KUBERNETES

PRZEWODNIK PO ORKIESTRACJI KONTENERÓW
I TWORZENIU NIEZAWODNYCH APLIKACJI

ALAN HOHN

no starch press

Helion

Tytuł oryginału: The Book of Kubernetes: A Complete Guide to Container Orchestration

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-1012-6

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

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

Printed in Poland.

Spis treści

1

Dlaczego kontenery są ważne?

Obecnie jest doskonały czas dla programistów oprogramowania. Tworzenie zupełnie nowych aplikacji i ich udostępnianie milionom użytkowników nigdy nie było tak łatwym zadaniem. Nowoczesne języki programowania, biblioteki otwartoźródłowe oraz platformy aplikacji umożliwiają tworzenie niewielkiej ilości kodu i otrzymywanie ogromnej funkcjonalności. Jednak pomimo ogromnej łatwości, z jaką można szybko rozpocząć pracę nad nową aplikacją, najlepsi programiści oprogramowania to ci, którzy wykraczają poza traktowanie platformy aplikacji jako rodzaju „czarnej skrzynki" i poznają sposób jej działania. Tworzenie niezawodnych, odpornych na awarie i skalowalnych aplikacji wymaga znacznie więcej niż umiejętności przygotowania komponentu typu Deployment w przeglądarce WWW bądź powłoce.

W tym rozdziale omówię architekturę aplikacji w świecie skalowanej, natywnej chmury. Wyjaśnię, dlaczego kontenery to preferowany sposób na opracowywanie i wdrażanie komponentów aplikacji, a także jak mechanizm koordynacji kontenerów pozwala spełnić podstawowe wymagania aplikacji umieszczanych w kontenerach. Na końcu przedstawię przykładową aplikację wdrożoną w Kubernetesie, aby w ten sposób dać Ci przedsmak możliwości technologii omówionych w książce.

Nowoczesna architektura aplikacji

Głównym motywem pojawiającym się na obszarze tworzenia nowoczesnego oprogramowania jest *skala*. Żyjemy w świecie aplikacji, z których równocześnie korzystają miliony użytkowników. Godna podziwu jest nie tylko możliwość osiągnięcia przez te aplikacje takiej skali, ale również oferowana przez nie stabilność. Poziom tej stabilności jest na tyle wysoki, że informacje o ewentualnej awarii trafiają na czołówki gazet, same awarie zaś są później przedmiotem analiz technicznych ciągnących się tygodniami bądź nawet miesiącami.

Gdy tak wiele nowoczesnych aplikacji działa na ogromną skalę, bardzo łatwo można przeoczyć ogromną ilość ciężkiej pracy, którą trzeba było włożyć w opracowanie, zbudowanie, wdrożenie i obsługę techniczną aplikacji tego kalibru, niezależnie od ich skali — przeznaczone dla tysięcy, milionów bądź miliardów użytkowników. Moim zadaniem jest wskazać w tym rozdziale, jakie wymagania musi spełniać platforma aplikacji, aby była w stanie zapewnić działanie skalowalnej i niezawodnej aplikacji. Przekonasz się, że konteneryzacja i Kubernetes spełniają te wymagania. Rozpocznę od omówienia trzech kluczowych atrybutów nowoczesnej architektury aplikacji. Następnie przejdę do przedstawienia trzech najważniejszych zalet tych atrybutów.

Atrybut: natywna chmura

Istnieje wiele sposobów na zdefiniowanie technologii **natywnej chmury** (dobrym punktem wyjścia będzie witryna fundacji Cloud Native Computing Foundation dostępna pod adresem *https://www.cncf.io/*). Dobrze jest rozpocząć od idei określenia, czym jest chmura i jakie są jej możliwości, co następnie pozwala ustalić, jakiego rodzaju architektura jest w stanie najlepiej wykorzystać chmurę.

W zasadzie chmurę można określić mianem abstrakcji. Temat abstrakcji pojawił się już we wprowadzeniu, więc w tym momencie doskonale już wiesz, że abstrakcje mają duże znaczenie dla przetwarzania. Konieczne jest dokładne zrozumienie abstrakcji, aby można było z nich poprawnie korzystać. W przypadku chmury dostawca zapewnia abstrakcję fizycznych procesorów, pamięci operacyjnej, pamięci masowej i sieci, pozwalając użytkownikom chmury, by jedynie zadeklarowali niezbędną ilość tych zasobów, które dzięki temu zostaną im przydzielone na żądanie. Aby otrzymać aplikację natywnej chmury, musi ona wykorzystać możliwości oferowane przez tę abstrakcję. Aplikacja, na ile to możliwe, nie powinna być powiązana z konkretnym hostem ani układem sieci, ponieważ nie chcesz ograniczać sobie elastyczności w zakresie podziału komponentów aplikacji między poszczególne hosty.

Atrybut: modułowość

Modułowość nie jest niczym nowym w architekturze aplikacji. Celem zawsze jest osiągnięcie *wysokiej spójności*, w której cała zawartość modułu będzie związana z pojedynczym celem, i *niskiego poziomu powiązania*, gdy moduły są zorganizowane w sposób minimalizujący komunikację między nimi. Jednak pomimo tego, że modułowość pozostaje ważnym celem projektowym, definicja modułu przedstawia się odmiennie. Zamiast po prostu traktować modułowość jako sposób organizacji kodu, nowoczesna architektura aplikacji obecnie preferuje przeniesienie modułowości do środowiska uruchomieniowego, dostarczając poszczególne moduły razem z oddzielnymi procesami systemu operacyjnego oraz zniechęcając do używania współdzielonego systemu plików bądź pamięci współdzielonej na potrzeby komunikacji. Skoro moduły są oddzielnymi procesami, komunikacja między nimi odbywa się za pomocą standardowych rozwiązań w zakresie komunikacji sieciowej (gniazda).

