O'REILLY®

Kubernetes
Wzorce projektowe

Komponenty wielokrotnego użycia do projektowania natywnych aplikacji chmurowych

Bilgin Ibryam
Roland Huß

Helion

Tytuł oryginału: Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications

Tłumaczenie: Krzysztof Rychlicki-Kicior

ISBN: 978-83-283-6403-5

Authorized Polish translation of the English edition of Kubernetes Patterns
ISBN 9781492050285 © 2019 Bilgin Ibryam and Roland Huß

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/kubewp.zip

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

Printed in Poland.

Spis treści

Część II. Wzorce zachowań

Część IV. Wzorce konfiguracyjne

Część V. Wzorce zaawansowane

ROZDZIAŁ 12.

Wykrywanie Usług

Wzorzec *Wykrywanie Usług* dostarcza stabilną końcówkę, za pośrednictwem której klienci usługi mogą uzyskać dostęp do konkretnych instancji zapewniających usługę. W związku z tym Kubernetes dostarcza liczne mechanizmy, stosowane w zależności od tego, czy konsumenci i producenci usługi znajdują się wewnątrz czy na zewnątrz klastra.

Problem

Aplikacje wdrożone w Kubernetesie rzadko funkcjonują samodzielnie. Z reguły dochodzi do interakcji z innymi usługami wewnątrz klastra lub z systemami poza nim. Do interakcji może dochodzić w wyniku działań wewnętrznych lub z inicjatywy zewnętrznej. Wewnętrzne interakcje są wykonywane zazwyczaj za pomocą konsumenta odpytującego: aplikacja łączy się z innym systemem w momencie startu (lub nieco później), a następnie zaczyna wysyłać i odbierać dane. Dobry przykład takiego zachowania stanowi aplikacja uruchomiona wewnątrz kapsuły, która łączy się z serwerem plików i rozpoczyna konsumpcję plików, lub łączy się z brokerem komunikatów, po czym rozpoczyna pobieranie lub wysyłanie komunikatów. To samo może dotyczyć połączenia z relacyjną bazą danych lub magazynem typu klucz-wartość w celu rozpoczęcia procesu odczytu i zapisu danych.

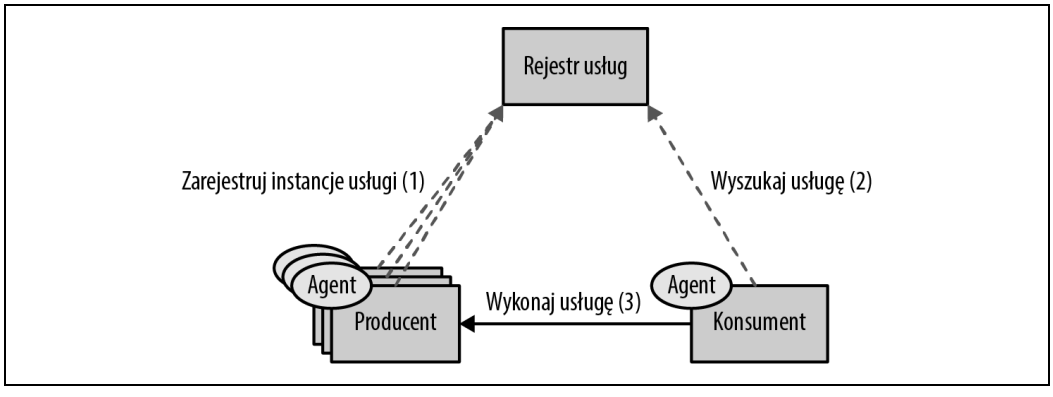
Najważniejszą cechą wspólną tych wszystkich przypadków jest fakt, że aplikacja uruchomiona w ramach kapsuły w pewnym momencie swojego działania decyduje się nawiązać połączenie z inną kapsułą lub systemem zewnętrznym, po czym rozpoczyna transfer danych. W takiej sytuacji aplikacja nie otrzymuje żadnego bodźca z zewnątrz i nie potrzebujemy żadnej dodatkowej konfiguracji w Kubernetesie.

Często korzystamy z tej techniki implementując wzorce przedstawione w rozdziałach 7. i 8. Co więcej, długo działające kapsuły w ramach kontrolerów `DaemonSet` i `ReplicaSet` czasami aktywnie łączą się z innymi systemami przez sieć. Najbardziej popularnym przypadkiem w Kubernetesie są długo działające procesy, które oczekują na otrzymanie sygnału z zewnątrz, najczęściej w postaci połączeń HTTP przychodzących z innych kapsuł z klastra lub zupełnie z zewnątrz. W takiej sytuacji konsumenci usługi muszą w jakiś sposób wykryć kapsuły rozmieszczane dynamicznie przez planistę, podlegające niekiedy skalowaniu w górę i w dół.

Samodzielne śledzenie, rejestrowanie i wykrywanie końcówek dynamicznych kapsuł byłoby niewątpliwie dużym wyzwaniem. W związku z tym Kubernetes implementuje wzorzec *Wykrywanie Usług* za pomocą różnych mechanizmów, które omawiamy w tym rozdziale.

Rozwiązanie

Jeśli przeanalizujemy rozwiązania tego problemu z czasów „przed Kubernetesem", najpopularniejszym z nich było wykrywanie realizowane po stronie klienta. W takim modelu, gdy konsument usługi musi wywołać inną, wyskalowaną na wiele instancji usługę, konieczne jest skorzystanie ze specjalnego agenta wykrywania, odpowiedzialnego za przeszukanie rejestru instancji usług i wybór jednej z nich. Zazwyczaj odbywa się to za pośrednictwem agenta wbudowanego w konsumenta usługi (np. klienta ZooKeepera, klienta Consula czy Ribbona) lub za pomocą innego, kolokowanego procesu — takiego jak Prana — poszukującego usługi w rejestrze (rysunek 12.1).

