opcji **help**:

docker-machine help

Usage: docker-machine [OPTIONS] COMMAND [arg...]

Create and manage machines running Docker.

Version: 0.16.2, build bd45ab13

Author:

Docker Machine Contributors - <<https://github.com/docker/machine>>

```
Options:
--debug, -D                                     Enable debug mode
--storage-path, -s "/home/darki/.docker/machine"  Configures storage path [$MACHINE_STORAGE_PATH]
--tls-ca-cert                                    CA to verify remotes against [$MACHINE_TLS_CA_CERT]
--tls-ca-key                                     Private key to generate certificates [$MACHINE_TLS_CA_KEY]
--tls-client-cert                                Client cert to use for TLS [$MACHINE_TLS_CLIENT_CERT]
--tls-client-key                                  Private key used in client TLS auth [$MACHINE_TLS_CLIENT_KEY]
--github-api-token                               Token to use for requests to the Github API [$MACHINE_GITHUB_API_TOKEN]
--native-ssh                                      Use the native (Go-based) SSH implementation. [$MACHINE_NATIVE_SSH]
--bugsnag-api-token                            BugSnag API token for crash reporting [$MACHINE_BUGSNAG_API_TOKEN]
--help, -h                                         show help
--version, -v                                       print the version
```

Commands:

| | |
|---------|--|
| active | Print which machine is active |
| config | Print the connection config for machine |
| create | Create a machine |
| env | Display the commands to set up the environment for the Docker client |
| inspect | Inspect information about a machine |
| ip | Get the IP address of a machine |

| | |
|------------------|---|
| kill | Kill a machine |
| ls | List machines |
| provision | Re-provision existing machines |
| regenerate-certs | Regenerate TLS Certificates for a machine |
| restart | Restart a machine |
| rm | Remove a machine |
| ssh | Log into or run a command on a machine with SSH. |
| scp | Copy files between machines |
| mount | Mount or unmount a directory from a machine with SSHFS. |
| start | Start a machine |
| status | Get the status of a machine |
| stop | Stop a machine |
| upgrade | Upgrade a machine to the latest version of Docker |
| url | Get the URL of a machine |
| version | Show the Docker Machine version or a machine docker version |
| help | Shows a list of commands or help for one command |

Run 'docker-machine COMMAND --help' for more information on a command.

Wiele dostępnych opcji powtarza się i działa na tej samej zasadzie co w dockerze, którego już znamy. Na liście znajduje się możliwość tworzenia przy pomocy polecenia **create**, usunięcia **rm**, startu **start**, zatrzymania **stop** i wielu innych identycznych poleceń jak przy dockerze. Działanie jest identyczne. To znaczy służą do tego samego, te same polecenia co w klasycznym dockerze. Jednak to, co tworzymy teraz jest trochę inne. Nie chcę tłumaczyć tego na suchej teorii, dlatego przejdźmy do przykładów.

10.6. Pierwszy etap tworzenia elementów klastra

Mam nadzieję, że dość wnikliwie zapoznałeś się z teorią z początku tego rozdziału. Jest ona bardzo istotna, ponieważ będę używał tam poznaną wiedzę w sposób bardziej praktyczny.

Jak wiesz to, co robi **Docker Machine** możemy porównać do **tworzenia klastra**. Choć nie jest to porównanie, a dokładnie klaster. W tym klastrze stworzymy **menadżera** oraz trzy **węzły pracownicze**.

Zaczniemy od stworzenia menadżera:

```
docker-machine create --driver virtualbox --virtualbox-memory "2048" --virtualbox-hostonly-cidr  
192.168.56.1/21 manager  
Running pre-create checks...  
Creating machine...  
(manager) Copying /home/darki/.docker/machine/cache/boot2docker.iso to  
/home/darki/.docker/machine/machines/manager/boot2docker.iso...  
(manager) Creating VirtualBox VM...  
(manager) Creating SSH key...  
(manager) Starting the VM...  
(manager) Check network to re-create if needed...  
(manager) Waiting for an IP...  
Waiting for machine to be running, this may take a few minutes...  
Detecting operating system of created instance...  
Waiting for SSH to be available...  
Detecting the provisioner...  
Provisioning with boot2docker...  
Copying certs to the local machine directory...  
Copying certs to the remote machine...  
Setting Docker configuration on the remote daemon...  
Checking connection to Docker...  
Docker is up and running!  
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run:  
docker-machine env manager
```

Wprowadzone w powyższy sposób polecenie możemy przetłumaczyć na język polski w następujący sposób:

docker machine stwórz maszynę wirtualną o przydzielonej pamięci 2 GB w sieci z określonego zakresu i nazwij ją manager.

Po poprawnym zainicjowaniu wszystkich funkcji oraz uruchomieniu, jeżeli spojrzesz do **VirtualBox** powinieneś mieć dostępną maszynę jak na poniższym obrazku:



Natomiast jeżeli wypiszesz listę uruchomionych maszyn w dockerze również otrzymasz informacje o jej uruchomieniu.

docker-machine ls

| NAME | ACTIVE | DRIVER | STATE | URL | SWARM | DOCKER | ERRORS |
|---------|--------|------------|---------|---------------------------|-------|-----------|--------|
| manager | - | virtualbox | Running | tcp://192.168.56.103:2376 | | v19.03.12 | |

Chociaż polecenie identyczne jak w przypadku podstawowego dockera, to informacje jakie otrzymaliśmy trochę się różnią. Jednak otrzymana lista jest bardzo intuicyjna i wszystko to opisałem w fazie wywołania polecenia.

Natomiast z istotnych dwóch kolumn mamy **State**, w której znajduje się słowo **Running**. Oznacza to, że nasza maszyna jest uruchomiona i działa.

Następnie w kolumnie **URL** znajduje się **adres IP** jaki został przydzielony do danej maszyny. Jednak to jeszcze nie wszystko. Nim przejdziemy do dalszych szczegółów związanych z wynikiem tego przykładu stworzymy sobie jeszcze 3 węzły:

```
docker-machine create --driver virtualbox --virtualbox-memory "2048" --virtualbox-hostonly-cidr  
192.168.56.1/21 wezel-1
```

```
docker-machine create --driver virtualbox --virtualbox-memory "2048" --virtualbox-hostonly-cidr  
192.168.56.1/21 wezel-2
```

```
docker-machine create --driver virtualbox --virtualbox-memory "2048" --virtualbox-hostonly-cidr  
192.168.56.1/21 wezel-3
```

Wyświetlamy ponownie listę dostępnych urządzeń:

docker-machine ls

| NAME | ACTIVE | DRIVER | STATE | URL | SWARM | DOCKER | ERRORS |
|---------|--------|------------|---------|---------------------------|-------|-----------|--------|
| manager | - | virtualbox | Running | tcp://192.168.56.103:2376 | | v19.03.12 | |
| wezel-1 | - | virtualbox | Running | tcp://192.168.56.104:2376 | | v19.03.12 | |
| wezel-2 | - | virtualbox | Running | tcp://192.168.56.105:2376 | | v19.03.12 | |
| wezel-3 | - | virtualbox | Running | tcp://192.168.56.106:2376 | | v19.03.12 | |

W **VirtualBox** pojawią się również tworzone obecnie przez nas węzły, tak jak pojawił się manager. W przypadku tworzonych maszyn wszystkie operacje będziemy wykonywać z pozycji wiersza poleceń.

Zobaczmy co stanie się jak maszyna naszego menadżera zostanie zatrzymana:

docker-machine stop manager

```
Stopping "manager"...
```

```
Machine "manager" was stopped.
```

Wypiszmy listę maszyn:

docker-machine ls

| NAME | ACTIVE | DRIVER | STATE | URL | SWARM | DOCKER | ERRORS |
|---------|--------|------------|---------|---------------------------|-------|-----------|--------|
| manager | - | virtualbox | Stopped | | | Unknown | |
| wezel-1 | - | virtualbox | Running | tcp://192.168.56.104:2376 | | v19.03.12 | |
| wezel-2 | - | virtualbox | Running | tcp://192.168.56.105:2376 | | v19.03.12 | |
| wezel-3 | - | virtualbox | Running | tcp://192.168.56.106:2376 | | v19.03.12 | |

Pod kolumną **state** nasza maszyna **manager** jest określona jako zatrzymana. Jeśli zerkniesz na **VirtualBox** to tam też wystąpiła ta sama czynność. To, co wykonaś przy pomocy **docker-machine** będzie miało wpływ na wirtualne maszyny w **VirtualBox**. W zakresie obsługi **docker-machine** oraz **VirtualBox** są ze sobą połączone, a oprogramowaniem do administrowania i wprowadzania zmian jest **docker-machine** w przypadku obydwu programów. W celu uruchomienia **kontenera, przepraszam maszyny wirtualnej, manager** używamy polecenia:

docker-machine start manager

Zostanie uruchomiony wcześniej zamknięty manager. Inną funkcją, którą powinieneś pamiętać z klasycznego dockera jest opcja **inspect**. Wykonajmy tak zwaną inspekcję naszego managera:

docker-machine inspect manager

Otrzymasz wszelkie informacje dotyczące tego **kontenera**. Jeżeli chcesz szybko uzyskać informację o przydzielonym adresie IP wystarczy użyć polecenia w prezentowanej poniżej formie:

docker-machine ip manager

```
192.168.56.103
```

Choć w wygodny sposób możesz to zrobić przy pomocy polecenia **ls**, to przy sporej ilości węzłów może okazać się szybszym rozwiązaniem.

10.7. Drugi etap tworzenie klastra

Nie wiem czy zauważłeś, ale elementy które stworzyliśmy nie współpracują w żaden sposób ze sobą. Są tak jakby oddzielnymiinstancjami. Są jedynie uruchomione. Tak naprawdę nie mają one z sobą jeszcze żadnego połączenia, są oddzielnymi maszynami wirtualnymi będącymi w tej samej sieci. Na tym etapie zajmiemy się konstruowaniem klastra ze stworzonych przed chwilą elementów.

10.7.1. Połaczenie ssh w przypadku ręcznej instalacji

Połaczenie ssh w przypadku instalacja i konfiguracja ręcznej w VirtualBox. Po wykonaniu powyższych czynności przy pomocy ssh logujemy się do każdej z maszyn:

```
ssh <nazwa-uzytkownika>@192.168.56.100  
ssh <nazwa-uzytkownika>@192.168.56.101  
ssh <nazwa-uzytkownika>@192.168.56.102  
ssh <nazwa-uzytkownika>@192.168.56.103
```

10.7.2. Połaczenie ssh z użyciem docker-machine

Do stworzenia oraz administracji naszych maszyn skorzystamy z możliwości połączenia przy pomocy **ssh**. Posiadamy jednego menadżera oraz trzy węzły. Do każdego z wymienionych chcemy się połączyć wymienioną metodą. W związku z tym istnieje konieczność uruchomienia kilku terminali. Chyba, że zainstalujecie oprogramowanie **terminator**. Jest to nakładka na terminal umożliwiająca podział naszej konsoli na kilka elementów. Oznacza to, że nie musimy uruchamiać kilku tylko dzielimy już uruchomioną konsolę na kilka części. Możemy to robić zarówno w pionie jak i w poziomie. Przy konieczności połączenia się ze wszystkimi elementami klastra taka funkcja bardzo się przyda. Aby połączyć się z naszymi maszynami wywołujemy polecenie:

```
docker-machine ssh manager  
( '>' )  
/) TC (\ Core is distributed with ABSOLUTELY NO WARRANTY.  
(/_--_\\) www.tinycorelinux.net
```

10.7.3. Dalsza konfiguracja niezależnie od wersji

Zostaniesz połączony ze wskazaną **maszyną wirtualną**. W ten sam sposób robimy z pozostałymi elementami naszego **klastra**. Zmieniamy oczywiście **manager** na nazwę jaką nadaliśmy naszym węzłom. Jeżeli korzystasz z **terminatora** wystarczy, że podzielisz go na cztery elementy tak jak w poniższym przykładzie:

The image shows four terminal windows side-by-side, each displaying a command-line interface for a Docker machine. The top-left window shows a command to start a 'manager' node. The other three windows show commands to start 'wezel-1', 'wezel-2', and 'wezel-3' nodes respectively. Each window displays a standard tinycorelinux net boot message.

```

darki@noishacking: ~ 76x18
darki@noishacking:~$ docker-machine ssh manager
( '>')
/) TC (\ Core is distributed with ABSOLUTELY NO WARRANTY.
(/_ _ _\_) www.tinycorelinux.net
docker@manager:~$ 

darki@noishacking: ~ 73x18
darki@noishacking:~$ docker-machine ssh wezel-1
( '>')
/) TC (\ Core is distributed with ABSOLUTELY NO WARRANTY.
(/_ _ _\_) www.tinycorelinux.net
docker@wezel-1:~$ 

darki@noishacking: ~ 76x20
darki@noishacking:~$ docker-machine ssh wezel-2
( '>')
/) TC (\ Core is distributed with ABSOLUTELY NO WARRANTY.
(/_ _ _\_) www.tinycorelinux.net
docker@wezel-2:~$ 

darki@noishacking: ~ 73x20
darki@noishacking:~$ docker-machine ssh wezel-3
( '>')
/) TC (\ Core is distributed with ABSOLUTELY NO WARRANTY.
(/_ _ _\_) www.tinycorelinux.net
docker@wezel-3:~$ 

```

W przypadku jeżeli nie chcesz korzystać ze wspomnianego oprogramowania musisz uruchomić cztery oddzielne terminale, lub cztery oddzielne zakładki i w nich połączyć się z każdym z elementów.

10.7.4. Ustawienie menadżera

Pomimo tego, że jednemu z uruchomionych maszyn nadaliśmy nazwę manager, to jednak nie jest on nim jak nazwa sugeruje. My określiliśmy mu tylko nazwę pod jaką występuje. Natomiast nie określiliśmy funkcji jaką ma pełnić. W celu określania jej, będąc w terminalu w którym połączylismy się z nim, wprowadzamy polecenie (zmieniasz tylko adres IP na swój):

docker swarm init --advertise-addr 192.168.56.103

Swarm initialized: current node (ve2ivo154s3vt83yy6tpqirx4) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join --token SWMTKN-1-Oghh5rze74h5je0n21hnecy6nwv1agtdxanfytwua8hvz2lutm-f3b38laq42dwheec6n8sstwk4 192.168.56.103:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

W ten sposób został wygenerowany indywidualny **token**, czyli klucz. Dzięki temu kluczowi jesteśmy w stanie dodać do naszego klastra węzły, które będą wykonywały wskazane im zadania. Token jest dość

skomplikowany w zapisie. W związku z tym nie uda się w łatwy sposób go zapamiętać. Oczywiście możemy sobie gdzieś go zapisać, ale zawsze może się zdarzyć, że po prostu zostanie zgubiony. Aby odzyskać utracony token wystarczy, skorzystać z polecenia:

docker swarm join-token worker

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-Oghh5rze74h5je0n21hnecy6nwv1agtdxanfytwua8hvz2lutm-f3b38laq42dwheec6n8sstwk4 192.168.56.103:2377
```

10.7.5. Dodanie węzłów wykonawczych

Teraz dzięki kluczowi możemy do naszego klastra dodać poszczególne węzły. Poniższe polecenie wprowadzamy do wszystkich urządzeń, które chcemy dodać po wcześniejszym połączeniu się przy pomocy **ssh**:

docker swarm join --token SWMTKN-1-

```
Oghh5rze74h5je0n21hnecy6nwv1agtdxanfytwua8hvz2lutm-f3b38laq42dwheec6n8sstwk4  
192.168.56.103:2377
```

This node joined a swarm as a worker.

Otrzymaliśmy informacje, że ten węzeł został dodany do naszego klastra jako *pracownik*. W ten sam sposób postępujemy z pozostałymi dwoma.

10.8. Wyświetlanie zawartości naszego klastra

Stworzyliśmy **menadżera** oraz trzy **węzły pracowników**. Następnie połączymy wszystko w jeden **klaster**. Aby wyświetlić listę urządzeń naszego klastra w maszynie będącej menadżerem wprowadzamy polecenie:

docker node ls

| ID | HOSTNAME | STATUS | AVAILABILITY | MANAGER STATUS | ENGINE | VERSION |
|-----------------------------|----------|---------|--------------|----------------|----------|----------|
| ve2ivo154s3vt83yy6tpqirx4 * | | manager | Ready | Active | Leader | 19.03.12 |
| q7f018wr0sq0jt24j6co9p1pt | wezel-1 | Ready | Active | | 19.03.12 | |
| iej9fq9ptpgxj5id79tww52 | wezel-2 | Ready | Active | | 19.03.12 | |
| httax6laurhpbjw9wsfasv6s | wezel-3 | Ready | Active | | 19.03.12 | |

Zwróć uwagę, że na powyższej liście mamy dokładnie określonego lidera, którego ustawiliśmy w jednym z poprzednich paragrafów. Wydaje mi się, że w związku z prezentowaną listą nie ma nic więcej do dopowiedzenia.

Możemy jeszcze uzyskać szczegółowe informacje o danej maszynie przy pomocy inspekcji. Aby dokonać tego na menadżerze wystarczy że posłużymy się poleceniem:

docker node inspect --pretty self

ID: ve2ivo154s3vt83yy6tpqirx4

Hostname: manager

Joined at: 2022-11-18 07:43:28.482160526 +0000 utc

Status:

State: Ready

Availability: Active

Address: 192.168.56.103

Manager Status:

Address: 192.168.56.103:2377

Raft Status: Reachable

Leader: Yes

Platform:

Operating System: linux

Architecture: x86_64

Resources:

CPUs: 1

Memory: 1.947GiB

Plugins:

Log: awslogs, fluentd, gcplogs, gelf, journald, json-file, local, logentries, splunk, syslog

Network: bridge, host, ipvlan, macvlan, null, overlay

Volume: local

Engine Version: 19.03.12

Engine Labels:

- provider=virtualbox

TLS Info:

TrustRoot:

-----BEGIN CERTIFICATE-----

MII BjCCARCgAwIBAgIUdCBFoqPZMdsQmzUf+ZMuwjpalu8wCgYIKoZIzj0EAwIw

EzERMA8GA1UEAxMIc3dhcm0tY2EwHhcNMjIxMTE0MDczODAwWhcNNDIxMTA5MDcz

ODAwWjATMREwDwYDVQQDEwhzd2FybS1jYTBZMBMGBYqGSM49AgEGCCqGSM49AwEH

A0IABFoMIZLEX41xFuyCUfZ5Eo95xerDeygUM8p/Pq98PWw1G/ojry2MXdo/LSWG
u3s9hshl0MvxQdOBI1HQC7VxVvajQjBAMA4GA1UdDwEB/wQEAvIBBjAPBgNVHRMB
Af8EBTADAQH/MB0GA1UdDgQWBBQdczXVp3240OeBduODtLzm7G1MQjAKBggqhkJ0
PQQDAgNIADBFAiEA1BrIGyYEMqVEtZ+NxFP1fNdCIzKZQK4odvKLJnpOURkCIDwd
IhXnZeVD39zV3Ifb+3sliFKU4hrg54xxdCTXcqdV
-----END CERTIFICATE-----

Issuer Subject: MBMxEТАPBgNVBAMTCHN3YXJtLWNh

Issuer Public Key:

MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEWgyVksRfjXEW7IJR9nkSj3nF6sN7KBQzyn8+r3w9bDUb+
iOvLYxd2j8tJYa7ez2GyGXQy/FB04GXUdALTXFW9g==

Skorzystałem z dodatkowej opcji **--pretty** ponieważ lista wyświetla się w znacznie czytelniejszy sposób. W przypadku inspekcji maszyny, w której się znajdujemy korzystamy również z nazwy **self** zamiast nazwy pod jaką występuje. Jeżeli chcesz to zamiast **self** możesz wprowadzić dowolną nazwę maszyny którą chcesz poddać inspekcji.

10.9. Tworzenie i uruchamianie serwisu

Po zapoznaniu się z dotychczasowymi elementami nadszedł czas na poznanie sposobu wykorzystania naszego klastra. Uruchomimy w nim serwer **apache2**. Nim jednak opiszę to wszystko zerknijmy na polecenie, dzięki któremu to osiągniemy (polecenie wykonujemy będąc zalogowanym do managera):

