

O'REILLY®

Kubernetes

– rozwiązania chmurowe w świecie DevOps

Tworzenie, wdrażanie i skalowanie
nowoczesnych aplikacji chmurowych



Helion

John Arundel
Justin Domingus

Tytuł oryginału: Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud

Tłumaczenie: Łukasz Wójcicki

ISBN: 978-83-283-6928-3

© 2021 Helion SA

Authorized Polish translation of the English edition of *Cloud Native DevOps with Kubernetes*
ISBN 9781492040767 © 2019 John Arundel and Justin Domingus

This translation is published and sold by permission of O'Reilly Media, Inc., which owns
or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form
or by any means, electronic or mechanical, including photocopying, recording or by
any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości
lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione.
Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie
książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie
praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi
bądź towarowymi ich właścicielami.

Autorzy oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje
były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich
wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych
lub autorskich. Autorzy oraz Helion SA nie ponoszą również żadnej odpowiedzialności
za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zairzyj pod adres

http://helion.pl/user/opinie/kubdev_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Kubernetes — rozwiązania chmurowe w świecie DevOps

Kubernetes — rozwiązania chmurowe w świecie DevOps to niezbędny przewodnik po teraźniejszych systemach rozproszonych. Jest to bardzo przejrzysta i pouczająca lektura, obejmująca wszystkie zagadnienia. Dużo się nauczyłem i na pewno skorzystam z kilku wskazówek!

— Will Thames, Platform Engineer, Skedulo

Najbardziej obszerna, wyczerpująca i praktyczna książka na temat obsługi infrastruktury Kubernetes. Absolutny must-have.

— Jeremy Yates, SRE Team, The Home Depot QuoteCenter

Żałuję, że nie miałem tej książki, kiedy zaczynałem! To lektura obowiązkowa dla wszystkich tworzących i uruchamiających aplikacje w Kubernetes.

— Paul van der Linden, Lead Developer, vdL Software Consultancy

Ta książka jest bardzo ekscytująca. To kopalnia informacji dla każdego, kto chce korzystać z Kubernetes. Czuję, że osiągnąłem kolejny poziom!

— Adam McPartlan (@mcparty), Senior Systems Engineer, NYnet

Naprawdę podobała mi się ta książka. Napisana jest w luźnym stylu, ale jednocześnie autorytatywna. Zawiera wiele praktycznych porad i dokładnie takie informacje, których każdy potrzebuje, ale nie wie, jak je zdobyć bez odpowiedniego doświadczenia.

— Nigel Brown, natywny trener chmurowy i autor kursów

Spis treści

| | |
|---|-----------|
| Przedmowa | 19 |
| Wstęp | 21 |
| 1. Rewolucja chmurowa | 25 |
| Tworzenie chmury | 25 |
| Czas kupowania | 26 |
| Infrastruktura jako usługa | 27 |
| Początki DevOps | 27 |
| Nikt nie rozumie DevOps | 28 |
| Przewaga biznesowa | 29 |
| Infrastruktura w postaci kodu | 30 |
| Wspólna nauka | 30 |
| Nadejście kontenerów | 31 |
| Stan aktualny | 31 |
| Myślenie pudełkowe | 31 |
| Umieszczeranie oprogramowania w kontenerach | 32 |
| Aplikacje Plug and Play | 33 |
| Uruchomienie orkiestratora | 33 |
| Kubernetes | 34 |
| Od Borga do Kubernetes | 34 |
| Co sprawia, że Kubernetes jest tak cenny? | 35 |
| Czy Kubernetes zniknie? | 36 |
| Kubernetes nie załatwia wszystkiego | 37 |
| Model Cloud Native | 38 |
| Przyszłość operacji | 39 |
| Rozproszone DevOps | 40 |
| Niektóre rzeczy pozostaną scentralizowane | 40 |

| | |
|--|-----------|
| Produktywni programiści | 40 |
| Jesteś przyszłością | 41 |
| Podsumowanie | 41 |
| 2. Pierwsze kroki z Kubernetes | 43 |
| Uruchamianie pierwszego kontenera | 43 |
| Instalowanie Docker Desktop | 43 |
| Co to jest Docker? | 44 |
| Uruchamianie obrazu kontenera | 44 |
| Aplikacja demonstracyjna | 45 |
| Oglądamy kod źródłowy | 45 |
| Wprowadzenie do języka Go | 46 |
| Jak działa aplikacja demonstracyjna? | 46 |
| Budowanie kontenera | 46 |
| Opis plików Dockerfile | 47 |
| Minimalne obrazy kontenerów | 47 |
| Uruchamianie skompilowanego obrazu Docker | 48 |
| Nazewnictwo obrazów | 48 |
| Przekierowanie portów (ang. port forwarding) | 49 |
| Rejestry kontenerowe | 49 |
| Uwierzytelnianie w rejestrze | 49 |
| Przydzielanie nazwy i dodawanie obrazu | 50 |
| Uruchamianie obrazu | 50 |
| Cześć, Kubernetes | 50 |
| Uruchamianie aplikacji demonstracyjnej | 50 |
| Jeśli kontener się nie uruchamia | 51 |
| Minikube | 52 |
| Podsumowanie | 52 |
| 3. Opis Kubernetes | 53 |
| Architektura klastrowa | 53 |
| Warstwa sterowania | 54 |
| Elementy węzła | 54 |
| System wysokiej niezawodności (ang. high availability) | 55 |
| Koszty samodzielnego hostingu Kubernetes | 56 |
| To więcej pracy niż myślisz | 57 |
| Nie chodzi tylko o początkową konfigurację | 58 |
| Narzędzia nie wykonają za Ciebie całej pracy | 58 |
| Kubernetes jest trudny | 58 |

| | |
|--|-----------|
| Koszty administracyjne | 59 |
| Zacznij od usług zarządzanych | 59 |
| Zarządzane usługi Kubernetes | 60 |
| Google Kubernetes Engine (GKE) | 60 |
| Automatyczne skalowanie klastra | 61 |
| Usługa Amazon Elastic Container Service dla Kubernetes (EKS) | 61 |
| Usługa Azure Kubernetes Service (AKS) | 61 |
| OpenShift | 62 |
| Usługa IBM Cloud Kubernetes Service | 62 |
| Rozwiązańa Kubernetes pod klucz | 62 |
| Containership Kubernetes Engine (CKE) | 62 |
| Instalatory Kubernetes | 63 |
| kops | 63 |
| Kubespray | 63 |
| TK8 | 63 |
| Kubernetes The Hard Way | 64 |
| kubeadm | 64 |
| Tarmak | 64 |
| Rancher Kubernetes Engine (RKE) | 64 |
| Moduł Puppet dla Kubernetes | 65 |
| Kubeformation | 65 |
| Kup lub zbuduj: nasze rekomendacje | 65 |
| Uruchom mniejsze oprogramowanie | 65 |
| Skorzystaj z zarządzanych usług Kubernetes, jeśli możesz | 66 |
| A co z uzależnieniem od jednego dostawcy? | 66 |
| Jeśli trzeba, użyj standardowych narzędzi do samodzielnego hostingu Kubernetes | 67 |
| Gdy Twoje wybory są ograniczone | 67 |
| Bare-metal i on-premise | 67 |
| Bezklasztowe usługi kontenerowe | 68 |
| Amazon Fargate | 68 |
| Azure Container Instances (ACI) | 69 |
| Podsumowanie | 69 |
| 4. Praca z obiektami Kubernetes | 71 |
| Zasoby Deployment | 71 |
| Nadzór i planowanie | 72 |
| Restart kontenerów | 72 |
| Zapytania obiektów Deployment | 72 |

| | |
|---|-----------|
| Pody | 73 |
| ReplicaSet | 74 |
| Utrzymanie pożdanego stanu | 74 |
| Scheduler | 75 |
| Manifesty zasobów w formacie YAML | 76 |
| Zasoby są danymi | 76 |
| Manifesty obiektu Deployment | 76 |
| Polecenie kubectl apply | 77 |
| Serwis | 77 |
| Odpytywanie klastra za pomocą polecenia kubectl | 80 |
| Przenoszenie zasobów na wyższy poziom | 80 |
| Helm: menadżer pakietów Kubernetes | 81 |
| Instalacja Helm | 81 |
| Instalacja wykresu Helm | 81 |
| Wykresy, repozytoria i wydania | 82 |
| Wyświetlanie listy wydań Helm | 82 |
| Podsumowanie | 83 |
| 5. Zarządzanie zasobami | 85 |
| Zrozumienie działania zasobów | 85 |
| Jednostki zasobów | 86 |
| Żądania zasobów | 86 |
| Limity zasobów | 86 |
| Utrzymuj małe kontenery | 87 |
| Zarządzanie cyklem życia kontenera | 88 |
| Sondy żywotności | 88 |
| Opóźnienie i częstotliwość sondy | 89 |
| Inne typy sond | 89 |
| Sondy gRPC | 89 |
| Sondy gotowości | 90 |
| Sondy gotowości na podstawie pliku | 90 |
| minReadySeconds | 91 |
| Budżety zakłóceń Poda | 91 |
| Korzystanie z przestrzeni nazw | 92 |
| Praca z przestrzeniami nazw | 93 |
| Jakich przestrzeni nazw powinieneś używać? | 93 |
| Adresy serwisów | 94 |
| Przydziły zasobów (ang. Resource Quotas) | 94 |
| Domyślne żądania zasobów i limity | 96 |

| | |
|--|------------|
| Optymalizacja kosztów klastra | 96 |
| Optymalizacja obiektów Deployment | 97 |
| Optymalizacja Podów | 97 |
| Vertical Pod Autoscaler | 98 |
| Optymalizacja węzłów | 98 |
| Optymalizacja przestrzeni dyskowej | 99 |
| Czyszczenie nieużywanych zasobów | 100 |
| Sprawdzanie wolnej pojemności | 102 |
| Korzystanie z instancji zastrzeżonych | 102 |
| Korzystanie z instancji w trybie wywłaszczeniowym | 103 |
| Utrzymywanie równowagi obciążeń | 105 |
| Podsumowanie | 106 |
| 6. Operacje na klastrach | 109 |
| Rozmiar i skalowanie klastra | 109 |
| Planowanie pojemności | 109 |
| Węzły i instancje | 112 |
| Skalowanie klastra | 114 |
| Sprawdzanie zgodności | 116 |
| Certyfikat CNCF | 116 |
| Testy zgodności z Sonobuoy | 118 |
| Walidacja i audyt | 118 |
| K8Guard | 118 |
| Copper | 119 |
| kube-bench | 119 |
| Dziennik kontroli Kubernetes (ang. Kubernetes audit log) | 119 |
| Testowanie chaosu | 120 |
| Tylko produkcja jest produkcją | 120 |
| chaoskube | 121 |
| kube-monkey | 121 |
| PowerfulSeal | 122 |
| Podsumowanie | 122 |
| 7. Narzędzia Kubernetes | 125 |
| Znowu o kubectl | 125 |
| Aliasy powłoki | 125 |
| Używanie przełączników | 126 |
| Skracanie typów zasobów | 126 |
| Automatyczne uzupełnianie poleceń kubectl | 126 |

| | |
|--|------------|
| Uzyskiwanie pomocy | 127 |
| Uzyskiwanie pomocy na temat zasobów Kubernetes | 127 |
| Bardziej szczegółowe wyniki | 127 |
| Praca z JSON Data i jq | 128 |
| Oglądanie obiektów | 129 |
| Opisywanie obiektów | 129 |
| Praca z zasobami | 129 |
| Imperatywne polecenia kubectl | 130 |
| Kiedy nie należy używać poleceń imperatywnych? | 130 |
| Generowanie manifestów zasobów | 131 |
| Eksportowanie zasobów | 131 |
| Śledzenie różnic w zasobach | 132 |
| Praca z kontenerami | 132 |
| Przeglądanie dzienników kontenera | 132 |
| Przyłączanie do kontenera | 134 |
| Oglądanie zasobów Kubernetes za pomocą kubespy | 134 |
| Przekierowanie portu kontenera | 134 |
| Wykonywanie poleceń w kontenerach | 135 |
| Rozwiązywanie problemów w kontenerach | 135 |
| Korzystanie z poleceń BusyBox | 136 |
| Dodawanie BusyBox do kontenerów | 137 |
| Instalowanie programów w kontenerze | 138 |
| Debugowanie w czasie rzeczywistym za pomocą kubesquash | 138 |
| Konteksty i przestrzenie nazw | 139 |
| kubectx i kubens | 140 |
| kube-ps1 | 140 |
| Powłoki i narzędzia Kubernetes | 141 |
| kube-shell | 141 |
| Click | 141 |
| kubed-sh | 141 |
| Stern | 142 |
| Budowanie własnych narzędzi Kubernetes | 142 |
| Podsumowanie | 143 |
| 8. Uruchamianie kontenerów | 145 |
| Kontenery i Pody | 145 |
| Co to jest kontener? | 146 |
| Co należy do kontenera? | 147 |
| Co należy do Poda? | 147 |

| | |
|---|------------|
| Manifesty kontenera | 148 |
| Identyfikatory obrazu | 149 |
| Tag latest | 149 |
| Skróty kontenerów | 150 |
| Podstawowe tagi obrazu | 150 |
| Porty | 151 |
| Żądania i limity dotyczące zasobów | 151 |
| Polityka pobierania obrazu | 151 |
| Zmienne środowiskowe | 152 |
| Bezpieczeństwo kontenerów | 152 |
| Uruchamianie kontenerów jako użytkownik inny niż root | 153 |
| Blokowanie kontenerów z uprawnieniami root | 153 |
| Ustawianie systemu plików tylko do odczytu | 154 |
| Wyłączanie eskalacji uprawnień | 154 |
| Mechanizm właściwości | 155 |
| Konteksty bezpieczeństwa Poda | 156 |
| Polityka bezpieczeństwa Poda | 156 |
| Konta usług Poda | 157 |
| Woluminy | 157 |
| Woluminy emptyDir | 158 |
| Woluminy trwałe | 159 |
| Restart polityk | 160 |
| Uwierzytelnianie przy pobieraniu obrazu | 160 |
| Podsumowanie | 160 |
| 9. Zarządzanie Podami | 163 |
| Etykiety | 163 |
| Co to są etykiety? | 163 |
| Selektory | 164 |
| Bardziej zaawansowane selektory | 164 |
| Inne zastosowania etykiet | 165 |
| Etykiety i adnotacje | 166 |
| Koligacje węzłów | 167 |
| Koligacje twarde | 167 |
| Koligacje miękkie | 168 |
| Koligacje Podów i antykoligacje | 168 |
| Trzymanie Podów razem | 169 |
| Trzymanie Podów oddzielnie | 169 |

| | |
|--|------------|
| Antykoligacje miękkie | 170 |
| Kiedy korzystać z koligacji Podów? | 170 |
| Skazy i tolerancje | 171 |
| Kontrolery Podów | 172 |
| DaemonSet | 172 |
| StatefulSet | 173 |
| Kontroler Job | 174 |
| CronJob | 175 |
| Horizontal Pod Autoscaler | 176 |
| PodPreset | 177 |
| Operatory i niestandardowe definicje zasobów (CRD) | 178 |
| Zasoby Ingress | 179 |
| Reguły Ingress | 180 |
| Zarządzanie połączeniami TLS za pomocą Ingress | 180 |
| Kontroler Ingress | 181 |
| Istio | 182 |
| Envoy | 182 |
| Podsumowanie | 183 |
| 10. Konfiguracja i obiekty Secret | 185 |
| ConfigMap | 185 |
| Tworzenie ConfigMap | 186 |
| Ustawianie zmiennych środowiskowych z obiektu ConfigMap | 186 |
| Ustawianie środowiska za pomocą ConfigMap | 188 |
| Używanie zmiennych środowiskowych w argumentach polecień | 189 |
| Tworzenie plików konfiguracji z ConfigMaps | 190 |
| Aktualizacja Podów po zmianie konfiguracji | 191 |
| Obiekty Secret aplikacji Kubernetes | 192 |
| Używanie obiektów Secret jako zmiennych środowiskowych | 192 |
| Zapisywanie obiektów Secret do plików | 193 |
| Odczyt obiektów Secret | 194 |
| Dostęp do obiektów Secret | 195 |
| Szyfrowanie w stanie spoczynku | 195 |
| Przechowywanie obiektów Secret | 195 |
| Strategie zarządzania obiektami Secret | 195 |
| Szyfrowanie Secret w systemach kontroli wersji | 196 |
| Zdalne przechowywanie Secret | 197 |
| Dedykowane narzędzie do zarządzania obiektami Secret | 197 |
| Rekomendacje | 198 |

| | |
|---|------------|
| Szyfrowanie obiektów Secret za pomocą Sops | 198 |
| Przedstawiamy Sops | 199 |
| Szyfrowanie pliku za pomocą Sops | 199 |
| Korzystanie z zaplecza KMS | 201 |
| Podsumowanie | 201 |
| 11. Bezpieczeństwo i kopia zapasowa | 203 |
| Kontrola dostępu i uprawnienia | 203 |
| Zarządzanie dostępem przez klaster | 203 |
| Kontrola dostępu oparta na rolach (RBAC) | 204 |
| Role | 204 |
| Wiązanie ról z użytkownikami | 205 |
| Jakich ról potrzebuję? | 206 |
| Ochrona dostępu do Cluster-Admin | 206 |
| Aplikacje i wdrażanie | 206 |
| Rozwiązywanie problemów z RBAC | 207 |
| Skanowanie bezpieczeństwa | 208 |
| Clair | 208 |
| Aqua | 208 |
| Anchore Engine | 209 |
| Kopie zapasowe | 209 |
| Czy muszę wykonać kopię zapasową? | 209 |
| Tworzenie kopii zapasowej etcd | 210 |
| Kopia zapasowa stanu zasobów | 210 |
| Tworzenie kopii zapasowej stanu klastra | 210 |
| Duże i małe katastrofy | 211 |
| Velero | 211 |
| Monitorowanie statusu klastra | 214 |
| kubectl | 214 |
| Wykorzystanie procesora i pamięci | 216 |
| Konsola dostawcy chmury | 216 |
| Pulpit Kubernetes (ang. Kubernetes Dashboard) | 216 |
| Weave Scope | 218 |
| kube-ops-view | 218 |
| node-problem-detector | 218 |
| Dalsza lektura | 219 |
| Podsumowanie | 220 |

| | |
|---|------------|
| 12. Wdrażanie aplikacji Kubernetes | 221 |
| Budowanie manifestów za pomocą wykresu Helm | 221 |
| Co znajduje się w wykresie narzędzia Helm? | 222 |
| Szablony Helm | 223 |
| Zmienne interpolacyjne | 223 |
| Wartości tekstowe w szablonach | 224 |
| Określanie zależności | 225 |
| Wdrażanie wykresów Helm | 225 |
| Ustawianie zmiennych | 225 |
| Określanie opcji podczas instalacji Helm | 226 |
| Aktualizowanie aplikacji za pomocą Helm | 226 |
| Powrót do poprzednich wersji | 227 |
| Tworzenie repozytorium wykresów Helm | 227 |
| Zarządzanie obiektami Secret wykresów Helm za pomocą Sops | 228 |
| Zarządzanie wieloma wykresami za pomocą Helmfile | 229 |
| Co znajduje się w pliku Helmfile? | 230 |
| Metadane wykresu | 231 |
| Stosowanie pliku Helmfile | 231 |
| Zaawansowane narzędzia do zarządzania manifestami | 232 |
| Tanka | 232 |
| kapitan | 233 |
| kustomize | 233 |
| kompose | 233 |
| Ansible | 234 |
| kubeval | 234 |
| Podsumowanie | 235 |
| 13. Proces tworzenia oprogramowania | 237 |
| Narzędzia programistyczne | 237 |
| Skaffold | 237 |
| Draft | 238 |
| Telepresence | 238 |
| Knative | 238 |
| Strategie wdrażania | 239 |
| Rolling Updates | 239 |
| Recreate | 240 |
| maxSurge i maxUnavailable | 240 |
| Wdrożenia niebiesko-zielone | 241 |

| | |
|--|------------|
| Wdrożenia rainbow | 242 |
| Wdrożenia kanarkowe | 242 |
| Obsługa migracji za pomocą Helm | 242 |
| Funkcja hook wykresu Helm | 243 |
| Obsługa nieudanych funkcji hook | 243 |
| Inne funkcje hook | 244 |
| Kolejność wykonywania funkcji hook | 244 |
| Podsumowanie | 244 |
| 14. Ciągłe wdrażanie w Kubernetes | 247 |
| Co to jest ciągłe wdrażanie? | 247 |
| Z którego narzędzia CD powiniem skorzystać? | 248 |
| Jenkins | 248 |
| Drone | 248 |
| Google Cloud Build | 248 |
| Concourse | 249 |
| Spinaker | 249 |
| GitLab CI | 249 |
| Codefresh | 249 |
| Azure Pipelines | 249 |
| Komponenty CD | 250 |
| Docker Hub | 250 |
| Gitkube | 250 |
| Flux | 250 |
| Keel | 250 |
| Potok CD z wykorzystaniem Google Cloud Build | 250 |
| Konfigurowanie Google Cloud i GKE | 251 |
| Kopiowanie repozytorium demo | 251 |
| Wprowadzenie do Cloud Build | 251 |
| Budowanie kontenera testowego | 252 |
| Uruchamianie testów | 252 |
| Budowanie kontenera aplikacji | 253 |
| Walidacja manifestów Kubernetes | 253 |
| Publikowanie obrazu | 253 |
| Tagi Git SHA | 254 |
| Tworzenie pierwszego triggera komplikacji | 254 |
| Testowanie triggera | 254 |
| Wdrożenie z potoku CD | 256 |

| | |
|--|------------|
| Tworzenie triggera wdrażania | 258 |
| Optymalizacja potoku komplikacji | 258 |
| Dostosowanie przykładowego potoku | 259 |
| Podsumowanie | 259 |
| 15. Obserwowałość i monitorowanie | 261 |
| Co to jest obserwowałość? | 261 |
| Co to jest monitorowanie? | 261 |
| Monitorowanie typu czarna skrzynka | 261 |
| Co oznacza określenie „działa”? | 263 |
| Zapisywanie logów | 264 |
| Przedstawiamy metryki | 265 |
| Śledzenie | 267 |
| Obserwowałość | 267 |
| Potok obserwowałości | 268 |
| Monitorowanie w Kubernetes | 269 |
| Zewnętrzny monitoring typu czarna skrzynka | 269 |
| Wewnętrzna kontrola aplikacji | 271 |
| Podsumowanie | 272 |
| 16. Metryki w Kubernetes | 275 |
| Czym są metryki? | 275 |
| Seria danych w czasie | 275 |
| Liczniki i mierniki | 276 |
| Co mogą powiedzieć metryki? | 277 |
| Wybór dobrych metryk | 277 |
| Usługi: wzorzec RED | 278 |
| Zasoby: wzorzec USE | 279 |
| Metryki biznesowe | 279 |
| Metryki Kubernetes | 280 |
| Analizowanie metryk | 283 |
| Dlaczego nie korzystać z wartości średniej? | 284 |
| Średnie arytmetyczne, mediany i wartości odstające | 284 |
| Odkrywanie percentylów | 284 |
| Stosowanie percentylów do danych metryk | 285 |
| Zwykle chcemy zobaczyć to, co najgorsze | 287 |
| Co zamiast percentylów? | 287 |

| | |
|--|------------|
| Tworzenie wykresów metryk w pulpicie | 288 |
| Użyj standardowego układu graficznego dla wszystkich serwisów | 288 |
| Zbuduj radiator informacji za pomocą pulpitu | 289 |
| Umieszczanie na pulpicie rzeczy, które ulegają awarii | 291 |
| Alerty na podstawie metryk | 291 |
| Jakie są problemy związane z alarmami? | 291 |
| Bezbolesna reakcja na żądanie | 292 |
| Pilne, ważne i przydatne alerty | 293 |
| Śledź swoje alerty, powiadomienia poza godzinami pracy i pobudki | 294 |
| Narzędzia i usługi metryczne | 294 |
| Prometheus | 294 |
| Google Stackdriver | 296 |
| AWS Cloudwatch | 297 |
| Azure Monitor | 297 |
| Datadog | 297 |
| New Relic | 298 |
| Podsumowanie | 299 |
| Posłowie | 301 |

Przedmowa

Witamy w książce *Kubernetes — rozwiązania chmurowe w świecie DevOps*.

Kubernetes to prawdziwa rewolucja branżowa. Cloud Native Computing Foundation (<https://landscape.cncf.io/>) skupia dane ponad 600 projektów ze świata cloud native. Podkreśla to znaczenie aplikacji Kubernetes w dzisiejszych czasach. Nie wszystkie te narzędzia zostały opracowane dla Kubernetes, nie wszystkich można nawet używać z Kubernetes, ale wszystkie są częścią ogromnego ekosystemu, w którym Kubernetes jest jedną z flagowych technologii.

Kubernetes zmienił sposób tworzenia i obsługiwanego aplikacji. Jest to obecnie kluczowy element w świecie DevOps. Kubernetes zapewnia programistom elastyczność i swobodę działania. Dziś możesz korzystać z Kubernetes u dowolnego dużego dostawcy rozwiązań chmurowych, w środowiskach lokalnych typu bare-metal, a także we własnym środowisku programistycznym. Stabilność, elastyczność, potężny interfejs API, otwarty kod i otwarta społeczność programistów to kilka powodów, dla których Kubernetes stał się standardem branżowym, podobnie jak Linux jest standardem w świecie systemów operacyjnych.

Kubernetes — rozwiązania chmurowe w świecie DevOps to świetny podręcznik dla osób, które wykonują codzienne czynności w Kubernetes lub dopiero rozpoczynają swoją podróż do tej aplikacji. John i Justin omawiają wszystkie główne aspekty wdrażania, konfigurowania i obsługi Kubernetes oraz najlepsze praktyki dotyczące tworzenia i uruchamiania aplikacji. Dają także doskonały przegląd powiązanych technologii, w tym Prometheus, Helm, a także podejścia związanego z ciągłym wdrażaniem. Jest to książka dla wszystkich pracujących w świecie DevOps.

Kubernetes to nie tylko kolejne eksytyujące narzędzie; jest to standard branżowy i fundament technologii nowych generacji, w tym bezserwerowych (OpenFaaS, Knative), i narzędzi do uczenia maszynowego (Kubeflow). Cały przemysł IT zmienia się z powodu rewolucji cloud native, a przeżywanie jej jest niezwykle eksytyjące.

— Ihor Dvoretskyi
Developer Advocate, Cloud Native
Computing Foundation
Grudzień 2018

Wstęp

W świecie IT kluczowe założenia DevOps zostały dobrze zrozumiane i powszechnie przyjęte. Przyszedł jednak czas na zmiany. Nowa platforma aplikacji o nazwie Kubernetes została szybko przyjęta przez firmy we wszystkich branżach na całym świecie. Ponieważ coraz więcej aplikacji i firm przenosi się z tradycyjnych serwerów do środowiska Kubernetes, ludzie pytają, jak do nowej rzeczywistości dostosować metodologię DevOps.

W książce tej wyjaśniamy, co pojęcie DevOps oznacza w świecie cloud native, w którym Kubernetes jest standardową platformą. Pomożemy Ci wybrać najlepsze narzędzia i framework z ekosystemu Kubernetes. Zaprezentujemy także spójny sposób korzystania z nich, oferując sprawdzone w praktyce rozwiązania, które obecnie działają w środowisku produkcyjnym, w realnym świecie.

Czego się nauczysz?

Dowiesz się, czym jest Kubernetes, skąd się bierze i co oznacza dla przyszłości tworzenia i działania oprogramowania. Dowiesz się, jak działają kontenery, jak je budować i nimi zarządzać oraz jak projektować usługi i infrastrukturę cloud native.

Zrozumiesz kompromisy między budowaniem i hostowaniem klastrów Kubernetes a korzystaniem z usług zarządzanych. Dowiesz się o możliwościach, ograniczeniach oraz zaletach i wadach popularnych narzędzi instalacyjnych Kubernetes, takich jak kops, kubeadm i Kubespray. Otrzymasz rzetelny przegląd głównych zarządzanych ofert Kubernetes od takich firm jak Amazon, Google i Microsoft.

Zdobędziesz praktyczne doświadczenie w pisaniu i wdrażaniu aplikacji Kubernetes, konfigurowaniu i obsłudze klastrów Kubernetes oraz automatyzacji infrastruktury i wdrożeń chmurowych za pomocą narzędzi, takich jak Helm. Zapoznasz się z usługą Kubernetes od strony bezpieczeństwa, uwierzytelniania i uprawnień, w tym kontroli dostępu opartej na rolach (RBAC). Poznasz sprawdzone metody zabezpieczania kontenerów oraz aplikacji Kubernetes w środowisku produkcyjnym.

Dowiesz się, jak skonfigurować ciągłą integrację i wdrożenia za pomocą Kubernetes, jak tworzyć kopie zapasowe i przywracać dane, jak testować kластer pod kątem zgodności i niezawodności, jak monitorować, śledzić, rejestrować i agregować metryki, a także jak utworzyć skalowalną, odporną na awarie i opłacalną infrastrukturę Kubernetes.

Aby zilustrować wszystkie rzeczy, o których mówimy, wykorzystaliśmy bardzo prostą aplikację demonstracyjną. Możesz śledzić wszystkie opisane przykłady, korzystając z kodu z autorskiego repozytorium Git.

Dla kogo jest ta książka?

Książka jest najbardziej odpowiednia dla personelu operacyjnego IT odpowiedzialnego za serwery, aplikacje i usługi oraz dla programistów odpowiedzialnych za tworzenie usług cloud native lub migrację istniejących aplikacji do Kubernetes i chmury. Zakładamy, że nie знаłeś wcześniej aplikacji Kubernetes ani kontenerów — nie martw się, przeprowadzimy Cię przez wszystko.

Doświadczeni użytkownicy Kubernetes powinni również znaleźć cenne informacje; w książce poruszamy także zaawansowane tematy, takie jak RBAC, ciągłe wdrażanie, zarządzanie obiektami Secret i obserwowalność. Niezależnie od Twojego poziomu wiedzy, mamy nadzieję, że znajdziesz coś przydatnego.

Na jakie pytania odpowiada ta książka?

Tworząc tę książkę, rozmawialiśmy z setkami ludzi na temat cloud native i Kubernetes — począwszy od liderów branży i ekspertów, kończąc na kompletnie początkujących.

Oto niektóre pytania, na które chcieliby oni uzyskać odpowiedź.

- Chciałbym dowiedzieć się, dlaczego powinienem poświęcić swój czas na tę technologię. Jakie problemy pomoże rozwiązać mnie i mojemu zespołowi?
- Kubernetes wydaje się świetny, ale chcemy dokładniej podejść do tematu. Konfiguracja szybkiego demo jest łatwa, ale obsługa i rozwiązywanie problemów wydaje się trudne. Chcielibyśmy uzyskać pewne solidne wskazówki na temat tego, jak ludzie zarządzają klastrami Kubernetes w prawdziwym świecie i jakie problemy możemy napotkać.
- Przydałyby się zaawansowane porady. Ecosystem Kubernetes ma zbyt wiele opcji do wyboru dla początkujących zespołów. Kiedy istnieje wiele sposobów na zrobienie tego samego, który z nich jest najlepszy? Jaki wybieramy?

I być może najważniejsze pytanie ze wszystkich.

- Jak korzystać z Kubernetes bez wystąpienia przerw w pracy firmy?

Pisząc tę książkę, pamiętaliśmy o tych i wielu innych kwestiach, dołożyliśmy wszelkich starań, aby na nie odpowiedzieć. Jak sobie poradziliśmy? Przejdz do następnej strony, aby się dowiedzieć.

Konwencje użyte w tej książce

W książce zastosowano następujące konwencje typograficzne.

Kursywa

Wskazuje nowe terminy, adresy URL, adresy e-mail, nazwy plików i rozszerzenia plików.

Stała szerokość

Używana w listingach programów, a także w akapitach w celu odniesienia do elementów programu, takich jak nazwy zmiennych lub funkcji, bazy danych, typy danych, zmienne środowiskowe, instrukcje i słowa kluczowe.

Pogrubienie o stałej szerokości

Pokazuje polecenia lub inny tekst, który powinien zostać wpisany przez użytkownika.

Kursywa o stałej szerokości

Pokazuje tekst, który należy zastąpić wartościami podanymi przez użytkownika lub wartościami określonymi przez kontekst.



Ten element oznacza wskazówkę lub sugestię.



Ten element oznacza ogólną uwagę.



Ten element oznacza ostrzeżenie lub przestroगę.

Korzystanie z przykładów

Materiały uzupełniające (przykłady kodu, ćwiczenia itp.) można pobrać pod adresem <ftp://ftp.helion.pl/przyklady/kubdev.zip>.

Książka ma Ci pomóc w wykonywaniu pracy. Zamieszczone przykłady możesz wykorzystać w swoich programach i dokumentacji. Nie musisz kontaktować się z nami w celu uzyskania zgody, chyba że reprodukujesz znaczną część kodu. Przykładowo napisanie programu wykorzystującego kilka fragmentów kodu z tej książki nie wymaga pozwolenia. Sprzedaż lub dystrybucja płyt CD-ROM z przykładami książek O'Reilly wymaga pozwolenia. Zacytowanie tej książki i podanie przykładowego kodu nie wymaga zgody. Włączenie znacznej ilości przykładowego kodu z tej książki do dokumentacji produktu wymaga pozwolenia.

Doceniamy , ale nie wymagamy podawania autorstwa. Atrybucja zwykle obejmuje tytuł, autora, wydawcę i numer ISBN. Przykładowo *Kubernetes — rozwiązania chmurowe w świecie DevOps*, John Arundel i Justin Domingus, Helion, Gliwice 2020.

Podziękowania

Nasze wielkie podziękowania należą się wielu osobom, które zapoznały się z wczesną wersją tej książki oraz udzielili nam bezcennych informacji i porad lub pomogły w inny sposób. Są to: Abby Bangser, Adam J. McPartlan, Adrienne Domingus, Alexis Richardson, Aron Trauring, Camilla Montonen, Gabriell Nascimento, Hannah Klemme, Hans Findel, Ian Crosby, Ian Shaw, Ihor Dvoretskyi, Ike Devolder, Jeremy Yates, Jérôme Petazzoni, Jessica Deen, John Harris, Jon Barber, Kitty Karate, Marco Lancini, Mark Ellens, Matt North, Michel Blanc, Mitchell Kent, Nicolas Steinmetz, Nigel Brown, Patrik Duditš, Paul van der Linden, Philippe Ensarguet, Pietro Mamberti, Richard Harper, Rick Highness, Sathyajith Bhat, Suresh Vishnoi, Thomas Liakos, Tim McGinnis, Toby Sullivan, Tom Hall, Vincent De Smet i Will Thames (i wszyscy inni).

Rewolucja chmurowa

Nie wiemy, kiedy świat zaczął istnieć, ponieważ krąży po okręgu, a okrąg nie ma początku.

— Alan Watts

Trwa rewolucja. Właściwie trzy rewolucje.

Pierwszą rewolucją jest utworzenie chmury — wyjaśnimy Ci, czym jest. Drugą — wprowadzenie metodyki *DevOps* — dowiesz się, jakie obejmuje operacje. Trzecią rewolucją jest powstanie kontenerów. Te trzy fale zmian razem tworzą nowy świat oprogramowania zwany *cloud native*. System operacyjny tego świata to *Kubernetes*.

W tym rozdziale krótko opiszemy historię i znaczenie tych zmian. Sprawdzimy także, w jaki sposób wpływają one na sposób wdrażania i obsługi oprogramowania. Wyjaśnimy także, co oznacza pojęcie *cloud native* i jakich zmian możesz spodziewać się w środowisku, w którym aktualnie pracujesz — o ile zajmujesz się rozwojem oraz wdrażaniem oprogramowania, bezpieczeństwem czy też administracją sieci.

Kiedy powiążemy powyższe trzy technologie, odkryjemy, że przyszłość rozwoju oprogramowania leży w rozwiązaniach chmurowych, konteneryzacji, systemach rozproszonych dynamicznie zarządzanych za pomocą automatyzacji na platformie *Kubernetes* (lub innej podobnej). Dalej w książce zapoznamy Cię ze sposobami obsługi tych aplikacji — rozwiązaniami chmurowymi opartymi na metodyce *DevOps*.

Jeżeli już znasz pojęcia podstawowe, możesz rozpocząć zabawę z *Kubernetes*, zaczynając od rozdziału 2. Jeśli nie, usiądź wygodnie z kubkiem swojego ulubionego napoju. Zaczynamy.

Tworzenie chmury

Początkowo (czyli w latach 60. ubiegłego wieku) komputery wypełniały szafy w olbrzymich, klimatyzowanych centrach danych — bez bezpośredniego dostępu użytkowników. Programiści pracowali na nich zdalnie. W taki sam sposób do takiej infrastruktury obliczeniowej mogliby się dostać wiele tysięcy użytkowników — za zużyty czas procesora lub inne zasoby otrzymaliby rachunek.

Dla wielu firm lub organizacji zakup i utrzymanie własnego sprzętu komputerowego nie byłby opłacalne. Dlatego pojawił się model biznesowy, w którym użytkownicy dzielą się mocą obliczeniową zdalnych maszyn będących własnością trzeciej strony.

Jeśli przypomina to dzisiejszą sytuację, nie jest to przypadek. Słowo rewolucja powstało od słowa „obrót”, a informatyka w pewnym sensie powróciła do miejsca, w którym się rozpoczęła. Komputery na przestrzeni lat stały się o wiele mocniejsze — dzisiejszy zegarek Apple stanowi odpowiednik około trzech komputerów mainframe. Jak pokazano na rysunku 1.1, współużytkowany dostęp do zasobów obliczeniowych typu pay-per-use to bardzo stary pomysł. Obecnie taki sposób dostępu nazwany został *rozwiązaniem chmurowym*. Rewolucja, która rozpoczęła się wraz ze współdzieleniem komputerów mainframe, zatoczyło koło.



Rysunek 1.1. Wczesny komputer chmurowy: IBM System/360 Model 91 w NASA Goddard Space Flight Center

Czas kupowania

Główna ideą rozwiązań chmurowych jest to, że zamiast zakupu komputera kupujesz *moc obliczeniową*. Oznacza to, że nie lokujesz sporej ilości kapitału w fizyczny zakup dużej ilości sprzętu komputerowego, który jest trudny do skalowania, łatwo się psuje oraz szybko starzeje. Zamiast tego płacisz tylko za czas pracy na komputerze, który jest własnością kogoś innego. Ten ktoś, a nie Ty, musi się zająć skalowaniem, konserwacją oraz aktualizacją tego sprzętu. W czasie epoki maszyn typu bare-metal — lub jeśli wolisz „epoki żelaza” — moc obliczeniowa była wydatkiem kapitałowym. Obecnie jest wydatkiem operacyjnym.

Chmura to nie tylko zdalna, wynajęta moc obliczeniowa. To także rozproszone systemy operacyjne. Możesz kupić tylko moc obliczeniową (tak jak w np. instancji Google Compute lub usłudze AWS Lambda) i użyć jej do uruchomienia własnego oprogramowania. Możesz także skorzystać z *usług chmurowych* (ang. *cloud services*) — czyli użyć czystego oprogramowania. Jeśli np. korzystasz z oprogramowania PagerDuty do monitorowania i ostrzegania (w przypadku wystąpienia awarii systemu), używasz usługi w chmurze (czasami nazywanej usługą *oprogramowanie jako usługa*, w skrócie SaaS).

Infrastruktura jako usługa

Kupując infrastrukturę chmurową dla swoich serwisów, naprawdę kupujesz usługę zwaną *infrastruktura jako usługa* (IaaS). Na jej zakup nie musisz korzystać z wydatków kapitałowych — nie musisz jej budować czy też ulepszać. To tylko koszt, taki jak elektryczność lub woda. Rozwiązania chmurowe to rewolucja w relacjach pomiędzy prowadzeniem interesu a infrastrukturą IT.

Outsourcing sprzętu to tylko część historii; chmura pozwala również na outsourcing oprogramowania, którego sam nie stworzyłeś; są to systemy operacyjne, bazy danych, oprogramowanie do klastrowania, replikacji, zarządzania siecią, monitorowania, zapewniające dostępność, obróbkę strumieni oraz wszystkie niezliczone warstwy oprogramowania i konfiguracji, które wypełniają lukę pomiędzy Twoim kodem a procesorem. Usługi zarządzane (ang. *managed services*) mogą obsłużyć prawie wszystkie te banalne ciężary (ang. *undifferentiated heavy lifting*) za Ciebie (o zaletach zarządzanych usług dowiesz się więcej w rozdziale 3.).

Rewolucja w chmurze wywołała także kolejną rewolucję w ludziach, którzy z niej korzystają; stał się nią ruch w kierunku DevOps.

Początki DevOps

Przed powstaniem metodyki DevOps tworzenie i obsługa oprogramowania były zasadniczo dwoma oddzielnymi zadaniami, obsługiwanyimi przez dwie różne grupy ludzi. Programiści po napisaniu przekazywali oprogramowanie personelowi *operacyjnemu*, który je uruchamiał i walczył z problemami w środowisku *produkcyjnym* (tzn. takim, które obsługuje prawdziwych użytkowników, zamiast po prostu działać w warunkach testowych). Podobnie jak komputery, które potrzebują własnej przestrzeni na podłodze, ta separacja ma swoje korzenie w połowie ubiegłego wieku. Tworzenie oprogramowania było bardzo specjalistycznym zajęciem — tak jak zarządzanie infrastrukturą informacyjną. Oba obszary nakładały się na siebie tylko w małym stopniu.

W rzeczywistości dwa działy miały zupełnie inne cele, które często kolidowały (patrz rysunek 1.2). Programiści zwykle koncentrują się na szybkim dostarczeniu nowych funkcjonalności, podczas gdy zespoły operacyjne dbają o to, aby dostarczone usługi były stabilne w dłuższym okresie czasu.

Kiedy na horyzoncie pojawiły się rozwiązania chmurowe, wszystko się zmieniło. Systemy rozproszone są złożone, a internet jest bardzo duży. Technika związane z usługą systemu — przywracanie po awarii, obsługa timeoutów, płynna aktualizacja wersji — nie są tak po prostu oddzielone od tworzenia, architektury oraz implementacji systemu.



Rysunek 1.2. Wśród dwóch zespołów może dochodzić do konfliktu interesów (autor Dave Roth)

Co więcej, „system” nie jest już tylko Twoim oprogramowaniem: obejmuje oprogramowanie wewnętrzne, usługi chmurowe, zasoby sieciowe, load balancery, monitorowanie, sieci dystrybucji, firewalle, DNS itd. Wszystkie te rzeczy są ściśle powiązane i wzajemnie zależne. Ludzie, którzy piszą oprogramowanie, muszą zrozumieć, w jaki sposób odnosi się do reszty systemu, a osoby obsługujące system muszą zrozumieć, jak działa oprogramowanie.

Ruch DevOps ma na celu połączenie tych dwóch grup. Dzięki współpracy można podzielić odpowiedzialność za systemy, poprawić niezawodność działania oraz zwiększyć skalowalność obu systemów i zespołów ludzi, którzy je zbudowali.

Nikt nie rozumie DevOps

DevOps był w pewnym okresie kontrowersyjnym pomysłem zarówno dla ludzi, którzy mówili, że to nic więcej, tylko nowoczesny opis już istniejących dobrych praktyk tworzenia oprogramowania, jak i dla tych, którzy odrzucali potrzebę ściślejszej współpracy między tworzeniem oprogramowania a działaniami operacyjnymi.

Istnieje również powszechnie nieporozumienie dotyczące tego, czym właściwie jest DevOps. Czy to rodzaj pracy? Czy zespół? Czy metodologia? Czy zestaw umiejętności? Wpływowy autor opisujący DevOps John Willis zidentyfikował cztery kluczowe filary DevOps, takie jak kultura, automatyzacja,

metryki i współdzielenie (CAMS — ang. *culture, automation, measurement, sharing*). Inaczej przedstawił to Brian Dawson. Stworzył on tzw. tróję DevOps: ludzie i kultura, proces i praktyka oraz narzędzia i technologia.

Niektórzy uważają, że istnienie chmur i kontenerów oznacza, że nie potrzebujemy już DevOps — taki punkt widzenia nazywany jest czasem *NoOps*. Oznacza to, że skoro wszystkie operacje IT są wynajmowane u usługodawcy w chmurze lub za pomocą usług trzeciej strony, to firmy nie potrzebują personelu operacyjnego w pełnym wymiarze godzin.

Założenia NoOps opierają się na niezrozumieniu tego, co faktycznie oferuje DevOps.

Dzięki DevOps większość tradycyjnych operacji informatycznych ma miejsce przed powstaniem kodu produkcyjnego. Każde wydanie nowej wersji oprogramowania obejmuje monitorowanie, rejestrowanie i testowanie A / B. Potoki CI/CD automatycznie uruchamiają testy jednostkowe, testy bezpieczeństwa oraz kontrole polityk przy każdym zatwierdzeniu każdej zmiany oprogramowania (ang. commit). Wdrożenia są automatyczne. Kontrole, zadania i wymagania niefunkcjonalne są teraz implementowane przed każdym wydaniem oprogramowania, co pozwala uniknąć niebezpiecznych następstw wystąpienia krytycznej awarii.

— Jordan Bach (AppDynamics (<https://blog.appdynamics.com/engineering/is-noops-the-end-of-devops-think-again/>))

Aby zrozumieć DevOps, trzeba wiedzieć, że dotyczy ono przede wszystkim kwestii organizacyjnych, ludzkich, a nie technicznych. Stwierdzenie to nawiązuje do jednego z praw o nazwie *Second Law of Consulting*, którego autorem jest Gerald Weinberg.

Mimo iż na początku może wyglądać inaczej, to zawsze jest problem ludzki.

— Gerald M. Weinberg, Secrets of Consulting

Przewaga biznesowa

Z biznesowego punktu widzenia DevOps został opisany jako „poprawa jakości oprogramowania poprzez przyspieszenie cykłów wydawniczych z użyciem chmurowych rozwiązań oraz wprowadzenie nowych korzyści dla oprogramowania, które jest obecnie w produkcji” (The Register (https://www.theregister.co.uk/2018/03/06/what_does_devops_do_to_decades_old_planning_processes_and_assumptions)).

Przyjęcie DevOps wymaga głębszej transformacji kulturowej dla firm. Muszą one rozpocząć zmiany na poziomie wykonawczym, strategicznym i stopniowo rozprzestrzeniać się w każdej części organizacji. Szybkość, zwinność, współpraca, automatyzacja i jakość oprogramowania są kluczowymi czynnikami DevOps — dla wielu firm oznacza to znaczną zmianę sposobu myślenia.

Jednak DevOps działa, a badania regularnie wykazują, że firmy, które przyjmują zasady DevOps, szybciej wypuszczają lepsze oprogramowanie, lepiej i szybciej reagują na awarie, są bardziej zwinne na rynku oraz znacznie poprawiają jakość swoich produktów.

DevOps nie jest chwilową modą; jest to raczej sposób, w jaki odnoszące sukcesy organizacje uprzedziałają dostarczanie wysokiej jakości oprogramowania; będzie to stanowiło także podstawę przez wiele kolejnych lat.

— Brian Dawson (Cloudbees), Computer Business Review

(<https://www.cbronline.com/enterprise-it/applications/devops-fad-stay>)

Infrastruktura w postaci kodu

Dawno, dawno temu programiści tworzyli oprogramowanie, a zespoły operacyjne zajmowały się sprzętem oraz systemami operacyjnymi.

Obecnie, gdy sprzęt znajduje się w chmurze, wszystko w pewnym sensie jest oprogramowaniem. Ruch DevOps przenosi tworzenie programowania do operacji; są to narzędzia i przepływy pracy do szybkiego, zwinnego, wspólnego budowania złożonych systemów. Z DevOps nierozerwalnie związane jest pojęcie *infrastruktury jako kodu*.

Fizyczna infrastruktura, w postaci komputerów, switchów, okablowania może zostać zastąpiona, za sprawą oprogramowania, przez infrastrukturę chmurową. Obowiązki inżynierów operacyjnych ulegają zmianie. Wcześniej musieli ręcznie dbać o upgrade sprzętu, a po zmianie są ludźmi tworzącymi oprogramowanie, które automatyzuje pracę w środowisku chmurowym.

Transformacja nie przebiega tylko w jednym kierunku. Programiści uczą się od zespołów operacyjnych, jak przewidywać awarie związane z rozproszonymi systemami opartymi na chmurze, jak złagodzić ich konsekwencje i jak zaprojektować bezpieczne i zgodne wstecz oprogramowanie.

Wspólna nauka

Zarówno zespoły programistów, jak i zespoły operacyjne uczą się, jak pracować razem. Uczę się, jak projektować i budować systemy, jak monitorować systemy produkcyjne oraz w jaki sposób wykorzystać uzyskane informacje w celu usprawnienia działania. Co ważniejsze, uczą się poprawiać jakość obsługi swoich użytkowników oraz podnoszą wartość firmy.

Ogromne możliwości środowiska chmurowego i oparte na kodzie charakter metodyki DevOps sprawiły, że wszelkie operacje stały się problemem oprogramowania. Jednocześnie oprogramowanie stało się problemem operacyjnym, zatem nasuwają się następujące pytania.

- W jaki sposób wdrażać i aktualizować oprogramowanie w dużych, różnorodnych sieciach, z różnymi architekturami serwerów i systemów operacyjnych?
- W jaki sposób wdrażać oprogramowanie w środowiskach rozproszonych w niezawodny i powtarzalny sposób, używając w dużej mierze standardowych komponentów?

Czas na trzecią rewolucję, czyli kontener.

Nadejście kontenerów

Aby wdrożyć oprogramowanie, potrzebujesz nie tylko samego oprogramowania, ale także składników wymaganych przez to oprogramowanie. Są to biblioteki, interpretery, pakiety, kompilatory, rozszerzenia itd.

Potrzebujesz także *konfiguracji*. Ustawienia, szczegółowe informacje dotyczące witryny, klucze licencyjne, hasła do bazy danych, wszystko, co zmienia oprogramowanie w użyteczną usługę.

Stan aktualny

Wcześniejsze próby rozwiązania tego problemu obejmują korzystanie z systemów *zarządzania konfiguracją*, takich jak Puppet lub Ansible. Narzędzia te dostarczają kod umożliwiający instalację, uruchomienie, konfigurację oraz aktualizację oprogramowania.

Niektóre języki zapewniają własny mechanizm archiwizacji, np. pliki JAR Javy, eggi Pythona lub gemy Ruby. Są one jednak specyficzne dla języka i nie rozwiązują całkowicie problemu zależności: aby np. uruchomić plik JAR, nadal potrzebujesz zainstalowanego środowiska wykonawczego Java.

Innym rozwiązaniem jest pakiet *omnibus*, który — jak sugeruje nazwa — próbuje wcisnąć wszystko, czego aplikacja potrzebuje, do jednego pliku. Taki pakiet zawiera oprogramowanie, jego konfigurację, zależne komponenty, ich konfigurację, ich zależności itd. (Przykładowo pakiet omnibus Java zawierałby środowisko wykonawcze Java, a także wszystkie pliki JAR aplikacji).

Niektórzy producenci poszli nawet o krok dalej i umieścili cały system komputerowy jako *obraz maszyny wirtualnej*. Obrazy te są jednak duże, czasochłonne w budowie i utrzymaniu, trudne we wdrażaniu oraz ogromnie nieefektywne i zasoboźne.

Z operacyjnego punktu widzenia nie tylko musisz zarządzać różnego rodzaju pakietami, ale także zarządzać grupą serwerów.

Serwery muszą być udostępniane, łączone w sieć, wdrażane, konfigurowane, aktualizowane za pomocą poprawek bezpieczeństwa, monitorowane, zarządzane itd.

Wszystkie powyższe działania wymagają znacznej ilości czasu, umiejętności i wysiłku. Czy nie ma lepszego sposobu?

Myślenie pudełkowe

Aby rozwiązać te problemy, przemysł technologiczny zapożyczył pomysł z branży morskiej, czyli kontener. W latach 50. ubiegłego wieku kierowca ciężarówki o nazwisku Malcolm McLean (<https://hbs.me/2Q0QCzb>) zaproponował, aby zamiast żmudnego rozładowywania towarów z ciężarówek ładować na statek od razu całą zawartość ciężarówki.

Przyczepa samochodowa to zasadniczo duża metalowa skrzynia na kołach. Jeśli moźesz oddzielić pudło — kontener — od kół i podwozia używanego do jego transportu, to uzyskujesz coś, co jest bardzo łatwe do podniesienia, załadowania, ułożenia i rozładowania. Może zostać bezpośrednio przeniesione na statek lub inną ciężarówkę (patrz rysunek 1.3).



Rysunek 1.3. Standaryzowane pojemniki znacznie obniżają koszty wysyłki towarów masowych (zdjęcie z serwisu Pixabay (<https://www.pexels.com/@pixabay>), licencja Creative Commons 2.0)

Sea-Land, firma kurierska McLeana odniosła duży sukces. Wykorzystała ten system do znacznie tańszej wysyłki towarów w kontenerach (<https://www.freightos.com/the-history-of-the-shipping-container>). Obecnie każdego roku wysyłane są setki milionów kontenerów, które przewożą towary o wartości trylionów dolarów.

Umieszczanie oprogramowania w kontenerach

Kontener oprogramowania to dokładnie ten sam pomysł: standardowy format pakowania i dystrybucji, który jest ogólny i rozpowszechniony. Umożliwia znacznie większą nośność, niższe koszty, skalowanie i łatwość obsługi. Kontener zawiera wszystko, czego aplikacja potrzebuje do uruchomienia. Zapisany jako plik *obrazu* może być wykonany przez środowisko *wykonawcze kontenera*.

Jaka jest różnica w stosunku do obrazu maszyny wirtualnej? On także zawiera wszystko, co aplikacja musi uruchomić — ale o wiele więcej. Typowy obraz maszyny wirtualnej to około 1 GiB¹. Z drugiej strony, dobrze zaprojektowany obraz kontenera może być sto razy mniejszy.

Ponieważ maszyna wirtualna zawiera wiele aplikacji, bibliotek i innych rzeczy, których nasze oprogramowanie nigdy nie będzie używać, większość miejsca jest marnowana. Przesyłanie obrazów maszyn wirtualnych w sieci jest znacznie wolniejsze niż zoptymalizowanych kontenerów.

¹ Gibibajt (GiB) to jednostka danych Międzynarodowej Komisji Elektrotechnicznej (IEC), zdefiniowana jako 1 024 mebibajtów (MiB). W tej książce będziemy używać jednostek IEC (GiB, MiB, KiB), aby uniknąć niejasności.

Co gorsza, maszyny wirtualne są tylko *wirtualne*: fizyczny CPU emuluje procesor, na którym działa maszyna wirtualna. Warstwa wirtualizacji ma dramatyczny, negatywny wpływ na wydajność (<https://www.stratoscale.com/blog/container/running-containers-on-bare-metal/>); w testach wirtualizacje działają o około 30% wolniej od równoważnych kontenerów.

Dla porównania, kontenery działają pod kontrolą rzeczywistego CPU, bez narzutów związanych z wirtualizacją, tak jak robią to zwykłe pliki binarne.

A ponieważ kontenery zawierają tylko potrzebne pliki, są znacznie mniejsze niż obrazy maszyn wirtualnych. Użyto w nich również sprytnej techniki adresowej warstw systemu plików, które mogą być współużytkowane i ponownie wykorzystywane między kontenerami.

Jeśli np. masz dwa kontenery i oba zawierają ten sam obraz Debian Linux, jest on pobierany tylko raz, a każdy kontener może po prostu się do niego odwoływać.

Środowisko wykonawcze kontenera zgromadzi wszystkie niezbędne warstwy i pobierze warstwę tylko wtedy, gdy nie jest ona lokalnie buforowana. W ten sposób bardzo efektywnie wykorzystywane jest zarówno miejsce na dysku, jak i przepustowość sieci.

Aplikacje Plug and Play

Kontener jest nie tylko jednostką wdrożenia oraz jednostką pakietu, jest to także jednostka ponownego wykorzystania (ten sam obraz kontenera może być wykorzystany jako składnik wielu różnych usług), jednostka skalowania i jednostka alokacji zasobów (kontener może działać wszędzie tam, gdzie dostępne są wystarczające zasoby na jego specyficzne potrzeby).

Programiści nie muszą się już martwić problemami związanymi z utrzymywaniem różnych wersji oprogramowania, tak aby działały w różnych dystrybucjach Linuksa, w różnych wersjach bibliotek, w różnych wersjach języka programowania itd. Praca z kontenerami zależy jedynie od jądra systemu operacyjnego (np. Linuksa).

Po prostu dodaj swoją aplikację do obrazu kontenera, a będzie ona działać na dowolnej platformie, która obsługuje standardowy format kontenera i ma kompatybilne jądro.

Brendan Burns i David Oppenheimer, programiści Kubernetes, opisali to w swoim artykule *Design Patterns for Container-based Distributed Systems* (<https://www.usenix.org/node/196347>).

Dzięki hermetycznemu zamknięciu, zadaniu o zależności i wprowadzeniu atomowego sygnału wdrażania („sukces” / „błąd”) [kontenery] znacznie poprawiają dotychczasowy stan wdrażania oprogramowania w centrum danych lub chmurze. Jednak kontenery mogą być potencjalnie czymś więcej niż tylko lepszym narzędziem do wdrażania — wierzymy, że ich działanie może być analogiczne do obiektów w obiektowych systemach oprogramowania i jako takie umożliwi opracowanie wzorców projektowania systemów rozproszonych.

Uruchomienie orkiestratora

Zespoły operacyjne również doszły do wniosku, że użycie kontenerów spowodowało, iż ich praca została uproszczona. Zamiast utrzymywać rozległą posiadłość maszyn różnego rodzaju, architektur i systemów operacyjnych, wystarczy uruchomić *orkiestrator kontenerów* (ang. *container orchestrator*),

czyli oprogramowanie zaprojektowane do łączenia wielu różnych maszyn w jeden kластер. Są to połączone jednostki obliczeniowe, postrzegane przez użytkownika jako pojedynczy bardzo wydajny komputer (na którym mogą działać kontenery).

Terminy *orkiestracja* (ang. *orchestration*) i *planowanie* (ang. *scheduling*) są często używane jako synonimy. Ścisłe mówiąc, *orkiestracja* w tym kontekście oznacza koordynację i sekwencjonowanie różnych działań na rzecz wspólnego celu (podobnie do pracy muzyków w orkiestrze). *Planowanie* oznacza zarządzanie dostępnymi zasobami i przypisywanie prac tam, gdzie można je najbardziej wydajnie uruchomić. (Nie należy mylić z planowaniem w sensie zaplanowanych zadań, które są wykonywane o ustalonych porach).

Trzecim ważnym działaniem jest *zarządzanie klastrami* (ang. *cluster management*), czyli łączenie wielu serwerów fizycznych lub wirtualnych w zunifikowaną, niezawodną, odporną na awarie grupę.

Termin „orkiestrator kontenerów” zwykle odnosi się do pojedynczej usługi, która zajmuje się planowaniem i koordynacją klastra oraz zarządzaniem nim.

Konteneryzacja (używanie kontenerów jako standardowej metody wdrażania i uruchamiania oprogramowania) zapewniała oczywiste korzyści, a standardowy format kontenerów umożliwiał wszelkiego rodzaju korzyści. Jednak wciąż jeden problem przeszkadzał w upowszechnieniu kontenerów; był to brak standardowego systemu orkiestracji kontenerów.

Tak długo, jak kilka różnych narzędzi do planowania i orkiestracji kontenerów konkurowało na rynku, firmy niechętnie decydowały się na wybór konkretnych technologii. Jednak to wszystko miało się zmienić.

Kubernetes

Google skorzystał z kontenerów na dużą skalę znacznie wcześniej niż ktokolwiek inny. Prawie wszystkie usługi Google działają w kontenerach: Gmail, wyszukiwarka Google, Mapy Google, Google App Engine itd. Ponieważ w tym czasie nie istniał żaden odpowiedni system orkiestracji kontenerów, Google został zmuszony do jego wynalezienia.

Od Borga do Kubernetes

Aby rozwiązać problem związany z uruchomieniem dużej liczby usług w skali globalnej na milionach serwerów, Google opracował prywatny, wewnętrzny system orkiestracji kontenerów o nazwie Borg (<https://pdos.csail.mit.edu/6.824/papers/borg.pdf>).

Borg zasadniczo jest decentralizowanym systemem zarządzania przydzielającym i planującym, które kontenery mają się uruchomić w danej puli serwerów. Bardzo potężny Borg jest ścisłe powiązany z wewnętrznymi i zastrzeżonymi technologiami Google, trudnymi do rozszerzenia i niemożliwymi do publicznego udostępnienia.

W 2014 r. Google założył projekt o otwartym kodzie źródłowym o nazwie Kubernetes (od greckiego słowa κυβερνητης, co oznacza sternik, pilot), który pozwala na wdrożenie orkiestratora kontenerów przez każdego, kto chce z niego skorzystać. Firma oparła się na wnioskach wyciągniętych z Borga i jego następcy Omegi (<https://research.google/pubs/pub41684/>).

Rozwój projektu Kubernetes był błyskawiczny. Mimo iż inne systemy orkiestracji kontenerów istniały przed Kubernetes, były to produkty komercyjne, zależne od producenta, co zawsze stanowiło barierę dla ich powszechnego wdrożenia. Wraz z pojawiением się prawdziwie bezpłatnego i otwartego oprogramowania do tworzenia kontenerów wykorzystanie zarówno kontenerów, jak i Kubernetes rosło w niesamowitym tempie.

Pod koniec 2017 r. walki producentów zakończyły się, Kubernetes wygrał. Inne systemy są nadal w użyciu, jednak od teraz firmy, które chcą przenieść swoją infrastrukturę do kontenerów, stawiają tylko na jedną platformę: Kubernetes.

Co sprawia, że Kubernetes jest tak cenny?

Kelsey Hightower, rzecznik programistów Google, współautor *Kubernetes Up & Running* (O'Reilly) i legenda społeczności Kubernetes, przedstawia to w następujący sposób.

Kubernetes robi rzeczy, które zrobiłyby najlepszy administrator systemu, takie jak automatyzacja, obsługa awarii, monitorowanie. Przejmuje wszystkie nauki ze społeczności DevOps i sprawia, że stają się domyślne.

— Kelsey Hightower

Wiele tradycyjnych zadań sysadminów, takich jak aktualizowanie serwerów, instalowanie poprawek bezpieczeństwa, konfigurowanie sieci i wykonywanie kopii zapasowych, nie stanowi problemu w środowisku chmurowym. Kubernetes może zautomatyzować te rzeczy, aby Twój zespół mógł skoncentrować się na wykonywaniu swojej podstawowej pracy.

Niektóre z tych funkcji, np. równoważenie obciążenia i automatyczne skalowanie, są wbudowane w rdzeń Kubernetes; inne są dostarczane przez dodatki, rozszerzenia i narzędzia innych firm korzystające z interfejsu API Kubernetes. Ekosystem Kubernetes jest duży i cały czas rośnie.

Kubernetes ułatwia wdrażanie

Z tych powodów załoga DevOps uwielbia Kubernetes. Są też pewne znaczące zalety dla programistów. Kubernetes znacznie skraca czas i wysiłek potrzebny do wdrożenia. W Kubernetes wdrożenia bez przestojów są powszechnie, ponieważ Kubernetes domyślnie wykonuje aktualizacje ciągłe (uruchamia kontenery z nową wersją, następnie czeka, aż będą gotowe, i zamyka stare).

Kubernetes zapewnia również udogodnienia, które pomagają wprowadzić praktyki ciągłego wdrażania, takie jak *canary deployments*, czyli stopniowe wdrażanie aktualizacji na serwer, tak aby wcześnie wychwycić problemy (patrz „Wdrożenia kanarkowe” w rozdziale 13.). Inną powszechną praktyką są wdrożenia metodą blue-green: równolegle uruchamianie nowej wersji systemu i przelączanie do niego ruchu po jego pełnym uruchomieniu (patrz „Wdrożenia niebiesko-zielone” w rozdziale 13.).

Nagle skoki zapotrzebowania na zasoby nie będą już obniżać poziomu usług, ponieważ Kubernetes obsługuje automatyczne skalowanie. Jeśli np. wykorzystanie procesora przez kontener osiągnie pewien poziom, Kubernetes może dodawać nowe repliki kontenera, dopóki wykorzystanie nie spadnie poniżej progu. Kiedy zapotrzebowanie spadnie, Kubernetes ponownie zmniejszy ilość replik, uwalniając pojemność klastra do uruchamiania innych prac.

Ponieważ Kubernetes ma wbudowaną redundancję i przełączanie awaryjne (ang. *failover*), aplikacja będzie bardziej niezawodna i odporna. Niektóre usługi zarządzane mogą nawet skalować sam kластer Kubernetes w górę i w dół w odpowiedzi na zapotrzebowanie, dzięki czemu nigdy nie płacisz za większy kластer niż potrzebujesz w danym momencie (patrz „Automatyczne skalowanie” w rozdziale 6.).

Firma pokocha Kubernetes, ponieważ obniża koszty infrastruktury i znacznie lepiej wykorzysta dany zestaw zasobów. Tradycyjne serwery, nawet serwery w chmurze, są zwykle bezczynne przez większość czasu. Nadwyżka mocy potrzebna do obsługi skoków zapotrzebowania na zasoby jest zasadniczo marnowana w normalnych warunkach.

Kubernetes wykorzystuje tę zmarnowaną pojemność i używa jej do uruchamiania zadań, a zatem maszyny są znacznie lepiej wykorzystywane — dodatkowo zyskujemy skalowanie, równoważenie obciążenia oraz obsługę awarii.

Chociaż niektóre z tych funkcji, takie jak automatyczne skalowanie, były dostępne przed Kubernetes, zawsze były powiązane z konkretnym dostawcą usług chmurowych lub usługą. Kubernetes działa w sposób niezależny od dostawcy (ang. *provider-agnostic*): po zdefiniowaniu zasobu do wykorzystania możesz go uruchomić w dowolnym klastrze Kubernetes, w sposób niezależny od dostawcy rozwiązań chmurowych.

To nie znaczy, że Kubernetes ogranicza Cię do najniższego wspólnego mianownika. Kubernetes mapuje Twoje zasoby do odpowiednich funkcji specyficznych dla dostawcy: np. dla usługi typu load-balance Kubernetes w Google Cloud zostanie utworzony jej odpowiednik, a w Amazon powstanie usługa AWS load balancer. Kubernetes wyodrębnia szczegóły dotyczące chmury, umożliwiając skupienie się na określeniu zachowania aplikacji.

Podobnie jak kontenery są przenośnym sposobem definiowania oprogramowania, zasoby Kubernetes zapewniają przenośną definicję działania tego oprogramowania.

Czy Kubernetes zniknie?

Co ciekawe, pomimo emocji, które obecnie budzi Kubernetes, naprawdę możemy o nim zapomnieć w najbliższych latach. Wiele rzeczy, które kiedyś były nowe i rewolucyjne, stanowi obecnie dużą część innych technologii, np: mikroprocesory, mysz, internet.

Kubernetes prawdopodobnie również zniknie i stanie się częścią infrastruktury. Gdy jednak dowiesz się, co musisz wiedzieć, aby wdrożyć aplikację w Kubernetes, będziesz posiadać już podstawową wiedzę.

Przyszłość Kubernetes prawdopodobnie leży w dużej mierze w dziedzinie usług zarządzanych. Wirtualizacja, która była kiedyś eksytującą nową technologią, teraz stała się po prostu narzędziem. Zamiast uruchamiać własną platformę wirtualizacji, taką jak vSphere lub Hyper-V, większość osób wynajmuje maszyny wirtualne od dostawcy chmury.

Uważamy, że w podobny sposób Kubernetes stanie się standardową częścią innej technologii.

Kubernetes nie załatwia wszystkiego

Czy infrastruktura przyszłości będzie w całości oparta na Kubernetes? Prawdopodobnie nie. Po pierwsze, niektóre rzeczy po prostu nie pasują do Kubernetes (np. bazy danych).

Oprogramowanie do orkiestracji kontenerów obejmuje rozwijanie nowych wymiennych instancji bez konieczności koordynacji między nimi. Jednak repliki baz danych nie są wymienne; każda z nich ma unikalny stan, a wdrożenie repliki bazy danych wymaga koordynacji innych węzłów w celu zapewnienia, że zmiany obejmą wszystkie miejsca.

— Sean Loiselle (<https://www.cockroachlabs.com/blog/kubernetes-state-of-stateful-apps>)
(Karakuchi Labs)

Chociaż Kubernetes pozwala na uruchamianie prac stanowych (ang. *stateful workloads*), takich jak bazy danych, z niezawodnością klasy korporacyjnej, wymaga to dużego nakładu czasu — więc pozbawione jest sensu (patrz „Uruchom mniejsze oprogramowanie” w rozdziale 3.). Korzystanie z usług zarządzanych jest zwykle bardziej opłacalne.

Po drugie, niektóre zadania nie potrzebują uruchomienia Kubernetes i mogą działać na tzw. platformach bezserwerowych, czyli na zasadzie *funkcja jako usługa* (FaaS — ang. *Functions as a Service*).

Funkcje chmurowe i funtainery

Przykładowo AWS Lambda jest platformą FaaS, która pozwala na uruchamianie kodu napisanego w językach Go, Python, Java, Node.js, C # i innych, bez konieczności komplikacji. Amazon robi to wszystko za Ciebie.

Ponieważ rachunek za korzystanie z usługi naliczany jest za każde 100 milisekund, model FaaS jest idealny do obliczeń, które działają tylko wtedy, gdy są potrzebne. Natomiast za wynajęcie serwera w chmurze płacisz raz i działa on przez cały czas, niezależnie od tego, czy go używasz, czy nie.

Takie *funkcje chmurowe* są pod pewnymi względami wygodniejsze niż kontenery (choć niektóre platformy FaaS mogą również uruchamiać kontenery). Jednak najlepiej nadają się do krótkich, niezależnych zadań (np. AWS Lambda ogranicza funkcje do 15 minut działania i około 50 MiB dla plików), zwłaszcza tych, które integrują się z istniejącymi usługami obliczeniowymi w chmurze, takimi jak Microsoft Cognitive Services lub interfejs API Google Cloud Vision.

Dlaczego nie lubimy nazywać tego modelu „bezserwerowym”? Ponieważ taki nie jest, gdyż to jestczył serwer. Chodzi o to, że nie musisz obsługiwać i utrzymywać tego serwera; dostawca usług chmurowych dba o to za Ciebie.

Nie dla każdego zadania odpowiednie jest działanie na platformach FaaS, ale nadal prawdopodobnie będzie kluczową technologią dla chmurowych aplikacji natywnych w przyszłości.

Funkcje chmurowe nie są ograniczone do publicznych platform FaaS, takich jak Lambda lub Azure Functions: jeśli masz już klaster Kubernetes i chcesz na nim uruchamiać aplikacje FaaS, OpenFaaS (<https://www.openfaas.com/>) i inne projekty open source, jest to możliwe. Taka hybryda funkcji i kontenerów jest czasem nazywana *funtainerami*.

Obecnie jest w fazie rozwoju *Knative*, bardziej zaawansowana platforma dostawy oprogramowania dla Kubernetes, która obejmuje zarówno kontenery, jak i funkcje chmurowe (patrz „*Knative*” w rozdziale 13.). To bardzo obiecujący projekt, co może oznaczać, że w przyszłości rozróżnienie między kontenerami a funkcjami może całkowicie się zacierać lub zanikać.

Model Cloud Native

Termin *cloud native* staje się coraz bardziej popularny w kontekście mówienia o nowoczesnych aplikacjach i usługach korzystających z chmury, kontenerów oraz orkiestracji, często opartych na oprogramowaniu open source.

Organizacja Cloud Native Computing Foundation (CNCF) (<https://www.cncf.io/>) została założona w 2015 r., aby „wspierać społeczność wokół konstelacji wysokiej jakości projektów, które orkiestrują kontenery jako część architektury mikrouslug” (<https://www.cncf.io/about/join/>).

CNCF, część organizacji Linux Foundation, istnieje po to, aby skupiać programistów, użytkowników końcowych i dostawców, w tym głównych dostawców publicznych chmur obliczeniowych. Najbardziej znanym projektem pod patronatem CNCF jest sam Kubernetes, ale fundacja również inkubuje i promuje inne kluczowe elementy rodzimego ekosystemu chmurowego, takie jak Prometheus, Envoy, Helm, Fluentd, gRPC i wiele innych.

Co dokładnie rozumiemy pod pojęciem *cloud native*? Jak większość takich terminów, oznacza to różne rzeczy dla różnych ludzi, ale być może istnieje jakaś wspólna płaszczyzna.

Aplikacje *cloud native* działają w chmurze; to nie jest kontrowersyjne. Jednak samo uruchomienie istniejącej aplikacji w instancji chmury obliczeniowej nie powoduje, że jest ona natywna. Nie chodzi również o uruchamianie w kontenerze lub korzystanie z usług w chmurze, takich jak Azure Cosmos DB lub Google Pub / Sub, chociaż mogą to być ważne aspekty aplikacji typu *cloud native*.

Spójrzmy więc na kilka cech charakterystycznych dla systemów *cloud native*, na które większość ludzi może się zgodzić.

Zdolność do automatyzacji

Jeśli aplikacje mają być wdrażane i zarządzane przez maszyny, a nie przez ludzi, muszą przestrzegać wspólnych standardów, formatów i interfejsów. Kubernetes zapewnia te standardowe interfejsy w taki sposób, że programiści aplikacji nie muszą się o nie martwić.

Wszechobecność i elastyczność

Ponieważ są one oddzielone od zasobów fizycznych, takich jak dyski, lub jakiekolwiek konkretnej wiedzy na temat węzła obliczeniowego, na którym akurat się uruchamiają, mikrosługi w kontenerach można łatwo przenosić z jednego węzła do drugiego, a nawet z jednego klastra do drugiego.

Odporność i skalowalność

W tradycyjnych aplikacjach występują pojedyncze punkty awarii: aplikacja przestaje działać, jeśli jej główny proces ulegnie awarii, jeśli komputer ulegnie awarii lub przy braku dostępu do zasobów. Aplikacje *cloud native* są z natury rozproszone, zapewniają wysoką dostępność dzięki redundancji i płynnemu rozkładowi.

Dynamiczność

Orkiestrator kontenerów, taki jak Kubernetes, może rozplanować użycie kontenerów tak, aby maksymalnie wykorzystać dostępne zasoby. Aby osiągnąć wysoką dostępność, może uruchomić wiele kopii. Może także przeprowadzać ciągłe aktualizacje, aby płynnie aktualizować usługi bez zmniejszania ruchu.

Obserwowanie

Aplikacje cloud native w swojej naturze są trudniejsze do sprawdzania i debugowania. Zatem kluczowym wymaganiem systemów rozproszonych jest obserwowanie — monitorowanie, rejestrowanie, śledzenie — czyli wszystkie pomiary, które pomagają inżynierom zrozumieć, co robią ich systemy (oraz co robią źle).

Rozproszenie

Cloud native to podejście do budowania i uruchamiania aplikacji, które wykorzystuje rozproszony i zdecentralizowany charakter chmury. Chodzi o to, jak działa Twoja aplikacja, a nie gdzie działa. Zamiast wdrażać kod jako pojedynczą jednostkę (zwaną *monolitem* — ang. *monolith*), aplikacje cloud native zwykle składają się z wielu współpracujących, rozproszonych *mikrouslug*. Mikrousluga to po prostu niezależna usługa, która robi jedną rzecz. Jeśli połączysz wystarczającą liczbę mikrouslug, otrzymasz aplikację.

Nie chodzi tylko o mikrouslugi

Jednak mikrouslugi nie są panaceum. Monolity są łatwiejsze do zrozumienia, ponieważ wszystko jest w jednym miejscu i można śledzić interakcje różnych części. Natomiast trudno skalować monolit zarówno pod względem samego kodu, jak i zespołów programistów, którzy go utrzymują. W miarę wzrostu kodu interakcje między jego różnymi częściami rosną wykładniczo, a system jako całość przestaje być czytelny.

Aplikacja cloud native składa się z mikrouslug, ale decydujące, jakie powinny być te mikrouslugi, gdzie są granice i jak różne usługi mają współpracować, nie jest łatwym problemem. Dobry projekt usług cloud native polega na dokonywaniu mądrych wyborów dotyczących dzielenia różnych części architektury. Jednak nawet dobrze zaprojektowana aplikacja cloud native jest nadal systemem rozproszonym, co czyni go z natury złożonym, trudnym do zaobserwowania i uzasadnienia oraz podatnym na awarie.

Chociaż systemy cloud native są zwykle rozprozone, nadal można uruchamiać monolityczne aplikacje w chmurze, używając kontenerów. Może to być krok na drodze do stopniowej migracji części monolitu na zewnątrz do nowoczesnych mikrouslug lub chwilowe oczekiwanie na przeprojektowanie systemu, tak aby był w pełni cloud native.

Przyszłość operacji

Operacje, zarządzanie infrastrukturą i administracja systemem to prace wymagające wysokich kwalifikacji. Czy taki personel jest zagrożony z uwagi na przyszłość cloud native? Myślimy, że nie.

Umiejętności te staną się jeszcze ważniejsze. Projektowanie systemów rozproszonych jest trudne. Sieci i orkiestratory kontenerów są skomplikowane. Każdy zespół tworzący aplikacje cloud native

będzie potrzebował umiejętności operacyjnych oraz wiedzy. Automatyzacja uwalnia pracowników od nudnej, powtarzalnej pracy ręcznej, aby radzili sobie z bardziej złożonymi oraz interesującymi problemami, których komputery nie są jeszcze w stanie rozwiązać samodzielnie.

Nie oznacza to, że wszystkie prace związane z zadaniami operacyjnymi są gwarantowane. Sysadmi potrafieli sobie radzić bez umiejętności kodowania, z wyjątkiem tworzenia skryptów w powłoce. W środowisku chmurowym to nie zadziała.

W świecie zdefiniowanym przez oprogramowanie umiejętność pisania, rozumienia i utrzymywania oprogramowania staje się krytyczna. Jeśli nie możesz opanować nowych umiejętności lub nie nauczysz się ich, świat Cię porzuci — zawsze tak było.

Rozproszone DevOps

Zamiast koncentrować się w jednym zespole operacyjnym, który obsługuje inne zespoły, wiedza operacyjna zostanie podzielona między wiele zespołów.

Każdy zespół programistów będzie potrzebował co najmniej jednego specjalisty ds. operacyjnych, odpowiedzialnego za stan systemów lub usług świadczonych przez zespół. Będzie także programistą, ale też ekspertem w dziedzinie sieci, Kubernetes, wydajności, odporności oraz narzędzi i systemów, które umożliwiają innym programistom dostarczanie kodu do chmury.

Dzięki rewolucji DevOps w większości organizacji nie będzie już miejsca dla programistów, którzy nie mogą realizować zadań operacyjnych, lub operatorów, którzy nie potrafią programować. Różnica między tymi dwoma zawodami szybko znika. Tworzenie i obsługa oprogramowania to zaledwie dwa aspekty tego samego.

Niektóre rzeczy pozostaną scentralizowane

Czy są ograniczenia wynikające ze stosowania DevOps? Może tradycyjny zespół operacji IT zniknie całkowicie, zamieniając się w grupę konsultantów wewnętrznych czy też ekspertów ds. coachingu?

Myślimy, że nie, a przynajmniej nie do końca. Niektóre rzeczy nadal korzystają z centralizacji. Nie ma sensu, aby każdy zespół ds. aplikacji lub usług miał własny sposób wykrywania zdarzeń oraz komunikowania np. o zdarzeniach produkcyjnych, taki jak własny system obsługi klienta czy narzędzia potrzebne do wdrożeń. Nie ma sensu wymyślać czegoś od początku.

Produktywni programiści

Chodzi o to, że samoobsługa ma swoje granice, a celem metodyki DevOps jest przyspieszenie pracy zespołów programistycznych, a nie spowalnianie ich niepotrzebną i zbędną pracą.

Tak, duża część tradycyjnych operacji może i powinna zostać przekazana innym zespołom, przede wszystkim tym, które dotyczą wdrażania kodu i reagowania na zdarzenia związane z kodem. Aby jednak tak się stało, musi istnieć silny, centralny zespół, który wspiera ekosystem DevOps, w jakim działają wszystkie pozostałe zespoły.

Aby nie nazywać tego zespołu operacyjnym, użyjemy nazwy *produktywni programiści* (DPE — ang. *developer productivity engineering*). Zespoły DPE robią wszystko, co konieczne, aby pomóc programistom w szybszym wykonywaniu pracy; zajmują się obsługą infrastruktury, narzędziami budowania, problemami z awarią.

I chociaż DPE posiada specjalistyczny zestaw umiejętności, sami inżynierowie mogą przenieść się do organizacji, aby dostarczyć tę wiedzę tam, gdzie jest potrzebna.

Matt Klein inżynier firmy Lyft zasugerował, że chociaż czysty model DevOps ma sens dla startupów i małych firm, to wraz z rozwojem organizacji zachodzi naturalna tendencja przenoszenia ekspertów ds. infrastruktury i niezawodności do zespołu centralnego. Twierdzi jednak, że zespołu nie można skalować w nieskończoność.

Do czasu, gdy zespół inżynierijny osiągnie ~ 75 osób, prawie na pewno działa centralny zespół ds. infrastruktury, który zaczyna budować wspólne właściwości wymagane przez zespoły tworzące mikrousługi. Jednak przychodzi moment, w którym centralny zespół ds. infrastruktury nie może dłużej zarówno budować infrastruktury krytycznej, jak i ją obsługiwać, jednocześnie utrzymując wsparcie dla zespołów ds. produktów w wykonywaniu zadań operacyjnych.

— Matt Klein (<https://medium.com/@mattklein123/the-human-scalability-of-devops-e36c37d3db6a>)

W tym momencie nie każdy programista może być ekspertem infrastruktury, tak jak pojedynczy zespół ekspertów ds. infrastruktury nie jest w stanie obsłużyć stale rosnącej liczby programistów. W większych organizacjach, mimo że centralny zespół ds. infrastruktury jest nadal potrzebny, istnieje również możliwośćłączenia inżynierów ds. niezawodności (SRE — ang. *site reliability engineer*) do każdego zespołu programistów lub produktów. Wnoszą swoją wiedzę do każdego zespołu jako konsultanci, a także stanowią pomost między rozwojem produktu a operacjami związanymi z infrastrukturą.

Jesteś przyszłością

Jeśli czytasz tę książkę, oznacza to, że będziesz częścią rodziny cloud native. W pozostałych rozdziałach omówimy całą wiedzę i umiejętności, których będziesz potrzebować jako programista lub inżynier operacyjny pracujący z infrastrukturą chmurową, kontenerami oraz Kubernetes.

Niektóre z tych rzeczy będą znane, a niektóre nowe. Mamy jednak nadzieję, że kiedy zapoznasz się z materiałem zawartym w książce, poczujesz się bardziej pewny własnych umiejętności związanych z cloud native. Tak, jest wiele do nauczenia się, ale dasz radę.

Czytaj dalej.

Podsumowanie

Zapoznałeś się z krótką prezentacją chmurowego środowiska DevOps. Mamy nadzieję, że wystarczy ona, abyś nabrał pewności, że będziesz sprawniej rozwiązywał problemy związane z środowiskiem chmurowym, kontenerami oraz Kubernetes.

Poniżej zamieściliśmy krótkie podsumowanie głównych tematów — potem przejdziesz do następnego rozdziału i zapoznasz się z Kubernetes.

- Przetwarzanie w chmurze uwalnia Cię od kosztów zarządzania własnym sprzętem, umożliwiając budowanie odpornych, elastycznych, skalowalnych systemów rozproszonych.
- DevOps zapewnia, że współczesne tworzenie oprogramowania nie kończy się w momencie napisania kodu: chodzi o utworzenie pętli sprzężenia zwrotnego między tymi, którzy piszą kod, a tymi, którzy go używają.
- DevOps wprowadza podejście zorientowane na kod oraz dobre praktyki inżynierii oprogramowania w świecie infrastruktury i operacji.
- Kontenery umożliwiają wdrażanie i uruchamianie oprogramowania w małych, znormalizowanych, samodzielnych jednostkach. To sprawia, że budowanie dużych, różnorodnych, rozproszonych systemów jest łatwiejsze i tańsze dzięki połączeniu mikrousług kontenerowych.
- Systemy orkiestracji zajmują się rozmieszczeniem kontenerów, planowaniem, skalowaniem, tworzeniem sieci i wszystkim, co zrobiłby dobry administrator systemu, ale w sposób zautomatyzowany i programalny.
- Kubernetes to de facto standardowy system orkiestracji kontenerów — gotowy do użycia już dziś w produkcji.
- Cloud native jest skrótem przydatnym, kiedy mówimy o chmurowych, kontenerowych, rozproszonych systemach, złożonych ze współpracujących mikrousług, dynamicznie zarządzanych przez zautomatyzowaną infrastrukturę w postaci kodu.
- Umiejętności związane z operacjami oraz obsługą infrastruktury, które nie są uważane za przestarzałe w terminologii cloud native, są i będą ważniejsze niż kiedykolwiek.
- Nadal istnieje sens, aby centralny zespół budował i utrzymywał platformy oraz narzędzia, które umożliwiają skorzystanie z metodyki DevOps wszystkim pozostałym zespołom.
- Zaniknie wyraźne rozróżnienie między inżynierami oprogramowania a inżynierami operacji. Od teraz wszystko związane jest z oprogramowaniem i wszyscy jesteśmy inżynierami.

Pierwsze kroki z Kubernetes

Aby zrobić coś naprawdę wartościowego, nie myśl o deszczach, zimie oraz niebezpieczeństwie, ale idę dalej z entuzjazmem, najlepiej, jak potrafię.

— Og Mandino

Dosyć teorii; zacznijmy pracę z Kubernetes i kontenerami. W tym rozdziale zbudujesz prostą aplikację kontenerową i wdrożysz ją w lokalnym klastrze Kubernetes działającym na Twoim komputerze. W tym rozdziale poznasz też kilka bardzo ważnych technologii cloud native; będą to Docker, Git, Go, rejestr kontenerów oaz narzędzie kubectl.



Ten rozdział jest interaktywny! Często w tej książce poprosimy Cię o podążanie za przykładami. Będziesz proszony o zainstalowanie różnych rzeczy na własnym komputerze, wpisywanie poleceń oraz uruchamianie kontenerów. Uważamy, że jest to o wiele bardziej skuteczny sposób nauki niż prosty opis słowny.

Uruchamianie pierwszego kontenera

Jak napisaliśmy w rozdziale 1., użycie kontenerów jest jedną z kluczowych koncepcji cloud native. Podstawowym narzędziem do budowania i uruchamiania kontenerów jest Docker. W tym rozdziale zbudujemy prostą aplikację i uruchomimy ją lokalnie. Obraz aplikacji przekażemy do rejestru kontenera. Skorzystamy z narzędzia Docker Desktop.

Jeśli już znasz kontenery, przejdź bezpośrednio do podrozdziału „Cześć, Kubernetes”, w którym zaczyna się prawdziwa zabawa. Jeśli chcesz dowiedzieć się, czym są kontenery i jak działają, oraz zdobyć trochę praktycznej wiedzy, czytaj dalej.

Instalowanie Docker Desktop

Docker Desktop to kompletne środowisko programistyczne Kubernetes dla komputerów pracujących w systemach Mac lub Windows, które działają na Twoim laptopie (lub komputerze stacjonarnym). Zawiera jednowęzłowy klaster Kubernetes, którego można używać do testowania aplikacji.

Zainstalujesz teraz Docker Desktop i użyjesz go do uruchomienia prostej aplikacji w kontenerze. Jeśli masz już zainstalowany Docker, pomiń ten punkt i przejdź od razu do punktu „Uruchamianie obrazu kontenera”.

Pobierz odpowiednią dla swojego komputera wersję Docker Desktop Community Edition (<https://hub.docker.com/search/?type=edition&offering=community>). Następnie postępuj zgodnie z instrukcjami.



Docker Desktop nie jest obecnie dostępny dla Linuksa. Dlatego użytkownicy tego systemu będą musieli zainstalować oprogramowanie Docker Engine (<https://www.docker.com/products/docker-engine>), a następnie Minikube (patrz podrozdział „Minikube” w tym rozdziale).

Gdy to zrobisz, otwórz terminal i uruchom następującą komendę (powinno się udać):

```
docker version
Client:
  Version: 19.03.8
...
...
```

Dokładne dane wyjściowe będą się różnić w zależności od platformy, ale jeśli Docker jest poprawnie zainstalowany i uruchomiony, zobaczysz komunikat podobny do powyższego. W systemach Linux może być konieczne uruchomienie polecenia z uprawnieniami superużytkownika (czyli root): sudo docker version.

Co to jest Docker?

Docker (<https://docs.docker.com/>) to kilka różnych, ale powiązanych rzeczy: format obrazu kontenera, biblioteka środowiska *wykonawczego kontenera*, która zarządza cyklem życia kontenerów, narzędzie wiersza polecenia do pakowania i uruchamiania kontenerów oraz interfejs API do zarządzania kontenerami. Szczegóły nie są ważne, ponieważ Kubernetes używa środowiska Docker jako jednego z wielu komponentów.

Uruchamianie obrazu kontenera

Czym dokładnie jest obraz kontenera? Dla naszych celów szczegóły techniczne nie mają znaczenia, ale możesz myśleć o obrazie jak o pliku ZIP. Jest to pojedynczy plik binarny, który ma unikalny identyfikator i zawiera wszystko, czego potrzeba do uruchomienia kontenera.

Niezależnie od tego, czy korzystasz z kontenera bezpośrednio za pomocą środowiska Docker, czy w klastrze Kubernetes, wszystko, co musisz podać, to ID obrazu kontenera lub adres URL, a system zajmie się znalezieniem, pobraniem, rozpakowaniem i uruchomieniem kontenera za Ciebie.

Napisaliśmy małą aplikację, z której będziemy korzystać w całej książce, aby zilustrować, o czym mówimy. Możesz pobrać i uruchomić aplikację za pomocą wcześniej przygotowanego obrazu kontenera. Uruchom następujące polecenie, aby spróbować:

```
docker container run -p 9999:8888 - name hello cloudnativized /demo:hello
```

Pozostaw to polecenie uruchomione i na przeglądarce otwórz adres <http://localhost:9999>.

Powinieneś zobaczyć komunikat:

```
Hello, 世界
```

Za każdym razem, gdy podasz ten adres URL, nasza aplikacja będzie wysyłała do Ciebie powyższe powitanie.

Gdy nacieszysz się już tym widokiem, zatrzymaj kontener, naciskając *Ctrl+C* w swoim terminalu.

Aplikacja demonstracyjna

Jak to działa? Pobierz kod źródłowy aplikacji demonstracyjnej działającej w tym kontenerze i zobacz.

W tej części musisz zainstalować narzędzie Git¹. Jeśli nie masz pewności, czy masz już to narzędzie, spróbuj wykonać następujące polecenie:

```
git version
git version 2.25.1
```

Jeśli nie masz jeszcze narzędzia Git, postępuj zgodnie z instrukcjami instalacji (<https://git-scm.com/download>) dla Twojej platformy.

Po zainstalowaniu narzędzia Git wykonaj następujące polecenie:

```
git clone https://github.com/cloudnativedevelopers/demo.git
Cloning into 'demo' ...
...
```

Oglądamy kod źródłowy

Repozytorium Git zawiera aplikację demonstracyjną, z której będziemy korzystać w tej książce. Aby łatwiej można było zobaczyć, jakie zmiany zachodzą na konkretnym etapie, repozytorium zawiera każdą kolejną wersję aplikacji w innym podkatalogu. Pierwsza nazywa się po prostu *hello*. Aby zobaczyć kod źródłowy, skorzystaj z polecenia:

```
cd demo / hello
ls
Dockerfile README.md
go.mod main.go
```

Otwórz plik *main.go* w swoim ulubionym edytorze (zalecamy Visual Studio Code — <https://code.visualstudio.com/>), bo obsługuje Go, Docker i Kubernetes). Zobaczysz poniższy kod źródłowy:

```
package main
import (
    "fmt"
    "log"
    "net/http"
)
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, 世界")
}
func main() {
```

¹ Jeśli nie znasz narzędzia Git, przeczytaj znakomitą książkę Scotta Chacona i Benia Strauba *Pro Git* (<https://git-scm.com/book/en/v2>) (Apress).

```
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8888", nil))
}
```

Wprowadzenie do języka Go

Nasza aplikacja demonstracyjna napisana jest w języku programowania Go.

Go to nowoczesny język programowania (opracowany w Google w 2009 r.). Jego cechy to prostota, bezpieczeństwo i czytelność. Jest przeznaczony do budowania współbieżnych aplikacji na dużą skalę, zwłaszcza usług sieciowych. Programowanie w nim daje dużo frajdy².

W Go napisany jest Kubernetes, podobnie jak Docker, Terraform i wiele innych popularnych projektów open source. To sprawia, że Go jest dobrym wyborem do tworzenia aplikacji cloud native.

Jak działa aplikacja demonstracyjna?

Jak widać, aplikacja demonstracyjna jest dość prosta, mimo że implementuje serwer HTTP (Go jest wyposażony w potężną bibliotekę standardową). Jego rdzeniem jest funkcja o nazwie `handler`:

```
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, 世界")
}
```

Jak sama nazwa wskazuje, powyższa funkcja obsługuje żądania HTTP. Żądanie jest przekazywane jako argument do funkcji (choć funkcja nic z tym nie robi).

Serwer HTTP potrzebuje również sposobu, aby wysłać coś z powrotem do klienta. Umożliwia to obiekt `http.ResponseWriter` — odsyła on użytkownikowi wiadomość do wyświetlenia w przeglądarce: w tym przypadku tylko ciąg `Hello, 世界`.

Pierwszy przykładowy program, w dowolnym języku, tradycyjnie wyświetla ciąg znaków: `Hello, world`. Ponieważ jednak język Go natywnie obsługuje kodowanie Unicode (międzynarodowy standard reprezentacji tekstu), przykładowe programy Go często drukują `Hello 世界`, aby taką możliwość zaprezentować. Jeśli nie mówisz po chińsku, to w porządku: Go potrafi!

Reszta programu zajmuje się rejestracją handlера funkcji jako handlера dla żądań HTTP i uruchamia serwer HTTP na porcie 8888.

To cała aplikacja! Nie robi jeszcze wiele, ale dodamy inne możliwości.

Budowanie kontenera

Już wiesz, że obraz kontenera to pojedynczy plik, który zawiera wszystko, czego kontener potrzebuje do uruchomienia. Jak w ogóle zbudować obraz? Aby to zrobić, użyj polecenia `docker image build`, które przyjmuje jako dane wejściowe specjalny plik tekstowy o nazwie `Dockerfile`. Plik `Dockerfile` określa dokładnie, co musi znajdować się w obrazie kontenera.

² Jeżeli jesteś dość doświadczonym programistą, ale nie znasz języka Go, zapoznaj się z przewodnikiem Alana Donovana i Briana Kernighana *The Go Programming Language* (<https://www.gopl.io/>) (Addison-Wesley).

Jedną z kluczowych zalet kontenerów jest możliwość korzystania z istniejących obrazów w celu tworzenia nowych obrazów. Możesz np. wziąć obraz kontenera zawierający kompletny system operacyjny Ubuntu, dodać do niego pojedynczy plik, a wynikiem będzie nowy obraz.

Generalnie plik *Dockerfile* zawiera instrukcje dotyczące tworzenia obrazu początkowego (tzw. obrazu podstawowego — ang. *base image*), przekształcania go w jakiś sposób i zapisywania wyniku jako nowego obrazu.

Opis plików Dockerfile

Przyjrzyjmy się plikowi *Dockerfile* z naszej aplikacji demonstracyjnej (znajduje się w podkatalogu *hello* repozytorium aplikacji):

```
FROM golang:1.14-alpine AS build

WORKDIR /src/
COPY main.go go.* /src/
RUN CGO_ENABLED=0 go build -o /bin/demo

From scratch
COPY --from=build /bin/demo /bin/demo
ENTRYPOINT ["/bin/demo"]
```

Dokładny sposób działania nie ma na razie znaczenia, ale wykorzystywany jest standardowy proces budowania kontenerów Go, zwany *multi-stage builds*. Pierwszy etap rozpoczyna się od oficjalnego obrazu kontenera *golang*, który zawiera tylko system operacyjny (w tym przypadku Alpine Linux) zainstalowanym językiem Go. Do komplikacji pliku *main.go*, który widzieliśmy wcześniej, wykonywane jest polecenie `go build`.

Wynikiem tych operacji jest wykonywalny plik binarny o nazwie *demo*. W drugim etapie operacji pobierany jest pusty obraz kontenera (zwany obrazem *scratch* — instrukcja `from scratch`) i kopiowany do niego plik binarny *demo*.

Minimalne obrazy kontenerów

Do czego potrzebny jest drugi etap komplikacji? Cóż, środowisko językowe Go i reszta systemu Alpine Linux są naprawdę potrzebne tylko do zbudowania programu. Aby uruchomić program, wystarczy plik binarny, więc plik *Dockerfile* tworzy nowy pusty kontener, w którym można go umieścić. Wynikowy obraz jest bardzo mały (zajmuje około 6 MiB) — i to jest obraz, który można wdrożyć w produkcji.