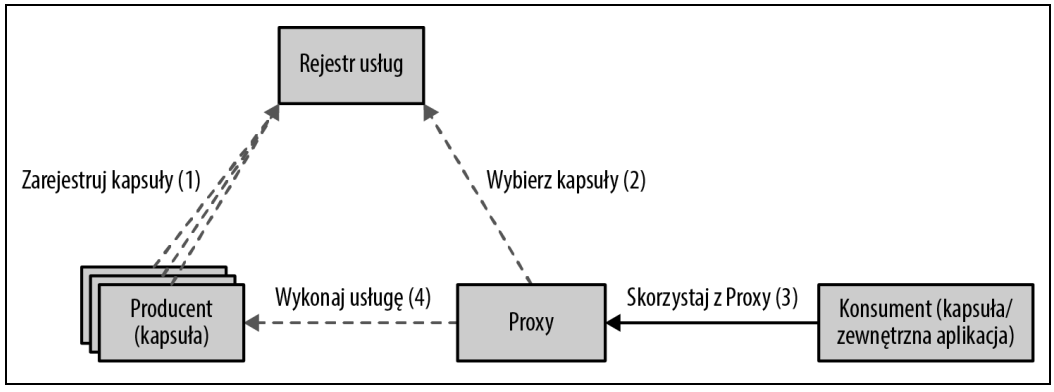


Rysunek 12.1. Wykrywanie usług po stronie klienta

W czasach „po Kubernetesie" wiele spośród niefunkcjonalnych wymagań systemów rozproszonych, takich jak rozmieszczanie, kontrola kondycji, proces przywracania sprawności czy izolacja zasobów, wykonywanych jest na poziomie platformy — to samo dotyczy *Wykrywania Usług* i równoważenia obciążenia. Jeśli odniesiemy się do definicji używanych w architekturze zorientowanej na usługi (SOA — ang. *Service-Oriented Architecture*), instancja dostawcy usługi musi zarejestrować się w rejestrze usług, równolegle dostarczając jej możliwości. Konsument usługi ma dostęp do informacji w rejestrze, aby skorzystać z usługi.

W świecie Kubernetesa wszystko to dzieje się za kulisami, dzięki czemu konsument usługi wywołuje jej wirtualną końcówkę, która dynamicznie wykrywa instancje usług implementowane przez kapsuły. Rysunek 12.2 przedstawia sposób obsługi rejestracji wyszukiwania przez Kubernetesa.

Na pierwszy rzut oka *Wykrywanie Usług* może wydawać się prostym w obsłudze wzorcem. W celu jego implementacji można skorzystać z wielu mechanizmów, w zależności od tego, czy konsument i dostawca usługi znajdują się wewnątrz czy na zewnątrz klastra.



Rysunek 12.2. Wykrywanie usług po stronie serwera

Wykrywanie usług wewnętrznych

Załóżmy, że dysponujesz aplikacją webową, którą chcesz uruchomić na Kubernetesie. Tuż po utworzeniu wdrożenia z kilkoma replikami, planista rozmieści kapsuły w odpowiednich węzłach, a każda z nich otrzyma adres IP klastra przed uruchomieniem. Jeśli inna usługa kliencka, z innej kapsuły, zechce skorzystać z końcówek aplikacji webowej, nie będzie w stanie z góry rozpoznać adresów IP kapsuł dostawcy usługi.

To wyzwanie rozwiązuje zasób usługi Kubernetesa. Usługa dostarcza stały i stabilny punkt wejścia dla kolekcji kapsuł oferujących te same funkcje. Najprostszym sposobem na utworzenie usługi jest użycie polecenia `kubectl expose`, które tworzy usługę dla pojedynczej kapsuły lub wielu kapsuł na podstawie obiektu `Deployment` lub `ReplicaSet`. Polecenie tworzy wirtualny adres IP, określany jako `clusterIP`, i pobiera selektory kapsuły wraz z numerami portów z zasobu, aby utworzyć definicję usługi. Aby uzyskać pełną kontrolę nad tworzoną definicją, należy utworzyć usługę ręcznie (listing 12.1).

Listing 12.1. Prosta usługa

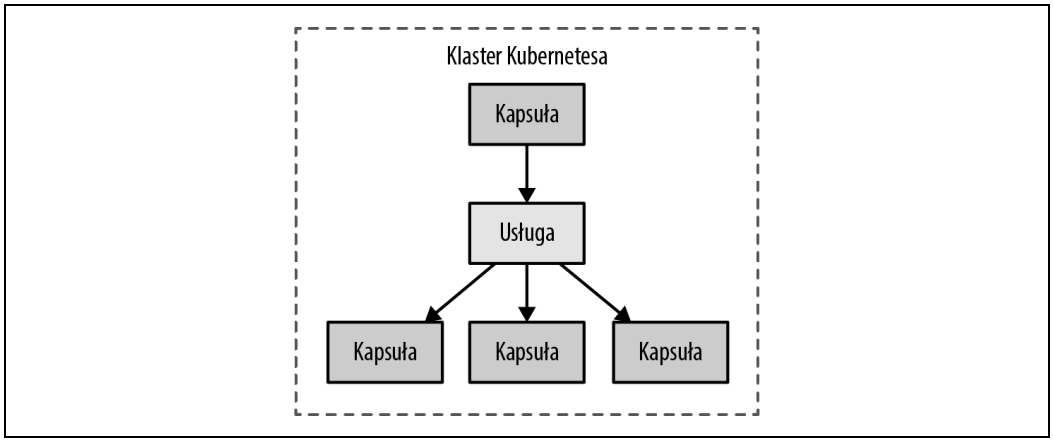
```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  selector:  ❶
    app: random-generator
  ports:
  - port: 80  ❷
    targetPort: 8080 ❸
    protocol: TCP
```

❶ Selektor dopasowany do etykiet kapsuły

❷ Port, na którym można skontaktować się z usługą

❸ Port, na którym nasłuchują kapsuły

Definicja w tym przykładzie tworzy usługę o nazwie `random-generator` (nazwa jest istotna dla późniejszego procesu wykrywania), a także ustawia atrybut `type` na wartość `ClusterIP` (jest to wartość domyślna). Oprócz tego akceptujemy połączenia TCP na porcie 80 i przekierowujemy na port 8080 dla wszystkich kapsuł pasujących do selektora `app: random-generator`. Nie ma znaczenia, kiedy czy jak kapsuły są tworzone — dowolna pasująca kapsuła stanie się celem trasowania (rysunek 12.3).



