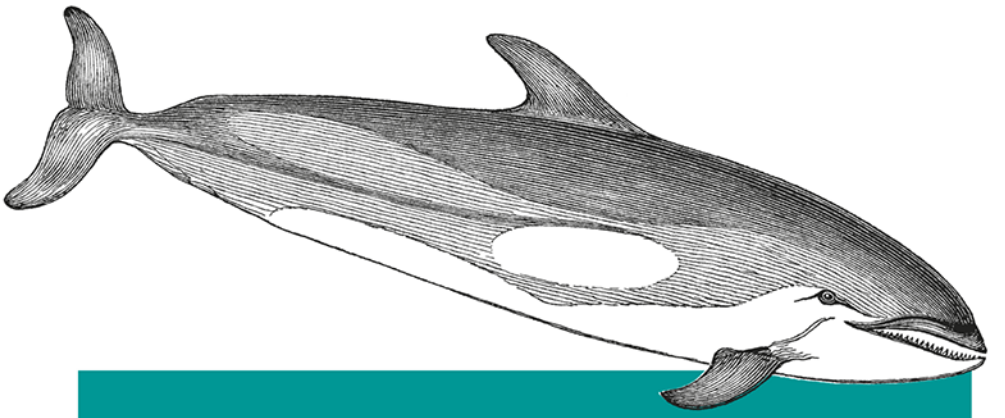
O'REILLY®



Kubernetes

Tworzenie niezawodnych systemów rozproszonych

Kelsey Hightower
Brendan Burns
Joe Beda

Helion

Tytuł oryginału: Kubernetes: Up and Running: Dive into the Future of Infrastructure

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-5235-3

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

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

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

Printed in Poland.

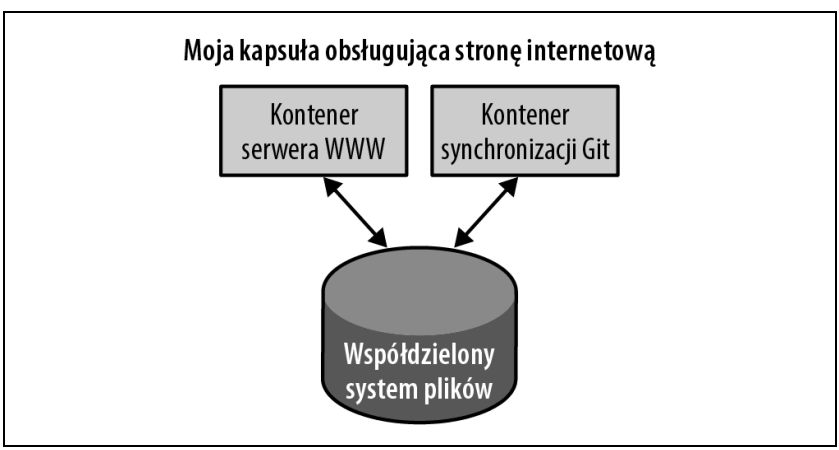
Spis treści

ROZDZIAŁ 5.

Kapsuły

W poprzednich rozdziałach omawialiśmy, jak można zabrać się za konteneryzowanie aplikacji, ale w rzeczywistych wdrożeniach kontenerowych aplikacji często trzeba kolokować wiele aplikacji w jednej niepodzielnej jednostce rozplanowanej na pojedynczej maszynie.

Na rysunku 5.1 przedstawiono kanoniczny przykład takiego wdrożenia; widać na nim kontener obsługujący żądania WWW i kontener synchronizujący system plików ze zdalnym repozytorium Git.



Rysunek 5.1. Przykładowa kapsuła z dwoma kontenerami i współdzielonym systemem plików

Na początku może wydawać się kuszące opakowanie zarówno serwera WWW, jak i synchronizatora Git w jeden kontener. Jednak po bliższej analizie jasne

stają się powody ich rozdzielenia. Po pierwsze, te dwa różne kontenery mają całkowicie odmienne wymagania dotyczące wykorzystania zasobów. Weźmy na przykład pamięć. Ponieważ serwer WWW obsługuje żądania użytkowników, chcemy zagwarantować, żeby zawsze był dostępny i responsywny. Z kolei synchronizator Git nie jest tak naprawdę skierowany do użytkownika i charakteryzuje się jakością usług typu *best effort*.

Załóżmy, że nasz synchronizator Git ma wyciek pamięci. Musimy zagwarantować, żeby synchronizator Git nie wykorzystywał pamięci, której chcemy używać dla serwera WWW, ponieważ mogłoby to wpłynąć na wydajność tego serwera, a nawet spowodować jego awarię.

Ten rodzaj izolacji zasobów jest dokładnie tym, w czym najlepiej sprawdzają się kontenery. Rozdzielając dwie aplikacje na dwa osobne kontenery, możemy zapewnić niezawodne działanie serwera WWW.

Oczywiście te dwa kontenery są symbiotyczne; nie ma sensu rozplanowywać serwera WWW na jednej maszynie, a synchronizatora Git na drugiej. Dlatego Kubernetes grupuje wiele kontenerów w pojedynczą niepodzielną jednostkę zwaną **kapsułą** (ang. *pod*).

Kapsuły w Kubernetes

Kapsuła to zbiór kontenerów aplikacji i woluminów działających w tym samym środowisku wykonawczym. Najmniejszymi możliwymi do wdrożenia w klastrze Kubernetes artefaktami są właśnie kapsuły, a nie kontenery. Oznacza to, że wszystkie kontenery w kapsule zawsze lądują na tej samej maszynie.

Poszczególne kontenery w ramach kapsuły działają we własnych grupach kontrolnych (cgroups), ale współdzielą wiele przestrzeni nazw Linuksa.

Aplikacje działające w tej samej kapsule współdzielą ten sam adres IP i przestrzeń portów (sieciową przestrzeń nazw), mają tę samą nazwę hosta (przestrzeń nazw UTS) i mogą komunikować się za pomocą natywnych kanałów komunikacji międzyprocesowej poprzez kolejki komunikatów System V IPC lub POSIX (przestrzeń nazw IPC). Aplikacje w różnych kapsułach są od siebie odizolowane; mają różne adresy IP, różne nazwy hostów itd. Kontenery w różnych kapsułach działających na tym samym węźle mogą równie dobrze znajdować się na różnych serwerach.

