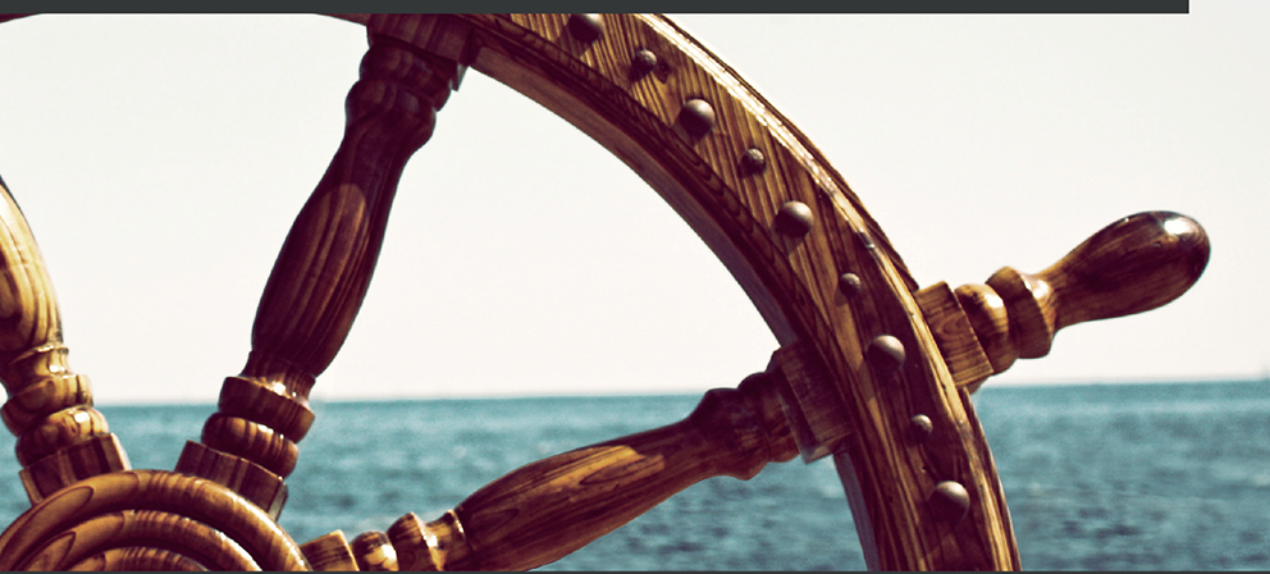
Kubernetes i Docker

w środowisku produkcyjnym przedsiębiorstwa

Konteneryzacja i skalowanie aplikacji
oraz jej integracja z systemami korporacyjnymi



Scott Surovich
Marc Boorshtein

Helion

Tytuł oryginału: Kubernetes and Docker - An Enterprise Guide: Effectively containerize applications, integrate enterprise systems, and scale applications in your enterprise

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-8629-7

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/kubdoc.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/kubdoc
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

O autorach 13

O recenzencie technicznym 14

Wprowadzenie 15

Część I. Wprowadzenie do Dockera i kontenerów 21

Rozdział 1. Podstawy Dockera i kontenerów 23

Wymagania techniczne 24
Zrozumienie potrzeby stosowania kontenerów 24
Pojawienie się Dockera 25
Poznajemy Dockera 26
Kontenery są tymczasowe 26
Obrazy Dockera 27
Warstwy obrazu 28
Trwałe przechowywanie danych 29
Uzyskiwanie dostępu do usług działających w kontenerach 29
Instalacja Dockera 30
Przygotowania do instalacji Dockera 31
Instalowanie Dockera w Ubuntu 31
Nadanie Dockerowi niezbędnych uprawnień 32
Używanie Dockera w powłoce 34
docker help 34
docker run 34
docker ps 35
docker start i docker stop 36

docker attach 37
docker exec 38
docker logs 39
docker rm 40
Podsumowanie 41
Pytania 41

Rozdział 2. Praca z danymi Dockera 43

Wymagania techniczne 43
Dlaczego w ogóle potrzebujesz mechanizmu trwałego przechowywania danych? 44
Woluminy Dockera 45
Tworzenie woluminu z poziomu powłoki 46
Montowanie woluminu w kontenerze 48
Montowanie istniejącego woluminu 49
Montowanie woluminu w wielu kontenerach 50
Wyświetlanie woluminów Dockera 51
Usuwanie woluminów 51
Dołączane punkty montowania w Dockerze 53
Tymczasowy system plików w Dockerze 55
Używanie systemu tmpfs w kontenerze 56
Podsumowanie 58
Pytania 58

Rozdział 3. Sieć w Dockerze 60

Wymagania techniczne 60
Obsługa sieci w Dockerze 61
Krótkie wprowadzenie do używania portów przez TCP/IP 61
Dołączanie portu do usługi 63
Sterowniki sieciowe Dockera 63
Domyślna sieć mostu 65
Wyświetlanie dostępnych sieci 66
Pobieranie informacji szczegółowych dotyczących sieci 67
Samodzielne tworzenie sieci typu most 68
Połączenie kontenera z siecią zdefiniowaną przez użytkownika 69
Zmiana sieci w uruchomionym kontenerze 69
Usunięcie sieci 70
Uruchomienie kontenera bez obsługi sieci 71
Udostępnianie usług kontenera 72
Udostępnianie portów za pomocą sieci hosta 72
Udostępnianie portów za pomocą sieci typu most 74
Podsumowanie 77
Pytania 77

Część II. Tworzenie klastra programistycznego Kubernetes, poznawanie obiektów i udostępnianie usług 79

Rozdział 4. Wdrażanie Kubernetes za pomocą KinD 81

Wymagania techniczne **82**
Wprowadzenie do obiektów i komponentów Kubernetes **83**
Praca z klastrem 84
Używanie klastrów programistycznych **84**
Dlaczego zdecydowaliśmy się na KinD? 85
Praca z podstawowym klastrem KinD Kubernetes 86
Poznajemy obraz węzła 89
KinD i sieć Dockera 90
Instalacja KinD **93**
Przygotowanie do instalacji KinD 93
Instalacja pliku binarnego KinD 94
Tworzenie klastra KinD **95**
Tworzenie prostego klastra 95
Usunięcie klastra 96
Tworzenie pliku konfiguracyjnego klastra 96
Konfiguracja klastra składającego się z wielu węzłów 97
Dostosowanie do własnych potrzeb warstwy sterowania i opcji kubelet 100
Tworzenie własnego klastra KinD 101
Instalacja Calico 102
Instalacja kontrolera Ingress 103
Analiza utworzonego klastra KinD **104**
Obiekty pamięci masowej KinD 105
Sterowniki pamięci masowej 106
Klasy pamięci masowej w KinD 106
Używanie komponentu KinD przygotowującego pamięć masową 107
Dodawanie niestandardowego mechanizmu równoważenia obciążenia dla kontrolera Ingress **109**
Przygotowanie do instalacji 110
Tworzenie konfiguracji klastra KinD 110
Wdrażanie niestandardowego kontenera HAProxy 111
Przepływ ruchu sieciowego HAProxy 113
Symulowanie awarii kubeletu 115
Usunięcie kontenera HAProxy 116
Podsumowanie **117**
Pytania **117**

Rozdział 5. Krótkie wprowadzenie do Kubernetes 119

Wymagania techniczne **120**
Ogólne omówienie komponentów Kubernetes **120**
Poznajemy warstwę sterowania **121**
Serwer API w Kubernetes 121
Baza danych Etcd 122
kube-scheduler 124

kube-controller-manager 124
cloud-controller-manager 125
Poznajemy sposób działania komponentów węzła roboczego 125
kubelet 125
kube-proxy 125
Środowisko uruchomieniowe kontenera 125
Współpraca z serwerem API 126
Praca z narzędziem Kubernetes kubectl 126
Poznajemy opcję verbose 127
Ogólne polecenia kubectl 128
Poznajemy obiekty Kubernetes 129
Manifest w Kubernetes 130
Czym jest obiekt Kubernetes? 130
Przegląd obiektów Kubernetes 132
Podsumowanie 148
Pytania 149

Rozdział 6. Usługi, mechanizm równoważenia obciążenia i zewnętrzny serwer DNS 151

Wymagania techniczne 152
Zapewnienie żądaniom dostępu do zadań 152
Jak działa usługa? 152
Poznajemy poszczególne typy usług 157
Wprowadzenie do mechanizmu równoważenia obciążenia 163
Poznajemy model OSI 163
Mechanizmy równoważenia obciążenia działające na warstwie siódmej 164
Określanie nazw i mechanizmy równoważenia obciążenia działające na warstwie siódmej 165
Używanie usługi nip.io do określania nazw 166
Tworzenie reguł Ingress 168
Mechanizmy równoważenia obciążenia działające na warstwie czwartej 172
Opcje w zakresie mechanizmu równoważenia obciążenia na warstwie czwartej 172
Używanie MetalLB jako działającego na warstwie czwartej mechanizmu równoważenia obciążenia 172
Tworzenie usługi typu LoadBalancer 176
Dodawanie wielu pul adresów IP do MetalLB 178
Problemy związane z używaniem wielu protokołów 180
Używanie wielu protokołów z MetalLB 181
Używanie współdzielonych adresów IP 182
Udostępnianie nazw usług na zewnątrz 184
Konfiguracja projektu ExternalDNS 185
Integracja projektu ExternalDNS z CoreDNS 185
Dodawanie strefy Etcd do CoreDNS 186
Tworzenie usługi typu LoadBalancer zintegrowanej z ExternalDNS 189
Podsumowanie 195
Pytania 195

Część III. Kubernetes w korporacjach 197

Rozdział 7. Integracja z klastrem mechanizmu uwierzytelniania 199

Wymagania techniczne **200**
Jak Kubernetes rozpoznaje użytkownika? **200**
Użytkownik zewnętrzny 200
Grupy w Kubernetes 201
Konta usług 201
Poznajemy protokół OpenID Connect **202**
Protokół OpenID Connect 203
Współdziałanie OIDC i API 205
Inne opcje w zakresie uwierzytelniania 209
Konfiguracja klastra KinD dla OpenID Connect **213**
Spełnienie wymagań 214
Wdrażanie OIDC 217
Wprowadzenie do funkcjonalności „wcielania się w rolę" w celu integracji systemu uwierzytelniania z klastrami zarządzanymi w chmurze **230**
Czym jest funkcjonalność wcielania się w rolę? 231
Kwestie związane z zapewnieniem bezpieczeństwa 232
Konfiguracja klastra do użycia funkcjonalności wcielania się w rolę **233**
Testowanie rozwiązania 235
Konfiguracja funkcjonalności wcielania się w rolę bez użycia OpenUnison **236**
Polityki modelu RBAC dotyczące funkcjonalności wcielania się w rolę 236
Grupy domyślne 237
Podsumowanie **237**
Pytania **237**

Rozdział 8. Polityki modelu RBAC i audyt 239

Wymagania techniczne **240**
Wprowadzenie do modelu RBAC **240**
Czym jest rola? **241**
Identyfikowanie roli 242
Role kontra obiekt typu ClusterRole 243
Odwrotność roli 244
Agregowane obiekty typu ClusterRole 245
Obiekty typu RoleBinding i ClusterRoleBinding 246
Mapowanie tożsamości użytkowników organizacji na polityki Kubernetes w celu autoryzacji dostępu do zasobów **248**
Implementacja wielodostępności za pomocą przestrzeni nazw **250**
Audyt w Kubernetes **252**
Zdefiniowanie polityki audytu 252
Włączanie audytu w klastrze 254
Używanie audit2rbac do debugowania polityk **256**
Podsumowanie **261**
Pytania **261**

Rozdział 9. Wdrażanie bezpiecznego panelu Kubernetes 263

Wymagania techniczne 264
Jak panel rozpoznaje użytkownika? 264
Architektura panelu 264
Metody uwierzytelniania 265
Niebezpieczeństwa związane z panelem Kubernetes 266
Wdrażanie niewystarczająco zabezpieczonego panelu 267
Używanie tokenu do logowania 273
Wdrożenie panelu z użyciem odwrotnego proxy 273
Panel lokalny 274
Inne aplikacje na poziomie klastra 275
Integracja panelu z OpenUnison 276
Podsumowanie 278
Pytania 278

Rozdział 10. Definiowanie polityki bezpieczeństwa poda 280

Wymagania techniczne 281
Czym jest PSP? 281
Różnice między kontenerem i maszyną wirtualną 281
Atak typu container breakout 282
Prawidłowe projektowanie kontenera 284
Czy coś się zmienia? 291
Włączenie PSP 292
Alternatywy dla PSP 297
Podsumowanie 298
Pytania 298

Rozdział 11. Poprawianie bezpieczeństwa za pomocą Open Policy Agent 300

Wymagania techniczne 301
Wprowadzenie do dynamicznych kontrolerów sterowania dopuszczeniem 301
Co to jest program typu OPA i na czym polega jego działanie? 303
Architektura OPA 303
Rego, czyli język polityki w OPA 304
GateKeeper 305
Zautomatyzowany framework testowania 306
Używanie Rego do definiowania polityki 306
Opracowanie polityki OPA 307
Testowanie polityki OPA 308
Wdrażanie polityki do GateKeeper 310
Tworzenie polityki dynamicznej 312
Debugowanie kodu w języku Rego 316
Używanie istniejącej polityki 317
Wymuszanie ograniczeń dotyczących pamięci 317
Włączanie bufora GateKeeper 318
Imitacja danych testowych 320
Budowanie i wdrażanie polityki 320

Wymuszanie PSP za pomocą OPA **322**
Podsumowanie **323**
Pytania **323**

Rozdział 12. Audyt za pomocą Falco i EFK **325**

Wymagania techniczne **326**
Poznajemy audyt **326**
Wprowadzenie do Falco **328**
Poznajemy pliki konfiguracyjne Falco **328**
Plik konfiguracyjny falco.yaml 329
Pliki konfiguracyjne reguł Falco 333
Definiowanie i dołączanie reguł niestandardowych 339
Wdrożenie Falco **340**
Moduł jądra Falco **341**
Tworzenie modułu jądra na podstawie zainstalowanych nagłówków jądra 342
Używanie nagłówków jądra do utworzenia modułu Falco 343
Tworzenie modułu jądra za pomocą narzędzia driverkit 344
Używanie modułu w klastrze 347
Używanie modułu w KinD 347
Wdrożenie z użyciem manifestu DaemonSet 348
Wdrażanie stosu EFK 351
Podsumowanie **369**
Pytania **369**

