O'REILLY®

Kubernetes w środowisku produkcyjnym

Jak budować efektywne platformy aplikacji

Josh Rosso, Rich Lander,
Alexander Brand, John Harris

Helion

Tytuł oryginału: Production Kubernetes: Building Successful Application Platforms

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-8549-8

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/kubwsr.zip

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

Printed in Poland.

Spis treści

ROZDZIAŁ 4.

Pamięć masowa kontenerów

Chociaż twórcy Kubernetesa mają do zaoferowania duże doświadczenie w świecie bezstanowych obciążeń roboczych, coraz bardziej powszechne staje się uruchamianie usług stanowych. Do klastrów Kubernetesa trafiają nawet złożone stanowe obciążenia robocze, takie jak bazy danych i kolejki komunikatów. Aby obsługiwać te obciążenia robocze, Kubernetes musi zapewnić możliwości przechowywania danych wykraczające poza opcje efemeryczne. Chodzi mianowicie o systemy, które będą miały zwiększoną odporność i dostępność w obliczu różnych zdarzeń, takich jak awaria aplikacji lub przeniesienie obciążenia roboczego na inny host.

W tym rozdziale zbadamy, w jaki sposób nasza platforma może oferować aplikacjom usługi pamięci masowej. Zanim przejdziemy do omówienia podstawowych funkcjonalności pamięci masowej dostępnych w Kubernetesie, zajmiemy się kluczowymi kwestiami dotyczącymi oczekiwań aplikacji w zakresie utrwalania danych i systemu pamięci masowej. Gdy będziemy zagłębiać się w bardziej zaawansowane wymagania związane z przechowywaniem danych, przyjrzymy się interfejsowi pamięci masowej kontenerów (CSI; *https://kubernetes-csi.github.io/docs*), który umożliwia integrację z różnymi dostawcami pamięci masowej. Na koniec omówimy użycie wtyczki CSI do zapewnienia aplikacjom samoobsługowej pamięci masowej.

Pamięć masowa jest obszernym tematem. Naszym zamiarem jest dostarczenie Ci wystarczającej ilości informacji do podejmowania świadomych decyzji dotyczących pamięci masowej, którą będziesz mógł zaoferować obciążeniom roboczym. Jeśli nie masz doświadczenia w tej dziedzinie, zdecydowanie zalecamy omówienie tych koncepcji z zespołem ds. infrastruktury (pamięci masowej). Kubernetes nie zwalnia Twojej organizacji z konieczności dysponowania specjalistyczną wiedzą na temat pamięci masowej!

Kwestie związane z pamięcią masową

Przed przejściem do wzorców i opcji pamięci masowej Kubernetesa powinniśmy zrobić krok w tył i przeanalizować kilka kluczowych kwestii związanych z potencjalnymi potrzebami w zakresie trwałego przechowywania danych. Na poziomie infrastruktury i aplikacji należy przemyśleć następujące wymagania:

- tryby dostępu,
- zwiększanie pojemności woluminu,
- alokowanie dynamiczne,

- kopia zapasowa i odzyskiwanie sprawności po awarii,
- urządzenia blokowe oraz plikowa i obiektowa pamięć masowa,
- dane efemeryczne,
- wybór dostawcy.

Tryby dostępu

Aplikacjom można zaoferować trzy tryby dostępu:

ReadWriteOnce (RWO)

Na danym woluminie operacje odczytu i zapisu może przeprowadzać pojedyncza kapsuła.

ReadOnlyMany (ROX)

Na danym woluminie operacje odczytu może przeprowadzać wiele kapsuł.

ReadWriteMany (RWX)

Na danym woluminie operacje odczytu i zapisu może przeprowadzać wiele kapsuł.

W przypadku aplikacji natywnych dla chmury najpowszechniejszym wzorcem jest zdecydowanie RWO. Kiedy korzysta się z popularnych dostawców usług chmurowych, takich jak Amazon Elastic Block Storage (EBS; *https://aws.amazon.com/ebs*) lub Azure Disk Storage (*https://azure.microsoft.com/en-us/services/storage/disks/*), jest się ograniczonym do RWO, ponieważ dysk może być podłączony tylko do jednego węzła. Chociaż to ograniczenie może się wydawać problematyczne, większość aplikacji natywnych dla chmury działa najlepiej z tego rodzaju pamięcią masową, w której wolumin należy wyłącznie do nich i oferuje wysoką wydajność operacji odczytu i zapisu.

Często napotykamy starsze aplikacje, które wymagają trybu dostępu RWX. Zazwyczaj są one zbudowane z założeniem dostępu do sieciowego systemu plików (ang. *Network File System* — NFS; *https://en.wikipedia.org/wiki/Network_File_System*). Gdy usługi muszą współdzielić stan, z reguły istnieją bardziej eleganckie rozwiązania niż udostępnianie danych za pośrednictwem NFS, np. użycie kolejek komunikatów lub baz danych. Ponadto jeśli aplikacja powinna udostępniać dane, zwykle najlepiej robić to za pośrednictwem interfejsu API, zamiast udzielać dostępu do jej systemu plików. To wszystko sprawia, że wiele przypadków użycia dla RWX bywa wątpliwych. Jeżeli NFS nie jest właściwym wyborem projektowym, zespoły zajmujące się platformami mogą stanąć przed trudnym wyborem: czy zaoferować pamięć masową zgodną z RWX, czy może poprosić programistów o przeprojektowanie architektury aplikacji. W przypadku podjęcia decyzji, że wymagana jest obsługa trybu dostępu ROX lub RWX, można przeprowadzić integrację z kilkoma dostępnymi dostawcami, takimi jak Amazon Elastic File System (EFS; *https://aws.amazon.com/efs*) i Azure File Share (*https://docs.microsoft.com/en-us/azure/storage/files/storage-files-introduction*).

Zwiększanie pojemności woluminu

Z czasem aplikacja może zapełnić swój wolumin. Może to stanowić wyzwanie, gdyż zastąpienie tego woluminu większym wymagałoby migracji danych. Jednym z rozwiązań problemu jest obsługa zwiększania pojemności woluminu. W przypadku orkiestratora kontenerów, takiego jak Kubernetes, wymaga to wykonania kilku czynności, do których należą:

1. Zażądanie od orkiestratora dodatkowej pamięci masowej, np. za pośrednictwem żądania woluminu trwałego (`PersistentVolumeClaim`).
2. Zwiększenie rozmiaru woluminu za pośrednictwem dostawcy pamięci masowej.
3. Rozszerzenie systemu plików, aby korzystał z większego woluminu.

Po zakończeniu tych czynności kapsuła będzie miała dostęp do dodatkowej przestrzeni dyskowej. Dostępność tej funkcjonalności zależy od wyboru back-endu pamięci masowej oraz tego, czy integracja w Kubernetesie umożliwia wykonanie powyższych kroków. Przykład zwiększania pojemności woluminu omówimy dalej w tym rozdziale.

Alokowanie woluminów

Dostępne są dwa modele alokowania: dynamiczny i statyczny. W przypadku alokowania statycznego (ang. *static provisioning*) na węzłach tworzone są woluminy, z których może korzystać Kubernetes. Alokowanie dynamiczne (ang. *dynamic provisioning*) zachodzi wtedy, gdy w klastrze uruchomiony jest sterownik, który komunikuje się z dostawcą pamięci masowej i może zaspokajać żądania alokowania pamięci wysyłane przez obciążenia robocze. Wszędzie tam, gdzie to możliwe, preferowane jest alokowanie dynamiczne. Często wybór między tymi modelami zależy od tego, czy bazowy system pamięci masowej ma kompatybilny sterownik dla Kubernetesa. Omówimy te sterowniki dalej w tym rozdziale.