Takie podejście wydaje się marnotrawstwem zasobów sprzętowych. Znacznie oszczędniejszym i szybszym podejściem jest współdzielenie pamięci niż kopiowanie danych poprzez gniazdo. Jednak istnieją dwa dobre powody, dla których preferuje się używanie oddzielnych procesów. Pierwszy jest taki, że nowoczesny sprzęt jest coraz szybszy, więc za postać przedwczesnej optymalizacji można uznać przekonanie, iż gniazda nie będą wystarczająco szybkie na potrzeby danej aplikacji. Drugim powodem jest to, że niezależnie od wielkości serwera zawsze będzie istniało ograniczenie dotyczące liczby procesów, które mogą być w nim przetworzone, więc model pamięci współdzielonej ostatecznie ogranicza możliwości w zakresie rozbudowy rozwiązania.

Atrybut: mikrousługi

Nowoczesna architektura aplikacji bazuje na modułach w postaci oddzielnych procesów — te poszczególne moduły są zwykle bardzo małe. Teoretycznie chmura może dostarczać serwery wirtualne oferujące potężne możliwości. Jednak w praktyce używanie kilku potężnych serwerów okazuje się znacznie kosztowniejsze i mniej elastyczne niż wykorzystywanie wielu małych serwerów. Jeżeli moduły będą wystarczająco małe, można je wdrażać w tanich i powszechnie dostępnych serwerach, co z kolei oznacza możliwość wykorzystania w pełni sprzętu oferowanego przez dostawców usług chmury. Wprawdzie nie istnieje konkretna odpowiedź wskazująca, na ile małe powinny być moduły, aby można było je określać mianem *mikrousług*, ale dobra reguła wyjściowa będzie brzmiała następująco: mały na tyle, aby zapewniać elastyczność niezależnie od środowiska wdrożenia.

Architektura mikrousług oferuje również praktyczne korzyści dla zespołów organizacji. Odkąd Fred Brooks napisał książkę *Legendarny osobomiesiąc. Opowieści o inżynierii oprogramowania*, architekci zrozumieli, że zarządzanie zasobami ludzkimi jest jednym z największych wyzwań podczas opracowywania ogromnych i skomplikowanych systemów. Zbudowanie systemu na bazie wielu małych komponentów zmniejsza poziom skomplikowania testów, a także pozwala na zorganizowanie ogromnego zespołu ludzi, którzy nie będą wchodzili sobie w drogę.

JAK SIĘ PRZEDSTAWIA KWESTIA SERWERÓW APLIKACJI?

Idea modułowych usług ma długą historię. Jednym z popularnych rozwiązań na implementację takiego podejścia było tworzenie modułów przeznaczonych do działania w serwerze aplikacji, np. w środowisku Java Enterprise. Dlaczego nie można więc po prostu kontynuować tego wzorca i zastosować go w przypadku aplikacji?

Wprawdzie serwery aplikacji sprawdzają się w wielu zastosowaniach, ale nie zapewniają tego samego poziomu izolacji, który można uzyskać za pomocą architektury mikrousług. W efekcie pojawia się więcej problemów związanych ze wzajemnymi zależnościami, co znacznie utrudnia testowanie rozwiązania i ogranicza niezależność zespołów. Ponadto w przypadku typowego modelu istnienia pojedynczego serwera aplikacji w hoście, gdy zachodzi potrzeba wdrożenia wielu aplikacji, muszą one współdzielić tę samą przestrzeń procesu. Takie podejście okazuje się mniej elastyczne niż podejście skonteneryzowane, które przedstawię w niniejszej książce.

To oczywiście nie oznacza, że należy natychmiast wyrzucić do kosza używaną architekturę serwerów aplikacji i przejść do rozwiązań opartych na kontenerach. Konteneryzacja każdej architektury niesie ze sobą wiele korzyści. Gdy na przestrzeni czasu będziesz wdrażać tę architekturę, sensowne będzie przeniesienie kodu do architektury prawdziwie bazującej na mikrousługach, aby dzięki temu w pełni wykorzystać możliwości kontenerów i Kubernetesa.

W ten sposób przedstawiłem trzy najważniejsze atrybuty nowoczesnej architektury. W kolejnych punktach omówię trzy najważniejsze zalety oferowane przez te atrybuty.

Zaleta: skalowalność

Zacznę od przedstawienia najprostszej możliwej aplikacji. Utworzony zostaje pojedynczy plik wykonywalny, który jest uruchomiony w pojedynczym komputerze i współpracuje jednocześnie tylko z jednym użytkownikiem. Załóżmy, że musimy rozbudować tę aplikację, aby mogła współpracować jednocześnie z milionami użytkowników. Nie ulega wątpliwości, że niezależnie od tego, jak potężny serwer zostanie użyty, to ostatecznie nadejdzie moment, w którym dostępne zasoby tego serwera staną się wąskim gardłem. Nie ma tutaj znaczenia, czego będzie dotyczyło wąskie gardło — mocy obliczeniowej, pamięci operacyjnej, pamięci masowej czy przepustowości łącza. Gdy tylko wąskie gardło się pojawi, aplikacja nie będzie w stanie zapewnić obsługi kolejnych użytkowników bez niekorzystnego wpływu na wydajność działania dla dotychczasowych.

Jedynym możliwym rozwiązaniem tego problemu jest zakończenie współdzielenia zasobu, który doprowadził do powstania wąskiego gardła. To oznacza konieczność znalezienia sposobu na rozproszenie aplikacji między wiele serwerów. Jeżeli jednak chcesz naprawdę przeprowadzić skalowanie w górę, nie możesz na tym poprzestać. Konieczne będzie więc rozproszenie aplikacji między wieloma sieciami, w przeciwnym razie dojdziesz do kresu możliwości pojedynczego przełącznika sieciowego. Ostatecznie trzeba będzie sięgnąć także po rozproszenie geograficzne, aby uniknąć przeciążenia sieci.

Aby zbudować aplikację pozbawioną ograniczeń w zakresie skalowalności, konieczne jest zastosowanie architektury pozwalającej na uruchamianie dodatkowych egzemplarzy aplikacji na żądanie. Skoro aplikacja jest na tyle szybka, na ile pozwala jej najwolniej działający komponent, trzeba znaleźć sposób na skalowanie *wszystkiego*, w tym także magazynów danych. Oczywiste staje się, że jedyny sposób pozwalający zrobić to efektywnie polega na utworzeniu aplikacji z wielu niezależnych elementów, które nie będą powiązane z żadnym konkretnym sprzętem. Innymi słowy — mikrousług natywnej chmury.