Rysunek 12.3. Wykrywanie wewnętrznych usług

W tym momencie musimy pamiętać, że po utworzeniu usługi otrzymuje ona `clusterIP`, który jest dostępny tylko z wewnątrz klastra Kubernetesa (stąd nazwa). Adres IP pozostanie niezmieniony, o ile istnieje definicja usługi. W jaki sposób możemy przekazać tę informację do innych aplikacji wewnątrz klastra? Mamy do wyboru jedną z dwóch metod:

Wykrycie za pomocą zmiennych środowiskowych
: Gdy Kubernetes uruchamia kapsułę, do zestawu jej zmiennych środowiskowych trafią wszystkie zmienne należące do dotychczas uruchomionych usług. Nasza usługa `random-generator`, która nasłuchuje na porcie 80, zostanie wstrzyknięta do nowo uruchamianej kapsuły, co widać na przykładzie zmiennych środowiskowych z listingu 12.2. Aplikacja, która jest uruchomiona w tej kapsule, będzie znać nazwę usługi, z której musi skorzystać — wystarczy odczytać wskazane zmienne środowiskowe. Taki mechanizm może być zaimplementowany w aplikacji napisanej w dowolnym języku programowania; łatwo go emulować poza klastrem Kubernetesa, w trakcie tworzenia i testowania oprogramowania. Główny problem związany z tym podejściem to kwestia czasowej zależności tworzenia usługi. Skoro zmienne środowiskowe nie mogą być wstrzykiwane do już uruchomionych kapsuł, koordynaty usług będą dostępne tylko dla kapsuł uruchomionych po starcie usługi. To wymaga zdefiniowania usługi przed uruchomieniem kapsuł, które od niej zależą — lub zrestartowania tychże kapsuł.

Listing 12.2. Związane z usługą zmienne środowiskowe ustawiane automatycznie w kapsule

```
RANDOM_GENERATOR_SERVICE_HOST=10.109.72.32
RANDOM_GENERATOR_SERVICE_PORT=8080
```

Wykrywanie za pomocą wyszukiwania w usłudze DNS
: Kubernetes uruchamia serwer DNS, z którego wszystkie kapsuły są w stanie automatycznie korzystać. Co więcej, w momencie tworzenia nowej usługi, automatycznie otrzymuje ona wpis DNS, którego wszystkie kapsuły mogą natychmiast używać. Zakładając, że klient zna nazwę usługi, z której chce skorzystać, może to uczynić podając pełną, jednoznaczną nazwę domenową (FQDN), np. `random-generator.default.svc.cluster.local`. W tym przypadku `random-generator` to nazwa usługi, `default` to nazwa przestrzeni nazw, `svc` oznacza zasób usługi, a `cluster.local` jest sufiksem danego klastra. Możemy ominąć sufiks klastra, jeśli chcemy, podobnie jak przestrzeń nazw (o ile korzystamy z usługi z tej samej przestrzeni).
  Mechanizm wykrywania DNS nie ma wad znanych z podejścia opartego na zmiennych środowiskowych, ponieważ serwer DNS pozwala na wykrywanie wszystkich usług przez wszystkie kapsuły tuż po zdefiniowaniu usługi. Mimo to wciąż może zaistnieć potrzeba skorzystania ze zmiennych środowiskowych, aby sprawdzić numer portu, jeśli jest on niestandardowy lub nieznany konsumentowi usługi.

Poniżej przedstawiamy wysokopoziomowe opisy usług z atrybutem `type` ustawionym na wartość `ClusterIP`, z których korzystają inne typy:

Wiele portów
: Pojedyncza definicja usługi może obsługiwać wiele portów źródłowych i docelowych. Na przykład, jeśli Twoja kapsuła obsługuje zarówno protokół HTTP na porcie 8080, jak również HTTPS na porcie 8443, nie ma potrzeby definiowania dwóch usług. Pojedyncza usługa może na przykład udostępnić oba protokoły na portach 80 i 443.

Pokrewieństwo sesji
: W momencie pojawienia się nowego żądania, sesja losowo wybiera kapsułę, która zajmie się jego obsługą. To zachowanie można zmienić ustawiając atrybut `sessionAffinity` na wartość `ClientIP`, co sprawi, że wszystkie żądania z tego samego IP klienta trafią do tej samej kapsuły. Pamiętaj, że usługi Kubernetesa równoważą obciążenie na poziomie czwartej warstwy modelu ISO/OSI (transportowej), przez co nie są w stanie przeanalizować pakietów sieciowych i realizować pokrewieństwa sesji np. na podstawie ciastek HTTP.

Sondy gotowości
: W rozdziale 4. nauczyliśmy się definiować sondę gotowości (`readinessProbe`) dla kontenera. Jeśli kapsuła ma zdefiniowane testy gotowości, które na niej nie przechodzą, jest usuwana z listy końcówek usługi, nawet jeśli dopasował ją selektor etykiety.

Wirtualny adres IP
: Kiedy usługa utworzona jest z atrybutem `type: ClusterIP`, otrzymuje ona stabilny, wirtualny adres IP. Ten adres nie ma jednak powiązania z żadnym interfejsem sieciowym i — w praktyce — nie istnieje. To `kube-proxy`, uruchomione na wszystkich węzłach, odpowiada za wykrycie nowej usługi i dodanie do `iptables` w węźle reguł, które przechwycą pakiety przeznaczone do danego wirtualnego adresu IP i podmienią go na prawdziwy adres IP kapsuły. Reguły w `iptables` nie dodają reguł ICMP, a jedynie reguły protokołu określonego w definicji usługi (TCP lub UDP). W związku z tym nie będziesz w stanie odpytać adresu IP usługi za pomocą polecenia `ping`, ponieważ korzysta ono z protokołu ICMP. Naturalnie jest możliwy dostęp do adresu IP usługi za pomocą protokołu TCP (np. poprzez wykonanie żądania HTTP).

Wybór wartości ClusterIP

Podczas tworzenia usługi możemy określić adres IP, który zostanie użyty w polu `.spec.clusterIP`. Musi to być prawidłowy adres IP z określonego zakresu. Choć nie jest to zalecane, dzięki tej opcji możemy poradzić sobie z przestarzałymi aplikacjami, które korzystają z określonego adresu IP, lub gdy chcemy użyć ponownie istniejącego wpisu DNS.