Kopia zapasowa i odzyskiwanie sprawności po awarii

Tworzenie kopii zapasowej to jeden z najbardziej złożonych aspektów obsługi pamięci masowej, zwłaszcza gdy wymagane jest automatyczne przywracanie. Ogólnie rzecz biorąc, kopia zapasowa to kopia danych, która jest przechowywana na wypadek ich utraty. Strategie tworzenia kopii zapasowych równoważy się zwykle za pomocą gwarancji dostępności systemów pamięci masowej. Chociaż kopie zapasowe są zawsze ważne, bywają mniej kluczowe, gdy system pamięci masowej ma np. gwarancję replikacji, gdzie utrata sprzętu *nie spowoduje* utraty danych. Należy też wziąć pod uwagę, że aplikacje mogą wymagać różnych procedur w celu ułatwienia tworzenia kopii zapasowych oraz przywracania danych. Pomysł, że można wykonywać kopię zapasową całego klastra i przywracać go w dowolnym momencie, jest często podejściem naiwnym lub przynajmniej takim, które wymaga ogromnego wysiłku inżynieryjnego.

Wybór osoby odpowiedzialnej za tworzenie kopii zapasowych aplikacji i odzyskiwanie sprawności po awarii może być jedną z najtrudniejszych debat w organizacji. Zapewne przyjemnie byłoby oferować funkcjonalności przywracania jako usługi platformy. Taka koncepcja może się jednak nie utrzymać, gdy uwzględnimy złożoność charakterystyczną dla poszczególnych aplikacji — aplikacja może mieć np. problem z ponownym uruchomieniem i wymagać podjęcia działań, które są znane tylko programistom.

Jednym z najpopularniejszych rozwiązań do tworzenia kopii zapasowych zarówno dla stanu Kubernetesa, jak i stanu aplikacji jest Project Velero (*https://velero.io*). Velero może tworzyć kopie zapasowe tych obiektów Kubernetesa, które chcemy przenieść w inną lokalizację lub przywrócić w różnych klastrach. Dodatkowo obsługuje planowanie wykonywania migawek woluminów. Gdy dalej w rozdziale przejdziemy do omawiania migawek woluminów, dowiesz się, że rozplanowywaniem tworzenia migawek i zarządzaniem nimi trzeba zająć się *samemu*. Co więcej, często otrzymujemy podstawowe

funkcjonalności migawek, ale musimy zdefiniować dla nich przepływ orkiestracji. Velero obsługuje ponadto wykonywanie kopii zapasowych i przywracanie zaczepów. Dzięki temu przed utworzeniem kopii zapasowej lub przeprowadzeniem odzyskiwania można uruchamiać w kontenerze polecenia. Niektóre aplikacje przed wykonaniem kopii zapasowej mogą wymagać np. zatrzymania ruchu lub uruchomienia czyszczenia pamięci podręcznej. Umożliwiają to właśnie zaczepy z Velero.

Urządzenia blokowe oraz plikowa i obiektowa pamięć masowa

Oczekiwane przez aplikacje typy pamięci masowej są kluczem do wyboru odpowiedniego podłoża pamięci masowej oraz integracji z Kubernetesem. Najpopularniejszym typem pamięci masowej używanym przez aplikacje jest plikowa pamięć masowa. Jest to urządzenie blokowe z nałożoną warstwą systemu plików. Dzięki temu aplikacje mogą zapisywać dane w plikach w sposób znany z dowolnego systemu operacyjnego.

Podłożem systemu plików jest urządzenie blokowe. Zamiast więc ustanawiać nad nim warstwę systemu plików, moglibyśmy zaoferować to urządzenie, by aplikacje mogły komunikować się bezpośrednio z surowym blokiem. Systemy plików z natury zwiększają obciążenie przy zapisywaniu danych, ale w nowoczesnym tworzeniu oprogramowania dość rzadko trzeba się przejmować obciążeniem generowanym przez te systemy. Jeśli jednak Twój przypadek użycia gwarantuje bezpośrednią interakcję z surowymi urządzeniami blokowymi, niektóre systemy pamięci masowej mogą to obsługiwać.

Ostatnim typem pamięci masowej jest obiektowa pamięć masowa. Różni się ona od pamięci plikowej pod tym względem, że nie ma konwencjonalnej hierarchii. Obiektowa pamięć masowa umożliwia programistom korzystanie z nieustrukturyzowanych danych przez nadawanie im unikatowego identyfikatora, dodawanie do nich metadanych i zapisywanie. Obiektowe magazyny danych dostawców chmury takich jak Amazon S3 (*https://aws.amazon.com/s3*) stały się dla organizacji popularnymi lokalizacjami do przechowywania obrazów, plików binarnych itd. Ta popularyzacja została przyspieszona dzięki w pełni funkcjonalnemu internetowemu API i kontroli dostępu. W interakcje z obiektowymi magazynami danych *najczęściej* wchodzą same aplikacje, używając do uwierzytelniania i komunikacji z dostawcą odpowiedniej biblioteki. Ponieważ interfejsy służące do interakcji z obiektowymi magazynami danych są mniej standaryzowane, rzadziej będą integrowane jako usługi platformy, z którymi aplikacje mogą komunikować się w transparentny sposób.

Dane efemeryczne

O ile zastosowanie pamięci masowej może sugerować poziom utrwalania danych wykraczający poza cykl życia kapsuły, o tyle istnieją uzasadnione przypadki użycia dla danych efemerycznych. Kontenery, które zapisują we własnym systemie plików, będą domyślnie korzystać z pamięci efemerycznej. Jeśli kontener zostanie zrestartowany, ta pamięć będzie utracona. Dla efemerycznej pamięci masowej dostępny jest jednak typ woluminu `emptyDir` (*https://kubernetes.io/docs/concepts/storage/volumes/#emptydir*), odporny na ponowne uruchamianie. Jest on nie tylko odporny na restarty kontenera, ale może być także używany do współdzielenia plików przez kontenery w tej samej kapsule.

Największym ryzykiem związanym z danymi efemerycznymi jest możliwość, że kapsuły będą wykorzystywać zbyt dużą ilość pamięci masowej hosta. Chociaż mogłoby się wydawać, że 4 GB na kapsułę to niedużo, należy wziąć pod uwagę, że węzeł może obsługiwać setki, a w niektórych przypadkach

nawet tysiące kapsuł. Kubernetes obsługuje możliwość ograniczania łącznej ilości pamięci efemerycznej dostępnej dla kapsuł w określonej przestrzeni nazw. Konfigurację tych elementów omówiliśmy w rozdziale 12.

Wybór dostawcy pamięci masowej

Dostawców pamięci masowej nie brakuje. Opcje rozciągają się od rozwiązań pamięci masowej, którymi można zarządzać samodzielnie, takich jak Ceph, po w pełni zarządzane systemy, takie jak Google Persistent Disk lub Amazon Elastic Block Store. Omówienie różnic w dostępnych opcjach wykracza daleko poza zakres książki. Zalecamy jednak zapoznanie się z funkcjonalnościami systemów pamięci masowej oraz sprawdzenie, które z tych funkcjonalności można łatwo zintegrować z Kubernetesem. Da Ci to pewne wyobrażenie o tym, jak poszczególne rozwiązania są dostosowane do wymagań Twojej aplikacji. Gdybyś jednak musiał zarządzać własnym systemem pamięci masowej, w miarę możliwości rozważ użycie czegoś, z czym masz doświadczenie operacyjne. Wprowadzenie Kubernetesa wraz z nowym systemem pamięci masowej znacznie zwiększy złożoność operacyjną Twojej organizacji.

Podstawowe funkcjonalności pamięci masowej Kubernetesa

Kubernetes zapewnia wiele podstawowych funkcjonalności do obsługi pamięci masowej obciążeń roboczych. Stanowią one elementy konstrukcyjne wykorzystywane do oferowania wyrafinowanych rozwiązań w zakresie pamięci masowej. W tym podrozdziale omówimy woluminy trwałe (ang. *Persistent Volume* — PV), żądania woluminów trwałych (ang. *Persistent Volume Claim* — PVC) i klasy pamięci masowej (ang. *Storage Class*), posługując się przykładem przydzielania kontenerom szybkiej, wstępnie alokowanej pamięci masowej.

