Tajniki Kubernetes

Rozwijaj umiejętności orkiestrowania kontenerów w Kubernetes, aby budować, uruchamiać, zabezpieczać i monitorować wielkoskalowe aplikacje rozproszone

Gigi Sayfan

Packt>

Tajniki Kubernetes

Rozwijaj umiejętności orkiestrowania kontenerów w Kubernetes, aby budować, uruchamiać, zabezpieczać i monitorować wielkoskalowe aplikacje rozproszone

Gigi Sayfan

Przekład: Marek Włodarz

APN Promise
Warszawa 2021

Tajniki Kubernetes

Original English language edition © 2020 Packt Publishing

All rights reserved. Authorised translation from the English language edition book
Mastering Kubernetes
ISBN 978-1-83921-125-6, published by Packt Publishing.

© Polish edition by APN PROMISE SA, Warszawa 2021

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiejkolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

Książka ta przedstawia poglądy i opinie autorów. Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń, chyba że zostanie jednoznacznie stwierdzone, że jest inaczej. Ewentualne podobieństwo do jakiejkolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

Wszystkie nazwy handlowe i towarowe występujące w niniejszej publikacji mogą być znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odpowiednich firm odnośnych właścicieli.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.
APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiekolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-450-9 (druk), 978-83-7541-451-6 (ebook)

Przekład: Marek Włodarz

Korekta: Ewa Swędrowska
Skład i łamanie: MAWART Marek Włodarz

O autorze

Gigi Sayfan zawodowo pisze programy od przeszło 20 lat, w tak różnorodnych dziedzinach, jak komunikatory, morfing, procesy sterujące produkcją układów scalonych, wbudowane aplikacje multimedialne dla konsol do gier, uczenie maszynowe inspirowane działaniem mózgu, przeglądarki internetowe, usługi Web dla rozproszonych platform gier trójwymiarowych, czujniki IoT, rzeczywistość wirtualna czy (ostatnio) genetyka – firma, w której aktualnie pracuje, stworzył jeden z najskuteczniejszych i szybko działających testów genetycznych wykrywających wirusa SARS-Cov-2. Pisał kod produkcyjny w wielu językach programowania, takich jak Go, Python, C, C++, C#, Java, Delphi, JavaScript, a nawet Cobol oraz PowerBuilder, dla bardzo różnorodnych systemów operacyjnych, że wymienimy tylko Windows (od wersji 3.11 aż po 7), Linux, macOS, Lynx (wbudowany) i Sony PlayStation. Jego wiedza techniczna obejmuje bazy danych, niskopoziomowe mechanizmy sieciowe, systemy rozproszone, niekonwencjonalne interfejsy użytkownika, DevOps oraz procesy wytwarzania oprogramowania ogólnego zastosowania. Jest również autorem kilku książek oraz setek artykułów i blogów technicznych.

O recenzencie

Onur Yilmaz jest starszym inżynierem oprogramowania w międzynarodowej firmie programistycznej. Posiada certyfikat *Certified Kubernetes Administrator* (*CKA*) i pracuje przy instalacjach Kubernetes i chmurowych systemach zarządzanych. Jest gorliwym zwolennikiem nowatorskich technologii, takich jak Docker, Kubernetes i aplikacje chmurowe. Jest autorem wielu książek, w tym *Introduction to DevOps with Kubernetes*, *Kubernetes Design Patterns and Extensions*, *Serverless Architectures with Kubernetes* oraz *Cloud-Native Continuous Integration and Delivery*. Uzyskał trzy tytuły naukowe w dziedzinie inżynierii oprogramowania.

Spis treści

O autorze iii
O recenzencie iii

Przedmowa 1

Rozdział 1: Poznajemy architekturę Kubernetes 7
Czym jest Kubernetes? 8
Czym Kubernetes nie jest 8
Istota orkiestrowania kontenerów 9
Koncepcje Kubernetes 12
Klastry 12
Węzły 12
Master 14
Pody 14
Etykiety 15
Adnotacje 15
Selektory etykiet 16
Usługi 16
Woluminy 17
Kontrolery replikacji i zestawy replik 17
StatefulSet 18
Sekrety 18
Nazwy 19
Przestrzenie nazw 19
Głębsze zanurzenie w architekturę Kubernetes 19
Wzorce projektowe systemów rozproszonych 20
API Kubernetes 22
Komponenty Kubernetes 26
Mechanizmy wykonawcze w Kubernetes 30
Container Runtime Interface (CRI) 31
Docker 33
rkt 34

CRI-O....35
Kontenery Hyper....35
Ciągła integracja i wdrażanie....36
Projektowanie potoku CI/CD dla Kubernetes....37
Podsumowanie....38

Rozdział 2: Tworzenie klastrów Kubernetes....39

Przegląd....39
Tworzenie jednowęzłowego klastra za pomocą Minikube....40
Poznajemy kubectl....40
Krótkie wprowadzenie do Minikube....41
Przygotowanie....41
W systemie Windows....41
W systemie macOS....42
Tworzenie klastra....43
Rozwiązywanie problemów....44
Sprawdzanie klastra....45
Wykonywanie pracy....46
Badanie klastra za pomocą tablicy kontrolnej....48
Tworzenie wielowęzłowego klastra przy użyciu KinD....49
Krótkie wprowadzenie do KinD....49
Instalowanie KinD....50
Tworzenie klastra w KinD....50
Wykonywanie pracy w KinD....53
Uzyskiwanie dostępu do usług Kubernetes lokalnie poprzez proxy....53
Tworzenie wielowęzłowego klastra przy użyciu k3d....55
Krótkie wprowadzenie do k3s i k3d....55
Instalowanie k3d....56
Tworzenie klastra za pomocą k3d....56
Porównanie Minikube, KinD i k3d....59
Tworzenie klastrów w chmurze (GCP, AWS, Azure)....59
Interfejs cloud-provider....60
GCP....60
AWS....61
Azure....63
Inni dostawcy chmurowi....64
Tworzenie fizycznego klastra od podstaw....66
Przypadki użycia fizycznych maszyn....66

Kiedy należy rozważyć utworzenie fizycznego klatra? ... 66
Istota procesu ... 67
Używanie infrastruktury wirtualnej chmury prywatnej ... 67
Budowanie własnego klastra przy użyciu Kubespray ... 68
Budowanie klastra przy użyciu KRIB ... 68
Budowanie klastra przy użyciu RKE ... 69
Bootkube ... 69
Podsumowanie ... 69
Źródła ... 70

Rozdział 3: Wysoka dostępność i niezawodność ... 71
Koncepcje wysokiej dostępności ... 72
Nadmiarowość ... 72
Wymiana na gorąco ... 72
Wybór lidera ... 73
Inteligentne równoważenie obciążeń ... 73
Idempotencja ... 73
Samonaprawy ... 74
Najlepsze praktyki wysokiej dostępności ... 74
Tworzenie klastrów wysokiej dostępności ... 75
Zapewnianie niezawodności węzłów ... 76
Ochrona stanu klastra ... 77
Ochrona danych ... 81
Uruchamianie nadmiarowych serwerów API ... 82
Realizowanie wyboru lidera w Kubernetes ... 82
Zapewnianie wysokiej dostępności środowiska przejściowego ... 83
Testowanie wysokiej dostępności ... 84
Planowanie wysokiej dostępności, skalowalności i pojemności ... 86
Instalowanie mechanizmu Cluster Autoscaler ... 87
Automatyczne skalowanie podów w pionie ... 88
Aktualizacje klastrów na żywo ... 89
Aktualizacje kroczące ... 90
Wdrożenia blue-green ... 93
Wdrożenia typu canary ... 94
Zarządzanie zmianami w kontraktach danych ... 95
Migrowanie danych ... 96
Wycofywanie przestarzałych API ... 96
Wydajność, koszty i kompromisy projektowe wielkich klastrów ... 97

Wymagania dostępności ... 98
Najlepsze starania ... 98
Okna konserwacji ... 99
Szybkie przywracanie ... 99
Zero przestojów ... 100
Inżynieria niezawodności lokacji ... 102
Wydajność i spójność danych ... 103
Podsumowanie ... 103
Źródła ... 104

Rozdział 4: Zabezpieczanie Kubernetes ... 105

Istota wyzwań zabezpieczeń w Kubernetes ... 106
Wyzwania węzłów ... 106
Wyzwania sieciowe ... 107
Wyzwania dotyczące obrazów ... 109
Wyzwania dotyczące konfiguracji i wdrażania ... 111
Wyzwania dotyczące podów i kontenerów ... 111
Wyzwania organizacyjne, kulturowe i procesowe ... 112
Wzmacnianie Kubernetes ... 114
Istota kont usługowych w Kubernetes ... 114
Uzyskiwanie dostępu do serwera API ... 116
Zabezpieczanie podów ... 123
Zarządzanie zasadami sieciowymi ... 130
Używanie sekretów ... 133
Klastry o wielu dzierżawcach ... 137
Przypadki użycia klastrów o wielu dzierżawcach ... 137
Wykorzystanie przestrzeni nazw dla bezpiecznego rozwiązania wielu dzierżawców ... 138
Unikanie pułapek dotyczących przestrzeni nazw ... 139
Podsumowanie ... 140
Źródła ... 140

Rozdział 5: Praktyczne używanie zasobów Kubernetes ... 141

Projektowanie platformy Hue ... 141
Definiowanie zakresu Hue ... 142
Planowanie przepływów pracy ... 147
Używanie Kubernetes do budowania platformy Hue ... 149
Efektywne używanie kubectl ... 149
Pliki konfiguracji zasobów kubectl ... 150

Wdrażanie długo działających mikrousług w podach ... 152
Separowanie usług wewnętrznych i zewnętrznych ... 157
Wdrażanie wewnętrznej usługi ... 158
Tworzenie usługi hue-reminders ... 159
Eksponowanie usługi na zewnątrz ... 161
Zaawansowane rozmieszczanie ... 163
Selektor węzłów ... 163
Skazy i tolerancje ... 164
Koligacje i antykoligacje węzła ... 166
Koligacja i antykoligacja podu ... 166
Używanie przestrzeni nazw do ograniczania dostępu ... 167
Wykorzystanie kustomize do hierarchizowania struktur klastra ... 169
Podstawy kustomize ... 170
Konfigurowanie struktury katalogów ... 170
Aplikowanie dostosowań kustomize ... 172
Uruchamianie zadań ... 175
Równoległe uruchamianie zadań ... 176
Sprzątanie ukończonych zadań ... 177
Planowanie zadań cron ... 178
Mieszanie nieklastrowych komponentów ... 179
Komponenty poza siecią klastra ... 180
Komponenty wewnątrz sieci klastra ... 180
Zarządzanie platformą Hue przy użyciu Kubernetes ... 180
Używanie sond gotowości do zarządzania zależnościami ... 182
Stosowanie kontenerów inicjujących w celu uporządkowanego podnoszenia podów ... 183
Gotowość podu i bramki gotowości ... 183
Współużytkowanie przy użyciu podów DaemonSet ... 184
Ewoluowanie platformy Hue z pomocą Kubernetes ... 186
Wykorzystanie Hue w przedsiębiorstwie ... 186
Postęp naukowy w Hue ... 186
Kształcenie dzieci ... 186
Podsumowanie ... 187
Źródła ... 187

Rozdział 6: Zarządzanie magazynem ... 189
Przegląd trwałych woluminów ... 190
Woluminy ... 190
Wyposażanie trwałych woluminów ... 196

Tworzenie trwałych woluminów 197
Tworzenie żądań trwałych woluminów 200
Montowanie żądań jako woluminów 203
Surowe woluminy blokowe 204
Klasy magazynowe....206
Pełna demonstracja stosowania trwałych woluminów 207
Typy woluminów magazynów chmur publicznych – GCE, AWS i Azure 213
Amazon EBS 213
Amazon EFS 214
Trwałe dyski GCE....216
Dyski danych Azure 217
Azure Files....218
Woluminy GlusterFS oraz Ceph w Kubernetes 219
Stosowanie GlusterFS 219
Używanie Ceph 222
Flocker jako klastrowy kontenerowy menedżer woluminów danych 225
Integrowanie magazynów klasy przedsiębiorstwa z Kubernetes 227
Rook – nowy gracz na boisku 228
Rzutowanie woluminów 229
Korzystanie z wtyczek woluminów out-of-tree przy użyciu FlexVolume 230
Container Storage Interface....230
Migawki i klonowanie woluminów 232
Podsumowanie 234

Rozdział 7: Uruchamianie aplikacji stanowych w Kubernetes 235
Aplikacje stanowe i bezstanowe w środowisku Kubernetes 235
Wspólne zmienne środowiskowe kontra rekordy DNS dla odkrywania....237
Uruchamianie klastra Cassandra w Kubernetes....243
Podsumowanie 260

Rozdział 8: Instalowanie i aktualizowanie aplikacji 261
Automatyczne skalowanie podów w poziomie 262
Deklarowanie HPA 263
Niestandardowe miary 265
Automatyczne skalowanie przy użyciu kubectl....266
Wykonywanie aktualizacji kroczących przy użyciu autoskalowania 269
Obsługiwanie deficytowych zasobów za pomocą limitów i przydziałów....271
Włączanie przydziałów zasobów 272

Typy przydziałów zasobów 272
Zakresy przydziałów 275
Przydziały zasobów i klasy pierwszeństwa 276
Żądania i limity 276
Posługiwanie się przydziałami 276
Wybieranie i zarządzanie pojemnością klastra 282
Wybieranie typów węzłów 282
Wybieranie rozwiązania magazynowego 283
Kompromis pomiędzy kosztami a czasem odpowiedzi 283
Efektywne używanie wielu konfiguracji węzłów 284
Czerpanie korzyści z elastycznych zasobów chmurowych 285
Uwzględnianie rozwiązań natywnych dla kontenerów 286
Przesuwanie granic wydajności w Kubernetes 288
Podnoszenie wydajności i skalowalności Kubernetes 288
Mierzenie wydajności i skalowalności Kubernetes 291
Testowanie Kubernetes w wielkiej skali 294
Podsumowanie 296

Rozdział 9: Pakowanie aplikacji 297
Czym jest Helm 297
Umotywowanie Helm 298
Architektura Helm 2 298
Komponenty Helm 2 298
Helm 3 299
Używanie Helm 300
Instalowanie Helm 300
Wyszukiwanie schematów 301
Instalowanie pakietów 304
Posługiwanie się repozytoriami 313
Zarządzanie schematami przy użyciu Helm 314
Tworzenie własnych schematów 315
Plik Chart.yaml 316
Pliki metadanych schematu 317
Zarządzanie zależnościami schematu 317
Używanie szablonów i wartości 320
Podsumowanie 327

Rozdział 10: Poznawanie zaawansowanych funkcji sieciowych 329

Model sieciowy Kubernetes 330
Komunikacja wewnątrz podu (między kontenerami) 330
Komunikacja między podami 330
Komunikacja pod-usługa 331
Dostęp z zewnątrz 331
Sieć Kubernetes a sieć Dockera 332
Wyszukiwanie i odkrywanie 334
Wtyczki sieciowe Kubernetes 336
Rozwiązania sieciowe Kubernetes 344
Mostkowanie w klastrach fizycznych 344
Contiv 345
Open vSwitch 345
Nuage Networks VCS 347
Flannel 347
Calico 348
Romana 349
Weave Net 350
Efektywne używanie zasad sieciowych 351
Istota projektu zasad sieciowych Kubernetes 351
Zasady sieciowe i wtyczki CNI 351
Konfigurowanie zasad sieciowych 352
Implementowanie zasad sieciowych 352
Opcje równoważenia obciążeń 353
Zewnętrzny mechanizm równoważący 354
Równoważenie obciążenia dla usługi 357
Ingress 358
Pisanie własnej wtyczki CNI 363
Pierwsze podejście – wtyczka loopback 363
Podsumowanie 372

Rozdział 11: Uruchamianie Kubernetes w wielu chmurach oraz federacja klastrów 375

Historia federacji klastrów w Kubernetes 376
Czym jest federacja klastrów 376
Ważne przypadki użycia dla federacji klastrów 378
Podstawy federacji Kubernetes 380
Warstwa sterowania KubeFed 382

