Sieć

Service, DNS, Ingress, Gateway API i polityki sieciowe.

- Service: jak ruch trafia do poda
- DNS w klastrze i wykrywanie usług
- Ingress i Gateway API: wejście z zewnątrz
- Sieć klastra i polityki sieciowe

Service: jak ruch trafia do poda

Poziom: średni **Dotyczy: Kubernetes 1.30+** **Czas: ~20 min**

Service daje podom stały adres i stałą nazwę, dzięki czemu reszta klastra może się z nimi komunikować mimo że same pody znikają i pojawiają się na nowo z innym adresem IP.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
DNS	Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)

Fiszka

TYLKO ODCZYT

Wylistowanie usług w przestrzeni nazw: kubectl get svc

```
kubectl get svc -n NazwaPrzestrzeni
```

Sprawdzamy, jakie usługi istnieją w danej przestrzeni nazw, jaki mają typ, jaki adres ClusterIP i na jakich portach nasłuchują.

Przykład:

```
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
sklep-api     ClusterIP   10.96.142.7     <none>        8080/TCP       14d
sklep-front   NodePort    10.96.201.33    <none>        80:31080/TCP   14d
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

PLIK KONFIGURACYJNY

Utworzenie usługi typu ClusterIP

```
apiVersion: v1
kind: Service
metadata:
  name: NazwaUslugi
  namespace: NazwaPrzestrzeni
spec:
  type: ClusterIP
  selector:
    app: NazwaAplikacji
  ports:
    - port: NumerPortu
      targetPort: NumerPortu
      protocol: TCP
```

```
kubectl apply -f service-clusterip.yaml
```

Tworzymy usługę dostępną tylko wewnątrz klastra, adresowaną przez pozostałe pody pod nazwą NazwaUslugi — dokładny mechanizm rozwiązywania tej nazwy opisuje strona DNS w klastrze i wykrywanie usług.

DNS = Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP ·**TCP** = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia ·**kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podejrzenie, które pody stoją za usługą: kubectl get endpoints

```
kubectl get endpoints NazwaUslugi -n NazwaPrzestrzeni
```

Sprawdzamy listę adresów IP i portów podów faktycznie dobranych przez selektor usługi — to jest pierwsze miejsce, w którym widzimy, czy usługa w ogóle ma do czego kierować ruch.

Przykład:

```
NAME        ENDPOINTS                           AGE
sklep-api   10.244.1.12:8080,10.244.2.7:8080    14d
```

IP = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podejrzenie tego samego zestawu w nowszym formacie: kubectl get endpointslices

```
kubectl get endpointslices -n NazwaPrzestrzeni -l kubernetes.io/service-name=NazwaUslugi
```

Sprawdzamy adresy podów w formacie, który realnie stoi pod maską usługi od wersji Kubernetesa 1.21 wzwyż, opisanym niżej.

Przykład:

```
NAME               ADDRESSTYPE   PORTS   ENDPOINTS                 AGE
sklep-api-x7k2p     IPv4          8080    10.244.1.12,10.244.2.7    14d
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

PLIK KONFIGURACYJNY

Utworzenie usługi typu NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: NazwaUslugi
  namespace: NazwaPrzestrzeni
spec:
  type: NodePort
  selector:
    app: NazwaAplikacji
  ports:
    - port: NumerPortu
      targetPort: NumerPortu
      nodePort: 31080
```

```
    protocol: TCP
```

Otwieramy port na każdym węźle klastra, przez który ruch trafia dalej do usługi — zakres portów NodePort i sytuacje, w których po niego sięgamy, opisujemy niżej w tabeli typów.

TCP = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

TYLKO ODCZYT

Diagnostyczne przekierowanie portu na usługę: kubectl port-forward

```
kubectl port-forward svc/NazwaUslugi NumerPortu:NumerPortu -n NazwaPrzestrzeni
```

Otwieramy tymczasowy tunel z naszej stacji roboczej do usługi wewnątrz klastra, ważny tylko na czas działania tej komendy.

Przykład:

```
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

OSTROŻNIE · NIEODWRACALNE

Usunięcie usługi

```
kubectl delete svc NazwaUslugi -n NazwaPrzestrzeni
```

Usuwamy obiekt Service — pody, które ona adresowała, dalej działają, tracą jedynie stały punkt wejścia.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

> O 10:40 aplikacja katalogu produktów na app01 przestaje odpowiadać na zapytania do wewnętrznej usługi cenowej, mimo że oba komponenty ewidentnie działają w `kubectl get pods`. Tomek sprawdza kod aplikacji i widzi, że ktoś kiedyś na szybko wpisał na sztywno adres IP jednego z podów usługi cenowej zamiast jej nazwy, a ten pod zniknął przy ostatniej aktualizacji obrazu i dostał nowy adres. Po zmianie adresu na nazwę usługi (`Service`) połączenie przestaje się psuć przy każdym kolejnym wdrożeniu, niezależnie od tego, które konkretnie pody akurat za nią stoją.

Po co Service istnieje

> **Pojęcie: Service.** Obiekt Kubernetesa, który daje stabilny adres IP i stabilną nazwę DNS zestawowi podów wybranych po etykiecie, niezależnie od tego, ile tych podów aktualnie żyje i pod jakimi adresami. Bez Service jedyny sposób, żeby jeden pod dogadał się z drugim, to znajomość jego adresu IP — a ten adres jest tymczasowy.

Każdy pod w Kubernetesie dostaje własny adres IP w momencie startu i traci go bezpowrotnie w momencie, w którym pod znika — restart poda po awarii węzła, aktualizacja obrazu przez Deployment albo zwykłe przeskalowanie tworzy nowy pod z nowym adresem, nie wskrzesza starego. Aplikacja, która zapamiętała adres IP innego poda i próbuje się z nim łączyć bezpośrednio, prędzej czy później zaczyna łączyć się donikąd — a najgorsze w tym jest to, że działa to poprawnie przez tygodnie, aż do pierwszego restartu.

Service rozwiązuje ten problem jedną warstwą pośrednią: zamiast pamiętać adres poda, zapamiętujemy nazwę usługi. Kubernetes sam utrzymuje aktualną listę adresów podów stojących za tą nazwą i sam kieruje na nie ruch, także wtedy, gdy pody są wymieniane w tle.

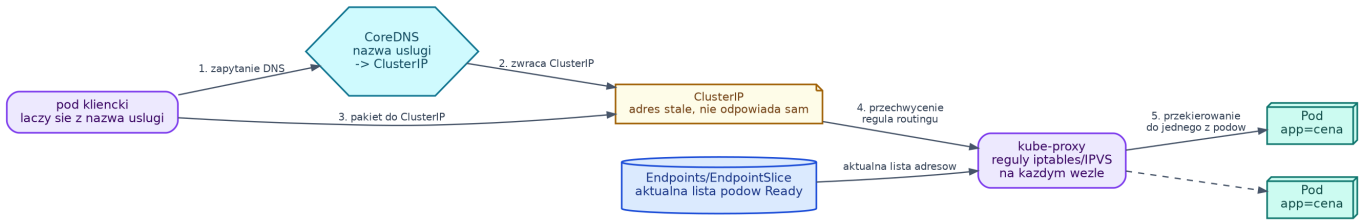
Jak to działa: selektor i Endpoints

Service łączy się z podami przez selektor etykiet — te same etykiety, które nosi Deployment albo StatefulSet zarządzający podami. Kontroler endpointów w płaszczyźnie sterowania bez przerwy obserwuje, które pody pasują do selektora danej usługi i są w stanie gotowości (`Ready`), i na bieżąco aktualizuje listę ich adresów.

> **Pojęcie: Endpoints i EndpointSlice.** Endpoints to starszy obiekt trzymający płaską listę adresów IP i portów podów pasujących do selektora usługi — jeden obiekt na całą usługę, niezależnie od liczby podów. EndpointSlice to jego następca, wprowadzony jako stabilny od wersji 1.21, który dzieli tę samą listę na mniejsze kawałki po maksymalnie 100 adresów, żeby aktualizacja jednego poda w usłudze z tysiącem replik nie wymagała przepisania

> całego, ogromnego obiektu za każdym razem. Dla nas jako administratorów oba obiekty pokazują to samo: rzeczywisty, aktualny zestaw adresów, do których usługa kieruje ruch.

Ruch do usługi nie przechodzi jednak przez żaden proces pośredniczący na ścieżce pakietu — `kube-proxy`, uruchomiony jako demon na każdym węźle, na bieżąco przekłada adres i port usługi na reguły systemu operacyjnego (`iptables` albo `IPVS`, w zależności od trybu), które przekierowują pakiet bezpośrednio do adresu jednego z podów. Adres ClusterIP usługi nigdy nie odpowiada sam — to wpis routingu na każdym węźle sprawia, że pakiet wysłany na ten adres ląduje na jednym z realnych podów.