```
docker service create --name serwer -p 8080:80 --replicas 4 httpd:latest
ixapw5sc2wbq395tmktg6unb8
overall progress: 4 out of 4 tasks
1/4: running
2/4: running
3/4: running
4/4: running
verify: Service converged
```

Zwróć uwagę na jego składnię. Możliwe, że Cię nie zaskoczyła jakimiś ogromnymi zmianami w porównaniu do tego, co wykonywaliśmy dotychczas. Nowością jest **service**. To dzięki niemu powstanie nasz serwis oraz dzięki słowu **create** występującym po nim. Nasz serwis musimy odpowiednio nazwać, dlatego użyłem polskiego słowa serwer. Jednak powinno się używać anglojęzycznych nazw. My się

uczymy, dlatego wydaje mi się łatwiej będzie zrozumieć z użyciem polskich nazw. Wracając do polecenia, serwis udostępniamy na porcie **80**, który występuje w **4 replikach**. Natomiast obraz z jakiego będziemy korzystali to najnowszy **httpd** czyli **apache2**.

Z całego polecenia chciałbym skupić się na **replikach**. To w tym miejscu informujesz **Docker Swarm** o tym z ilu twój serwer ma się składać.

Aby wyświetlić listę serwisów przez nas stworzonych korzystamy z polecenia:

| docker service ls | | | ID | NAME | MODE |
|-------------------|--------|------------|-----|--------------|----------------|
| REPLICAS | IMAGE | POROS | | | |
| ixapw5sc2wbq | serwer | replicated | 4/4 | httpd:latest | *:8080->80/tcp |

Wynik nie powinien nas zaskoczyć. Wszystko jest podobne jak w przypadku podstawowych zasad dockera, które poznawaliśmy przez cały ten kurs. Jedyną nowością jest kolumna **REPLICAS**, w której wypisana jest używana ilość replik.

Teraz możemy chcieć wyświetlić listę samych replik oraz maszyn do jakich zostały przydzielone. Do tego służy polecenie:

| docker service ps serwer | | | ID | NAME |
|--------------------------|----------|---------------|---------------|--------------------|
| IMAGE | NODE | DESIRED STATE | CURRENT STATE | ERROR |
| PORTS | | | | |
| xsntw68x3agx | serwer.1 | httpd:latest | manager | Running |
| ago | | | | Running 13 minutes |
| kg486s6p8hy6 | serwer.2 | httpd:latest | wezel-1 | Running |
| ago | | | | Running 13 minutes |
| gopoaclejyk2 | serwer.3 | httpd:latest | wezel-2 | Running |
| ago | | | | Running 13 minutes |
| 118yr2i3s24u | serwer.4 | httpd:latest | wezel-3 | Running |
| ago | | | | Running 13 minutes |

Zgodnie z zasadami o jakich do tej pory rozmawialiśmy wszystko się zgadza. Powstały cztery repliki **httpd-latest**, które zostały rozdzielone pomiędzy cztery dostępne maszyny. Po jednej do każdej z maszyn.

Stworzony przez nas serwer możemy też poddać tak zwanej inspekcji. Wykonujemy to przy pomocy następującego polecenia:

```
docker service inspect serwer
```

```
[  
  {  
    "ID": "ixapw5sc2wbq395tmktg6unb8",  
    "Version": {  
      "Index": 27  
    },  
    "CreatedAt": "2022-11-18T09:26:23.126873074Z",  
    "UpdatedAt": "2022-11-18T09:26:23.127689806Z",  
    "Spec": {  
      "Name": "serwer",  
      "Labels": {},  
      "TaskTemplate": {  
        "ContainerSpec": {  
          "Image":  
            "Init": false,  
            "StopGracePeriod": 10000000000,  
            "DNSConfig": {},  
            "Isolation": "default"  
          },  
          "Resources": {  
            "Limits": {},  
            "Reservations": {}  
          },  
          "RestartPolicy": {  
            "Condition": "any",  
            "Delay": 5000000000,  
            "MaxAttempts": 0  
          },  
          "Placement": {  
            "Platforms": [  
              {  
                "Architecture": "amd64",  
                "OS": "linux"  
              },  
              {  
                "OS": "linux"  
              }  
            ]  
          }  
        }  
      }  
    }  
  }  
]
```

```
        },
        {
          "OS": "linux"
        },
        {
          "Architecture": "arm64",
          "OS": "linux"
        },
        {
          "Architecture": "386",
          "OS": "linux"
        },
        {
          "Architecture": "mips64le",
          "OS": "linux"
        },
        {
          "Architecture": "ppc64le",
          "OS": "linux"
        },
        {
          "Architecture": "s390x",
          "OS": "linux"
        }
      ],
    },
    "ForceUpdate": 0,
    "Runtime": "container"
  },
  "Mode": {
    "Replicated": {
      "Replicas": 4
    },
    "UpdateConfig": {
      "Parallelism": 1,
      "FailureAction": "pause",
      "Monitor": 5000000000,
      "MinReadyTime": 0
    }
  }
}
```

```

    "MaxFailureRatio": 0,
    "Order": "stop-first"
  },
  "RollbackConfig": {
    "Parallelism": 1,
    "FailureAction": "pause",
    "Monitor": 5000000000,
    "MaxFailureRatio": 0,
    "Order": "stop-first"
  },
  "EndpointSpec": {
    "Mode": "vip",
    "Ports": [
      {
        "Protocol": "tcp",
        "TargetPort": 80,
        "PublishedPort": 8080,
        "PublishMode": "ingress"
      }
    ]
  }
},
"Endpoint": {
  "Spec": {
    "Mode": "vip",
    "Ports": [
      {
        "Protocol": "tcp",
        "TargetPort": 80,
        "PublishedPort": 8080,
        "PublishMode": "ingress"
      }
    ]
  }
},
"Ports": [
  {
    "Protocol": "tcp",
    "TargetPort": 80,

```

```

    "PublishedPort": 8080,
    "PublishMode": "ingress"
}
],
"VirtualIPs": [
{
    "NetworkID": "ocblig6rarrjh7dp1vq4shnhm",
    "Addr": "10.0.0.6/24"
}
]
}
]

```

Otrzymujemy podobne informacje tak jak w przypadku standardowego kontenera. Będziemy potrzebowali teraz adresy IP naszych maszyn klastra. Aby to zrobić, musisz być zalogowany do managera lub jednego z węzłów.

sudo docker node inspect self --format '{{ .Status.Addr }}' <- wyświetli adres noda w którym polecenie wpiszesz

sudo docker node inspect <nazwa-hosta-wezla> --format '{{ .Status.Addr }}' <- wyświetla adres wskazanego noda np.

sudo docker node inspect wezel-1 --format '{{ .Status.Addr }}' <- wypisze adres noda o nazwie hosta worker1.

Teraz chciałbym, abyś włączył przeglądarkę. Następnie otworzył cztery karty w niej i w każdej wprowadził adres IP każdej z maszyn. W moim przypadku są to:

```

http://192.168.56.103:8080/
http://192.168.56.104:8080/
http://192.168.56.105:8080/
http://192.168.56.106:8080/

```

Jeżeli otworzysz którykolwiek z powyższych adresów zostanie wyświetlona informacja *IT Works!* co oznacza, że nasz serwer **apache2** został uruchomiony. Może samo uruchomienie nas nie zaskakuje jednak zwróć uwagę, że *to samo* zostało uruchomione w 4 różnych instancjach. W związku z

powyższym mamy stworzony serwer, który posiada cztery repliki. Gdy jedna z nich przestanie działać nie będzie miało to wpływu na samo funkcjonowanie naszego serwera.

10.10. Odłączenie jednej z maszyn

Co się stanie jeśli jedno z urządzeń przestanie działać, zostanie odłączone lub po prostu nie będzie dostępne? Zgodnie z tym co przeczytaliśmy na początku tego materiału maszyna, która przestała w jakiś sposób działać powinna zostać zastąpiona przez inną. Sprawdźmy czy tak się stanie:

```
docker node update --availability drain wezel-3           wezel-3
```

Przy pomocy powyższego polecenia wprowadzamy nasz **wezel-3** w stan tak zwanego uśpienia. Oznacza to, że nie został on usunięty czy też wyłączony jednak nie może przyjmować ani wykonywać żadnych zadań. Zobaczmy, co teraz się stało w przypadku zadania które mu powierzono:

```
docker service ps serwer
```

| ID | NAME | IMAGE | NODE | DESIRED STATE | CURRENT STATE |
|-----------------------------|-------------|--------------|---------|---------------|---------------|
| ERROR | PORTS | | | | |
| xsntw68x3agx minutes ago | serwer.1 | httpd:latest | manager | Running | Running 52 |
| kg486s6p8hy6 minutes ago | serwer.2 | httpd:latest | wezel-1 | Running | Running 52 |
| gopoaclejyk2 minutes ago | serwer.3 | httpd:latest | wezel-2 | Running | Running 52 |
| ycl4t657zhxn minutes ago | serwer.4 | httpd:latest | manager | Running | Running 3 |
| 118yr2i3s24u minutes ago | _ serwer.4 | httpd:latest | wezel-3 | Shutdown | Shutdown 3 |

Zwróć uwagę na ostatni wynik. **Desired State** jest określony jako **Shutdown** czyli zamknięty. Niech Cię to nie zwiedzie. Ten węzeł działa jednak nie obsługuje i nie przyjmuje żadnych zadań. Dowodem tego, jest to że jesteś przy pomocy **ssh** z nim połączony.

Zadanie jakie wykonywał zostało przejęte przez menedżera, o czym świadczy przedostatni wynik. Natomiast pamiętasz pewnie, że replika **managera** była dostępna pod adresem <http://192.168.56.103:8080/>, natomiast **wezla-3** <http://192.168.56.106:8080/>. W przeglądarce uruchom ponownie te dwa adresy. Zarówno jeden, jak i drugi działa. Oznacza to, że zadania jakie wykonywał **wezel-3** zostały przekazane do **managera** łącznie z obsługą jego adresu IP.

10.11. Aktualizacja obrazu

Pomimo używanego obrazu możemy go w bardzo łatwy sposób zmienić. Korzystamy z najnowszej wersji obrazu **httpd** czyli **apache2**. Jeżeli musimy w jakiś sposób zmienić wersję obrazu, z którego korzystamy jesteśmy w stanie tego dokonać w bardzo łatwy sposób. Jednak zanim tego dokonamy zobaczymy jak wygląda to obecnie:

```
docker service inspect --pretty serwer
```

ContainerSpec:

Image:

httpd:latest@sha256:5fa96551b61359de5dfb7fd8c9e97e4153232eb520a8e883e2f47fc80dbfc33e

Init: false

W powyższym przykładzie pokazuję tylko część, która nas obecnie interesuje. Zgodnie z informacją z przykładu korzystamy z najnowszego obrazu. Dokonajmy teraz zmiany z najnowszego na **alpine**:

```
docker service update --image httpd:alpine serwer
```

serwer

overall progress: 4 out of 4 tasks

1/4: running

2/4: running

3/4: running

4/4: running

verify: Service converged

Obraz powinien zostać podmieniony. Sprawdźmy to:

```
docker service inspect --pretty serwer
```

ContainerSpec:

Image:

httpd:alpine@sha256:4658c554fe215c8a17d57adbd9f8595e4922abda40d1e67a276456d2f142400e

Init: false

Zgodnie z informacją obraz został zamieniony. Jeżeli chcesz, sprawdź czy wszystko działa wprowadzając adresy z poprzedniego paragrafu.

Jednak pamiętaj, że zmiana obrazu nie przywróci do działania węzła. Jak pamiętasz w ostatnim paragrafie **wezel-3** stał się niedostępny. Pomimo obrazu nadal tak jest:

docker node ls

| ENGINE VERSION | ID | HOSTNAME | STATUS | AVAILABILITY | MANAGER STATUS |
|-----------------------------|---------|----------|--------|--------------|----------------|
| ve2ivo154s3vt83yy6tpqirx4 * | manager | Ready | Active | Leader | 19.03.12 |
| q7f018wr0sq0jt24j6co9p1pt | wezel-1 | Ready | Active | | 19.03.12 |
| iej9fq9ptpgxjt5id79tww52 | wezel-2 | Ready | Active | | 19.03.12 |
| hztax6laurhpbjw9swsfavs6s | wezel-3 | Ready | Drain | | |

Aby ponownie działał musiał go aktywować w następujący sposób:

docker node update --availability active wezel-3

Teraz będzie on już aktywny:

docker node ls

| ID | HOSTNAME | STATUS | AVAILABILITY | MANAGER STATUS | ENGINE |
|-----------------------------|----------|--------|--------------|----------------|----------|
| VERSION | | | | | |
| ve2ivo154s3vt83yy6tpqirx4 * | manager | Ready | Active | Leader | 19.03.12 |
| q7f018wr0sq0jt24j6co9p1pt | wezel-1 | Ready | Active | | 19.03.12 |
| iej9fq9ptpgxjt5id79tww52 | wezel-2 | Ready | Active | | 19.03.12 |
| hztax6laurhpbjw9swsfavs6s | wezel-3 | Ready | Active | | 19.03.12 |

10.12. Usuwanie klastra

Przyczyn usunięcia klastra i jego elementów może być kilka. Nie chcę się zagłębiać nad samą przyczyną wykonania tych czynności tylko je wykonać. Pierwsze elementy do usunięcia są to trzy węzły, które stworzyliśmy. W każdym z nich musimy wpisać:

docker swarm leave

Node left the swarm.

Otrzymamy informację o tym, że dany węzeł opuścił nasz klaster. Po wykonaniu tej czynności możemy przystąpić do ich usunięcia. Będąc zalogowanym w manager wpisujemy polecenie:

docker node rm wezel-1

docker node rm wezel-2

docker node rm wezel-3

Jeżeli wyświetlisz listę dostępnych węzłów będzie na niej tylko manager. Natomiast jeżeli wyświetlisz listę replik zauważ, że wszystkie zadania zostały przekazane jemu. Po usunięciu węzłów, pozostaje usunąć nasz serwer:

docker service rm serwer

10.13. Usuwanie pozostałości z docker-machine

Następnie wylogowujemy się z każdego węzła oraz menadżera. Lista elementów pozostała nienaruszona. My tylko odłączyliśmy węzły oraz usunęliśmy zawartość. Natomiast nadal istnieją one co potwierdza lista:

docker-machine ls

| NAME | ACTIVE | DRIVER | STATE | URL | SWARM | DOCKER | ERRORS |
|---------|--------|------------|---------|---------------------------|-------|-----------|--------|
| manager | - | virtualbox | Running | tcp://192.168.56.103:2376 | | v19.03.12 | |
| wezel-1 | - | virtualbox | Running | tcp://192.168.56.104:2376 | | v19.03.12 | |
| wezel-2 | - | virtualbox | Running | tcp://192.168.56.105:2376 | | v19.03.12 | |
| wezel-3 | - | virtualbox | Running | tcp://192.168.56.106:2376 | | v19.03.12 | |

W związku z tym musimy zatrzymać każdy z elementów:

```
docker-machine stop manager  
docker-machine stop wezel-1  
docker-machine stop wezel-2  
docker-machine stop wezel-3
```

Po zatrzymaniu każdego z elementów pozostaje ich trwałe usunięcie:

```
docker-machine rm manager wezel-1 wezel-2 wezel-3
```

About to remove manager, wezel-1, wezel-2, wezel-3

WARNING: This action will delete both local reference and remote instance.

Are you sure? (y/n): y

Successfully removed manager

Successfully removed wezel-1

Successfully removed wezel-2

Successfully removed wezel-3

Jeśli zerkniesz zarówno na **VirtualBox**, jak i na listę dostępnych maszyn będzie ona pusta. Wszystkie elementy zostały usunięte.

10.14. Podsumowanie

Docker Swarm jest bardzo rozbudowanym narzędziem. Po zapoznaniu się z tym materiałem możesz stwierdzić, że zawiera on wszelkie możliwości jakie do tej pory poznaleś, ale wszystko to stosujemy na klastrze. I jeżeli doszedłeś do tego wniosku to wiedz, że tak właśnie jest.

Rozdział 11: Instalacja Kubernetes w dystrybucjach Red Hat Enterprise Linux i Debian

RHEL i **Debian** są najpopularniejszymi dystrybucjami Linuksa na świecie. Dlatego też przeprowadzenie instalacji narzędzia, jakim jest **Kubernetes**, wybrałem na tych dwóch dystrybucjach.

11.1. Red Hat Enterprise Linux

Wyróżnia się niezawodnością oraz bardzo długim okresem gwarancyjnym producenta. Niemniej jednak ze względu na ten tak zwany długoterminowy support jest on dystrybucją komercyjną. Czyli, aby korzystać z **RHEL**, należy opłacić subskrypcję. Oczywiście istnieje kilka alternatyw opartych o **RHEL** jak na przykład **Fedora**, **CentOS Stream**. Niestety w tych przypadkach, jak to niektórzy użytkownicy mawią, *to nie to samo*. Niestety konieczność opłacenia subskrypcji wielu użytkownikom uniemożliwia korzystanie z tej dystrybucji. Większość nie chce wdawać się w koszty, jeżeli mogą korzystać za darmo, tak jak w przypadku większości innych. Jednak nie wszyscy wiedzą, że firma **Red Hat** posiada swoje **wersje developerskie**, które **dostępne są całkowicie za darmo**. Jedyne co jest wymagane to rejestracja.Więcej o tej inicjatywie możesz poczytać pod adresem <https://developers.redhat.com/articles/faqs-no-cost-red-hat-enterprise-linux#general>. Natomiast sama rejestracja znajduje się na stronie <https://developers.redhat.com/register>.

Na dzień dzisiejszy każdy może wypróbować RHEL bez ponoszenia jakichkolwiek kosztów. Musisz jednak zdawać sobie sprawę, że jest to wersja developerska, czyli tak zwana testowa. Dlatego nie jest tak niezawodna, jak jej wersja płatna. Dlatego nim przystąpicie do instalacji, poczytajcie trochę o tej inicjatywie.

11.2. Debian

Jest to druga najpopularniejsza dystrybucja Linux. Tak samo, jak RHEL jest znana ze swojego podejścia do bezpieczeństwa oraz niezawodności. Jednak Debian jest dystrybucją całkowicie darmową. **Nie posiada wersji czy też subskrypcji, za którą tak jak w przypadku RHEL musisz zapłacić**. Wspierany jest przez społeczeństwo. Takie rozwiązanie ma swoje plusy, jak i minusy. Choć użytkowników jest bardzo dużo i w większości tematów możesz uzyskać pomoc za darmo, to jednak prawda jest taka, że przy płatnej subskrypcji RHEL-a w razie problemów masz jedno miejsce, w którym możesz zapytać o wszystko. W przypadku systemu Debian zdarzyć się może, żeby o bardziej techniczne sprawy zapytać, będziesz musiał poszukać jakiejś grupy czy to na Facebooku, czy też w innym miejscu. Oczywiście

istnieje również support płatny, jednak jest on prowadzony nie przez autorów Debiana, a przez prywatne osoby.

Pomimo braku oficjalnego supportu, Debian jest liderem wśród dystrybucji linuksowych. Od samego początku jego podejście co do wersji stabilnej jest bardzo rygorystyczne. W nim znajdują się programy często określane jako stare. Jednak wiąże się to z bezpieczeństwem. W tej wersji nie wychodzą aktualizacje wersji programów jedynie ich łatki bezpieczeństwa, jak i również łatki bezpieczeństwa systemu. Przez takie podejście jedni chwalą, drudzy krytykują. Jednak to, w której Ty się grupie znajdujesz, jest kwestią twojego podejścia.

W związku z popularnością oraz ogromnym podobieństwem co do instalacji Kuberneta w dystrybucjach pokrewnych poniżej opisuje sposób zarówno w jednej, jak i drugiej tak aby użyć dwa różniące się od siebie menadżery pakietów.

11.3. Instalacja Dockera

Pierwszą czynnością, jaką należy wykonać to odinstalować niepotrzebne oprogramowanie powiązane z **Dockerem** i **Podmanem**. Następnie dodać repozytorium i zainstalować **Dockera**.

11.3.1. Instalacja w systemie Red Hat Enterprise Linux

Powysze czynności wykonujemy przy pomocy polecenia:

```
sudo yum remove docker docker-client docker-client-latest docker-common docker-latest docker-latest-logrotate docker-logrotate docker-engine podman runc
```

W kolejnym kroku instalujemy paczkę, która umożliwia dodanie repozytorium:

```
sudo yum install -y yum-utils
```

Następnie dodajemy wspomniane repozytorium. I w tym miejscu powstaje pewien problem, ponieważ **oficjalne repozytorium do RHEL nie przewiduje wsparcia dla systemów 64-bitowych**. Wszystkie komputery są jednak w obecnych czasach wyposażone w procesory oraz systemy 64-bitowe. Można zainstalować wersję 32-bitową. Nie wprowadzi to większych zmian, ponieważ pomiędzy wersjami główna różnica polega na szybkości. Jednak ja chcę zainstalować wersję 64-bitową. **W związku z tym zainstalujemy Dockera przeznaczonego dla systemu CentOS Stream**. Dlatego dodajemy repozytorium przeznaczone do wspomnianego systemu w następujący sposób:

```
sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
```

I teraz możemy przejść do instalacji **Dockera**:

```
sudo yum install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

W kolejnym kroku uruchamiamy usługę zainstalowanego oprogramowania:

```
sudo systemctl start docker
```

Jeżeli chcemy, aby Docker automatycznie uruchamiał się przy starcie systemu, korzystamy z polecenia:

```
sudo systemctl enable docker
```

11.3.2. Instalacja w systemie Debian

Taką instalację przeprowadziliśmy w początkowej części szkolenia, dlatego aby się nie powtarzać, jeżeli nie masz zainstalowanego Dockera, to zajrzyj tam.

11.3.3. Dodanie do grupy oraz sprawdzenie czy Docker działa

Pozostało dodać nasze konto do grupy dockera tak, aby wszystko wykonywać bez uprawnień administracyjnych:

```
sudo usermod -aG docker $USER && newgrp docker
```

Sprawdzamy, czy prawidłowo zainstalowaliśmy Dockera, uruchamiając polecenie bez uprawnień konta root:

```
docker run hello-world
```

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

2db29710123e: Pull complete

Digest: sha256:18a657d0cc1c7d0678a3fbea8b7eb4918bba25968d3e1b0adef71caddbc346

Status: Downloaded newer image for hello-world:latest

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/get-started/
```

Po otrzymaniu powyższego komunikatu mamy potwierdzenie, że wszystko zostało prawidłowo zainstalowane i skonfigurowane.

11.4. Instalacja kubectl

Po poprawnym zainstalowaniu Dockera przyszła pora na instalację kubectl. Jest to nic innego jak narzędzie wiersza poleceń Kuberneta. Pozwoli na uruchamianie poleceń w klastrach. To dzięki temu programowi będziesz mógł wdrażać aplikacje oraz wykonywać inne niezbędne czynności z tym związane.

11.4.1. Instalacja kubectl w Red Hat Enterprise Linux

W tym przypadku podobnie jak przy Dockerze dodajemy odpowiednie repozytorium:

```
cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-\$basearch
enabled=1
gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
```

Po dodaniu repozytorium przystępujemy do instalacji:

```
sudo yum install -y kubectl
```

I w ten sposób mamy już dostępne narzędzie w linii poleceń.

11.4.2. Instalacja kubectl w Debian

Pierwszą czynność, jaką wykonujemy to aktualizacja repozytoriów oraz instalacja niezbędnych pakietów do Kuberntesesa:

```
sudo apt update && sudo apt install -y ca-certificates curl
```

Następnie pobieramy publiczny klucz **Google Cloud**:

```
sudo mkdir /etc/apt/keyrings && sudo curl -fsSLo /etc/apt/keyrings/kubernetes-archive-keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg
```

Dodajemy repozytorium:

```
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

Po dodaniu repozytorium aktualizujemy i instalujemy **kubectl**:

```
sudo apt update && sudo apt install -y kubectl
```

11.4.3. Weryfikacja kubectl

W celu weryfikacji zainstalowanego oprogramowania wpisujemy jego nazwę:

kubectl

Jeżeli pojawiła się lista dostępnych opcji polecenia, program został zainstalowany prawidłowo.

11.5. Minikube

Minikube jest to tak zwany lokalny **Kubernetes**, dzięki któremu łatwiej poznasz i nauczysz się używać tego narzędzia. To z jego powodu zainstalowaliśmy **Dockera**. Korzystamy z **minikube**, ponieważ za jego pomocą w naszym systemie hosta stworzymy **klaster**, którego wykorzystamy do nauki. Został opracowany przez zespół pracujący nad Kuberntesem dla **Google** w celu przetestowania aplikacji **bez konieczności korzystania ze środowiska chmurowego**. Dlatego jest idealnym narzędziem do nauki czy też testowania.

Dostępny jest na wiele platform takich jak **Windows**, **Linux** czy też **macOS**, a sama instalacja, jak i konfiguracja zalicza się do bardzo prostych. O tym przekonasz się, gdy za chwilę przejdiesz do dalszej części materiału. Po jego instalacji otrzymujemy klaster **Kubernetes**, z którego przy pomocy poleceń terminala jesteśmy w stanie używać.

Pomimo swojej prostoty **klaster minikube** zachowuje się w identyczny sposób jak każdy inny. Dlatego to czego nauczysz się za jego pomocą, będziesz w stanie wykorzystać we wspomnianych środowiskach chmurowych, jak i wszędzie indziej, gdzie chcesz skorzystać z tej technologii. Jego ogromną zaletą jest to, że jest narzędziem darmowym. Dlatego bez ponoszenia jakichkolwiek dodatkowych kosztów, czy samego programu lub licencji jesteś w stanie **na swoim domowym urządzeniu rozpoczęć naukę Kuberneta**.

Biorąc pod uwagę wszystkie wspomniane możliwości, **minikube jest idealnym narzędziem z którego powinniśmy skorzystać na samym początku naszej nauki Kuberneta**. Dzięki niemu jesteś w stanie odkryć moc i potencjał jaki drzemie w tym narzędziu do **orkiestracji**. Jak łatwo tego dokonać przekonasz się zapoznając z dalszą częścią przygotowanego materiału.

11.5.1. Instalacja w dystrybucji Red Hat Enterprise Linux

Instalację przeprowadzamy w następujący sposób:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-latest.x86_64.rpm
```

Następnie instalujemy pobrany pakiet:

```
sudo rpm -Uvh minikube-latest.x86_64.rpm
```

W ten prosty sposób zainstalowaliśmy nasze minikube.

11.5.2. Instalacja w dystrybucji Debian

Instalację przeprowadzamy w następujący sposób:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube_latest_amd64.deb
```

Następnie instalujemy pobrany pakiet:

```
sudo dpkg -i minikube_latest_amd64.deb
```

W ten prosty sposób zainstalowaliśmy minikube.

11.6. Konfiguracja minikube

Pozostało odpowiednio skonfigurować nasz klaster, następnie wypróbować czy wszystko działa poprawnie. Dlatego zacznijmy od ustawienia **Dockera** w naszym **minikube**:

```
minikube config set driver docker
```

Przyznam, że przy pisaniu tego artykułu wykonałem kilka instalacji. Czasami z nie wiadomo, jakich dla mnie przyczyn nie działało to ustawienie. Jeżeli spotkasz się z tym samym problemem, to skorzystaj z polecenia:

```
minikube start --driver=docker
```

Uwaga. Aby przejść do następnego kroku, potrzebujesz:

- Być zalogowanym na zwykłego użytkownika (wyloguj się z roota)
- 2 CPU
- Wsparcie dla CPU do wirtualizacji

Natomiast po ustawnieniu powyższej konfiguracji nasz minikube powinien zostać uruchomiony przy pomocy polecenia

```
minikube start
```

Po jakiejś chwili powinieneś otrzymać informację:

```
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

11.7. Podsumowanie

Choć sama instalacja **Kubernetes** wymaga poświęcenia trochę więcej czasu, to jednak nie jest w jakiś sposób skomplikowana. Dodam, że jeżeli zdarzy Ci się korzystać z Podmana, to wystarczy, że w dwóch prezentowanych przed chwilą poleceniach podmienisz **Dockera** na **Podmana**, tak jak w poniższym przykładzie:

```
minikube config set driver podman
```

```
minikube start --driver=podman
```

Natomiast do stworzenia klastra nie musisz korzystać z **Podmana** czy też **Dockera**. Do tego celu Możesz użyć **maszyn wirtualnych** na przykład za pomocą **VirtualBox**.

Rozdział 12: Kubernetes - Podstawy

Dotarłeś do tego miejsca, dlatego jak sądzę, poznałeś już dość dobrze Dockera. Aby osiągnąć wiecej, pozostaje Ci tylko praktyka. Jednak chciałbym pokazać Ci narzędzie, które jest następnym, jakie powinieneś poznać. Jak sam wiesz **przy pomocy Dockera, możesz tworzyć obrazy, a następnie kontenery**. Dlatego możemy go określić jako **narzędzie do tworzenia oraz uruchamiania pojedynczych kontenerów**. Możesz teraz się ze mną nie zgodzić, ponieważ Docker ma funkcję compose, dzięki której jesteś w stanie uruchomić multi-kontenery. Jednak compose jest dodatkiem do Dockera oraz, co jest najistotniejsze, po wprowadzeniu zmian musisz ręcznie wykonywać czynności, jak na przykład usuwać zbędne kontenery i zastępować je nowymi. Nie ma tutaj automatyzacji oraz w przypadku takiej aktualizacji, na jakiś czas aplikacja musi być nieaktywna. W wielu sytuacjach jest to problem.

Poznałeś **Docker Swarm**, który jest platformą do **orkiestracji**. Kubernetes jest również taką samą platformą, jednak o wiele większych możliwościach. Pomimo identycznej funkcjonalności Docker Swarm ma ograniczoną możliwość rozbudowy. Natomiast Kubernetes jest najczęściej wybierany w środowiskach produkcyjnych, ponieważ **ma możliwość uruchomienia większej ilości nodów niż Docker Swarm**. Dlatego, jeżeli środowisko produkcyjne będzie o dość małej strukturze można śmiało pomyśleć od Docker Swarm. Jest prostszy i nie wymaga większej technicznej wiedzy. Jednak jeżeli środowisko będzie posiadało dużą liczbę nodów, podów, czy też kontenerów lub z czasem liczba będzie się zwiększała to od razu najlepiej skorzystać z Kuberntesa. Wymaga on większej wiedzy i jest bardziej skomplikowany, jednak po spędzeniu dłuższej chwili może się okazać efektywniejszym rozwiązaniem.

12.1. Dokumentacja

Choć wydawać się może, że odbiegam od tematu, to jednak chciałbym, abyś wszedł i zapisał sobie w ulubionych stronę do dokumentacji Kuberntesa, która znajduje się pod adresem <https://kubernetes.io/pl/docs/home/>. Zapytasz pewnie dlaczego?

Otoż Kubernetes rozwija się bardzo dynamicznie i niektóre elementy potrafią się zmienić, dlatego czasami, gdy coś nie działa, warto tam zerknąć. Ten dynamizm rozwoju powoduje, że niektóre w miarę nowe publikacje mogą okazać się nieaktualne. Dokumentacja Kuberntesa będzie nieodzowną bazą wiedzy, dla każdego, kto chce z niego korzystać.

12.2. Podstawowe zagadnienia

Przez etap instalacji minikube przeszedłeś we wcześniejszym materiale. Masz już go przygotowanego i wiesz, że działa. Wraz z jego instalacją zainstalowaliśmy kubectl, dzięki któremu będziemy wykonywali większość działań w tym szkoleniu. Jednak wiedz, że dzięki przeprowadzonej instalacji stworzyliśmy klaster. Taki klaster należy rozumieć jako zestaw maszyn roboczych, które określane są jako węzły. Następnie na takich węzłach powstają kontenery, w których uruchamiane są aplikacje. W trakcie tworzenia klastra zawsze tworzony jest jeden węzeł. Taki węzeł powstaje zawsze, ponieważ jest to wymagane przez Kuberntesa.

W tym klastrze, znajdują się węzły, w których umieszczone są pody. Pod jest to pojemnik na kontenery, w którym może znajdować się kilkanaście kontenerów. Wszystkie kontenery aplikacji umieszczamy w takim podzie dlatego mamy możliwość ich zarządzania jako grupą.

12.3. Podstawowe polecenia Kuberntesa

Przyznam, że początkowo, gdy poznawałem Kuberntesa, to po przeczytaniu jego opisu nie rozumiałem go. Dopiero gdy zacząłem używać Kuberntesa w sposób bardziej praktyczny i wróciłem do wyjaśnienia, zrozumiałem, o co chodzi. Dlatego, jeżeli nie rozumiesz moich wyjaśnień, nie przejmuj się tym. Uważam, że w części praktycznej, do której zaraz przejdziemy, wszystko będzie bardziej zrozumiałe.

W Dockerze wywoływałyśmy polecenia w konsoli, aby pobrać obraz i stworzyć kontener. W przypadku Kuberntesa też wykonujemy te czynności za pomocą polecień. Kuberntes korzysta z tego samego źródła obrazów co Docker. Dlatego strona <https://hub.docker.com/> jest również bazą obrazów dla Kuberntesa.

Pobierzmy obraz i go uruchommy:

```
kubectl run apache --image=httpd
```

W związku z tym, że jest to nasze pierwsze uruchomienie, popatrzmy, co w tym przykładzie zrobiliśmy. **kubectl** jest to aplikacja, dzięki której będziemy tworzyli oraz uruchamiali wszystko w Kuberntes. Opcję **run** wykorzystujemy w identyczny sposób jak w Dockerze, jednak w tym wypadku to, co zrobiliśmy to **uruchomienie kontenera w podzie**. Po run wpisujemy **nazwę, pod jaką ma występować uruchomiony przez nas pod**. Następnie określamy, z jakiego obrazu chcemy skorzystać. W tym przykładzie uruchamiam kontener z **apache2**, dlatego korzystam z obrazu **httpd**. Tak samo, jak w przypadku Dockera możesz korzystać z określonej wersji obrazu. Robimy to w sposób:

```
kubectl run apache-older --image=httpd:2.4
```

Zmieniamy nazwę poda, bo dwie takie same nazwy nie mogą wystąpić i tak jak w przypadku Dockera po nazwie obrazu wpisujemy dwukropek i określamy jego wersję.

UWAGA! Tak naprawdę to dzięki Dockerowi są pobierane obrazy. Kubernetes w minikube wydaje polecenia Dockerowi, który wykonuje wskazane przez niego czynności.

12.4. Wstęp do podów

Stworzyliśmy **dwa pody**, w których umieściliśmy **dwie różne wersje apache**. Aby wyświetlić listę utworzonych podów, posługujemy się poleceniem:

```
kubectl get pod
```

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------|-------|---------|----------|-------|
| apache | 1/1 | Running | 0 | 8m42s |
| apache-older | 1/1 | Running | 0 | 8m32s |

W tym miejscu jest punkt, w jakim chciałbym się zatrzymać. Otóż możesz za bardzo nie rozumieć, czym jest pod i jak to jest z tymi kontenerami. Dlatego wyjaśnijmy to sobie bardziej szczegółowo.

Na liście znajdują się dwa przed chwilą stworzone pody. Zwróć uwagę na kolumnę **READY**. W niej znajduje się **ilość gotowych kontenerów w podzie**. Ta liczba może być większa, jeżeli do danego poda, zechcemy dodać większą liczbę kontenerów. W związku z tym **pod określamy jako pojemnik na kontenery**, które z sobą powinny w jakiś sposób współpracować. Użyłem słowa, powinny, ponieważ jest to jak najbardziej logiczne, jednak wcale tak nie musi być.

12.5. Restarty kontenerów w podach

Utworzyliśmy dwa pody, w których znajduje się uruchomiony serwer apache. Jak wiesz apache w związku z tym, że jest serwerem działa w trybie ciągłym. Czyli nie posiada instrukcji, która automatycznie kończy jego działanie tak jak na przykład, gdy uruchamiamy kontener Dockera z systemem. Stwórzmy sobie pod w którym pobierzemy i uruchomimy kontener z Ubuntu:

```
kubectl run ubuntu --image=ubuntu
```

Teraz, jeżeli wyświetlisz listę dostępnych kontenerów przy użyciu polecenia *get pod*, powinieneś jako pierwsze otrzymać:

kubectl get pod

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------|-------|-----------|----------|-----|
| apache | 1/1 | Running | 0 | 16m |
| apache-older | 1/1 | Running | 0 | 16m |
| ubuntu | 0/1 | Completed | 0 | 5s |

Jak pamiętasz, tego typu kontenery kończą swoje działanie, ponieważ nie mają żadnej instrukcji, jaką miałyby wykonać. Teraz ponownie wyświetl listę podów:

kubectl get pod

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------|-------|------------------|------------|-----|
| apache | 1/1 | Running | 0 | 16m |
| apache-older | 1/1 | Running | 0 | 16m |
| ubuntu | 0/1 | CrashLoopBackOff | 1 (2s ago) | 6s |

Otrzymaliśmy status **CrashLoopBackOff**. Oznacza to, że kontener restartuje się po zakończeniu swojego działania. Ma to związek z wbudowaną polityką Kurnetesa. W przypadku serwera jest to dość przydatna funkcja, jednak gdy chcemy uruchomić, pod który ma nie działać w trybie ciągłym, to restartowanie generuje błąd. By to udowodnić, po jakimś czasie jak wyświetlisz powyższą listę, powinieneśtrzymać informację:

kubectl get pod

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------|-------|------------------|---------------|-----|
| apache | 1/1 | Running | 0 | 26m |
| apache-older | 1/1 | Running | 0 | 26m |
| ubuntu | 0/1 | CrashLoopBackOff | 6 (4m24s ago) | 10m |

Zwróć uwagę na kolumnę **RESTARTS**. Określa ona, że wystąpiło 6 restartów, poda ubuntu. Błąd wygenerowałem specjalnie, a rozwiążanie podam na końcu. Dzięki powstałemu błędowi mamy możliwość poznania nowego polecenia, przy pomocy, którego wypiszemy szczegółowe informacje, dotyczące poda. Robimy to w następujący sposób:

kubectl describe pod ubuntu

Wyświetli się dość sporo informacji. Nie będę przykleiał otrzymanego wyniku, ponieważ masz go przed sobą. Po poznaniu Dockera powinieneś mniej więcej orientować się w informacjach, które wyświetliły Ci się dzięki temu poleceniu. To, co jest dla nas obecnie istotne, znajduje się na samym dole:

Warning BackOff 4m14s (x71 over 19m) kubelet Back-off restarting failed container

Oczywiście, u Ciebie otrzymany błąd będzie się różnił, jednak informacja końcowa powinna być identyczna: *Back-off restarting failed container*. Jak temu zaradzić? Należy, przy uruchomieniu poda, wyłączyć konieczność restartu:

```
kubectl run ubuntu-no-restart --image=ubuntu --restart=Never
```

Teraz, jeżeli ponownie wypiszesz listę:

```
kubectl get pod
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------|-------|------------------|---------------|-----|
| apache | 1/1 | Running | 0 | 40m |
| apache-older | 1/1 | Running | 0 | 39m |
| ubuntu | 0/1 | CrashLoopBackOff | 9 (2m36s ago) | 23m |
| ubuntu-no-restart | 0/1 | Completed | 0 | 10s |

W podzie, w którym **została wyłączona polityka restartowania**, po wykonaniu wszystkich czynności kończy działanie kontenera i ustawia status na *Completed*. Czyli zgodnie z tym, co występowało w kontenerach Dockera. Dlatego, jeżeli wyświetlisz, opis poda:

```
kubectl describe pod ubuntu-no-restart
```

Events:

| Type | Reason | Age | From | Message |
|--------|-----------|-------|-------------------|---|
| --- | --- | --- | --- | ----- |
| Normal | Scheduled | 6m14s | default-scheduler | Successfully assigned default/ubuntu-no-restart to minikube |
| Normal | Pulling | 6m14s | kubelet | Pulling image "ubuntu" |
| Normal | Pulled | 6m13s | kubelet | Successfully pulled image "ubuntu" in 1.480595472s |
| Normal | Created | 6m13s | kubelet | Created container ubuntu-no-restart |
| Normal | Started | 6m13s | kubelet | Started container ubuntu-no-restart |

Żaden błąd już się nie pojawią.

12.6. Możliwość skracania nazw

W przypadku poleceń wyświetlających możemy korzystać skrótowej jej formy. Oznacza to, że nie musisz wpisywać pełnej nazwy. Dla przykładu do tej pory korzystaliśmy z polecenia w taki sposób, by wyświetlić listę wszystkich podów:

```
kubectl get pod
```

Czy też do wyświetlania opisu:

```
kubectl describe pod ...
```

Możemy to skrócić w następujący sposób:

kubectl get po

kubectl describe po ...

W przypadku poda, pozbywamy się ostatniej litery. Wiem, że nie jest to wielka różnica, jednak przy dłuższych poleceniach, które poznamy w dalszej części szkolenia, tego typu skracanie ma sens.

12.7. Nody

Do tej pory tworzyliśmy pody i wykonywaliśmy je w nodzie, który stworzyliśmy przy instalacji minikube. Listę dostępnych nodów jesteś w stanie uzyskać przy pomocy polecenia:

kubectl get nodes

| NAME | STATUS | ROLES | AGE | VERSION |
|----------|--------|---------------|-----|---------|
| minikube | Ready | control-plane | 29d | v1.25.3 |

W tym miejscu wypisane będą wszystkie nody, jakie obsługujesz. My korzystamy tylko z jednego w postaci minikube, dlatego wszystkie dotychczasowe pody są obsługiwane przez niego. Wersja skrócona, do wyświetlania nodów wygląda następująco:

kubectl get no

Aby to sprawdzić, możemy zalogować się do naszego minikube przy pomocy ssh. Wystarczy, że wpiszemy:

minikube ssh

Zostaniesz zalogowany do sesji wiersza poleceń minikuba. W nim możesz wyświetlić wszystkie aktywne kontenery przy pomocy znanego już polecenia:

docker ps

Zostanie wyświetlona bardzo obszerna lista, dlatego jej tutaj nie wklejam. Aby wyświetlić kontenery dotyczące naszych podów, możemy skonstruować polecenie w następujący sposób:

docker ps | grep apache

Wyświetlą się tylko te, które dotyczą naszego serwera apache. Tak samo, możesz zrobić z ubuntu. Jak zauważyłeś, w nodzie korzystamy z Dockera. Czyli teraz masz już potwierdzenie, że Kubernetes wykorzystuje Dockera do tworzenia kontenerów. Aby opuścić sesję, wystarczy wpisać **exit**.

12.8. Usuwanie podów

Jeżeli pody nie są już nam do niczego potrzebne, należy je usunąć. Nie musimy ich zatrzymywać, a dopiero po tym usuwać. Polecenie **kubectl nie posiada funkcji zatrzymania**. Dlatego od razu przystępujemy do usuwania przy pomocy polecenia:

kubectl delete po ubuntu

```
pod "ubuntu" deleted
```

W ten sposób usuniemy, pod który restartował się po zamknięciu. Zwróć uwagę na składnię, jest ona równie przejrzysta, jak w Dockerze, chociaż delikatnie się różni. W ten sposób usunęliśmy jeden pod. Jeżeli chcemy, możemy usunąć je wszystkie przy pomocy polecenia:

kubectl delete pods --all

```
pod "apache" deleted
```

```
pod "apache-older" deleted
```

```
pod "ubuntu-no-restart" deleted
```

Tak wyczyściliśmy listę naszych podów. Jednak musisz z tym bardzo uważać. Polecenie do usuwania wszystkich elementów przydaje się, gdy się uczymy. W przypadku środowiska produkcyjnego może spowodować wiele problemów.

12.9. Podsumowanie

Pierwszy etap nauki Kuberntesera masz już za sobą. **Pody są najważniejszym typem w Kubernetesie** jaki należy bardzo dobrze poznać. Warto w tym momencie zatrzymać się i stworzyć kilka na własną rękę. Znasz Dockera, to wiesz jak można, pobierać obrazy. W tym materiale poznaleś sposób, w jaki robi się to w Kubernetesie. Dlatego nic nie stoi na przeszkodzie, byś się trochę tym *pobawił*.

Rozdział 13: Wstęp do plików yaml

Wiesz już, czym są pliki yaml i jaka jest ich składnia oraz w jaki sposób są wykorzystywane. W przypadku Kuberntesesa nie jest inaczej. To, co poznaleś dotychczas, jest podstawą podstaw. Ważne, że znasz, jednak prawda jest taka, że rzadko pobiera się i uruchamia pody w sposób prezentowany w poprzednim materiale. W Kuberntesie do wykonania większości czynności tworzy się odpowiednie pliki .yaml. Dlatego tym zajmiemy się w tym module.

13.1. Podstawowa konstrukcja pliku

Typów w Kuberntesie jest kilka. My poznaliśmy jeden, który nazywa się pod. W tym module pokażę Ci, w jaki sposób stworzyć pod przy pomocy pliku. Przy okazji zapoznam Cię z jego podstawową konstrukcją. Dlatego otwórz swój ulubiony edytor tekstowy i stwórz plik o nazwie *pod.yaml* lub nadaj mu nazwę według własnego uznania. W nim umieść:

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: <nazwa>
```

Opiszmy każdy z elementów występujący w przykładzie.

apiVersion: v1

Jest to interfejs, z jakiego będziemy korzystali w pliku. W Kuberntesie istnieje takich kilka. Każdy z nich ma różne opcje co do zasobów takich jak obsługa typów. Ja w tym przykładzie posłużyłem się wersją 1, ponieważ zawiera ona wszystkie niezbędne funkcje, jakie zamierzam użyć. Więcej o tym możesz przeczytać pod adresem <https://kubernetes.io/pl/docs/concepts/overview/kubernetes-api/>. **apiVersion można rozumieć jako repozytorium lub bibliotekę funkcji**, z których będziemy korzystali w pliku. Czyli dzięki określeniu wersji wszystko, co wpiszemy, będzie uruchomione, stworzone zgodnie z zasadami w tym repozytorium. Natomiast **najczęściej zmieniamy wersję api w Kuberntesie, aby uzyskać dostęp do jakiegoś nowego typu**.

kind: Pod

W tym miejscu **wpisujemy jaki typ chcemy stworzyć**. Jedynym jaki do tej pory poznaliśmy był pod dlatego też od razu wpisałem go w tę rubrykę.

metadata:

name: <nazwa>

Tu wprowadzamy niezbędne metadane jak na przykład nazwę naszego typu. Czyli w naszym przypadku pod będzie nosił nazwę, jaką określmy w etykiecie name.

Opisane elementy są konieczne i muszą występować w każdym z plików.

13.2. Pierwsze uruchomienie poda

Jeżeli będziemy chcieli stworzyć pod za pomocą dotychczasowej konstrukcji pliku, nie uda nam się to. Pod służy do uruchomienia kontenerów. My tych kontenerów nie sprecyzowaliśmy, dlatego też nie zostaną one utworzone. W związku z tym Kubernetes nie jest w stanie, stworzyć poda. Dlatego aby wszystko działało, do pliku musimy dodać:

spec:

containers:

- name: pierwszy-kontener

image: nginx

Po dyrektywie *spec* następuje konfiguracja wybranego typu. W naszym wypadku, w polu *kind*, określiliśmy, że chcemy stworzyć *pod*. Po etykiecie *containers* określamy kontenery dla wybranego typu. Następnie tworzymy listę przy pomocy znaku -. Lista jest to funkcja Kubernetesa, dzięki której określamy funkcje kontenera. Istotny jest jednak sposób zapisu, ponieważ jak zapewne pamiętasz, w pliku yaml bardzo ważne są odstępy. Dlatego myślnik powinien znajdować się na tym samym poziomie co *containers*. Po myślniku wprowadzamy spację i w *name* wpisujemy nazwę naszego kontenera. Poniżej na poziomie *name* wprowadzamy nazwę obrazu, co zapewne zauważysz. Ten typ formatowania stosowaliśmy w pliku *.yaml Docker Compose*, dlatego nie powinien być Ci obcy. Jednak w przypadku Kubernetesa częściej spotkasz się z dość zagnieżdzonymi odstępami.

W powyższym przykładzie użyjemy najnowszego obrazu nginx. Cały plik powinien wyglądać w następujący sposób:

apiVersion: v1

kind: Pod

metadata:

```
name: moj-pod

spec:

containers:

- name: pierwszy-kontener

  image: nginx
```

Zapisz plik. Następnie będąc w folderze, w którym tego dokonałeś, użyj polecenia:

kubectl apply -f pod.yaml

W poleceniu kubectl korzystamy z funkcji apply odpowiedzialnej za uruchomienie. Następnie przy pomocy opcji **-f** wskazujemy, że chcemy stworzyć typ Kubernetesowy z pliku noszącym nazwę **pod.yaml**. Po uruchomieniu polecenia, jeżeli wprowadziłeś dane w identyczny sposób jak ja, powinieneś otrzymać informację:

```
pod/moj-pod created
```

Oznacza to, że został stworzony pod. Zerknijmy na listę:

kubectl get po

```
NAME READY STATUS RESTARTS AGE
moj-pod 1/1 Running 0 76s
```

Jak prezentuje to przykład, pod został utworzony i działa. Czyli zrobiliśmy to samo co w poprzednim rozdziale przy pomocy polecenia *run*.

13.3. Polityka restartu typów

Drugim przykładem, jakim chciałbym się zająć, jest problem z restartowaniem, z którym spotkaliśmy się w poprzednim module. Dlatego stwórzmy plik korzystający z obrazu Ubuntu i go uruchommy:

```
apiVersion: v1

kind: Pod

metadata:

  name: ubuntu-pod

spec:
```

containers:

```
- name: ubuntu-kontener  
  image: ubuntu
```

Prezentowany kod zapisałem w pliku *ubuntu.yaml*. Teraz uruchomimy go przy pomocy polecenia:

kubectl apply -f ubuntu.yaml

pod/ubuntu-pod created

Jeżeli teraz szybko wyświetlimy listę dostępnych podów, powinniśmy otrzymać komunikat:

kubectl get po

| NAME | READY | STATUS | RESTARTS | AGE |
|------------|-------|-----------|------------|-----|
| moj-pod | 1/1 | Running | 0 | 55m |
| ubuntu-pod | 0/1 | Completed | 1 (3s ago) | 6s |

Następnie, jeżeli za chwilę ponownie wyświetlisz listę, otrzymasz komunikat o błędzie:

kubectl get po

| NAME | READY | STATUS | RESTARTS | AGE |
|------------|-------|------------------|------------|-----|
| moj-pod | 1/1 | Running | 0 | 55m |
| ubuntu-pod | 0/1 | CrashLoopBackOff | 1 (3s ago) | 8s |

Stworzyliśmy identyczną sytuację jak w przypadku gdy korzystaliśmy z poleceń. Kontener z Ubuntu kończy swoje działanie, ponieważ nie ma określonych zadań, natomiast Kubernetes w związku ze swoją polityką uruchamia go ponownie. Powoduje to wystąpienie błędu. Aby kontener po zakończeniu wszystkich swoich działań nie uruchomił się ponownie, musimy określić mu *restartPolicy*. Robimy to tak jak w poniższym przykładzie:

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: ubuntu-rp
```

```
spec:
```

```
  containers:
```

```
- name: ubuntu-rp
```

```
  image: ubuntu
```

```
restartPolicy: Never
```

Ważne jest, aby *restartPolicy* było na poziomie pola *containers*. Teraz zerknijmy na naszą listę podów:

kubectl get po

| NAME | READY | STATUS | RESTARTS | AGE |
|------------|-------|------------------|---------------|-----|
| moj-pod | 1/1 | Running | 0 | 71m |
| ubuntu-pod | 0/1 | CrashLoopBackOff | 7 (4m31s ago) | 15m |
| ubuntu-rp | 0/1 | Completed | 0 | 7s |

Jak widać, pod osiągnął status *Completed*, dlatego wszystko poszło zgodnie z naszym planem. Jednak nie jest to wszystko, co możemy zrobić.

13.4. Wywołanie polecenia w kontenerze

Zarówno w Dockerze, jak i w Kubernetes, przy tworzeniu kontenera możesz wywoływać polecenia w trakcie jego uruchamiania. Do tego służy dyrektywa *command*. Zerknij na poniższy przykład:

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: ubuntu-rp-cm
```

```
spec:
```

```
  containers:
```

```
    - name: ubuntu
```

```
      image: ubuntu
```

```
      command: ["sleep", "20"]
```

```
    restartPolicy: Never
```

Powyższy przykład stworzy pod o nazwie *ubuntu-rp-cm*. Następnie w nim stworzy kontener o nazwie *ubuntu*, który powstanie z najnowszego obrazu Ubuntu. W powstałym kontenerze zostanie wywołane polecenie *sleep 20*, które uśpi działania kontenera na 20 sekund. W związku z tym kontener będzie w stanie *Running* przez 20 sekund. Po osiągnięciu wyznaczonego czasu zakończy swoje działanie i osiągnie status *Completed*. Zerknij na poniższe przykłady:

Stworzenie poda i uruchomienie kontenera:

```
kubectl apply -f ubuntu-rp-cm.yaml
```

```
pod/ubuntu-rp-cm created
```

Sprawdzenie działania kontenera, po upływie kilku sekund:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------|-------|------------------|----------------|-----|
| moj-pod | 1/1 | Running | 0 | 86m |
| ubuntu-pod | 0/1 | CrashLoopBackOff | 10 (4m42s ago) | 31m |
| ubuntu-rp | 0/1 | Completed | 0 | 15m |
| ubuntu-rp-cm | 1/1 | Running | 0 | 11s |

Sprawdzenie statusu kontenera po upłynięciu 20 sekund:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------|-------|------------------|----------------|-----|
| moj-pod | 1/1 | Running | 0 | 86m |
| ubuntu-pod | 0/1 | CrashLoopBackOff | 10 (4m53s ago) | 31m |
| ubuntu-rp | 0/1 | Completed | 0 | 15m |
| ubuntu-rp-cm | 0/1 | Completed | 0 | 22s |

Pod zakończył swoje działanie w wyznaczonym czasie. Jeżeli chciałbyś, aby kontener był w tak zwanym stanie uśpienia przez cały czas, to zamiast określania mu ilości sekund wpisz frazę *inf*. Zaprezentuję to w następnym przykładzie.

13.5. Dwa kontenery w jednym podzie

Stworzymy teraz plik z dwoma kontenerami. Pierwszy z nich będzie używał nginx drugi Ubuntu. Zerknij na przykład poniżej:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-ubuntu
spec:
  containers:
    - name: nginx
      image: nginx
    - name: ubuntu
      image: ubuntu
      command: ["sleep", "inf"]
```

Zwróć uwagę, w jaki sposób określane są kontenery w podzie. Do tego korzystamy ze znaku - czyli z listy. Oczywiście w ten sposób możesz stworzyć o wiele więcej kontenerów. Wystarczy, że napiszesz kolejną listę. Uruchommy skonstruowany w powyższy sposób pod:

kubectl apply -f nginx-ubuntu.yaml

Zawartość przykładu umieściłem w pliku o nazwie *nginx-ubuntu.yaml*. Następnie go uruchomiłem. Teraz wyświetlamy listę podów:

kubectl get po

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------|-------|---------|----------|-----|
| nginx-ubuntu | 2/2 | Running | 0 | 6s |

Z listy usunąłem te, które zostały utworzone w poprzednich przykładach. Zwróć uwagę, mamy teraz 2/2, czyli w podzie zostały uruchomione dwa kontenery i działają. Jednak zajrzyjmy w opis poda:

Containers:

nginx:

...

ubuntu:

...

...

W miejscu, gdzie znajduje się etykieta *Containers*, odnajdziesz dwa kontenery oraz ich dokładny opis. Postanowiłem nie wkleić całości, ponieważ jest dość długa i na pewno będzie różniła się od twojej. Przynasz, że bardzo przydatna opcja. Pozwala na grupowanie kontenerów. Jednak teraz powstaje pewien problem. Jak dostać się do konkretnego kontenera?

13.6. Powłoka kontenera

Poznaj polecenie, dzięki któremu dostać się możesz do powłoki kontenera. Użyjmy go na podzie, w którym mamy uruchomione dwa kontenery:

```
kubectl exec -it nginx-ubuntu -- bash
```

```
Defaulted container "nginx" out of: nginx, ubuntu
```

```
root@nginx-ubuntu:/#
```

Znajdujemy się, w kontenerze poda, który mieliśmy ustawiony jako pierwszy. Zgodnie z tym, co nam podpowiada Kubernetes, jest to nginx. Jak się jednak dostać do tego drugiego? Otóż musimy użyć zbliżonego polecenia do powyższego, ale określamy konkretnie, o jaki kontener nam chodzi:

```
kubectl exec -it -c ubuntu nginx-ubuntu -- bash
```

Przy pomocy opcji **-c** określamy, do powłoki którego kontenera w podzie chcemy się dostać. W przypadku poda, w którym znajduje się kilka kontenerów, najlepiej zawsze jest określać kontener, do którego chcemy się zalogować. Dzięki temu unikniemy wielu nieprzyjemnych sytuacji.

13.7. Logi

Logi są ważnym elementem każdego systemu. W tym wypadku każdego kontenera. Wiesz, już, w jaki sposób uruchamiać pod za pomocą linii poleceń, jak i również wiesz w jaki sposób stworzyć taki przy pomocy pliku yaml. Aby sprawdzić logi kontenera w podzie, posługujemy się poleceniem:

kubectl logs moj-pod

W ten sposób wyświetlimy logi kontenera w podzie. Jeżeli znajduje się w nim kilka, to powinniśmy użyć poznanej przed chwilą opcji z nazwą kontenera, z którego chcemy wydobyć logi:

kubectl logs -c nginx nginx-ubuntu

W tym materiale korzystaliśmy z kilku podów. Nie będą one nam już potrzebne. Twoim zadaniem będzie usunąć wszystko, to, co do tej pory stworzyliśmy. Dlatego do dzieła!

13.8. Podsumowanie

Sama podstawowa konstrukcja Kuberntesa nie jest bardzo skomplikowana. Dotychczas poznaliśmy tylko jeden typ, jakim jest pod. Jednak na nim nauczyliśmy się wielu cennych rzeczy, które wykorzystamy w dalszej części szkolenia. Jeżeli jest Ci coś niejasne, to przejrzyj ponownie materiały. Do tej pory wszystkie przykłady były bardzo proste. W następnych rozdziałach będzie trochę trudniej, dlatego ważne jest, abyś zrozumiał wszystko to, co do tej pory pokazałem.

Rozdział 14: Deployment i wszystko co z nim związane

W tym materiale poznamy kilka rodzajów typów Kuberntesesa, jakie możemy stosować. W dotychczasowych materiałach zajmowaliśmy się tylko podami. Pod jest to podstawowy typ Kuberntesesa, o czym powinieneś już wiedzieć. Te, które poznasz w tym i kolejnych modułach są rozbudowanymi funkcjami, dzięki którym wszystko, co wykonasz w Kuberntesie, będzie łatwiejsze. Jednak nim do tego dojdziemy, w poprzednim module miałeś za zadanie usunięcie wszystkich podów. Jeżeli miałeś z tym problem, poniżej podaję rozwiązanie.

14.1. Usunięcie podów z poprzedniego rozdziału

Wyświetlenie listy podów:

kubectl get po

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------|-------|-----------|--------------|-----|
| moj-pod | 1/1 | Running | 1 (29s ago) | 18h |
| nginx-ubuntu | 2/2 | Running | 2 (29s ago) | 16h |
| ubuntu-pod | 0/1 | Completed | 27 (15h ago) | 17h |
| ubuntu-rp | 0/1 | Completed | 0 | 17h |
| ubuntu-rp-cm | 0/1 | Completed | 0 | 16h |

Usunięcie każdego poda:

kubectl delete po moj-pod nginx-ubuntu ubuntu-pod ubuntu-rp ubuntu-rp-cm

Po wprowadzeniu powyższego polecenia lista powinna być pusta.

14.2. Poznajemy typ ReplicaSet

Omawiany typ dba o to, aby wyznaczona ilość replik, została uruchomiona. Prościej można go opisać jako narzędzie dbające o to, by ilość identycznych podów zgadzała się z ilością określona Kuberntesowi. Typ ReplicaSet jest bardzo rzadko stosowany, ponieważ uruchamia się automatycznie z deploymentem, o którym będziemy rozmawiać za chwilę. Jednak chciałbym, abyś wiedział, że tego typu narzędzie możesz uruchomić oddzielnie oraz abyś zrozumiał, w jaki sposób ono działa. Pomoże Ci to w dalszym zrozumieniu typu deployment.

Pominę jednak temat ReplicaController, którego stosowanie nie jest już zalecane. Nie będę zajmował się opisem różnic pomiędzy tymi dwoma typami. Dopowiem, tylko że ReplicaSet jest nowszym rozwiązaniem wzbogaconym o funkcje, których brakowało u poprzednika. Wspominam o tym tylko bo możesz się spotkać z tym typem przy analizowaniu jakiegoś pliku yaml. Wiedz wtedy, że służy on temu samemu co ReplicaSet.

Nim pokażę cały plik, rozbiję go na kilka elementów. Zrozumienie tego jest o tyle istotne, że konstrukcja, jaką tu zaprezentuję, jest stosowane przy każdym innym typie. Dlatego, pomimo że w niektórych wypadkach powtórzę się, to ma to na celu lepsze zrozumienie tematu oraz jego utrwalenie. Dlatego zobacz, w jaki sposób konstruuje się taki plik:

apiVersion: apps/v1

Przykład prezentuje nagłówek pliku, o którym już rozmawialiśmy. ReplicaSet znajduje się w innej, jak to określiłem w poprzednim module bibliotece. Dlatego też nie wpisujemy tutaj v1 jak poprzednio, tylko apps/v1.

kind: ReplicaSet

metadata:

name: nginx-rs

Te dwa elementy są wymagane w każdym pliku yaml Kubernetesa. W poprzednim module korzystaliśmy z typu Pod, natomiast teraz wprowadzamy typ ReplicaSet. Pod name wpisałem nazwę, pod jaką będzie występował nasz pod. Do nazwy dodałem rs tak, aby można było go lepiej rozpoznać.

spec:

replicas: 3

selector:

matchLabels:

app: app-nginx-rs

Na pewno zauważyłeś nową etykietę replicas. Jak się domyślasz dzięki niej, ustawiamy ilość replik. ReplicaSet ma za zadanie kontrolowanie, aby określona w tym miejscu ilość podów została stworzona. W naszym wypadku jest to 3.

Natomiast w dalszej części przykładu musimy jeszcze określić, co ma kontrolować, a dokładniej wyznaczyć temu typowi, o który dokładnie pod nam chodzi. Wykonujemy to przy pomocy etykiety, którą umieszczamy w selektorze, tak jak pokazuję to w przykładzie. Czyli dodaję selektor matchLabels, w którym określamy nazwę etykiety aplikacji, którą ma ReplicaSet kontrolować. Teraz musimy przypisać, do poda za pomocą znacznika label nazwę app-nginx-rs, dzięki której określmy ReplicaSet, pod który ma kontrolować. Robimy to w następujący sposób:

template:

```
metadata:  
  name: nginx-pod-rs  
  
labels:  
  app: app-nginx-rs  
  
spec:  
  
  containers:  
    - name: nginx-rs-container  
  
      image: nginx
```

Aby móc przypisać etykietę podowi, musimy stworzyć szablon. Robimy to przy pomocy selektora template. Następne kroki są identyczne, jakie wykonywaliśmy w pierwszym module. Czyli nazywamy nasz pod. Dalej znajduje się miejsce, w którym przypisujemy przy pomocy selektora label identyczną nazwę, jaką nadaliśmy w app na początku pliku. W ten właśnie sposób określamy, o który pod nam chodzi. Następnie w spec określamy nazwę kontenera i obraz, z jakiego ma powstać. Po dokładnym omówieniu, poniżej prezentuje cały kod:

```
apiVersion: apps/v1  
kind: ReplicaSet  
  
metadata:  
  name: nginx-rs  
  
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      app: app-nginx-rs
```

```
template:  
  metadata:  
    name: nginx-pod-rs  
    labels:  
      app: app-nginx-rs  
  spec:  
    containers:  
      - name: nginx-rs-container  
        image: nginx
```

Napisaliśmy plik yaml, teraz nauczmy się go obsługiwać. W celu jego uruchomienia posługujesz się już poznanym wcześniej poleceniem:

```
kubectl apply -f nginx-rs.yaml
```

```
replicaset.apps/httpd-rs created
```

Wyświetlamy listę podów:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------|-------|---------|----------|-----|
| nginx-rs-29hvq | 1/1 | Running | 0 | 6s |
| nginx-rs-b9t6v | 1/1 | Running | 0 | 6s |
| nginx-rs-jwgz7 | 1/1 | Running | 0 | 6s |

Zgodnie z replikami, jakie ustawiliśmy, zostały stworzone 3 pody. Zgodnie z powyższym przykładem wszystko się zgadza. Pojawiły się one w odpowiedniej ilości.

4.2.1. Usuwanie podów

Jak pamiętasz, ReplicaSet dba o to, by istniała odpowiednia ilość replik. Dlatego co się stanie jak, usuniemy jeden z podów? Zobaczmy to na przykładzie:

```
kubectl delete po nginx-rs-29hvq
```

```
pod "nginx-rs-29hvq" deleted
```

Jeżeli teraz dość szybko użyjesz polecenia do wyświetlania listy podów, możesz zauważyc, że po usunięciu poda, na jego miejsce powstaje nowy:

kubectl get po

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------|-------|-------------------|----------|------|
| nginx-rs-b9t6v | 1/1 | Running | 0 | 3m5s |
| nginx-rs-jwgz7 | 1/1 | Running | 0 | 3m5s |
| nginx-rs-nvntf | 0/1 | ContainerCreating | 0 | 2s |

ContainerCreating, czyli kontener jest tworzony. O tym, że jest nowy, można wyczytać z kolumny AGE. Jednak to, co tutaj zaobserwowałeśmy, jest bardzo istotne. Jeżeli usuniemy pod to automatycznie, w jego miejsce jest tworzony nowy. Jest to wykonywane bez ingerencji z naszej strony. O to dba ReplicaSet.

14.2.2. Lista ReplicaSet

Tak samo, jak listę podów tak i możemy wypisać listę ReplicaSet. Robimy to za pomocą:

kubectl get replicaset

| NAME | DESIRED | CURRENT | READY | AGE |
|----------|---------|---------|-------|-------|
| nginx-rs | 3 | 3 | 3 | 9m29s |

lub w skróconej formie:

kubectl get rs

| NAME | DESIRED | CURRENT | READY | AGE |
|----------|---------|---------|-------|-------|
| nginx-rs | 3 | 3 | 3 | 9m55s |

Tak jak prezentuje to tabela z przykładu mamy ustawione 3 repliki, które są uruchomione i gotowe do działania. Na końcu mamy podany czas od ich uruchomienia.

14.2.3. Usuwanie ReplicaSet

Jak zapewne zauważycie, jeżeli pody są replikowane, od razu po ich usunięciu to nie możesz wykorzystać znanego Ci dotychczas sposobu, aby się ich pozbyć. W celu ich usunięcia musisz usunąć ReplicaSet. Robisz to w następujący sposób:

```
kubectl delete rs nginx-rs
```

Zostanie usunięta ReplicaSet oraz pody, które były przez ten typ kontrolowane.

14.3. Deployment

To, co pokazywałem w poprzednim części, jak już wspomniałem, nie jest często wykorzystywane, ponieważ jest automatycznie uruchamiane przez deployment. Teraz chcę, abyś poznał, właśnie ten wspomniany typ. Dba on tak samo o ilość replik, wykorzystując do tego celu ReplicaSet. Jednak w jego wypadku jesteśmy w stanie zaktualizować ich ilość w niezauważalny sposób dla użytkownika. Dlatego, gdy aplikacja powinna działać bez przerwy, powinniśmy skorzystać z właśnie tego typu.

14.3.1. Uruchomienie Deploymentu

Ze względu na to, że dość szczegółowo opisaliśmy sobie wszystko, chwilę temu pozwolę sobie od razu wkleić kod:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app-apache-deployment
  template:
    metadata:
      name: apache-pod-deployment
      labels:
        app: app-apache-deployment
    spec:
      containers:
        - name: apache-deployment-container
          image: httpd
```

Nie wiem, czy zwróciłeś uwagę, ale jest to identyczny przykład jak ten, który prezentowałem przy ReplicaSet. Różnica polega jedynie na typie, jaki tutaj zastosowaliśmy, a jest to Deployment. Drugą

zmianą jest rodzaj serwisu, z jakiego korzystamy. Wcześniej używaliśmy nginx teraz apache. Uruchamiamy nasz Deployment w identyczny sposób, jak to robiliśmy dotychczas:

```
kubectl apply -f apache-deployment.yaml
```

```
deployment.apps/apache-deployment created
```

14.3.2. Sprawdzenie funkcjonowania replik

Po wyświetleniu listy otrzymamy 3 pody:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|------------------------------------|-------|---------|----------|-----|
| apache-deployment-76687f8b5f-cbs45 | 1/1 | Running | 0 | 7s |
| apache-deployment-76687f8b5f-f2dj6 | 1/1 | Running | 0 | 7s |
| apache-deployment-76687f8b5f-xvhcb | 1/1 | Running | 0 | 7s |

Jeżeli usuniemy jeden:

```
kubectl delete po apache-deployment-845b49cb6c-8pcll
```

```
pod "apache-deployment-76687f8b5f-cbs45" deleted
```

To automatycznie zostanie stworzony nowy:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|------------------------------------|-------|---------|----------|-------|
| apache-deployment-76687f8b5f-blfxt | 1/1 | Running | 0 | 3s |
| apache-deployment-76687f8b5f-f2dj6 | 1/1 | Running | 0 | 4m58s |
| apache-deployment-76687f8b5f-xvhcb | 1/1 | Running | 0 | 4m58s |

Niestety w tym wypadku nie udało mi się wyłapać momentu, gdy pod jest ponownie generowany. Zawsze wyświetla się, gdy już był RUNNING. Jednak, że został stworzony nowy, świadczy o tym jego nazwa, jak i czas od jego uruchomienia, który jest krótszy niż pozostałe.

14.3.3. Lista z Deploymentami

Czyli typ Deployment działa w identyczny sposób jak ReplicaSet. W związku z tym, że tak jak poprzednik jest typem Kubernetesowym, posiada swoją własną listę, na której wyświetlane są wszystkie deploymenty. Aby wyświetlić wspomnianą, korzystamy z polecenia:

kubectl get deployment

| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|-------------------|-------|------------|-----------|-----|
| apache-deployment | 3/3 | 3 | 3 | 17m |

Natomiast jak tworzymy Deployment, to automatycznie jest tworzona ReplicaSet. Aby to potwierdzić, wyświetlamy jej listę:

kubectl get rs

| NAME | DESIRED | CURRENT | READY | AGE |
|------------------------------|---------|---------|-------|-----|
| apache-deployment-76687f8b5f | 3 | 3 | 3 | 24m |

Czyli jak prezentują to ostatnie przykłady, Deployment jest dość złożonym procesem tworzącym takie typy jak Pody, ReplicaSet i oczywiście generuje sam siebie. Zobaczmy teraz, co możemy, korzystając z niego, dodatkowo zrobić.

14.3.4. Aktualizacja wersji obrazu

W pliku korzystamy z wersji najnowszej httpd. Jeżeli chcielibyśmy ją zmienić, to możemy zmodyfikować plik yaml i następnie ponownie go uruchomić. Aktualizacja zostanie przeprowadzona automatycznie:

kubectl apply -f apache-deployment.yaml

deployment.apps/apache-deployment configured

Jednak w przypadku tego sposobu powstaje problem przy wyświetlaniu zmian. Poznajmy teraz polecenie do tego służące:

kubectl rollout history deployment apache-deployment

deployment.apps/apache-deployment

REVISION CHANGE-CAUSE

1 <none>

2 <none>

Wyświetlona została lista, z której można się domyślić, że coś zostało zmienione. Jednak nie wiadomo co. Dlatego, jeżeli chcemy dokonać jakichś zmian, lepiej jest to wykonać z użyciem polecenia, niż przez modyfikację pliku. Jeżeli zmodyfikowałeś plik i podałeś httpd jakąś wersję, to powróć teraz do najnowszej. I ponownie uruchom instrukcje z pliku. Aby dokonać zmian obrazu oraz nadać tej zmianie jakąś sensowną nazwę korzystamy z polecenia:

```
kubectl set image deployment/apache-deployment apache-deployment-container=httpd:2.4
```

Dzięki temu poleceniu jesteś w stanie zmienić obraz, jaki jest używany w danym kontenerze. Przeanalizujmy dokładnie przykład.

```
deployment/apache-deployment
```

Wskazujesz, że zmiana dotyczy określonego deploymentu.

```
apache-deployment-container=httpd:2.4
```

Podajesz nazwę kontenera, który wpisałeś w pliku yaml. Można się pomylić z nazwą obrazu. Przykład specjalnie skonstruowałem w powyższy sposób, by móc Ci zobrazować miejsce, w jakim możesz popełnić błąd. W większości przypadków w pliku yaml dobrze jest określić nazwę kontenera nazwą obrazu. Jednak to rada na przyszłość. Po uruchomieniu tego polecenia będzie działało się bardzo dużo istotnych rzeczy. Zaczniemy od podów. Jeżeli dość szybko po wprowadzeniu polecenia uruchomisz listę podów, możesz zaobserwować coś, co prezentuję na poniższym przykładzie:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|------------------------------------|-------|-------------------|----------|------|
| apache-deployment-74994f8c65-5g4pk | 0/1 | ContainerCreating | 0 | 2s |
| apache-deployment-74994f8c65-68pfw | 1/1 | Running | 0 | 5s |
| apache-deployment-74994f8c65-w4jlv | 1/1 | Running | 0 | 8s |
| apache-deployment-76687f8b5f-d2s8b | 1/1 | Terminating | 0 | 4m6s |
| apache-deployment-76687f8b5f-v429h | 1/1 | Running | 0 | 4m6s |

Nim kontener zostanie usunięty, musi powstać nowy na jego miejsce. Dzieje się to automatycznie. W ten sposób jest to wykonywane na wszystkich trzech kontenerach. Tworzony jest nowy, usuwany stary, a w trakcie usuwania powstaje kolejny itd.

Teraz zerknijmy do ReplicaSet:

```
kubectl get rs
```

| NAME | DESIRED | CURRENT | READY | AGE |
|------------------------------|---------|---------|-------|------|
| apache-deployment-74994f8c65 | 3 | 3 | 2 | 6s |
| apache-deployment-76687f8b5f | 1 | 1 | 1 | 4m4s |

W przypadku ReplicaSet jest tworzona nowa i nim zostanie ona zastąpiona, muszą wszystkie jej elementy zostać uruchomione. Po wykonaniu powyższych czynności ReplicaSet, która jest już nieużywana, nie zostaje usunięta. Nadal znajduje się na liście:

```
kubectl get rs
```

| NAME | DESIRED | CURRENT | READY | AGE |
|------------------------------|---------|---------|-------|-------|
| apache-deployment-74994f8c65 | 3 | 3 | 3 | 17s |
| apache-deployment-76687f8b5f | 0 | 0 | 0 | 4m15s |

Jednak wartości co do ilości uruchomionych podów wynoszą 0. Dlatego już nieużywany ReplicaSet został dezaktywowany, a na jego wszedł nowy z wprowadzonymi zmianami.

Teraz spójrzmy co dzieje się z deploymentem.

