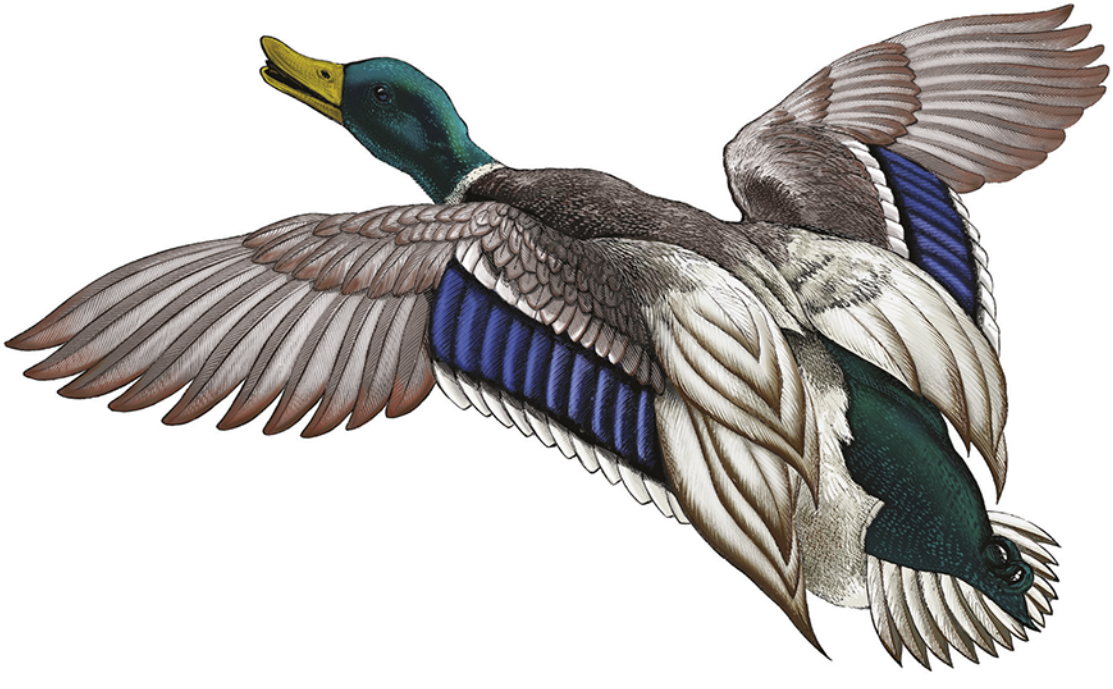
O'REILLY®

Najlepsze praktyki w Kubernetes

Jak budować udane aplikacje



Brendan Burns, Eddie Villalba,
Dave Strebel, Lachlan Evenson

Helion

Tytuł oryginału: Kubernetes Best Practices: Blueprints for Building Successful Applications on Kubernetes

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-7232-0

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

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 5.

Ciągła integracja, testowanie i ciągłe wdrażanie

W tym rozdziale zostaną przedstawione kluczowe koncepcje dotyczące integracji z mechanizmem ciągłej integracji (ang. *continuous integration*, CI) i ciągłego wdrażania (ang. *continuous deployment*, CD) podczas dostarczania aplikacji do Kubernetes. Przygotowanie doskonałego sposobu pracy pozwoli na sprawne dostarczanie aplikacji do środowiska produkcyjnego. Dlatego też w rozdziale będą zaprezentowane metody, narzędzia i procesy umożliwiające stosowanie technik CI i CD we własnym środowisku pracy. Celem technik CI i CD jest opracowanie w pełni zautomatyzowanego procesu, od programisty umieszczającego kod w repozytorium po przekazanie nowej aplikacji do środowiska produkcyjnego. Najlepiej będzie unikać ręcznego wdrażania w Kubernetes uaktualnionych aplikacji, ponieważ ten proces jest podatny na błędy. Ponadto ręczne zarządzanie uaktualnieniami aplikacji w Kubernetes prowadzi do zmian w konfiguracji i niepewnych uaktualnień oraz ogólnie do utraty zwinności podczas procesu dostarczania aplikacji.

W rozdziale zostaną omówione następujące zagadnienia:

- system kontroli wersji,
- ciągła integracja,
- testowanie,
- oznaczanie tagami obrazów kontenera,
- ciągłe wdrażanie,
- strategie wdrażania,
- testowanie wdrożeń,
- testowanie w chaosie.

Zaprezentujemy również przykładowe techniki CI i CD, na które składają się wymienione tutaj zadania:

- przekazywanie do repozytorium Git zmian wprowadzonych w kodzie źródłowym,
- kompilowanie kodu aplikacji,
- testowanie kodu,

- utworzenie obrazu kontenera po zakończeniu testów powodzeniem,
- przekazywanie obrazu kontenera do rejestru kontenerów,
- wdrażanie aplikacji w Kubernetes,
- przeprowadzanie testów wdrożonej aplikacji,
- nieustanne uaktualnianie wdrożenia.

System kontroli wersji

Każde rozwiązanie oparte na technikach CI i CD rozpoczyna się od systemu kontroli wersji, którego zadaniem jest obsługa historii zmian kodu aplikacji i jej konfiguracji. Git stał się standardem przemysłowym w dziedzinie platform zarządzania kodem źródłowym, a każde repozytorium Git ma tzw. *gałąź master* (ang. *master branch*), która jest gałęzią główną repozytorium i zawiera kod przeznaczony do wdrożenia w środowisku produkcyjnym. W repozytorium zwykle znajdują się także inne gałęzie przeznaczone do opracowywania kolejnych funkcjonalności aplikacji, a wprowadzone w nich zmiany ostatecznie i tak trafiają do gałęzi master. Istnieje wiele strategii związanych z tworzeniem gałęzi, a konkretna konfiguracja będzie zależała od struktury organizacji i separacji zadań. Według nas kod aplikacji i kod konfiguracji, czyli np. manifest Kubernetes lub plik menedżera Helm w formacie chart, pomagają w promowaniu dobrych praktyk DevOps w zakresie komunikacji i współpracy. Gdy programiści aplikacji i inżynierowie operacji współpracują nad kodem znajdującym się w jednym repozytorium, to daje przekonanie, że zespół będzie w stanie dostarczyć aplikację do środowiska produkcyjnego.

Ciągła integracja

Ciągła integracja to proces nieustannego integrowania zmian w kodzie z repozytorium systemu kontroli wersji. Zamiast rzadko przekazywać większe zmiany, znacznie częściej przekazuje się mniejsze. Każde przekazanie zmian do repozytorium powoduje rozpoczęcie kompilacji kodu źródłowego. Dzięki temu można o wiele szybciej otrzymać informacje o tym, co zostało zepsute w aplikacji, gdy wystąpi w niej problem. W tym miejscu prawdopodobnie zadajesz sobie pytanie w rodzaju: „Dlaczego miałbym poznawać szczegóły związane z kompilacją aplikacji, skoro to jest zadanie programisty?”. Tradycyjnie tak było, choć w ostatnim czasie można zaobserwować w firmach przesunięcie w stronę podejścia kultury DevOps, w którym zespół operacji jest bliżej kodu aplikacji i procesów związanych z jej tworzeniem.

Istnieje wiele rozwiązań w dziedzinie ciągłej integracji. Jednym z najpopularniejszych narzędzi tego typu jest Jenkins.

Testowanie

Celem testów jest szybkie dostarczenie informacji o zmianach w kodzie, które doprowadziły do uszkodzenia aplikacji. Używany język programowania będzie miał wpływ na framework testów, który wykorzystasz do ich przygotowania. Przykładowo aplikacje w języku Go używają `go test`

do uruchomienia zestawu testów jednostkowych dla bazy kodu. Opracowanie rozbudowanego zestawu testów pomaga unikać sytuacji, gdy do środowiska produkcyjnego zostaje przekazany niepoprawnie działający kod. Chcesz mieć pewność, że jeśli test zostanie niezaliczony w środowisku programistycznym, natychmiast po jego zakończeniu kompilacja zakończy się niepowodzeniem. Nie chcesz utworzyć obrazu kontenera i przekazać go do repozytorium, gdy jakikolwiek test bazy kodu kończy się niepowodzeniem.

Także w tym przypadku być może zadajesz sobie pytanie w rodzaju: „Czy tworzenie testów nie powinno być zadaniem programisty aplikacji?". Gdy zaczniesz stosować zautomatyzowaną infrastrukturę dostarczania aplikacji do środowiska produkcyjnego, musisz pomyśleć o przeprowadzaniu zautomatyzowanych testów całej bazy kodu. Przykładowo z rozdziału 2. dowiedziałeś się nieco o użyciu menedżera pakietów Helm w celu przygotowania aplikacji do umieszczenia w Kubernetes. Ten menedżer zawiera narzędzie o nazwie `helm lint`, którego działanie polega na wykonaniu serii testów względem pliku w formacie chart i przeanalizowaniu kodu pod kątem potencjalnych problemów. Istnieje wiele różnych testów do wykonania podczas przygotowywania aplikacji. Część z nich powinna być wykonywana przez programistów, inne zaś to wysiłek podejmowany wspólnie przez wszystkich. Testowanie bazy kodu i dostarczanie na jej podstawie gotowej aplikacji do środowiska produkcyjnego jest wysiłkiem całego zespołu i ta operacja powinna być zaimplementowana od początku do końca.

Kompilacja kontenera

Podczas tworzenia obrazów należy optymalizować ich wielkość. Mniejszy obraz oznacza skrócenie czasu potrzebnego na pobranie i wdrożenie obrazu, a ponadto zwiększa poziom jego bezpieczeństwa. Istnieje wiele sposobów na optymalizację wielkości obrazu, z których część wiąże się z pewnymi kompromisami. Zapoznaj się ze strategiami, które pomagają w tworzeniu możliwie małych obrazów zawierających budowane aplikacje.

Kompilacja wieloetapowa
: To pozwala na usunięcie zależności niepotrzebnych do działania aplikacji. Przykładowo w przypadku języka programowania Go nie potrzebujemy wszystkich narzędzi kompilacji używanych do utworzenia statycznych plików binarnych. Dlatego też kompilacja wieloetapowa pozwala na użycie jednego pliku *Dockerfile* do przeprowadzenia kompilacji, a ostateczny obraz będzie zawierał tylko statyczne pliki binarne wymagane do uruchomienia aplikacji.

Obraz bazowy nieoparty na żadnej dystrybucji
: To pozwala na usunięcie z obrazu wszystkich niepotrzebnych plików binarnych i powłok. Skutkiem jest znaczne zmniejszenie wielkości obrazu i zwiększony poziom bezpieczeństwa. Natomiast wadą obrazu nieopartego na żadnej dystrybucji jest to, że jeśli nie masz powłoki, wówczas nie będziesz mógł dołączyć debugera do obrazu. Być może uważasz, że to świetne rozwiązanie, ale w rzeczywistości znacznie utrudni debugowanie aplikacji. Takie obrazy nie zawierają żadnego menedżera pakietów, powłoki ani innych typowych pakietów systemu operacyjnego, więc możesz nie uzyskać dostępu do narzędzi debugowania znanych Ci z typowego systemu operacyjnego.

Zoptymalizowane obrazy bazowe
: W przypadku tych obrazów skoncentrowano się na usunięciu wszelkich elementów warstwy systemu operacyjnego i dostarczeniu minimalnej wersji obrazu. Przykładowo Alpine oferuje obraz bazowy o wielkości około 10 MB, a także pozwala na dołączenie debugera lokalnego podczas opracowywania aplikacji w lokalnym środowisku programistycznym. Inne dystrybucje również oferują zoptymalizowane obrazy bazowe; przykładem może być tutaj obraz Slim dystrybucji Debian. To może być doskonałe rozwiązanie, ponieważ zoptymalizowane obrazy zapewniają możliwości oczekiwane w środowisku programistycznym, a zarazem są zoptymalizowane pod względem wielkości obrazu i podatności na ataki.

Optymalizacja obrazów ma wyjątkowo duże znaczenie, choć często jest niedoceniana przez użytkowników. Mogą być ku temu pewne powody, np. wynikające z przyjętych w firmie standardów dotyczących dozwolonych do użycia systemów operacyjnych, ale warto je odłożyć na bok, aby maksymalizować wartość kontenerów.

Zauważyliśmy, że firmy, które zaczęły używać Kubernetes, zwykle są zadowolone ze stosowanego systemu operacyjnego, a mimo to decydują się na wybór znacznie bardziej zoptymalizowanego obrazu, takiego jak Debian Slim. Gdy zdobędziesz większe doświadczenie w tworzeniu aplikacji dla środowiska kontenerów, poczujesz się pewniej podczas pracy z obrazami nieopartymi na żadnych dystrybucjach.

Oznaczanie tagiem obrazu kontenera

Kolejnym krokiem w procesie ciągłej integracji jest utworzenie obrazu Dockera, aby mógł zostać wdrożony do wybranego środowiska. Bardzo duże znaczenie ma stosowanie strategii nadawania tagów obrazom, co pozwoli na łatwe identyfikowanie wersjonowanych obrazów wdrożonych w środowiskach. Jedną z najważniejszych kwestii jest zaprzestanie używania słowa *latest* jako tagu obrazu. Używanie tagu obrazu nieprzedstawiającego *wersji* oznacza brak możliwości ustalenia zmian, po których wprowadzeniu w kodzie nastąpiło wygenerowanie takiego obrazu. Każdy obraz tworzony w procesie CI powinien mieć unikatowy tag.

Istnieje wiele użytecznych strategii podczas oznaczania obrazów tagami w procesie ciągłej integracji. Wymienione tutaj strategie pozwalają bardzo łatwo identyfikować zmiany w kodzie i konkretnej kompilacji, z którą zmiany te są powiązane.

Identyfikator kompilacji
: Po rozpoczęciu kompilacji zostaje z nią powiązany pewien identyfikator. Użycie tej części tagu pozwala odwołać się do konkretnej kompilacji powiązanej z obrazem.

Identyfikator systemu kompilacji — identyfikator kompilacji
: To jest taki sam identyfikator jak poprzedni, ale zawiera także identyfikator systemu kompilacji, co okazuje się przydatne dla użytkowników, którzy mają wiele systemów kompilacji.

Wartość hash z repozytorium Git
: W przypadku nowej operacji przekazania kodu do repozytorium następuje wygenerowanie wartości hash w repozytorium Git. Następnie ta wartość hash jest używana jako tag, pozwalający na łatwe odwołanie się do operacji, która spowodowała zainicjowanie generowania obrazu.

Wartość hash dla identyfikatora kompilacji
: Ta wartość pozwala odwoływać się do operacji przekazania kodu do repozytorium i identyfikatora kompilacji, w której wyniku powstał obraz. Trzeba w tym miejscu dodać, że ten znacznik może być dość długi.

Ciągłe wdrażanie

Ciągłe wdrażanie to proces, w którym zmiany pasywnie przekazywane z sukcesem do systemu ciągłej integracji zostają wdrożone do środowiska produkcyjnego, bez konieczności udziału człowieka. Kontenery mają ogromną zaletę w zakresie wdrażania zmian w środowisku produkcyjnym. Obraz kontenera staje się obiektem niemodyfikowalnym, który poprzez środowiska programistyczne i robocze można promować do środowiska produkcyjnego. Przykładowo jednym z poważnych problemów, z którymi zawsze się stykamy, jest zapewnienie spójnych środowisk. Niemal każdy napotkał sytuację, w której zasób `Deployment` działał w środowisku roboczym, a przestał działać po jego przekazaniu do środowiska produkcyjnego. Tak się dzieje na skutek tzw. *przesunięcia w konfiguracji*, gdy biblioteki i wersjonowane komponenty różnią się w poszczególnych środowiskach. Kubernetes zapewnia deklaracyjny sposób opisywania obiektów `Deployments`, które mogą być wersjonowane i wdrażane w spójny sposób.

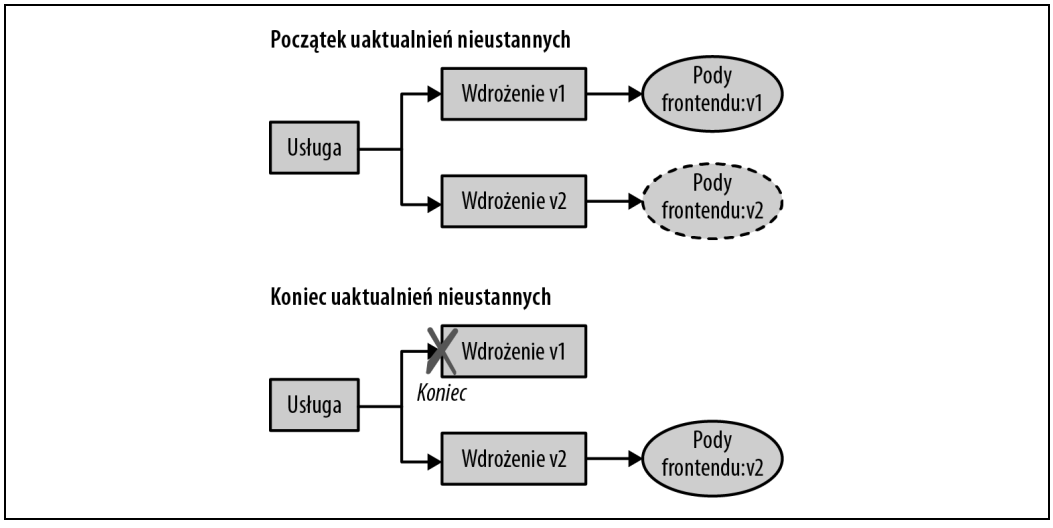
Trzeba pamiętać o tym, by najpierw zadbać o zachowanie spójnej konfiguracji procesu ciągłej integracji, a dopiero później zająć się nieustannym wdrażaniem. Jeżeli nie masz przygotowanego niezawodnego zestawu testów wychwytującego błędy na wstępnym etapie procesu, skutkiem może być przekazanie niepoprawnego kodu do wszystkich środowisk.

Strategie wdrażania

Skoro poznałeś podstawowe reguły nieustannego wdrażania, warto się zapoznać z różnymi strategiami, które są możliwe do zastosowania. Kubernetes oferuje wiele strategii przeznaczonych do wydawania nowych wersji aplikacji. Nawet jeśli masz wbudowany mechanizm przeznaczony do dostarczania uaktualnień, zawsze możesz skorzystać z nieco bardziej zaawansowanych strategii. W tym podrozdziale będą przeanalizowane następujące strategie związane z dostarczaniem uaktualnień aplikacji:

- dostarczanie uaktualnień,
- wdrożenie typu niebieski-zielony,
- wdrożenie kanarkowe.

Mechanizm *dostarczania uaktualnień* jest wbudowany w Kubernetes i pozwala na przeprowadzenie aktualizacji uruchomionej aplikacji bez jej zatrzymywania i przestoju. Przykładowo, jeśli masz uruchomioną aplikację frontendu w wersji 1 i uaktualnisz ją do wersji 2, wówczas Kubernetes przeprowadzi aktualizację tej aplikacji w replikach, jak pokazaliśmy na rysunku 5.1.



Rysunek 5.1. Uaktualnienia nieustanne w Kubernetes

Obiekt `Deployment` pozwala na skonfigurowanie maksymalnej liczby uaktualnianych replik i maksymalnej liczby podów niedostępnych podczas aktualizacji. Spójrz na przedstawiony tutaj manifest, który pokazuje, jak można zdefiniować strategię uaktualnień nieustannych.

```
kind: Deployment
apiVersion: v1
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: frontend
        image: brendanburns/frontend:v1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1 # Maksymalna liczba jednocześnie uaktualnianych replik.
      maxUnavailable: 1 # Maksymalna liczba replik niedostępnych podczas uaktualnienia.
```

Należy zachować ostrożność podczas uaktualnień nieustannych, ponieważ ta strategia może spowodować odrzucanie połączeń. Aby sobie z tym poradzić, można wykorzystać tzw. *próbkowanie odczytu* (ang. *readiness probe*) i zaczepy cyklu życiowego `preStop`. Próbkowanie odczytu ma na celu sprawdzenie, czy nowo wdrożona wersja aplikacji jest gotowa do przyjmowania ruchu sieciowego. Zaczep `preStop` może zaś zagwarantować, że połączenia zostały zamknięte w nowo wdrożonej aplikacji. Ten zaczep cyklu życiowego jest wywoływany przed zakończeniem działania kontenera i jest asynchroniczny, więc musi się zakończyć przed wysłaniem ostatecznego sygnału zakończenia pracy kontenera. Spójrz na przykład przedstawiający zastosowanie próbkowania odczytu i zaczepu cyklu życiowego.

```
kind: Deployment
apiVersion: v1
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: frontend
        image: brendanburns/frontend:v1
        livenessProbe:
          # ...
        readinessProbe:
          httpGet:
            path: /readiness # Punkt końcowy próbkowania.
            port: 8888
        lifecycle:
          preStop:
            exec:
              command: ["/usr/sbin/nginx","-s","quit"]
strategy:
  # ...
```

W omawianym przykładzie zaczep `preStop` cyklu życiowego spowoduje eleganckie zakończenie procesu NGINX, podczas gdy sygnał `SIGTERM` spowodowałby zakończenie szybkie i nieeleganckie.

Inną kwestią związaną z uaktualnieniami nieustannymi jest posiadanie dwóch wersji aplikacji działających jednocześnie podczas aktualizacji. Schemat bazy danych musi obsługiwać obie wersje aplikacji. Można również użyć strategii opcji właściwości, w której to schemat wskazuje nowe kolumny, utworzone przez nową wersję aplikacji. Po przeprowadzeniu uaktualnienia nieustannego stare kolumny mogą zostać usunięte.

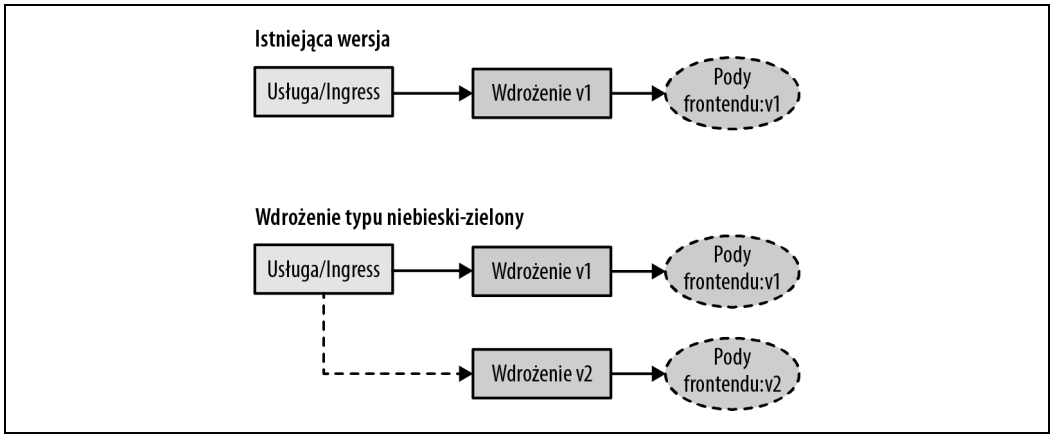
W manifeście wdrożenia zostało zdefiniowane próbkowanie odczytu i dostępności. Próbkowanie odczytu ma zagwarantować, że aplikacja jest gotowa do obsługi ruchu sieciowego, zanim zacznie działać w charakterze usługi dla punktu końcowego. Z kolei próbkowanie dostępności ma zagwarantować, że aplikacja działa poprawnie i że pod zostanie ponownie uruchomiony, jeśli próbkowanie zakończy się niepowodzeniem. Kubernetes potrafi automatycznie ponownie uruchomić niedziałającego poda tylko wtedy, gdy zostanie on zamknięty na skutek błędu. Przykładowo próbkowanie dostępności może sprawdzać punkt końcowy i ponownie go uruchomić po wykryciu zakleszczenia, z którego pod nie zdołał się wydostać.

Wdrożenie typu niebieski-zielony pozwala na wydawanie aplikacji w przewidywalny sposób. Dzięki wdrożeniu tego typu zachowujesz kontrolę nad przeniesieniem ruchu sieciowego do nowego środowiska, co oznacza większą kontrolę nad wydawaniem nowych wersji aplikacji. W przypadku wdrożenia typu niebieski-zielony musisz mieć do dyspozycji wystarczająco dużą pojemność, aby mogły jednocześnie być wdrożone środowiska istniejące i nowe. Taki rodzaj wdrożenia ma wiele zalet, takich jak łatwość cofnięcia do poprzedniej wersji aplikacji.

Stosując tę strategię wdrożenia, trzeba uwzględnić pewne kwestie:

- Migracja bazy danych może być trudna, ponieważ trzeba wziąć pod uwagę realizowane transakcje i zgodność uaktualnienia schematu.
- Istnieje niebezpieczeństwo usunięcia obu środowisk.
- Trzeba zapewnić pojemność wystarczającą dla obu środowisk.
- Możliwe są problemy związane z koordynacją wdrożeń hybrydowych, po których starsze aplikacje nie będą w stanie obsłużyć danego wdrożenia.

Wizualne przedstawienie wdrożenia typu niebieski-zielony pokazaliśmy na rysunku 5.2.



Rysunek 5.2. Wdrożenie typu niebieski-zielony

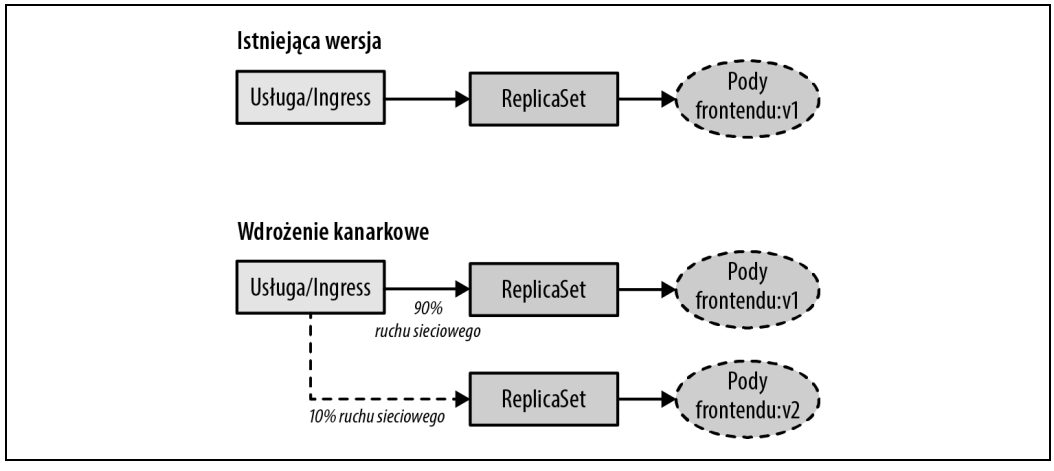
Wdrożenie kanarkowe jest bardzo podobne do wdrożenia typu niebieski-zielony, choć zapewnia większą kontrolę nad przesunięciem ruchu sieciowego do nowego wydania. Większość nowych implementacji usługi umożliwia przekierowanie pewnej, wyrażonej w procentach, ilości ruchu sieciowego do nowego wydania. Ponadto istnieje możliwość implementacji technologii Service Mesh — np. Istio, Linkerd lub HashiCorp Consul — która udostępnia pewną liczbę funkcjonalności pomagających w przygotowaniu takiej strategii wdrożenia.

Wdrożenie kanarkowe pozwala przetestować nowe funkcjonalności tylko na podzbiorze użytkowników. Przykładowo można wydać nową wersję aplikacji i przetestować ją jedynie dla 10% bazy użytkowników. Dzięki temu na niebezpieczeństwa związane z wprowadzeniem niepoprawnego wdrożenia lub niedziałających funkcjonalności będzie narażona znacznie mniejsza grupa użytkowników. Jeżeli we wdrożeniu lub w nowych funkcjonalnościach nie ma błędów, można zacząć przekierowywać do nowej wersji aplikacji coraz większy odsetek użytkowników. Istnieją również o wiele bardziej zaawansowane technologie przeznaczone do stosowania wraz z wdrożeniami kanarkowymi. Przykładowo aplikacja może zostać wydana dla użytkowników pochodzących z określonego regionu lub jedynie dla użytkowników o konkretnym profilu. Takie rodzaje wydań zwykle są określane jako A/B lub ciemne, ponieważ użytkownicy są nieświadomi tego, że testują nową funkcjonalność wdrożenia.

W przypadku wdrożenia kanarkowego trzeba wziąć pod uwagę pewne kwestie pojawiające się we wcześniej omówionym wdrożeniu typu niebieski-zielony, a także kilka nowych:

- Możliwość przesunięcia ruchu sieciowego do wyrażonej w procentach grupy użytkowników.
- Solidna wiedza pozwalająca na porównanie istniejącej wersji aplikacji z nową.
- Wskaźniki pozwalające na określenie, czy stan nowego wydania jest „dobry" czy też „zły".

Wizualne przedstawienie wdrożenia kanarkowego pokazaliśmy na rysunku 5.3.



Rysunek 5.3. Wdrożenie kanarkowe

Wdrożenie kanarkowe jest utrudnione w przypadku wielu uruchomionych jednocześnie wersji aplikacji. Schemat bazy danych musi mieć możliwość obsługi obu wersji aplikacji. Gdy stosujesz tę strategię, naprawdę musisz skoncentrować się na sposobie obsługi usług zależnych i na jednoczesnym działaniu wielu wersji. Dlatego trzeba mieć silne API i zagwarantować, że usługi danych obsługujące wiele wersji aplikacji zostaną wdrożone w tym samym czasie.

Testowanie w produkcji

Testowanie w produkcji pomaga się upewnić, że aplikacja jest niezawodna, skalowana i charakteryzuje się dobrym UX. Trzeba w tym miejscu dodać, że *testowanie w produkcji* wiąże się z pewnymi wyzwaniami i ryzykiem, choć warto ponieść ten wysiłek, aby zagwarantować niezawodność systemów. Istnieją pewne ważne aspekty, które trzeba wziąć pod uwagę podczas przygotowywania takiej implementacji. Przede wszystkim należy się upewnić, że istnieje strategia pozwalająca na dogłębną obserwację, co pozwoli sprawdzić efekty testowania w produkcji. Bez możliwości obserwacji wskaźników wpływających na wrażenia użytkowników końcowych aplikacji nie będziesz miał jasno określonego celu, na którym trzeba się skoncentrować podczas próby poprawienia odporności programu na awarie. Dobrze jest zastosować również wysoki stopień automatyzacji i umożliwić automatyczną naprawę po awarii w systemach.

Do dyspozycji masz wiele narzędzi, które trzeba będzie zaimplementować w celu zmniejszenia niebezpieczeństwa i efektywnego przetestowania systemów w produkcji. O części narzędzi już wspomnieliśmy w rozdziale; są też inne, m.in. służące do monitorowania rozproszonego, instrumentacji, inżynierii chaosu (ang. *chaos engineering*) i przesłaniania ruchu sieciowego (ang. *traffic shadowing*). Dla przypomnienia przedstawiamy listę narzędzi, które zostały już wspomniane w rozdziale:

- wdrożenie kanarkowe,
- testowanie A/B,
- przesunięcie ruchu sieciowego,
- opcje właściwości.

Inżynieria chaosu została opracowana przez firmę Netflix. Polega na wdrażaniu eksperymentów w działających systemach produkcyjnych i ma na celu odkrycie ich słabych stron. Inżynieria chaosu pozwala poznać sposób, w jaki system się zachowuje, przez jego obserwację podczas kontrolowanego eksperymentu. Zapoznaj się z wymienionymi tutaj krokami, które trzeba wykonać przed przystąpieniem do eksperymentów.

1. Opracowanie hipotezy i poznanie aktualnego stanu systemu.
2. Przygotowanie rzeczywistych zdarzeń, które mogą wpływać na system.
3. Utworzenie grupy kontrolnej i eksperymentowanie w celu porównania stanu.
4. Przeprowadzenie eksperymentów w celu sformułowania hipotezy.

Ogromne znaczenie ma to, aby podczas przeprowadzania eksperymentów zminimalizować „pole rażenia" i zagwarantować, że ewentualne problemy będą naprawdę minimalne. Chcesz mieć pewność, że gdy zaczniesz przeprowadzać eksperymenty, skoncentrujesz się na ich automatyzacji, ponieważ ich wykonywanie może być pracochłonne.

Jednak w tym miejscu być może zaczniesz zadawać sobie pytanie: „Dlaczego nie mogę po prostu wykonać testu w środowisku roboczym?". Przekonaliśmy się, że testowanie w środowisku roboczym wiąże się z pewnymi nieuchronnymi problemami. Są to:

- nieidentyczne zasoby wdrożenia,
- przesunięcie konfiguracji względem tej stosowanej w produkcji,
- nienaturalna symulacja ruchu sieciowego i sposobu zachowania użytkownika,
- liczba generowanych żądań nie odzwierciedla rzeczywistego obciążenia,
- brak monitorowania zaimplementowanego w środowisku roboczym,
- wdrożone usługi danych zawierają inne dane i wiążą się z innym obciążeniem niż w środowisku produkcyjnym.

Nie sposób podkreślić tego wystarczająco mocno, ale upewnij się o zastosowaniu w środowisku produkcyjnym solidnego rozwiązania w zakresie monitorowania, ponieważ użytkownicy, którzy nie mają odpowiednich możliwości obserwowania systemów produkcyjnych, są skazani na niepowodzenie. Ponadto rozpocznij od niewielkich eksperymentów, by zwiększyć zaufanie do środowiska produkcyjnego.

Stosowanie inżynierii chaosu i przygotowania

Pierwszym krokiem w omawianym procesie jest utworzenie rozwidlenia repozytorium GitHub. Dzięki temu będziesz mieć własne repozytorium przeznaczone do użycia w rozdziale. Konieczne będzie użycie interfejsu GitHub pozwalającego na rozwidlenie repozytorium (*https://github.com/dstrebel/kbp*).

Konfiguracja ciągłej integracji

Skoro poznałeś technikę ciągłej integracji, zajmiesz się kompilacją kodu, który sklonowaliśmy poprzednio.

Na potrzeby omawianego przykładu wykorzystamy serwis *https://drone.io/*. Będziesz musiał się zarejestrować (*https://cloud.drone.io/*) i utworzyć bezpłatne konto. Podczas logowania podaj dane uwierzytelniające z serwisu GitHub; w ten sposób zarejestrujesz swoje repozytoria w Drone i będziesz mógł je synchronizować. Po zalogowaniu się do Drone wybierz opcję *Active* dla rozwidlenia repozytorium. Pierwszym zadaniem, które prawdopodobnie będziesz musiał wykonać, jest dodanie do konfiguracji pewnych danych poufnych. To pozwoli na przekazanie aplikacji do rejestru Docker Hub i wdrożenie jej do klastra Kubernetes.

W ramach repozytorium w Drone kliknij *Settings* i dodaj następujące dane poufne (zobacz rysunek 5.4).

- `docker_username`,
- `docker_password`,
- `kubernetes_server`,
- `kubernetes_cert`,
- `kubernetes_token`.

Nazwa użytkownika i hasło w serwisie Docker będą tymi wartościami, których użyłeś podczas rejestrowania konta w Docker Hub. W kolejnych krokach zobaczysz, jak przebiega utworzenie konta usługi Kubernetes, certyfikacja i pobranie tokena.

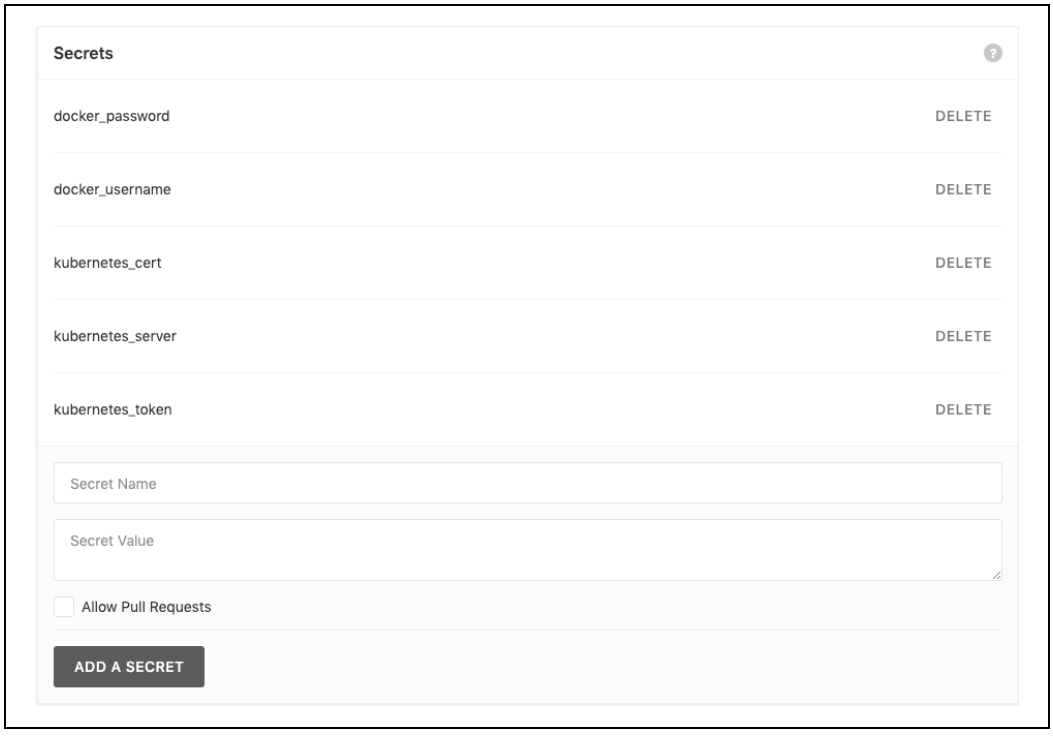
W przypadku serwera Kubernetes potrzebujesz publicznie dostępnego punktu końcowego API Kubernetes.

Do wykonania kroków omawianych w tej sekcji konieczne jest uprawnienie `cluster-admin` w klastrze Kubernetes.

W celu pobrania API punktu końcowego należy wydać następujące polecenie:

```
$ kubectl cluster-info
```

Powinieneś otrzymać komunikat informujący o działaniu Kubernetes pod adresem takim jak *https://kbp.centralus.azmk8s.io:443*. Ta wartość będzie przechowywana w postaci danych poufnych `kubernetes_server`.



Rysunek 5.4. Konfiguracja danych poufnych w Drone

Przechodzimy teraz do utworzenia konta usługi, które będzie używane przez Drone podczas nawiązywania połączenia z klastrem. Skorzystaj z przedstawionego tutaj polecenia, które tworzy `serviceaccount`.

```
$ kubectl create serviceaccount drone
```

Następne polecenie tworzy `clusterrolebinding` dla `serviceaccount`:

```
$ kubectl create clusterrolebinding drone-admin \
  --clusterrole=cluster-admin \
  --serviceaccount=default:drone
```

Kolejnym krokiem jest pobranie tokena dla `serviceaccount`:

```
$ TOKENNAME=`kubectl -n default get serviceaccount/drone -o jsonpath='{.secrets[0].name}'`
$ TOKEN=`kubectl -n default get secret $TOKENNAME -o jsonpath='{.data.token}' | base64 -d`
$ echo $TOKEN
```

Wygenerowane dane wyjściowe w postaci tokena należy przechowywać jako dane poufne `kubernetes_token`.

Potrzebny jest również certyfikat użytkownika w celu uwierzytelnienia w klastrze. Skorzystaj więc z przedstawionego tutaj polecenia i wklej zawartość *ca.crt* do danych poufnych `kubernetes_cert`.

```
$ kubectl get secret $TOKENNAME -o yaml | grep 'ca.crt:'
```

Teraz utwórz rozwiązanie Drone, a następnie przekaż aplikację do rejestru Docker Hub.

Pierwszym krokiem jest *etap kompilacji*, w trakcie którego powstanie frontend opracowany w Node.js. Drone wykorzystuje obrazy kontenera do wykonywania swoich zadań, co zapewnia ogromną elastyczność w zakresie dostępnych możliwości. Na etapie kompilacji skorzystaj z obrazu Node.js pochodzącego z rejestru Docker Hub:

```
pipeline:
  build:
    image: node
    commands:
      - cd frontend
      - npm i redis --save
```

Po zakończeniu kompilacji należy ją przetestować, co odbędzie się na *etapie testowania*. Polega on na wydaniu polecenia `npm` względem nowo utworzonej aplikacji.

```
test:
  image: node
  commands:
    - cd frontend
    - npm i redis --save
    - np
```

Gdy kompilacja i testowanie aplikacji zakończą się sukcesem, będzie można przejść do następnego kroku, którym jest *etap publikowania*. W tym kroku następuje utworzenie obrazu Dockera aplikacji i jego przekazanie do rejestru Docker Hub.

W pliku *.drone.yml* wprowadź poniższą zmianę w kodzie:

```
repo: <twój-rejestr>/frontend

publish:
  image: plugins/docker
  dockerfile: ./frontend/Dockerfile
  context: ./frontend
  repo: dstrebel/frontend
  tags: [latest, v2]
  secrets: [ docker_username, docker_password ]
```

Po zakończeniu operacji tworzenia obrazu Dockera można go przekazać do rejestru Dockera.

Konfiguracja ciągłego wdrażania

Na etapie wdrożenia gotowa aplikacja zostanie przekazana do klastra Kubernetes. W trakcie tego procesu będzie wykorzystany manifest wdrożenia, który znajduje się w katalogu aplikacji frontendu w repozytorium.

```
kubectl:
  image: dstrebel/drone-kubectl-helm
  secrets: [ kubernetes_server, kubernetes_cert, kubernetes_token ]
  kubectl: "apply -f ./frontend/deployment.yaml"
```

Gdy operacja wdrożenia zostanie zakończona, będziesz mógł zobaczyć pody działające w klastrze. Wydanie następującego polecenia pozwoli się upewnić, że pody działają:

```
$ kubectl get pods
```

Istnieje możliwość dodania etapu testowania, w trakcie którego zostaną pobrane informacje o stanie wdrożenia. To jest możliwe po dodaniu w definicji rozwiązania Drone następującego kodu:

```
test-deployment:
  image: dstrebel/drone-kubectl-helm
  secrets: [ kubernetes_server, kubernetes_cert, kubernetes_token ]
  kubectl: "get deployment frontend"
```

Przeprowadzanie operacji uaktualnienia

Teraz pokażemy, jak przeprowadzić uaktualnienie przez wprowadzenie jednej zmiany w kodzie frontendu. W pliku *server.js* dodaj przedstawiony poniżej wiersz kodu, a następnie przekaż zmiany do repozytorium.

```
console.log('Serwer API został uruchomiony.');
```

Gdy to zrobisz, powinieneś zobaczyć, jak przebiega wdrażanie i uaktualnianie oprogramowania w istniejących podach. Po zakończeniu operacji uaktualnienia nowa wersja aplikacji będzie wdrożona.

Prosty eksperyment z inżynierią chaosu

W ekosystemie Kubernetes mamy wiele narzędzi pomagających w przeprowadzaniu w wybranym środowisku eksperymentów związanych z inżynierią chaosu. Gama tych narzędzi jest naprawdę ogromna, od rozwiązań typu „Chaos as a Service" po proste narzędzia do eksperymentów, które doprowadzą do zakończenia działania podów w środowisku. W tym miejscu zdecydowaliśmy się na przedstawienie wybranych narzędzi, o których wiemy, że są z powodzeniem stosowane przez użytkowników.

Gremlin
: To hostingowana usługa zapewniająca zaawansowane funkcje do przeprowadzania eksperymentów związanych z inżynierią chaosu.

PowerfulSeal
: To projekt typu open source oferujący zaawansowane scenariusze inżynierii chaosu.

Chaos Toolkit
: To projekt typu open source, którego zadaniem jest zapewnienie bezpłatnego, otwartego i rozwijanego przez społeczność zestawu narzędzi oraz API dla różnych postaci narzędzi z zakresu inżynierii chaosu.

KubeMonkey
: To narzędzie typu open source oferujące podstawowe możliwości testowania odporności podów w klastrze.

Przeprowadzimy teraz szybki eksperyment pozwalający przetestować odporność aplikacji na awarie. W trakcie eksperymentu działanie podów zostanie zakończone. Do przeprowadzenia eksperymentu użyjemy narzędzia Chaos Toolkit.

```
$ pip install -U chaostoolkit
$ pip install chaostoolkit-kubernetes
$ export FRONTEND_URL="http://$(kubectl get svc frontend -o
jsonpath="{.status.loadBalancer.ingress[*].ip}"):8080/api/"
$ chaos run experiment.json
```

Najlepsze praktyki dotyczące technik ciągłej integracji i ciągłego wdrażania

Przygotowane rozwiązanie w zakresie ciągłej integracji i ciągłego wdrażania nie będzie od razu doskonałe. Dlatego też warto rozważyć zastosowanie wybranych praktyk z poniższej listy, aby iteracyjnie usprawniać to rozwiązanie.

- W przypadku technik ciągłej integracji należy skoncentrować się na automatyzacji i szybkiej kompilacji. Optymalizacja wydajności działania kompilacji zapewni programistom szybkie informacje o tym, czy wprowadzone przez nich zmiany nie doprowadziły do uszkodzenia aplikacji.
- Skoncentruj się na dostarczeniu w rozwiązaniu niezawodnych testów. Dzięki nim programiści będą szybko otrzymywali informacje o potencjalnych problemach w kodzie. Im szybciej takie informacje dotrą do programistów, tym większą osiągną oni produktywność w pracy.
- Podczas wybierania narzędzi z zakresu ciągłej integracji i ciągłego wdrażania upewnij się, że te narzędzia pozwolą zdefiniować rozwiązanie w postaci kodu. To umożliwi umieszczenie kodu rozwiązania w systemie kontroli wersji.
- Upewnij się co do optymalizacji obrazów. To pozwoli zmniejszyć wielkość obrazu, a tym samym płaszczyznę ataku po wdrożeniu danego obrazu do środowiska produkcyjnego. Wieloetapowa kompilacja w Dockerze umożliwia usunięcie pakietów niepotrzebnych do działania aplikacji. Przykładowo pakiet Maven może być potrzebny do skompilowania aplikacji, ale nie jest niezbędny do rzeczywistego uruchomienia obrazu.
- Unikaj używania słowa *latest* w tagu obrazu kontenera. Zamiast tego skorzystaj z *tagu* odwołującego się do identyfikatora kompilacji lub identyfikatora zatwierdzenia kodu w repozytorium Git.
- Jeżeli dopiero zaczynasz korzystanie z technik ciągłego wdrażania, użyj oferowanych przez Kubernetes możliwości w zakresie dostarczania uaktualnień. Rozpoczęcie pracy z nimi jest bardzo łatwe, podobnie jak ich zastosowanie we wdrożeniach. Gdy nabędziesz większej wprawy i większej pewności siebie w pracy z technikami ciągłego wdrażania, zainteresuj się strategiami wdrażania kanarkowego i typu niebieski-zielony.
- W trakcie stosowania technik ciągłego wdrażania upewnij się, że przetestowałeś, jak uaktualnienia połączeń klienta i schematu bazy danych są obsługiwane przez aplikację.
- Testowanie w produkcji pomoże w zagwarantowaniu niezawodności działania aplikacji. Upewnij się również, że dysponujesz dobrym rozwiązaniem w zakresie monitorowania. Testując w produkcji, rozpocznij od operacji na mniejszą skalę i postaraj się ograniczyć pole rażenia eksperymentu.

Podsumowanie

W rozdziale zostały omówione strategie związane z utworzeniem rozwiązania z zakresu ciągłej integracji i ciągłego wdrażania dla aplikacji. Dzięki takiemu rozwiązaniu można zapewnić niezawodny proces dostarczania oprogramowania. Stosowanie technik ciągłej integracji i ciągłego wdrażania pozwala ograniczyć ryzyko i zarazem zwiększyć częstotliwość dostarczania aplikacji do Kubernetes. Przedstawiliśmy także różne strategie wdrażania, które można stosować podczas dostarczania aplikacji.

Najlepsze praktyki w Kubernetes: poradzisz sobie z każdym wyzwaniem!

Systemy informatyczne oparte na chmurze stały się atrakcyjną alternatywą dla standardowej infrastruktury. Wymusiły jednak radykalne zmiany w praktykach tworzenia, wdrażania i utrzymywania aplikacji. Dziś uwaga profesjonalistów skupiona jest na Kubernetes, który w ciągu zaledwie kilku lat stał się faktycznym standardem wdrażania natywnej chmury. Aby tworzone aplikacje funkcjonowały wydajnie, bezawaryjnie i niezawodnie, warto wdrożyć i stosować wzorce oraz najlepsze praktyki. Konieczne jest również przemodelowanie sposobu pracy programistów.

Ta książka jest przeznaczona dla profesjonalnych użytkowników Kubernetes, którzy chcą poznać wzorce i najlepsze praktyki przy wdrażaniu rzeczywistych rozwiązań. Znalazły się tu informacje o jego działaniu w różnych skalach, topologiach i domenach, a także liczne przykłady zastosowania omawianych technologii. Sporo miejsca poświęcono zagadnieniom projektowania aplikacji, konfiguracji i działania usług Kubernetes, a także ciągłej integracji i testowania aplikacji. Ważnym zagadnieniem są takie aspekty zarządzania klastrem jak przydzielanie zasobów, zapewnienie bezpieczeństwa czy autoryzacja i dostęp do klastra. Prezentowane treści zilustrowano fragmentami przejrzystego kodu, co dodatkowo zwiększa przydatność tej książki w pracy inżyniera.

Najciekawsze zagadnienia:

- konfiguracja i projektowanie aplikacji w Kubernetes
- wzorce monitorowania i zarządzanie uaktualnieniami aplikacji
- wdrażanie i wycofywanie aplikacji Kubernetes
- polityka sieciowa i współpraca Kubernetes z architekturą Service Mesh

Brendan Burns jest inżynierem w Microsoft Azure oraz współzałożycielem projektu open source Kubernetes.

Eddie Villalba jest inżynierem oprogramowania w Microsoft Commercial Software Engineering. Specjalizuje się w pracy z Kubernetes i ze środowiskami chmurowymi.

Dave Strebel jest architektem natywnej chmury Microsoft Azure. Głęboko zaangażowany w projekt Kubernetes.

Lachlan Evenson jest głównym menedżerem programu w zespole kontenerów w Microsoft Azure. Prowadzi szkolenia i pomaga w rozpoczęciu pracy z Kubernetes.

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 978-83-283-7232-0

9 788328 372320

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 59,00 zł