O'REILLY®

Kubernetes

Tworzenie natywnych aplikacji działających w chmurze

Michael Hausenblas
Stefan Schimanski

Helion

Tytuł oryginału: Programming Kubernetes: Developing Cloud-Native Applications

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-6405-9

© 2020 Helion SA

Authorized Polish translation of the English edition of Programming Kubernetes
ISBN 9781492047100 © 2019 Michael Hausenblas and Stefan Schimanski

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 właścicieli.

Autor 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. Autor 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, zajrzyj pod adres
http://helion.pl/user/opinie/kubert
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa **11**

1. Wprowadzenie **15**
Czym jest programowanie dla Kubernetesa? 15
Przykład wprowadzający 17
Wzorce rozszerzania 18
Kontrolery i operatory 19
Pętla sterowania 19
Zdarzenia 20
Wyzwalacze sterowane zmianami i sterowane poziomem 22
Modyfikowanie świata zewnętrznego lub obiektów w klastrze 24
Współbieżność optymistyczna 27
Operatory 29
Podsumowanie 30

2. Podstawy API Kubernetesa **33**
Serwer API 33
Interfejs HTTP serwera API 34
Terminologia związana z API 35
Wersjonowanie API w Kubernetesie 38
Deklaratywne zarządzanie stanem 39
Używanie API w wierszu poleceń 39
W jaki sposób serwer API przetwarza żądania? 43
Podsumowanie 46

3. Podstawy klienta client-go **47**
Repozytoria 47
Biblioteka klienta 47
Typy w API Kubernetesa 49
Repozytorium API Machinery 50

Kup książkę Poleć książkę

Tworzenie i używanie klientów 50
Wersjonowanie i kompatybilność 52
Wersje API i gwarancje kompatybilności 55
Obiekty Kubernetesa w Go 56
TypeMeta 57
ObjectMeta 60
Sekcje spec i status 60
Zbiory klientów 61
Podzasoby status — UpdateStatus 63
Wyświetlanie i usuwanie obiektów 63
Czujki 63
Rozszerzanie klientów 64
Opcje klientów 65
Informatory i buforowanie 66
Kolejka zadań 70
Repozytorium API Machinery — szczegóły 72
Rodzaje 72
Zasoby 72
Odwzorowania REST 73
Schemat 74
Vendoring 75
glide 76
dep 76
Moduły języka Go 77
Podsumowanie 78

4. Używanie niestandardowych zasobów 79
Wykrywanie informacji 81
Definicje typów 82
Zaawansowane mechanizmy niestandardowych zasobów 84
Sprawdzanie poprawności niestandardowych zasobów 84
Kategorie i krótkie nazwy 86
Wyświetlane kolumny 88
Podzasoby 89
Niestandardowe zasoby z perspektywy programisty 93
Klient dynamiczny 93
Klienty typizowane 95
Klient controller-runtime z narzędzi Operator SDK i Kubebuilder 99
Podsumowanie 101

Kup książkę Poleć książkę

5. Automatyzowanie generowania kodu 103
Po co stosować generatory kodu? 103
Wywoływanie generatorów 103
Kontrolowanie generatorów za pomocą znaczników 105
Znaczniki globalne 106
Znaczniki lokalne 107
Znaczniki dla generatora deepcopy-gen 108
runtime.Object i DeepCopyObject 108
Znaczniki dla generatora client-gen 109
Generatory informer-gen i lister-gen 111
Podsumowanie 111

6. Narzędzia służące do tworzenia operatorów 113
Czynności wstępne 113
Wzorowanie się na projekcie sample-controller 114
Przygotowania 114
Logika biznesowa 115
Kubebuilder 121
Przygotowania 122
Logika biznesowa 126
Operator SDK 130
Przygotowania 131
Logika biznesowa 132
Inne podejścia 135
Wnioski i przyszłe kierunki rozwoju 136
Podsumowanie 136

7. Udostępnianie kontrolerów i operatorów 137
Zarządzanie cyklem życia i pakowanie 137
Pakowanie — trudności 137
Helm 138
Kustomize 140
Inne techniki pakowania kodu 142
Najlepsze praktyki z obszaru pakowania kodu 143
Zarządzanie cyklem życia 143
Instalacje gotowe do użytku w środowisku produkcyjnym 144
Odpowiednie uprawnienia 144
Zautomatyzowany proces budowania i testowania 147
Niestandardowe kontrolery i obserwowalność 148
Podsumowanie 151

Kup książkę Poleć książkę

8. Niestandardowe serwery API 153
Scenariusze stosowania niestandardowych serwerów API 153
Przykład — pizzeria 155
Architektura — agregowanie 156
Usługi API 157
Wewnętrzna struktura niestandardowego serwera API 160
Delegowane uwierzytelnianie i obsługa zaufania 161
Delegowana autoryzacja 162
Pisanie niestandardowych serwerów API 164
Wzorzec opcji i konfiguracji oraz szablonowy kod potrzebny do uruchomienia serwera 165
Pierwsze uruchomienie 171
Typy wewnętrzne i konwersja 172
Pisanie typów API 175
Konwersje 176
Ustawianie wartości domyślnych 179
Testowanie konwersji powrotnych 181
Sprawdzanie poprawności 183
Rejestr i strategia 185
Instalowanie API 189
Kontrola dostępu 192
Instalowanie niestandardowych serwerów API 201
Manifesty instalacji 202
Konfigurowanie systemu RBAC 204
Uruchamianie niestandardowego serwera API bez zabezpieczeń 205
Certyfikaty i zaufanie 207
Współdzielenie systemu etcd 209
Podsumowanie 211

9. Zaawansowane zasoby niestandardowe 213
Wersjonowanie niestandardowych zasobów 213
Poprawianie kodu do obsługi pizzerii 214
Architektura webhooków konwersji 216
Implementacja webhooka konwersji 220
Przygotowywanie serwera HTTPS 220
Instalowanie webhooka konwersji 226
Konwersja w praktyce 227
Webhooki kontroli dostępu 229
Wymogi związane z kontrolą dostępu w przykładzie 230
Architektura webhooków kontroli dostępu 231
Rejestrowanie webhooków kontroli dostępu 233

Kup książkę Poleć książkę

Implementowanie webhooka kontroli dostępu 234
Webhook kontroli dostępu w praktyce 239
Schematy strukturalne i przyszłość definicji CRD 240
Schematy strukturalne 240
Okrajanie a zachowywanie nieznanych pól 242
Sterowanie okrajaniem 243
IntOrString i RawExtension 244
Wartości domyślne 244
Podsumowanie 246

A Materiały 247

Kup książkę Poleć książkę

Kup książkę Poleć książkę

ROZDZIAŁ 1.

Wprowadzenie

Programowanie w Kubernetesie może oznaczać dla różnych osób co innego. W tym rozdziale najpierw określimy zakres i tematykę tej książki. Ponadto przedstawimy założenia dotyczące używanego środowiska i wyjaśnimy, czego będziesz potrzebować, aby optymalnie wykorzystać tę książkę. Zdefiniujemy, co rozumiemy przez programowanie dla Kubernetesa, czym są natywne aplikacje dla Kubernetesa, a także — posługując się konkretnym przykładem — jakie są ich cechy. Omówimy podstawy kontrolerów i operatorów oraz wyjaśnimy, jak działa sterowana zdarzeniami warstwa kontroli w Kubernetesie. Gotowy? Zaczynajmy.

Czym jest programowanie dla Kubernetesa?

Zakładamy, że masz dostęp do działającego klastra z Kubernetesem, np. do usługi Amazon EKS, Microsoft AKS, Google GKE lub jednego z rozwiązań z rodziny OpenShift.

Sporo czasu przeznaczysz na programowanie w środowisku *lokalnym* — na swoim laptopie lub komputerze stacjonarnym. W takim modelu klaster Kubernetesa działa lokalnie, a nie w chmurze lub w centrum danych. Gdy programujesz lokalnie, możesz korzystać z różnych narzędzi. W zależności od używanego systemu operacyjnego i innych preferencji możesz zdecydować się na jedno (lub na kilka) z wymienionych narzędzi, aby uruchamiać Kubernetesa lokalnie: kind (*https://kind.sigs.k8s.io*), k3d (*http://bit.ly/2Ja1LaH*) lub Docker Desktop (*https://dockr.ly/2PTJVLL*)[1].