Trudniejsze części ... 383
Zarządzanie federacją klastrów Kubernetes ... 388
Instalowanie kubefedctl ... 388
Tworzenie klastrów ... 389
Konfigurowanie klastra Host ... 390
Rejestrowanie klastrów w federacji ... 391
Posługiwanie się typami federacyjnego API ... 392
Federowanie zasobów ... 393
Używanie pola overrides ... 395
Używanie pola placement do kontrolowania federacji ... 396
Debugowanie błędów propagacji ... 397
Stosowanie zachowań wyższego rzędu ... 397
Wprowadzenie do projektu Gardener ... 402
Terminologia projektu Gardener ... 402
Poznajemy model koncepcyjny rozwiązania Gardener ... 403
Poznajemy architekturę Gardenera ... 404
Zarządzanie stanem klastra ... 404
Rozszerzanie Gardener ... 406
Pierścień Gardener ... 411
Podsumowanie ... 411

Rozdział 12: Przetwarzanie bezserwerowe w Kubernetes ... 413
Istota bezserwerowego przetwarzania ... 413
Uruchamianie długo działających usług w infrastrukturze„bezserwerowej” ... 414
Uruchamianie FaaS w infrastrukturze „bezserwerowej” ... 415
Bezserwerowa platforma Kubernetes w chmurze ... 416
Nie zapominajmy o autoskalowaniu klastra ... 416
Azure AKS oraz Azure Container Instances ... 416
AWS: EKS i Fargate ... 418
Google Cloud Run ... 419
Knative ... 420
Knative Serving ... 421
Knative Eventing ... 427
Próbna jazda z Knative ... 431
Frameworki FaaS w Kubernetes ... 435
Fission ... 435
Kubeless ... 440
Knative oraz riff ... 445
Podsumowanie ... 448

Rozdział 13: Monitorowanie klastrów Kubernetes 449
Obserwowalność 450
Rejestrowanie 450
Metryki 451
Śledzenie rozproszone 452
Raportowanie błędów aplikacji 453
Tablice kontrolne i wizualizacje 453
Alarmowanie 453
Rejestrowanie dzienników 454
Dzienniki kontenerów 454
Dzienniki komponentów Kubernetes 455
Scentralizowane rejestrowanie 456
Wykorzystanie Fluentd do gromadzenia dzienników 459
Gromadzenie metryk w Kubernetes 460
Monitorowanie przy użyciu serwera metryk 461
Przeglądanie klastra przy użyciu tablicy kontrolnej Kubernetes 463
Powstanie Prometheus 464
Rozproszone śledzenie przy użyciu Jaeger 474
Czym jest OpenTracing? 475
Przedstawiamy Jaeger 476
Instalowanie Jaegera 478
Rozwiązywanie problemów 481
Korzystanie ze środowisk pośrednich 481
Wykrywanie problemów na poziomie węzłów 482
Tablice kontrolne kontra alerty 483
Dzienniki kontra metryki kontra raporty błędów 484
Wykrywanie zakłóceń wydajności i źródłowych przyczyn za pomocą śledzenia rozproszonego 485
Podsumowanie 485

Rozdział 14: Korzystanie z Service Mesh 487
Czym jest Service Mesh? 487
Warstwa sterowania i warstwa danych 491
Wybieranie Service Mesh 491
Envoy 491
Linkerd 2 491
Kuma 492
AWS App Mesh 492

Maesh ... 492
Istio ... 492
Dołączanie Istio do klastra Kubernetes ... 493
Architektura Istio ... 493
Przygotowywanie klastra minikube dla Istio ... 496
Instalowanie Istio ... 497
Instalowanie Bookinfo ... 499
Zarządzanie ruchem ... 502
Zabezpieczenia ... 505
Zasady ... 511
Monitorowanie i obserwowalność ... 514
Podsumowanie ... 525

Rozdział 15: Rozszerzanie Kubernetes ... 527

Posługiwanie się API Kubernetes ... 527
Istota OpenAPI ... 528
Konfigurowanie proxy ... 528
Bezpośrednie badanie API Kubernetes ... 529
Tworzenie podu przy użyciu API Kubernetes ... 532
Uzyskiwanie dostępu do API Kubernetes poprzez klienta w Pythonie ... 533
Rozszerzanie API Kubernetes ... 540
Punkty i wzorce rozszerzania Kubernetes ... 540
Wprowadzenie do zasobów niestandardowych ... 545
Opracowywanie definicji zasobów niestandardowych ... 545
Integrowanie niestandardowych zasobów ... 547
Agregowanie serwera API ... 551
Korzystanie z wykazu usług ... 552
Tworzenie wtyczek Kubernetes ... 554
Pisanie niestandardowego schedulera ... 554
Tworzenie wtyczek dla kubectl ... 561
Pułapki związane z wtyczkami kubectl ... 564
Stosowanie webhooków kontroli dostępu ... 565
Używanie webhooka uwierzytelniania ... 566
Używanie webhooka autoryzacji ... 568
Używanie webhooka kontroli wejścia ... 570
Dostarczanie niestandardowych metryk dla autoskalowania podów ... 572
Rozszerzanie Kubernetes przy użyciu niestandardowego magazynu ... 573
Podsumowanie ... 574

Rozdział 16: Przyszłość Kubernetes 577

Dynamika Kubernetes 578

Ważność CNCF 578

Osprzęt 580

Powstanie zarządzanych platform Kubernetes 581

Platformy Kubernetes w chmurach publicznych 581

Instalacje fizyczne, chmury prywatne i KubeEdge 581

Kubernetes jako Platform as a Service (PaaS) 582

Nowe trendy 582

Zabezpieczenia 582

Sieci 583

Niestandardowy sprzęt i urządzenia 584

Service Mesh 584

Przetwarzanie bezserwerowe 585

Kubernetes dla rozwiązań brzegowych 585

Natywne CI/CD 586

Operatory 586

Podsumowanie 587

Źródła 587

Indeks 589

Przedmowa

Kubernetes to system o otwartych źródłach (*open source*), automatyzujący wdrażanie, utrzymywanie, skalowanie i – w ogólności – zarządzanie aplikacjami skonteneryzowanymi. Jeśli ktoś uruchamia więcej niż kilka kontenerów albo chce zautomatyzować zarządzanie tymi kontenerami, potrzebuje Kubernetes. Książka ta poprowadzi Czytelnika poprzez zaawansowane zagadnienia zarządzania klastrami Kubernetes.

Książka rozpoczyna się objaśnieniem fundamentów architektury Kubernetes, po czym przechodzimy do szczegółowego omówienia projektu i struktury systemu. Dowiemy się, jak uruchamiać w nim złożone mikrousługi, zarówno bezstanowe, jak i te z pamięcią stanu. Omówimy takie funkcjonalności, jak autoskalowanie aplikacji w poziomie, aktualizacje kroczące, przydziały zasobów i rozwiązania trwałej pamięci masowej. Pokażemy opcje sieci na praktycznych, rzeczywistych przypadkach użycia i dowiemy się, jak skonfigurować, utrzymywać, zabezpieczać i rozwiązywać problemy w klastrach Kubernetes. Na koniec poznamy takie zaawansowane zagadnienia, jak zasoby niestandardowe (definiowane przez użytkownika), agregowanie API, siatki usług i przetwarzanie bezserwerowe. Cała zawartość jest aktualna i zgodna z wersją Kubernetes 1.18. Po ukończeniu lektury Czytelnik będzie wiedział wszystko, co potrzebne do przejścia z poziomu początkującego na zaawansowany.

Dla kogo jest ta książka

Książka jest przeznaczona dla administratorów systemów i deweloperów, którzy dysponują już wstępną wiedzą na temat Kubernetes i przymierzają się do opanowania jego zaawansowanych funkcjonalności. Niezbędna będzie również przynajmniej podstawowa wiedza na temat sieci.

Zagadnienia omawiane w książce

W rozdziale 1, *Poznajemy architekturę Kubernetes*, zbudujemy wspólnie podstawy niezbędne do wykorzystania całego potencjału Kubernetes. Rozpoczniemy od dowiedzenia się, czym Kubernetes jest i czym *nie jest* oraz co właściwie oznacza orkiestracja kontenerów. Następnie omówimy ważne koncepcje Kubernetes w formie słowniczka, którego będziemy używać w dalszej części książki.

Rozdział 2, *Tworzenie klastrów Kubernetes*, stanowi przejście do ćwiczeń praktycznych. Zbudujemy w nim kilka klastrów przy użyciu rozwiązań minikube, KinD oraz k3d. Omówimy i ocenimy również inne narzędzia, takie jak Kubeadm, Kube-spray, bootkube i stackube. Przyjrzymy się również środowiskom wdrożeniowym, od maszyn fizycznych (*bare metal*) poprzez lokalną serwerownię, aż po chmury publiczne.

Rozdział 3, *Wysoka dostępność i niezawodność*, poświęcony jest zagadnieniu wysoce dostępnych klastrów. Jest to skomplikowane zagadnienie. Twórcy projektu Kubernetes i społeczność nie proponuje „jedynie słusznej" drogi do osiągnięcia wymarzonego celu. Zrealizowanie wysokiej dostępności klastrów Kubernetes obejmuje wiele aspektów, takich jak zagwarantowanie ciągłości działania warstwy sterowania w obliczu (nieuniknionych) awarii, ochrona stanu klastra przechowywanego w etcd, ochrona danych systemu oraz szybkie przywracanie pojemności i/lub wydajności. A wszystko to przy uwzględnieniu faktu, że różne systemy będą miały różne wymagania niezawodności i dostępności.

W rozdziale 4, *Zabezpieczanie Kubernetes*, zajmiemy się ważnym zagadnieniem zabezpieczeń. Klastry Kubernetes to skomplikowane systemy złożone z wzajemnie oddziałujących komponentów działających na wielu warstwach. Izolacja i odseparowanie różnych warstw jest niezwykle ważne przy uruchamianiu kluczowych aplikacji. Aby móc zabezpieczyć system i zagwarantować właściwy (ani zbyt duży, ani za mały) poziom dostępu do zasobów, funkcjonalności i danych, musimy najpierw zrozumieć unikatowe wyzwania, przed którymi stoi Kubernetes jako platforma orkiestracji ogólnego stosowania, mogąca uruchamiać dowolne, nieznane sobie obciążenia (aplikacje). Dopiero wtedy możemy prawdziwie wykorzystać różne mechanizmy zabezpieczeń, izolacji i kontroli dostępu, aby zagwarantować, że sam klaster, aplikacje w nim działające oraz dane są bezpieczne. Omówimy różne najlepsze praktyki oraz to, kiedy odpowiednie jest stosowanie poszczególnych mechanizmów.

Rozdział 5, *Praktyczne używanie zasobów Kubernetes*, poświęcony jest projektowaniu wielkoskalowej platformy, która będzie prawdziwym wyzwaniem dla możliwości i skalowalności Kubernetes. Fikcyjna platforma Hue ma być drogą do utworzenia wszechwiedzącego i wszechmocnego asystenta cyfrowego. Hue ma być cyfrowym rozszerzeniem nas samych. Hue będzie pomagać w robieniu czegokolwiek, wyszukiwania czegokolwiek, a w wielu przypadkach będzie robić mnóstwo rzeczy w naszym imieniu. Co oczywiste, musi przechowywać masę informacji, integrować się z wieloma zewnętrznymi usługami, reagować na powiadomienia i zdarzenia i inteligentnie współpracować z użytkownikiem (czyli z nami).

W rozdziale 6, *Zarządzanie magazynem*, przyjrzymy się temu, jak Kubernetes zarządza pamięcią masową. Magazynowanie danych znacząco różni się od ich przetwarzania (obliczeń), ale na dostatecznie wysokim poziomie jedno i drugie to po prostu

zasoby. Kubernetes jest platformą ogólną, która wykorzystuje abstrakcje konkretnych rozwiązań magazynowych poprzez modele programowe i zbiór wtyczek dla różnych dostawców magazynu.

Rozdział 7, *Uruchamianie aplikacji stanowych w Kubernetes*, pokazuje niełatwe zagadnienie utrzymywania aplikacji z pamięcią stanu. Kubernetes bierze na siebie wiele pracy, automatycznie uruchamiając i wznawiając pody w poszczególnych węzłach klastra, opierajac się na złożonych wymaganiach i konfiguracjach, takich jak przestrzenie nazw, limity i przydziały. Jednak w przypadku podów, w których działa oprogramowanie uzależnione od pamięci masowej (takie jak bazy danych czy kolejki), relokowanie podu mogłoby spowodować zupełną awarię systemu.

W rozdziale 8, *Instalowanie i aktualizowanie aplikacji*, zbadamy zapewniane przez Kubernetes funkcje automatycznego skalowania, wpływu ich na aktualizacje kroczące orz ich interakcje z przydziałami. Zajmiemy się ważnym zagadnieniem wyposażania oraz wyborem i zarządzaniem rozmiarami klastra. Na koniec przespacerujemy się po rozwiązaniach usprawniających wydajność Kubernetes i testowaniu możliwości systemu przy użyciu narzędzia Kubemark.

Rozdział 9, *Pakowanie aplikacji*, prezentuje narzędzie Helm, czyli menedżera pakietów dla Kubernetes. Każda udana i nietrywialna platforma potrzebuje dobrego systemu pakietów. Helm został opracowany przez firmę Deis (zakupioną przez Microsoft 4 kwietnia 2017), a następnie bezpośrednio dołączony do projektu Kubernetes. W roku 2018 stał się projektem CNCF. Rozpoczniemy od poznania celów Helm, jego architektury i jego komponentów.

W rozdziale 10, *Poznawanie zaawansowanych funkcji sieciowych*, będziemy badać ważne zagadnienia pracy sieciowej. Kubernetes, jako platforma orkiestracji, zarządza kontenerami (podami) uruchamianymi na różnych maszynach (fizycznych lub wirtualnych) i wymaga dobrze zdefiniowanego i niezawodnego modelu sieciowego.

W rozdziale 11, *Uruchamianie Kubernetes w wielu chmurach oraz federacja klastrów*, przejdziemy na kolejny poziom i zajmiemy się uruchamianiem Kubernetes w wielu chmurach, wielu klastrach oraz federowaniem klastrów. Klaster Kubernetes to ciasno powiązana jednostka, której wszystkie elementy działają we względnej bliskości i są połączone szybką siecią (typowo w tym samym fizycznym centrum danych albo w tej samej strefie dostępności dostawcy chmurowego). To doskonałe cechy dla wielu przypadków użycia, ale istnieją też takie ważne sytuacje, w których system potrzebuje rozbudowy poza granice pojedynczego klastra.

Rozdział 12, *Przetwarzanie bezserwerowe w Kubernetes*, poświęcony jest badaniu fascynującego świata przetwarzania bezserwerowego w chmurze. Termin „bezserwerowy” zdobywa ostatnio wiele uwagi, ale jest to w istocie błędna nazwa, używana do opisywania dwóch różnych paradygmatów. Prawdziwie bezserwerowa aplikacja

działa jako aplikacja Web w przeglądarce użytkownika lub apce mobilnej, jedynie komunikując się z zewnętrznymi usługami. Typy systemów bezserwerowych, które budujemy na bazie Kubernetes, są inne.

W rozdziale 13, *Monitorowanie klastrów Kubernetes*, zajmujemy się tym, jak zagwarantować, że nasze systemy będą działać z właściwą wydajnością i jak reagować, gdy tak nie jest. W rozdziale 3 omawialiśmy powiązane zagadnienia. Tym razem jednak skupimy się na tym, jak dowiadywać się, co dzieje się w naszym systemie i jakich narzędzi oraz praktyk można używać w tym celu.

Rozdział 14, *Korzystanie z Service Mesh*, zajmuje się nowym podejściem siatek usług i tym, jak pozwalają one na wydzielenie poza kod aplikacji takich krytycznych zagadnień, jak monitorowanie i obserwowalność. Koncepcja siatki usług (Service Mesh) jest prawdziwą zmianą paradygmatu, jeśli chodzi o podejście do projektowania, ewolucję i utrzymywanie systemów rozproszonych. Lubię myśleć o tym jako o *programowaniu aspektowym* chmurowych systemów rozproszonych.