Rozdział 13. Tworzenie kopii zapasowej **371**

Wymagania techniczne **372**
Kopie zapasowe w Kubernetes **372**
Tworzenie kopii zapasowej Etcd **373**
Tworzenie kopii zapasowej wymaganych certyfikatów 373
Tworzenie kopii zapasowej bazy danych Etcd 373
Poznajemy narzędzie Velero Heptio i jego konfigurację **375**
Wymagania Velero 375
Instalacja działającego w powłoce narzędzia Velero 376
Instalacja Velero 376
Używanie Velero do tworzenia kopii zapasowej **382**
Jednorazowe utworzenie kopii zapasowej klastra 382
Harmonogram tworzenia kopii zapasowej klastra 386
Tworzenie niestandardowej kopii zapasowej 387
Zarządzanie Velero za pomocą narzędzia działającego w powłoce **388**
Najczęściej używane polecenia Velero 390
Przywracanie z kopii zapasowej **392**
Przywracanie w akcji 392
Przywrócenie przestrzeni nazw 395
Używanie kopii zapasowej do przywrócenia danych w nowym klastrze 396
Przywrócenie kopii zapasowej w nowym klastrze 398
Podsumowanie **400**
Pytania **401**

Rozdział 14. Przygotowywanie platformy **402**

Wymagania techniczne **403**
Opracowanie potoku **403**
Najważniejsze istniejące platformy 405
Zabezpieczanie potoku 406
Określenie wymagań platformy 406
Wybór stosu technologii 409
Przygotowanie klastra **410**
Wdrażanie cert-manager 411
Wdrażanie rejestru kontenerów Dockera 413
Wdrażanie OpenUnison 414
Wdrażanie GitLab **417**
Tworzenie przykładowych projektów 420
Wdrażanie Tekton **421**
Tworzenie aplikacji typu Witaj, świecie! 422
Kompilacja automatyczna 427
Wdrażanie ArgoCD **428**
Automatyzacja tworzenia projektu z użyciem OpenUnison **431**
Integracja z GitLab 434
Integracja z ArgoCD 435
Uaktualnienie OpenUnison 436
Podsumowanie **437**
Pytania **438**

Odpowiedzi na pytania **439**

ROZDZIAŁ 4

Wdrażanie Kubernetes za pomocą KinD

Jedną z największych przeszkód podczas poznawania Kubernetes są niewystarczające zasoby wymagane w celu utworzenia klastra przeznaczonego do testów lub programowania. Podobnie jak większość profesjonalistów, także my lubimy mieć w naszych laptopach klastry Kubernetes przeznaczone do celów demonstracyjnych lub ogólnego testowania produktów.

Bardzo często może zachodzić potrzeba uruchomienia wielu klastrów niezbędnych w skomplikowanym przykładzie, np. składającej się z wielu klastrów usługi typu *mesh* lub podczas testowania `kubefed2`. W takich wypadkach wymagane będzie użycie wielu serwerów pozwalających na utworzenie niezbędnych klastrów, co z kolei przekłada się na zapotrzebowanie na dużą ilość pamięci RAM i użycie hipernadzorcy.

Aby móc w pełni przetestować produkt z użyciem wielu klastrów, dla każdego z nich trzeba utworzyć po sześć węzłów. Jeżeli tworzysz klastry z wykorzystaniem maszyn wirtualnych, musisz mieć zasoby pozwalające na uruchomienie sześciu takich maszyn. Każda z nich ma pewne wymagania dotyczące ilości potrzebnej przestrzeni dyskowej, pamięci RAM i zasobów procesora.

Jak to wygląda w sytuacji, w której klaster jest tworzony na podstawie kontenerów? Użycie kontenerów zamiast maszyn wirtualnych daje możliwość uruchomienia dodatkowych węzłów ze względu na mniejsze wymagania sprzętowe kontenerów. Tworzenie i usuwanie klastrów może się odbywać bardzo szybko za pomocą pojedynczego polecenia lub skryptu. Ponadto w pojedynczym hoście można uruchamiać wiele klastrów.

Użycie kontenerów do uruchomienia klastra Kubernetes oferuje środowisko, którego przygotowanie za pomocą maszyn wirtualnych lub fizycznych komputerów byłoby trudne dla większości osób ze względu na ograniczone dostępne zasoby. Aby wyjaśnić, jak można lokalnie uruchomić klaster na podstawie jedynie kontenerów, popularnego narzędzia KinD użyjemy

do utworzenia klastra Kubernetes w hoście Dockera. Zajmiemy się wdrożeniem składającego się z wielu węzłów klastra, który w kolejnych rozdziałach będzie używany do testowania i wdrażania komponentów takich jak kontrolery Ingress, mechanizm uwierzytelniania, model RBAC, polityki bezpieczeństwa itd.

Oto tematy, które zostały omówione w rozdziale:

- wprowadzenie do obiektów i komponentów Kubernetes,
- używanie klastrów programistycznych,
- instalowanie KinD,
- tworzenie klastra KinD,
- analiza klastra KinD,
- dodawanie własnego mechanizmu równoważenia obciążenia.

Zaczynamy!

Wymagania techniczne

Materiał zamieszczony w rozdziale wiąże się z pewnymi wymaganiami technicznymi.

- Host Dockera przygotowany na podstawie informacji zamieszczonych w rozdziale 1.
- Skrypty instalacyjne zamieszczone w materiałach przygotowanych dla książki.

Przykładowe fragmenty kodu omówione w tym rozdziale znajdziesz w materiałach dostępnych pod adresem *https://ftp.helion.pl/przyklady/kubdoc.zip*.

> Trzeba koniecznie wspomnieć, że w rozdziale odwołujemy się do wielu obiektów Kubernetes, czasem bez przedstawiania niezbędnego kontekstu. W rozdziale 5. zamieścimy wiele informacji szczegółowych dotyczących obiektów Kubernetes, a także zaprezentujemy liczne polecenia pomagające w poznaniu tych obiektów. Dlatego też warto mieć pod ręką gotowy klaster podczas lektury następnego rozdziału.
>
> Większość poruszonych w tym rozdziale podstawowych tematów dotyczących Kubernetes będzie dokładniej omawianych także w kolejnych rozdziałach. Dlatego też, jeśli nie wszystko będzie jasne, nie przejmuj się tym. Do wielu zagadnień jeszcze wrócimy w dalszej części książki.

Wprowadzenie do obiektów i komponentów Kubernetes

Skoro w rozdziale koncentrujemy się na najczęściej używanych komponentach i obiektach Kubernetes, uznaliśmy za stosowne zamieszczenie tabeli (zobacz tabela 4.1), w której zebraliśmy używane w tekście pojęcia wraz z ich krótkimi wyjaśnieniami, co ma na celu dostarczenie kontekstu.

Tabela 4.1. Komponenty i obiekty Kubernetes

Komponent	Opis
Warstwa sterowania	`kube-apiserver`: akceptuje żądania pochodzące od klientów frontendu warstwy sterowania
	`kube-scheduler`: ten komponent przypisuje zadania węzłom
	`etcd`: to baza danych zawierająca wszystkie dane klastra
	`kube-controller-manager`: ten komponent monitoruje stan węzła, replik poda, punktów końcowych, kont usług i tokenów
Węzeł	`kubelet`: ten agent uruchamia poda na podstawie informacji otrzymanych z warstwy sterowania
	`kube-proxy`: ten komponent pozwala na tworzenie i usuwanie reguł sieciowych przeznaczonych do obsługi komunikacji poda
	Środowisko uruchomieniowe kontenera: to jest komponent odpowiedzialny za działanie kontenera
Obiekt	**Opis**
Kontener	Pojedynczy i niemodyfikowalny obraz, który zawiera wszystko to, co jest wymagane do uruchomienia aplikacji
Pod	Najmniejszy obiekt, który może być kontrolowany przez Kubernetes. Pod zawiera jeden lub więcej kontenerów. Wszystkie kontenery w podzie są umieszczone na tym samym serwerze w kontekście współdzielonym (tzn. każdy kontener w podzie może uzyskać dostęp do innych podów za pomocą adresu 127.0.0.1)
Wdrożenie (ang. *deployment*)	Obiekt używany do wdrażania aplikacji w klastrze na podstawie żądanego stanu, czyli m.in. liczby podów i konfiguracji ciągłych uaktualnień
Klasa pamięci masowej	Ten obiekt definiuje dostawców pamięci masowej i dostarcza ich klastrowi
PV	Ten obiekt dostarcza docelową pamięć masową, która może być żądana przez PVR (ang. *persistent volume request*)
PVC	Ten obiekt nawiązuje połączenie z PV, aby pamięć masowa mogła być używana w podzie
CNI	Ten obiekt zapewnia połączenie sieciowe podom. Do najczęściej stosowanych obiektów typu CNI zaliczamy Flannel i Calico
CSI	Ten obiekt zapewnia połączenie między podami i systemami pamięci masowej

W rozdziale 5. dokładniej omówimy komponenty Kubernetes i bazowy zestaw obiektów znajdujących się w klastrze. Ponadto dowiesz się, jak pracować z klastrem Kubernetes za pomocą działającego w powłoce narzędzia `kubectl`.

Wprawdzie w tabeli 4.1 wymieniliśmy jedynie kilka obiektów dostępnych w klastrze Kubernetes, ale to podstawowe obiekty, o których będziemy wspominać w tym rozdziale. Poznanie poszczególnych obiektów i sposobu ich działania pomoże w przygotowaniu i wdrożeniu klastra KinD.

Praca z klastrem

Aby przetestować utworzony w rozdziale klaster KinD, skorzystamy z działającego w powłoce narzędzia o nazwie `kubectl`. Jego dokładne omówienie znajdzie się w rozdziale 5., w tym przedstawimy jedynie kilka poleceń. Te polecenia wraz z krótkimi wyjaśnieniami i wybranymi opcjami wymieniliśmy w tabeli 4.2.

Tabela 4.2. Podstawowe polecenia narzędzia kubectl

Polecenie kubectl	Opis
`kubectl get <obiekt>`	Pobiera listę żądanego obiektu Przykład: `kubectl get nodes`
`kubectl create -f <nazwa-manifestu>`	Tworzy obiekt na podstawie podanego manifestu. Polecenie `create` pozwala jedynie na tworzenie obiektów początkowych, a nie na uaktualnianie obiektów
`kubectl apply -f <nazwa-manifestu>`	Wdraża obiekty w podanym manifeście. W przeciwieństwie do `create` opcja `apply` może nie tylko tworzyć obiekty, ale również je uaktualniać
`kubectl patch <typ-obiektu> <nazwa-obiektu> -p {opcje}`	Stosuje podane opcje we wskazanym obiekcie

Tych podstawowych poleceń użyjemy dalej, podczas wdrażania klastra, który następnie będzie używany w pozostałych rozdziałach książki.

W kolejnym podrozdziale wprowadzimy koncepcję klastrów programistycznych, a następnie skoncentrujemy się na najpopularniejszym narzędziu używanym do tworzenia takich klastrów: KinD.

Używanie klastrów programistycznych

Na przestrzeni lat opracowano wiele różnych narzędzi przeznaczonych do tworzenia programistycznych klastrów Kubernetes. Narzędzia te pozwalają administratorom i programistom na testowanie rozwiązań w systemach lokalnych. Wprawdzie wiele z tych narzędzi sprawdzało

się w podstawowych testach Kubernetes, ale często nakładały pewne ograniczenia, z powodu których niezbyt sprawdzały się w przypadku zaawansowanych rozwiązań.

Oto wybrane z najczęściej stosowanych rozwiązań:

- Docker Desktop,
- minikube,
- kubeadm.

Każde z tych rozwiązań ma zalety, ograniczenia i własne zastosowania. Niektóre z nich ograniczają Cię do pojedynczego węzła, w którym działają zarówno warstwa sterowania, jak i węzły robocze. Z kolei inne rozwiązania oferują obsługę wielu węzłów, ale wymagają dodatkowych zasobów niezbędnych do utworzenia wielu maszyn wirtualnych. W zależności od wymagań projektu programistycznego lub testowego te rozwiązania niekoniecznie będą Ci odpowiadały.

Wydaje się, że w regularnych odstępach czasu (co kilka tygodni) pojawia się nowe rozwiązanie. Jednym z najnowszych i przeznaczonych do tworzenia klastrów programistycznych jest projekt określany mianem **Kubernetes in Docker** (KinD).

KinD pozwala przy użyciu pojedynczego hosta utworzyć wiele klastrów, z których każdy może zawierać wiele warstw sterowania i węzłów roboczych. Możliwość uruchomienia wielu węzłów pozwala na prowadzenie zaawansowanego testowania, które w przypadku innego rozwiązania wymagałoby większych zasobów. Projekt KinD jest bardzo dobrze oceniany przez społeczność i aktywnie przez nią rozwijany (*https://github.com/kubernetes-sigs/kind*). Dla tego projektu istnieje również kanał Slack (*#kind*).

> Nie należy używać KinD w klastrze produkcyjnym ani też udostępniać klastra KinD w internecie. Wprawdzie klaster KinD oferuje większość funkcjonalności, których się oczekuje w klastrze produkcyjnym, ale mimo to **nie** został on zaprojektowany do stosowania w środowisku produkcyjnym.

Dlaczego zdecydowaliśmy się na KinD?

Kiedy rozpoczynaliśmy pracę nad książką, chcieliśmy w niej umieścić nie tylko teorię, ale również przykłady praktyczne. KinD pozwala nam na dostarczenie skryptów przeznaczonych do uruchamiania i zatrzymywania klastrów. Wprawdzie inne rozwiązania oferują podobne możliwości, ale KinD pozwala na bardzo szybkie tworzenie klastrów składających się z wielu węzłów. Chcieliśmy przygotować oddzielną warstwę sterowania i węzły robocze, aby w ten sposób powstał bardziej „realistyczny" klaster. W celu ograniczenia wymagań sprzętowych i ułatwienia konfiguracji Ingress na potrzeby przykładów zamieszczonych w książce utworzymy klaster składający się jedynie z dwóch węzłów.

Utworzenie składającego się z wielu węzłów klastra może zająć zaledwie kilka minut, po zakończeniu testowania zaś można go usunąć w ciągu dosłownie kilku sekund. Możliwość uruchamiania i zamykania klastrów powoduje, że KinD stanowi doskonałą platformę dla przykładów

zamieszczonych w książce. Wymagania KinD są bardzo skromne: działający demon Docker w celu utworzenia klastra. To oznacza zgodność z większością systemów operacyjnych, m.in.:

- Linux,
- macOS z uruchomioną aplikacją Docker Desktop,
- Windows z uruchomioną aplikacją Docker Desktop,
- Windows z uruchomionym podsystemem WSL2.

> W chwili powstawania książki KinD nie oferuje obsługi Chrome OS.

