Nauka KUBERNETESA W MIESIĄC

ELTON STONEMAN

Tytuł oryginału: Learn Kubernetes in a Month of Lunches

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-7910-7

Original edition copyright © 2021 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2022 by Helion S.A.
All rights reserved.

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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

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

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Wstęp 11
Podziękowania 12
O książce 13

TYDZIEŃ I. SZYBKA DROGA DO OPANOWANIA KUBERNETESA 19

1. *Zanim zaczniesz* 21
   1.1. Jak działa Kubernetes? 22
   1.2. Czy ta książka jest dla Ciebie? 26
   1.3. Tworzenie środowiska laboratoryjnego 27
   1.4. Natychmiastowa efektywność 33
2. *Uruchamianie kontenerów w Kubernetesie za pomocą kapsuł i wdrożeń* 34
   2.1. Jak Kubernetes uruchamia kontenery i zarządza nimi? 35
   2.2. Uruchamianie kapsuł za pomocą kontrolerów 41
   2.3. Definiowanie wdrożeń w manifestach aplikacji 48
   2.4. Praca z aplikacjami działającymi w kapsułach 52
   2.5. Zarządzanie zasobami przez Kubernetes 56
   2.6. Laboratorium 59
3. *Łączenie kapsuł przez sieć za pomocą usług* 60
   3.1. Jak Kubernetes routuje ruch sieciowy? 61
   3.2. Routowanie ruchu między kapsułami 65
   3.3. Routowanie do kapsuł ruchu zewnętrznego 70
   3.4. Routowanie ruchu poza Kubernetes 75
   3.5. Jak działa rozwiązywanie usług w Kubernetesie? 81
   3.6. Laboratorium 86
4. *Konfigurowanie aplikacji za pomocą obiektów ConfigMap i Secret* 87
   4.1. Jak Kubernetes dostarcza konfigurację do aplikacji? 88
   4.2. Zapisywanie plików konfiguracyjnych w obiektach ConfigMap oraz ich używanie 93

Kup ksi k   Pole ksi k

4.3. Udostępnianie danych konfiguracyjnych z obiektów ConfigMap 98
4.4. Konfigurowanie poufnych danych za pomocą obiektów Secret 106
4.5. Zarządzanie konfiguracją aplikacji w Kubernetesie 113
4.6. Laboratorium 115

5. Przechowywanie danych przy użyciu woluminów, punktów montowania i żądań 117

5.1. Jak Kubernetes buduje system plików kontenera? 118
5.2. Przechowywanie danych na węźle za pomocą woluminów i punktów montowania 123
5.3. Użycie woluminów trwałych oraz żądań do przechowywania danych dla całego klastra 131
5.4. Dynamiczna alokacja woluminów i klasy pamięci masowej 141
5.5. Opcje wyboru pamięci masowej w Kubernetesie 147
5.6. Laboratorium 148

6. Używanie kontrolerów do skalowania aplikacji w celu rozproszenia ich na wiele kapsuł 149

6.1. Jak Kubernetes uruchamia skalowalne aplikacje? 150
6.2. Używanie wdrożeń i zbiorów replik do skalowania pod kątem obciążenia 156
6.3. Używanie kontrolerów DaemonSet do skalowania pod kątem zapewniania wysokiej dostępności 165
6.4. Własność obiektów w Kubernetesie 171
6.5. Laboratorium 174

TYDZIEŃ II. KUBERNETES W PRAWDZIWYM ŚWIECIE 175

7. Rozszerzanie aplikacji o wielokontenerowe kapsuły 177

7.1. Jak kontenery komunikują się w kapsule? 178
7.2. Konfigurowanie aplikacji za pomocą kontenerów inicjujących 184
7.3. Zapewnianie spójności za pomocą kontenerów adapterów 191
7.4. Tworzenie warstwy abstrakcji połączeń za pomocą kontenerów ambasadorów 195
7.5. Środowisko kapsuły 199
7.6. Laboratorium 204

Kup ksi k Pole ksi k

8. Wykorzystywanie kontrolerów StatefulSet i Job do uruchamiania aplikacji operujących na dużych ilościach danych 205

8.1. Jak Kubernetes modeluje stabilność za pomocą kontrolerów StatefulSet 206
8.2. Używanie kontenerów inicjujących do ładowania kapsuł w zbiorach stanowych 210
8.3. Żądanie pamięci masowej za pomocą szablonów PVC 216
8.4. Uruchamianie zadań konserwacyjnych za pomocą kontrolerów Job i CronJob 222
8.5. Wybór platformy dla aplikacji stanowych 229
8.6. Laboratorium 231

9. Zarządzanie wydawaniem nowych wersji aplikacji za pomocą rolloutów i rollbacków 233

9.1. Jak Kubernetes zarządza rolloutami? 234
9.2. Aktualizowanie wdrożeń za pomocą rolloutów i rollbacków 237
9.3. Konfigurowanie dla wdrożeń aktualizacji kroczących 245
9.4. Aktualizacje kroczące w zbiorach demonów i zbiorach stanowych 254
9.5. Strategie wydawania nowych wersji 259
9.6. Laboratorium 261

10. Pakowanie aplikacji i zarządzanie nimi za pomocą menedżera pakietów Helm 263

10.1. Jakie funkcjonalności Helm dodaje do Kubernetesa? 264
10.2. Pakowanie własnych aplikacji za pomocą menedżera pakietów Helm 270
10.3. Modelowanie zależności w wykresach 279
10.4. Wykonywanie uaktualnień i rollbacków wydań Helma 283
10.5. Zastosowania menedżera pakietów Helm 290
10.6. Laboratorium 291

11. Tworzenie aplikacji — programistyczne przepływy pracy oraz potok CI/CD 292

11.1. Programistyczny przepływ pracy oparty na Dockerze 293
11.2. Programistyczny przepływ pracy Kubernetesa jako usługi 298
11.3. Izolowanie obciążeń roboczych za pomocą kontekstów i przestrzeni nazw 305
11.4. Ciągłe dostarczanie w Kubernetesie bez Dockera 310

Kup ksi k Pole ksi k

11.5. Ocena programistycznych przepływów pracy w Kubernetesie 316
11.6. Laboratorium 318

TYDZIEŃ III. PRZYGOTOWANIE DO DZIAŁANIA W ŚRODOWISKU PRODUKCYJNYM 321

12. Konfigurowanie samonaprawiających się aplikacji 323

12.1. Routowanie ruchu do zdrowych kapsuł przy użyciu sond gotowości 324
12.2. Wykorzystanie sond żywotności do restartowania kapsuł, które uległy awarii 330
12.3. Bezpieczne wdrażanie uaktualnień za pomocą menedżera pakietów Helm 335
12.4. Chronienie aplikacji i węzłów za pomocą limitów zasobów 342
12.5. Ograniczenia samonaprawiających się aplikacji 350
12.6. Laboratorium 351

13. Centralizacja dzienników za pomocą oprogramowania Fluentd i Elasticsearch 352

13.1. Jak Kubernetes przechowuje wpisy dzienników? 353
13.2. Gromadzenie dzienników z węzłów za pomocą Fluentd 357
13.3. Wysyłanie dzienników do Elasticsearch 364
13.4. Parsowanie i filtrowanie wpisów dzienników 369
13.5. Opcje rejestrowania w Kubernetesie 374
13.6. Laboratorium 376

14. Monitorowanie aplikacji i Kubernetesa za pomocą pakietu narzędziowego Prometheus 377

14.1. Jak Prometheus monitoruje obciążenia robocze Kubernetesa? 378
14.2. Monitorowanie aplikacji zbudowanych przy użyciu bibliotek klienckich Prometheusa 384
14.3. Monitorowanie zewnętrznych aplikacji przy użyciu eksporterów wskaźników 391
14.4. Monitorowanie kontenerów i obiektów Kubernetesa 397
14.5. Inwestycje w monitorowanie 402
14.6. Laboratorium 404

15. Zarządzanie ruchem przychodzącym za pomocą obiektu Ingress 405

15.1. W jaki sposób Kubernetes routuje ruch za pomocą obiektu Ingress? 406
15.2. Routing ruchu HTTP za pomocą reguł obiektu Ingress 411
15.3. Porównanie kontrolerów ruchu przychodzącego 418
15.4. Używanie obiektu Ingress do zabezpieczania aplikacji za pomocą protokołu HTTPS 428
15.5. Obiekt Ingress i kontrolery ruchu przychodzącego 433
15.6. Laboratorium 434

16. Zabezpieczanie aplikacji za pomocą reguł, kontekstów i sterowania dostępem 436

16.1. Zabezpieczanie komunikacji za pomocą reguł sieciowych 437
16.2. Ograniczanie możliwości kontenerów za pomocą kontekstów bezpieczeństwa 445
16.3. Blokowanie i modyfikowanie obciążeń roboczych za pomocą zaczepów sieciowych 451
16.4. Sterowanie dostępem za pomocą silnika Open Policy Agent 458
16.5. Kwestie bezpieczeństwa w Kubernetesie 466
16.6. Laboratorium 467

TYDZIEŃ IV. CZYSTY KUBERNETES W PRAKTYCE 469

17. Zabezpieczanie zasobów za pomocą kontroli dostępu opartej na rolach 471

17.1. Jak Kubernetes zabezpiecza dostęp do zasobów? 472
17.2. Zabezpieczanie dostępu do zasobów wewnątrz klastra 479
17.3. Wiązanie ról z grupami użytkowników i kont usług 488
17.4. Wykrywanie i kontrolowanie uprawnień za pomocą wtyczek 496
17.5. Planowanie strategii RBAC 500
17.6. Laboratorium 501

18. Wdrażanie Kubernetesa: klastry wielowęzłowe i wieloarchitekturowe 503

18.1. Co się znajduje w klastrze Kubernetesa? 504
18.2. Inicjowanie płaszczyzny sterowania 508
18.3. Dodawanie węzłów i uruchamianie obciążeń roboczych na węzłach linuksowych 512
18.4. Dodawanie węzłów Windowsa i uruchamianie hybrydowych obciążeń roboczych 520

Kup ksi k Pole ksi k

18.5. Kubernetes na dużą skalę 529
18.6. Laboratorium 530

19. Kontrolowanie rozmieszczania obciążeń roboczych i automatyczne skalowanie 531

19.1. Jak Kubernetes rozdysponowuje obciążenia robocze? 532
19.2. Zarządzanie rozmieszczaniem kapsuł za pomocą powinowactwa i antypowinowactwa 537
19.3. Kontrolowanie wydajności za pomocą automatycznego skalowania 545
19.4. Ochrona zasobów za pomocą wywłaszczeń i priorytetów 552
19.5. Mechanizmy zarządzania obciążeniami roboczymi 559
19.6. Laboratorium 561

20. Rozszerzanie Kubernetesa o niestandardowe zasoby i operatory 562

20.1. Jak rozszerzać Kubernetes za pomocą niestandardowych zasobów? 563
20.2. Wyzwalanie przepływów pracy za pomocą niestandardowych kontrolerów 567
20.3. Zarządzanie zewnętrznymi komponentami przy użyciu operatorów 575
20.4. Budowanie operatorów dla własnych aplikacji 584
20.5. Kiedy rozszerzać Kubernetes? 591
20.6. Laboratorium 592

21. Uruchamianie w Kubernetesie funkcji bezserwerowych 594

21.1. Jak działają platformy bezserwerowe w Kubernetesie? 595
21.2. Wywoływanie funkcji za pomocą żądań HTTP 601
21.3. Wywoływanie funkcji za pomocą zdarzeń i harmonogramów 606
21.4. Tworzenie warstwy abstrakcji dla funkcji bezserwerowych przy użyciu projektu Serverless 613
21.5. Kiedy stosować funkcje bezserwerowe? 620
21.6. Laboratorium 621

22. Nauka nigdy się nie kończy 623

22.1. Dalsza lektura dla poszczególnych rozdziałów 623
22.2. Wybór platformy Kubernetesa 627
22.3. Jak jest zbudowany Kubernetes? 629
22.4. Dołączanie do społeczności 630

5 Przechowywanie danych przy użyciu woluminów, punktów montowania i żądań

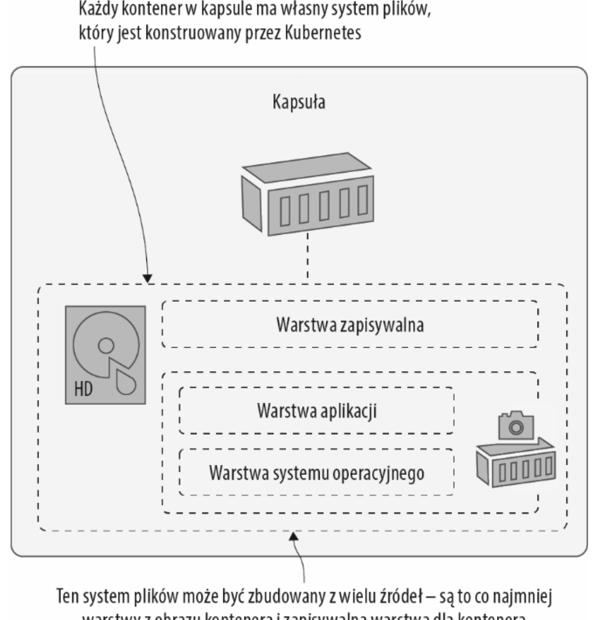
Dostęp do danych w środowisku klastrowym jest trudny. Przenoszenie obliczeń to łatwa część — API Kubernetesa jest w stałym kontakcie z węzłami, a gdy węzeł przestanie odpowiadać, Kubernetes może założyć, że jest on offline, i rozpocząć zastępowanie wszystkich jego kapsuł na innych węzłach. Jeśli jednak aplikacja w jednej z tych kapsuł przechowywała dane na węźle, wówczas kapsuła zastępcza nie będzie miała dostępu do tych danych, gdy zostanie uruchomiona na innym węźle, i byłoby rozczarowujące, gdyby te dane zawierały duże zamówienie, którego klient nie ukończył. Aby kapsuły miały dostęp do tych samych danych z dowolnego węzła, potrzebujesz pamięci masowej obejmującej cały klaster.

Kubernetes nie ma wbudowanej pamięci masowej dla całego klastra, ponieważ nie ma jednego rozwiązania, które działałoby w każdym scenariuszu. Aplikacje mają różne wymagania dotyczące pamięci masowej, a poszczególne platformy, na których można uruchomić Kubernetes, mają odmienne możliwości przechowywania. Dane zawsze wymagają równowag między szybkością dostępu a trwałością, a Kubernetes obsługuje to poprzez umożliwienie definiowania różnych klas pamięci masowej zapewnianej przez klaster oraz żądania określonej

klasy pamięci dla aplikacji. W tym rozdziale dowiesz się, jak pracować z różnymi typami pamięci masowej i w jaki sposób Kubernetes tworzy warstwę abstrakcji dla szczegółów implementacji pamięci masowej.

5.1. Jak Kubernetes buduje system plików kontenera?

Kontenery w kapsułach mają system plików skonstruowany przez Kubernetes przy użyciu wielu źródeł. Obraz kontenera zapewnia początkową zawartość systemu plików, a każdy kontener ma warstwę zapisywalnej pamięci masowej, której używa do zapisywania nowych plików lub aktualizowania dowolnych plików z obrazu. (Obrazy Dockera są tylko do odczytu, więc gdy kontener aktualizuje plik na podstawie obrazu, w rzeczywistości aktualizuje kopię pliku we własnej zapisywalnej warstwie). Na rysunku 5.1 pokazałem, jak to wygląda wewnątrz kapsuły.



Rysunek 5.1. *Kontenery nie są świadome tego, że ich system plików jest konstrukcją wirtualną zbudowaną przez Kubernetes*

Aplikacja działająca w kontenerze widzi tylko pojedynczy system plików, do którego ma dostęp z uprawnieniami odczytu i zapisu, a wszystkie szczegóły warstwy są ukryte. Świetnie sprawdza się to przy przenoszeniu aplikacji do Kubernetesa, ponieważ nie trzeba ich zmieniać, aby uruchomić je w kapsułach. Jeśli

jednak Twoje aplikacje zapisują dane, musisz zrozumieć, w jaki sposób wykorzystują pamięć masową, i zaprojektować kapsuły tak, by spełniały ich wymagania. W przeciwnym razie Twoje aplikacje z pozoru będą działały poprawnie, ale narażasz się na utratę danych, gdy wydarzy się coś nieoczekiwanego — np. zrestartowanie kapsuły z nowym kontenerem.

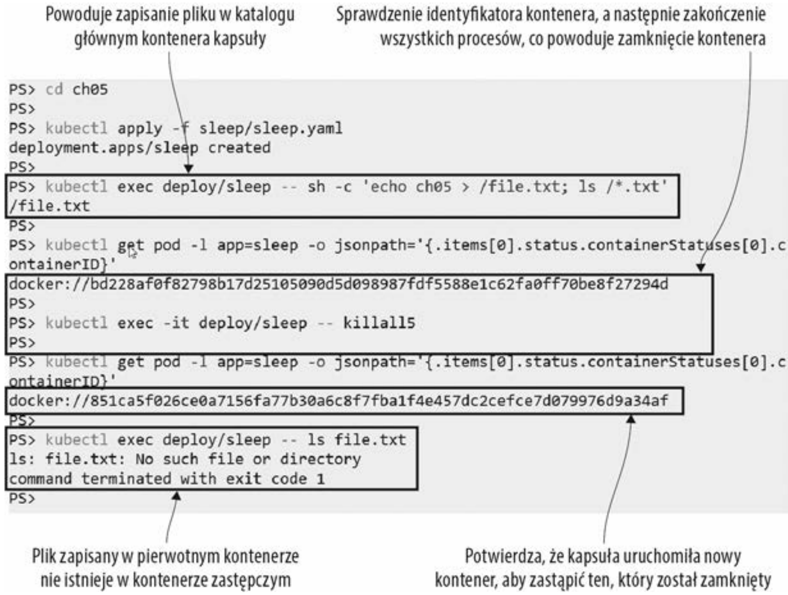
WYPRÓBUJ Jeżeli aplikacja w kontenerze ulegnie awarii i kontener zostanie zamknięty, kapsuła uruchomi kontener zastępczy. Nowy kontener rozpocznie działanie z systemem plików z obrazu kontenera i nową warstwą zapisywalną — znikną wszelkie dane zapisane przez poprzedni kontener w warstwie zapisywalnej.

```
# Przełącz się do katalogu ćwiczeń dla tego rozdziału:
cd ch05

# Wdróż kapsułę sleep:
kubectl apply -f sleep/sleep.yaml

# Zapisz plik wewnątrz kontenera:
kubectl exec deploy/sleep -- sh -c 'echo ch05 > /file.txt; ls /*.txt'

# Sprawdź identyfikator kontenera:
kubectl get pod -l app=sleep -o
        jsonpath='{.items[0].status.containerStatuses[0].containerID}'

# Zamknij wszystkie procesy w tym kontenerze, powodując restart kapsuły:
kubectl exec -it deploy/sleep -- killall5

# Sprawdź identyfikator kontenera zastępczego:
kubectl get pod -l app=sleep -o
  jsonpath='{.items[0].status.containerStatuses[0].containerID}'

# Poszukaj zapisanego pliku — nie znajdziesz go tutaj:
kubectl exec deploy/sleep -- ls file.txt
```

Z tego ćwiczenia zapamiętaj dwie istotne rzeczy. Po pierwsze, system plików kontenera kapsuły jest regulowany cyklem życia kontenera, a nie kapsuły; po drugie, gdy w kontekście Kubernetesa mówimy o restarcie kapsuły, w rzeczywistości odnosi się to do kontenera zastępczego. Jeśli Twoje aplikacje beztrosko zapisują dane w kontenerach, dane te nie są zapisywane na poziomie kapsuły — gdy kapsuła zostanie zrestartowana z nowym kontenerem, wszystkie dane przepadną. Widać to w moich danych wyjściowych, które pokazałem na rysunku 5.2.

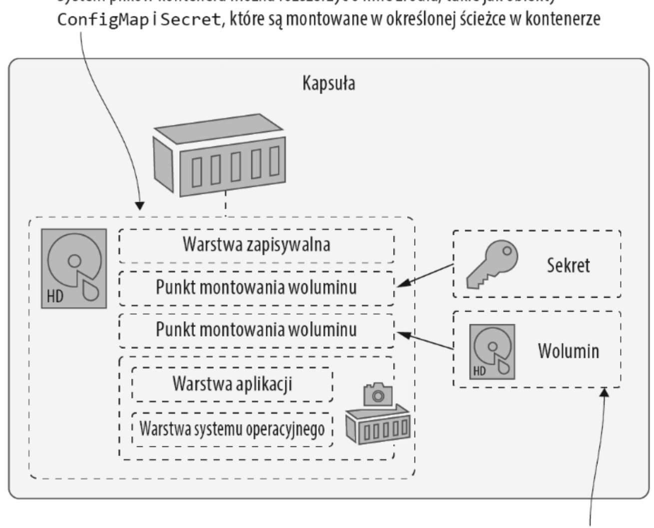
Wiesz już, że Kubernetes może budować system plików kontenera na podstawie innych źródeł — w rozdziale 4. udostępnialiśmy obiekty `ConfigMap` i `Secret` w katalogach systemu plików. Mechanizm ten polega na zdefiniowaniu na poziomie kapsuły woluminu zapewniającego kolejne źródło pamięci masowej, a następnie zamontowaniu go w systemie plików kontenera w określonej ścieżce. Mapy konfiguracji i sekrety to jednostki pamięci masowej z właściwością tylko do odczytu, ale Kubernetes obsługuje wiele innych typów zapisywalnych woluminów.

Kup ksi k Pole ksi k



Rysunek 5.2. *Warstwa zapisywalna jest regulowana cyklem życia kontenera, a nie kapsuły*

Na rysunku 5.3 pokazałem, jak zaprojektować kapsułę używającą woluminu do przechowywania danych, które nie są usuwane po restarcie i mogą być dostępne nawet w całym klastrze.



Rysunek 5.3. *Wirtualny system plików można zbudować z woluminów, które odwołują się do zewnętrznych elementów pamięci masowej*

Woluminy obejmujące całe klastry omówię dalej w rozdziale, ale na razie zacznę od dużo prostszego typu woluminu, który jest nadal przydatny w wielu scenariuszach. W listingu 5.1 pokazałem specyfikację kapsuły korzystającej z typu woluminu o nazwie `EmptyDir`, który jest po prostu pustym katalogiem, ale przechowywanym na poziomie kapsuły, a nie kontenera. Montuje się go jako wolumin w kontenerze, więc jest widoczny jako katalog, ale nie stanowi jednej z warstw obrazu czy kontenera.

Listing 5.1. Plik sleep-with-emptyDir.yaml; prosta specyfikacja woluminu

```
spec:
  containers:
    - name: sleep
      image: kiamol/ch03-sleep
      volumeMounts:
        - name: data              # Montuje wolumin o nazwie data
          mountPath: /data        # w katalogu /data
  volumes:
    - name: data                  # To jest specyfikacja woluminu data
      emptyDir: {}                # typu EmptyDir
```

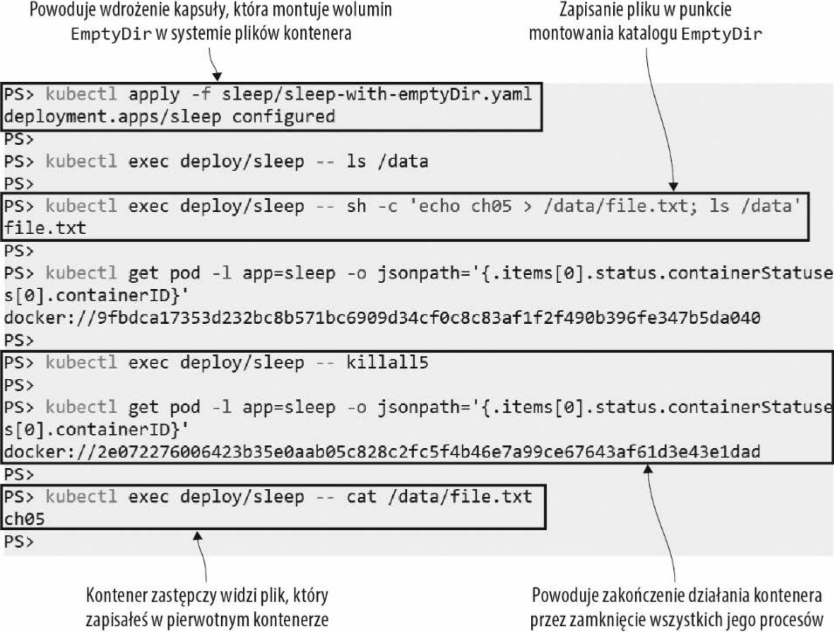
Pusty katalog brzmi jak najmniej przydatny element pamięci masowej, jaki można sobie wyobrazić, lecz w rzeczywistości ma wiele zastosowań, ponieważ ma taki sam cykl życia jak kapsuła. Wszelkie dane przechowywane w wolumenie `EmptyDir` pozostają w kapsule po restarcie, więc kontenery zastępcze mogą uzyskać dostęp do danych zapisanych przez swoje poprzedniki.

WYPRÓBUJ Zaktualizuj wdrożenie `sleep`, korzystając ze specyfikacji z listingu 5.1, dodając wolumin `EmptyDir`. Teraz możesz zapisać dane, zamknąć kontener, a kontener zastępczy będzie mógł odczytać te dane.

```
# Zaktualizuj kapsułę sleep, aby używała woluminu EmptyDir:
kubectl apply -f sleep/sleep-with-emptyDir.yaml

# Wyświetl listę zawartości punktu montowania woluminu:
kubectl exec deploy/sleep -- ls /data

# Utwórz plik w pustym katalogu:
kubectl exec deploy/sleep -- sh -c 'echo ch05 > /data/file.txt; ls /data'

# Sprawdź identyfikator kontenera:
kubectl get pod -l app=sleep -o
  jsonpath='{.items[0].status.containerStatuses[0].containerID}'

# Zamknij procesy kontenera:
kubectl exec deploy/sleep -- killall5

# Sprawdź identyfikator kontenera zastępczego:
kubectl get pod -l app=sleep -o
  jsonpath='{.items[0].status.containerStatuses[0].containerID}'

# Wyświetl zawartość pliku zapisanego przez pierwotny kontener:
kubectl exec deploy/sleep -- cat /data/file.txt
```

Kup ksi k   Pole ksi k

Swoje dane wyjściowe pokazałem na rysunku 5.4. Kontenery widzą po prostu katalog w systemie plików, ale wskazuje on na jednostkę pamięci masowej, która jest częścią kapsuły.



Rysunek 5.4. *Coś tak podstawowego jak pusty katalog jest nadal przydatne, ponieważ może on być współużytkowany przez kontenery*

Woluminów EmptyDir można używać dla dowolnych aplikacji, które do tymczasowego przechowywania wykorzystują systemy plików. Aplikacja może np. wywoływać interfejs API, co zajmuje kilka sekund, a odpowiedź jest ważna przez długi czas. Odpowiedź interfejsu API może zostać zapisana przez aplikację w pliku lokalnym, ponieważ odczyt z dysku jest szybszy niż powtarzanie wywołania interfejsu API. Wolumin EmptyDir jest rozsądnym źródłem lokalnej pamięci podręcznej, gdyż jeśli aplikacja ulegnie awarii, kontener zastępczy nadal będzie miał zbuforowane pliki i odnosił korzyści z przyspieszenia odczytu.

Woluminy EmptyDir współdzielą jedynie cykl życia kapsuły, więc jeśli kapsuła zostanie wymieniona, nowa kapsuła zostanie uruchomiona… z pustym katalogiem. Jeśli chcesz, aby Twoje dane przetrwały restarty kapsuł, możesz zamontować inne typy woluminów, które mają własne cykle życia.

Kup ksi k Pole ksi k

5.2. Przechowywanie danych na węźle za pomocą woluminów i punktów montowania

W tym miejscu praca z danymi staje się trudniejsza niż praca z obliczeniami, ponieważ musimy pomyśleć o tym, czy dane zostaną powiązane z określonym węzłem — co oznacza, że aby zobaczyć wszelkie dane zastępcze, kapsuły będą musiały być uruchamiane na tym węźle — lub czy dane mają dostęp do całego klastra, a kapsuła może być uruchamiana na dowolnym węźle. Kubernetes obsługuje wiele wariantów, ale musisz wiedzieć, czego chcesz i co obsługuje Twój klaster, oraz określić to dla kapsuły.

Najprostszą opcją pamięci masowej jest użycie woluminu, który jest mapowany na katalog na węźle, więc gdy kontener zapisuje dane w punkcie montowania woluminu, są one tak naprawdę zapisywane w znanym katalogu na dysku węzła. Zademonstruję to, uruchamiając najpierw prawdziwą aplikację, która używa woluminu `EmptyDir` do buforowania danych w pamięci podręcznej, i omówię związane z tym ograniczenia, a potem zaktualizuję ją, aby korzystała z pamięci masowej na poziomie węzła.

WYPRÓBUJ Uruchom aplikację internetową korzystającą z komponentu proxy, co poprawia wydajność. Aplikacja internetowa jest uruchomiona w kapsule z wewnętrzną usługą, a proxy działa w innej kapsule, która jest publicznie dostępna w usłudze `LoadBalancer`.

```
# Wdróż aplikację Pi:
kubectl apply -f pi/v1/

# Poczekaj, aż kapsuła będzie gotowa:
kubectl wait --for=condition=Ready pod -l app=pi-web

# Poszukaj adresów URL z usługi LoadBalancer:
kubectl get svc pi-proxy -o
  jsonpath='http://{.status.loadBalancer.ingress[0].*}:8080/?dp=30000'

# Teraz przejdź do tego adresu URL, poczekaj na odpowiedź, a następnie odśwież stronę

# Sprawdź pamięć podręczną proxy:
kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache
```

Jest to typowa konfiguracja dla aplikacji internetowych, w której serwer proxy zwiększa wydajność przez obsługiwanie odpowiedzi bezpośrednio ze swojej lokalnej pamięci podręcznej, a dodatkowo zmniejsza obciążenie aplikacji internetowej. Swoje dane wyjściowe pokazałem na rysunku 5.5. Wygenerowanie odpowiedzi dla pierwszego obliczenia liczby pi zajęło ponad 2 sekundy, a odświeżenie było prawie natychmiastowe, ponieważ pochodziło z serwera proxy i nie było wymagane ponowne obliczanie.

Kup ksi k Pole ksi k

Rysunek 5.5. *Buforowanie plików w woluminie EmptyDir oznacza, że pamięć podręczna potrafi przetrwać restarty kapsuł*

Użycie woluminu `EmptyDir` może być rozsądnym podejściem do tego rodzaju aplikacji, ponieważ dane przechowywane w woluminie nie są kluczowe. Jeśli nastąpi restart kapsuły, pamięć podręczna przetrwa, a nowy kontener proxy będzie mógł serwować odpowiedzi zapisane w pamięci podręcznej przez poprzedni kontener. Jeżeli zastąpiona zostanie kapsuła, pamięć podręczna zostanie utracona. Kapsuła zastępcza uruchomi się z pustym katalogiem pamięci podręcznej, ale nie jest ona wymagana — aplikacja nadal działa poprawnie; po prostu z początku działa powoli, aż pamięć podręczna zostanie ponownie zapełniona.

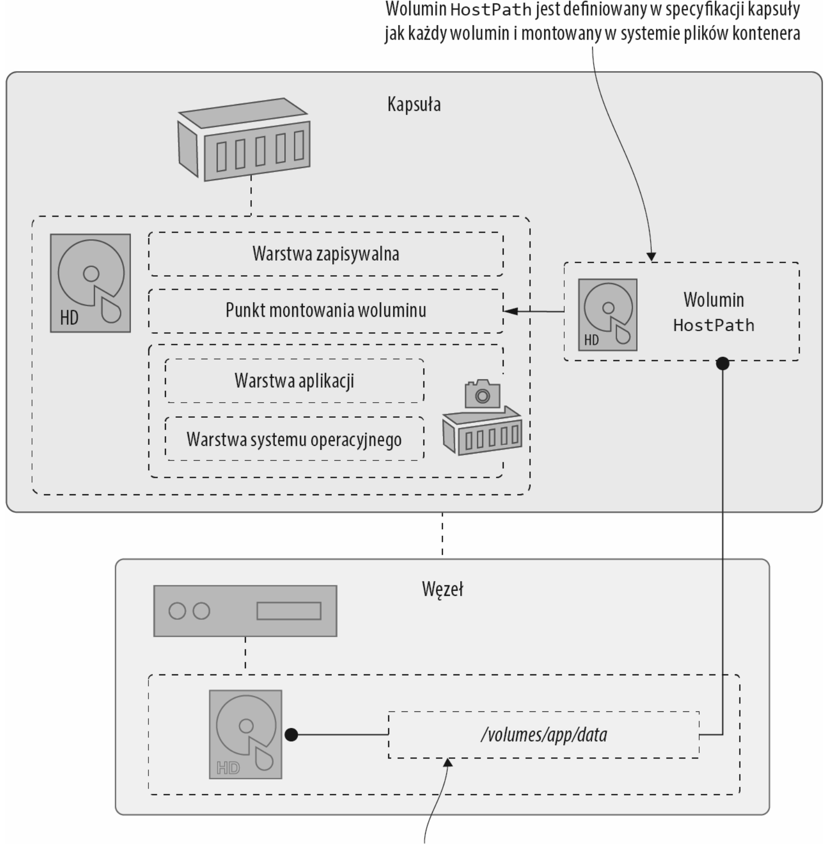
WYPRÓBUJ Usuń kapsułę proxy. Zostanie zastąpiona, ponieważ jest zarządzana przez kontroler wdrożenia. Kapsuła zastępcza zostanie uruchomiona z nowym woluminem `EmptyDir`, co dla tej aplikacji oznacza pustą pamięć podręczną proxy, więc żądania będą wysyłane do kapsuły internetowej.

```
# Usuń kapsułę proxy:
kubectl delete pod -l app=pi-proxy

# Sprawdź katalog pamięci podręcznej kapsuły zastępczej:
kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache

# Teraz odśwież swoją przeglądarkę na adresie URL aplikacji Pi
```

Swoje dane wyjściowe pokazałem na rysunku 5.6. Rezultat jest taki sam, ale znów musiałem czekać ponad 2 sekundy na obliczenie go przez aplikację internetową, ponieważ zastępcza kapsuła proxy została uruchomiona bez pamięci podręcznej.

Rysunek 5.6. *Nowa kapsuła jest uruchamiana z nowym pustym katalogiem*

Następny poziom utrwalania polega na użyciu woluminu, który jest mapowany na katalog na dysku węzła — Kubernetes nazywa to woluminem `HostPath`. Jest on określany jako wolumin w kapsule i montowany w systemie plików kontenera w standardowy sposób. Gdy kontener zapisuje dane w katalogu punktu montowania, w rzeczywistości są one zapisywane na dysku na węźle. Na rysunku 5.7 pokazałem relację między węzłem, kapsułą i woluminem.



Rysunek 5.7. *Woluminy HostPath zachowują dane z podmienianych kapsuł, ale tylko wtedy, gdy kapsuły używają tego samego węzła*

Woluminy `HostPath` mogą być przydatne, jednak trzeba zdawać sobie sprawę z ich ograniczeń. Dane są fizycznie przechowywane na węźle i to wszystko w tej kwestii. Kubernetes nie replikuje w magiczny sposób tych danych we wszystkich innych węzłach w klastrze. W listingu 5.2 przedstawiłem zaktualizowaną specyfikację kapsuły dla internetowego proxy sieci, która zamiast `EmptyDir` używa woluminu `HostPath`. Gdy kontener proxy zapisuje pliki pamięci podręcznej w lokalizacji */data/nginx/cache*, są one faktycznie zapisywane na węźle w lokalizacji */volume/nginx/cache*.

Listing 5.2. Plik nginx-with-hostPath.yaml; montowanie woluminu HostPath

```
spec:                    # To jest skrócona specyfikacja kapsuły;
  containers:            # pełna wersja zawiera również punkt montowania mapy konfiguracji
    - image: nginx:1.17-alpine
      name: nginx
```

```
      ports:
        - containerPort: 80
      volumeMounts:
        - name: cache-volume
          mountPath: /data/nginx/cache      # Ścieżka pamięci podręcznej proxy
  volumes:
    - name: cache-volume
      hostPath:                             # Użycie katalogu na węźle
        path: /volumes/nginx/cache          # Ścieżka woluminu na węźle
        type: DirectoryOrCreate             # Tworzy ścieżkę, jeśli ta nie istnieje
```

Ta metoda zwiększa poziom utrwalania danych, rozszerzając trwałość poza cykl życia kapsuły na cykl życia dysku węzła, pod warunkiem że kapsuły zastępcze zawsze będą uruchamiane na tym samym węźle. Tak będzie w przypadku jednowęzłowego klastra Twojego laboratorium, ponieważ ma on tylko jeden węzeł. Kapsuły zastępcze podczas uruchamiania ładują wolumin `HostPath`, a jeśli jest on zapełniony danymi z pamięci podręcznej z poprzedniej kapsuły, nowy serwer proxy może od razu rozpocząć serwowanie danych z pamięci podręcznej.

WYPRÓBUJ Zaktualizuj wdrożenie serwera proxy, aby wykorzystywało specyfikację kapsuły z listingu 5.2, a następnie pobaw się trochę aplikacją i usuń potem kapsułę. Kapsuła zastępcza będzie odpowiadać na żądania poprzez korzystanie z istniejącej pamięci podręcznej.

```
# Zaktualizuj kapsułę proxy, by używała woluminu HostPath:
kubectl apply -f pi/nginx-with-hostPath.yaml

# Wyświetl listę zawartości katalogu pamięci podręcznej:
kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache

# Teraz otwórz adres URL aplikacji

# Usuń kapsułę proxy:
kubectl delete pod -l app=pi-proxy

# Sprawdź katalog pamięci masowej kapsuły zastępczej:
kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache

# Odśwież przeglądarkę
```

Swoje dane wyjściowe pokazałem na rysunku 5.8. Uzyskanie odpowiedzi na pierwsze żądanie zajęło ponad 2 sekundy, ale odświeżenie było prawie natychmiastowe, ponieważ nowa kapsuła odziedziczyła po poprzedniej kapsule zbuforowaną odpowiedź przechowywaną na węźle.

Oczywisty problem związany z woluminami `HostPath` polega na tym, że nie mają one sensu w klastrze z więcej niż jednym węzłem, czyli w prawie każdym klastrze poza prostym środowiskiem laboratoryjnym. W specyfikacji kapsuły można co prawda zawrzeć wymóg uruchamiania jej zawsze na tym samym węźle, aby mieć pewność, że trafi tam, gdzie znajdują się dane. Ogranicza to jednak elastyczność rozwiązania — jeżeli węzeł przejdzie w tryb offline, kapsuła nie zostanie uruchomiona i tracimy aplikację.

Kup ksi k Pole ksi k

Powoduje użycie woluminu `HostPath` jako pamięci podręcznej serwera proxy

Katalog nie istnieje, dlatego Kubernetes tworzy pusty katalog na węźle hosta

```
PS> kubectl apply -f pi/nginx-with-hostPath.yaml
deployment.apps/pi-proxy configured
PS>
PS> kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache
total 0
PS>
PS> kubectl delete pod -l app=pi-proxy
pod "pi-proxy-7fc8dc9584-m8fw8" deleted
PS>
PS> kubectl exec deploy/pi-proxy -- ls -l /data/nginx/cache
total 0
drwx------    3 nginx    nginx           60 Jun 17 07:48 1
```

Korzystanie z aplikacji wypełnia wolumin pamięci podręcznej. Kapsuła zastępcza działa na tym samym węźle i używa tego samego woluminu, więc ma dostęp do pamięci podręcznej zapisanej przez poprzednią kapsułę

Rysunek 5.8. *W jednowęzłowym klastrze kapsuły zawsze działają na tym samym węźle, więc wszystkie mogą używać woluminu HostPath*

Mniej dostrzegalnym problemem jest to, że dana metoda tworzy dość sporą lukę w zabezpieczeniach. Kubernetes nie ogranicza, które katalogi na węźle mają być dostępne dla woluminów `HostPath`. Specyfikacja kapsuły pokazana w listingu 5.3 jest całkowicie poprawna, ale sprawia, że kontener kapsuły ma dostęp do całego systemu plików na węźle.

Listing 5.3. Plik sleep-with-hostPath.yaml; kapsuła z pełnym dostępem do dysku węzła

```
pec:
  containers:
    - name: sleep
      image: kiamol/ch03-sleep
      volumeMounts:
        - name: node-root
          mountPath: /node-root
  volumes:
    - name: node-root
      hostPath:
        path: /                 # Musi istnieć katalog główny
        type: Directory         # systemu plików węzła
```

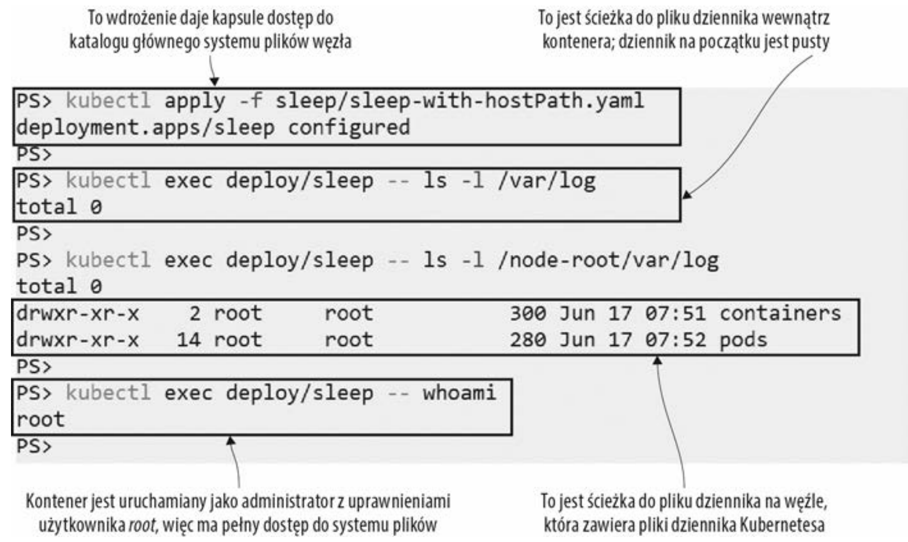
Każdy, kto ma możliwość utworzenia kapsuły na podstawie tej specyfikacji, będzie miał dostęp do całego systemu plików węzła, na którym kapsuła zostanie uruchomiona. Możesz ulec pokusie, aby użyć tego punktu montowania woluminu jako szybkiego sposobu na odczytywanie wielu ścieżek na hoście, ale jeśli Twoja aplikacja zostanie zhakowana i atakujący będzie mógł wykonywać polecenia w kontenerze, wówczas uzyska również dostęp do dysku węzła.

Kup ksi k Pole ksi k

WYPRÓBUJ Uruchom kapsułę z wykorzystaniem pliku YAML-a pokazanego w listingu 5.3, a następnie wykonaj kilka poleceń w kontenerze kapsuły, aby zbadać system plików węzła.

```
# Uruchom kapsułę z punktem montowania woluminu na hoście:
kubectl apply -f sleep/sleep-with-hostPath.yaml

# Sprawdź pliki dzienników kontenera:
kubectl exec deploy/sleep -- ls -l /var/log

# Korzystając z tego woluminu, sprawdź pliki dzienników na węźle:
kubectl exec deploy/sleep -- ls -l /node-root/var/log

# Sprawdź użytkownika kontenera:
kubectl exec deploy/sleep -- whoami
```

Jak pokazałem na rysunku 5.9, kontener kapsuły widzi pliki dziennika na węźle, który w tym przypadku zawiera również dzienniki Kubernetesa. To raczej nieszkodliwe, ale ten kontener jest uruchomiony z uprawnieniami użytkownika *root*, który jest mapowany na użytkownika *root* na węźle, więc kontener ma pełny dostęp do systemu plików.



Rysunek 5.9. *Niebezpieczeństwo! Montowanie woluminu HostPath może zapewnić pełny dostęp do danych na węźle*

Jeśli wydaje się to fatalnym pomysłem, pamiętaj, że Kubernetes jest platformą z szerokim zakresem funkcjonalności, które mają być dopasowane do wielu różnych aplikacji. Możesz mieć np. starszą aplikację, która musi uzyskiwać dostęp do określonych ścieżek plików na węźle, na którym jest uruchomiona, a wolumin `HostPath` to umożliwia. W takim scenariuszu można przyjąć bezpieczniejsze podejście, używając woluminu z dostępem do jednej ścieżki na węźle — zadeklarowanie

Kup ksi k Pole ksi k

podścieżek dla punktu montowania woluminu ogranicza elementy widoczne dla kontenera. Pokazałem to w listingu 5.4.

Listing 5.4. Plik sleep-with-hostPath-subPath.yaml; ograniczanie punktów montowania za pomocą podścieżek

```
spec:
  containers:
    - name: sleep
      image: kiamol/ch03-sleep
      volumeMounts:
        - name: node-root               # Nazwa woluminu do zamontowania
          mountPath: /pod-logs          # Ścieżka docelowa dla kontenera
          subPath: var/log/pods         # Ścieżka źródłowa w obrębie woluminu
        - name: node-root
          mountPath: /container-logs
          subPath: var/log/containers
  volumes:
    - name: node-root
      hostPath:
        path: /
        type: Directory
```

W tym przypadku wolumin nadal jest zdefiniowany w ścieżce głównej na węźle, ale można uzyskać do niej dostęp jedynie przez punkty montowania woluminów w kontenerze, które są ograniczone do zdefiniowanych ścieżek podrzędnych. Pomiędzy specyfikacją woluminu a specyfikacją montowania istnieje duża elastyczność w budowaniu i mapowaniu systemu plików kontenera.

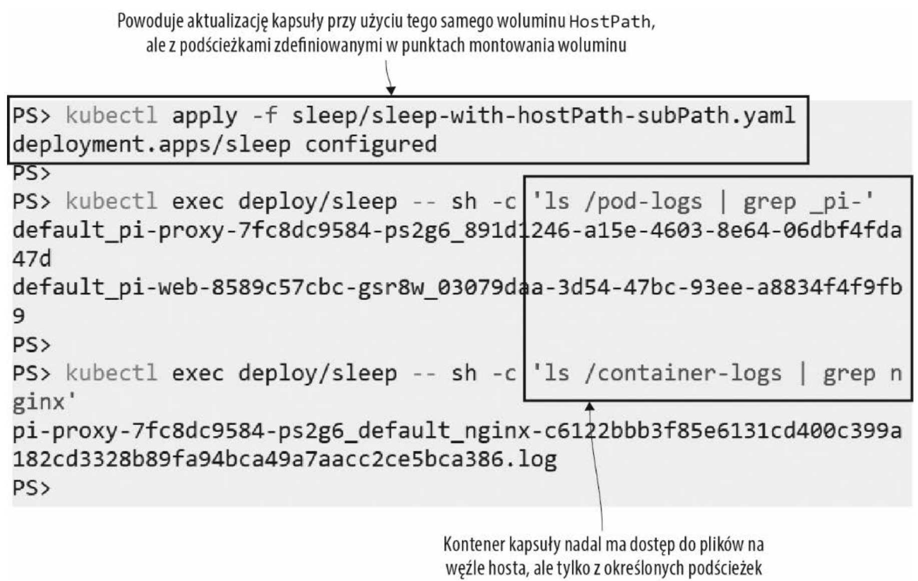
WYPRÓBUJ Zaktualizuj kapsułę `sleep` w taki sposób, by punkt montowania woluminu kontenera był ograniczony do podścieżek zdefiniowanych w listingu 5.4. Następnie sprawdź zawartości plików.

```
# Zaktualizuj specyfikację kapsuły:
kubectl apply -f sleep/sleep-with-hostPath-subPath.yaml

# Sprawdź dzienniki kapsuły na węźle:
kubectl exec deploy/sleep -- sh -c 'ls /pod-logs | grep _pi-'

# Sprawdź dzienniki kontenera:
kubectl exec deploy/sleep -- sh -c 'ls /container-logs | grep nginx'
```

W tym ćwiczeniu eksplorację systemu plików węzła można przeprowadzić jedynie za pośrednictwem punktów montowania do katalogów dziennika. Jak pokazałem na rysunku 5.10, kontener może uzyskiwać dostęp tylko do plików w podścieżkach.

Woluminy `HostPath` to dobry sposób na rozpoczęcie pracy z aplikacjami stanowymi. Są łatwe w użyciu i działają w ten sam sposób w każdym klastrze. Przydają się także w rzeczywistych aplikacjach, ale tylko wtedy, gdy używają stanu do tymczasowego przechowywania. Aby zapewnić przechowywanie trwałe, musimy przejść do woluminów, do których dostęp może uzyskać dowolny węzeł w klastrze.

Kup ksi k Pole ksi k



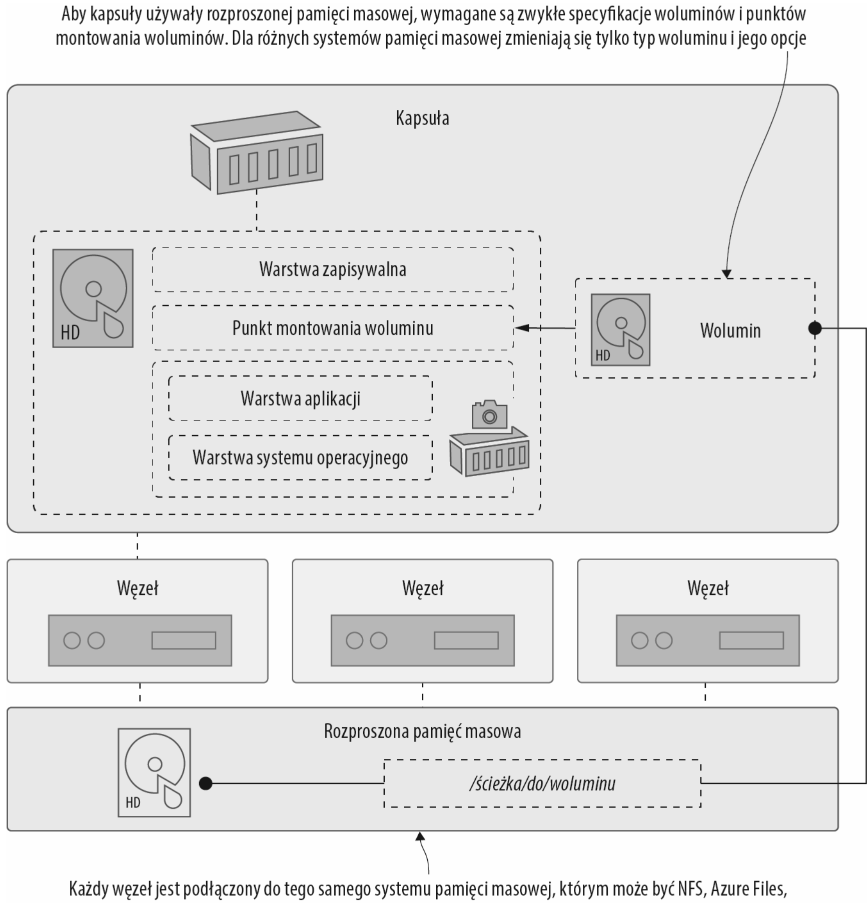
Rysunek 5.10. *Limitowanie dostępu do woluminów za pomocą podścieżek ogranicza możliwości kontenera*

5.3. Użycie woluminów trwałych oraz żądań do przechowywania danych dla całego klastra

Klaster Kubernetesa przypomina pulę zasobów: składa się z wielu węzłów udostępniających klastrowi pewną moc obliczeniową oraz ilość pamięci, których Kubernetes używa do uruchamiania aplikacji. Pamięć masowa jest po prostu kolejnym zasobem udostępnianym przez Kubernetes aplikacji, ale pamięć masową na poziomie całego klastra można zapewnić tylko wtedy, gdy węzły mogą podłączyć się do rozproszonego systemu pamięci masowej. Na rysunku 5.11 pokazałem, w jaki sposób kapsuły mogą uzyskać dostęp do woluminów z dowolnego węzła, jeśli woluminy korzystają z rozproszonej pamięci masowej.

Kubernetes obsługuje wiele typów woluminów obsługiwanych przez systemy rozproszonej pamięci masowej: klastry AKS mogą korzystać z Azure Files lub Azure Disk, klastry EKS mogą korzystać z Elastic Block Store, a w centrum danych można używać prostych udziałów NFS (ang. *Network File System*) lub sieciowego systemu plików takiego jak GlusterFS. Wszystkie te systemy mają różne wymagania konfiguracyjne i można je określić w specyfikacji woluminu dla kapsuły. Powoduje to jednak ścisłe powiązanie specyfikacji aplikacji z jedną implementacją pamięci masowej, a Kubernetes zapewnia bardziej elastyczne podejście.

Kup książkę Poleć książkę



Rysunek 5.11. *Rozproszona pamięć masowa zapewnia kapsułom dostęp do danych z dowolnego węzła, ale wymaga wsparcia określonej platformy*

Kapsuły są warstwą abstrakcji dla warstwy obliczeniowej, a usługi — dla warstwy sieciowej. W warstwie pamięci masowej abstrakcje stanowią woluminy trwałe (ang. *PersistentVolume* — PV) oraz żądania woluminów trwałych (ang. *PersistentVolumeClaim* — PVC). Wolumin trwały to obiekt Kubernetesa, który definiuje dostępną część pamięci masowej. Administrator klastra może utworzyć zestaw woluminów trwałych, z których każdy będzie zawierał specyfikację woluminu dla bazowego systemu pamięci masowej. W listingu 5.5 przedstawiłem specyfikację woluminu trwałego, która używa pamięci masowej NFS.

Listing 5.5. Plik persistentVolume-nfs.yaml; wolumin obsługiwany przez punkt montowania NFS

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
  name: pv01                      # Ogólna jednostka pamięci masowej z ogólną nazwą

spec:
  capacity:
    storage: 50Mi                 # Ilość pamięci masowej oferowanej przez PV
  accessModes:                    # Sposób uzyskiwania dostępu do woluminu przez kapsuły
    - ReadWriteOnce               # Może być używany tylko przez jedną kapsułę
  nfs:                            # Ten wolumin trwały jest obsługiwany przez NFS
    server: nfs.my.network        # Nazwa domenowa serwera NFS
    path: "/kubernetes-volumes"   # Ścieżka do udziału NFS
```

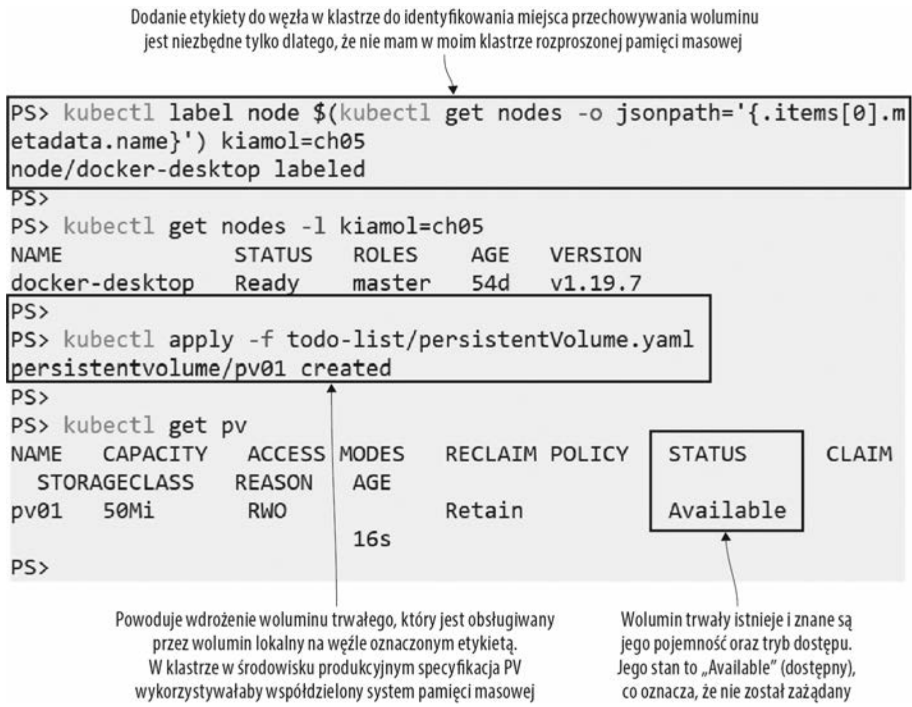
Nie będziesz mógł wdrożyć tej specyfikacji w swoim środowisku laboratoryjnym, chyba że masz akurat w sieci serwer NFS z nazwą domenową `nfs.my.network` i udziałem o nazwie `kubernetes-volume`. Twój Kubernetes może być uruchomiony na dowolnej platformie, więc w kolejnych ćwiczeniach użyję woluminu lokalnego, który będzie działał wszędzie. (Gdybym używał Azure Files, ćwiczenia działałyby tylko w klastrze AKS, ponieważ EKS i Docker Desktop oraz inne dystrybucje Kubernetesa nie są skonfigurowane dla typów woluminów platformy Azure).

WYPRÓBUJ Utwórz PV korzystający z lokalnego woluminu. PV obejmuje cały klaster, ale dany wolumin jest lokalny dla jednego węzła, dlatego trzeba zagwarantować, aby PV był powiązany z węzłem, w którym będzie się znajdował wolumin. Zrobię to za pomocą etykiet.

```
# Zastosuj niestandardową etykietę do pierwszego węzła w klastrze:
kubectl label node $(kubectl get nodes -o
  jsonpath='{.items[0].metadata.name}') kiamol=ch05

# Sprawdź węzły za pomocą selektora etykiet:
kubectl get nodes -l kiamol=ch05

# Wdróż PV, który używa woluminu lokalnego na oznaczonym etykietą węźle:
kubectl apply -f todo-list/persistentVolume.yaml

# Sprawdź ten wolumin trwały:
kubectl get pv
```

Swoje dane wyjściowe pokazałem na rysunku 5.12. Nadawanie węzłom etykiet jest konieczne tylko dlatego, że nie używam rozproszonego systemu pamięci masowej; zwykle wystarczy określić konfigurację woluminu NFS lub Azure Disk, która jest dostępna z dowolnego węzła. Wolumin lokalny istnieje tylko na jednym węźle, a PV identyfikuje ten węzeł za pomocą etykiety.

Teraz ten wolumin trwały istnieje w klastrze jako dostępna jednostka pamięci masowej ze znanym zestawem cech, w tym rozmiarem i trybem dostępu. Kapsuły nie mogą użyć tego PV bezpośrednio — ten zamiar muszą zgłosić za pośrednictwem żądania woluminu trwałego (ang. *PersistentVolumeClaim* — PVC). PVC jest abstrakcją pamięci masowej używaną przez kapsuły; polega to po prostu na zażądaniu dla aplikacji pewnej ilości miejsca w pamięci masowej. Kubernetes



Rysunek 5.12. *Jeśli nie masz rozproszonej pamięci masowej, możesz wykorzystać sztuczkę polegającą na podpięciu PV do woluminu lokalnego*

dopasowuje PVC do PV, a obsługę szczegółowych kwestii bazowych woluminu pozostawia PV. W listingu 5.6 przedstawiłem żądanie pamięci masowej, które zostanie dopasowane do utworzonego przeze mnie woluminu trwałego.

Listing 5.6. Plik postgres-persistentVolumeClaim.yaml; PVC dopasowujące PV

```
piVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc          # To żądanie zostanie użyte przez określoną aplikację
spec:
  accessModes:                # Wymagany tryb dostępu
    - ReadWriteOnce
  resources:
    requests:
      storage: 40Mi           # Żądana ilość pamięci masowej
  storageClassName: ""        # Pusta klasa oznacza, że wolumin trwały musi istnieć
```

Specyfikacja PVC zawiera tryb dostępu oraz ilość i klasę pamięci masowej. Jeśli nie została określona żadna klasa pamięci masowej, Kubernetes będzie próbował znaleźć istniejący wolumin trwały, który odpowiada wymaganiom podanym w żądaniu. Jeśli znajdzie dopasowanie, PVC zostaje powiązane z danym PV

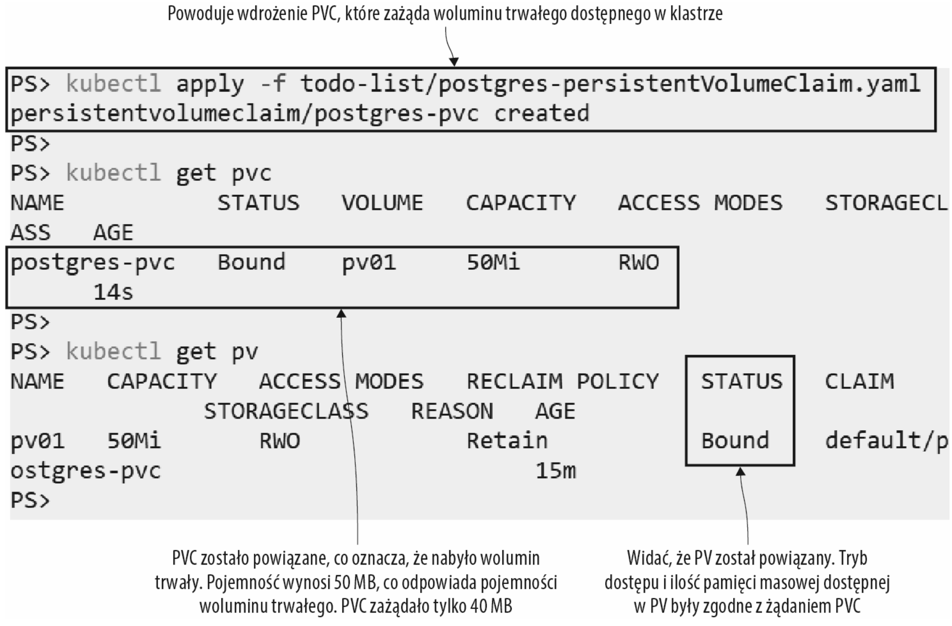
Kup ksi k Pole ksi k

— jest to powiązanie „jeden do jednego”, dlatego po przydzieleniu PV przestaje być dostępny dla innych PVC.

WYPRÓBUJ Wdróż PVC z listingu 5.6. Jego wymagania spełnia wolumin trwały, który utworzyłeś w poprzednim ćwiczeniu, więc z tym woluminem zostanie powiązane to żądanie.

```
# Utwórz nowe PVC, które zostanie powiązane z określonym PV:
kubectl apply -f todo-list/postgres-persistentVolumeClaim.yaml

# Sprawdź żądania woluminów trwałych:
kubectl get pvc

# Sprawdź woluminy trwałe:
kubectl get pv
```

Swoje dane wyjściowe pokazałem na rysunku 5.13. Można na nim zobaczyć powiązanie „jeden do jednego”: żądanie woluminu trwałego zostało powiązane z określonym woluminem trwałym, a wolumin został powiązany z żądaniem.



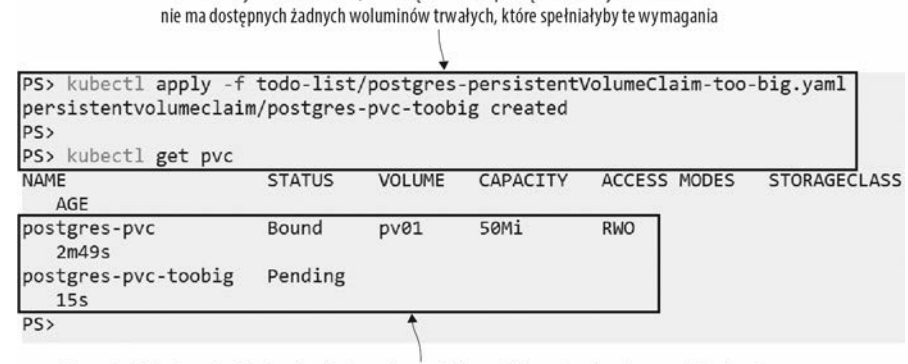
Rysunek 5.13. *Woluminy trwałe to po prostu jednostki pamięci masowej w klastrze; są przypisywane do aplikacji za pomocą żądań woluminów trwałych*

Jest to tzw. statyczna alokacja pamięci masowej (ang. *static provisioning*), w której wolumin trwały musi zostać utworzony bezpośrednio, aby Kubernetes mógł dokonać jego powiązania. Jeśli podczas tworzenia PVC nie będzie żadnego odpowiadającego mu PV, żądanie i tak zostanie utworzone, ale nie będzie się nadawało do użytku. Pozostanie w systemie, czekając na utworzenie woluminu trwałego spełniającego jego wymagania.

WYPRÓBUJ PV w Twoim klastrze został już powiązany z żądaniem, więc nie można go ponownie użyć. Utwórz kolejne żądanie woluminu trwałego, które pozostanie niepowiązane.

```
# Utwórz PVC, które nie odpowiada żadnemu z dostępnych woluminów trwałych:
kubectl apply -f todo-list/postgres-persistentVolumeClaim-too-big.yaml

# Sprawdź żądania:
kubectl get pvc
```

Na rysunku 5.14 pokazałem, że nowe PVC ma status `Pending` (oczekujące). Będzie pozostawać w tym stanie, dopóki nie pojawi się w klastrze wolumin trwały o pojemności co najmniej 100 MB, co odpowiada wymaganiom ilości pamięci masowej tego PVC.



Rysunek 5.14. *W przypadku alokacji statycznej PVC pozostanie niezdatne do użytku, dopóki nie pojawi się wolumin trwały, z którym będzie można powiązać to żądanie*

Zanim kapsuła będzie mogła użyć PVC, musi ono zostać powiązane. Jeśli wdrożysz kapsułę, która odwołuje się do niepowiązanego PVC, kapsuła będzie pozostawać w stanie oczekiwania, dopóki PVC nie zostanie powiązane, a tym samym aplikacja nie zostanie uruchomiona, póki nie uzyska wymaganej pamięci masowej. Pierwsze utworzone żądanie woluminu trwałego zostało powiązane, więc może zostać użyte, ale tylko przez jedną kapsułę. Tryb dostępu żądania to `ReadWriteOnce`, co oznacza, że wolumin jest zapisywalny, ale może go zamontować tylko jedna kapsuła. W listingu 5.7 pokazałem skróconą specyfikację kapsuły dla bazy danych Postgres, która do zapewnienia pamięci masowej używa PVC.

Listing 5.7. Plik todo-db.yaml; specyfikacja kapsuły korzystającej z PVC

```
spec:
  containers:
    - name: db
```

```
      image: postgres:11.6-alpine
      volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql/data
  volumes:
    - name: data
      persistentVolumeClaim:        # Wolumin korzystający z PVC
        claimName: postgres-pvc     # PVC, które ma zostać użyte
```

Teraz mamy wszystkie elementy potrzebne do wdrożenia kapsuły bazy danych Postgres z użyciem woluminu, który może (ale nie musi) być obsługiwany przez rozproszoną pamięć masową. Projektant aplikacji jest właścicielem specyfikacji kapsuły oraz żądania woluminu trwałego, ale nie przejmuje się samym woluminem trwałym — ten element jest zależny od infrastruktury klastra Kubernetesa i może być zarządzany przez inny zespół. W naszym środowisku laboratoryjnym jesteśmy właścicielami wszystkiego. Muszę zrobić jeszcze jeden krok: utworzyć ścieżkę katalogu na węźle, którego ma używać wolumin.

WYPRÓBUJ W prawdziwym klastrze Kubernetesa prawdopodobnie nie będziesz mieć dostępu do logowania się do węzłów, musisz więc użyć pewnej sztuczki i uruchomić kapsułę `sleep`, która ma punkt montowania `HostPath` w katalogu głównym węzła. Przy użyciu tego punktu montowania utworzysz katalog.

```
# Uruchom kapsułę sleep, która ma dostęp do dysku węzła:
kubectl apply -f sleep/sleep-with-hostPath.yaml

# Poczekaj, aż kapsuła będzie gotowa:
kubectl wait --for=condition=Ready pod -l app=sleep

# Utwórz ścieżkę katalogu na węźle, którego ma używać PV:
kubectl exec deploy/sleep -- mkdir -p /node-root/volumes/pv01
```

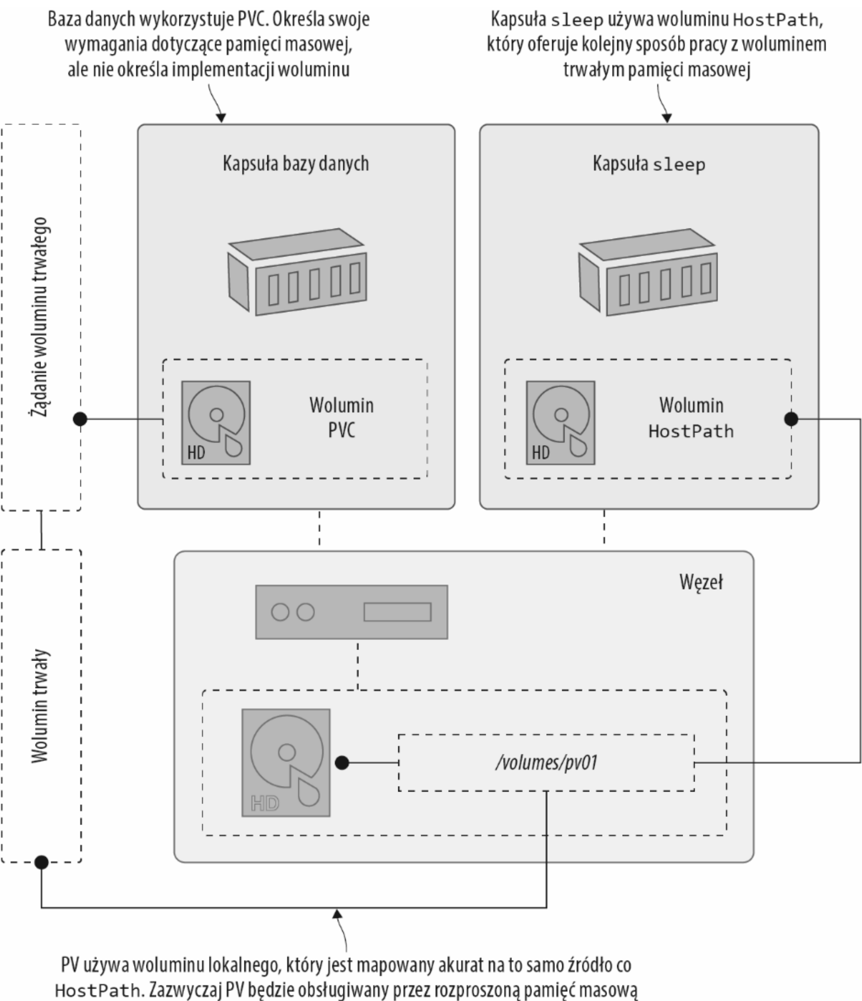
Na rysunku 5.15 pokazałem kapsułę `sleep` uruchomioną z uprawnieniami użytkownika *root*, dzięki czemu za jej pośrednictwem mogę utworzyć katalog na węźle, mimo że nie mam bezpośredniego dostępu do tego węzła.

```
PS> kubectl apply -f sleep/sleep-with-hostPath.yaml
deployment.apps/sleep configured
PS>
PS> kubectl exec deploy/sleep -- mkdir -p /node-root/volumes/pv01
PS>
```

To jest ścieżka na hoście, której ma używać wolumin trwały. Możemy ją utworzyć za pośrednictwem kapsuły, która ma dostęp do systemu plików węzła

Rysunek 5.15. *W tym przykładzie HostPath jest alternatywnym sposobem uzyskania dostępu do źródła PV na węźle*

Wszystko jest już gotowe, aby uruchomić aplikację listy zadań z trwałym magazynem. Zwykle nie będziesz musiał wykonywać tylu kroków, ponieważ będziesz wiedział, jakie możliwości oferuje Twój klaster. Ja jednak nie znam funkcjonalności Twojego klastra, więc te ćwiczenia będą działać na każdym klastrze i są przydatnym wprowadzeniem do wszystkich zasobów pamięci masowej. Na rysunku 5.16 pokazałem, co wdrożyłem do tej pory, wraz z bazą danych, którą zamierzam wdrożyć.



Rysunek 5.16. *Odrobinę skomplikowane — mapowanie PV i HostPath na tę samą lokalizację pamięci masowej*

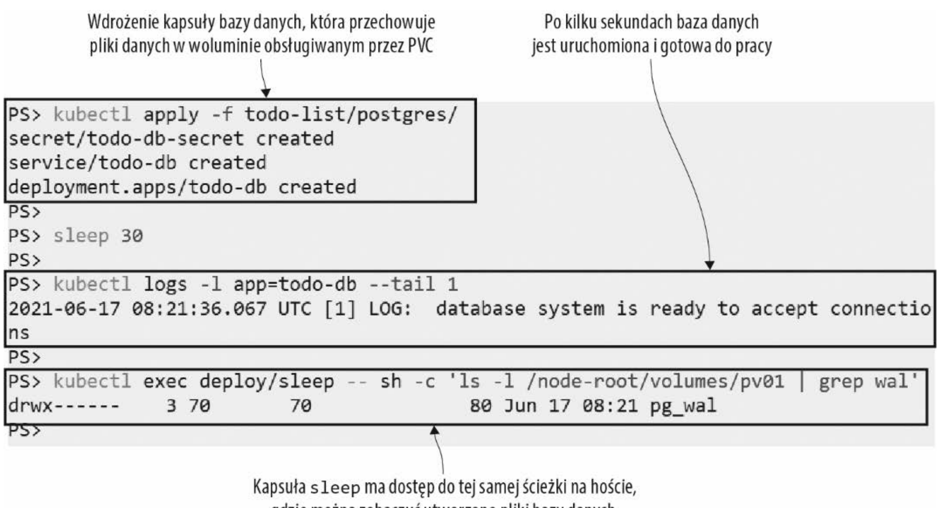
Uruchommy bazę danych. Po utworzeniu kontener Postgres montuje w kapsule wolumin obsługiwany przez PVC. Ten nowy kontener bazy danych łączy się z pustym woluminem, więc po uruchomieniu zainicjuje bazę danych i utworzy

dziennik rejestrowania z wyprzedzeniem (ang. *write-ahead log* — WAL), który jest głównym plikiem danych. Kapsuła bazy danych Postgres nie ma informacji o tym, że PVC jest obsługiwane przez lokalny wolumin na węźle, gdzie uruchomiona jest również kapsuła `sleep`, której możemy użyć do przeglądania plików tej bazy danych.

WYPRÓBUJ Wdróż bazę danych i daj jej czas na zainicjowanie plików danych. Potem za pośrednictwem kapsuły `sleep` sprawdź, co zostało zapisane w woluminie.

```
# Wdróż bazę danych:
kubectl apply -f todo-list/postgres/

# Poczekaj, aż Postgres przeprowadzi inicjowanie:
sleep 30

# Sprawdź dzienniki bazy danych:
kubectl logs -l app=todo-db --tail 1

# Sprawdź pliki danych w woluminie:
kubectl exec deploy/sleep -- sh -c 'ls -l /node-root/volumes/pv01 | grep wal'
```

Swoje dane wyjściowe pokazałem na rysunku 5.17. Widać, że serwer bazy danych uruchamia się poprawnie i po zapisaniu wszystkich swoich plików danych w woluminie czeka na połączenia.



Rysunek 5.17. *Kontener bazy danych zapisuje w lokalnej ścieżce danych, ale w rzeczywistości jest to punkt montowania dla PVC*

Kup ksi k Pole ksi k

Ostatnią rzeczą do zrobienia jest uruchomienie aplikacji, przetestowanie jej i potwierdzenie, że dane będą nadal istnieć, gdy kapsuła bazy danych zostanie zastąpiona.

WYPRÓBUJ Uruchom kapsułę internetową dla aplikacji `to-do`, która łączy się z bazą danych Postgres.

```
# Wdróż internetowe komponenty aplikacji:
kubectl apply -f todo-list/web/

# Poczekaj na kapsułę internetową:
kubectl wait --for=condition=Ready pod -l app=todo-web

# Pobierz z usługi adres URL aplikacji:
kubectl get svc todo-web -o
  jsonpath='http://{.status.loadBalancer.ingress[0].*}:8081/new'

# Otwórz stronę aplikacji i dodaj nowy element

# Usuń kapsułę bazy danych:
kubectl delete pod -l app=todo-db

# Sprawdź zawartość woluminu na węźle:
kubectl exec deploy/sleep -- ls -l /node-root/volumes/pv01/pg_wal

# Sprawdź, czy dodany element jest nadal na liście zadań
```

Na rysunku 5.18 widać, że moja aplikacja `to-do` pokazuje jakieś dane. Musisz uwierzyć mi na słowo, że te dane zostały dodane do pierwszej kapsuły bazy danych i ponownie załadowane z drugiej kapsuły bazy danych.

Mam teraz przyzwoicie podzieloną aplikację z kapsułą internetową, którą można aktualizować i skalować niezależnie od bazy danych, oraz z kapsułą bazy danych korzystającą z trwałego magazynu danych, który istnieje poza cyklem życia kapsuły. W tym ćwiczeniu jako bazowego magazynu dla trwałych danych użyłem woluminu lokalnego, ale w przypadku wdrożenia produkcyjnego wystarczyłoby zastąpić specyfikację woluminu w PV woluminem rozproszonym obsługiwanym przez dany klaster.

Kwestią uruchamiania w Kubernetesie relacyjnej bazy danych zajmę się pod koniec rozdziału, ale najpierw pokażę Ci prawdziwy rarytas związany z pamięcią masową: klaster dynamicznie alokujący woluminy na podstawie abstrakcyjnej klasy pamięci masowej.

Kup ksi k Pole ksi k

Rysunek 5.18. *Abstrakcje pamięci masowej oznaczają, że wystarczy zamontować PVC, by baza danych otrzymała trwały magazyn danych*

5.4. Dynamiczna alokacja woluminów i klasy pamięci masowej

Do tej pory korzystałem z przepływu pracy ze statycznym alokowaniem (ang. *static provisioning*). Tworzyłem PV bezpośrednio, a następnie tworzyłem PVC, które Kubernetes wiązał z PV. Sprawdza się to we wszystkich klastrach Kubernetesa i może być preferowanym przepływem pracy w organizacjach, w których dostęp do pamięci masowej jest ściśle kontrolowany, ale większość platform Kubernetesa obsługuje prostszą alternatywę z alokacją dynamiczną (ang. *dynamic provisioning*).

W przepływie pracy z dynamiczną alokacją wystarczy utworzyć żądanie woluminu trwałego, a obsługujący je PV jest tworzony na żądanie przez klaster. Klastry można konfigurować z wieloma klasami pamięci masowej, które odzwierciedlają różne oferowane możliwości woluminów, a także z domyślną klasą pamięci masowej. Żądania woluminów trwałych mogą określać wymaganą nazwę klasy pamięci masowej albo używać klasy domyślnej — w tym przypadku pomija się pole klasy pamięci masowej w specyfikacji żądania, jak pokazałem w listingu 5.8.

Listing 5.8. Plik postgres-persistentVolumeClaim-dynamic.yaml; dynamiczne PCV

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc-dynamic
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
      # Nie ma pola storageClassName, używana jest więc klasa domyślna
```

To PVC możesz wdrożyć w klastrze bez tworzenia woluminu trwałego, ale nie jestem w stanie powiedzieć Ci, co się stanie, ponieważ zależy to od konfiguracji Twojego klastra. Jeśli Twoja platforma Kubernetesa obsługuje alokację dynamiczną z domyślną klasą pamięci masowej, PV zostanie utworzony, powiązany z żądaniem i będzie używał takiego typu woluminu, jaki jest ustawiony w klastrze jako domyślny.

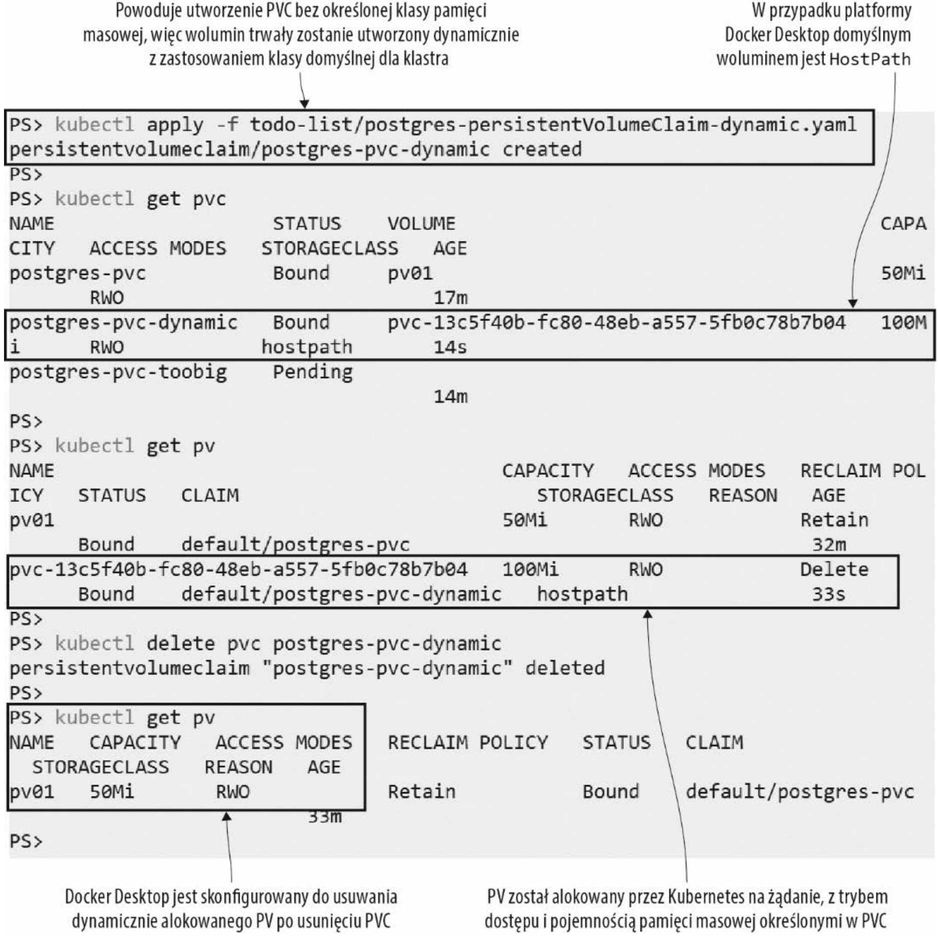
WYPRÓBUJ Wdróż żądanie woluminu trwałego i sprawdź, czy jest alokowane dynamicznie.

```
# Wdróż PVC z listingu 5.8:
kubectl apply -f todo-list/postgres-persistentVolumeClaim-dynamic.yaml

# Sprawdź żądania i woluminy:
kubectl get pvc
kubectl get pv

# Usuń żądanie:
kubectl delete pvc postgres-pvc-dynamic

# Ponownie sprawdź woluminy:
kubectl get pv
```

Co się dzieje, gdy wykonujesz ćwiczenie? Docker Desktop używa woluminu `HostPath` w domyślnej klasie pamięci masowej dla dynamicznie alokowanych woluminów trwałych. AKS używa Azure Files, a K3s `HostPath`, ale z inną konfiguracją niż Docker Desktop, co oznacza, że nie zobaczysz PV, ponieważ jest tworzony

Kup książkę Poleć książkę

tylko wtedy, gdy tworzona jest kapsuła wykorzystująca PVC. Na rysunku 5.19 przedstawiłem moje dane wyjściowe z platformy Docker Desktop. Wolumin trwały został utworzony i powiązany z PVC, a kiedy usunąłem PVC, usunięty został również wolumin trwały.



Rysunek 5.19. *Docker Desktop ma jeden zestaw zachowań dla domyślnej klasy pamięci; inne platformy różnią się w tym zakresie*

Klasy pamięci masowej zapewniają dużą elastyczność. Tworzy się je jako standardowe zasoby Kubernetesa, a w specyfikacji definiuje się dokładnie sposób działania klasy za pomocą następujących trzech pól:

- `provisioner` (alokator) — komponent, który tworzy woluminy trwałe na żądanie. Różne platformy mają różne alokatory, np. alokator w domyślnej klasie pamięci masowej AKS w celu tworzenia nowych udziałów plików integruje się z Azure Files.

Kup ksi k Pole ksi k

- `reclaimPolicy` (reguła odzyskiwania) — określa, co zrobić z dynamicznie utworzonymi woluminami, gdy żądanie zostanie usunięte. Wolumin bazowy może zostać również usunięty albo zachowany.
- `volumeBindingMode` (tryb wiązania woluminu) — określa, czy PV jest tworzony zaraz po utworzeniu PVC, czy dopiero po utworzeniu kapsuły, która używa danego PVC.

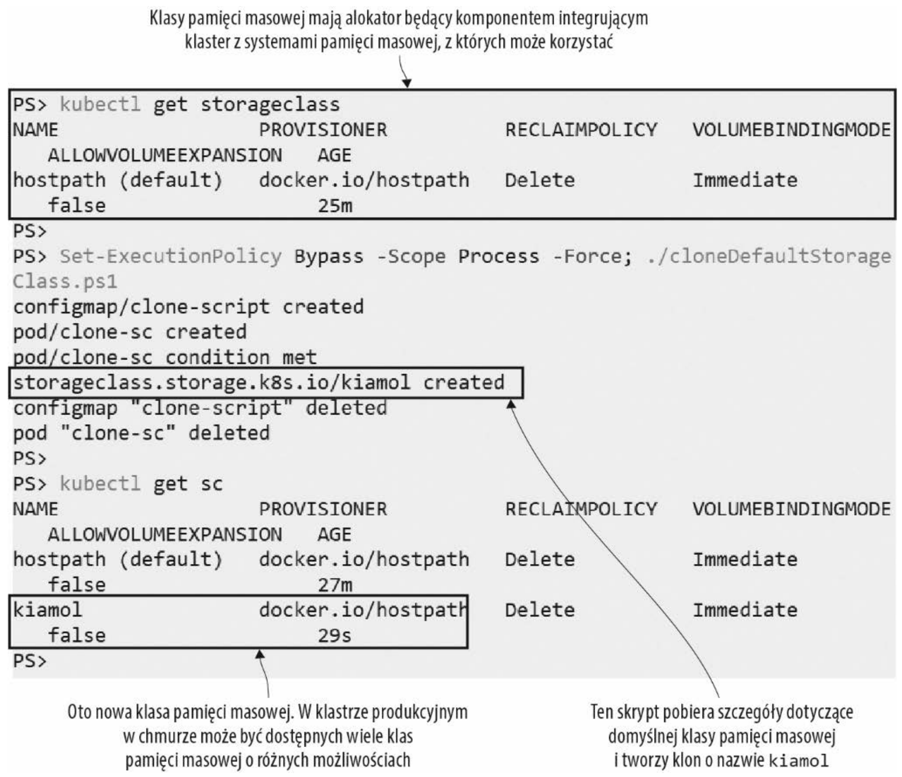
Kombinacja ustawień tych pól pozwala dokonać wyboru klasy pamięci masowej w klastrze, dzięki czemu aplikacje mogą żądać odpowiednich właściwości — od szybkiego magazynu lokalnego po pamięć klastrową o wysokiej dostępności — bez określania szczegółów dotyczących woluminu lub typu woluminu. Nie mogę dostarczyć Ci pliku YAML-a klasy pamięci masowej, który na pewno zadziała w Twoim klastrze, ponieważ nie wszystkie klastry mają te same alokatory. Zamiast tego utworzę nową klasę pamięci, klonując klasę domyślną.

WYPRÓBUJ Pobieranie domyślnej klasy pamięci masowej i klonowanie nie należy do najprzyjemniejszych zadań, więc zawarłem te kroki w skrypcie. Jeśli jesteś ciekaw, możesz sprawdzić zawartość skryptu, ale ostrzegam, że potem prawdopodobnie będziesz musiał się położyć i ochłonąć.

```
# Wyświetl listę klas pamięci masowej w klastrze:
kubectl get storageclass

# Sklonuj domyślną klasę w systemie Windows:
Set-ExecutionPolicy Bypass -Scope Process -Force; ./cloneDefaultStorageClass.ps1

# Sklonuj domyślną klasę w systemie Mac lub Linux:
chmod +x cloneDefaultStorageClass.sh && ./cloneDefaultStorageClass.sh

# Wyświetl listę klas pamięci masowej:
kubectl get sc
```

Dane wyjściowe z polecenia wyświetlenia listy klas pamięci masowej pokazują, co jest skonfigurowane w klastrze. Po uruchomieniu skryptu powinieneś mieć nową klasę o nazwie `kiamol`, która ma taką samą konfigurację jak klasa domyślna. Swoje dane wyjściowe z platformy Docker Desktop pokazałem na rysunku 5.20.

Mamy teraz niestandardową klasę pamięci masowej, której aplikacje mogą żądać w PVC. Jest to znacznie bardziej intuicyjny i elastyczny sposób zarządzania pamięcią masową, zwłaszcza na platformie chmurowej, gdzie alokacja dynamiczna jest prosta i szybka. W listingu 5.9 przedstawiłem specyfikację PVC żądającą nowej klasy pamięci.



Rysunek 5.20. *Klonowanie domyślnej klasy pamięci masowej w celu utworzenia klasy niestandardowej, której będzie można używać w specyfikacjach PVC*

Listing 5.9. Plik postgres-persistentVolumeClaim-storageClass.yaml

```
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: kiamol      # Klasa pamięci masowej jest abstrakcją
  resources:
    requests:
      storage: 100Mi
```

Klasy pamięci masowej w klastrze produkcyjnym będą miały bardziej wymowne nazwy, ale teraz mamy w naszych klastrach klasę pamięci o tej samej nazwie, dzięki czemu możemy zaktualizować bazę danych Postgres, aby używała tej bezpośrednio zdefiniowanej klasy.

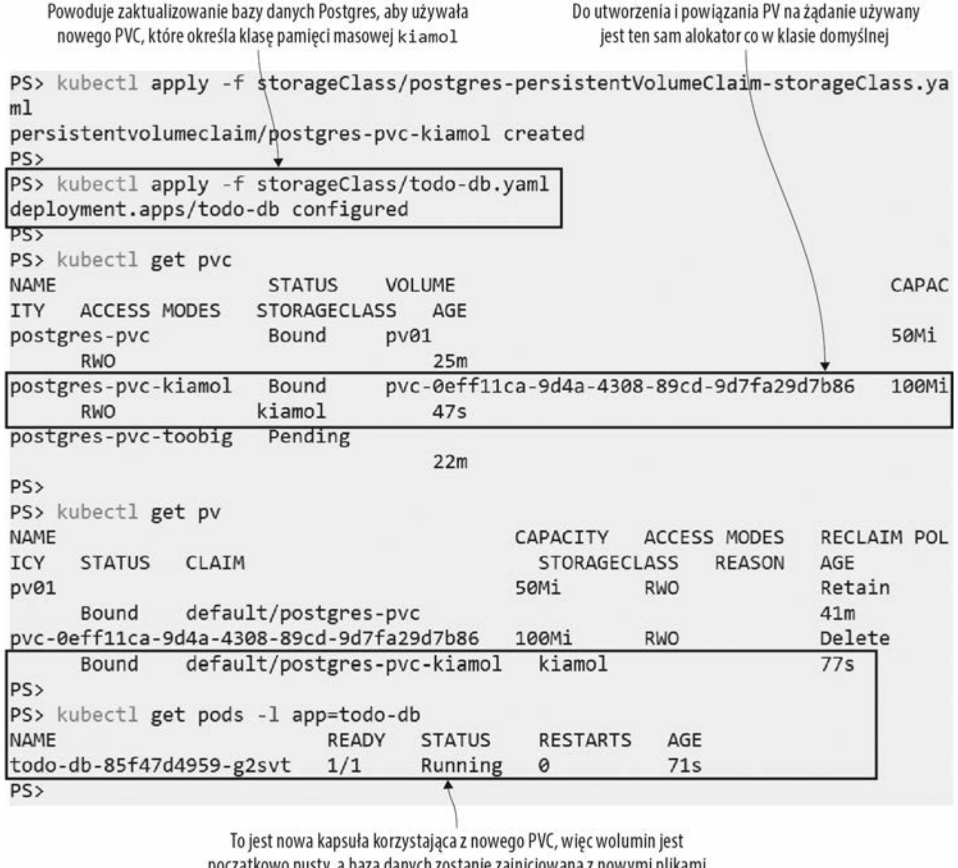
WYPRÓBUJ Utwórz nowe żądanie woluminu trwałego i zaktualizuj specyfikację kapsuły bazy danych, aby go używała.

```
# Utwórz nowe PVC, używając niestandardowej klasy pamięci masowej:
kubectl apply -f storageClass/postgres-persistentVolumeClaim-storageClass.yaml
```

Kup książkę Poleć książkę

```
# Zaktualizuj bazę danych, aby używała nowego PVC:
kubectl apply -f storageClass/todo-db.yaml

# Sprawdź pamięć masową:
kubectl get pvc
kubectl get pv

# Sprawdź kapsuły:
kubectl get pods -l app=todo-db

# Odśwież listę w aplikacji to-do
```

W tym ćwiczeniu przełączam kapsułę bazy danych na używanie nowego alokowanego dynamicznie PVC, co pokazałem w swoich danych wyjściowych na rysunku 5.21. Nowe PVC jest obsługiwane przez nowy wolumin, zostanie więc uruchomione z pustym woluminem i poprzednie dane zostaną utracone. Poprzedni wolumin nadal istnieje, mógłbym więc wdrożyć kolejną aktualizację kapsuły bazy danych, przywrócić stare PVC i ponownie zobaczyć swoje elementy.



Rysunek 5.21. *Korzystanie z klas pamięci masowej znacznie upraszcza specyfikację aplikacji; wystarczy nazwać klasę w PVC*

Kup ksi k Pole ksi k

5.5. Opcje wyboru pamięci masowej w Kubernetesie

Tak właśnie wygląda pamięć masowa w Kubernetesie. W swojej codziennej pracy będziesz definiować żądania woluminów trwałych dla kapsuł oraz określać wymagane rozmiary i klasy pamięci masowej, które mogą mieć wartości niestandardowe, takie jak `FastLocal` lub `Replicated`. W tym rozdziale zabrałem Cię w długą podróż, by pokazać Ci, co tak naprawdę się dzieje, gdy tworzysz żądania pamięci masowej, jakie inne zasoby są w to zaangażowane i jak je konfigurować.

Omówiłem także typy woluminów, ale ten temat będziesz musiał zgłębić samodzielnie, żeby poznać opcje dostępne na Twojej platformie Kubernetesa i oferowane przez nie funkcjonalności. Jeśli pracujesz w środowisku chmurowym, powinieneś mieć luksus wyboru wielu wariantów pamięci masowej obejmujących cały klaster, ale pamiętaj, że pamięć masowa kosztuje, a najdroższa jest szybka pamięć masowa. Musisz zrozumieć, że jeśli utworzysz PVC przy użyciu klasy szybkiej pamięci masowej skonfigurowanej tak, aby zachowywała bazowy wolumin, to po usunięciu wdrożenia nadal będziesz płacić za pamięć masową.

Nasuwa się więc ważne pytanie: „Czy w ogóle należy używać Kubernetesa do uruchamiania aplikacji stanowych, takich jak bazy danych?". Ta funkcjonalność ma zapewniać Ci wysoce dostępną, replikowaną pamięć masową (jeśli Twoja platforma to obsługuje), ale nie znaczy to, że powinieneś spieszyć się z likwidacją swoich zasobów Oracle i zastępować je bazą danych MySQL działającą w Kubernetesie. Zarządzanie danymi znacznie zwiększa złożoność aplikacji Kubernetesa, a uruchamianie aplikacji stanowych to tylko część problemu. Warto pomyśleć o kopiach zapasowych, migawkach i wycofywaniu zmian, a jeśli pracujesz w chmurze, zarządzana usługa bazy danych prawdopodobnie zapewni Ci te funkcjonalności od ręki. Dość kuszące jest jednak zdefiniowanie całego stosu w manifestach Kubernetesa, a niektóre nowoczesne serwery baz danych są zaprojektowane do działania na platformie kontenerowej. Możesz przyjrzeć się np. takim opcjom jak TiDB i CockroachDB.

Teraz pozostaje tylko posprzątać klaster, zanim przejdziemy do laboratorium.

WYPRÓBUJ Usuń wszystkie obiekty wdrożone za pomocą manifestów używanych w tym rozdziale. Możesz zignorować wszelkie otrzymane błędy, bo gdy będziesz uruchamiać poniższe polecenia, nie wszystkie obiekty muszą istnieć.

```
# Usuń wdrożenia, żądania woluminów trwałych, woluminy trwałe i usługi:
kubectl delete -f pi/v1 -f sleep/ -f storageClass/ -f todo-list/web -f
  todo-list/postgres -f todo-list/

# Usuń niestandardową klasę pamięci masowej:
kubectl delete sc kiamol
```

5.6. Laboratorium

Te laboratoria mają zapewnić Ci trochę doświadczenia w rzeczywistych problemach związanych z Kubernetesem, dlatego nie zaplanowałem powtarzania ćwiczenia w klonowaniu domyślnej klasy pamięci masowej. Zamiast tego masz nowe wdrożenie aplikacji `to-do`, które zawiera kilka problemów. Przed kapsułą internetową umieszczony jest serwer proxy, aby poprawić wydajność, a w kapsule używany jest lokalny plik bazy danych, ponieważ jest to tylko wdrożenie programistyczne. Potrzebujesz trwałego magazynu danych skonfigurowanego w warstwie proxy i warstwie internetowej, abyś mógł usuwać kapsuły i wdrożenia bez jednoczesnej utraty danych. Oto wskazówki:

- Zacznij od wdrożenia manifestów aplikacji z folderu *ch05/lab/todo-list*. Za ich pomocą utworzysz usługi i wdrożenia dla serwera proxy i komponentów internetowych.
- Znajdź adres URL usługi `LoadBalancer` i wypróbuj działanie aplikacji. Przekonasz się, że aplikacja nie reaguje, i będziesz musiał zbadać dzienniki, aby się dowiedzieć, co jest nie tak.
- Twoim zadaniem jest skonfigurowanie w kapsule internetowej trwałego magazynu danych dla plików pamięci podręcznej serwera proxy i pliku bazy danych. Na podstawie wpisów dziennika i specyfikacji kapsuły powinieneś być w stanie znaleźć miejsca docelowe punktów montowania.
- Gdy będziesz mieć już działającą aplikację, wprowadź jakieś dane, a potem usuń wszystkie kapsuły, odśwież przeglądarkę i sprawdź, czy Twoje dane nie zostały usunięte.
- Możesz użyć dowolnego typu woluminu lub klasy pamięci masowej. Jest to dobra okazja do zbadania możliwości Twojej platformy.

Swoje podejście możesz jak zwykle porównać z moim rozwiązaniem, które znajdziesz w repozytorium kodu źródłowego dołączonego do książki: *kiamol/ch05/lab/README.md*.

Kup ksi k Pole ksi k

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

KUBERNETES:
WYSTARCZĄ CI 22 GODZINY!

Kubernetes to system, który uruchamia aplikacje w kontenerach i nimi zarządza. Jest obsługiwany przez wszystkie ważne platformy chmurowe i znakomicie się sprawdza jako centrum danych. Został zbudowany w 2014 roku przez Google i do teraz prężnie się rozwija. Słynie ze skalowalności, elastyczności, wszechstronności i potężnego zestawu funkcjonalności. Biegłość w posługiwaniu się Kubernetesem jest dziś receptą na sukces. Aby ją zdobyć, musisz zainwestować trochę zaangażowania i nieco ponad 20 godzin. Resztę znajdziesz w tej książce.

Oto znakomity przewodnik po Kubernetesie. Dzięki niemu w ciągu 22 godzinnych lekcji poznasz najważniejsze możliwości Kubernetesa. Od początku będziesz się koncentrować na praktyce: dzięki codziennym ćwiczeniom, przykładom i laboratoriom zdobędziesz biegłość w używaniu najlepszych narzędzi Kubernetesa zgodnie ze sprawdzonymi praktykami. Dowiesz się, jak definiować aplikacje w manifestach YAML, nauczysz się konfigurować ruch sieciowy i uruchamiać zadania wsadowe. Płynnie przejdziesz do pracy w środowisku produkcyjnym i zapewnisz aplikacji wysoki poziom bezpieczeństwa. Zapoznasz się też z zagadnieniami zaawansowanymi, takimi jak skalowanie aplikacji w górę i w dół, kontrola dostępu oparta na rolach, a także używanie Kubernetesa jako platformy dla funkcji bezserwerowych i jako klastra wieloarchitekturowego.

Najciekawsze zagadnienia ujęte w książce:

- cykl życia aplikacji Kubernetesa
- bezpieczeństwo w Kubernetesie
- wdrażanie aplikacji w klastrach Kubernetes
- tworzenie aplikacji skalowalnych i odpornych na błędy
- Kubernetes jako platforma dla nowych technologii

Elton Stoneman jest niezależnym konsultantem, szkoleniowcem i wielokrotnym zdobywcą tytułu MVP przyznawanego przez Microsoft. Przez trzy ogromnie pracowite i niezwykle zabawne lata rozwijał Dockera. Obecnie pomaga organizacjom na wszystkich etapach ich podróży po technologii kontenerowej.

MANNING