Bez drugiego etapu powstałby obraz kontenera o wielkości około 350 MiB, którego 98% jest niepotrzebne i nigdy nie zostanie wykonane. Im mniejszy obraz kontenera, tym szybciej można go załadować i pobrać, i tym szybciej będzie można uruchomić.

Kontenery o mniejszej pojemności mają również mniejszą *podatność na ataki*. Im mniej programów jest w Twoim kontenerze, tym mniej potencjalnych luk.

Ponieważ Go to skompilowany język, który może tworzyć pliki samowykonywalne, idealnie nadaje się do pisania minimalnych kontenerów. Dla porównania, oficjalny obraz kontenera Ruby zajmuje 1,5 GiB; jest około 250 razy większy niż nasz obraz Go — jeszcze przed dodaniem programu!

Uruchamianie skompilowanego obrazu Docker

Wiesz już, że plik *Dockerfile* zawiera instrukcje dla narzędzia `docker image build` — do zmiany naszego kodu źródłowego Go w wykonywalny kontener. Pójdzmy dalej. W katalogu *hello* podaj następujące polecenie:

```
docker image build -t myhello .
Sending build context to Docker daemon 4.096kB
Step 1/7 : FROM golang:1.14-alpine AS build
...
Successfully built eeb7d1c2e2b7
Successfully tagged myhello:latest
```

Gratulacje, właśnie zbudowałeś swój pierwszy kontener! Na podstawie danych wyjściowych widać, że Docker wykonuje kolejno każdą akcję z pliku *Dockerfile* na nowo utworzonym kontenerze. W wyniku tych działań powstaje gotowy do użycia obraz.

Nazywanie obrazów

Kiedy budujesz obraz, domyślnie otrzymuje on po prostu szesnastkowy identyfikator. Aby się do niego później odwołać (aby go np. uruchomić), możesz z niego skorzystać. Te identyfikatory nie są szczególnie łatwe do zapamiętania ani proste do wpisania, więc Docker pozwala nadać obrazowi nazwę czytelną dla człowieka. Do tego celu służy przełącznik `-t`, podawany przy instrukcji `docker image build`. W poprzednim przykładzie nazwałeś obraz *myhello*, więc powinieneś użyć tej nazwy do uruchomienia obrazu.

Zobaczmy, czy to działa:

```
docker container run -p 9999:8888 myhello
```

Uruchomłeś właśnie własną kopię aplikacji demonstracyjnej i możesz to sprawdzić, przechodząc pod ten sam adres URL, co poprzednio (<http://localhost:9999/>).

Powinieneś zobaczyć ciąg znaków Hello, 世界. Naciśnij `Ctrl+C`, aby przerwać działanie polecenia `container run`.

Ćwiczenie

Jeśli masz ochotę na przygodę, zmodyfikuj plik *main.go* w aplikacji demonstracyjnej i zmień powitanie, tak aby brzmiało „Hello, world” lub inaczej. Zbuduj ponownie kontener i uruchom go, aby sprawdzić, czy działa.

Gratulacje, jesteś teraz programistą Go! Jednak nie poprzestawaj na tym: przejdź na stronę Tour of Go (<https://tour.golang.org/welcome/1>), aby dowiedzieć się więcej.

Przekierowanie portów (ang. port forwarding)

Programy działające w kontenerze są odizolowane od innych programów działających na tym samym komputerze, co oznacza, że nie mogą mieć bezpośredniego dostępu do zasobów, takich jak porty sieciowe.

Aplikacja demonstracyjna nasłuchuje na porcie 8888, ale jest to prywatny port *kontenera* 8888, a nie port działający na Twoim komputerze. Aby połączyć się z portem 8888 kontenera, musisz przekierować port na komputerze lokalnym do tego portu w kontenerze. Może to być dowolny port, w tym 8888, ale my użyjemy portu o nr 9999, aby wyjaśnić zasadę działania.

Aby Docker przekierował port, możesz użyć przełącznika `-p` — piszemy o tym wcześniej w punkcie „Uruchamianie obrazu kontenera”:

```
docker container run -p HOST_PORT:CONTAINER_PORT ...
```

Po uruchomieniu kontenera wszelkie żądania do portu `HOST_PORT` na komputerze lokalnym będą automatycznie przekazywane do portu `CONTAINER_PORT` w kontenerze — dzięki temu możesz połączyć się z aplikacją za pomocą przeglądarki.

Rejestry kontenerowe

W punkcie „Uruchamianie obrazu kontenera” uruchomiłeś obraz za pomocą jego nazwy. Docker pobrał go automatycznie.

Być może zastanawiasz się, skąd został pobrany. Choć możesz budować i uruchamiać lokalne obrazy, to o wiele bardziej przydatne jest dodawanie i pobieranie obrazów z *rejestru kontenera* (ang. *container registry*). Rejestr pozwala przechowywać obrazy i szukać ich przy użyciu unikalnej nazwy (np. `cloudnativelibrary/demo:hello`).

Domyślnym rejestrem dla polecenia `docker container run` jest Docker Hub, ale można określić inny lub skonfigurować własny.

Na razie pozostanmy przy Docker Hub. Możesz pobrać dowolny publiczny obraz kontenera z Docker Hub, jednak do umieszczania własnych obrazów potrzebujesz konta (o nazwie *Docker ID*). Aby utworzyć Docker ID, postępuj zgodnie z instrukcjami ze strony <https://hub.docker.com/>.

Uwierzytelnianie w rejestrze

Po uzyskaniu Docker ID nastepnym krokiem jest połączenie lokalnego demona Docker z Docker Hub przy użyciu Twojego ID oraz hasła:

```
docker login
```

Zaloguj się za pomocą swojego Docker ID, aby wysyłać i pobierać obrazy z Docker Hub. Jeśli nie masz identyfikatora Docker, przejdź na stronę <https://hub.docker.com/>, aby go utworzyć.

```
Username: TWÓJ_DOCKER_ID  
Password: TWOJE_HASŁO  
Login Succeeded
```

Przydzielanie nazwy i dodawanie obrazu

Aby przesłać lokalny obraz do rejestru, musisz nazwać go przy użyciu formatu TWÓJ_DOCKER_ID/myhello.

Aby utworzyć tę nazwę, nie trzeba odbudowywać obrazu; wystarczy uruchomić to polecenie:

```
docker image tag myhello YOUR_DOCKER_ID/myhello
```

Dzięki temu, po przesłaniu obrazu do rejestru Docker będzie wiedział, na którym koncie go przechowywać.

Za pomocą poniższego polecenia prześlij obraz do Docker Hub:

```
docker image push TWÓJ_DOCKER_ID/myhello
The push refers to repository [docker.io/TWÓJ_DOCKER_ID/myhello]
b2c591f16c33: Pushed
latest: digest:
sha256:7ac57776e2df70d62d7285124fbff039c9152d1bdfb36c75b5933057cefe4fc7
size: 528
```

Uruchamianie obrazu

Gratulacje! Obraz kontenera jest teraz dostępny i można go uruchomić z dowolnego miejsca (przynajmniej z dowolnego miejsca z dostępem do internetu) za pomocą polecenia:

```
docker container run -p 9999:8888 TWÓJ_DOCKER_ID/myhello
```

Cześć, Kubernetes

Po zbudowaniu i przesłaniu pierwszego obrazu kontenera możesz go uruchomić za pomocą polecenia docker container run. Nie jest to zbyt ekscytujące. Zróbmy coś bardziej szalonego i uruchommy go w Kubernetes.

Istnieje wiele sposobów na uzyskanie klastra Kubernetes. Niektóre z nich omówimy bardziej szczegółowo w rozdziale 3. Jeśli masz już dostęp do klastra Kubernetes, to świetnie. Jeśli chcesz, możesz z niego skorzystać w przypadku pozostałych przykładów zawartych w tym rozdziale.

Jeśli nie, nie martw się. Docker Desktop obsługuje Kubernetes (użytkownicy Linuksa muszą zainstalować podrozdział „Minikube”). Aby włączyć usługę Kubernetes, otwórz w Docker Desktop menu Preferences, wybierz kartę Kubernetes i zaznacz opcję Enable (patrz rysunek 2.1).

Instalacja i uruchomienie Kubernetes potrwa kilka minut. Gdy to zrobisz, możesz uruchomić aplikację demonstracyjną!

Uruchamianie aplikacji demonstracyjnej

Zacznijmy od uruchomienia obrazu, który zbudowałeś wcześniej. Otwórz terminal i wpisz polecenie kubectl z następującymi argumentami:

```
kubectl run demo --image=YOUR_DOCKER_ID/myhello --port=9999 --labels app=demo
pod/demo created
```



Rysunek 2.1. Włączanie obsługi Kubernetes w Docker Desktop

Nie przejmuj się na razie szczegółami tego polecenia: jest to w zasadzie odpowiednik komendy Kubernetes dla polecenia docker container run, którego użyłeś wcześniej do uruchomienia obrazu demonstracyjnego. Jeśli nie masz jeszcze własnego obrazu, możesz skorzystać z naszego: `--image=clouchnative/demo:hello`.

Przypomnij sobie, że musiałeś przekierować port 9999 na komputerze lokalnym do portu 8888 kontenera, aby połączyć się z nim za pomocą przeglądarki internetowej. Musisz zrobić to samo tutaj, używając opcji port-forward dla kubectl:

```
kubectl port-forward pod/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

Pozostaw to polecenie uruchomione i otwórz nowy terminal.

Z pomocą przeglądarki połącz się z adresem `http://localhost:9999/`. Zobaczysz komunikat Hello, 世界.

Uruchomienie kontenera i udostępnienie aplikacji może potrwać kilka sekund. Jeśli po około 30 sekundach aplikacja nie będzie dostępna, wypróbuj to polecenie:

```
kubectl get pods --selector app=demo
NAME READY STATUS RESTARTS AGE
demo 1/1 Running 0 9m
```

Gdy kontener jest uruchomiony i możesz się z nim połączyć za pomocą przeglądarki, zobaczysz w terminalu ten komunikat:

```
Handling connection for 9999
```

Jeśli kontener się nie uruchamia

Jeśli w polu STATUS nie jest widoczny napis Running, może to oznaczać jakiś problem. Jeśli pole STATUS ma wartość ErrImagePull lub ImagePullBackoff, oznacza to, że Kubernetes nie był w stanie

znaleźć i pobrać podanego obrazu. Być może zrobiłeś literówkę w nazwie obrazu; sprawdź składnię polecenia `kubectl run`.

Jeśli pole `STATUS` ma wartość `ContainerCreating`, wszystko jest w porządku; Kubernetes nadal pobiera i uruchamia obraz. Poczekaj kilka sekund i sprawdź ponownie.

Gdy zakończysz pracę, usuń demonstracyjną aplikację:

```
kubectl delete pod demo  
pod "demo" deleted
```

Minikube

Jeśli nie chcesz używać obsługi Kubernetes w Docker Desktop lub nie możesz korzystać z niej, istnieje alternatywa, czyli Minikube. Podobnie jak Docker Desktop, Minikube zapewnia kластer Kubernetes z jednym węzłem, który działa na Twoim komputerze (w rzeczywistości na maszynie wirtualnej, ale to nie ma znaczenia).

Aby zainstalować Minikube, postępuj zgodnie z instrukcjami zawartymi na stronie (<https://kubernetes.io/docs/tasks/tools/install-minikube/>).

Podsumowanie

Jeśli, podobnie jak my, szybko niecierpliwisz się długimi wypowiedziami na temat, dlaczego Kubernetes jest taki świetny, mamy nadzieję, że podobało Ci się nasze praktyczne podejście do tematu. Jeśli jesteś już zaawansowanym użytkownikiem narzędzi Docker lub Kubernetes, być może wybaczysz nam taki kurs przygotowawczy. Chcemy mieć pewność, że wszyscy czują się komfortowo w budowaniu i uruchamianiu kontenerów, oraz upewnić się, że posiadasz środowisko Kubernetes — zanim przejdziesz do bardziej zaawansowanych rzeczy.

Oto wiadomości, które powinieneś zapamiętać z tego rozdziału.

- Wszystkie przykłady kodu źródłowego (i wiele innych) są dostępne w repozytorium demonstracyjnym (<https://github.com/cloudnativedevelopers/demo>), powiązanym z tą książką.
- Narzędzie Docker pozwala budować kontenery lokalnie, dodawać je lub pobierać z rejestru kontenerów, takiego jak Docker Hub, oraz uruchamiać obrazy kontenerów lokalnie na komputerze.
- Obraz kontenera jest całkowicie określony w pliku Dockerfile; jest to plik tekstowy, który zawiera instrukcje dotyczące sposobu budowania kontenera.
- Docker Desktop pozwala na uruchomienie małego (jednowęzłowego) klastra Kubernetes na Twoim komputerze, który jest jednak zdolny do uruchamiania dowolnej aplikacji kontenerowej. Minikube to kolejna opcja.
- Narzędzie `kubectl` jest podstawowym narzędziem służącym do interakcji z klastrem Kubernetes i może być używane albo *imperatywnie* (aby np. uruchomić publiczny obraz kontenera i utworzyć niezbędne zasoby Kubernetes), albo *deklaratywnie*, aby zastosować plik konfiguracyjny Kubernetes w formacie YAML.

Opis Kubernetes

Problemy są początkiem wiedzy.

— Kahlil Gibran

Kubernetes to system operacyjny świata cloud native, zapewniający niezawodną i skalowalną platformę do uruchamiania zadań kontenerowych. Jak jednak uruchomić Kubernetes? Czy powinieneś sam go hostować? W instancjach chmurowych? Na serwerach typu bare-metal? A może powinieneś skorzystać z usług zarządzanych Kubernetes? Lub platformy opartej na Kubernetes, ale rozszerzającej go o narzędzia przepływu zadań, pulpity nawigacyjne i interfejsy sieciowe?

Jest wiele pytań, na które należy odpowiedzieć w jednym rozdziale. Spróbujemy.

Warto zauważyc, że nie zajmiemy się tutaj szczegółami technicznymi obsługi samego Kubernetes, takimi jak budowanie, tuning czy rozwiązywanie problemów związanych z klastrami. Istnieje wiele materiałów, które mogą Ci w tym pomóc, z których szczególnie polecamy książkę *Managing Kubernetes: Operating Kubernetes Clusters in the Real World* (O'Reilly), autorstwa współzałożyciela Kubernetes — Brendana Burnsa.

Zamiast tego skupimy się na wyjaśnieniu podstawowej architektury klastra i udzieleniu informacji potrzebnych do uruchomienia Kubernetes. Przedstawimy zalety i wady usług zarządzanych oraz przyjrzymy się niektórym popularnym dostawcom.

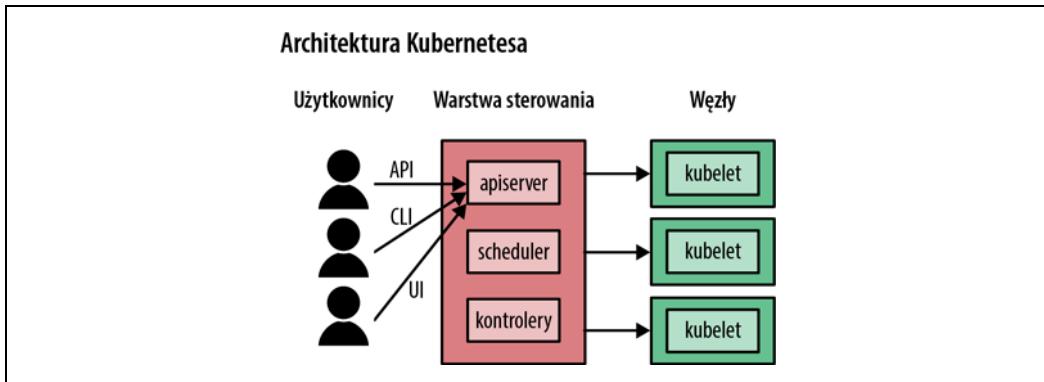
Jeśli chcesz uruchomić własny klaster Kubernetes, omówimy jedne z najlepszych dostępnych narzędzi instalacyjnych, które pomogą Ci w konfiguracji klastrów i zarządzaniu nimi.

Architektura klastrowa

Już wiesz, że Kubernetes łączy wiele serwerów w *klaster*, ale czym jest klaster i jak działa? Szczegóły techniczne nie mają znaczenia w tej książce, ale powinieneś zrozumieć działanie podstawowych elementów Kubernetes i ich wzajemne dopasowanie — aby wiedzieć, jakie masz możliwości w zakresie budowania lub zakupu klastrów Kubernetes.

Warstwa sterowania

Mózg klastra nazwany został *warstwą sterowania* (ang. *control plane*) i wykonuje wszystkie zadania wymagane przez Kubernetes do jego pracy: planowanie kontenerów, zarządzanie usługami, obsługa żądań API itd. (patrz rysunek 3.1).



Rysunek 3.1. Tak działa klanter Kubernetes

Warstwa sterowania składa się w rzeczywistości z kilku elementów.

kube-apiserver

Serwer frontend warstwy sterowania, obsługujący żądania API.

etcd

Baza danych, w której Kubernetes przechowuje wszystkie informacje: jakie węzły istnieją, jakie zasoby istnieją w klastrze itd.

kube-scheduler

Decyduje, gdzie uruchomić nowo utworzone Pody.

kube-controller-manager

Element odpowiedzialny za uruchamianie kontrolerów zasobów, takich jak Deployments.

cloud-controller-manager

Współdziała z dostawcą chmury (w klastrach opartych na chmurze), zarządzając zasobami, takimi jak usługi równoważenia obciążenia i woluminy dyskowe.

Elementy klastra, które uruchamiają komponenty warstwy sterowania, nazywane są *węzłami master*.

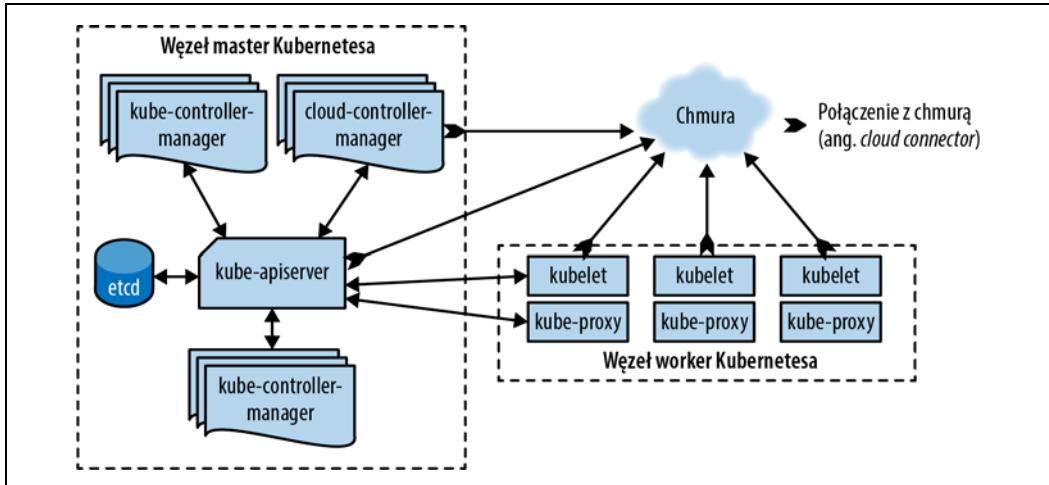
Elementy węzła

Elementy klastra, które uruchamiają zadania użytkowników, nazywane są *węzłami worker* (patrz rysunek 3.2).

Każdy węzeł worker w klastrze Kubernetes uruchamia następujące komponenty.

kubelet

Odpowiedzialny za uruchomienie środowiska wykonawczego kontenera w celu uruchomienia zadań zaplanowanych w węźle i monitorowania ich stanu.



Rysunek 3.2. Tak komponenty Kubernetes współpracują ze sobą

kube-proxy

Komponent odpowiedzialny za obsługę routingu żądań między Podami w różnych węzłach oraz między Podami a internetem.

Środowisko wykonawcze kontenera (ang. Container runtime)

Komponent, który uruchamia i zatrzymuje kontenery oraz obsługuje ich komunikację. Zazwyczaj to Docker, ale Kubernetes obsługuje inne środowiska wykonawcze, takie jak rkt i CRI-O.

Poza uruchomieniem różnych komponentów oprogramowania, nie ma istotnej różnicy między węzłami master a węzłami worker. Węzły master zwykle jednak nie uruchamiają zadań użytkowników, z wyjątkiem bardzo małych klastrów (takich jak Docker Desktop lub Minikube).

System wysokiej niezawodności (ang. high availability)

Prawidłowo skonfigurowana warstwa sterowania Kubernetes posiada wiele węzłów master, co czyni ją *wysocie niezawodną*. Jeśli dowolny pojedynczy węzeł master ulegnie awarii lub zostanie zamknięty albo jeden z jego komponentów warstwy sterowania przestanie działać, kластer nadal będzie działał poprawnie. Wysoka niezawodność warstwy sterowania poradzi sobie również z sytuacją, w której węzły master działają poprawnie, ale niektóre z nich nie mogą komunikować się z innymi z powodu awarii sieci (w przypadku *partyjci sieciowej* — ang. *network partition*).

Baza danych *etcd* jest replikowana na wielu węzłach i może przetrwać awarię poszczególnych węzłów, o ile kworum przekraczające połowę liczby replik *etcd* jest nadal dostępne.

Jeśli wszystko jest poprawnie skonfigurowane, warstwa sterowania może przetrwać ponowne uruchomienie lub tymczasową awarię poszczególnych węzłów master.

Awaria warstwy sterowania

Uszkodzona warstwa sterowania niekoniecznie oznacza, że aplikacje przestaną działać, chociaż może powodować dziwne i błędne zachowanie.

Jeśli np. zatrzymasz wszystkie węzły master w klastrze, na węzłach worker nadal będą działały Pody — przynajmniej przez jakiś czas. Nie można jednak wdrożyć żadnych nowych kontenerów ani zmienić żadnych zasobów Kubernetes, a kontrolery, takie jak Deployments, przestaną działać.

Dlatego wysoka niezawodność warstwy sterowania ma kluczowe znaczenie dla prawidłowo działającego klastra. Musisz mieć wystarczającą liczbę węzłów master, aby kластer mógł utrzymać kworum, nawet jeśli ulegnie awarii; w środowisku produkcyjnym muszą działać minimum trzy (patrz „Najmniejszy kластer” w rozdziale 6.).

Awaria węzła worker

Natomiast awaria dowolnego węzła worker nie ma znaczenia. Kubernetes wykryje awarię i przemieści Pody z uszkodzonego węzła gdzie indziej, o ile warstwa sterowania nadal działa.

Jeśli duża liczba węzłów ulegnie awarii jednocześnie, może to oznaczać, że kластer nie ma już wystarczających zasobów do uruchomienia wszystkich potrzebnych zadań. Na szczęście nie zdarza się to często. Jeśli nawet tak się dzieje, Kubernetes będzie utrzymywał jak najwięcej Podów do czasu, gdy zastąpisz brakujące węzły.

Warto jednak pamiętać, że im mniej masz węzłów worker, tym większy jest ich udział w każdym klastrze. Należy założyć, że awaria jednego węzła może wystąpić w dowolnym momencie, szczególnie w chmurze, a dwie jednocośne awarie nie są niespotykane.

Rzadkim, ale całkowicie możliwym rodzajem awarii jest utrata całej *strefy dostępności chmury* (ang. *availability zone*). Dostawcy chmur, tacy jak AWS i Google Cloud, zapewniają wiele stref dostępności w każdym regionie, z których każdy odpowiada w przybliżeniu pojedynczemu centrum danych. Z tego powodu zamiast umieszczać wszystkie węzły robocze w tej samej strefie, dobrym pomysłem jest rozdzielenie ich na dwie lub nawet trzy strefy.

Ufaj, ale sprawdzaj

Chociaż wysoka niezawodność powinna pozwolić klastrowi przetrwać utratę jednego węzła master lub kilku węzłów worker, zawsze warto to *przetestować*. W czasie zaplanowanej konserwacji lub poza godzinami szczytu spróbuj zresetować węzeł worker i sprawdź, co się stanie. (Mam nadzieję, że nic lub coś, co nie jest widoczne dla użytkowników Twoich aplikacji).

W bardziej wymagającym teście zrestartuj jeden z węzłów master. (Usługi zarządzane, takie jak Google Kubernetes Engine, które omówimy dalej w tym rozdziale, nie pozwolą Ci tego robić z oczywistych powodów). Mimo to kластer klasy produkcyjnej powinien przetrwać bez żadnych problemów.

Koszty samodzielnego hostingu Kubernetes

Najważniejszą decyzją, przed którą stoi każdy, kto rozważa uruchomienie zadań produkcyjnych w Kubernetes, jest decyzja o *zakupie czy samodzielnnej budowie*. Czy powinieneś uruchamiać własne klastry, czy płacić komuś innemu za ich zarządzanie? Spójrzmy na niektóre opcje.

Najprostszym wyborem jest utrzymanie własnego hostingu Kubernetes. Rozumiemy przez to, że osobiście Ty lub Twój zespół instalujecie i konfigurujecie Kubernetes na komputerach, które posiadasz

lub kontrolujesz, tak jak w przypadku każdego innego oprogramowania, którego używasz, takiego jak Redis, PostgreSQL lub Nginx.

Jest to opcja zapewniająca maksymalną elastyczność i kontrolę. Możesz zdecydować, które wersje Kubernetes mają być uruchamiane, jakie opcje i funkcje są włączone, kiedy i czy chcesz aktualizować klastry itd. Jednak ta opcja ma kilka istotnych wad, o czym piszemy w kolejnym punkcie.

To więcej pracy niż myślisz

Opcja własnego hostowania wymaga również maksymalnych zasobów ludzkich, umiejętności, czasu inżynierii, konserwacji i rozwiązywania problemów. Samo skonfigurowanie działającego klastra Kubernetes jest dość proste, ale od klastra gotowego do produkcji daleka droga. Musisz взять pod uwagę następujące problemy.

- Czy warstwa sterowania jest wysoko niezawodna? Co się stanie, gdy węzeł master ulegnie awarii lub przestanie odpowiadać? Czy klaster nadal działa? Czy nadal możesz wdrażać lub aktualizować aplikacje? Czy bez warstwy sterowania działające aplikacje nadal będą odporne na awarie (patrz „System wysokiej niezawodności” wcześniej w tym rozdziale)?
- Czy pula węzłów worker jest wysoce niezawodna? Jeżeli awaria wyeliminuje kilka węzłów worker, a nawet całą strefę dostępności chmury, czy zadania przestaną działać? Czy Twój klaster będzie nadal działał? Czy będzie w stanie automatycznie udostępnić nowe węzły, aby odzyskać sprawność, czy będzie wymagać ręcznej interwencji?
- Czy Twój klaster jest zabezpieczony? Czy jego wewnętrzne komponenty komunikują się przy użyciu szyfrowania TLS i zaufanych certyfikatów? Czy użytkownicy i aplikacje posiadają minimalne prawa i uprawnienia do operacji klastrowych? Czy domyślne wartości bezpieczeństwa kontenera są ustawione poprawnie? Czy węzły posiadają niezbędnego dostępu do komponentów warstwy sterowania? Czy dostęp do bazy danych *etcd* jest odpowiednio kontrolowany i uwierzytelniany?
- Czy wszystkie usługi w klastrze są bezpieczne? Jeśli są dostępne z internetu, czy są odpowiednio uwierzytelnione? Czy dostęp do API klastra jest ściśle ograniczony?
- Czy Twój klaster został skonfigurowany zgodnie ze standardami? Czy spełnia standardy dla klastrów Kubernetes określone przez Cloud Native Computing Foundation (patrz „Sprawdzanie zgodności” w rozdziale 6.)?
- Czy Twoje węzły klastra są w pełni zarządzane przez konfigurację (ang. *config-managed*), a nie konfigurowane przez imperatywne skrypty powłoki, a następnie pozostawione same sobie? System operacyjny i jądro muszą być w każdym węźle aktualizowane, muszą być stosowane poprawki bezpieczeństwa itd.
- Czy w klastrze są prawidłowo wykonywane kopie zapasowe? Jak wygląda proces przywracania z kopii? Jak często testujesz proces odzyskiwania danych?
- Jak już będziesz mieć działający klaster, to jak w miarę upływu czasu będzie wyglądał proces jego utrzymania? W jaki sposób przydzielasz nowe węzły? Jak są wdrażane zmiany konfiguracji w istniejących węzłach? A jak aktualizacje Kubernetes? Jak wygląda skalowanie oraz wdrażanie nowej polityki?

Cindy Sridharan, inżynier i twórca systemów rozproszonych, oszacowała (<https://twitter.com/copyconstruct/status/1020880388464377856>), że koszt utrzymania inżyniera, który konfiguruje (od zera do wersji produkcyjnej) Kubernetes, wynosi około miliona dolarów. Jest to temat do przemyśleń dla każdego dyrektora technicznego, który zastanawia się nad własną infrastrukturą Kubernetes.

Nie chodzi tylko o początkową konfigurację

Pamiętaj, że na powyższe czynniki należy zwrócić uwagę nie tylko podczas konfigurowania pierwszego klastra, ale trzeba o nich pamiętać cały czas i obejmować wszystkie klastry. Wprowadzając zmiany lub aktualizacje infrastruktury Kubernetes, należy wziąć pod uwagę ich wpływ na wysoką niezawodność, bezpieczeństwo itd.

Aby mieć pewność, że węzły klastra i wszystkie komponenty Kubernetes działają poprawnie, musisz wprowadzić ciągły monitoring oraz skuteczny system powiadomiania o awariach. Pracownicy muszą być informowani o awariach zarówno w dzień, jak i w nocy.

Kubernetes jest wciąż w fazie dynamicznego rozwoju — cały czas są publikowane nowe funkcje i aktualizacje. Musisz dbać o to, aby klaster był na bieżąco aktualizowany. Ponadto musisz rozumieć, w jaki sposób poszczególne zmiany wpływają na istniejącą konfigurację. Aby w pełni wykorzystać najnowszą funkcjonalność Kubernetes, konieczne może być ponowne skonfigurowanie klastra.

Aby odpowiednio skonfigurować klaster, nie wystarczy przeczytać kilku książek lub artykułów. Musisz regularnie testować i weryfikować konfigurację — np. unieruchamiając węzeł główny i upewniając się, że wszystko nadal działa.

Zautomatyzowane narzędzia do testowania odporności, takie jak opracowana przez Netflix koncepcja Chaos Monkey, mogą w tym pomóc — od czasu do czasu losowo unieruchamiane są węzły, Pody lub połączenia sieciowe. W zależności od stopnia niezawodności dostawcy usług chmurowych może się okazać, że Chaos Monkey jest zbędny, ponieważ regularne awarie w świecie rzeczywistym sprawdzą również odporność klastra i usług w nim działających (patrz „Testowanie chaosu” w rozdziale 6.).

Narzędzia nie wykonają za Ciebie całej pracy

Istnieją narzędzia — wiele narzędzi — które pomagają w konfigurowaniu klastrów Kubernetes, a wiele z nich reklamuje się, że oferuje konfigurację za pomocą „prostego klikania”. Smutnym faktem jest to, że — naszym zdaniem — znaczna większość tych narzędzi rozwiązuje tylko proste problemy i ignoruje trudne.

Z drugiej strony, wydajne, elastyczne narzędzia komercyjne klasy korporacyjnej są zwykle bardzo drogie lub nawet publicznie niedostępne, ponieważ można zarobić więcej na sprzedaży usługi zarządzanej niż na sprzedaży narzędzi do zarządzania klastrami ogólnego przeznaczenia.

Kubernetes jest trudny

Pomimo powszechnego przekonania, że konfiguracja i zarządzanie nim jest proste, prawda jest taka, że *Kubernetes jest trudny*. Biorąc pod uwagę to, co robi, jest niezwykle prosty i dobrze zaprojektowany, ale musi radzić sobie z bardzo złożonymi sytuacjami, co prowadzi do złożonego oprogramowania.

Codzienna nauka właściwego zarządzania własnymi klastrami wymaga znacznej inwestycji czasu i energii. Nie chcemy Cię zniechęcać do korzystania z Kubernetes, ale zależy nam, abyś dobrze rozumiał, z czym związane jest samodzielne uruchamianie Kubernetes. Pomoże to w podjęciu świadomiej decyzji.

Koszty administracyjne

Jeśli Twoja organizacja jest duża i ma do dyspozycji zasoby przeznaczone dla dedykowanego zespołu operacyjnego zajmującego się Kubernetes, może to nie stanowić dla Ciebie dużego problemu. Jednak w przypadku małych i średnich przedsiębiorstw, a nawet startupów z garstką inżynierów, nakłady administracyjne związane z prowadzeniem własnych klastrów Kubernetes mogą być zbyt duże.



Biorąc pod uwagę ograniczony budżet i liczbę personelu dostępnego do operacji IT, jaką część zasobów chcesz przeznaczyć na administrowanie samym Kubernetes? Czy te zasoby byłyby lepiej wykorzystane do obsługi innych zadań? Czy możesz obsługiwać Kubernetes w bardziej opłacalny sposób, korzystając z własnego personelu lub usług zarządzanych?

Zacznij od usług zarządzanych

Możesz być trochę zaskoczony tym, że w książce poświęconej Kubernetes zalecamy, abyś nie uruchamiał Kubernetes! Przynajmniej nie rób tego sam. Z powodów opisanych w poprzednich podrozdziałach uważamy, że korzystanie z usług zarządzanych może być znacznie bardziej opłacalne niż samodzielne hostowanie klastrów Kubernetes. Chyba że chcesz zrobić coś, czego nie zapewnia jeszcze żaden dostawca.



Z naszego doświadczenia oraz wielu osób, z którymi rozmawialiśmy, usługa zarządzana jest najlepszym sposobem na uruchomienie Kubernetes.

Jeśli zastanawiasz się, czy Kubernetes jest rozwiązaniem dla Ciebie, skorzystanie z usługi zarządzanej to świetny sposób na wypróbowanie. Możesz, w ciągu kilku minut oraz za kilka dolarów dziennie, uzyskać w pełni działający, bezpieczny, wysoce niezawodny kластer klasy produkcyjnej. (Większość dostawców usług chmurowych oferuje nawet bezpłatne rozwiązanie, która pozwala na uruchamianie klastra Kubernetes przez tygodnie lub miesiące bez ponoszenia żadnych opłat). Jeśli nawet po okresie próbny zdecydujesz, że wolisz uruchomić własny kластer Kubernetes, przy użyciu usług zarządzanych dowieś się, jak należy to zrobić.

Jeśli jednak już eksperymentowałeś z samodzielnym konfigurowaniem Kubernetes, będziesz zdumiony, o ile łatwiejsza jest praca za pomocą usług zarządzanych. Prawdopodobnie nie zbudowałeś własnego domu; po co zatem budować własny kластer, skoro taniej i szybciej zlecić to komuś innemu, a wyniki są lepsze?

W następnym podrozdziale przedstawimy niektóre z najpopularniejszych zarządzanych usług Kubernetes, powiemy Ci, co o nich myślimy, i polecimy naszą ulubioną. Jeśli nadal nie jesteś przekonany,

w drugiej połowie rozdziału omówimy instalatory Kubernetes, których możesz użyć do zbudowania własnych klastrów (patrz podrozdział „Instalatory Kubernetes”).

W tym miejscu powinniśmy powiedzieć, że żaden z autorów książki nie jest związany z żadnym dostawcą usług chmurowych czy też komercyjnym dostawcą Kubernetes. Nikt nie płaci nam za rekomendację swojego produktu lub usługi. Opinie zawarte tutaj są nasze, oparte na osobistym doświadczeniu oraz opiniach setek użytkowników Kubernetes, z którymi rozmawialiśmy podczas tworzenia tej książki.

Oczywiście, w świecie Kubernetes wszystko postępuje dynamicznie, a rynek usług zarządzanych jest szczególnie konkurencyjny. Bądź pewien, że funkcje i usługi opisane w tej książce szybko się zmienią. Przedstawiona tutaj lista nie jest kompletna, ale staraliśmy się uwzględnić usługi, które — naszym zdaniem — są najlepsze, najczęściej używane lub w inny sposób ważne.

Zarządzane usługi Kubernetes

Zarządzane usługi Kubernetes zwalniają Cię z prawie wszystkich kosztów administracyjnych związanych z konfigurowaniem i uruchamianiem klastrów Kubernetes, zwłaszcza warstwy sterowania. W rzeczywistości usługa zarządzana oznacza, że płacisz komuś innemu (np. Google) za uruchomienie klastra za Ciebie.

Google Kubernetes Engine (GKE)

Jak można oczekiwąć od twórców Kubernetes, Google oferuje w pełni zarządzaną usługę Kubernetes (<https://cloud.google.com/kubernetes-engine>), która jest całkowicie zintegrowana z Google Cloud Platform. Po prostu, aby utworzyć kластera za pomocą webowej konsoli GCP, wybierz oraz zatwierdź liczbę węzłów worker. Możesz także skorzystać z narzędzia Deployment Tool. W kilka minut klastera będzie gotowy do użycia.

Google dba o monitorowanie oraz wymianę awaryjnych węzłów, automatyczne stosowanie poprawek bezpieczeństwa oraz wysoką niezawodność dla warstwy sterowania itp. Możesz dla wszystkich swoich węzłów ustawić automatyczną aktualizację do najnowszej wersji Kubernetes podczas okresowej konserwacji.

Wysoka niezawodność

GKE zapewnia produkcyjny, wysoce niezawodny klastera Kubernetes — bez żadnych kosztów związanych z konfiguracją i konserwacją. Wszystko można kontrolować za pomocą interfejsu API Google Cloud, narzędzia Deployment Manager, Terraform lub innych narzędzi. Można także skorzystać z konsoli webowej GCP. Oczywiście GKE jest w pełni zintegrowana ze wszystkimi innymi usługami w Google Cloud.

W celu zwiększenia wysokiej niezawodności można tworzyć klastry *wielostrefowe*, które rozkładają węzły worker na wiele stref awarii (w przybliżeniu odpowiadających pojedynczym centrom danych). Twoje zadania będą działać, jeśli nawet awaria dotyczyć będzie całej strefy awarii.

Klastry regionalne (ang. *regional clusters*) rozszerzają ten pomysł, dystrybuując wiele węzłów master do innych stref, podobnie jak węzły worker.

Automatyczne skalowanie klastra

GKE oferuje również atrakcyjną opcję automatycznego skalowania klastra (patrz „Automatyczne skalowanie” w rozdziale 6.). Jeśli istnieją zadania, które oczekują na dostępność węzła, to po włączeniu automatycznego skalowania system automatycznie doda nowe węzły, aby zaspokoić zapotrzebowanie.

I odwrotnie, jeśli istnieją nadmiarowe zasoby, autoskaler skonsoliduje Pody na mniejszej liczbie węzłów i usunie nieużywane węzły. Ponieważ opłata zależna jest od liczby węzłów worker, pomaga to kontrolować koszty.

GKE jest najlepsza w swojej klasie

Google działa w branży Kubernetes dłużej niż ktokolwiek inny i to pokazuje — naszym zdaniem — że GKE jest najlepszą dostępną usługą zarządzaną Kubernetes. Jeśli masz już infrastrukturę w Google Cloud, warto skorzystać z GKE do uruchomienia Kubernetes. Jeśli korzystasz z innego rozwiązania chmurowego, nie musisz z niego rezygnować. Jednak powinieneś przyjrzeć się opcjom istniejącym u obecnego dostawcy rozwiązania chmurowego.

Jeśli jeszcze nie podjąłeś decyzji o wyborze rozwiązania, GKE jest przekonującym argumentem przemawiającym za wyborem Google Cloud.

Usługa Amazon Elastic Container Service dla Kubernetes (EKS)

Amazon od dawna świadczy usługi związane z kontenerami, ale do niedawna jedyną opcją była usługa Elastic Container Service (ECS), zastrzeżona technologią Amazon.

Chociaż ECS doskonale nadaje się do użytku (<https://aws.amazon.com/ecs>), nie jest on tak potężny ani elastyczny jak Kubernetes. Do takiego wniosku doszedł także sam Amazon i uruchomił usługę Elastic Container Service dla Kubernetes (EKS). (Tak, EKS powinien oznaczać Elastic Kubernetes Service, ale tak nie jest).

Praca nie przebiega tak płynnie jak w przypadku (<https://blog.hasura.io/gke-vs-aks-vs-eks-411f080640dc>) Google Kubernetes Engine, więc przygotuj się na samodzielne wykonanie większej części konfiguracji.

Jeśli masz już infrastrukturę w AWS lub uruchamiasz zadania w starszej usłudze ECS, którą chcesz przenieść do Kubernetes, to EKS jest rozsądnym wyborem. Jednak usługa ta ma sporo do nadrobienia w stosunku do rozwiązań Google oraz Microsoft.

Usługa Azure Kubernetes Service (AKS)

Chociaż Microsoft zaoferował usługi chmurowe nieco później niż Amazon czy Google, to szybko nadrabia zaległości. Usługa Azure Kubernetes Service (AKS) (<https://azure.microsoft.com/en-us/services/kubernetes-service/>) oferuje większość funkcji konkurentów, takich jak GKE Google. Możesz tworzyć klastry za pomocą interfejsu webowego lub narzędzia wiersza poleceń Azure az.

Podobnie jak w przypadku GKE i EKS, nie masz dostępu do węzłów master, którymi zarządza się wewnętrznie, a opłata zależy od liczby węzłów worker występujących w klastrze oraz dodatkowych kosztów związanych z węzłami master.

OpenShift

OpenShift (<https://www.openshift.com/>) to coś więcej niż tylko zarządzana usługa Kubernetes — jest to pełne rozwiązanie typu *platforma jako usługa* (PaaS), którego celem jest zarządzanie całym cyklem życia oprogramowania, wyposażone w narzędzia obsługujące proces ciągłej integracji oraz budowania, test runner, wdrażanie aplikacji, monitorowanie oraz orkiestrację.

Rozwiązanie OpenShift można wdrożyć na serwerach typu bare-metal, maszynach wirtualnych, chmurach prywatnych i publicznych. Zatem możesz utworzyć pojedynczy kластer Kubernetes obejmujący wszystkie te środowiska. To sprawia, że jest to dobry wybór dla bardzo dużych organizacji lub tych o bardzo heterogenicznej infrastrukturze.

Usługa IBM Cloud Kubernetes Service

Oczywiście, w dziedzinie zarządzanych usług Kubernetes nie można pominąć czcigodnego IBM. Usługa IBM Cloud Kubernetes Service (<https://www.ibm.com/cloud/container-service>) jest dość prosta i pozwala na skonfigurowanie klastra Kubernetes w środowisku IBM Cloud.

Möżesz uzyskać dostęp do klastra IBM Cloud i zarządzać nim za pomocą domyślnego interfejsu CLI Kubernetes oraz wiersza poleceń lub przy użyciu podstawowego interfejsu GUI. Oferta IBM niczym nie różni się od innych głównych dostawców rozwiązań chmurowych, ale jest to logiczny wybór, jeśli do tej pory korzystałeś już z rozwiązania IBM Cloud.

Rozwiązania Kubernetes pod klucz

Chociaż zarządzane usługi Kubernetes są odpowiednie dla większości wymagań biznesowych, mogą zaistnieć pewne okoliczności, w których korzystanie z usług zarządzanych nie jest możliwe. Istnieje rosnąca liczba ofert „pod klucz”, których celem jest dostarczenie gotowego do użycia, produkcyjnego klastra Kubernetes — wystarczy jeden raz kliknąć w przeglądarce internetowej.

Gotowe rozwiązania Kubernetes są atrakcyjne zarówno dla dużych przedsiębiorstw (ponieważ mogą one utrzymywać komercyjne relacje z dostawcą), jak i małych firm z ograniczonymi zasobami inżynieryjnymi i operacyjnymi. Oto kilka rozwiązań z obszaru „pod klucz”.

Containership Kubernetes Engine (CKE)

CKE (<https://blog.containership.io/introducing-containership-kubernetes-engine/>) to kolejny interfejs przeglądarkowy do obsługi Kubernetes w chmurze publicznej. Umożliwia uruchomienie klastra z rozsądnymi ustawieniami domyślnymi lub dostosowanie niemal każdego aspektu klastra do bardziej wymagających zastosowań.

Instalatory Kubernetes

Jeśli klastry zarządzane lub rozwiązania „pod klucz” nie są dla Ciebie rozsądny wyborem, musisz rozważyć pewien poziom samodzielnego hostingu Kubernetes: tzn. samodzielnie skonfigurować i uruchomić Kubernetes na własnych komputerach.

Jest bardzo mało prawdopodobne, że będziesz wdrażać i uruchamiać Kubernetes całkowicie od zera, z wyjątkiem celów edukacyjnych i demonstracyjnych. Zdecydowana większość osób korzysta z jednego lub wielu dostępnych narzędzi albo usług instalatora Kubernetes do konfigurowania swoich klastrów i zarządzania nimi.

kops

Narzędzie kops (<https://kubernetes.io/docs/setup/custom-cloud/kops/>) jest oparte na wierszu poleceń, służy do automatycznego udostępniania klastrów Kubernetes. To część projektu Kubernetes i już od dawna jest narzędziem specyficznym dla AWS. Jednak teraz dodano obsługę w wersji beta dla Google Cloud i planowana jest obsługa innych dostawców.

Narzędzie to obsługuje także tworzenie klastrów o wysokiej niezawodności, co sprawia, że jest odpowiednie dla produkcyjnych wdrożeń Kubernetes. Korzysta z konfiguracji deklaratywnej, podobnie jak same zasoby Kubernetes. Może nie tylko zapewnić niezbędne zasoby chmurowe i skonfigurować klaster, ale także skalować go w górę i w dół, zmieniać rozmiar węzłów, przeprowadzać aktualizacje i wykonywać inne przydatne zadania administracyjne.

Podobnie jak wszystko w świecie Kubernetes, kops jest w fazie dynamicznego rozwoju. Jest to jednak stosunkowo dojrzałe i wyrafinowane narzędzie, już szeroko stosowane. Jeśli planujesz uruchomić samodzielnie hostowane Kubernetes w AWS, dobrym pomysłem jest skorzystanie z kops.

Kubespray

Kubespray (<https://github.com/kubernetes-sigs/kubespray>) wcześniej znany jako Kargo, projekt pod patronatem Kubernetes, jest narzędziem do łatwego wdrażania klastrów. Oferuje wiele opcji, zapewnia wysoką niezawodność i obsługę wielu platform.

Kubespray koncentruje się na instalowaniu Kubernetes na istniejących maszynach, zwłaszcza na serwerach typu on-premise oraz bare-metal. Jednak stosować je można w dowolnym środowisku chmurowym, w tym chmurze prywatnej (maszyny wirtualne działające na Twoich serwerach).

TK8

TK8 (<https://github.com/kubernauts/tk8>) to narzędzie oparte na wierszu poleceń. Służy do udostępniania klastrów Kubernetes i wykorzystuje zarówno Terraform (do tworzenia serwerów chmurowych), jak i Kubespray (do instalowania na nich Kubernetes). Napisane zostało w języku Go (oczywiście), obsługuje instalację na serwerach AWS, OpenStack i bare-metal. W przygotowaniu jest obsługa platform Azure i Google Cloud.

TK8 nie tylko buduje kластер Kubernetes, ale także instaluje opcjonalne dodatki, w tym Jmeter Cluster do testowania obciążenia, Prometheus do monitorowania, Jaeger, Linkerd lub Zipkin do śledzenia, Ambassador API Gateway z narzędziem Envoy do obsługi load balancingu, Istio do obsługi service mesh, Jenkins-X na potrzeby CI/CD oraz Helm lub Kedge do przeprowadzanie archiwizacji w Kubernetes.

Kubernetes The Hard Way

Lepiej nie traktować samouczka *Kubernetes The Hard Way* autorstwa Kelseya Hightowera (<https://github.com/kelseyhightower/kubernetes-the-hard-way>) jako narzędzi instalacyjnego lub instrukcji instalacji Kubernetes. Jest to przemyślany przewodnik po procesie budowania klastra Kubernetes, który ilustruje złożoność różnych części. Niemniej jednak jest to bardzo pouczające ćwiczenie dla każdego, kto rozważa uruchomienie Kubernetes, nawet jako usługi zarządzanej. Pozwala zorientować się, jak to wszystko działa.

kubeadm

Narzędzie kubeadm (<https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/>) jest częścią dystrybucji Kubernetes i ma na celu pomoc w instalacji i utrzymaniu klastra Kubernetes — zgodnie z najlepszymi praktykami; kubeadm nie zapewnia infrastruktury dla samego klastra, więc nadaje się do instalowania Kubernetes na serwerach bare-metal lub dowolnego rodzaju instancjach chmurowych.

Wiele innych narzędzi i usług, o których wspomnimy w tym rozdziale, używa kubeadm wewnętrznie do obsługi operacji administracyjnych w klastrze. Nic jednak nie stoi na przeszkodzie, abyś używał go bezpośrednio, jeśli chcesz.

Tarmak

Tarmak (<https://blog.jetstack.io/blog/introducing-tarmak/>) to narzędzie do zarządzania cyklem życia klastra Kubernetes, którego celem jest ułatwienie modyfikacji i aktualizacji węzłów klastra. Wiele innych narzędzi radzi sobie z tym poprzez zastąpienie węzła, co może zająć dużo czasu i często wiąże się z przenoszeniem dużej ilości danych pomiędzy węzłami podczas procesu przebudowy. Zamiast tego Tarmak może naprawić lub zaktualizować węzeł na miejscu.

Do tworzenia węzłów klastra Tarmak używa Terraform. Natomiast Puppet używany jest do zarządzania konfiguracją w samych węzłach. Dzięki temu możliwe jest szybsze i bezpieczniejsze wprowadzanie zmian w konfiguracji węzła.

Rancher Kubernetes Engine (RKE)

RKE (<https://github.com/rancher/rke>) ma być prostym, szybkim instalatorem Kubernetes. Nie tworzy za Ciebie węzłów — zanim będziesz mógł skorzystać z RKE do zainstalowania klastra, musisz sam zainstalować Docker na węzłach. RKE obsługuje wysoką niezawodność warstwy sterowania Kubernetes.

Moduł Puppet dla Kubernetes

Puppet to potężne, dojrzałe i wyrafinowane narzędzie do ogólnego zarządzania konfiguracją. Jest bardzo szeroko stosowane oraz posiada duże wsparcie ze strony środowiska open source. Oficjalnie obsługiwany moduł Kubernetes (<https://forge.puppet.com/puppetlabs/kubernetes>) instaluje i konfiguruje Kubernetes na istniejących węzłach, włączając z obsługą wysokiej niezawodności dla warstwy sterowania oraz *etcd*.

Kubeformation

Kubeformation (<https://github.com/hasura/kubeformation>) to konfigurator online dla Kubernetes, który pozwala wybrać opcje klastra za pomocą przeglądarki internetowej, a następnie generuje szablony konfiguracji dla interfejsu API konkretnego dostawcy usług chmurowych (np. Deployment Manager dla Google Cloud lub Azure Resource Manager dla platformy Azure). Obsługa innych dostawców usług chmurowych jest w przygotowaniu.

Korzystanie z Kubeformation może nie być tak proste jak w przypadku innych narzędzi, ale ponieważ „opakowuje” inne istniejące narzędzia automatyzacji, takie jak Deployment Manager, jest to bardzo elastyczne oprogramowanie. Jeśli już zarządzasz infrastrukturą np. Google Cloud za pomocą Deployment Manager, Kubeformation idealnie spełni Twoje potrzeby.

Kup lub zbuduj: nasze rekomendacje

Poniżej przedstawiliśmy krótką prezentację niektórych (niezbędnych) opcji zarządzania klastrami Kubernetes — zakres wszystkich możliwych operacji jest duży i zróżnicowany, i cały czas rośnie. Możemy jednak sformułować kilka zaleceń opartych na zdrowych zasadach. Jedną z nich jest filozofia uruchamiania mniejszej ilości oprogramowania (ang. *run less software*) (<https://blog.intercom.com/run-less-software>).

Uruchom mniejsze oprogramowanie

Istnieją trzy filary filozofii „run less software”; wszystkie pomogą optymalizować czas i pokonać konkurencję.

1. Wybierz standardową technologię.
2. Zlecaj na zewnątrz „podnoszenie banalnych ciężarów”.
3. Stwórz trwałą przewagę konkurencyjną.

— Rich Archbold

Chociaż korzystanie z innowacyjnych technologii jest zabawne i eksytyujące, nie zawsze ma sens z biznesowego punktu widzenia. Korzystanie z popularnego oprogramowania, którego używają wszyscy inni, jest ogólnie dobrym wyborem, dlatego że prawdopodobnie działa, jest dobrze obsługiwane i nie będziesz ponosić ryzyka związanego z nieuniknionymi błędami.

Jeśli korzystasz z kontenerowych zadań oraz aplikacji natywnych w chmurze, Kubernetes jest naj-powszechniejszym wyborem — w najlepszy możliwy sposób. Biorąc to pod uwagę, powinieneś wybrać najbardziej dojrzałe, stabilne i powszechnie używane narzędzia i usługi Kubernetes.

„Podnoszenie banalnych ciężarów” to termin ukuty w Amazon, który oznacza całą ciężką pracę i wysiłek związany z instalowaniem oprogramowania i zarządzaniem nim, utrzymaniem infrastruktury itd. W tej pracy nie ma nic specjalnego; w każdej firmie wygląda tak samo. Kosztuje pieniądze, zamiast przynosić dochód.

Filozofia „run less software” mówi, że powinieneś zlecać „podnoszenie banalnych ciężarów”, ponieważ na dłuższą metę będzie to tańsze i zwalnia zasoby, które możesz wykorzystać do pracy w swojej podstawowej działalności.

Skorzystaj z zarządzanych usług Kubernetes, jeśli możesz

Mając na uwadze zasadę „run less software”, zalecamy outsourcing operacji klastra Kubernetes do usługi zarządzanej. Instalowanie, konfigurowanie, konserwacja, zabezpieczanie, aktualizowanie i zapewnianie niezawodności klastra Kubernetes to „podnoszenie banalnych ciężarów”. Dlatego prawie wszystkie firmy nie powinny robić tego samodzielnie.

Cloud native to termin, który nie oznacza dostawcy rozwiązań chmurowych, nie oznacza Kubernetes, to nie są kontenery, to nie jest technologia. Jest to praktyka polegająca na przy-spieszaniu działalności poprzez pomijanie powszechnych czynności.

— Justin Garrison

W zarządzanej przestrzeni Kubernetes Google Kubernetes Engine (GKE) jest wyraźnym zwycięzcą. Podczas gdy inni dostawcy usług w chmurze mogą nadrobić zaległości za rok lub dwa, Google wciąż jest daleko z przodu i będzie tam jeszcze przez jakiś czas.

A co z uzależnieniem od jednego dostawcy?

Jeśli zaangażujesz się w zarządzaną usługę Kubernetes od określonego dostawcy, taką jak Google Cloud, czy to uzależni Cię od dostawcy i ograniczy Twoje możliwości w przyszłości? Niekoniecznie. Kubernetes jest standardową platformą, więc wszelkie aplikacje i usługi, które zbudujesz w celu uruchomienia w Google Kubernetes Engine, będą również działać w każdym certyfikowanym systemie innego dostawcy Kubernetes. Samo korzystanie z Kubernetes to duży krok w kierunku uniknięcia uzależnienia od dostawcy.

Czy zarządzane Kubernetes sprawia, że jesteś bardziej podatny na uzależnienie od dostawcy niż w przypadku prowadzenia własnego klastra Kubernetes? Uważamy, że jest odwrotnie. Samodzielne hostowanie Kubernetes wymaga wielu maszyn i konfiguracji, które są ściśle powiązane z interfejsem API konkretnego dostawcy chmury. Przykładowo udostępnianie maszyn wirtualnych AWS do uruchamiania Kubernetes wymaga zupełnie innego kodu niż ta sama operacja w Google Cloud. Niektórzy asystenci konfiguracji Kubernetes, np. wspomniani w tym rozdziale, obsługują wielu dostawców usług chmurowych, ale nie wszystkich.

Jednym z założień Kubernetes jest ukrycie szczegółów technicznych platformy chmurowej i przedstawienie programistom standardowego, znanego interfejsu, który działa w ten sam sposób, niezależnie od tego, czy pracuje na platformie Azure, czy Google Cloud. Tak długo, jak projektujesz swoje aplikacje w oparciu o Kubernetes, nie będziesz uzależniony od konkretnego dostawcy.

Jeśli trzeba, użyj standardowych narzędzi do samodzielnego hostingu Kubernetes

Jeżeli masz specjalne wymagania, które oznaczają, że zarządzane oferty Kubernetes nie będą dla Ciebie odpowiednie, wtedy powinieneś rozważyć samodzielne uruchomienie Kubernetes.

W takim przypadku najlepiej skorzystać z najbardziej dojrzałych, potężnych i powszechnie używanych narzędzi. W zależności od wymagań zalecamy kops lub Kubespray.

Jeśli wiesz, że długoterminowo pozostaniesz u jednego dostawcy usług chmurowych (zwłaszcza wtedy, kiedy jest to AWS), skorzystaj z narzędzia kops.

Z drugiej strony, jeśli potrzebujesz, aby Twój klaster obejmował wiele chmur lub platform, w tym serwerów typu bare-metal, i chcesz mieć otwarte możliwości, powinieneś skorzystać z narzędzia Kubespray.

Gdy Twoje wybory są ograniczone

Mogą istnieć biznesowe, a nie techniczne powody, dla których w pełni zarządzane usługi Kubernetes nie są opcją dla Ciebie. Jeśli masz istniejące relacje biznesowe z firmą hostingową lub dostawcą chmury, która nie oferuje zarządzanej usługi Kubernetes, to z pewnością ograniczy Twoje możliwości.

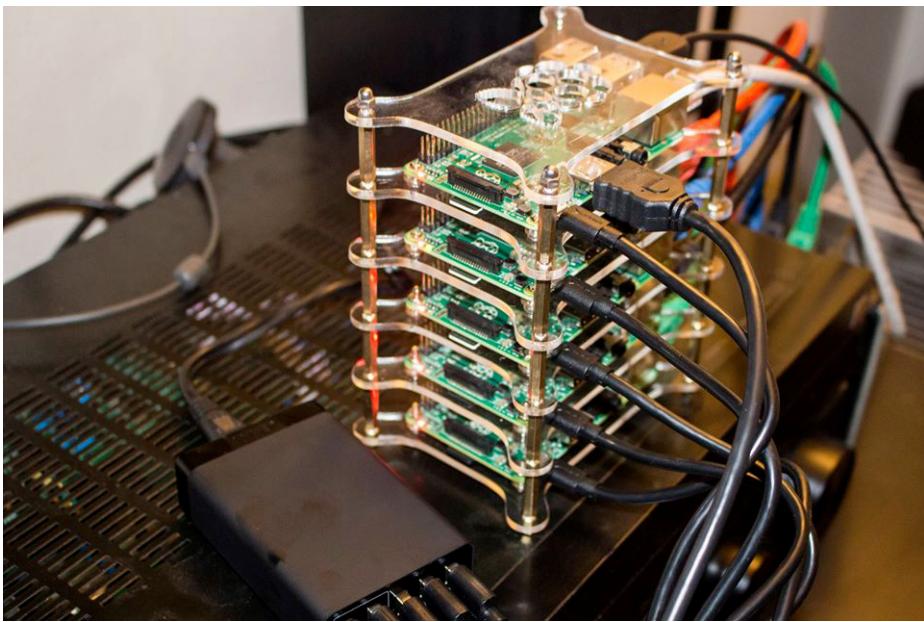
Możliwym rozwiązaniem jest skorzystanie z opcji „pod klucz”. Opcje te zapewniają usługę zarządzaną dla węzłów master Kubernetes, ale łączą ją z węzłami worker działającymi w Twojej infrastrukturze. Ponieważ większość kosztów administracyjnych związanych z Kubernetes polega na konfigurowaniu i utrzymywaniu węzłów master, jest to dobry kompromis.

Bare-metal i on-premise

Może Cię zaskoczyć, że bycie cloud native nie wymaga istnienia *w chmurze*, w sensie outsourcingu Twojej infrastruktury do publicznego dostawcy rozwiązań chmurowych, takiego jak Azure lub AWS.

Wiele organizacji obsługuje część lub całość infrastruktury na sprzęcie typu bare-metal, niezależnie od tego, czy jest zlokalizowana w centrach danych, czy na zasadzie modelu on-premise (własne środowisko informatyczne). Wszystko, co powiedzieliśmy w tej książce o Kubernetes i kontenerach, odnosi się zarówno do infrastruktury wewnętrznej, jak i do chmury.

Możesz uruchomić Kubernetes na własnych komputerach; jeśli Twój budżet jest ograniczony, możesz nawet uruchomić go na komputerach Raspberry Pi (patrz rysunek 3.3). Niektóre firmy działają w *chmurze prywatnej*, składającej się z maszyn wirtualnych hostowanych na lokalnym sprzęcie.



Rysunek 3.3. Budżetowy Kubernetes: kластер Raspberry Pi (fot. David Merrick)

Bezlastrowe usługi kontenerowe

Jeśli naprawdę chcesz zminimalizować obciążenie związane z uruchamianiem zadań kontenerowych, masz do wyboru jeszcze jeden poziom, który istnieje powyżej w pełni zarządzanych usług Kubernetes. Są to tzw. usługi *bezlastrowe*, takie jak Azure Container Instances lub Amazon's Fargate. Chociaż pod spodem znajduje się kластer, nie masz do niego dostępu za pomocą narzędzi, np. takich jak *kubectl*. Zamiast tego określasz obraz kontenera do uruchomienia i kilka parametrów, takich jak wymagania CPU i pamięci aplikacji. Usługa zajmie się resztą.

Amazon Fargate

Amazon twierdzi, że „Fargate jest jak EC2, ale zamiast maszyny wirtualnej dostajesz kontener”. W przeciwieństwie do ECS, nie ma potrzeby samodzielnego udostępniania węzłów klastra, a następnie podłączania ich do warstwy sterowania. Wystarczy zdefiniować zadanie, które jest zasadniczo zestawem instrukcji, jak utworzyć obraz kontenera i uruchomić go. Ceny naliczane są za sekundę, na podstawie zużywanych przez zadanie ilości zasobów procesora i pamięci.

Prawdopodobnie słusze jest stwierdzenie, że Fargate (<https://amzn.to/2SgQS9N>) ma sens w przypadku prostych, niezależnych, długo działających zadań obliczeniowych lub zadań wsadowych (takich jak przetwarzanie danych), które nie wymagają zbytniej modyfikacji lub integracji z innymi usługami. Jest również idealna do budowania kontenerów, które zwykle są krótkotrwałe, i do każdej sytuacji, w której narzut zarządzania węzłami worker nie jest uzasadniony.

Jeśli korzystasz już z ECS z węzłami worker EC2, przejście na Fargate uwolni Cię od konieczności zarządzania tymi węzłami. Fargate jest już dostępna w niektórych regionach do uruchamiania zadań ECS i ma obsługiwać EKS do 2019 r.

Azure Container Instances (ACI)

Usługa Azure Container Instances (ACI) firmy Microsoft (<https://azure.microsoft.com/en-gb/services/container-instances/>) jest podobna do usługi Fargate, ale oferuje również integrację z usługą Azure Kubernetes Service (AKS). Możesz np. skonfigurować klastera AKS, tak aby zapewniała tymczasowe dodatkowe Pody wewnętrz ACI do obsługi nagłych dodatkowych zapotrzebowan.

Podobnie ad hoc możesz uruchamiać zadania wsadowe w ACI, bez konieczności utrzymywania beczynnych węzłów. Microsoft nazywa tę ideę *kontenerami bezserwerowymi* (ang. *serverless containers*), ale stwierdzamy, że terminologia jest zarówno myląca (*bezserwerowa* zwykle odnosi się do działania chmury lub do usługi FaaS), jak i niedokładna (istnieją serwery, ale po prostu nie można uzyskać do nich dostępu).

Usługa ACI jest również zintegrowana z Azure Event Grid, zarządzaną przez Microsoft usługą routingu zdarzeń. Za pomocą Event Grid kontenery ACI mogą komunikować się z usługami w chmurze, funkcjami chmury lub aplikacjami Kubernetes działającymi w AKS.

Możesz tworzyć, uruchamiać zadania lub przekazywać dane do kontenerów ACI za pomocą Azure Functions. Zaletą jest to, że można uruchomić dowolne zadanie z funkcji chmurowej, nie tylko te, które korzystają z oficjalnie obsługiwanych języków, takich jak Python lub JavaScript.

Jeśli możesz skontenerować swoje zadania, możesz je uruchomić jako funkcje chmurowe, ze wszystkimi powiązanymi narzędziami. Przykładowo dzięki Microsoft Flow osoby, które nie są programistami, mogą w sposób graficzny tworzyć zadania, łącząc kontenery, funkcje i zdarzenia.

Podsumowanie

Kubernetes jest wszędzie! Nasza podróż przez rozległy krajobraz narzędzi, usług i produktów Kubernetes była z konieczności krótka, ale mamy nadzieję, że okaże się przydatna.

Chociaż nasze omówienie określonych produktów i funkcji jest tak aktualne, jak to możliwe, świat porusza się dość szybko i liczymy, że wiele się zmieni.

Uważamy jednak, że podstawową kwestią jest to, że nie warto samemu zarządzać klastrami Kubernetes, jeśli usługodawca może to zrobić lepiej i taniej.

Z naszego doświadczenia w doradztwie dla firm migrujących do Kubernetes często jest to zaskakujący pomysł dla wielu ludzi. Okazuje się, że organizacje, które zrobiły pierwsze kroki w kierunku własnego hostingu klastrów (używając narzędzi takich jak kops), nie zastanawiały się nad użyciem usługi zarządzanej, takiej jak GKE. Warto o tym pomyśleć.

Oto co powinieneś zapamiętać z tego rozdziału.

- Klastry Kubernetes składają się z węzłów *master*, które obsługują warstwę sterowania, oraz węzłów *worker*, które obsługują zadania.

- Klastry produkcyjne muszą być *wysoce niezawodne*, co oznacza, że awaria *węzła master* nie spowoduje utraty danych ani nie wpłynie na działanie klastra.
- Jest dłuża droga od prostego klastra demonstracyjnego do takiego, który jest gotowy na krytyczne obciążenia produkcyjne; wysoka niezawodność, bezpieczeństwo i zarządzanie węzłami to tylko niektóre z problemów.
- Zarządzanie własnymi klastrami wymaga znacznej inwestycji czasu, wysiłku i wiedzy; nawet wtedy, kiedy wszystko opanujesz, możesz jeszcze popełniać błędy.
- Usługi zarządzane, takie jak Google Kubernetes Engine, wykonują za Ciebie wszystkie zadania, znacznie niższym kosztem niż samodzielnny hosting.
- Usługi „pod klucz” stanowią dobry kompromis między własnym hostingiem a całkowicie zarządzanym Kubernetes; dostawcy „pod klucz” zarządzają za Ciebie *węzłami master*, podczas gdy uruchamiasz *węzły worker* na własnych komputerach.
- Jeśli musisz hostować własny klaster, kops jest dojrzałym i szeroko stosowanym narzędziem, które może udostępniać klastry produkcyjne w AWS oraz Google Cloud i zarządzać nimi.
- Jeśli możesz, powinieneś skorzystać z zarządzanego Kubernetes; jest to najlepsza opcja dla większości firm pod względem kosztów, nakładów pracy i jakości.
- Jeśli usługi zarządzane nie są dla Ciebie opcją, rozważ użycie usług „pod klucz” jako dobrego kompromisu.
- Nie hostuj swojego klastra bez uzasadnionych powodów biznesowych. Jeśli korzystasz z własnego hostingu, nie lekceważ czasu związanego z początkową konfiguracją i bieżącymi kosztami konserwacji.

Praca z obiektami Kubernetes

Nie rozumiem, dlaczego ludzie boją się nowych pomysłów. Boję się starych.

— John Cage

W rozdziale 2. zbudowałaś i wdrożyłaś aplikację na Kubernetes. W tym rozdziale poznasz podstawowe obiekty Kubernetes zaangażowane w ten proces, takie jak Pod, Deployment oraz Serwis (usługi). Dowiesz się również, jak korzystać z narzędzia Helm, niezbędnego do zarządzania aplikacjami w Kubernetes.

Po zapoznaniu się z przykładem w podrozdziale „Uruchamianie aplikacji demonstracyjnej” w rozdziale 2. powinieneś mieć obraz kontenera działający w klastrze Kubernetes. Jak on właściwie działa? Pod spodem polecenie `kubectl` tworzy zasób Kubernetes o nazwie *Deployment*. Co to jest? W jaki sposób Deployment rzeczywiście uruchamia obraz kontenera?

Zasoby Deployment

Przypomnij sobie, jak uruchomiłeś aplikację demo za pomocą Dockera. Polecenie `docker container run` uruchomiło kontener i działało, dopóki nie zakończono jego pracy poleceniem `docker stop`.

Załóżmy jednak, że kontener kończy pracę z innego powodu; być może program niespodziewanie zakończył pracę, wystąpił błąd systemu lub w komputerze zabrakło miejsca na dysku albo procesor, w niewłaściwym momencie, został trafiony promieniowaniem kosmicznym (mało prawdopodobne, ale tak się dzieje). Przy założeniu, że jest to aplikacja produkcyjna, oznacza to, że masz teraz na głowie niezadowolonych użytkowników, do momentu dopóki ktoś nie dostanie się do terminala i wpisze komendę `docker container run` w celu ponownego uruchomienia kontenera.

To nie jest zadowalające rozwiązanie. Naprawdę potrzebujesz czegoś w rodzaju programu nadzorującego, który stale sprawdza, czy kontener działa, a jeśli kiedykolwiek się zatrzyma, natychmiast uruchamia go ponownie. Na tradycyjnych serwerach możesz użyć narzędzia, takiego jak *systemd*, *runit* lub *supervisord*; Docker zawiera coś podobnego i nie zdziwi się, że Kubernetes ma również taki element nadzorczy, czyli Deployment.

Nadzór i planowanie

Dla każdego programu, który Kubernetes musi nadzorować, tworzony jest odpowiedni obiekt Deployment; obiekt rejestruje pewne informacje o programie: nazwę obrazu kontenera, liczbę replik, które chcesz uruchomić, oraz cokolwiek innego, co musi wiedzieć, aby uruchomić kontener.

Współdziałanie z zasobami Deployment kontrolowane jest przez komponent Kubernetes o nazwie kontroler (ang. *controller*). Kontrolery obserwują zasoby, za które są odpowiedzialne, upewniając się, że są obecne i działają. Jeśli dany Deployment nie uruchamia wystarczającej liczby replik, kontroler utworzy kilka nowych. (Gdyby z jakiegoś powodu było zbyt wiele replik, kontroler wyłączyłby nadmiarowe. Tak czy inaczej, kontroler upewnia się, że stan rzeczywisty odpowiada żądanemu stanowi).

W rzeczywistości obiekt Deployment nie zarządza bezpośrednio replikami: zamiast tego automatycznie tworzy powiązany obiekt o nazwie ReplicaSet, który to obsługuje. Za chwilę porozmawiamy o obiektach ReplicaSet, ale na razie omówimy obiekty Deployment.

Restart kontenerów

Na pierwszy rzut oka sposób działania obiektów Deployment może być nieco zaskakujący. Jeśli kontener zakończy swoje działanie, Deployment go zrestartuje. Jeśli ulegnie awarii lub zakończysz jego pracę, wysyłając do niego odpowiedni sygnał albo zakończysz jego pracę za pomocą kubectl, Deployment uruchomi go ponownie. (Jednak rzeczywistość jest nieco bardziej skomplikowana, jak zobaczymy).

Większość aplikacji Kubernetes zaprojektowano z myślą o długotrwałym działaniu i niezawodności, dlatego takie zachowanie ma sens: kontenery mogą zakończyć działanie z wielu różnych powodów, a w większości przypadków wystarczy tylko je zrestartować, więc domyślnie robi to Kubernetes.

Można zmienić tę zasadę dla pojedynczego kontenera: aby np. nigdy nie uruchamiać go ponownie lub uruchamiać go ponownie tylko w przypadku awarii, a nie w przypadku normalnego wyjścia (patrz „Restart polityk” w rozdziale 8.). Jednak domyślne zachowanie (zawsze restart) jest zwykle tym, czego chcesz.

Zadaniem obiektu Deployment jest obserwowanie powiązanych z nim kontenerów i upewnianie się, że określona liczba z nich jest zawsze uruchomiona. Jeśli będzie ich mniej, uruchomi więcej. Jeśli jest ich zbyt wiele, niektóre zakończą działanie. Jest to o wiele bardziej wydajne i elastyczne rozwiązanie niż tradycyjny program typu nadzorca.

Zapytania obiektów Deployment

Wszystkie aktywne obiekty Deployment w bieżącej przestrzeni nazw (ang. *namespace*) (patrz „Korzystanie z przestrzeni nazw” w rozdziale 5.) możesz zobaczyć za pomocą następującego polecenia:

```
kubectl get deployments
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
demo 1 1 1 1 21h
```

Aby zobaczyć bardziej szczegółowe informacje na temat konkretnego obiektu Deployment, użyj polecenia:

```
kubectl describe deployments/demo
Name:           demo
Namespace:      default
CreationTimestamp: Tue, 08 May 2018 12:20:50 +0100
...
```

Jak widzisz, masz dostęp do wielu informacji. W tej chwili jednak nie są one ważne. Przyjrzyjmy się bliżej sekcji Pod Template:

```
Pod Template:
  Labels: app=demo
  Containers:
    demo:
      Image: cloudnative/demo:hello
      Port: 8888/TCP
      Host Port: 0/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
```

Wiesz, że obiekt Deployment zawiera informacje, których Kubernetes potrzebuje do uruchomienia kontenera, i oto one. Jednak co to jest Pod Template? Zanim odpowiemy na to pytanie, wyjaśnimy, co to jest Pod.

Pody

Pod to obiekt Kubernetes, który reprezentuje grupę jednego lub wielu kontenerów (pod to także nazwa grupy wielorybów, która pasuje do morskiego slangu pojęć Kuberntesa).

Dlaczego obiekt Deployment nie zarządza bezpośrednio pojedynczym kontenerem? Odpowiedź jest taka, że czasami zestaw kontenerów musi działać razem, w tym samym węźle i komunikować się lokalnie, by móc też współużytkując pamięć.

Przykładowo aplikacja blogowa może mieć jeden kontener, który synchronizuje zawartość z repozytorium Git, oraz kontener serwera WWW Nginx, który udostępnia użytkownikom treść blogu. Ponieważ współużytkują dane, dwa kontenery muszą być ze sobą w Podzie. W praktyce jednak wiele Podów ma tylko jeden kontener, jak w tym przypadku (patrz „Co należy do Poda” w rozdziale 8., aby uzyskać więcej informacji na ten temat).

Tak więc specyfikacja Poda (w skrócie *spec*) zawiera listę kontenerów. W naszym przykładzie jest tylko jeden kontener, czyli *demo*:

```
demo:
  Image: cloudnative/demo:hello
  Port: 8888/TCP
  Host Port: 0/TCP
  Environment: <none>
  Mounts: <none>
```

Specyfikacją obrazu (*Image spec*) będzie w Twoim przypadku **YOUR_DOCKER_ID**/myhello i wraz z numerem portu będą to wszystkie informacje, jakich potrzebuje Deployment do uruchmienia i utrzymania Poda.

To jest ważny punkt. Komenda `kubectl run` nie utworzyła Poda bezpośrednio. Został utworzony obiekt Deployment, a *następnie* Deployment uruchomił Pod. Deployment to deklaracja pożdanego stanu: „Kontener `myhello` powinien działać wewnątrz Poda”.

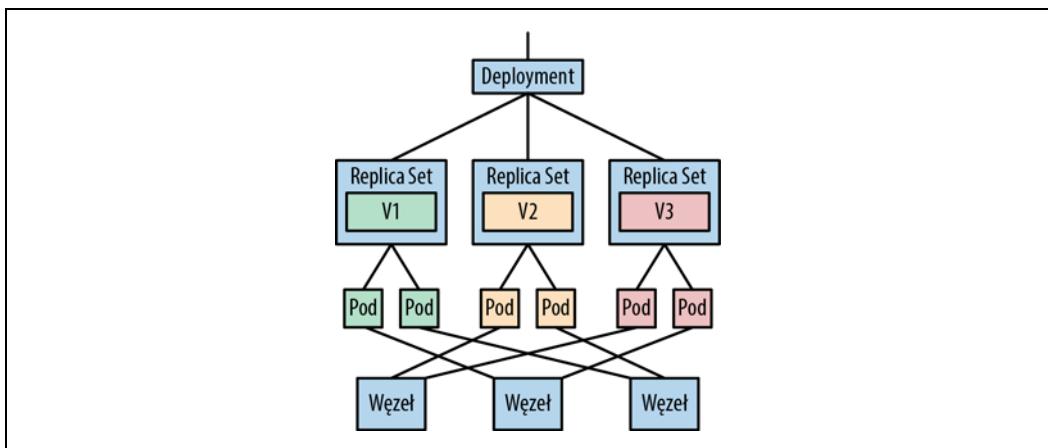
ReplicaSet

Powiedzieliśmy, że obiekty Deployment uruchamiają Pody, ale jest w tym coś więcej. W rzeczywistości obiekty Deployment nie zarządzają Podami bezpośrednio. To jest zadanie obiektu ReplicaSet.

ReplicaSet jest odpowiedzialny za grupę identycznych Podów lub *replik* (ang. *replicas*). Jeśli jest za mało (lub zbyt wiele) Podów w porównaniu ze specyfikacją, kontroler ReplicaSet uruchomi (lub zatrzyma) niektóre Pody w celu naprawy sytuacji.

Obiekty Deployment z kolei zarządzają obiektami ReplicaSet i kontrolują ich zachowanie podczas aktualizacji — np. poprzez wprowadzenie nowej wersji aplikacji (patrz „Strategie wdrażania” w rozdziale 13.). Kiedy aktualizujesz Deployment, tworzony jest nowy obiekt ReplicaSet w celu zarządzania nowymi Podami. Po zakończeniu aktualizacji stary obiekt ReplicaSet i jego Pody zostają zamkane.

Na rysunku 4.1 każdy obiekt ReplicaSet (V1, V2, V3) reprezentuje inną wersję aplikacji wraz z odpowiednimi Podami.



Rysunek 4.1. Obiekty Deployment, ReplicaSet oraz Pod

Zazwyczaj nie pracujesz bezpośrednio z obiektami ReplicaSet, ponieważ obiekty Deployment wykonują pracę za Ciebie — ale warto wiedzieć, czym one są.

Utrzymanie pożdanego stanu

Kontrolery Kubernetes stale sprawdzają, czy stan każdego zasobu jest zgodny z pożdanym, i wprowadzają niezbędne zmiany, aby utrzymać synchronizację. Proces ten nazywa się *pętlą uzgadniania* (ang. *reconciliation loop*), ponieważ zapętała się w nieskończoność, próbując pogodzić stan faktyczny z pożdanym.

Przykładowo po pierwszym utworzeniu obiektu Deployment *demo* nie ma działającego obiektu Pod *demo*. Tak więc Kubernetes natychmiast uruchomi wymagany Pod. Jeśli kiedykolwiek miałby się zatrzymać, Kubernetes uruchomi go ponownie, o ile obiekt Deployment nadal istnieje.

Sprawdźmy to teraz, zatrzymując Pod. Najpierw sprawdź, czy Pod rzeczywiście działa:

```
kubectl get pods --selector app=demo
NAME           READY STATUS   RESTARTS AGE
demo-54df94b7b7-qgtc6 1/1   Running 1        22h
```

Teraz uruchom następujące polecenie, aby zatrzymać Pod:

```
kubectl delete pods --selector app=demo
pod "demo-54df94b7b7-qgtc6" deleted
```

Ponownie pobierz listę Podów:

```
kubectl get pods --selector app=demo
NAME           READY STATUS   RESTARTS AGE
demo-54df94b7b7-hrspp 1/1   Running 0        5s
demo-54df94b7b7-qgtc6 0/1   Terminating 1      22h
```

Możesz zobaczyć, jak oryginalna kapsuła się wyłącza (jej status kończy się), ale została już zastąpiona nową kapsułą, która ma zaledwie pięć sekund. To pętla pojednania w pracy.

Powiedziałeś Kubernetesowi, za pomocą utworzonego przez Ciebie stanowiska, że *demo* Pod *musi zawsze działać*. Jeśli zatem sam usuniesz kapsułę, Kubernetes zakłada, że popełniłeś błąd, i pomaga założyć nową kapsułę, aby ją zastąpić.

Po zakończeniu eksperymentowania usuń obiekt Deployment za pomocą następującego polecenia:

```
kubectl delete all --selector app=demo
pod "demo-54df94b7b7-hrspp" deleted
service "demo" deleted
deployment.apps "demo" deleted
```

Scheduler

Powiedzieliśmy, że obiekty *Deployment tworzą Pody* oraz że *Kubernetes uruchomi wymagane Pody*, nie wyjaśniając dokładnie, jak to się dzieje.

Scheduler jest komponentem odpowiedzialnym za tę część procesu. Gdy obiekt Deployment (za pośrednictwem powiązanego obiektu ReplicaSet) zdecyduje, że potrzebna jest nowa replika, tworzy zasób Pod w bazie danych Kubernetes. Jednocześnie ten Pod jest dodawany do kolejki, która działa jak skrzynka odbiorcza komponentu scheduler.

Zadaniem tego komponentu jest obserwowanie kolejki nieprzydzielonych Podów, wybranie z niego kolejnego Poda i znalezienie węzła, na którym można go uruchomić. Aby wybrać odpowiedni węzeł, przy założeniu, że jest dostępny, skorzysta z kilku różnych kryteriów, w tym żądań Podów (więcej o tym procesie powiemy w rozdziale 5.).

Po przydzieleniu Poda do węzła proces kubelet działający w tym węźle pobiera go i dba o faktyczne uruchomienie kontenerów (patrz „Elementy węzła” w rozdziale 3.).

Po usunięciu Poda w podrozdziale „Utrzymaniu pożądanego stanu” ReplicaSet węzła zauważyl to działanie i uruchomił zapasowy Pod. On wie, że Pod o nazwie *demo* powinnien działać w swoim węźle — jeśli go nie znajdzie, uruchomi zapasowy. (Co by się stało, gdyby całkowicie zamknąć węzeł? Jego Pody wrócą do kolejki komponentu scheduler, aby ponownie przypisać je do innych węzłów).

Inżynier Stripe Julia Evans znakomicie wyjaśniła, jak działa scheduler w Kubernetes (<https://jvns.ca/blog/2017/07/27/how-does-the-kubernetes-scheduler-work/>).

Manifesty zasobów w formacie YAML

Czy wiesz już, jak uruchomić aplikację w Kubernetes? Czy to wszystko? Nie do końca. Użycie polecenia `kubectl run` do utworzenia obrazu obiektu Deployment jest przydatne, ale ograniczone. Założymy, że chcesz coś zmienić w specyfikacji Deployment, powiedzmy nazwę lub wersję. Możesz usunąć istniejący Deployment (za pomocą polecenia `kubectl delete`) i utworzyć nowy z odpowiednimi wartościami. Zobaczmy, czy można to zrobić lepiej.

Ponieważ Kubernetes jest z natury systemem *deklaratywnym*, stale uzgadniającym stan rzeczywisty z pożądanym, musisz tylko zmienić żądany stan — specyfikację obiektu Deployment — a Kubernetes wykona resztę. Jak to zrobić?

Zasoby są danymi

Wszystkie zasoby Kubernetes, takie jak Deployment lub Pod, są reprezentowane jako rekordy wewnętrznej bazie danych. Pętla uzgadniania śledzi bazę danych pod kątem zmian w tych rekordach i podejmuje odpowiednie działania. W rzeczywistości wszystko, co robi komenda `kubectl run`, polega na dodaniu do bazy danych nowego rekordu odpowiadającego obiekowi Deployment, a Kubernetes zajmuje się resztą.

Istnieje alternatywny sposób. Można utworzyć oraz edytować *manifest zasobu* (specyfikację żądanego stanu zasobu). Możesz przechowywać plik manifestu w systemie kontroli wersji i zamiast uruchamiać polecenia, możesz zmienić pliki manifestu, a następnie nakazać Kubernetesowi odczytanie zaktualizowanych danych.

Manifesty obiektu Deployment

Pliki manifestu Kubernetes zapisywane są w formacie YAML, chociaż możliwy jest także format JSON. Jak wygląda manifest YAML dla obiektu Deployment?

Spojrzymy na przykład naszej aplikacji demonstracyjnej (*hello-k8s/k8s/loymen.yaml*):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 1
  selector:
```

```
matchLabels:  
  app: demo  
template:  
  metadata:  
    labels:  
      app: demo  
spec:  
  containers:  
    - name: demo  
      image: cloudnativd/demo:hello  
      ports:  
        - containerPort: 8888
```

Na pierwszy rzut oka skomplikowane, ale w większości to szablon. Jedyne interesujące części to te same informacje, które już widziałeś w różnych formach, czyli nazwa i port obrazu kontenera. Gdy podałeś tę informację wcześniej do polecenia `kubectl run`, pod spodem utworzony został odpowiednik tego manifestu i przesłany do Kubernetes.

Polecenie `kubectl apply`

Aby wykorzystać pełną moc Kubernetes jako infrastruktury deklaratywnej w postaci kodu, prześlij manifesty YAML do klastra samodzielnie, używając polecenia `kubectl apply`.

Skorzystaj z naszego przykładowego manifestu dla obiektu Deployment, `hello-k8s/k8s/loymen.t.yaml1`.

Uruchom następujące polecenia w kopii repozytorium `demo`:

```
cd hello-k8s  
kubectl apply -f k8s/deployment.yaml  
deployment.apps "demo" created
```

Po kilku sekundach powinien uruchomić się Pod `demo`:

```
kubectl get pods --selector app=demo  
NAME          READY STATUS  RESTARTS AGE  
demo-6d99bf474d-z9zv6 1/1   Running 0       2m
```

To jeszcze nie jest koniec. Aby połączyć Pod `demo` z przeglądarką internetową, utworzymy Serwis (ang. *Service*), który jest zasobem Kubernetes pozwalającym łączyć się z Podami (więcej na ten temat za chwilę).

Najpierw sprawdźmy, czym jest Serwis i dlaczego go potrzebujemy.

Serwis

Założymy, że chcesz nawiązać połączenie sieciowe z Podem (takim jak nasza przykładowa aplikacja). Jak to zrobić? Możesz znaleźć adres IP Poda, numer portu i połączyć się z nim. Jednak adres IP może się zmienić po ponownym uruchomieniu Poda — musisz więc go ciągle sprawdzać, aby upewnić się, że jest aktualny.

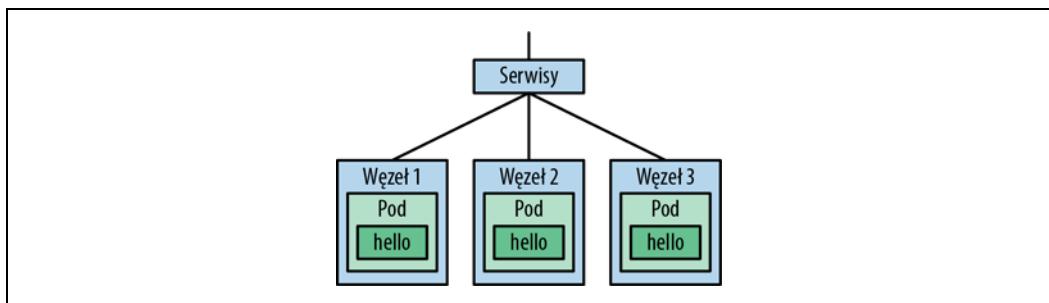
¹ k8s, wymawiane jak kates, jest powszechnym skrótem dla Kubernetes: pierwsza i ostatnia litera plus liczba liter pomiędzy (k-8-s). Zobacz także skróty: i18n (internacjonalizacja, ang. *internationalization*), a11y (dostępność — ang. *accessibility*) oraz o11y (obserwowałość — ang. *observability*).

Co gorsza, może istnieć wiele replik Podów, każda z innym adresem. Każda inna aplikacja, która musi skontaktować się z Podem, musiałaby przechowywać listę tych adresów, co nie wygląda na dobry pomysł.

Na szczęście istnieje lepszy sposób: zasób Serwis podaje pojedynczy, niezmienny adres IP lub nazwę DNS, które zostaną automatycznie przekierowane do dowolnego pasującego Poda. Dalej w książce, w rozdziale 9., omówimy zasób Ingress, który umożliwia bardziej zaawansowany routing oraz korzystanie z certyfikatów TLS.

Na razie jednak opiszemy bliżej, jak działa Serwis.

Możesz myśleć o zasobie Serwis jak o serwerze proxy lub load balancer, przesyłającym żądania do grupy Podów na *zapleczu* (patrz rysunek 4.2). Nie ogranicza się on jednak do portów webowych: Serwis może przekazywać ruch z dowolnego portu do dowolnego innego portu, zgodnie z opisem w części specyfikacji *ports*.



Rysunek 4.2. Serwis zapewnia stały punkt dostępowy dla grupy Podów

Oto manifest w formacie YAML dla naszej aplikacji *demo*:

```
apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
  ports:
    - port: 8888
      protocol: TCP
      targetPort: 8888
  selector:
    app: demo
  type: ClusterIP
```

Widać, że wygląda podobnie do zasobu Deployment, dla którego manifest pokazaliśmy wcześniej. Jednak pole `kind` ma wartość `Service`, a pole `spec` zawiera tylko listę portów (`ports`) oraz selektor i typ.

Jeśli przyjrzesz się uważniej, zobaczyś, że Serwis przekierowuje swój port 8888 do portu Poda o numerze 8888:

```
...
ports:
- port: 8888
  protocol: TCP
  targetPort: 8888
```

Pole selector informuje Serwis, jak kierować żądania do poszczególnych Podów. Żądania będą przekazywane do dowolnych Podów zgodnych z określonym zestawem etykiet (ang. *label*); w naszym przypadku tylko do app: demo (patrz „Etykiety” w rozdziale 9.). W naszym przykładzie istnieje tylko jeden Pod, który pasuje, ale jeśli istnieje wiele Podów, usługa wyśle każde żądanie do losowo wybranego².

Pod tym względem Serwis przypomina trochę tradycyjny load balancer — w rzeczywistości zarówno zasób Serwis, jak i Ingress mogą automatycznie tworzyć chmurowe load balancery (patrz „Zasoby Ingress” w rozdziale 9.).

Na razie najważniejsze jest, aby pamiętać, że Deployment zarządza zestawem Podów dla aplikacji, a Serwis zapewnia pojedynczy punkt wejścia dla żądań do tych Podów.

Pójdźmy dalej i zastosujmy teraz manifest tworzący Serwis:

```
kubectl apply -f k8s/service.yaml
service "demo" created

kubectl port-forward service/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

Tak jak poprzednio, komenda kubectl port-forward połączy Pod *demo* z portem na Twoim komputerze lokalnym, dzięki czemu będziesz mógł połączyć się z adresem <http://localhost:9999> za pomocą przeglądarki internetowej.

Po upewnieniu się, że wszystko działa poprawnie, uruchom następujące polecenie, aby program wyczyścić:

```
kubectl delete -f k8s /
```



Możesz skorzystać z polecenia kubectl delete z przełącznikiem label — jak to zrobiliśmy wcześniej — aby usunąć wszystkie zasoby, które pasują do etykiet (patrz „Etykiety” w rozdziale 9.). Alternatywnie możesz skorzystać z polecenia kubectl delete -f, jak powyżej, i podać katalog manifestów. Wszystkie pliki manifestu zostaną usunięte.

Ćwiczenie

Zmodyfikuj plik *k8s/loyment.yaml*, aby zmienić liczbę replik na 3. Załaduj manifest za pomocą polecenia kubectl apply i sprawdź (przy użyciu polecenia kubectl get pods), czy otrzymasz trzy Pody *demo*.

² Jest to domyślny algorytm równoważenia obciążenia; wersje Kubernetes 1.10+ obsługują również inne algorytmy, np. least connection; patrz <https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/>.

Odpypywanie klastra za pomocą polecenia kubectl

Narzędzie kubectl to „szwajcarski scyzoryk” Kubernetes: zmienia konfigurację, tworzy, modyfikuje i usuwa zasoby, a także może wysyłać zapytania do klastra o informacje na temat istniejących zasobów (w tym aktualny status).

Widziałeś już, jak korzystać z kubectl, aby odpytywać zasoby Pod i Deployment. Możesz również z niego skorzystać, aby sprawdzić, jakie węzły istnieją w klastrze:

```
kubectl get nodes
NAME           STATUS ROLES AGE     VERSION
my-machine     Ready  <none> 3d20h  v1.18.4-1+6f17be3f1fd54a
```

Jeśli chcesz zobaczyć zasoby wszystkich typów, użyj polecenia kubectl get all. (W rzeczywistości nie pokazuje ono dosłownie *wszystkich* zasobów, tylko najczęstsze typy, ale na razie nie będziemy się nad tym zastanawiać).

Aby zobaczyć wyczerpujące informacje na temat pojedynczego Poda (lub dowolnego innego zasobu), skorzystaj z polecenia kubectl describe:

```
kubectl describe pod/demo-dev-6c96484c48-69vss
Name:           demo-dev-6c96484c48-69vss
Namespace:      default
Node:          docker-for-desktop/10.0.2.15
Start Time:    Wed, 06 Jun 2018 10:48:50 +0100
...
Containers:
  demo:
    Container ID: docker://646aaf7c4baf6d...
    Image:        cloudnativized/demo:hello
...
Conditions:
  Type        Status
  Initialized  True
  Ready       True
  PodScheduled  True
...
Events:
  Type  Reason  Age   From            Message
  ----  -----  --   --              --
  Normal Scheduled 1d   default-scheduler  Successfully assigned demo-dev...
  Normal Pulling   1d   kubelet         pulling image "cloudnativized/demo...
  ...

```

W powyższym wyniku widać, że kubectl daje podstawowe informacje o samym kontenerze, w tym jego identyfikator obrazu oraz status, a także uporządkowaną listę zdarzeń, które miały miejsce w tym kontenerze. (O możliwościach kubectl więcej powiemy w rozdziale 7.).

Przenoszenie zasobów na wyższy poziom

Teraz wiesz wszystko, co musisz wiedzieć, aby za pomocą deklaratywnych manifestów YAML wdrażać aplikacje w klastrach Kubernetes. W tych plikach jest jednak wiele powtarzających się wyrażeń: np. kilkakrotnie powtórzyłeś nazwę demo, etykieta app: demo i port 8888.

Czy nie powinieneś określić tych wartości tylko raz, a następnie odwoływać się do nich wszędzie tam, gdzie występują w manifestach Kubernetes?

Przykładowo wspaniale byłoby zdefiniować zmienne o nazwach, takich jak `container.name` i `container.port`, a następnie używać ich tam, gdzie są potrzebne w plikach YAML. Jeśli będziesz chciał zmienić nazwę aplikacji lub numer portu, na którym nasłuchuje, zmienisz je tylko w jednym miejscu, a wszystkie manifesty zostaną zaktualizowane automatycznie.

Na szczęście jest odpowiednie narzędzie, a pod koniec tego rozdziału pokażemy Ci, co potrafi.

Helm: menadżer pakietów Kubernetes

Jeden z popularnych menadżerów pakietów dla Kubernetes nazywa się Helm i działa tak, jak opisano w poprzednim punkcie. Narzędzie `helm` udostępnia wiersz poleceń, za pomocą którego możesz instalować i konfigurować aplikacje (własne lub dowolne inne), a także tworzyć pakiety zwane *wykresami* (ang. *chart*) Helm, które całkowicie określają zasoby potrzebne do uruchomienia aplikacji, jej zależności oraz konfigurowalne ustawienia.

Helm jest częścią rodziny projektów organizacji Cloud Native Computing Foundation (patrz „Model Cloud Native” w rozdziale 1.), co stanowi o jego stabilności i szerokim zastosowaniu.



Musisz zdawać sobie sprawę, że wykres Helm, w przeciwieństwie do binarnych pakietów oprogramowania używanych przez narzędzia, takie jak APT lub Yum, nie zawiera samego obrazu kontenera. Zamiast tego zawiera po prostu metadane dotyczące tego, gdzie można znaleźć obraz, podobnie jak to robi Deployment.

Po zainstalowaniu wykresu Kubernetes sam zlokalizuje i pobierze obraz kontenera binarnego z określonego miejsca. W rzeczywistości wykres Helm jest wygodnym opakowaniem manifestów YAML obiektu Kubernetes.

Instalacja Helm

Postępuj zgodnie z instrukcjami instalacji Helm (https://docs.helm.sh/using_helm/#installing_helm) dla swojego systemu operacyjnego.

Aby sprawdzić, czy Helm działa, wpisz polecenie:

```
helm version
version.BuildInfo{Version:"v3.2.3",
GitCommit:"8f832046e258e2cb800894579b1b3b50c2d83492", GitTreeState:"clean",
GoVersion:"go1.13.12"}
```

Gdy to polecenie się powiedzie, możesz rozpocząć korzystanie z aplikacji Helm.

Instalacja wykresu Helm

Jak wyglądałby wykres Helm dla naszej aplikacji demonstracyjnej? W katalogu `hello-helm3` zobaczysz podkatalog `k8s`, który w poprzednim przykładzie (`hello-k8s`) zawierał tylko pliki manifestu Kubernetes, potrzebne do wdrożenia aplikacji. Teraz katalog `demo` zawiera wykres Helm:

```
ls k8s/demo
Chart.yaml      prod-values.yaml  staging-values.yaml  templates
values.yaml
```

Zobaczmy, do czego służą te wszystkie pliki w punkcie „Co znajduje się w wykresie narzędzia Helm?” w rozdziale 12., ale na razie użyjmy aplikacji Helm, aby zainstalować *demo*. Najpierw wyczyść ślady wszystkich poprzednich operacji:

```
kubectl delete all --selector app=demo
```

Następnie skorzystaj z następującego polecenia:

```
helm install demo ./k8s/demo
NAME: demo
LAST DEPLOYED: Fri Jul 3 08:06:01 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Możesz zobaczyć, że Helm utworzył zasoby Deployment (który uruchamia Pod) oraz Serwis, tak jak w poprzednim przykładzie. Instrukcja `helm install` utworzyła obiekt Secret typu `helm.sh/release` do śledzenia wydania.