Pod kliencki rozwiązuje nazwę usługi przez DNS na ClusterIP, a kube-proxy na podstawie aktualnej listy Endpoints przekierowuje pakiet do jednego z żywych podów (kliknięcie otwiera pełny rozmiar)

Typy Service

Typ	Jak wygląda ruch	Kiedy sięgamy
`ClusterIP` (domyślny)	stały adres widoczny tylko wewnątrz klastra	komunikacja między usługami wewnątrz klastra — zdecydowana większość przypadków
`NodePort`	dodatkowo ten sam port otwarty na każdym węźle klastra	szybki dostęp z zewnątrz bez zewnętrznego load balancera, testy, środowiska bez chmury publicznej
`LoadBalancer`	dodatkowo zewnętrzny load balancer chmury z publicznym adresem	publikacja usługi w chmurze publicznej, gdzie kontroler chmury sam tworzy load balancer
`ExternalName`	brak selektora i brak proxy — zwraca rekord `CNAME` na zewnętrzną nazwę	odwołanie się z klastra do usługi poza nim (np. zewnętrznej bazy danych) tą samą nazwą DNS co do usługi wewnętrznej

Uważaj: `LoadBalancer` bez kontrolera chmury (np. w klastrze on-premise postawionym `kubeadm`) zostaje trwale w stanie `<pending>` na polu `EXTERNAL-IP` — sam typ nic nie tworzy, potrzebuje

kontrolera, który umie rozmawiać z konkretną infrastrukturą (chmurą albo np. MetalLB w środowisku lokalnym).

Service bez selektora

Service nie musi mieć selektora wcale — wtedy Kubernetes nie tworzy dla niej automatycznie żadnych endpointów, a my utrzymujemy je ręcznie:

```
apiVersion: v1
kind: Service
metadata:
  name: NazwaUslugi
  namespace: NazwaPrzestrzeni
spec:
  ports:
    - port: NumerPortu
      protocol: TCP
---
apiVersion: v1
kind: Endpoints
metadata:
  name: NazwaUslugi
  namespace: NazwaPrzestrzeni
subsets:
  - addresses:
      - ip: 203.0.113.10
    ports:
      - port: NumerPortu
```

> **Pojęcie: Service bez selektora.** Wzorzec przydatny, gdy chcemy adresować z klastra coś, co żyje poza nim — bazę danych na fizycznym serwerze, usługę na innej platformie — tą samą nazwą DNS, jakiej używalibyśmy dla zwykłej usługi wewnętrznej. Aplikacja wewnątrz klastra łączy się z `NazwaUslugi`, nie wiedząc, że po drugiej stronie stoi adres spoza klastra.

Headless Service

> **Pojęcie: headless Service.** Usługa z `clusterIP: None` — nie dostaje własnego, stałego adresu IP, więc zapytanie DNS o jej nazwę zwraca nie jeden adres pośredniczący, tylko listę adresów IP wszystkich podów pasujących do selektora, każdego z osobna.

```
apiVersion: v1
kind: Service
metadata:
  name: NazwaUslugi
  namespace: NazwaPrzestrzeni
spec:
  clusterIP: None
  selector:
    app: NazwaAplikacji
  ports:
    - port: NumerPortu
```

Headless Service jest tym, czego realnie używa StatefulSet: każda replika w StatefulSecie ma stałą tożsamość sieciową w postaci własnej nazwy DNS (`nazwa-0.NazwaUslugi`, `nazwa-1.NazwaUslugi`), a nie wspólny, uśredniony adres. Ma to znaczenie dla baz danych i innych systemów rozproszonych, w których klient musi połączyć się z konkretnym egzemplarzem — np. wyłącznie z liderem klastra replikacji — zamiast trafić losowo na dowolną replikę przez zwykły ClusterIP.

kubectl port-forward jako narzędzie diagnostyczne

`kubectl port-forward` otwiera tunel na czas życia jednej komendy w naszym terminalu — to narzędzie do sprawdzenia, czy usługa w ogóle odpowiada, zanim zaczniemy szukać przyczyny gdzie indziej, a nie sposób na trwałe udostępnienie usługi. Tunel znika, gdy zamykamy terminal albo przerywamy komendę `Ctrl+C`, i nie przechodzi przez żaden z mechanizmów opisanych w tabeli typów wyżej — działa nawet dla usługi typu `ClusterIP`, bo cały ruch tuneluje się przez połączenie do API Kubernetesa, z pominięciem sieci klastra.

Nasz przykład: kolorowy nginx

Ten sam mechanizm selektora i Endpoints stoi za usługą `kolorowy-nginx`, opisaną na stronie k3s w laboratorium: pierwszy deploy, nginx w kolorach. Trzy pody nginksa noszą tam tę samą etykietę `app: kolorowy-nginx`

, a Service o tej nazwie kieruje na nie ruch, nie wiedząc nic ani o kolorach, ani o nazwach poszczególnych podów. Kolejne osobne połączenia trafiają losowo do innego z adresów z listy Endpoints tej usługi, co jest najprostszym możliwym sposobem, żeby zobaczyć działanie `kube-proxy` gołym okiem zamiast tylko na papierze.

```
apiVersion: v1
kind: Service
metadata:
  name: kolorowy-nginx
  namespace: kolory
spec:
  type: ClusterIP
  selector:
    app: kolorowy-nginx
  ports:
    - port: 80
      targetPort: 80
```

Częste błędy

Selektor usługi nie pasuje do żadnego poda. `kubectl get endpoints` pokazuje puste pole `<none>` mimo że pody wyglądają na działające — najczęstszą przyczyną jest literówka w etykiecie albo selektor skopiowany z innej usługi bez dostosowania.

Port usługi pomylony z portem kontenera. `port` to numer, pod którym usługa nasłuchuje, `targetPort` to numer portu, na którym faktycznie nasłuchuje kontener — jeśli aplikacja w kontenerze słucha na innym porcie niż podany w `targetPort`, usługa istnieje, endpointy istnieją, a każde połączenie kończy się odmową po stronie poda.

Oczekiwanie, że LoadBalancer zadziała bez kontrolera chmury. Pole `EXTERNAL-IP` zostaje w stanie `<pending>` bezterminowo w klastrze bez zintegrowanego dostawcy chmury albo bez lokalnego kontrolera typu MetalLB.

□□ Ćwiczenia

> Ćwiczenia zakładają dostęp do klastra testowego (np. `kind` albo `minikube`) i wdrożony przykładowy Deployment z co najmniej dwiema replikami.

Ćwiczenie 1 — obserwacja zmiany adresu poda

```
kubectl get pods -n NazwaPrzestrzeni -o wide
kubectl delete pod NazwaPoda -n NazwaPrzestrzeni
kubectl get pods -n NazwaPrzestrzeni -o wide
```

Zapisujemy adres IP jednego z podów, usuwamy go, po ponownym wystawieniu przez kontroler porównujemy nowy adres ze starym.

Odpowiedź

Nowy pod dostaje inny adres IP niż usunięty — sama usługa (jeśli jest) i jej adres ClusterIP pozostają bez zmian, bo to warstwa pośrednia, o którą chodzi na tej stronie.

Pytania kontrolne

1. Dlaczego usługa typu ClusterIP działa dla podów wewnątrz klastra, ale nie da się jej otworzyć bezpośrednio z przeglądarki poza klastrem?

Odpowiedź

Adres ClusterIP istnieje wyłącznie jako wpis w regułach routingu (`iptables` / `IPVS`) utrzymywanych przez `kube-proxy` na węzłach klastra — poza siecią klastra ten adres nikomu nic nie mówi, bo żaden węzeł spoza klastra nie ma takich reguł.

2. Po co w ogóle istnieje EndpointSlice, skoro Endpoints robi to samo?

Odpowiedź

Endpoints trzyma całą listę adresów w jednym obiekcie, więc przy usłudze z bardzo dużą liczbą replik każda drobna zmiana (jeden pod wchodzi lub wychodzi z gotowości) wymusza przesłanie całego, dużego obiektu do każdego obserwującego komponentu. EndpointSlice dzieli tę listę na kawałki, więc zmiana jednego adresu porusza tylko jeden mały kawałek.

Powiązane: DNS w klastrze i wykrywanie usług · Ingress i Gateway API: wejście z zewnątrz · Diagnostyka: objaw, przyczyna, pierwsze kroki

Źródła: dokumentacja kubernetes.io, rozdziały „Service", „EndpointSlice" i „Service Internals"

DNS w klastrze i wykrywanie usług

◫◫ Poziom: średni **◫◫ Dotyczy: Kubernetes 1.30+, CoreDNS 1.11+** **⏱ Czas: ~15 min**

Każda usługa w klastrze dostaje własną nazwę DNS rozwiązywalną z wnętrza dowolnego poda — ta strona pokazuje, jak wygląda ten schemat nazw, co go obsługuje i jak diagnozujemy, kiedy przestaje działać.