Zaleta: niezawodność

Powróćmy do najprostszej możliwej aplikacji. Pomijając ograniczenia związane ze skalowalnością, ta aplikacja ma jeszcze inną wadę: jest uruchomiona w pojedynczym serwerze, więc jeśli ulegnie on awarii, ten sam los czeka również aplikację. W przypadku takiej aplikacji można powiedzieć, że nie zapewnia ona niezawodności. Podobnie jak wcześniej jedynym możliwym rozwiązaniem tego problemu będzie zaprzestanie współdzielenia zasobu, który może ulec awarii. Na szczęście istnieje możliwość rozproszenia aplikacji na wielu serwerach, a tym samym uniknięcie pojedynczego punktu awarii sprzętowej, który mógłby doprowadzić do awarii całej aplikacji. Skoro niezawodność aplikacji odpowiada niezawodności jej najbardziej zawodnego komponentu, konieczne jest znalezienie sposobu na rozproszenie wszystkiego, łącznie z pamięcią masową i siecią. Także w tym przypadku potrzebne są mikrousługi natywnej chmury, które zapewniają elastyczność w zakresie środowiska ich uruchomienia i liczby jednocześnie działających egzemplarzy.

Zaleta: odporność na awarie

Istnieje jeszcze trzecia, subtelniejsza zaleta architektury mikrousług. Tym razem wyobraź sobie aplikację uruchomioną w pojedynczym serwerze, która może być łatwo zainstalowana jako pojedynczy pakiet w dowolnej liczbie serwerów. Każdy egzemplarz może zapewnić obsługę jednego użytkownika. Teoretycznie taka aplikacja będzie zapewniała dobrą skalowalność, biorąc pod uwagę możliwość, że zawsze można zainstalować ją na innym serwerze. Ostatecznie taka aplikacja będzie również niezawodna, ponieważ awaria serwera będzie miała wpływ tylko na jednego użytkownika, podczas gdy pozostali będą obsługiwani w zwykły sposób.

Jednak w tym podejściu brakuje koncepcji odporności na awarie, a dokładnie możliwości eleganckiej reakcji aplikacji na awarię. Prawdziwie odporna na awarie aplikacja jest w stanie poradzić sobie z problemami sprzętowymi bądź programowymi w sposób niezauważalny dla użytkownika końcowego. Wprawdzie oddzielne i niepowiązane ze sobą egzemplarze aplikacji będą działały w przypadku awarii jednego z nich, ale w takim podejściu nie istnieje pełna odporność aplikacji na awarie, przynajmniej nie z perspektywy pechowego użytkownika, który był obsługiwany przez egzemplarz, który uległ awarii.

Z drugiej strony jeśli aplikacja zostanie opracowana w postaci oddzielnych mikrousług, mających możliwość komunikowania się poprzez sieć z innymi mikrousługami działającymi na dowolnym serwerze, wówczas utrata jednego serwera może przekładać się na utratę wielu egzemplarzy mikrousług. Jednak w takim przypadku obsługa użytkowników końcowych może być automatycznie przeniesiona do innych egzemplarzy mikrousług uruchomionych na innych serwerach, tak że ci użytkownicy nawet nie zauważą awarii.

Dlaczego kontenery?

Przedstawione przeze mnie informacje na temat nowoczesnej architektury aplikacji z jej mikrousługami natywnej chmury być może powodują, że ta architektura wydaje się bardzo atrakcyjna. Jednak inżynieria jest pełna kompromisów i doświadczeni inżynierowie mogą podejrzewać istnienie poważnych kompromisów, które oczywiście istnieją.

Niezwykle trudnym zadaniem jest zbudowanie aplikacji na bazie wielu małych elementów. Wprawdzie organizacja zespołów pod kątem mikrousług, aby pozwolić im na niezależną pracę, może być doskonałym podejściem, ale gdy później trzeba wszystko połączyć w działającą aplikację, ogromna liczba elementów oznacza powstawanie kolejnych kwestii do rozwiązania. Jak spakować poszczególne elementy? Jak je dostarczyć do środowiska uruchomieniowego? Jak je skonfigurować? Jak je dostarczać razem z (potencjalnie powodującymi konflikty) zależnościami? Jak je uaktualniać? Jak je monitorować i upewniać się o poprawnym działaniu?

Tego rodzaju problem staje się poważniejszy, gdy trzeba uwzględnić wiele uruchomionych egzemplarzy poszczególnych mikrousług. Teraz należy zapewnić mikrousłudze możliwość znalezienia działającego egzemplarza innej mikrousługi oraz dodać obsługę mechanizmu równoważenia obciążenia dla wszystkich działających egzemplarzy. W przypadku awarii sprzętu bądź oprogramowania konfiguracja mechanizmu równoważenia obciążenia musi zostać zmieniona natychmiast. Konieczne będzie zapewnienie automatycznego rozwiązania awaryjnego i ponawiania próby wykonywania nieudanych zadań, aby ukryć przed użytkownikiem końcowym fakt wystąpienia awarii. Trzeba monitorować nie tylko poszczególne usługi, ale również sposób ich współdziałania ze sobą, gdy wykonują zlecone im zadania. Ostatecznie dla użytkowników nie ma znaczenia to, że 99% mikrousług działa poprawnie, jeśli 1% tych usług ulega awarii i uniemożliwia im korzystanie z aplikacji.

W przypadku budowania aplikacji składających się z wielu mikrousług trzeba rozwiązać naprawdę wiele problemów. Na pewno nie chcesz, aby każdy zespół mikrousługi musiał się zajmować tymi problemami, ponieważ wtedy nigdy nie będzie miał czasu na tworzenie rzeczywistego kodu źródłowego. Konieczne jest istnienie sposobu pozwalającego na zarządzanie opracowaniem, wdrożeniem, konfiguracją i obsługą techniczną wszystkich mikrousług. Zapoznaj się z dwiema kategoriami wymaganych atrybutów: stosowanymi dla pojedynczych mikrousług i stosowanymi do wielu współdziałających ze sobą mikrousług.