Wykresy, repozytoria i wydania

Oto trzy najważniejsze pojęcia narzędzia Helm, które musisz znać.

- *Wykres* to pakiet Helm, zawierający wszystkie definicje zasobów niezbędne do uruchomienia aplikacji w Kubernetes.
- *Repozytorium* to miejsce, w którym mogą być gromadzone i udostępniane wykresy.
- *Wydanie* jest szczególną instancją wykresu uruchomionego w klastrze Kubernetes.

Jeden wykres często można zainstalować wiele razy w tym samym klastrze. Przykładowo może być uruchomionych wiele kopii wykresu serwera WWW Nginx, z których każda obsługuje inną witrynę. Każda oddzielna instancja wykresu jest odrębnym wydaniem.

Wyświetlanie listy wydań Helm

Aby w dowolnym momencie sprawdzić, które wydania są uruchomione, wydaj polecenie `helm list`:

```
helm list
NAME NAMESPACE REVISION UPDATED      STATUS   CHART      APP VERSION
demo  default    1        2020-07-03 ...  DEPLOYED  demo-1.0.1
```

Aby zobaczyć dokładny status konkretnego wydania, skorzystaj z polecenia `helm status` z nazwą wydania. Zobaczysz te same informacje, które wyświetliłeś przy pierwszym uruchomieniu wydania.

Dalej w książce pokażemy, jak zbudować własne wykresy Helm dla aplikacji (patrz „Co znajduje się w wykresie narzędzia Helm?” w rozdziale 12.). Teraz musisz wiedzieć, że Helm to przydatny sposób instalowania aplikacji za pomocą publicznych wykresów.



Możesz zobaczyć pełną listę publicznych wykresów Helm (<https://github.com/helm/charts/tree/master/stable>) na GitHub.

Możesz także uzyskać listę dostępnych wykresów, uruchamiając polecenie `helm search repo` bez argumentów (lub np. `helm search repo redis`, aby wyszukać wykresy dla Redis).

Podsumowanie

To nie jest książka o wewnętrznych elementach Kubernetes (przepraszamy, nie przyjmujemy zwrotów). Naszym celem jest pokazanie, co potrafi Kubernetes, i szybkie doprowadzenie do momentu, w którym możesz uruchomić rzeczywiste zadania produkcyjne. Warto jednak znać przynajmniej niektóre główne elementy maszyny, z którymi będziesz pracować, takie jak Pod czy Deployment. W tym rozdziale krótko przedstawiliśmy niektóre z najważniejszych.

Choć technologia ta jest fascynująca dla geeków takich jak my, jesteśmy również zainteresowani częścią praktyczną. Dlatego nie poruszyliśmy wszystkich informacji związanych z zasobami dostarczonymi przez Kubernetes — ponieważ jest ich wiele, a wielu z nich prawie na pewno nie będziesz potrzebować (przynajmniej jeszcze nie teraz).

Oto wiadomości, które powinieneś zapamiętać z tego rozdziału.

- Pod jest podstawową jednostką pracy w Kubernetes, określającą pojedynczy kontener lub grupę komunikujących się kontenerów, wykonujących tę samą pracę.
- Deployment jest zasobem wysokiego poziomu Kubernetes, który deklaratywnie zarządza Podami — tworzy, uruchamia, aktualizuje i restartuje, jeśli trzeba.
- Serwis w Kubernetes jest odpowiednikiem load balansera lub serwera proxy; kieruje ruch do odpowiednich pasujących Podów poprzez jeden, znany i stały adres IP lub nazwę DNS.
- Scheduler w Kubernetes sprawdza, czy dany Pod nie jest jeszcze uruchomiony, znajduje odpowiedni węzeł i instruuje kubelet w tym węźle, aby uruchomił ten Pod.
- Zasoby, takie jak Deployment, są reprezentowane przez rekordy w wewnętrznej bazie danych Kubernetes; zewnętrznie zasoby te mogą być reprezentowane przez pliki tekstowe (znane jako *manifesty*) w formacie YAML; manifest jest deklaracją pożądanego stanu zasobu.
- `kubectl` jest głównym narzędziem służącym do interakcji z Kubernetes, umożliwiającym stosowanie manifestów, odpytywanie zasobów, wprowadzanie zmian, usuwanie zasobów i wykonywanie wielu innych zadań.
- Helm jest menadżerem pakietów Kubernetes; upraszcza konfigurowanie i wdrażanie aplikacji Kubernetes, umożliwiając korzystanie z jednego zestawu wartości (takich jak nazwa aplikacji lub port nasłuchiwanego) oraz zestawu szablonów do generowania plików YAML — bez konieczności samodzielnego tworzenia plików YAML od zera.

Zarządzanie zasobami

Nic nie jest wystarczające dla człowieka, dla którego wystarczająco to za mało.

— Epikur

W tym rozdziale przyjrzymy się, jak najlepiej wykorzystać kластer: jak zarządzać zasobami i optymalizować wykorzystanie zasobów, jak zarządzać cyklem życia kontenerów oraz jak dzielić kластer za pomocą przestrzeni nazw. Przedstawimy również niektóre techniki i najlepsze praktyki optymalizujące koszty klastra.

Dowiesz się, jak używać żądań, limitów i wartości domyślnych zasobów oraz jak je optymalizować za pomocą Vertical Pod Autoscaler; jak używać sond gotowości, sond żywotności i budżetów zakłóceń Poda do zarządzania kontenerami; jak zoptymalizować ilość miejsca w chmurze oraz w jaki sposób i kiedy używać instancji zastrzeżonych (ang. *reserved*) lub w trybie wywłaszczeniowym (ang. *pre-emptible*) do kontrolowania kosztów.

Zrozumienie działania zasobów

Założymy, że masz kластer Kubernetes o określonej pojemności i rozsądnej liczbie węzłów w odpowiednim rozmiarze. Jak z takiego rozwiązania osiągnąć jak największy zysk? W jaki sposób uzyskać najlepsze możliwe wykorzystanie dostępnych zasobów klastra, zapewniając jednocześnie wystarczającą rezerwę, aby obsłużyć skokowy ruch, awarie węzłów i błędy związane z wdrożeniami?

Aby odpowiedzieć na to pytanie, postaw się w miejscu schedulera Kubernetes i spróbuj zobaczyć rzeczy z jego punktu widzenia. Zadaniem schedulera jest podjęcie decyzji, gdzie uruchomić dany Pod. Czy są jakieś węzły z wystarczającą ilością wolnych zasobów, aby uruchomić na nich Pod?

Odpowiedź na to pytanie jest niemożliwa, chyba że scheduler otrzyma informację, ile zasobów Pod będzie musiał uruchomić. Pod, który wymaga 1 GiB pamięci, nie może zostać uruchomiony w węźle z jedynie 100 MiB wolnej pamięci.

Scheduler musi być także w stanie podjąć działania, gdy zachłanny Pod zużywa zbyt wiele zasobów i negatywnie wpływa na inne Pody w tym samym węźle. Jednak co to znaczy za dużo? Aby efektywnie zaplanować działania Podów, scheduler musi znać minimalne i maksymalne dopuszczalne wymagania zasobów dla każdego Poda.

Właśnie tutaj pojawiają się żądania i limity zasobów Kubernetes. Kubernetes radzi sobie z zarządzaniem dwoma rodzajami zasobów, czyli procesorem i pamięcią. Istnieją również inne ważne rodzaje zasobów, takie jak przepustowość sieci, operacje dyskowe I/O (IOPS) i miejsce na dysku. Dostęp do tych zasobów może powodować konflikty w klastrze. W tym miejscu w Kubernetes nie ma jeszcze sposobu na określenie tych wymagań dla Podów.

Jednostki zasobów

Wykorzystanie procesora dla Podów jest wyrażane, jak można się spodziewać, w jednostkach CPU. Jedna jednostka CPU jest równoważna jednemu procesorowi wirtualnemu AWS, jednemu Google Cloud Core, jednemu Azure vCore lub jednemu *wątkowi* w procesorze bare-metal obsługującym wielowątkowość. Innymi słowy, 1 CPU w kategoriach Kubernetes oznacza to, co według Ciebie działa.

Ponieważ większość Podów nie potrzebuje całego CPU, żądania i limity są zwykle wyrażane w *milicpusach* (czasem nazywanych *millicore'ami*). Pamięć mierzona jest w bajtach, a dokładniej w *mebibajtach* (MiB).

Żądania zasobów

W *żądaniu zasobu* Kubernetes określa minimalną ilość tego zasobu, jakiej potrzebuje Pod do uruchomienia. Przykładowo żądanie zasobu wynoszące 100m (100 milicpus) i 250Mi (250 MiB pamięci) oznacza, że Pod nie może zostać uruchomiony w węźle z mniejszą ilością dostępnych zasobów niż określone. Jeśli nie ma dostępnego węzła o wystarczającej pojemności, Pod pozostanie w *stanie oczekiwania* (ang. *pending*), dopóki takiego węzła nie będzie.

Jeśli np. wszystkie węzły klastra mają dwa rdzenie CPU i 4 GiB pamięci, kontener żądający 2,5 CPU lub 5 GiB pamięci nigdy nie zostanie uruchomiony.

Zobaczmy, jak wyglądałyby żądania zasobów w naszej aplikacji demonstracyjnej:

```
spec:  
  containers:  
    - name: demo  
      image: cloudnative/demo:hello  
      ports:  
        - containerPort: 8888  
      resources:  
        requests:  
          memory: "10Mi"  
          cpu: "100m"
```

Limity zasobów

Limit zasobów określa maksymalną ilość zasobów, z których Pod może korzystać. Pod, który próbuje wykorzystać więcej niż przydzielony limit CPU, zostanie „zdławiony” i obniży swoją wydajność.

Pod, który spróbuje wykorzystać więcej niż dozwolony limit pamięci, zostanie zakończony. Jeśli zakończony Pod może zostać ponownie uruchomiony, to tak będzie. W praktyce może to oznaczać, że Pod jest po prostu restartowany w tym samym węźle.

Niektóre aplikacje, takie jak serwery sieciowe, mogą z czasem zużywać coraz więcej zasobów w odpowiedzi na rosnące zapotrzebowanie. Określenie limitów zasobów to dobry sposób na zapobieganie wykorzystywaniu przestrzeni klastra przez tak zachłanne Pody.

Oto przykład ustawiania limitów zasobów w aplikacji demonstracyjnej:

```
spec:  
  containers:  
    - name: demo  
      image: cloudnativelabs/demo:hello  
      ports:  
        - containerPort: 8888  
      resources:  
        limits:  
          memory: "20Mi"  
          cpu: "250m"
```

Wiedza, jakie limity należy ustawić dla konkretnej aplikacji, jest kwestią obserwacji i osądu (patrz „Optymalizacja Podów” dalej w tym rozdziale).

Kubernetes pozwala na nadmierne *zaangażowanie zasobów*; tzn. suma wszystkich limitów zasobów kontenerów w węźle może przekraczać łączne zasoby tego węzła. Jest to rodzaj hazardu: scheduler obstawia, że przez większość czasu większość kontenerów nie będzie musiała przekraczać swoich limitów zasobów.

Jeśli zakład się nie powiedzie, a całkowite wykorzystanie zasobów zacznie zbliżać się do maksymalnej pojemności węzła, Kubernetes zacznie bardziej agresywnie kończyć pracę kontenerów. W warunkach nadmiernego zapotrzebowania na zasoby kontenery, które przekroczyły swoje wartości żądań, ale nie przekroczyły limitów, mogą być nadal zamkane¹.

Jeśli Kubernetes musi zakończyć pracę Podów, zacznie od tych, które najbardziej przekroczyły swoje wartości żądań. Pody mieściące się w swoich wartościach żądań nie zostaną zakończone, z wyjątkiem bardzo rzadkich sytuacji, w których Kubernetes nie byłby w stanie uruchomić komponentów systemu, takich jak kubelet.



Najlepsze praktyki

Zawsze określaj żądania zasobów i limity dla swoich kontenerów. Pomaga to aplikacji Kubernetes odpowiednio planować swoje Pody i zarządzać nimi.

Utrzymuj małe kontenery

W punkcie „Minimalne obrazy kontenerów” w rozdziale 2. wspomnieliśmy, że utrzymywanie jak najmniejszych obrazów kontenera, z wielu powodów, jest dobrym pomysłem.

- Małe kontenery budują się szybciej.
- Obrazy zajmują mniej miejsca.

¹ Dla poszczególnych kontenerów możliwe jest dostosowanie tego zachowania za pomocą klas jakości usługi (<https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/>) (QoS — ang. *Quality of Service*).

- Krócej trwa ładowanie obrazów.
- Są mniejsze możliwości ataku.

Jeśli korzystasz z języka Go, jesteś już na dobrej drodze, ponieważ Go może skompilować Twoją aplikację do jednego statycznego pliku binarnego. Jeśli w Twoim kontenerze jest tylko jeden plik, jest on tak mały, jak to tylko możliwe!

Zarządzanie cyklem życia kontenera

Wiesz już, że Kubernetes może najlepiej zarządzać Podami, gdy otrzymuje informacje, jakie są ich wymagania dotyczące procesora i pamięci. Jednak musi także wiedzieć, kiedy kontener działa, tzn. kiedy działa poprawnie i jest gotowy do obsługi żądań.

Aplikacje w kontenerze dosyć często się zawieszają, tzn. proces nadal jest uruchomiony, ale nie obsługuje żadnych żądań. Kubernetes potrzebuje sposobu na wykrycie tej sytuacji, aby mógł ponownie uruchomić kontener w celu rozwiązania problemu.

Sondy żywotności

Kubernetes pozwala wprowadzić *sondę żywotności* jako część specyfikacji kontenera, tj. kontrolę stanu, która określa, czy kontener działa.

W przypadku kontenera z serwerem HTTP specyfikacja sondy żywotności zwykle wygląda mniej więcej tak:

```
livenessProbe:  
  httpGet:  
    path: /healthz  
    port: 8888  
    initialDelaySeconds: 3  
    periodSeconds: 3
```

Sonda `httpGet` wysyła żądanie HTTP do identyfikatora URI i określonego portu; w tym przypadku `/healthz` na porcie 8888.

Jeśli Twoja aplikacja nie ma określonego punktu wejścia do sprawdzenia kontroli poprawności, możesz użyć / lub dowolnego poprawnego adresu URL dla swojej aplikacji. Jednak powszechną praktyką jest tworzenie punktu wejścia `/healthz`, służącego tylko do tego celu. (Dlaczego dodano z? Aby mieć pewność, że nie koliduje z istniejącą ścieżką, taką jak `health` — np. może to być strona z informacjami o zdrowiu).

Jeśli aplikacja odpowiada kodem stanu HTTP 2xx lub 3xx, Kubernetes uważa ją za działającą. Jeśli zareaguje inaczej lub w ogóle nie zareaguje, kontener będzie uznany za martwy i zostanie ponownie uruchomiony.

Opóźnienie i częstotliwość sondy

Jak często Kubernetes powinien używać sondy żywotności? Żadna aplikacja nie może zostać uruchomiona natychmiast. Jeśli Kubernetes wypróbuje sondę żywotności natychmiast po uruchomieniu kontenera, prawdopodobnie kontrola zawiedzie i spowoduje to ponowne uruchomienie kontenera — i ta pętla będzie się powtarzać w nieskończoność!

Pole `initialDelaySeconds` pozwala w Kubernetes określić, jak długo trzeba czekać przed wypróbowaniem pierwszej sondy żywotności, co pozwoli uniknąć tej *nieskończonej pętli*.

Podobnie, nie byłoby dobrym pomysłem, aby tysiące razy na sekundę wywoływać żądania do ścieżki `healthz`. Pole `periodSeconds` określa, jak często należy sprawdzać sondę żywotności; w tym przykładzie co trzy sekundy.

Inne typy sond

`httpGet` nie jest jedynym dostępnym rodzajem sondy; dla serwerów sieciowych, które nie obsługują protokołu HTTP, możesz skorzystać np. z `tcpSocket`:

```
livenessProbe:  
  tcpSocket:  
    port: 8888
```

Jeśli połączenie TCP z określonym portem powiedzie się, kontener działa.

Möżesz także uruchomić dowolne polecenie na kontenerze, używając sondy `exec`:

```
livenessProbe:  
  exec:  
    command:  
      - cat  
      - /tmp/healthy
```

Sonda `exec` uruchamia określoną komendę w kontenerze, a działanie sondy kończy się powodzeniem, jeśli komenda się powiedzie (tzn. kończy działanie z zerowym statusem). Sonda `exec` jest zwykle bardziej przydatna jako sonda gotowości. W jaki sposób jej używać, zobaczysz w następnym punkcie.

Sondy gRPC

Chociaż wiele aplikacji i usług komunikuje się przez HTTP, coraz popularniejsze jest stosowanie protokołu gRPC (<https://grpc.io/>), szczególnie w przypadku mikrousług. Protokół gRPC to wydajny, przenośny, binarny protokół sieciowy opracowany przez Google i obsługiwany przez Cloud Native Computing Foundation.

Sondy `httpGet` nie będą działać z serwerami gRPC i zamiast tego możesz skorzystać z sondy `tcpSocket`. Wtedy jednak dostaniesz tylko informację, że możesz nawiązać połączenie z danym socketem, a nie, że sam serwer działa.

gRPC ma standardowy protokół sprawdzania stanu, który obsługuje większość usług gRPC. Aby wykorzystać go jako sondę żywotności Kubernetes, można użyć narzędzia `grpc-health-probe` (<https://kubernetes.io/blog/2018/10/01/health-checking-grpc-servers-on-kubernetes/>). Jeśli dodasz to narzędzie do swojego kontenera, możesz sprawdzić działanie za pomocą sondy `exec`.

Sondygotowości

Sonda żywotności to sonda gotowości z inną semantyką. Czasami aplikacja musi zasygnalizować Kubernetes, że tymczasowo nie jest w stanie obsłużyć żądań; być może dlatego, że wykonuje długi proces inicjacji lub czeka na zakończenie podprocesu. Sonda gotowości spełnia tę funkcję.

Dopóki Twoja aplikacja nie zacznie nasłuchiwać połączeń HTTP, dopóty nie będzie gotowa do działania. Sonda gotowości może działać tak samo jak sonda żywotności:

```
readinessProbe:  
  httpGet:  
    path: /healthz  
    port: 8888  
  initialDelaySeconds: 3  
  periodSeconds: 3
```

Kontener, który nie przejdzie testu sondy gotowości, zostanie usunięty z dowolnych Serwisów obsługujących dany Pod. To jest jak usunięcie uszkodzonego węzła z puli load balancera: żaden ruch nie zostanie wysłany do Poda, dopóki jego sonda gotowości nie przejdzie pozytywnego testu.

Zwykle po uruchomieniu Poda Kubernetes rozpoczęcie wysyłanie ruchu, gdy tylko kontener będzie działał (`state: running`). Jeśli jednak kontener ma zaimplementowaną sondę gotowości, Kubernetes poczeka, aż sonda zakończy test z sukcesem, zanim wyśle do kontenera wszelkie żądania — aby użytkownicy nie widzieli błędów związanych z niegotowym kontenerem. Jest to niezwykle ważne w przypadku aktualizacji zero-downtime (aktualizacje bez przestojów). Aby znaleźć więcej informacji na ten temat, patrz „Strategie wdrażania” w rozdziale 13.

Kontener, który nie jest gotowy, nadal będzie miał status `Running`, ale w kolumnie `READY` pojawi się co najmniej jeden nieprzygotowany kontener w Podzie:

```
kubectl get pods  
NAME      READY STATUS RESTARTS AGE  
readiness-test 0/1  Running 0 56s
```



Sondygotowości powinny zwracać tylko status HTTP 200 OK. Chociaż Kubernetes uważa oba kody stanu 2xx i 3xx za poprawne, to usługi load balancera — nie. Jeśli korzystasz z zasobu Ingress w połączeniu z modułem load balancera w chmurze (patrz „Zasoby Ingress” w rozdziale 9.), a Twoja sonda gotowości zwraca np. kod 301 (przekierowanie), moduł load balancera może oznaczyć Pody jako niesprawne. Upewnij się, że Twoje sondygotowości zwracają tylko kod statusu 200.

Sondygotowości na podstawie pliku

Alternatywnie możesz poprosić aplikację o utworzenie pliku w systemie plików kontenera np. o nazwie `/tmp/healthy` i użyć sondygotowości `exec`, aby sprawdzić obecność tego pliku.

Ten rodzaj sondy gotowości może być przydatny: jeśli chcesz tymczasowo wyłączyć kontener z eksploatacji w celu zdebugowania jakiegoś problemu, możesz usunąć plik `/tmp/healthy`. Następna sonda gotowości zakończy się niepowodzeniem, a Kubernetes usunie kontener ze wszystkich pasujących Serwisów. (Lepszym rozwiązaniem jest jednak zmiana etykiet kontenera, aby nie pasowały już do Serwisu: patrz „Serwis” w rozdziale 4.).

W tym momencie możesz już sprawdzać i rozwiązywać problemy z kontenerem. Po zakończeniu prac możesz albo zamknąć kontener i wdrożyć poprawioną wersję, albo umieścić plik sondy z powrotem na swoim miejscu, aby kontener ponownie zaczął odbierać ruch.



Najlepsze praktyki

Użyj sond gotowości i sond żywotności, aby poinformować Kubernetes, kiedy aplikacja jest gotowa do obsługi żądań lub kiedy ma problem i wymaga ponownego uruchomienia.

minReadySeconds

Domyślnie kontener lub Pod są uważane za gotowe do użycia w momencie, gdy ich sonda gotowości zakończy się powodzeniem. W niektórych przypadkach możesz uruchomić kontener na chwilę, aby upewnić się, że jest stabilny. Podczas wdrażania Kubernetes przed uruchomieniem następnego Poda czeka, aż każdy nowy Pod będzie gotowy (patrz „Rolling updates” w rozdziale 13.). Jeśli wadliwy kontener natychmiast ulegnie awarii, spowoduje to zatrzymanie procesu wdrażania, ale jeśli do czasu wystąpienia awarii minie kilka sekund, wszystkie jego repliki mogą zostać wdrożone przed wykryciem problemu.

Aby tego uniknąć, możesz ustawić wartość dla pola `minReadySeconds` w kontenerze. Kontener lub Pod nie będą uważane za gotowe, dopóki sonda gotowości nie będzie gotowa przez `minReadySeconds` sekund (domyślnie 0).

Budżety zakłóceń Poda

Czasami Kubernetes musi zatrzymać Twoje Pody, nawet jeśli działają (proces eksmisji — *eviction*). Przykładowo gdy węzeł, na którym działają, jest opróżniany przed aktualizacją, a Pody należy przemieścić do innego węzła.

Nie musi to jednak powodować przestojów aplikacji, pod warunkiem że może zostać uruchomiona wystarczająca liczba replik. Możesz skorzystać z zasobu `PodDisruptionBudget`, aby dla danej aplikacji określić, na utratę ilu Podów w danym momencie pozwalasz.

Możesz np. ustalić, że nie więcej niż 10% Podów aplikacji może zostać zakłóconych jednocześnie. Możesz też określić, że Kubernetes może eksmitować dowolną liczbę kapsuł, pod warunkiem że zawsze działają co najmniej trzy repliki.

minAvailable

Oto przykład konfiguracji `PodDisruptionBudget`, określającej minimalną liczbę Podów, które mają działać — za pomocą pola `minAvailable`:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: demo-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: demo
```

W tym przykładzie `minAvailable: 3` określa, że przynajmniej trzy Pody pasujące do etykiety `app: demo` powinny zawsze być uruchomione. Kubernetes może eksmitować tyle Podów `demo`, ile chce — o ile zawsze pozostaą co najmniej trzy.

maxUnavailable

I odwrotnie, możesz użyć pola `maxUnavailable`, aby ograniczyć całkowitą liczbę lub odsetek Podów, które Kubernetes może eksmitować:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: demo-pdb
spec:
  maxUnavailable: 10%
  selector:
    matchLabels:
      app: demo
```

W tym przypadku nie można eksmitować więcej niż 10% Podów `demo`. Dotyczy to jednak tylko tzw. *eksmissions dobrowolnych* (ang. *voluntary evictions*); tzn. eksmissions zainicjowanych przez Kubernetes. Jeśli np. węzeł ulegnie awarii sprzętowej lub zostanie usunięty, Pody zostaną mimowolnie eksmitowane, nawet jeśli naruszałyby to budżet zakłóceń.

Ponieważ Kubernetes będzie miał tendencję do równomiernego rozmieszczenia Podów w węzłach, wszystkie inne rzeczy pozostają równe. Warto o tym pamiętać, biorąc pod uwagę liczbę węzłów potrzebnych w klastrze. Jeśli masz trzy węzły, awaria jednego z nich może spowodować utratę jednej trzeciej wszystkich Podów, a to może nie wystarczyć do utrzymania akceptowalnego poziomu usług (patrz „*Wysoka niezawodność*” w rozdziale 3.).



Najlepsze praktyki

Dla aplikacji o kluczowym znaczeniu dla firmy ustaw `PodDisruptionBudget`, aby upewnić się, że zawsze jest wystarczająca liczba replik do utrzymania usługi nawet wtedy, gdy eksmitowane są Pody.

Korzystanie z przestrzeni nazw

Innym bardzo przydatnym sposobem zarządzania wykorzystaniem zasobów w klastrze jest użycie przestrzeni *nazw* (ang. *namespace*). Przestrzeń nazw Kubernetes to sposób na podzielenie klastra na osobne podziały, niezależnie od przeznaczenia.

Możesz np. mieć przestrzeń nazw prod dla aplikacji produkcyjnych i testową przestrzeń nazw test do przetestowania różnych rzeczy. Jak sugeruje termin *przestrzeń nazw*, nazwy w jednej przestrzeni nazw nie są widoczne z innej przestrzeni nazw.

Oznacza to, że możesz mieć usługę o nazwie demo w przestrzeni nazw prod i inną usługę o nazwie demo w testowej przestrzeni nazw test, i nie będzie żadnego konfliktu.

Aby zobaczyć przestrzenie nazw istniejące w klastrze, uruchom następujące polecenie:

```
kubectl get namespaces
NAME      STATUS  AGE
default   Active  1y
kube-public   Active  1y
kube-system  Active  1y
```

Przestrzenie nazw można traktować tak samo jak foldery na dysku twardym komputera. Chociaż możesz przechowywać wszystkie pliki w tym samym folderze, jest to niewygodne. Poszukiwanie konkretnego pliku byłoby czasochłonne i niełatwo byłoby sprawdzić, które pliki należą do jakich przestrzeni nazw. Przestrzeń nazw grupuje powiązane zasoby i ułatwia pracę z nimi. Jednak w przeciwieństwie do folderów przestrzeni nazw nie można zagnieżdżać.

Praca z przestrzeniami nazw

Do tej pory, podczas pracy z Kubernetes, zawsze korzystaliśmy z *domyślnej przestrzeni nazw* (ang. *default namespace*). Jeśli w poleceniu kubectl run nie określisz przestrzeni nazw, polecenie będzie działać w domyślnej przestrzeni nazw. Jeśli zastanawiasz się, czym jest przestrzeń nazw *kube-system*, to działają tam wewnętrzne komponenty systemu Kubernetes, dzięki czemu są one odseparowane od Twoich aplikacji.

Jeśli określisz przestrzeń nazw za pomocą flagi --namespace (lub w skrócie -n), polecenie użyje tej przestrzeni nazw. Aby np. uzyskać listę Podów w przestrzeni nazw prod, wpisz:

```
kubectl get pods --namespace prod
```

Jakich przestrzeni nazw powinieneś używać?

Od Ciebie zależy, na jakie przestrzenie nazw podzieliś klasterek. Jednym z intuicyjnych pomysłów jest wydzielenie jednej przestrzeni nazw na aplikację lub na zespół. Możesz np. utworzyć przestrzeń nazw demo, w ramach której uruchomisz aplikację demonstracyjną. Możesz utworzyć przestrzeń nazw za pomocą zasobu *Namespace*, np.:

```
apiVersion: v1
kind: Namespace
metadata:
  name: demo
```

Aby wykorzystać powyższy manifest, skorzystaj z polecenia kubectl apply -f (aby znaleźć więcej informacji na ten temat, patrz „Manifesty zasobów w formacie YAML” w rozdziale 4.). Wszystkie manifesty YAML, dla wszystkich przykładów z tego rozdziału, znajdują się w repozytorium aplikacji demo, w katalogu *hello-namespace*:

```
cd demo/hello-namespace
ls k8s
```

```
deployment.yaml limitrange.yaml namespace.yaml resourcequota.yaml  
service.yaml
```

Możesz pójść krok dalej i utworzyć przestrzenie nazw dla każdego środowiska, w którym działa Twoja aplikacja, np. demo-prod, demo-staging, demo-test itd. Przestrzeń nazw może być wykorzystana jako rodzaj tymczasowego *wirtualnego klastra* (ang. *virtual cluster*) i usunięta po zakończeniu prac. Jednak bądź ostrożny! Usunięcie przestrzeni nazw spowoduje usunięcie wszystkich zasobów w niej zawartych. (Patrz „Kontrola dostępu oparta na rolach (RBAC)” w rozdziale 11., aby dowiedzieć się, jak udzielać lub odmawiać uprawnień użytkownikowi w poszczególnych przestrzeniach nazw).

W obecnej wersji Kubernetes nie ma sposobu na ochronę zasobu, takiego jak przestrzeń nazw, przed usunięciem (propozycja zmian dostępna jest pod adresem <https://github.com/kubernetes/kubernetes/issues/10179>). Więc nie usuwaj nazw, chyba że naprawdę są tymczasowe i na pewno nie zawierają żadnych zasobów produkcyjnych.



Najlepsze praktyki

Utwórz osobne przestrzenie nazw dla każdej aplikacji lub dla każdego logicznego elementu infrastruktury. Nie używaj domyślnej przestrzeni nazw: zbyt łatwo popełnić błąd!

Jeśli chcesz zablokować cały ruch sieciowy do określonej przestrzeni nazw lub z niej, możesz użyć zasad sieciowych Kubernetes (ang. *Kubernetes Network Policies*) (<https://kubernetes.io/docs/concepts/services-networking/network-policies/>).

Adresy serwisów

Chociaż przestrzenie nazw są odizolowane, nadal mogą komunikować się z Serwisami w innych przestrzeniach nazw. Patrz „Serwis” w rozdziale 4. — tam pisaliśmy, że każdy serwis Kubernetes ma powiązaną nazwę DNS, służącą do komunikacji. Łącząc się z hostem demo, połączysz się z Serwisem o nazwie demo. Jak to działa w różnych przestrzeniach nazw?

Nazwy usług DNS zawsze mają następujący format:

SERVICE.NAMESPACE.svc.cluster.local

Część `.svc.cluster.local` jest opcjonalna, podobnie jak przestrzeń nazw. Jeśli jednak np. chcesz skomunikować się z serwisem demo w przestrzeni nazw prod, możesz użyć nazwy:

demo.prod

Jeśli nawet masz tuzin różnych usług o nazwie demo, każda jest we własnej przestrzeni nazw. Możesz dodać przestrzeń nazw do nazwy DNS serwisu, aby dokładnie określić, o którą z nich chodzi.

Przydziły zasobów (ang. Resource Quotas)

Oprócz przydzielenia ograniczeń zużycia CPU i pamięci przez poszczególne kontenery, o czym dowiedziałeś się w punkcie „Żądania zasobów” w tym rozdziale, możesz (i powinieneś) ograniczyć wykorzystanie zasobów w danej przestrzeni nazw. Sposobem na to jest utworzenie *ResourceQuota* w przestrzeni nazw. Oto przykład *ResourceQuota*:

```
ResourceQuota:  
apiVersion: v1  
kind: ResourceQuota  
metadata:  
  name: demo-resourcequota  
spec:  
  hard:  
    pods: "100"
```

Zastosowanie tego manifestu do określonej przestrzeni nazw (np. `demo`) ustala twardy limit (ang. *hard limit*) określający, że jednocześnie w tej przestrzeni nazw może działać do 100 Podów. (Zauważ, że wartość `metadata.name` zasobu *ResourceQuota* może być dowolna. Przestrzenie nazw, na które wpływa ten limit, zależą od tego, do których przestrzeni nazw zastosujesz manifest).

```
cd demo/hello-namespace  
kubectl create namespace demo  
namespace "demo" created  
kubectl apply --namespace demo -f k8s/resourcequota.yaml  
resourcequota "demo-resourcequota" created
```

Teraz Kubernetes zablokuje wszelkie operacje API w przestrzeni nazw demonstracji, które przekroczyłyby limit. Powyższy przykład *ResourceQuota* ogranicza przestrzeń nazw do 100 Podów, więc jeśli działa już 100 Podów i spróbujesz uruchomić nowy, pojawi się komunikat o błędzie:

```
Error from server (Forbidden): pods "demo" is forbidden: exceeded quota:  
  demo-resourcequota, requested: pods=1, used: pods=100, limited: pods=100
```

Korzystanie z *ResourceQuota* jest dobrym sposobem, aby powstrzymać aplikację przed gromadzeniem zbyt wielu zasobów w jednej przestrzeni nazw i ograniczaniem ich w innych częściach klastra.

Chociaż możesz także całkowicie ograniczyć wykorzystanie CPU i pamięci Podów w przestrzeni nazw, nie zalecamy tego. Jeśli ustawisz limity na dość niskim poziomie, prawdopodobnie spowodują one nieoczekiwane i trudne do wykrycia problemy, gdy obciążenia zbliżą się do wartości granicznych. Jeśli ustawisz je bardzo wysoko, ich ustawianie nie ma większego sensu.

Ustawienie limitu Poda jest jednak przydatne, aby np. zapobiec generowaniu potencjalnie nieograniczonej liczby Podów — np. w wyniku błędnej konfiguracji. Łatwo zapomnieć o oczyszczeniu jakiegoś obiektu z typowych zadań. Któregoś dnia można się nagle przekonać, że tysiące z nich zapychają Twój klaster.



Najlepsze praktyki

Użyj *ResourceQuota* w każdej przestrzeni nazw, aby wymusić ograniczenie liczby Podów, które mogą działać w tej przestrzeni nazw.

Żeby sprawdzić, czy *ResourceQuota* jest aktywny w określonej przestrzeni nazw, użyj komendy `kubectl get resourcequotas`:

```
kubectl get resourcequotas -n demo  
NAME          AGE  
demo-resourcequota 15d
```

Domyślne żądania zasobów i limity

Nie zawsze łatwo przewidzieć, jakie będą wymagania dotyczące zasobów Twojego kontenera. Za pomocą zasobu *LimitRange* możesz ustawić domyślne żądania i limity dla wszystkich kontenerów w przestrzeni nazw:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: demo-limitrange
spec:
  limits:
    - default:
        cpu: "500m"
        memory: "256Mi"
    defaultRequest:
        cpu: "200m"
        memory: "128Mi"
  type: Container
```



Podobnie jak w przypadku *ResourceQuota*, wartość `metadata.name` zasobu *LimitRange* może być dowolna. Może np. nie być zgodna z nazwą przestrzeni nazw Kubernetes. *LimitRange* lub *ResourceQuota* działają w określonej przestrzeni nazw tylko wtedy, gdy zastosujesz do tej przestrzeni nazw dany manifest.

Każdy kontener w przestrzeni nazw, który nie określa limitu zasobów ani żądania, odziedziczy wartość domyślną z *LimitRange*. Przykładowo kontener bez określonej wartości żądania `cpu` odziedziczy wartość 200m zasobu *LimitRange*. Podobnie kontener bez określonego limitu `memory` odziedziczy wartość 256Mi zasobu *LimitRange*.

Theoretycznie można ustawić wartości domyślne w *LimitRange* i nie zwracać sobie głowy okresem żądań lub limitów dla poszczególnych kontenerów. Nie jest to jednak dobra praktyka: powinno być możliwe przejrzenie specyfikacji kontenera i sprawdzenie, jakie żądania i limity są ustawione, bez konieczności sprawdzania, czy obowiązuje *LimitRange*. Zasób *LimitRange* może być używany tylko jako zabezpieczenie w przypadku tych właścicieli kontenerów, którzy zapomnieli określić żądania i limity.



Najlepsze praktyki

Skorzystaj z *LimitRanges* w każdej przestrzeni nazw, aby ustawić domyślne żądania zasobów oraz limity dla kontenerów. Nie polegaj na nich, ale traktuj je jako zabezpieczenie. Zawsze określaj wyraźne żądania i limity w samej specyfikacji kontenera.

Optymalizacja kosztów klastra

W podrozdziale „Rozmiar i skalowanie klastra” w rozdziale 6. przedstawiliśmy kilka uwag dotyczących wyboru początkowego rozmiaru klastra i skalowania go w czasie w miarę ewolucji obciążień. Zakładamy, że kластер ma prawidłowy rozmiar i wystarczającą pojemność. Jak należy go uruchomić w najbardziej opłacalny sposób?

Optymalizacja obiektów Deployment

Czy naprawdę potrzebujesz tak wielu replik? To może wydawać się oczywiste, ale każdy Pod w klastrze zużywa zasoby, które są niedostępne dla innych Podów.

Uruchomienie dużej liczby replik może być kuszące, jeżeli chodzi o jakość usług, która nigdy nie ulegnie obniżeniu w przypadku awarii poszczególnych Podów lub podczas ciągłych aktualizacji. Im więcej replik, tym większy ruch mogą obsłużyć Twoje aplikacje.

Jednak replik powinieneś używać mądrze. Twój klaster może uruchomić tylko skońzoną liczbę Podów. Przydzieli ją aplikacjom, które naprawdę potrzebują maksymalnej dostępności i wydajności.

Jeśli naprawdę nie ma znaczenia to, że dany Deployment zostanie wyłączony na kilka sekund podczas aktualizacji, to nie wymaga dużej liczby replik. Zaskakująco duża liczba aplikacji i serwisów może doskonale sobie poradzić z jedną lub dwiema replikami.

Przejrzyj liczbę replik skonfigurowanych dla każdego obiektu Deployment i odpowiedz na poniższe pytania.

- Jakie są wymagania biznesowe dotyczące wydajności i dostępności tej usługi?
- Czy możemy spełnić te wymagania przy mniejszej liczbie replik?

Jeśli aplikacja ma problemy z obsługą żądań lub użytkownicy otrzymują zbyt wiele błędów podczas uaktualniania obiektów Deployment, potrzebuje więcej replik. Jednak w wielu przypadkach można znacznie zmniejszyć rozmiar obiektów Deployment, zanim degradacja zacznie być zauważalna.



Najlepsze praktyki

Użyj minimalnej liczby Podów (która spełni Twoje wymagania dotyczące wydajności i dostępności) dla danego obiektu Deployment. Ustal liczbę replik do poziomu nieco powyżej punktu, w którym osiągane są wymagane cele poziomu usług.

Optymalizacja Podów

Wcześniej w tym rozdziale, w punkcie „Żądania zasobów”, podkreśliśmy znaczenie ustawienia prawidłowych żądań zasobów i limitów dla Twoich kontenerów. Jeśli żądania zasobów są zbyt małe, wkrótce się o tym dowiesz, gdyż Pody przestaną się uruchamiać. Jeśli jednak są one zbyt duże, dowiesz się o tym fakcie dopiero wtedy, gdy otrzymasz miesięczny rachunek za usługi chmurowe.

Powinieneś regularnie sprawdzać żądania zasobów oraz limity dla różnych obciążień i porównywać je z faktycznie używanymi.

Większość zarządzanych usług Kubernetes oferuje pewnego rodzaju pulpit nawigacyjny pokazujący zużycie CPU i pamięci przez kontenery w czasie — więcej na ten temat w podrozdziale „Monitorowanie statusu klastra” w rozdziale 11.

Możesz także tworzyć własne pulpity nawigacyjne i statystyki za pomocą narzędzia *Prometheus and Grafana* — omówimy to szczegółowo w rozdziale 15.

Ustawienie optymalnych żądań i limitów zasobów jest sztuką — dla każdego rodzaju obciążenia potrzebne będą inne wartości. Niektóre kontenery mogą być bezczynne przez większość czasu, jedynie od czasu do czasu będą zwiększały zużycie zasobów w celu obsługi żądania. Inne mogą być ciągle zajęte i stopniowo będą wykorzystywały coraz więcej pamięci, aż osiągną swoje limity.

Ogólnie rzecz biorąc, należy ustawić limity zasobów dla kontenera nieco powyżej maksymalnego wykorzystania przy normalnym działaniu. Jeśli np. wykorzystanie pamięci danego kontenera przez kilka dni nigdy nie przekracza 500 MiB pamięci, możesz ustawić limit pamięci na 600 MiB.



Czy kontenery powinny mieć w ogóle ograniczenia? Jedna ze szkół mówi, że kontenery produkcyjne *nie powinny mieć żadnych limitów* lub że granice powinny być ustawione tak wysoko, żeby kontenery nigdy ich nie przekroczyły. Dla bardzo dużych i zasobnych kontenerów, których ponowne uruchomienie jest drogie, może to mieć sens, ale — naszym zdaniem — lepiej ustawić limity. Bez ustawionych limitów kontener z wyciekiem pamięci lub zużywający zbyt dużo CPU może pochłonąć wszystkie zasoby dostępne w węźle, uniemożliwiając pracę innych kontenerów (zostaje obiektem Pac-man — nazwa nawiązuje do gry, w której użytkownik musi zjeść wszystkie kulkki, aby przejść do następnego poziomu).

Aby uniknąć powyższego scenariusza, ustaw limity kontenera na nieco ponad 100% normalnego użytkowania. Zapewni to, że nie zostanie unieruchomiony tak długo, jak działa poprawnie, oraz minimalizuje siłę oddziaływania, jeśli coś pojedzie nie tak.

Ustawienia żądań są mniej krytyczne niż limity, ale nadal nie powinny być ustawione zbyt wysoko (ponieważ w takim wypadku Pod nigdy nie zostanie uruchomiony) ani zbyt nisko (ponieważ Pody przekraczające ich żądania są w pierwszej kolejności eksmitowane).

Vertical Pod Autoscaler

Istnieje dodatek Kubernetes o nazwie *Vertical Pod Autoscaler* (<https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler>), który może pomóc w ustaleniu idealnych wartości dla żądań zasobów. Podczas obserwacji danego obiektu Deployment narzędzie automatycznie dostosuje żądania zasobów dla Podów — w oparciu o to, czego faktycznie używają. Posiada tryb pracy próbnej (ang. *dry-run*), podczas której pokaże tylko sugestie zmian, ale bez faktycznej modyfikacji.

Optymalizacja węzłów

Kubernetes może pracować z szeroką gamą rozmiarów węzłów, ale niektóre będą działać lepiej niż inne. Aby uzyskać najlepszą wydajność klastra w stosunku do zainwestowanych finansów, musisz obserwować, jak Twoje węzły radzą sobie w praktyce — w warunkach rzeczywistego ruchu, przy określonych obciążeniach. Pomoże to określić najbardziej opłacalne rozwiązania.

Warto pamiętać, że każdy węzeł musi mieć system operacyjny, który zużywa zasoby dysku, pamięci i procesora. Podobne zasoby zużywają także komponenty systemu Kubernetes i środowisko wykonawcze kontenera. Im mniejszy węzeł, tym jest to bardziej zauważalne.

Większe węzły mogą więc być bardziej opłacalne, ponieważ większa część ich zasobów jest dostępna dla różnych zadań. Kompromis polega na tym, że utrata pojedynczego węzła ma większy wpływ na dostępną pojemność klastra.

Małe węzły mają również wyższy procent *zasobów osieroconych* (ang. *stranded resources*); są to fragmenty pamięci i czasu procesora, które są nieużywane, ale zbyt małe, aby mógł je zająć każdy istniejący Pod.

Dobrą zasadą (<https://medium.com/@dyachuk/why-do-kubernetes-clusters-inaws-cost-more-than-they-should-fa510c1964c6>) jest to, że węzły powinny być na tyle duże, aby uruchomić przynajmniej pięć typowych Podów, utrzymując proporcję zasobów osieroconych na poziomie około 10% lub mniej. Jeśli węzeł może uruchomić 10 lub więcej Podów, osierocone zasoby będą poniżej 5%.

Domyślnym limitem w Kubernetes jest 110 Podów na węzeł. Chociaż możesz zwiększyć ten limit, dostosowując ustawienie `--max-pod` narzędzia kubelet, może to nie być możliwe w przypadku niektórych usług zarządzanych i dobrym pomysłem jest trzymanie się wartości domyślnych Kubernetes — chyba że istnieje uzasadniony powód, aby je zmienić.

Limit liczby Podów na węzeł oznacza, że możesz nie być w stanie skorzystać z największej wielkości instancji dostawcy usług chmurowych. Zamiast tego, aby uzyskać lepsze wykorzystanie, rozważ uruchomienie większej liczby mniejszych węzłów (<https://medium.com/@brendanrius/scaling-kubernetes-for-25m-users-a7937e3536a0>), zamiast np. 6 węzłów z 8 vCPU uruchom 12 węzłów z 4 vCPU.



Korzystając z pulpitu nawigacyjnego dostawcy chmury lub polecenia `kubectl top nodes`, zobacz procent wykorzystania zasobów w każdym węźle. Im większy procent wykorzystanego procesora, tym lepiej. Jeśli większe węzły w klastrze mają lepsze wykorzystanie, może być wskazane usunięcie niektórych mniejszych węzłów i zastąpienie ich większymi.

Z drugiej strony, jeśli większe węzły mają niskie wykorzystanie, pojemność klastra może być przeszacowana i dlatego możesz albo usunąć niektóre węzły, albo je zmniejszyć, obniżając całkowity rachunek.



Najlepsze praktyki

Większe węzły wydają się być bardziej opłacalne, ponieważ w tym przypadku mniej zasobów zużywają operacje systemowe. Ocenij swoje węzły, patrząc na rzeczywiste dane dotyczące wykorzystania klastra, i dąż do uzyskania od 10 do 100 Podów na węzeł.

Optymalizacja przestrzeni dyskowej

Jednym z często pomijanych kosztów chmurowych jest miejsce na dysku. Dostawcy usług w chmurze oferują różne ilości przestrzeni dyskowej dla każdego rozmiaru instancji (tu cena także się różni).

Chociaż możliwe jest osiągnięcie dość wysokiego wykorzystania procesora i pamięci za pomocą żądań i limitów zasobów Kubernetes, to samo nie dotyczy miejsca na dysku. Wiele węzłów klastra dysponuje nadmierną ilością przestrzeni dyskowej.

Wiele węzłów ma nie tylko więcej miejsca niż potrzeba, ale także zbyt wysoką klasę dysku. Większość dostawców chmury oferuje różne klasy dysków w zależności od liczby przydzielonych operacji we/wy na sekundę (IOPS) lub przepustowości.

Przykładowo bazy danych, które używają trwałych woluminów dyskowych, często wymagają bardzo wysokiego stopnia IOPS, aby uzyskać szybki dostęp do magazynu o dużej przepustowości. To drogie rozwiązanie. Możesz zaoszczędzić na kosztach chmury, zapewniając pamięć o niskim IOPS dla obciążień, które nie potrzebują tak dużej przepustowości. Z drugiej strony, jeśli Twоя aplikacja działa słabo, ponieważ spędza dużo czasu na czekaniu na operacje we/wy pamięci, będziesz chciał zapewnić lepsze rozwiązanie.

Zazwyczaj pulpit rozwiązania chmurowego pokazuje, ile IOPS faktycznie jest używanych w Twoich węzłach. Możesz skorzystać z tych wartości, aby zadecydować, jak obniżyć koszty.

Najlepiej byłoby ustawić żądania zasobów dla kontenerów, które wymagają dużej przepustowości lub dużej ilości pamięci. Jednak Kubernetes obecnie tego nie obsługuje, chociaż w przyszłości może zostać dodana obsługa żądań IOPS.



Najlepsze praktyki

Nie używaj rodzajów instancji z większą ilością miejsca, niż potrzebujesz. Zapewnij najmniejsze woluminy dysków o najniższym IOPS, jakie możesz — na podstawie faktycznie używanej przepustowości i miejsca.

Czyszczenie nieużywanych zasobów

W miarę wzrostu liczby klastrów Kubernetes tworzy się wiele niewykorzystanych lub utraconych zasobów. Z czasem, jeśli utracone zasoby nie zostaną usunięte, zaczną stanowić znaczną część ogólnych kosztów.

Na najwyższym poziomie możesz znaleźć instancje chmury, które nie są częścią żadnego krastra; łatwo zapomnieć o zamknięciu maszyny, gdy nie jest już używana.

Inne rodzaje zasobów w chmurze, takie jak usługi równoważenia obciążenia, publiczne adresy IP i woluminy dyskowe, również kosztują, nawet jeśli nie są używane. Należy regularnie sprawdzać wykorzystanie każdego rodzaju zasobów, aby znaleźć i usunąć nieużywane instancje.

W krastrze Kubernetes mogą znajdować się również obiekty Deployment i Pody, do których nie odwołuje się żaden Serwis — więc nie obsługują ruchu.

Nawet obrazy kontenerów, które nie są uruchomione, zajmują miejsce na dysku w Twoich węzłach. Na szczęście Kubernetes automatycznie usunie nieużywane obrazy, gdy w węźle zacznie brakować miejsca na dysku².

² Możesz dostosować to zachowanie, dobierając odpowiednie ustawienia czyszczenia pamięci kubelet (<https://kubernetes.io/docs/concepts/cluster-administration/kubelet-garbage-collection/>).

Korzystanie z metadanych właściciela

Jednym z pomocnych sposobów na zminimalizowanie nieużywanych zasobów jest ustanowienie w całej organizacji zasady, że każdy zasób musi być oznaczony informacją o jego właścielcu. W tym celu można skorzystać z adnotacji Kubernetes (patrz „Etykiety i adnotacje” w rozdziale 9.).

Każdy Deployment może być opisany np. w taki sposób:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-brilliant-app
  annotations:
    example.com/owner: "Customer Apps Team"
...
...
```

Metadane właściciela powinny określać osobę lub zespół, z którymi należy się skontaktować w sprawie tego zasobu. Jest to pozyteczne i przydatne szczególnie do identyfikowania porzuconych lub nieużywanych zasobów. (Pamiętaj, że dobrym pomysłem jest poprzedzanie niestandardowych adnotacji nazwą domeny Twojej firmy, np. *Example.com*, aby uniknąć kolizji z innymi adnotacjami, które mogą mieć tę samą nazwę).

Możesz regularnie wysyłać zapytania do klastra o wszystkie zasoby, które nie mają adnotacji właściciela, i sporządzić ich listę w celu ewentualnego usunięcia. Szczególnie surowa polityka może natychmiast zakończyć wszystkie nieznane zasoby. Nie bądź jednak zbyt surowy, szczególnie na początku: wartość firmy programistycznej jest tak samo ważnym zasobem jak pojemność klastra, jeśli nie bardziej.



Najlepsze praktyki

Ustaw adnotacje właściciela na wszystkich swoich zasobach, podając informacje o tym, z kim się skontaktować, jeśli wystąpi problem z tym zasobem lub jeśli wydaje się on porzucony i podlega usunięciu.

Znajdowanie niewykorzystanych zasobów

Niektóre zasoby mogą odbierać bardzo niski poziom ruchu lub w ogóle go nie mieć. Być może zostały odłączone od interfejsu Serwisu z powodu zmiany etykiety, a być może były to zasoby tymczasowe lub eksperymentalne.

Każdy Pod powinien przedstawić liczbę otrzymanych żądań jako metrykę (więcej informacji na ten temat znajduje się w rozdziale 16.). Skorzystaj z tych danych, aby znaleźć Pody o niskim lub zero-wym natężeniu ruchu i utworzyć listę zasobów, które potencjalnie mogą zostać zakończone.

Możesz także sprawdzić dane dotyczące wykorzystania procesora i pamięci dla każdego Poda w konsole internetowej i znaleźć Pody, które są najmniej używane w klastrze. Pody, które nic nie robią, prawdopodobnie nie są dobrymi zasobami.

Jeśli Pody mają metadane właściciela, skontaktuj się z ich właścicielami, aby dowiedzieć się, czy są one rzeczywiście potrzebne (mogą być np. przeznaczone do aplikacji będącej w fazie rozwoju).

Możesz użyć innej niestandardowej adnotacji Kubernetes (np. `example.com/lowtraffic`), aby zidentyfikować Pody, które nie otrzymują żądań, ale nadal są potrzebne z tego czy innego powodu.



Najlepsze praktyki

Regularnie przeglądaj kластer, aby znaleźć niewykorzystane lub porzucone zasoby w celu ich eliminacji. Adnotacje o właściwym mogą wiele pomóc.

Czyszczenie zakończonych obiektów Job

Obiekty Job Kubernetes (patrz „Kontroler Job” w rozdziale 9.) to Pody, które uruchamiają się jeden raz i nie są ponownie uruchamiane. Jednak obiekty Job nadal istnieją w bazie danych Kubernetes, a gdy pojawi się znaczna liczba zakończonych zadań, może to mieć wpływ na wydajność interfejsu API. Przydatnym narzędziem do czyszczenia zakończonych kontrolerów Job jest narzędzie *kube-job-cleaner* (<https://github.com/hjacobs/kube-job-cleaner>).

Sprawdzanie wolnej pojemności

Aby obsłużyć awarię pojedynczego węzła worker, w klastrze zawsze powinna być wystarczająca wolna pojemność. Żeby to sprawdzić, spróbuj opróżnić swój największy węzeł (patrz „Skalowanie w dół” w rozdziale 6.). Gdy wszystkie Pody zostaną eksmitowane z węzła, sprawdź, czy wszystkie aplikacje nadal mają stan „forking” ze skonfigurowaną liczbą replik. Jeśli tak nie jest, musisz zmniejszyć pojemność klastra.

Jeśli w przypadku awarii węzła nie ma miejsca na przeplanowanie obciążen, usługi mogą zostać w najlepszym wypadku zdegradowane, a w najgorszym niedostępne.

Korzystanie z instancji zastrzeżonych

Niektórzy dostawcy usług w chmurze oferują różne klasy instancji w zależności od cyklu życia maszyny. Instancje *zastrzeżone* (ang. *reserved*) oferują kompromis między ceną i elastycznością.

Przykładowo instancje zastrzeżone AWS są o połowę tańsze niż instancje na żądanie (ang. *on-demand*; typ domyślny). Możesz zarezerwować instancje na różne okresy: rok, trzy lata itd. Instancje zastrzeżone AWS mają ustalony rozmiar, więc jeśli okaże się, że za trzy miesiące potrzebujesz większej instancji, Twoja rezerwacja zostanie w większości zmarnowana.

Ekwivalentem rozwiązania Google Cloud dla instancji zastrzeżonych jest *Committed Use Discounts*, czyli rabaty za zarezerwowanie zasobów. Umożliwiają opłacenie z góry określonej liczby vCPU i pewnej ilości pamięci. Jest to bardziej elastyczne rozwiązanie niż instancje zastrzeżone AWS, ponieważ możesz zużywać więcej zasobów niż zarezerwowałeś; po prostu płacisz normalną cenę na żądanie za wszystko, co nie jest objęte rezerwacją.

Oba powyższe rozwiązania mogą być dobrym wyborem, jeśli znasz swoje wymagania w dającej się przewidzieć przyszłości. Nie ma jednak zwrotu kosztów za rezerwacje, z których nie skorzystasz, musisz zapłacić z góry za cały okres rezerwacji. Dlatego powinieneś rezerwować instancje tylko na okres, w którym Twoje wymagania raczej się nie zmienią.

Jeśli jednak możesz zaplanować rok lub dwa, skorzystanie z instancji zastrzeżonych może przynieść znaczne oszczędności.



Najlepsze praktyki

Używaj instancji zastrzeżonych, gdy Twoje potrzeby prawdopodobnie nie zmienią się przez rok lub dwa — mądrze wybieraj takie rezerwacje, ponieważ nie można ich zmienić ani zwrócić, gdy zostaną wykupione.

Korzystanie z instancji w trybie wywłaszczeniowym

Instancje *spot*, jak je nazywa AWS, lub maszyny *wirtualne pracujące w trybie wywłaszczeniowym* (ang. *preemptible Vms*) w terminologii Google, nie zapewniają gwarancji dostępności i często mają ograniczony okres użytkowania. Stanowią zatem kompromis między ceną a dostępnością.

Instancja spot jest tania, ale może zostać w dowolnym momencie wstrzymana lub wznowiona, a także może zostać całkowicie zakończona. Na szczęście Kubernetes został zaprojektowany w celu zapewnienia usług wysokiej niezawodności, pomimo utraty poszczególnych węzłów klastra.

Zmienna cena lub zmienne wywłaszczenie

Instancje spot mogą być zatem opłacalnym wyborem dla klastra. W instancji spot AWS cena za godzinę różni się w zależności od zapotrzebowania. Gdy jest ono wysokie dla danego typu instancji w określonym regionie i strefie dostępności, cena wzrośnie.

Z drugiej strony, rozwiązywanie oferowane przez Google Cloud rozliczane jest według stałej stawki, ale z różnym wywłaszczeniem. Google twierdzi, że średnio od 5 do 15% Twoich węzłów może zostać wstrzymanych w danym tygodniu (<https://cloud.google.com/compute/docs/instances/preemptible>). Maszyny wirtualne pracujące w trybie wywłaszczenia mogą być nawet o 80% tańsze niż na żądanie, w zależności od typu instancji.

Węzły z wywłaszczeniem mogą zmniejszyć o połowę Twoje koszty

Korzystanie z węzłów z wywłaszczeniem dla klastra Kubernetes może być więc bardzo skutecznym sposobem na obniżenie kosztów. Chociaż może być konieczne uruchomienie kilku dodatkowych węzłów, aby upewnić się, że obciążenia będą w stanie przetrwać chwilowe zawieszenie, uważa się, że możliwe jest osiągnięcie całkowitego zmniejszenia kosztów na węzeł o 50%.

Może się również okazać, że użycie węzłów z wywłaszczeniem jest dobrym sposobem na wbudowanie w klaster odrabiny inżynierii chaosu (patrz „Testowanie chaosu” w rozdziale 6.) — pod warunkiem że aplikacja jest gotowa do testowania chaosu.

Pamiętaj jednak, że zawsze powinieneś mieć wystarczającą liczbę węzłów pracujących bez wywłaszczenia, aby obsłużyć minimalne obciążenie klastra. Nigdy nie stawiaj więcej, niż możesz stracić. Jeśli masz wiele węzłów pracujących w trybie wywłaszczenia, dobrym pomysłem może być skorzystanie z automatycznego skalowania klastra, aby upewnić się, że wszystkie zawieszone węzły zostaną jak najszybciej wymienione (patrz „Automatyczne skalowanie” w rozdziale 6.).

Teoretycznie *wszystkie* możliwe węzły *preemptible* mogą zniknąć w tym samym czasie. Dlatego pomimo oszczędności kosztów dobrym pomysłem jest ograniczenie liczby takich węzłów do nie więcej niż — powiedzmy — 2/3 klastra.



Najlepsze praktyki

Zmniejsz koszty, wykorzystując instancje spot lub preemptible dla niektórych swoich węzłów, zachowując optymalny poziom ewentualnych strat. Zawsze utrzymuj także węzły bez wywłaszczenia.

Używanie koligacji węzłów do kontroli uruchomień

Można skorzystać z *koligacji węzłów* (ang. *node affinities*) Kubernetes, aby upewnić się, że Pody, które nie tolerują awarii, nie będą uruchamiane w węzłach preemptible (<https://medium.com/google-cloud/using-preemptible-vms-to-cut-kubernetes-engine-bills-in-half-de2481b8e814>) (patrz „Koligacje węzłów” w rozdziale 9.).

Przykładowo węzły preemptible Google Kubernetes Engine posiadają etykietę `cloud.google.com/gke-preemptible`. Aby przekazać do Kubernetes informację, aby nigdy nie uruchamiał Poda na jednym z tych węzłów, dodaj następujące polecenie do specyfikacji Poda lub obiektu Deployment:

```
affinity:  
  nodeAffinity:  
    requiredDuringSchedulingIgnoredDuringExecution:  
      nodeSelectorTerms:  
        - matchExpressions:  
          - key: cloud.google.com/gke-preemptible  
            operator: DoesNotExist
```

Koligacja `requiredDuringScheduling` ... jest obowiązkowa: Pody z tą koligacją nigdy nie zostaną uruchomione w węźle niepasującym do wyrażenia selektora (*twarda koligačia* — ang. *hard affinity*).

Ewentualnie możesz powiedzieć Kubernetesowi, że niektóre z mniej krytycznych Podów mogą tolerować okazjonalne awarie oraz mogą być uruchamiane na węzłach preemptible. W takim przypadku możesz użyć *miękkiej koligacji* (ang. *soft affinity*):

```
affinity:  
  nodeAffinity:  
    preferredDuringSchedulingIgnoredDuringExecution:  
      - preference:  
          matchExpressions:  
            - key: cloud.google.com/gke-preemptible  
              operator: Exists  
        weight: 100
```

To oznacza: „Jeśli możesz, uruchom ten Pod w węźle preemptible; jeśli nie możesz, nie ma to znaczenia”.



Najlepsze praktyki

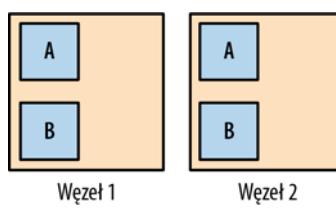
Jeśli korzystasz z węzłów preemptible, skorzystaj z koligacji węzłów Kubernetes, aby upewnić się, że krytyczne obciążenia nie zostaną zawieszone.

Utrzymywanie równowagi obciążen

Rozmawialiśmy o pracy, którą wykonuje scheduler Kubernetes, upewniając się, że obciążenia są sprawiedliwie rozmiieszczone na jak największej liczbie węzłów oraz że repliki kapsuł są ulokowane w różnych węzłach w celu zapewnienia wysokiej niezawodności.

Ogólnie rzecz biorąc, scheduler wykonuje świetną robotę, ale są pewne przypadki graniczne, na które trzeba uważać.

Załóżmy, że masz dwa węzły i dwa serwisy, A i B, każdy z dwiema replikami. W zbalansowanym klastrze będzie jedna replika serwisu A na każdym węźle i jeden serwis B na każdym węźle (patrz rysunek 5.1). Jeśli jeden węzeł ulegnie awarii, zarówno A, jak i B będą nadal dostępne.

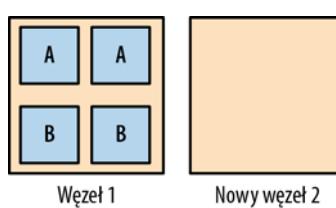


Rysunek 5.1. Usługi A i B są zrównoważone we wszystkich dostępnych węzłach

Na razie w porządku. Założymy jednak, że węzeł 2. nie działa. Scheduler zauważa, że zarówno A, jak i B potrzebują dodatkowej repliki, ale jest tylko jeden węzeł. Teraz w węźle 1. uruchomiono dwie repliki usługi A i dwie usługi B.

Załóżmy teraz, że uruchamiamy nowy węzeł, aby zastąpić uszkodzony węzeł 2. Jeśli nawet będzie dostępny, nie będzie na nim żadnych Podów. Scheduler nigdy nie przenosi działających Podów z jednego węzła do drugiego.

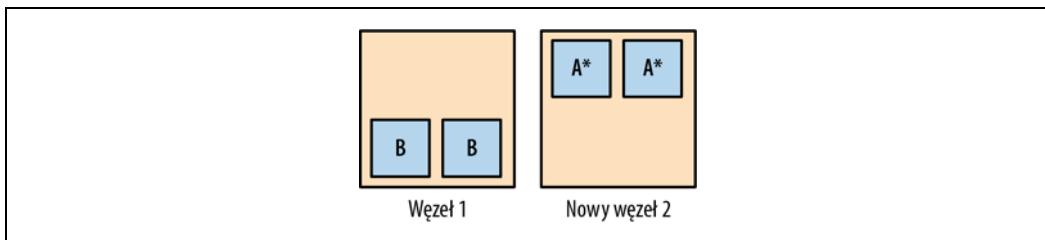
Mamy teraz niezrównoważony kластer (<https://itnext.io/keep-you-kubernetes-clusterbalanced-the-secret-to-high-availability-17edf60d9cb7>), w którym wszystkie Pody znajdują się w węźle 1., a żadnego nie ma w 2. węźle (patrz rysunek 5.2).



Rysunek 5.2. Gdy węzeł 2. zostaje uszkodzony, wszystkie repliki zostaną przeniesione na 1. węzeł

Jednak robi się coraz gorzej. Założymy, że wdrażasz aktualizację „na bieżąco” (ang. *rolling*) do serwisu A (nazwijmy nową wersję serwisu A*). Scheduler musi uruchomić dwie nowe repliki dla serwisu A*, poczekać, aż się pojawią, a następnie zakończyć stare. Gdzie powstaną nowe repliki? W nowym

węźle 2., ponieważ jest bezczynny, podczas gdy w węźle 1. są już uruchomione cztery Pody. Tak więc dwie nowe repliki serwisu A* są uruchamiane w węźle 2., a stare usuwane z 1. węzła (patrz rysunek 5.3).



Rysunek 5.3. Sytuacja po aktualizacji serwisu A, klastry ciągle nie są zrównoważone

Teraz znajdujesz się w złej sytuacji, ponieważ obie repliki serwisu B znajdują się w tym samym węźle (węzeł 1.), podczas gdy obie repliki serwisu A* znajdują się również w tym samym węźle (węzeł 2.). Chociaż masz dwa węzły, nie ma zapewnionej wysokiej niezawodności. Awaria węzła 1. lub węzła 2. spowoduje awarię serwisu.

Powodem tego problemu jest fakt, że scheduler nigdy nie przenosi Podów z jednego węzła do drugiego, chyba że z jakiegoś powodu zostaną one ponownie uruchomione. Ponadto cel komponentu scheduler, polegający na równomiernym rozmieszczeniu obciążenia między węzłami, czasami stoi w sprzeczności z utrzymaniem wysokiej niezawodności dla poszczególnych serwisów.

Jednym ze sposobów rozwiązyania tego problemu jest skorzystanie z narzędzia o nazwie Descheduler (<https://github.com/kubernetes-incubator/descheduler>). Możesz uruchamiać to narzędzie co jakiś czas, w postaci obiektu Job. Dokoła ono wszelkich starań, aby przywrócić równowagę klastra, znajdując Pody, które należy przenieść, a niepotrzebne usunąć.

Descheduler wyposażony jest w różne strategie i polityki, które możesz skonfigurować. Przykładowo jedna polityka szuka niewykorzystanych węzłów i usuwa Pody w innych węzłach, aby zmusić je do uruchomienia w bezczynnych węzłach.

Inna polityka szuka duplikatów Podów, w których dwie lub więcej replik tego samego Poda działają w tym samym węźle i eksmituje je. To rozwiązuje problem, który pojawił się w naszym przykładzie, w którym obciążenia były nominalnie zrównoważone, ale w rzeczywistości żaden Serwis nie był wysoce niezawodny.

Podsumowanie

Kubernetes radzi sobie z obciążeniami w niezawodny i wydajny sposób, bez faktycznej potrzeby ręcznej interwencji. Musisz tylko przekazać do komponentu scheduler dokładne szacunki zasobów kontenerów.

Dzięki temu czas poświęcony na rozwiązywanie problemów można znacznie lepiej wykorzystać — np. na tworzenie aplikacji. Dzięki, Kubernetes!

Zrozumienie, w jaki sposób Kubernetes zarządza zasobami, jest kluczem do prawidłowego budowania i uruchamiania klastra. Oto wiadomości do zapamietania z tego rozdziału.

- Kubernetes przydziela zasoby procesora i pamięci do kontenerów na podstawie żądań i limitów.
- Żądania kontenera to minimalna ilość zasobów niezbędna do uruchomienia, jego limity określają maksymalną dozwoloną ilość.
- Minimalne obrazy kontenerów są szybsze w budowie, wdrażaniu i uruchamianiu; im mniejszy kontener, tym mniej potencjalnych luk w zabezpieczeniach.
- Sondy żywotności informują Kubernetes, czy kontener działa poprawnie; jeśli sonda żywotności kontenera zawiedzie, kontener zostanie zrestartowany.
- Sondy gotowości informują Kubernetes, że kontener jest gotowy i może obsłużyć żądania; jeśli sonda gotowości zawiedzie, kontener zostanie usunięty z Serwisów, które się do niego odwołują, i odfłączony od ruchu użytkowników.
- Pod DisruptionBudgets pozwala ograniczyć liczbę Podów, które można zatrzymać jednocześnie podczas *eksmisji*, zachowując wysoką niezawodność Twojej aplikacji.
- Przestrzenie nazw to sposób logicznego podziału klastra na partie; możesz utworzyć przestrzeń nazw dla każdej aplikacji lub grupy powiązanych aplikacji.
- Aby odwołać się do usługi w innej przestrzeni nazw, możesz użyć adresu DNS, takiego jak SERVICE.NAMESPACE.
- ResourceQuota pozwala ustawić ogólne limity zasobów dla danej przestrzeni nazw.
- LimitRanges określa domyślne żądania zasobów i limity dla kontenerów w przestrzeni nazw.
- Limity zasobów należy ustawać tak, aby były wystarczające dla aplikacji, ale żeby nie były przekraczone w normalnym użytkowaniu.
- Nie należy przydzielać w chmurze więcej miejsca niż potrzeba oraz dużej przepustowości, chyba że ma to decydujące znaczenie dla wydajności Twojej aplikacji.
- Ustaw adnotacje właściciela na wszystkich swoich zasobach i regularnie skanuj klaster w poszukiwaniu zasobów nieposiadających przypisanego właściciela.
- Znajdź i wyczyść zasoby, które nie są używane (ale najpierw skontaktuj się z ich właścicielami).
- Instancje zastrzeżone mogą zaoszczędzić pieniądze, jeśli możesz zrobić plan w perspektywie długoterminowej.
- Instancje preemptible mogą zaoszczędzić pieniądze, ale bądź przygotowany na ich zniknięcie w krótkim okresie czasu; skorzystaj z koligacji węzłów, aby trzymać wrażliwe na awarie Pody z dala od węzłów preemptible.

Operacje na klastrach

Jeśli Tetris mnie czegoś nauczył, to tego, że błędy narastają, a osiągnięcia znikają.

— Andrew Clay Shafer

Jeśli masz klaster Kubernetes, skąd wiesz, że jest w dobrej formie i działa poprawnie? Jak go skalować, aby poradzić sobie z chwilowym wzrostem ruchu, ale ograniczyć koszty chmury do minimum? W tym rozdziale przyjrzymy się problemom związanym z obsługą klastrów Kubernetes w obciążeniach produkcyjnych oraz niektórym narzędziom, które mogą Ci w tym pomóc.

Jak pisaliśmy w rozdziale 3., jest wiele ważnych rzeczy, które należy wziąć pod uwagę przy obsłudze klastra Kubernetes; są to dostępność, uwierzytelnianie, aktualizacje itd. Jeśli korzystasz z dobrze zarządzanej usługi Kubernetes zgodnie z zaleceniami, większość z tych problemów powinna zostać rozwiązana.

Jednak to, co faktycznie zrobisz z klastrem, zależy od Ciebie. W tym rozdziale dowiesz się, jak dobrać rozmiar klastra oraz jak go skalować, jak znajdować problemy związane z bezpieczeństwem oraz testować odporność infrastruktury za pomocą narzędzia Chaos Monkey.

Rozmiar i skalowanie klastra

Jak duży musi być Twój klaster? Z wykorzystaniem samodzielnie hostowanych klastrów Kubernetes i prawie wszystkich usług zarządzanych bieżący koszt klastra zależy bezpośrednio od liczby i wielkości jego węzłów. Jeśli pojemność klastra jest zbyt mała, obciążenia nie będą działać poprawnie lub ulegną awarii przy dużym natężeniu ruchu. Jeśli pojemność jest zbyt duża, marnujesz pieniądze.

Właściwe dobranie rozmiaru i skalowanie klastra jest bardzo ważne, więc spójrzmy na niektóre problemy.

Planowanie pojemności

Jednym ze sposobów wstępnego oszacowania potrzebnej pojemności jest zastanowienie się, ile tradycyjnych serwerów potrzebujesz do uruchomienia tych samych aplikacji. Jeśli np. Twoja obecna architektura działa na 10 instancjach chmurowych, prawdopodobnie nie będziesz potrzebował więcej niż 10 węzłów w klastrze Kubernetes, aby uruchomić to samo obciążenie. W rzeczywistości

prawdopodobnie nie będziesz potrzebować aż tylu. Kubernetes może osiągnąć większe wykorzystanie niż w przypadku tradycyjnych serwerów, ponieważ może zrównoważyć pracę na różnych maszynach. Jednak taka optymalizacja może zająć trochę czasu i praktycznego doświadczenia.

Najmniejszy klaster

Gdy konfigurujesz klaster po raz pierwszy, prawdopodobnie będziesz go używać do zabawy, eksperymentów oraz aby zastanowić się, jak uruchomić aplikację. Więc prawdopodobnie na początku nie musisz marnować pieniędzy na duży klaster, bo nie masz pojęcia, jakiej pojemności będziesz potrzebować.

Najmniejszy możliwy klaster Kubernetes to pojedynczy węzeł. Pozwoli Ci wypróbować Kubernetes i uruchomić małe obciążenia dla developmentu (tzn. rozwoju oprogramowania), tak jak widzieliśmy w rozdziale 2. Jednak klaster z jednym węzłem nie jest odporny na awarie sprzętu węzła, serwera API Kubernetes lub *kubeleta* (demona agenta odpowiedzialnego za uruchamianie obciążeń na każdym węźle).

Jeśli korzystasz z zarządzanej usługi Kubernetes, takiej jak GKE (patrz „Google Kubernetes Engine (GKE)” w rozdziale 3.), nie musisz martwić się o obsługę węzłów master: jest to robione za Ciebie. Jeśli natomiast budujesz własny klaster, musisz zdecydować, z ilu węzłów master chcesz korzystać.

Minimalna liczba węzłów głównych potrzebnych do zapewnienia odpornego klastra Kubernetes wynosi trzy. Jeden nie byłby odporny, a dwa mogłyby nie zdecydować, który jest liderem — więc potrzebne są trzy węzły master.

Chociaż możesz wykonać pracę w tak małym klastrze Kubernetes, nie jest to zalecane. Lepszym pomysłem jest dodanie niektórych węzłów worker, aby Twoje własne obciążenia nie konkurowały o zasoby z warstwą sterowania Kubernetes.

Pod warunkiem, że Twoja warstwa sterowania klastrami jest wysoce niezawodna, *możesz sobie poradzić* z pojedynczym węzłem worker. Jednak dwa są rozsądnym minimum, chroniącym przed awarią węzła i umożliwiającym Kubernetes uruchomienie co najmniej dwóch replik każdego Poda. Im więcej węzłów, tym lepiej, zwłaszcza że scheduler Kubernetes nie zawsze może zapewnić, iż obciążenia są w pełni zrównoważone w dostępnych węzłach (patrz „Utrzymywanie równowagi obciążzeń” w rozdziale 5.).



Najlepsze praktyki

W celu zapewnienia wysokiej niezawodności klastry Kubernetes wymagają co najmniej trzech węzłów master — być może potrzeba będzie więcej do obsługi pracy większych klastrów. Dwa węzły worker to minimum wymagane do tego, aby obciążenia były odporne na awarie pojedynczego węzła, a lepiej posiadać trzy takie węzły.

Największy klaster

Czy istnieje limit wielkości dla klastrów Kubernetes? Tak, ale prawie na pewno nie musisz się o to martwić; Kubernetes w wersji 1.12 oficjalnie obsługuje klastry do 5000 węzłów.

Ponieważ operacja dzielenia na klastry wymaga komunikacji między węzłami, liczba możliwych ścieżek komunikacji i skumulowane obciążenie wewnętrznej bazy danych rośnie wykładniczo wraz z rozmiarem klastra. Chociaż Kubernetes *może* nadal działać z więcej niż 5000 węzłów, nie jest to zagwarantowane — może przestać wystarczająco reagować, aby poradzić sobie z obciążeniami produkcyjnymi.

Dokumentacja Kubernetes informuje, że obsługiwane konfiguracje klastra (<https://kubernetes.io/docs/setup/cluster-large>) muszą mieć nie więcej niż 5000 węzłów, nie więcej niż 150 000 Podów, nie więcej niż 300 000 kontenerów i nie więcej niż 100 Podów na węzeł. Warto pamiętać, że im większy kластер, tym większe obciążenie węzłów master; jeśli jesteś odpowiedzialny za swoje własne węzły master, muszą one być dość potężnymi maszynami, aby poradzić sobie z klastrem o tysiącach węzłów.



Najlepsze praktyki

Aby zapewnić maksymalną niezawodność, w klastrze Kubernetes powinno być mniej niż 5000 węzłów i 150 000 Podów (nie jest to problem dla większości użytkowników). Jeśli potrzebujesz więcej zasobów, uruchom wiele klastrów.

Klastry sfederowane

Jeśli masz bardzo wymagające obciążenia lub musisz działać na dużą skalę, limity te mogą stać się dla Ciebie praktycznym problemem. W takim przypadku możesz uruchomić wiele klastrów Kubernetes i jeśli to konieczne, *federować je* (ang. *federate*), tak aby obciążenia mogły być replikowane między klastrami.

Federacja umożliwia synchronizację dwóch lub więcej klastrów, z uruchamionym identycznym obciążeniem. Może to być przydatne, jeśli potrzebujesz klastrów Kubernetes od różnych dostawców chmur, w celu zapewnienia odporności, lub z różnych lokalizacji geograficznych do zmniejszenia opóźnienia użytkowników. Grupa klastrów sfederowanych może nadal działać nawet wtedy, kiedy pojedynczy kластer ulegnie awarii.

Więcej informacji na temat federacji klastrów można znaleźć w dokumentacji Kubernetes (<https://v1-16.docs.kubernetes.io/docs/concepts/cluster-administration/federation/>).

Dla większości użytkowników Kubernetes federacja nie jest czymś, czym powinni się zajmować, i w rzeczywistości większość użytkowników na bardzo dużą skalę jest w stanie obsłużyć swoje obciążenia za pomocą wielu niesfederowanych klastrów (od kilkuset do kilku tysięcy węzłów).



Najlepsze praktyki

Jeśli chcesz zreplikować obciążenia w wielu klastrach, być może ze względu na opóźnienia wynikające z różnych lokalizacji geograficznych, wykorzystaj federacje. Jednak większość użytkowników nie musi tego stosować.

Czy potrzebuję wielu klastrów?

O ile nie działasz na bardzo dużą skalę, jak wspomnieliśmy w poprzednim podpunkcie, prawdopodobnie nie potrzebujesz więcej niż jednego lub dwóch klastrów: może jednego do produkcji, a drugiego do rozwoju programowania i testowania.

Dla wygody i łatwości zarządzania zasobami kластer można podzielić na logiczne partycje za pomocą *przestrzeni nazw*, które omówiliśmy bardziej szczegółowo w rozdziale 5. (patrz „Korzystanie z przestrzeni nazw”). Zarządzanie wieloma klastrami (z kilkoma wyjątkami) zwykle nie jest warte wydatków administracyjnych.

Istnieją pewne szczególne sytuacje, takie jak bezpieczeństwo i zgodność z przepisami, w których możesz upewniać się, że usługi działające w jednym klastrze są absolutnie odizolowane od usług w innym (gdy np. mamy do czynienia z chronionymi informacjami zdrowotnymi lub danych nie można przesyłać z jednego położenia geograficznego do drugiego ze względów prawnych). W takich przypadkach musisz utworzyć osobne klastry. Dla większości użytkowników Kubernetes nie będzie to problemem.



Najlepsze praktyki

Użyj pojedynczego klastra produkcyjnego i pojedynczego klastra do rozwoju oprogramowania. Chyba że naprawdę potrzebujesz pełnej izolacji jednego zestawu obciążen lub zespołów od drugiego. Jeśli chcesz po prostu podzielić kластer na partie, aby ułatwić zarządzanie, skorzystaj z przestrzeni nazw.

Węzły i instancje

Im większa pojemność danego węzła, tym więcej pracy może on wykonać. Przy czym pojemność jest wyrażana w kategoriach liczby rdzeni procesora (wirtualnego lub innego), dostępnej pamięci oraz w mniejszym stopniu w kategorii miejsca na dysku. Czy jednak lepiej np. uruchamiać 10 bardzo dużych węzłów, czy 100 znacznie mniejszych?

Wybór odpowiedniego rozmiaru węzła

Dla klastrów Kubernetes nie ma uniwersalnie poprawnego rozmiaru węzła. Odpowiedź zależy od dostawcy chmury lub sprzętu oraz od konkretnych obciążień.

Koszt pojemności różnych rozmiarów instancji może mieć wpływ na sposób decydowania o wielkości węzłów. Przykładowo niektórzy dostawcy usług chmurowych mogą zaoferować niewielką zniżkę na większe rozmiary instancji, więc jeśli obciążenia są bardzo intensywne obliczeniowo, tańsze może być uruchomienie ich na kilku bardzo dużych węzłach zamiast na wielu mniejszych.

Wymagana w klastrze liczba węzłów wpływa również na wybór rozmiaru węzła. Aby uzyskać korzyści oferowane przez Kubernetes, takie jak replikacja Podów i wysoka niezawodność, musisz rozłożyć pracę na kilka węzłów. Jeśli jednak węzły mają zbyt dużo wolnej mocy, to strata pieniędzy.

Jeśli potrzebujesz — powiedzmy — co najmniej 10 węzłów do uzyskania wysokiej niezawodności, ale każdy węzeł musi uruchomić tylko kilka Podów, instancje węzła mogą być bardzo małe. Z drugiej strony, jeśli potrzebujesz tylko dwóch węzłów, możesz wybrać duże i potencjalnie zaoszczędzić pieniądze dzięki bardziej korzystnym cenom instancji.



Najlepsze praktyki

Użyj najbardziej opłacalnego typu węzła, który oferuje Twój dostawca. Często większe węzły są tańsze, ale jeśli masz tylko kilka węzłów, możesz dodać kilka mniejszych, aby pomóc w nadmiarowości.

Typy instancji chmurowych

Ponieważ same komponenty Kubernetes, takie jak kubelet, wykorzystują określona ilość zasobów, a do wykonania użytecznej pracy będziesz potrzebować wolnej mocy, najmniejsze rozmiary instancji oferowane przez dostawcę chmury prawdopodobnie nie będą odpowiednie dla Kubernetes.

Węzeł główny dla małych klastrów (do około pięciu węzłów) powinien mieć co najmniej jeden wirtualny procesor (vCPU) i 3 – 4 GiB pamięci, przy czym większe klastry wymagają więcej pamięci i procesorów dla każdego węzła master. Jest to odpowiednik instancji *n1-standard-1* w Google Cloud, *m3.medium* w AWS i Standard DS1 v2 na Azure.

Instancja pojedynczego procesora 4 GiB jest również rozsądny minimum dla węzła worker, chociaż — jak widzieliśmy — czasem może być bardziej opłacalny zakup większych węzłów. Przykładowo domyślny rozmiar węzła w Google Kubernetes Engine, który w przybliżeniu spełnia tę specyfikację, to *n1-standard-1*.

W przypadku większych klastrów, które mogą mieć kilkadziesiąt węzłów, sensowne może być zapewnienie dwóch lub trzech różnych rozmiarów instancji. Oznacza to, że Pody wykonujące intensywne obliczenia oraz wymagające dużej ilości pamięci mogą być uruchamiane przez Kubernetes na dużych węzłach, pozostawiając mniejszym węzłom swobodę obsługi mniejszych Podów (patrz „Koligacje węzłów” w rozdziale 9.). Daje to schedulerowi Kubernetes maksymalną swobodę wyboru przy podejmowaniu decyzji, gdzie uruchomić dany Pod.

Węzły heterogeniczne

Nie wszystkie węzły są sobie równe. Możesz potrzebować niektórych węzłów o specjalnych właściwościach, np. posiadających procesor graficzny (GPU). Procesory graficzne to wysokowydajne procesory równoległe, które są szeroko stosowane w przypadku problemów wymagających dużej mocy obliczeniowej — np. takich jak uczenie maszynowe lub analiza danych.

Możesz skorzystać z funkcjonalności, np. *limitów zasobów* (patrz „Limity zasobów” w rozdziale 5.), aby określić, że dany Pod wymaga przykładowo co najmniej jednego procesora graficznego. Zapewni to, że Pody będą działały tylko na węzłach obsługujących GPU oraz to, że będą miały priorytet w stosunku do innych Podów działających na innych węzłach.

Większość węzłów Kubernetes prawdopodobnie pracuje w Linuksie, co jest odpowiednie dla prawie wszystkich aplikacji. Pamiętaj, że kontenery *nie są maszynami wirtualnymi*, więc proces działający wewnętrz kontenera działa bezpośrednio na jądrze systemu operacyjnego tego węzła. Plik binarny systemu Windows nie będzie działał na linuksowym węźle Kubernetes, więc jeśli chcesz uruchomić kontenery z Windows, będziesz musiał uruchomić odpowiednie węzły.



Najlepsze praktyki

Większość kontenerów jest zbudowanych dla systemu Linux, więc prawdopodobnie będziesz chciał uruchamiać głównie węzły oparte na systemie Linux. Konieczne może być jednak dodanie jednego lub dwóch specjalnych typów węzłów dla określonych wymagań, np. zawierających GPU lub Windows.

Serwery bare-metal

Jedną z najbardziej przydatnych właściwości Kubernetes jest jego zdolność do łączenia wszelkiego rodzaju maszyn o różnych rozmiarach, architekturach i możliwościach w celu zapewnienia jednej, zunifikowanej maszyny logicznej, na której mogą działać różnego rodzaju obciążenia. Podczas gdy Kubernetes jest zwykle kojarzony z serwerami w chmurze, wiele organizacji ma dużą liczbę fizycznych maszyn bare-metal, które można potencjalnie wykorzystać w klastrach Kubernetes.

W rozdziale 1. pisaliśmy, że technologia chmury przekształca infrastrukturę *capex* (zakupy maszyny jako wydatek inwestycyjny) na infrastrukturę *opex* (leasing mocy obliczeniowej jako koszt operacyjny). Ma to sens finansowy, ale jeśli Twoja firma już posiada dużą liczbę serwerów bare-metal, nie musisz z nich jeszcze rezygnować: zamiast tego rozważ połączenie ich w klaster Kubernetes (patrz „Bare-metal i on-premise” w rozdziale 3.).



Najlepsze praktyki

Jeśli posiadasz własne serwery lub nie jesteś jeszcze gotowy do pełnej migracji do rozwiązania chmurowego, użyj Kubernetes, aby uruchomić obciążenia kontenerów na istniejących komputerach.

Skalowanie klastra

Czy po wybraniu rozsądniego rozmiaru początkowego dla klastra i odpowiedniej kombinacji rozmiarów instancji dla węzłów worker skończyłeś pracę? Niemal na pewno nie: z czasem konieczne może być zwiększenie lub zmniejszenie klastra, aby dopasować go do obsługi różnego rodzaju natężeń ruchu lub wymagań biznesowych.

Grupy instancji

Dodawanie węzłów do klastra Kubernetes jest łatwe. Jeśli prowadzisz klaster, który hostujesz, narzędzie do zarządzania klastrami, takie jak kops (patrz „kops” w rozdziale 3.), może to zrobić za Ciebie. Narzędzie kops obsługuje pojęcie *grupy instancji*, która jest zbiorem węzłów danego typu instancji (np. m3.medium). Usługi zarządzane, takie jak Google Kubernetes Engine, mają tę samą funkcję, zwianą *grupami węzłów* (ang. *node pools*).

Można skalować grupy instancji lub grupy węzłów, zmieniając minimalny i maksymalny rozmiar grupy, określony typ instancji lub oba te elementy.

Skalowanie w dół

Zasadniczo nie ma też problemu ze zmniejszeniem klastra Kubernetes. Możesz nakazać Kubernetes o opróżnienie węzłów, które chcesz usunąć, co spowoduje stopniowe zamykanie lub przenoszenie działających Podów na inne węzły.

Większość narzędzi do zarządzania klastrami automatycznie wykona opróżnianie węzłów. Możesz także zrobić to samodzielnie i skorzystać z polecenia `kubectl drain`, pod warunkiem że w pozostałej części klastra jest wystarczająca ilość wolnej pojemności, aby usuwane Pody ponownie uruchomić gdzie indziej. Po pomyślnym opróżnieniu węzłów możesz je zakończyć.

Aby uniknąć zbytniego zmniejszania liczby replik Podów dla danej usługi, możesz skorzystać z `PodDisruptionBudgets`, aby określić minimalną liczbę dostępnych Podów lub maksymalną liczbę Podów, które mogą być niedostępne w dowolnym momencie (patrz „*Budżety zakłóceń Poda*” w rozdziale 5.).

Jeśli opróżnienie węzła spowoduje, że Kubernetes przekroczy te limity, operacja ta będzie blokowana do momentu zmiany limitów lub zwolnienia dodatkowych zasobów w klastrze.

Opróżnianie umożliwia Podom poprawne zamknięcie, posprzątanie po sobie i zapisanie stanu. W przypadku większości aplikacji lepiej po prostu zamknąć węzeł, co spowoduje natychmiastowe zakończenie Podów.



Najlepsze praktyki

Nie zamykaj węzłów, kiedy już ich nie potrzebujesz. Opróżnij je najpierw, aby zapewnić migrację ich obciążeniu do innych węzłów i upewnić się, że w klastrze pozostała wystarczająca ilość wolnej pojemności.

Automatyczne skalowanie

Większość dostawców usług chmurowych obsługuje *automatyczne skalowanie*, czyli automatyczne powiększanie lub zmniejszanie liczby instancji w grupie — zgodnie z niektórymi parametrami lub planem. Przykładowo grupy automatycznego skalowania AWS (ASG — ang. *autoscaling groups*) mogą poprawić minimalną i maksymalną liczbę instancji w taki sposób, że jeśli jedna instancja ulegnie awarii, inna zostanie uruchomiona lub jeśli zostanie uruchomionych zbyt wiele instancji, kilka z nich zostanie zamkniętych.

Alternatywnie, jeśli Twoje potrzeby zmieniają się w zależności od pory dnia, możesz sprawić, aby grupa zmieniała rozmiar węzła w określonych porach. Możesz również skonfigurować grupę skalowania tak, aby rozmiar węzła zmieniał się dynamicznie w zależności od potrzeb: jeśli np. średnie użycie procesora przekroczy 90% w ciągu 15 minut, to instancje będą usuwane, a gdy obciążenie procesora nie spada poniżej progu, będą dodawane. Gdy ruch ponownie spadnie, grupa może skalować węzeł w dół, aby zaoszczędzić pieniądze.

Kubernetes ma dodatek o nazwie Cluster Autoscaler, który może korzystać z narzędzi do zarządzania klastrami, takich jak `kops` — w celu umożliwienia zastosowania funkcji skalowania w chmurze. Klastry zarządzane, takie jak Azure Kubernetes Service, również oferują automatyczne skalowanie.

Właściwe dobranie ustawień automatycznego skalowania może pochłonąć trochę czasu. Jednak dla wielu użytkowników nie jest to konieczne. Większość klastrów Kubernetes startuje od małego rozmiaru i stopniowo się rozwija, dodając węzły wraz ze wzrostem zużycia zasobów.

Jednak w przypadku dużych zastosowań, w których wymagania są bardzo zmienne, automatyczne skalowanie klastra jest bardzo przydatną funkcją.



Najlepsze praktyki

Nie włączaj opcji automatycznego skalowania klastra tylko dlatego, że tam jest. Prawdopodobnie nie będziesz jej potrzebować, aż obciążenia nie staną się bardzo zmienne. Dopóki nie zorientujesz się, jak potrzeby zmieniają się w czasie, skaluj kластer ręcznie.

Sprawdzanie zgodności

Kiedy Kubernetes nie jest aplikacją Kubernetes? Elastyczność Kubernetes oznacza, że istnieje wiele różnych metod konfigurowania klastrów Kubernetes — może to stanowić potencjalny problem. Jeśli Kubernetes ma być uniwersalną platformą, powinieneś być w stanie uruchomić dowolne obciążenie w dowolnym klastrze Kubernetes, a jego praca powinna być zgodna z Twoimi przewidywaniemi. Oznacza to, że są dostępne te same wywołania API oraz obiekty Kubernetes, które muszą zachowywać się tak samo oraz działać tak, jak powinny.

Na szczęście sam Kubernetes zawiera zestaw testów, które weryfikują *zgodność* (ang. *conformant*) danego klastra Kubernetes; oznacza to, że kластer spełnia podstawowy zestaw wymagań dla danej wersji Kubernetes. Te testy zgodności są bardzo przydatne dla administratorów Kubernetes.

Jeśli kластer ich nie przejdzie, oznacza to problem z konfiguracją, który należy rozwiązać. Jeśli przejdzie, wiedzą o tym, że kластer jest zgodny, daje pewność, że aplikacje zaprojektowane dla Kubernetes będą działać z klastrem, a rzeczy, które zbudujesz na klastrze, będą działały także gdzie indziej.

Certyfikat CNCF

Cloud Native Computing Foundation (CNCF) jest oficjalnym właścicielem projektu oraz znaku towarowego Kubernetes (patrz „Model Cloud Native” w rozdziale 1.) i zapewnia różne rodzaje certyfikatów dla produktów, inżynierów i dostawców powiązanych z Kubernetes.

Zgodne z Kubernetes

Jeśli korzystasz z zarządzanej lub częściowo zarządzanej usługi Kubernetes, sprawdź, czy nosi ona znak i logo Certified Kubernetes (patrz rysunek 6.1). Oznacza to, że dostawca i usługa spełniają standard Certified Kubernetes (<https://github.com/cncf/k8s-conformance>), określony przez Cloud Native Computing Foundation (CNCF).

Jeśli produkt ma w nazwie *Kubernetes*, musi być certyfikowany przez CNCF. Oznacza to, że klienci dokładnie wiedzą, co otrzymują, i mogą mieć pewność, że produkty będą zgodne z innymi usługami Kubernetes. Producenci mogą samodzielnie certyfikować swoje produkty, uruchamiając narzędzie do sprawdzania zgodności Sonobuoy (patrz „Testy zgodności z Sonobuoy” dalej w tym rozdziale).



Rysunek 6.1. Znak Certified Kubernetes oznacza, że dany produkt lub usługa zostały zaakceptowane przez organizację CNCF

Certyfikowane produkty Kubernetes muszą również śledzić zmiany w najnowszych wersjach Kubernetes, zapewniając aktualizacje co najmniej raz w roku. Nie tylko usługi zarządzane mogą nosić znak Certified Kubernetes, dystrybucje i narzędzia instalatora również.

Certyfikowany administrator Kubernetes (CKA — ang. Certified Kubernetes Administrator)

Aby zostać certyfikowanym administratorem Kubernetes, musisz wykazać się kluczowymi umiejętnościami zarządzania klastrami Kubernetes w środowisku produkcyjnym, w tym instalacją i konfiguracją, obsługą sieci, konserwacją, znajomością interfejsu API, bezpieczeństwem i rozwiązywaniem problemów. Każdy może przystąpić do egzaminu CKA. Jest on przeprowadzany online i obejmuje serię trudnych testów praktycznych.

CKA ma reputację trudnego, kompleksowego egzaminu, który naprawdę sprawdza Twoje umiejętności i wiedzę. Możesz być pewien, że każdy inżynier posiadający certyfikat CKA naprawdę zna Kubernetes. Jeśli prowadzisz działalność opierającą się na Kubernetes, zastanów się nad oddelegowaniem części pracowników na szkolenie CKA, szczególnie bezpośrednio odpowiedzialnych za zarządzanie klastrami.

Certyfikowany dostawca usług Kubernetes (KCSP — ang. Kubernetes Certified Service Provider)

Sami dostawcy mogą ubiegać się o udział w programie certyfikowanego dostawcy usług Kubernetes (KCSP). Aby to zrobić, sprzedawca musi być członkiem CNCF, zapewniać obsługę przedsiębiorstw (np. wsparcie klienta przez odpowiednich inżynierów), brać aktywny udział w społeczności Kubernetes oraz zatrudniać trzech lub więcej inżynierów posiadających certyfikat CKA.