W rozdziale 15, *Rozszerzanie Kubernetes*, wejdziemy głęboko we wnętrzności Kubernetes. Rozpoczniemy od omówienia API Kubernetes i dowiemy się, jak posługiwać się nim programowo, korzystając z bezpośredniego dostępu, klienta w Pythonie i automatyzowania narzędzia kubectl. Następnie przyjrzymy się rozszerzaniu API za pomocą niestandardowych zasobów. Ostatnia część rozdziału poświęcona jest różnym wtyczkom obsługiwanym przez Kubernetes. Wiele aspektów funkcjonowania Kubernetes wykorzystuje podejście modularne i zostało zaprojektowane pod kątem rozszerzania. Zbadamy warstwę agregowania API i wiele typów wtyczek, takich jak niestandardowe schedulery, wtyczki autoryzacyjne, kontroli wejścia, niestandardowe metryki i woluminy. Na koniec zajmiemy się rozszerzaniem kubectl i dodawaniem własnych poleceń.

Rozdział 16, *Przyszłość Kubernetes*, wypełnia przegląd trendów rozwojowych Kubernetes z wielu punktów widzenia. Zaczniemy od dynamiki rozwoju Kubernetes od chwili jego wynalezienia. Spoiler alert: Kubernetes wygrał wojnę pomiędzy systemami orkiestracji kontenerów, miażdżąc oponentów. W miarę rozwoju i uzyskiwania dojrzałości główne pole bitwy przesunęło się z walki z konkurentami do zmagań z własną złożonością Kubernetes. Użyteczność, narzędzia i edukacja będzie odgrywać główną rolę, jako że orkiestrowanie nadal stanowi nową, szybko zmieniającą się i nie tak jeszcze dobrze poznaną dziedzinę. Następnie przyjrzymy się pewnym bardzo interesującym wzorcom i trendom, a na koniec przejrzymy moje przewidywania z 2 wydania książki, a także przedstawię pewne nowe.

Jak uzyskać możliwie dużo korzyści z tej książki

Aby móc realizować przykłady zawarte w każdym rozdziale, potrzebna jest najnowsza wersja Dockera i Kubernetes, najlepiej wersja Kubernetes 1.18. Jeśli używany system operacyjny to Windows 10 Professional, można włączyć tryb hiperwizora; w innych sytuacjach konieczne jest zainstalowanie VirtualBox i użycie Linuksa jako systemu operacyjnego gościa. W przypadku korzystania z macOS (albo Linuksa) jesteśmy od razu przygotowani do działania.

Pobieranie przykładowych plików kodu

Przykładowe pliki kodu dla tej książki można pobrać z repozytorium GitHub pod adresem *https://github.com/PacktPublishing/Mastering-Kubernetes-Third-Edition*. Repozytorium to można sklonować do własnej instalacji Git albo pobrać spakowany (.zip) plik zawierający całość kodu. Po pobraniu pliku należy go rozpakować, używając najnowszej wersji:

- WinRAR / 7-Zip w przypadku Windows
- Zipeg / iZip / UnRarX w macOS
- 7-Zip / PeaZip w systemie Linux

Warto wspomnieć, że oferujemy również inne pakiety kodu dla naszych książek i wideo, dostępne pod adresem *https://github.com/PacktPublishing/*. Warto je sprawdzić!

Stosowane konwencje typograficzne

W książce tej wykorzystywanych jest kilka konwencji, które powinny poprawić czytelność tekstu.

`Kod_w_tekście`: Na przykład „Jeśli wybierzemy HyperKit zamiast VirtualBox, konieczne jest dodanie flagi `--vm-driver=hyperkit` podczas uruchamiania klastra”.

Bloki kodu zostały złamane jak poniżej:

```
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdCluster"
metadata:
  name: "example-etcd-cluster"
spec:
  size: 3
  version: "3.2.13"
```

Gdy chcemy zwrócić uwagę na konkretną część bloku kodu, odpowiednie wiersze lub elementy zostały złożone czcionką wytłuszczoną:

```
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdCluster"
metadata:
  name: "example-etcd-cluster"
spec:
  size: 3
  version: "3.2.13"
```

Polecenia i ich wyjście zostało złożone w następujący sposób:

```
$ k get pods
NAME                    READY   STATUS    RESTARTS   AGE
echo-855975f9c-r6kj8    1/1     Running   0          2m11s
```

Słowo kluczowe: Sygnalizuje nowy termin, ważne pojęcie albo słowa pojawiające się na ekranie, na przykład w menu lub oknach dialogowych.

W taki sposób wyróżniane są dodatkowe informacje lub ważne uwagi.

1
Poznajemy architekturę Kubernetes

Jednym zdaniem, Kubernetes to platforma orkiestracji wdrożeń, skalowania i zarządzania aplikacjami opartymi na kontenerach. Czytelnik zapewne czytał już wiele o Kubernetes, a być może nawet zetknął się z nim w praktyce i używał w jakimś pobocznym projekcie, a nawet w pracy. Jednak zrozumienie, czym naprawdę jest Kubernetes, jak używać go efektywnie i jakie są najlepsze praktyki, wymaga dużo więcej.

Kubernetes to wielki projekt o otwartych źródłach oraz ekosystem zawierający mnóstwo kodu i mnóstwo funkcjonalności. Kubernetes wywodzi się z Google, ale następnie został włączony do **Cloud Native Computing Foundation** (**CNCF**) i obecnie jest niekwestionowanym liderem, jeśli chodzi o aplikacje oparte na kontenerach.

W tym rozdziale przedstawimy podstawy niezbędne do pełnego wykorzystania potencjału Kubernetes. Zaczniemy od wyjaśnienia, czym Kubernetes *jest*, czym *nie jest* i co dokładnie oznacza orkiestracja kontenerów. Następnie zajmiemy się ważnymi koncepcjami Kubernetes, które pozwolą nam zbudować słownik pojęć wykorzystywany w dalszej części tej książki. Następnie zagłębimy się we właściwą architekturę Kubernetes i przyjrzymy się temu, jak umożliwia ona wszystkie funkcjonalności, które udostępnia użytkownikom. Później omówimy różne silniki czasu wykonywania i kontenerów obsługiwane przez Kubernetes (Docker to tylko jedna z możliwych opcji), a na koniec wyjaśnimy rolę Kubernetes w pełnym potoku ciągłej integracji i wdrażania.

Po przeczytaniu tego rozdziału Czytelnik będzie miał solidną wiedzę na temat orkiestrowania kontenerów, problemów rozwiązywanych przez Kubernetes, rozumowania kryjącego się za projektem i architekturą oraz różnych obsługiwanych środowisk

wykonawczych. Zapozna się również z ogólną strukturą repozytoriów open source i będzie gotowy do poszukiwania odpowiedzi na dowolne pytania.

Czym jest Kubernetes?

Kubernetes jest platformą obejmującą wielką liczbę usług i możliwości, które stale się rozrastają. Podstawowa funkcjonalność to rozmieszczanie obciążeń zawartych w kontenerach w naszej infrastrukturze, ale na tym się sprawy nie kończą. Oto kilka innych funkcjonalności, które Kubernetes kładzie na stół:

- Montowanie systemów magazynowania
- Dystrybuowanie sekretów
- Sprawdzanie kondycji i gotowości aplikacji
- Replikowanie wystąpień aplikacji
- Stosowanie automatycznego skalowania podów w poziomie (Horizontal Pod Autoscaler)
- Stosowanie automatycznego skalowania klastra (Cluster Autoscaling)
- Nazywanie i odkrywanie usług
- Równoważenie obciążeń
- Kroczące aktualizacje
- Monitorowanie zasobów
- Odczytywanie i konsumowanie dzienników
- Debugowanie aplikacji
- Zapewnianie uwierzytelniania i autoryzacji

Wszystkie te funkcjonalności omówimy szczegółowo w tej książce. Na razie wystarczy wstępne ich przedstawienie, aby móc docenić, jak dużo Kubernetes może zaoferować naszym systemom.

Zakres funkcjonalności Kubernetes jest imponujący, ale trzeba również zrozumieć, czego Kubernetes nie oferuje.

Czym Kubernetes nie jest

Kubernetes nie jest rozwiązaniem typu **Platform as a Service** (**PaaS**) (platforma jako usługa). Nie narzuca wielu ważnych aspektów, które są pozostawiane nam (jako projektantom) albo innym systemom budowanym na bazie Kubernetes, takim jak Deis, OpenShift i Eldarion. Przykładowo:

- Kubernetes nie wymaga używania określonego typu aplikacji ani środowiska programistycznego

- Kubernetes nie wymaga używania określonego języka programowania
- Kubernetes nie udostępnia baz danych ani kolejek komunikatów
- Kubernetes nie rozróżnia aplikacji od usług
- Kubernetes nie dysponuje „sklepem” usług typu „kliknij, aby wdrożyć”
- Kubernetes nie udostępnia wbudowanych funkcji jako rozwiązań usługowych
- Kubernetes nie narzuca systemu rejestrowania, monitorowania i alertów

Teraz, gdy mamy już jasny obraz tego, jakie są granice Kubernetes, przejdziemy do jego głównej odpowiedzialności – orkiestrowania kontenerów.

Istota orkiestrowania kontenerów

Główna odpowiedzialność spoczywająca na Kubernetes to orkiestrowanie kontenerów. Rozumiemy przez to zagwarantowanie, że wszystkie kontenery realizujące różne obciążenia zostaną rozmieszczone i uruchomione na fizycznych lub wirtualnych maszynach. Kontenery te muszą być efektywnie pakowane, zgodnie z ograniczeniami środowiska wdrożeniowego i konfiguracji klastra. Dodatkowo Kubernetes musi mieć oko na wszystkie uruchomione kontenery i zastępować nowymi te, które zginęły, przestały reagować lub jakkolwiek utraciły kondycję. Kubernetes udostępnia dużo więcej możliwości, które poznamy w następnych rozdziałach. Jednak w tym podrozdziale skupimy się na kontenerach i ich orkiestrowaniu.

Maszyny fizyczne, wirtualne i kontenery

Wszystko zaczyna się i kończy na sprzęcie. Aby uruchomianie naszych obciążeń było w ogóle możliwe, musimy mieć przygotowany jakiś rzeczywisty sprzęt. Obejmuje on faktyczne fizyczne maszyny dysponujące określonymi możliwościami przetwarzania (procesory i rdzenie), pamięcią oraz jakąś lokalną, trwałą pamięcią masową (dyski wirujące albo SSD). Dodatkowo będziemy potrzebować jakiejś współużytkowanej pamięci masowej oraz połączenia tych wszystkich maszyn za pomocą sieci, aby mogły się one porozumiewać ze sobą. W tym miejscu możemy uruchamiać liczne wirtualne maszyny na fizycznych komputerach albo pozostać na poziomie „żelaza” (tylko rzeczywisty sprzęt – żadnych maszyn wirtualnych). Kubernetes można wdrożyć w fizycznym klastrze (*bare-metal*) albo w klastrze maszyn wirtualnych. Następnie będzie mógł orkiestrować zarządzane przez siebie kontenery bezpośrednio na fizycznych komputerach albo na maszynach wirtualnych. W teorii klaster Kubernetes może składać się z mieszanki maszyn fizycznych i wirtualnych, ale nie jest to często spotykane podejście.

Zalety kontenerów

Kontenery reprezentują prawdziwą zmianę paradygmatu w dziedzinie opracowywania i użytkowania wielkich, złożonych systemów oprogramowania. Oto kilka korzyści wynikających ze stosowania kontenerów w stosunku do bardziej tradycyjnych modeli aplikacji:

- Zwinne tworzenie i wdrażanie aplikacji
- Ciągłe rozwijanie, integrowanie i wdrażanie
- Rozdzielenie odpowiedzialności Dev i Ops
- Spójność pomiędzy środowiskami deweloperskim, testowym, przejściowym i produkcyjnym
- Przenośność dystrybucji pomiędzy różnymi chmurami i systemami operacyjnymi
- Zarządzanie skoncentrowane na aplikacji (zależności pakowane są wraz z aplikacją)
- Izolacja zasobów (możliwe jest ograniczenie czasu procesora i pamięci na poziomie kontenera)
- Wykorzystanie zasobów (wiele kontenerów może zostać wdrożonych w tym samym węźle)

Zalety tworzenia i wdrażania oprogramowania opartego na kontenerach są znaczące w wielu kontekstach, ale stają się szczególnie istotne, gdy nasz system wdrażamy w chmurze.

Kontenery w chmurze

Kontenery są idealnym narzędziem do pakowania mikrousług, gdyż, zapewniając izolację tych usług, są jednocześnie bardzo lekkie i nie wprowadzają tak wielkiego narzutu przy wdrażaniu wielu mikrousług, jak w przypadku stosowania maszyn wirtualnych. Sprawia to, że kontenery są doskonałym podejściem dla wdrożeń chmurowych, gdzie alokowanie całej maszyny wirtualnej dla każdej mikrousługi byłoby niezwykle kosztowne.

Wszyscy główni dostawcy chmurowi, tacy jak **Amazon Web Services** (**AWS**), **Google Cloud Platform** (**GCP**) i Microsoft Azure, oferują obecnie usługi hostowania kontenerów. Wiele innych firm również wsiadło do tego pociągu i oferuje zarządzane usługi Kubernetes, w tym:

- IBM IKS
- Alibaba Cloud
- DigitalOcean DKS
- Oracle OKS

- OVH Managed Kubernetes
- Rackspace KaaS

Platforma **Google Kubernetes Engine** (**GKE**) od zawsze opiera się na Kubernetes. Platforma **Elastic Kubernetes Service** (**EKS**) firmy Amazon została dodana do własnego rozwiązania orkiestracji AWS ECS. Usługa kontenerów w Microsoft Azure była pierwotnie oparta na Apache Mesos, ale później również przestawiła się na Kubernetes wraz w wprowadzeniem **Azure Kubernetes Service** (**AKS**). Zawsze było możliwe wdrożenie Kubernetes w każdej z platform chmurowych, ale nie zapewniało to pogłębionej integracji z innymi usługami. Jednak już pod koniec roku 2017 wszyscy czołowi dostawcy chmurowi ogłosili bezpośrednie wsparcie dla Kubernetes. Microsoft uruchomił AKS, AWS udostępnił EKS, a Alibaba Cloud rozpoczął pracę nad sterownikiem menedżera Kubernetes, aby zapewnić gładką integrację Kubernetes.

Bydło czy zwierzęta domowe

W dawniejszych, choć przecież nieodległych czasach, gdy systemy były jeszcze stosunkowo małe, każdy serwer miał nazwę. Deweloperzy i użytkownicy wiedzieli dokładnie, jakie oprogramowanie działa na każdej maszynie. Pamiętam, że w wielu firmach, w których pracowałem, toczyły się wielodniowe dyskusje na temat schematu nazewniczego naszych serwerów. Jednym z popularnych wyborów były postacie z mitologii greckiej lub kompozytorzy. Wszystko to było bardzo emocjonalne. Traktowaliśmy swoje serwery, jak ulubione psy lub koty. Gdy serwer się popsuł, następował poważny kryzys. Wszyscy zrywali się, aby znaleźć gdzieś inny serwer, ustalić, co w ogóle działało na martwym serwerze i jak sprawić, aby zaczęło to znowu działać na nowym. Jeśli serwer utrzymywał jakieś ważne dane, pozostawała nadzieja, że mamy aktualną kopię zapasową i że uda się ją odtworzyć.

Oczywiste jest, że takie podejście nie skaluje się dobrze. Kiedy mamy dziesiątki lub setki serwerów, musimy zacząć je traktować jak zwierzęta hodowlane (bydło), a nie domowych pupilów. Musimy myśleć o nich zbiorczo (stado), a nie indywidualnie. Naturalnie, nadal będziemy mieli kilku pupilów, jak nasze maszyny CI/CD (choć zarządzane rozwiązania CI/CD stają się coraz powszechniejsze), ale serwery Web czy usługi zaplecza to po prostu zwierzęta hodowlane – innymi słowy, bydło.