Woluminy trwałe i żądania woluminów trwałych

Woluminy i żądania woluminów stanowią w Kubernetesie podstawy pamięci masowej. Są one udostępniane za pomocą interfejsów API Persistent Volume (*https://kubernetes.io/docs/concepts/storage/persistent-volumes/*) i Persistent Volume Claim (*https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims*). Zasób `PersistentVolume` reprezentuje wolumin pamięci masowej znany Kubernetesowi. Załóżmy, że administrator przygotował węzeł, który oferuje 30 GB szybkiej pamięci masowej na hoście. Przyjmijmy również, że udostępnił ten magazyn danych w lokalizacji */mnt/fast-disk/pod-0*. Do reprezentowania takiego woluminu w Kubernetesie administrator może wtedy utworzyć obiekt `PersistentVolume`:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0
spec:
  capacity:
    storage: 30Gi (1)
  volumeMode: Filesystem (2)
  accessModes:
```

```
  - ReadWriteOnce ❸
  storageClassName: local-storage ❹
  local:
    path: /mnt/fast-disk/pod-0
  nodeAffinity: ❺
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - test-w
```

❶ Ilość pamięci masowej dostępnej w tym woluminie. Służy do określania, czy z danym woluminem może zostać powiązane żądanie.

❷ Określa, czy wolumin jest urządzeniem blokowym (*https://en.wikipedia.org/wiki/Device_file#Block_devices*), czy systemem plików.

❸ Określa tryb dostępu woluminu. Możliwe wartości to `ReadWriteOnce`, `ReadMany` i `ReadWriteMany`.

❹ Kojarzy ten wolumin z klasą pamięci. Służy do łączenia z tym woluminem ewentualnego żądania.

❺ Identyfikuje węzeł, z którym ten wolumin powinien być skojarzony.

Jak widać, obiekt `PersistentVolume` zawiera szczegóły dotyczące implementacji woluminu. Aby zapewnić jeszcze jedną warstwę abstrakcji, wprowadzono obiekt `PersistentVolumeClaim`, który wiąże odpowiedni wolumin na podstawie jego żądania. Najczęściej żądanie woluminu trwałego jest definiowane przez zespół aplikacji, który dodaje je do swojej przestrzeni nazw i odwołuje się do niego ze swoich kapsuł:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc0
spec:
  storageClassName: local-storage ❶
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 30Gi ❷
---
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: fast-disk
      persistentVolumeClaim:
        claimName: pvc0 ❸
  containers:
    - name: ml-processer
      image: ml-processer-image
      volumeMounts:
        - mountPath: "/var/lib/db"
          name: fast-disk
```

❶ Szuka woluminu klasy `local-storage` z trybem dostępu `ReadWriteOnce`.

❷ Tworzy wiązanie z woluminem o pojemności co najmniej 30 GB.

❸ Deklaruje, że ta kapsuła jest konsumentem żądania woluminu trwałego.

Na podstawie ustawienia `nodeAffinity` (powinowactwa węzłów) obiektu `PersistentVolume` ta kapsuła zostanie automatycznie rozdysponowana na host, na którym dostępny jest ten wolumin. Od programisty nie wymaga się żadnej dodatkowej konfiguracji powinowactwa.

Ten proces ilustruje ręczny sposób, w jaki administratorzy mogą udostępniać pamięć masową programistom. Nazywamy to alokacją statyczną. Przy odpowiedniej automatyzacji może to być realny sposób na udostępnienie kapsułom szybkich dysków na hostach. W klastrze można np. wdrożyć alokator woluminów Local Persistence Volume Static Provisioner (*https://github.com/kubernetes-sigs/sig-storage-local-static-provisioner*), który będzie wykrywał wstępnie alokowaną pamięć masową i automatycznie udostępniał ją jako woluminy trwałe (`PersistentVolume`). Zapewnia on również pewne funkcjonalności zarządzania cyklem życia, np. usuwanie danych po zniszczeniu obiektu żądania woluminu trwałego (`PersistentVolumeClaim`).

Istnieje wiele sposobów na zapewnienie lokalnej pamięci masowej, które mogą prowadzić do złych praktyk. Atrakcyjne może się wydawać np. umożliwienie programistom korzystania z `hostPath` (*https://kubernetes.io/docs/concepts/storage/volumes/#hostpath*) w miejsce wstępnego alokowania lokalnej pamięci masowej. Wolumin `hostPath` pozwala określić na hoście ścieżkę, która może być użyta do wiązania bez konieczności tworzenia zasobów `PersistentVolume` i `PersistentVolumeClaim`. Stanowi to potencjalnie ogromne zagrożenie bezpieczeństwa, gdyż dopuszcza stosowanie przez programistów wiązania kapsuły z katalogami na hoście, co może mieć negatywny wpływ na dany host i pozostałe kapsuły. Jeśli chcesz zapewnić programistom efemeryczny magazyn danych, który przetrwa restart kapsuły, ale jej usunięcia lub przeniesienia na inny węzeł już nie, możesz użyć woluminu `emptyDir` (*https://kubernetes.io/docs/concepts/storage/volumes/#emptydir*). Spowoduje to alokowanie pamięci masowej w systemie plików zarządzanym przez Kube i będzie transparentne dla kapsuły.

Klasy pamięci

W wielu środowiskach nierealistyczne jest oczekiwanie na wcześniejsze przygotowywanie węzłów z dyskami i woluminami. Przypadki te często wymagają dynamicznej alokacji, w której woluminy mogą być udostępniane na podstawie potrzeb wyrażanych przez żądania. Aby umożliwić korzystanie z tego modelu, można udostępnić programistom klasy pamięci masowej. Są one definiowane za pomocą interfejsu API StorageClass (*https://kubernetes.io/docs/concepts/storage/storage-classes/*). Jeżeli założymy, że Twój klaster działa w usłudze AWS i chcesz dynamicznie oferować kapsułom woluminy EBS, możesz dodać następującą klasę `StorageClass`:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-standard ❶
  annotations:
    storageclass.kubernetes.io/is-default-class: true ❷
```