Najlepsze praktyki

Poszukaj znaku Certified Kubernetes, aby upewnić się, że produkt spełnia standardy CNCF. Poszukaj dostawców, którzy mają certyfikat KCSP, a jeśli zatrudniasz administratorów Kubernetes, poszukaj tych z kwalifikacjami CKA.

Testy zgodności z Sonobuoy

Jeśli zarządzasz własnym klastrzem lub nawet korzystasz z usługi zarządzanej, ale chcesz dokładnie sprawdzić, czy jest ona poprawnie skonfigurowana i aktualna, możesz uruchomić testy zgodności Kubernetes. Standardowym narzędziem do uruchamiania tych testów jest Sonobuoy.



Najlepsze praktyki

Po skonfigurowaniu klastra po raz pierwszy uruchom skaner Sonobuoy, aby sprawdzić, czy jest on zgodny ze standardami i czy wszystko działa. Uruchamiaj go co jakiś czas, aby upewnić się, że nie występują problemy ze zgodnością.

Walidacja i audyt

Zgodność klastra to podstawa: każdy kластер produkcyjny z pewnością powinien być zgodny. Istnieje jednak wiele typowych problemów z konfiguracjami Kubernetes i obciążeniami, których nie sprawdzą testy zgodności. Oto przykłady.

- Używanie zbyt dużych obrazów kontenerów może zmarnować dużo czasu i zasobów klastra.
- Obiekty Deployment, które posiadają tylko jedną replikę Poda, nie zapewniają wysokiej niezawodności.
- Uruchamianie procesów w kontenerach jako root jest potencjalnym zagrożeniem dla bezpieczeństwa (patrz „Bezpieczeństwo kontenerów” w rozdziale 8).

W tym punkcie przyjrzymy się niektórym narzędziom i technikom, które mogą pomóc w znalezieniu problemów związanych z klastrem. Powiemy Ci, co je powoduje.

K8Guard

Narzędzie K8Guard (<https://target.github.io/infrastructure/k8guard-the-guardian-angelfor-kuberentes>), opracowane w firmie Target, może sprawdzić typowe problemy związane z Twoim klastrem Kubernetes. Może podjąć działania naprawcze lub po prostu wysłać odpowiednie powiadomienie. Możesz skonfigurować je, dostosowując do własnych polityk (np. możesz określić, że K8Guard powinien Cię ostrzec, jeśli jakikolwiek obraz kontenera jest większy niż 1 GiB lub reguła ingress pozwala na dostęp z dowolnego miejsca).

K8Guard eksportuje również metryki, które mogą być gromadzone przez system monitorowania, takie jak Prometheus (patrz rozdział 16.), podające, jak duża liczba obiektów Deployment narusza zasady, a także sprawdzające wydajność odpowiedzi interfejsu API Kubernetes. Pomoże to we wcześniejszym wykryciu oraz naprawie problemu.

Dobrym pomysłem jest narzędzie K8Guard działające w klastrze; wtedy otrzymasz ostrzeżenia o wszelkich naruszeniach, gdy tylko się pojawią.

Copper

Copper (<https://copper.sh/>) to narzędzie do sprawdzania manifestów Kubernetes przed ich wdrożeniem oraz oznaczania typowych problemów lub egzekwowania niestandardowych zasad. Zawiera język DSL (ang. *Domain-Specific Language*) służący do definiowania reguł i zasad sprawdzania poprawności.

Przykładowo poniżej wyrażona jest reguła w języku Copper, która blokuje dowolny kontener przy użyciu tagu latest (patrz „Tag latest” w rozdziale 8.; sprawdzisz, dlaczego jest to zły pomysł):

```
rule NoLatest ensure {
    fetch("$.spec.template.spec.containers..image")
        .as(:image)
        .pick(:tag)
        .contains("latest") == false
}
```

Po uruchomieniu polecenia copper check na manifeście Kubernetes, który zawiera flagę latest w specyfikacji obrazu kontenera, zobaczysz komunikat o błędzie:

```
copper check --rules no_latest.cop --files deployment.yml
Validating part 0
NoLatest - FAIL
```

Dobrym pomysłem jest dodanie takich reguł Copper i uruchomienie tego narzędzia jako części systemu kontroli wersji (np. w celu sprawdzenia manifestów Kubernetes przed zatwierdzeniem lub w ramach automatycznych kontroli dla żądań pull).

Podobnym narzędziem, które sprawdza poprawność manifestów względem specyfikacji API Kubernetes, jest kubeval (patrz „kubeval” w rozdziale 12.).

kube-bench

Narzędzie kube-bench (<https://github.com/aquasecurity/kube-bench>) służy do kontrolowania klastra Kubernetes na podstawie testów opracowanych przez CIS (ang. *Center for Internet Security*). W efekcie działania sprawdzane jest, czy kластер został skonfigurowany zgodnie z najlepszymi praktykami bezpieczeństwa. Chociaż prawdopodobnie nie musisz tego robić, możesz sam skonfigurować testy, które uruchamiają kube-bench, a nawet dodać własne, określone jako dokumenty w formacie YAML.

Dziennik kontroli Kubernetes (ang. Kubernetes audit log)

Założymy, że znajdziesz problem w klastrze, np. odkryjesz Pod, którego nie rozpoznajesz, i chcesz wiedzieć, skąd się wziął. Jak dowiedzieć się, co robił, kiedy był w klastrze? Dziennik kontroli Kubernetes (<https://kubernetes.io/docs/tasks/debug-application-cluster/audit/>) powie Ci prawdę.

Po włączeniu dziennika kontroli wszystkie żądania do interfejsu API klastra będą rejestrowane razem ze znacznikiem czasu. Uzyskasz informacje o tym, kto wysłał żądanie (które konto usługi), szczegóły żądania, takie jak kwerenda, której dotyczy zapytanie, i jaka była odpowiedź.

Zdarzenia audytu można przesyłać do centralnego systemu rejestrowania, w którym można je filtrować i ostrzegać o nich, tak jak w przypadku innych danych dziennika (patrz rozdział 15.). Dobra usługa zarządzana, taka jak Google Kubernetes Engine, będzie mieć domyślnie włączony dziennik kontroli. W innym przypadku konieczne może być samodzielne skonfigurowanie klastra, aby włączyć tę funkcjonalność.

Testowanie chaosu

Wspomnieliśmy już o zasadzie „Ufaj, ale weryfikuj”. Wskazaliśmy, że jedynym prawdziwym sposobem na sprawdzenie wysokiej niezawodności jest unieruchomienie jednego węzła klastra lub wielu węzłów i sprawdzenie, co się stanie. To samo dotyczy wysokiej niezawodności Twoich Podów i aplikacji. Możesz losowo wybrać Pod, dla którego chcesz zakończyć działanie, a następnie sprawdzić, czy Kubernetes uruchamia go ponownie i czy nie ma to wpływu na poziom błędów.

Wykonanie tej operacji ręcznie jest czasochłonne. Nie zdając sobie z tego sprawy, możesz nieświadomie wpływać na zasoby, o których wiesz, że są krytyczne dla aplikacji. Aby test był rzetelny, proces musi być zautomatyzowany.

Ten rodzaj automatycznej, losowej ingerencji w usługi produkcyjne jest znany jako testowanie *Chaos Monkey*. Nazwa pochodzi od nazwy narzędzia opracowanego przez Netflix w celu przetestowania jego infrastruktury.

Wyobraź sobie małpę wchodząą do centrum danych; to te farmy serwerów, które obsługują wszystkie kluczowe funkcje naszej działalności online. Małpa losowo rozrywa kable, niszczy urządzenia. ...

Wyzwaniem dla kierowników działów IT jest takie zaprojektowanie systemu informatycznego, za który są odpowiedzialni, aby działał pomimo tych małp, o których nikdy nie wie, kiedy dotrą na miejsce i co zniszczą.

— Antonio Garcia Martinez, Chaos Monkeys

Oprócz samego testowania Chaos Monkey, które losowo unieruchamia serwery chmurowe, pakiet *Simian Army* Netfliksa zawiera także inne narzędzia *inżynierii chaosu*, takie jak Latency Monkey, które wprowadza opóźnienia komunikacyjne w celu symulacji problemów z siecią, Security Monkey, które szuka znanych podatności oraz Chaos Gorilla, które usuwa całą strefę dostępności AWS.

Tylko produkcja jest produkcją

Filozofię Chaos Monkey możesz również zastosować w aplikacjach Kubernetes. Aby uniknąć zakłócania środowiska produkcyjnego, możesz uruchomić narzędzia inżynierii chaosu w klastrze programistycznym. Jednak to za dużo Ci nie powie. Aby dowiedzieć się czegoś o ewentualnych problemach środowiska produkcyjnego, musisz przetestować właśnie to środowisko.

Wiele systemów jest zbyt dużych, złożonych i kosztownych, aby można je było sklonować. Wyobraź sobie, że na potrzeby testów próbujesz rozpakować kopię Facebooka (z wieloma, globalnie dystrybuowanymi centrami danych).

Nieprzewidywalność ruchu użytkownika uniemożliwia przeprowadzenie realnego testu; jeśli nawet potrafisz doskonale odtworzyć wczorajszy ruch, nadal nie możesz przewidzieć jutrzejszego ruchu. Tylko produkcja jest produkcją.

— Charity Majors (<https://opensource.com/article/17/8/testing-production>)

Ważne jest również, aby pamiętać, żeby eksperymenty z chaosem są najbardziej przydatne; należy je przeprowadzać w sposób zautomatyzowany i ciągły. Nie warto tego robić raz i na tej podstawie decydować, że Twój system jest niezawodny.

Cały sens automatyzacji eksperymentów z chaosem polega na tym, że aby zbudować zaufanie do swojego systemu, musisz je uruchamiać wielokrotnie. To nie tylko odkrywanie nowych słabości, ale przede wszystkim nabieranie pewności, że Twój system jest odporny na takie słabości.

— Ros Miles (ChaosIQ) (<https://medium.com/chaosiq/exploring-multi-levelweaknesses-using-automated-chaos-experiments-aa30f0605ce>)

Istnieje kilka narzędzi, które służą do przeprowadzenia zautomatyzowanego testu chaosu w Twoim klastrze.

Oto kilka z nich.

chaoskube

Narzędzie chaoskube (<https://github.com/linki/chaoskube>) losowo unieruchamia kapsuły w klastrze. Domyślnie działa w trybie dry-run, który pokazuje, jakie przeprowadziłby operacje — ale bez ich uruchamiania.

Narzędzie to możesz skonfigurować tak, aby włączało lub wyłączało Pody na podstawie etykiet (patrz „Etykiety” w rozdziale 9.), adnotacji i przestrzeni nazw oraz aby unikało określonych okresów lub dat (np. nie testuj niczego w wigilię Bożego Narodzenia). Domyślnie jednak chaoskube unieruchomi każdy Pod w dowolnej przestrzeni nazw, w tym Pody systemu Kubernetes, a nawet samo siebie.

Gdy będziesz zadowolony z konfiguracji filtra chaoskube, możesz wyłączyć tryb dry-run i pozwolić mu działać.

Narzędzie chaoskube jest proste w instalacji i konfiguracji oraz idealne do rozpoczęcia zabaw z inżynierią chaosu.

kube-monkey

Narzędzie kube-monkey (<https://github.com/asobti/kube-monkey>) działa w ustalonym czasie (domyślnie o 8. rano w dni powszednie) i buduje harmonogram obiektów Deployment, na których będą przeprowadzone testy na resztę dnia (domyślnie od 10. do 16.). W przeciwieństwie do niektórych

innych narzędzi, kube-monkey działa na podstawie wcześniej uzgodnionych działań: zostaną przetestowane tylko te Pody, które używają określonej adnotacji.

Oznacza to, że możesz dodawać testy kube-monkey do określonych aplikacji lub usług podczas ich tworzenia i ustawać różne poziomy częstotliwości oraz agresji w zależności od usługi. Przykładowo poniższa adnotacja na Podzie ustali średni czas między awariami (MTBF — ang. *mean time between failures*) wynoszący dwa dni:

```
kube-monkey/mtbf: 2
```

Adnotacja `kill-mode` pozwala określić, ile (lub jaki maksymalny procent) Podów na obiektach Deployment zostanie unieruchomionych. Następujące adnotacje unieruchomią do 50% kapsuł na docelowym obiekcie Deployment:

```
kube-monkey/kill-mode: "random-max-percent"  
kube-monkey/kill-value: 50
```

PowerfulSeal

PowerfulSeal (<https://github.com/bloomberg/powerfulseal>) to narzędzie inżynierii chaosu Kubernetes o otwartym kodzie, które działa w dwóch trybach — interaktywnym i autonomicznym. Tryb interaktywny pozwala eksplorować klaster i ręcznie psuć różne rzeczy, aby zobaczyć, co się stanie. Może unieruchamiać węzły, przestrzenie nazw, obiekty Deployment i pojedyncze Pody.

Tryb autonomiczny wykorzystuje określony przez Ciebie zestaw zasad: na jakich zasobach operować, których unikać, kiedy uruchomić (możesz skonfigurować je tak, aby działało np. tylko w godzinach pracy od poniedziałku do piątku) i w jakim stopniu ma być agresywne (unieruchamiaj np. określony procent wszystkich pasujących obiektów Deployment). Pliki zasad PowerfulSeal są bardzo elastyczne i pozwalają skonfigurować prawie każdy możliwy scenariusz inżynierii chaosu.



Najlepsze praktyki

Jeśli Twoje aplikacje wymagają wysokiej niezawodności, regularnie uruchamiaj narzędzie do testowania chaosu, takie jak chaokube, aby upewnić się, że nieoczekiwane awarie węzłów lub Podów nie powodują problemów. Upewnij się, że problemy rozwiązywane są przez osoby odpowiedzialne za obsługę klastra i testowanych aplikacji.

Podsumowanie

Naprawdę trudno ustalić rozmiar oraz konfigurację pierwszych klastrów Kubernetes. Istnieje wiele opcji i nie wiesz, czego będziesz potrzebować, dopóki nie zdobędziesz doświadczenia produkcyjnego.

Nie możemy podjąć tych decyzji za Ciebie, ale mamy nadzieję, że przynajmniej podaliśmy kilka pomocnych wskazówek do przemyślenia.

- Przed udostępnieniem produkcyjnego klastra Kubernetes zastanów się, ile węzłów oraz jakiego rozmiaru potrzebujesz.
- Potrzebujesz co najmniej trzech węzłów master (lub nie, jeśli korzystasz z usługi zarządzanej) i co najmniej dwóch (najlepiej trzech) węzłów worker. Tworzenie klastrów Kubernetes może

się początkowo wydawać trochę drogie, zwłaszcza gdy wykonujesz tylko kilka małych zadań, ale nie zapomnij o zaletach wbudowanej odporności i skalowania.

- Klastry Kubernetes można skalować do wielu tysięcy węzłów i setek tysięcy kontenerów.
- Jeśli chcesz skalować do większych wartości, użyj wielu klastrów (czasem musisz to zrobić również ze względów bezpieczeństwa lub zgodności). Jeśli chcesz zreplikować obciążenia w różnych klastrach, możesz połączyć klastry za pomocą federacji.
- Typowy rozmiar instancji dla węzła Kubernetes to 1 procesor, 4 GiB RAM. Warto jednak mieścić kilka różnych rozmiarów węzłów.
- Kubernetes nie jest przeznaczony tylko do rozwiązań chmurowych; działa również na serwerach bare-metal. Jeśli masz takie serwery, dlaczego ich nie użyć?
- Możesz skalować kластer ręcznie w górę i w dół bez większych problemów i prawdopodobnie nie będziesz musiał robić tego zbyt często. Miło stosować automatyczne skalowanie, ale nie jest to ważne.
- Istnieje dobrze zdefiniowany standard dla dostawców i produktów Kubernetes: logo *Certified Kubernetes*. Jeśli go nie widzisz, zapytaj.
- Testowanie chaosu polega na losowym unieruchamianiu Podów i sprawdzaniu, czy aplikacja nadal działa. Jest to użyteczne, ale środowisko chmurowe i tak przeprowadza własne testy chaosu, bez pytania o Twoją zgodę.

Narzędzia Kubernetes

Mój mechanik powiedział: „Nie mogłem naprawić hamulców, więc zwiększyłem głośność Twojego klaksonu”.

— Steven Wright

Ludzie zawsze pytają nas: „Co z tymi wszystkimi narzędziami Kubernetes? Czy ich potrzebuję? Jeśli tak, to których? I jak one działają?”.

W tym rozdziale omówimy kilka narzędzi, które pomagają w pracy z Kubernetes. Pokażemy kilka zaawansowanych zastosowań kubectl oraz kilka przydatnych narzędzi, takich jak jq, kubectlx, kubens, kube-ps1, kubeshell, Click, kubed-sh, Stern i BusyBox.

Znowu o kubectl

W rozdziale 2. poznaleś już kubectl, a ponieważ jest to podstawowe narzędzie służące do interakcji z Kubernetes, powinieneś już dobrze znać jego podstawy. Przyjrzyjmy się teraz bardziej zaawansowanym funkcjom kubectl, które mogą być dla Ciebie nowe.

Aliases powłoki

Jedną z pierwszych rzeczy, które większość użytkowników Kubernetes robi, aby ułatwić sobie życie, jest utworzenie aliasu dla polecenia kubectl. Mamy np. w pliku `.bash_profile` ustawiony następujący alias:

```
alias k = kubectl
```

Teraz, zamiast wpisywać kubectl w całości dla każdego polecenia, możemy po prostu użyć k:

```
k get pods
```

Jeśli często używasz komend kubectl, również możesz dla nich utworzyć aliasy. Oto kilka możliwych przykładów:

```
alias kg=kubectl get
alias kgdep=kubectl get deployment
alias ksys=kubectl --namespace=kube-system
alias kd=kubectl describe
```

Inżynier Google Ahmet Alp Balkan opracował logiczny system aliasów (<https://ahmet.im/blog/kubectl-aliases>) i utworzył skrypt do generowania ich (obecnie około 800 aliasów).

Jednak nie musisz z nich korzystać; sugerujemy zacząć od aliasu `k` i dodawać kolejne dla najczęściej używanych poleceń.

Używanie przełączników

Jak większość narzędzi wiersza poleceń, `kubectl` obsługuje skrócone formy wielu swoich przełączników. To pozwala zaoszczędzić dużo pisania.

Möżesz np. skrócić nazwę przełącznika `--namespace` do po prostu `-n` (patrz „Korzystanie z przestrzeni nazw” w rozdziale 5.):

```
kubectl get pods -n kube-system
```

Narzędzie `kubectl` często korzysta z zasobów pasujących do zestawu etykiet, wykorzystuje w tym celu przełącznik `--selector` (patrz „Etykiety” w rozdziale 9.). Na szczęście zapis ten można skrócić do `-l` (etykiety):

```
kubectl get pods -l "environment=staging"
```

Skracanie typów zasobów

Typowym zastosowaniem `kubectl` jest uzyskiwanie listy zasobów różnych typów, takich jak Pody, obiekty Deployment, Serwis i przestrzenie nazw. Służy do tego polecenie `kubectl get`, np. razem z `deployments`.

Aby to przyspieszyć, `kubectl` obsługuje krótkie formy tych typów zasobów:

```
kubectl get po
kubectl get deploy
kubectl get svc
kubectl get ns
```

Inne przydatne skróty to: `no` dla `nodes`, `cm` dla `configmaps`, `sa` dla `serviceaccounts`, `count`, `ds` dla `daemonsets` i `pv` dla `persistentvolumes`.

Automatyczne uzupełnianie poleceń `kubectl`

Jeśli korzystasz z powłok bash lub zsh, możesz używać automatycznego uzupełniania poleceń `kubectl`. Uruchom to polecenie, aby wyświetlić instrukcje dotyczące włączania autouzupełniania dla Twojej powłoki:

```
kubectl completion -h
```

Postępuj zgodnie z instrukcjami. Po ich wykonaniu powinieneś być w stanie za pomocą klawisza Tab wykonać autouzupełnianie polecenia `kubectl`. Spróbuj:

```
kubectl cl <TAB>
```

Polecenie powinno się uzupełnić do `kubectl cluster-info`.

Jeśli wpiszesz tylko kubectl i naciśniesz klawisz Tab dwa razy, zobaczysz wszystkie dostępne polecenia:

```
kubectl <TAB><TAB>
alpha attach cluster-info cordon describe ...
```

Możesz użyć tej samej techniki, aby wyświetlić listę wszystkich flag, z których możesz skorzystać w bieżącym poleceniu:

```
kubectl get pods --<TAB><TAB>
--all-namespaces --cluster= --label-columns= ...
```

Bardzo przydatne jest to, że kubectl będzie także automatycznie uzupełniał nazwy Podów, obiektów Deployment, przestrzeni nazw itd.:

```
kubectl -n kube-system describe pod <TAB><TAB>
event-exporter-v0.1.9-85bb4fd64d-2zjng
kube-dns-autoscaler-79b4b844b9-2wg1c
fluentd-gcp-scaler-7c5db745fc-h7ntr
...
...
```

Uzyskiwanie pomocy

Najlepsze narzędzia wiersza poleceń zawierają także dokładną dokumentację — kubectl nie jest tutaj wyjątkiem. Pełny przegląd dostępnych poleceń możesz uzyskać za pomocą kubectl -h:

```
kubectl -h
```

Możesz przejść dalej i uzyskać szczegółową dokumentację dla każdego polecenia, ze wszystkimi dostępnymi opcjami i zestawem przykładów, wpisując kubectl COMMAND -h:

```
kubectl get -h
```

Uzyskiwanie pomocy na temat zasobów Kubernetes

Oprócz dokumentacji na swój temat, kubectl może również wyświetlić pomoc dotyczącą obiektów Kubernetes, takich jak Deployment lub Pod. Polecenie kubectl explain wyświetli dokumentację dla określonego typu zasobu:

```
kubectl explain pods
```

Możesz uzyskać dodatkowe informacje na temat określonego pola zasobu za pomocą kubectl explain ZASÓB.POLE. W rzeczywistości przy użyciu explain możesz uzyskiwać informacje dla coraz niższych poziomów:

```
kubectl explain deploy.spec.template.spec.containers.livenessProbe.exec
```

Alternatywnie wypróbuj polecenie kubectl explain --recursive, które pokazuje pola w polach w obrębie pól... Uważaj, żebyś nie miał zawrotów głowy!

Bardziej szczegółowe wyniki

Wiesz już, że polecenie kubectl get wyświetli listę zasobów różnych typów, takich jak Pody:

```
kubectl get pods
NAME READY STATUS RESTARTS AGE
demo-54f4458547-pqdxn 1/1 Running 6 5d
```

Możesz zobaczyć dodatkowe informacje, takie jak węzeł, na którym działa każdy Pod, za pomocą flagi `-o wide`:

```
kubectl get pods -o wide
NAME ... IP NODE
demo-54ff4458547-pqdxn ... 10.76.1.88 gke-k8s-cluster-1-n1-standard...
```

(Pomineliśmy informacje, które widzisz bez użycia `-o wide`, tylko ze względu na miejsce).

W zależności od typu zasobu, opcja `-o wide` pokaże różne informacje. Oto przykład dla węzłów:

```
kubectl get nodes -o wide
NAME ... EXTERNAL-IP OS-IMAGE KERNEL-VERSION
gke-k8s-...816n ... 35.233.136.194 Container... 4.14.22+
gke-k8s-...dwtv ... 35.227.162.224 Container... 4.14.22+
gke-k8s-...67ch ... 35.233.212.49 Container... 4.14.22+
```

Praca z JSON Data i jq

Domyślny format wyjściowy dla polecenia `kubectl get` to zwykły tekst. Można jednak uzyskać takie informacje w formacie JSON:

```
kubectl get pods -n kube-system -o json
{
  "apiVersion": "v1",
  "items": [
    {
      "apiVersion": "v1",
      "kind": "Pod",
      "metadata": {
        "creationTimestamp": "2018-05-21T18:24:54Z",
        ...
      }
    }
  ]
}
```

Nic dziwnego, że uzyskany wynik jest dosyć duży (około 5000 linii na klaster). Ponieważ dane wyjściowe są sformatowane w formacie JSON, do ich filtrowania możesz użyć innych narzędzi, takich jak nieocenione jq.

Jeśli nie posiadasz jq (<https://stedolan.github.io/jq/manual>), zainstaluj je (<https://stedolan.github.io/jq/download>) w odpowiedni sposób, dostosowany dla Twojego systemu (brew install jq dla macOS, apt install jq dla Debian / Ubuntu itd.).

Narzędzia jq po zainstalowaniu możesz użyć do odpytywania i filtrowania danych wyjściowych `kubectl`:

```
kubectl get pods -n kube-system -o json | jq '.items[] .metadata.name'
"event-exporter-v0.1.9-85bb4fd64d-2zjng"
"fluentd-gcp-scaler-7c5db745fc-h7ntr"
"fluentd-gcp-v3.0.0-5m627"
"fluentd-gcp-v3.0.0-h5fjg"
...
```

jq to bardzo potężne narzędzie służące do odpytywania i przekształcania danych JSON.

Możesz np. wyświetlić listę najbardziej obciążonych węzłów według liczby Podów działających na każdym z nich:

```
kubectl get pods -o json --all-namespaces | jq '.items | group_by(.spec.nodeName) | map({"nodeName": .[0].spec.nodeName, "count": length}) | sort_by(.count) | reverse'
```

Dostępne jest środowisko testowe online (<https://jqplay.org/>) dla jq, w którym można wkleić dane JSON i wypróbować różne zapytania, aby uzyskać dokładnie pożądany wynik.

Jeśli nie masz dostępu do jq, narzędzie kubectl obsługuje również zapytania JSONPath (<https://kubernetes.io/docs/reference/kubectl/jsonpath/>). JSONPath jest językiem zapytań JSON, który nie jest tak potężny jak jq, ale jest przydatny do stosowania w jednoliniowych poleceniach:

```
kubectl get pods -o=jsonpath={.items[0].metadata.name}
demo-66ddf956b9-pnknx
```

Oglądzanie obiektów

Gdy czekasz na uruchomienie szeregu Podów, może być denerwujące, że musisz wpisywać kubectl get pods... co kilka sekund, aby zobaczyć, czy coś się stało.

Narzędzie kubectl udostępnia flagę --watch (w skrócie -w), aby zaoszczędzić czas, np.:

```
kubectl get pods --watch
```

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------------|---------------|-------------------|----------|-----|
| demo-95444875c-z9xv4 | 0/1 | ContainerCreating | 0 | 1s |
| ... | [time passes] | ... | | |
| demo-95444875c-z9xv4 | 0/1 | Completed | 0 | 2s |
| demo-95444875c-z9xv4 | 1/1 | Running | 0 | 2s |

Za każdym razem, gdy zmienia się stan jednego z pasujących Podów, na terminalu pojawi się aktualizacja. (Patrz „Oglądzanie zasobów Kubernetes za pomocą kubespy” dalej w tym rozdziale, aby uzyskać schludny sposób oglądania dowolnego rodzaju zasobów).

Opisywanie obiektów

Aby uzyskać naprawdę szczegółowe informacje o obiektach Kubernetes, możesz użyć polecenia kubectl describe:

```
kubectl describe pods demo-d94cffc44-gvgzm
```

Sekcja Events może być szczególnie przydatna do rozwiązywania problemów z kontenerami, które nie działają poprawnie, ponieważ rejestruje każdy etap cyklu życia kontenera, a także wszelkie występujące błędy.

Praca z zasobami

Do tej pory używałeś kubectl głównie do tworzenia zapytań lub list, a także do wczytywania deklaratywnych manifestów YAML za pomocą polecenia kubectl apply. Jednak kubectl zawiera również pełny zestaw poleceń *imperatywnych*; są to operacje, które bezpośrednio tworzą lub modyfikują zasoby.

Imperatywne polecenia kubectl

Jeden przykład pokazaliśmy w punkcie „Uruchamianie aplikacji demonstracyjnej” w rozdziale 2. — było to polecenie `kubectl run`, które domyślnie tworzy Deployment do uruchomienia w określonym kontenerze.

Możesz również jawnie utworzyć większość zasobów, używając polecenia `kubectl create`:

```
kubectl create namespace my-new-namespace
namespace "my-new-namespace" created
```

Podobnie polecenie `kubectl delete` usunie zasób:

```
kubectl delete namespace my-new-namespace
namespace "my-new-namespace" deleted
```

Polecenie `kubectl edit` umożliwia przeglądanie i modyfikowanie dowolnego zasobu:

```
kubectl edit deployments my-deployment
```

Spowoduje ono otwarcie domyślnego edytora z plikiem manifestu YAML, reprezentującym określony zasób.

Jest to dobry sposób, by szczegółowo zobaczyć konfigurację dowolnego zasobu, ale możesz także wprowadzić dowolne zmiany w edytorze. Kiedy zapiszesz plik i zamkniesz edytor, `kubectl` zaktualizuje zasób, dokładnie tak, jakbyś uruchomił polecenie `kubectl apply` dla pliku manifestu zasobu.

Jeśli wprowadziłeś jakieś błędy, np. w formacie YAML, `kubectl` poinformuje Cię i ponownie otworzy plik, aby naprawić problem.

Kiedy nie należy używać poleceń imperatywnych?

W tej książce podkreśliśmy znaczenie korzystania z infrastruktury *deklaratywnej* jako kodu. Nie powinno więc dziwić, że nie zalecamy używania imperatywnych poleceń `kubectl`.

Chociaż mogą być one bardzo przydatne do szybkiego testowania różnych rzeczy lub wypróbowywania pomysłów, głównym problemem związanym z poleceniami imperatywnymi jest to, że nie masz jednego źródła prawdy. Nie ma sposobu, aby dowiedzieć się, kto uruchomił polecenia w klasztre, w jakim czasie i jaki był ich efekt. Jak tylko uruchomisz jakieś polecenie imperatywne, stan klastra nie zostanie zsynchronizowany z plikami manifestu przechowywanymi w kontroli wersji.

Następnym razem, gdy ktoś zastosuje manifest YAML, wszelkie zmiany, które wprowadziłeś imperatywnie, zostaną nadpisane i utracone. Może to prowadzić do zaskakujących wyników oraz wywrieć potencjalnie niekorzystny wpływ na usługi krytyczne.

W czasie dyżuru Alice nagle następuje znaczny wzrost obciążenia usługi, którą zarządza. Alice korzysta z polecenia kubectl scale, aby zwiększyć liczbę replik z 5 do 10. Kilka dni później Bob edytuje manifest YAML w repozytorium, aby użyć nowego obrazu kontenera. Nie zauważa jednak, że liczba replik w pliku to obecnie 5, a nie 10 aktywnych w produkcji. Bob kontynuuje wdrażanie, które zmniejsza liczbę replik o połowę, powodując natychmiastowe przejęcie lub awarię.

— Kelsey Hightower i inni, *Kubernetes. Tworzenie niezawodnych systemów rozproszonych*

Alice zapomniała zaktualizować pliki w repozytorium po tym, jak wprowadziła konieczną zmianę. Jest to łatwe do przeoczenia, szczególnie pod wpływem stresu związanego z incydentem (patrz „Bezbolesna reakcja na żądanie” w rozdziale 16.). W prawdziwym życiu nie zawsze przestrzega się najlepszych praktyk.

Podobnie przed ponownym zastosowaniem plików manifestu Bob powinien był sprawdzić różnicę przy użyciu polecenia `kubectl diff` (patrz „Śledzenie różnic w zasobach” dalej w tym rozdziale), aby zobaczyć, co się zmieni. Jeśli jednak nie oczekujesz, że coś zostało zmienione, łatwo to przeoczyć. I może Bob nie przeczytał tej książki.

Najlepszym sposobem uniknięcia tego rodzaju problemów jest zawsze wprowadzanie zmian poprzez edycję i stosowanie plików zasobów z systemu kontroli wersji.



Najlepsze praktyki

Nie używaj poleceń imperatywnych `kubectl`, takich jak `create` lub `edit`, w klastrach produkcyjnych. Zamiast tego zawsze zarządzaj zasobami za pomocą manifestów YAML, weryfikowanych przez systemy kontroli wersji, stosowanych razem z poleceniem `kubectl apply` (lub za pomocą wykresów Helm).

Generowanie manifestów zasobów

Mimo że nie zalecamy używania `kubectl` w trybie imperatywnym do wprowadzania zmian w klastrze, polecenia imperatywne mogą znacznie zaoszczędzić czas podczas tworzenia pliku YAML Kubernetes od zera.

Zamiast wpisywać powtarzające się słowa kluczowe do pustego pliku, skorzystaj z `kubectl`, które wygeneruje manifest YAML za Ciebie:

```
kubectl create deployment demo --image=cloudnatived/demo:hello --dry-run=client -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
...
```

Flaga `--dry-run=client` informuje `kubectl`, aby nie tworzył zasobu, a jedynie poinformował, co by utworzył. Flaga `-o yaml` informuje, że utworzony manifest ma być w formacie YAML. Możesz zapisać te dane wyjściowe do pliku, edytować je (jeśli trzeba), a na koniec zastosować do utworzenia zasobu w klastrze:

```
kubectl create deployment demo --image=cloudnatived/demo:hello --dry-run=client -o yaml
>deployment.yaml
```

Teraz, używając swojego ulubionego edytora, wykonaj edycję, zapisz i zastosuj wynik:

```
kubectl apply -f deployment.yaml
deployment.apps/demo created
```

Eksportowanie zasobów

Oprócz pomocy w tworzeniu nowych manifestów zasobów, `kubectl` może również opracowywać pliki manifestów dla zasobów, które już istnieją w klastrze. Przykładowo może utworzyćś Deployment za pomocą polecień imperatywnych (`kubectl create`), zmodyfikowaćś i dostosowaćś je, aby uzyskać

odpowiednie ustawienia, a teraz chcesz dla niego zapisać deklaratywny manifest YAML i umieścić w systemie kontroli wersji.

Aby to zrobić, użyj flagi `-o` razem z `kubectl get`:

```
kubectl create deployment newdemo --image=cloudnative/demo:hello  
deployment.apps/newdemo created  
kubectl get deployments newdemo -o yaml >deployment.yaml
```

Dane wyjściowe będą zawierać dodatkowe informacje, jak np. sekcję status, które możesz bezpiecznie usunąć przed zapisaniem ich razem z innymi manifestami, zaktualizowaniem i zastosowaniem za pomocą polecenia `kubectl apply -f`.

Jeśli do tej pory korzystałeś z imperatywnych poleceń `kubectl` do zarządzania klastrami i chcesz przejść do stylu deklaratywnego, który zalecamy w tej książce, to jest świetny sposób na zrobienie tego. Wyeksportuj wszystkie zasoby w klastrze do plików manifestu za pomocą polecenia `kubectl` z flagą `-o`, jak pokazano w przykładzie, a wszystko będzie gotowe.

Śledzenie różnic w zasobach

Zanim zastosujesz manifesty Kubernetes za pomocą polecenia `kubectl apply`, zobacz dokładnie, co się zmieniło w klastrze. Komenda `kubectl diff Ci` w tym pomoże.

```
kubectl diff -f deployment.yaml  
- replicas: 10  
+ replicas: 5
```

Powyższy wynik może posłużyć Ci do sprawdzenia, czy wprowadzone zmiany rzeczywiście przyniosą oczekiwany efekt. Ostrzeże Cię również, jeśli obecny stan zasobu nie jest zsynchronizowany z manifestem YAML — być może dlatego, że ktoś go edytował imperatywnie.



Najlepsze praktyki

Przed zastosowaniem jakichkolwiek aktualizacji w klastrze produkcyjnym użyj polecenia `kubectl diff`, aby sprawdzić, co się zmieniło.

Praca z kontenerami

Większość tego, co dzieje się w klastrze Kubernetes, dzieje się w kontenerach, więc gdy coś pójdzie nie tak, trudno będzie zobaczyć, co się stało. Oto kilka przydatnych sposobów pracy z uruchomionymi kontenerami za pomocą polecenia `kubectl`.

Przeglądanie dzienników kontenera

Gdy próbujesz uruchomić kontener i nie działa on tak, jak powinien, jednym z najbardziej przydatnych źródeł informacji są dzienniki kontenera. W Kubernetes dziennikiem jest wszystko to, co kontener zapisuje do *standardowych danych wyjściowych* i *standardowych strumieni błędów*; jeśli program był uruchamiany w terminalu, są one wyświetlane w terminalu.

W aplikacjach produkcyjnych, zwłaszcza rozproszonych, musisz mieć możliwość agregowania dzienników z wielu usług, przechowywania ich w trwałe bazie danych oraz wyszukiwania i tworzenia wykresów. To duży temat, którym zajmiemy się bardziej szczegółowo w rozdziale 15.

Sprawdzanie komunikatów dziennika z określonych kontenerów jest jednak nadal bardzo przydatną techniką rozwiązywania problemów; można ją zastosować bezpośrednio za pomocą polecenia `kubectl logs`, z podaną nazwą konkretnego Poda:

```
kubectl logs -n kube-system --tail=20 kube-dns-autoscaler-69c5cbcd-94h7f
autoscaler.go:49] Scaling Namespace: kube-system, Target: deployment/kube-dns
autoscaler server.go:133] ConfigMap not found: configmaps "kube-dns-autoscaler"
k8sclient.go:117] Created ConfigMap kube-dns-autoscaler in namespace kube-system
plugin.go:50] Set control mode to linear
linear_controller.go:59] ConfigMap version change (old: new: 526) - rebuilding
```

Większość długo działających kontenerów generuje dużo danych wyjściowych dziennika, więc zwykle będziesz chciał ograniczyć je do najnowszych wierszy, używając flagi `--tail`, jak w tym przykładzie. (Dzienniki kontenerów będą wyświetlane ze znacznikami czasu, ale wyciągniemy je, żeby zmieścić komunikaty na stronie).

Aby obserwować kontener w trakcie jego działania i przesyłać strumieniowo dane wyjściowe dziennika do terminala, użądź flagę `--follow` (w skrócie `-f`):

```
kubectl logs --namespace kube-system --tail=10 --follow etcd-docker-for-desktop
etcdserver: starting server... [version: 3.1.12, cluster version: 3.1]
embed: ClientTLS: cert = /var/lib/localkube/certs/etcd/server.crt, key = ...
...
```

Tak długo, jak długo pozostawisz uruchomione polecenie `kubectl logs`, będziesz widzieć dane wyjściowe z kontenera `etcd-docker-for-desktop`.

Szczególnie przydatne może być przeglądanie dzienników serwera API Kubernetes; jeśli np. masz błędy uprawnień RBAC (patrz „Kontrola dostępu oparta na rolach (RBAC)” w rozdziale 11.), pojawią się właśnie w tym miejscu. Jeśli masz dostęp do swoich węzłów master, możesz znaleźć Pod `kube-apiserver` w przestrzeni nazw `kube-system` oraz skorzystać z polecenia `kubectl logs`, aby zobaczyć wyniki.

Jeśli korzystasz z usługi zarządzanej, takiej jak GKE, w której nie widzisz węzłów master, sprawdź dokumentację swojego dostawcy, aby dowiedzieć się, jak znaleźć dzienniki warstwy sterowania (np. w GKE będziesz mógł je zobaczyć za pomocą Stackdriver Logs Viewer).



Gdy w Podzie znajduje się wiele kontenerów, możesz określić, dla którego z nich wyświetlać dzienniki za pomocą flagi `--container` (w skrócie `-c`):

```
kubectl logs -n kube-system metrics-server
-c metrics-server-nanny
...
```

W celu bardziej zaawansowanego przeglądania dzienników możesz użyć dedykowanego narzędzia, takiego jak Stern (patrz „Stern” dalej w tym rozdziale).

Przyłączanie do kontenera

Samo oglądanie dzienników kontenera może być niewystarczające. Zamiast tego do kontenera możesz podłączyć lokalny terminal. Pozwala on na bezpośrednią obserwację działania kontenera. Aby to zrobić, użyj polecenia kubectl attach:

```
kubectl attach demo-54f4458547-fcx2n
Defaulting container name to demo.
Use kubectl describe pod/demo-54f4458547-fcx2n to see all of the containers in this pod.
If you don't see a command prompt, try pressing enter.
```

Oglądanie zasobów Kubernetes za pomocą kubespy

Kiedy wdrażasz zmiany w manifestach Kubernetes, zwykle czekasz z niecierpliwością, aby zobaczyć, co będzie dalej.

Często podczas wdrażania aplikacji wiele rzeczy musi się wydarzyć za kulisami, ponieważ Kubernetes tworzy zasoby, przeprowadza operacje na Podach itd.

Ponieważ dzieje się to *automagicznie*, jak lubią mówić inżynierowie, trudno dokładnie powiedzieć, co się dzieje. Polecenia kubectl get i kubectl describe mogą dać migawki poszczególnych zasobów, ale chcielibyśmy zobaczyć sposób zmieniania się stanu zasobów Kubernetes w czasie rzeczywistym.

Skorzystaj z kubespy (<https://github.com/pulumi/kubespy>), przydatnego narzędzia z projektu Pulumi¹. Z jego pomocą możesz oglądać pojedyncze zasoby w klastrze i widzieć, co się z nimi dzieje w czasie.

Jeśli np. w kubespy ustawisz obserwację zasobu Serwis, pokaże on, kiedy usługa jest tworzona, kiedy ma przydzielany adres IP, kiedy podłączają się do niego urządzenia końcowe itd.

Przekierowanie portu kontenera

W punkcie „Uruchamianie aplikacji demonstracyjnej” w rozdziale 2. korzystaliśmy z polecenia kubectl port-forward, aby przekierować Serwis Kubernetes do portu na komputerze lokalnym. Jednak możesz z niego również skorzystać w celu przekierowania portu kontenera, jeśli chcesz połączyć się bezpośrednio z konkretnym Podem. Wystarczy podać nazwę Poda oraz porty lokalne i zdalne:

```
kubectl port-forward demo-54f4458547-vm88z 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::]:9999 -> 8888
```

Teraz port 9999 na Twoim komputerze lokalnym zostanie przekierowany do portu 8888 na kontenerze i możesz np. połączyć się z nim za pomocą przeglądarki internetowej.

¹ Pulumi (<https://www.pulumi.com/>) to framework cloud native, jego infrastruktura ma postać kodu.

Wykonywanie poleceń w kontenerach

Izolowana natura kontenerów jest świetna, gdy chcesz uruchamiać niezawodne, bezpieczne obciążenia. Jednak może być trochę niewygodna, gdy coś nie działa poprawnie i nie możesz zrozumieć, dlaczego.

Gdy uruchamiasz program na komputerze lokalnym i działa on nieprawidłowo, masz do dyspozycji wiersz poleceń: za pomocą polecenia ps możesz zobaczyć uruchomione procesy, za pomocą polecenia ls oraz cat wylistować i wyświetlić zawartość plików, a nawet edytować je za pomocą narzędzia vi.

Bardzo często przy źle działającym kontenerze przydałaby się powłoka działająca w kontenerze, abyśmy mogli przeprowadzić tego rodzaju interaktywne debugowanie.

Za pomocą polecenia kubectl exec możesz uruchomić określone polecenie w dowolnym kontenerze, w tym w powłoce:

```
kubectl run alpine --image alpine --command -- sleep 999
deployment.apps "alpine" created
kubectl get pods
NAME           READY STATUS RESTARTS AGE
alpine-7fd44fc4bf-7gl4n 1/1   Running 0          4s
kubectl exec -it alpine-7fd44fc4bf-7gl4n /bin/sh
/ # ps
PID USER TIME COMMAND
1 root 0:00 sleep 999
7 root 0:00 /bin/sh
11 root 0:00 ps
```

Jeśli Pod zawiera więcej niż jeden kontener, kubectl exec domyślnie uruchomi polecenie w pierwszym kontenerze. Alternatywnie możesz określić kontener za pomocą flagi -c:

```
kubectl exec -it -c container2 POD_NAME /bin/sh
```

(Jeśli kontener nie ma powłoki, patrz „Dodawanie BusyBox do kontenerów” dalej w tym rozdziale).

Rozwiązywanie problemów w kontenerach

Oprócz uruchamiania poleceń na istniejącym kontenerze, czasem jest przydatna możliwość uruchamiania poleceń, takich jak wget lub nslookup, w klastrze, aby sprawdzić działanie aplikacji. Nauczyłeś się już, jak uruchamiać kontenery w klastrze za pomocą polecenia kubectl run. Teraz przedstawimy kilka przykładów uruchamiania jednorazowych poleceń kontenera służących do debugowania.

Najpierw uruchomimy instancję aplikacji demonstracyjnej, aby ją przetestować:

```
kubectl run demo --image cloudnatively/demo:hello --expose --port 8888
service/demo created
pod/demo created
```

Serwis demo powinien mieć przydzielony adres IP i nazwę DNS demo, która jest dostępna wewnętrz klastra. Sprawdźmy to, używając komendy nslookup w kontenerze:

```
kubectl run nslookup --image=busybox:1.28 --rm -it --restart=Never \
--command -- nslookup demo
Server: 10.79.240.10
```

```
Address 1: 10.79.240.10 kube-dns.kube-system.svc.cluster.local
Name: demo
Address 1: 10.79.242.119 demo.default.svc.cluster.local
```

Dobra wiadomość: nazwa DNS działa, więc można wysłać do niej żądanie HTTP za pomocą polecenia wget i zobaczyć wynik:

```
kubectl run wget --image=busybox:1.28 --rm -it --restart=Never \
--command -- wget -qO- http://demo:8888
Hello, 世界
pod "wget" deleted
```

Widac, że polecenie kubectl run używa znanego zestawu flag:

```
kubectl run NAME --image=IMAGE --rm -it --restart=Never --command -- ...
```

Co one robią?

--rm

Flaga informuje Kubernetes, że ma usunąć zasoby stworzone przez to polecenie dla powiązanych kontenerów po zakończeniu działania, aby nie zaśmiecał lokalnej pamięci Twoich węzłów.

-to

Dzięki niej kontener zostanie uruchomiony interaktywnie (i) w terminalu (t), zatem zobaczysz dane wyjściowe z kontenera we własnym terminalu i możesz wysyłać do niego kombinacje klawiszy, jeśli będzie trzeba.

--restart=Never

Informacja dla obiektu Kubernetes, aby pomijał swoje zwykłe zachowanie dotyczące restartowania kontenera przy każdym jego wyjściu. Ponieważ chcemy uruchomić kontener tylko raz, możemy wyłączyć domyślną zasadę restartu.

--command --

Okręśla polecenie do uruchomienia zamiast domyślnego punktu wejścia kontenera. Wszystko, co następuje po --, zostanie przekazane do kontenera jako linia poleceń wraz z argumentami.

Korzystanie z poleceń BusyBox

Chociaż możesz uruchomić dowolny dostępny kontener, obraz *busybox* jest szczególnie przydatny. Zawiera wiele najczęściej używanych komend uniksowych, takich jak cat, echo, find, grep i kill. Pełną listę poleceń BusyBox możesz zobaczyć na stronie internetowej (<https://busybox.net/downloads/BusyBox.html>).

BusyBox zawiera również lekką powłokę podobną do bash, zwaną ash, która jest kompatybilna ze standardowymi skryptami powłoki /bin/sh. Aby uzyskać interaktywną powłokę w klastrze, możesz uruchomić:

```
kubectl run busybox --image=busybox:1.28 --rm -it --restart=Never /bin/sh
```

Ponieważ wzorzec uruchamiania poleceń z obrazu BusyBox jest zawsze taki sam, możesz nawet utworzyć dla niego alias powłoki (zobacz „Aliasy powłoki” w tym rozdziale):

```
alias bb=kubectl run busybox --image=busybox:1.28 --rm -it --restart=Never \
--command --
bb nslookup demo
```

```
...
bb wget -qO- http://demo:8888
...
bb sh
If you don't see a command prompt, try pressing enter.
/ #
```

Dodawanie BusyBox do kontenerów

Jeśli Twój kontener ma już powłokę (bo np. został zbudowany z podstawowego obrazu Linuksa, takiego jak *alpine*), możesz uzyskać dostęp do powłoki w kontenerze, uruchamiając polecenie:

```
kubectl exec -it POD /bin/sh
```

Jeśli jednak nie ma `/bin/sh` w kontenerze? Dzieje się tak, kiedy używasz minimalnego obrazu, co omówiliśmy w „Opis plików Dockerfile” w rozdziale 2.

Najprostszym sposobem na łatwe debugowanie kontenerów, przy zachowaniu bardzo małych obrazów, jest skopiowanie do nich pliku wykonywalnego `busybox` podczas komplikacji. To tylko 1 MiB, czyli niewielka cena za posiadanie użytecznej powłoki i zestawu narzędzi uniksowych.

Z wcześniejszej dyskusji na temat wieloetapowych komplikacji nauczyłeś się, że możesz skopiować plik z wcześniej zbudowanego kontenera do nowego kontenera za pomocą polecenia `Dockerfile COPY --from`. Mniej znaną funkcją tego polecenia jest to, że możesz także skopiować plik z dowolnego obrazu publicznego, a nie tylko tego, który zbudowałeś lokalnie.

Poniższy Dockerfile pokazuje, jak to przeprowadzić z obrazem `demo`:

```
FROM golang:1.14-alpine AS build
WORKDIR /src/
COPY main.go go.* /src/
RUN CGO_ENABLED=0 go build -o /bin/demo
FROM scratch
COPY --from=build /bin/demo /bin/demo
COPY --from=busybox:1.28 /bin/busybox /bin/busybox
ENTRYPOINT ["/bin/demo"]
```

Tutaj `--from=busybox:1.28` odwołuje się do publicznego obrazu biblioteki BusyBox². Możesz skopiować plik z dowolnego obrazu, który Ci się podoba (np. z *alpine*).

Teraz nadal masz bardzo mały kontener, ale możesz również na nim skorzystać z powłoki:

```
kubectl exec -it POD_NAME /bin/busybox sh
```

Zamiast bezpośredniego uruchamiania `/bin/sh`, wykonujesz `/bin/busybox`, a następnie podajesz nazwę żądanego polecenia; w tym przypadku `sh`.

² Wersje obrazu BusyBox późniejsze niż 1.28 mają problem z wyszukiwaniem DNS (<https://github.com/kubernetes/kubernetes/issues/66924>) w aplikacji Kubernetes.

Instalowanie programów w kontenerze

Jeśli potrzebujesz programów, które nie są zawarte w BusyBox lub nie są dostępne w publicznym obrazie kontenera, możesz uruchomić obraz systemu Linux, taki jak *alpine* lub *ubuntu*, i zainstalować na nim wszystko, czego potrzebujesz:

```
kubectl run alpine --image alpine --rm -it --restart=Never /bin/sh  
If you don't see a command prompt, try pressing enter.  
/ # apk --update add emacs
```

Debugowanie w czasie rzeczywistym za pomocą kubesquash

W tym rozdziale rozmawialiśmy nieco luźno o *debugowaniu* kontenerów, aby dowiedzieć się, *co jest z nimi nie tak*. Co jednak zrobić, jeśli chcesz dołączyć prawdziwy debugger, taki jak *gdb* (debugger projektu GNU) lub *d1v* (debugger Go), do jednego z uruchomionych procesów w kontenerze?

Debugger, taki jak *d1v*, jest potężnym narzędziem, które może dołączyć do procesu, pokazać, które wiersze kodu źródłowego są wykonywane, sprawdzić i zmienić wartości zmiennych lokalnych, ustawić punkty przerwania i przechodzić program linia po linii. Jeśli dzieje się coś tajemniczego, czego nie możesz zrozumieć, prawdopodobnie w końcu będziesz musiał skorzystać z debugera.

Gdy uruchamiasz program na komputerze lokalnym, masz bezpośredni dostęp do jego procesów, więc nie stanowi to problemu. Jeśli problem wystąpi w kontenerze, to jak w przypadku większości rzeczy, sprawa się komplikuje.

Narzędzie *kubesquash* zostało zaprojektowane, by pomóc Ci dołączyć debugger do kontenera. Aby je zainstalować, postępuj zgodnie z instrukcjami (<https://github.com/solo-io/kubesquash>) na GitHub.

Po zainstalowaniu *kubesquash* wystarczy przekazać mu nazwę działającego kontenera:

```
/usr/local/bin/kubesquash-osx demo-6d7dff895c-x8pf  
? Going to attach d1v to pod demo-6d7dff895c-x8pf. continue? Yes  
If you don't see a command prompt, try pressing enter.  
(d1v)
```

Pod spodem *kubesquash* tworzy Pod w przestrzeni nazw *squash*, który uruchamia debugger i dba o dołączenie go do uruchomionego procesu w podanym Podzie.

Ze względów technicznych (<https://github.com/solo-io/kubesquash/blob/master/cmd/kubesquash/main.go#L13>) *kubesquash* polega na poleceniu *ls* dostępnym w badanym kontenerze. Jeśli go nie masz, możesz skorzystać z BusyBox, tak jak zrobiliśmy w punkcie „*Dodawanie BusyBox do kontenerów*”:

```
COPY --from=busybox:1.28 /bin/busybox /bin/ls
```

Zamiast kopować plik wykonywalny do */bin/busybox*, kopujemy go do */bin/ls*. Dzięki temu *kubesquash* działa idealnie.

Nie zajmiemy się teraz szczegółowo korzystania z *d1v*, ale jeśli piszesz aplikacje Kubernetes w Go, jest to nieocenione narzędzie, a *kubesquash* bardzo dobrze współpracuje z kontenerami.

Więcej na temat *d1v* możesz przeczytać w oficjalnej dokumentacji (<https://github.com/derekparker/delve/tree/master/Documentation>).

Konteksty i przestrzenie nazw

Do tej pory w tej książce pracowaliśmy z jednym klastrem Kubernetes, a wszystkie uruchomione przez Ciebie polecenia `kubectl` naturalnie stosowały się do tego klastra.

Co się dzieje, gdy masz więcej niż jeden kластer? Może np. masz kластer Kubernetes na swoim komputerze do testowania lokalnego i kластer produkcyjny w chmurze, a może inny kластer zdalny do testowania i programowania. Jak poinformować `kubectl`, który masz na myśli?

Aby rozwiązać ten problem, `kubectl` stosuje pojęcie *kontekstów* (ang. *contexts*). Kontekst jest kombinacją klastra, użytkownika i przestrzeni nazw (patrz „Korzystanie z przestrzeni nazw” w rozdziale 5.).

Polecenia `kubectl` po uruchomieniu są zawsze wykonywane w *biejącym kontekście*.

Oto przykład:

```
kubectl config get-contexts
CURRENT NAME          CLUSTER      AUTHINFO      NAMESPACE
          gke           gke_test_us-w  gke_test_us  myapp
*        docker-for-desktop docker-for-d  docker-for-d
```

Są to konteksty, o których `kubectl` coś wie. Każdy kontekst ma nazwę i odnosi się do określonego klastra, nazwy użytkownika uwierzytelnionego w klastrze oraz przestrzeni nazw w klastrze. Kontekst `docker-for-desktop`, jak można się spodziewać, odnosi się do mojego lokalnego klastra Kubernetes.

Bieżący kontekst jest wyświetlany ze znakiem * w pierwszej kolumnie (w przykładzie jest to `docker-for-desktop`). Jeśli teraz uruchomię polecenie `kubectl`, będzie ono działać w klastrze Docker Desktop w domyślnej przestrzeni nazw (ponieważ kolumna `NAMESPACE` jest pusta, co wskazuje, że kontekst odwołuje się do domyślnej przestrzeni nazw):

```
kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/...
```

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

Możesz przejść do innego kontekstu za pomocą komendy `kubectl config use-context`:

```
kubectl config use-context gke
Switched to context "gke".
```

Można myśleć o kontekstach jak o zakładkach: pozwalają one łatwo przełączać się na określony klaster i określoną przestrzeń nazw. Aby utworzyć nowy kontekst, użyj polecenia `kubectl config set-context`:

```
kubectl config set-context myapp --cluster=gke --namespace=myapp
Context "myapp" created.
```

Teraz, za każdym razem gdy przełączysz się na kontekst `myapp`, Twoim bieżącym kontekstem będzie przestrzeń nazw `myapp` w klastrze Docker Desktop.

Jeśli zapomnisz, jaki jest Twój obecny kontekst, `kubectl config current-context` Ci podpowie:

```
kubectl config current-context
myapp
```

kubectx i kubens

Jeśli, podobnie jak my, zarabiasz pisaniem na życie, prawdopodobnie nie lubisz pisać więcej, niż musisz. Aby szybciej przełączać konteksty kubectl, możesz użyć narzędzi kubectx i kubens. Postępuj zgodnie z instrukcjami (<https://github.com/ahmetb/kubectx>) na Git-Hub, aby zainstalować zarówno kubectx, jak i kubens.

Teraz możesz używać polecenia kubectx do przełączania kontekstów:

```
kubectx docker-for-desktop
Switched to context "docker-for-desktop".
```

Jedną fajną cechą kubectx jest to, że narzędzie to ma możliwość przełączenia się do poprzedniego kontekstu, dzięki czemu możesz szybko przełączać się między dwoma kontekstami:

```
kubectx -
Switched to context "gke".
kubectx -
Switched to context "docker-for-desktop".
```

Po prostu kubectx samodzielnie wyświetli listę wszystkich kontekstów, które zapisałś, z zaznaczonym bieżącym kontekstem.

Przełączanie przestrzeni nazw jest czymś, czego prawdopodobnie będziesz używać częściej niż przełączania kontekstów, więc narzędzie kubens jest do tego idealne:

```
kubens
default
kube-public
kube-system
kubens kube-system
Context "docker-for-desktop" modified.
Active namespace is "kube-system".
kubens -
Context "docker-for-desktop" modified.
Active namespace is "default".
```



Narzędzia kubectx i kubens robią jedną rzecz, ale są bardzo przydatnymi dodatkami do zestawu narzędzi Kubernetes.

kube-ps1

Jeśli używasz powłoki bash lub zsh, istnieje małe narzędzie (<https://github.com/jonmosco/kube-ps1>), które doda bieżący kontekst Kubernetes do Twojej ścieżki.

Po zainstalowaniu kube-ps1 nie zapomnisz, w jakim kontekście pracujesz:

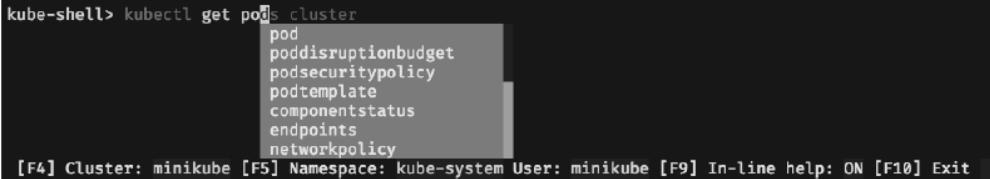
```
source "/usr/local/opt/kube-ps1/share/kube-ps1.sh"
PS1="[$(kube_ps1)]$ "
[(_ | docker-for-desktop:default)]
kubectx cloudnativedevops
Switched to context "cloudnativedevops".
(_ |cloudnativedevops:cloudnativedevopsblog)
```

Powłoki i narzędzia Kubernetes

Korzystanie z kubectl w zwyczajnej powłoce jest całkowicie wystarczające do większości rzeczy, które chcesz zrobić z klastrzem Kubernetes, jednak istnieją inne opcje.

kube-shell

Jeśli autouzupełnianie kubectl nie jest dla Ciebie wystarczające, możesz skorzystać z nakładki dla kubectl — kube-shell, która w okienku wyświetla możliwe uzupełnienia dla każdego polecenia (patrz rysunek 7.1).



The screenshot shows a terminal window titled 'kube-shell'. The command 'kubectl get pods cluster' is being typed. A dropdown menu is open over the word 'cluster', listing several Kubernetes resources: 'pod', 'poddisruptionbudget', 'podsecuritypolicy', 'podtemplate', 'componentstatus', 'endpoints', and 'networkpolicy'. At the bottom of the screen, there is status information: '[F4] Cluster: minikube [F5] Namespace: kube-system User: minikube [F9] In-line help: ON [F10] Exit'.

Rysunek 7.1. kube-shell to interaktywny klient Kubernetes

Click

Więcej opcji podczas korzystania z terminala Kubernetes zapewnia Click (<https://databricks.com/blog/2018/03/27/introducing-click-the-command-line-interactive-controller-for-kubernetes.html>).

Click działa jak interaktywna wersja kubectl zapamiętującą bieżący obiekt, z którym pracujesz. Jeśli np. chcesz znaleźć dany Pod w kubectl, zwykle musisz najpierw wyświetlić wszystkie pasujące Pody, a następnie skopiować i wkleić unikalną nazwę Poda, którym jesteś zainteresowany, do nowego polecenia.

Z pomocą Click możesz wybrać dowolny zasób z listy, wpisując jego numer (np. 1 dla pierwszego elementu). To jest teraz bieżący zasób, a następne polecenie Click domyślnie będzie działać na tym zasobie. Aby ułatwić znalezienie żądanego obiektu, Click obsługuje wyszukiwanie według wyrażeń regularnych.

Click to użyteczne narzędzie, które zapewnia bardzo przyjemne środowisko do pracy z Kubernetes. Chociaż jest opisane jako *beta i eksperymentalne*, jest już doskonale przydatne do codziennych zadań związanych z administrowaniem klastrami i warto je wypróbować.

kubed-sh

Podczas gdy kube-shell i Click zapewniają lokalne powłoki, które trochę wiedzą o Kubernetes, kubed-sh (wymawiane jako *kube-dash*) jest bardziej intrygującym pomysłem: w pewnym sensie powłoka działa na samym klastrze.

kubed-sh pobierze i uruchomi kontenery, niezbędne do działania języków JavaScript, Ruby lub Python w bieżącym klastrze. Możesz np. utworzyć skrypt Ruby na komputerze lokalnym i użyć kubed-sh, aby wykonać skrypt jako Deployment Kubernetes.

Stern

Chociaż kubectl logs jest przydatnym poleceniem (patrz „Przeglądanie dzienników kontenera” wcześniej w tym rozdziale), nie jest aż tak wygodne, jak mogłyby być. Przykładowo zanim będzie można go użyć, musisz najpierw znaleźć unikalną nazwę Poda i kontenera, którego dzienniki chcesz wyświetlić, i podać je w wierszu polecenia, co zwykle oznacza co najmniej jedną operację typu kopij i wklej.

Jeśli ponadto używasz opcji -f do śledzenia dzienników z określonego kontenera, za każdym razem gdy kontener zostanie zrestartowany, strumień dziennika zostanie zatrzymany. Musisz znaleźć nową nazwę kontenera i ponownie uruchomić polecenie kubectl logs. Dodatkowo możesz śledzić dzienniki tylko z jednego Poda jednocześnie.

Bardziej wyrafinowane narzędzie do śledzenia dzienników pozwala określić grupę Podów za pomocą wyrażenia regularnego czy zestawu etykiet, a dodatkowo obserwacja dzienników nie jest zakłócona nawet po ponownym uruchomieniu poszczególnych kontenerów.

Umożliwia to narzędzie Stern (<https://github.com/wercker/stern>). Stern śledzi dzienniki ze wszystkich Podów, których nazwa pasuje do wyrażenia regularnego (np. demo.*). Jeśli Pod zawiera wiele kontenerów, Stern pokaże dla każdego z nich osobne dzienniki, poprzedzone ich nazwą.

Flaga --since pozwala ograniczyć dane wyjściowe do ostatnich wiadomości (przykładowo do ostatnich 10 minut).

Zamiast dopasowywać określone nazwy Podów za pomocą wyrażenia regularnego, możesz użyć dowolnego selektora etykiety Kubernetes, tak jak w przypadku kubectl. W połączeniu z flagą --allnamespaces jest to idealne rozwiązanie do oglądania dzienników z wielu kontenerów.

Budowanie własnych narzędzi Kubernetes

W połączeniu z narzędziami do zapytań, takimi jak jq, i standardowym zestawem narzędzi uniwersyjnych (cut, grep, xargs itp.) kubectl może być używany do tworzenia dość skomplikowanych skryptów związanych z zasobami Kubernetes. Jak pokazaliśmy w tym rozdziale, istnieje również wiele narzędzi innych firm, z których można korzystać w ramach automatyzacji skryptów.

To podejście ma jednak swoje ograniczenia. Można tworzyć genialne skrypty jednowierszowe oraz skrypty powłoki do interaktywnego debugowania i eksploracji, ale mogą być one trudne do zrozumienia i utrzymania.

W przypadku prawdziwych programów systemowych, automatyzujących procesy produkcyjne, zdecydowanie zalecamy używanie prawdziwego języka programowania. Go jest logicznym wyborem, ponieważ był to wystarczająco dobry język dla autorów Kubernetes oraz zawiera w pełni funkcjonalną bibliotekę kliencką (<https://github.com/kubernetes/client-go>) do użytku w programach Go.

Ponieważ biblioteka client-go zapewnia pełny dostęp do interfejsu API Kubernetes, możesz w nim zrobić wszystko, co potrafi kubectl, i wiele więcej. Poniższy fragment pokazuje, jak wyświetlić listę wszystkich Podów w klastrze:

```
...
podList, err := clientset.CoreV1().Pods("").List metav1.ListOptions{}
if err != nil {
    log.Fatal(err)
}
fmt.Println("There are", len(podList.Items), "pods in the cluster:")
for _, i := range podList.Items {
    fmt.Println(i.ObjectMeta.Name)
}
...
```

Możesz także tworzyć lub usuwać Pody, obiekty Deployment lub dowolne inne zasoby. Możesz nawet zaimplementować własne niestandardowe typy zasobów.

Jeśli potrzebujesz funkcji, której brakuje w Kubernetes, możesz ją zaimplementować samodzielnie, korzystając z tej biblioteki.

Inne języki programowania, takie jak Ruby, Python i PHP, również mają swoje biblioteki w Kubernetes (<https://kubernetes.io/docs/reference/using-api/client-libraries/>), których można używać w ten sam sposób.

Podsumowanie

Dostępna jest oszałamiająca liczba narzędzi Kubernetes, a co tydzień publikowane są kolejne. Możesz czuć się trochę zmęczony, czytając o kolejnym nowym narzędziu, bez którego najwyraźniej nie możesz się obejść.

Faktem jest, że nie potrzebujesz większości z tych narzędzi. Sam Kubernetes przy użyciu kubectl może zrobić wszystko, co chcesz. Reszta jest po prostu dla zabawy i wygody.

Nikt nie wie wszystkiego, ale każdy coś wie. Pisząc ten rozdział, dodaliśmy porady i wskazówki wielu doświadczonych inżynierów Kubernetes, z książek, postów na blogu i dokumentacji oraz jednego lub dwóch własnych małych doświadczeń. Wszyscy, którym to pokazaliśmy, bez względu na to, jakimi są ekspertami, nauczyli się co najmniej jednej przydatnej rzeczy. To nas cieszy.

Warto poświęcić trochę czasu na zapoznanie się z kubectl i zbadanie jego możliwości; to najważniejsze narzędzie Kubernetes, które posiadasz, i będziesz z niego często korzystać.

Oto kilka najważniejszych rzeczy, o których warto wiedzieć.

- kubectl zawiera kompletną i wyczerpującą dokumentację, dostępną za pomocą polecenia kubectl -h, a o każdym zasobie, polu lub funkcji Kubernetes, za pomocą polecenia kubectl explain.
- Jeśli chcesz wykonać skomplikowane filtrowanie i transformacje na wyniku wyjściowym polecenia kubectl, np. w skryptach, wybierz format JSON z opcją -o json; do obróki danych w formacie JSON możesz wykorzystać narzędzia, takie jak jq (do wysyłania zapytań).

- Opcja `kubectl --dry-run=client`, w połączeniu z `-o YAML` (aby uzyskać format YAML na wyjściu), pozwala używać poleceń imperatywnych do generowania manifestów Kubernetes; podczas tworzenia plików manifestów dla nowych aplikacji zyskujemy dużą oszczędność czasu.
- Można również przekształcić istniejące zasoby do manifestów YAML, używając flagi `-o` w poleceniu `kubectl get`.
- Polecenie `kubectl diff` powie Ci, co by się zmieniło, gdybyś zastosował manifest, bez faktycznej jego zmiany.
- Za pomocą polecenia `kubectl logs` możesz zobaczyć komunikaty wyjściowe i komunikaty o błędach dla dowolnego kontenera, przesyłać je strumieniowo w sposób ciągły za pomocą flagi `--follow` lub śledzić dzienniki wielu Podów przy użyciu narzędzia Stern.
- Aby rozwiązać problemy z kontenerami, możesz się do nich przyłączyć za pomocą polecenia `kubectl attach` lub uzyskać dostęp do powłoki w kontenerze za pomocą polecenia `kubectl exec -it ... /bin/sh`.
- Aby rozwiązać problemy, możesz uruchomić dowolny publiczny obraz kontenera za pomocą narzędzia `kubectl run` — w tym narzędzia BusyBox, które zawiera wszystkie Twoje ulubione polecenia uniksowe.
- Konteksty Kubernetes są jak zakładki, oznaczające Twoje miejsce w określonym klastrze i przestrzeni nazw; możesz wygodnie przełączać się między kontekstami i przestrzeniami nazw za pomocą narzędzi `kubectx` i `kubens`.
- Click to potężna powłoka Kubernetes, która zapewnia funkcjonalność `kubectl`, ale z dodanym stanem: zapamiętuje aktualnie wybrany obiekt z jednego polecenia i przenosi do drugiego, więc nie musisz go podawać za każdym razem.
- Kubernetes został zaprojektowany do automatyzacji i kontrolowania go za pomocą kodu. Gdy potrzebujesz wyjść poza to, co zapewnia `kubectl`, biblioteka `client-go` Kubernetes daje Ci pełną kontrolę nad każdym aspektem klastra za pomocą języka Go.

Uruchamianie kontenerów

Jeśli masz trudne pytanie, na które nie możesz odpowiedzieć, zacznij od odpowiedzi na prostsze pytania, na które nie możesz odpowiedzieć.

— Max Tegmark

W poprzednich rozdziałach koncentrowaliśmy się głównie na aspektach operacyjnych Kubernetes, czyli gdzie zdobyć klastry, jak je utrzymywać i jak zarządzać zasobami klastra. Przejdziemy teraz do najbardziej podstawowego obiektu Kubernetes, do *kontenera*. Przyjrzymy się, jak kontenery działają na poziomie technicznym, jak powiązane są z Podami i jak wdrażać obrazy kontenerów do Kubernetes.

W tym rozdziale zajmiemy się również ważnym tematem dotyczącym bezpieczeństwa kontenerów oraz tego, jak używać funkcji bezpieczeństwa w Kubernetes do bezpiecznego wdrażania aplikacji zgodnie z najlepszymi praktykami. Na koniec przyjrzymy się, jak montować woluminy dyskowe w Podach, umożliwiając kontenerom udostępnianie i utrwalanie danych.

Kontenery i Pody

W rozdziale 2. wprowadziliśmy pojęcie Podów i wyjaśniliśmy, w jaki sposób obiekty Deployment wykorzystują obiekty ReplicaSet do utrzymania zestawu replik Podów, ale nie przyjrzaliśmy się dokładnie samym Podom. Pody są jednostkami zaplanowanymi do uruchamiania w Kubernetes. Obiekt Pod reprezentuje kontener lub grupę kontenerów, a wszystko, co działa w Kubernetes, zależy od Podów.

Obiekt Pod reprezentuje kolekcję kontenerów aplikacji i woluminów działających w tym samym środowisku wykonawczym. Pody, a nie kontenery, to najmniejsze artefakty, które można wdrożyć w klastrze Kubernetes. Oznacza to, że wszystkie kontenery zawarte w Podzie zawsze lądują na tej samej maszynie.

— Kelsey Hightower i inni, *Tworzenie niezawodnych systemów rozproszonych*

Do tej pory w książce terminy Pod i kontener były używane mniej lub bardziej zamienne — Pod aplikacji demo zawiera tylko jeden kontener. Jednak w bardziej złożonych aplikacjach w Podach znajdziemy dwa kontenery lub wiele kontenerów. Zobaczmy więc, jak to działa, oraz kiedy i dlaczego warto zgrupować kontenery w Podach.

Co to jest kontener?

Zanim wyjaśnimy, dlaczego warto mieć wiele kontenerów w Podzie, poświęcimy chwilę, aby ponownie sprawdzić, czym właściwie jest kontener.

W podrozdziale „Nadejście kontenerów” w rozdziale 1. napisaliśmy, że kontener to znormalizowany pakiet zawierający oprogramowanie wraz z zależnościami, konfiguracją, danymi itd., czyli wszystko, czego potrzebuje do uruchomienia. Jak to jednak działa?

W Linuksie i większości innych systemów operacyjnych wszystko, co działa na maszynie, nosi nazwę *procesu*. Proces reprezentuje kod binarny i stan pamięci uruchomionej aplikacji, takiej jak Chrome, iTunes lub Visual Studio Code. Wszystkie procesy istnieją w tej samej globalnej przestrzeni nazw; tzn. że wszystkie mogą widzieć inne i współdziałać z nimi, wszystkie współużytkują tę samą pulę zasobów, takich jak procesor, pamięć i system plików. (Przestrzeń nazw Linux jest trochę podobna do przestrzeni Kubernetes, choć technicznie nie jest taka sama).

Z punktu widzenia systemu operacyjnego, kontener reprezentuje izolowany proces (lub grupę procesów), który istnieje we własnej przestrzeni nazw. Procesy wewnętrz kontenera nie widzą procesów poza nim i vice versa. Kontener nie może uzyskać dostępu do zasobów należących do innego kontenera ani przetwarzają zasobów poza kontenerem. Granica kontenera jest jak ogrodzenie, które zatrzymuje wszystkie inne procesy.

Proces wewnętrz kontenera działa na własnej maszynie, z pełnym dostępem do wszystkich swoich zasobów i nie ma tam uruchomionych żadnych innych procesów. Możesz to zobaczyć, jeśli uruchomisz kilka poleceń w kontenerze:

```
kubectl run busybox --image busybox:1.28 --rm -it --restart=Never /bin/sh
If you don't see a command prompt, try pressing enter.
/ # ps ax
PID USER TIME COMMAND
1  root 0:00 /bin/sh
8  root 0:00 ps ax
/ # hostname
busybox
```

Zwykłe polecenie ps ax wyświetla listę wszystkich procesów uruchomionych na komputerze, a jest ich dużo (kilka set na typowym serwerze Linux). Jednak tutaj mamy tylko dwa procesy — /bin/sh i ps ax. Dlatego jedynymi procesami widocznymi w kontenerze są procesy faktycznie uruchomione w kontenerze.

Podobnie polecenie hostname, które normalnie wyświetla nazwę hosta, zwraca busybox — w rzeczywistości jest to nazwa kontenera. Polecenie zagląda do kontenera busybox, tak jakby to była maszyna o nazwie busybox. Dotyczy to każdego z kontenerów działających na tej samej maszynie.



Zabawne jest samodzielne tworzenie kontenera, bez korzyści wynikających z środowiska uruchomieniowego, takiego jak Docker. Doskonała rozmowa Liz Rice na temat „Co to właściwie jest kontener?” (<https://youtu.be/HPuvDm8IC-4>) pokazuje, jak to zrobić od zera w programie Go.

Co należy do kontenera?

Nie ma technicznych powodów, dla których nie możesz uruchomić tylu procesów, ile chcesz, wewnątrz kontenera: możesz uruchomić pełną dystrybucję Linuksa z wieloma uruchomionymi aplikacjami, usługami sieciowymi itd., wszystko w tym samym kontenerze. Dlatego czasami słyszysz, że kontenery zwane są *lekkimi maszynami wirtualnymi* (ang. *lightweight virtual machines*). Jednak nie jest to najlepszy sposób korzystania z kontenerów, ponieważ wtedy nie zyskujesz korzyści związanych z izolacją zasobów.

Jeśli procesy nie powinny o sobie nic wiedzieć, nie muszą działać w tym samym kontenerze. Dobrą zasadą dotyczącą kontenera jest to, że powinien on robić *jedną rzecz*. Przykładowo nasz kontener z aplikacją demo nasłuchiwał na porcie sieciowym i wysyłała ciąg Hello, 世界 do każdego, kto się z nim łączy. To prosta, samodzielna usługa: nie opiera się na żadnych innych programach ani usługach oraz nie jest od niczego zależna. Jest idealnym kandydatem do posiadania własnego kontenera.

Kontener ma również *punkt wejścia* (ang. *entrypoint*), czyli polecenie uruchamiane podczas startu kontenera. Zwykle powoduje to utworzenie jednego procesu do uruchomienia polecenia, chociaż niektóre aplikacje często uruchamiają kilka podprocesów, które działają jako pomocnicy lub pracownicy. Aby uruchomić wiele oddzielnych procesów w kontenerze, musisz napisać skrypt, który będzie działał jako punkt wejścia i uruchomi pożądane procesy.



Każdy kontener powinien uruchamiać tylko jeden główny proces. Jeśli prowadzisz dużą grupę niepowiązanych procesów w kontenerze, nie wykorzystujesz w pełni mocy kontenerów i powinieneś pomyśleć o podzieleniu aplikacji na wiele komunikujących się kontenerów.

Co należy do Poda?

Teraz, gdy wiesz, co to jest kontener, możesz zobaczyć, dlaczego warto grupować je w Podach. Pod reprezentuje grupę kontenerów, które muszą się komunikować i udostępniać sobie dane; muszą być planowane razem, muszą być uruchamiane i zatrzymywane razem oraz muszą działać na tej samej maszynie fizycznej.

Dobrym przykładem jest aplikacja, która przechowuje dane w lokalnej pamięci podręcznej, takiej jak Memcached (<https://memcached.org/about>). Musisz uruchomić dwa procesy — aplikację i proces serwera *memcached*, który obsługuje przechowywanie i pobieranie danych. Chociaż możesz uruchomić oba procesy w jednym kontenerze, nie jest to konieczne: muszą komunikować się tylko przez gniazdo sieciowe. Lepiej podzielić je na dwa osobne kontenery, z których każdy musi się martwić jedynie o zbudowanie i uruchomienie własnego procesu.

W rzeczywistości możesz użyć publicznego obrazu kontenera Memcached, dostępnego w Docker Hub, który jest już skonfigurowany do pracy jako część Poda z innym kontenerem.

Tworzysz Pod z dwoma kontenerami, Memcached i swoją aplikacją. Aplikacja może rozmawiać z Memcached, nawiązując połączenie sieciowe, a ponieważ dwa kontenery znajdują się w tym samym Podzie, połączenie to zawsze będzie lokalne, dwa kontenery będą zawsze działać w tym samym węźle.

Teraz wyobraź sobie aplikację blogową, która składa się z kontenera serwera WWW, takiego jak Nginx, i kontenera synchronizatora Git, który klonuje repozytorium Git zawierające dane blogu, takie jak pliki HTML, obrazki itd. Kontener blogu zapisuje dane na dysk, ponieważ jednak kontenery w Podzie mogą współużytkować wolumin dysku, dane mogą być również dostępne dla kontenera Nginx — do obsługi HTTP.

Ogólnie rzecz biorąc, właściwe pytanie, które należy sobie zadać podczas projektowania Podów, brzmi: „Czy te kontenery będą działać poprawnie, kiedy wylądują na różnych maszynach?”. Jeśli odpowiedź brzmi: „Nie”, Pod zapewni prawidłowe grupowanie kontenerów. Jeśli odpowiedź brzmi „Tak”, zastosowanie wielu Podów jest tutaj prawdopodobnie właściwym rozwiązańiem.

— Kelsey Hightower i inni, *Tworzenie niezawodnych systemów rozproszonych*

Wszystkie kontenery w Podzie powinny współpracować, aby wykonać jedną pracę. Jeśli potrzebujesz tylko jednego kontenera do wykonania tej pracy, użyj jednego kontenera. Jeśli potrzebujesz dwóch lub trzech, w porządku. Jeśli masz więcej, możesz zastanowić się, czy kontenery można podzielić na osobne Pody.

Manifesty kontenera

Opowiedzieliśmy, czym są kontenery, co ma być w kontenerze i kiedy kontenery powinny być zgrupowane w Pody. Jak więc faktycznie uruchomić kontener w Kubernetes?

Podczas tworzenia pierwszego obiektu Deployment w punkcie „Manifesty obiektu Deployment” w rozdziale 4. powstała sekcja `template.spec` określająca kontener do uruchomienia (w tym przykładzie tylko jeden kontener):

```
spec:  
  containers:  
    - name: demo  
      image: cloudnativized/demo:hello  
      ports:  
        - containerPort: 8888
```

Oto przykład, jak wyglądałaby sekcja `template.spec` dla obiektu Deployment z dwoma kontenerami:

```
spec:  
  containers:  
    - name: container1  
      image: example/container1  
    - name: container2  
      image: example/container2
```

Jednymi wymaganymi polami w specyfikacji każdego kontenera są pola `name` i `image`; kontener musi mieć nazwę, aby inne zasoby mogły się do niego odwoływać, a Ty musisz poinformować Kubernetes, jaki obraz ma zostać uruchomiony w kontenerze.

Identyfikatory obrazu

Do tej pory używałeś już różnych identyfikatorów obrazów kontenerów, np. `cloudnativized /demo:hello`, `alpine` i `busybox:1.28`.

Identyfikator obrazu składa się z czterech części; są to *nazwa hosta rejestru* (ang. *registry hostname*), *przestrzeń nazw repozytorium* (ang. *repository namespace*), *repozytorium obrazów* (ang. *image repository*) i *tag*. Wszystkie części, oprócz nazwy obrazu, są opcjonalne. Identyfikator obrazu wykorzystujący wszystkie te części wygląda następująco:

`docker.io/cloudnativized/demo:hello`

- Nazwa hosta rejestru w tym przykładzie to `docker.io`; w rzeczywistości jest to ustawienie domyślne dla obrazów Docker, więc nie musimy go określać. Jeśli Twój obraz jest przechowywany w innym rejestrze, musisz podać jego nazwę hosta. Przykładowo obrazy rejestru kontenera Google są poprzedzane przez `gcr.io`.
- Przestrzeń nazw repozytorium to `cloudnativized`: to my (cześc!). Jeśli nie określisz przestrzeni nazw repozytorium, zostanie użyta domyślna przestrzeń nazw (zwana biblioteką). Jest to zestaw oficjalnych obrazów (https://docs.docker.com/docker-hub/official_repos), które są zatwierdzone i utrzymywane przez Docker, Inc. Popularne oficjalne obrazy obejmują obrazy podstawowe systemu operacyjnego (`alpine`, `ubuntu`, `debian`, `centos`), środowiska językowe (`golang`, `python`, `ruby`, `php`, `java`) i powszechnie używane oprogramowanie (`mongo`, `mysql`, `nginx`, `redis`).
- Repozytorium obrazów to `demo`. Identyfikuje konkretny obraz kontenera w rejestrze i przestrzeni nazw. (Patrz też „Skróty kontenerów” dalej w tym rozdziale).
- Tag to `hello`. Tagi identyfikują różne wersje tego samego obrazu.

Od Ciebie zależy, jakie tagi umieścić w kontenerze. Niektóre typowe opcje obejmują:

- tag z nazwą wersji, taką jak `v1.3.0`; zwykle odnosi się to do wersji aplikacji,
- tag Git SHA, taki jak `5ba6bfd ...`; identyfikuje specyficzny commit w repozytorium źródłowym, który został użyty do zbudowania kontenera (patrz „Tagi Git SHA” w rozdziale 14.),
- środowisko, które reprezentuje, takie jak środowisko programistyczne lub produkcyjne.

Do danego obrazu możesz dodać dowolną liczbę tagów.

Tag latest

Jeśli nie określisz tagu podczas pobierania obrazu, domyślnym tagiem jest ten najnowszy. Gdy np. uruchomisz obraz `alpine` bez określonego tagu, otrzymasz `alpine:latest`.

Tag `latest` to domyślny tag, który jest dodawany do obrazu podczas jego tworzenia lub umieszczania bez określonego tagu. To niekoniecznie identyfikuje najnowszy obraz; to tylko najnowszy obraz, który nie został wyraźnie oznaczony, co sprawia, że tag `latest` jest raczej nieprzydatny (<https://vsupalov.com/docker-latest-tag>) jako identyfikator.

Dlatego ważne jest, aby zawsze używać określonego tagu podczas wdrażania kontenerów produkcyjnych na Kubernetes. Gdy uruchamiasz szybki, jednorazowy kontener, aby rozwiązać problem lub eksperyment (np. kontener `alpine`), możesz pominąć tag i pobrać najnowszy obraz. Jednak

w przypadku prawdziwych aplikacji chcesz mieć pewność, że jeśli jutro wdrożysz Pod, otrzymasz dokładnie taki sam obraz kontenera, jak przy dzisiejszym wdrożeniu.

Należy unikać używania tagu latest podczas wdrażania kontenerów w produkcji, ponieważ utrudnia to śledzenie, która wersja obrazu jest uruchomiona, i przywracanie.

— dokumentacja Kubernetes

(<https://kubernetes.io/docs/concepts/configuration/overview/#using-labels>)

Skróty kontenerów

Jak widzisz, tag `latest` nie zawsze oznacza to, co myślisz. Wersja semantyczna lub tag Git SHA również niejednoznacznie określają konkretny obraz kontenera. Jeśli twórca obrazu zdecyduje się przesyłać inny obraz z tym samym tagiem, przy następnym wdrożeniu otrzymasz ten zaktualizowany obraz. Z technicznego punktu widzenia tag jest *niedeterministyczny*.

Czasami pożąданie są deterministyczne wdrożenia; innymi słowy, są one po to, aby zagwarantować, że wdrożenie zawsze będzie odwoływać się do dokładnie określonego obrazu kontenera. Możesz to zrobić, korzystając ze *skrótu* (ang. *digest*) kontenera; jest to kryptograficzny skrót zawartości obrazu, który niezmiennie identyfikuje ten obraz.

Obrazy mogą mieć wiele tagów, ale tylko jeden skrót. Oznacza to, że jeśli manifest kontenera określa skrót obrazu, możesz zagwarantować deterministyczne wdrożenia. Identyfikator obrazu ze skrótem wygląda następująco:

```
cloudnatted /  
demo@sha256:aeae1e551a6cbd60bcfd56c3b4ffec732c45b8012b7cb758c6c4a34 ...
```

Podstawowe tagi obrazu

Gdy odwołujesz się do obrazu podstawowego w Dockerfile i nie określisz tagu, otrzymasz ten z tagiem `latest`, podobnie jak podczas wdrażania kontenera. Ze względu na trudną semantykę tagu `latest`, jak widzieliśmy, dobrym pomysłem jest użycie określonego podstawowego tagu obrazu, takiego jak `alpine:3.8`.

Po wprowadzeniu zmiany w aplikacji i przebudowaniu jej kontenera nie chcesz także otrzymywać nieoczekiwanych zmian w wyniku pobrania nowszego publicznego podstawowego obrazu. Może to powodować problemy, które są trudne do znalezienia i debugowania.

Aby komplikacje były jak najbardziej odtwarzalne, użyj określonego tagu lub skrótu.



Powiedzieliśmy, że należy unikać używania tagu `latest`, ale trzeba zauważyc istnienie kilku kwestii spornych. Nawet obecni autorzy mają różne preferencje. Używanie obrazów podstawowych z tagiem `latest` oznacza, że jeśli jakaś zmiana obrazu podstawowego spowoduje problem z Twoją komplikacją, przekonasz się o tym od razu. Z drugiej strony, użycie określonych tagów obrazu oznacza, że musisz aktualnić obraz podstawowy tylko wtedy, gdy chcesz, a nie wtedy, gdy zdecydują się na to inni twórcy. To zależy od Ciebie.

Porty

Zapoznałeś się już polem ports, używanym w naszej aplikacji demo — określa ono numery portów sieciowych, na których aplikacja będzie nasłuchiwać. Jest to tylko informacja i nie ma znaczenia dla Kubernetes, ale warto to uwzględnić.

Żądania i limity dotyczące zasobów

W rozdziale 5. omówiliśmy już szczegółowo żądania zasobów i limity dotyczące kontenerów, więc wystarczy krótkie podsumowanie.

Każdy kontener może dostarczyć jeden element lub więcej następujących elementów w ramach jego specyfikacji:

- `resources.requests.cpu`,
- `resources.requests.memory`,
- `resources.limits.cpu`,
- `resources.limits.memory`.

Chociaż żądania i limity są określone dla poszczególnych kontenerów, zwykle rozmawiamy o żądaniach i limitach zasobu Pod. Żądanie zasobu Poda jest sumą żądań zasobu dla wszystkich kontenerów Poda itd.

Polityka pobierania obrazu

Jak wiadomo, zanim kontener może zostać uruchomiony w węźle, obraz musi zostać *wyciągnięty* (ang. *pulled*) lub pobrany z odpowiedniego rejestru kontenera. Pole `imagePullPolicy` w kontenerze określa, jak często Kubernetes będzie to robić. Może przyjąć jedną z trzech wartości: `Always`, `IfNotPresent` lub `Never`.

- Dla wartości `Always` obraz jest wyciągany za każdym razem, gdy kontener jest uruchamiany. Zakładając, że określisz tag, który powinieneś (patrz „Tag latest” wcześniej w tym rozdziale), jest to niepotrzebne, marnuje czas i przepustowość.
- Domyślna wartość `IfNotPresent` jest poprawna w większości sytuacji. Jeśli obraz nie jest już obecny w węźle, zostanie pobrany. Potem, chyba że zmienisz specyfikację obrazu, zapisany obraz będzie używany przy każdym uruchomieniu kontenera, a Kubernetes nie będzie próbował go ponownie pobrać.
- Wartość `Never` w ogóle nie aktualizuje obrazu. Dzięki tej zasadzie Kubernetes nigdy nie pobierze obrazu z rejestru: jeśli jest już obecny w węźle, zostanie użyty, ale jeśli nie, kontener nie uruchomi się. Jest mało prawdopodobne, że tego chcesz.

Jeśli napotkasz dziwne problemy (np. Pod nie aktualizuje się po aktualizacji obrazu kontenera), sprawdź zasady pobierania obrazu.

Zmienne środowiskowe

Zmienne środowiskowe są powszechnym, choć ograniczonym sposobem przekazywania informacji do kontenerów w czasie wykonywania. Są sposobem powszechnym, bo wszystkie pliki wykonywalne systemu Linux mają dostęp do zmiennych środowiskowych, a nawet programy napisane na długo przed istnieniem kontenerów mogą używać swojego środowiska do konfiguracji. Są sposobem ograniczonym, ponieważ zmienne środowiskowe mogą być tylko ciągami znaków: bez tablic, bez kluczy i wartości, bez ogólnie uporządkowanych danych. Całkowity rozmiar środowiska procesu jest również ograniczony do 32 KiB, więc nie można przesyłać dużych plików danych.

Aby ustawić zmienną środowiskową, wpisz ją w polu env kontenera:

```
containers:  
- name: demo  
  image: cloudnativelabs/demo:hello  
  env:  
  - name: GREETING  
    value: "Hello from the environment"
```

Jeśli obraz kontenera określa zmienne środowiskowe (np. ustawione w Dockerfile), wówczas ustawienia env Kubernetes zastąpią je. Może to być przydatne do zmiany domyślnej konfiguracji kontenera.



Bardziej elastycznym sposobem przekazywania danych konfiguracyjnych do kontenerów jest użycie obiektów Kubernetes ConfigMap lub Secret: więcej informacji na ten temat znajdziesz w rozdziale 10.

Bezpieczeństwo kontenerów

Być może zauważyłeś w punkcie „Co to jest kontener?”, że kiedy spojrzałem na listę procesów w kontenerze za pomocą polecenia `ps aux`, wszystkie procesy działały jako użytkownik root. W Linuksie i innych systemach operacyjnych opartych na Unixie root jest superużytkownikiem, który ma uprawnienia do odczytu dowolnych danych, modyfikowania dowolnego pliku i wykonywania dowolnej operacji w systemie.

Podczas gdy w pełnym systemie Linux niektóre procesy muszą działać jako root (np. `init`, który zarządza wszystkimi innymi procesami), zwykle nie jest tak w przypadku kontenera.

Rzeczywiście uruchamianie procesów jako użytkownik root, gdy nie jest to konieczne, jest złym pomysłem. Jest to sprzeczne z zasadą najmniejszych uprawnień (https://pl.wikipedia.org/wiki/Zasada_najmniejszego_uprzywilejowania). Oznacza to, że program powinien mieć dostęp tylko do informacji i zasobów, których faktycznie potrzebuje do wykonania swojej pracy.

Programy zawierają błędy; to fakt z życia, widoczny dla każdego, kto napisał choć jeden. Niektóre błędy pozwalają złośliwym użytkownikom przejmować kontrolę nad programem, aby robić rzeczy, których nie powinien, np. czytać tajne dane lub wykonywać dowolny kod. Aby to ograniczyć, ważne jest uruchamianie kontenerów z minimalnymi możliwymi uprawnieniami.

Zaczyna się od niedopuszczenia ich do działania jako root. Przypisuje się im zwykłego użytkownika, takiego, który nie ma specjalnych uprawnień, np. do czytania plików innych użytkowników.

Podobnie jak nie uruchamiasz (lub nie powinieneś) niczego jako root na serwerze, nie powinieneś uruchamiać niczego jako root w kontenerze na serwerze. Uruchamianie plików binarnych utworzonych w innym miejscu wymaga znacznego zaufania, podobnie jest w przypadku plików binarnych w kontenerach.

— Marc Campbell (<https://medium.com/@mccode/processes-in-containers-shouldnot-run-as-root-2feae3f0df3b>)

Osoby atakujące mogą również wykorzystać błędy w środowisku wykonawczym kontenera do „ucieczki” z kontenera i uzyskania takich samych uprawnień na maszynie hosta, jak w kontenerze.

Uruchamianie kontenerów jako użytkownik inny niż root

Oto przykład specyfikacji kontenera, która każe Kubernetes uruchomić kontener jako konkretnego użytkownika:

```
containers:  
- name: demo  
  image: cloudnativd/demo:hello  
  securityContext:  
    runAsUser: 1000
```

Wartością runAsUser jest **UID** (numeryczny identyfikator użytkownika). W wielu systemach Linux identyfikator UID 1000 jest przypisany do pierwszego użytkownika, innego niż root, utworzonego w systemie. Więc ogólnie mówiąc, dla identyfikatorów UID kontenerów można bezpiecznie wybrać wartość 1000 lub większą. Nie ma znaczenia, czy użytkownik Unix z tym identyfikatorem *istnieje* w kontenerze, nawet jeśli w systemie jest system operacyjny; działa to równie dobrze w przypadku kontenerów minimalnych.

Aby uruchomić proces kontenera, Docker pozwala również określić użytkownika w Dockerfile. Nie musisz się jednak tym martwić. Łatwiej i elastyczniej jest ustawić pole runAsUser w specyfikacji Kubernetes.

Jeśli zostanie określony UID za pomocą runAsUser, zastąpi on dowolnego użytkownika skonfigurowanego w obrazie kontenera. Jeśli nie ma runAsUser, ale kontener określa użytkownika, Kubernetes uruchomi go jako tego użytkownika. Jeśli w manifeście lub obrazie nie zostanie określony żaden użytkownik, kontener będzie działał jako root (co, jak widzieliśmy, jest złym pomysłem).

Aby uzyskać maksymalne bezpieczeństwo, należy wybrać inny identyfikator UID dla każdego kontenera. W ten sposób, jeśli jakiś kontener zostanie w narażony na szwank lub przypadkowo zastąpi dane, ma jedynie uprawnienia dostępu do własnych danych, a nie innych kontenerów.

Z drugiej strony, jeśli chcesz, aby dwa kontenery lub więcej kontenerów miało dostęp do tych samych danych (np. przez zamontowany wolumin), powinieneś przypisać im ten sam UID.

Blokowanie kontenerów z uprawnieniami root

Aby zapobiec tej sytuacji, Kubernetes pozwala zablokować uruchamianie kontenerów, gdyby działały one jako użytkownik root.

Spowoduje to ustawienie runAsNonRoot: true:

```
containers:  
- name: demo  
  image: cloudnativived/demo:hello  
  securityContext:  
    runAsNonRoot: true
```

Podczas uruchamiania tego kontenera Kubernetes sprawdzi, czy kontener chce działać jako root. Jeśli tak, odmówi uruchomienia. Zapobiegnie to zapomnieniu o ustawieniu użytkownika innego niż root w kontenerach lub uruchamianiu kontenerów innych firm skonfigurowanych do działania jako root.

Jeśli tak się stanie, zobaczysz status Poda CreateContainerConfigError, a kiedy uruchomisz polecenie kubectl describe, zobaczysz następujący błąd:

```
Error: container has runAsNonRoot and image will run as root
```



Najlepsze praktyki

Uruchamiaj kontenery jako użytkownika innego niż root i blokuj uruchamianie kontenerów z uprawnieniami root za pomocą ustawienia runAsNonRoot: true.

Ustawianie systemu plików tylko do odczytu

Innym przydatnym ustawieniem w kontekście bezpieczeństwa jest readOnlyRootFilesystem, które uniemożliwi zapisy kontenera we własnym systemie plików. Można sobie wyobrazić kontener wykorzystujący błąd w Docker lub Kubernetes, w którym np. zapisywanie w jego systemie plików może wpływać na pliki w węźle hosta. Jeśli jego system plików jest tylko do odczytu, tak się nie stanie; kontener otrzyma błąd we/wy:

```
containers:  
- name: demo  
  image: cloudnativived/demo:hello  
  securityContext:  
    readOnlyRootFilesystem: true
```

Wiele kontenerów nie musi zapisywać niczego we własnym systemie plików, więc to ustawienie nie będzie im przeszkadzało. Dobrą praktyką (<https://kubernetes.io/blog/2016/08/security-best-practices-kubernetes-deployment/>) jest zawsze ustawianie readOnlyRootFilesystem, chyba że kontener naprawdę musi zapisywać do plików.