Kubernetes doprowadza to podejście traktowania serwerów jako bydła do samego skraju, przejmując całą odpowiedzialność za alokowanie kontenerów na określonych maszynach. Przez większość czasu nie występuje potrzeba żadnej interakcji z indywidualnymi maszynami (węzłami). Sprawdza się to najlepiej dla obciążeń bezstanowych. W przypadku aplikacji stanowych (*stateful*) sytuacja jest nieco inna, ale Kubernetes oferuje tu rozwiązanie o nazwie StatefulSet, które omówimy za chwilę.

W tym podrozdziale przedstawiliśmy ideę orkiestrowania kontenerów i omówiliśmy zależności pomiędzy hostami (fizycznymi lub wirtualnymi) a kontenerami, a także zalety uruchamiania kontenerów w chmurze. Zakończyliśmy omówieniem odmiennych podejść, określanych zwykle *bydło kontra zwierzęta domowe*. W kolejnym podrozdziale zaczniemy poznawać świat Kubernetes i uczyć się jego koncepcji i terminologii.

Koncepcje Kubernetes

W tym podrozdziale przedstawię skrótowo wiele ważnych koncepcji Kubernetes, zarówno w kontekście tego, dlaczego są one potrzebne, jak i ich interakcji z innymi koncepcjami. Celem jest opanowanie tych pojęć i pomysłów. Następnie zobaczymy, jak koncepcje te są splatane ze sobą i porządkowane w grupy API i kategorie zasobów, co prowadzi do niesamowitych efektów. Wiele z tych koncepcji można traktować jako cegiełki. Niektóre z nich, takie jak węzły i mastery, są implementowane jako zbiory komponentów Kubernetes. Komponenty te znajdują się na innym poziomie abstrakcji i omówimy je szczegółowo w dedykowanym punkcie w dalszej części tego rozdziału – *Komponenty Kubernetes*.

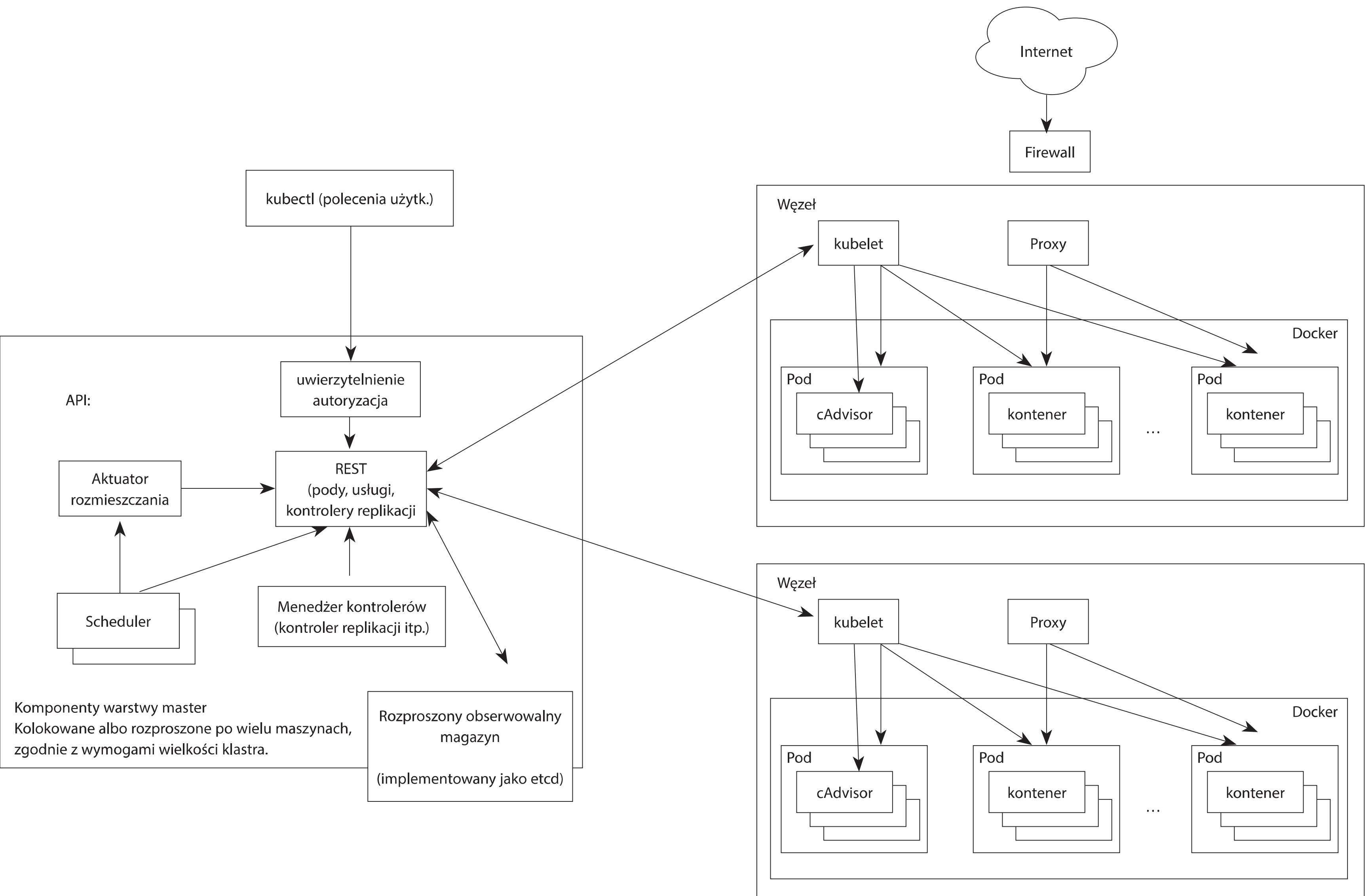
Rysunek 1.1 przedstawia dobrze znany diagram architektury Kubernetes:

Klastry

Klaster (*cluster*) jest zbiorem hostów (węzłów), które udostępniają zasoby obliczeniowe, pamięci, magazynowania i sieci. Kubernetes używa tych zasobów do uruchamiania różnych obciążeń składających się na nasz system. Zwróćmy uwagę, że cały system może składać się z wielu klastrów. Ten zaawansowany przypadek użycia, znany jako federację, omówimy szczegółowo w rozdziale 11, *Uruchamianie Kubernetes w wielu chmurach oraz federacja klastrów.*

Węzły

Węzeł (*node*) to pojedynczy host. Może to być maszyna fizyczna lub wirtualna. Jego zadaniem jest uruchamianie podów. W każdym węźle Kubernetes działa wiele komponentów Kubernetes, takich jak kubelet, mechanizm wykonawczy kontenerów czy kube-proxy. Węzły są zarządzane przez mastera Kubernetes. Jeśli potraktujemy Kubernetes jak ul pszczeli, węzły są robotnicami, dźwigającymi całość obciążenia. W przeszłości były nazywane **minionkami**. Jeśli ktoś zajrzy do starszej dokumentacji lub artykułów, nie powinien dać się zaskoczyć – minionki to po prostu węzły.



Rysunek 1.1: *Diagram architektury Kubernetes*

Master

Master to warstwa sterująca (*control plane*) Kubernetes. Składa się z wielu komponentów, takich jak serwer API, scheduler (dyspozytor) i controller manager (menedżer sterowania). Master jest odpowiedzialny za ogólny stan klastra, rozmieszczanie podów na poziomie klastra oraz obsługę zdarzeń. Zazwyczaj wszystkie komponenty mastera są skonfigurowane na jednym hoście. Jednak przy rozważaniu scenariuszy wysokiej dostępności albo bardzo wielkich klastrów możemy dążyć do uzyskania nadmiarowości mastera. Wysoko dostępne klastry omówimy szczegółowo w rozdziale 4, *Zabezpieczanie Kubernetes*.

Pody

Pod jest jednostką roboczą w Kubernetes. Każdy pod zawiera jeden lub więcej kontenerów. Kontenery w podzie są zawsze rozmieszczane łącznie (zawsze działają na tej samej maszynie). Wszystkie kontenery w podzie mają tę samą przestrzeń adresów IP i portów; mogą komunikować się między sobą przy użyciu localhost albo standardowej łączności międzyprocesowej. Dodatkowo wszystkie kontenery w podzie mogą mieć dostęp do wspólnej pamięci masowej węzła hostującego ten pod. Kontenery nie mają domyślnie dostępu do lokalnej pamięci masowej ani żadnej innej. Woluminy magazynu muszą zostać jawnie zamontowane do każdego kontenera wewnątrz podu. Pody są bardzo ważną funkcjonalnością Kubernetes. Możliwe jest uruchomienie wielu aplikacji wewnątrz jednego kontenera Docker poprzez zastosowanie czegoś w rodzaju nadzorcy jako głównego procesu Docker, który uruchamia wiele procesów, ale taka praktyka jest zwykle źle widziana z następujących powodów:

- **Przezroczystość** Sprawienie, że kontenery wewnątrz podu są widoczne dla infrastruktury powoduje, że możliwe jest udostępnienie usług tym kontenerom, takich jak zarządzanie procesami i monitorowanie zasobów. Zapewnia to użytkownikom wiele korzyści.
- **Oddzielanie zależności oprogramowania** Indywidualne kontenery mogą być wersjonowane, ponownie budowane i wdrażane niezależnie od siebie. Kubernetes pewnego dnia będzie zapewne nawet wspierać aktualizacje na żywo indywidualnych kontenerów.
- **Łatwość użycia** Użytkownicy nie muszą uruchamiać swoich własnych menedżerów procesów, martwić się o propagowanie sygnałów i kodów wyjściowych i tak dalej.
- **Wydajność** Jako że infrastruktura przejmuje więcej odpowiedzialności, kontenery mogą być lżejsze.

Pody stanowią doskonałe rozwiązanie dla zarządzania grupami ściśle powiązanych kontenerów, które są zależne od siebie nawzajem i muszą wspólnie działać na tym samym hoście, aby wykonać swoje zadania. Trzeba tu pamiętać, że pody są uważane za efemeryczne, tymczasowe byty, które mogą zostać odrzucone i zastąpione na życzenie. Dowolna pamięć masowa podu jest niszczona razem z nim. Każdy pod otrzymuje **unikatowy ID (UID)**, zatem nadal możliwe jest ich rozróżnianie w razie potrzeby.

Etykiety

Etykiety (*labels*) są parami klucz-wartość, które służą do grupowania razem zbiorów obiektów – bardzo często podów. Są one ważne dla wielu innych koncepcji, takich jak kontrolery replikacji, zestawów replikacji, wdrożeń i usług, które operują na dynamicznych grupach obiektów i potrzebują możliwości identyfikowania członków grupy. Pomiędzy obiektami i etykietami mamy do czynienia z relacją typu $N \times N$. Każdy obiekt może mieć wiele etykiet i każda etykieta może być stosowana do różnych obiektów. Etykiety z założenia podlegają określonym ograniczeniom. Każda etykieta obiektu musi mieć unikatowy klucz. Klucz etykiety musi stosować się do ściśle zdefiniowanej składni. Zawiera on dwie części: prefiks oraz nazwę, przy czym prefiks jest opcjonalny. Jeśli istnieje, oddzielany jest od nazwy ukośnikiem (/) i musi być poprawną nazwą poddomeny DNS. Prefiks nie może przekroczyć długości 253 znaków. Nazwa jest obowiązkowa i może mieć co najwyżej 63 znaki. Nazwy muszą zaczynać się i kończyć znakami alfanumerycznymi (a-z, A-Z, 0-9) i mogą zawierać tylko te znaki oraz kropki, dywizy (myślniki) i znaki podkreślenia. Wartości podlegają tym samym ograniczeniom, co nazwy. Zauważmy, że etykiety dedykowane są do identyfikowania obiektów, a nie do przypisywania im dowolnych metadanych. Do tego drugiego celu służą adnotacje.

Adnotacje

Adnotacje (*annotations*) pozwalają przypisać do obiektów Kubernetes dowolnie wybrane metadane. Kubernetes po prostu przechowuje adnotacje i sprawia, że zawarte w nich metadane są dostępne. Adnotacje, podobnie jak etykiety, są parami klucz-wartość, gdzie klucz może zawierać opcjonalny prefiks, oddzielony od nazwy klucza znakiem ukośnika (/). Nazwa i prefiks (jeśli istnieje) klucza muszą spełniać ścisłe reguły. Szczegóły można znaleźć w dokumentacji dostępnej pod adresem *https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/#syntax-and-character-set*.

Z mojego doświadczenia wynika, że tego typu metadane są zawsze potrzebne w złożonych systemach. To miło, że twórcy Kubernetes rozumieją tę potrzebę i udostępniają gotowe rozwiązanie, dzięki czemu nie musimy tworzyć własnego magazynu

metadanych i mapowania obiektów na te metadane. Jednak choć adnotacje są użyteczne, ich brak struktury może sprawiać pewne problemy, gdy próbujemy przetwarzać je w ogólny sposób. Stąd często jako alternatywa zalecane są niestandardowe definicje zasobów. Omówimy je później, w rozdziale 15, *Rozszerzanie Kubernetes*.

Selektory etykiet

Selektory etykiet służą do wybierania obiektów na podstawie ich etykiet. Selektory równościowe specyfikują nazwę klucza oraz wartość. Mamy tu dwa operatory, `=` oraz `!=`, odpowiednio dla równości lub nierówności bazującej na wartości, na przykład:

```
role = webserver
```

Taki selektor wybierze wszystkie obiekty, które mają ten klucz (`role`) i tę wartość (`webserver`).

Selektory etykiet mogą zawierać wiele warunków rozdzielanych przecinkami, na przykład:

```
role = webserver, application != foo
```

Selektory oparte na zbiorach rozszerzają te możliwości i umożliwiają wybieranie na podstawie wielu wartości. Na przykład poniższy selektor zostanie spełniony, gdy etykieta `role` będzie miała którąś z podanych wartości :

```
role in (webserver, backend)
```

Usługi

Usługi (*services*) mają na celu eksponowania użytkownikom lub innym usługom pewnej funkcjonalności. Zazwyczaj obejmują grupę podów, zwykle identyfikowane za pomocą – tak, właśnie – etykiety. Możemy mieć usługi zapewniające dostęp do zewnętrznych zasobów albo do podów, które kontrolujemy bezpośrednio na poziomie wirtualnego adresu IP. Natywne usługi Kubernetes są eksponowane poprzez wygodne punkty końcowe. Zwróćmy uwagę, że usługi działają w warstwie 3 stosu sieciowego (TCP/UDP). W wersji Kubernetes 1.2 dodany został obiekt Ingress, który zapewnia dostęp do obiektów HTTP. Więcej na ten temat powiemy w dalszej części rozdziału. Usługi są publikowane i odkrywane za pośrednictwem jednego z dwóch mechanizmów: DNS albo zmiennych środowiskowych. Usługi mogą być równoważone przez Kubernetes. Jednak deweloperzy mogą zdecydować się na samodzielne równoważenie obciążeń w przypadku usług, które wykorzystują zewnętrzne zasoby lub wymagają szczególnego traktowania.

W kontekście usług znajdziemy wiele niełatwych szczegółów dotyczących przestrzeni adresów IP, wirtualnych adresów IP i portów. Omówimy je szczegółowo w rozdziale 10, *Poznawanie zaawansowanych funkcji sieciowych*.

Woluminy

Lokalna pamięć masowa podu jest efemeryczna i znika wraz ze zniszczeniem podu. Niekiedy jest to wystarczające, jeśli jej celem jest tylko wymiana danych pomiędzy kontenerami w danym węźle, ale niekiedy ważne jest, aby dane przeżyły pod albo konieczne jest współdzielenie danych pomiędzy podami. Potrzebę tę zaspokaja koncepcja woluminu (*volume*). Zauważmy, że choć Docker również zawiera koncepcję woluminu, jest ona dość ograniczona (choć zdobywa coraz większe możliwości). Kubernetes używa swoich własnych, oddzielnych woluminów. Kubernetes również wspiera inne mechanizmy wykonawcze kontenerów, zatem nie możemy nawet w teorii bazować na woluminach Dockera.

Istnieje wiele typów woluminów. Kubernetes obecnie bezpośrednio wspiera różnorodne typy woluminów, ale nowe podejście do rozszerzania Kubernetes o większą liczbę typów opiera się na koncepcji **Container Storage Interface** (**CSI**), którą omówimy szczegółowo w dalszej części książki. Wbudowane typy woluminów będą stopniowo wycofywane na rzecz coraz liczniejszych wtyczek dostępnych poprzez CSI.