Wymagania dotyczące kontenerów

W przypadku pojedynczej mikrousługi mamy do czynienia z następującymi kwestiami:

- **Pakowanie** — połączenie elementów tworzących aplikację w celu jej dostarczenia użytkownikom końcowym. To oznacza konieczność uwzględnienia zależności, aby pakiet stał się przenośny i nie powodował konfliktów między mikrousługami.
- **Wersjonowanie** — unikatowe identyfikowanie wersji. Mikrousługi będą wymagały uaktualniania wraz z upływem czasu, więc trzeba znać wersje działających mikrousług.

- **Izolowanie** — mikrousługi nie powinny wzajemnie zakłócać swojego działania. Dzięki temu zyskujesz elastyczność w zakresie środowiska, w którym mogą być one wdrażane.
- **Szybkie uruchamianie** — nowa mikrousługa powinna być uruchamiana natychmiast. Jest to konieczne dla zapewnienia możliwości skalowania i szybkiego reagowania na awarie.
- **Małe obciążenie** — należy ograniczyć ilość zasobów wymaganych do działania mikrousługi, aby uniknąć ograniczeń dotyczących wielkości mikrousługi.

Kontenery zostały opracowane dokładnie po to, by spełnić te wymagania. Zapewniają izolację, a jednocześnie charakteryzują się małym obciążeniem i szybkim uruchamianiem. Ponadto, jak się przekonasz w rozdziale 5., kontener działa na podstawie obrazu kontenera, co zapewnia sposób na przygotowanie paczki aplikacji razem z jej zależnościami oraz możliwość unikatowego identyfikowania wersji tej paczki.

Wymagania dotyczące koordynacji

Aby wiele mikrousług mogło ze sobą współdziałać, konieczne jest spełnienie kilku wymagań:

- **Klaster.** Zapewnia moc obliczeniową, pamięć operacyjną i pamięć masową dla kontenerów na wielu serwerach.
- **Odkrywanie mikrousług.** Konieczne jest zapewnienie możliwości odkrywania mikrousług przez inne mikrousługi. Dzięki temu mikrousługi mogą działać w dowolnym miejscu klastra i zmieniać położenie w nim.
- **Konfiguracja.** Oddzielenie konfiguracji od środowiska uruchomieniowego, co pozwala na ponowne skonfigurowanie aplikacji bez konieczności ponownego kompilowania i wdrażania mikrousług.
- **Kontrola dostępu.** Zarządzanie autoryzacją w celu tworzenia kontenerów. To pozwala zagwarantować, że będą działały jedynie odpowiednie kontenery.
- **Mechanizm równoważenia obciążenia.** Rozproszenie obciążenia między wiele uruchomionych egzemplarzy. Dzięki temu użytkownik końcowy bądź inne mikrousługi nie muszą śledzić pozostałych mikrousług i samodzielnie się zajmować równoważeniem obciążenia.
- **Monitorowanie.** Identyfikowanie egzemplarzy mikrousług, które uległy awarii. Mechanizm równoważenia obciążenia nie będzie działał dobrze, jeśli ruch sieciowy jest kierowany do niefunkcjonujących egzemplarzy.
- **Odzyskiwanie po awarii.** Automatyczne odzyskiwanie mikrousługi po awarii. Jeżeli tej możliwości nie będzie, to kaskadowy łańcuch awarii może doprowadzić do zamknięcia aplikacji.

Takie wymagania pojawiają się w przypadku uruchamiania kontenerów na wielu serwerach. To jest zupełnie inny problem niż kwestia pakowania i uruchamiania pojedynczego kontenera. Aby spełnić te wymagania, konieczne jest użycie środowiska *koordynacji kontenerów*. Przykładem takiego środowiska jest Kubernetes. Pozwala ono traktować zbiór serwerów

jako pojedynczy zbiór zasobów przeznaczonych do uruchamiania kontenerów oraz dynamicznie alokować kontenery na dostępnych serwerach, a także zapewnia możliwości w zakresie rozproszonej komunikacji i pamięci masowej.

Uruchamianie kontenerów

Mam nadzieję, że nie możesz się doczekać tworzenia aplikacji z wykorzystaniem skonteneryzowanych mikrousług i frameworka Kubernetes. Warto więc zapoznać się z podstawami, aby przekonać się, jak te koncepcje sprawdzają się w praktyce. Dzięki temu zyskasz podstawy stanowiące doskonały punkt wyjścia do dalszego zgłębiania technologii kontenerów, na temat której informacje znajdziesz w pozostałej części książki.

Jak wygląda kontener?

W następnym rozdziale wyjaśnię różnice między platformą kontenerów i środowiskiem uruchomieniowym kontenerów, a także przedstawię przykłady uruchamiania kontenerów w różnych środowiskach uruchomieniowych. Natomiast teraz skoncentruję się na prostym przykładzie uruchomionym za pomocą najpopularniejszej platformy kontenerów, jaką jest *Docker*. Moim celem jest tutaj omówienie podstawowych poleceń Dockera i na ich podstawie wyjaśnienie uniwersalnych koncepcji związanych z kontenerami.

Uruchomienie kontenera

Pierwsze przedstawione polecenie Dockera, `run`, tworzy kontener i wykonuje w nim wskazane polecenie. W poleceniu `run` należy podać nazwę obrazu kontenera przeznaczonego do użycia. Więcej informacji na temat obrazu kontenera znajdziesz w rozdziale 5. W tym miejscu wystarczy wiedzieć, że zapewnia on unikatową nazwę i wersję, dzięki czemu Docker dokładnie wie, co ma zostać uruchomione. Rozpocznę od przedstawienia przykładu dla tego rozdziału.