Ponadto zakładamy, że programujesz w języku Go i masz doświadczenie w pracy z nim lub przynajmniej podstawową wiedzę na jego temat. Jeśli nie spełniasz któregoś z wymienionych warunków, teraz jest dobry czas na to, aby się doszkolić. Jeśli chodzi o język Go, polecamy książki *The Go Programming Language* (*https://www.gopl.io*) autorstwa Alana A.A. Donovana i Briana W. Kernighana (Addison-Wesley) oraz *Concurrency in Go* (*http://bit.ly/2tdCt5j*) autorstwa Katherine Cox-Buday (O'Reilly). Aby zapoznać się z Kubernetesem, sprawdź jedną lub kilka z następujących pozycji:

[1] Więcej na ten temat dowiesz się z tekstu Megan O Keefe: „A Kubernetes Developer Workflow for MacOS" (*http://bit.ly/2WXfzu1*), Medium, 24 stycznia 2019, a także z artykułu z bloga Aleksa Ellisa: „Be KinD to yourself" (*http://bit.ly/2XkK9C1*), 14 grudnia 2018.

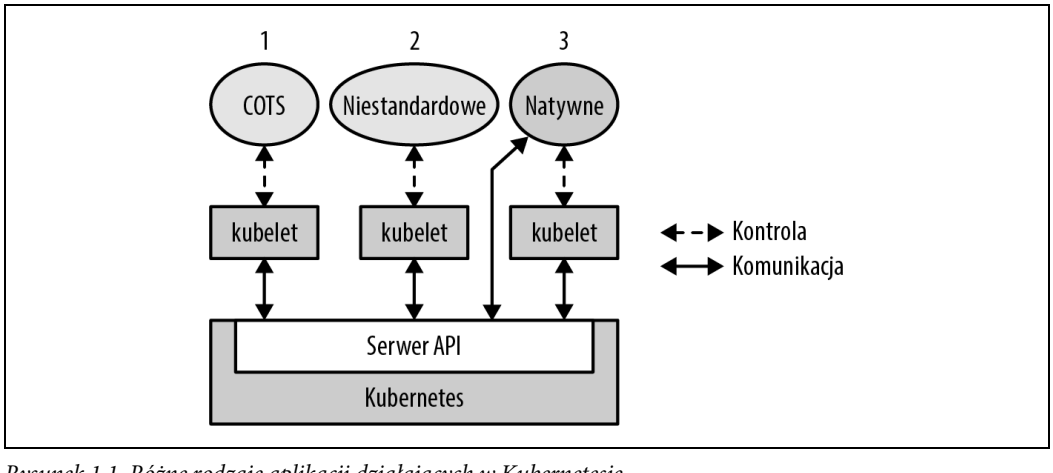
Kup książkę Poleć książkę

- *Kubernetes in Action*, Marko Lukša (Manning);
- *Kubernetes: Up and Running*, wydanie drugie (*https://oreil.ly/2SaANU4*) Kelsey Hightower i inni (O'Reilly);
- *Cloud Native DevOps with Kubernetes* (*https://oreil.ly/2BaE1iq*), John Arundel i Justin Domingus (O'Reilly);
- *Managing Kubernetes* (*https://oreil.ly/2wtHcAm*), Brendan Burns i Craig Tracey (O'Reilly);
- *Kubernetes Cookbook* (*http://bit.ly/2FTgJzk*), Sébastien Goasguen i Michael Hausenblas (O'Reilly).

Dlaczego skupiamy się na programowaniu dla Kubernetesa w języku Go? Przydatna może być tu analogia: Unix został napisany w języku C, a jeśli ktoś chciał pisać aplikacje lub narzędzia dla Uniksa, domyślnie używał właśnie C. Ponadto gdy ktoś chciał rozszerzać i dostosowywać Uniksa do własnych potrzeb, to nawet jeśli korzystał z języka innego niż C, musiał przynajmniej umieć czytać kod w C.

Kubernetes i wiele powiązanych technologii natywnych działających w chmurze (od środowisk uruchomieniowych dla kontenerów po narzędzia do monitorowania takie jak Prometheus) jest napisanych w Go. Uważamy, że większość aplikacji natywnych będzie oparta na języku Go, dlatego skupiamy się na nim w tej książce. Jeśli wolisz korzystać z innych języków, obserwuj repozytorium kubernetes-client w serwisie GitHub (*http://bit.ly/2xfSrfT*). W przyszłości może się tam znaleźć klient napisany w Twoim ulubionym języku programowania.

W tej książce „programowanie dla Kubernetesa" oznacza pisanie natywnych aplikacji dla platformy Kubernetes, które bezpośrednio komunikują się z serwerem API, kierując zapytania o stan zasobów i/lub aktualizując go. Nie mamy tu na myśli uruchamiania gotowych aplikacji takich jak WordPress, Rocket Chat lub Twój ulubiony system CRM dla przedsiębiorstw (takie narzędzia czasem są nazywane aplikacjami COTS — ang. *commercially available off-the-shelf*). Ponadto w rozdziale 7. skupiamy się nie na kwestiach operacyjnych, ale głównie na etapie pisania i testowania kodu. Tak więc w skrócie można stwierdzić, że ta książka dotyczy pisania natywnych aplikacji działających w chmurze. Rysunek 1.1 może pomóc Ci lepiej to zrozumieć.



Rysunek 1.1. Różne rodzaje aplikacji działających w Kubernetesie

Kup książkę Poleć książkę

Widać tu, że dostępne są różne rodzaje rozwiązań:

1. Możesz użyć aplikacji COTS, np. Rocket Chat, i uruchomić ją w Kubernetesie. Sama aplikacja nie wie, że działa w Kubernetesie, i zwykle nie musi tego wiedzieć. To Kubernetes kontroluje cykl życia aplikacji — znajduje węzeł, gdzie aplikacja jest uruchamiana, pobiera obraz, uruchamia kontenery, sprawdza stan, montuje woluminy itd.
2. Możesz uruchomić w Kubernetesie niestandardową aplikację, którą napisałeś od podstaw. Rozwijając ją, mogłeś myśleć o uruchamianiu jej w Kubernetesie, ale nie jest to konieczne. Sytuacja wygląda tu podobnie jak dla aplikacji COTS.
3. W tej książce skupiamy się na natywnych aplikacjach dla chmury lub Kubernetesa, które są w pełni świadome tego, że działają w Kubernetesie, a także wykorzystują w jakimś stopniu API oraz zasoby Kubernetesa.

Warto ponieść koszty programowania z użyciem API Kubernetesa. Po pierwsze zyskujesz przenośność, ponieważ aplikacja może działać w dowolnym środowisku (od lokalnych instalacji po chmury oferowane przez publicznych dostawców). Po drugie możesz korzystać z przejrzystych deklaratywnych mechanizmów dostępnych w Kubernetesie.

Przejdźmy teraz do konkretnego przykładu.

Przykład wprowadzający

Aby zademonstrować możliwości natywnych aplikacji dla Kubernetesa, załóżmy, że chcesz zaimplementować narzędzie `at`, pozwalające zaplanować wykonanie polecenia w określonym czasie (*http://bit.ly/2L4VqzU*).

Nowe narzędzie nazwiemy `cnat` (*http://bit.ly/2RpHhON*) (od ang. *cloud-native at*). Będzie ono działać w następujący sposób: załóżmy, że chcesz wykonać polecenie `echo "Natywne aplikacje dla Kubernetesa są super!"` o 2:00 w nocy 3 lipca 2020 r. W narzędziu `cnat` można to będzie zrobić tak:

```
$ cat cnat-rocks-example.yaml
apiVersion: cnat.programming-kubernetes.info/v1alpha1
kind: At
metadata:
  name: cnrex
spec:
  schedule: "2019-07-03T02:00:00Z"
  containers:
  - name: shell
    image: centos:7
    command:
    - "bin/bash"
    - "-c"
    - echo "Natywne aplikacje dla Kubernetesa są super!"

$ kubectl apply -f cnat-rocks-example.yaml
cnat.programming-kubernetes.info/cnrex created
```


Kup książkę Poleć książkę

Na zapleczu używane są tu następujące komponenty:

- Niestandardowy zasób `at.programming-kubernetes.info/cnrex` reprezentujący harmonogram.
- Kontroler wykonujący zaplanowane polecenie w odpowiednim czasie.

Ponadto przydatna byłaby wtyczka `kubectl` zapewniająca interfejs CLI i umożliwiająca prostą obsługę poleceń takich jak `kubectl at "02:00 Jul 3" echo "Natywne aplikacje dla Kubernetesa są super!"`. Nie napiszemy jej w tej książce, jednak potrzebne instrukcje znajdziesz w dokumentacji Kubernetesa (*http://bit.ly/2J1dPuN*).

W książce będziemy używać tego przykładu do omawiania aspektów Kubernetesa, jego wewnętrznych mechanizmów i rozszerzania go.

W bardziej zaawansowanych przykładach z rozdziałów 8. i 9. będziemy symulować działanie pizzerii z reprezentującymi pizzę i dodatki obiektami w klastrze. Szczegółowe informacje znajdziesz w podrozdziale „Przykład — pizzeria".

Wzorce rozszerzania

Kubernetes to dający duże możliwości i z natury rozszerzalny system. Istnieje wiele sposobów na modyfikowanie i/lub rozszerzanie Kubernetesa, na przykład używanie plików konfiguracyjnych i opcji (*http://bit.ly/2KteqbA*) dla komponentów kontrolnych takich jak `kubelet` lub serwer API Kubernetesa, a także stosowanie wielu zdefiniowanych punktów rozszerzeń, którymi są:

- Tak zwani dostawcy chmury (*http://bit.ly/2FpHInw*), w przeszłości będący wewnętrznym elementem menedżera kontrolera. W wersji 1.11 Kubernetes umożliwia rozwój zewnętrznych dostawców, udostępniając niestandardowy proces `cloud-controller-manager` do integracji rozwiązań z chmurą (*http://bit.ly/2WWlcxk*). Dostawca chmury umożliwia używanie narzędzi specyficznych dla danego dostawcy, np. równoważników obciążenia lub maszyn wirtualnych.
- Binarne wtyczki dla narzędzia `kubelet` do obsługi: sieci (*http://bit.ly/2L1tPzm*), urządzeń takich jak karty graficzne (*http://bit.ly/2XthLgM*), pamięci (*http://bit.ly/2x7Unaa*) i środowisk uruchomieniowych dla kontenerów (*http://bit.ly/2Zzh1Eq*).
- Binarne wtyczki dla narzędzia `kubectl` (*http://bit.ly/2FmH7mu*).
- Rozszerzenia do obsługi dostępu w serwerze API, np. dynamiczna kontrola dostępu z użyciem webhooków (*http://bit.ly/2DwR2Y3*; zob. rozdział 9.).
- Niestandardowe zasoby (zob. rozdział 4.) i niestandardowe kontrolery; zob. następny podrozdział.
- Niestandardowe serwery API (zob. rozdział 8.).
- Rozszerzenia programu szeregującego, np. używanie webhooków (*http://bit.ly/2xcg4FL*) do implementowania własnych mechanizmów szeregowania.
- Uwierzytelnianie (*http://bit.ly/2Oh6DPS*) z użyciem webhooków.

W tej książce koncentrujemy się na niestandardowych zasobach, kontrolerach, webhookach i niestandardowych serwerach API, a także na wzorcach rozszerzania Kubernetesa (*http://bit.ly/2L2SJ1C*). Jeśli interesują Cię inne punkty rozszerzeń, np. wtyczki związane z pamięcią lub siecią, zapoznaj się z oficjalną dokumentacją (*http://bit.ly/2Y0L1J9*).

Kup książkę Poleć książkę

Po zapoznaniu się z podstawami wzorców rozszerzeń Kubernetesa i zakresem tej książki pora przejść do istoty mechanizmów kontroli Kubernetesa i ich rozszerzania.

Kontrolery i operatory

W tym podrozdziale poznasz kontrolery i operatory Kubernetesa oraz dowiesz się, jak działają.

Zgodnie ze słowniczkiem pojęć związanych z Kubernetesem (*http://bit.ly/2IWGlxz*) *kontroler* zawiera pętlę sterowania. Obserwuje współdzielony stan klastra za pomocą serwera API i wprowadza zmiany, próbując przejść od bieżącego stanu do stanu docelowego.

Przed przejściem do mechanizmów kontrolerów warto zdefiniować terminologię:

- Kontrolery mogą operować podstawowymi zasobami, takimi jak instalacje lub usługi, które zwykle są częścią menedżera kontrolera Kubernetesa (*http://bit.ly/2WUAEVy*) w warstwie kontroli. Mogą też obserwować niestandardowe zasoby zdefiniowane przez użytkownika i operować nimi.
- Operatory to kontrolery obejmujące wiedzę operacyjną, np. z zakresu zarządzania cyklem życia aplikacji, a także niestandardowe zasoby (zdefiniowane w rozdziale 4.).

Naturalne jest to, że ponieważ drugie z tych pojęć jest oparte na pierwszym, najpierw omówimy kontrolery, a następnie opiszemy bardziej wyspecjalizowane operatory.

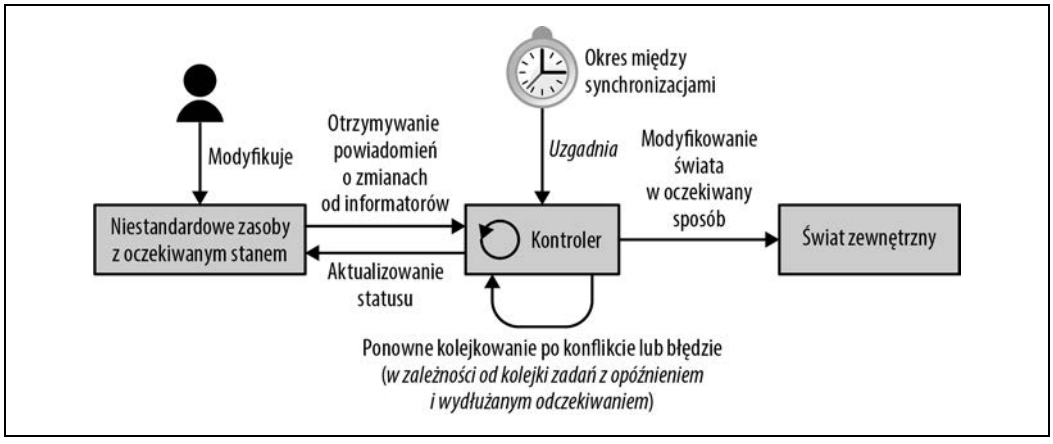
Pętla sterowania

Pętla sterowania na ogólnym poziomie wygląda tak:

1. Wczytywanie stanu zasobów, najlepiej w modelu sterowanym zdarzeniami (z użyciem czujek — ang. *watches* — co opisujemy w rozdziale 3.). Szczegółowe informacje znajdziesz w punktach „Zdarzenia” i „Wyzwalacze sterowane zmianami i sterowane poziomem”.
2. Zmiana stanu obiektów w klastrze lub poza klastrem (np. uruchomienie poda, utworzenie sieciowego punktu końcowego lub skierowanie zapytania do API chmury). Zobacz punkt „Modyfikowanie świata zewnętrznego lub obiektów w klastrze”.
3. Aktualizowanie za pomocą serwera API statusu zasobu z kroku 1. w systemie `etcd`. Zobacz punkt „Współbieżność optymistyczna”.
4. Powtórzenie cyklu (powrót do kroku 1.).

Niezależnie od tego, jak skomplikowany lub prosty jest kontroler, trzy wymienione tu kroki (wczytywanie stanu zasobów > modyfikowanie świata > aktualizowanie stanu zasobów) pozostają takie same. Przyjrzyjmy się bliżej temu, jak te kroki są realizowane w kontrolerze w Kubernetesie. Pętla sterowania jest pokazana na rysunku 1.2, gdzie widoczne są typowe komponenty. Pośrodku znajduje się główna pętla kontrolera. Ta główna pętla działa nieustannie w procesie kontrolera. Z kolei proces działa zwykle w podzie w klastrze.

Kup książkę Poleć książkę



Rysunek 1.2. Pętla sterowania w Kubernetesie

Jeśli chodzi o architekturę, kontroler zwykle używa następujących struktur danych (opisanych szczegółowo w rozdziale 3.).

Informatory
: Informatory obserwują oczekiwany stan zasobów w skalowalny i stabilny sposób. Obsługują też mechanizm ponownej synchronizacji (zob. punkt „Informatory i buforowanie"), który wymusza okresowe uzgadnianie stanu. Są często używane do upewniania się, że stan klastra i zakładany stan zapisany w buforze nie różnią się między sobą (np. z powodu błędów lub problemów z siecią).

Kolejki zadań
: Kolejka zadań to komponent, który może być używany przez mechanizm obsługi zdarzeń do obsługi kolejkowania zmian stanu i do ponawiania prób. W narzędziu `client-go` ten mechanizm jest dostępny w pakiecie *workqueue* (*http://bit.ly/2x7zyeK*; zob. punkt „Kolejka zadań"). Zasoby można ponownie umieszczać w kolejce po wystąpieniu błędów w trakcie aktualizowania świata zewnętrznego lub zapisywania stanu (kroki 2. i 3. w pętli) lub gdy z innych powodów trzeba ponownie przetworzyć zasób po upływie jakiegoś czasu.

Bardziej formalne omówienie Kubernetes jako deklaratywnego silnika i narzędzia do zmieniania stanu znajdziesz w tekście „The Mechanics of Kubernetes" (*http://bit.ly/2IV2lcb*) autorstwa Andrew Chena i Dominika Tornowa.

Przyjrzyjmy się teraz dokładniej pętli sterowania. Zaczniemy od sterowanej zdarzeniami architektury Kubernetesa.

Zdarzenia

Warstwa kontroli w Kubernetesie jest w dużym stopniu oparta na zdarzeniach i zasadzie luźnego powiązania komponentów. W innych systemach rozproszonych do uruchamiania operacji są używane wywołania RPC. W Kubernetesie jest inaczej. Kontrolery Kubernetesa obserwują zmiany w obiektach Kubernetesa na serwerze API: operacje dodawania, aktualizowania i usuwania. Gdy wystąpi takie zdarzenie, kontroler wykonuje logikę biznesową.

Kup książkę Poleć książkę

Na przykład aby uruchomić pod w instalacji, potrzebne jest współdziałanie wielu kontrolerów i innych komponentów z warstwy kontroli:

1. Kontroler instalacji (w menedżerze `kube-controller-manager`) zauważa (za pomocą informatora instalacji), że użytkownik tworzy instalację. Kontroler tworzy wtedy za pomocą logiki biznesowej zbiór replik.
2. Kontroler zbioru replik (także w menedżerze `kube-controller-manager`) zauważa (za pomocą informatora zbioru replik) nowy zbiór replik i uruchamia swoją logikę biznesową, która tworzy obiekt poda.
3. Program szeregujący (w programie binarnym `kube-scheduler`), który sam też jest kontrolerem, zauważa pod (za pomocą informatora poda) z pustym polem `spec.nodeName`. Logika biznesowa programu szeregującego umieszcza pod w kolejce.
4. W tym czasie `kubelet` (kolejny kontroler) zauważa nowy pod (za pomocą informatora poda). Jednak pole `spec.nodeName` nowego poda jest puste, dlatego nie pasuje do nazwy węzła z kontrolera `kubelet`. Kontroler ignoruje więc pod i wraca do trybu uśpienia (do czasu wystąpienia następnego zdarzenia).
5. Program szeregujący pobiera pod z kolejki zadań i szereguje go do wykonania w węźle, który ma wystarczającą ilość wolnych zasobów. W tym celu aktualizuje pole `spec.nodeName` w podzie i zapisuje pod na serwerze API.
6. Kontroler `kubelet` znów jest wybudzany w wyniku zdarzenia aktualizacji poda. Ponownie porównuje wtedy pole `spec.nodeName` z własną nazwą. Jeśli nazwy pasują, kontroler `kubelet` uruchamia kontenery poda i informuje serwer API o włączeniu tych kontenerów, zapisując to w stanie poda.
7. Kontroler zestawu replik wykrywa zmodyfikowany pod, ale nie musi nic z tym robić.
8. Ostatecznie pod kończy działanie. Kontroler `kubelet` to wykrywa, pobiera obiekt poda z serwera API, ustawia warunek „zakończono” w stanie poda i zapisuje pod z powrotem na serwerze API.
9. Kontroler zbioru replik wykrywa pod, który zakończył działanie, i uznaje, że dany pod musi zostać zastąpiony.
10. I tak dalej.

Widać tu, że liczne niezależne pętle sterowania komunikują się wyłącznie dzięki modyfikacjom obiektów na serwerze API i zdarzeń wywoływanych przez te zmiany za pomocą informatorów.

Zdarzenia są wysyłane z serwera API do informatorów w kontrolerach przy użyciu czujek (zob. punkt „Czujki”), a dokładnie zdarzeń czujek przesyłanych połączeniami strumieniowymi. Większość tych operacji jest niewidoczna dla użytkowników. Nawet mechanizm audytu serwera API nie uwidacznia takich zdarzeń. Widoczne są tylko aktualizacje obiektów. Jednak kontrolery często korzystają z danych z dziennika, gdy mają reagować na zdarzenia.

Jeśli chcesz dowiedzieć się więcej na temat zdarzeń, zapoznaj się z artykułem „Events, the DNA of Kubernetes” (*http://bit.ly/2MZwbl6*) z bloga Michaela Gascha. Znajdziesz tam więcej informacji i przykładów.

Kup książkę Poleć książkę

Zdarzenia czujek a obiekty zdarzeń

Zdarzenia czujek i obiekty typu `Event` z najwyższego poziomu są w Kubernetesie dwoma różnymi rzeczami:

- Zdarzenia czujek są przesyłane strumieniowymi połączeniami HTTP między serwerem API a kontrolerami na potrzeby informatorów.
- Obiekty typu `Event` z najwyższego poziomu to zasoby takie jak pody, instalacje lub usługi mające specjalną właściwość — ich czas życia to godzina, a następnie są automatycznie usuwane z systemu `etcd`.

Obiekty typu `Event` to widoczny dla użytkowników mechanizm rejestrowania zdarzeń. Liczne kontrolery tworzą takie zdarzenia, aby informować użytkowników o różnych aspektach logiki biznesowej. Na przykład `kubelet` informuje o zdarzeniach dotyczących cyklu życia podów (np. o uruchomieniu, ponownym uruchomieniu i zamknięciu kontenera).

Za pomocą narzędzia `kubectl` można wyświetlić drugą kategorię zdarzeń zachodzących w klastrze. Poniższe polecenie pozwala zobaczyć, co dzieje się w przestrzeni nazw `kube-system`:

```
$ kubectl -n kube-system get events
LAST SEEN   FIRST SEEN   COUNT   NAME                                                KIND
3m          3m           1       kube-controller-manager-master.15932b6faba8e5ad     Pod
3m          3m           1       kube-apiserver-master.15932b6fa3f3fbbc              Pod
3m          3m           1       etcd-master.15932b6fa8a9a776                        Pod
…
2m          3m           2       weave-net-7nvnf.15932b73e61f5bc6                    Pod
2m          3m           2       weave-net-7nvnf.15932b73efeec0b3                    Pod
2m          3m           2       weave-net-7nvnf.15932b73e8f7d318                    Pod
```

Wyzwalacze sterowane zmianami i sterowane poziomem

Przyjrzyjmy się teraz na bardziej abstrakcyjnym poziomie temu, jak można ustrukturyzować logikę biznesową implementowaną w kontrolerach i dlaczego w Kubernetesie zdecydowano się stosować zdarzenia (np. zmiany stanu) do sterowania logiką.

Istnieją dwa podstawowe sposoby wykrywania zmian stanu (czyli samych zdarzeń):

Wyzwalacze sterowane zmianami
: Mechanizm obsługi jest wyzwalany w momencie, gdy zachodzi zmiana stanu — np. z nieuruchomionego poda na uruchomiony pod.

Wyzwalacze sterowane poziomem
: Stan jest sprawdzany w regularnych odstępach czasu i jeśli spełnione są określone warunki (np. pod działa), wyzwalany jest mechanizm obsługi.

Drugie z tych podejść jest formą odpytywania. Technika ta nie skaluje się dobrze wraz ze wzrostem liczby obiektów, a opóźnienie w kontrolerach wykrywających zmiany zależy od odstępów między operacjami odpytywania i szybkości odpowiadania przez serwer API. Gdy używanych jest wiele asynchronicznych kontrolerów, co opisano w punkcie „Zdarzenia”, powstaje system, który z dużym opóźnieniem reaguje na żądania użytkowników.

Kup książkę Poleć książkę

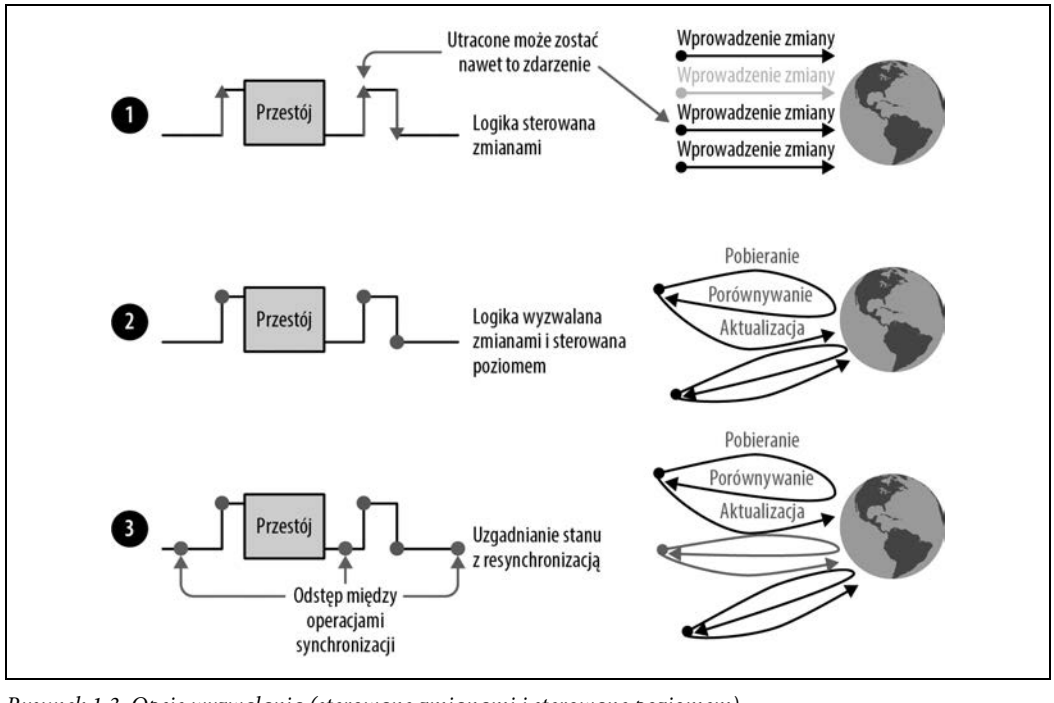
Gdy używanych jest wiele obiektów, znacznie wydajniejsza jest pierwsza z tych technik. Opóźnienie zależy wtedy głównie od liczby wątków roboczych w kontrolerach przetwarzających zdarzenia. Dlatego Kubernetes jest oparty na zdarzeniach (czyli używane są wyzwalacze sterowane zmianami).

W warstwie kontroli w Kubernetesie liczne komponenty modyfikują obiekty na serwerze API, a każda zmiana prowadzi do zdarzenia (czyli zmiany). Takie komponenty są nazywane *źródłami zdarzeń* lub *generatorami zdarzeń*. W kontrolerach istotne jest konsumowanie tych zdarzeń, czyli to, kiedy i jak reagować na zdarzenia (z użyciem informatorów).

W systemach rozproszonych równolegle działa wiele jednostek, a zdarzenia nadchodzą asynchronicznie w dowolnej kolejności. Gdy logika kontrolera zawiera błędy, gdy używana jest nieprawidłowa maszyna stanowa lub gdy nastąpi awaria zewnętrznej usługi, łatwo jest utracić zdarzenia (w tym sensie, że nie zostają one w pełni przetworzone). Dlatego trzeba dobrze przyjrzeć się sposobom radzenia sobie z błędami.

Na rysunku 1.3 pokazane są trzy różne strategie:

1. Przykład logiki sterowanej tylko zmianami, gdzie utracona może zostać druga zmiana stanu.
2. Przykład logiki wyzwalanej zmianami, gdzie jednak przy przetwarzaniu zdarzenia zawsze pobierany jest najnowszy stan (czyli poziom). Oznacza to, że logika jest wyzwalana zmianami, ale sterowana poziomem.
3. Przykład logiki wyzwalanej zmianami i sterowanej poziomem z dodatkową resynchronizacją.



Rysunek 1.3. Opcje wyzwalania (sterowane zmianami i sterowane poziomem)

Kup książkę Poleć książkę

Strategia nr 1 nie radzi sobie dobrze z pominiętymi zdarzeniami. Zdarzenia mogą zostać pominięte z powodu utraty zdarzeń w uszkodzonej sieci, błędów w samym kontrolerze lub przestoju API w zewnętrznej chmurze. Wyobraź sobie, że kontroler zbioru replik zastępuje pody tylko po zakończeniu przez nie pracy. Pominięcie zdarzeń może oznaczać, że w zbiorze replik zawsze będzie uruchamiana mniejsza liczba podów, ponieważ nigdy nie jest uzgadniany pełny stan.

Strategia nr 2 pozwala po takich problemach przywrócić stan, gdy odebrane zostanie następne zdarzenie. Wynika to z tego, że logika jest oparta na najnowszym stanie klastra. Kontroler zbioru replik zawsze porównuje wtedy określoną liczbę replik z liczbą podów uruchomionych w klastrze. Po utracie zdarzenia wszystkie brakujące pody zostaną zastąpione po otrzymaniu następnej aktualizacji podów.

W strategii nr 3 dodana została ciągła resynchronizacja (np. co pięć minut). Jeśli nie zostaną zgłoszone nowe zdarzenia dotyczące podów, system uzgodni stan przynajmniej co pięć minut, nawet jeśli aplikacja działa bardzo stabilnie i nie generuje wielu zdarzeń dotyczących podów.

Z powodu problemów związanych z wyzwalaczami sterowanymi tylko zmianami w kontrolerach Kubernetesa zwykle stosowana jest strategia nr 3.

Jeśli chcesz dowiedzieć się czegoś więcej na temat źródeł wyzwalaczy oraz powodów zastosowania w Kubernetesie wyzwalaczy opartych na poziomach i uzgadniania stanu, zapoznaj się z artykułem „Level Triggering and Reconciliation in Kubernetes” (*http://bit.ly/2FmLLAW*) Jamesa Bowesa.

To kończy omawianie różnych abstrakcyjnych sposobów wykrywania zewnętrznych zmian i reagowania na nie. Następny krok w pętli sterowania z rysunku 1.2 dotyczy modyfikowania obiektów w klastrze lub zmieniania świata zewnętrznego zgodnie ze specyfikacją. Zajmijmy się tym teraz.

Modyfikowanie świata zewnętrznego lub obiektów w klastrze

W tej fazie kontroler zmienia stan nadzorowanych obiektów. Na przykład kontroler `ReplicaSet` w menedżerze kontrolera (*http://bit.ly/2WUAEVy*) nadzoruje pody. Po każdym zdarzeniu (wyzwalanie zmianami) kontroler wykrywa bieżący stan nadzorowanych podów i porównuje go z oczekiwanym stanem (sterowanie poziomem).

Ponieważ proces zmiany stanu zasobów może być specyficzny dla domeny lub dla zadania, trudno jest coś doradzać w tym obszarze. Zamiast tego należy przyjrzeć się wprowadzonemu wcześniej kontrolerowi `ReplicaSet`. Takie kontrolery są używane w instalacjach, a podstawowe zadanie takiego kontrolera to utrzymywanie zdefiniowanej przez użytkownika liczby identycznych replik podów. Oznacza to, że gdy liczba podów jest mniejsza od podanej (np. z powodu awarii poda lub zwiększenia liczby replik), kontroler uruchamia nowe pody. Jeśli jednak podów jest zbyt wiele, kontroler zamyka niektóre z nich. Cała logika biznesowa kontrolera jest dostępna w pakiecie *replica_set.go* (*http://bit.ly/2L4eKxa*), a poniższy fragment kodu w Go dotyczy zmian stanu (kod został zmodyfikowany pod kątem przejrzystości):

```
// Funkcja manageReplicas sprawdza i aktualizuje repliki z danego zbioru ReplicaSet.
// Funkcja NIE modyfikuje kolekcji <filteredPods>.
// Błąd w trakcie tworzenia lub usuwania podów skutkuje ponownym zakolejkowaniem danego zbioru replik.
func (rsc *ReplicaSetController) manageReplicas(
        filteredPods []*v1.Pod, rs *apps.ReplicaSet,
```


Kup książkę Poleć książkę

```
) error {
    diff := len(filteredPods) - int(*(rs.Spec.Replicas))
    rsKey, err := controller.KeyFunc(rs)
    if err != nil {
        utilruntime.HandleError(
            fmt.Errorf("Nie można pobrać klucza dla %v %#v: %v", rsc.Kind, rs, err),
        )
        return nil
    }
    if diff < 0 {
        diff *= -1
        if diff > rsc.burstReplicas {
            diff = rsc.burstReplicas
        }
        rsc.expectations.ExpectCreations(rsKey, diff)
        klog.V(2).Infof("Za mało replik dla %v %s/%s, potrzebnych %d, tworzenie %d",
            rsc.Kind, rs.Namespace, rs.Name, *(rs.Spec.Replicas), diff,
        )
        successfulCreations, err := slowStartBatch(
            diff,
            controller.SlowStartInitialBatchSize,
            func() error {
                ref := metav1.NewControllerRef(rs, rsc.GroupVersionKind)
                err := rsc.podControl.CreatePodsWithControllerRef(
                    rs.Namespace, &rs.Spec.Template, rs, ref,
                )
                if err != nil && errors.IsTimeout(err) {
                    return nil
                }
                return err
            },
        )
        if skippedPods := diff - successfulCreations; skippedPods > 0 {
            klog.V(2).Infof("Niepowodzenie przy powolnym rozruchu. Pominięto " +
                "tworzenie %d podów, zmniejszanie oczekiwań dla %v %v/%v",
                skippedPods, rsc.Kind, rs.Namespace, rs.Name,
            )
            for i := 0; i < skippedPods; i++ {
                rsc.expectations.CreationObserved(rsKey)
            }
        }
        return err
    } else if diff > 0 {
        if diff > rsc.burstReplicas {
            diff = rsc.burstReplicas
        }
        klog.V(2).Infof("Zbyt wiele replik dla %v %s/%s, potrzebnych %d, usuwanie %d",
            rsc.Kind, rs.Namespace, rs.Name, *(rs.Spec.Replicas), diff,
        )

        podsToDelete := getPodsToDelete(filteredPods, diff)
        rsc.expectations.ExpectDeletions(rsKey, getPodKeys(podsToDelete))
        errCh := make(chan error, diff)
        var wg sync.WaitGroup
        wg.Add(diff)
        for _, pod := range podsToDelete {
            go func(targetPod *v1.Pod) {
                defer wg.Done()
```


Kup książkę Poleć książkę

```
                if err := rsc.podControl.DeletePod(
                    rs.Namespace,
                    targetPod.Name,
                    rs,
                ); err != nil {
                    podKey := controller.PodKey(targetPod)
                    klog.V(2).Infof("Nieudane usuwanie %v, zmniejszanie " +
                        "oczekiwań dla %v %s/%s",
                        podKey, rsc.Kind, rs.Namespace, rs.Name,
                    )
                    rsc.expectations.DeletionObserved(rsKey, podKey)
                    errCh <- err
                }
            }(pod)
        }
        wg.Wait()

        select {
        case err := <-errCh:
            if err != nil {
                return err
            }
        default:
        }
    }
    return nil
}
```

Widać tu, że w wierszu `diff := len(filteredPods) - int(*(rs.Spec.Replicas))` kontroler oblicza różnicę między specyfikacją a bieżącym stanem, a następnie na podstawie wyniku realizuje dwa scenariusze:

- `diff < 0`: replik jest za mało, trzeba utworzyć dodatkowe pody.
- `diff > 0`: replik jest za dużo, trzeba usunąć pody.

W funkcji `getPodsToDelete` zaimplementowana jest strategia wyboru podów, których usunięcie będzie najmniej szkodliwe.

Jednak zmiana stanu zasobu nie musi oznaczać, że dany zasób jest częścią klastra Kubernetesa. Kontroler może więc modyfikować stan zasobów zlokalizowanych poza Kubernetesem, np. w usłudze przechowywania danych w chmurze. Na przykład narzędzie AWS Service Operator (*http://bit.ly/2ItJcif*) pozwala zarządzać zasobami w usłudze AWS. Między innymi możliwe jest zarządzanie komorami S3. Oznacza to, że kontroler S3 nadzoruje zasób (komorę S3), która istnieje poza Kubernetesem, a zmiany stanu odzwierciedlają konkretne etapy cyklu życia: komora S3 jest tworzona i w określonym momencie usuwana.

To powinno Cię przekonać, że niestandardowy kontroler pozwala zarządzać nie tylko podstawowymi zasobami (takimi jak pody) i niestandardowymi zasobami (takimi jak przykładowe narzędzie `cnat`), ale nawet przeprowadzać obliczenia lub przechowywać zasoby poza Kubernetesem. Dzięki temu kontrolery są bardzo elastycznymi i oferującymi duże możliwości mechanizmami integracji rozwiązań oraz zapewniają jednolity sposób używania zasobów w różnych platformach i środowiskach.

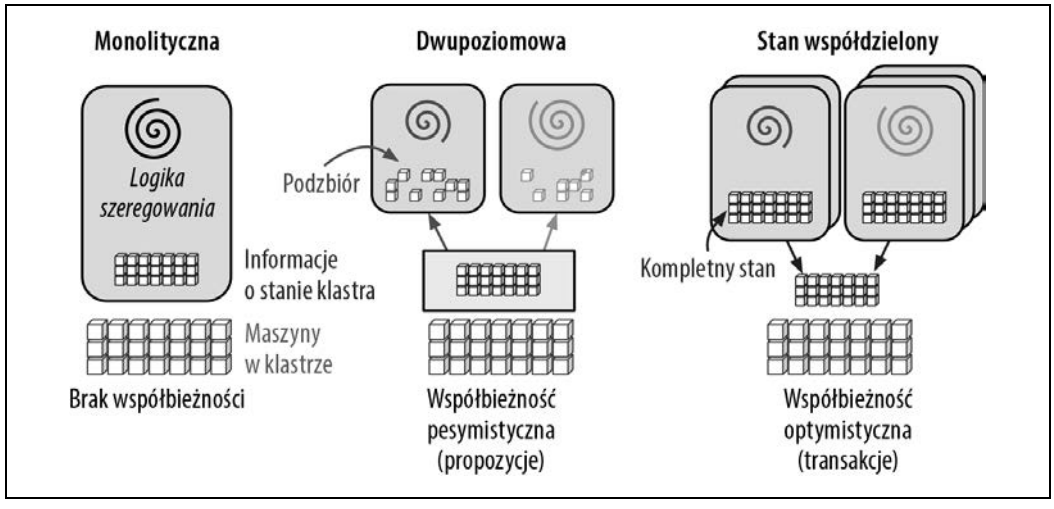
Kup książkę Poleć książkę

Współbieżność optymistyczna

W punkcie „Pętla sterowania" opisano w kroku 3., że kontroler (po zaktualizowaniu zgodnie ze specyfikacją obiektów klastra i/lub świata zewnętrznego) zapisuje wyniki w stanie zasobu, który wyzwolił uruchomienie kontrolera w kroku 1.

Ten i niemal dowolny inny zapis (także w kroku 2.) może zakończyć się niepowodzeniem. W systemie rozproszonym kontroler jest zwykle tylko jedną z wielu jednostek aktualizujących zasoby. Współbieżne zapisy mogą się nie powieść z powodu konfliktów zapisu.

Aby lepiej zrozumieć, co się nie powiodło, warto przyjrzeć się rysunkowi 1.4[2].



Rysunek 1.4. Architektury szeregowania w systemach rozproszonych

W źródłowym tekście architektura równoległego programu szeregującego z systemu Omega została zdefiniowana tak:

> Nasze rozwiązanie to nowa architektura równoległego programu szeregującego oparta na współdzielonym stanie i wykorzystująca optymistyczną kontrolę współbieżności bez blokad, aby uzyskać zarówno rozszerzalność implementacji, jak i skalowalność wydajności. Ta architektura jest używana w Omedze — systemie zarządzania klastrami nowej generacji opracowanym w firmie Google.

Choć w Kubernetesie wykorzystano wiele cech i lekcji z systemu Borg (*http://bit.ly/2XNSv5p*), ten konkretny transakcyjny mechanizm z warstwy kontroli pochodzi z Omegi. W celu wykonywania współbieżnych operacji bez blokad serwer API Kubernetesa używa współbieżności optymistycznej.

W skrócie oznacza to, że jeśli (i kiedy) serwer API wykryje próbę współbieżnego zapisu, odrzuca późniejszą z dwóch operacji zapisu. Następnie to klient (kontroler, program szeregujący, narzędzie `kubectl` itd.) musi obsłużyć konflikt i ewentualnie ponowić próbę operacji zapisu.

[2] Źródło: „Omega: Flexible, Scalable Schedulers for Large Compute Clusters" (*http://bit.ly/2PjYZ59*), Malte Schwarzkopf i inni, Google AI, 2013.

Kup książkę Poleć książkę

Poniżej zilustrowano działanie optymistycznej współbieżności w Kubernetesie:

```
var err error
for retries := 0; retries < 10; retries++ {
    foo, err = client.Get("foo", metav1.GetOptions{})
    if err != nil {
        break
    }

    <aktualizowanie-świata-i-inne>

    _, err = client.Update(foo)
    if err != nil && errors.IsConflict(err) {
        continue
    } else if err != nil {
        break
    }
}
```

W tym kodzie pokazana jest pętla z ponawianiem prób, która w każdej iteracji pobiera najnowszy obiekt `foo`, a następnie próbuje zaktualizować świat zewnętrzny i stan obiektu `foo` zgodnie ze specyfikacją tego obiektu. Zmiany przed wywołaniem `Update` są wprowadzane w trybie optymistycznym.

Obiekt `foo` zwracany przez wywołanie `client.Get` zawiera *wersję zasobu* (część zagnieżdżonej struktury `ObjectMeta`; szczegóły znajdziesz w punkcie „ObjectMeta"), informującą system `etcd` związany z operacją zapisu z wywołania `client.Update`, że inna jednostka w klastrze w międzyczasie zapisała obiekt `foo`. Jeśli taki inny zapis miał miejsce, w pętli z ponawianiem prób występuje *błąd konfliktu wersji zasobu*. To oznacza, że logika optymistycznej współbieżności zawiodła. Wywołanie `client.Update` jest więc optymistyczne.

Wersja zasobu to para klucz-wartość z systemu `etcd`. Wersją zasobu dla każdego obiektu jest łańcuch znaków z Kubernetesa zawierający liczbę całkowitą. Ta liczba całkowita pochodzi bezpośrednio z systemu `etcd`. Ten system zawiera licznik zwiększany za każdym razem, gdy modyfikowana jest powiązana z kluczem wartość (przechowująca zserializowany obiekt).

W kodzie API wersja zasobu jest (dość konsekwentnie) obsługiwana jak zwykły, ale uporządkowany w określony sposób łańcuch znaków. To, że ten łańcuch znaków zawiera liczby całkowite, jest jedynie szczegółem implementacji obecnie używanego backendu systemu `etcd`.

Przyjrzyj się konkretnemu przykładowi. Wyobraź sobie, że Twój klient nie jest jedyną jednostką w klastrze modyfikującą pod. Istnieje też inna jednostka, `kubelet`, która nieustannie modyfikuje jakieś pola, ponieważ kontener stale ulega awarii. Twój kontroler wczytuje najnowszy stan poda:

```
kind: Pod
metadata:
  name: foo
  resourceVersion: 57
spec:
  ...
status:
  ...
```


Kup książkę Poleć książkę

Teraz załóżmy, że kontroler potrzebuje kilku sekund na wprowadzenie zmian w świecie zewnętrznym. Po siedmiu sekundach kontroler próbuje zaktualizować wczytany pod — np. zapisuje w nim adnotacje. W tym czasie `kubelet` wykrywa restart innego kontenera i aktualizuje stan poda, aby to odzwierciedlić. Oznacza to, że wartość pola `resourceVersion` rośnie do 58.

Obiekt przesłany przez kontroler w żądaniu aktualizacji ma pole `resourceVersion: 57`. Serwer API próbuje ustawić w systemie `etcd` klucz dla poda o tej wartości. System `etcd` wykrywa niedopasowanie wersji zasobu i zgłasza konflikt wersji 57 i 58. Aktualizacja kończy się niepowodzeniem.

Podsumowanie tego przykładu jest takie, że programista kontrolera odpowiada za implementacją strategii ponawiania prób i uwzględniania niepowodzenia optymistycznych operacji. Nigdy nie wiadomo, jakie jeszcze jednostki modyfikują stan. Mogą to być inne kontrolery lub kontrolery podstawowe, np. kontroler instalacji.

Oto istota tego fragmentu: *błędy spowodowane konfliktami są w kontrolerach czymś zupełnie normalnym. Zawsze powinieneś się ich spodziewać i obsługiwać je w kontrolowany sposób.*

Należy zwrócić uwagę na to, że współbieżność optymistyczna doskonale współdziała z logiką opartą na poziomach, ponieważ dzięki temu można bez dodatkowych zabiegów ponownie uruchomić pętlę sterowania (zob. punkt „Wyzwalacze sterowane zmianami i sterowane poziomem”). W następnym przebiegu pętla automatycznie wycofa optymistyczne zmiany wprowadzone w nieudanej optymistycznej próbie i spróbuje wprowadzić najnowszy stan w systemie.

Przejdźmy teraz do konkretnego rodzaju niestandardowych kontrolerów (powiązanego z niestandardowymi zasobami) — do operatorów.

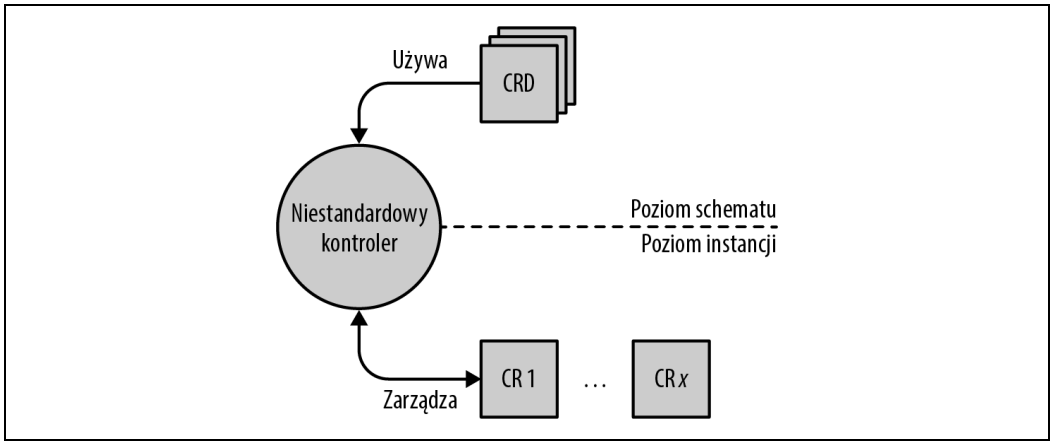
Operatory

Operatory to mechanizm Kubernetesa wprowadzony przez firmę CoreOS w 2016 r. W swoim przełomowym wpisie na blogu „Introducing Operators: Putting Operational Knowledge into Software” (*http://bit.ly/2ZC4Rui*) Brandon Philips, dyrektor techniczny firmy CoreOS, zdefiniował operatory tak:

> Inżynier SRE to osoba, która operuje aplikacją, pisząc oprogramowanie. Jest to inżynier (programista), który wie, jak pisać oprogramowanie w dziedzinie powiązanej z konkretną aplikacją. W wynikowym oprogramowaniu wbudowana jest wiedza z dziedziny działania aplikacji.
>
> […]
>
> Tę nową kategorię oprogramowania nazywamy operatorami. Operator to specyficzny dla aplikacji kontroler, który rozszerza API Kubernetesa, aby tworzyć, konfigurować i obsługiwać instancje złożonych aplikacji stanowych na rzecz użytkownika Kubernetesa. Operator jest oparty na podstawowych mechanizmach Kubernetesa, zasobach i kontrolerach, ale uwzględnia dziedzinę lub wiedzę specyficzną dla aplikacji, aby automatyzować powtarzalne zadania.