Kontrolery replikacji i zestawy replik

Kontrolery replikacji (ReplicationController) i zestawy replik (ReplicaSet) zarządzają grupami podów identyfikowanych poprzez selektor etykiet i gwarantują, że w każdym momencie będzie uruchomionych i działających określona liczba podów. Główna różnica pomiędzy nimi polega na tym, że kontrolery replikacji testują członkostwo przy użyciu warunków równościowych, zaś zestawy replik mogą używać selektorów opartych na zbiorach. Zestawy replik są właściwym wyborem, jako że są nadzbiorem kontrolerów replikacji. Można oczekiwać, że kontrolery replikacji zostaną w pewnym momencie uznane za przestarzałe. Kubernetes gwarantuje, że zawsze będzie uruchomiona liczba podów wyspecyfikowana w kontrolerze replikacji albo zestawie replik. Ilekroć liczba ta spadnie z powodu problemów w hostującym podzie albo nieprawności samego podu, Kubernetes uruchomi nowe wystąpienia. Zwróćmy uwagę, że jeśli ręcznie uruchomimy pody i przekroczymy wskazaną liczbę, kontroler replikacji (albo zestaw replik) zabije nadmiarowe pody – choć niekoniecznie te, które właśnie uruchomiliśmy.

Kontrolery replikacji były ośrodkiem wielu typowych przepływów pracy, takich jak kroczące aktualizacje i realizacja zadań jednorazowych. W miarę ewolucji Kubernetes

wprowadzona została bezpośrednia obsługa wielu z takich przepływów w postaci dedykowanych obiektów, takich jak Deployment, Job, CronJob i DaemonSet. Z wszystkimi spotkamy się jeszcze później.

StatefulSet

Pody przychodzą i odchodzą, a jeśli zależy nam na ich danych, możemy użyć trwałej pamięci masowej. To dobrze. Jednak czasami będziemy potrzebowali zarządzać rozproszonym magazynem danych, takim jak Cassandra lub MySQL Galera. Takie klastrowe magazyny utrzymują dane rozproszone pomiędzy unikatowo identyfikowanymi węzłami. Nie możemy tego wymodelować za pomocą zwykłych podów i usługi – w tym miejscu wkraczają obiekty StatefulSet.

Jak Czytelnik pamięta, omawialiśmy rozróżnienie podejść „domowe zwierzęta kontra bydło" i że podejście hodowlane jest właściwą drogą. Jednak StatefulSet znajdują się gdzieś pośrodku. StatefulSet gwarantuje (podobnie jak zestaw replik), że w każdym momencie będzie działać określona liczba wystąpień o unikatowych tożsamościach. Elementy należące do StatefulSet mają następujące właściwości:

- Stabilna nazwa hosta, dostępna w DNS
- Indeks porządkowy
- Stabilna pamięć masowa powiązana z tym indeksem i nazwą hosta

Obiekty StatefulSet pomagają również w odkrywaniu partnerów, podobnie jak w bezpiecznym dodawaniu i usuwaniu członków zestawu.

Sekrety

Sekrety (*secrets*) to niewielkie obiekty, które przechowują poufne informacje, takie jak poświadczenia i tokeny. Są one domyślnie przechowywane w postaci jawnego tekstu w etcd, dostępne dla serwera API Kubernetes i mogą być montowane jako pliki (przy użyciu dedykowanych woluminów sekretów, które są dołączane do zwykłych woluminów danych) w podach, które potrzebują dostępu do nich. Ten sam sekret może być montowany w wielu podach. Kubernetes sam tworzy sekrety dla swoich komponentów, a my możemy tworzyć swoje własne. Inne podejście polega na użyciu sekretów jako zmiennych środowiskowych. Zauważmy, że sekrety w podzie są zawsze przechowywane w pamięci, a nie na dysku (w tmpfs w przypadku montowanych sekretów) dla lepszego bezpieczeństwa.

Nazwy

Każdy obiekt w Kubernetes jest identyfikowany poprzez UID oraz nazwę. Nazwa wykorzystywana jest przy odwoływaniu się do obiektu w wywołaniach API. Nazwy powinny mieć długość do 253 znaków i używać jedynie małych liter, cyfr, dywizu (-) oraz kropki. Jeśli usuniemy obiekt, możemy utworzyć nowy o tej samej nazwie, ale UID muszą być unikatowe w całym życiu klastra. Identyfikatory UID są generowane przez Kubernetes, zatem nie potrzebujemy się nimi zajmować.

Przestrzenie nazw

Przestrzenie nazw (*namespace*) tworzą pewnego rodzaju klastry wirtualne. Możemy mieć pojedynczy klaster fizyczny, który zawiera wiele wirtualnych klastrów, oddzielonych od siebie poprzez przestrzenie nazw. Domyślnie pody w jednej przestrzeni nazw mogą uzyskiwać dostęp do podów i usług w innych przestrzeniach nazw. Jednak w scenariuszach o wielu dzierżawcach, gdy ważna jest całkowita izolacja przestrzeni nazw, można to osiągnąć poprzez odpowiednie zasady sieciowe. Zauważmy, że obiekty węzłów i trwałe woluminy nie należą do żadnych przestrzeni nazw. Kubernetes może rozmieścić pody z innej przestrzeni nazw i uruchomić je w tym samym węźle. Analogicznie pody z różnych przestrzeni nazw mogą używać tej samej trwałej pamięci masowej.

Przy korzystaniu z przestrzeni nazw konieczne jest rozważenie stosowania zasad sieciowych i limitów przydziałów, aby zagwarantować właściwy dostęp i dystrybucję zasobów fizycznego klastra.

Przedstawiliśmy większość podstawowych koncepcji Kubernetes. Istnieje jeszcze kilka innych, o których tylko wspomniałem. W kolejnym podrozdziale będziemy kontynuować naszą wycieczkę po architekturze Kubernetes, przyglądając się motywacjom projektowym, mechanizmom wewnętrznym i implementacji, a nawet sięgniemy do kodu źródłowego.

Głębsze zanurzenie w architekturę Kubernetes

Kubernetes ma bardzo ambitne cele. Jest ukierunkowany na zarządzanie i uproszczenie orkiestracji, wdrażania i zarządzania systemami rozproszonymi w szerokiej gamie środowisk i dostawców chmurowych. Udostępnia wiele możliwości i usług, które powinny działać ponad całą tą różnorodnością, a jednocześnie ewoluując i pozostając dostatecznie proste, aby zwykli śmiertelnicy mogli się nimi posługiwać. To trudne wyzwanie. Kubernetes osiąga to, trzymając się krystalicznie przejrzystego projektu wysokiego poziomu i dobrze przemyślanej architektury, która promuje rozszerzalność

i dołączalność nowych funkcji. Wiele części Kubernetes jest nadal zakodowanych na stałe lub zależnych od środowiska, ale widoczny jest trend refaktoryzacji ich do postaci wtyczek, tak by jądro pozostało niewielkie, ogólne i abstrakcyjne. W tym podrozdziale rozbierzemy Kubernetes jak cebulę, zaczynając od różnych wzorców projektowych systemów rozproszonych i tego, jak są one wspierane w Kubernetes, po czym przejdziemy na powierzchnię Kubernetes, którą stanowi zbiór jego API. Następnie przyjrzymy się rzeczywistym komponentom tworzącym Kubernetes. Na koniec odbędziemy krótką wycieczkę po drzewie kodu źródłowego, aby uzyskać jeszcze lepszy wgląd w strukturę samego systemu Kubernetes.

Na koniec tego podrozdziału Czytelnik powinien opanować solidne podstawy architektury i implementacji Kubernetes, a także zrozumie powody podjęcia określonych decyzji projektowych.

Wzorce projektowe systemów rozproszonych

Parafrazując Tołstoja w *Annie Kareninie*, wszelkie szczęśliwe (działające) systemy rozproszone są podobne do siebie. Oznacza to, że aby funkcjonować poprawnie, wszystkie dobrze zaprojektowane systemy rozproszone muszą stosować się do pewnych najlepszych praktyk i zasad. Kubernetes nie ma na celu być jedynie systemem zarządzania; chce wspierać i umożliwiać te najlepsze praktyki oraz udostępniać usługi wysokiego poziomu dla deweloperów i administratorów. Przyjrzyjmy się zatem niektórym z tych najlepszych praktyk, określanych mianem wzorców projektowych.

Wzorzec przyczepki

Wzorzec przyczepki (*sidecar pattern*) dotyczy kolokacji w podzie innego kontenera obok głównego kontenera aplikacji. Kontener aplikacji nie jest świadomy obecności doczepionego kontenera i po prostu zajmuje się swoimi sprawami. Doskonałym przykładem jest agent scentralizowanego rejestrowania. Główny kontener może po prostu zapisywać zdarzenia do stdout, a kontener doczepiony będzie przechwytywać wszystkie dzienniki i przesyłać je do centralnej usługi rejestrującej, w której zostaną one zagregowane z dziennikami z całego systemu. Korzyści ze stosowania w tym celu doczepionego kontenera zamiast dodawania mechanizmu rejestrowania do głównego kontenera aplikacji są ogromne. Po pierwsze, aplikacje nie są więcej obciążane zadaniami scentralizowanego rejestrowania, co mogłoby być uciążliwe. Jeśli kiedyś zechcemy uaktualnić albo zmienić zasady scentralizowanego rejestrowania albo nawet przełączyć się na zupełnie nowego dostawcę, będziemy musieli tylko uaktualnić kontener doczepiony i go wdrożyć. Nie będą potrzebne żadne zmiany w kontenerach aplikacji, zatem nie ma ryzyka, że przypadkowo uszkodzimy je przy

tej operacji. Service Mesh Istio używa wzorca przyczepki do wstrzykiwania swoich pośredników do każdego podu.

Wzorzec ambasadora

Wzorzec ambasadora (*ambassador pattern*) związany jest z reprezentowaniem usługi zdalnej w taki sposób, jakby była lokalna i potencjalnie wymuszała jakąś zasadę. Dobrym przykładem wzorca ambasadora jest sytuacja, gdy mamy klaster Redis z jednym węzłem nadrzędnym do zapisywania i wieloma replikami do odczytywania. Lokalny kontener ambasadora może służyć jako proxy i eksponować Redis głównemu kontenerowi aplikacji poprzez połączenie localhost. Główny kontener aplikacji po prostu łączy się z Redis poprzez `localhost:6379` (domyślny port Redis), ale w rzeczywistości łączy się z ambasadorem uruchomionym w tym samym podzie, który filtruje żądania i przesyła żądania zapisu do rzeczywistego węzła głównego Redis, a żądania odczytu losowo do jednej z replik do odczytu. Podobnie jak w przypadku wzorca przyczepki, główna aplikacja nie wie, co się odbywa w tle. Może to być bardzo pomocne, gdy testy wykonywane są wobec rzeczywistej, lokalnej instalacji Redis. Ponadto jeśli konfiguracja klastra Redis się zmieni, konieczna będzie tylko modyfikacja ambasadora; główna aplikacja pozostanie w błogiej nieświadomości zmian.

Wzorzec adaptera

Wzorzec adaptera dotyczy standaryzacji wyjścia z głównego kontenera aplikacji. Rozważmy przypadek usługi, która jest stopniowo aktualizowana: może ona generować raporty w formacie, który nie jest zgodny z poprzednią wersją. Inne usługi i aplikacje, które wykorzystują to wyjście, nie zostały jeszcze zaktualizowane. W takiej sytuacji można wdrożyć kontener adaptera w tym samym podzie, co nowy kontener aplikacji, i dostosowywać wyjście, aby pasowało do starszej wersji, dopóki wszyscy odbiorcy nie zostaną zaktualizowani. Kontener adaptera wykorzystuje wspólny system plików z głównym kontenerem aplikacji, zatem może obserwować lokalny system plików, zatem ilekroć nowa aplikacja coś zapisze, będzie mógł to natychmiast dostosować.

Wzorce wielowęzłowe

Wszystkie wzorce jednowęzłowe są bezpośrednio wspierane przez Kubernetes poprzez pody. Wzorce wielowęzłowe, takie jak wybór lidera (*leader election*), kolejki robocze czy zbieranie i rozpraszanie (*scatter-gather*) nie są obsługiwane bezpośrednio, ale komponowanie podów ze standardowymi interfejsami w celu ich realizacji w Kubernetes jest realistycznym podejściem.

Wiele narzędzi, platform i dodatków, które są głęboko zintegrowane z Kubernetes, wykorzystuje te wzorce projektowe. Piękno tych wzorców leży w tym, że wszystkie one są luźno powiązane i nie wymagają modyfikowania Kubernetes ani nawet tego, by był świadomy tych integracji. Tętniący życiem ekostystem wokół Kubernetes jest bezpośrednim rezultatem jego architektury. Przejdźmy zatem jeden poziom głębiej i poznajmy API Kubernetes.

API Kubernetes

Chcąc zrozumieć funkcjonalności systemu i co on naprawdę udostępnia, musimy poświęcić wiele uwagi jego interfejsom programowania aplikacji – czyli API. Te API zapewniają wyczerpujący obraz, co jako użytkownicy możemy robić w systemie. Kubernetes eksponuje kilka zestawów API typu REST, przeznaczonych do różnych celów i różnych grup użytkowników. Niektóre z tych API jest wykorzystywanych głównie przez narzędzia, podczas gdy inne mogą być bezpośrednio używane przez deweloperów. Ważnym aspektem API jest to, że podlegają nieustannemu rozwojowi. Deweloperzy Kubernetes dbają o możliwość zapanowania nad tym, starając się je rozszerzać (dodawać nowe obiekty i pola do istniejących obiektów) i unikać przemianowywania lub odrzucania istniejących obiektów i pól. Dodatkowo wszystkie punkty końcowe API są wersjonowane i często zawierają także oznaczenia fazy alpha lub beta; na przykład:

```
/api/v1
/api/v2alpha1
```

Dostęp do API możemy uzyskiwać poprzez interfejs wiersza polecenia (CLI) kubectl, biblioteki klienckie albo bezpośrednio poprzez wywołania REST. Dostępne są rozbudowane mechanizmy uwierzytelniania i autoryzacji, które zbadamy bliżej w późniejszych rozdziałach. O ile mamy odpowiednie uprawnienia, możemy wyliczać, przeglądać, tworzyć, aktualizować i usuwać rozmaite obiekty Kubernetes. W tym miejscu przyjrzymy się pobieżnie zewnętrznej warstwie tych API. Najlepszą drogą do poznawania API jest wykorzystanie grup API. Niektóre grupy są domyślnie włączone. Inne grupy można włączać/wyłączać za pomocą flag. Na przykład w celu wyłączenia grupy batch V1 i włączenia grupy batch V2 Alpha można ustawić flagę `--runtime-config` przy uruchamianiu serwera API, jak poniżej:

```
--runtime-config=batch/v1=false,batch/v2alpha=true
```

Oprócz zasobów jądra domyślnie włączone są poniższe zasoby:

- DaemonSet
- Deployment

- HorizontalPodAutoscaler
- Ingress
- Job
- ReplicaSet

Inną użyteczną klasyfikacją jest podział API według ich funkcjonalności. Przedstawiamy kategorie zasobów.

Kategorie zasobów

API Kubernetes są ogromne i podział na kategorie jest bardzo pomocny, gdy próbujemy znaleźć właściwą drogę. Kubernetes definiuje następujące kategorie zasobów:

- **Obciążenia (Workloads)** Obiekty, których używamy do kontrolowania i uruchamiania kontenerów w klastrze.
- **Odkrywanie i równoważenie obciążeń (Discovery and Load Balancing)** Obiekty używane w celu eksponowania światu naszych obciążeń jako zewnętrznie dostępnych, równoważonych usług.
- **Konfiguracja i magazyn (Konfiguracja i pamięć masowa)** Obiekty wykorzystywane do inicjowania i konfigurowania aplikacji oraz utrwalania danych znajdujących się poza kontenerami.
- **Klaster (Cluster)** Obiekty definiujące konfigurację klastra jako takiego; są one typowo wykorzystywane tylko przez operatorów klastrów.
- **Metadane (Metadata)** Obiekty służące do konfigurowania zachowania innych zasobów wewnątrz klastra, takich jak `HorizontalPodAutoscaler` używany do skalowania obciążeń.