Wprawdzie KinD oferuje obsługę większości systemów operacyjnych, ale my zdecydowaliśmy się na użycie Ubuntu 18.04 jako systemu operacyjnego w komputerze gospodarza. Część przykładów zamieszczonych w książce będzie wymagała istnienia pewnych plików w określonych katalogach systemu, a wybór konkretnej dystrybucji systemu Linux pomógł nam zagwarantować działanie tych przykładów zgodnie z oczekiwaniami. Jeżeli nie masz dostępu do serwera Ubuntu, możesz utworzyć maszynę wirtualną w chmurze, np. korzystając z platformy GCP (ang. *Google Cloud Platform*). Google oferuje kredyt o równowartości 300 USD, co w zupełności wystarcza do działania pojedynczego serwera Google przez kilka tygodni. Więcej informacji na temat bezpłatnych opcji w ofercie GCP znajdziesz na stronie *https://cloud.google.com/free/*.

W następnej sekcji wyjaśnimy sposób działania KinD i pokażemy podstawowy klaster KinD Kubernetes.

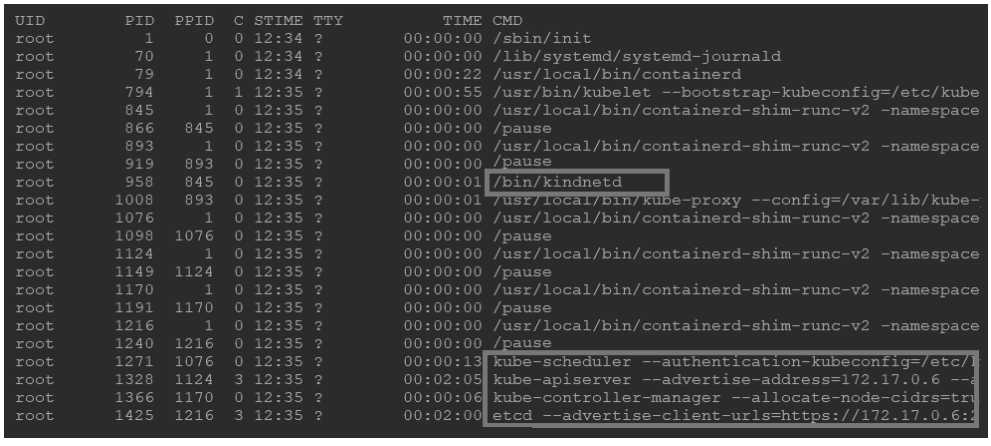
Praca z podstawowym klastrem KinD Kubernetes

Na wysokim poziomie klaster KinD można traktować jako składający się z **pojedynczego** kontenera Dockera, zawierającego warstwę sterowania i węzeł roboczy, które tworzą klaster Kubernetes. Aby wdrożenie było łatwe i niezawodne, KinD umieszcza każdy obiekt Kubernetes w oddzielnym obrazie nazywanym obrazem węzła. Taki obraz węzła zawiera wszystkie komponenty Kubernetes wymagane do utworzenia klastra w postaci pojedynczego węzła lub składającego się z wielu węzłów.

Po uruchomieniu klastra można za pomocą Dockera przejść do kontenera węzła warstwy sterowania i wyświetlić listę procesów (rysunek 4.1). Na tej liście zobaczysz standardowe komponenty Kubernetes znajdujące się w działającym węźle warstwy sterowania.

Jeżeli przejdziesz do węzła roboczego i sprawdzisz działające w nim komponenty, otrzymasz listę zawierającą wszystkie standardowe komponenty węzła roboczego, jak pokazaliśmy na rysunku 4.2.

W rozdziale 5. znajdziesz dokładne omówienie standardowych komponentów Kubernetes, m.in. `kube-apiserver`, `kubelets`, `kube-proxy`, `kube-scheduler` i `kube-controller-manager`.



Rysunek 4.1. Lista procesów zawierająca m.in. komponenty warstwy sterowania Kubernetes

Rysunek 4.2. Lista procesów zawierająca m.in. komponenty węzła roboczego

Poza standardowymi komponentami Kubernetes oba węzły KinD zawierają dodatkowy komponent, który nie zalicza się do większości instalacji standardowych: Kindnet. Po zainstalowaniu bazowego klastra KinD Kindnet to domyślny komponent typu CNI. Mimo to masz możliwość jego wyłączenia i użycia komponentu alternatywnego, takiego jak Calico.

Skoro przedstawiliśmy poszczególne węzły i komponenty Kubernetes, możemy przejść do prezentacji zawartości bazowego klastra KinD. W celu pokazania pełnego klastra i wszystkich uruchomionych komponentów użyliśmy polecenia `kubectl get podst --all-namespaces`. Wyświetla ono wszystkie uruchomione komponenty klastra, w tym komponenty bazowe, które zostaną dokładnie omówione w rozdziale 5. Poza komponentami klastra bazowego w przestrzeni nazw local-path-storage można zauważyć uruchomionego poda o nazwie `local-path-provisioner` (zobacz rysunek 4.3). Ten pod zapewnia działanie jednego z dodatków KinD i pozwala klastrowi na automatyczne przygotowywanie `PersistentVolumeClaims`.

Większość klastrów programistycznych oferuje podobną funkcjonalność, wymaganą do testowania wdrożeń w Kubernetes. Dostarczane są więc warstwa sterowania Kubernetes i węzły robocze, a dodatkowo większość klastrów ma domyślny komponent typu CNI przeznaczony do obsługi sieci. W przypadku kilku oferowanych rozwiązań otrzymujemy nieco więcej poza tą funkcjonalnością bazową. Wraz z upływem czasu można się spodziewać, że zajdzie potrzeba

```
NAMESPACE            NAME                                                 READY   STATUS    RESTARTS   AGE
kube-system          coredns-6955765f44-425sl                             1/1     Running   0          68m
kube-system          coredns-6955765f44-sclfh                             1/1     Running   0          68m
kube-system          etcd-config2-control-plane                           1/1     Running   0          68m
kube-system          etcd-config2-control-plane2                          1/1     Running   0          68m
kube-system          etcd-config2-control-plane3                          1/1     Running   0          67m
kube-system          kindnet-4lzz5                                        1/1     Running   0          68m
kube-system          kindnet-blcwt                                        1/1     Running   0          68m
kube-system          kindnet-bp6jj                                        1/1     Running   0          66m
kube-system          kindnet-p2jw8                                        1/1     Running   0          67m
kube-system          kindnet-pl2gg                                        1/1     Running   0          66m
kube-system          kindnet-vfcpm                                        1/1     Running   0          66m
kube-system          kube-apiserver-config2-control-plane                 1/1     Running   0          68m
kube-system          kube-apiserver-config2-control-plane2                1/1     Running   0          68m
kube-system          kube-apiserver-config2-control-plane3                1/1     Running   1          67m
kube-system          kube-controller-manager-config2-control-plane        1/1     Running   1          68m
kube-system          kube-controller-manager-config2-control-plane2       1/1     Running   0          68m
kube-system          kube-controller-manager-config2-control-plane3       1/1     Running   0          66m
kube-system          kube-proxy-c77z6                                     1/1     Running   0          66m
kube-system          kube-proxy-fvvbb                                     1/1     Running   0          68m
kube-system          kube-proxy-llfpl                                     1/1     Running   0          66m
kube-system          kube-proxy-lpfnw                                     1/1     Running   0          68m
kube-system          kube-proxy-pfk46                                     1/1     Running   0          67m
kube-system          kube-proxy-rd26p                                     1/1     Running   0          66m
kube-system          kube-scheduler-config2-control-plane                 1/1     Running   1          68m
kube-system          kube-scheduler-config2-control-plane2                1/1     Running   0          68m
kube-system          kube-scheduler-config2-control-plane3                1/1     Running   0          66m
local-path-storage   local-path-provisioner-7745554f7f-g5mj8              1/1     Running   1          68m
```

Rysunek 4.3. Dane wyjściowe polecenia kubectl get pods pokazują istnienie poda local-path-provisioner

użycia wtyczek dodatkowych, takich jak `local-path-provisioner`. Wymieniony komponent jest dość intensywnie używany w niektórych przykładach zamieszczonych w książce, ponieważ bez niego utworzenie pewnych procedur byłoby utrudnione.

Dlaczego w klastrze należy zwracać uwagę na woluminy trwałego magazynu danych? Większość produkcyjnych klastrów Kubernetes będzie zapewniała programistom dostęp do trwałego magazynu danych. Zwykle takie magazyny opierają się na systemach działających na podstawie bloków, np. S3 lub NFS. Większość domowych laboratoriów rzadko ma do dyspozycji zasoby pozwalające na istnienie w pełni wyposażonego systemu pamięci masowej. W takich wypadkach wtyczka `local-path-provisioner` okazuje się dużym ułatwieniem dla użytkowników, ponieważ dostarcza klastrowi KinD wszystkie funkcje znajdujące się w drogich rozwiązaniach pamięci masowej.

W rozdziale 5. omówimy kilka obiektów API — `CSIdrivers`, `CSInodes` i `StorageClass` — będących częścią systemu pamięci masowej w Kubernetes. Wymienione obiekty są używane przez klaster w celu zapewnienia dostępu do istniejącego w backendzie systemu pamięci masowej. Po zainstalowaniu i skonfigurowaniu pody mogą korzystać z pamięci masowej za pomocą obiektów `PersistentVolumes` i `PersistentVolumeClaims`. Poznanie obiektów pamięci masowej ma istotne znaczenie, choć przy pierwszym zetknięciu praca z nimi jest dla większości osób trudna, ponieważ te obiekty nie znajdują się w większości oferowanych rozwiązań bazujących na Kubernetes.

Twórcy KinD dostrzegli to ograniczenie i zdecydowali się na dołączenie do oferowanego przez siebie rozwiązania opracowanego przez firmę Rancher projektu o nazwie `local-path-provi`
`↪sioner`, który bazuje na wprowadzonych w Kubernetes 1.10 lokalnych trwałych woluminach.

Być może się zastanawiasz, dlaczego ktokolwiek miałby potrzebę używania wtyczki, skoro Kubernetes zapewnia natywną obsługę trwałych woluminów w hoście lokalnym. Wprawdzie

wspomniana obsługa trwałej pamięci masowej została dodana, ale mimo to Kubernetes nie oferuje możliwości automatycznego przygotowania pamięci masowej. Taką możliwość daje komponent CNCF, ale najpierw musi być on zainstalowany i skonfigurowany jako oddzielny komponent Kubernetes. KinD bardzo ułatwia automatyczne przygotowanie pamięci masowej, ponieważ odpowiedni komponent znajduje się we wszystkich instalacjach bazowych.

Opracowany przez firmę Rancher projekt zapewnia KinD takie możliwości:

- automatyczne tworzenie `PersistentVolume` podczas obsługi żądania PVC,
- domyślna klasa `StorageClass`.

Gdy komponent automatycznie przygotowujący pamięć masową napotyka w serwerze API żądanie `PersistentVolumeClaim`, wówczas następuje utworzenie egzemplarza `PersistentVolume` i dołączenie do niego PVC poda.

Wtyczka `local-path-provisioner` dodaje do KinD funkcjonalność znacznie rozbudowującą potencjalne scenariusze testowe, które można wykonać. Bez możliwości automatycznego przygotowania dysków trwałego magazynu danych przetestowanie wielu wymagających go wbudowanych wdrożeń stanowiłoby duże wyzwanie.

Dzięki użyciu wymienionej wtyczki KinD oferuje rozwiązanie pozwalające na przeprowadzanie eksperymentów z woluminami dynamicznymi, klasami pamięci masowej i innymi testami pamięci masowej, których wykonanie poza centrum danych byłoby w innym wypadku niemożliwe. Wtyczka `local-path-provisioner` będzie używana w wielu rozdziałach w celu dostarczenia woluminów różnych wdrożeniom. Będziemy wyraźnie wskazywać na jej wykorzystanie, aby w ten sposób podkreślić korzyści płynące z automatycznego przygotowywania pamięci masowej.

Poznajemy obraz węzła

Obraz węzła pozwala KinD na uruchamianie Kubernetes w kontenerze Dockera. Należy to uznać za imponujące osiągnięcie, ponieważ działanie Dockera opiera się na `systemd` i innych komponentach, które nie są dołączane do większości obrazów kontenerów.

Na początku KinD oferuje obraz bazowy, czyli przygotowany przez zespół tworzący KinD obraz zawierający wszystko to, co jest potrzebne do działania Dockera, Kubernetes i `systemd`. Skoro obraz bazowy został opracowany na podstawie obrazu Ubuntu, zespół KinD usunął niepotrzebne usługi i skonfigurował `systemd` na potrzeby Dockera. Następnie obraz węzła jest tworzony na podstawie tego obrazu bazowego.

> Jeżeli interesują Cię szczegóły związane ze sposobem tworzenia obrazu bazowego, możesz się zapoznać z plikiem *Dockerfile* używanym przez zespół rozwijający KinD. Ten plik znajdziesz pod adresem *https://github.com/kubernetes-sigs/kind/blob/main/images/base/Dockerfile*.

KinD i sieć Dockera

Ponieważ KinD używa Dockera jako silnika kontenerów przeznaczonego do uruchamiania węzłów klastra, wszystkich klastrów dotyczą te same ograniczenia sieci, które pojawiają się w przypadku standardowego kontenera Dockera. W rozdziale 3. przedstawiliśmy krótkie wprowadzenie do tematu sieci w Dockerze i potencjalnych ograniczeń domyślnego stosu sieciowego w Dockerze. Oczywiście te ograniczenia nie będą uniemożliwiały przetestowania klastra KinD Kubernetes w hoście lokalnym, choć jednocześnie mogą prowadzić do problemów, gdy zajdzie potrzeba przetestowania kontenerów z poziomu innych komputerów znajdujących się w sieci.

W trakcie rozważań dotyczących sieci Dockera pod uwagę trzeba wziąć również interfejs **CNI** (ang. *Container Network Interface*) w Kubernetes. Oficjalnie KinD ogranicza opcje sieciowe do zaledwie dwóch: Kindnet i Calico. Kindnet to jedyny obsługiwany typ CNI, przy czym masz możliwość wyłączenia domyślnej instalacji Kindnet, co spowoduje utworzenie klastra bez obsługi CNI. Następnie po wdrożeniu klastra można dodać obsługę CNI, np. za pomocą Calico.