Myślenie w kategoriach kapsuł

Jednym z najczęstszych pytań, które pojawiają się przy adaptowaniu Kubernetes, jest: „Co należy umieścić w kapsule?”.

Czasami ludzie widzą kapsuły i myślą: „Aha! Kontener WordPress i kontener bazy danych MySQL powinny znajdować się w tej samej kapsule”. Jednak taki rodzaj kapsuły jest w rzeczywistości przykładem antywzorca dla budowy kapsuł. Istnieją ku temu dwa powody. Po pierwsze, WordPress i jego baza danych nie są tak naprawdę symbiotyczne. Jeżeli kontener WordPress i kontener bazy danych trafią na różne maszyny, nadal mogą ze sobą całkiem skutecznie współpracować, ponieważ komunikują się przez połączenie sieciowe. Po drugie, niekoniecznie powinieneś skalować WordPressa i bazę danych jako jednostkę. Sam WordPress jest głównie bezstanowy, w odpowiedzi na obciążenie frontendu możesz więc chcieć skalować frontendy WordPressa, tworząc kolejne kapsuły WordPressa. Skalowanie bazy danych MySQL jest znacznie trudniejsze i prawdopodobnie należałoby raczej zwiększyć ilość zasobów przypisanych do pojedynczej kapsuły MySQL. Jeśli zgrupujesz kontenery WordPress i MySQL w jednej kapsule, będziesz musiał użyć tej samej strategii skalowania dla obu kontenerów, co nie jest zbyt dobrym rozwiązaniem.

Ogólnie rzecz biorąc, pytanie, które należy zadać sobie przy projektowaniu kapsuł, jest następujące: „Czy te kontenery będą działać poprawnie, jeżeli wylądują na różnych maszynach?”. Jeśli odpowiedź brzmi „nie”, wtedy właściwym sposobem pogrupowania tych kontenerów jest kapsuła. Jeżeli odpowiedź brzmi „tak”, to prawdopodobnie prawidłowym rozwiązaniem jest wiele kapsuł. W przykładzie przywołanym na początku tego rozdziału dwa kontenery współdziałają poprzez lokalny system plików. Nie mogłyby działać poprawnie, gdyby zostały rozplanowane na różnych maszynach.

W pozostałej części tego rozdziału opiszemy, jak tworzyć, analizować i usuwać kapsuły w Kubernetes oraz zarządzać nimi.

Manifest kapsuły

Kapsuły są opisane w manifeście kapsuły. Manifest kapsuły jest po prostu plikiem tekstowym reprezentującym obiekt API Kubernetes. Twórcy Kubernetes mocno wierzą w **konfigurację deklaratywną**. Deklaratywna konfiguracja oznacza zapisywanie żądanego stanu świata w pliku konfiguracyjnym, a następnie

przesyłanie tej konfiguracji do usługi, która podejmuje działania mające zapewnić, że pożądany stan stanie się stanem rzeczywistym.

Konfiguracja deklaratywna różni się od **konfiguracji imperatywnej**, w której po prostu wykonuje się serię czynności (na przykład `apt-get install foo`), aby zmodyfikować świat. Lata doświadczeń w środowisku produkcyjnym nauczyły nas, że utrzymywanie pisemnego zapisu żądanego stanu systemu prowadzi do uzyskania łatwiejszego w zarządzaniu, niezawodnego systemu. Konfiguracja deklaratywna zapewnia wiele korzyści, w tym możliwość przeglądania kodu pod kątem konfiguracji i dokumentowania aktualnego stanu świata dla rozproszonych zespołów. Dodatkowo jest podstawą wszystkich zachowań samonaprawiania się w Kubernetes, które pozwalają aplikacji działać bez podejmowania działań przez użytkownika.

Serwer interfejsu API Kubernetes akceptuje i przetwarza manifesty kapsuł przed ich zapisaniem w pamięci trwałej (`etcd`). Planista również wykorzystuje API Kubernetes do znajdowania kapsuł, które nie zostały rozplanowane dla jakiegoś węzła. Następnie planista umieszcza te kapsuły na węzłach w zależności od zasobów i innych ograniczeń wyrażonych w manifestach kapsuł. Na tej samej maszynie można umieścić wiele kapsuł, o ile dostępne są wystarczające zasoby. Jednak rozplanowywanie wielu replik tej samej aplikacji na tę samą maszynę jest gorsze pod kątem niezawodności, ponieważ taka maszyna jest pojedynczą domeną awarii. W związku z tym planista Kubernetes próbuje sprawić, żeby kapsuły z tej samej aplikacji były rozmieszczane na różnych maszynach w celu zapewnienia niezawodności w przypadku wystąpienia awarii. Po rozplanowaniu na węzeł kapsuły nie przemieszczają się i aby je przenieść, trzeba je bezpośrednio zniszczyć i ponownie rozplanować.

Wiele instancji kapsuły można wdrożyć, powtarzając opisany tutaj przepływ pracy. Jednak do uruchamiania wielu instancji kapsuły lepiej nadają się obiekty `ReplicaSet` (zobacz rozdział 8.). (Okazuje się, że są one również lepsze przy uruchamianiu pojedynczej kapsuły, ale do tego dojdziemy później).

Tworzenie kapsuły

Najprostszym sposobem na utworzenie kapsuły jest wykonanie imperatywnego polecenia `kubectl run`. Aby uruchomić na przykład nasz serwer `kuard`, użyj następującego polecenia:

```
$ kubectl run kuard --image=gcr.io/kuar-demo/kuard-amd64:1
```

W ten sposób możesz sprawdzić status tej kapsuły:

```
$ kubectl get pods
```

Początkowo możesz zobaczyć kontener jako oczekujący (`Pending`), ale ostatecznie przekonasz się, że jego status zmieni się na działający (`Running`), co oznacza, że kapsuła i jej kontenery zostały pomyślnie utworzone.

Nie przejmuj się zanadto losowymi łańcuchami znaków dołączonymi na końcach nazw kapsuł. W tej metodzie kapsuły są tak naprawdę tworzone za pomocą obiektów `Deployment` i `ReplicaSet`, które omówimy w kolejnych rozdziałach.