W kolejnych podpunktach wyliczymy zasoby należące do każdej kategorii wraz ze wskazaniem grupy API, do której należą. Wykorzystamy w tym celu następujący format: <nazwa zasobu>: <grupa API >, na przykład `Container: core`, gdzie zasobem jest `Container`, a jego grupą API jest `core`. Nie będziemy tu specyfikować wersji, gdyż API przechodzą bardzo szybko od wersji alpha do beta i następnie do GA (*general availability* – ogólna dostępność), od V1 do V2 i tak dalej.

API obciążeń

API obciążeń zawiera wiele zasobów. Oto lista wszystkich tych zasobów wraz z grupami API, do których one należą:

- `Container: core`
- `CronJob: batch`

- `DaemonSet: apps`
- `Deployment: apps`
- `Job: batch`
- `Pod: core`
- `ReplicaSet: apps`
- `ReplicationController: core`
- `StatefulSet: apps`

Kontenery są tworzone przez kontrolery za pośrednictwem podów. Pody uruchamiają kontenery i zapewniają zależności środowiskowe, takie jak współużytkowane lub trwałe woluminy pamięci masowej oraz dane konfiguracji lub sekretów, wstrzykiwane do kontenera.

Poniższy przykład demonstruje jedną z najbardziej powszechnych operacji – uzyskania listy wszystkich podów poprzez wywołanie REST API:

```
GET /api/v1/pods
```

Wywołanie to akceptuje rozmaite parametry zapytania (wszystkie są opcjonalne):

- **pretty** Jeśli ma wartość `true`, wyjście zostanie elegancko sformatowane
- **labelSelector** Wyrażenie selektora w celu ograniczenia wyników
- **watch** Jeśli `true`, włącza śledzenie zmian i zwraca strumień zdarzeń
- **resourceVersion** Użyte łącznie z `watch` zwraca tylko zdarzenia, które wystąpiły dla zasobów tej wersji i wyższych
- **timeoutSeconds** Czas wygaśnięcia dla operacji `list` albo `watch`

Kolejna kategoria zasobów dotyczy zagadnień sieciowych (wysokiego poziomu).

Odkrywanie i równoważenie obciążeń

Ta kategoria znana jest również jako API usług. Domyślnie obciążenia są dostępne tylko wewnątrz klastra i muszą być jakoś eksponowane na zewnątrz, na przykład za pośrednictwem usług `LoadBalancer` albo `NodePort`.

W trakcie projektowania można uzyskiwać dostęp do wewnętrznie dostępnych obciążeń poprzez proxy uruchomione przy użyciu polecenia `kubectl proxy`:

- `Endpoints: core`
- `Ingress: networking.k8s.io`
- `Service: core`

Następna kategoria zasobów dotyczy pamięci masowej i zarządzania stanem wewnętrznym.

Konfiguracja i pamięć masowa

Dynamiczne konfigurowanie bez konieczności ponownego wdrażania jest kamieniem węgielnym Kubernetes i uruchamiania złożonych, rozproszonych aplikacji w klastrze Kubernetes. Przechowywanie danych jest innym istotnym zagadnieniem w każdym nietrywialnym systemie. Kategoria konfiguracji i pamięci masowej udostępnia wiele zasobów, które pozwalają rozwiązać te zagadnienia:

- `ConfigMap: core`
- `CSIDriver: storage.k8s.io`
- `CSINode: storage.k8s.io`
- `Secret: core`
- `PersistentVolumeClaim: core`
- `StorageClass: storage.k8s.io`
- `Wolumin: storage.k8s.io`
- `VolumeAttachment: storage.k8s.io`

Następna kategoria zasobów dotyczy zasobów pomocniczych, które zazwyczaj stanowią części innych zasobów wysokiego poziomu.

Metadane

Zasoby metadanych typowo ukazują się jako podzasoby obiektów konfiguracji. Na przykład zakres ograniczeń będzie częścią konfiguracji podu. W większości przypadków nie będziemy bezpośrednio odwoływać się do tych obiektów. Istnieje wiele zasobów metadanych i wyliczanie ich wszystkich w tym miejscu nie miałoby większego sensu. Wyczerpującą listę można znaleźć w dokumentacji dostępnej pod następującym adresem: *https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.16/#-strong-metadata-apis-strong-*.

Klastry

Zasoby należące do kategorii klaster są przeznaczone do wykorzystania przez operatorów klastra, a nie deweloperów. Również ta kategoria zawiera wiele zasobów. Oto niektóre spośród najważniejszych zasobów z tej kategorii:

- `Namespace: core`
- `Node: core`
- `PersistentVolume: core`
- `ResourceQuota: core`
- `Role: rbac.authorization.k8s.io`

- `RoleBinding: rbac.authorization.k8s.io`
- `ClusterRole: rbac.authorization.k8s.io`
- `ClusterRoleBinding: rbac.authorization.k8s.io`
- `NetworkPolicy: networking.k8s.io`

Po tym skrótowym przeglądzie tego, jak Kubernetes porządkuje i eksponuje swoje funkcjonalności za pośrednictwem grup API i kategorii zasobów, przyjrzyjmy się temu, jak zarządza infrastrukturą fizyczną i utrzymuje aktualność stanu klastra.

Komponenty Kubernetes

Klaster Kubernetes zawiera kilka głównych komponentów używanych do sterowania klastrem (master), a także komponenty węzłów, które są uruchamiane w każdym węźle roboczym. Poznamy teraz wszystkie te komponenty i ich współdziałanie.

Komponenty master

Komponenty warstwy sterowania (master) mogą wszystkie działać w jednym węźle, ale w konfiguracji o wysokiej dostępności lub w bardzo wielkich klastrach mogą być rozproszone pomiędzy wiele węzłów.

Serwer API

Serwer API Kubernetes udostępnia API typu REST systemu. Komponent ten może być łatwo skalowany w poziomie, gdyż jest bezstanowy, a wszystkie dane przechowuje w klastrze etcd. Serwer API jest ucieleśnieniem warstwy sterowania Kubernetes.

Etcd

Etcd to rozproszony magazyn danych o wysokiej niezawodności. Kubernetes wykorzystuje go do przechowywania pełnego stanu klastra. W małym, tymczasowym klastrze wystarcza pojedyncze wystąpienie etcd, uruchomione w tym samym węźle, co wszystkie pozostałe komponenty master. Jednak w bardziej rozbudowanych klastrach typowo występują trzy-, a nawet pięciowęzłowe klastry etcd w celu zapewnienia nadmiarowości i wysokiej dostępności.

kube-controller-manager

Menedżer kontrolerów Kube jest zbiorem różnych kontrolerów, które zostały zebrane razem w pojedynczym pliku binarnym. Zawiera kontrolery replikacji, podów, usług, punktów końcowych i inne. Wszystkie te menedżery śledzą stan klastra

za pośrednictwem API; ich zadaniem jest takie sterowanie klastrem, aby uzyskiwał stan pożądany.

cloud-controller-manager

Przy uruchomieniu w chmurze Kubernetes umożliwia dostawcom chmurowym integrację z ich platformą w celu zapewnienia zarządzania węzłami, trasami, usługami i woluminami. Kod dostawcy chmurowego współpracuje z kodem Kubernetes. Zastępuje część funkcjonalności menedżera kontrolerów Kube. Przy uruchamianiu Kubernetes z chmurowym menedżerem kontrolerów trzeba ustawić flagę menedżera Kube `--cloud-provider` jako `"external"`. Wyłącza to pętlę sterowania, w którą przejmuje chmurowy menedżer kontrolerów. Chmurowe menedżery kontrolerów zostały wprowadzone w wersji Kubernetes 1.6 i są obecnie wykorzystywane przez wielu dostawców usług chmurowych, takich jak:

- GCP
- AWS
- Azure
- Baidu Cloud
- DigitalOcean
- Oracle
- Linode

Krótka uwaga na temat języka Go, która powinna pomóc w analizowaniu kodu: nazwa metody podawana jest jako pierwsza, a po niej znajdziemy parametry tej metody w nawiasach. Każdy parametr stanowi parę złożoną z nazwy uzupełnionej typem. Na koniec specyfikowane są zwracane wartości. Język Go pozwala na wiele zwracanych typów. Bardzo typowe jest zwrócenie obiektu błędu jako uzupełnienie rzeczywistego wyniku. Jeśli wszystko działa dobrze, obiekt błędu będzie pusty (`nil`).

Poniższy listing prezentuje główny interfejs pakietu `cloudprovider`:

```
package cloudprovider
import (
    "errors"
    "fmt"
    "strings"
    "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/types"
    "k8s.io/client-go/informers"
    "k8s.io/kubernetes/pkg/controller"
)
```

```
// Interfejs jest abstrakcyjnym, dołączalnym interfejsem
// dla dostawców chmurowych.
type Interface interface {
    Initialize(clientBuilder controller.ControllerClientBuilder)
    LoadBalancer() (LoadBalancer, bool)
    Instances() (Instances, bool)
    Zones() (Zones, bool)
    Clusters() (Clusters, bool)
    Routes() (Routes, bool)
    ProviderName() string
    HasClusterID() bool
}
```

Większość metod zwraca inne interfejsy ze swoimi własnymi metodami. Dla przykładu przyjrzyjmy się interfejsowi `LoadBalancer`:

```
type LoadBalancer interface {
GetLoadBalancer(clusterName string,
    service \*v1.Service) (status \*v1.LoadBalancerStatus,
    exists bool,
    err error)

EnsureLoadBalancer(clusterName string,

    service \*v1.Service,
    nodes []\*v1.Node) (\*v1.LoadBalancerStatus, error)

UpdateLoadBalancer(clusterName string, service \*v1.Service, nodes
[]\*v1.Node) error

EnsureLoadBalancerDeleted(clusterName string, service \*v1.Service) error
}
```

Chmurowy menedżer kontrolerów jest niezbędnym instrumentem pozwalającym na włączenie Kubernetes do wszystkich głównych dostawców chmurowych, ale prawdziwym sednem Kubernetes jest scheduler.

Uwaga Termin *scheduling* (planowanie) w Kubernetes oznacza *przypisanie podu do węzła*. Pod jest uruchamiany natychmiast, a nie w jakiejś chwili w przyszłości, co mógłby sugerować ten termin. W dalszej części książki zazwyczaj będziemy używać terminu *rozmieszczanie* na określenie tej operacji.

kube-scheduler

Kube-scheduler jest odpowiedzialny za rozmieszczanie podów pomiędzy węzły. Jest to bardzo złożone zadanie, gdyż konieczne jest uwzględnienie wielu wzajemnie zależnych czynników, takich jak następujące:

- Żądania zasobów
- Żądania usług
- Sprzętowe/programowe zasady ograniczeń
- Specyfikacje koligacji i antykoligacji węzłów
- Specyfikacji koligacji i antykoligacji podów
- Skazy i tolerancje
- Lokalność danych
- Terminy realizacji

Jeśli potrzebna jest jakaś specjalna logika rozmieszczania, która nie jest obsługiwana przez domyślny kube-scheduler, możliwe jest zastąpienie tego kontrolera swoim własnym, niestandardowym schedulerem. Możliwe jest również stosowanie niestandardowego schedulera równolegle z domyślnym i zadbanie o to, aby niestandardowy scheduler rozmieszczał tylko pewien podzbiór podów.

DNS

Począwszy od wersji Kubernetes 1.3 usługa DNS jest częścią standardowego klastra Kubernetes. Jest rozmieszczana jako zwykły pod. Każda usługa (poza usługami *headless*) otrzymuje nazwę DNS. Również pody mogą otrzymywać nazwę DNS. Jest to bardzo użyteczne przy realizowaniu automatycznego odkrywania usług i zasobów.

Komponenty węzłów

Węzły klastra potrzebują kilku komponentów zapewniających interakcję z komponentami master, otrzymywanie obciążeń do wykonywania i aktualizowania swojego statusu na serwerze API Kubernetes.

Proxy

Kube-proxy realizuje niskopoziomową obsługę sieci w każdym węźle. Odwzorowuje lokalnie usługi Kubernetes i może przekierowywać ruch TCP i UDP. Adresy IP klastra odnajduje przy użyciu zmiennych środowiskowych albo DNS.

Kubelet

Kubelet jest przedstawicielem systemu Kubernetes w danym węźle. Nadzoruje komunikację z komponentami warstwy sterowania i zarządza działającymi podami. Do jego obowiązków należą:

- Odbieranie specyfikacji podu
- Pobieranie sekretów podu z serwera API
- Montowanie woluminów
- Uruchamianie kontenerów podu (za pośrednictwem skonfigurowanego mechanizmu wykonawczego)
- Raportowanie statusu węzła i wszystkich podów
- Uruchamianie sond rozruchu, żywotności i gotowości kontenerów

W tym podrozdziale zagłębiliśmy się we wnętrze Kubernetes i zbadaliśmy jego architekturę z bardzo wysokiego punktu widzenia, a także obsługiwane tu wzorce projektowe. Przejrzeliśmy też jego API oraz komponenty używane do sterowania i zarządzania klastrem. W kolejnym podrozdziale przyjrzymy się różnym mechanizmom wykonawczym wspieranym przez Kubernetes.

Mechanizmy wykonawcze w Kubernetes

Oryginalnie Kubernetes wspierał jedynie Dockera jako mechanizm wykonawczy kontenerów. Ta sytuacja jednak uległa zmianie. Obecnie Kubernetes wspiera kilka różnych mechanizmów wykonawczych:

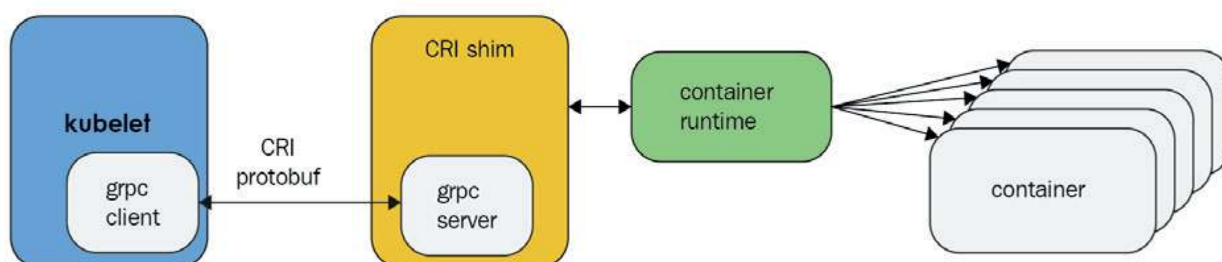
- Docker poprzez nakładkę (*shim*) CRI
- rkt (bezpośrednia integracja, która ma być zastąpiona przez rktlet)
- CRI-O
- Frakti (Kubernetes on the Hypervisor, wcześniej Hypernetes)
- rktlet (implementacja zgodna z CRI dla rkt)
- CRI-containerd

Główna zasada projektowa głosi, że sam Kubernetes powinien być całkowicie oddzielony od konkretnych mechanizmów wykonawczych. Umożliwia to **Container Runtime Interface** (CRI).

W tym podrozdziale przyjrzymy się bliżej koncepcji CRI oraz indywidualnym silnikom wykonawczym. Po jej przeczytaniu będziemy w stanie zdecydować, jaki silnik wykonawczy kontenerów jest odpowiedni dla naszego przypadku i w jakich okolicznościach moglibyśmy go zmienić, a być może nawet łączyć wiele mechanizmów wykonawczych w ramach jednego systemu.