Uwaga

Repozytorium z materiałami przygotowanymi dla książki znajduje się pod adresem https://github.com/book-of-kubernetes/examples. *We „Wprowadzeniu" zamieściłem więcej informacji na temat przygotowania środowiska pracy.*

Najważniejszą koncepcją jest tutaj to, że kontener wygląda jak zupełnie oddzielny system. Aby to zilustrować, przed uruchomieniem kontenera spójrz na system hosta:

```
root@host01:~# cat /etc/os-release
NAME="Ubuntu"
...
root@host01:~# ps -ef
UID          PID    PPID   C  STIME  TTY            TIME  CMD
root           1       0   0  12:59  ?          00:00:07  /sbin/init
...
```

```
root@host01:~# uname -v
#...-Ubuntu  SMP  ...
root@host01:~# ip addr
1:  lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 ...
     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
     inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
     inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
...
3:  enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel ...
     link/ether 08:00:27:bf:63:1f brd ff:ff:ff:ff:ff:ff
     inet 192.168.61.11/24 brd 192.168.61.255 scope global enp0s8
        valid_lft forever preferred_lft forever
     inet6 fe80::a00:27ff:febf:631f/64 scope link
        valid_lft forever preferred_lft forever
...
```

Pierwsze polecenie wyświetla zawartość pliku o nazwie */etc/os-release*, który zawiera informacje o zainstalowanej dystrybucji Linuksa. W omawianym przykładzie jest to maszyna wirtualna działająca pod kontrolą systemu operacyjnego Ubuntu. Te informacje odpowiadają danym wyjściowym jednego z kolejnych poleceń, potwierdzającego, że mamy do czynienia z jądrem systemu bazującego na Ubuntu. Następnie zostaje wyświetlona lista interfejsów sieciowych i można zobaczyć, że adres IP to `192.168.61.11`.

Zautomatyzowane skrypty przygotowujące środowisko pracy dla tego rozdziału automatycznie zainstalowały oprogramowanie Docker, które tym samym jest gotowe do użycia. Trzeba zacząć od pobrania i uruchomienia kontenera zawierającego system Rocky Linux. W tym celu wystarczy wydać pojedyncze polecenie:

```
root@host01:~# docker run -ti rockylinux:8
Unable to find image 'rockylinux:8' locally
8: Pulling from library/rockylinux
...
Status: Downloaded newer image for rockylinux:8
```

W poleceniu `docker run` została użyta opcja `-ti`, wskazująca Dockerowi, że ma udostępnić terminal interaktywny do wydawania poleceń w kontenerze. Poza tą opcją została jeszcze podana nazwa obrazu kontenera, `rockylinux:8`, określająca jego nazwę `rockylinux` i wersję `8`. Ponieważ nie podano polecenia przeznaczonego do wykonania w kontenerze, dla tego obrazu kontenera domyślnie zostało wykonane polecenie `bash` uruchamiające powłokę Bash.

Dzięki temu otrzymujesz znak zachęty powłoki w kontenerze, co pozwala wydawać w nim polecenia. Wydanie polecenia `exit` spowoduje opuszczenie powłoki kontenera i jego zatrzymanie:

```
[root@18f20e2d7e49 /]# cat /etc/os-release ❶
NAME="Rocky Linux" ❷
...
[root@18f20e2d7e49 /]# yum install -y procps iproute ❸
```

```
...
[root@18f20e2d7e49 /]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 13:30 pts/0    00:00:00 /bin/bash ❹
root          19       1  0 13:46 pts/0    00:00:00 ps -ef
[root@18f20e2d7e49 /]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 ...
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
   valid_lft forever preferred_lft forever
18: eth0@if19: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ... ❺
link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
   valid_lft forever preferred_lft forever
[root@18f20e2d7e49 /]# uname -v
#...-Ubuntu SMP ... ❻
[root@18f20e2d7e49 /]# exit
```

Podczas wydawania poleceń w kontenerze wygląda na to, że są one wykonywane w systemie Rocky Linux. W porównaniu do systemu hosta istnieje kilka różnic:

- Inna nazwa hosta wyświetlona w znaku zachęty powłoki ❶ (w moim przypadku to `18f20e2d7e49`, u Ciebie będzie inna).
- Inna zawartość systemu plików ❷, obejmująca m.in. podstawowe pliki typu */etc/os-release*.
- Użycie polecenia `yum` ❸ podczas instalacji pakietów oraz konieczność zainstalowania pakietów nawet dla najbardziej podstawowych poleceń.
- Ograniczony zbiór uruchomionych procesów, brak podstawowych usług systemowych, a powłoka Bash ❹ jest procesem o identyfikatorze (PID) `1`.
- Inne urządzenia sieciowe ❺, m.in. inne adresy MAC i IP.

Co ciekawe, po wydaniu polecenia `uname -v` wyświetlane są dokładnie te same dane wyjściowe wskazujące na jądro systemu Ubuntu ❻ jak w przypadku systemu hosta. Nie ulega wątpliwości, że kontener nie jest zupełnie oddzielnym systemem, jak można by sądzić.

Obraz kontenera i punkty montowania woluminów

Na pierwszy rzut oka wydaje się, że kontener to rozwiązanie pośrednie między zwykłym procesem i maszyną wirtualną. Sposób pracy z Dockerem jedynie utwierdza w tym przekonaniu. Zilustruję to na przykładzie uruchomienia kontenera systemu Alpine Linux. Trzeba rozpocząć od pobrania (ang. *pull*) obrazu kontenera, co przypomina proces pobierania obrazu maszyny wirtualnej:

```
root@host01:~# docker pull alpine:3
3: Pulling from library/alpine
...
docker.io/library/alpine:3
```

Następnym krokiem jest uruchomienie kontenera na podstawie tego obrazu. W tym przykładzie użyję operacji **montowania woluminu**, aby mieć dostęp do plików znajdujących się w hoście. Jest to często spotykane rozwiązanie w pracy z maszynami wirtualnymi. Docker otrzymuje polecenie zdefiniowania zmiennej środowiskowej, co jest zadaniem wykonywanym podczas pracy ze zwykłymi procesami:

```
root@host01:~# docker run -ti -v /:/host -e hello=world alpine:3
/ # hostname
75b51510ab61
```

Podobnie jak w przypadku kontenera z systemem Rocky Linux, także w kontenerze Alpine Linux zostaje wyświetlona zawartość pliku */etc/os-release*:

```
/ # cat /etc/os-release
NAME="Alpine Linux"
ID=alpine
...
```

Jednak tym razem można również wyświetlić zawartość pliku */etc/os-release* hosta, ponieważ system plików hosta został zamontowany jako */host*:

```
/ # cat /host/etc/os-release
NAME="Ubuntu"
...
```

W kontenerze masz również dostęp do zmiennej środowiskowej, która została do niego przekazana podczas uruchamiania kontenera:

```
/ # echo $hello
world
/ # exit
```

Połączenie koncepcji pochodzących z maszyn wirtualnych i zwykłych procesów czasami powoduje, że początkujący użytkownicy kontenerów zadają sobie pytanie: „Dlaczego nie mogę dostać się do kontenera za pomocą SSH?”. Najważniejszym celem kilku kolejnych rozdziałów jest wyjaśnienie, czym tak naprawdę jest kontener.

Czym tak naprawdę jest kontener?

Pomimo tego, jak kontener wygląda — ma własną nazwę hosta, system plików, przestrzeń procesów i sieć — nie jest maszyną wirtualną. Nie posiada oddzielnego jądra, a tym samym nie może mieć oddzielnych modułów jądra ani sterowników urządzeń. Wprawdzie kontener może mieć wiele procesów, ale jednocześnie muszą być one jawnie uruchomione przez pierwszy proces o identyfikatorze (PID) 1. Dlatego też domyślnie kontener nie posiada serwera SSH, a ponadto większość kontenerów nie ma uruchomionych żadnych usług systemowych.

W kilku następnych rozdziałach wyjaśnię, jak to możliwe, że kontener wygląda na zupełnie oddzielny system, podczas gdy tak naprawdę jest grupą powiązanych ze sobą procesów. W tym momencie wystarczające będzie wypróbowanie kilku kolejnych przykładów Dockera, aby przekonać się, jak kontener Dockera przedstawia się z perspektywy systemu hosta.

Zacznij od wydania pojedynczego polecenia, które spowoduje pobranie i uruchomienie kontenera zawierającego serwer WWW NGINX:

```
root@host01:~# docker run -d -p 8080:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
...
Status: Downloaded newer image for nginx:latest
e9c5e87020372a23ce31ad10bd87011ed29882f65f97f3af8d32438a8340f936
```

Ten przykład pokazuje dodatkowe, użyteczne opcje polecenia `docker`. Podobnie jak wcześniej mamy tutaj połączenie koncepcji związanych z maszynami wirtualnymi i zwykłymi procesami. Użycie opcji `-d` nakazuje Dockerowi uruchomienie tego kontenera w **trybie demona** (w tle), co jest spotykanym rozwiązaniem podczas pracy ze zwykłymi procesami. Natomiast polecenie `-p 8080:80` przywołuje kolejną koncepcję z maszyn wirtualnych, ponieważ nakazuje Dockerowi przekierowanie ruchu sieciowego z portu 8080 hosta do portu 80 Dockera. Dzięki temu można z poziomu hosta nawiązać połączenie z serwerem NGINX w kontenerze, pomimo tego, że kontener ma własne interfejsy sieciowe.

W tym momencie serwer NGINX działa w tle jako kontener Dockera. Aby się o tym przekonać, wydaj następujące polecenie:

```
root@host01:~# docker ps
CONTAINER ID IMAGE ... PORTS                  NAMES
e9c5e8702037 nginx ... 0.0.0.0:8080->80/tcp   funny_montalcini
```

Dzięki mechanizmowi przekazywania portów z serwerem NGINX w kontenerze można się połączyć z poziomu hosta, np. za pomocą polecenia `curl`:

```
root@host01:~# curl http://localhost:8080/
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Za sprawą tego przykładu zaczniesz dostrzegać, że konteneryzacja pozwala spełnić niektóre wymagania przedstawione nieco wcześniej w tym rozdziale. Skoro serwer NGINX został umieszczony w obrazie kontenera, można go pobrać i uruchomić, używając tylko jednego polecenia. Nie musisz się przy tym obawiać, że ten serwer będzie powodował konflikty z innym oprogramowaniem potencjalnie zainstalowanym w hoście.

Wydaj jeszcze jedno polecenie, aby nieco poeksperymentować z serwerem NGINX:

```
root@host01:~# ps -fC nginx
UID          PID    PPID  C STIME TTY          TIME CMD
root       22812   22777  1 15:01 ?        00:00:00 nginx: master ...
```

Jeżeli serwer NGINX zostałby uruchomiony na maszynie wirtualnej, to jego proces nie pojawiłby się w danych wyjściowych polecenia `ps` wydanego w systemie hosta. Nie ma żadnych wątpliwości, że serwer NGINX w kontenerze działa jako zwykły proces. Jednocześnie nie ma konieczności instalowania serwera NGINX w systemie hosta. Innymi słowy otrzymujemy korzyści oferowane przez maszyny wirtualne, ale bez powodowanego przez nie obciążenia.

Wdrażanie kontenerów w Kubernetesie

Aby mieć zapewniony mechanizm równoważenia obciążenia i odporności na awarie w skonteneryzowanych aplikacjach, potrzebny jest framework koordynacji kontenerów taki jak Kubernetes. Nasz przykładowy system również posiada automatycznie zainstalowany klaster, w którym została wdrożona aplikacja internetowa i baza danych. Warto zapoznać się z tą aplikacją jako formą przygotowań przed dokładniejszym poznawaniem frameworka Kubernetes w części II książki.

Istnieje wiele różnych opcji podczas instalowania i konfigurowania klastra Kubernetesa, z wykorzystaniem dystrybucji oferowanych przez wiele firm. W rozdziale 6. omówię różne opcje związane z dystrybucjami Kubernetesa. Natomiast w tym rozdziale wykorzystam lekką dystrybucję o nazwie K3s, opracowaną przez firmę Rancher.

W celu użycia środowiska koordynacji kontenerów takiego jak Kubernetes konieczne jest oddanie pewnej kontroli nad kontenerami. Zamiast bezpośrednio wykonywać polecenia przeznaczone do uruchamiania kontenerów, wskazujesz Kubernetesowi kontenery które mają działać, a framework decyduje, kiedy będą uruchamiane poszczególne kontenery. Następnie Kubernetes zajmuje się monitorowaniem kontenerów oraz obsługą automatycznego ponownego uruchomienia, zapewnieniem poprawnej pracy pomimo awarii, uaktualnieniami do nowszych wersji, a nawet automatycznym skalowaniem na podstawie obciążenia. Taki styl konfiguracji jest określany mianem **deklaratywnego**.

Komunikacja z klastrem Kubernetesa

Klaster Kubernetesa ma serwer API, który można wykorzystać do pobierania informacji o stanie oraz do zmiany konfiguracji klastra. Praca z tym serwerem odbywa się poprzez aplikację klienta w postaci polecenia powłoki o nazwie `kubectl`. Dystrybucja K3s jest dostarczana razem z własną osadzoną wersją `kubectl`, z której będę korzystać w przedstawionych tutaj przykładach. Na początek pokażę, jak można zebrać nieco podstawowych informacji na temat klastra Kubernetesa:

```
root@host01:~# k3s kubectl version
Client Version: version.Info{Major:"1", ...
Server Version: version.Info{Major:"1", ...
root@host01:~# k3s kubectl get nodes
NAME     STATUS   ROLES             AGE   VERSION
host01   Ready    control-plane...  2d    v1...
```

Jak widzisz, w omawianym przykładzie praca odbywa się z klastrem Kubernetesa składającym się z pojedynczego węzła. Oczywiście to nie spełnia potrzeb w zakresie zapewnienia wysokiej dostępności. Większość dystrybucji Kubernetesa, w tym także K3s, obsługuje wiele węzłów i klaster wysokiej dostępności — w części II książki dokładnie wyjaśnię sposób ich działania.

Ogólne omówienie przykładowej aplikacji

Przykładowa aplikacja dostarcza „listę rzeczy do zrobienia" razem z interfejsem przeglądarki WWW, trwałym magazynem danych oraz możliwością śledzenia stanu elementu. Uruchomienie tego rozwiązania w klastrze Kubernetesa może zająć kilka minut, nawet pomimo wykorzystania zautomatyzowanych skryptów przygotowujących niezbędne środowisko. Gdy aplikacja zostanie uruchomiona, dostęp do niej odbywa się za pomocą przeglądarki WWW, w której otrzymasz interfejs pokazany na rysunku 1.1.

Rysunek 1.1. Przykładowa aplikacja działająca w klastrze Kubernetesa

Ta aplikacja została podzielona na dwa typy kontenerów, po jednym dla każdego komponentu aplikacji. Aplikacja Node.js udostępnia pliki przeglądarce WWW i zapewnia obsługę API REST. Ta aplikacja komunikuje się z bazą danych PostgreSQL. Komponent Node.js jest bezstanowy, więc bardzo łatwo można go skalować w górę do dowolnej

liczby niezbędnych egzemplarzy, w zależności od liczby użytkowników. W omawianym przykładzie zasób Deployment w Kubernetesie spowodował uruchomienie trzech kontenerów Node.js:

```
root@host01:~# k3s kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
todo-db-7df8b44d65-744mt    1/1     Running   0          2d
todo-655ff549f8-l4dxt       1/1     Running   0          2d
todo-655ff549f8-gc7b6       1/1     Running   1          2d
todo-655ff549f8-qq8ff       1/1     Running   1          2d
```

Polecenie `get pods` nakazuje Kubernetesowi wyświetlenie listy tzw. **podów**. Jest to grupa składająca się z co najmniej jednego kontenera, która przez Kubernetesa jest traktowana jako pojedyncza jednostka na potrzeby szeregowania i monitorowania. Dokładne omówienie podów znajdziesz w części II książki.

W tym przykładzie istnieje jeden pod o nazwie rozpoczynającej się od `todo-db`, którym jest baza danych PostgreSQL. Trzy kolejne pody o nazwach rozpoczynających się od `todo` to kontenery Node.js. (Później dowiesz się, dlaczego nazwy podów zawierają losowo wybrane znaki. Na razie możesz to zignorować).

Według Kubernetesa kontenery komponentów przykładowej aplikacji są uruchomione, więc dostęp do aplikacji powinien być możliwy za pomocą przeglądarki WWW. Sposób, w jaki będziesz mieć ten dostęp, zależy od używanego środowiska, AWS lub Vagrant. Skrypty przykładowej konfiguracji spowodują wyświetlenie adresu URL, pod który należy przejść w przeglądarce WWW. Po przejściu pod ten adres zobaczysz interfejs pokazany na rysunku 1.1.

Funkcje Kubernetesa

Gdyby jedynym celem było uruchomienie czterech kontenerów, można byłoby to zrobić, używając Dockera i wcześniej przedstawionych poleceń. Kubernetes oferuje znacznie większą funkcjonalność. Warto pokrótce zapoznać się z najważniejszymi funkcjami Kubernetesa.

Pomijając uruchamianie kontenerów, Kubernetes zajmuje się również ich monitorowaniem. Skoro zostały wskazane trzy egzemplarze, Kubernetes będzie pilnować, aby w danej chwili działały trzy egzemplarze. Usuń jeden z nich i zobacz, jak Kubernetes automatycznie przeprowadzi operację uruchomienia nowego egzemplarza, aby zastąpić nim ten, który uległ awarii:

```
root@host01:~# k3s kubectl delete pod todo-655ff549f8-qq8ff
pod "todo-655ff549f8-qq8ff" deleted
root@host01:~# k3s kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
todo-db-7df8b44d65-744mt    1/1     Running   0          2d
todo-655ff549f8-l4dxt       1/1     Running   0          2d
todo-655ff549f8-gc7b6       1/1     Running   1          2d
todo-655ff549f8-rm8sh       1/1     Running   0          11s
```

Aby wykonać to polecenie, musisz skopiować i wkleić pełną nazwę jednego ze swoich podów. Ta nazwa będzie nieco inna niż w przedstawionym tutaj przykładzie. Po usunięciu poda zobaczysz, że Kubernetes natychmiast tworzy nowego. (Nowego poda można zidentyfikować na podstawie wartości kolumny `AGE`).

Następnym krokiem jest potwierdzenie, że Kubernetes potrafi automatycznie skalować aplikację w górę. W dalszej części książki dowiesz się, jak Kubernetes może się tym zajmować automatycznie, natomiast w tym momencie zrobisz to ręcznie. Załóżmy, że potrzebnych jest pięć podów, a nie jedynie trzech. Skalowanie można przeprowadzić poprzez wydanie następującego polecenia:

```
root@host01:~# k3s kubectl scale --replicas=5 deployment todo
deployment.apps/todo scaled
root@host01:~# k3s kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
todo-db-7df8b44d65-744mt    1/1     Running   0          2d
todo-655ff549f8-l4dxt       1/1     Running   0          2d
todo-655ff549f8-gc7b6       1/1     Running   1          2d
todo-655ff549f8-rm8sh       1/1     Running   0          5m13s
todo-655ff549f8-g7lxg       1/1     Running   0          6s
todo-655ff549f8-zsqp6       1/1     Running   0          6s
```

To polecenie nakazało Kubernetesowi skalowanie zasobu *Deployment* odpowiedzialnego za zarządzanie podami. W tym miejscu zasób Deployment możesz potraktować jako „właściciela" podów — monitoruje je i określa ich liczbę. W omawianym przykładzie nastąpiło natychmiastowe utworzenie dwóch dodatkowych podów. W ten sposób odbyło się skalowanie aplikacji w górę.

Zanim zakończę ten rozdział, chciałbym zwrócić uwagę na jeszcze jedną niezwykle ważną funkcjonalność Kubernetesa. Gdy wczytujesz aplikację w przeglądarce WWW, żądanie wykonane przez tę przeglądarkę Kubernetes przekazuje do jednego z dostępnych podów. W przypadku odświeżenia strony w przeglądarce żądanie może być przekierowane do zupełnie innego poda, ponieważ Kubernetes automatycznie zajmuje się obsługą mechanizmu równoważenia obciążenia aplikacji. Aby tak się działo, podczas wdrażania aplikacji w Kubernetesie jej plik konfiguracyjny zawiera **usługę** (ang. *service*).

```
root@host01:~# k3s kubectl describe service todo
Name:       todo
...
IPs:        10.43.231.177
Port:       <unset>  80/TCP
TargetPort: 5000/TCP
Endpoints:  10.42.0.10:5000,10.42.0.11:5000,10.42.0.14:5000 + 2 more...
...
```

Usługa ma własny adres IP i przekierowuje ruch sieciowy do jednego bądź więcej punktów końcowych. W omawianym przykładzie, skoro rozwiązanie zostało wyskalowane w górę do pięciu podów, usługa zajmuje się równoważeniem ruchu sieciowego między wszystkimi pięcioma punktami końcowymi.

Podsumowanie

Nowoczesna architektura aplikacji zapewnia skalowalność i niezawodność, ponieważ bazuje na mikrousługach, które mogą być wdrażane niezależnie i dynamicznie w dostępnych zasobach sprzętowych, m.in. w zasobach chmury. Dzięki wykorzystaniu kontenerów i mechanizmów koordynacji kontenerów do uruchamiania mikrousług udało się osiągnąć powszechnie stosowane podejście w zakresie pakowania, skalowania, monitorowania i obsługi technicznej mikrousług. To sprawia, że zespoły programistyczne mogą skoncentrować się na ciężkiej pracy związanej z faktycznym budowaniem aplikacji.

W tym rozdziale wyjaśniłem, jak konteneryzacja może sprawiać wrażenie pracy z oddzielnym systemem, podczas gdy tak naprawdę masz do czynienia ze zwykłym procesem działającym w odizolowany sposób. Pokazałem, jak można użyć Kubernetesa do wdrożenia całej aplikacji jako zbioru kontenerów, który ma możliwość skalowania i automatycznego odzyskiwania po awarii. Oczywiście Kubernetes oferuje znacznie więcej ważnych funkcjonalności niż tutaj przedstawiłem, a ich omówienie zajmie pozostałą część książki. Mam nadzieję, że tym krótkim wprowadzeniem zaciekawiłem Cię i wywołałem chęć dalszego poznawania technologii kontenerów i Kubernetesa, aby tworzyć aplikacje zapewniające świetną wydajność działania i wysoką niezawodność.

Do Kubernetesa powrócę w części II książki. Natomiast w tej skoncentruję się na tym, jak kontenery tworzą iluzję użycia oddzielnego systemu. Rozpocznę od przedstawienia zagadnień związanych z izolacją procesów za pomocą funkcji przestrzeni nazw w Linuksie.


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

CZY NA PEWNO ROZUMIESZ DZIAŁANIE KONTENERA KUBERNETES?

Kontenery i framework Kubernetes gwarantują niezawodne działanie oprogramowania i przejrzystą kontrolę jego funkcjonowania. Największą zaletą platformy Kubernetes jest ukrywanie za warstwą abstrakcji mechanizmu uruchamiania kontenerów w różnych klastrach. Ułatwia to wdrażanie aplikacji i zarządzanie nimi, ale jednocześnie utrudnia zrozumienie, co naprawdę dzieje się w klastrze.

W tej książce omówiono wewnętrzny sposób działania frameworka Kubernetes i pokazano, jak za jego pomocą budować wydajne, niezawodne i odporne na awarie aplikacje natywnej chmury. Dowiesz się, jak kontenery używają przestrzeni nazw w celu izolowania procesów, a także jak korzystają z funkcjonalności ograniczania zasobów, aby zagwarantować, że proces będzie się opierał jedynie na tych, które zostały mu przydzielone. Nauczysz się instalować klaster Kubernetes, wdrażać kontenery i zrozumiesz, na czym polega przepływ pakietów między kontenerami w sieci hosta. Ponadto poznasz strategie tworzenia i uruchamiania kontenerów, które zapewnią oprogramowaniu optymalną wydajność, jak również sposoby identyfikowania i usuwania potencjalnych problemów.

W książce między innymi:

- zapewnienie większej wydajności działania aplikacji
- automatyczne skalowanie i mechanizm równoważenia obciążenia
- konfigurowanie mechanizmu kontroli dostępu
- wykrywanie problemów i ich usuwanie
- wdrażanie nowych kontenerów i konfigurowanie routingu w sieci
- rozszerzanie klastra Kubernetes przez dodawanie nowych funkcjonalności

Alan Hohn od ponad 25 lat zajmuje się tworzeniem oprogramowania. Jest inżynierem oprogramowania, architektem i menedżerem, a także trenerem DevSecOps, architektury oprogramowania i Kubernetes. Pracował w Google, gdzie zajmował się rozwojem frameworka Kubernetes, potem współzałożył firmę Heptio działającą w zakresie narzędzi i usług dla tej platformy.

Helion

helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej! ▶

ISBN 978-83-289-1012-6

9 788328 910126

Cena: 89,00 zł

no starch press