Wyłączanie eskalacji uprawnień

Zwykle pliki binarne systemu Linux działają z tymi samymi uprawnieniami, co użytkownik, który je wykonuje. Istnieje jednak wyjątek: pliki binarne korzystające z mechanizmu *setuid* mogą tymczasowo uzyskać uprawnienia użytkownika, który jest właścicielem pliku binarnego (zwykle root).

Jest to potencjalny problem występujący w kontenerach, ponieważ nawet jeśli kontener działa jako zwykły użytkownik (np. UID 1000) i zawiera plik binarny *setuid*, ten plik binarny może domyślnie uzyskać uprawnienia root.

Aby temu zapobiec, ustaw pole `allowPrivilegeEscalation` w polityce bezpieczeństwa kontenera na `false`:

```
containers:
- name: demo
  image: cloudnativelabs/demo:hello
  securityContext:
    allowPrivilegeEscalation: false
```

Jak włączyć to ustalenie w całym klastrze, a nie w pojedynczym kontenerze, wyjaśniamy w punkcie „Polityka bezpieczeństwa Poda” dalej w tym rozdziale.

Nowoczesne programy linuksowe nie potrzebują `setuid`; aby osiągnąć to samo, mogą korzystać z bardziej elastycznego i precyzyjnego mechanizmu przywilejów zwanego *mechanizmem właściwości* (ang. *capabilities*).

Mechanizm właściwości

Tradycyjnie programy uniksowe miały dwa poziomy uprawnień — *normalny* i *superużytkownik*. Normalne programy nie mają większych uprawnień od użytkownika, który je uruchamia. Jednakże programy superużytkownika mogą zrobić wszystko, omijając kontrole bezpieczeństwa jądra.

Mechanizm właściwości systemu Linux poprawia to, definiując różne konkretne rzeczy, które program może zrobić; chodzi o ładowanie modułów jądra, wykonywanie bezpośrednich operacji sieciowych, uzyskiwanie dostępu do urządzeń systemowych itd. Każdemu programowi, który potrzebuje określonego uprawnienia, można je przypisać, żadnemu innemu nie.

Przykładowo serwer WWW, który nasłuchiwał na porcie 80, musiałby normalnie zostać uruchomiony na koncie root; numery portów poniżej 1024 są uważane za uprzywilejowane porty systemowe. Zamiast tego programowi można nadać funkcję `NET_BIND_SERVICE`, która pozwala mu łączyć się z dowolnym portem, ale nie daje mu żadnych innych specjalnych uprawnień.

Domyślny zestaw możliwości kontenerów Docker jest dość hojny. Jest to pragmatyczna decyzja oparta na kompromisie bezpieczeństwa z użytecznością: domyślny brak mechanizmu właściwości dla kontenerów wymagałby od operatorów ustawienia właściwości wielu kontenerów w celu ich uruchomienia.

Z drugiej strony, zasada najmniejszego uprzywilejowania mówi, że kontener nie powinien mieć możliwości, których nie potrzebuje. Konteksty bezpieczeństwa Kubernetes pozwalają usunąć dowolne właściwości z zestawu domyślnego i dodawać te, które są potrzebne, tak jak w tym przykładzie:

```
containers:
- name: demo
  image: cloudnativelabs/demo:hello
  securityContext:
    capabilities:
      drop: ["CHOWN", "NET_RAW", "SETPCAP"]
      add: ["NET_ADMIN"]
```

W kontenerze zostaną usunięte właściwości `CHOWN`, `NET_RAW` i `SETPCAP`, a dodana właściwość `NET_ADMIN`.

Dokumentacja Docker (<https://docs.docker.com/engine/reference/run/#runtimeprivilege-and-linux-capabilities>) zawiera listę wszystkich funkcji ustawionych domyślnie na kontenerach, które można dodać, jeśli trzeba.

Aby zapewnić maksymalne bezpieczeństwo, należy usunąć dla każdego kontenera wszystkie właściwości i dodać tylko określone, jeśli są potrzebne:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    capabilities:
      drop: ["all"]
      add: ["NET_BIND_SERVICE"]
```

Mechanizm właściwości nakłada twarde ograniczenie na to, co mogą wykonywać procesy wewnętrz kontenera, nawet jeśli działają jako root. Po usunięciu właściwości na poziomie kontenera nie można jej odzyskać, nawet w przypadku złośliwego procesu z maksymalnymi uprawnieniami.

Konteksty bezpieczeństwa Poda

Omówiliśmy ustawienia kontekstu zabezpieczeń na poziomie pojedynczych kontenerów, ale możesz też ustawić niektóre z nich na poziomie Poda:

```
apiVersion: v1
kind: Pod
...
spec:
  securityContext:
    runAsUser: 1000
    runAsNonRoot: false
    allowPrivilegeEscalation: false
```

Powyższe ustawienia będą miały zastosowanie do wszystkich kontenerów w Podzie, chyba że kontener zastąpi dane ustawienie we własnym kontekście bezpieczeństwa.



Najlepsze praktyki

Ustaw konteksty bezpieczeństwa na wszystkich swoich Podach i kontenerach. Wyłącz eskalację uprawnień i wyłącz wszystkie właściwości. Dodaj tylko te właściwości, których potrzebuje dany kontener.

Polityka bezpieczeństwa Poda

Zamiast określać wszystkie ustawienia zabezpieczeń dla każdego kontenera lub Poda, można je określić na poziomie klastra za pomocą zasobu PodSecurityPolicy. PodSecurityPolicy wygląda następująco:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
```

```
privileged: false
# Wreszcie kodu wypełniamy niektóre wymagane pola
seLinux:
  rule: RunAsAny
supplementalGroups:
  rule: RunAsAny
runAsUser:
  rule: RunAsAny
fsGroup:
  rule: RunAsAny
volumes:
- *
```

Ta prosta polityka blokuje wszelkie uprzywilejowane kontenery (te z flagą `privileged` ustawioną w ich `securityContext`, co dałoby im prawie wszystkie możliwości procesu działającego natywnie na węźle).

Korzystanie z PodSecurityPolicy jest nieco bardziej skomplikowane, ponieważ musisz utworzyć zasady, przyznać odpowiednim kontom usług dostęp do zasad za pośrednictwem RBAC (patrz „Kontrola dostępu oparta na rolach (RBAC)” w rozdziale 11.) i włączyć kontroler dostępu PodSecurityPolicy w klastrze. W większych infrastrukturach lub tam, gdzie nie masz bezpośredniej kontroli nad konfiguracją bezpieczeństwa poszczególnych modułów, korzystanie z PodSecurityPolicy jest dobrym pomysłem.

Możesz przeczytać o tym, jak tworzyć i włączać PodSecurityPolicy w dokumentacji Kubernetes (<https://kubernetes.io/docs/concepts/policy/pod-security-policy/>).

Konta usług Poda

Pody działają z uprawnieniami domyślnego konta usługi dla przestrzeni nazw, chyba że określono inaczej (patrz „Aplikacje i wdrażanie” w rozdziale 11.). Jeśli z jakiegoś powodu musisz udzielić dodatkowych uprawnień (takich jak wyświetlanie Poda w innych przestrzeniach nazw), utwórz dedykowane konto usługi dla aplikacji, powiąż je z wymaganymi rolami i skonfiguruj Pod, aby korzystał z nowego konta usługi.

Aby to zrobić, ustaw pole `serviceAccountName` w specyfikacji Poda na nazwę konta usługi:

```
apiVersion: v1
kind: Pod
...
spec:
  serviceAccountName: deploy-tool
```

Woluminy

Jak zapewne pamiętasz, każdy kontener ma własny system plików, dostępny tylko dla tego kontenera i *efemeryczny*, co oznacza, że wszelkie dane, które nie są częścią obrazu kontenera, zostaną utracone po ponownym uruchomieniu kontenera.

Często jest to w porządku; np. aplikacja demo jest serwerem bezstanowym, który nie wymaga stałej przestrzeni dyskowej. Nie musi też współdzielić plików z żadnym innym kontenerem.

Bardziej złożone aplikacje mogą jednak wymagać zarówno możliwości udostępniania danych innym kontenerom w tym samym zasobniku, jak i utrzymywania ich podczas ponownego uruchamiania. Obiekt wolumin w Kubernetes może zapewnić oba te elementy.

Istnieje wiele różnych rodzajów woluminów, które można podłączyć do Poda. Niekotarne od nośnika pamięci, wolumin zamontowany w Podzie jest dostępny dla wszystkich kontenerów w tymże Podzie. Kontenery, które muszą komunikować się za pomocą udostępniania plików, mogą to zrobić przy użyciu woluminu tego lub innego rodzaju. W poniższych punktach omówimy niektóre z ważniejszych typów.

Woluminy emptyDir

Najprostszym typem woluminu jest emptyDir. To efemeryczna pamięć, na początku pusta — stąd nazwa — która przechowuje dane w węźle (w pamięci lub na dysku węzła). Utrzymuje się tak dugo, jak długo Pod działa w tym węźle.

emptyDir jest przydatny, gdy chcesz zapewnić dodatkowe miejsce do przechowywania kontenera, ale nie jest ważne, aby dane były przechowywane na zawsze lub przenosiły się wraz z kontenerem. Niektóre przykłady zastosowań obejmują buforowanie pobranych plików lub wygenerowanej zawartości albo używanie przestrzeni do zadań przetwarzania danych.

Podobnie, jeśli chcesz po prostu udostępniać pliki między kontenerami w Podzie, ale nie musisz przechowywać danych przez długi czas, wolumin emptyDir jest idealnym rozwiązaniem.

Oto przykład Poda, który tworzy wolumin emptyDir i montuje go w kontenerze:

```
apiVersion: v1
kind: Pod
...
spec:
  volumes:
    - name: cache-volume
      emptyDir: {}
  containers:
    - name: demo
      image: cloudnativived/demo:hello
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
```

Najpierw w sekcji volumes specyfikacji Poda tworzymy wolumin emptyDir o nazwie cache-volume:

```
volumes:
  - name: cache-volume
    emptyDir: {}
```

Teraz wolumin cache-volume jest dostępny dla dowolnego kontenera w Podzie do zamontowania i użycia. Aby to zrobić, podajemy go w sekcji volumeMounts kontenera demo:

```
name: demo
image: cloudnativived/demo:hello
volumeMounts:
  - mountPath: /cache
    name: cache-volume
```

Żeby korzystać z nowego magazynu, kontener nie musi robić nic specjalnego: wszystko, co zapisuje w ścieżce `/cache`, zostanie zapisane w woluminie i będzie widoczne dla innych kontenerów, które zamontują ten sam wolumin. Wszystkie kontenery instalujące wolumin mogą go odczytywać i zapisywać w nim.



Zachowaj ostrożność, zapisując w udostępnionych woluminach. Kubernetes nie wymusza żadnego blokowania zapisów na dysku. Jeśli dwa kontenery spróbują jednocześnie zapisać w tym samym pliku, może to spowodować uszkodzenie danych. Aby tego uniknąć, zaimplementuj własny mechanizm blokady zapisu lub użyj typu woluminu obsługującego blokowanie, takiego jak `nfs` lub `glusterfs`.

Woluminy trwałe

Chociaż efemeryczny wolumin `emptyDir` jest idealny do buforowania i tymczasowego udostępniania plików, niektóre aplikacje muszą przechowywać trwałe dane; przykładem może być dowolny rodzaj bazy danych. Zasadniczo nie zalecamy uruchamiania baz danych w Kubernetes. Prawie zawsze lepiej korzystać z usługi w chmurze; np. większość dostawców usług w chmurze zarządza rozwiązaniami dla relacyjnych baz danych, takich jak MySQL i PostgreSQL, a także baz danych typu klucz-wartość (NoSQL).

Jak pisaliśmy w „Kubernetes nie załatwia wszystkiego” w rozdziale 1., Kubernetes jest najlepszy w zarządzaniu aplikacjami bezstanowymi, co oznacza brak trwałych danych. Przechowywanie trwałych danych znacznie komplikuje konfigurację Kubernetes dla Twojej aplikacji, zużywa dodatkowe zasoby chmurowe oraz wymaga wykonywania dla nich kopii zapasowej.

Jeśli jednak chcesz korzystać z trwałych woluminów, zasób `PersistentVolume` jest tym, czego szukasz. Nie będziemy tutaj omawiać go szczegółowo, ponieważ szczegóły różnią się w zależności od używanego przez Ciebie rozwiązania chmurowego; więcej na temat `PersistentVolumes` możesz dowiedzieć się z dokumentacją Kubernetes (<https://kubernetes.io/docs/concepts/storage/persistent-volumes/>).

Najbardziej elastycznym sposobem korzystania z `PersistentVolumes` w Kubernetes jest utworzenie obiektu `PersistentVolumeClaim`. Reprezentuje on żądanie do utworzenia `PersistentVolume` o określonym typie oraz wielkości, np. woluminu o pojemności 10 GiB z szybką pamięcią do odczytu i zapisu.

Pod może następnie dodać `PersistentVolumeClaim` jako wolumin, który będzie dostępny dla kontenerów do zamontowania i używania:

```
volumes:
- name: data-volume
  persistentVolumeClaim:
    claimName: data-pvc
```

W swoim klastrze możesz utworzyć pulę `PersistentVolumes`. Alternatywnie można skonfigurować dynamiczne przydzielanie (<https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/>): po zamontowaniu `PersistentVolumeClaim` odpowiednia część zostanie automatycznie przydzielona i połączona z Podem.

Restart polityk

W punkcie „Rozwiązywanie problemów w kontenerach” w rozdziale 7. zauważylismy, że Kubernetes zawsze ponownie uruchamia Pod, gdy ten zakończy pracę, chyba że określiłeś inaczej. Domyślną polityką ponownego uruchamiania jest wartość Always, ale możesz zmienić ją na OnFailure (restartuj tylko wtedy, gdy kontener zakończył pracę z niezerowym statusem) lub Never:

```
apiVersion: v1
kind: Pod
...
spec:
  restartPolicy: OnFailure
```

Jeśli chcesz, by Pod wykonał pracę do końca, a następnie zakończył pracę — ale bez jego restartu — możesz użyć zasobu Job do wykonania tej czynności (patrz „Kontroler Job” w rozdziale 9.).

Uwierzytelnianie przy pobieraniu obrazu

Jak wiesz, Kubernetes pobierze określony obraz z rejestru kontenerów — jeśli nie jest już obecny w węźle. Co jednak się stanie, kiedy używasz prywatnego rejestru? Jak przekazać do Kubernetes dane uwierzytelniające do rejestru?

Do tego celu służy pole `imagePullSecrets` w Podzie. Najpierw w obiekcie Secret musisz umieścić dane uwierzytelniające do rejestru (więcej informacji na ten temat znajduje się w podrozdziale „Obiekty Secret aplikacji Kubernetes” w rozdziale 10.). Teraz możesz poinformować Kubernetes, aby używał tego obiektu podczas wyciągania dowolnych kontenerów w Podzie. Oto przykład, kiedy nazwą obiektu Secret jest `registry-creds`:

```
apiVersion: v1
kind: Pod
...
spec:
  imagePullSecrets:
    - name: registry-creds
```

Dokładny format danych uwierzytelniających rejestru opisano w dokumentacji Kubernetes (<https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/>).

Pole `imagePullSecrets` możesz także dołączyć do konta usługi (patrz „Konta usług Poda” wcześniej w tym rozdziale). Wszelkie Pody utworzone za pomocą tego konta usługi będą miały automatycznie dołączone poświadczenia rejestru.

Podsumowanie

Aby zrozumieć Kubernetes, musisz najpierw zrozumieć działanie kontenerów. W tym rozdziale przedstawiliśmy podstawowe dane na temat kontenerów; napisaliśmy, jak współpracują w Podach oraz jakie opcje są dostępne, aby kontrolować sposób działania kontenerów w Kubernetes.

Oto podstawowe zagadnienia.

- Kontener Linux, na poziomie jądra, to izolowany zestaw procesów z wydzielonymi zasobami. Z wnętrza kontenera wygląda to tak, jakby ten kontener miał dla siebie maszynę z systemem Linux.
- Kontenery nie są maszynami wirtualnymi. Każdy kontener powinien uruchamiać jeden proces podstawowy.
- Pod zwykle zawiera jeden kontener z podstawową aplikacją oraz opcjonalne kontenery *pomocnicze*, które ją obsługują.
- Specyfikacje obrazu kontenera mogą obejmować nazwę hosta rejestru, przestrzeń nazw repozytorium, repozytorium obrazów i tag, np. docker.io/cloudnatively/demo:hello. Wymagana jest tylko nazwa obrazu.
- W przypadku powtarzalnych wdrożeń zawsze podawaj tag obrazu kontenera. W przeciwnym razie podczas aktualizacji otrzymasz wszystko oznaczone tagiem `latest`.
- Programy w kontenerach nie powinny działać jako użytkownik root. Zamiast tego przydziel im uprawnienia zwykłego użytkownika.
- Możesz ustawić pole `runAsNonRoot: true` na kontenerze, aby zablokować dowolny kontener, który chce działać jako root.
- Inne przydatne ustawienia zabezpieczeń w kontenerach obejmują `readOnlyRootFilesystem: true` i `allowPrivilegeEscalation: false`.
- Mechanizm właściwości Linuksa zapewnia precyzyjną kontrolę uprawnień, ale domyślne ustawienia kontenerów są zbyt duże. Zaczni od usunięcia wszystkich właściwości kontenerów, a następnie przydziel tylko te, których kontener potrzebuje.
- Kontenery działające w tym samym Podzie mogą udostępniać dane, odczytując i zapisując zamontowany wolumin. Najprostszym woluminem jest `emptyDir`, który jest pusty i zachowuje swoją zawartość tylko tak długo, jak długo działa Pod.
- Wolumin `PersistentVolume` zachowuje swoją zawartość tak długo, jak to konieczne. Pody mogą dynamicznie udostępniać nowe `PersistentVolumes` za pomocą obiektów `PersistentVolume-Claims`.

Zarządzanie Podami

Nie ma wielkich problemów, jest tylko wiele małych.

— Henry Ford

W poprzednim rozdziale szczegółowo omówiliśmy kontenery i wyjaśniliśmy, w jaki sposób są wykorzystywane do tworzenia Podów. Istnieje kilka innych interesujących aspektów Podów, do których przejdziemy w tym rozdziale; są to etykiety, uruchamianie Podów przy użyciu koligacji węzłów, blokowanie Podów przed uruchomieniem na niektórych węzłach z pewnymi skazami (ang. *taints*) i tolerancjami, utrzymywanie Podów razem lub osobno przy użyciu koligacji Podów oraz orkiestracja aplikacji za pomocą kontrolerów Pod, takich jak DaemonSets i StatefulSets.

Omówimy również kilka zaawansowanych opcji sieciowych, w tym zasoby Ingress, Istio oraz Envoy.

Etykiety

Wiesz, że Pody (i inne zasoby Kubernetes) mogą mieć dołączone etykiety oraz że odgrywają one ważną rolę w łączaniu powiązanych zasobów (np. podczas wysyłania żądań z zasobu Serwis do odpowiednich backendów). Przyjrzymy się teraz bliżej etykietom i selektorom.

Co to są etykiety?

Etykiety to pary klucz-wartość, które są dołączone do obiektów, takich jak Pody. Etykiety są przeznaczone do stosowania w celu określenia atrybutów identyfikujących obiekty, które są znaczące i istotne dla użytkowników, ale nie przekazują bezpośrednio semantyki do rdzenia systemu.

— dokumentacja Kubernetes (<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>)

Innymi słowy, etykiety służą do oznaczenia zasobów informacjami, które są istotne dla nas, ale dla Kubernetes nic nie znaczą. Przykładowo etykiety Podów często zawierają nazwę aplikacji, która jest na nich uruchomiona:

```
apiVersion: v1
kind: Pod
```

```
metadata:  
  labels:  
    app: demo
```

W tym momencie etykieta nie przynosi żadnej korzyści. Nadal jest przydatna jako forma dokumentacji: ktoś może spojrzeć na ten Pod i zobaczyć, jaką aplikację obsługuje. Jednak prawdziwa moc etykiet pojawi się, gdy używamy jej z *selektorem*.

Selektory

Selektor to wyrażenie pasujące do etykiety (lub zestawu etykiet). Jest to sposób na określenie grupy zasobów według ich etykiet. Przykładowo zasób Serwis posiada selektor identyfikujący Pody, do których będzie wysyłał żądania. Pamiętasz Serwis demo z punktu „Serwis” w rozdziale 4.?

```
apiVersion: v1  
kind: Service  
...  
spec:  
  ...  
  selector:  
    app: demo
```

Jest to bardzo prosty selektor, pasujący do dowolnego zasobu, który ma etykietę app o wartości demo. Jeśli zasób w ogóle nie ma etykiety app, nie będzie pasował do tego selektora. Jeśli ma etykietę app, ale jej wartość to nie demo, również nie będzie pasować do selektora. Tylko odpowiednie zasoby (w tym przypadku Pody) z etykietą app: demo będą pasować, a wszystkie takie zasoby zostaną wybrane przez Serwis.

Etykiety są używane nie tylko do łączenia zasobów Serwis i Podów; możesz z nich skorzystać bezpośrednio podczas odpytywania klastra za pomocą polecenia kubectl get, przy zastosowaniu flagi --selector:

```
kubectl get pods --all-namespaces --selector app=demo  
NAMESPACE NAME READY STATUS RESTARTS AGE  
demo demo-5cb7d6bfdd-9dckm 1/1 Running 0 20s
```

W punkcie „Używanie przełączników” w rozdziale 7. pisaliśmy, że --selector może być skrócony do -l (dla *etykiet*).

Jeśli chcesz zobaczyć, jakie etykiety są zdefiniowane w Podach, skorzystaj z flagi --show-labels razem z poleceniem kubectl get:

```
kubectl get pods --show-labels  
NAME ... LABELS  
demo-5cb7d6bfdd-9dckm ... app=demo,environment=development
```

Bardziej zaawansowane selektory

W większości przypadków wystarczy prosty selektor podobny do app: demo (znany jako *selektor równości*). Możesz łączyć różne etykiety, aby tworzyć bardziej szczegółowe selektory:

```
kubectl get pods -l app=demo,environment=production
```

Powyższe polecenie zwróci tylko Pody, które mają ustawione zarówno etykiety app: demo, jak i environment: production. W formacie YAML (np. dla Serwis) wyglądałoby to następująco:

```
selector:  
  app: demo  
  environment: production
```

Dla zasobu Serwis dostępne są tylko selektory równości. Natomiast dla zapytań z użyciem kubectl lub bardziej wyrafinowanych zasobów, takich jak Deployment, istnieją inne opcje.

Jedną z nich jest *nierówność*:

```
kubectl get pods -l app!=demo
```

Polecenie spowoduje zwrócenie wszystkich Podów, które mają etykietę app z inną wartością niż demo, lub tych, które w ogóle nie mają etykiety app.

Możesz także zapytać o zbiór wartości etykiet:

```
kubectl get pods -l environment in (staging, production)
```

W formacie YAML wyglądałoby to następująco:

```
selector:  
  matchExpressions:  
  - {key: environment, operator: In, values: [staging, production]}
```

Możesz również poprosić o wartości etykiet spoza danego zbioru:

```
kubectl get pods -l environment notin (production)
```

W formacie YAML wyglądałoby to następująco:

```
selector:  
  matchExpressions:  
  - {key: environment, operator: NotIn, values: [production]}
```

Inny przykład zastosowania matchExpressions możesz zobaczyć w rozdziale 5., w podpunkcie „Używanie koligacji węzłów do kontroli uruchomień”.

Inne zastosowania etykiet

Pokazaliśmy, w jaki sposób połączyć Pody z zasobami Serwis za pomocą etykiety app (właściwie możesz użyć dowolnej etykiety, ale app jest powszechna). Jednak jakie są inne zastosowania etykiet?

W naszym wykresie Helm dla aplikacji demo (patrz „Co znajduje się w wykresie narzędzia Helm” w rozdziale 12.) ustawiamy etykietę environment, która może przyjmować wartości staging lub production. Jeśli korzystasz z Podów programistycznych i produkcyjnych w tym samym klastrze (patrz „Czy potrzebuję wielu klastrów?” w rozdziale 6.), możesz skorzystać właśnie z takiej etykiety, aby rozróżnić dwa środowiska. Przykładowo selektor Serwis dla środowiska produkcyjnego może wyglądać następująco:

```
selector:  
  app: demo  
  environment: production
```

Bez dodatkowego selektora `environment` Serwis pasowałby do wszystkich Podów z etykietą `app: demo`, w tym również do programistycznych.

W zależności od aplikacji możesz użyć etykiet do wybierania zasobów na wiele różnych sposobów. Oto kilka przykładów:

```
metadata:  
  labels:  
    app: demo  
    tier: frontend  
    environment: production  
    version: v1.12.0  
    role: primary
```

Dzięki temu możesz zapytać klaster z różnymi parametrami, aby zobaczyć, co się dzieje.

Möesz także użyć etykiet jako sposobu wykonywania wdrożeń kanarkowych (ang. *canary deployments*) (patrz „Wdrożenia kanarkowe” w rozdziale 13.). Jeśli chcesz wdrożyć nową wersję aplikacji tylko dla niewielkiego odsetka Podów, możesz użyć etykiet, takich jak `track: stable` i `track: canary` dla dwóch osobnych wdrożeń.

Jeśli selektor Serwis pasuje tylko do etykiety `app`, będzie wysyłać ruch do wszystkich Podów pasujących do tego selektora, w tym zarówno do `stable`, jak i `canary`. Możesz zmienić liczbę replik dla obu wdrożeń, aby stopniowo zwiększać odsetek Podów `canary`. Gdy wszystkie działające Pody znajdą się pod etykietą `track: canary`, możesz ponownie oznaczyć je jako `stable` i rozpocząć proces od nowa dla nowej wersji.

Etykiety i adnotacje

Być może zastanawiasz się, jaka jest różnica między etykietami a adnotacjami. Obie reprezentują zestaw par klucz-wartość, które określają metadane dla zasobów.

Różnica polega na tym, że *etykiety identyfikują zasoby*. Służą do wybierania grup powiązanych zasobów, jak np. selektor Serwis. Natomiast adnotacje są używane przez narzędzia lub usługi poza Kubernetes. W punkcie „Funkcja hook wykresu Helm” w rozdziale 13. znajduje się przykład użycia adnotacji do sterowania pracami Helm.

Ponieważ etykiety są często używane wewnętrznych zapytaniach, które mają krytyczne znaczenie dla Kubernetes, istnieją pewne dość ścisłe ograniczenia dotyczące prawidłowości etykiet. Przykładowo nazwy etykiet są ograniczone do 63 znaków, chociaż mogą mieć opcjonalny prefiks o długości 253 znaków — w postaci poddomeny DNS, oddzielonej od etykiety ukośnikiem. Etykiety mogą zaczynać się tylko od znaku alfanumerycznego (litery lub cyfry) i zawierać wyłącznie znaki alfanumeryczne oraz myślniki, podkreślenia i kropki. Wartości etykiet mają podobne ograniczenia (<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#syntax-and-character-set>).

W praktyce wątpimy, aby zabrakło znaków dla nazwy etykiety, ponieważ większość powszechnie używanych etykiet to tylko jedno słowo (np. `app`).

Koligacje węzłów

Koligacje węzłów w skrócie opisaliśmy w podpunkcie „Używanie koligacji węzłów do kontroli uruchomień” w rozdziale 5. odniesieniu do węzłów preemptible. Wtedy nauczyłeś się, jak używać koligacji węzłów, aby preferencyjnie uruchamiać Pody w określonych węzłach (lub nie). Przyjrzymy się teraz bardziej szczegółowo koligacji węzłów.

W większości przypadków nie potrzebujesz koligacji węzłów. Kubernetes stosuje sprytne podejście do uruchamiania Podów w odpowiednich węzłach. Jeśli wszystkie Twoje węzły nadają się do obsługi danego Poda, nie przejmuj się nimi.

Istnieją jednak wyjątki, takie jak węzły preemptible. Jeśli ponowne uruchomienie Poda jest kosztowne, prawdopodobnie w miarę możliwości należy unikać uruchamiania w węźle preemptible; węzły te mogą zniknąć z klastra bez ostrzeżenia. Możesz wyrazić tego rodzaju preferencje za pomocą koligacji węzłów.

Istnieją dwa rodzaje koligacji — twarda i miękka. Ponieważ inżynierowie oprogramowania nie zawsze są najlepsi w nadawaniu nazw różnym rzeczom, w Kubernetes są one nazywane:

- `requiredDuringSchedulingIgnoredDuringExecution` (twarda),
- `preferredDuringSchedulingIgnoredDuringExecution` (miękka).

Aby to zapamiętać, wystarczy skojarzyć, że `required` oznacza twardą koligację (reguła *musi* być spełniona, aby uruchomić ten Pod), a `preferred` oznacza miękką koligację (byłoby *miło*, gdyby reguła była spełniona, ale nie jest krytyczna).



Długie nazwy twardych i miękkich typów koligacji wskazują, że reguły te mają zastosowanie *podczas planowania*, ale nie *podczas wykonywania*. Oznacza to, że gdy Pod zostanie przypisany do określonego węzła spełniającego koligację, pozostanie tam. Jeśli coś się zmieni, gdy Pod jest uruchomiony, reguła przestanie być przestrzegana i Kubernetes nie przeniesie Poda. (Ta funkcja może zostać dodana w przyszłości).

Koligacje twarde

Koligacja wyrażana jest przez opisanie rodzaju węzłów, na których ma działać Pod. Może istnieć kilka zasad dotyczących tego, w jaki sposób Kubernetes wybiera węzły dla Poda. Każda z nich jest wyrażana za pomocą pola `nodeSelectorTerms`. Oto przykład:

```
apiVersion: v1
kind: Pod
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "failure-domain.beta.kubernetes.io/zone"
                operator: In
                values: ["us-central1-a"]
```

Tylko węzły znajdujące się w strefie us-central1-a będą pasować do tej reguły, więc ogólnym efektem jest upewnienie się, że Pod jest zaplanowany tylko w tej strefie.

Koligacje miękkie

Koligacje miękkie są wyrażane w bardzo podobny sposób, z tym wyjątkiem, że każdej regule przypisuje się *wagę* liczbową od 1 do 100, która określa wpływ, jaki ma ona na wynik. Oto przykład:

```
preferredDuringSchedulingIgnoredDuringExecution:  
- weight: 10  
  preference:  
    matchExpressions:  
      - key: "failure-domain.beta.kubernetes.io/zone"  
        operator: In  
        values: ["us-central1-a"]  
- weight: 100  
  preference:  
    matchExpressions:  
      - key: "failure-domain.beta.kubernetes.io/zone"  
        operator: In  
        values: ["us-central1-b"]
```

Ponieważ to zasada preferred..., jest to koligacja miękka, czyli Kubernetes może zaplanować zasobnik na dowolnym węźle, ale da pierwszeństwo węzłom, które pasują do tych reguł.

Widać, że te dwie reguły mają różne wartości wagi. Pierwsza reguła ma wagę 10, ale druga ma wagę 100. Jeśli istnieją węzły, które pasują do obu reguł, Kubernetes da 10 razy większy priorytet węzłom, które pasują do drugiej reguły (będąc w strefie dostępności us-central1-b).

Wagi są użytecznym sposobem wyrażenia względnej ważności Twoich preferencji.

Koligacje Podów i antykoligacje

Widziałeś, jak można wykorzystać koligacje węzłów, aby wpłynąć na uruchomienia Podów na niektórych rodzajach węzłów. Czy jednak można wpływać na takie decyzje w oparciu o to, jakie inne Pody już działają w węźle?

Czasami są takie pary Podów, które działają lepiej, gdy są razem w tym samym węźle, np. serwer WWW i baza danych, taka jak Redis. Byłoby użyteczne, gdybyś mógł dodać do specyfikacji Poda informacje, które podpowiadają schedulerowi, że dany Pod wolałby być kolokowany z Podem pasującym do określonego zestawu etykiet.

I odwrotnie, czasami chcesz, aby Pody unikały się nawzajem. W rozdziale 5., w punkcie „Utrzymywanie równowagi obciążenia”, widzieliśmy problemy, które mogą powstać, jeśli repliki Poda skończyłyby pracę razem w tym samym węźle, a nie byłyby rozproszone w klastrze. Czy można powiedzieć schedulerowi, aby unikał rozplanowania Podów w tych samych węzłach?

Temu właśnie służą koligacje Podów. Podobnie jak koligacje węzłów, koligacje Podów są wyrażane jako zbiór reguł: albo twarde wymagania, albo miękkie preferencje z zestawem wag.

Trzymanie Podów razem

Spójrzmy na pierwszy przypadek, czyli wspólne działanie Podów. Założymy, że masz jeden Pod z etykietą app: server, która jest serwerem webowym, oraz inny z etykietą app: cache, który jest pamięcią podręczną dla zawartości serwera. Mogą nadal współpracować, nawet jeśli znajdują się na osobnych węzłach, ale lepiej, gdy znajdują się w tym samym węźle, ponieważ mogą się komunikować bez konieczności łączenia z siecią. Jak nakazać schedulerowi ich kolokację?

Oto przykład wymaganej koligacji Podów, wyrażonej jako część specyfikacji Poda server. Efekt byłby taki sam, jeśli dodasz go do specyfikacji Poda cache lub do obu Podów:

```
apiVersion: v1
kind: Pod
metadata:
  name: server
  labels:
    app: server
...
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
          - matchExpressions:
            - key: app
              operator: In
              values: ["cache"]
        topologyKey: kubernetes.io/hostname
```

Ogólnym efektem tej koligacji jest zapewnienie, że Pod sever będzie uruchomiony, jeśli to możliwe, w węźle, w którym będzie uruchomiony Pod oznaczony etykietą cache. Jeśli nie ma takiego węzła lub nie ma pasującego węzła, który ma wystarczające wolne zasoby do uruchomienia Poda, nie będzie on mógł działać.

Prawdopodobnie nie jest to zachowanie, którego oczekujesz w rzeczywistej sytuacji. Jeśli dwa Pody absolutnie muszą zostać kolokowane, umieść ich kontenery w tym samym Podzie. Jeśli kolokacja jest tylko preferowana, skorzystaj z koligacji miękkiej (preferredDuringSchedulingIgnoredDuringExecution).

Trzymanie Podów oddzielnie

Rozważmy teraz przypadek antykoligacji, czyli oddzielenie niektórych Podów. Zamiast podAffinity używamy podAntiAffinity:

```
apiVersion: v1
kind: Pod
metadata:
  name: server
  labels:
    app: server
...
spec:
  affinity:
    podAntiAffinity:
```

```
requiredDuringSchedulingIgnoredDuringExecution:  
  labelSelector:  
    - matchExpressions:  
      - key: app  
        operator: In  
        values: ["server"]  
  topologyKey: kubernetes.io/hostname
```

Jest to przykład bardzo podobny do poprzedniego, z tym wyjątkiem, że wykorzystano podAntiAffinity. Wartość ta wyraża przeciwny sens, a wyrażenie dopasowania jest inne. Tym razem wyrażenie brzmi: „Etykieta app musi mieć wartość server”.

Efektem tej koligacji jest zapewnienie, że Pod *nie zostanie uruchomiony* w żadnym węźle pasującym do tej reguły. Innymi słowy, żaden Pod z etykietą app: server nie może być uruchomiony w węźle, na którym działa już Pod z taką etykietą. Wymusi to równomierną dystrybucję Podów server w klasztre, przy możliwym koszcie pożąданie liczby replik.

Antykoligacje miękkie

Zazwyczaj jednak bardziej zależy nam na tym, aby mieć wystarczającą liczbę dostępnych replik niż na sprawiedliwej dystrybucji. Jednak to nie jest to, czego tutaj chcemy. Zmodyfikujmy nieznacznie przykład tak, aby stała się antykoligacją miękką:

```
affinity:  
  podAntiAffinity:  
    preferredDuringSchedulingIgnoredDuringExecution:  
      - weight: 1  
        podAffinityTerm:  
          labelSelector:  
            - matchExpressions:  
              - key: app  
                operator: In  
                values: ["server"]  
        topologyKey: kubernetes.io/hostname
```

Zauważ, że teraz reguła jest typu preferred..., not required..., dzięki czemu stała się antykoligacją miękką. Jeśli reguła może być spełniona, będzie, ale jeśli nie, Kubernetes i tak uruchomi Pod.

Ponieważ jest to preferencja, określamy wartość wagi, tak jak robiliśmy to dla miękkiej koligacji węzłów. Jeśli trzeba rozważyć wiele reguł koligacji, Kubernetes uszereguje je według wagi przypisanej do każdej z reguł.

Kiedy korzystać z koligacji Podów?

Podobnie jak koligację węzłów, powinieneś traktować koligację Podów jako ulepszenie dostrajające w szczególnych przypadkach. Scheduler zapewnia już najlepszą wydajność i dostępność w zakresie rozmieszczenia Podów. Koligacje Podów ograniczają swobodę schedulera, narzucając ograniczenia dotyczące aplikacji. Korzystanie z koligacji powinno zajść w momencie, gdy zauważysz problem w produkcji, a koligacja Podów jest jedynym sposobem na jego rozwiązanie.

Skazy i tolerancje

W tym rozdziale, w podrozdziale „Koligacje węzłów”, dowiedziałeś się o właściwości Poda, która pozwala na dołączanie do zbioru węzłów (lub ich odłączanie). I odwrotnie, *skazy* (ang. *taints*) pozwalają węźłowi unikać zbioru Podów, w oparciu o pewne właściwości węzła.

Możesz np. wykorzystać skazy do utworzenia dedykowanych węzłów, czyli węzłów, które są zarezerwowane tylko dla określonych rodzajów Podów. Kubernetes tworzy również skazy, jeśli w węźle występują pewne problemy, takie jak brak pamięci lub brak połączenia sieciowego.

Aby dodać skazę do określonego węzła, użyj polecenia kubectl taint:

```
kubectl taint nodes docker-for-desktop dedicated=true:NoSchedule
```

Polecenie to dodaje skazę o nazwie dedicated=true do węzła docker-for-desktop, z efektem NoSchedule; żaden Pod nie może teraz zostać uruchomiony, jeśli nie ma pasującej tolerancji (ang. *toleration*).

Aby zobaczyć skazy skonfigurowane w danym węźle, użyj polecenia kubectl describe node

Aby usunąć skazę z węzła, powtóż polecenie kubectl taint, ale ze znakiem minus po nazwie skazy:

```
kubectl taint nodes docker-for-desktop dedicated:NoSchedule-
```

Tolerancje to właściwości Podów opisujące skazy, które są z nimi kompatybilne. Aby np. utworzyć Pod tolerujący skazę dedicated=true, dodaj do specyfikacji Poda poniższy fragment:

```
apiVersion: v1
kind: Pod
...
spec:
  tolerations:
    - key: "dedicated"
      operator: "Equal"
      value: "true"
      effect: "NoSchedule"
```

Oznacza to: „Ten Pod może działać na węzłach, które mają skazę dedicated=true z efektem NoSchedule”. Ponieważ tolerancja pasuje do skazy, Pod może zostać uruchomiony. Jakiekolwiek Pody bez tej tolerancji nie będą mogły działać w węźle z daną skazą.

Kiedy Pod nie może w ogóle działać z powodu skazy w węzłach, pozostanie w stanie Pending, a w opisie Poda zobaczymy taki komunikat:

```
Warning FailedScheduling 4s (x10 over 2m) default-scheduler 0/1 nodes are available: 1
  ↳node(s) had taints that the pod didn't tolerate.
```

Innym zastosowaniem skaz i tolerancji może być oznaczanie specjalistycznych węzłów (np. GPU) oraz pozwalanie niektórym Podom na tolerowanie pewnych rodzajów problemów z węzłami.

Jeśli np. węzeł utraci dostęp do sieci, Kubernetes automatycznie dodaje skazę node.kubernetes.io/unreachable. Normalnie spowodowałoby to, że kubelet usunąłby wszystkie Pody z węzła. Możesz jednak zechcieć utrzymać działanie niektórych Podów, mając nadzieję, że sieć wróci w rozsądny czasie. Aby to zrobić, możesz dodać do tych Podów tolerancję, pasującą do skazy unreachable.

Więcej na temat skaz i tolerancji możesz przeczytać w dokumentacji Kubernetes (<https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/>).

Kontrolery Podów

W tym rozdziale dużo pisaliśmy o Podach i to ma sens, gdyż wszystkie aplikacje Kubernetes działają w Podzie. Możesz się zastanawiać, dlaczego w ogóle potrzebujemy innych rodzajów obiektów. Czy nie wystarczy tylko utworzyć Pod dla aplikacji i uruchomić go?

Właśnie to uzyskujesz, uruchamiając kontener bezpośrednio za pomocą polecenia `docker container run`, jak to zrobiliśmy w rozdziale 2., w punkcie „Uruchamianie obrazu kontenera”. To działa, ale jest bardzo ograniczone.

- Jeśli kontener z jakiegoś powodu kończy pracę, musisz go ponownie uruchomić ręcznie.
- Istnieje tylko jedna replika kontenera i nie ma możliwości równoważenia obciążenia ruchu w wielu replikach, jeśli uruchomiono je ręcznie.
- Jeśli chcesz mieć wysoce niezawodne repliki, musisz zdecydować, na których węzłach je uruchomić, i zadbać o utrzymanie równowagi klastra.
- Po aktualizowaniu kontenera musisz zadbać o to, aby po kolej zatrzymać każdy uruchomiony obraz, pobrać nowy obraz i ponownie go uruchomić.

Kubernetes wyręcza nas przy tej pracy za pomocą *kontrolerów*. W rozdziale 4., w podrozdziale „ReplicaSet”, przedstawiliśmy kontroler ReplicaSet, który zarządza grupą replik konkretnego Poda. Działa w sposób ciągły, aby upewnić się, że zawsze jest określona liczba replik; uruchamia nowe, jeśli jest ich za mało, a jeśli jest zbyt wiele — kończy ich pracę.

Poznałeś również obiekty Deployment, które — jak widziałeś w rozdziale 4., w podrozdziale „Zasoby Deployment” — zarządzają obiektami ReplicaSet, aby kontrolować wdrażanie aktualizacji aplikacji. Przykładowo po wykonaniu aktualizacji obiektu Deployment wgrywana jest nowa specyfikacja kontenera, tworzony nowy obiekt ReplicaSet, aby uruchomić nowe Pody, oraz ewentualnie zamknięty ReplicaSet, który zarządzał starymi Podami.

W przypadku najprostszych aplikacji wystarczy wdrożenie. Jednak istnieje kilka innych przydatnych rodzajów kontrolerów Poda. W tym rozdziale omówimy pokrótkę kilka z nich.

DaemonSet

Załóżmy, że chcesz wysłać dzienniki ze wszystkich aplikacji do scentralizowanego serwera, takiego jak Elasticsearch-Logstash-Kibana (ELK) lub aplikacji SaaS, takiej jak Datadog (patrz „Datadog” w rozdziale 16.). Można to zrobić na kilka sposobów.

Każda aplikacja może zawierać odpowiedni kawałek kodu służącego do łączenia się z usługą logowania, uwierzytelniania, zapisywania dzienników itd. Przez to może istnieć wiele zduplikowanych fragmentów kodu, co jest nieefektywne.

Alternatywnie możesz uruchomić w każdym Podzie dodatkowy kontener, działający w charakterze agenta rejestrującego (określany jako wzór *przyczepki* — ang. *sidecar pattern*). Oznacza to, że żadna

aplikacja nie musi mieć wbudowanej wiedzy o tym, jak rozmawiać z usługą rejestrowania. Jednak znaczy to, że potencjalnie masz kilka kopii agenta rejestrowania działających w każdym węźle.

Ponieważ wszystko, co robi dany kontener, to zarządzanie połączeniem z usługą rejestrowania i przekazywanie do niej komunikatów dziennika, potrzebujesz tylko jednej kopii agenta rejestrowania w każdym węźle. Jest to tak powszechny wymóg, że Kubernetes udostępnia dla niego specjalny obiekt kontrolera, czyli *DaemonSet*.



Termin *demon* tradycyjnie odnosi się do długotrwałych procesów działających w tle na serwerze, które obsługują takie funkcje jak rejestrowanie, więc analogicznie DaemonSet uruchamia kontener *daemona* na każdym węźle w klastrze.

Manifest dla DaemonSet, jak można się spodziewać, wygląda bardzo podobnie jak dla obiektu Deployment:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
...
spec:
...
template:
...
spec:
  containers:
    - name: fluentd-elasticsearch
...
```

Użyj DaemonSet, gdy chcesz uruchomić jedną kopię Poda na każdym z węzłów w klastrze. Jeśli korzystasz z aplikacji, w której utrzymanie określonej liczby replik jest ważniejsze niż to, w którym węże działały Pody, użyj zamiast tego obiektu Deployment.

StatefulSet

Podobnie jak Deployment lub DaemonSet, StatefulSet jest rodzajem kontrolera Poda. StatefulSet dodaje możliwość uruchamiania i zatrzymywania Podów w określonej kolejności.

W przypadku obiektu Deployment wszystkie Twoje Pody są uruchamiane i zatrzymywane w losowej kolejności. Jest to dobre w przypadku usług bezstanowych, w których każda replika jest identyczna i wykonuje tę samą pracę.

Czasami jednak musisz uruchomić Pody w określonej numerowanej sekwencji i być w stanie rozpoznać je za pomocą numeru. Przykładowo aplikacje rozproszone, takie jak Redis, MongoDB lub Cassandra, tworzą własne klastry i muszą być w stanie zidentyfikować lidera klastra za pomocą przewidywalnej nazwy.

StatefulSet jest do tego idealny. Jeśli np. utworzysz StatefulSet o nazwie `redis`, pierwszy uruchomiony Pod będzie miał nazwę `redis-0`, a przed uruchomieniem następnego, `redis-1`, Kubernetes poczeka, aż poprzedni Pod będzie gotowy.

W zależności od aplikacji możesz użyć tej właściwości do niezawodnego grupowania Podów. Przykładowo każdy Pod może uruchomić skrypt startowy, który sprawdza, czy działa na redis-0. Jeśli tak, znaczy to, że jest to lider klastra. Jeśli nie, spróbuje dołączyć do klastra, kontaktując się z redis-0.

Każda replika w StatefulSet musi być uruchomiona i gotowa, zanim Kubernetes uruchomi następną, i podobnie po zakończeniu StatefulSet repliki zostaną zamknięte w odwrotnej kolejności, czekając na zakończenie każdego Poda przed przejściem do następnego.

Oprócz tych specjalnych właściwości, StatefulSet przypomina normalny obiekt Deployment:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  selector:
    matchLabels:
      app: redis
  serviceName: "redis"
  replicas: 3
  template:
    ...

```

Aby można było adresować każdy z Podów za pomocą przewidywalnej nazwy DNS, takiej jak redis-1, musisz także utworzyć Serwis z polem cUserIP o wartości None (znany jako *usługa headless*).

Z zasobem Serwis typu nonheadless otrzymujesz pojedynczy wpis DNS (taki jak redis), który równoważy obciążenia we wszystkich backendowych Podach. Przy usłudze headless nadal otrzymujesz nazwę DNS pojedynczej usługi, ale dostajesz także indywidualne wpisy DNS dla każdego ponumerowanego Poda, takie jak redis-0, redis-1, redis-2 itd.

Pody, które muszą dołączyć do klastra Redis, mogą szczegółowo skontaktować się z redis-0, ale aplikacje, które potrzebują po prostu usługi Redis z równoważeniem obciążenia, mogą używać nazwy DNS redis do komunikowania się z losowo wybranym zasobnikiem Redis.

StatefulSets może również zarządzać pamięcią dyskową dla swoich Podów, używając obiektu VolumeClaimTemplate, który automatycznie tworzy PersistentVolumeClaim (patrz „Woluminy trwałe” w rozdziale 8.).

Kontroler Job

Innym przydatnym typem kontrolera Pod w Kubernetes jest Job. Podczas gdy Deployment uruchamia określoną liczbę Podów i restartuje je w sposób ciągły, Job uruchamia Pod tylko określoną liczbę razy. Następnie kończy pracę.

Przykładowo zadanie przetwarzania wsadowego lub Pod workera kolejki zwykłe uruchamia się, wykonuje swoją pracę, a następnie kończy działanie. Jest to idealny kandydat do zarządzania przez Job.

Istnieją dwa pola sterujące wykonywaniem zadania — completions i parallelism. Pierwsze, comple-
tions, ustala, ile razy określony Pod musi zadziałać, zanim zadanie zostanie uznane za ukończone. Wartością domyślną jest 1, co oznacza, że Pod uruchomi się raz.

Pole `parallelism` określa, ile Podów powinno działać jednocześnie. Ponownie domyślną wartością jest 1, co oznacza, że domyślnie będzie działać tylko jeden Pod.

Założymy, że chcesz uruchomić Job workera kolejki, którego celem jest obsługa poszczególnych elementów kolejki. Możesz ustawić `parallelism` na 10 i pozostawić pole `completions` puste. Spowoduje to uruchomienie 10 Podów, z których każdy będzie obsługiwał elementy kolejki, dopóki nie będzie już pracy do wykonania. W tym momencie Job zostanie zakończony:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: queue-worker
spec:
  parallelism: 10
  template:
    metadata:
      name: queue-worker
    spec:
      containers:
        ...

```

Alternatywnie, jeśli chcesz uruchomić zadanie przetwarzania wsadowego, możesz pozostawić pola `completions` i `parallelism` z wartością 1. Spowoduje to uruchomienie tylko jednej kopii Poda. Jeśli ulegnie awarii lub zakończy się niepowodzeniem, Job zrestartuje go, podobnie jak w przypadku obiektu Deployment. Wartość `completions` określa tylko pozytywne zakończenia działań.

Jak uruchomić kontroler Job? Możesz to zrobić ręcznie, stosując manifest Job za pomocą polecenia `kubectl` lub wykres Helm. Job może też zostać uruchomiony automatycznie; przy użyciu automatycznego procesu wdrożenia (patrz rozdział 14.).

Prawdopodobnie najczęstszym sposobem korzystania z kontrolera Job jest uruchamianie go określowo, o określonej porze dnia lub w ustalonych odstępach czasu. Kubernetes został wyposażony w specjalny typ Job, służący właśnie do tego; jest to CronJob.

CronJob

W środowiskach uniksowych zaplanowane zadania są uruchamiane przez demon `cron` (którego nazwa pochodzi od greckiego słowa χρόνος oznaczającego „czas”). W związku z tym są one znane jako *zadania cron*, a obiekt Kubernetes — CronJob — robi dokładnie to samo.

CronJob wygląda następująco:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: demo-cron
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      ...

```

Dwa ważne pola, występujące w manifeście CronJob to `spec.schedule` i `spec.jobTemplate`. Pole `schedule` określa, kiedy Job zostanie uruchomiony. Stosowany jest taki sam format (<https://en.wikipedia.org/wiki/Cron>) jak w przypadku uniksowego narzędzia `cron`.

`jobTemplate` określa szablon Job, który ma zostać uruchomiony i jest dokładnie taki sam jak normalny manifest Job (patrz „Kontroler Job” w tym rozdziale).

Horizontal Pod Autoscaler

Pamiętaj, że kontroler obiektu Deployment utrzymuje określoną liczbę replik Pod. Jeśli jedna replika ulegnie awarii, inna zostanie uruchomiona, aby ją zastąpić. Jeśli z jakiegoś powodu jest zbyt wiele Podów, Deployment zatrzyma nadmiarowe Pody, aby osiągnąć docelową liczbę replik.

Pożądana liczba replik jest ustawiana w manifeście obiektu Deployment. Wiesz już, że możesz ją zmienić, aby zwiększyć liczbę Podów w przypadku dużego natężenia ruchu, lub zmniejszyć ją, gdy istnieją bezczynne Pody.

Czy Kubernetes mógłby automatycznie dostosować liczbę replik za Ciebie, w zależności od aktualnego ruchu? Właśnie to robi Horizontal Pod Autoskaler. Skalowanie w poziomie (ang. *horizontal*) odnosi się do dostosowania liczby replik usługi, w przeciwieństwie do skalowania w pionie (ang. *vertical*), co powoduje, że poszczególne repliki są większe lub mniejsze.

Horizontal Pod Autoscaler (HPA) obserwuje konkretny Deployment, stale monitorując dane, aby sprawdzić, czy konieczne jest zwiększenie lub zmniejszenie liczby replik.

Jednym z najczęstszych wskaźników wykorzystywanych przez automatyczne skalowanie jest stopień wykorzystania procesora. Pamiętasz z punktu „Żądania zasobów” w rozdziale 5., że Pody mogą zażądać określonej mocy obliczeniowej procesora, np. 500 milicpus. Podczas działania Poda zużycie procesora będzie się zmieniać, co oznacza, że w danym momencie Pod faktycznie wykorzystuje pewien procent swojego pierwotnego żądania CPU.

Możesz automatycznie skalować Deployment w oparciu o tę wartość; przykładowo możesz utworzyć HPA, którego celem będzie 80% wykorzystanie procesora w Podach. Jeśli średnie zużycie procesora we wszystkich Podach Deployment wynosi tylko 70% ich żądanej ilości, HPA zmniejszy docelową liczbę replik. Jeśli Pody nie pracują ciężko, nie potrzebujemy ich tak wielu.

Z drugiej strony, jeśli średnie wykorzystanie procesora wynosi 90%, przekracza to ustawione 80%, więc musimy dodawać więcej replik, dopóki średnie zużycie procesora nie spadnie. HPA zmodyfikuje Deployment, aby zwiększyć docelową liczbę replik.

Za każdym razem, gdy HPA stwierdza, że musi wykonać operację skalowania, dostosowuje ilość replik w oparciu o stosunek rzeczywistej wartości do wartości ustawionej. Jeśli aktualna wartość obiektu Deployment jest bardzo zbliżona do docelowego wykorzystania procesora, HPA doda lub usunie tylko niewielką liczbę replik; jeśli jest znacznie poza skalą, HPA do dostosowania wykorzysta większą liczbę.

Oto przykład HPA wykorzystujący zużycie procesora:

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
```

```

name: demo-hpa
namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: demo
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
      targetAverageUtilization: 80

```

Interesujące pola to:

- `spec.scaleTargetRef`, które określa Deployment do skalowania,
- `spec.minReplicas` i `spec.maxReplicas`, które określają granice skalowania,
- `spec.metrics`, określające metryki, które będą używane do skalowania.

Chociaż wykorzystanie procesora jest najczęstszym wskaźnikiem skalowania, możesz użyć dowolnych wskaźników dostępnych dla Kubernetes, w tym zarówno wbudowanych *metryk systemowych*, takich jak użycie procesora i pamięci, jak i *mierników usług* specyficznych dla aplikacji, które definiujesz i eksportujesz ze swojej aplikacji (patrz rozdział 16.). Możesz np. skalować na podstawie poziomu błędu aplikacji.

Więcej informacji na temat autoskalerów i niestandardowych metryk można znaleźć w dokumentacji Kubernetes (<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/>).

PodPreset

PodPreset to eksperymentalna funkcja w wersji alfa dla Kubernetes, która pozwala wstrzykiwać informacje do Podów podczas ich tworzenia. Można np. utworzyć PodPreset, który montuje wolumin na wszystkich Podach pasujących do danego zestawu etykiet.

PodPreset to rodzaj obiektu zwany *kontrolerem dostępu* (ang. *admission controller*). Kontrolery dostępu obserwują budowanie Podów i podejmują pewne działania, gdy tworzone są Pody pasujące do ich selektora. Przykładowo niektóre kontrolery dostępu mogą blokować tworzenie Poda, jeśli narusza to politykę. Natomiast inne wprowadzają dodatkową konfigurację do Poda.

Oto przykład PodPreset, który zwiększa ilość pamięci cache dla wszystkich Podów pasujących do selektora `tier: frontend`:

```

apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: add-cache
spec:
  selector:
    matchLabels:
      tier: frontend
  volumeMounts:

```

```
- mountPath: /cache
  name: cache-volume
volumes:
- name: cache-volume
  emptyDir: {}
```

Ustawienia zdefiniowane przez PodPreset są scalane z ustawieniami każdego Poda. Jeśli Pod zostanie zmodyfikowany przez PodPreset, zobaczysz taką adnotację:

```
podpreset.admission.kubernetes.io/podpreset-add-cache: "<resource version>"
```

Co się stanie, jeśli ustawienia własne kapsuły są sprzeczne z ustawieniami zdefiniowanymi w PodPreset lub jeśli wiele funkcji PodPreset powoduje konflikt ustawień? W takim przypadku Kubernetes odmówi modyfikacji Poda, a w opisie Poda zobaczysz zdarzenie z komunikatem `Conflict on pod preset`.

Z tego powodu funkcje PodPreset nie mogą być używane do zastępowania własnej konfiguracji Poda, a jedynie do uzupełniania ustawień, których sam Pod nie określa. Pod może zrezygnować z modyfikacji przez PodPreset, poprzez ustawienie adnotacji:

```
podpreset.admission.kubernetes.io/exclude: "true"
```

Ponieważ funkcje PodPreset są nadal w fazie eksperymentalnej, mogą nie być dostępne w zarządzanych klastrach Kubernetes i może być konieczne wykonanie dodatkowych kroków w celu włączenia ich w klastrach samodzielnie hostowanych. Wtedy należy z wiersza poleceń wysłać odpowiednie polecenia do serwera API. Szczegółowe informacje znajdują się w dokumentacji Kubernetes (<https://kubernetes.io/docs/concepts/workloads/pods/podpreset/>).

Operatory i niestandardowe definicje zasobów (CRD)

W punkcie „StatefulSets” w tym rozdziale napisaliśmy, że chociaż standardowe obiekty Kubernetes, takie jak Deployment i Serwis, nadają się do prostych, bezstanowych aplikacji, mają swoje ograniczenia. Niektóre aplikacje wymagają wielu współpracujących Podów, które muszą być inicjowane w określonej kolejności (np. replikowane bazy danych lub usługi klastrowe).

W przypadku aplikacji, które wymagają bardziej skomplikowanego zarządzania, niż może zapewnić StatefulSets, Kubernetes umożliwia tworzenie własnych nowych typów obiektów. Są to tzw. *niestandardowe definicje zasobów* (CRD — ang. *custom resource definitions*). Przykładowo narzędzie do tworzenia kopii zapasowych Velero buduje niestandardowe obiekty Kubernetes, takie jak Konfiguracja i Backup (patrz „Velero” w rozdziale 11.).

Kubernetes został zaprojektowany w taki sposób, aby był rozszerzalny, dlatego możesz dowolnie definiować i tworzyć obiekty za pomocą mechanizmu CRD. Niektóre CRD istnieją tylko do przechowywania danych, np. obiekt Velero `BackupStorageLocation`. Jednak możesz tworzyć obiekty, które działają jak kontrolery Poda, podobnie jak Deployment lub StatefulSet.

Jeśli np. chcesz utworzyć obiekt kontrolera, który konfiguruje replikowane klastry bazy danych MySQL o wysokiej niezawodności w Kubernetes, jak to zrobić?

Pierwszym krokiem byłoby utworzenie CRD dla niestandardowego obiektu kontrolera. Aby to zrobić, musisz napisać program, który komunikuje się z interfejsem API Kubernetes. Jest to łatwe

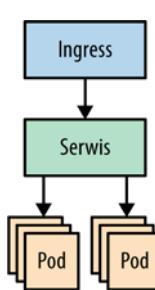
do zrobienia, jak widziałeś w podrozdziale „Budowanie własnych narzędzi Kubernetes” w rozdziale 7. Taki program nazywany jest operatorem (ang. *operator*), być może dlatego, że automatyzuje działania, które może wykonywać operator, będący człowiekiem.

Aby utworzyć operatora, nie potrzebujesz żadnych niestandardowych obiektów; inżynier DevOps Michael Treacher napisał dobry przykładowy operator (<https://medium.com/@mtreacher/writing-a-kubernetes-operator-a9b86f19bfb9>), który obserwuje tworzone przestrzenie nazw i automatycznie dodaje RoleBinding do każdej nowej przestrzeni nazw (patrz „Kontrola dostępu oparta na rolach (RBAC)” w rozdziale 11., tam uzyskasz więcej informacji na temat powiązań ról).

Ogólnie jednak operatory używają jednego lub więcej niestandardowych obiektów utworzonych za pomocą CRD, których zachowanie jest następnie implementowane przez program komunikujący się z interfejsem API Kubernetes.

Zasoby Ingress

Możesz myśleć o zasobie Ingress jak o module równoważenia obciążenia, który znajduje się przed zasobem Serwis (patrz rysunek 9.1). Ingress odbiera żądania od klientów i wysyła je do zasobu Serwis. Serwis rozsyła je następnie do odpowiednich Podów, na podstawie selektora etykiet (patrz „Serwis” w rozdziale 4.).



Rysunek 9.1. Zasób Ingress

Oto bardzo prosty przykład zasobu Ingress:

```
apiVersion: apps/v1
kind: Ingress
metadata:
  name: demo-ingress
spec:
  backend:
    serviceName: demo-service
    servicePort: 80
```

Powyższy Ingress przekazuje ruch do zasobu Serwis o nazwie `demo-service` działającego na porcie 80. (W rzeczywistości żądania przesyłane są bezpośrednio z Ingress do odpowiedniego Poda, ale warto myśleć o tym koncepcyjnie jak o przesyłaniu za pośrednictwem Serwis).

Sam przykład nie wydaje się bardzo przydatny. Zasoby Ingress mogą robić znacznie więcej.

Reguły Ingress

Podczas gdy zasoby Serwis są przydatne do kierowania ruchem *wewnętrzny* w klastrze (np. z jednej mikrouslugi do drugiej), Ingress jest wykorzystywany do kierowania ruchu *zewnętrznego* do klastra i do odpowiedniej mikrouslugi.

Ingress może przekazywać ruch do różnych usług, w zależności od określonych reguł. Jednym z typowych zastosowań jest kierowanie żądań do różnych miejsc, w zależności od adresu URL żądania (żądania zwane są *fanout*):

```
apiVersion: apps/v1
kind: Ingress
metadata:
  name: fanout-ingress
spec:
  rules:
  - http:
    paths:
    - path: /hello
      backend:
        serviceName: hello
        servicePort: 80
    - path: /goodbye
      backend:
        serviceName: goodbye
        servicePort: 80
```

Znajdują one wiele zastosowań. Wysoko niezawodne moduły równoważące obciążenie mogą być drogie, więc dzięki fanoutowi Ingress możesz mieć jeden moduł równoważenia obciążenia (i powiązany Ingress) kierujący ruch do dużej liczby usług.

Nie ograniczasz się tylko do routingu opartego na adresach URL; możesz także użyć nagłówka HTTP Host (odpowiednik praktyki zwanej *wirtualnym hostingiem opartym na nazwie*). Żądania dotyczące witryny z różnymi domenami (np. *example.com*) zostaną przekierowane do odpowiedniego zasobu Serwis na podstawie domeny.

Zarządzanie połączeniami TLS za pomocą Ingress

Ponadto Ingress może obsługiwać bezpieczne połączenia za pomocą protokołu TLS (protokół wcześniej znany jako SSL). Jeśli masz wiele różnych usług i aplikacji w tej samej domenie, wszystkie one mogą współużytkować certyfikat TLS, a pojedynczy zasób Ingress może zarządzać tymi połączeniami (ang. *TLS termination*):

```
apiVersion: apps/v1
kind: Ingress
metadata:
  name: demo-ingress
spec:
  tls:
  - secretName: demo-tls-secret
    backend:
      serviceName: demo-service
      servicePort: 80
```

Tutaj dodaliśmy nową sekcję `tls`, która instruuje Ingress, aby używał certyfikatu TLS do zabezpieczenia ruchu z klientami. Sam certyfikat jest przechowywany jako zasób Secret (patrz „Obiekty Secret aplikacji Kubernetes” w rozdziale 10.).

Korzystanie z istniejących certyfikatów TLS

Jeśli posiadasz już certyfikat TLS lub zamierzasz go kupić, możesz go wykorzystać w swoim zasobie Ingress. Utwórz Secret, który wygląda następująco:

```
apiVersion: v1
kind: Secret
type: kubernetes.io/tls
metadata:
  name: demo-tls-secret
data:
  tls.crt: LS0tLS1CRUdJTiBDRV...LS0tCg==
  tls.key: LS0tLS1CRUdJTiBSU0...LS0tCg==
```

Umieść zawartość certyfikatu w polu `tls.crt`, a klucz w `tls.key`. Zwykle w przypadku zasobów Secret powinieneś zakodować certyfikat i dane klucza w base64 — przed dodaniem ich do manifestu (patrz „base64” w rozdziale 10.).

Automatyzacja certyfikatów LetsEncrypt za pomocą cert-manager

Jeśli chcesz automatycznie żądać certyfikatów TLS przy użyciu popularnego serwisu LetsEncrypt (lub innego dostawcy certyfikatu ACME) i odnawiać je, możesz skorzystać z narzędzia cert-manager (<http://docs.cert-manager.io/en/latest/>).

Jeśli w klastrze uruchomisz cert-manager, automatycznie zostaną wykryte zasoby Ingress TLS, które nie mają certyfikatu. W wyniku tego zażąda on od określonego dostawcy (np. LetsEncrypt) jednego certyfikatu. Narzędzie cert-manager jest bardziej nowoczesnym następcą popularnego narzędzia kube-lego.

Dokładny sposób obsługi połączeń TLS zależy od czegoś, co nazywa się *kontrolerem Ingress*.

Kontroler Ingress

Kontroler Ingress jest odpowiedzialny za zarządzanie zasobami Ingress w klastrze. W zależności od miejsca, w którym działają klastry, używany kontroler może być inny.

Zazwyczaj dostosowywanie zachowania Ingress odbywa się przez dodanie określonych adnotacji rozpoznawanych przez kontroler Ingress.

Klastry działające w Google GKE mają opcję korzystania z Google Compute Load Balancer dla Ingress. AWS ma podobny produkt o nazwie Application Load Balancer. Te zarządzane usługi zapewniają publiczny adres IP, na którym Ingress nasłuchiwa żądań.

Jeśli potrzebujesz skorzystać z Ingress i używasz Kubernetes w Google Cloud lub AWS — są to dobre miejsca do rozpoczęcia pracy. Możesz przeczytać dokumentację każdego produktu w odpowiednich repozytoriach:

- dokumentacja Google Ingress (<https://github.com/kubernetes/ingress-gce>),
- dokumentacja AWS Ingress (<https://github.com/kubernetes-sigs/aws-alb-ingress-controller>).

Możesz także zainstalować i uruchomić własny kontroler Ingress w klastrze, a nawet uruchomić wiele kontrolerów, jeśli chcesz. Oto niektóre popularne opcje.

nginx-ingress (<https://github.com/nginxinc/kubernetes-ingress>)

NGINX od dawna jest popularnym narzędziem służącym do równoważenia obciążenia; działał zanim Kubernetes pojawił się na rynku. Ten kontroler zapewnia wiele możliwości oferowanych przez NGINX. Istnieją inne kontrolery Ingress oparte na NGINX, ale ten jest oficjalny.

Contour (<https://github.com/heptio/contour>)

Contour jest kontrolerem Ingress, który pod spodem faktycznie używa innego narzędzia o nazwie Envoy, aby wysyłać żądania proxy między klientami a Podami.

Traefik (<https://docs.traefik.io/user-guides/crd-acme/>)

Jest to lekkie narzędzie proxy, które może automatycznie zarządzać certyfikatami TLS dla Twojego Ingress.

Każdy z tych kontrolerów ma inne funkcje i zawiera własne instrukcje dotyczące instalacji oraz konfiguracji, a także własne sposoby obsługi tras routingu oraz certyfikatów. Przeczytaj o różnych opcjach i wypróbuj je we własnym klastrze ze swoimi aplikacjami, aby przekonać się, jak działają.

Istio

Istio jest przykładem tego, co często określa się mianem *usługi mesh*; staje się bardzo przydatne, gdy zespoły mają wiele aplikacji i usług, które komunikują się ze sobą. Obsługuje routing i szyfrowanie ruchu sieciowego między usługami oraz dodaje ważne funkcje, takie jak metryki, dzienniki i równoważenie obciążenia.

Istio jest opcjonalnym składnikiem wielu hostowanych klastrów Kubernetes, w tym Google Kubernetes Engine (sprawdź dokumentację swojego dostawcy, aby dowiedzieć się, jak włączyć Istio).

Jeśli chcesz zainstalować Istio w hostowanym przez siebie klastrze, skorzystaj z oficjalnego wykresu Helm Istio (<https://istio.io/v1.5/docs/setup/install/helm/>).

Jeśli Twoje aplikacje w dużej mierze komunikują się, Istio może być warte zbadania. Usługa zasługuje na własną książkę i prawdopodobnie tak będzie, ale w międzyczasie doskonałym źródłem wiedzy jest dokumentacja (<https://istio.io/docs/concepts/what-is-istio/>).

Envoy

Większość zarządzanych usług Kubernetes, takich jak Google Kubernetes Engine, zapewnia pewnego rodzaju integrację chmurowej usługi równoważenia obciążenia. Gdy np. tworzysz Serwis typu *LoadBalancer* w GKE lub Ingress, Google Cloud Load Balancer jest automatycznie budowany i łączony z Twoją usługą.

Te standardowe usługi równoważenia obciążenia w chmurze dobrze się skalują, są bardzo proste i nie dają wielu możliwości konfiguracji. Przykładowo domyślny algorytm równoważenia obciążenia jest zwykle *losowy* (patrz „Serwis” w rozdziale 4.). Żądania przekazywane są losowo do różnych backendów.

Jednak *losowość* nie zawsze jest tym, czego chcesz. Jeśli np. żądania kierowane do Twojej usługi mogą być długotrwałe i obciążają procesor, niektóre z Twoich węzłów zaplecza mogą zostać przeciążone, podczas gdy inne pozostają bezczynne.

Inteligentniejszy algorytm kierowałby żądania do backendu, który jest najmniej zajęty. Zagadnienie to jest znane pod pojęciami *leastconn* lub *LEAST_REQUEST*.

Do takiego bardziej wyrafinowanego równoważenia obciążenia możesz użyć produktu o nazwie Envoy (<https://www.envoyproxy.io/>). Nie jest to część samego Kubernetes, ale jest powszechnie używany z aplikacjami Kubernetes.

Envoy jest wysokowydajnym rozproszonym proxy C ++ zaprojektowanym dla pojedynczych usług i aplikacji, ale może być również używany jako część architektury usług mesh (patrz „Istio” w tym rozdziale).

Programista Mark Vincze napisał świetny post na blogu (<https://blog.markvincze.com/how-to-use-envoy-as-a-load-balancer-in-kubernetes/>) opisujący, jak skonfigurować Envoy w Kubernetes.

Podsumowanie

Ostatecznie wszystko w Kubernetes dotyczy uruchamiania Podów. W związku z tym omówiliśmy je szczegółowo i przepraszamy, jeśli to za dużo. Nie musisz rozumieć wszystkiego ani pamiętać o wszystkim, co omówiliśmy w tym rozdziale, przynajmniej na razie.

Później możesz napotkać problemy, które pomogą Ci rozwiązać bardziej zaawansowane tematy poruszone w tym rozdziale.

Oto podstawowe zagadnienia do zapamiętania.

- Etykiety to pary klucz-wartość, które identyfikują zasoby i mogą być używane z selektorami w celu dopasowania do określonej grupy zasobów.
- Koligacje węzłów przyłączają lub odłączają Pody do lub z węzłów o określonych atrybutach. Przykładowo można określić, że Pod może działać tylko w węźle o określonej strefie dostępności.
- Podczas gdy koligacje węzłów twardych mogą blokować działanie Poda, koligacje węzłów miękkich działają bardziej jako sugestie dla schedulera. Możesz stosować wiele koligacji miękkich z różnymi wartościami wag.
- Koligacje Podów wyrażają preferencję, aby Pody były uruchamiane w tym samym węźle, co inne Pody. Przykładowo Pody, które działają w tym samym węźle, mogą wyrazić to, używając wzajemnej koligacji Podów.
- Antykoligacje Podów odłączają inne Pody, zamiast je dołączać. Przykładowo antykoligacja replik tego samego Poda może pomóc równomiernie rozłożyć repliki w klastrze.

- Skazy to sposób oznaczania węzłów konkretnymi informacjami; zwykle dotyczy problemów lub awarii węzłów. Domyślnie Pody nie będą uruchamiane na węzłach ze skazą.
- Tolerancje pozwalają na uruchomienie Poda w węzłach o określonej skazie. Za pomocą tego mechanizmu można uruchamiać niektóre Pody tylko w dedykowanych węzłach.
- Zasoby DaemonSet pozwalają zaplanować jedną kopię Poda na każdym węźle (np. agenta rejestrującego).
- StatefulSets uruchamiają i zatrzymują repliki Poda w określonej numerowanej sekwencji, umożliwiając adresowanie każdego z nich za pomocą przewidywalnej nazwy DNS. Jest to idealne rozwiązanie dla aplikacji klastrowych, takich jak bazy danych.
- Zasoby Job uruchamiają Pod jeden raz (lub określoną liczbę razy). Podobnie zasoby CronJob okresowo uruchamiają Pod w określonych godzinach.
- Horizontal Pod Autoscaler obserwuje zestaw Podów, próbując zoptymalizować daną metrykę (np. wykorzystanie procesora). Zwiększają lub zmniejszają żądaną liczbę replik, aby osiągnąć określony cel.
- PodPreset mogą wstrzykiwać fragmenty konfiguracji do wszystkich wybranych Podów w czasie ich tworzenia. Przykładowo PodPreset może zostać wykorzystany do zamontowania określonego woluminu na wszystkich pasujących Podach.
- Niestandardowe definicje zasobów (CRD) umożliwiają tworzenie własnych niestandardowych obiektów Kubernetes w celu przechowywania dowolnych danych. Operatory to programy klienckie Kubernetes, które mogą implementować orkiestrację dla określonej aplikacji (np. MySQL).
- Zasoby Ingress kierują żądania do różnych usług, w zależności od zestawu reguł, np. na podstawie adresu URL żądania. Mogą również obsługiwać połączenia TLS dla Twojej aplikacji.
- Istio to narzędzie, które zapewnia zaawansowane funkcje sieciowe dla mikroserwisów i może być instalowane, podobnie jak każda aplikacja Kubernetes, za pomocą Helm.
- Envoy oferuje bardziej wyrafinowane funkcje równoważenia obciążenia niż standardowe usługi rozwiązań chmurowego. Świadczy także obsługę usługi mesh.

Konfiguracja i obiekty Secret

Jeśli chcesz zachować tajemnicę, musisz ją również ukryć przed sobą.

— George Orwell, 1984

Bardzo przydatna jest możliwość oddzielenia logiki aplikacji Kubernetes od jej *konfiguracji*, tzn. od wszelkich wartości lub ustawień, które mogą ulec zmianie w trakcie użytkowania aplikacji. Wartości konfiguracyjne zwykle obejmują takie ustawienia jak ustawienia środowiskowe, adresy DNS usług stron trzecich i poświadczania uwierzytelnienia.

Choć możesz umieścić te wartości bezpośrednio w kodzie, nie jest to zbyt elastyczne podejście. Po pierwsze, zmiana wartości konfiguracji wymagałaby całkowitej przebudowy i ponownego wdrożenia aplikacji. O wiele lepiej oddzielić te wartości od kodu i wczytać je z pliku lub ze zmiennych środowiskowych.

Kubernetes oferuje kilka różnych sposobów zarządzania konfiguracją. Jednym z nich jest przekazanie wartości do aplikacji za pomocą zmiennych środowiskowych w specyfikacji Pod (patrz „Zmienne środowiskowe” w rozdziale 8.). Innym rozwiązaniem jest przechowywanie danych konfiguracyjnych bezpośrednio w Kubernetes przy użyciu obiektów ConfigMap i Secret.

W tym rozdziale szczegółowo zbadamy obiekty ConfigMap i Secret oraz przyjrzymy się praktycznym technikom zarządzania konfiguracją i obiektami Secret w aplikacjach — za przykład posłuży aplikacja demo.

ConfigMap

ConfigMap jest podstawowym obiektem służącym do przechowywania danych konfiguracyjnych w Kubernetes. Można go traktować jak zestaw par klucz-wartość przechowujący dane konfiguracyjne. Po utworzeniu ConfigMap możesz dostarczyć te dane do aplikacji przy użyciu pliku lub za pomocą wstrzyknięcia ich do środowiska Poda.

W tym rozdziale przyjrzymy się różnym sposobom umieszczenia danych w ConfigMap, a następnie zbadamy sposoby wyodrębnienia tych danych i wprowadzenia ich do aplikacji Kubernetes.

Tworzenie ConfigMap

Załóżmy, że chcesz utworzyć plik konfiguracyjny YAML o nazwie *config.yaml* w systemie plików Poda, o następującej treści:

```
autoSaveInterval: 60
batchSize: 128
protocols:
  - http
  - https
```

Jak powyższy zestaw wartości zmienić w zasób ConfigMap, który następnie można zastosować w Kubernetes?

Jednym ze sposobów jest określenie tych danych jako dosłownych wartości YAML w manifeście ConfigMap.

Tak wygląda manifest dla obiektu ConfigMap:

```
apiVersion: v1
data:
  config.yaml: |
    autoSaveInterval: 60
    batchSize: 128
    protocols:
      - http
      - https
kind: ConfigMap
metadata:
  name: demo-config
  namespace: demo
```

Możesz utworzyć mapę ConfigMap, pisząc manifest od zera i dodając wartości z *config.yaml* do sekcji data, jak to zrobiliśmy w powyższym przykładzie.

Jednak łatwiejszym sposobem jest wykorzystanie polecenia `kubectl`. Możesz utworzyć ConfigMap bezpośrednio z pliku YAML w następujący sposób:

```
kubectl create configmap demo-config --namespace=demo --from-file=config.yaml
configmap "demo-config" created
```

Aby wyeksportować plik manifestu odpowiadający temu ConfigMap, uruchom polecenie:

```
kubectl get configmap/demo-config --namespace=demo -o yaml
>demo-config.yaml
```

Spowoduje to zapisanie manifestu YAML reprezentującego zasób ConfigMap klastra, w pliku *demo-config.yaml*. Wynikowy plik będzie zawierać dodatkowe informacje, jak np. sekcję status, które możesz chcieć usunąć przed ponownym zastosowaniem konfiguracji (patrz „Eksportowanie zasobów” w rozdziale 7.).

Ustawianie zmiennych środowiskowych z obiektu ConfigMap

Mamy już wymagane dane konfiguracyjne w obiekcie ConfigMap. W jaki sposób możemy wykorzystać te dane w kontenerze? Spójrzmy na przykład z wykorzystaniem naszej aplikacji demonstracyjnej. Kod znajdziesz w katalogu *hello-config-env* repozytorium demo.

Jest to ta sama aplikacja demonstracyjna, z której korzystaliśmy w poprzednich rozdziałach — nasłuchuje żądań HTTP i odpowiada powitaniem (patrz „Oglądamy kod źródłowy” w rozdziale 2.).

Tym razem jednak, zamiast zakodować na stałe napis Hello w aplikacji, chcielibyśmy, by był konfigurowalny. Wprowadziliśmy więc niewielką modyfikację funkcji modułu obsługi, aby odczytała tę wartość ze zmiennej środowiskowej GREETING:

```
func handler(w http.ResponseWriter, r *http.Request) {  
    greeting := os.Getenv("GREETING")  
    fmt.Fprintf(w, "%s, 世界\n", greeting)  
}
```

Nie martw się o szczegóły kodu Go; to tylko wersja demonstracyjna. Wystarczy powiedzieć, że jeśli zmienna środowiskowa GREETING jest obecna podczas działania programu, aplikacja użyje tej wartości w trakcie generowania odpowiedzi na żądanie. Niezależnie od tego, jakiego języka używasz do pisania aplikacji, będziesz w stanie odczytać za jego pomocą zmienne środowiskowe.

Teraz utworzymy obiekt ConfigMap do przechowywania wartości powitania. Plik manifestu dla ConfigMap, wraz ze zmodyfikowaną aplikacją Go, znajduje się w katalogu *helloconfig-env* repozytorium demo.

Wygląda to tak:

```
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: demo-config  
data:  
  greeting: Hola
```

Aby dane te były widoczne w środowisku kontenera, musimy nieco zmodyfikować Deployment. Oto odpowiednia część obiektu Deployment demo:

```
spec:  
  containers:  
    - name: demo  
      image: cloudnative/demo:hello-config-env  
      ports:  
        - containerPort: 8888  
      env:  
        - name: GREETING  
          valueFrom:  
            configMapKeyRef:  
              name: demo-config  
              key: greeting
```

Pamiętaj, że używamy innego tagu obrazu kontenera niż w poprzednich przykładach (patrz „Identyfikatory obrazu” w rozdziale 8.). Obraz oznaczony tagiem hello-config-env zawiera zmodyfikowaną wersję aplikacji demo, która odczytuje zmienną GREETING: cloudnative /demo: hello-config-env.

Drugim interesującym punktem jest sekcja env. W punkcie „Zmienne środowiskowe” w rozdziale 8. dowiedziałeś się, że możesz tworzyć zmienne środowiskowe o wartościach literalnych — dodając parę nazwa-wartość.

Wciąż mamy tutaj pole name, ale zamiast wartości podaliśmy valueFrom. To dla Kubernetes informacja, że zamiast brać dosłowną wartość zmiennej, powinien poszukać jej w innym miejscu.

configMapKeyRef nakazuje mu odwoływać się do określonego klucza w określonym ConfigMap. Nazwą tego ConfigMap jest config-demo, a kluczem, który chcemy sprawdzić, jest greeting. Dane te utworzyliśmy za pomocą manifestu ConfigMap, więc powinny być teraz dostępne do odczytu dla środowiska kontenera.

Jeśli ConfigMap nie istnieje, Deployment nie będzie mógł zostać uruchomiony (jego Pod pokaże status CreateContainerConfigError).

To wszystko, czego potrzebujesz, aby zaktualizowana aplikacja działała. Wdroż więc manifesty w klastrze Kubernetes. W katalogu repozytorium demo uruchom następującą komendę:

```
kubectl apply -f hello-config-env/k8s/  
configmap "demo-config" created  
deployment.extensions "demo" created
```

Tak jak poprzednio, aby zobaczyć aplikację w przeglądarce internetowej, musisz przekierować port lokalny do portu Poda 8888:

```
kubectl port-forward deploy/demo 9999:8888  
Forwarding from 127.0.0.1:9999 -> 8888  
Forwarding from [::1]:9999 -> 8888
```

(Tym razem nie zwracaliśmy sobie głowy tworzeniem zasobu Serwis; podczas gdy ty używałbyś zasobu Serwis w przypadku prawdziwej aplikacji produkcyjnej, w tym przykładzie skorzystaliśmy z polecenia kubectl w celu przekazania portu lokalnego bezpośrednio do obiektu Deployment demo).

Jeśli w przeglądarce wpiszesz adres <http://localhost: 9999/>, powinieneś zobaczyć napis (jeśli wszystko jest w porządku):

Hola, 世界

Ćwiczenie

W innym terminalu (musisz pozostawić uruchomione polecenie kubectl port-forward) edytuj plik configmap.yaml, aby zmienić powitanie. Ponownie załaduj plik za pomocą polecenia kubectl. Odśwież przeglądarkę internetową. Czy pozdrowienie się zmienia? Jeśli nie, dlaczego? Co musisz zrobić, aby aplikacja odczytała zaktualizowaną wartość? (Rozwiążanie znajdziesz w punkcie „Aktualizacja Podów po zmianie konfiguracji” w tym rozdziale).

Ustawianie środowiska za pomocą ConfigMap

Jak widziałeś w poprzednim przykładzie, za pomocą kluczy ConfigMap można ustawić jedną lub dwie zmienne środowiskowe. Może to być uciążliwe dla dużej liczby zmiennych.

Na szczęście istnieje prosty sposób na pobranie wszystkich kluczów z ConfigMap i przekształcenie ich w zmienne środowiskowe za pomocą envFrom:

```
spec:  
  containers:  
    - name: demo
```

```
image: cloudnatived/demo:hello-config-env
ports:
  - containerPort: 8888
envFrom:
  - configMapRef:
      name: demo-config
```

Teraz każde ustawienie w `demo-config` ConfigMap będzie zmienną w środowisku kontenera. Ponieważ w naszym przykładzie nazwą klucza ConfigMap jest `greeting`, zmienna środowiskowa będzie się również tak nazywać. Aby nazwy zmiennych środowiskowych były pisane dużymi literami (gdy używasz `envFrom`), zmień je w ConfigMap.

Możesz również ustawić inne zmienne środowiskowe dla kontenera w normalny sposób, za pomocą `env`: albo przez umieszczenie literalnych wartości w pliku manifestu, albo za pomocą `Config Map → KeyRef` — tak jak w naszym poprzednim przykładzie. Aby ustawić zmienne środowiskowe, Kubernetes pozwala używać `env`, `env From` lub obu naraz.

Jeśli zmienna ustawiona w `env` ma taką samą nazwę jak ustawiona w `envFrom`, to będzie miała pierwszeństwo. Jeśli np. ustawisz zmienną `GREETING` zarówno w `env`, jak i `ConfigMap`, do której odwołuje się `envFrom`, wartość określona w `env` zastąpi tę z `ConfigMap`.

Używanie zmiennych środowiskowych w argumentach poleceń

Chociaż użytkczne jest umieszczanie danych konfiguracyjnych w środowisku kontenera, czasem trzeba podać je jako argumenty polecenia.

Możesz to zrobić, pobierając zmienne środowiskowe z `ConfigMap`, jak w poprzednim przykładzie, ale używając specjalnej składni Kubernetes `$(VARIABLE)`.

W katalogu `hello-config-args` repozytorium `demo` znajdziesz plik `deployment.yaml` z poniższą zawartością:

```
spec:
  containers:
    - name: demo
      image: cloudnatived/demo:hello-config-args
      args:
        - "-greeting"
        - "$(GREETING)"
      ports:
        - containerPort: 8888
      env:
        - name: GREETING
          valueFrom:
            configMapKeyRef:
              name: demo-config
              key: greeting
```

W specyfikacji kontenera dodaliśmy pole `args`, które przekaże nasze niestandardowe argumenty do domyślnego punktu wejścia kontenera (`/bin/demo`).

Kubernetes zastępuje wszystko w postaci `$(VARIABLE)` w manifeście wartością zmiennej środowiskowej `VARIABLE`. Ponieważ utworzyliśmy zmienną `GREETING` i ustawiliśmy jej wartość z `ConfigMap`, jest ona dostępna w wierszu poleceń kontenera.

Po zastosowaniu tych manifestów wartość GREETING zostanie przekazana do aplikacji demo:

```
kubectl apply -f hello-config-args/k8s/
configmap "demo-config" configured
deployment.extensions "demo" configured
```

Powinieneś zobaczyć efekt w swojej przeglądarce:

Salut, 世界

Tworzenie plików konfiguracji z ConfigMaps

Pokazaliśmy kilka różnych sposobów przenoszenia danych z ConfigMaps do aplikacji: przy użyciu środowiska i za pomocą wiersza polecenia kontenera. Jednak bardziej złożone aplikacje często oczekują odczytu swojej konfiguracji z plików.

Na szczęście Kubernetes umożliwia tworzenie takich plików bezpośrednio z ConfigMap. Po pierwsze, zmieńmy nasz ConfigMap tak, aby zamiast jednego klucza przechowywał pełny plik YAML (który zawiera tylko jeden klucz):

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  config: |
    greeting: Buongiorno
```

Zamiast ustawać klucz greeting, tak jak w poprzednim przykładzie, tworzymy nowy klucz o nazwie config i przypisujemy mu blok danych (symbol potoku | w YAML wskazuje, że to, co po nim występuje, to blok surowych danych). Oto te dane:

```
greeting: Buongiorno
```

Może to być format YAML, ale też JSON, TOML, zwykły tekst lub dowolny inny format. Cokolwiek to jest, Kubernetes ostatecznie zapisze cały blok danych, tak jak jest, do pliku na naszym kontenerze.

Teraz, gdy przechowujemy niezbędne dane, wróźmy je w Kubernetes. W katalogu pliku *hello-config-file* repozytorium demo znajdziesz szablon Deployment zawierający poniższy kod:

```
spec:
  containers:
    - name: demo
      image: cloudnativated/demo:hello-config-file
      ports:
        - containerPort: 8888
      volumeMounts:
        - mountPath: /config/
          name: demo-config-volume
          readOnly: true
  volumes:
    - name: demo-config-volume
      configMap:
        name: demo-config
        items:
          - key: config
            path: demo.yaml
```

Patrząc na sekcję volumes, możesz zobaczyć, że tworzymy wolumin o nazwie demo-config-volume, za pomocą demo-config z ConfigMap.

W sekcji volumeMounts kontenera montujemy ten wolumin w ścieżce mountPath: /config/, wybieramy klucz config i zapisujemy go do pliku demo.yaml. W wyniku tego Kubernetes utworzy plik w kontenerze, w ścieżce /config/demo.yaml, zawierający dane demo-config w formacie YAML:

```
greeting: Buongiorno
```

Aplikacja demo po uruchomieniu odczyta swoją konfigurację z tego pliku. Tak jak poprzednio, zastosuj manifesty za pomocą poniższego polecenia:

```
kubectl apply -f hello-config-file/k8s/
configmap "demo-config" configured
deployment.extensions "demo" configured
```

Powinieneś zobaczyć wynik w swojej przeglądarce:

Buongiorno, 世界

Jeśli chcesz zobaczyć, jak wyglądają dane ConfigMap w klastrze, uruchom następujące polecenie:

```
kubectl describe configmap/demo-config
Name:           demo-config
Namespace:      default
Labels:         <none>
Annotations:
kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1",
"data":{"config":"greeting: Buongiorno\\n"}, "kind":"ConfigMap", "metadata": {
"annotations":{}, "name": "demo-config", "namespace": "default...
Data
====
config:
greeting: Buongiorno

Events: <none>
```

Jeśli zaktualizujesz ConfigMap i zmienisz jego wartości, odpowiedni plik (/config/demo.yaml w naszym przykładzie) zostanie zaktualizowany automatycznie. Niektóre aplikacje mogą automatycznie wykryć, że ich plik konfiguracyjny zmienił się, i załadują go ponownie; inne mogą tego nie zrobić.

Jedną z opcji aktualizacji zmian w aplikacji jest jej ponowne wdrożenie (patrz „Aktualizacja Podów po zmianie konfiguracji” w tym rozdziale). Jednak może to nie być konieczne. Aplikacja może ponownie załadować konfigurację w czasie działania za pomocą sygnału uniksowego (np. SIGHUP) lub z wykorzystaniem odpowiedniego polecenia w kontenerze.

Aktualizacja Podów po zmianie konfiguracji

Założymy, że masz w klastrze działający Deployment i chcesz zmienić niektóre wartości w jego ConfigMap. Jeśli korzystasz z wykresu Helm (patrz „Helm: menadżer pakietów Kubernetes” w rozdziale 4.), istnieje pewna sztuczka polegająca na tym, że automatycznie wykrywana jest zmiana konfiguracji i ponownie ładowane są Pody. Dodaj tę adnotację do specyfikacji Deployment:

```
checksum/config: {{ include (print $.Template.BasePath "/configmap.yaml") }}  
| sha256sum }}
```

Ponieważ szablon Deployment zawiera teraz sumę kontrolną ustawień konfiguracyjnych, jeśli te ustawienia się zmienią, zmieni się również suma kontrolna. Po wykonaniu polecenia `helm upgrade` Helm wykryje, że specyfikacja Deployment uległa zmianie i ponownie uruchomi wszystkie Pody.

Obiekty Secret aplikacji Kubernetes

Widzieliśmy, że obiekt ConfigMap zapewnia elastyczny sposób przechowywania i dostępu do danych konfiguracyjnych w klastrze. Jednak większość aplikacji ma pewne tajne i poufne dane konfiguracyjne, takie jak hasła lub klucze API. Chociaż moglibyśmy użyć ConfigMaps do ich przechowywania, nie jest to idealne rozwiązanie.

Zamiast tego Kubernetes zapewnia specjalny typ obiektu przeznaczony do przechowywania tajnych danych, jest to Secret. Zobaczmy przykład korzystania z niego w aplikacji demo.

Oto manifest Kubernetes dla obiektu Secret (patrz `hello-secret-env/k8s/secret.yaml`):

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: demo-secret  
stringData:  
  magicWord: xyzzy
```

W tym przykładzie tajnym kluczem jest `magicWord`, a tajną wartością jest słowo `xyzzy` ([https://en.wikipedia.org/wiki/Xyzz_\(computing\)](https://en.wikipedia.org/wiki/Xyzz_(computing)) — bardzo przydatne słowo w informatyce).

Podobnie jak w przypadku ConfigMap, możesz umieścić wiele kluczów i wartości w obiektach Secret. Tutaj, dla uproszczenia, używamy tylko jednej pary klucz-wartość.

Używanie obiektów Secret jako zmiennych środowiskowych

Podobnie jak w przypadku ConfigMap, obiekty Secret mogą być widoczne dla kontenerów poprzez umieszczenie ich w zmiennych środowiskowych lub zamontowanie ich jako pliku w systemie plików kontenera. W tym przykładzie ustawimy zmienną środowiskową na wartość z Secret:

```
spec:  
  containers:  
    - name: demo  
      image: cloudnativified/demo:hello-secret-env  
      ports:  
        - containerPort: 8888  
      env:  
        - name: MAGIC_WORD  
          valueFrom:  
            secretKeyRef:  
              name: demo-secret  
              key: magicWord
```

Ustawiamy zmienną środowiskową MAGIC_WORD dokładnie tak, jak robiliśmy to podczas korzystania z ConfigMap, tyle że teraz zamiast configMapKeyRef korzystamy z secretKeyRef (patrz „Ustawianie zmiennych środowiskowych z obiektu ConfigMap” w tym rozdziale).

Uruchom następującą komendę w katalogu repozytorium demo, aby zastosować te manifesty:

```
kubectl apply -f hello-secret-env/k8s/  
deployment.extensions "demo" configured  
secret "demo-secret" created
```

Tak jak poprzednio, przekieruj port lokalny do obiektu Deployment, abyś mógł zobaczyć wyniki w swojej przeglądarce internetowej:

```
kubectl port-forward deploy/demo 9999:8888  
Forwarding from 127.0.0.1:9999 -> 8888  
Forwarding from [::1]:9999 -> 8888
```

Przejdz do adresu <http://localhost:9999/>, powinieneś zobaczyć:

```
The magic word is "xyzzy"
```

Zapisywanie obiektów Secret do plików

W tym przykładzie zamontujemy Secret na kontenerze jako plik. Kod dla tego przykładu znajdziesz w folderze *hello-secret-file* repozytorium demo.

Aby zamontować Secret w pliku w kontenerze, używamy takiego obiektu Deployment:

```
spec:  
  containers:  
    - name: demo  
      image: cloudnativelabs/demo:hello-secret-file  
      ports:  
        - containerPort: 8888  
  volumeMounts:  
    - name: demo-secret-volume  
      mountPath: "/secrets/"  
      readOnly: true  
  volumes:  
    - name: demo-secret-volume  
      secret:  
        secretName: demo-secret
```

Podobnie jak w punkcie „Tworzenie plików konfiguracji z ConfigMap” w tym rozdziale, budujemy wolumin (demo-secret-wolumen w tym przykładzie) i montujemy go na kontenerze w sekcji `volumeMounts` specyfikacji. Pole `mountPath` ma wartość `/secrets`. Kubernetes utworzy w tym katalogu jeden plik dla każdej pary klucz-wartość zdefiniowanej w obiekcie Secret.

W przykładzie obieku Secret zdefiniowaliśmy tylko jedną parę klucz-wartość o nazwie `magicWord`. Manifest utworzy na kontenerze plik tylko do odczytu `/secrets/magicWord`, a zawartość pliku będzie zawierała tajne dane.

Jeśli zastosujesz ten manifest w taki sam sposób jak w poprzednim przykładzie, powinieneś zobaczyć ten sam wynik:

```
The magic word is "xyzzy"
```

Odczyt obiektów Secret

Wcześniej skorzystaliśmy z polecenia kubectl describe, aby zobaczyć dane w ConfigMap. Czy możemy zrobić tak samo z obiektem Secret?