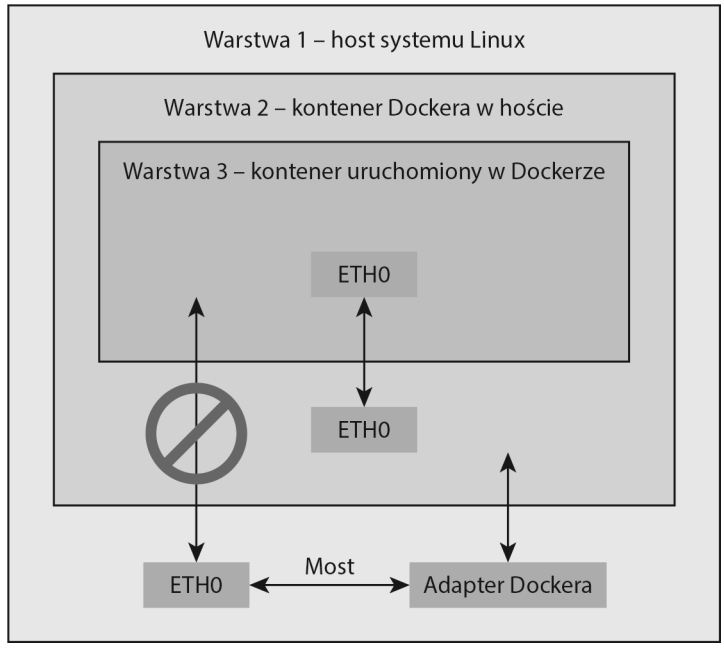
Wiele instalacji Kubernetes — w przypadku zarówno niewielkich wdrożeń, jak i klastrów w dużych firmach — używa Tigera Calico do obsługi CNI. Dlatego też zdecydowaliśmy się na zastosowanie takiego samego rozwiązania w przykładach zamieszczonych w książce.

Monitorowanie zagnieżdżonych kontenerów

Uruchomienie rozwiązania opartego na KinD może być dezorientujące, ponieważ mamy tutaj do czynienia z wdrożeniem typu „kontener w kontenerze”. Można to porównać do rosyjskich laleczek, tzw. matrioszek: jedna lalka jest włożona w drugą, w której środku jest następna itd. Gdy zaczniesz eksperymentować z KinD we własnym klastrze, możesz się pogubić w ścieżkach komunikacji między hostem, Dockerem i węzłami Kubernetes. Aby zachować jasność umysłu, należy doskonale wiedzieć, gdzie zostały uruchomione poszczególne komponenty, a także jak one ze sobą współdziałają.

Na rysunku 4.4 pokazaliśmy trzy warstwy, które muszą być uruchomione w klastrze KinD. Trzeba w tym miejscu koniecznie dodać, że każda z tych warstw może się komunikować tylko z warstwą znajdującą się powyżej. Dlatego też kontener KinD na warstwie trzeciej ma dostęp jedynie do obrazu Dockera działającego na warstwie drugiej, mającego z kolei dostęp do hosta systemu Linux działającego na warstwie pierwszej. Jeżeli chcesz zapewnić możliwość bezpośredniej komunikacji hosta z kontenerem uruchomionym w klastrze KinD, musisz przejść przez warstwę Dockera, a następnie przejść do kontenera Kubernetes działającego na warstwie trzeciej.

Tę koncepcję trzeba koniecznie zrozumieć, aby można było efektywnie używać KinD jako środowiska testowego.

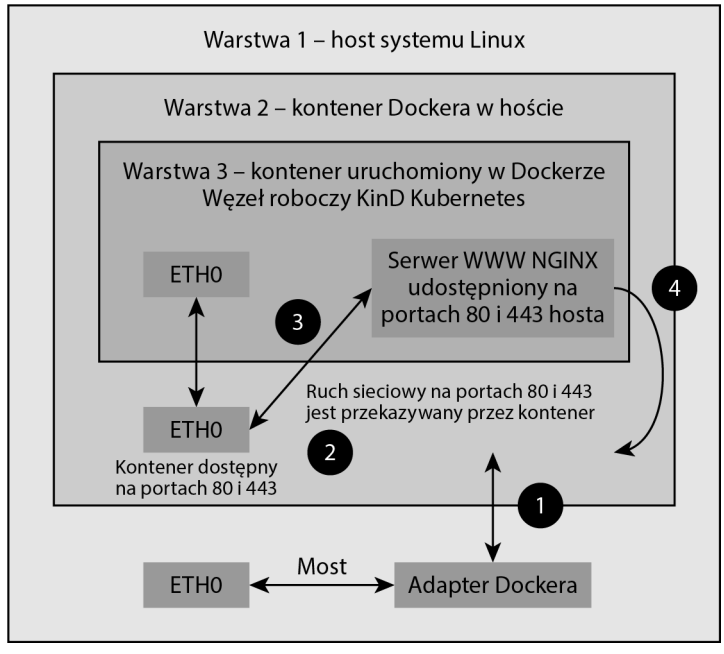


Rysunek 4.4. Host nie może się bezpośrednio komunikować z KinD

Przeanalizujmy np. sytuację, w której zachodzi potrzeba wdrożenia serwera WWW w klastrze Kubernetes. Przeprowadzasz wdrożenie kontrolera Ingress w klastrze KinD, a następnie chcesz przetestować witrynę internetową za pomocą hosta Docker lub innej stacji roboczej w sieci. Próba przejścia do portu 80 w hoście docelowym kończy się wygenerowaniem komunikatu błędu w przeglądarce WWW. Dlaczego?

Pod odpowiedzialny za działanie serwera WWW znajduje się na warstwie trzeciej i nie może bezpośrednio otrzymywać ruchu sieciowego z hosta lub innych komputerów znajdujących się w sieci. Aby z poziomu hosta mieć dostęp do tego serwera WWW, konieczne jest przekazanie ruchu sieciowego z warstwy Dockera na warstwę KinD. W rozdziale 3. wyjaśniliśmy, że udostępnienie kontenera w sieci odbywa się przez dodanie do niego portu, na którym kontener nasłuchuje przychodzącego ruchu sieciowego. W omawianym przykładzie to muszą być porty 80 i 443. Po uruchomieniu kontenera nasłuchującego na portach demon Dockera będzie przychodzący ruchy sieciowy przekazywał z hosta do uruchomionego kontenera Dockera (zobacz rysunek 4.5).

Po udostępnieniu portów 80 i 443 w kontenerze Dockera demon Dockera będzie akceptował żądania przychodzące do wymienionych portów, a kontroler Ingress serwera WWW NGINX otrzyma ruch sieciowy. Takie rozwiązanie jest możliwe, ponieważ porty 80 i 443 zostały udostępnione w dwóch miejscach na warstwie Dockera. Są dostępne na warstwie Kubernetes dzięki uruchomieniu kontenera serwera NGINX z użyciem portów 80 i 443 hosta. Ten proces instalacji zostanie dokładnie wyjaśniony w dalszej części rozdziału, natomiast w tym miejscu najważniejsze jest zrozumienie ogólnego sposobu działania takiego rozwiązania.



Rysunek 4.5. Host komunikuje się z warstwą KinD za pomocą kontrolera Ingress

Po stronie hosta jest wykonywane żądanie do serwera WWW, który ma zdefiniowaną regułę Ingress w klastrze Kubernetes:

1. Żądanie sprawdza oczekiwany adres IP (w omawianym przykładzie to jest lokalny adres IP).
2. Kontener Docker z uruchomionym węzłem Kubernetes nasłuchuje na adresie IP porty 80 i 443, więc żądanie zostaje zaakceptowane i przekazane do uruchomionego kontenera.
3. Pod serwera WWW NGINX w klastrze Kubernetes został skonfigurowany do użycia portów 80 i 443 hosta, więc ruch sieciowy zostaje przekazany do poda.
4. Za pomocą kontrolera Ingress użytkownik otrzymuje żądaną stronę internetową z serwera WWW NGINX.

Ta procedura może być nieco dezorientująca, ale im dłużej pracuje się z KinD, tym bardziej staje się zrozumiała.

Aby używać klasta KinD w projekcie programistycznym, trzeba koniecznie poznać sposób działania platformy KinD. Dotychczas wyjaśniliśmy koncepcję obrazu węzła i to, jak ten obraz jest używany podczas tworzenia klastra. Przedstawiliśmy również przepływ ruchu sieciowego między hostem Dockera i kontenerami odpowiedzialnymi za działanie klastra. Mając tę wiedzę, możesz przejść do tematu tworzenia klastra Kubernetes za pomocą KinD.

Instalacja KinD

Pliki wykorzystane w tym rozdziale znajdują się w katalogu *Rozdzial04* w materiałach przygotowanych dla książki. Możesz użyć tych plików lub też utworzyć własne na podstawie informacji zamieszczonych w rozdziale. W tym podrozdziale wyjaśnimy poszczególne kroki procesu instalacyjnego.

> W chwili powstawania rozdziału bieżącą wersją KinD była 0.8.1. W wersji 0.8.0 wprowadzono nową funkcjonalność pozwalającą na zachowanie stanu klastra między ponownymi uruchomieniami systemu i Dockera.

Przygotowanie do instalacji KinD

KinD wymaga pewnych przygotowań, zanim będzie można utworzyć klaster. W tej sekcji szczegółowo omówimy poszczególne wymagania i pokażemy, jak zainstalować niezbędne komponenty.

Instalacja kubectl

Skoro KinD to pojedynczy plik wykonywalny, narzędzie `kubectl` nie zostanie zainstalowane. Jeżeli wymienionego narzędzia nie masz jeszcze w systemie Ubuntu 18.04, możesz je zainstalować za pomocą tego polecenia:

```
$ sudo snap install kubectl --classic
```

Instalacja języka programowania Go

Zanim będzie można utworzyć klaster KinD, w komputerze gospodarza musi być dostępny język programowania Go. Jeżeli spełniasz to wymaganie, możesz przejść do następnego punktu. Instalacja języka Go wymaga pobrania odpowiedniego archiwum, wyodrębnienia pliku wykonywalnego i zdefiniowanie ścieżki dostępu projektu. Polecenia wymienione w kolejnym fragmencie kodu pozwalają na zainstalowanie języka Go w komputerze.

```
$ wget https://dl.google.com/go/go1.13.3.linux-amd64.tar.gz
$ tar -xzf go1.13.3.linux-amd64.tar.gz
$ sudo mv go /usr/local
$ mkdir -p $HOME/Projects/Project1

$ cat << 'EOF' >> ~/.bash_profile
export -p GOROOT=/usr/local/go
export -p GOPATH=$HOME/Projects/Project1
export -p PATH=$GOPATH/bin:$GOROOT/bin:$PATH
EOF
$ source ~/.bash_profile
```

Skrypt instalacyjny zawierający niezbędne polecenia znajduje się w materiałach przygotowanych dla książki (*Rozdzial04/install-go.sh*).

Oto wyjaśnienie sposobu działania przedstawionego fragmentu kodu:

- Pobranie archiwum języka Go do komputera gospodarza, rozpakowanie archiwum i przeniesienie plików do katalogu */usr/local*.
- Utworzenie w katalogu domowym użytkownika katalogu projektu Go o nazwie *Projects/Project1*.
- Dodanie do pliku *.bash_profile* zmiennej środowiskowej Go niezbędnej do uruchamiania aplikacji w języku Go.

Po przygotowaniu komputera gospodarza można przystąpić do instalacji KinD.

Instalacja pliku binarnego KinD

Instalacja KinD to łatwy proces, który można przeprowadzić za pomocą pojedynczego polecenia. Ta instalacja może się odbywać przez wykonanie skryptu umieszczonego w materiałach przygotowanych dla książki (zapoznaj się z plikiem *Rozdzial04/install-kind.sh*). Ewentualnie przejdź do powłoki, a następnie zastosuj przedstawione tutaj polecenie:

```
$ GO111MODULE="on" go get sigs.k8s.io/kind@v0.7.0
```

Poprawność procesu instalacji KinD można potwierdzić przez przejście do powłoki i użycie w niej tego polecenia:

```
$ kind version
```

Powinno ono wygenerować dane wyjściowe podobne do tych:

```
$ kind v0.7.0 go1.13.3 linux/amd64
```

Plik wykonywalny KinD zapewnia dostęp do wszystkich opcji niezbędnych do zarządzania cyklem życiowym klastra. Oczywiście ten plik pozwala nie tylko na tworzenie i usuwanie klastrów, ale również na przeprowadzanie innych operacji, takich jak:

- tworzenie niestandardowych obrazów bazowych i węzłów,
- eksportowanie *kubeconfig* lub plików dzienników zdarzeń,
- pobieranie klastrów, węzłów lub plików *kubeconfig*,
- wczytywanie obrazów do węzłów.

Po zainstalowaniu narzędzia KinD masz wszystko to, co jest potrzebne do tworzenia klastrów KinD. Jednak zanim użyjesz kilku poleceń `create cluster`, najpierw omówimy wybrane z oferowanych przez KinD opcji związanych z tworzeniem klastra.

Tworzenie klastra KinD

Po spełnieniu wszystkich wymagań można przystąpić do utworzenia pierwszego klastra za pomocą pliku wykonywalnego KinD. To narzędzie pozwala na tworzenie nie tylko klastra składającego się z pojedynczego węzła, ale również skomplikowanych klastrów z wieloma węzłami, warstwami sterowania i węzłami roboczymi. W tym podrozdziale omówimy wybrane opcje obsługiwane przez plik wykonywalny KinD. Gdy zakończysz lekturę rozdziału, będziesz mieć działający klaster składający się z dwóch węzłów: warstwy sterowania i węzła roboczego.

> W przykładach zamieszczonych w książce tworzymy klaster składający się z wielu węzłów. Konfiguracja prostego klastra służy jedynie jako przykład i nie będzie używana w innych fragmentach kodu.

Tworzenie prostego klastra

W celu utworzenia prostego klastra składającego się z warstwy sterowania i węzła roboczego uruchomionych w pojedynczym kontenerze trzeba użyć pliku wykonywalnego KinD jedynie z opcją `create cluster`.

Przystępujemy teraz do utworzenia takiego klastra, co pozwoli pokazać, jak bardzo KinD może ułatwić pracę. W komputerze gospodarza utworzenie prostego klastra następuje po wprowadzeniu przedstawionego tutaj polecenia:

```
$ kind create cluster
```

W wyniku działania tego polecenia w pojedynczym kontenerze Dockera zostanie utworzony klaster Kubernetes o nazwie `kind` z wszystkimi niezbędnymi komponentami. Temu klastrowi będzie przypisany kontener Dockera o nazwie `kind-control-plane`. Jeżeli zamiast domyślnej nazwy klastra chcesz użyć innej, do polecenia `create cluster` musisz dodać opcję `--name <nazwa klastra>`. Oto dane wyjściowe wygenerowane przez wymienione wcześniej polecenie:

```
Creating cluster "kind" ...
  Ensuring node image (kindest/node:v1.18.2)
  Preparing nodes
  Writing configuration
  Starting control-plane
  Installing CNI
  Installing StorageClass
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind
```

Polecenie `create` tworzy klaster i modyfikuje zawartość pliku *kubeconfig* narzędzia `kubectl`. KinD dodaje ten nowy klaster do istniejącego pliku *kubeconfig* i określi ten klaster jako kontekst domyślny.