W tej książce używamy operatorów zgodnie z opisem Philipsa. W bardziej formalnym ujęciu operatory muszą spełniać trzy warunki (zob. też rysunek 1.5):

Kup książkę Poleć książkę



Rysunek 1.5. Mechanizm operatora

- Istnieje specyficzna dla dziedziny wiedza operacyjna, którą chcesz zautomatyzować.
- Najlepsze praktyki związane z tą wiedzą operacyjną są znane i można je opisać. Na przykład w operatorze Cassandra ta wiedza dotyczy tego, kiedy i jak równoważyć węzły, a w operatorze systemu service mesh — jak tworzyć trasy.
- Jednostki udostępniane razem z operatorem to:
  - Zestaw *definicji CRD* reprezentujących schemat specyficzny dla dziedziny, a także niestandardowych zasobów zgodnych z definicjami CRD, które na poziomie instancji reprezentują daną dziedzinę.
  - Niestandardowy kontroler nadzorujący niestandardowe zasoby, a czasem także zasoby podstawowe. Niestandardowy kontroler może np. uruchamiać pody.

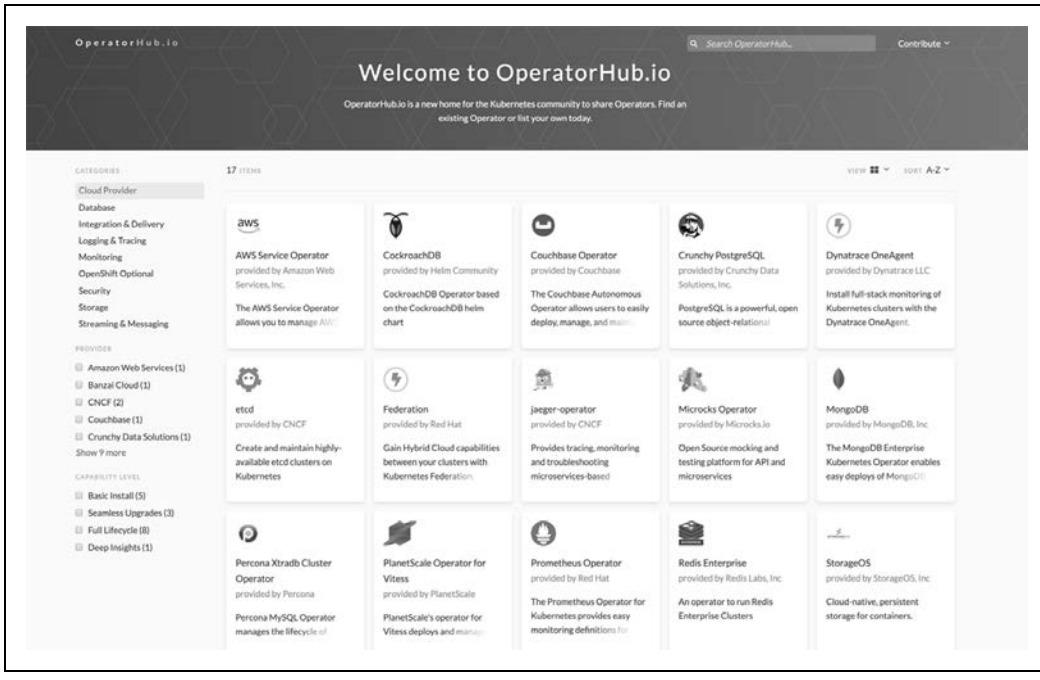
Operatory przeszły długą drogę (*http://bit.ly/2x5TSNw*) od koncepcji i prototypów w 2016 r. do uruchomienia na początku 2019 r. serwisu OperatorHub.io (*https://operatorhub.io*) przez firmę Red Hat (która w 2018 r. przejęła CoreOS i rozwija omawiany mechanizm). Na rysunku 1.6 pokazany jest zrzut tego serwisu z połowy 2019 r. Dostępnych jest na nim 17 operatorów gotowych do użycia.

Podsumowanie

W pierwszym rozdziale zdefiniowaliśmy zakres książki i to, czego od Ciebie oczekujemy. Wyjaśniliśmy, co oznacza tu programowanie dla Kubernetesa. Zdefiniowaliśmy też, czym są natywne aplikacje dla Kubernetesa. W ramach przygotowań do dalszych przykładów przedstawiliśmy też ogólne wprowadzenie do kontrolerów i operatorów.

Teraz gdy wiesz już, czego możesz oczekiwać od tej książki i jakie korzyści może przynieść Ci jej lektura, pora przejść do szczegółów. W następnym rozdziale przyjrzymy się API Kubernetesa, wewnętrznym mechanizmom serwera API, a także interakcjom z tym API z użyciem narzędzi wiersza poleceń takich jak `curl`.

Kup książkę Poleć książkę



Rysunek 1.6. Zrzut serwisu OperatorHub.io

Kup książkę Poleć książkę

Kup książkę Poleć książkę

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

Kubernetes: wykorzystaj w pełni jego potencjał!

Mimo że w 2018 roku Kubernetes zyskał reputację standardowego narzędzia do zarządzania kontenerami, wciąż należy traktować tę technologię jako znajdującą się w dość wczesnej fazie rozwoju. Możliwości tej platformy z czasem rosną i stają się coraz ciekawsze. Standardowym działaniem jest uruchamianie klastrów w Kubernetesie, jednak o wiele bardziej interesujące wydaje się samodzielne rozszerzanie tej platformy o własne kontrolery i rozbudowa API Kubernetesa w celu dostosowania do swoich wymagań. W przypadku niektórych złożonych projektów programowanie potrzebnych komponentów jest jedynym racjonalnym rozwiązaniem.

To książka przeznaczona dla programistów, którzy chcą maksymalnie wykorzystać możliwości Kubernetesa, również poprzez pisanie własnych zasobów. W praktyczny sposób pokazuje, jak rozwijać natywne, działające w chmurze aplikacje dla Kubernetesa. Wyjaśnia, w jaki sposób działa biblioteka API client-go i jak należy budować zasoby niestandardowe. Znalazło się tu obszerne i szczegółowe omówienie interfejsu programowania i działania platformy Kubernetes, a także pisania stabilnego oprogramowania w języku Go. Nie zabrakło szeregu wskazówek dotyczących samego pisania kodu oraz przeprowadzania testów. Dużo uwagi poświęcono niestandardowym zasobom, kontrolerom, webhookom i niestandardowym serwerom API oraz wzorcom rozszerzania Kubernetesa.

W tej książce między innymi:

- zasady programowania dla Kubernetesa
- API Kubernetesa i client-go
- korzystanie z niestandardowych zasobów
- pisanie i udostępnianie operatorów
- tworzenie niestandardowych serwerów API

Michael Hausenblas pracuje w Amazon Web Services, gdzie wraz z zespołem zajmuje się bezpieczeństwem kontenerów. Ma bogate doświadczenie w tworzeniu natywnej infrastruktury i natywnych aplikacji dla chmury. Pisze artykuły i książki, prowadzi prelekcje i współtworzy otwarte oprogramowanie.

Stefan Schimanski pracuje w firmie Red Hat jako główny inżynier oprogramowania w obszarze technologii Go, Kubernetes i OpenShift. Koncentruje się na serwerze API Kubernetesa, a przede wszystkim na implementowaniu definicji CRD, bibliotece API Machinery i publikowaniu repozytoriów roboczych Kubernetesa: client-go, apimachinery, api i innych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-6405-9

INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 54,90 zł