```
kubectl describe secret/demo-secret
Name: demo-secret
Namespace: default
Labels: <none>
Annotations:
Type: Opaque
Data
=====
magicWord: 5 bytes
```

Zauważ, że tym razem rzeczywiste dane nie są wyświetlane. Obiekty Secret Kubernetes są nieprzezroczyste (ang. opaque), co oznacza, że nie są pokazywane w kubectl describe, logach lub terminalu. Zapobiega to przypadkowemu ujawnieniu tajnych danych.

Zakodowaną wersję tajnych danych w formacie YAML możesz zobaczyć, używając polecenia kubectl get:

```
kubectl get secret/demo-secret -o yaml
apiVersion: v1
data:
  magicWord: eHl6enk=
kind: Secret
metadata:
...
type: Opaque
```

base64

Co to jest eHl6enk=? To nie przypomina naszych oryginalnych tajnych danych. W rzeczywistości jest to reprezentacja obiektu Secret w *base64*. Base64 to schemat kodowania dowolnych danych binarnych do postaci ciągu znaków.

Ponieważ tajne dane mogą być niedrukowalnymi danymi binarnymi (np. kluczem szyfrującym TLS), obiekty Secret zawsze są przechowywane w formacie base64.

Tekst eHl6enk= to nasze tajne słowo xyzzy zakodowane w standardzie base64. Możesz to sprawdzić za pomocą komendy base64 --decode w terminalu:

```
echo "eHl6enk=" | base64 --decode
xyzzy
```

Chociaż Kubernetes chroni Cię przed przypadkowym wyświetleniem tajnych danych na terminalu lub w logach, jeśli posiadasz uprawnienia do odczytu obiektów Secret w określonej przestrzeni nazw, możesz pobrać dane w formacie base64, a następnie je zdekodować.

Jeśli chcesz zakodować jakiś tekst do base64 (np. aby dodać go do Secret), użyj narzędzia base64 z flagą -n by zapobiec dodaniu znaku nowej linii:

```
echo -n xyzzy | base64
eHl6enkK
```

Dostęp do obiektów Secret

Kto może czytać lub edytować obiekty Secret? Jest to kontrolowane przez mechanizm kontroli dostępu Kubernetes, RBAC, który omówimy bardziej szczegółowo w podrozdziale „Kontrola dostępu oparta na rolach (RBAC)” w rozdziale 11. Jeśli używasz klastra, który nie obsługuje RBAC lub nie ma włączonej takiej opcji, wszystkie obiekty Secret są dostępne dla dowolnego użytkownika lub dowolnego kontenera. (Jak później wyjaśnimy, absolutnie nie powinieneś uruchamiać żadnego klastra produkcyjnego bez RBAC).

Szyfrowanie w stanie spoczynku

A co zrobić z przypadkiem, gdy ktoś ma dostęp do bazy danych *etcd*, w której przechowywane są wszystkie informacje Kubernetes? Czy może uzyskać dostęp do tajnych danych, nawet bez uprawnień API do odczytu obiektu Secret?

Począwszy od wersji 1.7, w Kubernetes jest obsługiwane szyfrowanie w *stanie spoczynku* (ang. *encryption at rest*). Oznacza to, że tajne dane w bazie danych *etcd* są przechowywane zaszyfrowane na dysku i nieczytelne nawet dla osób, które mają bezpośredni dostęp do bazy danych. Tylko serwer API Kubernetes ma klucz do odszyfrowania tych danych. W prawidłowo skonfigurowanym klastrze szyfrowanie w stanie spoczynku powinno być włączone.

Możesz sprawdzić, czy szyfrowanie w stanie spoczynku jest włączone w klastrze. Wpisz polecenie:

```
kubectl describe pod -n kube-system -l component=kube-apiserver |grep encryption  
--experimental-encryption-provider-config=...
```

Jeśli nie widzisz flagi `experimental-encryption-provider-config`, szyfrowanie w stanie spoczynku nie jest włączone. (Jeśli używasz Google Kubernetes Engine lub innych zarządzanych usług Kubernetes, Twoje dane są szyfrowane przy użyciu innego mechanizmu i nie zobaczysz tej flagi. Skontaktuj się z dostawcą Kubernetes, aby dowiedzieć się, czy dane *etcd* są zaszyfrowane).

Przechowywanie obiektów Secret

Czasami będziesz posiadać zasoby Kubernetes, których nigdy nie będziesz chciał usunąć z klastra — np. szczególnie ważny Secret. Za pomocą adnotacji specyficznej dla narzędzia Helm możesz zapobiec usunięciu zasobu:

```
kind: Secret  
metadata:  
  annotations:  
    "helm.sh/resource-policy": keep
```

Strategie zarządzania obiektami Secret

W przykładzie z poprzedniego punktu nasze tajne dane były chronione przed nieautoryzowanym dostępem, gdy były przechowywane w klastrze. Jednak w naszych plikach manifestu tajne dane były przechowywane jako zwykły tekst.

Nigdy nie należy ujawniać takich tajnych danych w plikach, które są umieszczane w repozytoriach kodu źródłowego. Jak więc bezpiecznie zarządzać tajnymi danymi i przechowywać je, zanim zostaną zastosowane w klastrze Kubernetes?

Bez względu na to, jakie narzędzie lub jaką strategię wybierzesz do zarządzania tajnymi danymi w swoich aplikacjach, będziesz potrzebować odpowiedzi na następujące pytania.

1. Gdzie przechowujesz obiekty Secret, zachowując wysoką niezawodność?
2. W jaki sposób udostępniasz obiekty Secret działającym aplikacjom?
3. Co musi się stać z działającymi aplikacjami, gdy zmieniasz Secret?

W tym podrozdziale przyjrzymy się trzem najpopularniejszym strategiom zarządzania obiektami Secret i sprawdzimy, w jaki sposób każda z nich odpowiada na powyższe pytania.

Szyfrowanie Secret w systemach kontroli wersji

Pierwszą opcją jest przechowywanie obiektów Secret w postaci zaszyfrowanej bezpośrednio w kodzie, w repozytoriach kontroli wersji i odszyfrowanie ich w czasie wdrażania.

Prawdopodobnie jest to najprostsze rozwiązanie. Obiekty te są umieszczane bezpośrednio w repozytoriach kodu źródłowego, ale nigdy w postaci zwykłego tekstu. Zamiast tego są szyfrowane w formie, którą można odszyfrować tylko przy użyciu pewnego zaufanego klucza.

Podczas wdrażania aplikacji obiekty Secret są odszyfrowywane tuż przed zastosowaniem manifestów Kubernetes do klastra. Aplikacja może następnie odczytywać te obiekty i używać ich tak samo jak w przypadku innych danych konfiguracyjnych.

Szyfrowanie obiektów Secret w systemach kontroli wersji pozwala przeglądać i śledzić zmiany w nich wykonywane, podobnie jak w przypadku śledzenia zmian w kodzie aplikacji. I tak długo, jak repozytoria kontroli wersji są wysoce niezawodne, Twoje obiekty Secret będą również wysoce niezawodne.

Aby zmienić obiekty Secret, po prostu odszyfruj je w lokalnej kopii źródła, zaktualizuj, ponownie zaszyfruj i zatwierdź zmianę w systemie kontroli wersji.

Chociaż strategia ta jest łatwa do wdrożenia i nie ma żadnych zależności z wyjątkiem klucza i narzędzia szyfrowania/deszyfrowania (patrz „Szyfrowanie obiektów Secret za pomocą Sops” w tym rozdziale), istnieje jedna potencjalna wada. Jeśli ten sam klucz jest używany przez wiele aplikacji, wszystkie potrzebują jego kopii w kodzie źródłowym. Oznacza to, że więcej pracy kosztuje aktualizacja klucza, ponieważ musisz upewnić się, że znalazłeś i zmieniłeś wszystkie jego wystąpienia.

Istnieje również poważne ryzyko przypadkowego przekazania obiektów w postaci zwykłego tekstu do systemu kontroli wersji. Błędy się zdarzają i nawet w przypadku prywatnych repozytoriów kontroli wersji każdy taki Secret powinien zostać uznany za skompromitowany i należy go jak najszybciej zmienić. Możesz ograniczyć dostęp do klucza szyfrowania tylko do niektórych osób, zamiast rozdawać go wszystkim programistom.

Niemniej jednak strategia szyfrowania obiektów Secret w kodzie źródłowym jest dobrym rozwiązaniem dla małych organizacji z niekrytycznymi obiektami. Wymaga stosunko małej pracy oraz jest

łatwa w konfiguracji, a jednocześnie wystarczająco elastyczna, aby obsługiwać wiele aplikacji i różne rodzaje tajnych danych. W końcowej części tego rozdziału przedstawimy kilka opcji narzędzi szyfrujących/deszyfrujących, których możesz użyć do tego celu. Najpierw jednak opiszemy krótko inne strategie zarządzania obiektami Secret.

Zdalne przechowywanie Secret

Inną opcję zarządzania obiektami Secret jest przechowywanie ich w pliku (lub wielu plikach) w zdalnym, bezpiecznym magazynie plików, takim jak AWS S3 lub Google Cloud Storage. Podczas wdrażania aplikacji pliki zostaną pobrane, odszyfrowane i dostarczone do aplikacji. Jest to podobne rozwiązanie do *opcji szyfrowania Secret w systemie kontroli wersji*. Z tym wyjątkiem, że obiekty Secret są przechowywane centralnie, a nie w kodzie źródłowym. Możesz użyć tego samego narzędzia do szyfrowania/deszyfrowania dla obu strategii.

To rozwiązuje problem duplikowania Secret w wielu repozytoriach kodu, ale potrzeba trochę dodatkowej inżynierii i koordynacji, aby ściągnąć odpowiedni plik Secret w czasie wdrażania. Są pewne korzyści z używania dedykowanego narzędzia do zarządzania obiektami Secret, bez konieczności konfigurowania dodatkowego komponentu oprogramowania i zarządzania nim lub refaktoryzacji aplikacji.

W związku z tym, że Twoje obiekty Secret nie znajdują się w systemie kontroli wersji, będziesz potrzebować procesu do obsługi aktualizacji tych obiektów w uporządkowany sposób — najlepiej z logowaniem (kto co zmienił, kiedy i dlaczego) — oraz pewnego rodzaju równoważnej procedury kontroli zmian, aby przeglądać żądania pobrań oraz je zatwierdzać.

Dedykowane narzędzie do zarządzania obiektami Secret

Szyfrowanie Secret w kodzie źródłowym i *przechowywanie ich w formie zdalnej* jest dobre dla większości organizacji. Jednak do większych rozwiązań możesz pomyśleć o skorzystaniu z dedykowanego narzędzia do zarządzania tymi obietami, takiego jak Vault Hashicorp, Squares Keywhiz, AWS Secrets Manager lub Azure Vault Key Vault. Narzędzia te obsługują bezpieczne przechowywanie wszystkich obiektów Secret aplikacji w jednym centralnym miejscu w wysoce niezawodny sposób. Mogą także kontrolować, którzy użytkownicy i jakie konta usług mają uprawnienia do dodawania, usuwania, zmian lub przeglądania Secret.

W systemie zarządzania obiektami Secret wszystkie działania są kontrolowane i weryfikowane, co ułatwia analizę naruszeń bezpieczeństwa i wykazanie zgodności z przepisami. Niektóre z tych narzędzi zapewniają także możliwość automatycznej zmiany Secret w regularnych odstępach czasu. To nie tylko dobry pomysł, ale jest to również wymagane przez zasady bezpieczeństwa wielu firm.

W jaki sposób aplikacje pobierają dane z narzędzia do zarządzania Secret? Jednym z powszechnych sposobów jest użycie konta usługi z dostępem tylko do odczytu, dzięki czemu każda aplikacja może tylko odczytywać potrzebne mu Secret. Programiści mogą mieć własne dane uwierzytelniające, z uprawnieniami do odczytu lub zapisu Secret tylko dla aplikacji, za które są odpowiedzialni.

Chociaż centralny system zarządzania obiektami Secret jest najbardziej wydajną i elastyczną opcją, dodaje również znaczną złożoność do infrastruktury. Oprócz konfigurowania i uruchamiania magazynu

Secret, konieczne będzie dodanie narzędzi lub oprogramowania pośredniego do każdej aplikacji i usługi wykorzystującej Secret. Aplikacje można ponownie przebudować lub przeprojektować w celu uzyskania bezpośredniego dostępu do magazynu obiektów Secret, ale może to być droższe i bardziej czasochłonne niż zwykłe dodanie warstwy, która pobiera Secret i umieszcza je w środowisku aplikacji lub w pliku konfiguracyjnym.

Spośród różnych rozwiązań, jednym z najpopularniejszych jest Vault (<https://www.vaultproject.io/>) firmy Hashicorp.

Rekomendacje

Chociaż na pierwszy rzut oka logiczny wydaje się wybór dedykowanego systemu zarządzania obiektami Secret, takiego jak Vault, nie zalecamy rozpoczęcia od tego rozwiązania. Zamiast tego wypróbuj lekkie narzędzie do szyfrowania, takie jak Sops (patrz „Szyfrowanie obiektów Secret za pomocą Sops” w tym rozdziale), szyfrujące Secret bezpośrednio w kodzie źródłowym.

Dlaczego? Cóż, może nie masz tak wielu obiektów Secret do zarządzania. O ile Twoja infrastruktura nie jest bardzo złożona i współzależna (której i tak powinieneś unikać), każda pojedyncza aplikacja powinna potrzebować tylko jednej lub dwóch części danych Secret; będą to klucze API i np. tokeny dla innych usług lub poświadczania dostępu do bazy danych. Jeśli dana aplikacja naprawdę potrzebuje wielu różnych obiektów Secret, możesz rozważyć umieszczenie ich wszystkich w jednym pliku i jego zaszyfrowanie.

Do zarządzania obiektami Secret podchodzimy pragmatycznie, podobnie jak w przypadku większości problemów w tej książce. Jeśli prosty, łatwy w użyciu system rozwiązuje problem, zacznij od niego. Zawsze możesz później przejść do bardziej wydajnej lub skomplikowanej konfiguracji. Na początku projektu często trudno dokładnie określić, ile Secret będzie potrzebnych. Jeśli nie masz pewności, wybierz opcję, która zapewni Ci najszybsze uruchomienie bez ograniczania możliwości wyboru w przyszłości.

Jeśli wiesz od samego początku, że istnieją ograniczenia prawne lub dotyczące zgodności, związane z obsługą obiektów Secret, najlepiej zapoznać się z dedykowanym rozwiązaniem do zarządzania tymi obiektami.

Szyfrowanie obiektów Secret za pomocą Sops

Przy założeniu, że zamierzasz wykonać własne szyfrowanie, przynajmniej na początek potrzebujesz narzędzia szyfrującego, które może współpracować z Twoim kodem źródłowym i plikami danych. Sops (skrót od *secret operations*) z projektu Mozilla jest narzędziem do szyfrowania/deszyfrowania, które może współpracować z plikami YAML, JSON lub plikami binarnymi. Obsługuje także wiele backendów szyfrowania, w tym PGP/GnuPG, Azure Key Vault, AWSs Key Management Service (KMS) i Google Cloud KMS.

Przedstawiamy Sops

Pokażemy teraz, co potrafi Sops. Zamiast szyfrować cały plik, Sops szyfruje tylko poszczególne wartości Secret. Jeśli np. Twój plik tekstowy zawiera:

```
password: foo
```

to kiedy zaszyfrujesz go za pomocą Sops, w pliku będzie:

```
password: ENC[AES256_GCM,data:p673w==,iv:YY=,aad:UQ=,tag:A=]
```

Ułatwia to edycję i przeglądanie kodu, szczególnie w czasie żądań pobierania danych, bez konieczności ich odszyfrowywania.

Odwiedź stronę główną projektu Sops (<https://github.com/mozilla/sops>), aby uzyskać instrukcje dotyczące instalacji i użytkowania.

W pozostałej części tego rozdziału omówimy kilka przykładów użycia Sops. Zobaczmy, jak działa z Kubernetes oraz dodamy do naszej aplikacji demo kilka obiektów Secret zarządzanych przez Sops. Jednak najpierw powinniśmy wspomnieć, że dostępne są inne narzędzia służące do szyfrowania Secret. Jeśli używasz już innego narzędzia, to dobrze; dopóki możesz szyfrować i deszyfrować obiekty Secret w plikach tekstowych w taki sam sposób jak Sops, użyj go.

Jesteśmy fanami narzędzia Helm (jeżeli doszedłeś do tego rozdziału książki — już to wiesz), więc jeśli potrzebujesz zarządzać zaszyfrowanymi obiektami Secret w wykresie Helm, możesz to zrobić za pomocą Sops z użyciem rozszerzenia `helm-secrets`. Gdy uruchomisz polecenie `helm upgrade` lub `helm install`, wtyczka `helm-secrets` odszyfruje Secret do wdrożenia. Aby uzyskać więcej informacji na temat `helm-secrets`, w tym instrukcje instalacji i użytkowania, zapoznaj się z repozytorium GitHub (<https://github.com/futuresimple/helm-secrets>).

Szyfrowanie pliku za pomocą Sops

Wypróbujmy Sops, szyfrując plik. Jak wspomnieliśmy, Sops nie obsługuje samego szyfrowania; przekazuje to do backendu, takiego jak GnuPG (popularna implementacja open source Pretty Good Privacy lub protokołu PGP). W tym przykładzie użyjemy Sops z GnuPG, aby zaszyfrować plik zawierający Secret. Wynikiem końcowym będzie plik, który można bezpiecznie zatwierdzić do systemu kontroli wersji.

Nie zajmiemy się szczegółami działania szyfrowania PGP, ale wiemy, że podobnie jak SSH i TLS, jest to kryptosystem klucza publicznego. Zamiast szyfrować dane za pomocą jednego klucza, w rzeczywistości wykorzystywana jest para kluczy — jeden publiczny, jeden prywatny. Możesz bezpiecznie udostępniać swój klucz publiczny innym osobom, ale nigdy nie powinieneś rozdawać swojego klucza prywatnego.

Wygenerujmy teraz Twoją parę kluczy. Najpierw zainstaluj GnuPG (<https://gnupg.org/download>), jeśli jeszcze go nie masz.

Po zainstalowaniu uruchom poniższe polecenie, aby wygenerować nową parę kluczy:

```
gpg --gen-key
```

Po pomylnym wygenerowaniu klucza zanotuj *odcisk palca* (ang. *fingerprint*) klucza (ciąg cyfr szesnastkowych): to jednoznacznie identyfikuje klucz, którego będziesz potrzebować w następnym kroku.

Teraz, gdy masz parę kluczy, zaszyfrujemy plik przy użyciu Sops i nowego klucza PGP. Jeśli jeszcze tego nie zrobiłeś, musisz zainstalować Sops na swoim komputerze. W tym celu możesz pobrać pliki instalacyjne (<https://github.com/mozilla/sops/releases>) lub zainstalować go za pomocą Go:

```
go get -u go.mozilla.org/sops/cmd/sops
sops -v
sops 3.0.5 (latest)
```

Teraz utworzymy testowy plik Secret, który zaszyfrujemy:

```
echo "password: secret123" > test.yaml
cat test.yaml
password: secret123
```

Na koniec użyj Sops, aby go zaszyfrować. Przekaż swój odcisk palca klucza za pomocą przełącznika --pgp, z usuniętymi spacjami:

```
sops --encrypt --in-place --pgp E0A9AF924D5A0C123F32108EAF3AA2B4935EA0AB
test.yaml cat test.yaml
password: ENC[AES256_GCM,data:Ny220M18JoqP,iv:HMkwA8eFFmdUU1D1e6NTpVgy8v1Qu/
Encrypting Secrets with Sops | 197
6Zqx95Cd/+NL4=,tag:Udg9Wef8coZRbPb0fo0OSA==,type:str]
sops:
...
```

Sukces! Teraz plik test.yaml jest zaszyfrowany. Wartość password jest chroniona hasłem i można ją odszyfrować tylko za pomocą klucza prywatnego. Zauważ również, że Sops dodał kilka metadanych na dole pliku, aby wiedzieć, jak je odszyfrować w przyszłości.

Fajne jest to, że tylko *wartość* password jest szyfrowana. Format pliku YAML zostaje zachowany, zaszyfrowane są tylko dane oznaczone etykietą password. Jeśli w pliku YAML masz długą listę par klucz-wartość, Sops zaszyfruje tylko wartości, pozostawiając klucze w spokoju.

Aby upewnić się, że możemy odzyskać zaszyfrowane dane i sprawdzić, czy są one zgodne z tym, co wprowadziliśmy, uruchom:

```
sops --decrypt test.yaml
You need a passphrase to unlock the secret key for
user: "Justin Domingus <justin@example.com>"
2048-bit RSA key, ID 8200750F, created 2018-07-27 (main key ID 935EA0AB)
Enter passphrase: *highly secret passphrase*

password: secret123
```

Pamiętasz hasło, które wybrałeś podczas generowania pary kluczów? Mamy taką nadzieję, ponieważ musisz je teraz wpisać! Jeśli dobrze je zapamiętałeś, zobaczysz odszyfrowaną wartość password: secret123.

Teraz wiesz, jak korzystać z Sops. Możesz szyfrować dowolne poufne dane w kodzie źródłowym, czy to pliki konfiguracyjne aplikacji, czy zasoby YAML Kubernetes, czy cokolwiek innego.

Kiedy przychodzi czas na wdrożenie aplikacji, użyj Sops w trybie odszyfrowywania, aby utworzyć potrzebne jawne obiekty Secret (pamiętaj jednak o usunięciu tych plików tekstowych i nie umieszczaj ich w systemie kontroli wersji!).

W dalszej części książki pokażemy Ci, jak używać Sops razem z wykresami Helm. Za pomocą Helm możesz nie tylko odszyfrować Secret podczas wdrażania aplikacji, ale także wykorzystać różne zestawy obiektów Secret, w zależności od środowiska wdrażania, np. *produkcyjnego* lub *programistycznego* (patrz „Zarządzanie obiekttami Secret wykresów Helm za pomocą Sops” w rozdziale 12.).

Korzystanie z zaplecza KMS

Jeśli używasz usługi Amazon KMS lub Google Cloud KMS do zarządzania kluczami w chmurze, możesz również korzystać z nich razem z Sops. Korzystanie z klucza KMS działa dokładnie tak samo jak w naszym przykładzie PGP — będą tylko inne metadane w pliku. Sekcja `sops`: na dole pliku może wyglądać tak:

```
sops:  
  kms:  
    - created_at: 1441570389.775376  
      enc: CiC....Pm1Hm  
      arn: arn:aws:kms:us-east-1:656532927350:key/920aff2e...
```

Podobnie jak w naszym przykładzie PGP, identyfikator klucza (`arn: aws: kms ...`) jest osadzony w pliku, zatem Sops ma informację, jak go później odszyfrować.

Podsumowanie

Konfiguracja i obiekty Secret to jeden z tematów, o który ludzie pytają nas najczęściej, gdy omawiamy Kubernetes. Cieszymy się, że mogliśmy poświęcić temu zagadnieniu rozdział i przedstawić kilka sposobów łączenia aplikacji z potrzebnymi ustawieniami i danymi.

Oto najważniejsze rzeczy, których się nauczyłeś.

- Oddziel swoje dane konfiguracyjne od kodu aplikacji i wdróż je za pomocą Kubernetes ConfigMap i Secret. W ten sposób nie musisz ponownie wdrażać aplikacji za każdym razem, gdy zmienisz hasło.
- Możesz pobrać dane do ConfigMap, pisząc je bezpośrednio w pliku manifestu Kubernetes lub za pomocą polecenia `kubectl`, aby przekonwertować istniejący plik YAML na specyfikację ConfigMap.
- Gdy dane znajdą się w ConfigMap, możesz wstawić je do środowiska kontenera lub do argumentów polecenia. Alternatywnie możesz zapisać dane do pliku zamontowanego w kontenerze.
- Obiekty Secret działają podobnie jak ConfigMaps. Z tym wyjątkiem, że dane są szyfrowane w stanie spoczynku i w wynikach polecenia `kubectl` występują w postaci zakodowanej.
- Prostym, elastycznym sposobem zarządzania obiekttami Secret jest przechowywanie ich bezpośrednio w repozytorium kodu źródłowego w postaci zaszyfrowanej za pomocą Sops lub innego narzędzia do szyfrowania opartego na tekście.

- Nie zastanawiaj się nad zarządzaniem obiektami Secret, zwłaszcza na początku. Zaczni od czegoś prostego, co jest łatwe do skonfigurowania dla programistów.
- Tam, gdzie wiele aplikacji współdzieli obiekty Secret, możesz je przechowywać (zaszyfrowane) w chmurze i pobierać w czasie wdrażania.
- Do zarządzania obiektami Secret na poziomie przedsiębiorstwa potrzebujesz dedykowanej usługi, takiej jak Vault. Jednak nie zaczynaj od Vault, ponieważ możesz jej nie potrzebować. Zawsze możesz przenieść się do Vault później.
- Sops to narzędzie do szyfrowania, które działa z plikami klucz-wartość, takimi jak YAML i JSON. Klucz szyfrujący możesz uzyskać za pomocą GnuPG lub usług zarządzania kluczami w chmurze, takich jak Amazon KMS i Google Cloud KMS.

Bezpieczeństwo i kopia zapasowa

Jeśli uważasz, że technologia może rozwiązać Twoje problemy z bezpieczeństwem, oznacza to, że nie rozumiesz problemów i technologii.

— Bruce Schneier, *Applied Cryptography*

W tym rozdziale omówimy mechanizmy bezpieczeństwa i kontroli dostępu w Kubernetes, w tym kontrolę dostępu opartą na rolach (RBAC), przedstawimy niektóre narzędzia i usługi wyszukujące luki w systemie oraz wyjaśnimy, jak wykonać kopię zapasową danych i stanu Kubernetes (a co ważniejsze, jak je przywrócić). Przyjrzymy się również pewnym przydatnym sposobom uzyskiwania informacji o tym, co dzieje się w Twoim klastrze.

Kontrola dostępu i uprawnienia

Na początku małe firmy technologiczne zatrudniają kilku pracowników i każdy z nich posiada uprawnienia administratora do każdego systemu.

Jednak wraz z rozwojem organizacji staje się jasne, że nie wszyscy powinni mieć uprawnienia administratora, gdyż każdy może zbyt łatwo popełnić błąd i zmienić coś, czego nie powinien. To samo dotyczy Kubernetes.

Zarządzanie dostępem przez kластer

Jedną z najłatwiejszych i najskuteczniejszych rzeczy, które możesz zrobić, aby zabezpieczyć klasterek Kubernetes, jest ograniczenie dostępu użytkowników. Zasadniczo istnieją dwie grupy osób, które muszą uzyskać dostęp do klastrów Kubernetes; są to *operatorzy klastrów* i *programiści aplikacji*. Często potrzebują różnych uprawnień w ramach swojej pracy.

Ponadto możesz mieć wiele środowisk, takich jak produkcja i rozwój oprogramowania. Te oddzielne środowiska będą wymagać różnych zasad, w zależności od organizacji. Produkcja może być ograniczona tylko do niektórych osób, podczas gdy środowisko rozwoju oprogramowania może być otwarte dla szerszej grupy inżynierów.

Jak napisaliśmy w podpunkcie „Czy potrzebuję wielu klastrów?” w rozdziale 6., często dobrym pomysłem jest posiadanie oddzielnego klastru do produkcji, rozwoju oprogramowania lub testowania.

Jeśli ktoś w środowisku programistycznym wykona zmianę, która obniży wydajność węzłów klastra, nie wpłynie to na produkcję.

Jeśli żaden zespół nie powinien mieć dostępu do innego zespołu i procesu wdrażania, każdy zespół może mieć własny dedykowany klasztar, ale nie powinien posiadać poświadczeń w klastrach drugiego zespołu.

Jest to z pewnością najbezpieczniejsze podejście, ale posiadanie dodatkowych klastrów generuje dodatkowe problemy. Każdy z nich musi być aktualizowany i monitorowany. Wiele małych klastrów działa również mniej wydajnie niż większe klastry.

Kontrola dostępu oparta na rolach (RBAC)

Innym sposobem zarządzania dostępem jest kontrolowanie, kto może wykonywać określone operacje w klastrze. Możliwe jest to za pomocą systemu kontroli dostępu opartego na rolach Kubernetes (RBAC).

RBAC ma na celu przyznawanie określonych uprawnień konkretnym użytkownikom (lub kontom usług, które są kontami użytkowników powiązanymi z automatycznymi systemami). Możesz np. przyznać możliwość wyświetlania listy wszystkich Podów w klastrze konkretnemu użytkownikowi, jeśli tego potrzebuje.

Pierwszą i najważniejszą rzeczą, którą należy wiedzieć o RBAC, jest to, że należy go włączyć. RBAC został wprowadzony w Kubernetes 1.6 jako opcja konfiguracyjna klastrów. Jednak to, czy ta opcja jest faktycznie włączona w klastrze, zależy od dostawcy chmury lub instalatora Kubernetes.

Jeśli sam hostujesz klasztar, wypróbuj to polecenie, aby sprawdzić, czy funkcja RBAC jest włączona:

```
kubectl describe pod -n kube-system -l component=kube-apiserver
Name: kube-apiserver-docker-for-desktop
Namespace: kube-system
...
Containers:
  kube-apiserver:
    ...
    Command:
      kube-apiserver
    ...
    --authorization-mode=Node,RBAC
```

Jeśli --authorization-mode nie zawiera RBAC, to RBAC nie jest włączony dla Twojego klastra. Sprawdź dokumentację usługodawcy lub instalatora, aby zobaczyć, jak przebudować klasztar z włączoną funkcją RBAC.

Bez RBAC każdy, kto ma dostęp do klastru, jest w stanie zrobić wszystko, w tym uruchomić dowolny kod lub usunąć zadania. To prawdopodobnie nie jest to, czego chcesz.

Role

Jak działa RBAC? Najważniejszymi pojęciami są użytkownicy, role i powiązania ról.

Za każdym razem, gdy łączysz się z klastrem Kubernetes, robisz to jako określony użytkownik. Dokładny sposób uwierzytelnienia w klastrze zależy od dostawcy; np. w Google Kubernetes Engine używasz narzędzia `gcloud`, aby uzyskać token dostępu do określonego krastra.

W klastrze są skonfigurowani inni użytkownicy; np. dla każdej przestrzeni nazw istnieje domyślne konto usługi. Wszyscy ci użytkownicy mogą potencjalnie mieć różne zestawy uprawnień.

Są one regulowane przez *role* Kubernetes. Rola opisuje określony zestaw uprawnień. Kubernetes zawiera kilka predefiniowanych ról. Przykładowo rola `cluster-admin` przeznaczona dla superużytkowników ma dostęp do odczytu i zmiany dowolnych zasobów w klastrze. Natomiast rola `view` uprawnia do wyświetlenia listy większości obiektów w danej przestrzeni nazw, ale nie może ich modyfikować.

Można zdefiniować role na poziomie przestrzeni nazw (za pomocą obiektu `Role`) lub w całym krastrze (za pomocą obiektu `ClusterRole`). Oto przykład manifestu `ClusterRole`, który zapewnia dostęp do odczytu obiektów `Secret` w dowolnej przestrzeni nazw:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: secret-reader
rules:
- apiGroups: []
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

Wiązanie ról z użytkownikami

Jak skojarzyć użytkownika z rolą? Możesz to zrobić za pomocą *powiązania roli* (ang. *role binding*). Podobnie jak w przypadku ról, można utworzyć obiekt `RoleBinding`, który odnosi się do określonej przestrzeni nazw, lub `ClusterRoleBinding`, który działa na poziomie krastra.

Oto manifest `RoleBinding`, który nadaje użytkownikowi `daisy` rolę `edit` tylko w przestrzeni nazw `demo`:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: daisy-edit
  namespace: demo
subjects:
- kind: User
  name: daisy
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit
  apiGroup: rbac.authorization.k8s.io
```

W Kubernetes uprawnienia są *addywne*; użytkownicy zaczynają bez uprawnień i można je dodawać za pomocą ról i `RoleBindingu`. Nie możesz ograniczyć uprawnień osobie, która już je ma.



Więcej informacji na temat RBAC oraz dostępnych ról i uprawnień można znaleźć w dokumentacji Kubernetes (<https://kubernetes.io/docs/reference/access-authn-authz/rbac/>).

Jakich ról potrzebuję?

Jakie role i powiązania należy skonfigurować w klastrze? Wstępnie zdefiniowane role `cluster-admin`, `edit` oraz `view`, prawdopodobnie spełnią większość wymagań. Aby zobaczyć, jakie uprawnienia ma dana rola, użyj polecenia `kubectl describe`:

```
kubectl describe clusterrole/edit
Name: edit
Labels: kubernetes.io/bootstrapping=rbac-defaults
Annotations: rbac.authorization.kubernetes.io/autoupdate=true
PolicyRule:
  Resources ... Verbs
  -----
  bindings ... [get list watch]
  configmaps ... [create delete deletecollection get list patch update watch]
  endpoints ... [create delete deletecollection get list patch update watch]
  ...
  ...
```

Możesz tworzyć role dla określonych osób lub zadań w organizacji (np. rolę programisty) lub dla poszczególnych zespołów (np. dla kontroli jakości lub bezpieczeństwa).

Ochrona dostępu do Cluster-Admin

Bądź bardzo ostrożny, kiedy przydzielasz rolę `cluster-admin`. Jest to administrator klastrów, równoważny z użytkownikiem `root` w systemach Unix. Może zrobić wszystko. Nigdy nie przydzielaj tej roli użytkownikom, którzy nie są operatorami klastrów, a zwłaszcza nie przydzielaj ich do kont serwisów dla aplikacji, które mogą mieć dostęp do internetu, np. Kubernetes Dashboard (patrz „Pulpit Kubernetes (ang. Kubernetes Dashboard)” w tym rozdziale).



Nie naprawiaj problemów, niepotrzebnie przyznając rolę `cluster-admin`. Kilka złych porad na ten temat znajdziesz na stronach, takich jak Stack Overflow. Przy błędach związanych z uprawnieniami Kubernetes częstą odpowiedzią jest przyznanie aplikacji roli `cluster-admin`. *Nie rób tego*. Tak, powoduje to, że błędy znikają, ale kosztem obejścia wszystkich kontroli bezpieczeństwa i potencjalnego otwarcia klastra dla atakującego. Zamiast tego nadaj aplikacji rolę z najmniejszymi uprawnieniami niezbędnymi do wykonywania swojej pracy.

Aplikacje i wdrażanie

Aplikacje działające w Kubernetes zwykle nie potrzebują żadnych uprawnień RBAC. O ile nie określono inaczej, wszystkie Pody będą działać jako konto usługi `default` w obszarze nazw, z którym nie są związane żadne role.

Jeśli z jakiegoś powodu Twoja aplikacja potrzebuje dostępu do interfejsu API Kubernetes (np. narzędzia do monitorowania, które musi wyświetlić listę Podów), utwórz dedykowane konto usługi dla

aplikacji, użyj RoleBinding, aby powiązać ją z niezbędną rolą (np. `view`), i ogranicz je do określonych przestrzeni nazw.

Co z uprawnieniami wymaganymi do wdrażania aplikacji w klastrze? Najbezpieczniejszym sposobem jest zezwolenie na wdrażanie aplikacji tylko przez narzędzie ciągłego wdrażania (patrz rozdział 14.). Może ono korzystać z dedykowanego konta usługi, z uprawnieniami do tworzenia i usuwania Podów w określonej przestrzeni nazw.

Rola `edit` jest do tego idealna. Użytkownicy z rolą `edit` mogą tworzyć i niszczyć zasoby w przestrzeni nazw, ale nie mogą tworzyć nowych ról ani udzielać uprawnień innym użytkownikom.

Jeśli nie masz zautomatyzowanego narzędzia do wdrażania, a programiści muszą wdrażać aplikacje bezpośrednio w klastrze, będą również potrzebować uprawnień do edycji odpowiednich obszarów nazw. Udzielaj tych zezwoleń po kolei; nie udzielaj nikomu uprawnień do edycji w całym klastrze. Osoby, które nie muszą wdrażać aplikacji, powinny domyślnie korzystać z roli `view`.



Najlepsze praktyki

Upewnij się, że funkcja RBAC jest włączona we wszystkich klastrach. Przyznaj uprawnienia `cluster-admin` tylko tym użytkownikom, którzy faktycznie potrzebują praw do usuwania wszystkiego w klastrze. Jeśli Twoja aplikacja potrzebuje dostępu do zasobów klastra, utwórz dla niej konto usługi i powiąż z rolą posiadającą tylko potrzebne uprawnienia, tylko w tych przestrzeniach nazw, w których ich potrzebuje.

Rozwiązywanie problemów z RBAC

Jeśli używasz starszej aplikacji innej firmy, która nie obsługuje RBAC, lub nadal pracujesz nad wymaganymi uprawnieniami dla własnej aplikacji, możesz napotkać błędy uprawnień RBAC. Jak to wygląda?

Jeśli aplikacja zgłosi żądanie interfejsu API dotyczące czegoś, do czego nie ma uprawnień (np. listy węzłów), serwer API zwróci błąd typu *Forbidden* (błąd HTTP 403):

```
Error from server (Forbidden): nodes.metrics.k8s.io is forbidden:  
User "demo" cannot list nodes.metrics.k8s.io at the cluster scope.
```

Jeśli aplikacja nie rejestruje tych informacji lub nie masz pewności, która aplikacja ulega awarii, możesz sprawdzić logi serwera API (więcej informacji na ten temat można znaleźć w punkcie „Przeglądanie dzienników kontenera” w rozdziale 7.). Poniższe polecenie zwróci komunikaty zawierające ciąg znaków RBAC `DENY`:

```
kubectl logs -n kube-system -l component=kube-apiserver | grep "RBAC DENY"  
RBAC DENY: user "demo" cannot "list" resource "nodes" cluster-wide
```

(Nie będziesz w stanie tego zrobić w klastrze GKE ani w żadnej innej zarządzanej usłudze Kubernetes, która nie daje dostępu do płaszczyzny sterowania: zapoznaj się z dokumentacją dostawcy Kubernetes, aby dowiedzieć się, jak uzyskać dostęp do dzienników serwera API). RBAC posiada opinię skomplikowanego zagadnienia, ale tak naprawdę nie jest. Po prostu przyznaj użytkownikom minimalne uprawnienia, których potrzebują, zapewnić bezpieczeństwo uprawnień `cluster-admin`, a wszystko będzie dobrze.

Skanowanie bezpieczeństwa

Jeśli w klastrze korzystasz z oprogramowania innych firm, warto sprawdzić, czy nie występują problemy związane z bezpieczeństwem. Nawet Twoje własne kontenery mogą zawierać oprogramowanie, o którym nie wiesz. Należy to sprawdzić.

Clair

Clair (<https://github.com/coreos/clair>) to skaner kontenerów typu open source, opracowany w ramach projektu CoreOS. Statycznie analizuje obrazy kontenerów, zanim zostaną faktycznie uruchomione, aby sprawdzić, czy zawierają oprogramowanie lub wersje, które nie są uważane za bezpieczne.

Clair może zostać uruchomiony ręcznie w celu sprawdzenia określonych obrazów pod kątem problemów. Można go także zintegrować z potokiem CD, aby przetestować wszystkie obrazy przed ich wdrożeniem (patrz rozdział 14.).

Ewentualnie Clair możesz podłączyć do rejestru kontenera, aby skanować obrazy, które są do niego przekazywane.

Warto wspomnieć, że nie należy automatycznie ufać obrazom podstawowym, takim jak *alpine*. Clair ma wbudowane opcje bezpieczeństwa dla wielu popularnych obrazów podstawowych i natychmiast poinformuje Cię, jeśli używasz takiego, który ma znaną lukę.

Aqua

Aquas Container Security Platform (<https://www.aquasec.com/products/aqua-container-security-platform/>) to kompleksowa oferta bezpieczeństwa kontenerów komercyjnych, umożliwiająca organizacjom skanowanie kontenerów w poszukiwaniu luk, złośliwego oprogramowania i podejrzanych działań, zgodnie z zapewnieniem egzekwowania zasad i zgodności z przepisami.

Jak można się spodziewać, platforma Aqua integruje się z rejestrem kontenera, potokiem CI/CD oraz wieloma systemami orkiestracji, w tym Kubernetes.

Aqua oferuje również bezpłatne narzędzie o nazwie MicroScanner (<https://github.com/aquasecurity/microscanner>), które można dodawać do obrazów kontenerów w celu skanowania zainstalowanych pakietów pod kątem znanych luk. Korzysta z tej samej bazy danych, co platforma Aqua Security Platform.

MicroScanner można zainstalować, dodając go do pliku Dockerfile:

```
ADD https://get.aquasec.com/microscanner /  
RUN chmod +x /microscanner  
RUN /microscanner <TOKEN> [--continue-on-failure]
```

MicroScanner wyświetla w formacie JSON listę wykrytych luk, którą można wykorzystać i zgłosić przy użyciu innych narzędzi.

Innym przydatnym narzędziem open source platformy Aqua jest kube-hunter (<https://kubehunter.aquasec.com/>), zaprojektowany do szukania problemów bezpieczeństwa w samym klastrze Kubernetes. Jeśli uruchomisz go jako kontener na komputerze poza klastrem (tak jak może to zrobić

osoba atakująca), sprawdzi różne rodzaje problemów, takie jak ujawnione adresy e-mail w certyfikatach, niezabezpieczone pulpity, otwarte porty i punkty końcowe itd.

Anchore Engine

Anchore Engine (<https://github.com/anchore/anchore-engine>) to narzędzie typu open source do skanowania obrazów kontenerów, nie tylko pod kątem znanych słabych punktów, ale także do identyfikacji *wszystkich składników*, które zawiera kontener — w tym bibliotek, plików konfiguracyjnych i uprawnień do plików. Można to wykorzystać do weryfikacji kontenerów pod kątem zasad zdefiniowanych przez użytkownika; można np. zablokować wszelkie obrazy zawierające poświadczenia bezpieczeństwa lub kod źródłowy aplikacji.



Najlepsze praktyki

Nie uruchamiaj kontenerów z niezaufanych źródeł lub gdy nie masz pewności, co w nich jest. Na wszystkich kontenerach (nawet tych, które sam zbudujesz) uruchom narzędzie skanujące, takie jak Clair lub MicroScanner — aby mieć pewność, że nie ma znanych luk w podstawowych obrazach lub zależnościach.

Kopie zapasowe

Być może zastanawiasz się, czy nadal potrzebujesz kopii zapasowych w natywnych architekturach chmurowych. W końcu Kubernetes jest z natury niezawodny i może poradzić sobie z utratą kilku węzłów jednocześnie bez utraty stanu, a nawet nadmiernego obniżenia wydajności aplikacji.

Ponadto Kubernetes jest deklaratywną infrastrukturą w postaci kodu. Wszystkie zasoby Kubernetes są opisane danymi przechowywanymi w niezawodnej bazie danych (*etcd*). Jeśli przypadkowo usunięte zostaną niektóre Pody, Deployment ponownie utworzy je ze specyfikacji przechowywanej w bazie danych.

Czy muszę wykonać kopię zapasową?

Czy nadal potrzebujesz kopii zapasowych? Tak. Przykładowo dane przechowywane na woluminach trwałych są podatne na awarie (patrz „Woluminy trwałe” w rozdziale 8.). Twój dostawca usług w chmurze może zapewniać wysoko niezawodne woluminy (np. replikację danych w dwóch różnych strefach dostępności), ale to nie to samo, co tworzenie kopii zapasowych.

Powtórzmy ten akapit, ponieważ nie jest oczywisty.



Replikacja nie jest kopią zapasową. Chociaż replikacja może uchronić Cię przed awarią podstawowego woluminu dyskowego, nie zabezpieczy przed przypadkowym usunięciem woluminu, np. po nieprawidłowym wyborze opcji w konsoli webowej.

Replikacja nie zapobiegnie również nadpisywaniu danych przez źle skonfigurowaną aplikację, a także uruchamianiu przez operatora polecenia z nieprawidłowymi zmiennymi środowiskowymi i przypadkowemu usunięciu produkcyjnej bazy danych — zamiast programistycznej. (To się dzieje

(<https://thenewstack.io/junior-dev-deleted-production-database>) prawdopodobnie częściej, niż ktoś kiedykolwiek jest skłonny to przyznać).

Tworzenie kopii zapasowej etcd

Jak pisaliśmy w podpunkcie „Wysoka niezawodność” w rozdziale 3., Kubernetes przechowuje cały swój stan w bazie danych *etcd*, więc każda awaria lub utrata danych może być katastrofalna. To jeden bardzo dobry powód, dla którego zalecamy korzystanie z usług zarządzanych, które gwarantują dostępność *etcd* oraz płaszczyzny sterowania (patrz „Skorzystaj z zarządzanych usług Kubernetes, jeśli możesz” w rozdziale 3.).

Jeśli posiadasz własne węzły master, jesteś odpowiedzialny za zarządzanie *etcd*, replikację i tworzenie kopii zapasowych. Nawet w przypadku zwykłych migawek danych (ang. *snapshot*) pobieranie i weryfikacja migawki, odbudowywanie klastra i przywracanie danych zajmują pewien czas. W tym czasie klaster prawdopodobnie będzie niedostępny lub poważnie zdegradowany.



Najlepsze praktyki

Skorzystaj z dostawcy usług „pod klucz” lub usług zarządzanych, aby uruchomić węzły master z kopią zapasową oraz klastrowaniem bazy *etcd*. Jeśli je uruchomisz, upewnij się, że wiesz, co robisz. Elastyczne zarządzanie *etcd* jest pracą specjalistyczną, a konsekwencje popełnienia błędu mogą być poważne.

Kopia zapasowa stanu zasobów

Oprócz awarii *etcd* istnieje także kwestia zapisu stanu Twoich indywidualnych zasobów. Jeśli np. usuniesz niewłaściwy Deployment, w jaki sposób go odtworzysz?

W tej książce kładziemy nacisk na wartość *infrastruktury* jako paradygmatu kodu i zalecamy, aby zawsze zarządzać zasobami Kubernetes deklaratywnie, stosując manifesty YAML lub wykresy Helm, przechowywane w systemie kontroli wersji.

Teoretycznie więc, aby odtworzyć całkowity stan obciążen klastra, powinieneś mieć możliwość sprawdzenia odpowiednich repozytoriów kontroli wersji i zastosowania w nich wszystkich zasobów. Teoretycznie.

Tworzenie kopii zapasowej stanu klastra

W praktyce nie wszystko, co masz w repozytorium, działa teraz w klastrze. Niektóre aplikacje mogą zostać wyłączone z użycia lub zastąpione nowszymi wersjami. Niektóre mogą nie być gotowe do wdrożenia.

W tej książce zalecamy unikanie wprowadzania bezpośrednich zmian w zasobach i stosowanie zmian w aktualizowanych plikach manifestów (patrz „Kiedy nie należy używać poleceń imperatywnych” w rozdziale 7.). Jednak ludzie nie zawsze stosują się do dobrych rad (lament konsultanta na przestrzeni wieków).

W każdym razie jest prawdopodobne, że podczas początkowego wdrażania i testowania aplikacji inżynierowie mogą dostosowywać ustawienia, takie jak liczba replik i powinowactwa węzłów w locie, i przechowują je w repozytorium dopiero po osiągnięciu właściwych wartości.

Założymy, że klaster został całkowicie zamknięty lub wszystkie jego zasoby zostały usunięte (mamy nadzieję, że jest to mało prawdopodobny scenariusz, ale przydatny jako eksperyment myślowy). Jak byś go ponownie utworzył?

Jeśli nawet masz znakomicie zaprojektowany i aktualny system automatyzacji klastra, który może przenieść wszystko do nowego klastra, skąd *wiesz*, że stan tego klastra pasuje do utraconego?

Jednym ze sposobów, aby to zapewnić, jest utworzenie migawki działającego klastra, do której można się później odwołać w przypadku problemów.

Duże i małe katastrofy

Jest mało prawdopodobne, że stracisz cały klaster: tysiące pracowników Kubernetes ciężko pracowały, aby mieć pewność, że tak się nie stanie.

Bardziej prawdopodobne jest, że Ty (lub najnowszy członek Twojego zespołu) przypadkowo usuniesz przestrzeń nazw, wyłączysz Deployment lub określisz zły zestaw etykiet dla polecenia kubectl delete, usuwając więcej, niż zamierzałeś.

Bez względu na przyczynę zdarzają się katastrofy. Przyjrzyjmy się więc narzędziu do tworzenia kopii zapasowych, które pomoże Ci ich uniknąć.

Velero

Velero (wcześniej znane jako Ark) to bezpłatne narzędzie typu open source, które może tworzyć kopie zapasowe i przywracać stan klastra oraz trwałe dane.

Velero działa w klastrze i łączy się z wybraną usługą przechowywania w chmurze (np. Amazon S3 lub Azure Storage).

Postępuj zgodnie z instrukcjami (<https://velero.io/>), aby skonfigurować Velero dla swojej platformy.

Konfigurowanie Velero

Przed użyciem Velero musisz utworzyć obiekt BackupStorageLocation w klastrze Kubernetes, informując go, gdzie przechowywać kopie zapasowe (np. bucket w chmurze AWS S3). Oto przykład, który konfiguruje Velero do tworzenia kopii zapasowej w bucket demo-backup:

```
apiVersion: velero.io/v1
kind: BackupStorageLocation
metadata:
  name: default
  namespace: velero
spec:
  provider: aws
  objectStorage:
    bucket: demo-backup
  config:
    region: us-east-1
```

Miejsce do przechowywania danych nosi nazwę default, ale możesz dodawać inne o dowolnych nazwach.

Velero może również wykonać kopię zapasową zawartości woluminów trwałych. Aby poinformować, gdzie je przechowywać, musisz utworzyć obiekt VolumeSnapshotLocation:

```
apiVersion: velero.io/v1
kind: VolumeSnapshotLocation
metadata:
  name: aws-default
  namespace: velero
spec:
  provider: aws
  config:
    region: us-east-1
```

Tworzenie kopii zapasowej Velero

Podczas tworzenia kopii zapasowej za pomocą polecenia `velero backup` serwer Velero wysyła zapytanie do interfejsu API Kubernetes w celu pobrania zasobów pasujących do podanego selektora (domyślnie tworzy kopię zapasową wszystkich zasobów). Możesz wykonać kopię zapasową danych przestrzeni nazw lub całego klastra:

```
velero backup create demo-backup --include-namespaces demo
```

Następnie wyeksportuje wszystkie te zasoby do pliku w Twoim chmurowym pojemniku bucket, zgodnie ze skonfigurowanym BackupStorageLocation. Kopie zapasowe metadanych i zawartości woluminów trwałych zostaną również zapisane w skonfigurowanej lokalizacji VolumeSnapshot.

Alternatywnie możesz wykonać kopię zapasową wszystkiego w klastrze, oprócz określonych przestrzeni nazw (np. system-kube). Możesz także zaplanować automatyczne tworzenie kopii zapasowych; przykładowo Velero może wykonywać kopię zapasową klastra co noc, a nawet co godzinę.

Każda kopia zapasowa Velero jest sama w sobie kompletna, a nie przyrostowa. Aby przywrócić kopię zapasową, potrzebujesz tylko najnowszego pliku kopii zapasowej.

Przywracanie danych

Za pomocą polecenia `velero backup get` możesz wyświetlić listę dostępnych kopii zapasowych:

```
velero backup get
NAME      STATUS     CREATED          EXPIRES   SELECTOR
demo-backup Completed 2018-07-14 10:54:20 +0100 BST 29d      <none>
```

Aby zobaczyć, co zawiera konkretna kopia zapasowa, użyj polecenia `velero backup download`:

```
velero backup download demo-backup
Backup demo-backup has been successfully downloaded to
$PWD/demo-backup-data.tar.gz
```

Pobrany plik to archiwum `tar.gz`, które można rozpakować i sprawdzić przy użyciu standardowych narzędzi. Jeśli np. potrzebujesz manifestu tylko dla określonego zasobu, możesz go wyodrębnić z pliku kopii zapasowej i zastosować indywidualnie za pomocą `kubectl apply -f`.

Aby przywrócić całą kopię zapasową, polecenie `velero restore` rozpocznie proces, a Velero odtworzy wszystkie zasoby i woluminy opisane w określonej migawce, pomijając wszystko, co już istnieje.

Jeśli zasób istnieje, ale różni się od zasobu zapasowego, Velero ostrzeże Cię, ale nie zastąpi istniejącego zasobu. Jeśli np. chcesz zresetować stan działającego obiektu Deployment do stanu z najnowszej migawki, najpierw usuń działający Deployment, a następnie przywrócić go za pomocą Velero.

Alternatywnie, jeśli przywracasz kopię zapasową przestrzeni nazw, możesz najpierw usunąć przestrzeń nazw, a następnie przywrócić kopię zapasową.

Procedury przywracania i testy

Powinieneś napisać szczegółową procedurę opisującą sposób przywracania danych z kopii zapasowych i upewnić się, że wszyscy pracownicy wiedzą, gdzie znaleźć ten dokument. Kiedy zdarza się katastrofa, zwykle dzieje się to w niewygodnej chwili, kluczowe osoby są niedostępne i wszyscy wpadają w panikę. Twoja procedura powinna być tak jasna i precyzyjna, że będzie mógł ją prowadzić ktoś, kto nie zna Velero lub nawet Kubernetes.

W każdym miesiącu testuj przywracanie danych, w którym inny członek zespołu wykonuje procedurę przywracania dla klastra tymczasowego. Sprawdzisz, czy kopie zapasowe są dobre, czy procedura przywracania jest poprawna i będziesz mieć pewność, że wszyscy wiedzą, jak to zrobić.

Planowanie wykonywania kopii zapasowych Velero

Wszystkie kopie zapasowe powinny być zautomatyzowane, a Velero nie jest wyjątkiem. Możesz zaplanować regularną kopię zapasową za pomocą polecenia `velero schedule create`:

```
velero schedule create demo-schedule --schedule="0 1 * * *" --include-namespaces demo
Schedule "demo-schedule" created successfully.
```

Argument `schedule` określa, kiedy należy wykonać kopię zapasową. Ma on format uniksowego narzędzia cron (patrz „CronJob” w rozdziale 9.). W tym przykładzie `0 1 * * *` uruchamia tworzenie kopii zapasowej codziennie o 01:00.

Aby zobaczyć zaplanowane kopie zapasowe, użyj polecenia `velero schedule get`:

```
velero schedule get
NAME      STATUS    CREATED      SCHEDULE      BACKUP    TTL LAST BACKUP SELECTOR
demo-schedule  Enabled  2018-07-14  * 10 * * *  720h0m0s 10h ago <none>
```

Pole `BACKUP TTL` pokazuje, jak długo kopia zapasowa będzie przechowywana, zanim zostanie automatycznie usunięta (domyślnie 720 godzin, co odpowiada jednemu miesiącowi).

Inne zastosowania Velero

Chociaż Velero jest niezwykle przydatne do odzyskiwania danych po awarii, możesz go również użyć do migracji zasobów i danych z jednego klastra do drugiego — proces ten nazywany jest czasem *lift and shift*.

Regularne tworzenie kopii zapasowych Velero może również pomóc zrozumieć, jak zmienia się Twoje użycie Kubernetes w czasie; możesz porównać obecny stan ze stanem sprzed miesiąca, sześciu miesięcy i sprzed roku.

Migawki mogą być również użytecznym źródłem informacji kontrolnych; możesz np. dowiedzieć się, co działało w klastrze w danym dniu lub czasie oraz jak i kiedy zmienił się stan klastra.



Najlepsze praktyki

Używaj Velero do regularnego tworzenia kopii zapasowych stanu klastra i trwałych danych; przynajmniej co noc. Uruchamiaj test przywracania co najmniej raz w miesiącu.

Monitorowanie statusu klastra

Monitorowanie rodzimych aplikacji chmurowych to duży temat, który — jak zobaczymy w rozdziale 15. — obejmuje m.in. obserwowalność, pomiary, rejestrowanie, śledzenie i tradycyjne monitorowanie czarnej skrzynki (ang. *black-box*).

Jednak w tym rozdziale zajmiemy się tylko monitorowaniem samego klastra Kubernetes: kondycją klastra, statusem poszczególnych węzłów oraz wykorzystaniem klastra i stanem obciążen.

kubectl

W rozdziale 2. wprowadziliśmy polecenie `kubectl`. Jednak jeszcze nie wyczerpaliśmy jego możliwości. Oprócz ogólnego narzędzia administrowania zasobami Kubernetes, `kubectl` może także raportować przydatne informacje o stanie komponentów klastra.

Status płaszczyzny sterowania

Komenda `kubectl get componentstatuses` (lub w skrócie `kubectl get cs`) podaje informacje o stanie zdrowia komponentów płaszczyzny sterowania — harmonogramu, menadżera kontrolera i `etcd`:

```
kubectl get componentstatuses
NAME           STATUS  MESSAGE           ERROR
controller-manager  Healthy   ok
scheduler        Healthy   ok
etcd-0          Healthy   {"health": "true"}
```

Jeśli pojawiłby się poważny problem z dowolnym komponentem płaszczyzny sterowania, wkrótce i tak stałby się widoczny. Jednak przydatna jest możliwość sprawdzenia takiego stanu wcześniej.

Jeśli którykolwiek ze składników płaszczyzny sterowania nie znajduje się w stanie `Healthy`, należy go naprawić. Taki przypadek nigdy nie powinien nastąpić w zarządzanej usłudze Kubernetes, ale kiedy hostujesz klastry samodzielnie, musisz się tym zająć.

Status węzła

Innym przydatnym poleceniem jest `kubectl get nodes`, które wyświetli listę wszystkich węzłów w klastrze i poda ich status oraz wersję Kubernetes:

```
kubectl get nodes
NAME           STATUS  ROLES   AGE  VERSION
docker-for-desktop  Ready   master  5d   v1.10.0
```

Ponieważ klastry Docker Desktop mają tylko jeden węzeł, dane wyjściowe nie są szczególnie pouczające; spójrzmy na wyniki z małego klastra Google Kubernetes Engine, aby uzyskać coś bardziej realistycznego:

```
kubectl get nodes
NAME                               STATUS  ROLES   AGE    VERSION
gke-k8s-cluster-1-n1-standard-2-pool--816n Ready   <none>  9d    v1.10.2-gke.1
gke-k8s-cluster-1-n1-standard-2-pool--dwtv Ready   <none>  19d   v1.10.2-gke.1
gke-k8s-cluster-1-n1-standard-2-pool--67ch Ready   <none>  20d   v1.10.2-gke.1
...
```

Zauważ, że w Docker Desktop w wyniku polecenia `get nodes` rola węzła została pokazana jako `master`. Oczywiście, ponieważ istnieje tylko jeden węzeł, musi mieć status `master` oraz jednocześnie `worker`.

W Google Kubernetes Engine i niektórych innych zarządzanych usługach Kubernetes nie masz bezpośredniego dostępu do węzłów `master`. W związku z tym polecenie `kubectl get nodes` wyświetla tylko węzły `worker` (rola `<none>` wskazuje węzeł `worker`).

Jeśli którykolwiek z węzłów ma status `NotReady`, oznacza to, że występuje problem. Ponowne uruchomienie węzła może go naprawić, ale jeśli tak się nie stanie, może wymagać dalszego debugowania. Możesz też po prostu go usunąć i zamiast tego utworzyć nowy węzeł.

Aby uzyskać szczegółowe informacje na temat problemów związanych z węzłami, możesz użyć polecenia `kubectl describe node`:

```
kubectl describe nodes/gke-k8s-cluster-1-n1-standard-2-pool--816n
```

W wyniku dostaniemy pamięć i pojemność procesora węzła oraz zasoby aktualnie używane przez Pody.

Obciążenia

W punkcie „Odpytywanie klastra za pomocą polecenia `kubectl`” w rozdziale 4. pisaliśmy, że możesz także użyć `kubectl`, aby wyświetlić listę wszystkich Podów (lub dowolnych zasobów) w klastrze. W tym przykładzie wylistowano tylko Pody z domyślnej przestrzeni nazw. Flaga `--all-namespaces` pozwoli zobaczyć wszystkie Pody w całym klastrze:

```
kubectl get pods --all-namespaces
NAMESPACE      NAME           READY   STATUS        RESTARTS   AGE
cert-manager   cert-manager   1/1     Running       1          10d
pa-test        permissions-auditor-15281892 0/1     CrashLoopBackOff 1720      6d
freshtacks     freshtracks-agent-779758f445 3/3     Running       5          20d
...
```

Dzięki temu można uzyskać wygodny przegląd tego, co działa w klastrze, oraz podgląd wszelkich problemów związanych z Podem. Jeśli istnieją jakiekolwiek Pody, które nie mają statusu `Running`, tak jak Pod `permissions-auditor` w przykładzie, należy to zbadać.

Kolumna `READY` pokazuje, ile kontenerów w Podzie faktycznie działa, w porównaniu do skonfigurowanej liczby. Przykładowo Pod `freshtacks-agent` pokazuje 3/3, a to oznacza, że 3 z 3 kontenerów działają — wszystko jest w porządku.

Z drugiej strony, `permissions-auditor` pokazuje, że 0/1 kontenerów jest gotowych: 0 kontenerów jest uruchomionych, a wymagany jest 1. Przyczyna jest pokazana w kolumnie `STATUS`: `CrashLoopBackOff`. Kontener nie uruchamia się poprawnie.

Kiedy kontener ulega awarii, Kubernetes będzie próbował go ponownie uruchamiać w coraz większych odstępach czasu, zaczynając od 10 sekund i podwajając za każdym razem, do 5 minut. Strategia ta nazywa się *wykładniczym wycofywaniem* (ang. *exponential backoff*), stąd komunikat o stanie CrashLoopBackOff.

Wykorzystanie procesora i pamięci

Kolejny przydatny widok klastra zapewnia komenda `kubectl top`. W przypadku węzłów pokaże CPU oraz ilość pamięci każdego węzła oraz ilość aktualnie używanego:

```
kubectl top nodes
NAME          CPU(cores)   CPU%   MEMORY(bytes)  MEMORY%
gke-k8s-cluster-1-n1...8l6n  151m      7%    2783Mi        49%
gke-k8s-cluster-1-n1...dwtv  155m      8%    3449Mi        61%
gke-k8s-cluster-1-n1...67ch  580m     30%   3172Mi        56%
...
```

Poniższe polecenie pokaże wykorzystanie procesora i pamięci dla określonego Poda:

```
kubectl top pods -n kube-system
NAME          CPU(cores)   MEMORY(bytes)
event-exporter-v0.1.9-85bb4fd64d-2zjng  0m       27Mi
fluentd-gcp-scaler-7c5db745fc-h7ntr    10m      27Mi
fluentd-gcp-v3.0.0-5m627                11m      171Mi
...
```

Konsola dostawcy chmury

Jeśli korzystasz z zarządzanej usługi Kubernetes oferowanej przez dostawcę usług chmurowych, będziesz mieć dostęp do konsoli przeglądarkowej. Może ona wyświetlać przydatne informacje o klastrze, jego węzłach i obciążeniach.

Przykładowo konsola Google Kubernetes Engine (GKE) wyświetla wszystkie klastry, szczegółowe informacje o każdym klastrze, pule węzłów itd. (patrz rysunek 11.1).

Można także wyświetlić listę obciążień, usług i szczegóły konfiguracji klastra. Są to te same informacje, które można uzyskać za pomocą narzędzia `kubectl`. Ponadto konsola GKE umożliwia także wykonywanie zadań administracyjnych, czyli tworzenie klastrów, aktualizowanie węzłów i wszystko, czego potrzebujesz do codziennego zarządzania klastrem.

Usługi Azure Kubernetes Service, AWS Elastic Container Service for Kubernetes itp. mają podobne funkcje. Warto zapoznać się z konsolą zarządzania dla konkretnej usługi Kubernetes, ponieważ będziesz jej często używać.

Pulpit Kubernetes (ang. Kubernetes Dashboard)

Pulpit Kubernetes (<https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>) to przeglądarkowy interfejs użytkownika dla klastrów Kubernetes (patrz rysunek 11.2). Jeśli prowadzisz własny klaszt Kubernetes, a nie korzystasz z usługi zarządzanej, możesz uruchomić pulpit Kubernetes, aby uzyskać mniej więcej te same informacje, jakich dostarczy konsola usług zarządzanych.

The screenshot shows the 'Kubernetes clusters' management interface. At the top, there's a back arrow, the title 'Kubernetes clusters', and three buttons: 'EDIT', 'DELETE', and 'CONNECT'. Below the title, a cluster named 'k8s-cluster-1' is selected, indicated by a checked checkbox. There are three tabs: 'Details' (selected), 'Storage', and 'Nodes'. The 'Cluster' section contains the following information:

| | | |
|---------------------------|--|-------------------|
| Master version | 1.10.2-gke.1 | Upgrade available |
| Endpoint | 35.185.228.127 | Show credentials |
| Client certificate | Enabled | |
| Kubernetes alpha features | Disabled | |
| Current total size | 7 | |
| Master zone | us-west1-a | |
| Node zones | us-west1-a us-west1-b us-west1-c | |
| Network | network | |
| Subnet | subnet-3 | |

Rysunek 11.1. Konsola Google Kubernetes Engine

The screenshot shows the Kubernetes dashboard interface. On the left, a sidebar navigation menu includes: Cluster (Namespaces, Nodes, Persistent Volumes, Roles, Storage Classes), Namespace (All namespaces, Overview), Workloads (Cron Jobs, Daemon Sets, Deployments, Jobs, Pods, Replica Sets, Replication Controllers, Stateful Sets), Discovery and Load Balancing (Ingresses, Services), Config and Storage (Config Maps, Persistent Volume Claims, Secrets), Settings, and About.

The main area displays several dashboards:

- CPU usage:** A line chart showing CPU usage over time from 11:49 to 12:13. The usage fluctuates between 0.05 and 0.08 cores.
- Memory usage:** A line chart showing memory usage over time from 11:49 to 12:13. The usage fluctuates between 114 M and 343 M.
- Workloads Statuses:** A row of seven donut charts showing the status of different workloads:
 - Cron Jobs: 100.0%
 - Daemon Sets: 100.0%
 - Deployments: 100.0%
 - Jobs: 100.0%
 - Pods: 93.3%
 - Replica Sets: 100.0%
 - Replication Controllers: 100.0%
- Cron Jobs:** A table listing two cron jobs:

| Name | Namespace | Labels | Schedule | Suspend | Active | Last Schedule | Age |
|---------|-----------|--------|-----------|---------|--------|---------------|--------|
| hello | default | - | * * * * * | false | 77 | 7 days | 7 days |
| missign | default | - | 0 0 * * * | false | 0 | - | 7 days |
- Daemon Sets:** A table listing one daemon set:

| Name | Namespace | Labels | Pods | Age | Images |
|--------------|-----------|--------------------|-------|---------|--------|
| nginx-daemon | default | name: nginx-daemon | 1 / 1 | a month | nginx |
- Deployments:** A table listing one deployment:

| Name | Namespace | Labels | Pods | Age | Updated |
|-------|-----------|--------|-------|---------|------------|
| nginx | default | - | 1 / 1 | 1 month | 2018-05-10 |

Rysunek 11.2. Pulpit nawigacyjny Kubernetes wyświetla przydatne informacje o klastrze

Jak można się spodziewać, pulpit pozwala zobaczyć status klastrów, węzłów i obciążień, w podobny sposób jak narzędzie kubectl, ale za pomocą interfejsu graficznego. Korzystając z pulpitu, możesz także tworzyć i usuwać zasoby.

Ponieważ pulpit udostępnia wiele informacji o klastrze i obciążeniach, bardzo ważne jest, aby odpo-wiednio go zabezpieczyć i nigdy nie udostępniać publicznie w internecie. Pulpit umożliwia prze-głądanie zawartości zasobów ConfigMap oraz Secret, które mogą zawierać dane uwierzytelniające i klucze kryptograficzne. Musisz więc ściśle kontrolować dostęp do pulpitu.

W 2018 r. firma RedLock (zajmująca się bezpieczeństwem) znalazła setki niezabezpieczonych konsoli Kubernetes Dashboard (<https://redlock.io/blog/cryptojacking-tesla>) dostępnych w internecie, w tym jednej należącej do Tesla, Inc. Dzięki temu mogli pozyskać dane uwierzytelniające do usług chmurowych i wykorzystać je w celu uzyskania dostępu do dalszych poufnych informacji.



Najlepsze praktyki

Jeśli nie musisz uruchamiać pulpitu Kubernetes (np. masz już konsolę Kubernetes dostarczaną przez usługę zarządzaną, taką jak GKE), nie uruchamiaj go. Jeśli go uru-chomisz, upewnij się, że ma minimalne uprawnienia (<https://blog.heptio.com/on-securig-the-kubernetes-dashboard-16b09b1b7aca>) i nie jest udostępniony w interne-cie. Zamiast tego uzyskaj do niego dostęp za pośrednictwem kubectl proxy.

Weave Scope

Weave Scope (<https://github.com/weaveworks/scope>) to świetne narzędzie do wizualizacji i moni-torowania klastra, pokazujące w czasie rzeczywistym mapę Twoich węzłów, kontenerów i procesów. Możesz także zobaczyć metryki i metadane, a nawet uruchomić lub zatrzymać kontenery.

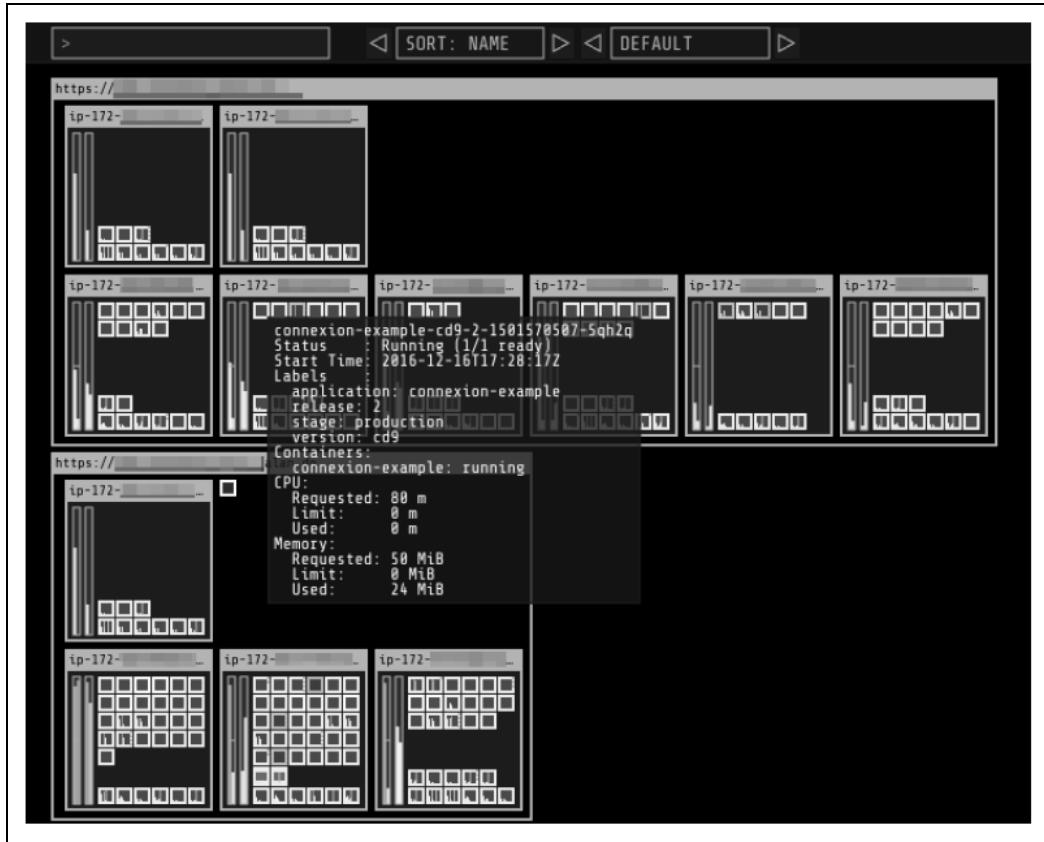
kube-ops-view

W przeciwnieństwie do pulpitu Kubernetes, kube-ops-view (<https://github.com/hjacobs/kube-ops-view>) nie jest narzędziem ogólnego zastosowania do zarządzania klastrami. Zamiast tego daje wizualizację tego, co dzieje się w Twoim klastrze: jakie są węzły, wykorzystanie procesora i pamięci w każdym węźle, ile na każdym z nich działa Podów oraz jaki jest ich status (patrz rysunek 11.3).

node-problem-detector

Narzędzie node-problem-detector (<https://github.com/kubernetes/node-problem-detector>) to dodatek Kubernetes, który może wykrywać i raportować kilka rodzajów problemów na poziomie węzłów; mogą to być problemy ze sprzętem, takie jak błędy procesora lub pamięci, uszkodzenie systemu plików oraz błędy związane z środowiskiem wykonawczym kontenera.

Obecnie node-problem-detector zgłasza problemy, wysyłając zdarzenia do interfejsu API Kuber-netes. Ponadto jest wyposażony w bibliotekę kliencką Go, której można użyć do integracji z wła-snymi narzędziami.



Rysunek 11.3. Narzędzie `kube-ops-view` wyświetla obraz operacyjny klastra Kubernetes

Chociaż Kubernetes obecnie nie podejmuje żadnych działań w odpowiedzi na zdarzenia z node-problem-detector, w przyszłości może nastąpić dalsza integracja, która pozwoli schedulerowi np. na uniknięcie uruchamiania Podów w problematycznych węzłach.

Jest to świetny sposób na uzyskanie ogólnego przeglądu klastra i jego działań. Chociaż nie zastępuje pulpitu ani specjalistycznych narzędzi do monitorowania, jest dobrym uzupełnieniem.

Dalsza lektura

Bezpieczeństwo Kubernetes jest złożonym i wyspecjalizowanym tematem, a my tylko zarysowaliśmy to zagadnienie. Zasługuje ono na własną książkę..., a obecnie jest tylko jedna. Ekspertki ds. bezpieczeństwa Liz Rice i Michael Hausenblas napisali doskonałą książkę *Kubernetes Security* (O'Reilly), która obejmuje bezpieczne konfigurowanie klastra, bezpieczeństwo kontenerów, zarządzanie obiektami Seckret i wiele innych zagadnień. Polecamy.

Podsumowanie

Bezpieczeństwo nie jest produktem ani celem końcowym, ale ciągłym procesem wymagającym wiedzy, przemyślenia i uwagi. Jeśli przeczytałeś ten rozdział i zrozumiałeś informacje w nim zawarte, wiesz wszystko, co musisz wiedzieć, aby bezpiecznie skonfigurować swoje kontenery w Kubernetes. Jesteśmy pewni, że rozumiesz, iż powinien to być początek, a nie koniec Twojego procesu zapewnienia bezpieczeństwa.

Oto najważniejsze rzeczy, o których należy pamiętać.

- Kontrola dostępu oparta na rolach (RBAC) zapewnia dokładne zarządzanie uprawnieniami w Kubernetes. Upewnij się, że jest włączona i użyj ról RBAC, aby przyznać określonym użytkownikom i aplikacjom tylko minimalne uprawnienia, których potrzebują do wykonywania swoich zadań.
- Kontenery nie są wolne od problemów związanych z bezpieczeństwem i złośliwym oprogramowaniem. Za pomocą skanera sprawdź wszystkie kontenery, które uruchomisz w produkcji.
- Kubernetes jest świetny, ale nadal potrzebujesz kopii zapasowych. Użyj narzędzi Velero, aby wykonać kopię zapasową danych i stanu klastra. Jest także przydatne do przenoszenia zasobów między klastrami.
- `kubectl` to potężne narzędzie do sprawdzania i raportowania wszystkich aspektów klastra i obciążień. Zaprzyjaźnij się z `kubectl`. Spędzisz z nim dużo czasu.
- Skorzystaj z konsoli przeglądarkowej dostawcy Kubernetes i `kube-ops-view`, aby uzyskać graficzny przegląd tego, co się dzieje. Jeśli korzystasz z pulpitu Kubernetes, zabezpiecz go tak dokładnie, jak swoje dane uwierzytelniające w chmurze i klucze kryptograficzne.

Wdrażanie aplikacji Kubernetes

Leżałem na plecach, zaskoczony uczuciem spokoju i skupienia, przywiązanym do czterech i pół miliona funtów materiałów wybuchowych.

— Ron Garan, astronauta

W tym rozdziale odpowiemy na pytanie, jak zamienić pliki manifestu w działające aplikacje. Dowiesz się, jak budować wykresy Helm dla aplikacji i jak posługiwać się alternatywnymi narzędziami do zarządzania manifestami, takimi jak tanka, kustomize, kapitan i kompose.

Budowanie manifestów za pomocą wykresu Helm

W rozdziale 2. napisaliśmy, jak wdrażać aplikacje przy użyciu zasobów Kubernetes utworzonych z manifestów YAML i nimi zarządzać. Nic nie stoi na przeszkodzie, aby w ten sposób zarządzać wszystkimi aplikacjami Kubernetes przy użyciu tylko czystych plików YAML. Jednak nie jest to idealne rozwiązanie. Utrzymanie tych plików jest nie tylko trudne, ale istnieje również problem z dystrybucją.

Załóżmy, że chcesz udostępnić swoją aplikację innym osobom, aby działała w ich własnych klastrach. Możesz dystrybuować do nich pliki manifestu. Jednak będą musiały dostosować niektóre ustawienia dla własnego środowiska.

Aby to zrobić, Twoi odbiorcy będą musieli utworzyć własną kopię konfiguracji Kubernetes, znaleźć, gdzie zdefiniowane są różne ustawienia (być może zduplikowane w kilku miejscach) i edytować je.

Z czasem będą musieli utrzymywać własne kopie plików, a kiedy udostępnisz aktualizację, będą musieli ją pobrać i ręcznie dopasować do lokalnych zmian.

To w końcu zaczyna być trudne. Chcielibyście mieć możliwość oddzielenia nieprzetworzonych plików manifestu od określonych ustawień i zmiennych, które Ty lub dowolny użytkownik aplikacji możecie dostosować. Idealnie byłoby, gdyby były dostępne w standardowym formacie, w którym każdy może je pobrać i zainstalować w klastrze Kubernetes.

Gdy już to osiągniemy, każda aplikacja może udostępnić nie tylko wartości konfiguracyjne, ale także wszelkie zależności od innych aplikacji lub usług. Następnie intelligentne narzędzie do zarządzania pakietami może zainstalować i uruchomić aplikację wraz ze wszystkimi jej zależnościami za pomocą pojedynczego polecenia.

W podrozdziale „Helm: menadżer pakietów Kubernetes” w rozdziale 4. przedstawiliśmy narzędzie Helm i pokazaliśmy, jak wykorzystać je do instalowania publicznych wykresów. Teraz przeanalizujemy je nieco bardziej szczegółowo i napiszemy, jak opracować własne.

Co znajduje się w wykresie narzędzia Helm?

W repozytorium demo otwórz katalog *hello-helm3/k8s*, aby zobaczyć, co znajduje się w wykresie Helm.

Każdy wykres Helm ma standardową strukturę. Wykres jest zawarty w katalogu o tej samej nazwie, co wykres (w tym przypadku *demo*):

```
demo
  Chart.yaml
  production-values.yaml
  staging-values.yaml
  templates
    deployment.yaml
    service.yaml
  values.yaml
```

Plik *Chart.yaml*

Pod spodem znajduje się plik o nazwie *Chart.yaml*, który określa nazwę i wersję wykresu:

```
name: demo
sources:
  - https://github.com/cloudnative-devops/demo
version: 1.0.1
```

W pliku *Chart.yaml* możesz zdefiniować wiele opcjonalnych pól, w tym link do kodu źródłowego projektu, tak jak wyżej. Jedynymi wymaganymi informacjami są nazwa i wersja.

Plik *values.yaml*

Istnieje również plik o nazwie *values.yaml*, zawierający ustawienia modyfikowalne przez użytkownika, które udostępnił autor wykresu:

```
environment: development
container:
  name: demo
  port: 8888
  image: cloudnative-devops/demo
  tag: hello
  replicas: 1
```

Plik przypomina trochę manifest YAML Kubernetes, ale jest jedna ważna różnica. Plik *values.yaml* ma całkowicie wolny format YAML, bez predefiniowanego schematu; to Ty decydujesz, które zmienne są zdefiniowane, jakie są ich nazwy i wartości.

W wykresie Helm nie muszą znajdować się żadne zmienne, ale jeśli są, możesz umieścić je w pliku *values.yaml*, a następnie odwoływać się do nich w innym miejscu na wykresie.

Na razie zignoruj pliki *production-values.yaml* i *staging-values.yaml*; wkrótce wyjaśnimy, po co istnieją.

Szablony Helm

Gdzie znajdują się referencje do tych zmiennych? Jeśli spojrzysz na podkatalog *templates*, zobaczyś kilka znajomych plików:

```
ls k8s/demo/templates  
deployment.yaml service.yaml
```

Są one dokładnie takie same jak pliki manifestów Deployment oraz Serwis z poprzedniego przykładu, tyle że teraz są to *szablony* (ang. *templates*): zamiast odwoływać się np. do nazwy kontenera bezpośrednio, zawierają symbol zastępczy, który Helm zastąpi rzeczywistą wartością z pliku *values.yaml*.

Oto szablon Deployment:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: {{ .Values.container.name }}-{{ .Values.environment }}  
spec:  
  replicas: {{ .Values.replicas }}  
  selector:  
    matchLabels:  
      app: {{ .Values.container.name }}  
  template:  
    metadata:  
      labels:  
        app: {{ .Values.container.name }}  
        environment: {{ .Values.environment }}  
    spec:  
      containers:  
        - name: {{ .Values.container.name }}  
          image: {{ .Values.container.image }}:{{ .Values.container.tag }}  
          ports:  
            - containerPort: {{ .Values.container.port }}  
      env:  
        - name: ENVIRONMENT  
          value: {{ .Values.environment }}
```



Nawiasy klamrowe wskazują miejsce, w którym Helm powinien zastąpić wartość zmiennej, ale w rzeczywistości są one częścią *składni szablonu Go*.

(Tak, Go jest wszędzie. Kubernetes i Helm są napisane w Go, więc nic dziwnego, że wykresy Helm używają szablonów Go).

Zmienne interpolacyjne

W tym szablonie znajduje się kilka zmiennych:

```
...  
metadata:  
  name: {{ .Values.container.name }}-{{ .Values.environment }}
```

Ta cała sekcja tekstu, w tym nawiasy klamrowe, zostanie *interpolowana* (tzn. zastąpiona) wartościami `container.name` i `environment`, zaczerpniętymi z pliku `values.yaml`. Wygenerowany wynik będzie wyglądał następująco:

```
...
metadata:
  name: demo-development
```

Jest to bardzo przydatne, ponieważ do takich wartości jak `container.name` jest dużo więcej odwołań w szablonie. W szablonie Serwis również występuje takie odwołanie:

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.container.name }}-service-{{ .Values.environment }}
  labels:
    app: {{ .Values.container.name }}
spec:
  ports:
  - port: {{ .Values.container.port }}
    protocol: TCP
    targetPort: {{ .Values.container.port }}
  selector:
    app: {{ .Values.container.name }}
  type: ClusterIP
```

Dla przykładu możesz zobaczyć, jak dużo odwołań występuje do `.Values.container.name`. Nawet w takim prostym wykresie musisz powtarzać te same fragmenty informacji wiele razy. Zastosowanie zmiennych Helm eliminuje to powielanie. Aby np. zmienić nazwę kontenera, musisz tylko wyedytać plik `values.yaml` i ponownie zainstalować wykres. Zmiana zostanie rozpropagowana we wszystkich szablonach.

Format szablonu Go jest bardzo rozbudowany i można go wykorzystywać do wykonywania znacznie większych operacji niż tylko prostego zastępowania zmiennych; obsługuje pętle, wyrażenia, instrukcje warunkowe, a nawet wywołuje funkcje. Wykresy Helm mogą używać tych funkcji do generowania dość złożonej konfiguracji na podstawie wartości wejściowych, w przeciwieństwie do prostych podstawień z naszego przykładu.

Więcej na ten temat możesz przeczytać w dokumentacji Helm (https://docs.helm.sh/chart_template_guide).

Wartości tekstowe w szablonach

Aby automatycznie otoczyć daną wartość znakiem cudzysłowu, możesz wykorzystać funkcję `quote`:

```
name: {{.Values.MyName | quote }}
```

Cudzysłowu można użyć tylko dla wartości tekstowych; nie stosuj funkcji `quote` z wartościami liczbowymi, takimi jak numery portów.

Określanie zależności

Co zrobić, jeśli Twój wykres opiera się na wartościach innych wykresów? Jeśli np. Twoja aplikacja korzysta z Redis, wykres Helm dla tej aplikacji może wymagać określenia wykresu redis jako zależności.

Możesz to zrobić za pomocą pliku *requirements.yaml*:

```
dependencies:  
  - name: redis  
    version: 1.2.3  
  - name: nginx  
    version: 3.2.1
```

Teraz uruchom polecenie `helm dependency update`, a Helm pobierze te wykresy jako gotowe do zainstalowania wraz z własną aplikacją.

Wdrażanie wykresów Helm

Spójrzmy na to, co faktycznie wiąże się z użyciem wykresu Helm w czasie wdrożenia aplikacji. Jedeną z najcenniejszych funkcji tego wykresu jest możliwość określania, zmiany, aktualizacji i zastępowania ustawień konfiguracji. W tym podrozdziale pokażemy, jak to działa.

Ustawianie zmiennych

Widzieliśmy, że autor wykresu Helm może umieścić wszystkie modyfikowalne przez użytkownika ustawienia w pliku *values.yaml* wraz z wartościami domyślnymi dla tych ustawień. Jak więc użytkownik wykresu zmienia lub zastępuje te ustawienia, aby dopasować je do swojego lokalnego środowiska? Polecenie `helm install` pozwala określić w wierszu polecenia dodatkowe pliki wartości, które zastąpią wartości domyślne w pliku *values.yaml*. Spójrz na przykład.

Tworzenie zmiennej środowiskowej

Założymy, że chcesz wdrożyć wersję aplikacji w środowisku staging. Na potrzeby naszego przykładu naprawdę nie ma znaczenia, co to oznacza w praktyce, ale założymy, że aplikacja rozpoznaje, czy jest w fazie staging, czy produkcyjnej w oparciu o wartość zmiennej środowiskowej o nazwie ENVIRONMENT i odpowiednio dostosowuje swoje zachowanie. Jak tworzona jest ta zmienna środowiskowa?

Popatrzmy ponownie na szablon *deployment.yaml*. Zmienna środowiskowa jest dostarczana do kontenera za pomocą tego kodu:

```
...  
env:  
  - name: ENVIRONMENT  
    value: {{ .Values.environment }}
```

Wartość environment pochodzi z pliku *values.yaml*:

```
environment: development  
...  
...
```

Tak więc zainstalowanie wykresu z wartościami domyślnymi spowoduje, że zmienna ENVIRONMENT kontenera będzie mieć wartość development. Założmy, że chcesz zmienić tę wartość na staging. Możesz wyedytować plik *values.yaml*, jak widziałeś, ale lepszym sposobem jest utworzenie dodatkowego pliku YAML zawierającego wartość tylko jednej zmiennej:

```
environment: staging
```

Tę wartość znajdziesz w pliku *k8s/demo/staging-values.yaml*. Nie jest on częścią wykresu Helm — udostępniłyśmy go, aby zaoszczędzić trochę pisania.

Określanie opcji podczas instalacji Helm

Aby określić dodatkowe opcje w poleceniu `helm install`, użyj opcji `--values`, tak jak poniżej:

```
helm install --name demo-staging --values=./k8s/demo/staging-values.yaml  
./k8s/demo ...
```

Spowoduje to utworzenie nowej wersji o nowej nazwie (`demo-staging`), a zmienna ENVIRONMENT uruchomionego kontenera zostanie ustawiona na wersję `staging` zamiast `development`. Zmienne wymienione w pliku dodatkowych wartości, który określiliśmy za pomocą `--values`, są łączone ze zmiennymi z pliku wartości domyślnych (*values.yaml*). W tym przypadku jest tylko jedna zmienna (`environment`), a wartości z pliku *staging-values.yaml* zastępują te z pliku wartości domyślnych.

Za pomocą flagi `--set` możesz także określić wartości, które podaje się bezpośrednio przy poleceniu `helm install` — nie jest to jednak zgodne z duchem pojęcia infrastruktury jako kodu. Zamiast tego, aby dostosować ustawienia wykresu Helm, utwórz plik YAML, który zastępuje wartość domyślną, np. plik *staging-values.yaml* — tak jak w przykładzie — i zastosuj go w wierszu poleceń za pomocą flagi `--values`.

Taki sposób ustawiania konfiguracji może być także zastosowany w przypadku wykresów publicznych. Aby wyświetlić listę wartości, które można ustawić dla wykresu, uruchom polecenie `helm inspect values` z nazwą wykresu:

```
helm inspect values stable/prometheus
```

Aktualizowanie aplikacji za pomocą Helm

Nauczyłeś się, jak instalować wykres Helm z wartościami domyślnymi i plikiem wartości niestandardowych, ale jak zmienić niektóre wartości dla aplikacji, która już działa?

Służy do tego polecenie `helm upgrade`. Założmy, że chcesz zmienić liczbę replik (liczbę kopii Poda, które Kubernetes powinien uruchomić) dla aplikacji demo. Domyślna wartość to 1, co widać w pliku *values.yaml*:

```
replicas: 1
```

Wiesz, jak to zmienić za pomocą pliku wartości niestandardowych. Wyedytuj więc plik *staging-values.yaml*, aby dodać odpowiednie ustawienie:

```
environment: staging  
replicas: 2
```

Uruchom następującą komendę, aby zastosować zmiany w *istniejącym* obiekcie Deployment demo-staging:

```
helm upgrade demo-staging --values=./k8s/demo/staging-values.yaml ./k8s/demo
Release "demo-staging" has been upgraded. Happy Helm-ing!
```

Możesz uruchomić polecenie helm upgrade tyle razy, ile chcesz, aby zaktualizować działający Deployment.

Powrót do poprzednich wersji

Jeśli uznasz, że nie podoba Ci się właśnie wdrożona wersja lub okazuje się, że jest z nią problem, możesz łatwo przywrócić poprzednią wersję; skorzystaj z polecenia helm rollback i określ numer poprzedniej wersji (co pokazano w wyniku polecenia helm history):

```
helm rollback demo-staging 1
Rollback was a success! Happy Helm-ing!
```

W rzeczywistości nie musisz wracać do poprzedniej wersji; powiedzmy, że wycofujesz się do wersji 1., a następnie zdecydujesz, że chcesz przejść do wersji 2. Jeśli uruchomisz polecenie helm rollback demo-staging 2, dokładnie tak się stanie.

Automatyczny powrót za pomocą helm-monitor

Helm może także automatycznie powrócić do poprzedniej wersji, bazując na odpowiednich metrykach (patrz rozdział 16.). Założmy np., że używasz wykresu Helm w ciągłym procesie wdrażania (patrz rozdział 14.). Możesz automatycznie cofnąć wydanie, jeśli liczba błędów zarejestrowanych przez system monitorowania przekroczy określona wartość.

Aby to zrobić, możesz użyć wtyczki helm-monitor. Może ona wysyłać zapytania do serwera Prometheus (patrz „Prometheus” w rozdziale 16.) w poszukiwaniu dowolnej metryki i wywołać proces wycofania, jeśli zapytanie się powiedzie. Moduł rozszerzający helm-monitor będzie obserwował metrykę przez pięć minut i wywoła proces cofania wersji, jeśli wykryje problem w tym czasie. Więcej informacji na temat helm-monitor znajduje się w tym poście na blogu (<https://containersolutions.com/automated-rollback-helm-releases-based-logs-metrics/>).

Tworzenie repozytorium wykresów Helm

Do tej pory używaliśmy wykresów Helm do instalowania wykresów z lokalnego katalogu lub ze stałego repozytorium. Aby korzystać z tych wykresów, nie potrzebujesz własnego repozytorium wykresów, ponieważ często przechowuje się wykres Helm aplikacji w repozytorium aplikacji.

Jeśli jednak chcesz utworzyć własne repozytorium wykresów Helm, jest to bardzo proste. Wykresy muszą być dostępne przez HTTP. Możesz to osiągnąć na wiele sposobów: umieść je w obiekcie bucket w chmurze, skorzystaj z GitHub Pages lub istniejącego serwera internetowego, jeśli taki masz.

Po zebraniu wszystkich wykresów w jednym katalogu uruchom w nim polecenie helm repo index, aby utworzyć plik index.yaml zawierający metadane repo.

Twoje repozytorium wykresów jest gotowe do użycia! Więcej informacji na temat zarządzania repozytoriami wykresów znajduje się w dokumentacji Helm (https://docs.helm.sh/developing_charts/#the-chart-repository-guide).

Aby zainstalować wykresy z repozytorium, musisz przede wszystkim dodać repozytorium do listy Helm:

```
helm repo add myrepo http://myrepo.example.com
helm install myrepo/myapp
```

Zarządzanie obiektami Secret wykresów Helm za pomocą Sops

W rozdziale 10. napisaliśmy, jak przechowywać poufne dane w Kubernetes i jak przekazywać je do aplikacji za pomocą zmiennych środowiskowych lub montowanych plików. Jeśli masz więcej niż jeden lub dwa obiekty Secret do zarządzania, łatwiejsze może być utworzenie jednego pliku zawierającego wszystkie te obiekty — zamiast pojedynczych plików, z których każdy zawiera jeden Secret. A jeśli używasz wykresu Helm do wdrożenia aplikacji, możesz sprawić, by ten plik stał się plikiem wartości, i zaszyfrować go za pomocą Sops (patrz „Szyfrowanie obiektów Secret za pomocą Sops” w rozdziale 10.).

Opracowaliśmy przykład, który możesz pobrać z repozytorium demo, z katalogu *hello-sops*:

```
cd hello-sops
tree
.
  k8s
    demo
      Chart.yaml
      production-secrets.yaml
      production-values.yaml
      staging-secrets.yaml
      staging-values.yaml
      templates
        deployment.yaml
        secrets.yaml
        values.yaml
        temp.yaml
  3 directories, 9 files
```

Wygląd wykresu Helm jest podobny do wykresu z naszego wcześniejszego przykładu (patrz „Co znajduje się w wykresie narzędzia Helm?” w tym rozdziale). Tutaj zdefiniowaliśmy Deployment i Secret. Jednak w tym przypadku dodaliśmy modyfikację, aby ułatwić zarządzanie wieloma obiektami Secret dla różnych środowisk.

Zobaczmy te obiekty Secret, których nasza aplikacja będzie potrzebować:

```
cat k8s/demo/production-secrets.yaml
secret_one: ENC[AES256_GCM,data:ekH3xIdCFiS4j1I2ja8=,iv:C95Ki1XL...1g==,type:str]
secret_two: ENC[AES256_GCM,data:0Xcmmlcdv3TbfM3mIkA=,iv:PQ0cI9vX...XQ==,type:str]
...
...
```

W tym miejscu wykorzystaliśmy Sops do szyfrowania wartości wielu obiektów Secret użytych przez naszą aplikację.

Teraz spójrz na plik Kubernetes *secrets.yaml*:

```
cat k8s/demo/templates/secrets.yaml
apiVersion: v1
kind: Secret
metadata:
  name: {{ .Values.container.name }}-secrets
type: Opaque
data:
  {{ $environment := .Values.environment }}
  app_secrets.yaml: {{ .Files.Get (nospace (cat $environment "-secrets.yaml"))
    | b64enc }}
```

W dwóch ostatnich wierszach dodaliśmy szablon Go do wykresu Helm, aby odczytać obiekty Secret z plików *production-secrets.yaml* lub *staging-secrets.yaml*, w zależności od tego, które środowisko jest ustawione w pliku *values.yaml*.

Rezultatem końcowym będzie pojedynczy manifest Kubernetes Secret o nazwie *demo-secrets*, zawierający *wszystkie* pary klucz-wartość zdefiniowane w obu plikach z obiektami Secret. Sekret zostanie zamontowany w obiekcie Deployment jako pojedynczy plik o nazwie *secrets.yaml* do użycia przez aplikację.

Dodaliśmy również ... | *b64enc* na końcu ostatniej linii. Jest to kolejny przydatny skrót wykorzystujący szablon Helm w Go do automatycznej konwersji danych Secret z postaci zwykłego tekstu na base64 —domyślne kodowanie Kubernetes dla Secret (patrz „base64” w rozdziale 10.).

Najpierw przy użyciu Sops musimy tymczasowo odszyfrować pliki, a następnie zastosować zmiany w klastrze Kubernetes. Oto potok poleceń służący do wdrażania wersji staging aplikacji demo — z obiektami Secret staging:

```
sops -d k8s/demo/staging-secrets.yaml > temp-staging-secrets.yaml && \
helm upgrade --install staging-demo --values staging-values.yaml \
--values temp-staging-secrets.yaml ./k8s/demo && rm temp-staging-secrets.yaml
```

A tak to działa.

1. Sops odszyfrowuje plik *staging-secrets* i zapisuje odszyfrowane dane wyjściowe do *temp-staging-secrets*.
2. Helm instaluje wykres demo przy użyciu wartości ze *staging-secrets* oraz *temp-staging-secrets*.
3. Plik *temp-staging-secrets* jest usuwany.

Ponieważ wszystko to dzieje się w jednym kroku, nie pozostawiamy pliku z jawnym tekstem, aby nie dostał się w niepowołane ręce.

Zarządzanie wieloma wykresami za pomocą Helmfile

Kiedy wprowadzaliśmy Helm w podrozdziale „Helm: menadżer pakietów Kubernetes” w rozdziale 4., pokazaliśmy, jak wdrożyć wykres Helm aplikacji demo w klastrze Kubernetes. Helm działa tylko na jednym wykresie jednocześnie. Skąd wiesz, jakie aplikacje powinny działać w klastrze wraz z niestandardowymi ustawieniami zastosowanymi podczas instalacji z Helm?