```
provisioner: kubernetes.io/aws-ebs ❸
parameters: ❹
  type: io2
  iopsPerGB: "17"
  fsType: ext4
```

❶ Nazwa `StorageClass`, do której można się odwoływać z poziomu żądań.

❷ Ustawia tę `StorageClass` jako domyślną. Jeśli żądanie nie określa klasy, stosowana jest klasa domyślna.

❸ Do tworzenia woluminów na podstawie żądań używa alokatora `aws-ebs`.

❹ Konfiguracja sposobu alokacji woluminów charakterystyczna dla danego dostawcy.

Udostępniając wiele klas `StorageClass`, można zaoferować programistom różne opcje pamięci masowej. Obejmuje to obsługę więcej niż jednego dostawcy w jednym klastrze, np. uruchamianie Ceph wraz z VMware vSAN. Ewentualnie można oferować różne warstwy pamięci masowej za pośrednictwem tego samego dostawcy. Przykładem może być proponowanie tańszych i droższych opcji pamięci masowej. Niestety Kubernetes nie ma szczegółowych ustawień do ograniczania klas, których mógłby zażądać programista. Tego typu kontrolę można zaimplementować za pomocą walidacyjnego sterowania dostępem (ang. *validating admission control*), co omówimy w rozdziale 8.

Kubernetes zapewnia licznych dostawców, takich jak AWS EBS, Glusterfs, GCE PD, Ceph RBD itd. Dostawcy ci byli wcześniej implementowani w drzewie, co oznaczało, że musieli zaimplementować swoją logikę w głównym projekcie Kubernetesa. Kod implementacji był następnie dostarczany w odpowiednich komponentach płaszczyzny sterowania Kubernetesa.

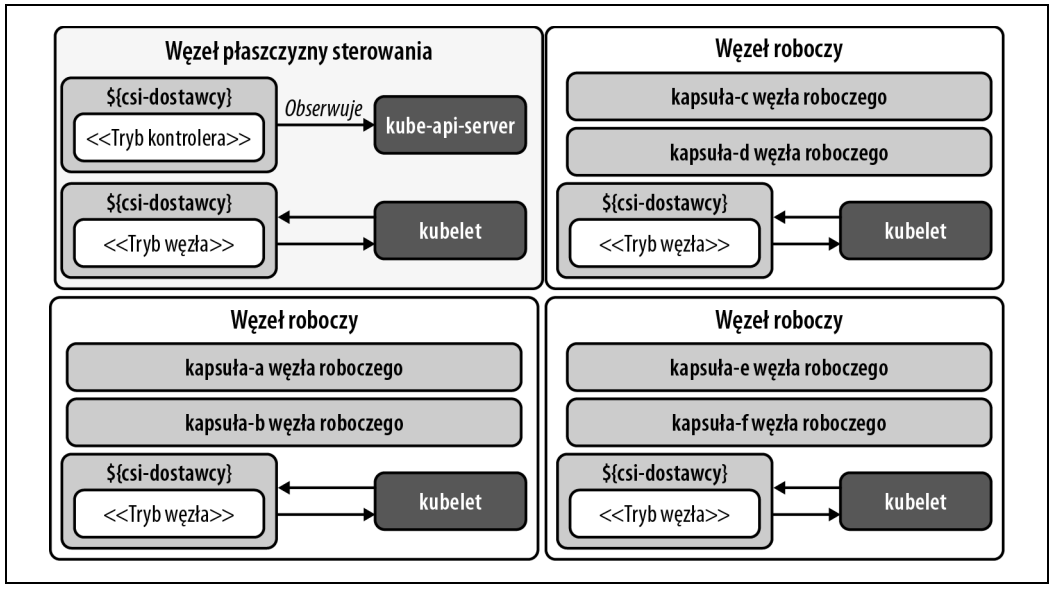
Ten model miał jednak kilka wad. Przede wszystkim dostawcą pamięci masowej nie można było zarządzać spoza projektu. Wszystkie zmiany w dostawcy musiały być powiązane z wydaniem Kubernetesa. Ponadto każde wdrożenie Kubernetesa było dostarczane z niepotrzebnym kodem. Klastry z systemem AWS nadal miały np. kod dostawcy do interakcji z GCE PD. Szybko okazało się, że eksternalizacja integracji tych dostawców i wycofywanie funkcjonalności umieszczania ich w drzewie mają dużą wartość. Początkowo ten problem miały rozwiązać sterowniki FlexVolume (*https://kubernetes.io/docs/concepts/storage/volumes/#flexVolume*), które były specyfikacją implementacji spoza drzewa. Prace nad nimi zostały jednak przeniesione w tryb utrzymywania na rzecz naszego następnego tematu: interfejsu pamięci masowej kontenerów.

Interfejs pamięci masowej kontenerów

Interfejs pamięci masowej kontenerów (ang. *Container Storage Interface* — CSI) jest odpowiedzią na to, w jaki sposób zapewnia się obciążeniom roboczym blokową i plikową pamięć masową. Implementacje CSI są traktowane jak sterowniki, które mają informacje operacyjne niezbędne do komunikowania się z dostawcami pamięci masowej. Dostawcami mogą być zarówno systemy w chmurze, np. Google Persistent Disks (*https://cloud.google.com/persistent-disk*), jak i samodzielnie wdrażane i zarządzane systemy pamięci masowej, do których należy chociażby Ceph (*https://ceph.io*). Sterowniki są implementowane przez dostawców pamięci masowej w projektach utrzymywanych poza drzewem. Mogą być zarządzane całkowicie poza klastrem, w którym są wdrożone.

Ogólnie rzecz biorąc, implementacje CSI zawierają wtyczkę kontrolera i wtyczkę węzła. Twórcy sterowników CSI mają dużą elastyczność w sposobie implementacji tych komponentów. Zazwyczaj implementacje łączą wtyczki kontrolera i węzła w tym samym pliku binarnym i włączają poszczególne tryby za pomocą zmiennej środowiskowej, takiej jak `X_CSI_MODE`. Jedyne wymagania to zarejestrowanie sterownika w agencie `kubelet` oraz zaimplementowanie punktów końcowych ze specyfikacji CSI.

Usługa kontrolera jest odpowiedzialna za zarządzanie w dostawcy pamięci masowej tworzeniem i usuwaniem woluminów. Ta funkcjonalność może być rozszerzana o (opcjonalne) funkcje, takie jak wykonywanie migawek woluminów i zwiększanie pojemności woluminów. Usługa węzła odpowiada za przygotowanie woluminów do wykorzystania przez działające na węźle kapsuły. Często oznacza to konfigurowanie punktów montowania i raportowanie informacji o woluminach znajdujących się na węźle. Zarówno usługa węzła, jak i kontrolera implementują także usługi tożsamości, które dostarczają informacji o wtyczce, jej funkcjonalnościach i kondycji. Na rysunku 4.1 przedstawiliśmy architekturę klastra, który ma wdrożone te komponenty.



Rysunek 4.1. Klaster z uruchomioną wtyczką CSI. Sterownik działa w trybach węzła i kontrolera. Kontroler jest zwykle uruchamiany jako wdrożenie, a usługa węzła jest wdrażana jako zbiór demonów, który umieszcza na każdym hoście po jednej kapsule

Przyjrzyjmy się bliżej tym dwóm komponentom: kontrolerowi i węzłowi.

Kontroler CSI

Usługa kontrolera CSI zapewnia interfejsy API do zarządzania woluminami w systemie trwałej pamięci masowej. Płaszczyzna sterowania Kubernetesa *nie wchodzi* w interakcje bezpośrednio z usługą kontrolera CSI. Zamiast tego kontrolery utrzymywane przez społeczność pamięci masowej Kubernetesa reagują na zdarzenia tej platformy i tłumaczą je na instrukcje CSI, takie jak `CreateVolumeRequest` (utwórz żądanie woluminu) w przypadku zdarzenia utworzenia nowego obiektu `PersistentVolumeClaim`.

Ponieważ usługa kontrolera CSI udostępnia swoje interfejsy API za pośrednictwem gniazd uniksowych, kontrolery te są zwykle wdrażane jako tzw. przyczepy (ang. *sidecar*) równolegle z usługą kontrolera CSI. Istnieje wiele kontrolerów zewnętrznych, a każdy z nich zachowuje się inaczej:

external-provisioner

Utworzenie obiektu `PersistentVolumeClaim` powoduje wysłanie do sterownika CSI żądania utworzenia woluminu. Po utworzeniu woluminu u dostawcy pamięci masowej ten kontroler tworzy w Kubernetesie obiekt `PersistentVolume`.

external-attacher

Obserwuje obiekty `VolumeAttachment`, które deklarują, czy wolumin powinien zostać dołączony do węzła lub od niego odłączony. Wysyła do sterownika CSI żądanie dołączenia lub odłączenia.

external-resizer

Wykrywa zmiany rozmiaru magazynu danych wprowadzane w obiektach `PersistentVolumeClaim`. Wysyła do sterownika CSI żądania rozszerzenia woluminu.

external-snapshotter

Wysyła do sterownika żądania wykonania migawki, gdy tworzone są obiekty `VolumeSnapshotContent`.

Podczas implementowania wtyczek CSI programiści nie muszą używać wyżej wymienionych kontrolerów. Zaleca się jednak ich stosowanie, aby zapobiec powielaniu logiki w każdej wtyczce CSI.

Węzeł CSI

Wtyczka węzła uruchamia zazwyczaj ten sam kod sterownika co wtyczka kontrolera. Działanie „w trybie węzła" oznacza jednak koncentrowanie się na takich zadaniach, jak montowanie dołączanych woluminów, ustanawianie ich systemu plików i montowanie woluminów w kapsułach. Żądania tych zachowań są realizowane za pośrednictwem komponentu `kubelet`. Wraz ze sterownikiem w kapsule często znajdują się następujące przyczepy:

node-driver-registrar

Wysyła żądanie rejestracji (*https://oreil.ly/kmkJh*) do komponentu `kubelet`, aby poinformować go o uruchomieniu sterownika CSI.

liveness-probe

Dostarcza informacji o kondycji sterownika CSI.

Implementowanie pamięci masowej jako usługi

Omówiliśmy kluczowe kwestie dotyczące pamięci masowej aplikacji, podstawowych funkcjonalności pamięci masowej dostępnych w Kubernetesie oraz integracji sterowników przy użyciu CSI. Nadszedł czas, aby złożyć te koncepcje i przyjrzeć się implementacji, która oferuje programistom pamięć masową jako usługę. Chcemy zapewnić deklaratywny sposób tworzenia żądań pamięci masowej

i udostępniania jej obciążeniom roboczym. Ponadto wolimy robić to dynamicznie, nie wymagając od administratora wstępnej alokacji i dołączania woluminów. W dodatku chcemy, by odbywało się to na żądanie na bazie potrzeb obciążeń roboczych.

Aby rozpocząć tę implementację, skorzystamy z usługi Amazon Web Services (AWS). Ta implementacja zapewnia integrację z systemem elastycznej blokowej pamięci masowej AWS (*https://oreil.ly/I4VVw*). Jeśli dokonałeś wyboru innego dostawcy, większość treści i tak pozostaje aktualna! Używamy akurat tego dostawcy po prostu jako konkretnego przykładu sposobu współpracy wszystkich elementów.

W następnym punkcie rozdziału zgłębimy kwestię instalowania integracji (sterownika) udostępniającej programistom opcje pamięci masowej, konsumującej pamięć masową za pomocą obciążeń roboczych, zmieniającej rozmiary woluminów i wykonującej migawki woluminów.

Instalacja

Instalacja to dość prosty proces składający się z dwóch kluczowych kroków:

1. Skonfigurowanie dostępu do dostawcy.
2. Wdrożenie w klastrze komponentów sterownika.

Dostawca — w naszej implementacji AWS — będzie wymagał od sterownika przedstawienia tożsamości, aby się upewnić, że ma on odpowiednie uprawnienia dostępu. W tym przypadku mamy do dyspozycji trzy opcje. Pierwszą z nich jest aktualizacja profilu instancji (*https://oreil.ly/fGWYd*) węzłów Kubernetesa. Dzięki temu nie będziemy musieli przejmować się poświadczeniami na poziomie Kubernetesa, konfigurując uniwersalne uprawnienia dla obciążeń roboczych, które będą mogły komunikować się z API usługi AWS. Drugą i prawdopodobnie najbezpieczniejszą opcją jest wprowadzenie usługi tożsamości, która będzie mogła zapewniać uprawnienia IAM (ang. *Identity and Access Management*) określonym obciążeniom roboczym. Przykładem może być projekt kiam (*https://github.com/uswitch/kiam*). To podejście omówiliśmy w rozdziale 10. Ostatnia opcja polega na dodaniu poświadczeń w obiekcie `Secret` (sekret), który zostanie zamontowany w sterowniku CSI. W tym modelu sekret mógłby wyglądać następująco:

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-secret
  namespace: kube-system
stringData:
  key_id: "AKIAWJQHICPELCJVKYNU"
  access_key: "jqWi1ut4KyrAHADIOrhH2Pd/vXpgqA9OZ3bCZ"
```

To konto będzie miało uprawnienia do manipulowania bazowym systemem pamięci masowej. Należy starannie zarządzać dostępem do tego sekretu. Więcej informacji na ten temat znajdziesz w rozdziale 7.

Po zaimplementowaniu tej konfiguracji można zainstalować komponenty CSI. Najpierw instalowany jest kontroler jako wdrożenie (`Deployment`). Przy uruchamianiu wielu replik stosowany jest wybór kontrolera głównego, aby określić, która instancja powinna być aktywna. Następnie instalowana jest wtyczka węzła w postaci zbioru demonów (`DaemonSet`), który uruchamia na każdym węźle po jednej

kapsule. Po zainicjowaniu instancje wtyczki węzła zarejestrują się w swoich komponentach `kubelet`. Następnie `kubelet` zgłosi węzeł z obsługą CSI, tworząc obiekt `CSINode` dla każdego węzła Kubernetesa. Dane wyjściowe klastra z trzema węzłami są następujące:

```
$ kubectl get csinode

NAME                                         DRIVERS  AGE
ip-10-0-0-205.us-west-2.compute.internal     1        97m
ip-10-0-0-224.us-west-2.compute.internal     1        79m
ip-10-0-0-236.us-west-2.compute.internal     1        98m
```

Jak widać, na liście znajdują się trzy węzły, a każdy z nich ma po jednym zarejestrowanym sterowniku. Zbadanie YAML-a jednego z obiektów `CSINode` ujawnia następujące elementy:

```
apiVersion: storage.k8s.io/v1
kind: CSINode
metadata:
  name: ip-10-0-0-205.us-west-2.compute.internal
spec:
  drivers:
    - allocatable:
        count: 25 (1)
      name: ebs.csi.aws.com
      nodeID: i-0284ac0df4da1d584
      topologyKeys:
        - topology.ebs.csi.aws.com/zone (2)
```

(1) Maksymalna liczba woluminów dozwolona w tym węźle.

(2) Gdy węzeł zostanie wybrany dla jakiegoś obciążenia roboczego, ta wartość zostanie przekazana w obiekcie `CreateVolumeRequest`, aby sterownik miał informacje, *gdzie* utworzyć wolumin. Jest to istotne w przypadku systemów pamięci masowej, w których węzły w klastrze nie będą miały dostępu do tego samego magazynu danych — np. gdy w AWS kapsuła zostanie rozplanowana w strefie dostępności, wolumin musi zostać utworzony w tej samej strefie.

Ponadto sterownik jest oficjalnie zarejestrowany w klastrze. Szczegóły można znaleźć w obiekcie `CSIDriver`:

```
apiVersion: storage.k8s.io/v1
kind: CSIDriver
metadata:
  name: aws-ebs-csi-driver (1)
  labels:
    app.kubernetes.io/name: aws-ebs-csi-driver
spec:
  attachRequired: true (2)
  podInfoOnMount: false (3)
  volumeLifecycleModes:
  - Persistent (4)
```

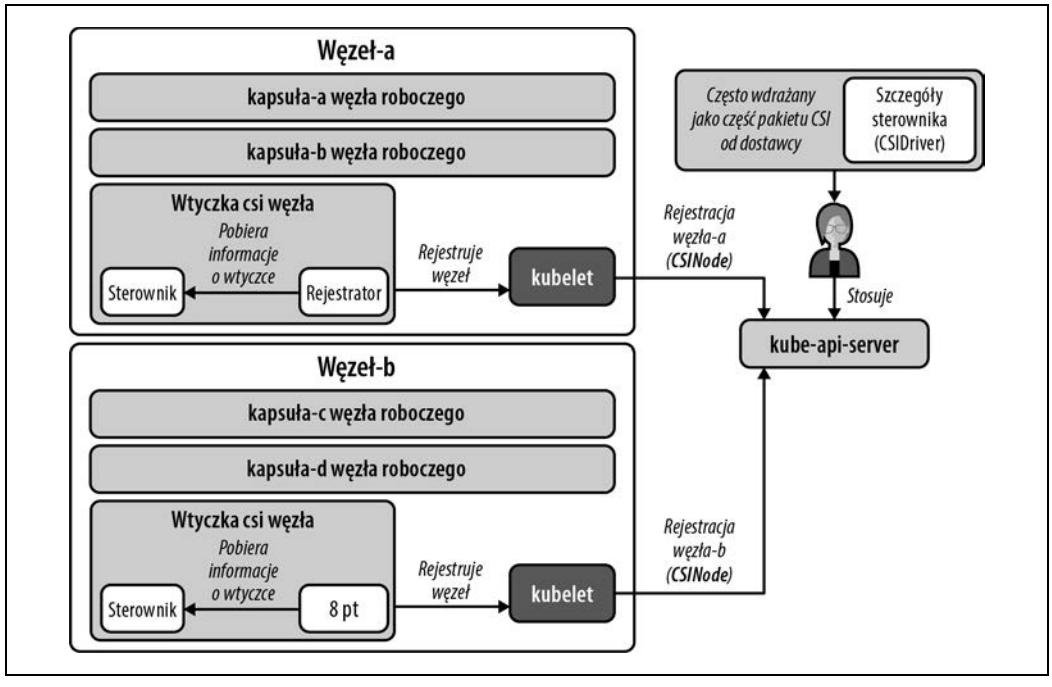
(1) Nazwa dostawcy, którego reprezentuje ten sterownik. Ta nazwa będzie powiązana z klasami pamięci oferowanymi użytkownikom platformy.

(2) Określa, że operacja dołączania musi zostać zakończona przed zamontowaniem woluminów.

(3) Podczas konfigurowania punktu montowania nie jest wymagane przekazywanie jako kontekstu metadanych kapsuły.

❹ Domyślny model alokowania woluminów trwałych. Wsparcie dla woluminów określanych bezpośrednio w definicji kapsuły (*https://kubernetes-csi.github.io/docs/ephemeral-local-volumes.html*) można włączyć przez ustawienie tej opcji na `Ephemeral`. W trybie efemerycznym pamięć masowa jest utrzymywana tylko w trakcie istnienia kapsuły.

Ustawienia i obiekty, które do tej pory zbadaliśmy, to artefakty naszego procesu ładowania początkowego. Obiekt `CSIDriver` ułatwia odnajdywanie szczegółów sterownika i został dołączony do pakietu wdrażania sterownika. Obiekty `CSINode` są zarządzane przez `kubelet`. Przyczepa rejestrowania jest zawarta w kapsule wtyczki węzła. Pobiera szczegóły ze sterownika CSI i rejestruje sterownik w agencie `kubelet`. Następnie `kubelet` zgłasza liczbę sterowników CSI dostępnych na każdym hoście. Ten proces ładowania początkowego przedstawiliśmy na rysunku 4.2.



Rysunek 4.2. Obiekt CSIDriver jest wdrażany i stanowi część pakietu, podczas gdy wtyczka węzła rejestruje się w agencie kubelet, który z kolei tworzy obiekty CSINode i zarządza nimi

Udostępnianie opcji pamięci masowej

Aby zapewnić programistom opcje pamięci masowej, trzeba utworzyć obiekty `StorageClass`. W tym scenariuszu założymy, że istnieją dwa typy pamięci masowej, które chcemy udostępnić. Pierwszą opcją jest udostępnienie taniego dysku, który może być używany do tych potrzeb obciążeń roboczych, które są związane z utrwalaniem danych. Często aplikacje nie potrzebują dysku SSD, ponieważ utrwalają po prostu pewne pliki, niewymagające szybkich operacji odczytu i zapisu. W związku z tym tani dysk (HDD) będzie opcją domyślną. Chcielibyśmy zaoferować również szybszy dysk SSD ze skonfigurowanymi niestandardowymi IOPS (*https://en.wikipedia.org/wiki/IOPS*) na gigabajt. Naszą ofertę przedstawiliśmy w tabeli 4.1; ceny odzwierciedlają koszty usługi AWS w chwili, gdy powstawał ten rozdział.

Tabela 4.1. Oferty pamięci masowej

Nazwa oferty	Typ pamięci masowej	Maksymalna przepustowość na wolumin	Koszt usługi AWS
default-block	HDD (optimized)	40 – 90 MB/s	0,045 dolara za gigabajt na miesiąc
performance-block	SSD (io1)	~1000 MB/s	0,125 dolara za gigabajt na miesiąc + 0,065 dolara za alokowane IOPS na miesiąc

Aby przygotować te oferty, dla każdej z nich utworzymy klasę pamięci masowej. Wewnątrz każdej klasy pamięci masowej znajduje się pole `parameters`. W tym miejscu możemy skonfigurować ustawienia z tabeli 4.1.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: default-block ❶
  annotations:
    storageclass.kubernetes.io/is-default-class: "true" ❷
provisioner: ebs.csi.aws.com ❸
allowVolumeExpansion: true ❹
volumeBindingMode: WaitForFirstConsumer ❺
parameters:
  type: st1 ❻
---
kind: StorageClass ❼
apiVersion: storage.k8s.io/v1
metadata:
  name: performance-block
provisioner: ebs.csi.aws.com
parameters:
  type: io1
  iopsPerGB: "20"
```

❶ Jest to nazwa oferty pamięci masowej, którą zapewniamy użytkownikom platformy. Będzie można odwoływać się do niej w obiektach `PersistentVolumeClaim`.

❷ Ustawia tę ofertę jako domyślną. Jeśli obiekt `PersistentVolumeClaim` zostanie utworzony *bez* określonej klasy pamięci masowej, użyta zostanie klasa `default-block`.

❸ Mapowanie na sterownik CSI, który powinien być używany.

❹ Dopuszcza zwiększanie rozmiaru woluminu przez zmiany w obiekcie `PersistentVolumeClaim`.

❺ Wolumin nie zostanie alokowany, dopóki jakaś kapsuła nie użyje obiektu `PersistentVolumeClaim`. Spowoduje to utworzenie woluminu w odpowiedniej strefie dostępności rozplanowanej kapsuły. Zapobiega to również tworzeniu w AWS woluminów przez osierocone PVC, za co byłaby naliczana opłata.

❻ Określa rodzaj pamięci, którą sterownik powinien pozyskać w celu zaspokojenia żądań.

❼ Druga klasa, która została dostosowana do dysku SSD o dużej wydajności.

Korzystanie z pamięci masowej

Gdy wymienione w poprzednim punkcie elementy zostaną przygotowane, użytkownicy będą mogli korzystać ze zdefiniowanych klas `StorageClass`. Zaczniemy od przyjrzenia się doświadczeniu programistycznemu związanemu z żądaniami pamięci masowej. Potem przeanalizujemy wewnętrzne mechanizmy zaspokajania tych żądań. Na początek zobaczmy, co otrzymuje programista, gdy wyświetli listę dostępnych klas `StorageClass`:

```
$ kubectl get storageclasses.storage.k8s.io

NAME                          PROVISIONER        RECLAIMPOLICY    VOLUMEBINDINGMODE
default-block (default)       ebs.csi.aws.com    Delete           Immediate
performance-block             ebs.csi.aws.com    Delete           WaitForFirstConsumer

ALLOWVOLUMEEXPANSION
true
true
```

Przez umożliwienie programistom tworzenia żądań PVC zezwalasz im na odwoływanie się do *dowolnej* klasy pamięci masowej. Jeśli jest to problematyczne, możesz rozważyć wdrożenie walidacyjnego sterowania dostępem, aby oceniać poprawność żądań. Omówimy ten temat w rozdziale 8.

Załóżmy, że programista chce udostępnić aplikacji tańszy dysk HDD i bardziej wydajny SSD. W takim przypadku tworzymy dwa obiekty `PersistentVolumeClaim`. Nazywamy je odpowiednio `pvc0` i `pvc1`:

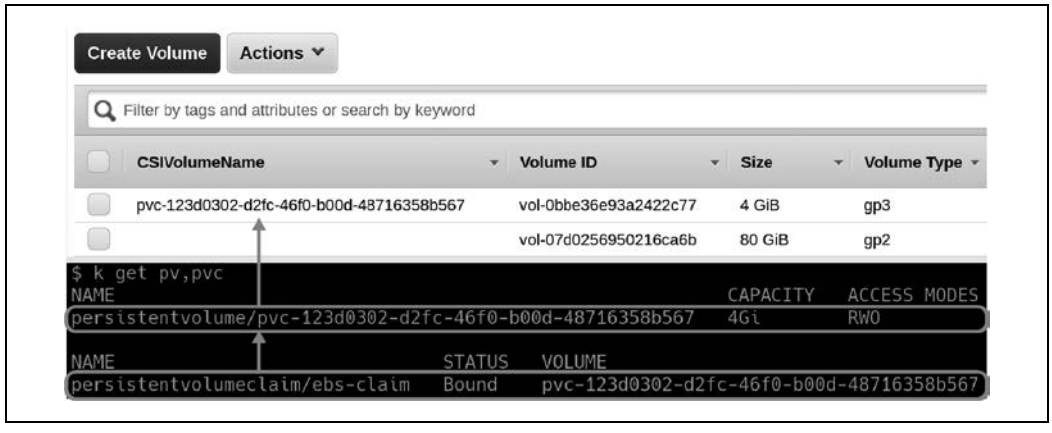
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc0 ❶
spec:
  resources:
    requests:
      storage: 11Gi
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc1
spec:
  resources:
    requests:
      storage: 14Gi
  storageClassName: performance-block ❷
```

❶ Użycie domyślnej klasy pamięci masowej (`default-block`), co zakłada również zastosowanie innych wartości domyślnych, takich jak RWO i typ pamięci masowej systemu plików.

❷ Upewnienie się, że od sterownika zażądana zostanie klasa `performance-block`, a nie `default-block`.

Na podstawie konfiguracji obiektów `StorageClass` te dwie pamięci masowe będą się charakteryzować różnymi zachowaniami w zakresie alokacji. Wydajna pamięć masowa (z obiektu `pvc1`) jest tworzona w AWS jako niepodłączony wolumin. Ten wolumin można szybko podłączyć i będzie gotowy do użycia. Domyślna pamięć masowa (`pvc0`) będzie się znajdować w stanie oczekiwania (`Pending`), w którym

klaster czeka z alokowaniem magazynu danych w AWS, aż jakaś kapsuła użyje tego żądania PVC. Gdy kapsuła w końcu użyje żądania, alokowanie będzie wymagało więcej pracy, mimo to nie zostaniesz obciążony opłatą za niewykorzystaną przestrzeń dyskową! Relację między żądaniem z Kubernetesa a woluminem w AWS pokazaliśmy na rysunku 4.3.



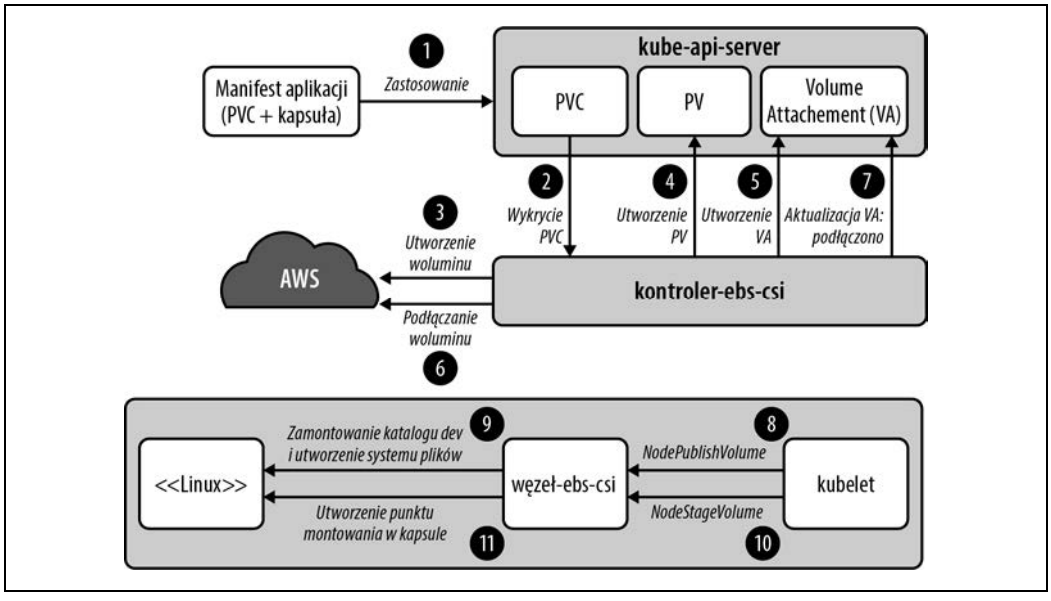
Rysunek 4.3. Pamięć masowa pv1 została alokowana w usłudze AWS jako wolumin i propagowany jest obiekt CSIVolumeName w celu ułatwienia korelacji. pv0 nie będzie miała utworzonego odpowiedniego woluminu, dopóki nie odwoła się do niej jakaś kapsuła

Załóżmy teraz, że programista tworzy dwie kapsuły. Jedna odwołuje się do obiektu `pv0`, a druga do `pv1`. Gdy obie kapsuły zostaną rozdysponowane na węzeł, zostanie do niego dołączony wolumin. Zanim to nastąpi w przypadku `pv0`, wolumin zostanie utworzony również w AWS. Po rozplanowaniu kapsuł i dołączeniu woluminów tworzony jest system plików, a pamięć masowa jest montowana w kontenerze. Ponieważ są to woluminy trwałe, wprowadziliśmy teraz model, w którym wolumin może być przenoszony wraz z kapsułą, nawet jeśli zostanie ona rozdysponowana ponownie na inny węzeł. Całościowy przepływ realizacji żądania przydzielenia samoobsługowej pamięci masowej pokazaliśmy na rysunku 4.4.

W debugowaniu interakcji pamięci masowej z CSI szczególnie przydatne są zdarzenia. Ponieważ alokowanie, podłączanie i montowanie są celami zaspokojenia żądania PVC, należy przeglądać zdarzenia z tych obiektów, gdyż różne komponenty informują o tym, co zrobiły. Łatwym sposobem na przeglądanie tych zdarzeń jest użycie polecenia `kubectl describe -n $PRZESTRZEŃ_NAZW pvc $NAZWA_PVC`.

Zmiana rozmiaru woluminu

Sterownik `aws-ebs-csi-driver` obsługuje funkcjonalność zmiany rozmiaru woluminu. W większości implementacji CSI do wykrywania zmian w obiektach `PersistentVolumeClaim` używany jest kontroler `external-resizer`. Informacja o wykryciu zmiany rozmiaru jest przekazywana do sterownika, który zwiększa pojemność woluminu. W tym przypadku sterownik uruchomiony we wtyczce kontrolera umożliwia zmianę rozmiaru za pomocą interfejsu API AWS EBS.



Rysunek 4.4. Kompleksowy przepływ współpracy sterownika i Kubernetesa w celu zaspokojenia żądania pamięci masowej

Gdy pojemność woluminu zostanie zwiększona w usłudze EBS, nowe miejsce *nie będzie* od razu dostępne dla kontenera. Dzieje się tak, ponieważ system plików nadal zajmuje jedynie pierwotną przestrzeń. Trzeba poczekać, aż zostanie on rozszerzony przez instancję sterownika wtyczki węzła. To wszystko może się odbyć *bez* zamykania kapsuły. Rozszerzenie systemu plików można zobaczyć w poniższych dziennikach sterownika CSI wtyczki węzła:

```
mount_linux.go: Attempting to determine if disk "/dev/nvme1n1" is formatted
using blkid with args: ([-p -s TYPE -s PTTYPE -o export /dev/nvme1n1])

mount_linux.go: Output: "DEVNAME=/dev/nvme1n1\nTYPE=ext4\n", err: <nil>

resizefs_linux.go: ResizeFS.Resize - Expanding mounted volume /dev/nvme1n1

resizefs_linux.go: Device /dev/nvme1n1 resized successfully
```

Kubernetes nie obsługuje zmniejszania rozmiaru woluminu przez dokonanie zmiany w odpowiednim polu obiektu PVC. Jeżeli sterownik CSI nie zapewnia obejścia tego problemu, to zmniejszenie rozmiaru bez ponownego utworzenia woluminu może nie być możliwe. Pamiętaj o tym podczas zwiększania pojemności woluminów.

Migawki

Okresowe tworzenie kopii zapasowych danych woluminów używanych przez kontenery jest możliwe dzięki funkcjonalności migawki. Funkcjonalność ta jest często rozdzielona na dwa kontrolery, które odpowiadają za dwa różne zasoby CRD, obejmujące obiekty `VolumeSnapshot` i `VolumeContentSnapshot`. Generalnie obiekty `VolumeSnapshot` odpowiadają za cykl życia woluminów. Na podstawie tych obiektów kontroler `external-snapshotter` zarządza obiektami `VolumeContentSnapshot`. Ten kontroler jest zwykle uruchamiany jako przyczepa we wtyczce kontrolera CSI i przekazuje żądania do sterownika.

W chwili gdy powstaje ten rozdział, obiekty te są implementowane jako CRD, a nie jako podstawowe obiekty API Kubernetesa. Wymaga to wcześniejszego wdrożenia definicji CRD przez sterownik CSI lub dystrybucję Kubernetesa.