Usługi Kubernetesa z atrybutem `type: ClusterIP` są dostępne tylko z wnętrza klastra. Są one używane do wykrywania kapsuł przez dopasowanie selektorów i jednocześnie są najczęściej stosowanym rodzajem usług. Teraz możemy przeanalizować pozostałe rodzaje usług, które pozwalają na wykrywanie końcówek zdefiniowanych ręcznie.

Ręczne wykrywanie usług

Gdy tworzymy usługę za pomocą atrybutu `selector`, Kubernetes śledzi listę pasujących i gotowych do działania kapsuł na liście zasobów końcówek. W przykładzie z listingu 12.1 możesz sprawdzić wszystkie utworzone przez usługę końcówki, używając w tym celu polecenia `kubectl get endpoints random-generator`. Zamiast przekierowania połączeń do kapsuł wewnątrz klastra, możemy przekierować je do zewnętrznych adresów IP i portów. Możemy to osiągnąć pomijając definicję atrybutu `selector` usługi i ręcznie tworząc zasoby końcówek (listing 12.3).

Listing 12.3. Usługa bez selektora

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ClusterIP
  ports:
  - protocol: TCP
    port: 80
```

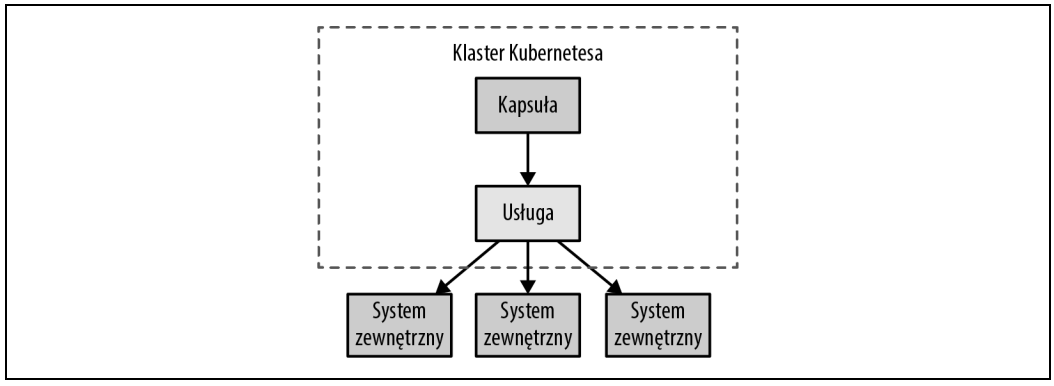
Następnie, za pomocą kodu z listingu 12.4, definiujemy zasób końcówek z tą samą nazwą, jaką ma usługa (zasób ten zawiera docelowe adresy IP i porty).

Listing 12.4. Końcówki zewnętrznej usługi

```
apiVersion: v1
kind: Endpoints
metadata:
  name: external-service ❶
subsets:
  - addresses:
    - ip: 1.1.1.1
    - ip: 2.2.2.2
    ports:
    - port: 8080
```

❶ Nazwa musi być identyczna z tą, jaką ma usługa, która uzyskuje dostęp do tych końcówek.

Ta usługa jest dostępna tylko wewnątrz klastra i może być konsumowana w ten sam sposób co poprzednie, za pomocą zmiennych środowiskowych lub wyszukiwania wpisów DNS. Różnica polega na tym, że lista końcówek jest zarządzana ręcznie, a wartości w niej umieszczone wskazują z reguły na adresy IP spoza klastra (rysunek 12.4).



Rysunek 12.4. Ręczne wykrywanie usług

Choć połączenie z zasobem zewnętrznym stanowi najczęstszy przypadek użycia, nie jest on jedynym. Końcówki mogą przechowywać adresy IP kapsuł, ale nie wirtualne adresy IP innych usług. Zaletą usług jest możliwość dodawania i usuwania selektorów, a także wskazywania na zewnętrznych lub wewnętrznych dostawców bez potrzeby usuwania definicji zasobu, co prowadziłoby do zmiany adresu IP usługi. W związku z tym konsumenci usług mogą korzystać z tego samego adresu IP usługi, na który wskazywali na początku, mimo że faktyczna implementacja dostawcy usługi jest przenoszona ze środowiska on-premise na Kubernetesa bez wpływu na klienta.

Do grupy usług wymagających ręcznej konfiguracji destynacji (celu) możemy zaliczyć tę z listingu 12.5.

Listing 12.5. Usługa z zewnętrzną destynacją (celem)

```
apiVersion: v1
kind: Service
metadata:
  name: database-service
spec:
  type: ExternalName
  externalName: my.database.example.com
  ports:
  - port: 80
```

Ta definicja usługi również nie ma definicji selektora, ale za to jej typ ma wartość `ExternalName`. Z punktu widzenia implementacji jest to istotna różnica. Ta definicja usługi wskazuje na treści określone w atrybucie `externalName` jedynie za pomocą usługi DNS. W ten sposób tworzymy alias dla zewnętrznej końcówki za pomocą rekordu DNS CNAME zamiast przechodzić przez proxy przy użyciu adresu IP. Koniec końców, jest to kolejna metoda dostarczenia abstrakcji Kubernetesa dla końcówek zlokalizowanych poza klastrem.

Wykrywanie usług spoza klastra

Omówione do tej pory mechanizmy wykrywania usług korzystają z wirtualnego adresu IP, który wskazuje na kapsuły lub zewnętrzne końcówki. Wirtualny adres IP jest jednak dostępny jedynie z wnętrza klastra Kubernetesa. Klaster nie jest całym naszym światem; poza połączeniami wychodzącymi z kapsuł do zasobów zewnętrznych, bardzo często konieczne jest dopuszczenie operacji odwrotnej — zewnętrzne aplikacje chcą uzyskać dostęp do końcówek dostarczonych przez kapsuły. Zobaczmy, jak można udostępnić kapsuły klientom funkcjonującym poza klastrem.

Pierwszym ze sposobów na udostępnienie usługi na zewnątrz klastra jest ustawienie atrybutu `type` na wartość `NodePort`.

Definicja z listingu 12.6 tworzy usługę podobnie jak poprzednio, czyli obsługując kapsuły pasujące do selektora `app: random-generator`. Usługa akceptuje połączenia na porcie 80 i wirtualnym adresie IP, przekierowując żądania na port 8080 wybranej kapsuły. Dodatkowo rezerwujemy także port 30036 we wszystkich kapsułach, przekierowując połączenia przychodzące do usługi. Taka operacja sprawia, że usługa jest dostępna wewnętrznie za pomocą wirtualnego adresu IP, a także zewnętrznie, dzięki dedykowanemu portowi dostępnemu w każdym węźle.

Listing 12.6. Usługa typu NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  type: NodePort ❶
  selector:
    app: random-generator
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30036 ❷
    protocol: TCP
```

❶ Otwórz port we wszystkich węzłach.