◫◫ Skróty użyte na tej stronie (4)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
DNS	Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP
FQDN	Fully Qualified Domain Name: pełna nazwa hosta razem z domeną, np. docker01.ad.zielona.example
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)

Fiszka

◫◫ OSTROŻNIE · NIEODWRACALNE

Sprawdzenie rozwiązywania nazwy usługi z wnętrza poda: nslookup

```
kubectl run diagnostyka --rm -it --image=busybox:1.36 --restart=Never -- nslookup
NazwaUslugi.NazwaPrzestrzeni.svc.cluster.local
```

Uruchamiamy jednorazowy pod diagnostyczny i sprawdzamy, czy nazwa usługi rozwiązuje się na adres ClusterIP — usuwa się sam po zakończeniu (`--rm`).

Przykład:

```
Server:    10.96.0.10
Address:   10.96.0.10:53

Name:      sklep-api.produkcja.svc.cluster.local
Address:   10.96.142.7
```

□□ **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

□□ TYLKO ODCZYT

Sprawdzenie pliku resolv.conf poda: cat

```
kubectl exec NazwaPoda -n NazwaPrzestrzeni -- cat /etc/resolv.conf
```

Sprawdzamy, na jaki adres serwera DNS pod jest skierowany i jaka wartość `ndots` została mu wstrzyknięta — obie sprawy opisujemy niżej.

Przykład:

```
nameserver 10.96.0.10
search produkcja.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

□□ **DNS** = Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

□□ TYLKO ODCZYT

Podejrzenie usługi CoreDNS: kubectl get pods

```
kubectl get pods -n kube-system -l k8s-app=kube-dns
```

Sprawdzamy, czy pody CoreDNS w ogóle działają — bez nich żadna nazwa w klastrze się nie rozwiąże, nawet jeśli reszta klastra wygląda zdrowo.

Przykład:

```
NAME                       READY   STATUS    RESTARTS   AGE
coredns-6f9c4b8d5f-2xk9m   1/1     Running   0          21d
coredns-6f9c4b8d5f-p7qwl   1/1     Running   0          21d
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podgląd konfiguracji CoreDNS: kubectl get configmap

```
kubectl get configmap coredns -n kube-system -o yaml
```

Sprawdzamy plik `Corefile`, czyli faktyczną konfigurację, według której CoreDNS rozwiązuje zapytania — miejsce, do którego zaglądamy, gdy ktoś dodał niestandardowe przekierowanie dla domeny wewnętrznej firmy.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Sprawdzenie logów CoreDNS: kubectl logs

```
kubectl logs -n kube-system -l k8s-app=kube-dns --tail=100
```

Sprawdzamy błędy rozwiązywania nazw i ewentualne przeciążenie — powtarzające się `SERVFAIL` dla tej samej nazwy wskazuje na konkretny problem opisany niżej, a nie na przypadkową awarię sieci.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 9:50 nowo wdrożona usługa raportowania na klastrze produkcyjnym nie może połączyć się z bazą kolejki w przestrzeni `cache`, mimo że obie przestrzenie są w tym samym klastrze i baza działa. Tomek sprawdza kod i widzi, że deweloper skopiował adres usługi z przykładu w innej przestrzeni nazw, więc krótka nazwa `baza-kolejki` bez dopisku przestrzeni po prostu nie istnieje tam, gdzie próbuje jej szukać resolwer poda. Po dopisaniu pełnej nazwy `baza-kolejki.cache` połączenie działa od razu, bez żadnej zmiany po stronie samej bazy.

Czym jest CoreDNS

> **Pojęcie: CoreDNS.** Serwer DNS działający jako zwykły Deployment wewnątrz klastra, w przestrzeni nazw `kube-system`, który odpowiada na zapytania DNS ze wszystkich podów w klastrze — to on tłumaczy nazwę usługi na jej adres ClusterIP i to przez niego przechodzi każde zapytanie o domenę spoza klastra wysłane z dowolnego poda. Usługa `kube-dns` (nazwa zachowana z czasów, gdy rolę tę pełnił inny serwer) wystawia CoreDNS pod stałym adresem, wpisanym automatycznie do `/etc/resolv.conf` każdego poda.

Każdy nowo utworzony obiekt Service dostaje wpis w CoreDNS automatycznie, bez naszej ingerencji — kontroler CoreDNS obserwuje API Kubernetesa i na bieżąco aktualizuje odpowiedzi na zapytania, w tym samym rytmie, w jakim usługi i pody powstają i znikają.

Schemat nazw

Pełna, w pełni kwalifikowana nazwa usługi w klastrze ma zawsze tę samą budowę:

```
NazwaUslugi.NazwaPrzestrzeni.svc.cluster.local
```

> **Pojęcie: cluster.local.** Domena bazowa klastra — domyślna wartość, którą można zmienić przy instalacji klastra, ale w zdecydowanej większości instalacji zostaje bez zmian. Segment `svc` odróżnia usługi od innych typów rekordów, jakie CoreDNS potrafi wystawić (np. `pod` dla adresów samych podów).

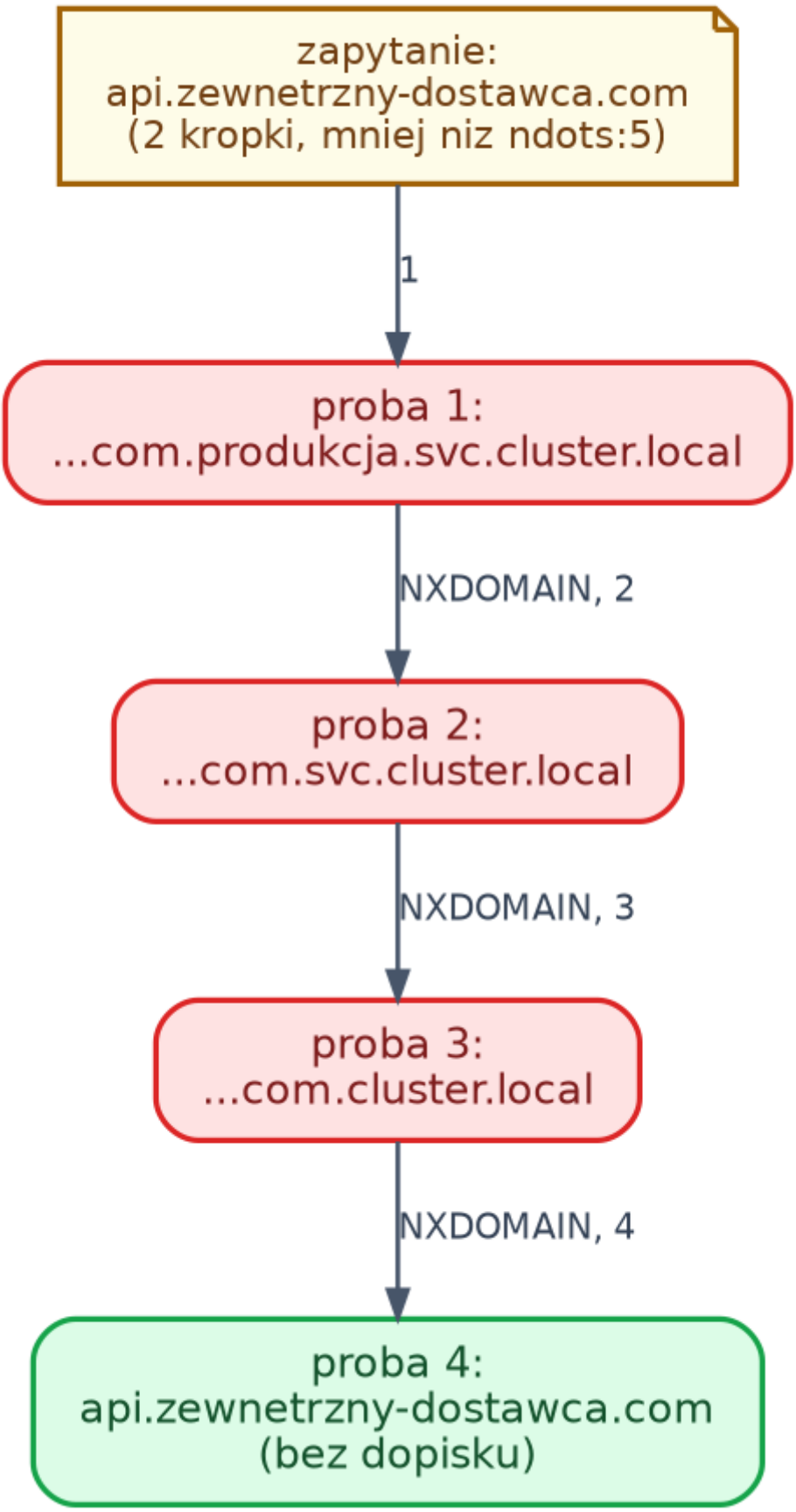
Z tego schematu wynika wprost, jak wygląda komunikacja między przestrzeniami nazw: pod w przestrzeni `produkcja` odwołujący się do usługi `sklep-api` w tej samej przestrzeni może użyć samej krótkiej nazwy `sklep-api` — mechanizm `search` z `resolv.conf`, pokazany w Fiszce, dokleja resztę. Ten sam pod odwołujący się do usługi `redis` żyjącej w przestrzeni `cache` musi użyć co najmniej `redis.cache`, bo krótka nazwa `redis` w przestrzeni `produkcja` nie istnieje. Pomijanie przestrzeni nazw przy odwołaniach między namespace'ami jest jednym z najczęstszych błędów przy pierwszym kontakcie z Kubernetesem — działa lokalnie w tej samej przestrzeni, przestaje działać, gdy usługę przenosimy albo gdy aplikacja próbuje odwołać się do usługi sąsiada.