```
kubectl get deployments
```

| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|-------------------|-------|------------|-----------|------|
| apache-deployment | 3/3 | 2 | 3 | 4m2s |

Pod, nim zostanie usunięty, na jego miejsce musi powstać nowy z wprowadzonymi zmianami. Zwróć uwagę, wszystkie pody są gotowe, bo mamy 3/3, oraz 3 są dostępne, jednak udało mi się wyłapać, gdy jeszcze wszystkie nie były zaktualizowane. Po przeprowadzeniu wszystkich działań, które zakończyły się powodzeniem, otrzymamy wynik:

```
kubectl get deployments
```

| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|-------------------|-------|------------|-----------|-------|
| apache-deployment | 3/3 | 3 | 3 | 4m14s |

14.3.5. Opis do wprowadzonych zmian

Wiesz już jak zachowuje się Deployment. Na jakie zdarzenia ma wpływ i jakich typów Kubernetesowych dotyczy. Gdy wprowadziłem zmianę w obrazie, została ona odnotowana na liście.

Nim ją uruchomimy, chciałbym, abyś poznał, polecenie, dzięki któremu będziesz w stanie opisać, na czym ta zmiana polegała:

```
kubectl annotate deployment/apache-deployment kubernetes.io/change-cause="image httpd updated to 2.4"
```

Powysze polecenie umieści komentarz do deploymentu noszącego nazwę apache-deployment o treści image httpd updated to 2.4. Oczywiście my określamy treść komentarza, a to, co tutaj zaprezentowałem, jest przykładem. W celu wyświetlenia listy wprowadzonych zmian do deploymentu używamy polecenia:

```
kubectl rollout history deployment/apache-deployment
```

```
deployment.apps/apache-deployment
```

```
REVISION CHANGE-CAUSE
```

```
1      <none>
2      image httpd updated to 2.4
```

Po otrzymanym wyniku można się domyślić, że służy do wyświetlenia historii rolek określonego deploymentu. Zwróć uwagę, że pierwsza wartość jest określona jako <none>. Po uruchomieniu pierwszy raz deploymentu został stworzony taki zapis. Jednak Kubernetes nie dodaje komentarzy, dlatego jest tylko widoczny <none>. Po wprowadzeniu zmian, a dokładniej, gdy zmieniliśmy wersję obrazu z latest na 2.4, dodaliśmy komentarz, który jest widoczny pod drugą pozycją. Gdybyśmy tego nie zrobili to w tym miejscu, również pojawiłby się opis <none>.

14.3.6. Cofanie wprowadzonych zmian

W pewnym momencie możemy zechcieć wrócić do pierwotnego stanu. Czyli wycofać wprowadzone zmiany. W tym pomoże nam polecenie:

```
kubectl rollout undo deployment/apache-deployment
```

```
deployment.apps/apache-deployment rolled back
```

Teraz zerknijmy ponownie na listę wprowadzonych zmian:

```
kubectl rollout history deployment/apache-deployment
```

```
deployment.apps/apache-deployment
```

```
REVISION CHANGE-CAUSE
```

```
2      image httpd updated to 2.4
3      <none>
```

Pierwsza już się nie pojawiła na liście, ponieważ jest identyczna z 3. Jednak nadal pojawia się ona jako <none>. Wprowadźmy teraz komentarz do naszej rolki:

```
kubectl annotate deployment/apache-deployment kubernetes.io/change-cause="rollback to image httpd:latest"
```

deployment.apps/apache-deployment annotated

I ponownie wyświetlamy listę:

```
kubectl rollout history deployment/apache-deployment
```

deployment.apps/apache-deployment

REVISION CHANGE-CAUSE

2 image httpd updated to 2.4

3 rollback to image httpd:latest

Mam nadzieję, że rozumiesz sens dodawania komentarzy tego typu. Bez nich z <none> nie jesteśmy w stanie określić jaką zmianą została wprowadzona. Jest to informacje dla nas i dla innych jakie zmiany zostały wprowadzone.

14.3.7. Sprawdzanie ustawień deploymentu

Wiemy jak wprowadzić zmiany, jednak nie wiemy, w jaki sposób sprawdzić co znajduje się w danym deploymencie. Przyznam, że są na to dwa sposoby. Pierwszy najczęściej stosowany polega na wyświetlaniu opisu deploymentu przy pomocy polecenia:

```
kubectl describe deployment apache-deployment
```

W wyświetlonym wyniku odnajdujemy rubrykę:

...

Pod Template:

Labels: app=app-apache-deployment

Containers:

apache-deployment-container:

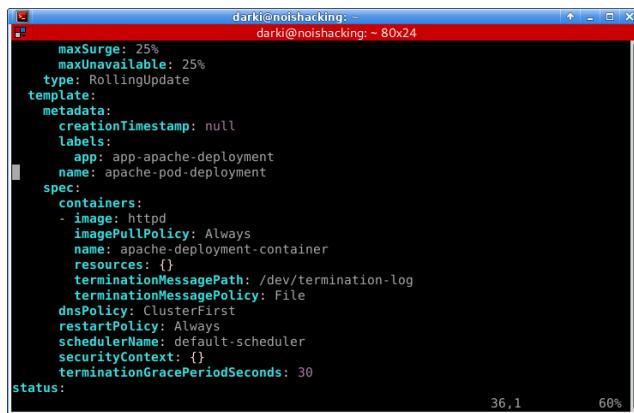
Image: httpd

...

Niepotrzebny kod został pominięty. Zgodnie z informacją, jaką otrzymaliśmy obraz, z jakiego korzystamy w podach to httpd, czyli najnowszy obraz apache2. Drugim sposobem jest skorzystanie z polecenia edit w następujący sposób:

kubectl edit deployments apache-deployment

Zostanie uruchomiony edytor vim, a w nim pojawi się kompletna konfiguracja deploymentu w postaci pliku yaml. Jednak ten plik nie zawiera tylko informacji, które my w nim umieściliśmy. Deployment czy też pod posiada szereg instrukcji, których standardowo nie musimy ustawiać, jeżeli nie chcemy. Są one automatycznie uzupełniane, co udowadnia edytowany przez nas plik. Zerknij na obrazek poniżej:



```
darki@noishacking: ~
darki@noishacking: ~ 80x24

maxSurge: 25%
maxUnavailable: 25%
type: RollingUpdate
template:
  metadata:
    creationTimestamp: null
    labels:
      app: app-apache-deployment
      name: apache-pod-deployment
  spec:
    containers:
      - image: httpd
        imagePullPolicy: Always
        name: apache-deployment-container
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
status: 36,1 60%
```

Przeanalizuj plik, zauważ ile możliwości konfiguracyjnych, znajduje się w nim. Natomiast na obrazku powyżej odnajdziesz również wersję obrazu, z jakiego korzysta deployment.

UWAGA: Aby wyjść z edytora vi czy też vim należy wcisnąć klawisz ESC, następnie przytrzymując klawisz SHIFT odnaleźć klawisz odpowiadający za : (dwukropka). Gdy w dolnej części zacznie migać kursor, wpisujemy !q, co umożliwi zamknięcie pliku i powrót do powłoki.

Sprawdźmy teraz, czy to działa i ponownie zmieńmy obraz na 2.4:

kubectl set image deployment/apache-deployment apache-deployment-container=httpd:2.4

I użyjmy describe na deployment:

kubectl describe deployment apache-deployment

Pod Template:

Labels: app=app-apache-deployment

Containers:

apache-deployment-container:

Image: httpd:2.4

Jak widać po przykładzie, obraz został zmieniony. Teraz, jeżeli chcesz, to sprawdź, to samo przy pomocy polecenia edit. W ramach ćwiczeń dodaj opis do wprowadzonej zmiany, a następnie wykonaj rolkę do wersji najnowszej. Na końcu posprawdzaj czy wszystkie zmiany zostały prawidłowo wprowadzone.

14.3.8. Zmiana ilości replik

W ramach tego materiału chciałbym jeszcze pokazać jedną pożyteczną opcję. A chodzi o możliwość zmiany ilości replik. Obecnie mamy 3, zmieńmy na 5:

```
kubectl scale --replicas=5 deployment/apache-deployment
```

Sprawdzamy po kolej, jak zachowują się poznane w tym materiale typy:

Pody:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|------------------------------------|-------|-------------------|----------|-----|
| apache-deployment-74994f8c65-4bldf | 1/1 | Running | 0 | 13m |
| apache-deployment-74994f8c65-jxdmk | 1/1 | Running | 0 | 13m |
| apache-deployment-74994f8c65-scb7t | 1/1 | Running | 0 | 13m |
| apache-deployment-74994f8c65-tg8k9 | 0/1 | ContainerCreating | 0 | 3s |
| apache-deployment-74994f8c65-trxdv | 0/1 | ContainerCreating | 0 | 3s |

Tworzone są dodatkowe 2 pody.

ReplicaSet:

```
kubectl get rs
```

| NAME | DESIRED | CURRENT | READY | AGE |
|------------------------------|---------|---------|-------|-----|
| apache-deployment-74994f8c65 | 5 | 5 | 3 | 20h |
| apache-deployment-76687f8b5f | 0 | 0 | 0 | 20h |

Czyli mamy 5 replik, z czego 3 są gotowe. Za chwilę zamiast 3 będzie 5.

Deployment:

```
kubectl get deployments
```

| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|-------------------|-------|------------|-----------|-----|
| apache-deployment | 3/5 | 5 | 3 | 20h |

Taka sama sytuacja, deployment czeka tylko na stworzenie dodatkowych podów. Jeżeli uważałesz przez cały ten moduł, to wynik nie powinien Cię w jakiś sposób zaskoczyć. Natomiast zobaczymy, co się stanie, jak wpiszemy ilość replik 0:

```
kubectl scale --replicas=0 deployment/apache-deployment
```

Pody natychmiast zostaną usuwane:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|------------------------------------|-------|-------------|----------|-----|
| apache-deployment-74994f8c65-4bldf | 1/1 | Terminating | 0 | 24m |
| apache-deployment-74994f8c65-jxdmk | 1/1 | Terminating | 0 | 24m |
| apache-deployment-74994f8c65-scb7t | 1/1 | Terminating | 0 | 24m |
| apache-deployment-74994f8c65-tg8k9 | 1/1 | Terminating | 0 | 11m |
| apache-deployment-74994f8c65-trxdv | 1/1 | Terminating | 0 | 11m |

Po krótkim czasie jak ponownie wyświetlisz listę podów, otrzymasz informację:

```
kubectl get po
```

No resources found in default namespace.

W przypadku ReplicaSet:

```
kubectl get rs
```

| NAME | DESIRED | CURRENT | READY | AGE |
|------------------------------|---------|---------|-------|-----|
| apache-deployment-74994f8c65 | 0 | 0 | 0 | 20h |
| apache-deployment-76687f8b5f | 0 | 0 | 0 | 20h |

Deployment:

```
kubectl get deployments
```

| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|-------------------|-------|------------|-----------|-----|
| apache-deployment | 0/0 | 0 | 0 | 20h |

W przypadku deploymentu zwróć uwagę, że nie został on usunięty. Jedynie pody, które obsługiwał, zostały usunięte, natomiast on nadal istnieje. Dlatego to, co zrobiliśmy, możemy określić jako tymczasowe wyłączenie deploymentu. Aby go aktywować, wystarczy ustawić mu odpowiednią ilość replik.

14.3.9. Inny sposób wprowadzenia ustawień

Jednak polecenie, z którego korzystamy, nie jest jedyną możliwością ustawienia replik w Kubernetesie. Pamiętasz jakąś chwilę temu, używaliśmy polecenia edit do sprawdzenia wersji obrazu. Tak naprawdę dzięki temu narzędziu możemy edytować wartości tam ustalone. Czyli nie musimy posługiwać się poleceniem do zmiany obrazu, czy też poleceniem do zmiany replik. Wystarczy, że do odpowiedniej instrukcji wpiszemy odpowiednią wartość. Dlatego teraz edytujemy nasz obraz:

UWAGA: Edytor vim czy też vi dla osób, które nigdy z niego nie korzystały, może sprawić pewne problemy. Jak zakończyć działanie programu pokazywałem w poprzedniej uwadze. Natomiast w celu edycji pliku najlepiej jest wcisnąć klawisz ESC, następnie klawisz i. Wtedy wejdziemy w tak zwany tryb edycji, dzięki któremu będziemy mogli zmieniać wartości tak jak w klasycznym edytorze tekstu. Aby wprowadzić zmiany, klikamy klawisz ESC, następnie przytrzymując klawisz SHIFT, odnajdujemy i wciskamy klawisz dwukropka (:). Na dole pojawi się migający kurSOR. Wpisujemy wq i wciskamy enter. Zmiany zostaną zachowane i przejdziemy do powłoki systemowej.

kubectl edit deployments apache-deployment

Odnajdujemy miejsce, gdzie znajduje się ilość replik:

```
spec:  
  progressDeadlineSeconds: 600  
  replicas: 0  
  revisionHistoryLimit: 10  
  selector:  
    ...
```

Jak widzisz, wpisana jest tutaj wartość 0, czyli taka, jaką ustawiliśmy w deploymencie. Zmieńmy ją teraz na 3:

```
spec:  
  progressDeadlineSeconds: 600  
  replicas: 3  
  revisionHistoryLimit: 10  
  selector:  
    matchLabels:  
      app: app-apache-deployment
```

Pobawmy się trochę tym. Odnajdźmy miejsce, gdzie znajduje się obraz:

```
spec:  
  containers:  
  - image: httpd:2.4  
    imagePullPolicy: Always  
    name: apache-deployment-container  
    resources: {}
```

Usuwamy wersję razem z dwukropkiem:

```
spec:  
  containers:  
  - image: httpd  
    imagePullPolicy: Always  
    name: apache-deployment-container  
    ...
```

Jeżeli masz obraz w wersji najnowszej, tzn. bez określenia wersji to wpisz zamiast samego httpd na przykład httpd:2.4. Po wprowadzeniu powyższych zmian zapisujemy plik. Jeżeli szybko wyświetlisz listę podów, zauważysz, że są one w trakcie tworzenia:

kubectl get po

| NAME | READY | STATUS | RESTARTS | AGE |
|------------------------------------|-------|-------------------|----------|-----|
| apache-deployment-76687f8b5f-89xpl | 0/1 | ContainerCreating | 0 | 5s |
| apache-deployment-76687f8b5f-fnc6l | 1/1 | Running | 0 | 5s |
| apache-deployment-76687f8b5f-j2g2t | 0/1 | ContainerCreating | 0 | 5s |

Pozostałe przykłady typów tym razem pominę. Sprawdź sam, jak to wszystko wygląda. W tym miejscu jeszcze chciałbym sprawdzić, czy został podmieniony również obraz:

kubectl describe deployment apache-deployment

Pod Template:

Labels: app=app-apache-deployment

Containers:

apache-deployment-container:

Image: httpd

Jak widzisz, zmiany zostały wprowadzone. Jest to najnowszy obraz httpd. Natomiast jeżeli zerknijmy na listę wprowadzonych zmian, to otrzymamy:

kubectl rollout history deployment/apache-deployment

deployment.apps/apache-deployment

REVISION CHANGE-CAUSE

4 rollback to image httpd:latest

5 rollback to image httpd:latest

Zauważ, że dwa razy pojawił się komentarz, który ustawiliśmy, gdy wykonywaliśmy rolkę. Niestety ten element nie jest na tyle inteligentny, by wprowadzić inny komentarz. Dlatego, jeżeli ustawiśmy jakiś komentarz, następnie dokonaliśmy jakichkolwiek zmian, w prezentowane sposoby spowoduje ponowne jego umieszczenie na liście. Dlatego tekst komentarza określamy jako statyczny. Pierwotna jego wartość była <none>, a my dokonaliśmy zmian i zastąpiliśmy wartość <none> własną. Jest to dość ważna informacja. Musisz pamiętać niezależnie od tego, w jaki sposób wprowadzasz zmiany w deploymencie, wypada odpowiednio to opisać w komentarzu. Inaczej po jakimś czasie gdy do tego zajrzysz, nie będziesz miał zielonego pojęcia, jakie zmiany w nim wprowadziłeś.

14.3.10. Usuwanie Deploymentu

Jeżeli chcemy usunąć deployment, korzystamy z polecenia:

```
kubectl delete deployments apache-deployment
```

```
deployment.apps "apache-deployment" deleted
```

Usunie to zarówno pody, jak i ReplicaSet. Jak pamiętasz, były dwie i dwie zostaną usunięte. To polecenie usuwa wszystko, co związane jest z określonym deploymentem, niezależnie od tego, czy jest aktywne, czy też nie. Wystarczy by, miało tylko z nim związek.

Ostatnie, o czym chciałbym wspomnieć to to, że deployment automatycznie uruchamia, ReplicaSet, co sobie potwierdziliśmy już nie raz. Jednak by była jasność Deployment automatycznie uruchamia ReplicaSet i tak naprawdę to ono dba o ilość replik w naszym deploymencie. My dzięki typowi deployment nie musimy o tym myśleć.

14.4. Podsumowanie

Jak zapewne zauważłeś, bardzo dużo rozmawialiśmy o Deploymencie. Jest to jeden z najważniejszych typów po podzie w Kubernetesie. Zrozumienie tego tematu jest bardzo istotne dlatego, jeżeli czegoś nie rozumiesz, przejrzyj ten go ponownie. Zapewniam Cię, że jeżeli dobrze zrozumiesz deployment, to pozostałe będą jedynie poznawaniem dodatkowych możliwości lub innych zastosowań Kuberntesa.

Rozdział 15: Wykonywanie pojedynczych zadań

W poprzednim, module zajęliśmy się bardzo ważną częścią Kuberntesa. Otóż poznaleś typ Deployment. W tym miejscu powinieneś wiedzieć, czym jest, z czego się składa, w jaki sposób go uruchomić oraz sprawdzić, czy odpowiednio działa. Jeżeli do tego typu podsiedłeś bardzo uważnie i ze wszystkim, co tam napisałem, zapoznałeś się w najmniejszych szczegółach, to teraz powinno być Ci łatwiej. Technika rozumowania czym jest Kubernetes i jak działa, została zawarta właśnie w tamtym miejscu. My teraz będziemy poznawali typy, które działają w podobny sposób lub będziemy poznawali typy, które wspomagają to, co już, znasz. Dlatego, jeżeli nie czujesz się dobrze w deploymencie, to wróć do poprzedniego modułu i przerób jeszcze raz tamten materiał.

15.1. Typ Job

Pierwszym typem, jakim się zajmiemy, będzie Job. **Jest on typem, który służy do wykonania jakiegoś pojedynczego zadania.** Po jego wykonaniu kończy swoje działanie. Można go porównać z Dockerowym rozwiązaniem, gdy uruchamiamy kontener, on wykonuje swoje czynności, a następnie po wykonaniu ich kończy swoje działanie. Zerknij na poniższy przykład:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: date-job
spec:
  template:
    spec:
      containers:
        - name: date-job-container
          image: debian
          command: ["sh", "-c", "date > /root/date.txt"]
      volumeMounts:
        - mountPath: /root
          name: date-volume
    restartPolicy: Never
    volumes:
      - name: date-volume
        hostPath:
          path: /tmp
        type: Directory
```

Dzięki niemu omówimy dwa tematy. Otóż jeżeli przeanalizujesz powyższy przykład, to wiele z elementów w nim zawartych już omawialiśmy, jednak są **pewne nowości jak woluminy**.

15.1.1. Ponownie o konstrukcji

Choć dużo o tym rozmawialiśmy, to w ramach przypomnienia omówimy sobie krok po kroku cały przykład.

```
apiVersion: batch/v1
```

```
kind: Job
```

```
metadata:
```

```
  name: date-job
```

O tej części kodu mówiłem w pierwszym module. **Jest to podstawowa konstrukcja pliku yaml Kuberntesesa.** Każdy z plików musi rozpoczynać się od tych elementów. Natomiast *apiVersion* określiliśmy jako pewnego rodzaju repozytorium, w którym są zawarte informacje o typach i ich obsłudze. **W tym przykładzie korzystamy z typu Job, który wpisaliśmy w etykietę kind.** W swoich przykładach od tej pory zawsze będę korzystał z języka angielskiego i Ty również powinieneś. Wiem, że wcześniej używałem polskich nazw. Było to w celu łatwiejszego zrozumienia tematu. Od teraz posługiwać się będę tylko angielskimi nazwami.

15.1.2. Zapisanie pliku z wynikiem

```
spec:
```

```
  template:
```

```
    spec:
```

Tworzymy szablon, który ma obsługiwać nasz typ, czyli job.

```
      containers:
```

```
        - name: date-job-container
```

```
          image: debian
```

```
          command: ["sh", "-c", "date > /root/date.txt"]
```

Określamy, że będzie tworzony kontener noszący nazwę *date-job-container* z najnowszego obrazu *debian*. **Następnie w nim będzie wykonywało się polecenie wypisujące aktualną datę.** Ta informacja zostanie zapisana do pliku *date.txt* w katalogu *root* kontenera. Tutaj zwróć uwagę, na ostatnie zdanie, **ten plik będzie znajdował się w katalogu root kontenera, nie węzła, nie hosta, tylko kontenera.** Powtarzam, ponieważ jest to bardzo istotne.

15.1.3. Udostępnienie katalogu za pomocą wolumina

volumeMounts:

- mountPath: /root

- name: date-volume

Tu wykonujemy zupełnie nową czynność. O woluminach rozmawialiśmy w części o Dockerze. Nie różnią się one od tego, co poznaliśmy tam. Jednak sam zapis jest trochę inny. W przykładzie informujemy, że chcemy, aby udostępniony został katalog *root*. Popatrz, stworzyliśmy listę, dzięki znakowi - (*myślnik*). Pod tą listą określiliśmy nazwę, pod którą czynność udostępnienia następuje. Dzięki temu, że nazwaliśmy tę czynność, będziemy mogli do niej w dalszej części pliku się odwołać. Jednak w tym miejscu jeszcze chcę z Tobą porozmawiać, na temat tego, co do tej pory zrobiliśmy. Otóż we wcześniej omawianym kodzie wywołaliśmy polecenie:

```
command: ["sh", "-c", "date > /root/date.txt"]
```

Dzięki niemu wynik zostanie zapisany do pliku *date.txt* w folderze *root*. O co mi teraz chodzi! Jeżeli do katalogu */root* zapisujesz jakieś dane, to ten sam katalog musisz udostępnić. Tak jak to zrobiliśmy w naszym przykładzie. Pozwolisz, że się powtórzę:

```
command: ["sh", "-c", "date > /root/date.txt"]
```

W powyższy sposób tworzymy plik w katalogu *root*.

- mountPath: /root

W ten sposób udostępniamy ten katalog. Jest to bardzo ważne, bo jeżeli udostępnisz inny katalog, to nie będziesz miał dostępu do stworzonych danych przez kontener w trakcie jego stworzenia.

15.1.4. Restart Policy

Zerknijmy na dalszą część przykładu.

restartPolicy: Never

O *restartPolicy* również już rozmawialiśmy. Job służy do wykonania jakiegoś zadania, następnie po jego wykonaniu kończy swoje działanie. Dlatego wartość w przypadku szczególnie typu Job

powinniśmy ustawić na *Never*, by uniemożliwić mu restart. Jeżeli nie określisz tego elementu przy uruchomieniu, otrzymasz informacje:

```
The Job "date-job" is invalid: spec.template.spec.restartPolicy: Required value: valid values: "OnFailure", "Never"
```

Dlatego zawsze musisz ustawić tę wartość. Jak tego nie zrobisz, Kubernetes Ci o tym przypomni.

5.1.5. Określenie miejsca udostępnienia

volumes:

```
- name: date-volume
```

hostPath:

```
path: /tmp
```

```
type: Directory
```

Jest to moment, w którym określamy gdzie katalog */root*, który udostępnia kontener, ma być zamontowany. W tym miejscu posługujemy się nazwą, jaką określiliśmy w kontenerze. W związku z tym, że Kubernetes wie, czego dotyczy dane ustawienie i wie, że udostępniliśmy katalog *root* z kontenera, przy pomocy etykiety *path* określamy miejsce w węźle, gdzie mają być udostępnione pliki. W związku z tym, że udostępniamy katalog, to jako *type* wpisujemy *Directory*.

15.1.6. Uruchomienie Job

Na tym kończy się nasz plik. Pozostało tylko go uruchomić. Dlatego bez dalszej zwłoki robimy to przy pomocy znanego już nam polecenia:

kubectl apply -f job.yaml

Tak jak w przypadku każdego typu Kubernetesowego, job posiada swoją oddzielną listę. Wywołujemy ją za pomocą polecenia:

kubectl get job

Po uruchomieniu naszego Job'a zostanie utworzony pod:

kubectl get pod

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------|-------|-------------------|----------|-----|
| date-job-7t7w2 | 0/1 | ContainerCreating | 0 | 1s |

Po jego utworzeniu oraz wykonaniu zleconych mu zadań zakończy swoje działanie:

```
kubectl get pod
```

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------|-------|-----------|----------|-----|
| date-job-7t7w2 | 0/1 | Completed | 0 | 5s |

Adekwaźnie do tego przykładu, na liście job będzie podobnie. Nim wykona swoje czynności, będzie to wyglądało w następujący sposób:

```
kubectl get job
```

| NAME | COMPLETIONS | DURATION | AGE |
|----------|-------------|----------|-----|
| date-job | 0/1 | 2s | 2s |

Po wykonaniu:

```
kubectl get job
```

| NAME | COMPLETIONS | DURATION | AGE |
|----------|-------------|----------|-----|
| date-job | 1/1 | 6s | 6s |

Czyli wszystko zostało wykonane poprawnie.

15.1.7. Miejsce zapisania plików

Teraz pytanie brzmi jak dostać się do danych, które udostępniliśmy. Jeżeli zerkniesz do folderu `tmp`, w swoim systemie hosta to nie odnajdziesz tam nic takiego. Nie wiem, czy pamiętasz, jak opisywałem przed chwilą część kodu odpowiedzialną za montowanie katalogu. **Katalog kontenera `/root` jest montowany w katalogu `/tmp` węzła**. Czyli nie w naszym systemie hosta tylko węzła, a naszym węzłem jest minikube. Dlatego, aby uzyskać dostęp do danych, musimy wejść do minikube. W jednym z modułów pokazywałem, jak to się robi, jednak dla przypomnienia zróbmy to ponownie. Aby wejść do naszego minikube, korzystamy z połączenia ssh w następujący sposób:

```
minikube ssh
```

Zostaniesz zalogowany do powłoki minikube. Teraz przejdźmy do określonego przez nas folderu:

```
cd /tmp/
```

Następnie sprawdźmy, czy plik znajduje się w katalogu:

```
ls date.txt
```

Wyświetlamy jego zawartość:

```
cat date.txt
```

Otrzymamy pożądana przez nas informację. Oczywiście wszystko to możemy zrobić w znacznie skrócony sposób:

```
ls /tmp/date.txt  
cat /tmp/date.txt
```

Jednak chciałem zrobić to mniejszymi krokami. Aby wyjść z minikube, wpisujemy *exit*.

15.1.8. Usuwanie Joba

Prezentowany typ tak jak wspomniałem, służy do wykonania jakieś czynności, a następnie na zakończeniu swojego działania. Natomiast w celu jego usunięcia po wykonaniu wszystkich zadań możemy skorzystać z dwóch sposobów. Pierwszy z nich polega na wpisaniu polecenia:

kubectl delete job date-job

Drugi na wprowadzeniu do etykiety informacji, że ma się usunąć automatycznie, po wykonaniu wszystkich czynności. Taka etykieta wyglądać powinna następująco:

```
ttlSecondsAfterFinished: 5
```

Po 5 sekundach, od wykonania wszystkich czynności, job usunie się sam. Cały plik yaml z tą etykietą powinien wyglądać następująco:

```
apiVersion: batch/v1  
kind: Job  
metadata:  
  name: date-job  
spec:  
  ttlSecondsAfterFinished: 5  
  template:  
    spec:  
      containers:  
        - name: date-job-container  
          image: debian  
          command: ["sh", "-c", "date > /root/date.txt"]  
      volumeMounts:  
        - mountPath: /root  
          name: date-volume  
    restartPolicy: Never  
    volumes:
```

```
- name: date-volume  
hostPath:  
path: /tmp  
type: Directory
```

Jeżeli nie usunąłeś poprzedniego joba, zrób to teraz. Zwróć uwagę na umiejscowienie opisanej przed chwilą etykiety. Nie znajduje się ona w template tylko przed. Jest to spowodowane tym, że usunięcie dotyczy całego Job'a, dlatego ta funkcja jest umiejscowiona przed *template*. Pozostało uruchomić nasz Job:

kubectl apply -f job.yaml

Zerknijmy, co się będzie dzieło. Na początku kontenery:

kubectl get pod

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------|-------|-----------|----------|-----|
| date-job-p79kd | 0/1 | Completed | 0 | 5s |

Czynności, jakie wykonuje Job, są kończone po 5 sekundach. Udało mi się to wyłapać. Zerknij na przykład z listą Jobów:

kubectl get job

| NAME | COMPLETIONS | DURATION | AGE |
|----------|-------------|----------|-----|
| date-job | 1/1 | 6s | 6s |

Natomiast analogicznie, gdy osiągniemy 10 sekund:

kubectl get job

| NAME | COMPLETIONS | DURATION | AGE |
|----------|-------------|----------|-----|
| date-job | 1/1 | 6s | 10s |

Job nadal istniał, jednak po tym, gdy ponownie wyświetlić chciałem listę:

kubectl get job

No resources found in default namespace.

Oznacza to, że Job został usunięty i wystąpiło to najprawdopodobniej w 11 sekundzie. Jeżeli zerkniesz na listę podów, będzie ona też pusta.

Teraz mam zagadkę dla Ciebie. Co się stało z plikiem, który stworzył Job i został on udostępniony w katalogu */tmp* węzła? To już pozostawiam Ci do sprawdzenia, jednak woluminy działają identycznie jak w Dockerze. Dlatego, jeżeli pamiętasz tamtą część, to odpowiedź już znasz.

Stosowanie elementu automatycznego usuwania Joba jest dobrą praktyką. Pamiętaj o sensie tego typu, wykonaj swoje i zakończ działanie. Poza tym nie ma on żadnego większego sensu, dlatego ważne

jest, aby usunąć go po tym, jak wykona swoje zadania. Jest to o tyle wygodne, że dbamy w tej sposób o porządek zarówno w jobach, jak i podach. Job był idealny do wytłumaczenia woluminów w Kubernetesie, dlatego też pozwoliłem sobie upiec dwie pieczenie na jednym ogniu.

15.2. CronJob

Znasz narzędzie **Cron**? Dzięki niemu w systemie Linux jesteś w stanie wskazać zadania, jakie ma wykonać w określonym czasie. W Kubernetesie również występuje możliwość korzystania z crona. Zerknij na poniższy kod:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: update-cronjob
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      ttlSecondsAfterFinished: 5
      template:
        spec:
          containers:
            - name: update-container
              image: debian
              command: ["/bin/sh", "-c"]
              args: ["apt update && apt upgrade -y"]
        restartPolicy: OnFailure
```

Tym razem początek pozwolę sobie pominąć. Opis pojawiał się we wszystkich modułach.

15.2.1. Ustawienie harmonogramu

```
spec:
  schedule: "*/1 * * * *"
```

Jest to miejsce, w którym określasz CronJob, w jaki sposób ma działać. Reguły ustawiasz w identyczny sposób jak w klasycznym cronie. W tym przykładzie ustawiamy harmonogram dla crona, by uruchamiał się co minutę.

15.2.2. Automatyczne usunięcie poda po zakończeniu zadania

```
jobTemplate:  
  spec:  
    ttlSecondsAfterFinished: 5
```

Tym razem nie korzystamy z samego template, tylko pojawia się *jobTemplate*. Jest to spowodowane wskazanymi zadaniami, jakie ma wykonać Job. Wcześniej ustawialiśmy harmonogram i przy pomocy *jobTemplate* tworzymy szablon dla tego zadania. W przypadku samego Job nie musielibyśmy tego robić, w przypadku CronJob jest to wymagane. Dla tego schematu dodatkowo ustawiamy opcję, by po wykonaniu wszystkich zadań sam się usunął. W tym wypadku jest to wymóg. Wyobraź sobie, że co minutę tworzy Ci się nowy kontener. Po godzinie masz ich 60. Dlatego w przypadku CronJob pamiętaj, by z tego zawsze korzystać.

15.2.3. Tworzymy kontener

```
template:  
  spec:  
    containers:  
      - name: update-container  
        image: debian
```

Dalej już standardowa konstrukcja, z którą się spotkałeś niejednokrotnie w tym materiale. Tworzymy kontener noszący nazwę *update-container* z obrazu *debian*.

15.2.4. Czynności jakie mają być wykonane

```
command: ["/bin/sh", "-c"]  
args: ["apt update && apt upgrade -y"]
```

W tym miejscu przy pomocy *command* ustawiamy powłokę, jakiej chcemy użyć, natomiast przy pomocy *args* przekazujemy polecenia. W ten sposób jesteśmy w stanie wywołać każde polecenie powłoki Linux.

15.2.5. Restart Policy

```
restartPolicy: OnFailure
```

Natomiast na końcu ustawiamy *restartPolicy* tak, aby kontener uruchomił się ponownie, gdy nie uda mu się wykonać zadania.

15.2.6. Uruchomienie CronJob'a

Omówiliśmy cały przykład. Pozostaje go tylko uruchomić. Robimy to w znany już nam sposób:

```
kubectl apply -f cronjob.yaml
```

Teraz musisz odczekać minutę, aby zadanie się uruchomiło:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------------|-------|-------------------|----------|-----|
| update-cronjob-27903306-29mtk | 0/1 | ContainerCreating | 0 | 0s |

Po stworzeniu kontenera są wykonywane wszystkie zadania, jakie mu wyznaczyliśmy:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------------|-------|---------|----------|-----|
| update-cronjob-27903306-29mtk | 1/1 | Running | 0 | 3s |

Po ich wykonaniu kontener kończy swoje działanie:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------------|-------|-----------|----------|-----|
| update-cronjob-27903306-29mtk | 0/1 | Completed | 0 | 5s |

I jest usuwany po 11 sekundzie:

```
kubectl get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------------|-------|-----------|----------|-----|
| update-cronjob-27903306-29mtk | 0/1 | Completed | 0 | 11s |

Po tym, jak wyświetliłem ponownie listę, była ona pusta:

```
kubectl get po
```

No resources found in default namespace.

Natomiast tak jak w każdym typie, CronJob posiada swoją listę. Wyświetlamy ją za pomocą dwóch poleceń:

```
kubectl get cronjob
```

```
kubectl get cj
```

| NAME | SCHEDULE | SUSPEND | ACTIVE | LAST SCHEDULE | AGE |
|----------------|-------------|---------|--------|---------------|-------|
| update-cronjob | */1 * * * * | False | 1 | 3s | 5m34s |

Osobiście preferuję wersję skróconą, jednak jeżeli jest Ci wygodniej, możesz również używać dłuższej.

15.2.7. Usunięcie CronJob

W celu usunięcia CronJob używamy polecenia:

```
kubectl delete cj update-cronjob
```

```
cronjob.batch "update-cronjob" deleted
```

Oczywiście możesz użyć *cronjob* zamiast *cj*.

15.3. Podsumowanie

Poznaliśmy kolejna dwa typy, dzięki którym wykonywane są zadania. Następnie Kubernetes kończy działanie każdego z nich. Poznałeś sposoby ich obsługi, jak i również powinieneś wiedzieć, do czego służą i jak nimi zarządzać. Jednak to jeszcze nie wszystko...

Rozdział 16: Przestrzenie nazw, zmienne środowiskowe i sekrety

Kubernetes posiada bardzo wiele typów, o czym przekonałeś się, przerabiając dotychczasowy materiał. W tym chciałbym, jeszcze byś poznał trzy, które uważam, za ważne szczególnie na samym początku jego poznawania. Dlatego bez dłuższego wstępu przejdźmy do szczegółów.

16.1. Namespaces - przestrzenie nazw

Pozwolę sobie przytoczyć jeden z przykładów z ostatniego modułu:

kubectl get po

No resources found in default namespace.

Taką informację otrzymaliśmy, gdy typ CronJob zakończył swoje działanie oraz usunął pod. Tłumacząc informację z przykładu, na język polski *Nie znaleziono zasobów w domyślnej przestrzeni nazw.*

16.1.1. Co to są namespaces - przestrzenie nazw?

Namespaces, czy też jeżeli wolisz przestrzenie nazw, służą do organizowania i oddzielania różnych zasobów w klastrze. Umożliwiają one grupowanie zasobów jakie powstają dzięki tworzeniu różnorodnych typów. Można na przykład stworzyć przestrzeń nazw dla określonej aplikacji, a dokładniej dla typów z której się ona składa. Korzystając z namespaces, można również uniknąć konfliktów nazw jakie mogą powstać przy tworzeniu typów. Każda przestrzeń nazw ma swoją unikatową nazwę, dzięki czemu można w niej używać tych samych nazw dla zasobów jak na przykład w tej z której do tej pory korzystaliśmy. Inaczej mówiąc w dwóch przestrzeniach nazw mogą występować te same nazwy podów, deploymentów itd. Tworząc jakikolwiek typ w Kubernetes, możesz określić, do której przestrzeni ma zostać on przypisany. Jeśli tego nie zrobisz, zasób zostanie dodany do domyślnej. Tak jak to było do tej pory. W jednym zdaniu przestrzenie nazw pozwalają na lepszą organizację i zarządzanie zasobami w klastrze.