Na razie możesz usunąć tę kapsułę, uruchamiając następujące polecenie:

```
$ kubectl delete deployments/kuard
```

Przejdziemy teraz do ręcznego napisania pełnego manifestu kapsuły.

Tworzenie manifestu kapsuły

Manifesty kapsuł można pisać przy użyciu formatów YAML lub JSON, ale zasadniczo preferowany jest YAML, ponieważ daje nieco większe możliwości edycji i oferuje opcję dodawanie komentarzy. Manifesty kapsuł (i inne obiekty API Kubernetes) należy tak naprawdę traktować w ten sam sposób, w jaki traktuje się kod źródłowy, a elementy takie jak komentarze pomagają objaśnić kapsułę nowym członkom zespołu, którzy stykają się z nią po raz pierwszy.

Manifesty kapsuł zawierają kilka kluczowych pól i atrybutów, między innymi: sekcję `metadata` do opisania kapsuły i jej etykiet, sekcję `spec` opisującą woluminy oraz listę kontenerów, które będą działały w kapsule.

W rozdziale 2. wdrożyliśmy `kuard` za pomocą następującego polecenia Dockera:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  gcr.io/kuar-demo/kuard-amd64:1
```

Podobny rezultat można osiągnąć, wpisując kod z listingu 5.1 do pliku o nazwie *kuard-pod.yaml*, a następnie używając poleceń `kubectl` w celu załadowania tego manifestu do Kubernetes.

Listing 5.1. kuard-pod.yaml

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Uruchamianie kapsuł

W poprzednim podrozdziale utworzyliśmy manifest kapsuły, który może być użyty do zainicjowania kapsuły uruchamiającej `kuard`. Aby uruchomić pojedynczą instancję `kuard`, skorzystaj z polecenia `kubectl apply`:

```
$ kubectl apply -f kuard-pod.yaml
```

Manifest kapsuły zostanie przesłany na serwer API Kubernetes. Następnie system Kubernetes rozplanuje tę kapsułę, aby została uruchomiona na zdrowym węźle w klastrze, gdzie będzie monitorowana przez proces demona `kubelet`. Nie przejmuj się, jeśli na razie nie rozumiesz tych wszystkich ruchomych części Kubernetes; w miarę lektury tej książki będziesz poznawał coraz więcej szczegółów.

Wyświetlanie listy kapsuł

Skoro mamy już uruchomioną kapsułę, dowiedzmy się o niej czegoś więcej. Używając narzędzia wiersza poleceń `kubectl`, możemy wyświetlić wszystkie kapsuły działające w klastrze. Na razie powinna w nim być tylko jedna kapsuła, którą utworzyliśmy w poprzednim kroku:

```
$ kubectl get pods
NAME        READY       STATUS        RESTARTS     AGE
kuard       1/1         Running       0            44s
```

Możesz zobaczyć nazwę kapsuły (`kuard`), którą nadaliśmy jej w poprzednim pliku YAML. Oprócz liczby gotowych kontenerów (`1/1`) dane wyjściowe pokazują również status, liczbę ponownych uruchomień kapsuły oraz wiek kapsuły.

Jeśli uruchomiłeś to polecenie natychmiast po utworzeniu kapsuły, możesz zobaczyć następujące dane wyjściowe:

```
NAME        READY       STATUS        RESTARTS     AGE
kuard       0/1         Pending       0            1s
```

Stan oczekujący (`Pending`) wskazuje, że kapsuła została przesłana, ale nie została jeszcze rozplanowana.

Jeżeli wystąpi jakiś bardziej znaczący błąd (na przykład próba utworzenia kapsuły z obrazem kontenera, który nie istnieje), również zostanie wyświetlony w polu statusu.

Domyślnie narzędzie wiersza poleceń `kubectl` zwraca zwięzłe informacje, ale możesz uzyskać ich więcej, stosując odpowiednią flagę wiersza poleceń. Dodanie flagi `-o wide` do jakiegokolwiek polecenia `kubectl` spowoduje wydrukowanie nieco większej ilości danych wyjściowych (nadal mieszczących się w pojedynczej linii). Dodanie flagi `-o json` lub `-o yaml` spowoduje wydrukowanie kompletnych obiektów odpowiednio w formacie JSON lub YAML.

Szczegółowe informacje o kapsule

Czasami jednoliniowy wydruk jest niewystarczający, ponieważ jest zbyt lapidarny. Kubernetes przechowuje dodatkowo różne zdarzenia związane z kapsułami, które są obecne w strumieniu zdarzeń, ale nie są dołączone do obiektu kapsuły.

W celu uzyskania większej ilości informacji na temat kapsuły (lub dowolnego obiektu Kubernetes) możesz użyć polecenia `kubectl describe`. Aby opisać na przykład kapsułę, którą wcześniej utworzyliśmy, możemy uruchomić następujące polecenie:

```
$ kubectl describe pods kuard
```

Te dane wyjściowe dostarczają wiele informacji o kapsule w różnych sekcjach. Na początku znajdują się informacje podstawowe:

```
Name:           kuard
Namespace:      default
Node:           node1/10.0.15.185
Start Time:     Sun, 02 Jul 2017 15:00:38 -0700
Labels:         <none>
Annotations:    <none>
Status:         Running
IP:             192.168.199.238
Controllers:    <none>
```

Następnie mamy informacje o kontenerach działających w kapsule:

```
Containers:
  kuard:
    Container ID:    docker://055095…
    Image:           gcr.io/kuar-demo/kuard-amd64:1
    Image ID:        docker-pullable://gcr.io/kuar-demo/
    ↪kuard-amd64@sha256:a580…
    Port:            8080/TCP
    State:           Running
      Started:       Sun, 02 Jul 2017 15:00:41 -0700
    Ready:           True
    Restart Count:   0
    Environment:     <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from
      ↪default-token-cg5f5 (ro)
```

Na koniec podawane są zdarzenia związane z kapsułą, takie jak czas jej rozplanowania, czas pobrania obrazu oraz informacje, czy i kiedy kapsuła musiała zostać ponownie uruchomiona z powodu niepowodzenia kontroli poprawności działania:

```
Events:
  Seen  From             SubObjectPath           Type     Reason     Message
  ----  ----             -------------           -------- ------     -------
  50s   default-scheduler                        Normal
  ↪Scheduled   Success…
  49s   kubelet, node1     spec.containers{kuard}   Normal
  ↪Pulling     pulling…
  47s   kubelet, node1     spec.containers{kuard}   Normal
  ↪Pulled      Success…
  47s   kubelet, node1     spec.containers{kuard}   Normal
  ↪Created     Created…
  47s   kubelet, node1     spec.containers{kuard}   Normal
  ↪Started     Started…
```

Usuwanie kapsuły

Gdy nadejdzie czas usunięcia kapsuły, możesz ją usunąć na podstawie nazwy:

```
$ kubectl delete pods/kuard
```

Możesz również użyć tego samego pliku, który został wykorzystany do jej utworzenia:

```
$ kubectl delete -f kuard-pod.yaml
```

Po usunięciu kapsuła **nie** jest natychmiast niszczona. Jeżeli uruchomisz w tym momencie polecenie `kubectl get pods`, zobaczysz, że kapsuła jest w stanie kończenia działania (`Terminating`). Wszystkie kapsuły mają **okres karencji** związany z kończeniem działania. Domyślnie jest to 30 sekund. Po wejściu kapsuły w stan `Terminating` nie otrzymuje już ona żadnych nowych żądań. W scenariuszu serwowania okres karencji jest ważny dla niezawodności, ponieważ pozwala kapsule przed usunięciem zakończyć wszelkie aktywne żądania, które mogą być w trakcie przetwarzania.

Należy zwrócić uwagę, że po usunięciu kapsuły wszelkie dane przechowywane w kontenerach związanych z tą kapsułą również zostaną usunięte. Jeśli chcesz zachować dane na wielu instancjach kapsuły, musisz użyć obiektu `Persistent ↪Volumes` opisanego na końcu tego rozdziału.

Uzyskiwanie dostępu do kapsuły

Gdy kapsuła jest już uruchomiona, możesz chcieć z różnych powodów uzyskać do niej dostęp. Na przykład by załadować usługę internetową, która działa w kapsule. Możesz chcieć przejrzeć jej dzienniki w celu debugowania napotkanego problemu lub nawet wykonać inne polecenia w kontekście kapsuły, które pomogą w debugowaniu. Poniższe punkty rozdziału opisują szczegółowo różne sposoby interakcji z kodem i danymi działającymi wewnątrz kapsuły.

Korzystanie z przekierowania portów

W dalszej części książki pokażemy, jak udostępnić usługę światu lub innym kontenerom korzystającym z mechanizmu równoważenia obciążenia, ale często chcemy po prostu uzyskać dostęp do konkretnej kapsuły, nawet jeśli nie obsługuje ona ruchu w internecie.

W tym celu możesz użyć funkcji przekierowania portów wbudowanej w API Kubernetes oraz narzędzia wiersza poleceń.

Gdy uruchomisz poniższe polecenie, zostanie utworzony bezpieczny tunel z Twojego lokalnego komputera, przez węzeł główny Kubernetes, do instancji kapsuły działającej na jednym z węzłów roboczych:

```
$ kubectl port-forward kuard 8080:8080
```

Dopóki polecenie przekierowania portów jest uruchomione, możesz uzyskać dostęp do kapsuły (w tym przypadku do interfejsu WWW kontenera `kuard`), wpisując w przeglądarce adres *http://localhost:8080*.

Uzyskiwanie większej ilości informacji za pomocą dzienników

Gdy aplikacja wymaga debugowania, pomocne jest pozyskanie większej ilości informacji, niż zapewnia `describe`, aby zrozumieć, co robi aplikacja. Kubernetes oferuje dwa polecenia do debugowania uruchomionych kontenerów. Polecenie `kubectl logs` pozwala pobrać bieżące dzienniki z działającej instancji:

```
$ kubectl logs kuard
```

Dodanie flagi `-f` pozwoli na ciągłe przesyłanie dzienników.

Użycie polecenia `kubectl logs` zawsze powoduje próbę uzyskania dzienników z aktualnie uruchomionego kontenera. Po dodaniu flagi `--previous` pobrane zostaną dzienniki z poprzedniej instancji kontenera. Jest to użyteczne na przykład wtedy, gdy kontenery są ciągle restartowane z powodu problemu z rozruchem kontenera.

Chociaż polecenie `kubectl logs` jest użyteczne w przypadku jednorazowego debugowania kontenerów w środowiskach produkcyjnych, zasadniczo bardziej przydaje się usługa agregacji dzienników. Istnieje kilka open source'owych narzędzi agregacji dzienników, takich jak `fluentd` i `elasticsearch`, a także wielu chmurowych dostawców rejestrowania dzienników. Usługi agregacji dzienników zapewniają więcej pojemności do przechowywania dzienników z dłuższego przedziału czasu oraz wszechstronne funkcje wyszukiwania i filtrowania dzienników. Ponadto oferują one możliwość agregowania dzienników z wielu kapsuł w jeden widok.

Uruchamianie poleceń w kontenerze przy użyciu exec

Czasami dzienniki są niewystarczające i żeby naprawdę określić, co się dzieje, musisz wykonywać polecenia w kontekście samego kontenera. W tym celu możesz użyć poniższego polecenia:

```
$ kubectl exec kuard date
```

Możesz również uzyskać interaktywną sesję, dodając flagi `-it`:

```
$ kubectl exec -it kuard ash
```

Kopiowanie plików do i z kontenerów

Czasami konieczne może być skopiowanie plików ze zdalnego kontenera na lokalny komputer w celu przeprowadzenia bardziej dogłębnej eksploracji. Do wizualizacji przechwytywania pakietów `tcpdump` możesz użyć na przykład narzędzia takiego jak Wireshark. Załóżmy, że wewnątrz kontenera w kapsule masz plik o nazwie */captures/capture3.txt*. Możesz bezpiecznie skopiować ten plik na lokalną maszynę, uruchamiając następujące polecenia:

```
$ kubectl cp <nazwa_kapsuły>:/captures/capture3.txt ./capture3.txt
```

Innym razem możesz potrzebować skopiować pliki z komputera lokalnego do kontenera. Powiedzmy, że chcesz skopiować plik *$HOME/config.txt* do zdalnego kontenera. W tym przypadku możesz uruchomić takie polecenie:

```
$ kubectl cp $HOME/config.txt <nazwa_kapsuły>:/config.txt
```

Ogólnie rzecz biorąc, kopiowanie plików do kontenera jest antywzorcem. Naprawdę powinieneś traktować zawartość kontenera jako niemutowalną. Jednak czasami jest to sposób na natychmiastowe powstrzymanie krwawienia i przywrócenie usługi do zdrowia, ponieważ jest szybszy niż zbudowanie, przesłanie do rejestru i wdrożenie nowego obrazu. Gdy krwawienie zostanie już zatrzymane, zawsze bardzo ważne jest, abyś od razu przystąpił do tworzenia i wdrażania obrazu, ponieważ w przeciwnym razie na pewno zapomnisz o lokalnej zmianie, którą wprowadziłeś do swojego kontenera, i nadpiszesz ją podczas późniejszego regularnego wprowadzania kolejnej wersji.

Kontrole działania

Po uruchomieniu aplikacji jako kontenera w Kubernetes aplikacja ta jest automatycznie utrzymywana przy życiu za pomocą **kontroli działania procesu** (ang. *process health check*). Ta kontrola po prostu zapewnia, że główny proces aplikacji będzie zawsze uruchomiony. Jeśli proces zostanie wyłączony, Kubernetes uruchomi go ponownie.

Jednak w większości przypadków prosta kontrola działania procesu jest niewystarczająca. Jeżeli proces ulegnie na przykład zakleszczeniu i nie będzie mógł

obsługiwać żądań, system sprawdzania działania nadal będzie traktował aplikację jako zdrową, ponieważ jej proces będzie wciąż uruchomiony.

Aby rozwiązać ten problem, Kubernetes wprowadził kontrole działania pod kątem **żywotności** aplikacji. Kontrole żywotności uruchamiają logikę charakterystyczną dla danej aplikacji (na przykład ładowanie strony internetowej), by zweryfikować, czy aplikacja nie tylko nadal działa, ale także czy działa poprawnie. Ponieważ kontrole żywotności są charakterystyczne dla danej aplikacji, należy je zdefiniować w manifeście kapsuły.

Sonda żywotności

Po uruchomieniu procesu `kuard` potrzebujemy sposobu weryfikacji, czy proces ten naprawdę działa poprawnie i nie powinien zostać zrestartowany. Sondy żywotności są definiowane dla poszczególnych kontenerów, co oznacza, że każdy kontener wewnątrz kapsuły jest sprawdzany osobno. W kodzie z listingu 5.2 dodajemy do naszego kontenera `kuard` sondę żywotności, która uruchamia zażądanie HTTP dla ścieżki */healthy* w kontenerze.

Listing 5.2. kuard-pod-health.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      livenessProbe:
        httpGet:
          path: /healthy
          port: 8080
        initialDelaySeconds: 5
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Powyższy manifest kapsuły używa sondy `httpGet` do wykonania żądania `GET` HTTP dla punktu końcowego `/healthy` na porcie `8080` kontenera `kuard`. Sonda ustawia początkowe opóźnienie `initialDelaySeconds` na `5`, a zatem wywoływanie

zostanie wykonane po upływie pięciu sekund od momentu utworzenia wszystkich kontenerów w kapsule. Sonda musi odpowiedzieć w ciągu jednosekundowego limitu czasu, a kod statusu HTTP musi być równy lub większy niż 200 i mniejszy niż 400, aby wykonanie żądania zostało uznane za udane. Kubernetes będzie wywoływał tę sondę co 10 sekund. Jeżeli zawiodą więcej niż trzy sondy, kontener ulegnie awarii i zostanie uruchomiony ponownie.

Możesz zobaczyć to w akcji, otwierając stronę statusu kontenera `kuard`. Utwórz kapsułę przy użyciu tego manifestu, a następnie przekieruj port do tej kapsuły:

```
$ kubectl apply -f kuard-pod-health.yaml
$ kubectl port-forward kuard 8080:8080
```

Wpisz w przeglądarce adres *http://localhost:8080*. Kliknij zakładkę *Liveness Probe* (sonda żywotności). Powinna wyświetlić się tabela zawierającą listę wszystkich sond, które otrzymała ta instancja `kuard`. Jeżeli klikniesz link *fail* na tej stronie, `kuard` zacznie „oblewać" kontrole poprawności działania. Poczekaj, aż Kubernetes zrestartuje kontener. W tym momencie wyświetlacz zostanie zresetowany i zainicjowany od nowa. Szczegółowe informacje na temat restartu można uzyskać za pomocą polecenia `kubectl describe kuard`. Sekcja *Events* (zdarzenia) będzie zawierała dane wyjściowe podobne do tych:

```
Killing container with id docker://2ac946...:pod "kuard_default(9ee84...)"
container "kuard" is unhealthy, it will be killed and re-created.
```

Sonda gotowości

Oczywiście sonda żywotności nie jest jedynym rodzajem kontroli działania, który chcemy wykorzystać. Kubernetes dokonuje rozróżnienia pomiędzy **żywotnością** a **gotowością**. Żywotność określa, czy aplikacja działa poprawnie. Kontenery, które nie przejdą kontroli żywotności, są restartowane. Gotowość opisuje, kiedy kontener jest gotowy do obsługi żądań użytkownika. Kontenery, które nie przejdą kontroli gotowości, są usuwane z mechanizmów równoważenia obciążenia usług. Sondy gotowości są konfigurowane podobnie do sond żywotności. Szczegółowo zbadamy usługi Kubernetes w rozdziale 7.

Połączenie sond gotowości i żywotności pomaga zagwarantować, że w klastrze będą działały tylko zdrowe kontenery.

Rodzaje kontroli działania

Oprócz kontroli HTTP Kubernetes obsługuje także kontrole działania `tcpSocket`, które otwierają gniazdo TCP; jeśli połączenie się powiedzie, kontrola się powiedzie. Ten styl sondowania jest przydatny dla aplikacji innych niż HTTP, na przykład baz danych lub interfejsów API innych niż oparte na HTTP.

Ponadto Kubernetes umożliwia przeprowadzanie sond `exec`. Wykonują one jakiś skrypt lub program w kontekście kontenera. Zgodnie z powszechnie przyjętą konwencją, jeżeli skrypt zwróci zerowy kod wyjścia, kontrola została wykonana z powodzeniem. W przeciwnym razie wykonanie sondy nie powiedzie się. Skrypty `exec` są często przydatne dla niestandardowych logik walidacji aplikacji, których nie można zweryfikować za pomocą wywołania HTTP.

Zarządzanie zasobami

Większość programistów przechodzi na kontenery i orkiestratory takie jak Kubernetes z powodu zapewnianych przez nie znacznych usprawnień w zakresie pakowania obrazów i niezawodnego wdrażania. Oprócz podstawowych elementów ukierunkowanych na aplikacje, które upraszczają tworzenie systemów rozproszonych, równie ważna jest możliwość zwiększenia ogólnego wykorzystania węzłów obliczeniowych, które tworzą klaster. Podstawowe koszty eksploatacji maszyny, wirtualnej lub fizycznej, są zasadniczo stałe niezależnie od tego, czy jest ona bezczynna, czy w pełni obciążona. Dlatego zapewnienie maksymalnej aktywności maszyn zwiększa efektywność każdego dolara wydanego na infrastrukturę.

Ogólnie rzecz biorąc, tę efektywność mierzy się za pomocą wskaźnika **wykorzystania**. Wykorzystanie jest definiowane jako ilość aktywnie używanych zasobów podzielona przez ilość zakupionych zasobów. Jeśli zakupisz na przykład maszynę jednordzeniową, a Twoja aplikacja będzie używać jedną dziesiątą rdzenia, wskaźnik wykorzystania wyniesie 10%.

Dzięki systemom planowania, takim jak Kubernetes, które zarządzają pakowaniem zasobów, można zwiększyć poziom wykorzystania do ponad 50%.

W tym celu musisz dostarczyć systemowi Kubernetes informacje o zasobach wymaganych przez aplikację, aby mógł ustalić optymalny sposób spakowania kontenerów na zakupionych maszynach.

Kubernetes umożliwia użytkownikom określenie dwóch różnych metryk zasobów. **Żądania** zasobów określają minimalną ilość zasobu wymaganą do uruchomienia aplikacji. **Limity** zasobów określają maksymalną ilość zasobów, które aplikacja może wykorzystywać. Obie te definicje zasobów są opisane bardziej szczegółowo w kolejnych punktach rozdziału.

Żądania zasobów: minimalne wymagane zasoby

W Kubernetes kapsuła żąda zasobów wymaganych do uruchomienia swoich kontenerów. Kubernetes gwarantuje, że te zasoby będą dostępne dla kapsuły. Najczęściej żądanymi zasobami są CPU i pamięć, ale Kubernetes obsługuje również inne typy zasobów, między innymi GPU.

Aby na przykład zażądać, żeby kontener `kuard` wylądował na maszynie z wolną połową zasobów procesora i otrzymał przydział 128 MB pamięci, definiujemy kapsułę tak, jak pokazano w listingu 5.3.

Listing 5.3. kuard-pod-resreq.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Zasoby są żądane dla poszczególnych kontenerów, a nie dla kapsuł. Całkowita ilość zasobów zażądanych przez kapsułę jest sumą wszystkich zasobów żądanych przez wszystkie kontenery w kapsule. Jest tak dlatego, że w wielu przypadkach różne kontenery mają bardzo różne wymagania odnośnie do zasobów procesora. Przykładowo w kapsule serwera WWW i synchronizatora danych serwer WWW jest skierowany do użytkownika i prawdopodobnie wymaga dużej mocy obliczeniowej, a synchronizator danych może obejść się mniejszą ilością zasobów procesora.

Żądania i limity

Żądania są używane podczas rozplanowywania kapsuł na węzły. Planista Kubernetes gwarantuje, że suma wszystkich żądań wszystkich kapsuł na węźle nie przekroczy pojemności węzła. Dlatego kapsuła działająca na węźle może mieć zagwarantowane posiadanie przynajmniej wymaganych zasobów. Co ważne, „żądanie" określa minimum. Nie określa górnego limitu zasobów, z których może korzystać kapsuła. Aby dowiedzieć się, co to oznacza, spójrz na przykład.

Wyobraź sobie, że mamy kontener, którego kod próbuje wykorzystać wszystkie dostępne rdzenie procesora. Załóżmy, że tworzymy kapsułę z tym kontenerem, która żąda połowy procesora. Kubernetes rozplanowuje tę kapsułę na maszynę z dwoma rdzeniami procesora.

Dopóki jest to jedyna kapsuła na maszynie, będzie wykorzystywać wszystkie dwa dostępne rdzenie, mimo że zażądała tylko połowy procesora.

Jeżeli na tę maszynę trafi druga kapsuła z tym samym kontenerem i tym samym żądaniem 0,5 CPU, wtedy każda kapsuła otrzyma jeden rdzeń.

Jeśli rozplanowana zostanie trzecia identyczna kapsuła, każda kapsuła otrzyma 0,66 rdzenia. Wreszcie, jeśli rozplanowana zostanie czwarta identyczna kapsuła, każda kapsuła otrzyma pół rdzenia, którego żąda, a węzeł będzie miał komplet.

Żądania CPU są realizowane za pomocą funkcjonalności `cpu-share` jądra Linuksa.

Żądania pamięci są obsługiwane podobnie do żądań procesora, ale istnieje pewna ważna różnica. Jeśli kontener przekracza swoją żądaną pamięć, system operacyjny nie może po prostu usunąć pamięci z procesu, ponieważ została ona przydzielona. Dlatego gdy w systemie zabraknie pamięci, `kubelet` kończy działanie kontenerów, których użycie pamięci jest większe niż żądana pamięć. Te kontenery są automatycznie restartowane, ale z mniejszą dostępną dla nich pamięcią na komputerze.

Ponieważ żądania zasobów gwarantują dostępność zasobów dla kapsuły, są one niezbędne w celu zapewnienia, by kontenery miały wystarczające zasoby w sytuacjach dużego obciążenia.

Ograniczanie wykorzystania zasobów za pomocą limitów

Oprócz ustawiania zasobów wymaganych przez kapsułę, co ustanawia minimalną ilość zasobów dostępnych dla kapsuły, możesz także definiować maksimum wykorzystania zasobów przez kapsułę, określając **limity** zasobów.

W naszym poprzednim przykładzie utworzyliśmy kapsułę `kuard`, która zażądała co najmniej pół rdzenia i 128 MB pamięci. W manifeście kapsuły w listingu 5.4 rozszerzamy tę konfigurację, aby dodać limit 1 CPU i 256 MB pamięci.

Listing 5.4. kuard-pod-reslim.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard

      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
          memory: "256Mi"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Po ustanowieniu ograniczeń dla kontenera jądro jest skonfigurowane w taki sposób, aby uniemożliwić wykorzystanie przekraczające te limity. Kontener z limitem procesora wynoszącym pół rdzenia będzie otrzymywał tylko pół rdzenia, nawet jeśli procesor będzie bezczynny. Kontener z limitem pamięci 256 MB nie będzie mógł wykorzystywać dodatkowej pamięci (nie powiedzie się na przykład `malloc`), jeśli jego wykorzystanie pamięci przekroczy 256 MB.

Utrwalanie danych za pomocą woluminów

Po usunięciu kapsuły lub ponownym uruchomieniu kontenera usuwane są również wszystkie dane z systemu plików kontenera. Często jest to dobre rozwiązanie, ponieważ raczej nie należy pozostawiać śmieciowego kodu, który akurat

zapisała jakaś bezstanowa aplikacja internetowa. W innych przypadkach posiadanie dostępu do trwałego magazynu danych jest ważną cechą zdrowej aplikacji. Kubernetes modeluje takie trwałe magazyny przechowywania danych.

Używanie woluminów z kapsułami

Aby dodać wolumin do manifestu kapsuły, musimy dodać do naszej konfiguracji dwa nowe fragmenty. Pierwszy to nowa sekcja `spec.volumes`. Ta tablica definiuje w manifeście kapsuły wszystkie woluminy, do których mogą uzyskiwać dostęp kontenery. Należy zwrócić uwagę, że nie wszystkie kontenery muszą montować wszystkie woluminy zdefiniowane w kapsule. Drugim dodatkiem jest tablica `volumeMounts` w definicji kontenera. Ta tablica określa woluminy, które są montowane w konkretnym kontenerze, oraz ścieżki montowania poszczególnych woluminów. Zauważ, że dwa różne kontenery w kapsule mogą montować ten sam wolumin na różnych ścieżkach.

Manifest w listingu 5.5 definiuje pojedynczy nowy wolumin o nazwie `kuard-data`, który kontener `kuard` montuje w ścieżce `/data`.

Listing 5.5. kuard-pod-vol.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  volumes:
    - name: "kuard-data"
      hostPath:
        path: "/var/lib/kuard"
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      volumeMounts:
        - mountPath: "/data"
          name: "kuard-data"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Różne sposoby używania woluminów z kapsułami

Istnieje wiele sposobów wykorzystania danych w aplikacji. Poniżej opisanych zostało kilka zalecanych wzorców dla Kubernetes.

Komunikacja/synchronizacja

W pierwszym przykładzie kapsuły zobaczyłeś, w jaki sposób dwa kontenery korzystały ze współdzielonego woluminu, aby serwować witrynę, zachowując przy tym synchronizację ze zdalną lokalizacją Git. W tym celu kapsuła używa woluminu `emptyDir`. Zakresem takiego woluminu jest czas życia kapsuły, ale może on być współdzielony między dwoma kontenerami, tworząc podstawę komunikacji między kontenerami synchronizacji Git i serwera WWW.

Pamięć podręczna

Aplikacja może używać woluminu, który jest istotny pod kątem wydajności, ale nie jest wymagany do poprawnego działania aplikacji. Aplikacja może na przykład przechowywać wstępnie zrenderowane miniatury większych obrazów. Oczywiście można je rekonstruować z oryginalnych obrazów, ale wówczas obsługa miniatur jest bardziej kosztowna. Taka pamięć podręczna powinna przetrwać ponowne uruchomienie kontenera spowodowane niepowodzeniami kontroli poprawności działania, dlatego również w przypadku użycia pamięci podręcznej dobrze sprawdza się `emptyDir`.

Trwałe dane

Czasami będziesz używać woluminu dla prawdziwie trwałych danych, czyli takich, które są niezależne od czasu życia konkretnej kapsuły i powinny być przesuwane między węzłami w klastrze, jeśli jakiś węzeł ulegnie awarii lub kapsuła z jakiegoś powodu zostanie przeniesiona na inną maszynę. W tym celu Kubernetes obsługuje wiele różnych zdalnych woluminów sieciowej pamięci masowej, w tym szeroko obsługiwane protokoły, takie jak NFS lub iSCSI, a także sieciowe pamięci masowe dostawców chmury, takie jak Elastic Block Store firmy Amazon, Files and Disk Storage od Azure i Persistent Disk firmy Google.

Montowanie systemu plików hosta

Inne aplikacje w rzeczywistości nie potrzebują trwałego woluminu, ale niektóre z nich wymagają dostępu do bazowego systemu plików hosta. Mogą na przykład potrzebować dostępu do systemu plików */dev* w celu uzyskiwania nieprzetworzonego dostępu na poziomie bloków do urządzenia w systemie. Dla takich przypadków Kubernetes obsługuje wolumin `hostDir`, który może montować w kontenerze dowolne lokalizacje w węźle roboczym.

W poprzednim przykładzie użyliśmy typu woluminu `hostDir`. Utworzony wolumin to */var/lib/kuard* na hoście.

Utrwalanie danych przy użyciu dysków zdalnych

Często wymagane jest, aby dane używane przez kapsułę pozostawały z kapsułą, nawet jeśli zostanie ona ponownie uruchomiona na innej maszynie hosta.

Aby to osiągnąć, można zamontować w kapsule zdalny wolumin sieciowy. Podczas korzystania z sieciowej pamięci masowej Kubernetes automatycznie montuje i demontuje odpowiednie woluminy za każdym razem, gdy korzystająca z nich kapsuła zostanie rozplanowana na konkretnej maszynie.

Istnieje wiele metod montowania woluminów w sieci. Kubernetes obsługuje standardowe protokoły, takie jak NFS i iSCSI, a także interfejsy API chmurowej pamięci masowej dla głównych dostawców chmury (zarówno publicznej, jak i prywatnej). W wielu przypadkach dostawcy usług w chmurze utworzą dla Ciebie również dysk, jeśli jeszcze go nie masz.

Oto przykład użycia serwera NFS:

```
...
# Powyżej reszta definicji kapsuły
volumes:
    - name: "kuard-data"
      nfs:
        server: my.nfs.server.local
        path: "/exports"
```

Wszystko razem

Wiele aplikacji ma charakter stanowy i jako takie muszą zachowywać wszelkie dane i zapewniać dostęp do bazowej pamięci masowej niezależnie od tego, na jakiej maszynie działa aplikacja. Jak widziałeś wcześniej, można to osiągnąć za

pomocą trwałego woluminu obsługiwanego przez sieciową pamięć masową. Chcemy również zapewnić, żeby przez cały czas działała zdrowa instancja aplikacji, co oznacza, iż chcemy mieć pewność, że kontener uruchamiający `kuard` będzie gotowy, zanim udostępnimy go klientom.

Poprzez połączenie trwałych woluminów, sond gotowości i żywotności oraz ograniczenia zasobów Kubernetes zapewnia wszystko, co jest potrzebne do niezawodnego działania stanowej aplikacji. Listing 5.6 zbiera to wszystko w jeden manifest.

Listing 5.6. kuard-pod-full.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  volumes:
    - name: "kuard-data"
      nfs:
        server: my.nfs.server.local
        path: "/exports"
  containers:
      - image: gcr.io/kuar-demo/kuard-amd64:1
        name: kuard
        ports:
          - containerPort: 8080
            name: http
            protocol: TCP
        resources:
          requests:
            cpu: "500m"
            memory: "128Mi"
          limits:
            cpu: "1000m"
             memory: "256Mi"
        volumeMounts:
          - mountPath: "/data"
            name: "kuard-data"
        livenessProbe:
          httpGet:
            path: /healthy
            port: 8080
          initialDelaySeconds: 5
          timeoutSeconds: 1
          periodSeconds: 10
          failureThreshold: 3
        readinessProbe:
```

```
    httpGet:
      path: /ready
      port: 8080
    initialDelaySeconds: 30
    timeoutSeconds: 1
    periodSeconds: 10
    failureThreshold: 3
```