❷ Określ konkretny port (musi być dostępny) lub pomiń ten atrybut, aby port został określony losowo.

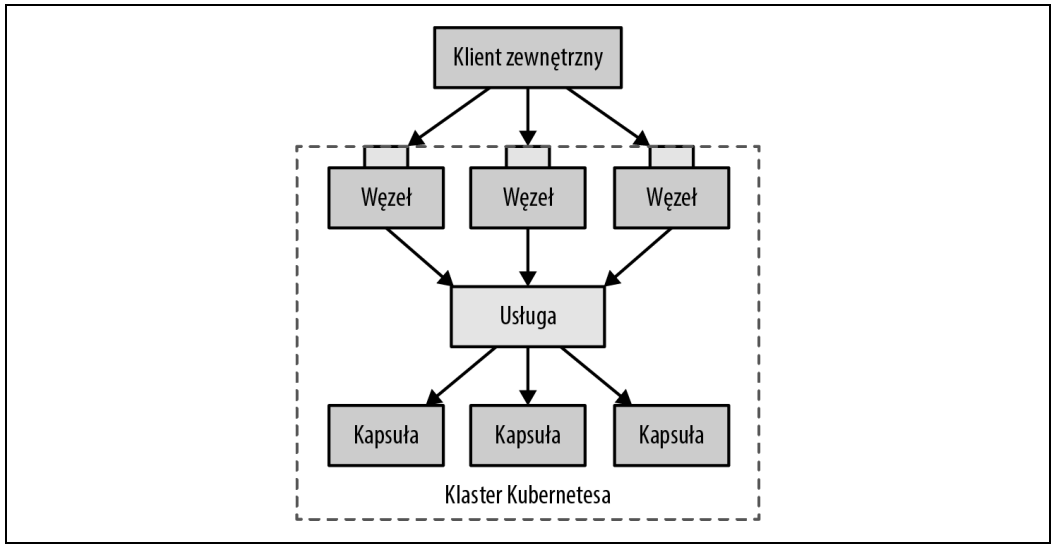
Ta metoda udostępniania usług, przedstawiona na rysunku 12.5, może wydawać się dobrym podejściem, jednak ma ona swoje wady. Przeanalizujmy jej istotne cechy:

Numer portu
: Zamiast określać konkretny numer portu (`nodePort: 30036`), możesz pozwolić Kubernetesowi na wybranie dowolnego dostępnego portu.

Reguły zapory sieciowej
: Skoro ta metoda otwiera port we wszystkich węzłach, rozsądne wydaje się skonfigurowanie dodatkowych reguł zapory sieciowej, aby umożliwić zewnętrznym klientom dostęp do portów węzła.

Wybór węzła
: Zewnętrzny klient może otworzyć połączenie do dowolnego węzła w klastrze. Jeśli jednak węzeł nie jest dostępny, to do aplikacji klienckiej należy obowiązek połączenia się z innym zdrowym węzłem. W związku z tym dobrym pomysłem jest umieszczenie równoważnika obciążenia przed węzłami, aby to jego zadaniem był wybór zdrowych węzłów.

Wybór kapsuły
: Gdy klient nawiązuje połączenie przy użyciu portu węzła, jest ono kierowane do losowo wybranej kapsuły — może ono trafić do tego samego węzła, na którym połączenie zostało otwarte, lub do innego. Można uniknąć tego dodatkowego przeskoku i wymusić na Kubernetesie wybór



Rysunek 12.5. Wykrywanie usług typu NodePort

węzła, na którym połączenie zostało otwarte, przez dodanie atrybutu `externalTrafficPolicy: Local` do definicji usługi. Po ustawieniu tej opcji Kubernetes nie pozwoli połączyć się z kapsułami zlokalizowanymi na innych węzłach, co czasami może być problemem. Aby rozwiązać ten problem, musisz upewnić się, że kapsuły są rozmieszczone na wszystkich węzłach (np. za pomocą *Usług Demona*) lub upewniając się, że klient wie, na których węzłach zostały rozmieszczone zdrowe kapsuły.