16.1.2. Wyświetlanie dostępnych przestrzeni nazw

Jak można wywnioskować, jeżeli jest domyślna przestrzeń nazw, to mogą być też inne. W celu wyświetlenia wszystkich, dostępnych korzystamy z polecenia:

kubectl get namespaces

| NAME | STATUS | AGE |
|-----------------|--------|-----|
| default | Active | 35d |
| kube-node-lease | Active | 35d |

```
kube-public     Active  35d  
kube-system     Active  35d
```

Jeżeli chcesz, możesz skorzystać ze skróconej wersji nazwy:

```
kubectl get ns
```

Wynik będzie identyczny. Standardowo wszystko, co do tej pory robiliśmy, było wykonywane w standardowej (*default*) przestrzeni nazw. Jeżeli zapoznałeś się z sieciami w Docker, to może kojarzysz pewną zbieżność.

16.1.3. Tworzenie własnej przestrzeni nazw

Jeżeli wszystkie pody były uruchamiane w standardowej przestrzeni nazw, to znaczy, że wiedziały o swoim istnieniu i mogły się ze sobą komunikować. Taka możliwość często może okazać się problemem. Dlatego, jeżeli nie chcemy, aby tak było, musimy stworzyć własną. Następnie typom, które tworzymy powiedzieć, że mają się w niej uruchamiać. W ten sposób wszystko, co znajduje się w danej przestrzeni nazw, będzie mogło komunikować się tylko między sobą. Nie trafią do niej jakieś przypadkowe elementy. Aby taką utworzyć, powinniśmy napisać podobny plik yaml do poniższego:

```
apiVersion: v1  
  
kind: Namespace  
  
metadata:  
  
  name: workplace
```

Jak widać po przykładzie, nie jest to nic skomplikowanego. Powyższy schemat występował w każdym przykładzie, jaki do tej pory wykonywaliśmy. W tym wypadku jedyne co wpisaliśmy innego, to jako typ ustawiliśmy *Namespace*. Pozostaje nam tylko uruchomić powyższy plik yaml w tradycyjny już sposób:

```
kubectl apply -f namespaces.yaml
```

```
namespace/workplace created
```

Zerknijmy teraz na listę przestrzeni nazw:

```
kubectl get ns
```

| NAME | STATUS | AGE |
|-----------------|--------|-----|
| default | Active | 35d |
| kube-node-lease | Active | 35d |
| kube-public | Active | 35d |
| kube-system | Active | 35d |
| workplace | Active | 35s |

Pojawiła się nowa, stworzona przez nas przestrzeń nazw.

16.1.4. Dodajemy namespace do deploymentu - sposób z użyciem polecenia

Teraz dodajmy do niej deployment, który stworzyliśmy i omawialiśmy przez bardzo długi okres. W ramach przypomnienia kod pliku yaml, o którym mówię to:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app-apache-deployment
  template:
    metadata:
      name: apache-pod-deployment
      labels:
        app: app-apache-deployment
    spec:
      containers:
        - name: apache-deployment-container
          image: httpd
```

Dlatego, jeżeli nie masz go zapisanego, to teraz uruchom swój ulubiony edytor tekstowy, zapisz w nim prezentowaną zawartość. Po wykonaniu powyższych czynności pozostało uruchomić deployment w stworzonej przez nas przestrzeni nazw:

```
kubectl apply -n workplace -f apache-deployment.yaml
```

```
deployment.apps/apache-deployment created
```

Przy pomocy znanego już polecenia oraz opcji **-n** określiliśmy naszemu deploymentowi, w jakiej przestrzeni nazw ma się uruchomić. Dlatego, jeżeli wpiszemy polecenie do wyświetlania podów czy też deploymentów otrzymamy informację:

```
kubectl get deployments
```

```
No resources found in default namespace.
```

Taką samą informację uzyskasz w ReplicaSet. Nie korzystamy z standardowej przestrzeni nazw, dlatego nie pojawia się nic na liście.

16.1.5. Wyświetlanie listy naszej przestrzeni nazw

Aby wyświetlić zawartość, jaką dodaliśmy do naszej przestrzeni *workplace*, dodajemy opcję **-n** w następujący sposób:

kubectl get -n workplace deployments

```
NAME           READY  UP-TO-DATE AVAILABLE AGE
apache-deployment 3/3    3        3   3m46s
```

Tak samo robimy z podami:

kubectl get -n workplace pod

```
NAME           READY STATUS RESTARTS AGE
apache-deployment-76687f8b5f-cqrhx 1/1 Running 0   4m17s
apache-deployment-76687f8b5f-xlt6h 1/1 Running 0   4m17s
apache-deployment-76687f8b5f-z2bx8 1/1 Running 0   4m17s
```

Jak i również z ReplicaSet:

kubectl get -n workplace rs

```
NAME           DESIRED CURRENT READY AGE
apache-deployment-76687f8b5f 3     3     3   4m48s
```

Wszystko zostało uruchomione w wyznaczonej przestrzeni nazw.

16.1.6. Usuwanie przestrzeni nazw

Natomiast jeżeli chcemy pozbyć się stworzonej przestrzeni nazw oraz wszystkiego, co z nią związane używamy polecenia:

kubectl delete ns workplace

```
namespace "workplace" deleted
```

Tak jak wspomniałem, w ten sposób pozbywamy się wszystkiego, co związane z wybraną przestrzenią nazw. I tutaj jedna uwaga. Osobiście uważam taką możliwość za problem. Usunięcie całej przestrzeni nazw jest wygodne w sytuacji, gdy na pewno wiemy, że wszystko w niej jest do usunięcia. Czy zawsze jednak jesteśmy w stanie to określić? W przypadku usuwania elementów tego typu zalecam ostrożność. Czasami lepiej jest dokładnie zerknąć, co znajduje się w niej i usunąć zbędne elementy

ręcznie, niż od razu całą przestrzeń. W tym przykładzie nie jest to żaden problem. To, co robiliśmy, nie jest skomplikowane, a co ważniejsze stworzyliśmy własną przestrzeń oraz jej zawartość kilka minut temu. Dlatego pamiętamy, o co tam chodziło. Jednak czy za miesiąc będziesz pamiętał, *co autor miał na myśli?*

16.1.7. Określamy namespace w pliku

Informację o tym, że dany typ ma się uruchomić w określonej przestrzeni nazw, możesz określić w pliku yaml. Wystarczy dodać tylko etykietę namespace i wprowadzić nazwę przestrzeni nazw. Tworzymy nowy plik **apache-ns-deployment.yaml**. Oto znany już przez nas schemat pliku z dodatkową etykietą namespace:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
  namespace: workplace
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app-apache-deployment
  template:
    metadata:
      name: apache-pod-deployment
      labels:
        app: app-apache-deployment
    spec:
      containers:
        - name: apache-deployment-container
          image: httpd
```

Jednak jeśli usunęliśmy wcześniej stworzoną przestrzeń, tak jak robiłem to przed chwilą to przy próbie uruchomienia otrzymamy informację:

```
Error from server (NotFound): error when creating "apache-ns-deployment.yaml": namespaces "workplace" not found
```

Oznacza to, że jeżeli chcemy dodać do określonej przestrzeni nazw to ona musi istnieć. Dlatego ponownie ją tworzymy:

```
kubectl apply -f namespaces.yaml
```

```
namespace/workplace created
```

Następnie ponownie uruchamiamy nasz deployment z określoną przestrzenią nazw w pliku:

```
kubectl apply -f apache-ns-deployment.yaml
```

```
deployment.apps/apache-deployment created
```

Teraz w ramach zadania dla Ciebie. Sprawdź czy wszystko zostało prawidłowo dodane. Skorzystać z dotychczas poznanych poleceń. Następnie jeżeli chcesz usuń stworzoną namespace.

16.1.8. Wyświetlanie listy z wszystkich dostępnych przestrzeni nazw

Ostatnie polecenie pomocne przy namespaces umożliwia wyświetlenie wszystkiego z danego typu, niezależnie od przynależności do określonej przestrzeni nazw:

```
kubectl get pod --all-namespaces
```

```
kubectl get rs --all-namespaces
```

```
kubectl get deployments --all-namespaces
```

```
...
```

Do każdego typu możemy zastosować opcję **--all-namespaces**, dzięki której wyświetlimy wszystko niezależnie od przestrzeni nazw.

16.2. Zmienne środowiskowe

Tak jak w systemach poza kontenerami, tak i w nich zdarza się, że musimy ustawić zmienną środowiskową. Posiadasz już bardzo dużą wiedzę na temat Kuberntesa, dlatego przeanalizuj sam przykład:

```
apiVersion: v1
kind: Pod
metadata:
  name: env-app-pod
spec:
  containers:
```

```
- name: env-app
  image: alpine
  env:
    - name: APP_NAME
      value: "env-app"
    command: ["env"]
  restartPolicy: Never
```

Zapewne zauważysz nowy element *env*. W tym miejscu chciałbym Ci zwrócić uwagę na pewną schematyczność plików yaml Kuberntesu. O ile oczywiście na to nie zwróciłeś uwagi. Gdy pojawia się jakiś element i chcemy dokonać w nim jakichś czynności, na początku pojawia się jego nazwa, czyli, dla przykładu:

env:

Następnie występuje znacznik listy z nazwą danego ustawienia, kontenera czy też innego elementu. W naszym wypadku jest to nazwa funkcji, jaką chcemy ustawić:

env:

```
- name: APP_NAME
```

Po tym na poziomie name możemy wykonać dostępne ustawienia. W naszym wypadku zmiennej ustawionej w name, czyli APP_NAME przypisujemy wartość:

value: "env-app"

Abyśmy mogli wyświetlić zawartość wszystkich ustawionych zmiennych środowiskowych, wywołujemy polecenie:

command: ["env"]

I na samym końcu ustawiamy *restartPolicy* na *Never*, ponieważ jak już zapewne wiesz, pod tego typu wykonuje swoje czynności i kończy działanie. Jeżeli nie ustawimy mu wartości, Never będzie chciał ponownie się uruchomić, co spowoduje błąd. Pozostaje tylko uruchomić pod:

kubectl apply -f pod-env.yaml

W celu sprawdzenia, czy nasza zmienna środowiskowa została ustawiona, korzystamy z:

kubectl logs env-app-pod

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=env-app-pod
APP_NAME=env-app
MY_POD_PORT=tcp://10.104.192.27:80
```

```
MY_POD_PORT_80_TCP=tcp://10.104.192.27:80
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
MY_POD_PORT_80_TCP_PROTO=tcp
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
MY_POD_PORT_80_TCP_PORT=80
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
MY_POD_SERVICE_HOST=10.104.192.27
MY_POD_SERVICE_PORT=80
MY_POD_PORT_80_TCP_ADDR=10.104.192.27
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
HOME=/root
```

W związku, z tym że użyliśmy polecenia `env`, które wypisuje listę wszystkich zmiennych środowiskowych, otrzymaliśmy ją jako wynik przy pomocy funkcji `kubectl logs`. **Przy pomocy logs wyświetlacza logi danego poda.**

Nie jest to nic skomplikowanego. Natomiast bardzo przydaje się w sytuacji, kiedy takie zmienne środowiskowe musimy ustawić ze względu na wymogi obrazu. Najlepszym przykładem będzie MySQL, o którym sobie zaraz opowiemy.

16.3. Typ Secret

Ostatnim elementem Kuberntesa opisywanym w tym szkoleniu będzie **Secret**. Najlepszym dla tego typu przykładem będzie baza danych MySQL. O ustawieniach, jakie są konieczne, wspominałem w jednym z modułów Dockera. Niemniej jednak było to dość dawno, dlatego zerknijmy na stronę obrazu: https://hub.docker.com/_/mysql. Aby uruchomić obraz, musimy ustawić przynajmniej wartość `MYSQL_ROOT_PASSWORD`. Jednak jak wiadomo, hasła zaliczają się do danych poufnych, dlatego powinny być w jakiś sposób ukryte. I w tym miejscu z pomocą przychodzi typ `Secret`. Na początku stwórzmy sobie plik `yaml`, w którym ustawimy hasło:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret
```

```
data:
```

```
mysql-root-password: "VmVyeVNlY3JldFBhc3N3b3Jk"
```

Jak widzisz, nie jest to nic skomplikowanego. W tym wypadku ustalamy hasło dla użytkownika root. Jednak jak sama konstrukcja nie jest złożona, to musisz wiedzieć, że wprowadzone tutaj hasło musi być zaszyfrowane przy pomocy **base64**.

Aby tego dokonać, możesz wejść na stronę: <https://www.base64encode.org/>. W górnej jej części wybierz Encode i wprowadź swoje tajne hasło. Ja wprowadziłem jako hasło *VerySecretPassword* i otrzymałem *VmVyeVNlY3JldFBhc3N3b3Jk*.

Po odpowiednim ustawieniu wartości musimy uaktywnić sekret. Robimy to w znany nam sposób:

```
kubectl apply -f secret.yaml
```

```
secret/mysql-secret created
```

Secret posiada swoją oddzielną listę, którą możesz wywołać przy pomocy:

```
kubectl get secret
```

| NAME | TYPE | DATA | AGE |
|--------------|--------|------|------|
| mysql-secret | Opaque | 1 | 2m1s |

Teraz w jaki sposób możemy to wykorzystać przy tworzeniu bazy danych:

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
  name: mysql-deployment
```

```
spec:
```

```
  replicas: 1
```

```
  selector:
```

```
    matchLabels:
```

```
      app: mysql
```

```
  template:
```

```
    metadata:
```

```
      labels:
```

```
        app: mysql
```

```
    spec:
```

```
      containers:
```

```
        - name: mysql
```

```

image: mysql
env:
- name: MYSQL_ROOT_PASSWORD
  valueFrom:
    secretKeyRef:
      name: mysql-secret
      key: mysql-root-password
ports:
- containerPort: 3306

```

Początkowa część pliku nie różni się niczym, od tego, co już poznaliśmy. Nazywamy nasz deployment, ustawiamy mu ilość replik, następnie wykonujemy ustawienia naszego obrazu oraz kontenera. Nowość znajduje się w poniższym miejscu:

```

env:
- name: MYSQL_ROOT_PASSWORD
  valueFrom:
    secretKeyRef:
      name: mysql-secret
      key: mysql-root-password
ports:
- containerPort: 3306

```

Ustawiamy zmienną środowiskową, która będzie występowała pod nazwą **MYSQL_ROOT_PASSWORD**, czyli pod identyczną nazwą jak wymaganie, jeżeli chcemy stworzyć kontener z obrazu mysql. Natomiast przy pomocy *secretKeyRef* i określamy nazwę sekretu, który stworzyliśmy w poprzednim przykładzie *mysql-secret* i pobieramy z niego klucz, hasło do konta root. Na końcu udostępniamy port kontenera 3306, ponieważ standardowo MySQL działa na tym porcie. Po omówieniu pliku uruchamiamy go:

kubectl apply -f mysql.yaml

```
deployment.apps/mysql-deployment created
```

Jak sprawdzić czy wszystko poprawnie działa i hasło zostało ustawione? Otóż jeżeli otrzymasz informację, że pod uruchomił się i działa oznacza to, że wszystko wyszło po naszej myśli:

kubectl get po

| NAME | READY | STATUS | RESTARTS | AGE |
|-----------------------------------|-------|---------|----------|-----|
| mysql-deployment-6c46bd8b78-mpd97 | 1/1 | Running | 0 | 18s |

Status, Running sam w sobie świadczy o prawidłowym ustawieniu zmiennej środowiskowej oraz ustawieniu hasła, bo w innym wypadku baza nie mogłaby zostać uruchomiona. Ustawienia hasła do

roota w tym obrazie jest wymogiem. Jednak możesz to jeszcze sprawdzić, przy pomocy describe poda lub deploymentu:

kubectl describe deployments mysql-deployment

Wystarczy odnaleźć informację mówiącą o tym, co zostało wykonane:

Environment:

```
MYSQL_ROOT_PASSWORD: <set to the key 'mysql-root-password' in secret 'mysql-secret'> Optional:  
false
```

Jest to już dowód, na piśmie, że hasło zostało pobrane z secretu oraz ustawione. Tak jak każdy typ Kuberntesu, secret możemy również usunąć:

kubectl delete secret mysql-secret

```
secret "mysql-secret" deleted
```

Jednak w ten sposób usuwasz sam secret. Pozostałe elementy musisz usunąć sam. Wiesz już jak to robić, dlatego posprzątanie pozostawiam już Tobie.

Prezentowany sposób jest przykładem, którego nie powinno stosować się w środowisku produkcyjnym. W takich miejscach zaleca się stosowanie bardziej złożonych technik zabezpieczających.

16.4. Podsumowanie

Kubernetes jest to bardzo potężne narzędzie, dzięki któremu jesteś w stanie działać wiele. Starałem się pokazać jego zalety, byś dzięki nim poznął najważniejsze zagadnienia związane z Kuberntesem. Jednak jak się sam domyślasz, nie są to wszystkie możliwości. Ważnym miejscem, które powinieneś często odwiedzać, jest dokumentacja Kuberntesa znajdująca się pod adresem <https://kubernetes.io/docs/home/>. Kubernetes rozwija się bardzo szybko i niektóre elementy nawet tutaj opisane za jakiś czas mogą zostać inaczej zaimplementowane. Jednak dokumentacja nadąża nad zmianami, niestety książki czy też kursy nie są w stanie. Dlatego, jeżeli coś nie działa, zerknij tam, najczęściej *apiVersion* danego typu lub funkcji zmieniło swoje położenie.

To jeszcze nie koniec.

O Dockerze napisać można bardzo dużo. W e-booku zawarłem podstawowe informacje. Na ten temat często wspominam w swoich kanałach social media. Jeśli chcesz wiedzieć więcej na ten temat i być na bieżąco, to zapraszam serdecznie do śledzenia moich kanałów:

YouTube: <https://www.youtube.com/c/ArkadiuszSiczek/>

Facebook: <https://www.facebook.com/siczkarek/>

Instagram: https://www.instagram.com/arek_siczek/

TikTok: <https://www.tiktok.com/@grupaadm>

Pinterest: <https://pl.pinterest.com/arekask/askomputer/>

Twitter: <https://twitter.com/arkady86>

Blog: <https://blog.askomputer.pl/>

Szkolenia i kursy

Informatyka jest niezwykle szybko rozwijającą się dziedziną wiedzy. Z tego względu każdy informatyk powinien stale brać udział w szkoleniach czy kursach i pogłębiać swoje kompetencje. W trakcie moich szkoleń staram się jak najbardziej skupiać na praktycznym podejściu do materiału, a teorię ograniczam do absolutnego minimum. Zapoznaj się z ofertą aktualnych kursów [TUTAJ](#).

The advertisement features a man in a blue blazer standing next to a computer monitor displaying the ADM website. The website has a dark theme with orange highlights and shows various course offerings. The background is orange, and the ADM logo is at the top left.