ndots i wolne zapytania do domen zewnętrznych

> **Pojęcie: ndots.** Parametr resolwera DNS systemu Linux mówiący, ile kropek musi zawierać nazwa, żeby resolwer spróbował rozwiązać ją jako w pełni kwalifikowaną, zanim zacznie

> doklejać do niej domeny wyszukiwania z listy `search`. Kubernetes ustawia domyślnie `ndots:5` w każdym podzie.

Konsekwencja tej wartości jest konkretna: nazwa `api.zewnetrzny-dostawca.com` ma tylko dwie kropki, czyli mniej niż `ndots:5`, więc resolwer w pierwszej kolejności próbuje ją rozwiązać jako `api.zewnetrzny-dostawca.com.produkcja.svc.cluster.local`, potem `api.zewnetrzny-dostawca.com.svc.cluster.local`, potem `api.zewnetrzny-dostawca.com.cluster.local` — dopiero po tych trzech nieudanych próbach próbuje samej nazwy bez żadnego dopisku. Każda z tych prób to osobne zapytanie DNS z osobnym oczekiwaniem na odpowiedź (albo na timeout), więc pierwsze połączenie aplikacji do zewnętrznego API bywa zauważalnie wolniejsze niż to samo połączenie wykonane poza klastrem, mimo że docelowy serwer odpowiada natychmiast.


zapytanie:
api.zewnetrzny-dostawca.com
(2 kropki, mniej niz ndots:5)
1
proba 1:
...com.produkcja.svc.cluster.local
NXDOMAIN, 2
proba 2:
...com.svc.cluster.local
NXDOMAIN, 3
proba 3:
...com.cluster.local
NXDOMAIN, 4
proba 4:
api.zewnetrzny-dostawca.com
(bez dopisku)

Resolwer przy ndots 5 próbuje kolejno trzy domeny wyszukiwania i dopiero po trzech nieudanych próbach rozwiązuje nazwę zewnętrzną bez żadnego dopisku (kliknięcie otwiera pełny rozmiar)

Uważaj: dla aplikacji intensywnie korzystających z domen zewnętrznych rozwiązaniem bywa dopisanie kropki na końcu w pełni kwalifikowanej nazwy w kodzie aplikacji (`api.zewnetrzny-dostawca.com.`, z kropką na końcu) albo obniżenie `ndots` dla konkretnego poda przez pole `dnsConfig` — nie zmieniamy globalnie `ndots` dla całego klastra bez pełnego zrozumienia konsekwencji dla wszystkich pozostałych aplikacji.

Zmienne środowiskowe z adresami usług

> **Pojęcie: zmienne środowiskowe usług.** Starszy, poprzedzający CoreDNS mechanizm wykrywania usług — Kubernetes wstrzykuje do każdego poda zmienne środowiskowe typu `NAZWAUSLUGI_SERVICE_HOST` i `NAZWAUSLUGI_SERVICE_PORT` dla każdej usługi, która istniała w klastrze w momencie startu poda.

Mechanizm ten ma istotne ograniczenie, przez które dziś jest tylko ciekawostką historyczną: zmienne dostają wyłącznie pody uruchomione po utworzeniu danej usługi, bo są wstrzykiwane raz, przy starcie kontenera, i nigdy się nie odświeżają. Usługa utworzona już po starcie poda jest dla niego niewidoczna tą drogą, a jedynym sposobem na jej dostrzeżenie jest restart poda. DNS nie ma tego ograniczenia — rozwiązuje nazwę na bieżąco, przy każdym zapytaniu.

Diagnostyka awarii DNS w klastrze

Najczęstsze objawy awarii DNS i to, co sprawdzamy w pierwszej kolejności:

Objaw	Co sprawdzamy
każde połączenie z poda kończy się błędem `Temporary failure in name resolution`	czy pody CoreDNS w ogóle działają (`kubectl get pods -n kube-system -l k8s-app=kube-dns`)
rozwiązują się nazwy usług, nie rozwiązują się domeny zewnętrzne	reguły forward w `Corefile`, dostęp sieciowy z podów CoreDNS na zewnątrz, `NetworkPolicy` blokująca ruch wyjściowy opisana na stronie Sieć klastra i polityki sieciowe

Objaw	Co sprawdzamy
rozwiązywanie nazw działa, ale bardzo wolno	wartość `ndots`, obciążenie samych podów CoreDNS (`kubectl top pod -n kube-system`)
działa z jednego węzła, nie działa z innego	reguły `NetworkPolicy` albo CNI ograniczające ruch do adresu CoreDNS z konkretnego węzła, temat rozwinięty na stronie Sieć klastra i polityki sieciowe
nazwa usługi rozwiązuje się na stary adres po jej odtworzeniu	krótkotrwałe opóźnienie propagacji — sprawdzamy ponownie po kilku sekundach, zanim uznamy to za awarię

Częste błędy

Odwołanie do usługi w innej przestrzeni nazw bez podania przestrzeni. Aplikacja działa poprawnie w środowisku, gdzie wszystko stoi w jednej przestrzeni nazw, i zaczyna zwracać błędy połączenia natychmiast po rozdzieleniu usług na osobne przestrzenie — naprawą jest dopisanie przestrzeni nazw do adresu albo pełnej nazwy FQDN.

Twarde zakodowanie adresu ClusterIP w konfiguracji aplikacji. Adres ClusterIP potrafi się zmienić przy odtworzeniu usługi (np. po jej usunięciu i utworzeniu na nowo) — jedynym stabilnym punktem odniesienia jest nazwa, nigdy sam adres.

Poleganie na zmiennych środowiskowych usług przy usługach tworzonych dynamicznie. Pod uruchomiony przed daną usługą nigdy nie dostanie jej zmiennych środowiskowych bez restartu — DNS jest jedynym mechanizmem odpornym na kolejność startu.

□□ Ćwiczenia

> Ćwiczenia zakładają dostęp do klastra testowego z co najmniej dwiema przestrzeniami nazw i jedną usługą w każdej z nich.

Ćwiczenie 1 — porównanie krótkiej i pełnej nazwy

```
kubectl run diagnostyka --rm -it --image=busybox:1.36 --restart=Never -n NazwaPrzestrzeni --
sh -c "nslookup NazwaUslugi; nslookup NazwaUslugi.inna-przestrzen"
```

Uruchamiamy pod w jednej przestrzeni nazw i próbujemy rozwiązać nazwę usługi z tej samej przestrzeni krótką formą, a potem tę samą krótką formę wobec usługi w innej przestrzeni.

Odpowiedź

Krótka nazwa rozwiązuje się poprawnie tylko dla usługi we własnej przestrzeni nazw — dla usługi w innej przestrzeni krótka forma zwraca `NXDOMAIN`, dopóki nie dopiszemy nazwy przestrzeni.

Pytania kontrolne

1. Dlaczego pierwsze połączenie z poda do zewnętrznego API bywa wolniejsze niż kolejne?

Odpowiedź

Przy domyślnym `ndots:5` resolwer najpierw próbuje doklejać kolejne domeny wyszukiwania z listy `search` do nazwy zewnętrznej, zanim spróbuje jej bez żadnego dopisku — każda nieudana próba to osobne zapytanie DNS z czasem oczekiwania na odpowiedź.

2. Co się stanie z aplikacją, jeśli wszystkie pody CoreDNS ulegną awarii jednocześnie?

Odpowiedź

Żadna nowa nazwa się nie rozwiąże — połączenia korzystające z już otwartych gniazd i zapamiętanych adresów IP mogą jeszcze działać, ale każde nowe połączenie po nazwie (w tym do innych usług w klastrze) zacznie kończyć się błędem rozwiązywania nazwy.