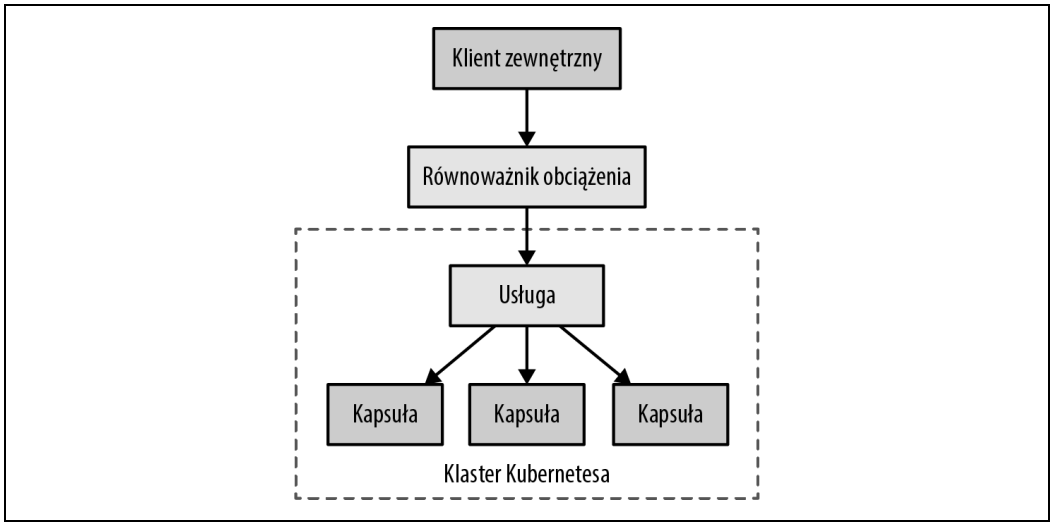
Adresy źródłowe

Z adresami źródłowymi pakietów wysyłanych do różnych typów usług wiążą się pewne osobliwości. Gdy korzystamy z typu `NodePort`, źródłowe adresy klientów w pakietach sieciowych, zawierające adresy IP klientów, są zamieniane na adresy wewnętrzne węzła (za pomocą mechanizmu NAT — Network Address Translation, translacja adresów sieciowych). Gdy aplikacja kliencka wysyła pakiet do węzła nr 1, adres źródłowy jest podmieniany na adres węzła, a adres docelowy — na adres kapsuły. Następnie pakiet jest przekierowywany do węzła nr 2, na którym znajduje się kapsuła. Gdy kapsuła otrzymuje pakiet sieciowy, adres źródłowy nie zawiera oryginalnego adresu klienta — zamiast tego znajdziemy w nim adres węzła nr 1. Aby uniknąć tego zachowania, możemy ustawić atrybut `externalTrafficPolicy` na wartość `Local`, przekierowując ruch tylko do kapsuł zlokalizowanych na węźle nr 1.

Innym sposobem obsługi procesu wykrywania usług dla zewnętrznych klientów jest użycie równoważnika obciążenia (ang. *load balancer*). Widzieliśmy już, jak usługa typu `NodePort` funkcjonuje na bazie zwykłej usługi (typu `ClusterIP`), przez otwarcie portu w każdym węźle. Ograniczeniem tego podejścia jest konieczność zastosowania mechanizmu równoważenia obciążenia dla aplikacji klienckich w celu wyboru zdrowego węzła. Usługa typu `LoadBalancer` rozwiązuje ten problem.

Poza utworzeniem zwykłej usługi i otwarciem portu na wszystkich węzłach typu `NodePort`, udostępniamy usługę na zewnątrz, za pomocą mechanizmu równoważenia obciążenia dostarczonego przez

dostawcę chmury. Rysunek 12.6 przedstawia ten schemat: zamknięty (dostarczony przez dostawcę chmury) *load balancer* funkcjonuje jako brama do klastra Kubernetesa.



Rysunek 12.6. Wykrywanie usług z równoważeniem obciążenia

Ten rodzaj usługi działa tylko jeżeli dostawca chmury obsługuje Kubernetesa i dostarcza rozwiązanie do równoważenia obciążenia.

W kodzie z listingu 12.7 tworzymy usługę równoważącą obciążenie, nadając jej typ `LoadBalancer`. Kubernetes następnie doda adresy IP do pól `.spec` i `.status`.

Listing 12.7. Usługa typu LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  type: LoadBalancer
  clusterIP: 10.0.171.239 ❶
  loadBalancerIP: 78.11.24.19
  selector:
    app: random-generator
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
status: ❷
  loadBalancer:
    ingress:
    - ip: 146.148.47.155
```

❶ Kubernetes przypisuje adresy `clusterIP` i `loadBalancerIP`, gdy staną się one dostępne.

❷ Pole `status` jest zarządzane przez Kubernetesa i dodaje adres IP Ingressa.

Dysponując taką definicją, zewnętrzna aplikacja kliencka może nawiązać połączenie z równoważnikiem obciążenia, który wybierze węzeł i zlokalizuje kapsułę. Szczegóły procesu równoważenia obciążenia i wykrywania usług zmieniają się w zależności od dostawcy chmury. Niektórzy dostawcy pozwalają na definiowanie adresu równoważnika, a inni nie. Jedni oferują mechanizmy do zachowywania adresów źródłowych, a drudzy podmieniają je na adres równoważnika. Szczegóły implementacji będą z pewnością opisane w dokumentacji Twojego dostawcy chmury.

Można skorzystać z jeszcze jednego rodzaju usług — typu *headless* (bezgłowych), dla których nie musimy prosić o dedykowany adres IP. Usługę typu *headless* tworzy się ustawiając atrybut `clusterIP` na wartość `None` wewnątrz sekcji `spec` usługi. W przypadku usług typu *headless* kapsuły wspierające są dodawane do wewnętrznego serwera DNS i jako takie są one najbardziej użyteczne w implementacji usług dla obiektów `StatefulSet` (opisanych w rozdziale 11.).

Wykrywanie usług w warstwie aplikacji

W przeciwieństwie do omówionych do tej pory mechanizmów, Ingress nie jest rodzajem usługi, a osobnym zasobem Kubernetesa, który funkcjonuje na przedzie usług, działając jako inteligentny router i punkt wejściowy dla całego klastra. Ingress z reguły daje dostęp do usług oparty na protokole HTTP, za pomocą dostępnego z zewnątrz adresu URL. Oprócz tego zapewnia równoważenie obciążenia, terminację SSL i wirtualny hosting na bazie nazw. Można jednak spotkać się też z innymi, specjalistycznymi implementacji Ingressa.

Do działania Ingressa konieczne jest dodanie do klastra minimum jednego kontrolera Ingress. Prosty Ingress, który udostępnia pojedynczą usługę, przedstawia listing 12.8.

Listing 12.8. Prosta definicja Ingressa

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: random-generator
spec:
  backend:
    serviceName: random-generator
    servicePort: 8080
```

W zależności od infrastruktury, w której funkcjonuje Kubernetes, a także implementacji kontrolera Ingress, definicja zarezerwuje dostępny na zewnątrz adres IP i udostępni usługę `random-generator` na porcie 80. Ta sytuacja nie różni się zbytnio od usługi z atrybutem `type: LoadBalancer`, która wymaga podania zewnętrznego adresu IP dla każdej definicji usługi. Prawdziwa siła Ingressa tkwi w możliwości użycia pojedynczego, zewnętrznego równoważnika obciążenia i adresu IP do obsługi wielu usług i zmniejszenia kosztów infrastruktury.