Container Runtime Interface (CRI)

CRI stanowi kolekcję API typu gRPC, specyfikacji/wymagań oraz bibliotek mechanizmów wykonawczych kontenerów, które umożliwiają integrację ze składnikiem kubelet w węźle. W wersji Kubernetes 1.7 wewnętrzna integracja Dockera została zastąpiona rozwiązaniem integracji opartej na CRI. To znacząca zmiana, jako że otwiera drzwi licznym implementacjom, które mogą korzystać z najnowszych osiągnięć świata kontenerów. Kubelet nie potrzebuje już bezpośrednio komunikować się z wieloma mechanizmami wykonawczymi. Zamiast tego może porozumieć się z każdym mechanizmem wykonawczym zgodnym z CRI. Przepływ pracy ilustruje poniższy diagram:



Rysunek 1.2: *Diagram przepływu pracy w Container Runtime Interface (CRI)**

Każdy mechanizm wykonawczy kontenerów zgodny z CRI (albo podkładka) musi implementować dwa interfejsy usług gRPC, `ImageService` oraz `RuntimeService`. `ImageService` odpowiedzialna jest za zarządzanie obrazami kontenerów. Poniżej pokazany jest interfejs `gRPC/protobuf` (jest to język specyfikacji Protobuf opracowany przez Google, a nie Go):

```
service ImageService {
    rpc ListImages(ListImagesRequest) returns (ListImagesResponse) {}
    rpc ImageStatus(ImageStatusRequest) returns (ImageStatusResponse) {}
    rpc PullImage(PullImageRequest) returns (PullImageResponse) {}
    rpc RemoveImage(RemoveImageRequest) returns (RemoveImageResponse) {}
    rpc ImageFsInfo(ImageFsInfoRequest) returns (ImageFsInfoResponse) {}
}
```

Interfejs `RuntimeService` jest odpowiedzialny za zarządzanie podami i kontenerami. Oto ten interfejs w formacie `gRPC/protobuf`:

```
service RuntimeService {
```

* Źródło: *https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/*

```
    rpc Version(VersionRequest) returns (VersionResponse) {}
    rpc RunPodsandbox(RunPodsandboxRequest) returns
(RunPodsandboxResponse) {}
    rpc StopPodsandbox(StopPodsandboxRequest) returns
(StopPodsandboxResponse) {}
    rpc RemovePodsandbox(RemovePodsandboxRequest) returns
(RemovePodsandboxResponse) {}
    rpc PodsandboxStatus(PodsandboxStatusRequest) returns
(PodsandboxStatusResponse) {}
    rpc ListPodsandbox(ListPodsandboxRequest) returns
(ListPodsandboxResponse) {}
    rpc CreateContainer(CreateContainerRequest) returns
(CreateContainerResponse) {}
    rpc StartContainer(StartContainerRequest) returns
(StartContainerResponse) {}
    rpc StopContainer(StopContainerRequest) returns
(StopContainerResponse) {}
    rpc RemoveContainer(RemoveContainerRequest) returns
(RemoveContainerResponse) {}
    rpc ListContainers(ListContainersRequest) returns
(ListContainersResponse) {}
    rpc ContainerStatus(ContainerStatusRequest) returns
(ContainerStatusResponse) {}
    rpc UpdateContainerResources(UpdateContainerResourcesRequest) returns
(UpdateContainerResourcesResponse) {}
    rpc ExecSync(ExecSyncRequest) returns (ExecSyncResponse) {}
    rpc Exec(ExecRequest) returns (ExecResponse) {}
    rpc Attach(AttachRequest) returns (AttachResponse) {}
    rpc PortForward(PortForwardRequest) returns (PortForwardResponse) {}
    rpc ContainerStats(ContainerStatsRequest) returns
(ContainerStatsResponse) {}
    rpc ListContainerStats(ListContainerStatsRequest) returns
(ListContainerStatsResponse) {}
    rpc UpdateRuntimeConfig(UpdateRuntimeConfigRequest) returns
(UpdateRuntimeConfigResponse) {}
```

```
    rpc Status(StatusRequest) returns (StatusResponse) {}
}
```

Typy danych używane jako typy argumentów i wartości zwracanych nazywane są komunikatami (`message`) i również są definiowane jako część API. Oto jeden z nich:

```
message CreateContainerRequest {
    string pod\_sandbox\_id = 1;
    ContainerConfig config = 2;
    PodsandboxConfig sandbox\_config = 3;
}
```

Jak można zauważyć, komunikaty mogą być osadzane w sobie nawzajem. Komunikat `CreateContainerRequest` zawiera jedno pole typu `string` oraz dwa inne pola, które same są komunikatami `ContainerConfig` i `PodsandboxConfig`.

Teraz, gdy wiemy już, co na poziomie kodu Kubernetes uważa za mechanizm wykonawczy, przyjrzymy się pokrótce poszczególnym silnikom.

Docker

W świecie kontenerów Docker jest oczywiście zawodnikiem wagi superciężkiej. Oryginalnie Kubernetes został zaprojektowany jedynie do zarządzania kontenerami Dockera. Możliwość stosowania wielu mechanizmów wykonawczych pojawiła się najpierw w wersji Kubernetes 1.3, a CRI zostało wprowadzone w Kubernetes 1.5. Do tego czasu Kubernetes mógł zarządzać tylko kontenerami Dockera.

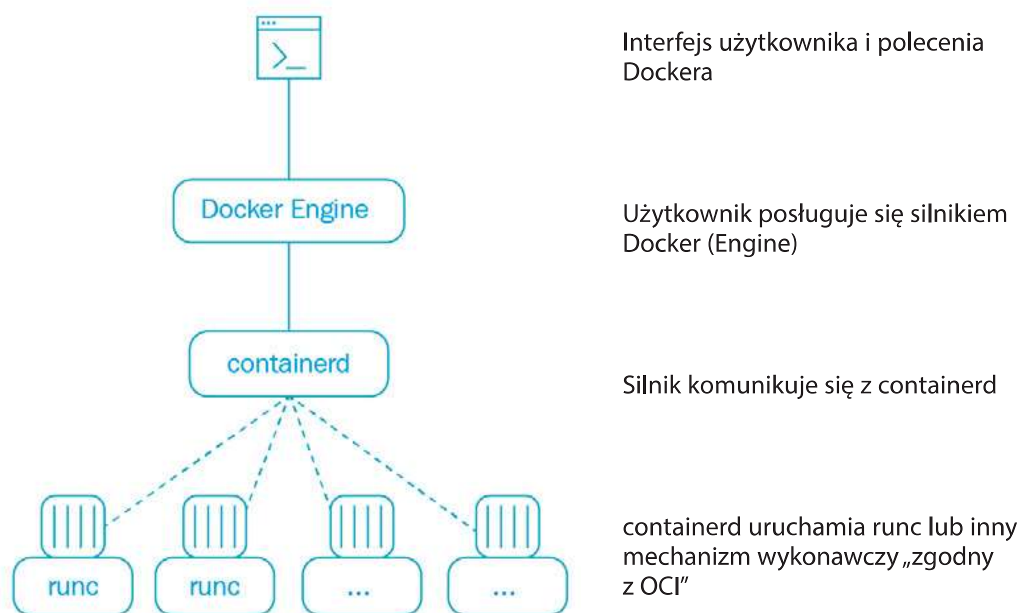
Zakładam, że Czytelnik dobrze zna środowisko Docker i co ono oferuje, skoro zapragnął przeczytać tę książkę. Docker cieszy się ogromną popularnością i rozwojem, ale nie oznacza to braku krytyki. Krytycy często wymieniają następujące zastrzeżenia:

- Bezpieczeństwo
- Trudność konfigurowania wielokontenerowych aplikacji (a w szczególności funkcji sieciowych)
- Trudności projektowania, monitorowania i rejestrowania
- Ograniczenie kontenerów Dockera do uruchomienia jednego polecenia
- Zbyt szybkie publikowanie częściowo opracowanych funkcjonalności

Środowisko Docker jest świadome tej krytyki i podejmuje próby rozwiązania przynajmniej części tych zastrzeżeń. W szczególności Docker zainwestował w produkt o nazwie Docker Swarm (rój). Jest to natywne rozwiązanie orkiestracji kontenerów, konkurencyjne dla Kubernetes. Jest prostsze w użyciu od Kubernetes, ale zdecydowanie nie tak efektywne ani dojrzałe.

Począwszy od wersji Docker 1.12 tryb roju jest natywnie dołączony do demona Docker, co wywołało irytację części użytkowników ze względu na rozdęcie systemu i rozmycie zakresu. W rezultacie więcej osób sięgnęło po mechanizm CoreOS rkt jako rozwiązanie alternatywne.

Zaczynając od wydanej w kwietniu 2016 roku wersji 1.11, Docker zmienił sposób uruchamiania kontenerów. Mechanizm wykonawczy używa obecnie **containerd** oraz **runC** do uruchamiania obrazów zgodnych z **Open Container Initiative** (**OCI**) w kontenerach:



Rysunek 1.3: *Architektura Docker 1.11 po oparciu go na runC oraz containerd**

rkt

rkt jest menedżerem kontenerów opracowanym przez CoreOS (twórców dystrybucji CoreOS Linux, etcd, flannel i wielu innych rozwiązań). Nie jest już dalej rozwijany, odkąd CoreOS zostało wchłonięte przez Red Hat, który następnie został kupiony przez IBM. Tym niemniej dziedzictwem rkt jest rozpowszechnienie się wielu mechanizmów wykonawczych kontenerów poza Dockerem, co popchnęło Dockera w stronę standardu OCI.

Mechanizm wykonawczy rkt wyróżnia się prostotą oraz silnym skupieniem na bezpieczeństwie i izolacji. Nie zawiera on demona, takiego jak Docker Engine, i opiera się na mechanizmie inicjowania systemu operacyjnego, takim jak systemd, do uruchamiania pliku wykonywalnego. rkt może pobierać obrazy (zarówno obrazy App

* Źródło: *https://www.docker.com/blog/docker-engine-1-11-runc/*

Container (appc), jak i obrazy OCI), weryfikować je i uruchamiać w kontenerach. Jego architektura jest znacznie prostsza.

App Container

W grudniu 2014 roku firma CoreOS rozpoczęła starania standaryzacyjne, nazywane appc. Obejmują one standardowy format obrazu (**ACI – Application Container Image**), mechanizm wykonawczy, podpisywanie i odkrywanie. Kilka lat później Docker rozpoczął swoje własne działania standaryzacyjne, czyli OCI. Obecnie wydaje się, że te działania zbiegną się kiedyś. Byłoby wspaniale, gdyby narzędzia, obrazy i mechanizmy wykonawcze mogły swobodnie współpracować ze sobą. Jednak jeszcze nie dotarliśmy do tego miejsca.

CRI-O

CRI-O jest inkubacyjnym projektem Kubernetes. Ma na celu zapewnienie ścieżki integracji pomiędzy Kubernetes a zgodnymi z OCI mechanizmami wykonawczymi kontenerów, takimi jak Docker. CRI-O zapewnia następujące funkcjonalności:

- Obsługę wielu formatów obrazów, włącznie z istniejącym formatem obrazu Dockera.
- Obsługę wielu sposobów pobierania obrazów, w tym relacje zaufania i weryfikację obrazów.
- Zarządzanie obrazami kontenerów (zarządzanie warstwami obrazów, nakładowymi systemami plików i tak dalej).
- Zarządzanie cyklem życia procesu kontenera.
- Monitorowanie i rejestrowane, wymagane przez CRI.
- Izolowanie zasobów, wymagane przez CRI.

Jak dotąd, wspierane są kontenery runc oraz Kata, ale dowolny mechanizm wykonawczy zgodny z OCI może zostać dołączony jako wtyczka i zintegrowany z Kubernetes.

Kontenery Hyper

Kontenery Hyper to kolejna opcja. Kontener Hyper zawiera lekką maszynę wirtualną (jego własne jądro gościa) i może być uruchamiany w maszynie fizycznej. Zamiast opierania się na cgroups systemu Linux w celu zapewnienia izolacji, polega na wykorzystaniu hiperwizora. To podejście stanowi ciekawe połączenie, jeśli porównać je ze standardowymi klastrami fizycznymi, bardzo trudnymi do skonfigurowania, i chmurami publicznymi, w których kontenery są wdrażane na masywnych maszynach wirtualnych.

Frakti

Frakti pozwala Kubernetes używać hiperwizorów za pośrednictwem zgodnego z OCI projektu runV w celu uruchamiania swoich podów i kontenerów. Jest to lekkie, przenośne i bezpieczne podejście, które zapewnia silniejszą izolację z własnym jądrem każdego kontenera, gdy porównamy ją z tradycyjnym podejściem linuksowym opartym na przestrzeniach nazw, ale nie jest tak ciężkie, jak w pełni wyposażone maszyny wirtualne.

Stackube

Stackube (wcześniej znane pod nazwą Hypernetes) jest wielodostępną dystrybucją, która wykorzystuje kontenery Hyper, a także pewne komponenty OpenStack w celu zapewnienia uwierzytelniania, trwałej pamięci masowej i usług sieciowych. Ponieważ kontenery nie współużytkują jądra hosta, bezpieczne jest uruchamianie kontenerów różnych dzierżawców na tym samym hoście fizycznym. Stackube oczywiście używa Frakti jako swojego mechanizmu wykonawczego.

W tym podrozdziale przedstawiliśmy różne silniki wykonawcze obsługiwane przez Kubernetes, a także zaprezentowaliśmy trend w kierunku standaryzacji, zbieżności i wydzielenia obsługi mechanizmów wykonawczych poza jądro Kubernetes. W kolejnym podrozdziale cofniemy się o krok, aby uzyskać ogólniejszy obraz. Zastanowimy się tu, jak Kubernetes wpasowuje się w potok CI/CD.

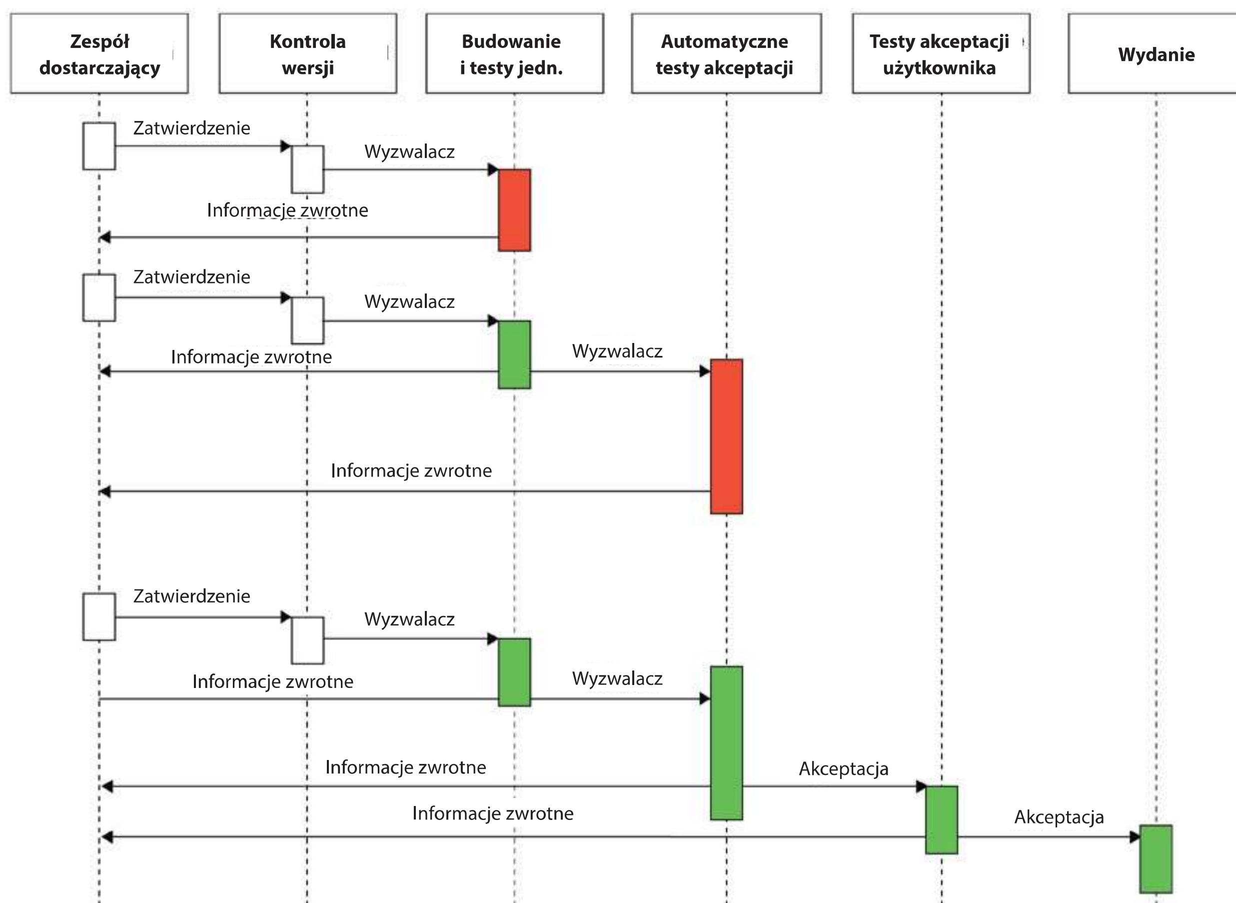
Ciągła integracja i wdrażanie

Kubernetes jest doskonałą platformą dla uruchamiania aplikacji opartych na mikrousługach. Jednak ostatecznie jest to jedynie szczegół implementacyjny. Użytkownicy, a coraz częściej również deweloperzy, mogą nie zdawać sobie sprawy, że system został wdrożony w Kubernetes. Jednak Kubernetes może zmienić tę sytuację i sprawić, że możliwe staną się rzeczy, które dotychczas były zbyt trudne, aby je zrealizować.