Podobnie jak w przypadku oferowania pamięci masowej za pośrednictwem klas `StorageClass`, tworzenie migawek jest obsługiwane przez wprowadzenie klasy `Snapshot`. Tę klasę reprezentuje następujący YAML:

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshotClass
metadata:
  name: default-snapshots
driver: ebs.csi.aws.com (1)
deletionPolicy: Delete (2)
```

(1) Sterownik, do którego ma być delegowane żądanie migawki.

(2) Określa, czy wraz z obiektem `VolumeSnapshot` ma być usuwany także obiekt `VolumeSnapshotContent`. W efekcie może zostać usunięty rzeczywisty wolumin (w zależności od obsługi zapewnianej przez dostawcę).

Obiekt `VolumeSnapshot` można utworzyć w przestrzeni nazw aplikacji i obiektu `PersistentVolumeClaim`. Przykład pokazaliśmy w poniższym listingu:

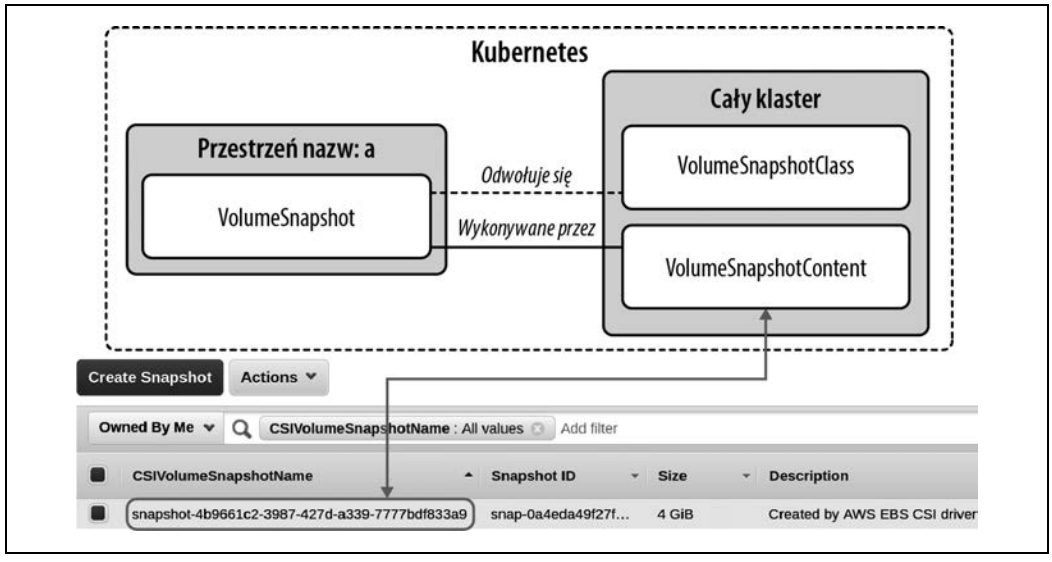
```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshot
metadata:
  name: snap1
spec:
  volumeSnapshotClassName: default-snapshots (1)
  source:
    persistentVolumeClaimName: pvc0 (2)
```

(1) Klasa wskazująca sterownik, którego należy użyć.

(2) Żądanie woluminu wskazujące wolumin, którego migawkę należy zrobić.

Istnienie tego obiektu informuje o potrzebie utworzenia obiektu `VolumeSnapshotContent`, który ma zasięg obejmujący cały klaster. Wykrycie obiektu `VolumeSnapshotContent` spowoduje zażądanie utworzenia migawki, a sterownik spełni je, komunikując się z usługą EBS Amazona. Po spełnieniu żądania `VolumeSnapshot` zgłosi stan `readyToUse` (gotowy do użytku). Relacje między opisanymi obiektami przedstawiliśmy na rysunku 4.5.

Mając migawkę, możemy zbadać scenariusz utraty danych. Niezależnie od tego, czy oryginalny wolumin został przypadkowo usunięty, miał awarię, czy też został usunięty z powodu przypadkowego wykasowania obiektu `PersistentVolumeClaim`, możemy przywrócić dane. W tym celu tworzymy nowy obiekt `PersistentVolumeClaim` z określonym polem `spec.dataSource`. Pole to obsługuje odwoływanie się do obiektu `VolumeSnapshot`, który może zapełnić danymi nowe żądanie. Z poprzednio utworzonej migawki odtwarzany jest następujący manifest:



Rysunek 4.5. Różne obiekty i ich relacje, tworzące przepływ wykonywania dla migawki

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-reclaim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: default-block
  resources:
    requests:
      storage: 600Gi
  dataSource:
    name: snap1 (1)
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
```

(1) Instancja `VolumeSnapshot`, która odwołuje się do migawki EBS w celu uzupełnienia nowego PVC.

Gdy odtworzona zostanie kapsuła, która ma się odwoływać do tego nowego żądania, do kontenera powróci stan utrwalony w ostatniej migawce! Mamy teraz dostęp do wszystkich podstawowych funkcjonalności pomocnych w budowaniu solidnych rozwiązań do tworzenia kopii zapasowych i odzyskiwania sprawności po awarii. Rozwiązania te mogą obejmować planowanie migawek za pomocą obiektów `CronJob`, napisanie niestandardowego kontrolera lub użycie do tworzenia kopii zapasowych obiektów Kubernetesa wraz z woluminami danych zgodnie z harmonogramem takich narzędzi jak Velero (*https://velero.io*).

Podsumowanie

W tym rozdziale omówiliśmy różne tematy związane z pamięcią masową kontenerów. Po pierwsze, musimy dobrze zrozumieć wymagania aplikacji, aby opierać nasze decyzje techniczne na jak największej ilości informacji. Po drugie, powinniśmy mieć pewność, że nasz bazowy dostawca pamięci masowej może zaspokoić te wymagania, a my dysponujemy specjalistyczną wiedzą operacyjną (jeśli jest potrzebna) do ich obsługi. A po trzecie, musimy zadbać o integrację między orkiestratorem a bazowym systemem pamięci masowej, aby bez biegłości w tym drugim zapewnić programistom dostęp do takiej pamięci masowej, jakiej potrzebują.

Kubernetes: zbuduj najlepsze środowisko dla aplikacji w swojej firmie

Kubernetes zmienił podejście do budowy i wdrażania oprogramowania korporacyjnego. Przedsiębiorstwa szybko zrozumiały, że dzięki tej potężnej technologii mogą korzystać z zalet wysoce dostępnych, samonaprawiających się i autoskalujących wdrożeń oprogramowania. Systemy są zdolne do automatycznego zapewniania pożądanych reakcji na podstawie zadanych warunków, a przy tym są szybsze i bardziej niezawodne od jakichkolwiek operacji wykonywanych ręcznie. Jednak ceną za ten postęp jest konieczność poradzenia sobie z większą złożonością.

To książka przeznaczona dla osób, które chcą z powodzeniem uruchomić Kubernetes w środowisku produkcyjnym jako platformę dla aplikacji przedsiębiorstwa. Zawiera wiele wniosków płynących z praktycznych doświadczeń autorów, omawia też kluczowe wyzwania i najlepsze praktyki. Pokazuje, w jaki sposób można sobie poradzić z różnymi kwestiami związanymi z technologiami, abstrakcjami i ze wzorcami, aby bez zbędnych problemów osiągnąć sukces w swoim wdrożeniu. Proces projektowy i wdrożeniowy potraktowano tu z dużą dozą pragmatyzmu i zwrócono uwagę na jego wczesne etapy. Omówiono także wiele punktów decyzyjnych i potencjalne przyczyny problemów, a poszczególne zagadnienia zostały poparte praktycznymi przykładami.

W książce między innymi:

- podstawy projektowania platform opartych na Kubernetesie
- praktyczne aspekty rozwiązywania problemów podczas budowania platformy
- wykorzystywanie architektury Kubernetesa w rozwijaniu platformy
- prowadzenie analiz przedprojektowych
- zasady wyboru stosowanych narzędzi i abstrakcji podczas pracy z Kubernetesem

Josh Rosso jest inżynierem oprogramowania. Pracował z Kubernetesem w CoreOS (Red Hat), Heptio i VMware.

Rich Lander jest inżynierem terenowym VMware. Pomaga przedsiębiorstwom wdrażać Kubernetes i technologie natywne dla chmury.

Alexander Brand jest inżynierem oprogramowania. Zajmuje się Kubernetesem i technologiami natywnymi chmury.

John Harris jest inżynierem personelu. Ma doświadczenie w pracy z narzędziami natywnymi dla chmury, platformami i ze wzorcami.

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

ISBN 978-83-283-8549-8
9 788328 385498

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 99,00 zł