Prosta konfiguracja, za pomocą której wiążemy jeden adres IP z wieloma usługami na podstawie ścieżek adresu HTTP URI, jest przedstawiona na listingu 12.9.

Listing 12.9. Definicja Ingressa z powiązaniami

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
```

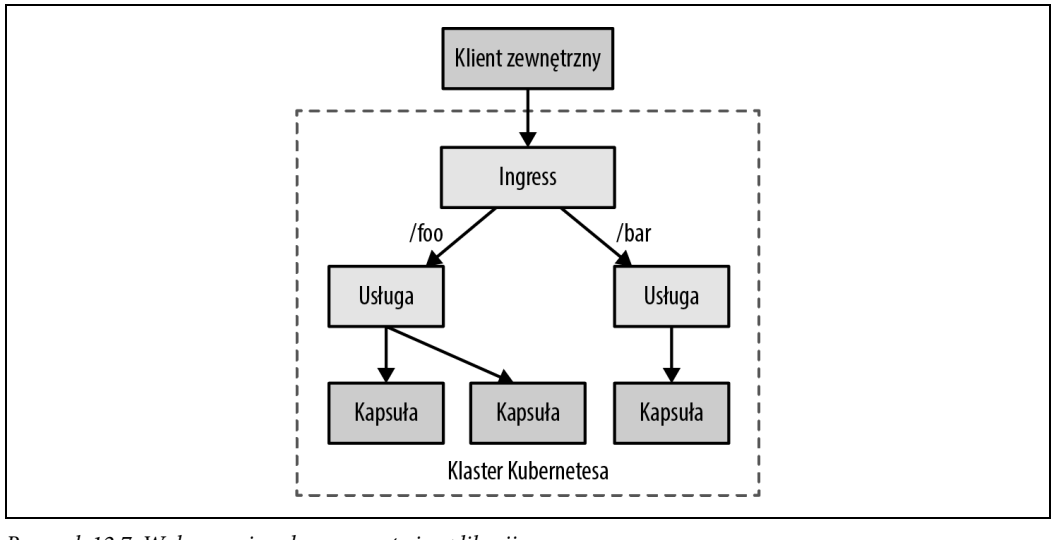
```
    name: random-generator
    annotations:
      nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:  ❶
  - http:
      paths:
      - path: /      ❷
        backend:
          serviceName: random-generator
          servicePort: 8080
      - path: /cluster-status  ❸
        backend:
          serviceName: cluster-status
          servicePort: 80
```

❶ Dedykowane reguły dla kontrolera Ingress, które pozwalają na rozdzielenie żądań na podstawie ścieżki żądania.

❷ Przekierowanie wszystkich żądań do usługi `random-generator`…

❸ …poza ścieżką `/cluster-status`, która kieruje do innej usługi.

Każda implementacja kontrolera Ingress jest inna, dlatego poza standardową definicją, kontroler może wymagać podania dodatkowych parametrów konfiguracji, przekazywanych za pomocą adnotacji. Zakładając, że Ingress jest skonfigurowany prawidłowo, poprzednia definicja dostarczy równoważnik obciążenia i pozyska zewnętrzny adres IP, który obsługuje dwie różne usługi pod dwoma różnymi ścieżkami (rysunek 12.7).



Rysunek 12.7. Wykrywanie usług w warstwie aplikacji

Ingress jest najpotężniejszym, a zarazem najbardziej złożonym mechanizmem *Wykrywania Usług* w Kubernetesie. Użyteczność tego rozwiązania wynika z możliwości udostępniania wielu usług pod tym samym adresem IP, gdy wszystkie usługi korzystają z tego samego protokołu warstwy aplikacji (z reguły jest to HTTP).

Trasy OpenShift

OpenShift firmy Red Hat to popularna, korporacyjna dystrybucja Kubernetesa. Poza pełną zgodnością ze standardowym Kubernetesem, OpenShift dostarcza dodatkowe funkcje. Jedną z nich jest mechanizm tras (ang. *Routes*), który bardzo przypomina w działaniu Ingressa. Oba mechanizmy są tak podobne, że w praktyce trudno zauważyć różnice. Przede wszystkim mechanizm Routes powstał przed wdrożeniem obiektu Ingress w Kubernetesie, dlatego można go określać mianem przodka Ingressa.

Mimo licznych podobieństw, można jednak odnotować wiele różnic pomiędzy obydwoma rodzajami obiektów:

- Trasa jest wykrywana automatycznie przez zintegrowany z OpenShiftem równoważnik obciążenia HAProxy, dlatego nie ma konieczności instalowania dodatkowego kontrolera Ingress. Mimo to możesz podmienić zbudowaną wersję także w równoważniku obciążenia w OpenShift.
- Możesz skorzystać z dodatkowych trybów terminacji TLS, takich jak ponowne szyfrowanie (ang. *re-encryption*) czy przejście (ang. *pass-through*) do usługi.
- Można skorzystać z wielu ważonych komponentów serwerowych do dzielenia ruchu.
- Obsługiwane są domeny wieloznaczne (typu *wildcard*).

Nic nie stoi na przeszkodzie, aby w OpenShift również korzystać z Ingressa. Wszystko jest kwestią wyboru.

Dyskusja

W tym rozdziale omówiliśmy ulubione mechanizmy wykrywania usług w Kubernetesie. Wykrywanie dynamicznych kapsuł z poziomu klastra realizujemy za pomocą zasobu usługi, choć różne opcje w tym zakresie mogą poprowadzić nas do różnych implementacji. Abstrakcja usługi jest wysokopoziomowym, natywnym, typowym dla środowiska chmurowego sposobem na konfigurację szczegółów niskopoziomowych, takich jak wirtualne adresy IP, iptables, rekordy DNS czy zmienne środowiskowe. *Wykrywanie Usług* spoza klastra funkcjonuje na podstawie abstrakcji usługi, koncentrując się na udostępnianiu usług na zewnątrz klastra. Mimo że usługa typu `NodePort` dostarcza podstawowe mechanizmy w zakresie udostępniania usług, stworzenie wysoce dostępnej konfiguracji wymaga integracji z dostawcą infrastruktury.

Tabela 12.1 podsumowuje różne omówione w tym rozdziale sposoby implementacji *Wykrywania Usług* w Kubernetesie, począwszy od prostszych, aż do tych bardziej skomplikowanych. Mamy nadzieję, że dzięki niej łatwiej będzie Ci je zrozumieć.

W tym rozdziale w szczegółowy sposób omówiliśmy kluczowe koncepty związane z dostępem do usług i ich wykrywaniem. Nasza podróż nie kończy się jednak w tym miejscu. Dzięki projektowi *Knative* wprowadzono nowe prymitywy bazujące na Kubernetesie, które pomagają twórcom aplikacji w zaawansowanych aspektach obsługi klientów, budowania aplikacji i obsługi komunikacji.