W tym podrozdziale przyjrzymy się potokowi CI/CD i tym, co Kubernetes wnosi do gry. Na koniec tego podrozdziału będziemy w stanie zaprojektować potoki CI/CD, które wykorzystują takie cechy Kubernetes, jak łatwość skalowania oraz zgodność środowiska deweloperskiego i produkcyjnego, aby podnieść produktywność i solidność codziennej pracy deweloperskiej i wdrożeniowej.

Potok CI/CD jest zbiorem narzędzi i kroków, określających, jak pewien zbiór zmian jest wprowadzany przez deweloperów lub operatorów modyfikujących kod, dane lub konfigurację systemu, następnie jest testowany, po czym wdrażany w środowisku produkcyjnym (a być może również w innych środowiskach). Niektóre potoki są w pełni

zautomatyzowane, a inne częściowo, z uwzględnieniem sprawdzeń wykonywanych przez ludzi. W dużych organizacjach mogą występować środowiska testowe i pośrednie, w których zmiany są wdrażane automatycznie, ale wydanie do środowiska produkcyjnego wymaga ręcznej interwencji. Poniższy diagram ilustruje typowy potok:



Rysunek 1.4: *Diagram potoku CI/CD*

Warto tu wspomnieć, że deweloperzy mogą być całkowicie odizolowani od infrastruktury produkcyjnej. Ich interfejsem jest po prostu przepływ pracy Git, czego dobrym przykładem jest Deis Workflow (PaaS oparty na Kubernetes, analogiczny do Heroku).

Projektowanie potoku CI/CD dla Kubernetes

Jeśli nasze wdrożenie ma nastąpić w klastrze Kubernetes, warto ponownie przemyśleć kilka tradycyjnych praktyk. Początkującym możemy powiedzieć, że sposób pakowania aplikacji będzie odmienny. Musimy przygotować obrazy dla naszych kontenerów. Wycofywanie zmian w kodzie jest niezwykle łatwe i natychmiastowe przy inteligentnym użyciu etykiet. Daje to sporo pewności, że jeśli jakaś błędna zmiana prześliźnie się przez testy, będziemy w stanie natychmiast powrócić do poprzedniej wersji.

Trzeba tu jednak zachować staranność. Nie można automatycznie wycofywać zmian schematu i migracji danych.

Inna unikatowa możliwość oferowana przez Kubernetes polega na tym, że deweloperzy mogą lokalnie uruchamiać cały klaster. Zaprojektowanie własnego klastra wymaga nieco pracy, jednak ponieważ mikrousługi tworzące nasz (budowany) system działają w kontenerach, a te kontenery współpracują ze sobą poprzez API, jest to możliwe i praktyczne podejście. Jak zwykle, jeśli nasz system jest bardzo silnie uzależniony od danych, będziemy musieli się dostosować i udostępnić migawki oraz dane syntetyczne, z których deweloperzy będą mogli korzystać.

Istnieje wiele komercyjnych rozwiązań CI/CD, które wspierają Kubernetes, ale dostępnych jest również szereg rozwiązań natywnych dla Kubernetes, takich jak Tekton, Argo CD i Jenkins X.

Natywne dla Kubernetes rozwiązania CI/CD działają wewnątrz klastra, są specyfikowane przy użyciu Kubernetes CRD i wykorzystują kontenery do wykonywania poszczególnych kroków. Dzięki wykorzystaniu takich rozwiązań CI/CD uzyskujemy korzyści wynikające z zarządzania przez Kubernetes, w tym łatwość skalowania potoków CI/CD, co w innych sytuacjach często okazuje się nietrywialnym zadaniem.

Podsumowanie

W tym rozdziale omówiliśmy wiele podstawowych pojęć i Czytelnik powinien teraz rozumieć projekt i architekturę Kubernetes. Przypomnijmy, że Kubernetes jest platformą orkiestracji dla aplikacji opartych na mikrousługach, uruchamianych jako kontenery. Klastry Kubernetes zawierają węzeł master oraz węzły robocze. Kontenery uruchamiane są w podach. Każdy pod działa na pojedynczej maszynie, fizycznej albo wirtualnej. Kubernetes bezpośrednio wspiera wiele koncepcji, takich jak usługi, etykiety czy trwała pamięć masowa. W Kubernetes możemy implementować różne wzorce projektowe systemów rozproszonych. Mechanizmy wykonawcze kontenerów muszą jedynie implementować standard CRI. Wspierane są kontenery Docker, rkt, Hyper i wiele innych.

W rozdziale 2, *Tworzenie klastrów Kubernetes*, poznamy rozmaite sposoby tworzenia klastrów Kubernetes, przedyskutujemy stosowalność różnych opcji i zbudujemy przykładowy, wielowęzłowy klaster.

2
Tworzenie klastrów Kubernetes

Przegląd

W poprzednim rozdziale dowiedzieliśmy się, czym w ogóle jest Kubernetes, jak został zaprojektowany, poznaliśmy wspierane koncepcje i silniki wykonawcze oraz to, jak wpasowuje się w potok CI/CD.

Tworzenie klastra Kubernetes od zera jest nietrywialnym zadaniem. Dostępnych jest wiele opcji i narzędzi, pomiędzy którymi musimy dokonywać wyborów. Trzeba też uwzględnić wiele czynników. W tym rozdziale zawiniemy rękawy i zbudujemy kilka klastrów Kubernetes, wykorzystując Minikube, KinD oraz K3d. Omówimy też i ocenimy inne narzędzia, takie jak Kubeadm, Kubespray, KRIB, RKE i bootkube. Przyjrzymy się również różnym środowiskom wdrażania, takim jak lokalne, chmurowe i fizyczne („bare metal”). Łącznie przedstawimy tu następujące zagadnienia:

- Tworzenie jednowęzłowego klastra za pomocą Minikube
- Tworzenie wielowęzłowego klastra przy użyciu KinD
- Tworzenie wielowęzłowego klastra przy użyciu k3d
- Tworzenie klastrów w chmurze
- Tworzenie fizycznych klastrów od podstaw
- Przegląd innych opcji tworzenia klastrów Kubernetes

Po ukończeniu tego rozdziału Czytelnik będzie miał solidną wiedzę o różnych opcjach budowania klastrów Kubernetes i będzie znać najlepsze w swojej dziedzinie narzędzia

wspomagające tworzenie klastrów. Przejdzie też przez praktyczne ćwiczenia budowania kilku klastrów, zarówno jedno-, jak i wielowęzłowych.

Tworzenie jednowęzłowego klastra za pomocą Minikube

W tym podrozdziale utworzymy lokalny, jednowęzłowy klaster przy użyciu Minikube. Lokalne klastry są przydatne głównie dla deweloperów, którzy chcą realizować szybkie cykle edycji- testów- wdrożenia-debugowania na swoim własnym komputerze, zanim zdecydują się zatwierdzić swoje zmiany. Lokalne klastry są również przydatne dla DevOps i operatorów, którzy chcą wypróbowywać różne opcje Kubernetes bez obawiania się, że zakłócą działanie współużytkowanego środowiska. Choć w środowiskach produkcyjnych Kubernetes typowo wdrażany jest w systemie Linux, wielu deweloperów pracuje na komputerach Windows albo na Makach. Tym niemniej nie będzie zbyt wielkiej różnicy, jeśli zechcemy zainstalować Minikube w Linuksie:

Rysunek 2.1: *Logo Minikube*

Poznajemy kubectl

Zanim jeszcze zaczniemy tworzyć klastry, warto poświęcić nieco czasu na omówienie kubectl. Jest to oficjalny interfejs wiersza polecenia (CLI) Kubernetes, zapewniający interakcję z serwerem API Kubernetes. Domyślnie jest konfigurowany poprzez plik `~/.kube/config`, czyli plik w formacie YAML zawierający metadane, informacje o połączeniu oraz tokeny uwierzytelniania albo certyfikaty dla jednego lub więcej klastrów. Kubectl udostępnia polecenia pozwalające wyświetlać konfigurację i przełączać się pomiędzy klastrami, jeśli konfiguracja ta zawiera więcej niż jeden. Można również przekierować do kubectl inny plik konfiguracyjny, ustawiając zmienną środowiskową `KUBECONFIG`. Osobiście preferuję trzecie podejście, polegające na utrzymywaniu oddzielnego pliku `config` dla każdego klastra i kopiowaniu pliku aktywnego klastra do `~/.kube/config` (ważne: łącza symboliczne nie działają w tym przypadku).

Całość funkcjonalności kubectl będziemy poznawać w trakcie tego i kolejnych rozdziałów. Powodem tej początkowej wzmianki jest po prostu uniknięcie pomyłek związanych z posługiwaniem się różnymi klastrami i plikami konfiguracyjnymi.

Krótkie wprowadzenie do Minikube

Minikube jest najbardziej dojrzałym lokalnym klastrem Kubernetes. Uruchamia najnowsze, stabilne wydanie Kubernetes i działa w systemach operacyjnych Windows, macOS i Linux. Obsługuje następujące funkcjonalności:

- Typ usługi `LoadBalancer` za pośrednictwem tunelu Minikube
- Typ usługi `NodePort` za pośrednictwem usługi Minikube
- Wiele klastrów w jednym systemie hosta
- Montowanie systemów plików
- Wsparcie dla GPU w zastosowaniach uczenia maszynowego
- RBAC
- Trwałe woluminy
- `Ingress`
- Tablicę kontrolną
- Niestandardowe mechanizmy wykonawcze kontenerów poprzez flagę `start --container-runtime`
- Konfigurowanie opcji serwera API i kubelet poprzez flagi wiersza polecenia
- Dodatki

Przygotowanie

Istnieje kilka wymagań wstępnych, które trzeba spełnić (zainstalować), zanim będziemy mogli utworzyć sam klaster. Należą do nich VirtualBox, narzędzie wiersza poleceń kubectl oraz, oczywiście, sam Minikube. Poniższe strony zawierają najnowsze wersje tych narzędzi:

- VirtualBox: *https://www.virtualbox.org/wiki/Downloads*
- Kubectl: *https://kubernetes.io/docs/tasks/tools/install-kubectl/*
- Minikube: *https://kubernetes.io/docs/tasks/tools/install-minikube/*

W systemie Windows

Instalujemy VirtualBox i upewniamy się, że programy kubectl i Minikube znajdują się w naszej ścieżce dostępu. Osobiście po prostu umieszczam wszystkie używane programy wiersza polecenia na `c:`. Czytelnik może preferować inne podejście. Do zarządzania wieloma konsolami, terminalami i sesjami SSH wykorzystuję doskonałe narzędzie ConEMU. Działa ono doskonale z powłokami cmd, PowerShell, PuTTY, Cygwin, msys i Git-Bash. Nie ma chyba nic lepszego dla Windows.

W przypadku Windows 10 Pro mamy opcję użycia hiperwizora Hyper-V. Od strony technicznej jest to lepsze rozwiązanie, niż VirtualBox, ale wymaga posiadania wersji Pro systemu Windows i jest całkowicie specyficzne dla Windows. Przy posługiwaniu się VirtualBox przedstawiane instrukcje są uniwersalne i można je łatwo zaadaptować do innych wersji Windows lub innych systemów operacyjnych. Warto też zauważyć, że jeśli mamy włączoną funkcję Hyper-V, musimy ją wyłączyć – VirtualBox nie może koegzystować z Hyper-V.

Rekomenduję tu wykorzystanie PowerShell w trybie administratora. Do naszego profilu PowerShell warto dodać następujący alias oraz funkcję:

```
Set-Alias -Name k -Value kubectl
function mk
{
  minikube-windows-amd64 `
  --show-libmachine-logs `
  --alsologtostderr      `
  @args
}
```

W systemie macOS

W systemie macOS dostępna jest opcja wykorzystania HyperKit zamiast VirtualBox:

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/docker-
machine-driver-hyperkit \
  && chmod +x docker-machine-driver-hyperkit \
  && sudo mv docker-machine-driver-hyperkit /usr/local/bin/ \
  && sudo chown root:wheel /usr/local/bin/docker-machine-driver-hyperkit \
  && sudo chmod u+s /usr/local/bin/docker-machine-driver-hyperkit
```

Możemy następnie dodać aliasy do swojego pliku `.bashrc` (analogiczne do aliasu i funkcji PowerShell w Windows):

```
alias k='kubectl'
alias mk='/usr/local/bin/minikube'
```

Jeśli ktoś zdecyduje się na używanie HyperKit zamiast VirtualBox, powinien dodać flagę `--vm-driver=hyperkit` przy uruchamianiu klastra.

Ważne jest również wyłączenie dowolnych funkcji VPN przy korzystaniu z HyperKit.

Możemy teraz używać poleceń `k` oraz `mk`, oszczędzając sobie wpisywania. Flagi przekazywane do Minikube w funkcji `mk` zapewniają lepsze rejestrowanie zdarzeń i przekierowywanie ich do konsoli oprócz zapisywania do plików (analogicznie do `tee`).

Wpisujemy `mk version`, aby się upewnić, że Minikube jest poprawnie zainstalowany i funkcjonalny:

```
$ mk version
minikube version: v1.10.1
```

Wpisanie `k version` pozwala zweryfikować, że kubectl jest poprawnie zainstalowany i funkcjonalny:

```
$ k version
Client Version: version.Info{Major:"1", Minor:"18", GitVersion:"v1.18.3", Gi
tCommit:"641856db18352033a0d96dbc99153fa3b27298e5", GitTreeState:"clean",
BuildDate:"2020-05-20T12:52:00Z", GoVersion:"go1.13.9", Compiler:"gc",
Platform:"darwin/amd64"}
The connection to the server localhost:8080 was refused - did you specify
the right host or port?
Unable to connect to the server: dial tcp 192.168.99.100:8443: getsockopt:
operation timed out
```

Nie trzeba się przejmować komunikatem błędu zwróconym w ostatnim wierszu. Nie mamy jeszcze żadnego uruchomionego klastra, zatem kubectl nie może się z niczym połączyć. Tego właśnie oczekiwaliśmy.

Sugeruję zbadanie dostępnych poleceń i flag, zarówno dla Minikube, jak i kubectl. Nie będę tu szczegółowo omawiać wszystkich poleceń, a jedynie tych, których będziemy używać.

Tworzenie klastra

Narzędzie Minikube obsługuje wiele wersji Kubernetes. W chwili pisania tych słów najnowsza była wersja 1.18.0, która jest również wersją domyślną:

```
$ mk start
😄 minikube v1.10.1 on darwin (amd64)
🔥  Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
🐳  Configuring environment for Kubernetes v1.18.0 on Docker 19.03.8
🚜  Pulling images ...
🚀  Launching Kubernetes ...
⌛  Verifying: apiserver proxy etcd scheduler controller dns
🏄  Done! kubectl is now configured to use "minikube"
```

Kiedy ponownie uruchamiamy istniejący już, ale zatrzymany klaster, zobaczymy następujące wyjście:

```
$ mk start
😄 minikube v1.10.1 on darwin (amd64)
```