Istnieje użyteczne narzędzie o nazwie Helmfile (<https://github.com/roboll/helmfile>), które może Ci w tym pomóc. O ile Helm umożliwia wdrażanie jednej aplikacji przy użyciu szablonów i zmiennych, Helmfile pozwala na wdrażanie wszystkiego, co powinno być zainstalowane w klastrze, za pomocą jednego polecenia.

Co znajduje się w pliku Helmfile?

W repozytorium demo znajduje się przykład użycia Helmfile. W folderze *hello-helmfile* znajdziesz plik *helmfile.yaml*:

```
repositories:
  - name: stable
    url: https://kubernetes-charts.storage.googleapis.com/
releases:
  - name: demo
    namespace: demo
    chart: ./hello-helm3/k8s/demo
    values:
      - "../hello-helm3/k8s/demo/production-values.yaml"
  - name: kube-state-metrics
    namespace: kube-state-metrics
    chart: stable/kube-state-metrics
  - name: prometheus
    namespace: prometheus
    chart: stable/prometheus
    set:
      - name: rbac.create
        value: true
```

Sekcja *repositories* definiuje repozytoria wykresów Helm, do których będziemy się odnosić. W tym przypadku jedyne repozytorium to *stable*, oficjalne stabilne repozytorium wykresów Kubernetes. Jeśli używasz własnego repozytorium wykresów Helm (patrz „Tworzenie repozytorium wykresów Helm” w tym rozdziale), dodaj je tutaj.

Następnie definiujemy zestaw wersji aplikacji (ang. *releases*), czyli tych aplikacji, które chcielibyśmy wdrożyć w klastrze.

Każda z tych wersji określa następujące metadane:

- nazwę (name) wykresu Helm do wdrożenia,
- przestrzeń nazw (namespace), w której ma zostać wdrożony,
- chart — adres URL lub ścieżkę do samego wykresu,
- values — ścieżkę do pliku *values.yaml*, który ma być używany z obiektem Deployment,
- set — dodatkowe wartości oprócz zawartych w pliku wartości.

Zdefiniowaliśmy tutaj trzy wersje: aplikację *demo*, *Prometheus* (patrz „Prometheus” w rozdziale 16.) oraz *kube-state-metrics* (patrz „Metryki Kubernetes” w rozdziale 16.).

Metadane wykresu

Pamiętaj, że podaliśmy ścieżkę względną do wykresu demo i plików wartości:

```
- name: demo
  namespace: demo
  chart: ../hello-helm3/k8s/demo
  values:
    - '../hello-helm3/k8s/demo/production-values.yaml"
```

Aby *Helmfile* mógł zarządzać wykresami, Twoje wykresy nie muszą znajdować się w repozytorium wykresów; możesz np. zachować je wszystkie w tym samym repozytorium kodu źródłowego.

Dla wykresu prometheus podaliśmy po prostu wartość stable/prometheus. Ponieważ nie jest to ścieżka systemu plików, *Helmfile* szuka wykresu w repozytorium stable, które zdefiniowaliśmy wcześniej w sekcji repositories:

```
- name: stable
  url: https://kubernetes-charts.storage.googleapis.com/
```

Wszystkie wykresy mają różne wartości domyślne ustawione w odpowiednich plikach *values.yaml*. W sekcji set: pliku *Helmfile* możesz określić dowolne wartości, które chcesz zastąpić podczas instalowania aplikacji.

W tym przykładzie dla wersji prometheus zmieniliśmy wartość domyślną dla rbac.create z false na true:

```
- name: prometheus
  namespace: prometheus
  chart: stable/prometheus
  set:
    - name: rbac.create
      value: true
```

Stosowanie pliku Helmfile

Plik *helmfile.yaml* określa zatem wszystko, co powinno być uruchomione w klastrze (lub przynajmniej jego podzbiorze) w sposób deklaratywny, tak jak w przypadku manifestów Kubernetes. Kiedy zastosujesz ten deklaratywny manifest, *Helmfile* dostosuje klaster do Twojej specyfikacji.

Aby to zrobić, uruchom:

```
helmfile sync
exec: helm repo add stable https://kubernetes-charts.storage.googleapis.com/ "stable" has
↳ been added to your repositories
exec: helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "cloudnative-devops" chart repository
...Successfully got an update from the "stable" chart repository

Update Complete. * Happy Helming!*
```

```
exec: helm dependency update .../demo/hello-helm3/k8s/demo
...
```

To tak, jakbyś uruchomił polecenia `helm install`/`helm upgrade` dla każdego zdefiniowanego wykresu Helm.

Możesz np. automatycznie uruchamiać polecenie `helm sync` jako część ciągłego procesu wdrażania (patrz rozdział 14.). Zamiast ręcznie uruchamiać polecenie `helm install` w celu dodania nowej aplikacji do klastra, możesz po prostu wyedytować plik *Helmfile*, zatwierdzić do repozytorium i poczekać, aż automatyzacja wprowadzi zmiany.



Używaj tylko jednego źródła. Nie wdrażaj ręcznie pojedynczych wykresów Helm. Zamiast tego deklaratywnie zarządzaj wszystkimi swoimi wykresami w klastrze za pomocą *Helmfile*. Wybierz jedno lub drugie. Jeśli zastosujesz plik *Helmfile*, a następnie użyjesz Helm do wdrożenia lub modyfikacji aplikacji poza nim, plik *Helmfile* nie będzie już jedynym źródłem dla Twojego klastra. To z pewnością doprowadzi do problemów. Jeśli zatem używasz *Helmfile*, wykonaj wszystkie wdrożenia tylko przy użyciu *Helmfile*.

Jeśli plik *Helmfile* nie przypadnie Ci do gustu, istnieje kilka innych narzędzi, które działają mniej więcej tak samo:

- *Landscaper* (<https://github.com/Eneco/landscaper>),
- *Helmsman* (<https://github.com/Praqma/helmsman>).

Podobnie jak w przypadku każdego nowego narzędzia, zalecamy zapoznanie się z dokumentacją, porównanie różnych opcji, wypróbowanie ich, a następnie podjęcie decyzji, które z nich najbardziej Ci odpowiada.

Zaawansowane narzędzia do zarządzania manifestami

Chociaż Helm jest doskonałym i szeroko stosowanym narzędziem, posiada kilka ograniczeń. Tworzenie i edycja szablonów Helm nie są łatwe. Pliki YAML Kubernetes są skomplikowane, duże i powtarzalne, podobnie szablony Helm.

W przygotowaniu jest kilka nowych narzędzi, które próbują rozwiązać te problemy i mają ułatwić pracę z manifestami Kubernetes: używają bardziej zaawansowanego języka niż YAML, np. *Jsonnet*, lub grupują pliki YAML w podstawowe wzorce i dostosowują je za pomocą dodatkowych plików.

Tanka

Czasami zastosowanie deklaratywnego YAML nie jest wystarczające; dzieje się tak szczególnie w dużych i złożonych wdrożeniach, w których musisz wykorzystywać obliczenia i logikę. Przykładowo masz ustawać liczbę replik dynamicznie w zależności od wielkości klastra.

Do tego potrzebny jest prawdziwy język programowania.

Narzędzie Tanka pozwala tworzyć manifesty Kubernetes za pomocą języka o nazwie *Jsonnet* — rozszerzonej wersji JSON (który jest deklaratywnym formatem danych równoważnym YAML, a Kubernetes może również zrozumieć manifesty formatu JSON). *Jsonnet* dodaje do JSON ważne możliwości, takie jak zmienne, pętle, arytmetykę, instrukcje warunkowe, obsługę błędów itd.

kapitan

Narzędzie kapitan (<https://github.com/deepmind/kapitan>) to kolejne narzędzie manifestu oparte na Jsonnet, koncentrujące się na udostępnianiu wartości konfiguracyjnych w wielu aplikacjach, a nawet klastrach. Ma ono hierarchiczną bazę danych wartości konfiguracji (zwaną *inwentarzem*), która pozwala ponownie używać wzorców manifestów, podłączając różne wartości, w zależności od środowiska lub aplikacji:

```
local kube = import "lib/kube.libjsonnet";
local kap = import "lib/kapitan.libjsonnet";
local inventory = kap.inventory();
local p = inventory.parameters;

{
    "00_namespace": kube.Namespace(p.namespace),
    "10_serviceaccount": kube.ServiceAccount("default")
}
```

kustomize

Narzędzie kustomize (<https://github.com/kubernetes-sigs/kustomize>) to kolejne narzędzie do zarządzania manifestami, które używa zwykłego YAML zamiast szablonów lub alternatywnego języka, takiego jak Jsonnet. Zaczynasz od podstawowego manifestu YAML, a następnie korzystasz z nakładek (ang. *overlays*), aby odpowiednio dostosować manifest dla różnych środowisk lub konfiguracji. Narzędzie kustomize wygeneruje ostateczne manifesty z plików podstawowych oraz nakładek:

```
namePrefix: staging
commonLabels:
  environment: staging
  org: acmeCorporation
commonAnnotations:
  note: Hello, I am staging!
bases:
- ../../base
patchesStrategicMerge:
- map.yaml
EOF
```

Oznacza to, że wdrażanie manifestów może być tak proste jak uruchomienie polecenia:

```
kustomize build /myApp/overlays/staging | kubectl apply -f -
```

Jeśli szablony lub Jsonnet nie przemawiają do Ciebie i chcesz mieć możliwość pracy z prostymi manifestami Kubernetes, warto to narzędzie sprawdzić.

kompose

Jeśli korzystasz z usług produkcyjnych w kontenerach Docker, ale nie używasz Kubernetes, możesz znać Docker Compose.

Compose pozwala definiować i wdrażać zestawy kontenerów, które współpracują; przykładem mogą być serwer WWW, backend i baza danych, taka jak Redis. Pojedynczego pliku *docker-compose.yml* można użyć do zdefiniowania, w jaki sposób te kontenery komunikują się ze sobą.

Narzędzie kompose (<https://github.com/kubernetes/kompose>) to narzędzie służące do konwersji pliku `docker-compose.yml` do manifestów Kubernetes. Pomoże Ci w migracji z Docker Compose do Kubernetes, bez konieczności pisania własnych manifestów Kubernetes lub wykresów Helm od zera.

Ansible

Być może znasz już popularne narzędzie do automatyzacji zwane Ansible. Nie zostało utworzone z myślą o Kubernetes, ale może zarządzać wieloma różnymi rodzajami zasobów, wykorzystując moduły rozszerzeń — podobnie jak Puppet (patrz „Moduł Puppet dla Kubernetes” w rozdziale 3.).

Oprócz instalowania i konfigurowania klastrów Kubernetes, Ansible może bezpośrednio zarządzać zasobami Kubernetes, takimi jak Deployment i Serwis przy użyciu modułu `k8s` (https://docs.ansible.com/ansible/latest/modules/k8s_module.html).

Podobnie jak Helm, Ansible może korzystać z szablonów manifestu Kubernetes przy użyciu standardowego języka szablonów (Jinja). Zawiera zaawansowaną obsługę zmiennych przy użyciu systemu hierarchicznego. Można np. ustawić wspólne wartości dla grupy aplikacji lub środowiska wdrażania, takiego jak *staging*.

Jeśli już używasz Ansible w swojej organizacji, zdecydowanie warto je zastosować również do zarządzania zasobami Kubernetes. Ansible może być zbyt rozbudowanym narzędziem, jeśli Twój infrastruktury oparta jest wyłącznie na Kubernetes. Jednak w przypadku infrastruktur mieszanych pomocne może być użycie tylko jednego narzędzia do zarządzania wszystkim:

```
kube_resource_configmaps:  
  my-resource-env: "{{ lookup(template, template_dir +  
    '/my-resource-env.j2) }}"  
  kube_resource_manifest_files: "{{ lookup(fileglob, template_dir +  
    '/manifest.yml) }}"  
  - hosts: "{{ application }}-{{ env }}-runner"  
    roles:  
      - kube-resource
```

Ekspert ds. Ansible Will Thamess pokazuje niektóre możliwości zarządzania Kubernetes za pomocą Ansible (<http://willthames.github.io/ansiblefest2018>).

kubeval

W przeciwieństwie do innych narzędzi, które omówiliśmy w tym podrozdziale, kubeval (<https://github.com/garethr/kubeval>) nie służy do generowania ani tworzenia szablonów manifestów Kubernetes, ale do ich sprawdzania.

Każda wersja Kubernetes ma inny schemat dla manifestów YAML lub JSON i ważne jest, aby można było automatycznie sprawdzać, czy manifesty są zgodne ze schematem. Przykładowo kubeval sprawdzi, czy podałeś wszystkie wymagane pola dla określonego obiektu i czy wartości są poprawnego typu.

`kubectl` również sprawdza manifesty po ich zastosowaniu i wyświetla błąd podczas próby użycia nieprawidłowego manifestu, ale bardzo przydatna jest też ich wcześniejsza weryfikacja. Narzędzie kubeval nie potrzebuje dostępu do klastra, a także może sprawdzać poprawność względem dowolnej wersji Kubernetes.

Dobrym pomysłem jest dodanie kubeval do potoku ciągłego wdrażania, aby manifesty były automatycznie sprawdzane przy każdej zmianie. Możesz także użyć kubeval do przetestowania np. tego, czy Twoje manifesty wymagają jakichkolwiek poprawek, aby działać na najnowszej wersji Kubernetes przed faktyczną aktualizacją.

Podsumowanie

Chociaż możesz wdrażać aplikacje w Kubernetes przy użyciu samych manifestów YAML, jest to niewygodne. Helm to potężne narzędzie, które może Ci w tym pomóc — pod warunkiem że wiesz, jak z niego korzystać.

Obecnie opracowywanych jest wiele nowych narzędzi, które znacznie ułatwiają wdrożenia w Kubernetes w przyszłości. Z niektórych funkcji można także skorzystać za pomocą Helm. Tak czy inaczej ważne jest, aby zapoznać się z podstawami korzystania z wykresu Helm.

- Wykres jest specyfikacją pakietu Helm, obejmującą metadane dotyczące pakietu, niektóre jego wartości konfiguracyjne oraz szablony obiektów Kubernetes, które odwołują się do tych wartości.
- Zainstalowanie wykresu tworzy wersję Helm. Za każdym razem, gdy instalujesz instancję wykresu, tworzona jest nowa wersja. Gdy ją aktualizujesz, Helm zwiększa numer wersji.
- Aby dostosować wykres Helm do własnych wymagań, utwórz plik wartości niestandardowych przesłaniający tylko te ustawienia, które są dla Ciebie ważne, i dodaj go do polecenia `helm install` lub `helm upgrade`.
- Możesz użyć zmiennej (np. `environment`), aby wybrać różne zestawy wartości lub obiekty Secret w zależności od środowiska wdrażania, takiego jak produkcyjne, testowe itd.
- Za pomocą *Helmfile* można deklaratywnie określić zestaw wykresów Helma i wartości, które mają być zastosowane w klastrze, oraz zainstalować lub zaktualizować je wszystkie za pomocą jednego polecenia.
- Helm może być używany razem z Sops do obsługi konfiguracji obiektów Secret dla Twoich wykresów. Może także używać funkcji do automatycznego kodowania Secret za pomocą base64 — standardowo stosowanego w Kubernetes.
- Helm nie jest jedynym dostępnym narzędziem do zarządzania manifestami Kubernetes. Są dostępne narzędzia Tanka oraz Kapitan, które używają Jsonnet, innego języka szablonów; kustomize stosuje inne podejście i zamiast interpolować zmienne, używa nakładek YAML do konfigurowania manifestów.
- Do szybkiego testowania i sprawdzania manifestów służy kubeval. Narzędzie kubeval sprawdzi poprawną składnię i typowe błędy w manifestach.

Proces tworzenia oprogramowania

Surfing to taka niesamowita koncepcja. Radzisz sobie z naturą za pomocą malej dechy, mówiąc: Będę pływał! A natura wiele razy odpowiada: Wcale nie! I sprowadza Cię na dno.

— Jolene Blalock

Ten rozdział jest kontynuacją rozdziału 12. Skupimy się w nim na całym cyklu życia aplikacji, od rozwoju w środowisku lokalnym do wdrażania aktualizacji w klastrze Kubernetes; poruszmy w nim też trudny temat migracji baz danych. Omówimy niektóre narzędzia, które pomogą Ci rozwijać, testować i wdrażać aplikacje, takie jak Skaffold, Draft, Telepresence i Knative. Gdy będziesz gotowy do wdrożenia aplikacji w klastrze, powiemy o bardziej złożonych wdrożeniach za pomocą wykresów Helm i funkcji hook.

Narzędzia programistyczne

W rozdziale 12. poznaleś niektóre narzędzia, które pomagają pisać, budować i wdrażać manifesty zasobów Kubernetes. To dobre rozwiązania. Jednak często, gdy tworzysz aplikację działającą w Kubernetes, chcesz wypróbować i natychmiast zobaczyć zmiany, bez przechodzenia przez pełną pętlę komplikacji-wysyłania-wdrażania-aktualizacji.

Skaffold

Skaffold (<https://github.com/GoogleContainerTools/skaffold>) to narzędzie Google o otwartym kodzie źródłowym, zaprojektowane w celu zapewnienia szybkiego przepływu pracy w środowisku lokalnym. Automatycznie odbudowuje kontenery i wdraża te zmiany w klastrze lokalnym lub zdalnym.

Żądany przepływ pracy definiujesz w pliku `skaffold.yaml` w Twoim repozytorium. Aby rozpocząć zdefiniowany proces, wystarczy uruchomić narzędzie `skaffold`. Gdy wprowadzasz zmiany do plików w lokalnym katalogu, Skaffold wznawia pracę, buduje nowy kontener ze zmianami, a następnie wdraża go automatycznie, oszczędzając Twój czas.

Draft

Draft (<https://github.com/Azure/draft>) to narzędzie typu open source obsługiwane przez zespół Azure w firmie Microsoft. Podobnie jak Skaffold, może wykorzystywać Helm do automatycznego wdrażania aktualizacji w klastrze po zmianie kodu.

Draft wprowadza również koncepcję Draft Packs, czyli pliki Dockerfile i wykresy Helm dla dowolnego języka używanego przez Twoją aplikację. Obecnie dostępne są pakiety dla .NET, Go, Node, Erlang, Clojure, C #, PHP, Java, Python, Rust, Swift i Ruby.

Jeśli dopiero zaczynasz pracę z nową aplikacją, a nie masz jeszcze Dockerfile lub wykresów Helm, Draft może być idealnym narzędziem do szybkiego uruchomienia. Po wykonaniu polecenia `draft init && draft create` Draft sprawdzi pliki w lokalnym katalogu aplikacji i spróbuje ustalić, w jakim języku został napisany Twój kod. Następnie utworzy plik Dockerfile oraz wykres Helm dla Twojego języka.

Aby zastosować zmiany, należy uruchomić polecenie `draft up`. Draft zbuduje lokalny kontener Docker za pomocą utworzonego pliku Dockerfile i wdroży go w klastrze Kubernetes.

Telepresence

Telepresence (<https://www.telepresence.io/>) stosuje nieco inne podejście niż Skaffold i Draft. Nie potrzebujesz lokalnego klastra Kubernetes; Pod Telepresence działa w Twoim prawdziwym klastrze, oferując miejsce dla Twojej aplikacji. Ruch dla aplikacji Pod jest następnie przechwytywany i kierowany do kontenera uruchomionego na komputerze lokalnym.

Umożliwia to programistom włączenie lokalnej maszyny w zdalnym klastrze. Gdy wprowadzisz zmiany w kodzie aplikacji, zostaną one odzwierciedlone w żywym klastrze, bez konieczności wdrażania do niego nowego kontenera.

Knative

Podczas gdy inne narzędzia, które omówiliśmy, skupią się na przyspieszeniu lokalnego procesu programistycznego, Knative (<https://github.com/knative/docs>) jest bardziej ambitne. Ma na celu zapewnienie standardowego mechanizmu wdrażania wszelkiego rodzaju obciążen na Kubernetes, nie tylko aplikacji skonteneryzowanych, ale także funkcji w stylu *serverless*.

Knative integruje się zarówno z Kubernetes, jak i Istio (patrz „Istio” w rozdziale 9), aby zapewnić kompletną platformę wdrażania aplikacji lub funkcji, w tym konfigurację procesu komplikacji, automatyczne wdrażanie i mechanizm zdarzeń, który standaryzuje sposób, w jaki aplikacje używają systemów przesyłania i kolejkowania komunikatów (np. Pub/Sub, Kafka lub RabbitMQ).

Projekt Knative jest na wczesnym etapie, ale należy go obserwować.

Strategie wdrażania

Jeśli aktualizujesz działającą aplikację ręcznie, bez wykorzystania Kubernetes, jednym ze sposobów jest zamknięcie aplikacji, zainstalowanie nowej wersji i ponowne jej uruchomienie. Jednak to oznacza wystąpienie przestoju.

Jeśli masz wiele replik, lepszym sposobem byłoby uaktualnienie każdej z nich po kolei, tak aby nie było żadnych przerw w działaniu; to tzw. *zero-downtime deployment*.

Nie wszystkie aplikacje wymagają takiej strategii; usługi wewnętrzne, które np. obsługują kolejki komunikatów, są idempotentne, więc można je aktualizować jednocześnie. Oznacza to, że aktualizacja odbywa się szybciej, ale w przypadku aplikacji zorientowanych na użytkownika zwykle bardziej zależy nam na uniknięciu przestojów niż na szybkich aktualizacjach.

W Kubernetes możesz wybrać jedną z tych strategii, która jest najbardziej odpowiednia. Aktualizacje typu Rolling Update (`RollingUpdate`) oznaczają zero przestojów (aktualizacja Poda po Podzie), podczas gdy `Recreate` to szybka opcja aktualizacji wszystkich Podów naraz. Istnieje również kilka innych opcji, które możesz dostosować, aby uzyskać dokładnie takie zachowanie, jakiego potrzebujesz w swojej aplikacji.

W Kubernetes strategia wdrażania aplikacji jest zdefiniowana w manifeście Deployment. Domyślnie wybrana jest opcja `RollingUpdate`. Aby zmienić strategię na `Recreate`, ustaw ją w następujący sposób:

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  strategy:
    type: Recreate
```

Teraz przyjrzyjmy się bliżej tym strategiom wdrażania i zobaczymy, jak działają.

Rolling Updates

W aktualizacjach typu Rolling Updates Pody są aktualizowane pojedynczo, dopóki wszystkie repliki nie zostaną zastąpione przez nową wersję.

Wyobraźmy sobie np. aplikację z trzema replikami, z których każda działa w wersji v1. Programista rozpoczyna aktualizację do wersji v2 — za pomocą polecenia `kubectl apply ...` lub `helm upgrade ...`. Co się wtedy dzieje?

Najpierw jeden z trzech Podów v1 kończy swoje działanie. Kubernetes oznaczy go odpowiednią flagą i przestanie kierować do niego ruch. Nowy Pod v2 zajmie jego miejsce. Tymczasem pozostałe Pody v1 nadal otrzymują przychodzące żądania. Gdy czekamy, aż pierwszy Pod v2 będzie gotowy, mamy dwa działające Pody i nadal obsługujemy użytkowników.

Kiedy Pod v2 będzie gotowy, Kubernetes zacznie do niego wysyłać ruch użytkownika, tak jak w przypadku pozostałych dwóch Podów v1. Teraz wróciliśmy do pełnego zestawu trzech Podów.

Proces ten będzie kontynuowany, Pod po Podzie, aż wszystkie Pody v1 zostaną zastąpione Podami v2. Chociaż zdarzają się sytuacje, że dostępnych jest mniej niż zwykle Podów do obsługi ruchu, aplikacja jako całość nigdy nie jest wyłączona. To wdrożenie bez przestojów.



Podczas aktualizacji Rolling Update zarówno stare, jak i nowe wersje aplikacji będą działać jednocześnie. Chociaż zwykle nie stanowi to problemu, konieczne może być podjęcie kroków w celu zapewnienia bezpieczeństwa; gdyby zmiany dotyczyły np. migracji bazy danych (patrz „Obsługa migracji za pomocą Helm” w tym rozdziale), normalna aktualizacja typu Rolling Update nie będzie możliwa.

Jeśli Twoje Pody mogą ulec awarii w krótkim czasie po stanie gotowości, skorzystaj z pola `minReadySeconds`, aby poczekać z kontynuacją aktualizacji do momentu, aż każdy Pod się ustabilizuje (patrz „`minReadySeconds`” w rozdziale 5.).

Recreate

W trybie Recreate wszystkie działające repliki są kończone jednocześnie, a następnie tworzone są nowe.

W przypadku aplikacji, które nie obsługują żądań bezpośrednio, powinno to być dopuszczalne. Jedną z zalet trybu Recreate jest to, że pozwala uniknąć sytuacji, w której dwie różne wersje aplikacji działają jednocześnie (patrz „Rolling Updates” w tym rozdziale).

maxSurge i maxUnavailable

W miarę postępu przebiegu aktualizacji Rolling Update czasami będziesz mieć uruchomionych więcej Podów niż nominalna wartość replik, a czasem mniej. Zachowanie to regulują dwie ważne opcje — `maxSurge` i `maxUnavailable`.

- `maxSurge` ustawia maksymalną liczbę nadmiarowych Podów. Jeśli np. masz 10 replik i `maxSurge` ustawione na 30%, wówczas nie będzie można uruchomić więcej niż 13 Podów w tym samym czasie.
- `maxUnavailable` ustawia maksymalną liczbę Podów, która może być niedostępna. Przy nominalnych 10 replikach i `maxUnavailable` ustawionym na 20%, Kubernetes nigdy nie pozwoli, aby liczba dostępnych Podów spadła poniżej 8.

Opcje możesz ustawić jako wartość całkowitą lub jako procent:

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 20%
      maxUnavailable: 3
```

Zwykle wartości domyślne dla obu opcji (25% lub 1, w zależności od wersji Kubernetes) są w porządku i nie trzeba ich dostosowywać. W niektórych sytuacjach konieczna może być ich zmiana, aby upewnić się, że aplikacja będzie w stanie utrzymać akceptowalną pojemność podczas aktualizacji. Na bardzo

dużą skalę może się okazać, że działanie przy 75% dostępności jest niewystarczające i trzeba nieco zmniejszyć wartość `maxUnavailable`.

Im większa wartość `maxSurge`, tym szybciej odbywa się wdrażanie, ale tym więcej dodatkowego obciążenia ląduje na zasobach klastra. Duże wartości `maxUnavailable` także przyspieszają wdrażanie kosztem wydajności Twojej aplikacji.

Z drugiej strony, małe wartości `maxSurge` i `maxUnavailable` zmniejszają wpływ na kластer i użytkowników, ale mogą znacznie wydłużyć czas wdrażania. Tylko Ty możesz zdecydować, jaki kompromis jest odpowiedni dla Twojej aplikacji.

Wdrożenia niebiesko-zielone

We *wdrożeniach niebiesko-zielonych* zamiast unieruchamiać i zastępować Pody pojedynczo, tworzony jest zupełnie nowy Deployment. Obok istniejącego obiektu Deployment v1 uruchamiany jest nowy osobny stos Podów z wersją v2.

Zaletą rozwiązania jest to, że nie musisz jednocześnie obsługiwać zarówno starych, jak i nowych wersji aplikacji. Jednak klasterek będzie musiał być wystarczająco duży, aby uruchomić dwukrotnie większą liczbę replik wymaganych dla aplikacji, co może być kosztowne i oznaczać dużą ilość niewykorzystanej pojemności przez większość czasu (chyba że skalujesz klasterek w górę i jeśli trzeba, w dół).

W rozdziale 4., w punkcie „Serwis” pisaliśmy, że Kubernetes używa etykiet do decydowania, które Pody powinny odbierać ruch z zasobu Serwis. Jednym ze sposobów wdrożenia niebiesko-zielonego jest ustawienie różnych etykiet na starych i nowych Podach (patrz „Etykiety” w rozdziale 9.).

Po drobnym dopracowaniu definicji Serwis dla naszej przykładowej aplikacji możesz wysyłać ruch tylko do Podów oznaczonych etykietą `deployment: blue`:

```
apiVersion: v1
kind: Service
metadata:
  name: demo
spec:
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: demo
    deployment: blue
  type: ClusterIP
```

Wdrażając nową wersję, możesz oznaczyć ją etykietą `deployment: green`. Nie otrzyma żadnego ruchu, nawet gdy jest w pełni uruchomiona, ponieważ Serwis wysyła ruch tylko do niebieskich Podów. Możesz ją przetestować i upewnić się, że jest gotowa do podmiany.

Aby przełączyć się na nowy Deployment, wyedytuj Serwis i zmień selektor na `deployment: green`. Teraz nowe Pody `green` zaczną otrzymywać ruch; gdy wszystkie stare Pody `blue` będą bezczynne, możesz je usunąć.

Wdrożenia rainbow

W niektórych rzadkich przypadkach, szczególnie wtedy, gdy Pody utrzymują bardzo długotrwałe połączenia (np. gniazda sieciowe), tylko wdrożenie niebiesko-zielone może nie wystarczyć. Może być konieczne jednoczesne utrzymanie trzech lub więcej wersji aplikacji w tym samym czasie.

Taka strategia nazywana jest czasami *wdrożeniem rainbow*. Za każdym razem, gdy wdrażasz aktualizację, dostajesz nowy zestaw kolorów Podów. Gdy najstarsze Pody zamkną połączenia, będzie można je usunąć.

Brandon Dimcheff szczegółowo opisuje przykład wdrożenia rainbow (<https://github.com/bdimcheff/rainbow-deploys>).

Wdrożenia kanarkowe

Zaletą wdrożeń niebiesko-zielonych (lub rainbow) jest to, że jeśli zdecydujesz, iż nie podoba Ci się nowa wersja lub nie działa ona poprawnie, możesz po prostu wrócić do starej działającej wersji. Jest to jednak kosztowne, ponieważ potrzebujesz zdolności do jednoczesnego uruchamiania obu wersji.

Jednym z alternatywnych podejść, które pozwala uniknąć tego problemu, jest *wdrożenie kanarkowe* (ang. *canary deployment*). Podobnie jak w przypadku kanarka w kopalni węgla, niewielka grupa nowych Podów zostaje wystawiona w środowisku produkcyjnym, aby sprawdzić, co się z nimi stanie. Jeśli przeżyją, wdrożenie może być kontynuowane. Jeśli występuje problem, promień rażenia jest ścisłe ograniczony.

Podobnie jak w przypadku wdrożeń niebiesko-zielonych, można to zrobić za pomocą etykiet (patrz „Etykiety” w rozdziale 9.). Szczegółowy przykład uruchomienia wdrożenia kanarkowego znajduje się w dokumentacji Kubernetes (<https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#canary-deployments>).

Bardziej wyrafinowanym sposobem jest wykorzystanie usługi Istio (patrz „Istio” w rozdziale 9.), która pozwala losowo kierować zmienną część ruchu do jednego lub większej ilości serwisów. Ułatwia to także wykonywanie takich czynności jak testowanie A/B.

Obsługa migracji za pomocą Helm

Aplikacje bezstanowe można łatwo wdrażać i aktualizować. Jednak w przypadku bazy danych sytuacja może być bardziej skomplikowana. Zmiany w schemacie bazy danych zwykle wymagają uruchomienia zadania *migracji* w określonym momencie wdrażania. Przykładowo w aplikacjach Rails musisz uruchomić polecenie `rake db:migrate` przed uruchomieniem nowych Podów.

W Kubernetes możesz wykorzystać do tego zasób Job (zobacz „Kontroler Job” w rozdziale 9.). Możesz to zrobić za pomocą polecenia `kubectl` jako części procesu aktualizacji lub, jeśli używasz wykresu Helm, możesz użyć wbudowanej funkcji o nazwie *hook*.

Funkcja hook wykresu Helm

Funkcje hook pozwalają kontrolować kolejność zdarzeń podczas wdrażania. Pozwalają także przerwać aktualizację, jeśli coś pójdzie nie tak.

Oto przykład zadania migracji bazy danych dla aplikacji Rails, wdrożonej za pomocą wykresu Helm:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ .Values.appName }}-db-migrate
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  activeDeadlineSeconds: 60
  template:
    spec:
      containers:
        - name: {{ .Values.appName }}-migration-job
          image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
          command:
            - bundle
            - exec
            - rails
            - db:migrate
```

Właściwości `helm.sh/hook` są zdefiniowane w sekcji `annotations`:

```
annotations:
  "helm.sh/hook": pre-upgrade
  "helm.sh/hook-delete-policy": hook-succeeded
```

Ustawienie `pre-upgrade` to dla wykresu Helm informacja, aby zastosował ten manifest Job przed wykonaniem aktualizacji. Job uruchomi standardowe polecenie migracji Rails.

Opcja `"helm.sh/hook-delete-policy": hook-succeeded` informuje Helm, aby usunął Job, jeśli zakończy się pomyślnie (tzn. skończy pracę ze statusem 0).

Obsługa nieudanych funkcji hook

Jeśli Job zwróci niezerowy kod wyjścia, oznacza to, że wystąpił błąd i migracja nie zakończyła się pomyślnie. Helm pozostawi Job w miejscu, w którym przestał działać, abyś mógł debugować to, co poszło źle.

Jeśli tak się stanie, proces związany z nową wersją zostanie zatrzymany, a aplikacja nie zostanie zaktualizowana. Polecenie `kubectl get pods` spowoduje wyświetlenie Poda z błędem, umożliwiając sprawdzenie dzienników.

Po rozwiązaniu problemu możesz usunąć Job z błędem (`kubectl delete job <nazwa-job'a>`), a następnie spróbować ponownie wykonać aktualizację.

Inne funkcje hook

Funkcje hook mają inne opcje niż tylko pre-upgrade. Możesz użyć funkcji hook na dowolnym etapie aktualizacji wersji.

- pre-install jest wykonywana po renderowaniu szablonów, ale przed utworzeniem jakichkolwiek zasobów.
- post-install wykonuje się po załadowaniu wszystkich zasobów.
- pre-delete jest wykonywana przy żądaniu usunięcia, ale przed usunięciem jakichkolwiek zasobów.
- post-delete wykonuje się przy żądaniu usunięcia, ale po usunięciu wszystkich zasobów wersji.
- pre-upgrade wykonuje się na żądanie aktualizacji po renderowaniu szablonów, ale przed załadowaniem jakichkolwiek zasobów (np. przed polecienniem kubectl apply).
- post-upgrade wykonuje się przy żądaniu aktualizacji, ale po aktualizacji wszystkich zasobów.
- pre-rollback jest wykonywana przy żądaniu wycofania aktualizacji, po renderowaniu szablonów, ale przed cofnięciem jakichkolwiek zasobów.
- post-rollback jest wykonywana przy żądaniu wycofania aktualizacji, ale po zmodyfikowaniu wszystkich zasobów.

Kolejność wykonywania funkcji hook

Funkcje hook mają również możliwość wykonywania w określonej kolejności przy użyciu właściwości helm.sh/hook-weight. Funkcje te będą uruchamiane w kolejności od najniższej do najwyższej, więc Job z wartością hook-weight równą 0 będzie działać przed funkcją hook z wartością 1:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ .Values.appName }}-stage-0
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook-delete-policy": hook-succeeded
    "helm.sh/hook-weight": "0"
```

Wszystko, co musisz wiedzieć o funkcjach hook, znajdziesz w dokumentacji wykresu Helm (https://docs.helm.sh/developing_charts/#hooks).

Podsumowanie

Tworzenie aplikacji Kubernetes może być żmudne, jeśli trzeba zbudować i wdrożyć obraz kontenera w celu przetestowania każdej małej zmiany kodu. Narzędzia, takie jak Draft, Skaffold i Telepresence, sprawiają, że proces ten jest znacznie szybszy i przyspiesza cały rozwój.

Szczególnie wprowadzanie zmian w produkcji jest znacznie łatwiejsze w Kubernetes niż w tradycyjnych serwerach, pod warunkiem że rozumiesz podstawowe pojęcia oraz wiesz, jak możesz je dostosować do swojej aplikacji.

- Domyślna strategia wdrażania Rolling Update w Kubernetes aktualizuje kilka Podów jednocześnie, czekając przed zamknięciem starego, aż każdy nowy Pod będzie gotowy.
- Aktualizacje Rolling Updates zapobiegają przestojom kosztem wydłużenia procesu wdrażania. Oznacza to również, że zarówno stare, jak i nowe wersje aplikacji będą działały jednocześnie podczas wdrażania.
- Możesz ustawić odpowiednie wartości dla `maxSurge` i `maxUnavailable`, aby dostosować bieżące aktualizacje. W zależności od wersji interfejsu API Kubernetes, którego używasz, wartości domyślne mogą, ale nie muszą być odpowiednie dla Twojej sytuacji.
- Strategia Recreate zamyka jednocześnie wszystkie stare Pody i uruchamia nowe. Jest to szybka opcja, ale powoduje przestoje, więc nie nadaje się do aplikacji zorientowanych na użytkownika.
- W przypadku wdrożenia niebiesko-zielonego wszystkie nowe Pody są uruchamiane, ale bez odbierania ruchu od użytkowników. Następnie cały ruch jest przełączany na nowe Pody za jednym razem, przed wycofaniem starych Podów.
- Wdrożenia rainbow są podobne do wdrożeń niebiesko-zielonych, ale jednocześnie obsługują więcej niż dwie wersje.
- W Kubernetes możesz skorzystać z wdrożenia niebiesko-zielonego oraz rainbow, dostosowując etykiety na swoich Podach i zmieniając selektor w obiekcie Serwis, aby skierować ruch do odpowiedniego zestawu Podów.
- Funkcje hook i wykres Helm zapewniają sposób na zastosowanie niektórych zasobów Kubernetes (zwykle Job) na określonym etapie wdrożenia, np. w celu uruchomienia migracji bazy danych. Funkcje hook mogą określać kolejność, w jakiej zasoby powinny być stosowane podczas wdrażania, i powodować zatrzymanie wdrażania, jeśli coś się nie powiedzie.

Ciągłe wdrażanie w Kubernetes

Tao niczego nie robi i nie zostawia niczego niezrobionego.

— Lao Tzu

W tym rozdziale przyjrzymy się kluczowej zasadzie DevOps, czyli *ciągłemu wdrażaniu* (ang. *continuous deployment*), i zobaczymy, jak możemy to osiągnąć w środowisku cloud native opartym na Kubernetes. Przedstawimy niektóre opcje konfigurowania potoków ciągłego wdrażania (CD) do pracy z Kubernetes i pokażemy w pełni działający przykład z wykorzystaniem Google Cloud Build.

Co to jest ciągłe wdrażanie?

Ciągłe wdrażanie (CD) to automatyczne wdrażanie udanych komplikacji do produkcji. Podobnie jak w test suite, wdrażanie powinno być również zarządzane centralnie i zautomatyzowane. Programiści powinni mieć możliwość wdrażania nowych wersji po naciśnięciu jednego przycisku.

CD jest często kojarzone z *ciągłą integracją* (CI), czyli automatyczną integracją i testowaniem zmian programistów w stosunku do gałęzi głównej. Chodzi o to, że jeśli wprowadzasz zmiany w gałęzi, które po połączeniu z linią główną złamałyby komplikację, ciągła integracja poinformuje o tym od razu, zamiast czekać na zakończenie gałęzi i ostateczne scalenie. Połączenie ciągłej integracji i wdrażania jest często nazywane *CI/CD*.

Mechanizm ciągłego wdrażania często określa się mianem *potoku*; jest to seria zautomatyzowanych działań, które przenoszą kod ze stacji roboczej programisty na produkcję, poprzez sekwencję testów i etapów akceptacji.

Typowy potok dla aplikacji kontenerowych może wyglądać następująco.

1. Programista umieszcza zmiany swojego kodu w Git.
2. System komplikacji automatycznie buduje bieżącą wersję kodu i uruchamia testy.
3. Jeśli wszystkie testy zakończą się pomyślnie, obraz kontenera zostanie opublikowany w centralnym rejestrze kontenerów.
4. Nowo zbudowany kontener jest wdrażany automatycznie w środowisku testowym.
5. Środowisko testowe przechodzi pewne automatyczne testy akceptacyjne.
6. Obraz zweryfikowanego kontenera jest wdrażany do produkcji.

Kluczową kwestią jest to, że artefakt, który jest testowany i wdrażany w różnych środowiskach, nie jest kodem źródłowym, ale kontenerem. Istnieje wiele sposobów wkradania się błędów między kodem źródłowym a działającym plikiem binarnym. Testowanie kontenera zamiast kodu może pomóc w wychwyceniu wielu z nich.

Wielką zaletą CD jest *brak wystąpienia niespodzianek w produkcji*; nic nie zostanie wdrożone, jeśli dokładny obraz binarny nie został pomyślnie przetestowany.

Szczegółowy przykład takiego potoku CD można zobaczyć w tym rozdziale, w podrozdziale „Potok CD z wykorzystaniem Google Cloud Build”.

Z którego narzędzia CD powinienem skorzystać?

Jak zwykle problemem nie jest brak dostępnych narzędzi, ale sam wybór odpowiedniej opcji. Istnieje kilka narzędzi CD zaprojektowanych specjalnie dla aplikacji cloud native, a tradycyjne narzędzia do budowania, takie jak Jenkins, mają teraz rozszerzenia umożliwiające pracę z Kubernetes i kontenerami.

W rezultacie, jeśli już korzystasz z CD, prawdopodobnie nie musisz przełączać się na zupełnie nowy system. Jeśli wykonujesz migrację istniejących aplikacji do Kubernetes, prawie na pewno możesz to zrobić za pomocą kilku zmian w istniejącym systemie komplikacji.

Jeśli nie korzystasz jeszcze z systemu CD, w tym rozdziale przedstawimy niektóre z opcji.

Jenkins

Jenkins (<https://jenkins.io/>) jest bardzo popularnym narzędziem CD, istniejącym od lat. Posiada rozszerzenia do prawie wszystkiego, czego możesz potrzebować do pracy z CD, w tym obsługę Docker, kubectl i Helm.

Istnieje również nowszy dedykowany projekt służący do uruchamiania narzędzia Jenkins w klasztorze Kubernetes, JenkinsX (<https://jenkins-x.io/>).

Drone

Drone (<https://github.com/drone/drone>) to nowsze narzędzie CD zbudowane z kontenerów i dla nich przeznaczone. Jest proste i lekkie, z potokiem zdefiniowanym przez pojedynczy plik YAML. Ponieważ każdy krok komplikacji polega na uruchomieniu kontenera, oznacza to, że wszystko, co możesz uruchomić w kontenerze, możesz uruchomić w Drone.

Google Cloud Build

Jeśli korzystasz z infrastruktury na Google Cloud Platform, to Twoim pierwszym wyborem powinno być narzędzie CD Google Cloud Build (<https://cloud.google.com/cloud-build>). Podobnie jak Drone, Cloud Build uruchamia kontenery jako różne etapy komplikacji, a konfiguracja YAML znajduje się w repozytorium kodu.

Możesz skonfigurować Cloud Build, aby obserwowało Twoje repozytorium Git (integracja z GitHub jest dostępna). Kiedy zostaną ustawione wstępne warunki, takie jak wysyłanie do określonej gałęzi lub tagu, Cloud Build uruchomi określony potok, zbuduje nowy kontener, uruchomii test suite, opublikuje obraz i być może wdroży nową wersję w Kubernetes.

Kompletny działający przykład potoku CD w Cloud Build zobaczysz w tym rozdziale, w podrozdziale „Potok CD z wykorzystaniem Google Cloud Build”.

Concourse

Concourse (<https://concourse-ci.org/>) to narzędzie CD typu open source napisane w Go. Przyjmuje także deklaratywne podejście potokowe, podobnie jak narzędzia Drone oraz Cloud Build, przy użyciu pliku YAML do definiowania i wykonywania kroków komplikacji. Concourse ma już oficjalny stabilny wykres Helm. Można je wdrożyć w Kubernetes, dzięki czemu szybko uruchomisz potok kontenerowy.

Spinaker

Spinaker jest bardzo potężnym i elastycznym narzędziem — jednak na pierwszy rzut oka może być nieco zniechęcające. Opracowane przez Netflix, przoduje w dużych i złożonych wdrożeniach, takich jak wdrożenia niebiesko-zielone (patrz „Wdrożenia niebiesko-zielone” w rozdziale 13.). Istnieje również darmowy ebook (<https://www.spinnaker.io/ebook>) poświęcony narzędziu Spinnaker, który powinien dać Ci pojęcie, czy Spinnaker pasuje do Twoich potrzeb.

GitLab CI

GitLab jest popularną alternatywą dla GitHub służącą do hostowania repozytoriów Git. Jest również wyposażone w potężne, wbudowane narzędzie CD, o nazwie GitLab CI (<https://about.gitlab.com/features/gitlab-ci-ci>), które można wykorzystać do testowania i wdrażania kodu.

Jeśli już korzystasz z GitLab, warto spojrzeć na GitLab CI w celu implementacji potoku ciągłego wdrażania.

Codefresh

Codefresh (<https://codefresh.io/>) to zarządzana usługa CD do testowania i wdrażania aplikacji na Kubernetes. Jedną z interesujących funkcji jest możliwość wdrażania tymczasowych środowisk testowych dla każdej zmiany w gałęzi.

Z pomocą kontenerów Codefresh możesz budować, testować i wdrażać środowiska na żądanie, a następnie możesz skonfigurować sposób wdrażania kontenerów w różnych środowiskach w klastrach.

Azure Pipelines

Usługa Azure DevOps firmy Microsoft (wcześniej znana jako Visual Studio Team Services) zawiera funkcję ciągłego dostarczania potoków, zwaną Azure Pipelines (<https://azure.microsoft.com/services/devops/pipelines>), podobnie jak Google Cloud Build.

Komponenty CD

Jeśli masz już działający system CD i po prostu musisz dodać komponent do budowania lub wdrażania kontenerów, przedstawiamy kilka opcji, które możesz zintegrować z istniejącym systemem.

Docker Hub

Jednym z najprostszych sposobów automatycznego budowania nowych kontenerów na podstawie zmian w kodzie jest wykorzystanie Docker Hub (<https://docs.docker.com/docker-hub/builds/>). Jeśli masz konto w Docker Hub (patrz „Rejestry kontenerowe” w rozdziale 2.), możesz utworzyć trigger dla repozytorium GitHub lub BitBucket, które automatycznie zbuduje i opublikuje nowe kontenery w Docker Hub.

Gitkube

Gitkube (<https://gitkube.sh/>) to samoobsługowe narzędzie, które działa w klastrze Kubernetes. Obserwuje repozytorium Git oraz automatycznie buduje i montuje nowy kontener po uruchomieniu jednego z Twoich triggerów. Jest bardzo proste i przenośne oraz łatwe w konfiguracji.

Flux

Wzorzec uruchamiania potoku CD (lub innych zautomatyzowanych procesów) w gałęziach lub tagach Git jest czasem nazywany *GitOps* (<https://www.weave.works/blog/gitops-operations-by-pull-request>). Flux (<https://github.com/weaveworks/flux>) rozszerza ten pomysł, obserwując zmiany w rejestrze kontenera, a nie w repozytorium Git. Po utworzeniu nowego kontenera Flux automatycznie wdroży go w klastrze Kubernetes.

Keel

Keel (<https://keel.sh/>), podobnie jak Flux, zajmuje się wdrażaniem nowych obrazów kontenerów z rejestru. Można go skonfigurować do reagowania na haki internetowe (ang. *webhooks*), wysyłania i odbierania wiadomości Slack, czekania na zatwierdzenie przed wdrożeniem i wykonywania innych przydatnych przepływów pracy.

Potok CD z wykorzystaniem Google Cloud Build

Teraz, gdy znasz ogólne zasady CD oraz narzędzia, spójrzmy na kompletny przykład potoku CD.

Chodzi o to, że nie należy koniecznie używać dokładnie takich samych narzędzi i konfiguracji, jakie mamy tutaj; raczej mamy nadzieję, że zrozumiesz, jak wszystko do siebie pasuje. Będziesz też mógł dostosować niektóre części tego przykładu do własnego środowiska.

W przykładzie będziemy używać Google Cloud Platform (GCP), Google Kubernetes Engine (GKE) i Google Cloud Build, ale nie polegamy na żadnych konkretnych funkcjach tych produktów. Możesz replikować tego rodzaju potok za pomocą dowolnych narzędzi.

Jeśli chcesz przejść przez ten przykład przy użyciu własnego konta GCP, pamiętaj, że używa on pewnych zasobów podlegających opłacie. Nie będzie to kosztować fortuny, ale później będziesz musiał usunąć i wyczyścić zasoby chmury, aby upewnić się, że nie zostaniesz obciążony kosztami wyższymi niż to konieczne.

Konfigurowanie Google Cloud i GKE

Jeśli rejestrujesz się w Google Cloud po raz pierwszy, będziesz uprawniony do dość znacznego bezpłatnego kredytu, który powinien umożliwić Ci uruchamianie klastrów Kubernetes i innych zasobów przez dłuższy czas — bez ponoszenia opłat. Możesz dowiedzieć się więcej i założyć konto w witrynie Google Cloud Platform (<https://cloud.google.com/free>).

Po zarejestrowaniu się i zalogowaniu do własnego projektu Google Cloud postępuj zgodnie z instrukcjami (<https://cloud.google.com/kubernetes-engine/docs/how-to/creating-a-cluster>), aby utworzyć klaster GKE.

Następnie zainicjuj wykres Helm w klastrze (patrz „Helm: menadżer pakietów Kubernetes” w rozdziale 4.).

Teraz, aby skonfigurować potok, wykonaj następujące kroki.

1. Skopiuj (ang. *fork*) repozytorium demo na swoje osobiste konto GitHub.
2. Utwórz trigger Cloud Build służący do budowania i testowania dowolnej gałęzi Git.
3. Utwórz trigger wdrażania w GKE na podstawie tagów Git.

Kopiowanie repozytorium demo

Jeśli masz konto GitHub, użyj interfejsu GitHub, aby skopiować repozytorium demo (<https://github.com/cloudnativedevops/demo>).

Jeśli nie korzystasz z GitHub, zrób kopię naszego repozytorium i prześlij go na własny serwer Git.

Wprowadzenie do Cloud Build

W Cloud Build, podobnie jak w Drone i wielu innych nowoczesnych platformach CD, każdy krok w procesie budowania składa się z uruchomienia kontenera. Kroki komplikacji są zdefiniowane przy użyciu pliku YAML, który znajduje się w repozytorium Git.

Gdy potok jest uruchamiany po zatwierdzeniu do repozytorium (ang. *commit*), Cloud Build tworzy kopię repozytorium w tym konkretnym zatwierdzeniu i wykonuje kolejno każdy krok potoku.

W repozytorium demo znajduje się folder o nazwie *hello-cloudbuild*. W tym folderze znajdziesz plik *cloudbuild.yaml*, który definiuje nasz potok Cloud Build.

Spójrzmy kolejno na każdy krok komplikacji w tym pliku.

Budowanie kontenera testowego

Oto pierwszy krok:

```
- id: build-test-image
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/docker
  entrypoint: bash
  args:
    - -c
    - |
      docker image build --target build --tag demo:test .
```

Podobnie jak wszystkie kroki komplikacji w chmurze, składa się on z zestawu par klucz-wartość YAML.

- `id` nadaje przyjazną dla człowieka etykietę kroku komplikacji.
- `dir` określa podkatalog repozytorium Git, w którym ma działać.
- `name` identyfikuje kontener do uruchomienia dla tego kroku.
- `entrypoint` określa polecenie do uruchomienia w kontenerze, jeśli ma nie być domyślne.
- `args` podaje niezbędne argumenty do polecenia `entrypoint`.

I to wszystko!

Celem tego kroku jest zbudowanie kontenera, którego możemy użyć do uruchomienia testów naszej aplikacji. Używamy komplikacji wieloetapowej (patrz „Opis plików Dockerfile” w rozdziale 2.), jednak teraz chcemy zbudować tylko pierwszy etap. Więc uruchamiamy polecenie:

```
docker image build --target build --tag demo:test .
```

Argument `--target build` to informacja dla Docker, aby zbudował tylko część pliku Dockerfile pod sekcją `FROM golang:1.14-alpine AS build` i zatrzymał się przed przejściem do następnego kroku.

Oznacza to, że w powstałym kontenerze będzie zainstalowany Go, wraz z dowolnymi pakietami lub plikami użytymi w kroku oznaczonym ... `AS build`. Będzie to zasadniczo kontener jednorazowy, używany tylko do uruchomienia zestawu testowego naszej aplikacji, a następnie zostanie odrzucony.

Uruchamianie testów

Oto następny krok:

```
- id: run-tests
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/docker
  entrypoint: bash
  args:
    - -c
    - |
      docker container run demo:test go test
```

Ponieważ oznaczyliśmy nasz kontener jako `demo:test`, ten tymczasowy obraz będzie nadal dostępny dla reszty tej komplikacji w Cloud Build, a ten krok uruchomi test (`go test`) w kontenerze. Jeśli jakieś testy zakończą się niepowodzeniem, komplikacja zakończy działanie i zgłosi awarię.

W przeciwnym razie przejdzie do następnego kroku.

Budowanie kontenera aplikacji

Ponownie uruchamiamy polecenie docker build, ale bez flagi --target, dzięki czemu uruchamiamy komplikację wieloetapową. Otrzymamy ostateczny kontener aplikacji:

```
- id: build-app
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/docker
  entrypoint: bash
  args:
    - -c
    - |
      docker build --tag gcr.io/${PROJECT_ID}/demo:${COMMIT_SHA} .
```

Walidacja manifestów Kubernetes

W tym momencie mamy kontener, który przeszedł testy i jest gotowy do uruchomienia w Kubernetes. Aby go wdrożyć, korzystamy z wykresu Helm. W tym kroku wykonamy polecenie helm template w celu wygenerowania manifestów Kubernetes, a następnie skorzystamy z narzędzia kubeval, aby je sprawdzić (patrz „kubeval” w rozdziale 12.) :

```
- id: kubeval
  dir: hello-cloudbuild
  name: cloudnatively/helm-cloudbuilder
  entrypoint: bash
  args:
    - -c
    - |
      helm template ./k8s/demo/ | kubeval
```



Pamiętaj, że używamy tutaj naszego własnego obrazu kontenera Helm (cloud native/ ↴helm-cloudbuilder). Dziwne jest to, że Helm jako narzędzie przeznaczone specjalnie do wdrażania kontenerów nie ma oficjalnego obrazu kontenera. Możesz skorzystać z naszego obrazu, ale w produkcji prawdopodobnie będziesz chciał zbudować własny.

Publikowanie obrazu

Przy założeniu, że potok zakończy się pomyślnie, Cloud Build automatycznie opublikuje wynikowy obraz kontenera w rejestrze. Aby określić, które obrazy chcesz opublikować, umieść je pod sekcją images w pliku Cloud Build:

```
images:
  - gcr.io/${PROJECT_ID}/demo:${COMMIT_SHA}
```

Tagi Git SHA

O co chodzi z tagiem COMMIT_SHA? W Git każde zatwierdzenie ma unikalny identyfikator zwany SHA (ang. *Secure Hash Algorithm* — algorytm skrótu). SHA to długi ciąg cyfr szesnastkowych, np. takich jak 5ba6bfd64a31eb4013ccaba27d95cddd15d50ba3.

Jeśli użyjesz SHA do tagowania obrazu, będzie on zawierał link do zatwierdzenia Git, które go wygenerowało; jest on również migawką dokładnego kodu znajdującego się w kontenerze. Zaletą tagowania artefaktów komplikacji za pomocą Git SHA jest to, że możesz budować i testować wiele gałęzi jednocześnie, nie powodując żadnych konfliktów.

Teraz, gdy zobaczyłeś, jak działa potok, zwróć uwagę na triggery komplikacji, które faktycznie wykonają potok w oparciu o nasze określone warunki.

Tworzenie pierwszego triggera komplikacji

Trigger Cloud Build określa repozytorium Git, które będzie obserwowane, warunek, od którego zależy jego aktywacja (np. publikowanie do określonej gałęzi lub tagu), oraz plik potoku do wykonania.

Utwórz teraz nowy trigger. Zaloguj się do projektu Google Cloud i przejdź pod adres <https://console.cloud.google.com/cloud-build/triggers?pli=1>.

Kliknij przycisk *Add Trigger*, aby utworzyć nowy trigger komplikacji, i wybierz GitHub jako repozytorium źródłowe.

Zostaniesz poproszony o udzielenie uprawnień dla Google Cloud na dostęp do Twojego repozytorium GitHub.

Wybierz TWOJA_NAZWA_UŻYTKOWNIKA_GITHUB/demo, a Google Cloud połączy się z Twoim repozytorium.

Następnie skonfiguruj trigger, tak jak pokazano na rysunku 14.1.

Możesz nazwać trigger, jak chcesz. W sekcji branch zachowaj wartość domyślną `*`; ta wartość pasuje do dowolnej gałęzi.

Zmień sekcję `Build configuration` z `Dockerfile` na `cloudbuild.yaml`.

Pole `Location` w `cloudbuild.yaml` informuje Cloud Build, gdzie znaleźć nasz plik potoku zawierający kroki komplikacji. W tym przypadku będzie to `hello-cloudbuild/cloudbuild.yaml`.

Po zakończeniu kliknij przycisk *Create trigger*. Teraz możesz przetestować trigger i zobaczyć, co się stanie!

Testowanie triggera

Wprowadź zmiany w kopii repozytorium demo. Przykładowo utwórz nową gałąź i zmień powitanie z Hello naHola:

```
cd hello-cloudbuild  
git checkout -b hola  
Switched to a new branch hola
```

Trigger settings

Source: GitHub Repository: <https://github.com/domingusj/demo> 

Name (Optional)

build

Trigger type 

Branch

Tag

Branch (regex) 

Matches 2 branches: master, john

.*

Included files filter (glob) (Optional)

Changes affecting at least one included file will trigger builds

glob pattern example: src/*

Ignored files filter (glob) (Optional)

Changes only affecting ignored files won't trigger builds

glob pattern example: .gitignore

 Hide included and ignored files filters

Build configuration

Dockerfile

Specify the path within the Git repo

cloudbuild.yaml

Specify the path to a Cloud Build configuration file in the Git repo [Learn more](#)

cloudbuild.yaml location 

/ hello-cloudbuild/cloudbuild.yaml

Rysunek 14.1. Tworzenie triggera

Edytuj zarówno `main.go`, jak i `main_test.go`, zamień Hello naHola lub dowolne inne powitanie i zapisz oba pliki.

Przeprowadź testy samodzielnie, aby upewnić się, że wszystko działa:

```
go test
PASS
ok  github.com/cloudnative/devops/demo/hello-cloudbuild 0.011s
```

Teraz zatwierdź zmiany i wyślij je do rozdzielonego repozytorium. Jeśli wszystko jest w porządku, powinno to wyzwolić trigger Cloud Build do rozpoczęcia nowej komplikacji. Przejdz pod adres <https://console.cloud.google.com/cloud-build/builds>.

Zobaczysz listę ostatnich komplikacji w swoim projekcie. Najwyżej na liście umieszczona jest aktualnie wprowadzona zmiana. Być może nadal działa lub już się zakończyła.

Mamy nadzieję, że zobaczysz zielony znacznik wskazujący, że wszystkie kroki zostały zakończone. Jeśli nie, sprawdź dane w dzienniku komplikacji i zobacz, co się nie udało.

Przy założeniu, że wszystko się udało, kontener powinien zostać opublikowany w Twoim prywatnym rejestrze kontenerów Google (Google Container Registry) i otagowany razem ze skrótem SHA dla Twojej zmiany.

Wdrożenie z potoku CD

Teraz możesz wyzwolić proces komplikacji w momencie publikacji w Git, uruchomić testy i opublikować końcowy kontener w rejestrze. Na tym etapie jesteś gotowy do wdrożenia tego kontenera w Kubernetes.

W tym przykładzie wyobrażamy sobie, że istnieją dwa środowiska, jedno produkcyjne i jedno testowe. Wdrożymy je w oddzielnach przestrzeniach nazw — `staging-demo` i `production-demo`.

Skonfigurujemy Cloud Build, aby wdrażał do środowiska testowego, gdy zobaczy tag Git zawierający `staging`, oraz do produkcji, gdy zobaczy tag `production-demo`. Wymaga to nowego potoku, w osobnym pliku YAML, `cloudbuild-deploy.yaml`. Oto niezbędne kroki.

Uzyskiwanie poświadczeń dla klastra Kubernetes

Aby wdrożyć do Kubernetes za pomocą wykresu Helm, musimy skonfigurować `kubectl`, aby skomunikować się z naszym klastrem:

```
- id: get-kube-config
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/kubectl
  env:
    - CLOUDSDK_CORE_PROJECT=${_CLOUDSDK_CORE_PROJECT}
    - CLOUDSDK_COMPUTE_ZONE=${_CLOUDSDK_COMPUTE_ZONE}
    - CLOUDSDK_CONTAINER_CLUSTER=${_CLOUDSDK_CONTAINER_CLUSTER}
    - KUBECONFIG=/workspace/.kube/config
  args:
    - cluster-info
```

W tym kroku odwołujemy się do niektórych zmiennych, takich jak `$_CLOUDSDK_CORE_PROJECT`. Możemy zdefiniować te zmienne w triggerze komplikacji, tak jak w tym przykładzie, lub w samym pliku potoku, pod nagłówkiem `substitutions`:

```
substitutions:
  _CLOUDSDK_CORE_PROJECT=demo_project
```

Zmienne zdefiniowane przez użytkownika muszą zaczynać się znakiem podkreślenia (`_`) i używać wyłącznie wielkich liter i cyfr. Cloud Build daje nam również pewne predefiniowane zmienne, takie jak `$PROJECT_ID` i `$COMMIT_SHA` (pełna lista znajduje się tutaj: <https://cloud.google.com/cloud-build/docs/configuring-builds/substitute-variable-values>).

Musisz także autoryzować konto usługi Cloud Build, aby mieć uprawnienia do wprowadzania zmian w klastrze Kubernetes Engine. W sekcji IAM w GCP przyznaj do konta usługi Cloud Build rolę *IAM Kubernetes Engine Developer* — dla Twojego projektu.

Dodanie tagu środowiska

W tym kroku oznaczymy kontener tym samym tagiem Git, który uruchomił wdrożenie:

```
- id: update-deploy-tag
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/gcloud
  args:
    - container
    - images
    - add-tag
    - gcr.io/${PROJECT_ID}/demo:${COMMIT_SHA}
    - gcr.io/${PROJECT_ID}/demo:${TAG_NAME}
```

Wdrażanie w klastrze

Tutaj uruchamiamy Helm, aby faktycznie zaktualizować aplikację w klastrze, używając wcześniej uzyskanych poświadczeń Kubernetes:

```
- id: deploy
  dir: hello-cloudbuild
  name: cloudnatives/helm-cloudbuilder
  env:
    - KUBECONFIG=/workspace/.kube/config
  args:
    - helm
    - upgrade
    - --install
    - ${TAG_NAME}-demo
    - --namespace=${TAG_NAME}-demo
    - --values
    - k8s/demo/${TAG_NAME}-values.yaml
    - --set
    - container.image=gcr.io/${PROJECT_ID}/demo
    - --set
    - container.tag=${COMMIT_SHA}
    - ./k8s/demo
```

Do polecenia `helm upgrade` przekazujemy kilka dodatkowych flag.

namespace

Przestrzeń nazw, w której aplikacja powinna zostać wdrożona.

values

Plik wartości Helm do użycia w tym środowisku.

set container.image

Ustawia nazwę kontenera do wdrożenia.

set container.tag

Wdraża obraz z tym konkretnym tagiem (pochodzącym z Git SHA).

Tworzenie triggera wdrażania

Dodajmy teraz triggery służące do wdrażania w środowiskach `staging` oraz `production`.

Utwórz nowy trigger w Cloud Build, tak jak to zrobiłeś w punkcie „Tworzenie pierwszego triggera komplikacji” wcześniej w tym rozdziale. Jednak tym razem skonfiguruj go tak, aby wyzwalał komplikację w momencie nadania tagu.

Ponadto zamiast korzystać z pliku `hello-cloudbuild/cloudbuild.yaml`, do tej komplikacji użyjemy `hello-cloudbuild/cloudbuild-deploy.yaml`.

W sekcji `Substitution variables` ustawimy wartości specyficzne dla środowiska testowego.

- `_CLOUDSDK_CORE_PROJECT` musi mieć wartość identyfikatora projektu Google Cloud, w którym działa klanster GKE.
- `_CLOUDSDK_COMPUTE_ZONE` powinien obejmować strefę dostępności klastra (lub regionu, jeśli jest to klanster regionalny).
- `_CLOUDSDK_CONTAINER_CLUSTER` to nazwa Twojego klastra GKE.

Jak widać, możemy użyć powyższych zmiennych w tym samym pliku YAML zarówno do wdrażania w środowisku testowym, jak i produkcyjnym, nawet jeśli chcielibyśmy uruchomić te środowiska w oddzielnich klastrach lub nawet osobnych projektach GCP.

Po utworzeniu triggera dla tagu `staging` wypróbuj go, wysyłając zmiany z tagiem `staging` do repozytorium:

```
git tag -f staging
git push -f origin refs/tags/staging
Total 0 (delta 0), reused 0 (delta 0)
To github.com:domingusj/demo.git
 * [new tag] staging -> staging
```

Tak jak poprzednio, możesz obserwować postęp komplikacji (<https://console.cloud.google.com/projectselector/cloud-build/builds?supportedpurview=project>).

Jeśli wszystko pójdzie zgodnie z planem, Cloud Build powinien pomyślnie uwierzytelnić się w klastrze GKE i wdrożyć wersję testową aplikacji w przestrzeni nazw `staging-demo`.

Możesz to zweryfikować, sprawdzając pulpit GKE (<https://console.cloud.google.com/kubernetes/workload>) (lub za pomocą polecenia `helm status`).

Na koniec wykonaj te same kroki, aby utworzyć trigger, który wyzwoli proces wdrożenia do środowiska produkcyjnego po przesłaniu zmian z tagiem `production`.

Optymalizacja potoku komplikacji

Jeśli korzystasz z kontenerowych narzędzi do tworzenia potoków CD, np. Cloud Build, ważne jest, aby kontener używany w każdym kroku był jak najmniejszy (patrz „Minimalne obrazy kontenerów” w rozdziale 2.). Gdy używasz dziesiątek lub setek komplikacji dziennie, czas pracy z dużymi kontenerami naprawdę się wydłuża.

Jeśli np. korzystasz z Sops do odszyfrowywania obiektów Secret (patrz „Szyfrowanie obiektów Secret za pomocą Sops” w rozdziale 10.), oficjalny obraz kontenera `mozilla/sops` ma rozmiar około 800 MiB. Budując własny niestandardowy obraz za pomocą komplikacji wieloetapowej, możesz zmniejszyć rozmiar obrazu do około 20 MiB. Ponieważ ten obraz zostanie pobrany przy każdej komplikacji, warto go 40-krotnie zmniejszyć.

Dostępna jest wersja Sops ze zmodyfikowanym plikiem Dockerfile do budowy minimalnego obrazu kontenera (<https://github.com/bitfield/sops>).

Dostosowanie przykładowego potoku

Mamy nadzieję, że ten przykład pokazuje kluczowe koncepcje potoku CD. Jeśli korzystasz z Cloud Build, możesz użyć przykładowego kodu jako punktu wyjścia do skonfigurowania własnego potoku. Jeśli korzystasz z innych narzędzi, dostosowanie pokazanych tutaj kroków do pracy we własnym środowisku powinno być stosunkowo łatwe.

Podsumowanie

Konfigurowanie ciągłego procesu wdrażania aplikacji umożliwia konsekwentne, niezawodne i szybkie wdrażanie oprogramowania. Programiści powinni mieć możliwość przesyłania kodu do repozytorium, a wszystkie fazy komplikacji, testowania i wdrażania będą odbywać się automatycznie w centralizowanym potoku.

Ponieważ jest tak wiele opcji oprogramowania CD i technik, nie możemy dać Ci jednego przepisu, który zadziała dla wszystkich. Chcieliśmy pokazać, dlaczego potok CD może przynieść Ci korzyści, oraz dać kilka ważnych tematów do przemyśleń, jeżeli zdecydujesz się wdrożyć go we własnej firmie.

- Decyzja o wyborze narzędzi CD jest ważnym procesem przy budowie nowego potoku. Wszystkie narzędzia, o których wspominamy w tej książce, mogą być prawdopodobnie wykorzystane w prawie każdym istniejącym narzędziu CD.
- Jenkins, GitLab, Drone, Cloud Build i Spinnaker to tylko niektóre z popularnych narzędzi CD, które dobrze współpracują z Kubernetes. Istnieje również wiele nowszych narzędzi, takich jak Gitkube, Flux i Keel, które zostały utworzone specjalnie do automatyzacji wdrożeń w klastrach Kubernetes.
- Definiowanie kroków potoku komplikacji za pomocą kodu umożliwia śledzenie i modyfikowanie tych kroków wraz z kodem aplikacji.
- Kontenery wspomagają proces budowania komponentów za pośrednictwem środowiska testowego oraz ewentualnie produkcyjnego, bez konieczności przebudowywania nowego kontenera.
- Nasz przykładowy potok korzystający z Cloud Build powinien łatwo dostosowywać się do innych narzędzi i typów aplikacji. Ogólne kroki budowania, testowania i wdrażania są w dużej mierze takie same w każdym potoku CD, niezależnie od użytych narzędzi lub rodzaju oprogramowania.

Obserwowałość i monitorowanie

Na pokładzie statku nigdy nic nie jest do końca w porządku.

— William Langewiesche, *The Outlaw Sea*

W tym rozdziale rozważymy kwestię obserwowałości i monitorowania aplikacji cloud native. Co to jest obserwowałość? Jak to się ma do monitorowania? Jak monitorujesz Kubernetes, jak zapisujesz w nim logi?

Co to jest obserwowałość?

Obserwowałość może nie być znanim Ci terminem, choć staje się on coraz popularniejszy jako sposób na wyrażenie czegoś więcej niż tradycyjne monitorowanie. Najpierw zajmiemy się monitorowaniem, potem dowiesz się, w jaki sposób rozszerza go termin obserwowałość.

Co to jest monitorowanie?

Czy Twoja strona teraz działa? Idź, sprawdź; poczekamy. Najbardziej podstawowym sposobem na sprawdzenie, czy wszystkie aplikacje i usługi działają, tak jak powinny, jest ich samodzielne sprawdzenie. Kiedy jednak mówimy o monitorowaniu w kontekście DevOps, mamy na myśli głównie *monitorowanie automatyczne*.

Zautomatyzowane monitorowanie polega na sprawdzeniu dostępności lub zachowania strony internetowej czy usługi w jakiś programowy sposób, zwykle zgodnie z regularnym harmonogramem i zazwyczaj za pomocą zautomatyzowanego sposobu powiadamiania inżynierów o problemach.

Monitorowanie typu czarna skrzynka

Przeanalizujmy prosty przypadek statycznej strony internetowej; powiedzmy, że jest to blog związany z tą książką (<https://cloudnative.devopsblog.com/>).

Jeśli w ogóle nie działa, po prostu nie odpowiada lub zobaczysz w przeglądarce komunikat informujący o błędzie (mamy nadzieję, że nie, ale nikt nie jest doskonali). Zatem dla tej witryny najprostszym możliwym monitorowaniem jest pobranie strony głównej i sprawdzenie kodu stanu HTTP (200 oznacza

pomyślne żądanie). Możesz to zrobić za pomocą klienta HTTP, takiego jak `httipe` lub `curl`. Jeśli status wyjścia z klienta jest niezerowy, wystąpił problem z pobraniem strony internetowej.

Załóżmy jednak, że z konfiguracją serwera WWW coś poszło nie tak i chociaż serwer działa i odpowiada statusem HTTP 200 OK, w rzeczywistości wyświetla pustą stronę (lub jakąś stronę domyślną czy powitalną, a może zupełnie niewłaściwą stronę). Nasz prosty monitoring nie będzie zgłaszał problemu, ponieważ żądanie HTTP się powiedzie. Witryna jest jednak niedostępna dla użytkowników i nie mogą czytać naszych fascynujących oraz pouczających postów na blogu.

Bardziej zaawansowany monitoring może szukać określonego tekstu na stronie, takiego jak *Cloud Native DevOps*. To wychwyciłoby problem źle skonfigurowanego, ale działającego serwera WWW.

Strony dynamiczne

Latwo sobie wyobrazić, że bardziej złożone strony internetowe mogą wymagać bardziej złożonego monitorowania.

Jeśli np. witryna ma możliwość logowania się użytkowników, system monitoringu może również próbować zalogować się przy użyciu wstępnie utworzonego konta użytkownika i ostrzec, jeśli logowanie się nie powiedzie. A jeśli witryna ma wbudowaną wyszukiwarkę, sprawdzenie może polegać na wypełnieniu pola wyszukiwania, symulacji kliknięcia przycisku wyszukiwania i sprawdzeniu, czy wyniki zawierają oczekiwany tekst.

W przypadku prostych stron internetowych odpowiedź typu „tak lub nie” na pytanie „Czy działa?” może wystarczyć. W aplikacjach cloud native, które zwykle są bardziej złożonymi systemami rozproszonymi, pytanie może obejmować wiele zagadnień.

- Czy moja aplikacja jest dostępna na całym świecie? Czy tylko w niektórych regionach?
- Jak długo trwa ładowanie strony dla większości moich użytkowników?
- Co z użytkownikami, którzy mogą mieć wolne łącze?
- Czy wszystkie funkcje mojej witryny działają zgodnie z przeznaczeniem?
- Czy niektóre funkcje działają wolno, czy wcale, i ilu użytkowników to dotyczy?
- Jeśli strona opiera się na usługach stron trzecich, co dzieje się z moją aplikacją, gdy ta usługa zewnętrzna działa źle lub jest niedostępna?
- Co się stanie, gdy mój dostawca usług chmurowych ma awarię?

Zaczyna być oczywiste, że w świecie monitorowania systemów rozproszonych cloud native nic do końca nie jest jasne.

Ograniczenia monitorowania typu czarna skrzynka

Jednak bez względu na to, jak skomplikowane są te kontrole, wszystkie należą do tej samej kategorii monitorowania, czyli *monitorowania typu czarna skrzynka* (ang. *black-box monitoring*). Jak sama nazwa wskazuje, ten typ monitoringu obserwuje tylko zewnętrzne zachowanie systemu, bez żadnej próby zaobserwowania, co się z nim dzieje w środku.

Jeszcze kilka lat temu monitoring black-box, wykonywany przez popularne narzędzia, takie jak Nagios, Icinga, Zabbix, Sensu i Check_MK, był uważany za najnowocześniejszy. Z pewnością posiadanie wszelkiego rodzaju zautomatyzowanego monitorowania systemów to ogromna przewaga w porównaniu z jego brakiem. Istnieje jednak kilka ograniczeń takiego systemu.

- Może wykryć tylko przewidywalne awarie (np. strona internetowa nie odpowiada).
- Sprawdza tylko zachowanie części systemu, tych które są wystawione na zewnątrz.
- Jest pasywny i reaktywny; informuje o problemie dopiero po jego wystąpieniu.
- Może odpowiedzieć na pytanie: „Co jest zepsute?”, ale nie na ważniejsze pytanie: „Dlaczego?“.

Aby odpowiedzieć na pytanie *dłaczego*, musimy wyjść poza monitorowanie typu czarna skrzynka.

Jest jeszcze jeden problem związany z testem *działa/nie działa*. Co to oznacza ?

Co oznacza określenie „*działa*”?

W operacjach jesteśmy przyzwyczajeni do mierzenia odporności i dostępności naszych aplikacji w czasie pracy (ang. *uptime*), zwykle mierzonych procentowo. Przykładowo aplikacja o dostępności 99% była niedostępna przez nie więcej niż 1% odpowiedniego okresu. 99,9% czasu przestoju, zwanego *trzy dziewiątki*, przekłada się na około dziewięć godzin przestoju rocznie, co byłoby dobrym wskaźnikiem dla przeciętnej aplikacji internetowej. Cztery dziewiątki (99,99%) to mniej niż godzina przestoju rocznie, a pięć dziewiątek (99,999%) to około pięciu minut.

Możesz pomyśleć, że im więcej dziewiątek, tym lepiej. Jednak patrząc na to w ten sposób, pomijasz ważny punkt.

Dziewiątki nie mają znaczenia, jeśli użytkownicy nie są zadowoleni.

— Charity Majors (<https://red.ht/2FMZcMZ>)

Dziewiątki nie mają znaczenia, jeśli użytkownicy nie są zadowoleni

Jak mówi powiedzenie, to, co jest mierzone, zostaje zmaksymalizowane. Lepiej więc uważaj, co mierzysz. Jeśli Twoja usługa nie działa dla użytkowników, nie ma znaczenia, co mówią Twoje wewnętrzne dane, gdyż *usługa nie działa*. Istnieje wiele sposobów, za sprawą których usługa może sprawiać, że użytkownicy są niezadowoleni, nawet jeśli nominalnie *działa*.

Jednym z przypadków jest ładowanie witryny w czasie 10 sekund. Po tym czasie witryna może działać dobrze. Jeśli jednak jest zbyt wolna, równie dobrze może być całkowicie wyłączona. Użytkownicy pójdą do konkurencji.

Tradycyjne monitorowanie typu czarna skrzynka może podjąć próbę poradzenia sobie z tym problemem, definiując czas ładowania, powiedzmy, pięciu sekund w góre, a wszystko ponadto jest uważane za wyłączone i generowany jest alert. Co jednak zrobić, jeśli użytkownicy doświadczają różnego rodzaju czasów ładowania, od 2 sekund do 10 sekund? Przy takim progu usługa nie będzie działać dla niektórych użytkowników, ale dla innych będzie działać. Co zrobić, jeśli czasy ładowania są akceptowalne dla użytkowników z USA, a nie są dla użytkowników z Europy czy Azji ?

Aplikacje cloud native zawsze nie działają

Chociaż możesz udoskonalać bardziej złożone zasady i progi, aby umożliwić udzielanie odpowiedzi działa, czy nie działa na temat statusu usługi, prawda jest taka, że pytanie jest nieodwracalnie błędne. Systemy rozproszone, takie jak aplikacje cloud native, nigdy nie działają (<http://red.ht/2hMHwSL>); istnieją w stałym stanie częściowo zdegradowanej usługi.

Jest to przykład klasy problemów zwanych *gray failures* (<https://blog.acolyer.org/2017/06/15/gray-failure-the-achilles-heel-of-cloud-scale-systems/>). Takie awarie są z definicji trudne do wykrycia, szczególnie z jednego punktu widzenia lub z poziomu pojedynczej obserwacji.

Chociaż monitorowanie typu czarna skrzynka może być dobrym miejscem na rozpoczęcie przygody związanej z obserwonalnością, ważne jest, aby pamiętać, że nie powinieneś się na tym zagadnieniu zatrzymywać. Zobaczmy, czy możemy to zrobić lepiej.

Zapisywanie logów

Większość aplikacji tworzy pewnego rodzaju *dzienniki*. Dzienniki to seria rekordów, zwykle z pewnymi znacznikami czasu wskazującymi, kiedy rekordy zostały zapisane i w jakiej kolejności.

Przykładowo serwer WWW zapisuje w swoich dziennikach każde żądanie, w tym informacje, takie jak:

- żądany identyfikator URI,
- adres IP klienta,
- status HTTP odpowiedzi.

Jeśli aplikacja napotka błąd, zwykle rejestruje ten fakt wraz z pewnymi informacjami, które mogą, ale nie muszą być pomocne dla operatorów, aby dowiedzieli się, co spowodowało problem.

Często logi z szerokiej gamy aplikacji i usług są *agregowane* w centralnej bazie danych (np. Elasticsearch), gdzie można je przeszukiwać i wyświetlać na wykresach, co pomaga w rozwiązywaniu problemów. Narzędzia, takie jak Logstash i Kibana, lub usługi hostowane, takie jak Splunk i Loggly, zostały zaprojektowane w celu ułatwienia zbierania i analizowania dużych ilości danych dziennika.

Limity logowania

Dzienniki mogą być przydatne, ale mają też swoje ograniczenia. Decyzję o tym, co należy zalogować, a czego nie, trzeba podjąć w momencie pisania aplikacji przez programistę. Dlatego, podobnie jak w przypadku czarnej skrzynki, dzienniki mogą tylko odpowiadać na pytania lub wykrywać problemy, które można przewidzieć z góry.

Wyodrębnianie informacji z dzienników może być również trudne, ponieważ każda aplikacja zapisuje dzienniki w innym formacie, a operatorzy często muszą pisać niestandardowe analizatory składni dla każdego typu rekordu dziennika, aby przekształcić go w przydatne dane liczbowe lub dane zdarzeń.

Ponieważ dzienniki muszą rejestrować wystarczającą ilość informacji, aby zdiagnozować każdy możliwy problem, zazwyczaj mają słaby stosunek sygnału do szumu. Jeśli zarejestrujesz wszystko, przejrzenie setek stron dzienników w celu znalezienia jednego komunikatu o błędzie jest trudne i czasochłonne. Jeśli rejestrujesz tylko sporadyczne błędy, trudno określić, jak wygląda sytuacja *normalna*.

Dzienniki są trudne do skalowania

Dzienniki również nie skalują się dobrze, a zależy to od ruchu. Jeśli każde żądanie użytkownika generuje wiersz dziennika, który musi zostać wysłany do agregatora, musisz użyć dużej przepustowości sieci (która jest w związku z tym niedostępna dla użytkowników), a agregator dziennika może stać się wąskim gardłem.

Wielu dostawców hostowanych dzienników pobiera opłaty za liczbę wygenerowanych dzienników, co jest zrozumiałe, ale niefortunne, gdyż zachęca do rejestrowania mniejszej liczby informacji oraz posiadania mniejszej liczby użytkowników i mniejszego ruchu!

To samo dotyczy rozwiązań do rejestrowania na własnym serwerze; im więcej danych przechowujesz, tym więcej sprzętu, pamięci i zasobów sieciowych musisz opłacić oraz więcej czasu inżynierskiego zajmuje jedynie utrzymanie działania agregacji dzienników.

Czy logowanie jest przydatne w Kubernetes?

Napisaliśmy trochę o tym, jak kontenery generują dzienniki i jak można je sprawdzać bezpośrednio w Kubernetes, w punkcie „Przeglądanie dzienników kontenera” w rozdziale 7. Jest to przydatna technika debugowania dla poszczególnych kontenerów.

Jeśli korzystasz z logowania, to zamiast zapisywać rekordy w postaci zwykłego tekstu, powinieneś użyć pewnej formy danych strukturalnych, takich jak JSON, które można automatycznie przeanalizować (patrz „Potok obserwacyjny” w tym rozdziale).

Podczas gdy scentralizowane agregowanie logów (dla usług, takich jak ELK) może być przydatne w aplikacjach Kubernetes, to nie wyczerpuje wszystkich możliwości. Chociaż istnieją pewne biznesowe przypadki użycia dla scentralizowanego rejestrowania (np. wymagania dotyczące audytu i bezpieczeństwa lub analizy klienta), dzienniki nie zapewniają wszystkich informacji potrzebnych dla prawdziwej obserwacyjności.

W tym celu musimy spojrzeć na coś znacznie bardziej użytecznego.

Przedstawiamy metryki

Bardziej wyrafinowanym sposobem gromadzenia informacji o Twoich usługach jest użycie *metryk*. Jak sama nazwa wskazuje, metryka jest liczbową miarą czegoś. W zależności od aplikacji odpowiednie wskaźniki mogą obejmować:

- liczbę aktualnie przetwarzanych żądań,
- liczbę obsługiwanych żądań na minutę (lub na sekundę czy na godzinę),
- liczbę napotkanych błędów podczas obsługi żądań,
- średni czas potrzebny na obsługę żądań (lub wartości maksymalne albo 99 percentylów).

Przydatne jest również zebranie danych na temat infrastruktury i aplikacji, czyli:

- wykorzystanie procesora przez poszczególne procesy lub kontenery,
- aktywność dyskowa operacji we/wy węzłów i serwerów,
- przychodzący i wychodzący ruch sieciowy maszyn, klastrów lub modułów równoważenia obciążenia.

Metryki pomagają odpowiedzieć na pytanie dlaczego?

Metryki otwierają nowy wymiar monitorowania poza zwykłym *działaniem/niedziała niem*. Podobnie jak prędkościomierz w samochodzie lub skala temperatury na termometrze, dostarczają liczbowe informacje o tym, co się dzieje. W przeciwieństwie do dzienników, metryki można łatwo przetwarzać na wiele użytecznych sposobów: rysować wykresy, pobierać statystyki lub powiadamiać o przekroczeniu zdefiniowanych progów. Przykładowo system monitorowania może ostrzec, jeśli poziom błędu dla aplikacji przekroczy 10% w danym okresie.

Dane mogą również pomóc w odpowiedzi na pytanie *dlaczego*. Założmy, że użytkownicy doświadczają długich czasów odpowiedzi (duże *opóźnienia*) z Twojej aplikacji. Sprawdzasz swoje metryki i widzisz, że skok metryki *opóźnienia* pokrywa się z podobnym skokiem metryki *wykorzystanie procesora* dla określonej maszyny lub komponentu. To natychmiast daje wskazówkę, od czego zacząć. Komponent może być zaklinowany lub wielokrotnie ponawiać niektóre nieudane operacje albo jego węzeł może mieć problemy sprzętowe.

Metryki pomagają przewidywać problemy

Ponadto metryki mogą pomóc w *przewidywaniu problemów*: gdy coś pójdzie nie tak, zwykle nie dzieje się to od razu. Zanim problem zostanieauważony przez Ciebie lub Twoich użytkowników, wzrost niektórych metryk może wskazywać, że problem wystąpi niebawem.

Przykładowo metryka informująca o wykorzystaniu dysku na serwerze może powoli rosnąć. Wraz z upływem czasu może ostatecznie osiągnąć moment, w którym na dysku zabraknie miejsca. W tym momencie dochodzi do awarii całego serwera. Jeśli metryka ostrzegła Cię przed wystąpieniem problemu, możesz całkowicie zapobiec awarii.

Niektóre systemy do analizowania metryk wykorzystują nawet techniki uczenia maszynowego, aby wykryć anomalie i uzasadnić przyczynę jej wystąpienia. Może to być pomocne, szczególnie w złożonych systemach rozproszonych. Jednak dla większości zastosowań zbieranie danych, tworzenie wykresów oraz alarmy generowane przez metryki są wystarczająco dobrym rozwiązaniem.

Metryki monitorują aplikacje od wewnętrz

W przypadku monitoringu typu czarna skrzynka operatorzy muszą domyślać się, jak dana aplikacja lub usługa działają, przewidywać, jakiego typu awarie mogą się zdarzyć, oraz jaki to będzie miało wpływ na zachowanie zewnętrzne. Z kolei metryki umożliwiają twórcom aplikacji eksportowanie kluczowych informacji o ukrytych aspektach systemu w oparciu o znajomość ich działania (i nie-działania).

Zrezygnuj z inżynierii wstępnej i rozpoczęj monitorowanie od wewnętrz.

— Kelsey Hightower, Monitorama 2016 (<https://vimeo.com/173610242>)

Narzędzia, takie jak Prometheus, statsd i Graphite, lub usługi hostowane, takie jak Datadog, New Relic i Dynatrace, są szeroko stosowane do zbierania i zarządzania danymi metryk.

W rozdziale 16. napiszemy znacznie więcej o metrykach, również o tym, na jakich rodzajach powinieneś się skupić i co z nimi zrobić. Na razie wracamy do obserwonalności i śledzenia.

Śledzenie

Inna przydatna technika w zestawie narzędzi do monitorowania to *śledzenie* (ang. *tracing*), które jest szczególnie ważne w systemach rozproszonych. Podczas gdy metryki i dzienniki informują o tym, co się dzieje z poszczególnymi komponentami systemu, śledzenie koncentruje się na jednym żądaniu użytkownika oraz całym cyklu jego życia.

Załóżmy, że próbujesz dowiedzieć się, dlaczego niektóre żądania użytkowników mają bardzo duże opóźnienia. Sprawdzasz metryki dla każdego z komponentów systemu: load balancer, ingress, serwer WWW, serwer aplikacji, baza danych, magistrala komunikatów itd. — i wszystko wygląda normalnie. Więc co się dzieje?

Kiedy prześledzisz pojedyncze (miejmy nadzieję, że jest reprezentatywne) żądanie od momentu otwarcia połączenia przez użytkownika do momentu jego zamknięcia, zobaczysz obraz tego, jak ogólne opóźnienie rozkłada się na każdym etapie życia w w systemie.

Może np. okazać się, że czas poświęcony na obsługę żądania na każdym etapie potoku jest normalny, z wyjątkiem czasu przetwarzania przez bazę danych, który jest 100 razy dłuższy niż normalnie. Chociaż baza danych działa poprawnie, a jej wskaźniki nie wykazują problemów, z jakiegoś powodu serwer aplikacji musi czekać bardzo długo na zakończenie obsługi żądania przez bazę.

Ewentualnie odkryjesz problem związany z nadmierną utratą pakietów na połączeniu pomiędzy serwerami aplikacji a serwerem bazy danych. Bez odpowiedniego *widoku żądania* wynikającego z rozproszonego śledzenia trudno znaleźć podobne problemy.

Niektóre popularne rozproszone narzędzia śledzenia oferowane są przez narzędzia Zipkin, Jaeger i LightStep. Inżynier Masroor Hasan napisał przydatny post na blogu (<https://medium.com/@masroor.hasan/tracing-infrastructure-with-jaeger-on-kubernetes-6800132a677>) opisujący, jak używać narzędzia Jaeger do rozproszonego śledzenia w Kubernetes.

Framework OpenTracing (<https://opentracing.io/> — część Cloud Native Computing Foundation) ma na celu zapewnienie standardowego zestawu interfejsów API i bibliotek do rozproszonego śledzenia.

Obserwowałość

Ponieważ pojęcie *monitorowania* oznacza odmienne rzeczy dla różnych osób, od zwykłego monitoringu typu czarna skrzynka po kombinację metryk, logowania i śledzenia, powszechnie staje się stosowanie *obserwowałości* jako terminu obejmującego wszystkie te techniki. Obserwowałość Twojego systemu jest miarą tego, jak dobrze jest zarządzany i jak łatwo możesz dowiedzieć się, co się w nim dzieje. Niektórzy twierdzą, że obserwowałość jest nadzbiorem monitorowania, inni zaś, że obserwowałość odzwierciedla zupełnie inny sposób myślenia niż tradycyjne monitorowanie.

Być może najbardziej użytecznym sposobem rozróżnienia tych terminów jest stwierdzenie, że monitorowanie informuje, czy *system działa*, a obserwowałość zachęca do odpowiedzi na pytanie, *dlaczego nie działa*.

Obserwowałość polega na zrozumieniu

Mówiąc bardziej ogólnie, obserwowałość dotyczy *zrozumienia*: zrozumienia tego, co robi Twój system oraz jak to robi. Jeśli np. wprowadzisz zmianę w kodzie, która ma poprawić wydajność określonej funkcji o 10%, obserwowałość może powiedzieć, czy zadziałała. Jeśli wydajność tylko nieznacznie wzrosła lub — gorzej — spadła nieznacznie, musisz ponownie sprawdzić kod.

Z drugiej strony, jeśli wydajność wzrośnie o 20%, zmiana zakończy się sukcesem i może powinieneś pomyśleć o tym, dlaczego Twoje prognozy nie były wystarczające. Obserwowałość pomaga zbudować i udoskonalić mentalny model interakcji różnych części systemu.

Obserwowałość dotyczy również *danych*. Musimy wiedzieć, jakie dane generować, co gromadzić, jak je agregować (w stosownych przypadkach), na jakich wynikach się skupić oraz jak je wyszukiwać i wyświetlać.

Oprogramowanie jest nieprzezroczyste

W tradycyjnym monitorowaniu mamy wiele danych na temat *maszyny*; są to obciążenia procesora, aktywność dysku, pakiety sieciowe itd. Trudno jednak wywnioskować z tego, co robi nasze oprogramowanie. Aby to zrobić, musimy je zbadać.

Oprogramowanie jest domyślnie nieprzezroczyste; musi generować dane, abyśmy mogli zorientować się, co robi. Obserwowe systemy pozwalają ludziom odpowiedzieć na pytanie: „Czy działa poprawnie?”. A jeśli odpowiedź brzmi: „Nie”, musisz określić zakres działania i zidentyfikować, co się dzieje.

— Christine Spang (<https://twitter.com/jetarrant/status/1025122034735435776>) (Nylas)

Budowanie kultury obserwowałości

Jeszcze bardziej ogólnie obserwowałość dotyczy *kultury*. To kluczowa zasada filozofii DevOps polegająca na zamknięciu pętli między tworzeniem kodu a uruchamianiem go na dużą skalę w środowisku produkcyjnym. Obserwowałość jest podstawowym narzędziem do zamykania tej pętli. Programiści i pracownicy operacyjni muszą ściśle współpracować, aby przygotować usługi na potrzeby obserwowałości, a następnie wymyśleć najlepszy sposób przetwarzania i działania na podstawie dostarczonych informacji.

Celem zespołu obserwowałości nie jest zbieranie dzienników, metryk ani śladów. Ma on na celu zbudowanie kultury inżynierii opartej na faktach i opiniach, a następnie rozpowszechnienie tej kultury w całej organizacji.

— Brian Knox (<https://twitter.com/taotetek/status/974989022115323904>) (DigitalOcean)

Potok obserwowałości

Jak działa obserwowałość z praktycznego punktu widzenia? Często zdarza się, że wiele źródeł danych (dzienniki, metryki itp.) zostało podłączonych ad hoc do różnych magazynów danych.

Przykładowo dzienniki mogą być przesyłane do serwera ELK, podczas gdy metryki przychodzą do trzech lub czterech różnych usług zarządzanych, a tradycyjny monitoring raportuje do innej usługi. Nie jest to idealne rozwiązanie.

Po pierwsze, trudno je skalować. Im więcej masz źródeł danych i sklepów, tym więcej połączeń i większy ruch przez te połączenia. Nie ma sensu poświęcać czasu na inżynierię, aby wszystkie te rodzaje połączeń były stabilne i niezawodne.

Ponadto im ściślej systemy zintegrowane są z konkretnymi rozwiązaniami lub dostawcami, tym bardziej je zmienić lub wypróbować alternatywne rozwiązania.

Coraz popularniejszym sposobem rozwiązań tego problemu jest zastosowanie *potoku obserwacyjności* (<https://dzone.com/articles/the-observability-pipeline>).

Za pomocą potoku obserwacyjności oddzielamy źródła danych od miejsc docelowych i udostępniamy bufor. To sprawia, że dane obserwacyjności są łatwo dostępne. Nie musimy już zastanawiać się, jakie dane wysłać z kontenerów, maszyn wirtualnych i infrastruktury, gdzie je wysłać i jak je wysłać. Wszystkie dane są raczej przesyłane do potoku, który obsługuje ich filtrowanie i dostarczanie we właściwe miejsca. Daje nam to również większą elastyczność w zakresie dodawania lub usuwania ujęć danych oraz zapewnia bufor między producentami danych a konsumentami.

— Tyler Treat

Potok obserwacyjności ma ogromne zalety. Dodanie nowego źródła danych to tylko kwestia podłączenia go do potoku. Nowa usługa wizualizacji lub ostrzegania o awariach staje się kolejnym odbiorcą potoku.

Ponieważ potok buforuje dane, nic nie zostaje utracone. Jeśli nastąpi nagły wzrost ruchu i przeciążenie danych metryk, potok buforuje je i niczego nie traci.

Korzystanie z potoku obserwacyjności wymaga standardowego formatu metryk (patrz „Prometheus” w rozdziale 16.) i najlepiej ustrukturyzowanego rejestrowania z aplikacji korzystających z JSON lub innego rozsądniego serializowanego formatu danych. Zamiast emitować zwykłe logi tekstowe i analizować je później przy użyciu delikatnych wyrażeń regularnych, zacznij od samego początku, korzystać ze strukturyzowanych danych.

Monitorowanie w Kubernetes

Teraz, gdy trochę lepiej rozumiesz, czym jest monitorowanie typu czarna skrzynka oraz jak ogólnie odnosi się do obserwacyjności, zobaczymy, jak to się ma do aplikacji Kubernetes.

Zewnętrzny monitoring typu czarna skrzynka

Jak widzieliśmy, za pomocą monitorowania black-box możemy tylko określić, czy Twoja aplikacja jest wyłączona. Jest to wciąż bardzo przydatna informacja. Pomimo że w aplikacjach cloud native może wystąpić dowolny błąd, niektóre żądania nadal mogą być obsługiwane w akceptowalny sposób.

Inżynierowie mogą pracować nad rozwiązywaniem problemów wewnętrznych, takich jak powolne zapytania i podwyższone poziomy błędów, w czasie gdy użytkownicy naprawdę nie są świadomi problemu.

Jednakże poważniejsza klasa problemów powoduje awarię na pełną skalę; aplikacja jest niedostępna lub nie działa dla większości użytkowników. Nie jest to dobre dla użytkowników, a w zależności od aplikacji może być również złe dla Twojej firmy. Aby wykryć awarię, monitorowanie musi korzystać z usługi w taki sam sposób jak użytkownik.

Monitorowanie naśladuje zachowanie użytkownika

Jeśli jest to np. usługa HTTP, system monitorowania musi wysyłać do niej żądania HTTP, a nie tylko nawiązywać połączenia TCP. Jeśli usługa zwraca tylko tekst statyczny, monitoring może sprawdzić, czy tekst pasuje do oczekiwanej ciągi. Zwykle jest to trochę bardziej skomplikowane, co opisaliśmy w punkcie „Monitorowaniu typu czarna skrzynka” w tym rozdziale.