Powiązane: Service: jak ruch trafia do poda · Sieć klastra i polityki sieciowe · Diagnostyka: objaw, przyczyna, pierwsze kroki

Źródła: dokumentacja kubernetes.io, rozdziały „DNS for Services and Pods" i „Debugging DNS Resolution" · dokumentacja coredns.io

Ingress i Gateway API: wejście z zewnątrz

Poziom: średni **Dotyczy: Kubernetes 1.30+, Gateway API 1.5** **Czas: ~25 min**

Ingress i Gateway API opisują, jak ruch HTTP/HTTPS z zewnątrz klastra trafia do właściwej usługi na podstawie hosta i ścieżki — ta strona pokazuje oba mechanizmy i to, w którym kierunku dziś idzie ekosystem.

Skróty użyte na tej stronie (6)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
HTTP	Hypertext Transfer Protocol: protokół stron WWW
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
RBAC	Role-Based Access Control: uprawnienia nadawane rolom, a role ludziom
TLS	Transport Layer Security: szyfrowanie połączeń, np. w HTTPS
URI	ogólny adres zasobu w internecie, część adresu URL bez samej domeny

Fiszka

TYLKO ODCZYT

Wylistowanie obiektów Ingress: kubectl get ingress

```
kubectl get ingress -n NazwaPrzestrzeni
```

Sprawdzamy, jakie reguły wejściowe istnieją w danej przestrzeni nazw i jaki adres przydzielił im kontroler.

Przykład:

```
NAME         CLASS   HOSTS                    ADDRESS        PORTS     AGE
sklep-web    nginx   kolory.zielona.example   203.0.113.50   80, 443   30d
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

PLIK KONFIGURACYJNY

Utworzenie reguły Ingress po hoście i ścieżce

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: NazwaUslugi
  namespace: NazwaPrzestrzeni
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - host: NazwaHosta
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: NazwaUslugi
                port:
                  number: NumerPortu
```

Kierujemy ruch przychodzący na dany host i ścieżkę do wskazanej usługi — kontroler Ingress opisany niżej zamienia ten obiekt na konkretną konfigurację działającego serwera.

PLIK KONFIGURACYJNY

Dodanie TLS do Ingressa

```
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: NazwaUslugi
  namespace: NazwaPrzestrzeni
spec:
  ingressClassName: nginx
  tls:
    - hosts:
        - NazwaHosta
      secretName: NazwaSekretu
  rules:
    - host: NazwaHosta
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: NazwaUslugi
                port:
                  number: NumerPortu
```

Wskazujemy sekret typu `kubernetes.io/tls` zawierający certyfikat i klucz prywatny — sposób jego przygotowania opisuje strona Certyfikaty SSL: wystawienie, konwersja, weryfikacja.

TLS = Transport Layer Security: szyfrowanie połączeń, np. w HTTPS · **SSL** = Secure Sockets Layer: dawna nazwa szyfrowania, dziś formalnie TLS

TYLKO ODCZYT

Sprawdzenie dostępnych klas Ingress: kubectl get ingressclass

```
kubectl get ingressclass
```

Sprawdzamy, które kontrolery Ingress są zainstalowane w klastrze i którego z nich obiekt Ingress ma faktycznie użyć.

Przykład:

```
NAME    CONTROLLER             PARAMETERS   AGE
nginx   k8s.io/ingress-nginx   <none>       90d
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wylistowanie obiektów Gateway API: kubectl get gateway, httproute

```
kubectl get gateway,httproute -n NazwaPrzestrzeni
```

Sprawdzamy stan bramy i tras HTTP zdefiniowanych w Gateway API — odpowiednik dwóch pierwszych komend tej strony, tyle że dla nowszego modelu opisanego niżej.

Przykład:

```
NAME                                      CLASS   ADDRESS        PROGRAMMED   AGE
gateway.gateway.networking.k8s.io/main    nginx   203.0.113.50   True         12d

NAME                                          HOSTNAMES                    AGE
httproute.gateway.networking.k8s.io/sklep-api  ["kolory.zielona.example"]  12d
```

API = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **HTTP** = Hypertext Transfer Protocol: protokół stron WWW · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

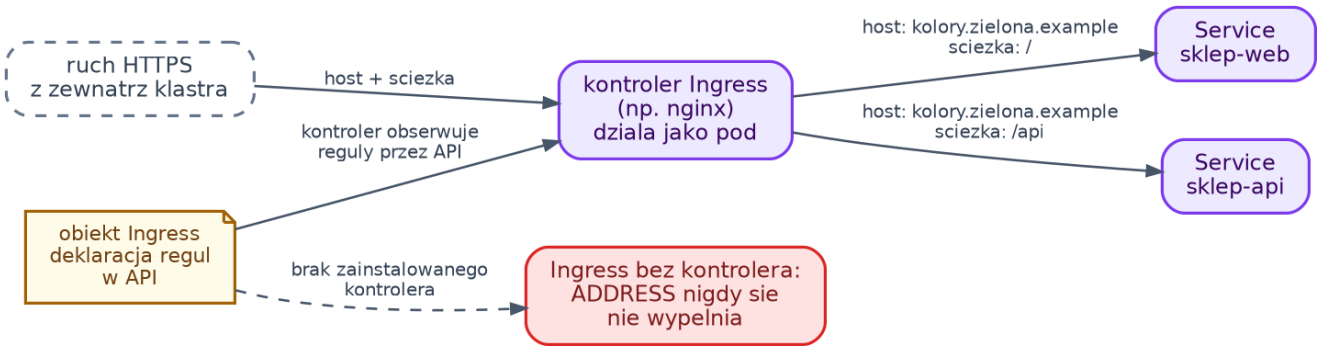
Z ŻYCIA WZIĘTE

O 15:20 nowa reguła Ingressa dla aplikacji raportowej wisi z pustym polem `ADDRESS` od dwóch godzin, a zespół deweloperski pyta Tomka, dlaczego strona wciąż zwraca błąd połączenia. Tomek sprawdza `kubectl get ingressclass` i widzi pustą listę, bo klaster testowy nigdy nie miał zainstalowanego żadnego kontrolera Ingress, tylko sam obiekt API. Po zainstalowaniu kontrolera Nginx Ingress pole adresu wypełnia się w ciągu kilku sekund, a reguła, która od początku była poprawna składniowo, wreszcie zaczyna kierować ruch.

Ingress i kontroler Ingress

> **Pojęcie: Ingress.** Obiekt Kubernetesa opisujący, jak ruch HTTP/HTTPS z zewnątrz klastra ma być kierowany do usług wewnątrz niego, na podstawie hosta i ścieżki żądania. Sam Ingress jest wyłącznie deklaracją zamiaru — zbiorem reguł zapisanym w API Kubernetesa, tak jak lista zakupów jest deklaracją zamiaru, a nie samymi zakupami.

To jest najczęstsze nieporozumienie wokół tego obiektu: utworzenie samego Ingressa w klastrze bez zainstalowanego kontrolera Ingress nie robi kompletnie nic. Obiekt zostaje zapisany w API, `kubectl get ingress` go pokazuje, ale żaden ruch nigdy do niego nie dotrze, bo nie ma nikogo, kto przełożyłby te reguły na działającą konfigurację serwera. Kontroler Ingress to osobny, samodzielnie instalowany komponent — najpopularniejszy to Nginx Ingress Controller, obok niego Traefik i Contour — który obserwuje obiekty Ingress przez API i na ich podstawie na bieżąco przepisuje konfigurację prawdziwego serwera proxy, opisanego pod kątem samej konfiguracji Nginksa na stronie Reverse proxy i równoważenie obciążenia. Bez kontrolera adres w polu `ADDRESS` z pierwszej komendy tej strony nigdy się nie pojawi.



Kontroler Ingress obserwuje reguły w API i kieruje ruch do właściwego Service po hoście i ścieżce, ale bez zainstalowanego kontrolera reguła zostaje martwą deklaracją (kliknięcie otwiera pełny rozmiar)

Reguły po hoście i po ścieżce, pathType

Reguła Ingressa dopasowuje żądanie po dwóch niezależnych kryteriach: nazwie hosta z nagłówka `Host` i ścieżce URI. Jeden obiekt Ingress może zawierać wiele reguł, kierując różne hosty albo różne ścieżki tego samego hosta do różnych usług — typowy sposób, w jaki kilka aplikacji dzieli jeden publiczny adres IP i jeden certyfikat.

> **Pojęcie: pathType.** Pole określające sposób dopasowania ścieżki. `Exact` wymaga dokładnej zgodności całej ścieżki, `Prefix` dopasowuje każdą ścieżkę zaczynającą się od podanego

> prefiksu podzielonego na segmenty (`/api` dopasowuje `/api/v1`, ale nie `/apix`), a `ImplementationSpecific` oddaje interpretację dopasowania konkretnemu kontrolerowi — ten ostatni wariant jest źródłem różnic w zachowaniu między różnymi kontrolerami przy tej samej regule i dlatego w nowych regułach sięgamy po `Prefix` albo `Exact`, gdzie tylko się da.

Adnotacje specyficzne dla kontrolera

Ponieważ sam obiekt Ingress w wersji stabilnej API opisuje tylko podstawowe dopasowanie hosta, ścieżki i TLS, każdy kontroler rozszerza jego możliwości przez adnotacje w `metadata.annotations`, których klucz zaczyna się od nazwy kontrolera (np. `nginx.ingress.kubernetes.io/`). Adnotacja `rewrite-target` z przykładu wyżej działa wyłącznie z kontrolerem Nginksa — ta sama adnotacja wpisana przy kontrolerze Traefik nie ma żadnego znaczenia, bo Traefik jej po prostu nie czyta. To jest praktyczny koszt modelu Ingress, do którego odpowiedzią jest Gateway API opisane niżej.

Gateway API

> **Pojęcie: Gateway API.** Następca modelu Ingress, rozwijany jako osobny projekt w ramach Kubernetesa, który dzieli konfigurację wejścia do klastra na kilka obiektów o osobnych właścicielach zamiast jednego płaskiego obiektu Ingress z adnotacjami. Problem, który Gateway API rozwiązuje, jest dokładnie ten opisany wyżej: brak wspólnego, przenośnego sposobu na wyrażenie funkcji wykraczających poza proste dopasowanie hosta i ścieżki (przepisywanie nagłówków, podział ruchu między wersjami, dopasowanie po metodzie HTTP) bez ucieczki w adnotacje właściwe tylko jednemu kontrolerowi.

Podział ról

Obiekt	Kto go tworzy	Co opisuje
`GatewayClass`	zespół platformy / dostawca infrastruktury	typ implementacji bramy dostępny w klastrze, odpowiednik `IngressClass`
`Gateway`	zespół platformy	konkretny punkt wejścia — adres, porty nasłuchu, certyfikaty TLS
`HTTPRoute`	zespół aplikacji	reguły routingu HTTP dla konkretnej aplikacji, dowiązane do istniejącej bramy

Ten podział ma konkretny cel praktyczny: zespół platformy zarządza `Gateway` i decyduje, kto na jakim porcie nasłuchuje i jakim certyfikatem, a zespoły aplikacyjne dostają uprawnienia wyłącznie

do `HTTPRoute` we własnej przestrzeni nazw, bez możliwości ingerencji w resztę konfiguracji brzegowej — coś, czego model Ingress z jednym płaskim obiektem nie rozdzielał w ogóle.

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: NazwaUslugi
  namespace: NazwaPrzestrzeni
spec:
  parentRefs:
    - name: main
      sectionName: https
  hostnames:
    - NazwaHosta
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /api
      backendRefs:
        - name: NazwaUslugi
          port: NumerPortu
```

Aktualny status wdrożeniowy

Stan na wrzesień 2026, po wydaniu Gateway API 1.5 (luty 2026): `HTTPRoute`, `GatewayClass`, `Gateway`, `ReferenceGrant`, `TLSRoute` i kilka nowszych funkcji (m.in. `ListenerSet`) znajdują się w kanale Standard, czyli traktowane są jako stabilne API. `GRPCRoute` i część funkcji routingu warstwy 4 wciąż rozwija się w kanale eksperymentalnym i bywa niekompletnie wspierana przez poszczególne implementacje — przed użyciem w produkcji warto sprawdzić w dokumentacji konkretnego kontrolera (Nginx Gateway Fabric, Cilium, Istio, Envoy Gateway), które funkcje faktycznie zaimplementował, bo status API i status wsparcia konkretnej implementacji to dwie różne rzeczy.

Sam obiekt Ingress formalnie nie został usunięty ani wycofany z Kubernetesa i istniejące wdrożenia dalej działają bez zmian, ale kierunek rozwoju ekosystemu przesunął się wyraźnie w stronę Gateway API — najbardziej dobitnym sygnałem jest zakończenie utrzymania projektu Ingress NGINX (najpopularniejszego kontrolera Ingress) z początkiem marca 2026, ogłoszone z rekomendacją migracji właśnie na Gateway API, ze względu na trudne do bezpiecznego załatania luki w mechanizmie adnotacji-snippetów tego kontrolera oraz brak wystarczających zasobów do dalszego utrzymania. Istniejące instalacje Ingress NGINX nadal działają, ale nie dostają już poprawek bezpieczeństwa. Dla nowych wdrożeń w klastrach, których dystrybucja i wybrany kontroler już wspierają Gateway API w wersji Standard, jest to dziś rozsądny punkt startowy — dla

klastrów zależnych od istniejących, mocno rozbudowanych konfiguracji Ingressa migracja jest raczej stopniowym procesem niż jednorazową zmianą, wspieranym narzędziem `ingress2gateway`.

Uważaj: to jest obszar, który zmienia się szybciej niż większość reszty API Kubernetesa — przed decyzją dla konkretnego klastra warto sprawdzić aktualny status wsparcia w dokumentacji zainstalowanego kontrolera, nie tylko status samego API Gateway.

Nasz przykład: kolorowy nginx

Regułę Ingress po hoście widzimy w praktyce na stronie k3s w laboratorium: pierwszy deploy, nginx w kolorach, gdzie wbudowany w k3s Traefik kieruje ruch z hosta `kolory.zielona.example` do usługi `kolorowy-nginx`. To ten sam kontroler i ten sam mechanizm dopasowania po nagłówku `Host`, opisany wyżej, tyle że zamiast rozbudowanej aplikacji stoi za nim prosty, kolorowy nginx, na którym łatwiej zobaczyć samą mechanikę routingu niż śledzić błąd w złożonej aplikacji. Fragment:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kolorowy-nginx
  namespace: kolory
spec:
  ingressClassName: traefik
  rules:
    - host: kolory.zielona.example
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: kolorowy-nginx
                port:
                  number: 80
```

Częste błędy

Utworzenie Ingressa bez zainstalowanego kontrolera. Obiekt istnieje w API, pole `ADDRESS` nigdy się nie wypełnia, żaden ruch nie dociera — pierwsza rzecz do sprawdzenia to `kubectl get`

`pods -n <przestrzeń kontrolera>` i `kubectl get ingressclass`.

pathType ImplementationSpecific traktowany jako przenośny. Ta sama reguła zachowuje się inaczej po zmianie kontrolera Ingress — jeśli reguła ma zachowywać się identycznie niezależnie od implementacji, wybieramy `Prefix` albo `Exact`.

Sekret TLS w złej przestrzeni nazw. Sekret wskazany w `tls.secretName` musi żyć w tej samej przestrzeni nazw co obiekt Ingress — sekret o tej samej nazwie w innej przestrzeni jest dla tego Ingressa niewidoczny, a kontroler zwraca certyfikat domyślny albo zgłasza błąd.

□□ Ćwiczenia

> Ćwiczenia zakładają zainstalowany kontroler Ingress (np. `ingress-nginx`) w klastrze testowym.

Ćwiczenie 1 — Ingress bez kontrolera

```
kubectl get ingressclass
kubectl delete ingressclass nginx
kubectl apply -f ingress-test.yaml
kubectl get ingress ingress-test -w
```

Usuwamy tymczasowo `IngressClass` (albo testujemy w klastrze, gdzie kontroler w ogóle nie jest zainstalowany) i obserwujemy, że pole `ADDRESS` nigdy się nie wypełnia.

Odpowiedź

Obiekt Ingress zostaje przyjęty przez API bez żadnego błędu składniowego — brak kontrolera nie jest błędem walidacji, tylko brakiem kogokolwiek, kto przetworzyłby regułę na działającą konfigurację. Dlatego pole adresu zostaje puste bezterminowo.

Pytania kontrolne

1. Dlaczego dwie instalacje tego samego Ingressa na dwóch różnych kontrolerach mogą zachowywać się różnie mimo identycznego pliku YAML?

Odpowiedź

Bo adnotacje specyficzne dla kontrolera i `pathType: ImplementationSpecific` są interpretowane przez każdy kontroler po swojemu — sam obiekt Ingress w wersji stabilnej opisuje tylko wspólny, minimalny zestaw pól.

2. Jaki problem realnie rozwiązuje rozdzielenie Gateway na `Gateway` i `HTTPRoute`, którego model Ingress nie miał?

Odpowiedź

Rozdzielenie uprawnień między zespołem platformy (zarządza punktem wejścia, portami, certyfikatami) a zespołami aplikacyjnymi (zarządzają wyłącznie regułami routingu własnej aplikacji) — w modelu Ingress oba te zakresy siedziały w jednym obiekcie, do którego RBAC nie potrafił rozdzielić dostępu na tym poziomie szczegółowości.

Powiązane: Reverse proxy i równoważenie obciążenia · Certyfikaty SSL: wystawienie, konwersja, weryfikacja · Service: jak ruch trafia do poda · RBAC i bezpieczeństwo klastra

Źródła: dokumentacja kubernetes.io, rozdziały „Ingress" i „Gateway API" · gateway-api.sigs.k8s.io, dokument „API Channels" i notatki wydania Gateway API 1.5 · ogłoszenie zakończenia utrzymania projektu Ingress NGINX (opensource.googleblog.com, luty 2026)

Sieć klastra i polityki sieciowe

Poziom: zaawansowany **Dotyczy: Kubernetes 1.30+** **Czas: ~25 min**

Model sieci Kubernetesa zakłada, że każdy pod widzi każdy inny pod bez żadnych przeszkód — ta strona pokazuje, kto realnie to zapewnia i jak świadomie to ograniczamy przez NetworkPolicy.

Skróty użyte na tej stronie (7)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CIDR	Classless Inter-Domain Routing: zapis podsieci z długością maski, na przykład 10.20.10.0/24
CNI	Container Network Interface: wtyczka, która daje sieć podom w Kubernetesie
DNS	Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
NAT	Network Address Translation: podmiana adresów IP na granicy sieci
TCP	Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

Fiszka

TYLKO ODCZYT

Sprawdzenie zainstalowanej wtyczki CNI: kubectl get pods

```
kubectl get pods -n kube-system -l k8s-app=cilium
```

Sprawdzamy, czy pody wtyczki sieciowej (tu: Cilium) działają na wszystkich węzłach — nazwa etykiety i przestrzeń nazw zależą od wybranej wtyczki, opisanej niżej.

Przykład:

```
NAME              READY   STATUS    RESTARTS   AGE
cilium-2k9wp      1/1     Running   0          40d
cilium-7xqm4      1/1     Running   0          40d
```

CNI = Container Network Interface: wtyczka, która daje sieć podom w Kubernetesie · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

PLIK KONFIGURACYJNY

Utworzenie polityki „domyślnie odrzucaj" dla przestrzeni nazw

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: domyslnie-odrzucaj
  namespace: NazwaPrzestrzeni
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

Blokujemy cały ruch wejściowy i wyjściowy dla wszystkich podów w przestrzeni nazw — punkt startowy, do którego dopisujemy jawne wyjątki, opisany niżej.

PLIK KONFIGURACYJNY

Zezwolenie na ruch tylko z określonych podów

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: NazwaUslugi-wejscie
  namespace: NazwaPrzestrzeni
spec:
  podSelector:
    matchLabels:
      app: NazwaAplikacji
  policyTypes:
```

```
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: sklep-front
      ports:
        - protocol: TCP
          port: NumerPortu
```

Otwieramy ruch przychodzący do podów z etykietą `app: NazwaAplikacji` wyłącznie od podów z etykietą `app: sklep-front`, na jednym konkretnym porcie.

TCP = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

PLIK KONFIGURACYJNY

Zezwolenie na ruch wyjściowy do konkretnej przestrzeni nazw

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: NazwaUslugi-wyjscie-dns
  namespace: NazwaPrzestrzeni
spec:
  podSelector:
    matchLabels:
      app: NazwaAplikacji
  policyTypes:
    - Egress
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: kube-system
      ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
```

```
      port: 53
```

Otwieramy ruch wyjściowy do przestrzeni nazw `kube-system` wyłącznie na portach DNS — bez tego wyjątku polityka „domyślnie odrzucaj” z pierwszej fiszki blokuje też rozwiązywanie nazw, opisane na stronie DNS w klastrze i wykrywanie usług.

DNS = Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP · **UDP** = User Datagram Protocol: szybki protokół sieciowy bez potwierdzeń · **TCP** = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

PLIK KONFIGURACYJNY

Zezwolenie na ruch wyjściowy do zewnętrznego bloku IP

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: NazwaUslugi-wyjscie-zewnetrzne
  namespace: NazwaPrzestrzeni
spec:
  podSelector:
    matchLabels:
      app: NazwaAplikacji
  policyTypes:
    - Egress
  egress:
    - to:
        - ipBlock:
            cidr: 203.0.113.0/24
      ports:
        - protocol: TCP
          port: 443
```

Otwieramy ruch wyjściowy do konkretnej publicznej podsieci — typowy wyjątek dla poda łączącego się z zewnętrznym API dostawcy.

IP = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **TCP** = Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia

TYLKO ODCZYT

Diagnostyka łączności między podami: kubectl exec i curl

```
kubectl exec NazwaPoda -n NazwaPrzestrzeni -- curl -sv --max-time 5
http://NazwaUslugi.NazwaPrzestrzeni.svc.cluster.local:NumerPortu
```

Sprawdzamy z wnętrza jednego poda, czy dociera do usługi innego poda — najszybszy sposób odróżnienia awarii sieci od awarii samej aplikacji.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 8:30 aplikacja katalogu produktów przestaje odpowiadać na zapytania DNS zaraz po tym, jak zespół bezpieczeństwa wdraża nową politykę ograniczającą ruch wejściowy do tej usługi. Tomek sprawdza `kubectl get networkpolicy` i widzi, że polityka wymienia w `policyTypes` zarówno `Ingress`, jak i `Egress`, mimo że w zamyśle miała ograniczyć tylko ruch przychodzący, a brak jakiejkolwiek reguły `egress` blokuje przy okazji cały ruch wyjściowy, łącznie z zapytaniami do CoreDNS. Po usunięciu `Egress` z `policyTypes` i dodaniu osobnej, jawnej polityki dla `Ingress` aplikacja znowu rozwiązuje nazwy i łączy się z bazą bez przeszkód.

Model sieci Kubernetesa

Pojęcie: model sieci Kubernetesa. Zestaw założeń, na których stoi cała reszta mechanizmów sieciowych klastra: każdy pod dostaje własny, unikalny w całym klastrze adres IP, i każdy pod może nawiązać połączenie z każdym innym podem po tym adresie bezpośrednio, bez translacji adresów (NAT) po drodze. Pod widzi adres drugiego poda dokładnie taki, jaki ten drugi pod sam o sobie zgłasza — inaczej niż w typowej sieci Dockera na pojedynczym hoście, gdzie kontenery na różnych hostach w ogóle się nie widzą bez ręcznie skonfigurowanego routingu albo publikacji portów.

To założenie upraszcza drastycznie wszystko, co dzieje się wyżej: Service, opisany na stronie Service: jak ruch trafia do poda, mógłby w ogóle nie istnieć jako osobna warstwa pośrednia, gdyby

nie to, że adresy podów są nietrwałe — sama łączność między nimi jest już zapewniona przez ten model, zanim jeszcze dojdzie do jakiegokolwiek Service.

Czym jest CNI

> **Pojęcie: CNI (Container Network Interface).** Specyfikacja i zestaw wtyczek odpowiedzialnych za faktyczne wprowadzenie w życie modelu sieci opisanego wyżej — to CNI nadaje podom adresy IP, tworzy interfejsy sieciowe wewnątrz nich i ustawia trasy routingu, tak żeby założenie „każdy pod widzi każdy" było prawdą na poziomie prawdziwych pakietów, a nie tylko dokumentacji. Kubernetes sam z siebie nie implementuje żadnej sieci — kubelet na każdym węźle wywołuje przy starcie poda skonfigurowaną wtyczkę CNI i to ona wykonuje całą pracę.

Bez zainstalowanej wtyczki CNI pody utkną trwale w stanie `Pending` albo `ContainerCreating` — kubelet nie ma jak nadać im adresu IP, więc kontener nigdy nie startuje, mimo że reszta klastra (API, scheduler) działa poprawnie.

Porównanie popularnych wtyczek CNI

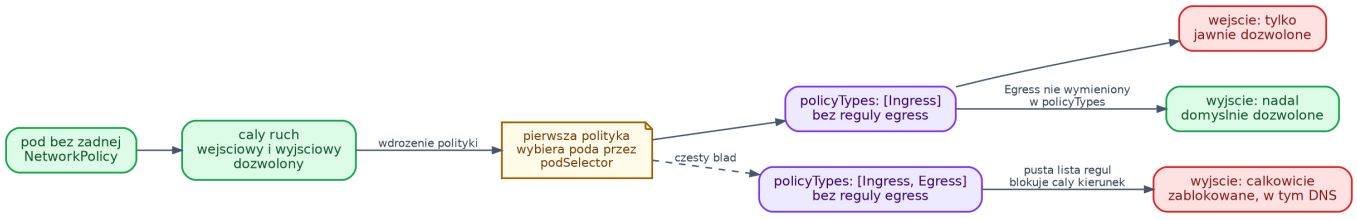
Wtyczka	Model	Charakterystyka
Flannel	prosty overlay (VXLAN)	najmniej skomplikowana konfiguracja, brak natywnego wsparcia dla `NetworkPolicy` bez dodatkowego komponentu
Calico	routing (BGP) albo overlay (VXLAN/IP-in-IP)	dojrzałe, pełne wsparcie `NetworkPolicy`, popularna w środowiskach on-premise z własnym routingiem
Cilium	eBPF, bez klasycznego `iptables`/`kube-proxy`	najwyższa wydajność przy dużej skali, rozszerzone polityki sieciowe (`CiliumNetworkPolicy`) wykraczające poza standard, wbudowana obserwowalność ruchu (Hubble)
Weave Net	overlay z szyfrowaniem między węzłami	prostota konfiguracji zbliżona do Flannela, szyfrowanie ruchu między węzłami bez dodatkowej warstwy

Uważaj: wtyczka CNI decyduje też o tym, czy `NetworkPolicy` w ogóle zadziała — sam obiekt `NetworkPolicy` jest częścią standardowego API Kubernetesa i przyjmuje się bez błędu niezależnie od zainstalowanej wtyczki, ale reguły w nim opisane egzekwuje dopiero wtyczka, która to potrafi. Klaster z samym Flannelem bez dodatkowego komponentu (np. Calico obok niego wyłącznie do polityk) przyjmie obiekt `NetworkPolicy` i będzie go całkowicie ignorował, bez żadnego ostrzeżenia.

NetworkPolicy

> **Pojęcie: NetworkPolicy.** Obiekt Kubernetesa opisujący, jaki ruch sieciowy wejściowy i wyjściowy jest dozwolony dla wybranych podów — jedyny wbudowany mechanizm ograniczania łączności między podami w standardowym API, w kontrze do domyślnej otwartości opisanej wyżej.

Uważaj: dopóki żadna polityka nie wybiera danego poda, cały ruch do niego i od niego jest dozwolony — to bezpośrednia konsekwencja modelu sieci z góry tej strony i kluczowa zasada, od której zaczyna się zrozumienie całego mechanizmu. W momencie, gdy pierwsza polityka zaczyna wybierać danego poda przez `podSelector` (nawet jeśli reguła w niej opisuje coś zupełnie innego, np. tylko ruch wejściowy), Kubernetes zaczyna domyślnie blokować cały pozostały ruch tego typu, którego dana polityka dotyczy (`Ingress` albo `Egress`), i przepuszcza tylko to, co jawnie opisano. To jest najczęstsza pułapka tego mechanizmu: dodanie jednej wąskiej polityki `Ingress` do poda, który wcześniej nie miał żadnej, potrafi przy okazji odciąć mu też cały ruch wyjściowy, jeśli ta sama polityka wymienia `Egress` w `policyTypes` bez żadnej reguły `egress` — pusta lista reguł przy obecnym typie polityki oznacza całkowitą blokadę tego kierunku.



Dopóki żadna polityka nie wybiera poda, ruch jest domyślnie otwarty w obu kierunkach, a wymienienie Egress w policyTypes bez żadnej reguły egress blokuje cały ruch wyjściowy, w tym DNS (kliknięcie otwiera pełny rozmiar)

Selektory: pody, przestrzenie nazw, bloki IP

Reguła `ingress` albo `egress` opisuje źródło (albo cel) ruchu przez jeden z trzech selektorów, które można też łączyć: `podSelector` (pody po etykietach, domyślnie w tej samej przestrzeni nazw), `namespaceSelector` (wszystkie pody w przestrzeniach nazw pasujących do etykiety) i `ipBlock` (zakres adresów IP w notacji CIDR, jedyny sposób na dopuszczenie ruchu spoza klastra). Połączenie `podSelector` i `namespaceSelector` w jednym wpisie `from` zawęża dopasowanie do konkretnych podów w konkretnych przestrzeniach nazw, a nie do sumy obu warunków osobno — to rozróżnienie (jeden wpis kontra dwa osobne wpisy w liście `from`) jest źródłem błędów, które przepuszczają więcej ruchu niż zamierzono.

Diagnostyka łączności między podami

Kolejność sprawdzania, gdy dwa pody, które powinny się widzieć, nie widzą się:

1. Sprawdzamy, czy oba pody w ogóle są w stanie `Running` i `Ready` (`kubectl get pods -o wide`).
2. Sprawdzamy łączność z wnętrza jednego poda do adresu IP drugiego bezpośrednio, z pominięciem DNS i Service, żeby wykluczyć te dwie warstwy z podejrzeń.
3. Sprawdzamy, czy istnieje `NetworkPolicy` obejmująca któregoś z podów (`kubectl get networkpolicy -n NazwaPrzestrzeni`) i czy jej reguły faktycznie dopuszczają ten konkretny ruch.
4. Sprawdzamy stan podów wtyczki CNI na obu węzłach, na których żyją nasze pody — awaria CNI na jednym konkretnym węźle tłumaczy asymetryczne problemy z łącznością (działa do jednych podów, nie działa do innych).

```
kubectl exec NazwaPoda -n NazwaPrzestrzeni -- nc -zv -w 3 10.244.2.7 NumerPortu
```

Sprawdzamy gołe połączenie TCP do adresu IP drugiego poda, z pominięciem nazwy DNS i Service — jeśli to się nie udaje, problem leży w samej sieci albo w `NetworkPolicy`, a nie w warstwach wyższych.

Częste błędy

Przekonanie, że NetworkPolicy działa bez wtyczki CNI, która ją wspiera. Obiekt zostaje przyjęty przez API bez błędu i sprawia wrażenie aktywnej ochrony, podczas gdy realnie żaden pakiet nie jest przez nią zatrzymywany — jedynym sposobem, żeby się upewnić, jest świadome sprawdzenie dokumentacji zainstalowanej wtyczki.

Polityka Ingress blokująca przy okazji Egress. Dodanie `Egress` do `policyTypes` bez choćby jednej reguły `egress` blokuje cały ruch wyjściowy poda, łącznie z zapytaniami DNS — objawem jest aplikacja, która przestaje cokolwiek rozwiązywać i łączyć się na zewnątrz zaraz po wdrożeniu polityki dotyczącej z pozoru tylko ruchu wejściowego.

Brak wyjątku na DNS przy polityce „domyślnie odrzucaj”. Klasyczny efekt uboczny pierwszej fiszki tej strony — bez osobnej reguły otwierającej port 53 do `kube-system` cała przestrzeń nazw traci możliwość rozwiązywania jakichkolwiek nazw, włącznie z nazwami usług we własnej przestrzeni.

□□ Ćwiczenia

> Ćwiczenia zakładają klaster testowy z wtyczką CNI wspierającą `NetworkPolicy` (Calico albo Cilium) i dwa pody w tej samej przestrzeni nazw.

Ćwiczenie 1 — efekt polityki domyślnie odrzucaj

```
kubectl exec pod-a -n NazwaPrzestrzeni -- curl -sv --max-time 3 http://pod-b-ip:80
kubectl apply -f domyslnie-odrzucaj.yaml
kubectl exec pod-a -n NazwaPrzestrzeni -- curl -sv --max-time 3 http://pod-b-ip:80
```

Sprawdzamy łączność między dwoma podami przed i po wdrożeniu polityki „domyślnie odrzucaj” dla całej przestrzeni nazw.

Odpowiedź

Pierwsze połączenie się udaje — bez żadnej polityki ruch jest domyślnie dozwolony. Po wdrożeniu polityki drugie połączenie kończy się przekroczeniem czasu oczekiwania, bo polityka z pustą listą reguł blokuje cały ruch tego typu dla wszystkich podów w przestrzeni nazw.

Pytania kontrolne

1. Dlaczego pod bez żadnej dedykowanej dla niego polityki NetworkPolicy może mimo to stracić łączność po wdrożeniu polityki gdzieś indziej w tej samej przestrzeni nazw?

Odpowiedź

Jeśli ta polityka wybiera pody przez pusty `podSelector: {}`, obejmuje wszystkie pody w przestrzeni nazw, nie tylko te, dla których pisaliśmy ją w zamyśle.

2. Czym różni się blokada ruchu przez NetworkPolicy od blokady przez firewall systemu operacyjnego na węźle?

Odpowiedź

NetworkPolicy działa na poziomie tożsamości poda (etykiety, przestrzeń nazw) i jest egzekwowana przez wtyczkę CNI, niezależnie od tego, na którym węźle pod aktualnie żyje — reguła podąża za podem przy jego przeniesieniu. Firewall na węźle operuje na adresach IP i interfejsach konkretnej maszyny, bez wiedzy o tym, czym jest pod ani do którego namespace'u należy.

Powiązane: Service: jak ruch trafia do poda · DNS w klastrze i wykrywanie usług · Diagnostyka: objaw, przyczyna, pierwsze kroki

Źródła: dokumentacja kubernetes.io, rozdziały „Cluster Networking", „Network Policies" i „Network Plugins" · dokumentacja projektów Cilium, Calico i Flannel