Trwałe woluminy to szeroki temat, który obejmuje wiele różnych szczegółowych kwestii, między innymi sposób, w jaki współpracują ze sobą trwałe woluminy, żądania trwałych woluminów i dynamiczne zapewnianie woluminów. Ten temat został omówiony szczegółowo w rozdziale 13.

Podsumowanie

Kapsuły reprezentują niepodzielną jednostkę pracy w klastrze Kubernetes. Składają się z jednego kontenera lub większej liczby kontenerów współpracujących symbiotycznie. Aby utworzyć kapsułę, należy napisać manifest kapsuły i przesłać go do serwera API Kubernetes za pomocą jakiegoś narzędzia wiersza poleceń lub (rzadziej) poprzez wywołania HTTP i JSON wysyłane bezpośrednio do serwera.

Po przesłaniu manifestu do serwera interfejsu API planista Kubernetes znajduje maszynę, na której może zmieścić się kapsuła, i rozplanowuje kapsułę na tę maszynę. Po rozplanowaniu kapsuły demon `kubelet` na tej maszynie jest odpowiedzialny za utworzenie odpowiadających tej kapsule kontenerów oraz przeprowadzanie kontroli działania określonych w manifeście kapsuły.

Gdy kapsuła zostanie rozplanowana na węzeł, nie jest przeprowadzane ponowne rozplanowanie, jeśli ten węzeł ulegnie awarii. W celu uzyskania wielu replik tej samej kapsuły trzeba je utworzyć i nazwać ręcznie. W jednym z kolejnych rozdziałów przedstawimy obiekt `ReplicaSet` i pokażemy, jak może on zautomatyzować tworzenie wielu identycznych kapsuł i pomóc upewnić się, że będą one odtwarzane w przypadku awarii maszyny węzła.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

O'REILLY®

Niezawodny system rozproszony? Kubernetes, koniecznie!

Systemy rozproszone to odpowiedź na zwiększone wymagania wobec systemów informatycznych. Chodziło o takie cechy jak łatwe współużytkowanie zasobów, odporność na awarie, prostota rozbudowy czy skalowalność. Równocześnie jednak architekci i administratorzy systemów dostrzegli, że projektowanie, budowa i utrzymywanie systemu rozproszonego niesie ze sobą wyzwania, o jakich wcześniej nie mieli pojęcia. Wyjściem z tej trudnej sytuacji mogą być rozwijane w ostatnich latach kontenery i interfejsy API orkiestracji kontenerów.

Sięgnij po to znakomite wprowadzenie do Kubernetesa — udostępnionego na licencji open source orkiestratora klastrów (ang. *orchestrator*). Ta młoda technologia umożliwia programistom bezproblemowe tworzenie aplikacji. Dowiesz się, jak używać Kubernetesa do tworzenia aplikacji rozproszonej. Nauczysz się wykorzystywać narzędzia i interfejsy API do automatyzacji skalowalnych systemów rozproszonych, niezależnie od tego, czy są to usługi internetowe, aplikacje do uczenia maszynowego, czy klastry komputerów Raspberry Pi. Kubernetes i technologia kontenerowa poprawią zwinność, niezawodność i wydajność Twojej aplikacji!

W tej książce między innymi:

- zakres działania Kubernetesa
- tworzenie aplikacji kontenerowych za pomocą Dockera
- kontenery w Kubernetesie i ich środowisko wykonawcze
- uruchamianie aplikacji w środowisku produkcyjnym
- przykłady wdrażania rzeczywistych aplikacji w Kubernetesie

Kelsey Hightower — zajmował się różnymi technikami informatycznymi. Jest orędownikiem idei open source i prostych narzędzi, które ułatwiają ludziom życie. Namiętnie pisze kod w Go i prowadzi warsztaty techniczne.

Brendan Burns — jest znakomitym inżynierem, obecnie zajmuje się technologią Azure w Microsofcie. Jest jednym z inicjatorów projektu Kubernetes w Google.

Joe Beda — jest jednym z inicjatorów projektu Kubernetes, pełnił kluczową funkcję w tworzeniu Google Compute Engine (usługi VM w chmurze Google). Pracuje jako dyrektor techniczny firmy Heptio.

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

ISBN 978-83-283-5235-3

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 44,90 zł