W kontekście *Wykrywania Usług* szczególne znaczenie ma podprojekt *Knative serving*, ponieważ wprowadza on nowy zasób usługi, tego samego rodzaju co przedstawione w tym rozdziale (ale z inną grupą API). *Knative serving* dostarcza obsługę dla różnych rewizji aplikacji, umożliwiając bardzo elastyczne skalowanie usług za mechanizmem równoważenia obciążenia. Nieco więcej na temat tego

Tabela 12.1. Mechanizmy wykrywania usług

Nazwa	Konfiguracja	Rodzaj klienta	Podsumowanie
`ClusterIP`	`type: ClusterIP` `.spec.selector`	Wewnętrzny	Najczęściej używany mechanizm wykrywania wewnętrznego
Ręczny IP	`type: ClusterIP` `kind: Endpoints`	Wewnętrzny	Wykrywanie zewnętrznego IP
Ręczna pełna nazwa domenowa	`type: ExternalName` `.spec.externalName`	Wewnętrzny	Wykrywanie zewnętrznej nazwy domenowej
Usługa typu *headless*	`type: ClusterIP` `.spec.clusterIP: None`	Wewnętrzny	Wykrywanie na podstawie usługi DNS bez wirtualnego adresu IP
`NodePort`	`type: NodePort`	Zewnętrzny	Preferowane dla ruchu innego niż HTTP
`LoadBalancer`	`type: LoadBalancer`	Zewnętrzny	Wymaga obsługi po stronie infrastruktury chmurowej
Ingress	`kind: Ingress`	Zewnętrzny	Inteligentny mechanizm trasowania na podstawie protokołu warstwy aplikacji (z reguły HTTP).

projektu znajdziesz w punktach „Udostępnianie w Knative” w rozdziale 24. i „Budowanie w Knative” w rozdziale 25., jednak szczegółowa dyskusja na jego temat wykracza poza zakres tej książki. W sekcji „Więcej informacji” w rozdziale 24. znajdziesz linki, dzięki którym dowiesz się więcej na temat Knative.

Więcej informacji

- Przykład *Wykrywania Usług*: *http://bit.ly/2TeXzcr*
- Usługi Kubernetesa: *http://bit.ly/2q7AbUD*
- Usługa DNS dla usług i kapsuł: *http://bit.ly/2Y5jUwL*
- Usługi pomocne w debugowaniu: *http://bit.ly/2r0igMX*
- Stosowanie źródłowego IP: *https://kubernetes.io/docs/tutorials/services/*
- Utwórz zewnętrzny mechanizm typu *load balancer*: *http://bit.ly/2Gs05Wh*
- Porównanie — `NodePort` kontra `LoadBalancer` kontra `Ingress`: *http://bit.ly/2GrVio2*
- Ingress: *https://kubernetes.io/docs/concepts/services-networking/ingress/*
- Ingress w Kubernetesie kontra Route w OpenShift: *https://red.ht/2JDDflo*

Kubernetes: rzeczywiste rozwiązanie istotnych problemów!

Kubernetes jest platformą do orkiestracji kontenerów. Projekt ten należy dziś do najpopularniejszych i najbogatszych narzędzi w swojej klasie, stanowi także podstawę dla wielu innych platform, znanych jako systemy typu PaaS. Dzięki nim Kubernetes zyskał możliwość tworzenia aplikacji, jednak tego rodzaju narzędzia wymagają od programistów i architektów zastosowania odpowiednich wzorców projektowych. Opisują one schematy rozwiązywania problemów na różnych poziomach dokładności, a tym samym umożliwiają efektywne projektowanie i implementację nowoczesnych, elastycznych natywnych aplikacji chmurowych w Kubernetesie.

To książka przeznaczona dla programistów, którzy chcą rozwijać chmurowe aplikacje dla Kubernetesa. Opisano w niej wiele przydatnych wzorców, przedstawiono ich możliwości i wytyczne dotyczące sposobów stosowania. Poszczególne zagadnienia zostały zilustrowane praktycznymi przykładami. Wśród wzorców znalazły się te, które ułatwiają tworzenie aplikacji chmurowych, oraz umożliwiające zarządzanie interakcjami między kontenerami i platformami. Opisano tu różne metody konfiguracji aplikacji w Kubernetesie oraz zasady organizowania kontenerów w ramach kapsuły. Wprowadzono również szereg bardziej zaawansowanych tematów, takich jak techniki rozszerzania platformy czy tworzenie obrazów kontenerów. Poszczególne wzorce nadają się do wielokrotnego użytku i są szczególnie przydatne w przypadku natywnych środowisk chmurowych.

W książce opisano następujące kategorie wzorców:

- wzorce podstawowe
- wzorce zachowań
- wzorce strukturalne
- wzorce konfiguracji
- wzorce zaawansowane

Bilgin Ibryam jest starszym architektem w firmie Red Hat. Kieruje też wieloma projektami w Apache Software Foundation. Jest blogerem, często występuje na różnych konferencjach. Pasjonuje się oprogramowaniem open source, systemami rozproszonymi i mikrousługami.

Dr Roland Huß jest starszym inżynierem oprogramowania w firmie Red Hat i członkiem zespołu serverless pracującego nad projektem Knative. Jest też jednym z głównych autorów kilku popularnych narzędzi do programowania w Javie.

ISBN 978-83-283-6403-5