Poprawność utworzenia nowego klastra można sprawdzić przez wyświetlenie listy węzłów za pomocą narzędzia `kubectl`, jak pokazaliśmy w kolejnym poleceniu.

```
$ kubectl get nodes
```

Wygenerowane przez nie dane wyjściowe zawierają informacje o działających węzłach. W omawianym przykładzie będzie to prosty klaster z pojedynczym węzłem:

```
NAME                 STATUS   ROLES    AGE   VERSION
kind-control-plane Ready    master   130m  v1.18.2
```

Naszym głównym celem, jeśli chodzi o wdrożenie tego klastra składającego się z pojedynczego węzła, było pokazanie, jak łatwo za pomocą KinD można utworzyć klaster, który później będzie wykorzystywany do testów. Na potrzeby przykładów zamieszczonych w książce musimy rozdzielić warstwę sterowania i węzeł roboczy, więc usuniemy ten prosty klaster za pomocą procedury omówionej w następnej sekcji.

Usunięcie klastra

Po zakończeniu testów klaster można usunąć za pomocą polecenia `delete`.

```
$ kind delete cluster –name <nazwa klastra>
```

Działanie polecenia `delete` polega na szybkim usunięciu klastra, łącznie z dodanymi dla niego wpisami w pliku *kubeconfig*.

Składający się z pojedynczego węzła klaster okazuje się przydatny w wielu sytuacjach. Jednak w przypadku różnych scenariuszy testowych zwykle trzeba tworzyć klastry zawierające wiele węzłów. Utworzenie bardziej złożonego klastra wymaga przygotowania pliku konfiguracyjnego.

Tworzenie pliku konfiguracyjnego klastra

Gdy chcesz utworzyć klaster składający się z wielu węzłów, np. zawierający dwa węzły klaster wykorzystujący opcje niestandardowe, wówczas konieczne jest przygotowanie pliku konfiguracyjnego klastra. Ten plik powinien być utworzony w doskonale znanym formacie YAML. Zdefiniowanie wartości w tym pliku pozwoli na dostosowanie klastra KinD do własnych potrzeb m.in. przez określenie liczby węzłów, opcji API itd. W kolejnym fragmencie kodu zamieściliśmy plik konfiguracyjny, którego użyjemy do utworzenia klastra wykorzystywanego w przykładach omawianych w książce. Ten plik znajdziesz również w materiałach przygotowanych dla książki (zobacz *Rozdzial04/cluster01-kind.yaml*).

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  apiServerAddress: "0.0.0.0"
disableDefaultCNI: true kubeadmConfigPatches:
-|
  apiVersion: kubeadm.k8s.io/v1beta2
  kind: ClusterConfiguration
```

```
metadata:
  name: config
networking:
  serviceSubnet: "10.96.0.1/12"
  podSubnet: "192.168.0.0/16"
nodes:
  - role: control-plane
  - role: worker
    extraPortMappings:
    - containerPort: 80
      hostPort: 80
    - containerPort: 443
      hostPort: 443
    extraMounts:
    - hostPath: /usr/src
      containerPath: /usr/src
```

W tabeli 4.3 wyjaśniliśmy znaczenie poszczególnych opcji użytych w przedstawionym pliku konfiguracyjnym.

Jeżeli planujesz utworzenie klastra wykraczającego poza prosty klaster składający się z pojedynczego węzła i nie chcesz przy tym używać opcji zaawansowanych, konieczne będzie utworzenie pliku konfiguracyjnego. Zrozumienie przeznaczenia dostępnych opcji pozwala na utworzenie klastra Kubernetes wykorzystującego komponenty zaawansowane, np. kontrolery Ingress, lub wiele węzłów w celu przetestowania procedur obsługi awarii i przywracania wdrożenia po awarii.

W ten sposób już wiesz, jak można utworzyć prosty klaster, którego cała zawartość mieści się w pojedynczym kontenerze, a także jak za pomocą pliku konfiguracyjnego tworzyć klaster składający się z wielu węzłów. Możemy więc przejść do przykładu znacznie bardziej złożonego klastra.

Konfiguracja klastra składającego się z wielu węzłów

Jeżeli chcesz jedynie utworzyć klaster składający się z wielu węzłów i nie chcesz przy tym korzystać z żadnych opcji dodatkowych, możesz przygotować prosty plik konfiguracyjny zawierający liczbę i typy węzłów, które mają się znaleźć w klastrze. Zamieszczony w kolejnym fragmencie kodu plik konfiguracyjny prowadzi do utworzenia klastra z trzema węzłami warstwy sterowania i trzema węzłami roboczymi.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: control-plane
- role: control-plane
- role: worker
- role: worker
- role: worker
```

Tabela 4.3. Opcje konfiguracyjne KinD użyte podczas tworzenia przykładowego klastra

Opcja konfiguracyjna	Opis
`apiServerAddress`	Pozwala na określenie adresu IP, na którym nasłuchuje serwer API. Domyślnie jest używana wartość `127.0.0.1`. Skoro w omawianym przykładzie planujemy używanie klastra z poziomu innych komputerów znajdujących się w sieci, to zdecydowaliśmy o nasłuchiwaniu na wszystkich adresach IP
`disableDefaultCNI`	Pozwala na włączenie lub wyłączenie instalacji Kindnet. Wartością domyślną tej opcji jest `false`. Skoro w omawianym przykładzie chcemy użyć Calico jako CNI, to omawianej opcji została przypisana wartość `true`
`kubeadmConfigPatches`	Ta sekcja pozwala na zdefiniowanie wartości dla określonej grupy ustawień podczas procesu instalacji. W przypadku naszej warstwy sterowania zdefiniowaliśmy CIDR dla `ServiceSubnet` i `podSubnet`
`nodes`	W tej sekcji są definiowane węzły. W omawianym przykładzie zostanie utworzony jeden węzeł warstwy sterowania i jeden węzeł roboczy
`- role: control-plane`	W tym miejscu definiujemy warstwę sterowania w klastrze, która składa się z pojedynczego węzła. Można tutaj podać opcje związane z warstwą sterowania. Więcej informacji na ich temat znajdziesz w następnym wierszu tabeli
`kubeadmConfigPatches`	Ta sekcja pozwala na zdefiniowanie opcji warstwy sterowania. W omawianym przykładzie dodaliśmy ustawienia dla serwera API, aby zawierał parametry OIDC, które będą używane w późniejszych rozdziałach
`- role: worker`	Ta sekcja definiuje węzeł roboczy
`extraPortMappings`	Ta sekcja pozwala na utworzenie dodatkowego mapowania portów dla węzła roboczego. W omawianym przykładzie nakazujemy Dockerowi na dołączenie portów 80 i 443 do kontenera węzła roboczego
`extraMounts`	Ta sekcja nakazuje Dockerowi utworzenie dodatkowych punktów montowania między hostem i kontenerem. W omawianym przykładzie użyliśmy wartości pozwalającej kontenerowi utworzonemu w dalszej części książki na wykorzystanie plików z katalogu */usr/src* komputera gospodarza

Użycie warstwy sterowania składającej się z wielu serwerów powoduje zwiększenie poziomu skomplikowania projektu, ponieważ w pliku konfiguracyjnym można podawać tylko pojedynczy host lub adres IP. Aby taki plik konfiguracyjny był dla nas użyteczny, konieczne jest wdrożenie dla klastra mechanizmu równoważenia obciążenia.

Platforma KinD również to uwzględnia i jeśli przeprowadzasz wdrożenie wielu węzłów warstwy sterowania, w trakcie procesu instalacji następuje utworzenie dodatkowego kontenera zawierającego mechanizm równoważenia obciążenia HAProxy. Jeżeli spojrzysz na listę uruchomionych

kontenerów po skonfigurowaniu klastra składającego się z wielu węzłów, w omawianym przykładzie zobaczysz sześć kontenerów węzłów i kontener mechanizmu równoważenia obciążenia HAProxy, jak pokazaliśmy w tabeli 4.4.

Tabela 4.4. Lista kontenerów w omawianym przykładzie

Kontener	Identyfikator	Port	Nazwa
29b28504239b	kindest/node:v1.17.0		config2-worker2
ac3efb14fd51	kindest/node:v1.17.0		config2-worker
c1beba396fe8	kindest/node:v1.17.0	127.0.0.1:32792->6443/tcp	config2-control-plane3
26ceb0c672a5	kindest/node:v1.17.0	127.0.0.1:32794->6443/tcp	config2-control-plane
99dd75667824	kindest/haproxy:2.1.1-alpine	127.0.0.1:32791->6443/tcp	config2-external-load-balancer
25c9dalb7ffa	kindest/node:v1.17.0	127.0.0.1:32793->6443/tcp	config2-control-plane2
b8e201938cbe	kindest/node:v1.17.0		config2-worker3

Przypomnij sobie zamieszczone w rozdziale 3. informacje dotyczące portów i gniazd. Skoro w omawianym przykładzie mamy pojedynczy komputer gospodarza, każdy węzeł warstwy sterowania i kontener HAProxy nasłuchują na unikatowych portach. Poszczególne kontenery muszą być udostępnione komputerowi gospodarza, aby mogły otrzymywać żądania przychodzące. W budowanym tutaj rozwiązaniu trzeba koniecznie zwrócić uwagę na port przypisany HAProxy, ponieważ to jest port docelowy dla klastra. Jeżeli przyjrzysz się plikowi konfiguracyjnemu Kubernetes, zauważysz adres docelowy *https://127.0.0.1:32791*, który zawiera numer portu przypisany kontenerowi HAProxy.

Po wykonaniu polecenia za pomocą `kubectl` zostaje ono przekazane bezpośrednio do serwera HAProxy. Korzystając z pliku konfiguracyjnego utworzonego przez KinD razem z klastrem, kontener HAProxy pozyskuje informacje pozwalające mu na przekazywanie ruchu sieciowego między trzema węzłami warstwy sterowania.

```
# Dane wygenerowane przez KinD.
global
  log /dev/log local0
  log /dev/log local1 notice
  daemon

defaults
  log global
  mode tcp
  option dontlognull
  # TODO: dostosowanie tych wartości do własnych potrzeb.
  timeout connect 5000
  timeout client 50000
  timeout server 50000

frontend control-plane
```

```
    bind *:6443

    default_backend kube-apiservers

backend kube-apiservers
    option httpchk GET /healthz
    # TODO: należy przeprowadzić weryfikację (!).

    server config2-control-plane 172.17.0.8:6443 check check-ssl verify none
    server config2-control-plane2 172.17.0.6:6443 check check-ssl verify none
    server config2-control-plane3 172.17.0.5:6443 check check-ssl verify none
```

Jak można zobaczyć w przedstawionym pliku konfiguracyjnym, mamy sekcję backendu o nazwie `kube-apiservers`, zawierającą trzy kontenery warstwy sterowania. Każdy element tej sekcji zawiera adres IP Dockera dla węzła warstwy sterowania razem z przypisanym portem 6443, prowadzącym do uruchomionego w kontenerze serwera API. Po wykonaniu żądania pod adresem *https://127.0.0.1:32791* dotrze ono do kontenera HAProxy. Z wykorzystaniem reguł zdefiniowanych w pliku konfiguracyjnym HAProxy żądanie zostanie przekierowane do jednego z trzech węzłów znajdujących się na liście.

Skoro nasz przykładowy klaster korzysta z mechanizmu równoważenia obciążenia, otrzymujemy do dyspozycji wysoce konfigurowalną warstwę sterowania do testów.

> Dołączony obraz HAProxy nie jest konfigurowalny. Został dostarczony jedynie w celu zapewnienia obsługi warstwy sterowania i mechanizmu równoważenia obciążenia serwerów API. Ze względu na to ograniczenie, jeżeli mechanizmu równoważenia obciążenia chcesz używać z węzłami roboczymi, musisz samodzielnie przygotować odpowiednie rozwiązanie.
>
> Przykładem zastosowania może być sytuacja, w której chcesz użyć kontrolera Ingress w wielu węzłach roboczych. Dla węzłów roboczych jest więc potrzebny mechanizm równoważenia obciążenia, który będzie akceptować żądania przychodzące do portów 80 i 443, a następnie będzie je kierował do poszczególnych węzłów z uruchomionym serwerem WWW NGINX. W dalszej części rozdziału przedstawimy przykładową konfigurację obejmującą niestandardowe ustawienia HAProxy pozwalające na równoważenie obciążenia ruchu sieciowego dla węzłów roboczych.

Dostosowanie do własnych potrzeb warstwy sterowania i opcji kubelet

Można pójść jeszcze o krok dalej i przetestować funkcjonalność taką jak integracja z OIDC lub bramy Kubernetes. KinD używa tej samej konfiguracji, z której korzysta instalacja `kubadm`. Jeśli np. chcesz zintegrować klaster z dostawcą OIDC, odpowiednie opcje możesz dodać do sekcji `kubeadmConfigPatches`, jak pokazaliśmy w kolejnym fragmencie kodu.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4 kubeadmConfigPatches:
-|
  kind: ClusterConfiguration
  metadata:
    name: config
  apiServer:
extraArgs:
      oidc-issuer-url: "https://oidc.testdomain.com/auth/idp/k8sIdp"
      oidc-client-id: "kubernetes"
      oidc-username-claim: sub
      oidc-client-id: kubernetes
      oidc-ca-file: /etc/oidc/ca.crt
nodes:
- role: control-plane
- role: control-plane
- role: control-plane
- role: worker
- role: worker
- rol: worker
```

Lista dostępnych opcji została dokładnie omówiona w dokumencie *Customizing control plane configuration with kubeadm* w dokumentacji Kubernetes zamieszczonej na stronie *https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/control-plane-flags/*.
W ten sposób masz przygotowany plik klastra. Kolejnym krokiem jest utworzenie klastra KinD.

Tworzenie własnego klastra KinD

Wreszcie! Skoro znasz już możliwości oferowane przez KinD, możesz przejść dalej i przystąpić do utworzenia klastra.

Konieczne jest utworzenie kontrolowanego, doskonale znanego środowiska. Dlatego też klastrowi nadamy nazwę i wykorzystamy plik konfiguracyjny, który został dokładnie omówiony w poprzedniej sekcji.

Upewnij się, że w katalogu bieżącym masz pliki pobrane z materiałów przygotowanych dla książki.

W celu utworzenia klastra KinD razem z żądanymi opcjami konieczne jest uruchomienie programu instalacyjnego KinD z takimi opcjami:

```
$ kind create cluster --name cluster01 --config c luster01-kind.yaml
```

Opcja `--name` spowoduje nadanie klastrowi nazwy `cluster01`, natomiast opcja `--config` nakazuje programowi instalacyjnemu użycie pliku konfiguracyjnego o nazwie *cluster01-kind.yaml*.

Po uruchomieniu programu instalacyjnego w komputerze gospodarza KinD rozpocznie proces instalacji i będzie wyświetlać informacje o kolejnych wykonywanych krokach. Cała operacja utworzenia klastra powinna trwać poniżej 2 minut (zobacz rysunek 4.6).

```
Creating cluster "cluster01" ...
 ✓ Ensuring node image (kindest/node:v1.17.0)
 ✓ Preparing nodes
 ✓ Writing configuration
 ✓ Starting control-plane
 ✓ Installing StorageClass
 ✓ Joining worker nodes
Set kubectl context to "kind-cluster01"
You can now use your cluster with:

kubectl cluster-info --context kind-cluster01

Not sure what to do next? Check out https://kind.sigs.k8s.io/docs/user/quick-start/
```

Rysunek 4.6. Dane wyjściowe wygenerowane podczas tworzenia klastra KinD

W ostatnim kroku wdrożenia następuje utworzenie lub zmodyfikowanie istniejącego pliku konfiguracyjnego Kubernetes. W obu tych przypadkach program instalacyjny tworzy nowy kontekst o nazwie `kind-<nazwa klastra>` i ustawia go jako kontekst domyślny.

Wprawdzie może się wydawać, że na tym etapie procedura tworzenia nowego klastra jest zakończona, ale klaster **nie** jest jeszcze gotowy. Pewne zadania wymagają kilku minut, aby w pełni zainicjalizować klaster. Ponadto, skoro wyłączyliśmy domyślny interfejs CNI, aby użyć Calico, trzeba się zająć wdrożeniem Calico i zapewnić klastrowi obsługę sieci.

Instalacja Calico

W celu zapewnienia obsługi sieci podom klastra konieczne jest zainstalowanie interfejsu CNI. Zdecydowaliśmy się na instalację Calico jako naszego CNI. Ponieważ KinD oferuje jedynie Kindnet CNI, to instalację Calico trzeba przeprowadzić ręcznie.

Jeżeli po etapie tworzenia klastra zatrzymasz się na chwilę i przeanalizujesz klaster, zauważysz, że niektóre pody znajdują się w stanie oczekiwania:

```
coredns-6955765f44-86l77   0/1   Pending   0   10m
coredns-6955765f44-bznjl   0/1   Pending   0   10m
local-path-provisioner-7   0/1   Pending   0   11m
745554f7f-jgmxv
```

Uruchomienie wymienionych tutaj podów wymaga działającego interfejsu CNI. Dlatego też pody znajdują się w stanie oczekiwania na zapewnienie im obsługi sieci. Skoro nie zdecydowaliśmy się na wdrożenie domyślnego interfejsu CNI, nasz klaster jeszcze nie zapewnia obsługi sieci. Aby stan podów zmienił się z oczekiwania na działanie, konieczne jest zainstalowanie CNI — w przypadku naszego klastra będzie to Calico.

Instalacja Calico wykorzystuje domyślne wdrożenie Calico, które wymaga tylko jednego manifestu. Rozpoczęcie procedury wdrażania Calico następuje po wprowadzeniu przedstawionego tutaj polecenia:

```
$ kubectl apply -f https://docs.projectcalico.org/v3.11/manifests/calico.yaml
```

Działanie tego polecenia polega na pobraniu manifestu z internetu i zastosowaniu go w klastrze. Podczas operacji wdrażania będzie można zobaczyć, że zostało utworzonych wiele obiektów Kubernetes, jak pokazaliśmy na rysunku 4.7.

```
configmap/calico-config created
customresourcedefinition.apiextensions.k8s.io/felixconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamblocks.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/blockaffinities.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamhandles.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamconfigs.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgppeers.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgpconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ippools.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/hostendpoints.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/clusterinformations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networksets.crd.projectcalico.org created
clusterrole.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrolebinding.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrole.rbac.authorization.k8s.io/calico-node created
clusterrolebinding.rbac.authorization.k8s.io/calico-node created
daemonset.apps/calico-node created
serviceaccount/calico-node created
deployment.apps/calico-kube-controllers created
serviceaccount/calico-kube-controllers created
```

Rysunek 4.7. Dane wyjściowe wygenerowane podczas instalacji Calico

Proces instalacji zabiera około minuty, a jego stan można sprawdzić za pomocą polecenia `kubectl get pods -n kube-system`. Zobaczysz, że zostały utworzone trzy pody Calico: dwa `calico-node` i jeden `calico-kube-controller`:

```
NAME                       READY STATUS RESTARTS AGE
calico-kube-controllers    1/1   Running    0    64s
-5b644bc49c-nm5wn
calico-node-4dqnv          1/1   Running    0    64s
calico-node-vwbpf          1/1   Running    0    64s
```

Po ponownym sprawdzeniu dwóch podów CoreDNS w przestrzeni nazw `kube-system` zauważysz, że ich stan zmienił się z oczekiwania (przed instalacją Calico) na działanie.

```
coredns-6955765f44-86l77    1/1   Running    0   18m
coredns-6955765f44-bznjl    1/1   Running    0   18m
```

Po zainstalowaniu CNI w klastrze wszystkie pody wymagające sieci do działania będą się znajdowały w stanie działania.

Instalacja kontrolera Ingress

W książce znajduje się poświęcony kontrolerowi Ingress rozdział, w którym dokładnie wyjaśniamy wszystkie dotyczące go szczegóły techniczne. Ponieważ teraz zajmujemy się wdrażaniem klastra, a kontroler Ingress będzie potrzebny w przykładach zamieszczonych w późniejszych rozdziałach, musimy go wdrożyć, aby otrzymać w pełni przygotowany klaster. Wszystkie szczegóły dotyczące kontrolera Ingress znajdziesz w rozdziale 6.

Instalacja kontrolera Ingress dla serwera WWW NGINX wymaga jedynie dwóch plików manifestów, które zostaną pobrane z internetu, aby maksymalnie uprościć całą procedurę. W powłoce użyj dwóch przedstawionych tutaj poleceń, które przeprowadzą instalację niezbędnych składników.

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/
↪nginx-0.28.0/deploy/static/mandatory.yaml
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/
↪nginx-0.27.0/deploy/static/provider/baremetal/service-nodeport.yaml
```

Wdrożenie spowoduje utworzenie w przestrzeni nazw `ingress-nginx` kilku obiektów Kubernetes wymaganych przez kontrolera Ingress, jak pokazaliśmy na rysunku 4.8.

```
namespace/ingress-nginx created
configmap/nginx-configuration created
configmap/tcp-services created
configmap/udp-services created
serviceaccount/nginx-ingress-serviceaccount created
clusterrole.rbac.authorization.k8s.io/nginx-ingress-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-role created
rolebinding.rbac.authorization.k8s.io/nginx-ingress-role-nisa-binding created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress-clusterrole-nisa-binding created
deployment.apps/nginx-ingress-controller created
limitrange/ingress-nginx created
```

Rysunek 4.8. Dane wyjściowe wygenerowane podczas instalacji kontrolera Ingress dla serwera WWW NGINX

Aby mieć w pełni funkcjonujący kontroler Ingress, konieczne jest wykonanie jeszcze tylko jednego kroku: udostępnienie portów 80 i 443 uruchomionemu podowi. To można zrobić przez modyfikację wdrożenia. Spójrz na polecenie wprowadzające niezbędne zmiany:

```
$ kubectl patch deployments -n ingress-nginx nginx-ingress-controller -p
↪'{"spec":{"template":{"spec":{"containers":[{"name":"nginx-ingress-controller",
↪"ports":[{"containerPort":80,"hostPort":80},{"containerPort":443,"hostPort":443}
↪]}]}}}}'
```

Gratulacje! W ten sposób masz w pełni funkcjonujący złożony z dwóch węzłów klaster Kubernetes, który używa Calico i kontrolera Ingress.

Analiza utworzonego klastra KinD

Skoro mamy przygotowany klaster Kubernetes, możemy przeanalizować znajdujące się w nim obiekty. To pomoże w zrozumieniu materiału przedstawionego w poprzednim rozdziale, w którym omówiliśmy wiele obiektów bazowych umieszczanych w klastrze Kubernetes. Szczególnie interesują nas obiekty pamięci masowej znajdujące się w klastrze KinD.

Obiekty pamięci masowej KinD

Przypomnij sobie, że KinD zawiera wtyczkę przeznaczoną do automatycznego zarządzania trwałą pamięcią masową w klastrze. W rozdziale 5. znajdziesz dokładne omówienie obiektów związanych z pamięcią masową. W tym momencie dysponujemy klastrem razem ze skonfigurowanym systemem pamięci masowej, więc warto dowiedzieć się o nich nieco więcej.

Istnieje jeden obiekt niewymagany przez wtyczkę automatycznie zarządzającą trwałą pamięcią masową, ponieważ jest używana standardowa funkcjonalność Kubernetes: `CSIdriver`. Skoro Kubernetes pozwala na używanie ścieżek dostępu hosta lokalnego jako PVC, to w klastrze KinD nie zobaczymy żadnych obiektów `CSIdriver`.

Analizę obiektów klastra KinD rozpoczniemy od `CSInode`. Wcześniej wspomnieliśmy, że ten obiekt jest tworzony w celu odłączenia wszelkich obiektów CSI od obiektów węzła bazowego. Każdy węzeł przeznaczony do wykonywania zadań będzie miał obiekt `CSInode`. W omawianym przykładzie klastra KinD oba węzły zawierają obiekty `CSInode`, co można potwierdzić za pomocą polecenia `kubectl get csinodes`.

```
NAME                    CREATED AT
cluster01-control-plane 2020-03-27T15:18:19Z
cluster01-worker        2020-03-27T15:19:01Z
```

Dokładne informacje o wybranym obiekcie węzła zostaną wyświetlone po użyciu polecenia `kubectl describe csinodes <nazwa węzła>`, jak pokazaliśmy na rysunku 4.9.

```
[root@localhost yaml]# kubectl describe csinodes cluster01-worker
Name:         cluster01-worker
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  storage.k8s.io/v1
Kind:         CSINode
Metadata:
  Creation Timestamp:  2020-03-27T15:19:01Z
  Owner References:
    API Version:     v1
    Kind:            Node
    Name:            cluster01-worker
    UID:             85af82eb-0b55-45f7-8b18-12ed20ca9b40
  Resource Version:  480
  Self Link:         /apis/storage.k8s.io/v1/csinodes/cluster01-worker
  UID:               05ebab79-89e5-44d9-b04d-b80a70741002
Spec:
  Drivers:  <nil>
Events:     <none>
```

Rysunek 4.9. Informacje szczegółowe o wybranym węźle

Zwróć uwagę na sekcję `Spec` wygenerowanych danych wyjściowych. Zawiera ona szczegóły dotyczące wszelkich sterowników, które mogły zostać zainstalowane w celu obsługi systemów

pamięci masowej backendu. W omawianym przykładzie nie mamy żadnych takich systemów, więc nasz klaster nie wymaga dodatkowych sterowników.

Aby pokazać przykład informacji o węźle, na rysunku 4.10 zaprezentowaliśmy dane wyjściowe wygenerowane w klastrze, w którym są zainstalowane dwa sterowniki przeznaczone do obsługi dwóch odmiennych rozwiązań pamięci masowej.

```
Name:         home-k8s-master3.k8shome.com
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  storage.k8s.io/v1beta1
Kind:         CSINode
Metadata:
  Creation Timestamp:  2019-10-29T12:38:05Z
  Owner References:
    API Version:     v1
    Kind:            Node
    Name:            home-k8s-master3.k8shome.com
    UID:             c7131855-e465-49e7-a700-debfcccc2ef2
  Resource Version:  28926386
  Self Link:         /apis/storage.k8s.io/v1beta1/csinodes/home-k8s-master3.k8shome.com
  UID:               7487170e-d069-4b59-9589-8f12c9f6a98a
Spec:
  Drivers:
    Name:           csi.trident.netapp.io
    Node ID:        home-k8s-master3.k8shome.com
    Topology Keys:  <nil>
    Name:           reduxio.magellan
    Node ID:        home-k8s-master3.k8shome.com
    Topology Keys:  <nil>
Events:             <none>
```

Rysunek 4.10. Przykład pokazujący zainstalowane sterowniki obsługi systemów pamięci masowej

Jeżeli spojrzysz na sekcję `spec.drivers` tego węzła, zauważysz dwie odmienne podsekcje `Name`. Pierwsza dotyczy zainstalowanego sterownika przeznaczonego do obsługi systemu NetApp SolidFire, podczas gdy druga zapewnia obsługę systemu pamięci masowej Reduxio.

Sterowniki pamięci masowej

Jak wcześniej wspomnieliśmy, klaster KinD nie zawiera zainstalowanych żadnych dodatkowych sterowników obsługi systemów pamięci masowej. Po użyciu polecenia `kubectl get csidrivers` wygenerowana lista nie zawiera żadnych zasobów API.

Klasy pamięci masowej w KinD

W celu dołączenia do dowolnej pamięci masowej oferowanej przez klaster wymagany jest obiekt `StorageClass`. Opracowany przez firmę Rancher dostawca tworzy domyślną klasę pamięci masowej. Działa ona w charakterze domyślnej klasy `StorageClass`, więc w żądaniach PVC nie trzeba podawać nazwy klasy `StorageClass`. Jeżeli domyślna klasa `StorageClass` nie zostanie zdefiniowana, każde żądanie PVC będzie musiało zawierać nazwę klasy `StorageClass`.

Ponadto, jeśli klasa domyślna będzie niedostępna, a żądanie PVC nie zawiera nazwy klasy `StorageClass`, alokacja PVC zakończy się niepowodzeniem, ponieważ serwer API nie będzie w stanie powiązać żądania z klasą `StorageClass`.

> W klastrze produkcyjnym za dobrą praktykę uznaje się pominięcie przypisania domyślnej klasy `StorageClass`. W zależności od użytkowników mogą istnieć wdrożenia, w których klasa nie została zdefiniowana, a domyślny system pamięci masowej okaże się nie pasować do wymagań danego wdrożenia. Taki problem może nie istnieć przed wdrożeniem rozwiązania w środowisku produkcyjnym, a będzie miał wpływ na zyski i reputację firmy. Jeżeli nie przypiszesz domyślnej klasy pamięci masowej, programista zobaczy zakończone niepowodzeniem żądanie PVC i odkryje problem na długo przed tym, zanim stanie się kosztowny dla firmy.

Aby wyświetlić dostępne klasy pamięci masowej w klastrze, należy skorzystać z polecenia `kubectl get storageclasses` lub użyć jego skróconej wersji z `sc` zamiast `storageclasses`. Na rysunku 4.11 pokazaliśmy przykładowe dane wyjściowe tego polecenia.

```
NAME                 PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
standard (default)   rancher.io/local-path   Delete          WaitForFirstConsumer   false                  66m
```

Rysunek 4.11. Wyświetlona lista domyślnych klas pamięci masowej

Przejdziemy teraz do sposobu pracy z komponentem przygotowującym pamięć masową.

Używanie komponentu KinD przygotowującego pamięć masową

Użycie dostarczonego komponentu jest bardzo łatwe. Skoro potrafi on automatycznie przygotować pamięć masową i jest zdefiniowany jako klasa domyślna, to wszystkie żądania PVC będą obsługiwane przez poda provisioningu, który następnie utworzy obiekty `PersistentVolume` i `PersistentVolumeClaim`.

Aby pokazać ten proces, omówimy niezbędne kroki. Na rysunku 4.12 zaprezentowaliśmy dane wyjściowe wygenerowane po użyciu poleceń `kubectl get pv` i `kube get pvc` w bazowym klastrze KinD.

```
[root@localhost yaml]# kubectl get pv
No resources found in default namespace.
[root@localhost yaml]# kubectl get pvc --all-namespaces
No resources found
```

Rysunek 4.12. Przykłady użycia PV i PVC

Pamiętaj, że `PersistentVolume` to nie jest obiekt w przestrzeni nazw, więc do polecenia nie trzeba dodawać opcji zawierającej przestrzeń nazw. Z kolei PVC to obiekty stosujące przestrzenie nazw, zatem w omawianym przykładzie nakazaliśmy Kubernetes wyświetlenie obiektów PVC dostępnych we wszystkich przestrzeniach nazw. Ponieważ mamy do czynienia z nowym klastrem, a żadne z zadań nie wymagało trwałej pamięci masowej, to nie istnieją jeszcze żadne obiekty PV lub PVC.

Jeżeli nie jest używany komponent automatycznie przygotowujący pamięć masową, to trzeba będzie utworzyć obiekt PV, zanim obiekt PVC będzie mógł żądać dostępu do woluminu. W klastrze mamy działający komponent firmy Rancher, więc proces tworzenia pamięci masowej można przetestować przez wdrożenie poda z żądaniem PVC, np. tak jak pokazaliśmy w kolejnym fragmencie kodu.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
storage: 1Mi
---
kind: Pod
apiVersion: v1
metadata:
  name: test-pvc-claim
spec:
  containers:
  - name: test-pod
    image: busybox
    command:
      - "/bin/sh"
    args:
      - "-c"
      - "touch /mnt/test && exit 0 || exit 1"
    volumeMounts:
      - name: test-pvc
        mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
    - name: test-pvc
      persistentVolumeClaim:
        claimName: test-claim
```

Żądanie PVC nosi nazwę `test-claim`, dotyczy domyślnej przestrzeni nazw i woluminu o pojemności 1 MB. Konieczne jest dołączenie opcji `StorageClass`, ponieważ platforma KinD zdefiniowała domyślną klasę `StorageClass` dla klastra.

W celu utworzenia żądania PVC można zastosować polecenie `create` w narzędziu `kubectl`, np. `kubectl create -f pvctest.yaml`, po którego wykonaniu Kubernetes zgłosi utworzenie

obiektu PVC. W tym miejscu trzeba jednak koniecznie dodać, że to nie oznacza istnienia w pełni działającego obiektu PVC. Wprawdzie obiekt PVC został utworzony, ale jeśli w żądaniu PVC zabraknie jakichkolwiek zależności, to obiekt będzie utworzony, przy czym próba pełnego wykonania żądania PVC zakończy się niepowodzeniem.

Po utworzeniu obiektu PVC jego stan można sprawdzić za pomocą jednego z dwóch dostępnych poleceń. Pierwsze to polecenie `get` używane następująco: `kubectl get pvc`. Ponieważ żądanie dotyczy domyślnej przestrzeni nazw, nie trzeba jej podawać w poleceniu. (Zauważ, że skróciliśmy nazwę woluminu, aby zmieściła się na stronie książki).

```
NAME          STATUS   VOLUME                    CAPACITY   ACCESS MODES
STORAGECLASS   AGE
test-claim    Bound    pvc-9c56cf65-d661-49e3-     1Mi           RWO       standard
2s
```

Doskonale wiemy o utworzeniu żądania PVC w manifeście, ale nie utworzyliśmy żądania PV. Jeżeli teraz przeanalizujemy obiekt PV, to okaże się, że na skutek wykonania naszego żądania PVC powstał pojedynczy obiekt PV. Także tym razem skróciliśmy nazwę obiektu, aby zmieścił się na stronie książki.

```
NAME                      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
pvc-9c56cf65-d661-49e3-      1Mi          RWO       Delete           Bound
default/test-claim
```

W ten sposób dotarliśmy do końca sekcji dotyczącej pamięci masowej w KinD.

Ponieważ sporo zadań wymaga dostępu do trwałej pamięci masowej, bardzo duże znaczenie ma poznanie sposobu, w jaki Kubernetes integruje te zadania z systemami pamięci masowej. W tej sekcji wyjaśniliśmy, jak KinD dodaje do klastra komponent pozwalający na automatyczne przygotowanie takiego systemu. Wiedzę z zakresu obiektów pamięci masowej w Kubernetes uzupełnisz w następnym rozdziale.

Dodawanie niestandardowego mechanizmu równoważenia obciążenia dla kontrolera Ingress

Materiał zamieszczony w tym podrozdziale dotyczy skomplikowanego tematu dodawania niestandardowego kontenera HAProxy, który można wykorzystać jako mechanizm równoważenia obciążenia węzłów roboczych w klastrze KinD. *Przedstawionych tutaj poleceń nie należy wykonywać w klastrze KinD, który będzie używany w pozostałych rozdziałach.*

Ten podrozdział został napisany z myślą o czytelnikach, którzy chcą się dowiedzieć więcej na temat mechanizmu równoważenia obciążenia między wieloma węzłami roboczymi.

KinD nie oferuje mechanizmu równoważenia obciążenia dla węzłów roboczych. Dołączony kontener HAProxy powoduje jedynie utworzenie pliku konfiguracyjnego dla serwera API. Oficjalnie nie są obsługiwane żadne modyfikacje domyślnego obrazu czy konfiguracji. Skoro w codziennej pracy masz do czynienia z mechanizmami równoważenia obciążenia, to chcieliśmy zamieścić w książce informacje dotyczące konfiguracji niestandardowego kontenera HAProxy w celu zapewnienia mechanizmu równoważenia obciążenia dla trzech węzłów klastra KinD.

Chcemy w tym miejscu wyraźnie podkreślić, że przedstawiona tutaj konfiguracja nie zostanie użyta w pozostałych rozdziałach książki. Przykłady mają być dostępne dla każdego, więc w celu ograniczenia niezbędnych zasobów zawsze będziemy korzystać z utworzonego we wcześniejszej części rozdziału klastra składającego się z dwóch węzłów. Jeżeli chcesz przetestować węzły KinD z mechanizmem równoważenia obciążenia, sugerujemy użycie innego hosta Dockera lub wstrzymanie się do chwili zakończenia lektury książki i usunięcia używanego w niej klastra KinD.

Przygotowanie do instalacji

Przyjęliśmy założenie, że masz do dyspozycji klaster KinD o przedstawionej tutaj konfiguracji:

- dowolna liczba węzłów warstwy sterowania,
- trzy węzły robocze,
- klaster o nazwie `cluster01`,
- działająca wersja **Kindnet** lub **Calico** (CNI),
- zainstalowany kontroler Ingress dla serwera WWW — nasłuchujący na portach 80 i 443 hosta.

Tworzenie konfiguracji klastra KinD

Skoro używasz kontenera HAproxy udostępnionego na portach 80 i 443 hosta Dockera, w pliku konfiguracyjnym nie musisz udostępniać żadnych portów.

Aby ułatwić sobie proces wdrożenia, możesz skorzystać z przedstawionej tutaj przykładowej konfiguracji klastra, która powoduje utworzenie składającego się z sześciu węzłów klastra z wyłączoną obsługą Kindnet.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  apiServerAddress: "0.0.0.0"
disableDefaultCNI: true kubeadmConfigPatches:
-|
  apiVersion: kubeadm.k8s.io/v1beta2
```

```
  kind: ClusterConfiguration
  metadata:
    name: config
  networking:
    serviceSubnet: "10.96.0.1/12"
    podSubnet: "192.168.0.0/16"
nodes:
- role: control-plane
- role: control-plane
- role: control-plane
- role: worker
- role: worker
- role: worker
```

Konieczne jest zainstalowanie oprogramowania Calico za pomocą tego samego manifestu, który został użyty we wcześniejszej części rozdziału. Po zainstalowaniu Calico kolejnym zadaniem jest instalacja kontrolera Ingress serwera WWW NGINX — odpowiednia procedura została zamieszczona wcześniej w rozdziale.

Po wdrożeniu Calico i NGINX otrzymasz działający klaster bazowy. Teraz możesz przejść do wdrożenia niestandardowego kontenera HAProxy.

Wdrażanie niestandardowego kontenera HAProxy

HAProxy oferuje umieszczony w serwisie Docker Hub kontener, który jest łatwy do wdrożenia, a którego uruchomienie wymaga tylko jednego pliku konfiguracyjnego.

By utworzyć plik konfiguracyjny, trzeba znać adresy IP poszczególnych węzłów roboczych klastra. W materiałach przygotowanych dla książki umieściliśmy plik skryptu, którego działanie polega na wyszukaniu za Ciebie niezbędnych informacji, utworzeniu pliku konfiguracyjnego i uruchomieniu kontenera HAProxy. Ten plik nosi nazwę *HAProxy-ingress.sh* i znajduje się w katalogu *Rozdzial04/HAProxy* w materiałach przygotowanych dla książki.

Aby pomóc Ci w zrozumieniu sposobu działania tego skryptu, omówimy tworzące go poszczególne sekcje. Blok kodu znajdujący się na początku skryptu pobiera adresy IP wszystkich węzłów roboczych klastra i zapisuje je w zmiennych. Te dane będą potrzebne na późniejszym etapie działania skryptu.

```
#!/bin/bash

worker1=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}' cluster01-
↪worker)
worker2=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}' cluster01-
↪worker2)
worker3=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}' cluster01-
↪worker3)
```

Następnie, skoro podczas uruchamiania kontenera będziemy korzystać z dołączania punktu montowania, konieczne jest umieszczenie pliku konfiguracyjnego w doskonale znanym położeniu. Zdecydowaliśmy się na podkatalog o nazwie *HAProxy* w katalogu domowym użytkownika.

```
# Utworzenie podkatalogu HAProxy w katalogu domowym użytkownika.
mkdir ~/HAProxy
```

Kolejnym krokiem jest utworzenie pliku konfiguracyjnego w nowym katalogu.

```
# Utworzenie pliku HAProxy.cfg dla węzłów roboczych.
tee ~/HAProxy/HAProxy.cfg <<EOF
```

Sekcja `global` konfiguracji powoduje zdefiniowanie dotyczących bezpieczeństwa i wydajności ustawień dla całego procesu.

```
global
  log /dev/log local0
  log /dev/log local1 notice
  daemon
```

Z kolei sekcja `defaults` jest używana do skonfigurowania wartości, które będą miały zastosowanie dla wszystkich sekcji frontendu i backendu.

```
defaults
  log global
  mode tcp
  timeout connect 5000
  timeout client 50000
  timeout server 50000

frontend workers_https
  bind *:443
  mode tcp
  use_backend ingress_https

backend ingress_https
  option httpchk GET /healthz
  mode tcp
  server worker $worker1:443 check port 80
  server worker2 $worker2:443 check port 80
  server worker3 $worker3:443 check port 80
```

Te polecenia nakazują HAProxy utworzenie frontendu o nazwie `workers_https`, a także zdefiniowanie, że adresy IP i porty dołączane do żądań przychodzących mają używać trybu TCP i backendu o nazwie `ingress_https`.

Backend `ingress_https` zawiera trzy węzły robocze używające portu 443. Opcja `check port 80` powoduje przetestowanie portu 80. Jeżeli serwer udzieli odpowiedzi na porcie 80, zostanie dodany jako cel dla żądań. Wprawdzie istnieje reguła dla portu 443 protokołu HTTPS, ale do sprawdzenia odpowiedzi pochodzącej z poda serwera WWW NGINX używamy jedynie portu 80.

```
frontend workers_http
  bind *:80
  use_backend ingress_http

backend ingress_http
  mode http
  option httpchk GET /healthz
```

```
    server worker $worker1:80 check port 80
    server worker2 $worker2:80 check port 80
    server worker3 $worker3:80 check port 80
```

Działanie sekcji `frontend` polega na utworzeniu frontendu akceptującego ruch sieciowy HTTP przychodzący do portu 80. Następnie jako punkty końcowe jest używana lista serwerów backendu, `ingress_http`. Podobnie jak w przypadku sekcji HTTPS, także tutaj wykorzystujemy port 80 do sprawdzenia pod kątem wszystkich węzłów nasłuchujących na podanym porcie. Każdy punkt końcowy odpowiadający na to żądanie zostanie dodany jako cel dla ruchu HTTP. Węzły nieposiadające uruchomionego serwera WWW NGINX nie udzielą odpowiedzi, więc nie zostaną dodane jako cel żądań.

```
EOF
```

Na tym kończymy omówienie pliku. Ostateczny plik zostanie utworzony w katalogu *HAProxy*:

```
# Uruchomienie kontenera HAProxy dla węzłów roboczych.
$ docker run --name HAProxy-workers-lb -d -p 80:80 -p 443:443 -v ~/HAProxy:/
↪usr/local/etc/HAProxy:ro HAProxy -f /usr/local/etc/HAProxy/HAProxy.cfg
```

Ostatnim krokiem jest uruchomienie kontenera Dockera z HAProxy wykorzystującym wcześniej utworzony plik konfiguracyjny, który definiuje trzy węzły robocze, udostępniane na portach 80 i 443 hosta Dockera.

W ten sposób wyjaśniliśmy temat instalacji niestandardowego mechanizmu równoważenia obciążenia HAProxy dla węzłów roboczych. Możemy przejść do analizy sposobu działania konfiguracji.

Przepływ ruchu sieciowego HAProxy

Nasz przykładowy klaster ma w sumie osiem działających kontenerów. Sześć z nich to standardowe komponenty Kubernetes, czyli trzy serwery warstwy sterowania i trzy węzły robocze. Dwa pozostałe kontenery to standardowy serwer HAProxy w KinD i niestandardowy kontener HAProxy, jak pokazaliśmy na rysunku 4.13.

```
IMAGE                                  PORTS                                       NAMES
haproxy                                0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp    haproxy-workers-lb
kindest/haproxy:2.1.1-alpine           0.0.0.0:32776->6443/tcp                     cluster01-external-load-balance

kindest/node:v1.17.0                                                               cluster01-worker
kindest/node:v1.17.0                   127.0.0.1:32801->6443/tcp                   cluster01-control-plane
kindest/node:v1.17.0                   127.0.0.1:32799->6443/tcp                   cluster01-control-plane3
kindest/node:v1.17.0                                                               cluster01-worker2
kindest/node:v1.17.0                   127.0.0.1:32800->6443/tcp                   cluster01-control-plane2
kindest/node:v1.17.0                                                               cluster01-worker3
```

Rysunek 4.13. Potwierdzenie działania niestandardowego kontenera HAProxy

Istnieje kilka różnic między danymi wyjściowymi, które zostały wygenerowane dla tego klastra, a wygenerowanymi dla składającego się z dwóch węzłów klastra używanego w pozostałych przykładach książki. Zwróć uwagę, że węzły robocze nie zostały udostępnione na żadnych portach komputera gospodarza. Dzięki uruchomieniu nowego serwera HAProxy węzły robocze nie wymagają mapowania. Jeżeli spojrzysz na utworzony kontener HAProxy, jest

udostępniony na portach 80 i 443. To oznacza, że wszystkie żądania przychodzące do portów 80 i 443 hosta zostaną przekierowane do naszego niestandardowego kontenera HAProxy.

Domyślne wdrożenie serwera WWW NGINX ma tylko jedną replikę, więc kontroler Ingress działa w pojedynczym węźle. Jeżeli zajrzysz do dzienników zdarzeń kontenera HAProxy, zauważysz coś interesującego:

```
[NOTICE] 093/191701 (1) : New worker #1 (6) forked
[WARNING] 093/191701 (6) : Server ingress_https/worker is DOWN,
reason: Layer4 connection problem, info: "SSL handshake failure
(Connection refused)", check duration: 0ms. 2 active and 0
backup servers left. 0 sessions active, 0 requeued, 0 remaining
in queue.
[WARNING] 093/191702 (6) : Server ingress_https/worker3 is
DOWN, reason: Layer4 connection problem, info: "SSL handshake
failure (Connection refused)", check duration: 0ms. 1 active
and 0 backup servers left. 0 sessions active, 0 requeued, 0
remaining in queue.
[WARNING] 093/191702 (6) : Server ingress_http/worker is DOWN,
reason: Layer4 connection problem, info: "Connection refused",
check duration: 0ms. 2 active and 0 backup servers left. 0
sessions active, 0 requeued, 0 remaining in queue.
[WARNING] 093/191703 (6) : Server ingress_http/worker3 is DOWN,
reason: Layer4 connection problem, info: "Connection refused",
check duration: 0ms. 1 active and 0 backup servers left. 0
sessions active, 0 requeued, 0 remaining in queue.
```

W dziennikach zdarzeń mogą się znajdować informacje o kilku błędach, np. dotyczących SSL i nieudanego połączenia. Wprawdzie te komunikaty wyglądają jak błędy, ale w rzeczywistości dotyczą nieudanych zdarzeń sprawdzenia w węzłach roboczych. Nie zapominaj, że serwer WWW NGINX działa w tylko jednym podzie, a ponieważ w konfiguracji backendu HAProxy są wymienione trzy węzły, sprawdzane będą porty w nich wszystkich. Jeżeli węzeł nie udzieli odpowiedzi, nie będzie użyty przez mechanizm równoważenia obciążenia. W naszej aktualnej konfiguracji to i tak bez znaczenia — serwer WWW NGINX mamy w tylko jednym węźle. Jednak mamy zapewnioną wysoką dostępność kontrolera Ingress.

Jeżeli dokładnie przeanalizujesz dane wyjściowe, zobaczysz, ile serwerów jest aktywnych w zdefiniowanym backendzie, np.:

```
check duration: 0ms. 1 active and 0 backup servers left.
```

Każda pula serwerów w danych wyjściowych pokazuje jeden aktywny punkt końcowy, więc wiemy, że serwer HAProxy z powodzeniem odnalazł kontrolera NGINX na portach 80 i 443.

Do ustalenia, do którego węzła roboczego został podłączony serwer HAProxy, można użyć znajdujących się w dziennikach zdarzeń informacji o nieudanych połączeniach. Każdy backend będzie zawierał listę nieudanych połączeń. Przykładowo wiemy o działaniu węzła `cluster01-`
`↪worker2`, ponieważ w dziennikach zdarzeń znajdują się informacje, że stan dwóch pozostałych węzłów to `DOWN`:

```
Server ingress_https/worker is DOWN
Server ingress_https/worker3 is DOWN
```

Zasymulujemy teraz awarię węzła, aby sprawdzić, czy HAProxy faktycznie zapewnia wysoką dostępność NGINX.

Symulowanie awarii kubeletu

Zapewne pamiętasz, że węzły KinD są tymczasowe i zatrzymanie dowolnego kontenera może uniemożliwić jego ponowne uruchomienie. W jaki sposób więc symulować awarię węzła roboczego, skoro nie można zatrzymać kontenera?

Aby symulować awarię, można zatrzymać usługę kubeletu w węźle. To spowoduje poinformowanie `kube-apiserver` o braku możliwości wykorzystania dodatkowych podów w węźle. W naszym przykładzie chcemy sprawdzić, czy serwer HAProxy zapewnia obsługę wysokiej dostępności dla serwera WWW NGINX. Ponieważ wiemy, że działa kontener `worker2`, to jego musimy „uszkodzić".

Najłatwiejszym sposobem na zatrzymanie kubeletu jest wykonanie polecenia `docker exec` w kontenerze:

```
$ docker exec cluster01-worker2 systemctl stop kubelet
```

To polecenie nie spowoduje wygenerowania żadnych danych wyjściowych. Jeżeli jednak odczekasz kilka minut, do momentu otrzymania przez klaster informacji o stanie węzła, to będzie można potwierdzić jego awarię poprzez analizę listy węzłów:

```
$ kubectl get nodes
```

To polecenie spowoduje wygenerowanie danych wyjściowych pokazanych na rysunku 4.14.

```
NAME                       STATUS     ROLES    AGE   VERSION
cluster01-control-plane    Ready      master   45m   v1.17.0
cluster01-control-plane2   Ready      master   45m   v1.17.0
cluster01-control-plane3   Ready      master   43m   v1.17.0
cluster01-worker           Ready      <none>   43m   v1.17.0
cluster01-worker2          NotReady   <none>   43m   v1.17.0
cluster01-worker3          Ready      <none>   43m   v1.17.0
```

Rysunek 4.14. Węzeł worker2 jest w stanie NotReady (niegotowy)

To stanowi potwierdzenie udanej symulacji awarii kubeletu i niedziałania węzła `worker2` (stan `NotReady`).

Wszystkie pody działające przed „awarią" kubeletu nadal będą działać, ale `kube-scheduler` nie będzie w stanie uruchamiać kolejnych zadań w węźle aż do chwili usunięcia problemu z kubeletem. Skoro wiemy, że pod nie zostanie ponownie uruchomiony w węźle, możemy usunąć poda, aby nie został przesunięty do innego węzła.

Potrzebujemy nazwy poda, a następnie możemy go usunąć i tym samym wymusić jego ponowne uruchomienie.

```
$ kubectl get pods -n ingress-nginx nginx-ingress-controller-7d6bf88c86-r7ztq
$ kubectl delete pod nginx-ingress-controller-7d6bf88c86-r7ztq -n ingress-nginx
```

To wymusi na `kube-scheduler` uruchomienie kontenera w innym węźle roboczym. Jednocześnie kontener HAProxy uaktualni listę backendu, ponieważ kontroler NGINX został przeniesiony do innego węzła roboczego.

Jeżeli ponownie zajrzysz do dzienników zdarzeń HAProxy, to zobaczysz uaktualnioną listę backendu HAproxy, która teraz zawiera `cluster01-worker3` i nie zawiera `cluster01-worker2` na liście aktywnych serwerów.

```
[WARNING] 093/194006 (6) : Server ingress_https/worker3 is
UP, reason: Layer7 check passed, code: 200, info: "OK", check
duration: 4ms. 2 active and 0 backup servers online. 0 sessions
requeued, 0 total in queue.
[WARNING] 093/194008 (6) : Server ingress_http/worker3 is UP,
reason: Layer7 check passed, code: 200, info: "OK", check
duration: 0ms. 2 active and 0 backup servers online. 0 sessions
requeued, 0 total in queue.
[WARNING] 093/195130 (6) : Server ingress_http/worker2 is DOWN,
reason: Layer4 timeout, check duration: 2000ms. 1 active and 0
backup servers left. 0 sessions active, 0 requeued, 0 remaining
in queue.
[WARNING] 093/195131 (6) : Server ingress_https/worker2 is
DOWN, reason: Layer4 timeout, check duration: 2001ms. 1 active
and 0 backup servers left. 0 sessions active, 0 requeued, 0
remaining in queue.
```

Jeżeli planujesz użycie tego klastra do wykonania dodatkowych testów, musisz ponownie uruchomić kubelet w `cluster01-worker2`. Jeśli natomiast chcesz usunąć ten klaster, możesz po prostu użyć polecenia KinD usuwającego klaster, co spowoduje usunięcie także wszystkich węzłów.

Usunięcie kontenera HAProxy

Po usunięciu klastra KinD trzeba będzie ręcznie usunąć dodany wcześniej kontener HAProxy. Ponieważ platforma KinD nie utworzyła naszego mechanizmu równoważenia obciążenia, usunięcie klastra nie spowoduje usunięcia kontenera mechanizmu równoważenia obciążenia. Aby usunąć niestandardowy kontener HAProxy, należy zastosować polecenie `docker rm`:

```
$ docker rm HAProxy-workers-lb –force
```

To spowoduje zatrzymanie kontenera i jego usunięcie z listy Dockera. W ten sposób uzyskasz możliwość ponownego uruchomienia w przyszłości kontenera o takiej samej nazwie dla nowego klastra KinD.

Podsumowanie

W rozdziale omówiliśmy projekt Kubernetes SIG o nazwie KinD. Dość dokładnie wyjaśniliśmy procedurę instalacji komponentów dodatkowych w klastrze KinD, m.in. Calico jako CNI i NGINX jako kontrolera Ingress. Przedstawiliśmy również szczegóły dotyczące obiektów pamięci masowej w Kubernetes, które są dostępne w klastrze KinD.

Mamy nadzieję, że dzięki lekturze tego rozdziału rozumiesz potężne możliwości, jakie Twojej organizacji może przynieść użycie KinD. To rozwiązanie oferuje łatwy do wdrożenia i w pełni konfigurowalny klaster Kubernetes. Liczba klastrów działających w pojedynczym hoście jest teoretycznie ograniczona jedynie przez zasoby dostępne w komputerze gospodarza.

W następnym rozdziale omówimy obiekty Kubernetes. Jego tytuł to *Krótkie wprowadzenie do Kubernetes*, ponieważ zamieściliśmy w nim omówienie większości standardowych obiektów Kubernetes i ich przeznaczenia. Następny rozdział można więc uznać za przewodnik po Kubernetes — znajdziesz tam informacje o obiektach Kubernetes, ich przeznaczeniu i sposobie użycia.

Ten rozdział był dość długi i został przygotowany w charakterze ściągi dla czytelników, którzy mają doświadczenie w pracy z Kubernetes, a także jako wprowadzenie dla dopiero rozpoczynających z nim pracę. Naszym zamierzeniem w książce jest wykroczenie poza standardowe obiekty Kubernetes, ponieważ w wielu obecnie dostępnych opracowaniach bardzo dobrze wyjaśniono podstawy pracy z tą platformą.

Pytania

1. Który obiekt trzeba utworzyć najpierw, zanim będzie można utworzyć obiekt `PersistentVolumeClaim`?
   a) PVC
   b) Dysk
   c) `PersistentVolume`
   d) `VirtualDisk`
2. KinD oferuje dynamiczne przygotowanie dysku. Która firma utworzyła ten komponent?
   a) Microsoft
   b) CNCF
   c) VMware
   d) Rancher

3. Jeżeli utworzysz klaster KinD z wieloma węzłami roboczymi, co musisz jeszcze zainstalować, aby mieć możliwość bezpośredniego kierowania ruchu sieciowego do poszczególnych węzłów?
   a) Mechanizm równoważenia obciążenia
   b) Serwer proxy
   c) Nic
   d) Sieciowy komponent równoważenia obciążenia
4. Klaster Kubernetes może mieć zainstalowany tylko jeden sterownik CSIdriver.
   a) Prawda
   b) Fałsz

Kubernetes i Docker: tak działają systemy najpotężniejszych korporacji!

Technologie Kubernetes i Docker szybko zdobyły zaufanie dużych firm i dziś są standardową infrastrukturą pozwalającą na tworzenie, testowanie i uruchamianie aplikacji. W porównaniu z wcześniej stosowanymi rozwiązaniami wymagają jednak zupełnie innego podejścia do budowy i wdrażania oprogramowania. Oznacza to, że jeśli korporacja chce w pełni skorzystać z potencjału Kubernetesa i Dockera, musi znaleźć osoby dysponujące wiedzą i umiejętnościami pozwalającymi na zintegrowanie klastrów Kubernetes z istniejącymi systemami organizacji.

To książka przeznaczona dla osób, które chcą poszerzyć swoją wiedzę i umiejętności potrzebne do pracy z klastrami. Omówiono tu podstawy dotyczące konteneryzacji, Dockera i Kubernetesa, jednak więcej miejsca poświęcono bardziej zaawansowanym zagadnieniom, między innymi integracji kontenera z platformą chmury czy integracji z takimi narzędziami jak MetalLB, ExternalDNS i OpenID Connect (OICD). Zaprezentowano również zasady stosowania Pod Security Policy (PSP), Open Policy Agent (OPA), Falco i Velero, a także sposób, w jaki przebiega wdrażanie całej platformy w chmurze z użyciem mechanizmów ciągłej integracji i ciągłego wdrażania (CI/CD). Dowiesz się też, jak testować aplikacje i komponenty Kubernetes i jak implementować różne rozwiązania *open source*.

Najciekawsze zagadnienia:

- tworzenie wielowęzłowego klastra Kubernetes za pomocą KinD
- implementacja narzędzi: Ingress, MetalLB i ExternalDNS
- konfiguracja klastra OIDC i uwierzytelnianie w Kubernetesie
- zabezpieczanie i audyty klastrów
- wdrażanie platformy z użyciem projektów: Tekton, GitLab i Argo CD

Scott Surovich — jest głównym inżynierem kontenerów w jednym z największych banków. Wcześniej wprowadzał rozwiązania oparte na Kubernetesie między innymi w Kasten, Reduxio, VMware i Google. Jako jeden z pierwszych otrzymał certyfikat Google — Cloud Certified Fellow: Hybrid Multicloud.

Marc Boorshtein — jest dyrektorem technicznym w firmie Tremolo Security. Specjalizuje się w stosowaniu DevOps i Kubernetesa do automatyzacji infrastruktury bezpieczeństwa. Zdobył certyfikat CKAD.

ISBN 978-83-283-8629-7
9 788328 386297
Cena: 99,00 zł

Packt