Jednak w przypadku wykrywania awarii prawdopodobnie wystarczy zwykłe dopasowanie tekstu, aby stwierdzić, że aplikacja jest wyłączona. Jednak wykonanie monitoringu black-box wewnętrz Twojej infrastruktury (np. w Kubernetes) nie wystarczy. Błąd może wynikać z różnego rodzaju problemów oraz awarii między użytkownikiem a zewnętrznymi elementami infrastruktury; może dotyczyć m.in.:

- nieprawidłowych rekordów DNS,
- partycji sieciowych,
- utraty pakietów,
- źle skonfigurowanych routerów,
- brakujących lub złych reguł zapory ogniowej,
- awarii dostawcy chmury.

W przypadku wszystkich tych sytuacji wewnętrzne metryki oraz monitorowanie mogą nie wykazywać żadnych problemów. Dlatego głównym zadaniem obserwonalności powinno być monitorowanie dostępności Twoich usług z pewnego miejsca zewnętrznego w stosunku do Twojej infrastruktury. Istnieje wiele usług innych firm, które mogą wykonywać tego rodzaju monitorowanie (czasem nazywane *monitorowaniem jako usługą* lub MaaS), w tym Uptime Robot, Pingdom i Wormly.

Nie buduj własnej infrastruktury monitorowania

Większość tych usług jest na pewnym poziomie darmowa albo posiada nieskrypcje, więc cokolwiek za nie płacisz powinieneś uznać za niezbędny wydatek operacyjny. Nie zawracaj sobie głowy budowaniem własnej zewnętrznej infrastruktury monitorowania, nie jest tego warta.

Koszt rocznej subskrypcji Pro w usłudze Uptime Robot prawdopodobnie nie wynosi tyle, co godzina pracy Twojego inżyniera.

Sprawdź, czy zewnętrzny dostawca monitorowania obsługuje następujące krytyczne funkcje:

- sprawdzanie HTTP/HTTPS,
- sprawdzanie, czy Twój certyfikat TLS jest nieważny lub wygasł,

- dopasowanie słowa kluczowego (ostrzeżenie, gdy brakuje słowa kluczowego lub gdy jest obecne),
- automatyczne tworzenie lub aktualizowanie rodzaju sprawdzeń za pośrednictwem interfejsu API,
- alerty przez e-mail, SMS, webhook lub inny prosty mechanizm.

W tej ksiązce promujemy ideę infrastruktury jako kodu, więc powinna istnieć możliwość automatyzacji zewnętrznych kontroli monitorowania za pomocą kodu. Przykładowo Uptime Robot ma prosty interfejs API REST do tworzenia nowych rodzajów sprawdzeń i można go zautomatyzować za pomocą bibliotek lub polecenia, takiego jak `uptimerobot` (<https://github.com/bitfield/uptimerobot>).

Nie ma znaczenia, z której usługi monitorowania zewnętrznego korzystasz, o ile z niej korzystasz. Nie poprzestawaj na tym. W kolejnym punkcie piszemy, co możesz zrobić, aby monitorować kondycję aplikacji w samym klastrze Kubernetes.

Wewnętrzna kontrola aplikacji

Do awarii aplikacji cloud native często dochodzi w skomplikowany, nieprzewidywalny i trudny do wykrycia sposób. Aplikacje muszą być zaprojektowane tak, aby były odporne i zdolne do degradacji w obliczu nieoczekiwanych awarii. Jednak — jak na ironię — im bardziej są odporne, tym trudniej wykryć awarie za pomocą monitorowania typu czarna skrzynka.

Aby rozwiązać ten problem, aplikacje mogą i powinny przeprowadzać własne testy kondycji. Twórca określonej funkcji lub usługi najlepiej wie, co musi być sprawne. Może utworzyć kod, który sprawdza tę funkcję oraz udostępnia wyniki kontroli w taki sposób, aby można było monitorować ją na zewnątrz kontenera (np. przy użyciu HTTP).

Czy użytkownicy są zadowoleni?

Kubernetes zawiera prosty mechanizm informowania o aktywności lub gotowości aplikacji, co widzieliśmy w „Sondach żywotności” w rozdziale 5. Zazwyczaj sondy żywotności lub gotowości Kubernetes są dość proste; aplikacja zawsze odpowiada „OK” na wszelkie żądania. Jeśli nie odpowiada, Kubernetes uważa, że jest wyłączena lub nie jest gotowa.

Jednak — jak wielu programistów wie ze swojego doświadczenia — to, że program działa, niekoniecznie oznacza, że działa poprawnie. Bardziej zaawansowana sonda gotowości powinna zapytać: „Czego aplikacja potrzebuje do wykonania swojej pracy?”.

Jeśli np. musi komunikować się z bazą danych, może sprawdzić, czy ma prawidłowe i responsywne połączenie z bazą danych. Jeżeli zależy to od innych usług, może sprawdzić dostępność usług. (Ponieważ kontrole aktywności są przeprowadzane często, nie powinny robić nic zbyt kosztownego, co mogłoby wpływać na obsługę żądań od rzeczywistych użytkowników).

Pamiętaj, że nadal odbieramy binarną odpowiedź tak/nie od sondy gotowości. To tylko bardziej świadoma odpowiedź. Staramy się odpowiedzieć na pytanie: „Czy użytkownicy są zadowoleni?” tak dokładnie, jak to możliwe.

Usługi i wyłączniki

Jak wiesz, jeśli sprawdzenie *żywotności* kontenera nie powiedzie się, Kubernetes uruchomi go ponownie. Nie jest to tak bardzo pomocne w sytuacji, gdy w kontenerze nie dzieje się nic złego, a zawiezie tylko jeden składnik. Z drugiej strony, semantyka nieudanego sprawdzenia *gotowości* oznacza: „Wszystko w porządku, ale w tej chwili nie mogę obsłużyć żądań użytkowników”.

W tej sytuacji kontener zostanie usunięty z dowolnych zasobów Serwis, a Kubernetes przestanie wysyłać żądania, dopóki nie będzie ponownie gotowy. Jest to lepszy sposób radzenia sobie z taką awarią.

Załóżmy, że posiadasz 10 mikrousług, z których każda zależy od następnej krytycznej części jej pracy. Ostatnia usługa w łańcuchu kończy się niepowodzeniem. Przedostatnia usługa wykryje to i sonda gotowości zacznie zgłaszać błąd. Kubernetes rozłączy ją, a następna usługa w linii wykryje to itd. W końcu usługa frontendowa zawiedzie i (miejmy nadzieję) zostanie wyzwolony alert monitorowania black-box.

Gdy problem z usługą podstawową zostanie naprawiony lub pomoże automatyczny restart, wszystkie pozostałe usługi w łańcuchu automatycznie staną się znów gotowe bez ponownego uruchamiania lub utraty jakiegokolwiek stanu. Jest to przykład tzw. *wzorca wyłącznika* (ang. *circuit breaker pattern*) (<https://martinfowler.com/bliki/CircuitBreaker.html>). Gdy aplikacja wykryje awarię na niższym poziomie, wyłącza się z działania (poprzez sprawdzenie gotowości), aby zapobiec wysyłaniu do niej kolejnych żądań, dopóki problem nie zostanie rozwiązyany.

Wdzięczna degradacja

Chociaż wyłącznik jest przydatny do rozwiązywania problemów tak szybko, jak to możliwe, należy tak zaprojektować swoje usługi, aby uniknąć awarii całego systemu — w przypadku gdy jedna lub więcej usług składowych jest niedostępnych. Zamiast tego postaraj się, aby Twoje usługi uległy *wdzięcznej degradacji* (ang. *degrade gracefully*), co oznacza, że nawet jeśli nie mogą zrobić wszystkiego, nadal mogą coś zrobić.

W systemach rozproszonych musimy założyć, że usługi, komponenty i połączenia zawodzą, mniej więcej przez cały czas, w sposób tajemniczy i sporadyczny. Odporny system może sobie z tym poradzić, nie zawodząc całkowicie.

Podsumowanie

O monitorowaniu można powiedzieć wiele. Nie mieliśmy miejsca, by powiedzieć tyle, ile chcieliśmy, ale mamy nadzieję, że ten rozdział dostarczył kilku użytecznych informacji na temat tradycyjnych technik monitorowania; napisaliśmy w nim, co mogą zrobić i czego nie mogą zrobić, a także, jak rzeczy muszą się zmienić w środowisku cloud native.

Pojęcie obserwonalności wprowadza szersze spojrzenie na tradycyjne pliki logowania i kontrole typu czarna skrzynka. Metryki stanowią ważną część tego obszaru, a w następnym i ostatnim rozdziale bardziej szczegółowo się im przyjrzymy.

Zanim jednak przewrócisz stronę, możesz przypomnieć sobie następujące kluczowe punkty.

- Monitoring typu czarna skrzynka obserwuje zachowanie zewnętrzne systemu w celu wykrycia przewidywalnych awarii.
- Systemy rozproszone ujawniają ograniczenia tradycyjnego monitorowania, ponieważ nie dotyczy ich stan typu *działa/nie działa*: istnieją w stałym stanie częściowo zdegradowanej usługi. Innymi słowy, na pokładzie statku nic nie jest do końca w porządku.
- Logi mogą być przydatne do rozwiązywania problemów po incydencie, ale trudno je skalować.
- Metryki otwierają nowy wymiar poza zwykłą odpowiedzią *działa/nie działa* i dostarczają ciągłych wartości liczbowych, które dotyczą setek lub tysięcy aspektów Twojego systemu.
- Metryki mogą pomóc Ci odpowiedzieć na pytanie *dlaczego*, a także zidentyfikować problematyczne trendy, zanim doprowadzą do awarii.
- Śledzenie rejestruje zdarzenia z dokładnym pomiarem czasu w cyklu życia pojedynczego żądania, aby pomóc w debugowaniu problemów z wydajnością.
- Obserwowałość to połączenie tradycyjnego monitorowania, rejestrowania, pomiarów i śledzenia oraz wszystkich innych sposobów rozumienia systemu.
- Obserwowałość oznacza także przejście w kierunku kultury zespołowej inżynierii opartej na faktach i opiniach.
- Należy sprawdzać, czy Twoje usługi skierowane do użytkowników działają, korzystając z zewnętrznych kontroli black-box. Nie próbuj tworzyć własnych rozwiązań; skorzystaj z usługi monitorowania innej firmy, np. takiej jak Uptime Robot.
- Dziewiątki nie mają znaczenia, jeśli użytkownicy nie są zadowoleni.

Metryki w Kubernetes

Możliwe, iż będziesz posiadał tyle wiedzy na dany temat, że staniesz się ignorantem.

— Frank Herbert, *Chapterhouse: Dune*

W tym rozdziale zajmiemy się metrykami, o których zaczeliśmy pisać w rozdziale 15. Teraz zagłębimy się w szczegóły i powiemy, jakie są rodzaje metryk, które z nich są ważne dla usług cloud native, które metryki wybrać, jak analizować dane metryk, aby uzyskać przydatne informacje oraz jak prezentować metryki i generować alerty. Na koniec przedstawimy niektóre opcje narzędzi i platform.

Czym są metryki?

Ponieważ podejście do obserwonalności zorientowane na metryki jest stosunkowo nowe w świecie DevOps, poświęćmy chwilę, aby porozmawiać o tym, czym dokładnie są metryki i jak najlepiej z nich korzystać.

Jak pisaliśmy w punkcie „Przedstawiamy metryki” w rozdziale 15., metryki są liczbowymi miarami konkretnych rzeczy. Znanym przykładem ze świata tradycyjnych serwerów jest wykorzystanie pamięci przez konkretną maszynę. Jeśli procesy użytkownika zużywają tylko 10% fizycznej pamięci, maszyna ma wolne moce produkcyjne. Jeśli jednak 90% pamięci pracuje, urządzenie jest prawdopodobnie dość zajęte.

Jedną z cennych informacji, które mogą dać metryki, jest migawka tego, co dzieje się w danym momencie, ale możemy zrobić więcej. Zużycie pamięci rośnie i maleje przez cały czas, gdy obciążenia zaczynają się i kończą. Czasem jednak interesuje nas zmiana wykorzystania pamięci w czasie.

Seria danych w czasie

Jeśli regularnie próbujesz użycie pamięci, możesz utworzyć szereg czasowy tych danych.

Na rysunku 16.1 pokazujemy wykres serii danych w zależności od czasu, dotyczą one wykorzystania pamięci w węźle Google Kubernetes Engine w ciągu tygodnia. Daje to znacznie bardziej zrozumiałą obraz tego, co się dzieje, niż garść chwilowych wartości.

Memory

Aug 2, 2018 2:51 PM



- gke-k8s-cluster-1-n1-standard-2-pool-2ac3f6e1-dwtv:
1.303G

Rysunek 16.1. Wykres serii danych wykorzystania pamięci dla węzła GKE w czasie

Większość metryk, którymi jesteśmy zainteresowani do celów obserwacyjności, jest wyrażona jako szeregi czasowe. Wszystkie są również numeryczne. Przykładowo w przeciwieństwie do danych dziennika metryki są wartościami, na których można wykonywać obliczenia matematyczne i statystyczne.

Liczniki i mierniki

Podczas gdy niektóre wielkości mogą być reprezentowane przez liczby całkowite (np. liczba fizycznych procesorów w maszynie), większość wymaga części dziesiętnej. Dlatego, aby zaoszczędzić konieczności obsługi dwóch różnych prezentacji liczb, metryki są prawie zawsze przedstawiane jako dziesiętne wartości zmiennoprzecinkowe.

Istnieją dwa główne typy wartości metryk, takie jak *liczniki* i *mierniki*. Liczniki mogą tylko rosnąć (lub resetować się do zera); nadają się do mierzenia liczby obsługiwanych żądań i liczby otrzymanych błędów. Mierniki mogą się zmieniać w górę i w dół; są przydatne do pomiaru ciągle zmieniających się wartości, takich jak zużycie pamięci, lub do wyrażania proporcji innych wielkości.

Odpowiedzi na niektóre pytania to prostu *tak* lub *nie*: np. czy dany punkt końcowy odpowiada na połączenia HTTP. W takim przypadku odpowiednią metryką będzie miernik o ograniczonym zakresie wartości, być może 0 i 1.

Przykładowo miernik sprawdzający połączenia HTTP dla punktu końcowego może mieć nazwę `http.can_connect`, a jego wartość może wynosić 1, gdy punkt końcowy odpowiada, lub 0 w przeciwnym razie.

Co mogą powiedzieć metryki?

Jakie zastosowanie mają metryki? Cóż, jak widzieliśmy wcześniej w tym rozdziale, metryki dają znak, kiedy coś się zepsuje. Jeśli np. poziom błędów nagle wzrośnie (lub żądania do Twojej strony nagle wzrosną), może to oznaczać problem. Dla niektórych metryk opartych na takim progu możesz automatycznie generować alerty.

Metryki mogą również informować o aktualnym stanie pracy, np. o tym, ilu równocześnie użytkowników obsługuje obecnie Twoja aplikacja. Długoterminowe trendy tych wartości mogą być przydatne zarówno w podejmowaniu decyzji operacyjnych, jak i w analizach biznesowych.

Wybór dobrych metryk

Możesz pomyśleć: „Jeśli metryki są dobre, lepiej używać ich więcej!”, ale to nie działa w ten sposób. Nie możesz monitorować wszystkiego. Przykładowo Google Stackdriver udostępnia dosłownie setki danych dotyczących zasobów w chmurze. Oto przykłady.

`instance/network/sent_packets_count`

Liczba pakietów sieciowych wysłanych przez każdą jednostkę obliczeniową.

`storage/object_count`

Łączna liczba obiektów w każdym zasobie bucket danych.

`container/cpu/utilization`

Aktualny stopień użycia CPU przez kontener.

Lista jest dłuża (<https://cloud.google.com/monitoring/api/metrics>). Nawet gdybyś mógł wyświetlić wykresy wszystkich tych danych jednocześnie, co wymagałoby ekranu monitora wielkości domu, nigdy nie byłbyś w stanie zebrać wszystkich tych informacji i wydedukować z nich coś przydatnego. Musisz skupić się na podzbiorze danych, na których zależy Ci najbardziej.

Na czym więc powinieneś się skoncentrować podczas monitorowania własnej aplikacji? Tylko Ty możesz odpowiedzieć na to pytanie, ale mamy kilka sugestii, które mogą być pomocne. W pozostałej części tego porozdziału przedstawimy niektóre typowe wzorce metryk dotyczących obserwonalności, skierowane do różnych odbiorców i zaprojektowane tak, aby spełniały różne wymagania.

Warto powiedzieć, że jest to doskonała okazja do współpracy DevOps i powinieneś zacząć myśleć i mówić o tym, jakie mierniki będą potrzebne na początku rozwoju oprogramowania, a nie na końcu (patrz „Wspólna nauka” w rozdziale 1.).

Usługi: wzorzec RED

Większość osób korzystających z Kubernetes używa jakiejś usługi webowej; i tak użytkownicy wysyłają żądania, a aplikacja wysyła odpowiedzi. Użytkownikami mogą być programy lub inne usługi; w systemie rozproszonym opartym na mikrousługach każda usługa wysyła żądania do innych usług i wykorzystuje wyniki do dostarczenia informacji z powrotem do jeszcze większej liczby usług. Tak czy inaczej jest to system sterowany żądaniami.

Co warto wiedzieć o systemie opartym na żądaniami?

- Pierwszą, oczywistą rzeczą jest liczba otrzymywanych żądań.
- Kolejną jest liczba żądań, które zakończyły się niepowodzeniem na różne sposoby; tzn. liczba błędów.
- Trzecią przydatną metryką jest *czas trwania* każdego żądania. Daje on wyobrażenie o tym, jak dobrze Twoja usługa działa i jak bardzo nieszczęśliwi mogą być Twoi użytkownicy.

Wzorzec żądania-błędy-czas trwania (ang. *Requests-Errors-Duration* w skrócie RED) to klasyczne narzędzie obserwacyjności, które sięga początków usług online. Książka firmy Google *Site Reliability Engineering* omawia cztery złote sygnały, które są w zasadzie żądaniami, błędami, czasem trwania i nasyceniem (za chwilę omówimy nasycenie).

Inżynier Tom Wilkie, który wymyślił skrót RED, w swoim blogu przedstawił uzasadnienie tego wzorca.

Dlaczego warto mierzyć te same metryki dla każdej usługi? Z pewnością każda usługa jest wyjątkowa. Z punktu widzenia monitorowania korzyścią wynikającą z traktowania każdej usługi tak samo jest skalowalność zespołów operacyjnych. Sprawiając, że każda usługa wygląda, czuje i smakuje tak samo, zmniejszamy obciążenie poznawcze osób reagujących na zdarzenie. Nawiasem mówiąc, jeśli traktujesz wszystkie swoje usługi tak samo, wiele powtarzalnych zadań staje się żadaniami automatycznymi.

— Tom Wilkie

Jak dokładnie mierzmy te wartości? Ponieważ łączna liczba żądań stale rośnie, bardziej przydatne jest sprawdzenie częstotliwości żądań, np. liczby żądań na sekundę. Daje to sensowny obraz ruchu, jaki system obsługuje w danym przedziale czasu.

Ponieważ poziom błędów jest powiązany z częstotliwością żądań, warto mierzyć błędy procentowo. Przykładowo typowy pulpit usługi może pokazywać:

- żądania na sekundę,
- procent żądań, które zwróciły błąd,
- czas trwania żądań (znany również jako opóźnienie).

Zasoby: wzorzec USE

Dowiedziałeś się, że wzorzec RED oferuje użyteczne informacje na temat wydajności Twoich usług oraz ich wpływu na użytkowników. Można to uznać za odgórny sposób patrzenia na dane dotyczące obserwonalności.

Z drugiej strony, wzorzec USE (<http://www.brendangregg.com/usemethod.html>), opracowany przez inżyniera ds. wydajności Netflixa Brendana Gregg'a, jest podejściem oddolnym, które ma pomóc w analizie problemów z wydajnością i w znalezieniu wąskich gardel. USE oznacza *wykorzystanie, nasycenie i błędy* (ang. *Utilization, Saturation, and Errors*).

W USE zamiast usług interesują nas *zasoby*: fizyczne składniki serwera, takie jak procesor i dyski lub interfejsy sieciowe i łącza. Każdy z nich może stanowić wąskie gardło w wydajności systemu, a wskaźniki USE pomogą dowiedzieć się, który.

Wykorzystanie

Średni czas, w którym zasób był zajęty obsługą żądań, lub ilość aktualnie wykorzystywanej pojemności zasobu. Przykładowo dysk, który jest zapełniony w 90%, miałby wykorzystanie w 90%.

Nasycenie

Stopień przeciążenia zasobu lub długość kolejki żądań oczekujących na dostępność tego zasobu. Jeśli np. na uruchomienie czeka 10 procesów, wartość nasycenia wynosi 10.

Błędy

Liczba niepowodzeń operacji na tym zasobie. Przykładowo dysk z niektórymi uszkodzonymi sektorami może mieć liczbę błędów jako 25 nieudanych odczytów.

Mierzenie tych danych dla kluczowych zasobów w systemie jest dobrym sposobem na wykrycie wąskich gardel. Zasoby o niskim zużyciu, bez nasycenia i bez błędów są prawdopodobnie w porządku. Wszystko, co odbiega od tego, jest warte uwagi. Jeśli np. jeden z linków sieciowych jest nasycony lub zawiera dużą liczbę błędów, może przyczyniać się do ogólnych problemów z wydajnością.

Metoda USE to prosta strategia, za pomocą której można przeprowadzić pełną kontrolę stanu systemu i zidentyfikować typowe wąskie gardła oraz błędy. Można ją wdrożyć na wczesnym etapie dochodzenia i szybko zidentyfikować obszary problemowe, które można bardziej szczegółowo zbadać za pomocą innych metodologii.

Sila USE polega na szybkości i wizualizacji; jeśli weźmiesz pod uwagę wszystkie zasoby, prawdopodobnie nie przeocysz żadnych problemów. Jednak USE wykryje tylko niektóre rodzaje problemów — wąskie gardła i błędy — i należy ją traktować jako jedno narzędzie w większym zestawie.

— Brendan Gregg

Metryki biznesowe

Przyjrzaliśmy się metrykom aplikacji i usług („Usługi: wzorzec RED” w tym rozdziale), które prawdopodobnie będą najbardziej interesujące dla programistów, oraz metrykom sprzętowym („Zasoby: wzorzec USE” też w rozdziale), które przydadzą się inżynierom operacyjnym i inżynierom

infrastruktury. A co z biznesem? Czy obserwowałość może pomóc menadżerom zrozumieć rozwój firmy i dać użyteczny wkład w podejmowanie decyzji biznesowych? Jakie metryki przyczyniłyby się do tego?

Większość firm śledzi kluczowe mierniki wydajności (KPI), takie jak przychody ze sprzedaży, marża zysku i koszty pozyskania klientów. Dane te zwykle pochodzą z działu finansowego i nie wymagają obsługi ze strony programistów i personelu infrastruktury.

Istnieją jednak inne przydatne wskaźniki biznesowe, które mogą być generowane przez aplikacje i usługi. Przykładowo firma zajmująca się subskrypcją, czyli produktem SaaS (ang. *Software-as-a-service*), musi znać dane o swoich subskrybentach. Oto dane brane pod uwagę.

- Analiza lejka sprzedawczo-wyszukującego (ile osób trafiło na stronę docelową, ile kliknięć przechodzi na stronę rejestracji, ile osób kończy transakcję itd.).
- Wskaźnik rejestracji i rezygnacji.
- Przychody na klienta (przydatne do obliczania miesięcznych przychodów cyklicznych, średniego przychodu na klienta).
- Skuteczność stron pomocy i wsparcia (np. odsetek osób, które odpowiedziały twierdząco na pytanie: „Czy ta strona rozwiązała problem?”).
- Ruch do strony informującej o *stanie systemu* (który często gwałtownie wzrasta w przypadku awarii lub pogorszenia jakości usług).

Wiele z tych informacji łatwiej zebrać, generując dane metryk w czasie rzeczywistym z aplikacji, niż próbować analizować po fakcie, przetwarzając dzienniki i zapytania do baz danych. Kiedy instrumentujesz swoje aplikacje w celu generowania metryk, nie zaniedbij informacji ważnych dla firmy (patrz „Tworzenie wykresów metryk w pulpicie” w tym rozdziale) oraz raportów dla każdej zaangażowanej grupy.

Metryki Kubernetes

Pisaliśmy o obserwowałości i metrykach w ujęciu ogólnym oraz przyjrzelismy się różnym rodzajom danych i sposobom ich analizy. Jak to wszystko odnosi się do aplikacji Kubernetes? Jakie metryki warto śledzić w przypadku klastrów Kubernetes i jakie decyzje mogą pomóc?

Na najbliższym poziomie narzędzie o nazwie cAdvisor monitoruje wykorzystanie zasobów i statystyki wydajności dla kontenerów działających w każdym węźle klastra — np. ile zasobów procesora, pamięci i miejsca na dysku zużywa każdy kontener. Narzędzie cAdvisor jest częścią kubelet.

Kubernetes wykorzystuje dane cAdvisor, wysyłając zapytanie do kubelet i podejmuje decyzje dotyczące planowania, automatycznego skalowania itd. Możesz również wyeksportować te dane do usługi innej firmy, gdzie je przedstawisz na wykresie i wygenerujesz alarm. Przykładowo przydatne byłoby śledzenie, ile zasobów procesora i pamięci zużywa każdy kontener.

Możesz również monitorować sam Kubernetes za pomocą narzędzia o nazwie kube-state-metrics. Nasłuchuje ono interfejsu API Kubernetes i zgłasza informacje o obiektach logicznych, takich jak węzeł, Pod i Deployment. Dane te mogą być również bardzo przydatne dla obserwowałości klastra.

Jeśli np. dla obiektu Deployment skonfigurowano repliki, których z jakiegoś powodu nie można obecnie zaplanować (być może klaster nie ma wystarczającej pojemności), prawdopodobnie chcesz o tym wiedzieć.

Jak zwykle problemem nie jest brak danych metryki, ale decyzja o tym, na których metrykach powiniśmy się skupić, które śledzić i wizualizować. Oto pewne sugestie.

Wskaźniki kondycji klastra

Aby monitorować kondycję i wydajność klastra na najwyższym poziomie, powinieneś spojrzeć przynajmniej na kwestie, takie jak:

- liczba węzłów,
- kondycja węzła,
- liczba Podów na węzel i ogółem,
- wykorzystanie/alokacja zasobów na węzel i ogólnie.

Powyższe metryki pomogą Ci poznać wydajność klastra, tzn. czy ma wystarczającą pojemność, jak zmienia się jego użycie w czasie oraz czy musisz go rozszerzyć, czy też zmniejszyć.

Jeśli korzystasz z zarządzanej usługi Kubernetes, takiej jak GKE, węzły w złej kondycji zostaną wykryte automatycznie i automatycznie naprawione (pod warunkiem, że dla klastra i puli węzłów włączona jest funkcja automatycznej naprawy). Jednak nadal warto wiedzieć, czy liczba awarii odbiega od normy, gdyż może to wskazywać na poważniejszy problem.

Metryki obiektu Deployment

W przypadku wszystkich obiektów Deployment ważne są:

- liczba obiektów Deployment,
- liczba skonfigurowanych replik na Deployment,
- liczba niedostępnych replik na Deployment.

Jeśli włączyłeś niektóre z różnych opcji automatycznego skalowania dostępnych w Kubernetes, szczególnie przydatne jest śledzenie tych informacji w czasie (patrz „Automatyczne skalowanie” w rozdziale 6.). Warto zwrócić uwagę na dane dotyczące niedostępnych replik — ostrzegają o problemach związanych z pojemnością.

Metryki kontenera

Na poziomie kontenera ważne są:

- liczba kontenerów lub Podów na węzel i ogółem,
- wykorzystanie zasobów dla każdego kontenera w stosunku do jego żądań czy limitów (patrz „Żądania zasobów” w rozdziale 5.),
- żywotność lub gotowość kontenerów,
- liczba restartów kontenera lub Poda,
- ruch sieciowy i błędy dla każdego kontenera.

Ponieważ Kubernetes automatycznie restartuje te kontenery, które uległy awarii lub przekroczyły swoje limity zasobów, musisz wiedzieć, jak często to się dzieje. Nadmierna liczba restartów może wskazywać na problem z konkretnym kontenerem. Jeśli kontener regularnie przekracza swoje limity zasobów, może to oznaczać błąd programu, a może po prostu musisz nieznacznie zwiększyć limit.

Metryki aplikacji

Niezależnie od tego, jakiego języka lub platformy oprogramowania używa Twoja aplikacja, prawdopodobnie dostępna jest biblioteka lub narzędzie umożliwiające eksport niestandardowych danych. Są one szczególnie przydatne dla programistów i zespołów operacyjnych, aby mogli zobaczyć, co robi aplikacja, jak często to robi i jak długo to trwa. Są to kluczowe wskaźniki problemów związanych z wydajnością lub dostępnością.

Wybór metryk aplikacji do przechwytywania i eksportowania zależy od tego, co dokładnie aplikacja robi. Jednak są pewne wspólne wzorce. Jeśli np. Twoja usługa obsługuje wiadomości kolejki, przetwarza je i podejmuje pewne działania w oparciu o komunikat, możesz obserwować metryki, takie jak:

- liczba otrzymanych komunikatów,
- liczba pomyślnie przetworzonych komunikatów,
- liczba nieprawidłowych lub błędnych komunikatów,
- czas przetwarzania każdego komunikatu i reakcji na każdy z nich,
- liczba wygenerowanych udanych akcji,
- liczba nieudanych działań.

Jeśli Twoja aplikacja jest oparta głównie na żądaniach, możesz użyć wzorca RED (patrz „Usługi: w rzec RED” w tym rozdziale) i obserwować:

- otrzymane żądania,
- zwrócone błędy,
- czas trwania (czas obsługi każdego żądania).

Trudno określić, jakie metryki będą przydatne, gdy jesteś na wczesnym etapie rozwoju. W razie wątpliwości rejestruj wszystko. Dane są tanie; możesz odkryć nieprzewidziany problem z produkcją dzięki danym metryk, które na początku nie wydawały się ważne.

Jeśli się zmienia, zanotuj. Jeśli nawet się nie zmienia i tak zanotuj, bo może kiedyś się zmieni.

— Laurie Denness (<https://twitter.com/lozzd/status/604064191603834880>)
(Bloomberg)

Jeśli chcesz, aby aplikacja generowała metryki biznesowe (patrz „Metryki biznesowe” w tym rozdziale), możesz je również obliczyć i wyeksportować jako metryki niestandardowe.

Inną rzeczą, która może być przydatna dla firmy, jest sprawdzenie, jak aplikacje radzą sobie z dowolnymi celami poziomu usług (SLO — ang. *Service Level Objectives*) lub umowami SLA (ang. *Service Level Agreements*), które możesz mieć z klientami, lub jak usługi dostawcy radzą sobie z SLO. Możesz utworzyć własną metrykę, która będzie wyświetlać docelowy czas trwania żądania (np. 200 ms), i utworzyć odpowiedni pulpit, który będzie nakładał to na rzeczywistą bieżącą wydajność.

Metryki czasu wykonywania

Na poziomie środowiska wykonawczego większość bibliotek metryk będzie również raportować przydatne dane o tym, co robi program, takie jak:

- liczba procesów/wątków/goroutines,
- wykorzystanie stosów i sterty,
- wykorzystanie pozostałej pamięci,
- pule buforów sieciowych,
- uruchamianie modułu odśmiecania pamięci (ang. *garbage collector*) i okresy pauzy (w przypadku języków z takich modułem),
- używane deskryptory plików/gniazda sieciowych.

Tego rodzaju informacje mogą być bardzo cenne w diagnozowaniu słabej wydajności, a nawet awarii. Przykładowo dość często w dugo działających aplikacjach stopniowo wykorzystuje się coraz więcej pamięci, dopóki nie zostaną zrestartowane z powodu przekroczenia limitów zasobów Kubernetes. Pomiary czasu wykonywania aplikacji mogą pomóc Ci dokładnie określić, gdzie ta pamięć wycieka, szczególnie w połączeniu z niestandardowymi pomiarami tego, co robi aplikacja.

Teraz, gdy już orientujesz się, jakie dane metryki warto obserwować, przejdziemy do następnego podrozdziału, aby dowiedzieć się, co zrobić z tymi danymi, innymi słowy, jak je analizować.

Analizowanie metryk

Dane to nie to samo co informacja. Aby uzyskać przydatne informacje z przechwyconych nieprzewrózonych danych, musimy je agregować, przetwarzać i analizować, co oznacza tworzenie *statystyk* na ich temat. Statystyki mogą dawać dosyć abstrakcyjne pojęcie, dlatego zilustrujemy tę dyskusję konkretnym przykładem, czyli czasem trwania zapytania.

W tym rozdziale, w punkcie „Usługi: wzorzec RED”, wspomnialiśmy, że należy śledzić metrykę czasu trwania zgłoszeń serwisowych, ale nie powiedzieliśmy dokładnie, jak to zrobić. Co dokładnie rozumiemy przez czas trwania? Zazwyczaj interesuje nas czas, w którym użytkownik musi czekać na odpowiedź na jakieś żądanie.

Przykładowo dla witryny internetowej możemy zdefiniować czas trwania jako czas między połączeniem się użytkownika z serwerem a momentem, w którym serwer zacznie odpowiadać.

(Całkowity czas oczekiwania użytkownika jest dłuższy niż ten, ponieważ nawiązanie połączenia zajmuje trochę czasu, podobnie jak odczyt danych odpowiedzi i renderowanie ich w przeglądarce. Zazwyczaj jednak nie mamy dostępu do tych danych, więc po prostu notujemy to, co możemy).

Każde żądanie ma inny czas trwania, więc w jaki sposób agregować dane setek, a nawet tysięcy żądań do jednej wartości?

Dlaczego nie korzystać z wartości średniej?

Oczywistą odpowiedzią jest wyciągnięcie średniej. Jednak przy bliższym poznaniu, co oznacza średnia, możemy stwierdzić, że niekoniecznie przedstawia ona rzetelną wartość. Stary żart o statystykach mówi, że przeciętny człowiek ma nieco mniej niż dwie nogi. Innymi słowy, większość ludzi ma więcej niż średnia liczba nóg. Dlaczego?

Większość ludzi ma dwie nogi, ale niektórzy mają jedną lub nie mają nóg, co obniża ogólną średnią. (Być może niektóre osoby mają więcej niż dwie, ale o wiele więcej osób ma mniej niż dwie). Prosta średnia nie daje wielu użytecznych informacji na temat rozmieszczenia nóg w populacji ani o doświadczeniu większości osób w posiadaniu nóg.

Istnieje również więcej niż jeden rodzaj średniej. Prawdopodobnie wiesz, że powszechnie pojęcie średniej odnosi się do średniej arytmetycznej. Średnia arytmetyczna zestawu wartości jest sumą wszystkich wartości podzieloną przez liczbę wartości. Przykładowo średni wiek grupy trzech osób to suma ich wieku podzielona przez 3.

Z drugiej strony, mediana odnosi się do wartości, która dzieli zbiór na dwie równe połówki, jedna zawiera wartości większe niż mediana, a druga zawiera mniejsze wartości. Przykładowo w dowolnej grupie osób połowa z nich jest z definicji wyższego wzrostu niż mediana wzrostu, a połowa z nich jest niższego wzrostu.

Średnie arytmetyczne, mediany i wartości odstające

Dlaczego dla czasu trwania żądania nie możemy przyjmować wartości średniej? Jednym z ważnych problemów jest to, że średnią łatwo wypaczają wartości *odstające*, tzn. jedna lub dwie skrajne wartości mogą znacząco zniekształcić średnią.

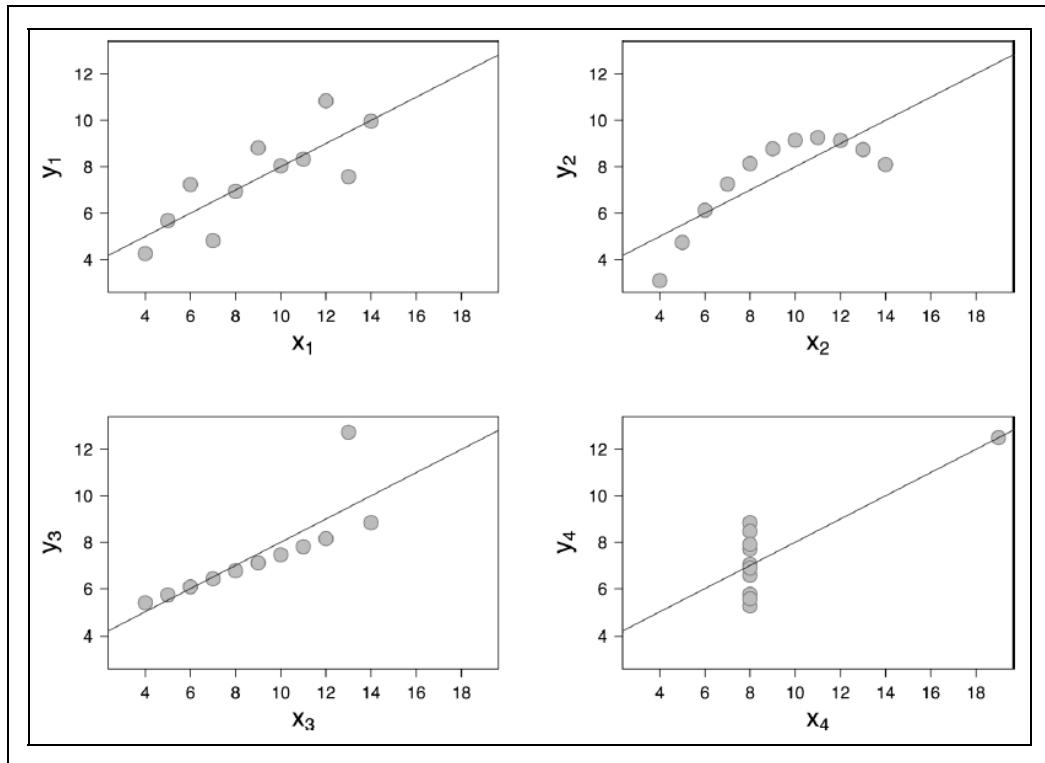
W związku z tym mediana, na którą wartości odstające mają mniejszy wpływ, jest bardziej pomocnym sposobem uśrednienia wskaźników niż średnia. Jeśli mediana opóźnienia dla usługi wynosi 1 sekundę, połowa użytkowników ma opóźnienie poniżej 1 sekundy, a połowa większe.

Rysunek 16.2 pokazuje, w jaki sposób średnie mogą być mylące. Wszystkie cztery zestawy danych mają tę samą wartość średnią, ale wygląda to zupełnie inaczej, gdy są przedstawione graficznie (statystycy znają ten przykład jako *kwartet Anscombe'a*). Nawiasem mówiąc, jest to również dobry sposób, aby zademonstrować ważność wykresów danych; nie warto patrzeć tylko na surowe liczby.

Odkrywanie percentylów

Kiedy mówimy o metrykach służących do obserwowania systemów opartych na żądaniach, zwykle interesuje nas to, jakie są *najgorsze* opóźnienia w obsłudze użytkowników, a nie średnia. W końcu mediana opóźnienia wynosząca 1 sekundę dla wszystkich użytkowników nie stanowi pocieszenia dla małej grupy, która może doświadczać opóźnień wynoszących 10 sekund lub więcej.

Sposobem na uzyskanie tych informacji jest rozbicie danych na *percentyle*. Opóźnienie 90 percentyla (często nazywane P90) jest wartością większą niż ta, której doświadczyło 90% użytkowników. Innymi słowy, 10% użytkowników doświadcza opóźnienia większego niż wartość P90.



Rysunek 16.2. Te cztery zestawy danych mają tę samą średnią wartość (obrazek (https://en.wikipedia.org/wiki/Anscombe%27s_quartet#/media/File:Anscombe%27s_quartet_3.svg) autorstwa Schutza, CC BY-SA 3.0.)

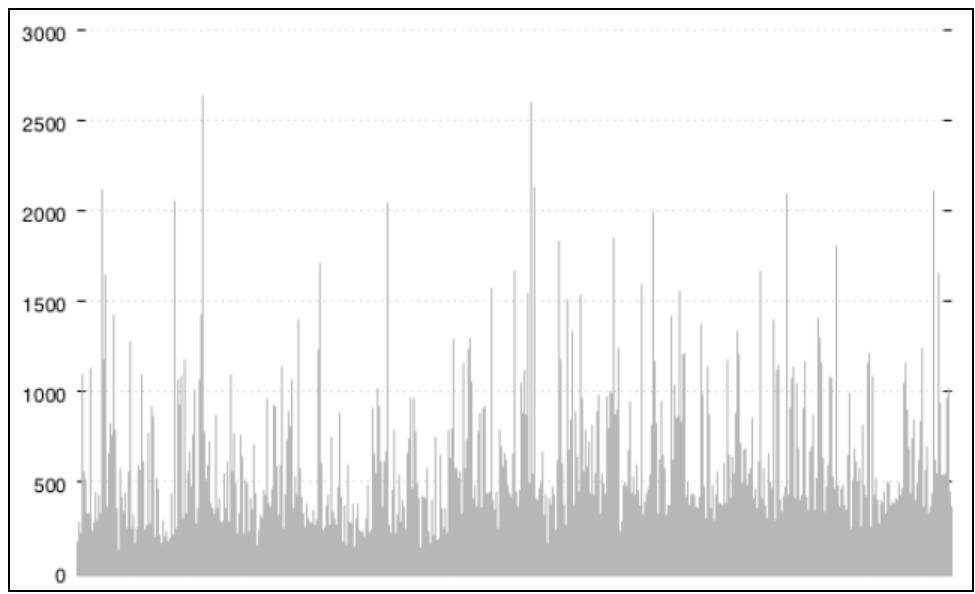
Medianą wyrażoną w tym języku jest 50 percentylem lub P50. Inne percentyle, które często są mierzone obserwowalnością, to P95 i P99, odpowiednio 95 i 99 percentyla.

Sposobanie percentyl do danych metryk

Igor Wiedler z Travis CI przygotował dobrą demonstrację (<https://igor.io/latency>), która wszystko objaśnia. W prezentacji ujęto zestaw danych obejmujący 135 000 żądań do usługi produkcyjnej wysyłanych w ciągu 10 minut (patrz rysunek 16.3). Jak widać, dane te są zaszumione, rozstrzelone i niełatwo wyciągnąć z nich użyteczne wnioski.

Zobaczmy teraz, co się stanie, jeśli uśrednimy te dane w odstępach 10-sekundowych (patrz rysunek 16.4). Wygląda wspaniale: wszystkie punkty danych są poniżej 50 ms. Wygląda więc na to, że większość naszych użytkowników ma opóźnienia poniżej 50 ms. Czy tak jest naprawdę?

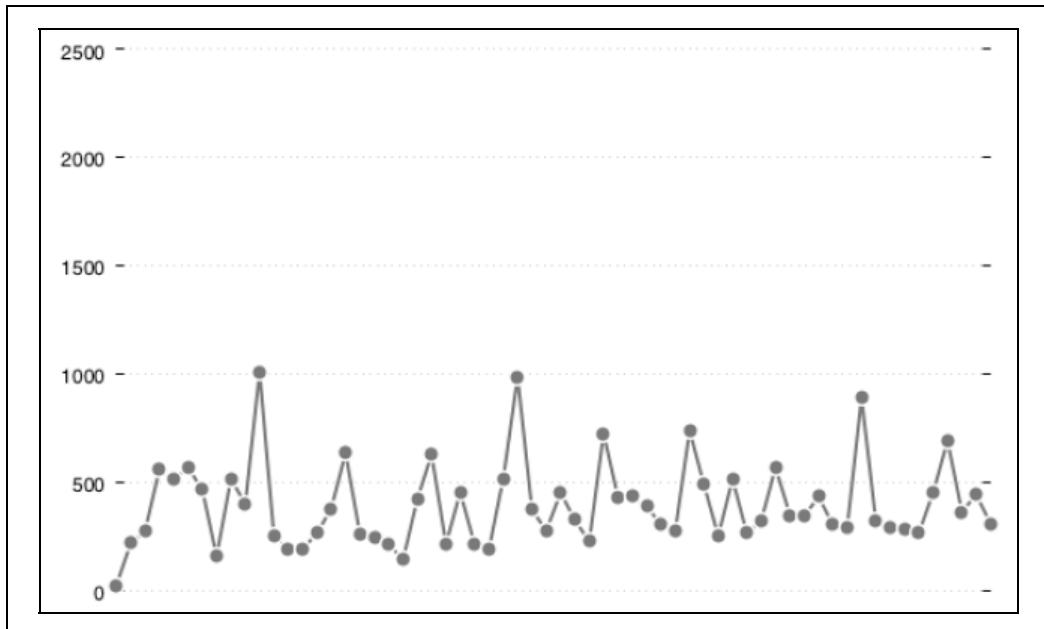
Zamiast tego, wykreślmy opóźnienie P99. Jest to maksymalne zaobserwowane opóźnienie, jeśli odrzucimy najwyższy 1% próbek. Wygląda zupełnie inaczej (patrz rysunek 16.5). Teraz widzimy wykres, w którym większość wartości jest grupowana w przedziale od 0 do 500 ms, a dla kilku wartości dochodzi do 1000 ms.



Rysunek 16.3. Surowe dane opóźnień dla 135 000 żądań, w ms



Rysunek 16.4. Średnie opóźnienie dla tych samych danych w odstępach 10-sekundowych



Rysunek 16.5. Opóźnienie P99 (99 percentyl) dla tych samych danych

Zwykle chcemy zobaczyć to, co najgorsze

Ponieważ przypadkowo obserwujemy wartości wolniejszych żądań sieciowych, dane P99 prawdopodobnie zapewniają bardziej realistyczny obraz opóźnień doświadczanych przez użytkowników. Weźmy np. witrynę o dużym ruchu z 1 milionem odsłon dziennie. Jeśli opóźnienie P99 wynosi 10 sekund, to 10 000 odsłon trwa dłużej niż 10 sekund. To wielu niezadowolonych użytkowników.

Jednak jest coraz gorzej: w systemach rozproszonych każde wyświetlenie strony może wymagać dziesiątek, a nawet setek żądań wewnętrznych. Jeśli opóźnienie P99 każdej usługi wewnętrznej wynosi 10 sekund, a 1 wyświetlenie strony wysyła 10 żądań wewnętrznych, liczba powolnych wyświetleń strony wzrasta do 100 000 dziennie. W tym przypadku około 10% użytkowników jest niezadowolonych, co stanowi duży problem (<https://engineering.linkedin.com/performance/who-moved-my-99th-percentile-latency>).

Co zamiast percentylów?

Jednym z problemów związanym z opóźnieniami percentylowymi, wdrażanymi przez wiele usług metrycznych, jest to, że żądania są próbkowane lokalnie, a statystyki są następnie agregowane centralnie. W rezultacie opóźnienie P99 jest średnim opóźnieniem P99 zgłoszonym przez każdego agenta, potencjalnie z grupy setek agentów.

Cóż, percentyl jest już wartością średnią, a próba uśrednienia średnich jest dobrze znaną pułapką statystyczną (https://pl.wikipedia.org/wiki/Paradoks_Simpsona). Wynik niekoniecznie jest taki sam jak rzeczywista średnia.

W zależności od tego, w jaki sposób agregujemy dane, ostateczna wartość opóźnienia P99 może różnić się aż o współczynnik 10. To dobrze nie wróży. Jeśli usługa metryk nie uwzględnia każdego pojedynczego nieprzetworzonego zdarzenia i nie generuje prawdziwej średniej, liczba ta będzie niewiarygodna.

Inżynier Yan Cui (<https://medium.com/theburningmonk-com/we-can-do-better-thanpercentile-latencies-2257d20c3b39>) sugeruje, że lepszym podejściem jest monitorowanie tego, co jest złe.

Z czego moglibyśmy skorzystać zamiast percentylów jako podstawowej miary do monitorowania wydajności naszej aplikacji i ostrzegania nas, gdy zacznie się ona pogarszać?

Jeśli wrócisz do SLO lub SLA, prawdopodobnie uzyskujesz coś takiego: „99% żądań powinno zostać zakończonych w ciągu 1s lub mniej”. Innymi słowy, mniej niż 1% żądań może zająć więcej niż 1 sekundę.

A jeśli zamiast tego monitorujemy odsetek żądań przekraczających próg? Aby ostrzec nas o naruszeniu naszych umów SLA, możemy wywoływać alarmy, gdy odsetek ten jest większy niż 1% w określonym przedziale czasu.

— Yan Cui

Jeśli każdy agent prześle metrykę całkowitych żądań i liczbę żądań, które przekroczyły próg, *możemy* uśrednić te dane, aby wygenerować procent żądań, które przekroczyły SLO — i o tym ostrzec.

Tworzenie wykresów metryk w pulpicie

W tym rozdziale dowiedziałeś się już, dlaczego metryki są przydatne, jakie metryki powinieneś zapisać, oraz poznałeś kilka przydatnych technik statystycznych do ich analizy zbiorczej. Wszystko wygląda dobrze, ale co właściwie *zrobimy* z tymi wszystkimi danymi?

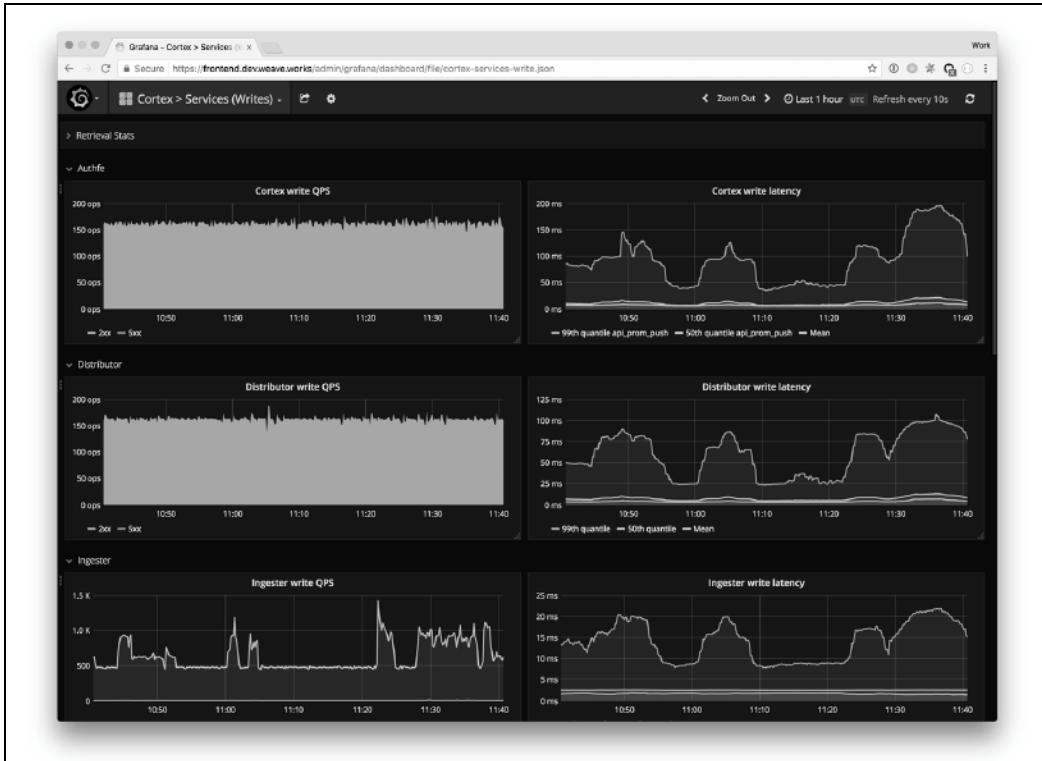
Odpowiedź jest prosta: zamierzamy je przedstawić na wykresie, umieścić na pulpicie i być może generować odpowiednie alarmy. O generowaniu alarmów porozmawiamy w następnym podrozdziale, a na razie przyjrzyjmy się niektórym narzędziom i technikom służącym do tworzenia wykresów i pulpitów.

Użyj standardowego układu graficznego dla wszystkich serwisów

Jeśli masz więcej niż kilka serwisów, warto ułożyć pulpity w taki sam sposób dla każdego z nich. Ktoś, kto pełni dyżur, gdy zerknie na pulpit danego serwisu, zinterpretuje dane natychmiast, bez konieczności znajomości tego konkretnego serwisu.

Tom Wilkie w poście blogu Weaveworks (<https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture/>) sugeruje następujący standardowy format (patrz rysunek 16.6).

- Jeden rzząd na serwis.
- Liczba żądań i błędów po lewej stronie, z błędami wyrażonymi jako procent żądań.
- Opóźnienie po prawej stronie.



Rysunek 16.6. Weaveworks zaproponował układ pulpitu dla serwisów

Nie musisz używać dokładnie tego układu; ważne jest to, żeby zawsze używać tego samego układu dla każdego pulpu i mieć pewność, że wszyscy go znają. Powinieneś regularnie przeglądać kluczowe pulpity (przynajmniej raz w tygodniu), sprawdzając dane z poprzedniego tygodnia, aby wszyscy wiedzieli, jaki jest *normalny* wygląd.

Pulpit żądań, błędów i czasu trwania działa dobrze w przypadku usług (patrz „Usługi: wzorzec RED” w tym rozdziale). W przypadku zasobów, takich jak węzły klastra, dyski i sieć, najbardziej przydatne są (zazwyczaj) stopień wykorzystania, nasycenie, błędy (patrz „Zasoby: wzorzec USE” w tym rozdziale).

Zbuduj radiator informacji za pomocą pulpitu

Jeśli masz sto serwisów, masz sto pulpitów, na które prawdopodobnie nie będziesz często patrzeć. Ważne jest, aby te informacje były dostępne (np. w celu wykrycia, który serwis nie działa), ale w tej skali potrzebujesz bardziej ogólnego przeglądu.

Aby to zrobić, utwórz główny pulpit, który pokazuje żądania, błędy i czas trwania dla *wszystkich* serwisów łącznie. Nie rób nic fantazyjnego, takiego jak mapy warstwowe; trzymaj się prostych wykresów liniowych. Są łatwiejsze do interpretacji oraz lepiej wizualizują dane niż złożone wykresy.

Najlepiej będzie użyć *radiatora informacji* (ang. *information radiator*). Jest to duży ekran pokazujący kluczowe dane obserwacyjności, które są widoczne dla wszystkich w odpowiednim zespole lub biurze. Celem radiatora informacji jest:

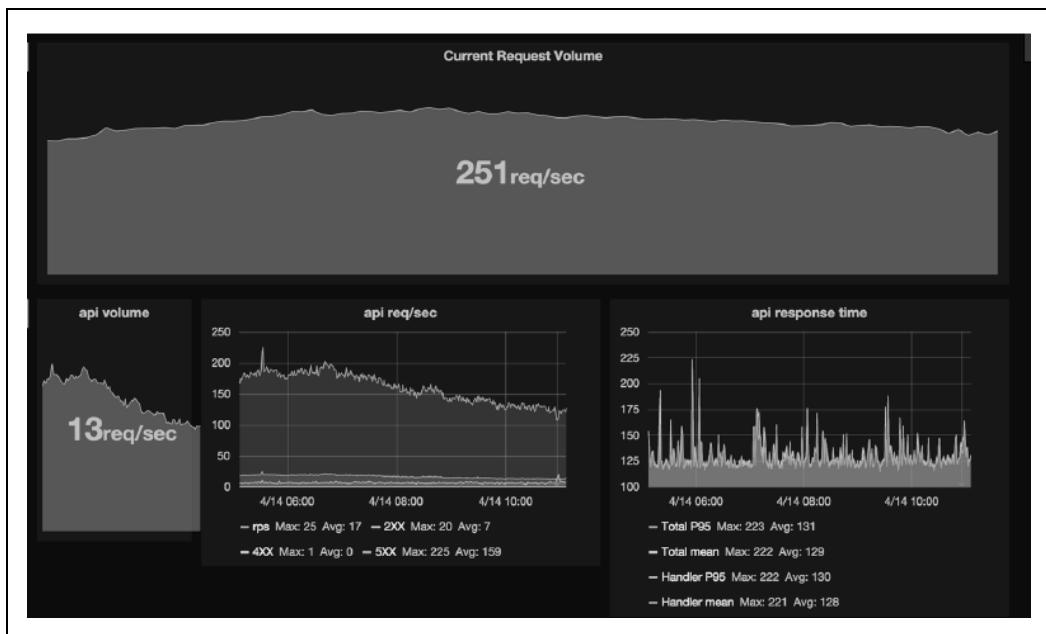
- szybkie pokazanie aktualnego stanu systemu,
- pokazanie jasnej informacji o tym, które wskaźniki zespół uważa za ważne,
- zapoznanie ludzi z tym, jak wygląda normalna sytuacja.

Co należy uwzględnić na ekranie radiatora? Tylko istotne informacje. Istotne w sensie *naprawdę ważnym*, ale także w znaczeniu *znaków życiowych*, czyli informacji, które mówią o życiu systemu.

Dobrym przykładem są monitory czynności życiowych, które zobaczysz przy łóżku szpitalnym. Pokazują kluczowe wskaźniki dla ludzi: tętno, ciśnienie krwi, nasycenie tlenem, temperaturę i częstotliwość oddechów. Istnieje wiele innych wskaźników, które można śledzić u pacjenta i mają one medycznie ważne zastosowania, ale na poziomie głównego pulpitu nie są kluczowe. Każdy poważny problem medyczny pojawi się w co najmniej jednym z tych wskaźników; wszystko inne jest kwestią diagnostyki.

Podobnie radiator informacji powinien pokazywać istotne cechy Twojej firmy lub serwisu. Jeśli prezentowane są wartości liczbowe, prawdopodobnie nie powinno być ich więcej niż cztery lub pięć. Jeśli prezentowane są wykresy, nie powinno być ich więcej niż cztery lub pięć.

Często kuszące jest wrzucanie zbyt dużej ilości informacji. To nie może być celem. Celem jest skupienie się na kilku kluczowych rzeczach i sprawienie, aby były łatwo widoczne z drugiego końca pokoju (patrz rysunek 16.7).



Rysunek 16.7. Przykładowy radiator informacji wygenerowany przez narzędzie Grafana Dash Gen (<https://github.com/uber/grafana-dash-gen>)

Umieszczanie na pulpicie rzeczy, które ulegają awarii

Oprócz głównego radiatora informacji oraz pulpitów dla poszczególnych serwisów i zasobów, możesz utworzyć pulpity dla określonych metryk, które informują o ważnych rzeczach w systemie. Mogą to być rzeczy bazujące na architekturze systemu. Jednak innym przydatnym źródłem informacji są rzeczy, które *ulegają awarii*.

Z każdym razem, gdy wystąpi jakieś zdarzenie lub awaria, poszukaj metryki lub kombinacji metryk, które z góry powiadomiąby Cię o tym problemie. Jeśli np. masz przestój produkcyjny spowodowany brakiem miejsca na serwerze, możliwe, że wykres pokazujący ilość miejsca na dysku na tym serwerze uprzedziłby Cię wcześniej, że dostępne miejsce wykazuje tendencję spadkową i zmierza w kierunku wystąpienia przestoju.

Nie mówimy tutaj o problemach, które pojawiają się w ciągu kilku minut lub nawet godzin; są one zazwyczaj przechwytywane przez automatyczne alerty (patrz „Alerty na podstawie metryk” w tym rozdziale). Interesują nas raczej wolno poruszające się tendencje, które zbliżają się w ciągu dni lub tygodni, a jeśli ich nie zauważysz i nie podejmiesz żadnych działań, Twój system ulegnie awarii w najgorszym możliwym momencie.

Po wystąpieniu incydentu zawsze zadawaj pytanie: „Co ostrzegłoby nas wcześniej o tym problemie, gdybyśmy byli tego świadomi?”. Jeśli odpowiedź zawiera dane, które już posiadałeś, ale na które nie zwracałeś uwagi, podejmij działania, aby je podświetlić. Wykorzystanie pulpitu jest jednym z możliwych sposobów, by to zrobić.

Chociaż alerty informują, że pewna wartość przekroczyła ustalony próg, nie zawsze możesz z góry wiedzieć, jaki jest poziom zagrożenia. Wykres umożliwia wizualizację zachowania tej wartości przez długi czas i pomaga wykryć problematyczne trendy, zanim faktycznie wpłyną na system.

Alerty na podstawie metryk

Möesz być zaskoczony, że większość tego rozdziału poświęciliśmy na obserwowanie i monitorowanie, a nie wspominaliśmy o alarmach. Dla niektórych osób alerty są istotą monitorowania. Uważamy, że takie podejście z wielu powodów musi się zmienić.

Jakie są problemy związane z alarmami?

Alerty wskazują pewne nieoczekiwane odchylenie od stabilnego stanu roboczego. Cóż, systemy rozproszone nie mają takich stanów!

Jak już wspomnieliśmy, duże systemy rozproszone nigdy nie są w stanie *działać*; prawie zawsze znajdują się w stanie częściowo zdegradowanej usługi (patrz „Aplikacje cloud native zawsze nie działają” w rozdziale 15.). Mają tak wiele metryk, że jeśli zaalarmujesz za każdym razem, gdy niektóre z nich przekroczą normalne limity, wygenerujesz setki raportów dziennie, bez żadnego celu.

Ludzie sami nadmiernie raportują, ponieważ ich obserwonalność maleje i przestały ufać narzędziom, które pozwalają im niezawodnie debugować i diagnozować problem. Aby poznać przyczynę problemu, otrzymujesz dziesiątki lub setki alertów, które dopasowują do wzorca.

*Błędzą. W chaotycznej przyszłości, do której wszyscy zmierzamy, musisz mieć określone zasady, aby generować znacznie **mniej** alarmów: częstotliwość żądań, opóźnienie, współczynnik błędów, nasycenie.*

— Charity Majors (<https://www.infoq.com/articles/charity-majors-observability-failure>)

Dla niektórych osób alarmowe powiadomienia telefoniczne to sposób życia. To zła rzecz, nie tylko z oczywistych ludzkich powodów. Alarm zmęczenia jest dobrze opisanym problemem występującym w medycynie. Kiedy alarmy występują ciągle, lekarze stają się wobec nich obojętni, co zwiększa prawdopodobieństwo, że przeoczą poważny problem, gdy się pojawi. Aby system monitorowania był użyteczny, musi mieć bardzo wysoki stosunek sygnału do szumu.

Fałszywe alarmy są nie tylko irytujące, ale i niebezpieczne, gdyż zmniejszają zaufanie do systemu i uczą, że alarmy można bezpiecznie zignorować.

Nadmierne, nieustanne i nieistotne alarmy były głównym czynnikiem w katastrofie Three Mile Island (<https://humanisticsystems.com/2015/10/16/fit-for-purpose-questions-about-alarm-system-design-from-theory-and-practice/>). Jeśli nawet poszczególne alarmy są dobrze zaprojektowane, operatorzy mogą zostać przytłoczeni zbyt dużą liczbą jednocięsnych alarmów.

Ostrzeżenie powinno oznaczać jedną bardzo prostą rzec: *osoba musi podjąć działanie* (<https://www.infoworld.com/article/3268126/devops/beware-the-danger-of-alarm-fatigue-in-it-monitoring.html>).

Jeśli nie jest wymagane żadne działanie, nie jest potrzebny alarm. Jeśli jakieś działanie musi się *kiedyś* zdarzyć, ale nie teraz, alarm można zmienić na wiadomość e-mail lub czat. Jeśli działanie może podjąć automatyczny system, to zautomatyzuj je: nie angażuj cennego człowieka.

Bezbolesna reakcja na żądanie

Chociaż pomysł reagowania na żądanie to klucz do filozofii DevOps, równie ważne jest, aby dyżutowanie było bezbolesnym doświadczeniem.

Powiadomienia o wystąpieniu awarii powinny być rzadkim i wyjątkowym wydarzeniem. Kiedy tak się dzieje, powinna istnieć dobrze ustalona i skuteczna procedura postępowania z nimi, która w jak najmniejszym stopniu obciąża osobę rozwiązyującą problem.

Nikt nie powinien dyżurować przez cały czas. W takim przypadku przydziel więcej osób. Nie musisz być ekspertem, aby pełnić dyżur: Twoim głównym zadaniem jest rozwiązywanie problemu oraz podejmowanie decyzji, czy należy podjąć działania i przekazać problem dalej odpowiednim osobom.

Ciężar dyżuru powinien być sprawiedliwie rozłożony, jednak sytuacja osobista ludzi jest inna. Jeśli masz rodzinę lub inne zobowiązania poza pracą, zmiany na dyżurze wcale nie mogą być tak łatwo przyjęte. Dobre zarządzanie wymaga zorganizowania dyżurów w sposób sprawiedliwy dla wszystkich.

Jeśli praca wymaga dyżuru, należy to wyjaśnić osobie zatrudnionej. Oczekiwania dotyczące częstotliwości i okoliczności zmian dyżurów powinny być zapisane w umowie. To niesprawiedliwe, gdy zatrudniamy kogoś do pracy w godzinach 9 – 17, a następnie wymagamy, aby pełnił także dyżury w nocy i w weekendy.

Taki czas pracy trzeba odpowiednio kompensować wynagrodzeniem, czasem wolnym lub inną istotną korzyścią. Niezależnie od tego, czy otrzymujesz jakieś powiadomienia, czy też nie, kiedy jesteś na dyżurze, do pewnego stopnia jesteś w pracy.

Powinien również istnieć twardy limit czasu, który ktoś może spędzić na dyżurze. Ludzie, którzy mają więcej wolnego czasu lub energii, mogą zgłosić się na ochotnika, aby zmniejszyć stres swoich współpracowników. Jest to wspaniałe, ale nie pozwól nikomu wziąć na siebie zbyt wiele.

Uznaj, że kiedy wysydasz ludzi na dyżur, wydajesz kapitał ludzki. Wydaj go mądrze.

Pilne, ważne i przydatne alarmy

Dlaczego w ogóle o mówimy o alarmach? Nadal potrzebujesz alarmów. Różne rzeczy się psują — zwykle w najbardziej niewygodnym momencie.

Obserwowalność jest cudowna, ale nie możesz znaleźć problemu, gdy go nie szukasz. Pulpity są świetne, ale nie płacisz, aby ktoś siedział i patrzył na pulpit przez cały dzień. Aby wykryć awarię lub problem, który się teraz dzieje, nie możesz pozbyć się automatycznych alertów opartych na progach.

Możesz np. chcieć, aby system ostrzegał Cię, jeśli poziom błędu dla danej usługi przekroczy 10% przez pewien okres czasu, przykładowo 5 minut. Możesz wygenerować alarm, gdy opóźnienie P99 dla usługi przekroczy pewną stałą wartość, np. 1000 ms.

Ogólnie rzecz biorąc, jeśli problem ma rzeczywisty lub potencjalny wpływ na biznes oraz wymaga ludzkiego działania, może być potencjalnym kandydatem do wygenerowania alarmu.

Nie ostrzegaj o każdej metryce. Z setek, a może tysięcy metryk powinieneś mieć tylko garść tych, które mogą generować alerty. Jeśli nawet je generują, to niekoniecznie oznacza, że musisz kogoś informować.

Ostrzeżenia o alarmach powinny być ograniczone tylko do *pilnych, ważnych i możliwych do rozwiązania alarmów*.

- Alarmy, które są ważne, ale nie są pilne, można rozpatrywać w normalnych godzinach pracy. Należy informować tylko o takich alarmach, które nie powinny czekać do rana.
- Alarmy, które są pilne, ale nie są ważne, nie sprawiedliwiają wybudzenia kogoś. Przykładem może być awaria rzadko używanej usługi wewnętrznej, która nie wpływa na klientów.
- Jeśli nie można podjąć natychmiastowych działań, aby wykonać naprawę, nie ma sensu informowanie o alarmie.

Poza tym możesz wysyłać asynchroniczne powiadomienia: e-maile, wiadomości typu Slack, zgłoszenia do pomocy technicznej, problemy z projektami itd. Będą one widoczne i obsługiwane w odpowiednim czasie, jeśli Twój system działa poprawnie. Nie musisz podnosić czyjegoś poziomu kortyzolu gwałtownie w górę, wybudzając go w środku nocy z powodu alarmu.

Śledź swoje alarmy, powiadomienia poza godzinami pracy i pobudki

Twoi ludzie są tak samo krytyczni dla Twojej infrastruktury jak serwery w chmurze i klastry Kubernetes, jeśli nie bardziej. Są prawdopodobnie drożsi i na pewno trudniejsi do wymiany. Sensowne jest zatem monitorowanie tego, co dzieje się z Twoimi ludźmi, w taki sam sposób, w jaki monitorujesz to, co dzieje się z Twoimi usługami.

Liczba alertów wysłanych w danym tygodniu jest dobrym wskaźnikiem ogólnego stanu zdrowia i stabilności Twojego systemu. Liczba pilnych zgłoszeń, a zwłaszcza liczba powiadomień wysyłanych poza godzinami pracy, w weekendy i podczas normalnego snu, jest dobrym wskaźnikiem ogólnego stanu zdrowia i morale Twojego zespołu.

Należy ustawić budżet na liczbę pilnych zgłoszeń, zwłaszcza poza godzinami pracy; powinien on być bardzo niski. Prawdopodobnie limitem powinny być jedno lub dwa zgłoszenia poza godzinami pracy na inżyniera na tydzień. Jeśli regularnie to przekraczasz, musisz naprawić alerty, naprawić system lub zatrudnić więcej inżynierów.

Przeglądaj wszystkie pilne zgłoszenia, co najmniej raz w tygodniu, i napraw lub wyeliminuj fałszywe lub niepotrzebne alarmy. Jeśli nie potraktujesz tego poważnie, ludzie nie będą traktować Twoich alertów poważnie. A jeśli regularnie przerywasz sen i życie prywatne niepotrzebnymi alertami, zaczną szukać lepszej pracy.

Narzędzia i usługi metryczne

Przejdzmy zatem do kilku szczegółów. Jakich narzędzi lub usług należy używać do zbierania, analizowania i przekazywania danych? W podpunkcie „Nie buduj własnej infrastruktury monitorowania” w rozdziale 15. zwróciliśmy uwagę, że w przypadku problemu z materiałami należy zastosować rozwiązanie materiałowe. Czy to oznacza, że powinieneś koniecznie korzystać z usług hostowanych metryk innych firm, takich jak Datadog lub New Relic?

Odpowiedź nie jest jednoznaczna. Chociaż usługi te oferują wiele zaawansowanych funkcji, mogą być drogie, zwłaszcza na dużą skalę. Nie ma przekonującego argumentu biznesowego przeciwko uruchomieniu własnego serwera metryk, ponieważ jest dostępny doskonały darmowy produkt o otwartym kodzie źródłowym.

Prometheus

Standardowym rozwiązaniem dla metryk w świecie cloud native jest Prometheus. Jest bardzo szeroko stosowany, szczególnie w Kubernetes (prawie wszystko może w jakiś sposób współdziałać z produktem Prometheus), więc jest to pierwsza rzecz, którą powinieneś wziąć pod uwagę, gdy myślisz o opcjach monitorowania metryk.

Prometheus to zestaw narzędzi open source służący do monitorowania i ostrzegania o problemach w systemie, na podstawie danych metryk w czasie. Rdzeniem narzędzia Prometheus jest serwer, który gromadzi i przechowuje metryki. Zawiera też różne inne opcjonalne komponenty, takie jak narzędzie alarmujące (Alertmanager) oraz biblioteki dla języków programowania, takich jak Go, których można użyć do instrumentowania aplikacji.

Wszystko wygląda na dość skomplikowane, ale w praktyce jest bardzo proste. Możesz zainstalować narzędzie Prometheus w klastrze Kubernetes za pomocą jednego polecenia — przy użyciu standardowego wykresu Helm (patrz „Helm: menadżer pakietów Kubernetes” w rozdziale 4.). Następnie narzędzie zbierze metryki automatycznie z klastra, a także z dowolnych aplikacji za pomocą procesu zwanego *scrapingiem*.

W wyniku tego procesu Prometheus wykonuje połączenie HTTP z aplikacją na wcześniej ustalonym porcie i pobiera dostępne dane. Następnie przechowuje dane w swojej bazie danych, gdzie będą dostępne dla Ciebie — dla zapytań, prezentacji wykresu lub wygenerowania alarmu.



System monitoringu narzędzia Prometheus może zbierać metryki w oparciu o model *pull*. W tym podejściu serwer monitorowania kontaktuje się z aplikacją i żąda danych metryk. Odwrotne podejście, zwane modelem *push*, używane przez niektóre inne narzędzia monitorowania, takie jak StatsD, działa w inny sposób: aplikacje kontaktują się z serwerem monitorowania, aby wysłać metryki.

Podobnie jak sam Kubernetes, Prometheus jest inspirowany infrastrukturą Google. Został opracowany w SoundCloud, ale wiele pomysłów czerpie z narzędzia o nazwie Borgmon. Borgmon, jak sama nazwa wskazuje, został zaprojektowany do monitorowania systemu orkiestracji kontenerów Borg firmy Google (patrz „Od Borga do Kubernetes” w rozdziale 1.).

Kubernetes korzysta bezpośrednio z dziesięcioletniego doświadczenia Google związanego z własnym systemem planowania klastrów o nazwie Borg. Powiązanie narzędzia Prometheus z Google jest znacznie luźniejsze, ale czerpie wiele inspiracji z Borgmon, wewnętrznego systemu monitorowania, który Google wymyślił mniej więcej w tym samym czasie, co Borg. Można powiedzieć, że Kubernetes to Borg dla zwykłych śmiertelników, podczas gdy Prometheus to Borgmon dla tychże śmiertelników. Oba są „drugimi systemami”, które próbują iterować dobre części, unikając błędów swoich przodków.

— Björn Rabenstein (SoundCloud)

Więcej na temat narzędzia Prometheus możesz przeczytać na stronie (<https://prometheus.io/>); znajdziesz tam też instrukcje dotyczące instalacji i konfiguracji dla Twojego środowiska.

Sam Prometheus skupia się na gromadzeniu i przechowywaniu metryk, jednak istnieją także inne rozwiązania open source, które umożliwiają tworzenie wykresów, pulpitów oraz alarmów. Grafana (<https://grafana.com/>) jest potężnym i wydajnym silnikiem graficznym obsługującym dane zbierane w określonym czasie (patrz rysunek 16.8).

Projekt Prometheus zawiera narzędzie o nazwie Alertmanager (<https://prometheus.io/docs/alerting/alertmanager/>), które współpracuje z Prometheus. Może również działać niezależnie od niego. Zadaniem narzędzia Alertmanager jest otrzymywanie alertów z różnych źródeł, w tym z serwerów Prometheus, oraz ich przetwarzanie (patrz „Alarmy na podstawie metryk” w tym rozdziale).

Pierwszym krokiem przetwarzania alarmów jest ich deduplikacja. Alertmanager może grupować alarmy, które są powiązane; np. poważna awaria sieci może spowodować setki pojedynczych alarmów, ale Alertmanager może zgromadzić je wszystkie w jedną wiadomość, aby osoby dyżurujące nie były przytłoczone dużą ilością zgłoszeń.



Rysunek 16.8. Pulpit Grafana prezentujący dane narzędzia Prometheus

Na koniec Alertmanager przekieruje przetworzone alerty do odpowiedniej usługi powiadamiania, takiej jak PagerDuty, Slack lub e-mail.

Format metryk Prometheus jest obsługiwany przez bardzo szeroką gamę narzędzi i usług. Obecnie ten standard stanowi założenia OpenMetrics (<https://openmetrics.io/>), projektu Cloud Native Computing Foundation, który ma na celu opracowanie neutralnego standardowego formatu danych metryk. Jednak nie musisz czekać, aż OpenMetrics stanie się rzeczywistością; obecnie prawie każda usługa metryk, w tym Stackdriver, Cloudwatch, Datadog i New Relic, może importować i przetwarzać dane Prometheus.

Google Stackdriver

Chociaż Stackdriver pochodzi z firmy Google, nie ogranicza się do Google Cloud: działa również z AWS. Stackdriver może zbierać metryki, sporządzać wykresy, generować alerty oraz rejestrować dane z różnych źródeł. Będzie automatycznie wykrywać i monitorować zasoby chmurowe, w tym maszyny wirtualne, bazy danych i klastry Kubernetes. Stackdriver przenosi wszystkie te dane do centralnej konsoli webowej, w której można tworzyć niestandardowe pulpity oraz alerty.

Stackdriver potrafi uzyskać metryki operacyjne z tak popularnych narzędzi jak Apache, Nginx, Cassandra i Elasticsearch. Jeśli chcesz dołączyć własne niestandardowe dane z aplikacji, możesz użyć biblioteki Stackdriver do wyeksportowania dowolnych danych.

Jeśli korzystasz z Google Cloud, Stackdriver jest bezpłatny dla wszystkich danych związanych z GCP; w przypadku niestandardowych danych lub danych z innych platform chmurowych płacisz za megabajt danych monitorowania miesięcznie.

Stackdriver (<https://cloud.google.com/monitoring>) nie jest tak elastyczny jak Prometheus ani tak wyrafinowany jak droższe narzędzia, np. Datadog, ale jest świetnym sposobem na rozpoczęcie pomiarów bez konieczności instalowania lub konfigurowania czegokolwiek.

AWS Cloudwatch

Produkt Amazona służący do monitorowania chmury, zwany Cloudwatch, ma właściwości podobne do Stackdriver. Integruje się ze wszystkimi usługami AWS; można eksportować niestandardowe metryki za pomocą SDK Cloudwatch lub aplikacji konsolowej.

Cloudwatch posiada darmową warstwę, która pozwala zbierać *podstawowe* metryki (takie jak wykorzystanie procesora dla maszyn wirtualnych) w pięciominutowych odstępach, pewną liczbę pulpitów i alarmów itd. Poza powyższymi, płacisz za metrykę, pulpit lub alarm, możesz także płacić za metryki pracujące w wyższym interwale (interwały jednominutowe) dla poszczególnych instancji.

Podobnie jak Stackdriver, Cloudwatch (<https://aws.amazon.com/cloudwatch>) jest prosty, ale skuteczny. Jeśli Twoją podstawową infrastrukturą chmurową jest AWS, Cloudwatch jest dobrym miejscem do rozpoczęcia pracy z metrykami. Dla małych wdrożeń może zawierać wszystko, czego potrzebujesz.

Azure Monitor

Monitor (<https://docs.microsoft.com/en-us/azure/azure-monitor/overview>) to oferowany przez Azure odpowiednik Google Stackdriver lub AWS Cloudwatch. Gromadzi dzienniki i dane metryk ze wszystkich zasobów platformy Azure, w tym klastrów Kubernetes. Umożliwia również wizualizację metryk oraz generowanie alarmów na podstawie tych danych.

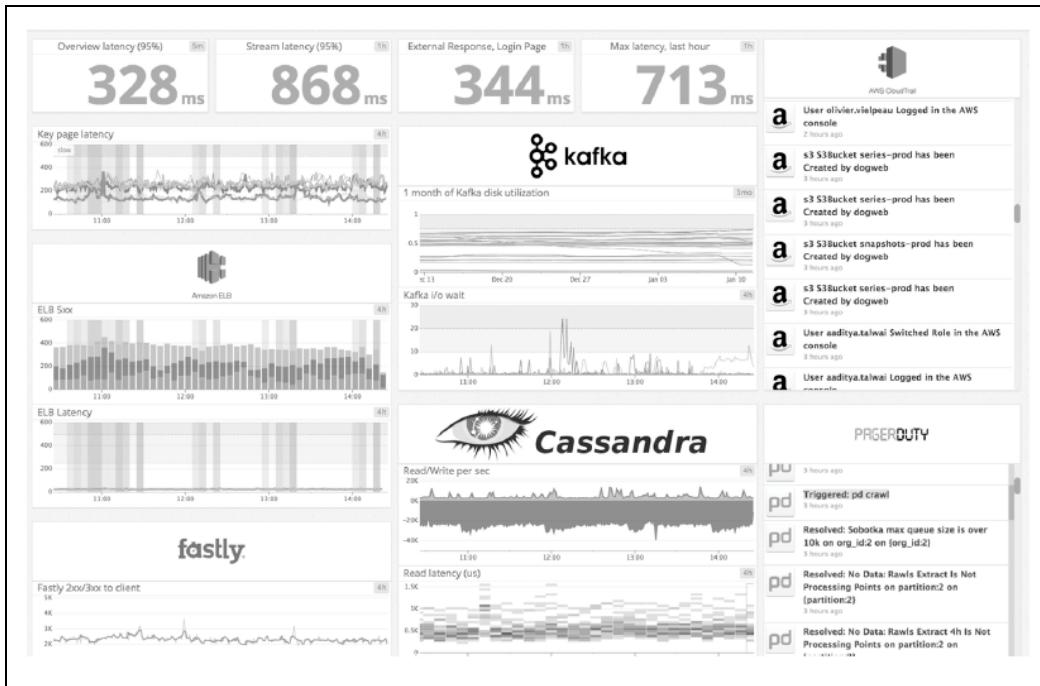
Datadog

W porównaniu z wbudowanymi narzędziami dostawców usług chmurowych, takimi jak Stackdriver i Cloudwatch, Datadog (<https://www.datadoghq.com/>) jest bardzo zaawansowaną i potężną platformą służącą do monitorowania i analizy. Oferuje integracje z ponad 250 platformami i usługami, w tym wszystkimi usługami chmurowymi głównych dostawców, oraz z popularnym oprogramowaniem, takim jak Jenkins, Varnish, Puppet, Consul i MySQL.

Datadog oferuje również komponent monitorowania wydajności aplikacji (APM), który pomaga monitorować i analizować wydajność aplikacji. Bez względu na to, czy korzystasz z Go, Java, Ruby, czy jakiekolwiek innej platformy oprogramowania, Datadog może zbierać dane, dzienniki, śledzić Twoje oprogramowanie oraz odpowiadać na pytania, takie jak poniższe.

- Jak wygląda użyteczność dla konkretnego, indywidualnego użytkownika mojej usługi?
- Których 10 klientów dostaje najwolniejsze odpowiedzi w danym punkcie końcowym?
- Która z moich różnych usług rozproszonych przyczynia się do ogólnego opóźnienia w żądaniach?

Wraz ze zwykłymi pulpitami (patrz rysunek 16.9) i funkcjami alarmowania (automatyzacja za pośrednictwem interfejsu API Datadog i bibliotek, w tym Terraform) Datadog zapewnia również funkcje, takie jak wykrywanie anomalii oparte na uczeniu maszynowym.



Rysunek 16.9. Pulpit Datadog

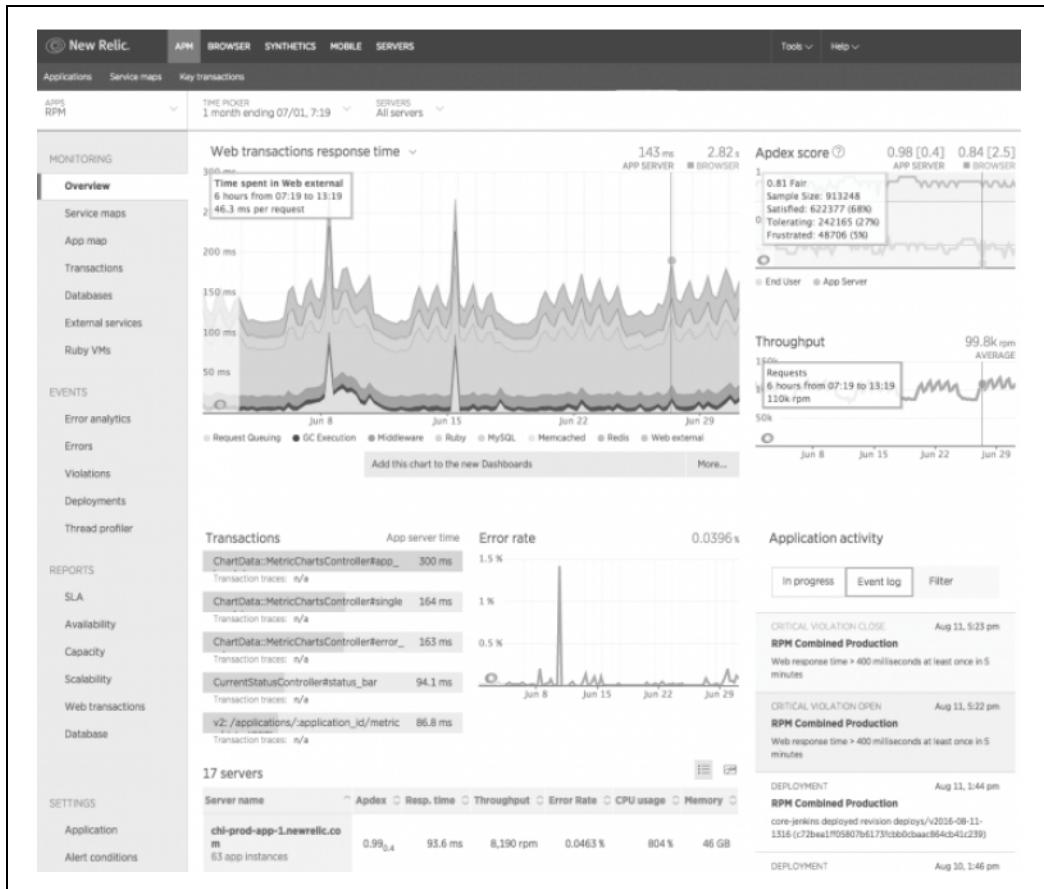
Jak można się spodziewać, Datadog jest również jedną z droższych dostępnych usług monitorowania. Jeśli jednak poważnie myślisz o obserwowalności i masz złożoną infrastrukturę z aplikacjami o klu-czowym znaczeniu dla wydajności, koszt może być tego wart.

Więcej o Datadog możesz przeczytać na stronie internetowej (<https://www.datadoghq.com/>).

New Relic

New Relic jest bardzo dobrze ugruntowaną i szeroko stosowaną platformą pomiarową skoncen-trowaną na monitorowaniu wydajności aplikacji (APM). Jego główną siłą jest diagnozowanie pro-blemów z wydajnością i wykrywanie wąskich gardeł w aplikacjach i systemach rozproszonych (patrz rysunek 16.10). Oferuje jednak również metryki związane z infrastrukturą oraz monitorowanie, gene-rowanie alarmów, analizę oprogramowania i wszystko, czego jeszcze oczekujesz.

Podobnie jak w przypadku każdej usługi klasy korporacyjnej, spodziewaj się, że za monitorowanie za pomocą New Relic dużo zapłacisz, szczególnie na dużą skalę. Jeśli szukasz platformy korpora-cyjnej klasy premium, prawdopodobnie zainteresujesz się New Relic (nieco bardziej skoncentro-wanym na aplikacjach) lub Datadog (nieco bardziej skoncentrowanym na infrastrukturze). Oba oferują również dobrą infrastrukturę z wykorzystaniem kodu; np. przy użyciu oficjalnych dostawców Terraform można tworzyć pulpity monitorowania i alarmy.



Rysunek 16.10. Pulpit APM New Relic

Podsumowanie

Zmierz dwa razy, tnij raz to ulubione powiedzenie wielu inżynierów. W świecie cloud native bez odpowiednich danych i danych obserwacyjnych bardzo trudno stwierdzić, co się dzieje. Z drugiej strony, gdy obserwujemy dużo metryk, zbyt wiele informacji może być tak samo bezużytecznych, jak zbyt mało.

Przede wszystkim sztuką jest zebranie odpowiednich danych, przetworzenie ich we właściwy sposób, wykorzystanie do odpowiedzi na właściwe pytania, wizualizacja w odpowiedni sposób i zastosowanie do ostrzeżenia właściwych ludzi we właściwym czasie.

Po przeczytaniu tego rozdziału musisz pamiętać o niżej spisanych zasadach.

- Skoncentruj się na kluczowych metrykach dla każdej usługi: żądaniach, błędach i czasie trwania (RED) i dla każdego zasobu: wykorzystanie, nasycenie i błędy (USE).
- Poinstruj swoje aplikacje tak, aby wyświetlały niestandardowe metryki zarówno dla wewnętrznej obserwowalności, jak i dla biznesowych wskaźników KPI.

- Przydatne metryki Kubernetes, na poziomie klastra, obejmują liczbę węzłów, liczbę Podów na węzel i zużycie zasobów przez węzły.
- Na poziomie obiektu Deployment śledź te obiekty Deployment i repliki, szczególnie niedostępne repliki, które mogą wskazywać na problem z pojemnością.
- Na poziomie kontenera śledź zużycie zasobów na kontener, stany żywotności lub gotowości, restarty, ruch sieciowy i błędy sieciowe.
- Zbuduj pulpit dla każdej usługi, korzystając ze standardowego layoutu i głównego radiatorka informacji, który zgłasza istotne parametry całego systemu.
- Jeśli ostrzegasz o metrykach, alerty powinny być pilne, ważne i wykonalne. Alarmowy chaos powoduje zmęczenie i obniża morale zespołu.
- Śledź i przeglądaj liczbę pilnych zgłoszeń, które otrzymuje Twój zespół, szczególnie w trakcie snu i weekendu.
- W świecie cloud native standardowym rozwiązaniem związanym z metrykami jest Prometheus. Prawie wszystkie narzędzia stosują format danych Prometheus.
- Usługi zarządzane związane z metrykami to także Google Stackdriver, Amazon Cloudwatch, Datadog i New Relic.

Postowie

Nie ma nic trudniejszego oraz bardziej niebezpiecznego niż przewodzenie we wprowadzaniu nowego porządku rzeczy.

— Niccolò Machiavelli

To była niezła jazda. Omówiliśmy w tej książce wiele tematów. Jednak wszystko, co zawarliśmy, zostało określone przez jedną prostą zasadę: uważamy, że musisz to wiedzieć, jeśli korzystasz z aplikacji Kubernetes w środowisku produkcyjnym.

Mówią, że ekspertem jest ten, kto jest o jedną stronę do przodu. Jest całkiem prawdopodobne, że jeśli czytasz tę książkę, przynajmniej na początku będziesz ekspertem Kubernetes w swojej organizacji. Mamy nadzieję, że ta książka okaże się przydatna, ale jest to tylko punkt wyjścia.

Co dalej?

Poniższe zasoby mogą okazać się przydatne zarówno w celu uzyskania dodatkowych informacji na temat Kubernetes i cloud native, jak i po to, aby być na bieżąco z najnowszymi wiadomościami i zmianami.

<http://slack.k8s.io/>

Oficjalna grupa Kubernetes na Slacku. To dobre miejsce do zadawania pytań i czatowania z innymi użytkownikami.

<https://discuss.kubernetes.io/>

Publiczne forum do dyskusji na temat wszystkich rzeczy związanych z Kubernetes.

<https://kubernetespodcast.com/>

Cotygodniowy podcast prowadzony przez Google. Odcinki zwykle trwają około 20 minut i obejmują cotygodniowe wiadomości. Często przeprowadzane są wywiady z kimś zaangażowanym w projekt Kubernetes.

<https://github.com/vmware-tanzu/tgik>

TGIK8s to cotygodniowy strumień wideo na żywo, zapoczątkowany przez Joego Beda w Heptio. Format zwykle obejmuje około godziny demonstracji na żywo w ekosystemie Kubernetes. Wszystkie filmy są archiwizowane i dostępne do oglądania na żądanie.

Nie można pominąć faktu, że mamy blog (<https://cloudnativedevelopsblog.com/>) powiązany z tą książką. Od czasu do czasu sprawdzaj najnowsze wiadomości, aktualizacje oraz posty.

Oto niektóre biuletyny e-mailowe, które możesz zasubskrybować. Dotyczą takich tematów jak tworzenie oprogramowania, bezpieczeństwo, DevOps i Kubernetes.

- KubeWeekly (<https://twitter.com/kubeweekly>) (prowadzony przez CNCF).
- SRE Weekly (<https://sreweekly.com/>).
- DevOps Weekly (<https://www.devopsweekly.com/>).
- DevOps'ish (<https://devopsish.com/>).
- Security Newsletter (<https://securitynewsletter.co/>).

Witamy na pokładzie

Nic się nie nauczysz, jeżeli wszystko przebiega bezproblemowo.

— Elizabeth Bibesco

Podstawowym priorytetem podczas nauki Kubernetes powinno być zdobywanie dużego doświadczenia, takiego jak to tylko możliwe, i uczenie się jak najwięcej od innych. Nikt z nas nie wie wszystkiego, ale każdy coś wie. Razem możemy po prostu coś wymyślić.

I nie bój się eksperymentować. Utwórz własną aplikację demonstracyjną lub pożycz naszą i przewicz te rzeczy, których prawdopodobnie będziesz potrzebować w produkcji. Jeśli wszystko, co robisz, działa idealnie, oznacza to, że nie eksperymentujesz wystarczająco. Prawdziwe uczenie się bierze się z niepowodzeń, prób ustalenia, co jest nie tak, i prób naprawienia. Im więcej popełniasz błędów, tym więcej się uczysz.

Jeśli *dowiedzieliśmy się* czegoś o Kubernetes, to tylko dlatego, że popełniliśmy wiele błędów. Mamy nadzieję, że Ty też. Baw się dobrze!

O Autorach

John Arundel jest konsultantem z 30-letnim doświadczeniem w branży komputerowej. Jest autorem kilku książek i współpracuje z wieloma firmami na całym świecie, doradzając w zakresie infrastruktury cloud native oraz aplikacji Kubernetes. W czasie wolnym od pracy jest zapalonym surferem, fanem broni krótkiej i długiej oraz zdecydowanie bezkonkurencyjnym pianistą. Mieszka w bajkowym domku w Kornwalii w Anglii.

Justin Domingus jest inżynierem operacyjnym pracującym w środowiskach DevOps z Kubernetes i technologiami chmurowymi. Lubi spędzać czas na świeżym powietrzu, uczyć się nowych rzeczy, przepada za kawą, krabami i komputerami. Mieszka w Seattle w stanie Waszyngton ze wspaniałym kotem oraz jeszcze wspanialszą żoną i najlepszą przyjaciółką Adrienne.

Kolofon

Zwierzęciem na okładce książki *Cloud Native DevOps z Kubernetes* jest fregata orla (*Fregata aquila*). To ptak morski występujący tylko na Wyspie Wniebowstąpienia i pobliskiej Boatswain Bird Island na południowym Oceanie Atlantyckim, mniej więcej między Angolą a Brazylią.

Wyspa, którą zamieskuje tytułowy ptak, została nazwana od dnia, w którym została odkryta — Dnia Wniebowstąpienia według kalendarza chrześcijańskiego.

Z rozpiętością skrzydeł ponad 2 metry, ale wagą około 1,25 kg, fregata orla szybuje bez wysiłku nad oceanem, lwiąc ryby pływające blisko powierzchni, zwłaszcza ryby latające. Czasami żywi się kałamarnicami, małymi żółwiami i zdobyczą skradzioną innym ptakom. Jej pióra są czarne i błyszczące, z nutami zieleni i fioletu. Samiec wyróżnia się jasnoczerwonym pogardlem, które nadyma, gdy szuka partnerki. Samica ma nieco matowe pióra, z brązowymi, a czasem białymi plamami pod spodem. Podobnie jak inne fregaty, ma haczykowaty dziób, rozwidlony ogon i ostro zakończone skrzydła.

Fregata orla rozmnaża się na klifach lub skalistych piargach swojego siedliska na wyspie. Zamiast budować gniazdo, wykopuje w ziemi zagłębienie i chroni je piórami, kamyczkami i kośćmi. Samica składa jedno jajko, a oboje rodzice opiekują się pisklęciem przez sześć lub siedem miesięcy, zanim w końcu nauczy się latać. Ponieważ sukces lęgowy jest niski, a siedlisko ograniczone, gatunek jest zwykle klasyfikowany jako wrażliwy.

Wyspa Wniebowstąpienia została po raz pierwszy zasiedlona przez Brytyjczyków do celów wojskowych na początku XIX wieku. Dziś na wyspie znajdują się stacje śledzące NASA i Europejskiej Agencji Kosmicznej, stacja przekaźnikowa dla BBC World Service oraz jedna z czterech anten GPS na świecie. Przez większość XIX i XX wieku fregaty ograniczały się do rozmnażania się na małej, skalistej wyspie Boatswain Bird Island koło wschodniego brzegu Wyspy Wniebowstąpienia, ponieważ zdzieciły koty zabijaly ich pisklęta. W 2002 roku Królewskie Towarzystwo Ochrony Ptaków rozpoczęło kampanię mającą na celu wyeliminowanie kotów z wyspy, a kilka lat później fregaty ponownie zaczęły budować gniazda na Wyspie Wniebowstąpienia.

Wiele zwierząt na okładkach O'Reilly jest zagrożonych; wszystkie są ważne dla świata. Aby dowiedzieć się więcej o tym, jak możesz pomóc, wejdź na stronę animals.oreilly.com.

Ilustracja na okładce jest autorstwa Karen Montgomery na podstawie czarno-białej rycinie z książki Cassell's Natural History.

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
<http://program-partnerski.helion.pl>

GRUPA
Helion 



KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie z dostępem do nowoczesnych narzędzi - videokursów, e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL