Podstawy i architektura

Po co orkiestracja, jak zbudowany jest klaster i czym są obiekty.

- Po co Kubernetes, skoro mamy Dockera
- Architektura klastra: płaszczyzna sterowania i węzły
- Obiekty, manifesty i przestrzenie nazw

Po co Kubernetes, skoro mamy Dockera

Poziom: podstawy
Dotyczy: koncepcji niezależnych od wersji, przykłady z Kubernetes 1.37 **Czas: ~20 min**

Po przeczytaniu tej strony wiemy, jaki konkretny problem rozwiązuje orkiestracja kontenerów, czym różni się podejście deklaratywne od imperatywnego i kiedy sięgnięcie po Kubernetesa jest przerostem formy nad treścią.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
CPU	Central Processing Unit: procesor
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Z życia wzięte

Z ŻYCIA WZIĘTE

O 9:20 na serwerze app01 pada proces jednej z trzech replik wewnętrznej aplikacji Javy, a Tomek zauważa to dopiero po zgłoszeniu od zespołu sprzedaży, że raporty ładują się wolniej niż zwykle. Ręczne uruchomienie brakującej instancji zajmuje mu pięć minut, ale przez te pięć minut nikt nic nie wiedział ani nie reagował automatycznie. Po tym incydencie Tomek zaczyna czytać o Kubernetesie właśnie dlatego, że chce mechanizmu, który sam zauważy brakującą replikę i sam ją odtworzy, zanim ktokolwiek zgłosi wolne raporty.

Docker wystarcza, dopóki kontenerów jest mało

Docker (opisany od podstaw na stronie Obraz, kontener, warstwa, wolumen: cztery pojęcia) świetnie odpowiada na pytanie „jak uruchomić jeden kontener na jednym hoście, żeby był odizolowany i powtarzalny". Nie odpowiada na pytanie „co zrobić, żeby sto takich kontenerów rozrzuconych po dziesięciu hostach działało jako jedna spójna całość, przetrwało awarię pojedynczej maszyny i dało się aktualizować bez przerwy w działaniu". To drugie pytanie to już nie jest pytanie o pojedynczy kontener, tylko o system złożony z wielu kontenerów i wielu maszyn naraz — i właśnie ten poziom problemu adresuje orkiestracja.

> **Pojęcie: orkiestracja kontenerów.** Warstwa oprogramowania, która zarządza cyklem życia wielu kontenerów rozłożonych na wielu maszynach jako jednym systemem — decyduje, na którym hoście uruchomić który kontener, pilnuje, żeby zadeklarowana liczba instancji faktycznie działała, wymienia niedziałające instancje na nowe, aktualizuje wersje bez przerywania ruchu i pozwala kontenerom znajdować się nawzajem, mimo że każdy restart może zmienić im adres IP. Docker (albo dowolny inny silnik kontenerów) pozostaje pod spodem — orkiestrator nie zastępuje go, tylko nim zarządza na większą skalę.

Co przestaje działać, gdy kontenerów jest dużo

Poniższe pięć problemów pojawia się dopiero przy skali — na jednym hoście z kilkoma kontenerami każdy z nich da się rozwiązać ręcznie i nikt tego nie zauważy. Przy stu kontenerach na dziesięciu hostach ręczne rozwiązywanie każdego z nich zaczyna zajmować więcej czasu niż sama praca, którą te kontenery mają wykonywać.

Rozmieszczenie kontenerów po hostach

Który z dziesięciu hostów ma dość wolnej pamięci i CPU, żeby przyjąć kolejny kontener? Ręczne śledzenie zajętości zasobów na każdym hoście z osobna i decydowanie „ten kontener wystawiam na host numer 7" skaluje się do kilku hostów, a przy większej liczbie zamienia się w arkusz kalkulacyjny, który i tak szybko przestaje odzwierciedlać rzeczywistość, bo ktoś inny w międzyczasie uruchomił coś ręcznie na hoście numer 7.

Restarty po awarii

`docker run --restart unless-stopped` (opisane na stronie Codzienne komendy) odtwarza kontener, który padł na tym samym hoście. Nie pomaga, gdy padł cały host — kontenery, które na nim działały, nie wstaną same na żadnej innej maszynie, bo nic nie obserwuje stanu całego klastra hostów, tylko stan pojedynczego demona Dockera na pojedynczej maszynie.

Aktualizacje bez przerwy w działaniu

Podmiana obrazu na nowszą wersję w prostym scenariuszu oznacza `docker stop` starego kontenera i `docker run` nowego — a między tymi dwiema komendami usługa jest niedostępna. Przy jednym kontenerze to ułamek sekundy, którego nikt nie zauważy. Przy usłudze złożonej z dwudziestu replik za load balancerem chcemy podmieniać je pojedynczo albo małymi grupami, tak żeby w każdej chwili część starych i część nowych instancji obsługiwała ruch — ręczna koordynacja takiej sekwencji dla dwudziestu kontenerów na kilku hostach jest źródłem błędów, nie procedurą do wykonywania co tydzień.

Wykrywanie usług

Kontener przy każdym restarcie może dostać nowy adres IP wewnątrz sieci dockerowej. Aplikacja, która łączy się z bazą danych po zapisanym na sztywno adresie IP kontenera, przestaje działać po pierwszym restarcie tej bazy. Potrzebny jest mechanizm, który daje stały punkt odniesienia — nazwę albo adres, który nie zmienia się mimo że konkretne instancje za nim się wymieniają.

Skalowanie w górę i w dół

Wzrost ruchu o godzinie szczytu wymaga uruchomienia kolejnych kopii aplikacji, a spadek ruchu w nocy pozwala je zwolnić i oddać zasoby innym zadaniom. Zrobienie tego ręcznie wymaga kogoś, kto o trzeciej w nocy zauważy wzrost obciążenia i wyda komendę — albo automatyzacji, która obserwuje metryki i sama decyduje, ile kopii ma teraz działać.

Kubernetes nie wymyśla żadnego z tych problemów — one istnieją niezależnie od tego, czy używamy orkiestratora. Wymyśla natomiast spójny sposób opisania pożądanego rezultatu i mechanizm, który sam dba o to, żeby rzeczywistość ten opis odzwierciedlała. To jest właśnie różnica między modelem imperatywnym a deklaratywnym.

Model deklaratywny kontra imperatywny

> **Pojęcie: model imperatywny.** Podejście, w którym mówimy systemowi dokładnie jakie kroki ma wykonać, w jakiej kolejności — `docker run`, potem `docker stop`, potem `docker run` z nowym obrazem. System wykonuje polecenia jedno po drugim i nie pamięta, po co je wykonuje ani jaki stan miał w efekcie osiągnąć. Jeśli coś między krokami pójdzie nie tak, nikt tego nie naprawi — trzeba wrócić i wydać kolejne polecenie ręcznie.

> **Pojęcie: model deklaratywny.** Podejście, w którym opisujemy pożądany stan końcowy — „chcę, żeby zawsze działały trzy kopie tego kontenera, każda z tym obrazem, każda z takim limitem pamięci" — i zostawiamy systemowi decyzję, jakich kroków użyć, żeby ten stan osiągnąć i, co ważniejsze, utrzymać w czasie. Nie mówimy „uruchom kontener", tylko „niech ten kontener istnieje".

Różnica ujawnia się w momencie awarii. W modelu imperatywnym awaria hosta z jednym z trzech kontenerów oznacza, że mamy teraz dwa kontenery i nic samo z siebie tego nie zmieni, dopóki ktoś nie zauważy i nie wyda kolejnego `docker run`. W modelu deklaratywnym system bez przerwy porównuje to, co zadeklarowaliśmy (trzy kopie), z tym, co faktycznie obserwuje (dwie kopie) — i sam, bez naszego udziału, uruchamia trzecią. Ten mechanizm porównywania i korygowania ma swoją nazwę i jest sednem tego, jak działa Kubernetes.

Pętla uzgadniania (reconciliation loop)

> **Pojęcie: pętla uzgadniania (reconciliation loop).** Mechanizm pracujący bez przerwy w tle, który dla każdego obiektu w klastrze porównuje stan pożądany (to, co zadeklarowaliśmy w `spec`) ze stanem aktualnym (to, co faktycznie się dzieje, widoczne w `status`) i w razie rozbieżności wykonuje akcje mające tę rozbieżność zlikwidować — po czym sprawdza ponownie, w kółko, bez końca. Nie jest to jednorazowy skrypt uruchamiany raz na jakiś czas, tylko ciągła, nieustająca kontrola.

Prześledźmy to na konkretnym przykładzie. Deklarujemy obiekt Deployment (opisany szczegółowo na stronie Deployment, ReplicaSet i aktualizacje bez przerwy) z polem `replicas: 3` i obrazem

`nginx:1.27`. Kontroler odpowiedzialny za ten obiekt działa mniej więcej tak:



Kontroler w pętli porównuje pożądany stan z pola spec z aktualnym stanem z pola status i przy rozbieżności tworzy brakujący pod w klastrze (kliknięcie otwiera pełny rozmiar)

1. Kontroler odczytuje `spec`: pożądany stan to trzy działające pody z obrazem `nginx:1.27`.
2. Kontroler odczytuje `status`: ile podów z tym obrazem faktycznie teraz działa. Załóżmy, że trzy — zgadza się ze `spec`, kontroler nic nie robi i czeka na kolejne zdarzenie.
3. Węzeł, na którym działał jeden z trzech podów, przestaje odpowiadać (awaria sprzętu, wyłączenie maszyny, cokolwiek). Po upływie skonfigurowanego czasu ten pod znika ze stanu aktualnego.
4. Przy kolejnym przebiegu pętli kontroler widzi rozbieżność: `spec` mówi trzy, `status` pokazuje dwa.
5. Kontroler tworzy nowy obiekt poda z tym samym obrazem. Ten nowy pod trafia do kolejki nieprzydzielonych podów, skąd odbiera go planista (`kube-scheduler`, opisany na stronie Architektura klastra: płaszczyzna sterowania i węzły) i przydziela do jednego z pozostałych zdrowych węzłów.
6. Nowy pod startuje, zaczyna działać, `status` znów pokazuje trzy — rozbieżność zniknęła, pętla wraca do bezczynnego oczekiwania na kolejną zmianę.

Nikt nie zalogował się na żaden serwer, nikt nie wydał ręcznej komendy — cały ten ciąg zdarzeń wynikł z jednego zdania zapisanego wcześniej w manifeście: „mają być trzy". Ta sama pętla działa identycznie, gdy zamiast awarii zmienimy `replicas: 3` na `replicas: 5` (kontroler dokłada dwa pody) albo obraz na `nginx:1.28` (kontroler stopniowo wymienia stare pody na nowe — to już mechanizm aktualizacji, rozwinięty na stronie o Deploymentach). Jeden mechanizm uzgadniania obsługuje skalowanie, samoleczenie po awarii i aktualizacje wersji — to nie trzy osobne funkcje, tylko trzy przejawy tej samej zasady.

Kiedy Kubernetes to przerost formy

Kubernetes rozwiązuje problemy skali — a rozwiązywanie ich ma swoją cenę. Płaszczyzna sterowania (control plane), o której szczegółowo mówi kolejna strona, to samo w sobie kilka dodatkowych usług do utrzymania, zrozumienia i aktualizowania, zanim jeszcze uruchomimy

pierwszy kontener z właściwą aplikacją.

Jedna aplikacja na jednym serwerze, bez wymogu przetrwania awarii tego serwera, bez potrzeby skalowania w poziomie, z aktualizacjami raz na tydzień w oknie serwisowym — to scenariusz, w którym `docker compose up -d` (temat strony Docker Compose) daje dokładnie to, co potrzebne, jednym plikiem YAML i bez uczenia się nowego słownika pojęć, nowego API i nowego narzędzia CLI. Uruchamianie klastra Kubernetes pod tak prosty przypadek to instalowanie systemu kolejkowego w sklepie, do którego wchodzi jeden klient dziennie — działa, ale koszt utrzymania przewyższa problem, który miał rozwiązać.

Granica nie jest sztywna i nie przebiega dokładnie po liczbie kontenerów — bierzemy pod uwagę też to, czy zespół już zna Kubernetesa (koszt nauki jest jednorazowy, nie ponosi się go przy każdym kolejnym projekcie), czy firma ma już działający klaster, na którym wystarczy dołożyć kolejną aplikację (wtedy krańcowy koszt jest bliski zeru), i czy w planach jest wzrost, który i tak zmusi nas do migracji za pół roku. Reguła praktyczna: jeśli odpowiedź na pytanie „co się stanie, gdy ten jeden serwer padnie" brzmi „poczekamy do rana i podniesiemy ręcznie", Kubernetes nie jest tu potrzebny.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Wyjaśniamy, dlaczego `docker run --restart unless-stopped` nie rozwiązuje problemu awarii całego hosta, na którym kontener działał.
2. Opisujemy różnicę między poleceniem „uruchom pięć kontenerów tym obrazem" a deklaracją „chcę, żeby zawsze działało pięć kontenerów tym obrazem" — i wskazujemy, co się dzieje w każdym z tych podejść, gdy jeden z kontenerów ulegnie awarii.
3. Na własnych słowach opisujemy pętlę uzgadniania dla scenariusza: zadeklarowaliśmy `replicas: 4`, a administrator przez pomyłkę ręcznie usunął jeden z czterech podów poleceniem `kubectl delete pod`.
4. Podajemy przykład aplikacji z własnego środowiska, dla której `docker compose` w zupełności wystarcza, i uzasadniamy dlaczego.

Powiązane: Obraz, kontener, warstwa, wolumen: cztery pojęcia · Architektura klastra: płaszczyzna sterowania i węzły · Deployment, ReplicaSet i aktualizacje bez przerwy

Źródła: dokumentacja Kubernetes, rozdział „Kubernetes Components" i „What is Kubernetes" (kubernetes.io/docs/concepts/overview/)

Architektura klastra: płaszczyzna sterowania i węzły

Poziom: podstawy
Dotyczy: Kubernetes 1.37, containerd jako środowisko uruchomieniowe **Czas: ~25 min**

Po przeczytaniu tej strony wiemy, z jakich procesów składa się klaster, co robi każdy z nich, co się dzieje, gdy któryś z nich padnie, i jaką drogę przechodzi żądanie od `kubectl apply` do działającego kontenera.

Skróty użyte na tej stronie (11)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CPU	Central Processing Unit: procesor
CRI	Container Runtime Interface: standard, którym Kubernetes rozmawia z silnikiem kontenerów
etcd	etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft
FAQ	Frequently Asked Questions: zbiór najczęściej zadawanych pytań z odpowiedziami
HTTPS	HTTP zabezpieczony szyfrowaniem TLS
JSON	JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami
OCI	Open Container Initiative: standard formatu obrazów kontenerów i rejestrów, z którego korzystają też rejestry wykresów Helm (adresy zaczynające się od oci://)
RBAC	Role-Based Access Control: uprawnienia nadawane rolom, a role ludziom
REST	sposób budowania API przez zwykłe żądania HTTP (GET, POST) zamiast osobnego protokołu
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Fiszka

TYLKO ODCZYT

Wylistowanie węzłów klastra z rolami i wersjami: kubectl get nodes -o wide

```
kubectl get nodes -o wide
```

Zwracamy listę węzłów należących do klastra wraz z ich rolą, statusem gotowości, wersją `kubelet` i używanym środowiskiem uruchomieniowym kontenerów.

Przykład:

```
NAME    STATUS   ROLES           AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-
IMAGE             KERNEL-VERSION       CONTAINER-RUNTIME
k3s01   Ready    control-plane   62d   v1.37.1   10.20.10.44   <none>        Ubuntu
22.04.4 LTS   5.15.0-131-generic   containerd://1.7.22
k3s02   Ready    <none>          62d   v1.37.1   10.20.10.67   <none>        Ubuntu
22.04.4 LTS   5.15.0-131-generic   containerd://1.7.22
k3s03   Ready    <none>          62d   v1.37.1   10.20.10.68   <none>        Ubuntu
22.04.4 LTS   5.15.0-131-generic   containerd://1.7.22
```

> Laboratorium Zielonej to k3s: węzły `k3s01` (10.20.10.44), `k3s02` (10.20.10.67) i `k3s03` (10.20.10.68). Sposób budowy klastra przez `kubeadm` opisujemy jako ogólny, dlatego listy podów niżej pokazują układ typowy dla `kubeadm`, z nazwami węzłów z naszego laboratorium.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wylistowanie podów płaszczyzny sterowania: kubectl get pods -n kube-system

```
kubectl get pods -n kube-system -o wide
```

Na klastrze postawionym przez `kubeadm` większość składników płaszczyzny sterowania działa jako zwykłe pody, uruchamiane bezpośrednio przez `kubelet` z manifestów statycznych, a nie przez planistę — dzięki temu widzimy je tą samą komendą, którą sprawdzalibyśmy dowolny inny pod.

Przykład:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
etcd-k3s01	1/1	Running	0	62d	10.20.10.44	k3s01
kube-apiserver-k3s01	1/1	Running	0	62d	10.20.10.44	k3s01
kube-controller-manager-k3s01	1/1	Running	2	62d	10.20.10.44	k3s01
kube-scheduler-k3s01	1/1	Running	1	62d	10.20.10.44	k3s01
kube-proxy-4f2xz	1/1	Running	0	62d	10.20.10.44	k3s01
kube-proxy-9jd7q	1/1	Running	0	62d	10.20.10.67	k3s02
kube-proxy-p2kmn	1/1	Running	0	62d	10.20.10.68	k3s03
coredns-7db6d8ff4d-2xqvz	1/1	Running	0	62d	10.244.0.5	k3s01
coredns-7db6d8ff4d-8mnzs	1/1	Running	0	62d	10.244.0.6	k3s01

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 14:40 Tomek wykonuje `kubectl apply` na klastrze testowym i przez dwie minuty nic się nie dzieje, więc zaczyna podejrzewać zawieszony proces. Sprawdza `kubectl get pods -n kube-system -o wide` i widzi, że `kube-scheduler` ma status `CrashLoopBackOff` po nieudanej aktualizacji obrazu na węźle sterującym. Nowy pod wisi w fazie `Pending`, bo nikt nie decyduje, na który węzeł roboczy go przydzielić, a dopiero po przywróceniu poprzedniej wersji obrazu planisty pod dostaje przydział i rusza.

Klaster, węzeł sterujący i węzeł roboczy

Pojęcie: klaster. Zbiór maszyn (fizycznych albo wirtualnych), które razem tworzą jeden Kubernetesowy system — z zewnątrz widziany i zarządzany jako jedna całość przez jedno API, mimo że w środku pracuje wiele niezależnych procesów rozłożonych na wielu hostach.

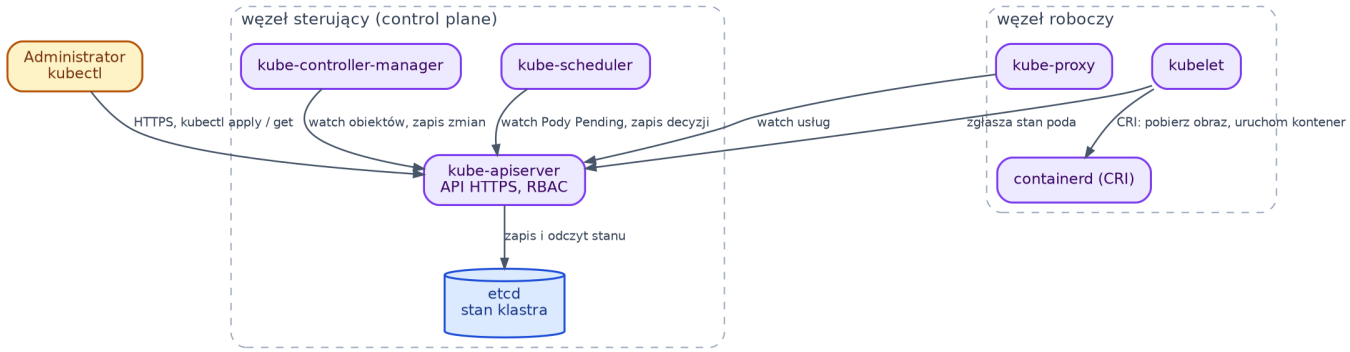
> **Pojęcie: węzeł (node).** Pojedyncza maszyna należąca do klastra. Kubernetes rozróżnia dwie role węzła, które mogą, ale nie muszą, być rozdzielone na osobne maszyny.

Węzeł sterujący (control-plane node) uruchamia procesy odpowiedzialne za zarządzanie całym klastrem — podejmowanie decyzji, przechowywanie stanu, reagowanie na zmiany. Na małych klastrach testowych (patrz sekcja o klastrach lokalnych na stronie kubectl: narzędzie codzienne) bywa jednocześnie jedynym węzłem, na którym da się też uruchamiać zwykłe aplikacje; na produkcji zwykle jest odseparowany i nie przyjmuje obciążenia aplikacyjnego.

Węzeł roboczy (worker node) uruchamia rzeczywiste kontenery naszych aplikacji. Nie podejmuje samodzielnie decyzji o tym, co ma uruchomić — czeka, aż płaszczyzna sterowania mu to przydzieli, i wykonuje.

Składniki płaszczyzny sterowania

> **Pojęcie: płaszczyzna sterowania (control plane).** Zestaw procesów pełniących rolę mózgu klastra — przyjmują żądania, przechowują pożądany i aktualny stan każdego obiektu, podejmują decyzje o przydziale zasobów i uruchamiają pętlę uzgadniania opisaną na stronie Po co Kubernetes, skoro mamy Dockera. Żaden z tych procesów sam nie uruchamia kontenerów naszej aplikacji — tym zajmują się wyłącznie węzły robocze.



Węzeł sterujący trzyma apiserver, etcd, scheduler i controller-manager, a węzeł roboczy kubelet, kube-proxy i containerd, przy czym wszystkie procesy rozmawiają wyłącznie przez apiserver (kliknięcie otwiera pełny rozmiar)

kube-apiserver

Jedyny punkt wejścia do klastra. Wszystko, co dzieje się w Kubernetesie — `kubectl apply`, odczyt stanu przez `kubectl get`, komunikacja kontrolerów między sobą, zgłoszenia `kubelet` o stanie podów — przechodzi przez ten jeden proces, wystawiony jako REST API po HTTPS. `kube-apiserver` uwierzytelnia żądanie (certyfikat klienta, token, moduł zewnętrzny), sprawdza autoryzację (RBAC), przepuszcza żądanie przez kontrolery przyjęcia (admission controllers, w tym walidację standardów bezpieczeństwa poda, patrz niżej) i dopiero wtedy zapisuje albo odczytuje dane z `etcd`. Jest bezstanowy — nie trzyma niczego wyłącznie we własnej pamięci — dlatego można go swobodnie skalować poziomo za load balancerem.

Co się dzieje, gdy padnie: żadne nowe żądanie nie przejdzie — `kubectl` przestaje odpowiadać, kontrolery nie widzą zmian i nie mogą reagować na awarie, `kubelet` na każdym węźle traci możliwość zgłaszania stanu i pobierania nowych przydziałów. Pody, które już działały w momencie awarii, nie znikają — `kubelet` na każdym węźle kontynuuje pilnowanie ich samodzielnie, na podstawie ostatniego znanego przydziału — ale klaster przestaje reagować na cokolwiek nowego, dopóki `kube-apiserver` nie wróci.

etcd

> **Pojęcie: etcd.** Rozproszona baza danych typu klucz-wartość, w której `kube-apiserver` przechowuje kompletny stan klastra — każdy obiekt, jego `spec` i `status`, w jednym miejscu. Nic w Kubernetesie nie trzyma stanu poza `etcd` — nawet planista i kontrolery przy każdym restarcie odtwarzają swoją wiedzę o świecie, czytając ją z powrotem z `etcd` przez `kube-apiserver`.

Co się dzieje, gdy padnie: `kube-apiserver` przestaje móc odczytywać i zapisywać stan, więc w praktyce klaster przestaje przyjmować jakiekolwiek zmiany, nawet jeśli sam `kube-apiserver` formalnie odpowiada na żądania. Utrata wszystkich kopii `etcd` bez kopii zapasowej oznacza utratę całej wiedzy o tym, co powinno działać w klastrze — kontenery na węzłach fizycznie nadal pracują, ale nikt już nie wie, że powinny.

kube-scheduler

> **Pojęcie: planista (scheduler).** Proces, który obserwuje nowo utworzone pody bez przypisanego węzła (pole `nodeName` puste) i decyduje, na którym węźle każdy z nich ma zostać uruchomiony. Decyzja bierze pod uwagę dostępne zasoby węzła (ile CPU i pamięci już jest zarezerwowane przez inne pody), reguły przypisania (affinity/anti-affinity), taints i tolerations oraz inne ograniczenia zadeklarowane w manifeście. Planista nie uruchamia niczego sam — jedynie zapisuje w `etcd` decyzję „ten pod należy do tego węzła", a wykonanie przejmuje `kubelet` na wskazanym węźle.

Co się dzieje, gdy padnie: pody już przypisane do węzłów działają dalej bez zmian. Nowe pody utworzone w tym czasie zostają w fazie `Pending` — istnieją jako obiekty w `etcd`, ale nikt nie decyduje, gdzie mają trafić, dopóki planista nie wróci i nie nadrobi zaległości.

kube-controller-manager

Pojedynczy proces, wewnątrz którego działa w rzeczywistości kilkanaście niezależnych kontrolerów — kontroler węzłów (zauważa, gdy węzeł przestaje odpowiadać), kontroler ReplicaSet (pilnuje liczby replik, patrz strona Deployment, ReplicaSet i aktualizacje bez przerwy), kontroler zadań (`Job` / `CronJob`) i wiele innych. Każdy z nich realizuje osobną pętlę uzgadniania dla swojego rodzaju obiektu, ale wszystkie działają w ramach tego samego procesu.

Co się dzieje, gdy padnie: żadna pętla uzgadniania nie działa. Istniejące pody nadal pracują, ale awaria węzła nie zostanie wykryta i skompensowana nowym podem gdzie indziej, ręczne skalowanie przez zmianę `replicas` w `etcd` (teoretycznie, przez bezpośredni zapis) nie zostanie zrealizowane, a zakończone zadania `Job` nie zostaną posprzątane.

cloud-controller-manager

Składnik opcjonalny, obecny wyłącznie na klastrach uruchomionych w chmurze publicznej albo zintegrowanych z konkretnym dostawcą infrastruktury. Odseparowuje logikę specyficzną dla dostawcy (tworzenie load balancerów przy typie usługi `LoadBalancer`, przydzielanie woluminów blokowych, oznaczanie węzła jako niedostępnego, gdy odpowiadająca mu maszyna wirtualna zniknęła z chmury) od reszty `kube-controller-manager`, która ma pozostać identyczna niezależnie od tego, gdzie klaster działa. Klaster postawiony samodzielnie na gołym metalu (`kubeadm` na własnych serwerach) zwykle w ogóle go nie uruchamia.

Co się dzieje, gdy padnie: funkcje integracji z chmurą (nowe load balancery, nowe woluminy) przestają się realizować, reszta klastra działa bez zmian.

Składniki węzła

kubelet

> **Pojęcie: kubelet.** Agent działający na każdym węźle (sterującym i roboczym), który utrzymuje stałe połączenie z `kube-apiserver`, obserwuje, jakie pody zostały przypisane do jego węzła, i dba o to, żeby faktycznie działały — pobiera obrazy, uruchamia kontenery przez

> środowisko uruchomieniowe, wykonuje sondy (patrz strona Pod: najmniejsza jednostka), restartuje kontenery zgodnie z `restartPolicy` i zgłasza z powrotem do `kube-apiserver` bieżący stan każdego poda. `kubelet` to jedyny składnik płaszczyzny sterowania, który ma bezpośredni kontakt z rzeczywistością na danym węźle — wszystko inne działa na poziomie deklaracji w `etcd`.

kube-proxy

Proces sieciowy działający na każdym węźle, odpowiedzialny za realizację obiektów `Service` — utrzymuje reguły przekierowania ruchu (przez `iptables` albo `IPVS`, w zależności od trybu) tak, żeby ruch skierowany na stały adres usługi trafiał do jednego z aktualnie działających podów za tą usługą. Szczegóły mechanizmu sieciowego i typów usług rozwija część poświęcona sieci klastra — tutaj wystarczy wiedzieć, że to `kube-proxy` sprawia, że adres usługi działa niezależnie od tego, które konkretnie pody aktualnie za nią stoją.

Środowisko uruchomieniowe kontenerów i los Dockershim

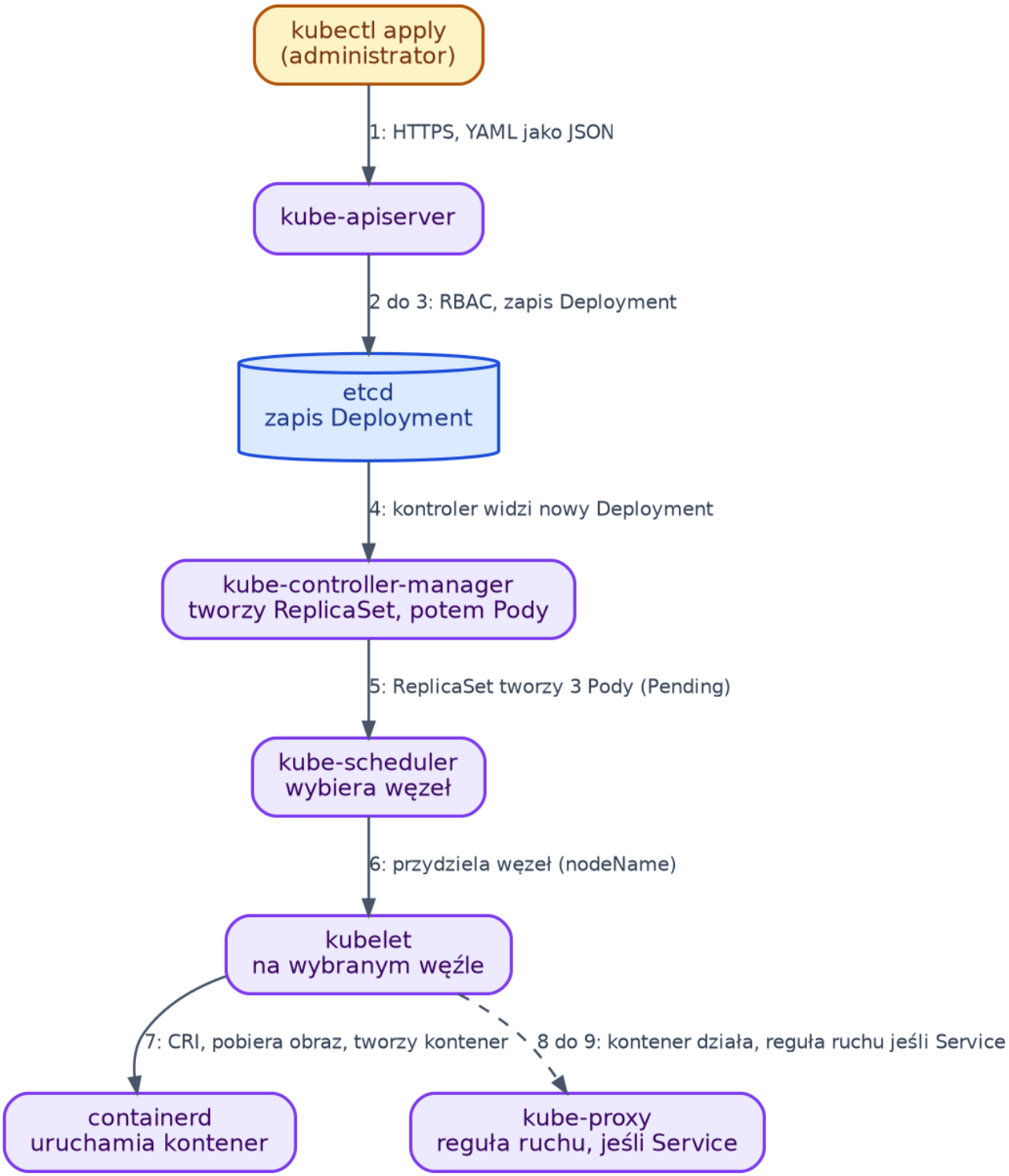
> **Pojęcie: CRI (Container Runtime Interface).** Standardowy interfejs, przez który `kubelet` komunikuje się z dowolnym środowiskiem uruchomieniowym kontenerów, nie wiążąc się na stałe z konkretną implementacją. Dzięki temu `kubelet` obsługuje `containerd`, `CRI-O` czy dowolne inne środowisko zgodne z CRI, wywołując dokładnie te same operacje — utwórz kontener, uruchom, zatrzymaj, pobierz obraz.

Docker Engine nigdy nie mówił natywnie w CRI — powstał wcześniej niż ten standard. Żeby Kubernetes mógł używać Dockera jako środowiska uruchomieniowego, `kubelet` przez lata ładował dodatkową warstwę tłumaczącą, `dockershim`, która przekładała wywołania CRI na wywołania API Dockera. Ta warstwa została usunięta z `kubelet` w wydaniu **Kubernetes 1.24** (kwiecień 2022) — od tej wersji Docker Engine przestał być obsługiwanym środowiskiem uruchomieniowym węzła.

To nie oznacza, że obrazy budowane Dockerem przestały działać w Kubernetesie — obrazy w formacie OCI, które produkuje `docker build`, są identyczne niezależnie od tego, jaki silnik je uruchamia, i `containerd` czy `CRI-O` uruchamiają je bez żadnej różnicy. Zniknął tylko sam demon Dockera jako proces zarządzający kontenerami na węźle roboczym — w jego miejsce większość dystrybucji Kubernetesa (w tym `kubeadm`) domyślnie instaluje `containerd`, czyli w istocie tę samą bibliotekę wykonawczą, na której Docker Engine sam był zbudowany.

Droga żądania: od kubectl apply do działającego kontenera

To jest najważniejszy fragment tej strony — spina wszystkie powyższe składniki w jeden przebieg. Zakładamy, że wykonujemy `kubectl apply -f wdrozenie.yaml`, gdzie manifest opisuje Deployment z trzema replikami.



Żądanie kubectl apply przechodzi kolejno przez apiserver, etcd, controller-manager, scheduler, kubelet i containerd, aż do działającego kontenera (kliknięcie otwiera pełny rozmiar)

1. **`kubectl`** wczytuje plik YAML, serializuje go do JSON i wysyła żądanie HTTPS do `kube-apiserver`, dołączając poświadczenia z aktualnego kontekstu w `~/.kube/config` (temat rozwija strona kubectl: narzędzie codzienne).

2. **`kube-apiserver`** uwierzytelnia żądanie i sprawdza w RBAC, czy to konto ma prawo tworzyć obiekty typu Deployment w tej przestrzeni nazw. Następnie przepuszcza obiekt przez kontrolery przyjęcia — walidujące (np. zgodność ze standardami bezpieczeństwa poda) i mutujące (mogą dopisać domyślne wartości) — i waliduje zgodność ze schematem danego rodzaju zasobu.
3. Obiekt Deployment trafia do **`etcd`** jako nowy wpis — to jest jedyny moment, w którym cokolwiek zostaje trwale zapisane.
4. Kontroler Deploymentu wewnątrz **`kube-controller-manager`** obserwuje (mechanizmem `watch` na `kube-apiserver`) nowo utworzone obiekty Deployment. Widząc nowy, tworzy odpowiadający mu obiekt ReplicaSet z `replicas: 3`.
5. Kontroler ReplicaSet, również wewnątrz `kube-controller-manager`, widzi nowy ReplicaSet bez żadnych podów i tworzy trzy obiekty Pod — na razie bez przypisanego węzła (`nodeName` puste, faza `Pending`).
6. **`kube-scheduler`** obserwuje pody bez przypisanego węzła. Dla każdego z trzech ocenia dostępne węzły robocze pod kątem wolnych zasobów i reguł przypisania, wybiera najlepszy i zapisuje decyzję z powrotem przez `kube-apiserver` do `etcd` — uzupełniając pole `nodeName`.
7. **`kubelet`** na wskazanym węźle, obserwujący pody przypisane do siebie, widzi nowy pod. Każe środowisku uruchomieniowemu (**`containerd`** przez CRI) pobrać wskazany obraz z rejestru, jeśli go jeszcze nie ma lokalnie, i utworzyć kontener — z własnymi przestrzeniami nazw jądra i `cgroups`, dokładnie tak jak przy `docker run`, tylko zainicjowane przez `kubelet`, a nie ręcznie.
8. Kontener startuje. `kubelet` zaczyna wykonywać zadeklarowane sondy (startową, gotowości, żywotności) i cyklicznie zgłasza aktualny stan poda z powrotem do `kube-apiserver`, który zapisuje go w `etcd` jako pole `status`.
9. Jeśli pod jest celem jakiegoś obiektu `Service`, **`kube-proxy`** na każdym węźle dostaje przez `watch` informację o nowym gotowym podzie i dokłada regułę przekierowania ruchu do niego.

Ten sam przebieg, w odwrotną stronę, tłumaczy dlaczego `kubectl get pods` zaraz po `apply` pokazuje pody w fazie `Pending`, a dopiero po chwili `Running` — obserwujemy krok po kroku tę samą sekwencję zdarzeń, którą właśnie opisaliśmy, tylko oglądaną z zewnątrz.

Wysoka dostępność płaszczyzny sterowania

Pojedynczy węzeł sterujący jest pojedynczym punktem awarii dla wszystkiego, co wymaga podjęcia nowej decyzji — dlatego produkcyjne klastry stawia się z co najmniej trzema węzłami sterującymi.

`etcd` działa wtedy jako klaster rozproszony, w którym zapis wymaga zgody większości członków (kworum) — stąd nieparzysta liczba węzłów (3 albo 5), bo przy parzystej liczbie łatwiej o sytuację,

w której żadna ze stron podziału sieci nie ma większości. `kube-apiserver` uruchamiamy na każdym węźle sterującym niezależnie i stawiamy przed nimi load balancer — każda replika jest bezstanowa i w pełni równoprawna, więc żaden dodatkowy mechanizm wyboru lidera nie jest jej potrzebny. `kube-scheduler` i `kube-controller-manager` natomiast muszą działać jako dokładnie jedna aktywna instancja naraz (uruchomienie dwóch planistów decydujących niezależnie o tym samym podzie prowadziłoby do konfliktów) — dlatego każda replika rywalizuje o dzierżawę (lease) w `etcd` metodą leader election, i tylko ta, która ją aktualnie trzyma, faktycznie działa; pozostałe stoją w gotowości i przejmują rolę natychmiast, gdy lider przestanie odnawiać dzierżawę.

Częste błędy

Zakładanie, że awaria `kube-apiserver` natychmiast zatrzymuje działające aplikacje. Pody, które już działały, kontynuuje `kubelet` na swoim węźle niezależnie od dostępności płaszczyzny sterowania — przestaje działać wyłącznie zdolność do wprowadzania nowych zmian i reagowania na nowe awarie.

Mylenie kopii zapasowej węzła sterującego z kopią zapasową `etcd`. Stan całego klastra leży w `etcd`, nie w systemie plików węzła sterującego — kopia zapasowa `etcd` (opisana osobno w części o eksploatacji) jest tym, co faktycznie pozwala odtworzyć klaster po katastrofie.

Przekonanie, że usunięcie Dockera z węzłów oznacza koniec obsługi obrazów budowanych Dockerem. Znika demon Dockera jako środowisko uruchomieniowe węzła, nie format obrazu — `docker build` i `docker push` na stacji roboczej administratora działają bez zmian, zmienia się wyłącznie to, czym te obrazy są uruchamiane na węzłach klastra.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Opisujemy, co dokładnie przestaje działać w klastrze, gdy pada jedyny węzeł sterujący, a co nadal działa bez zmian.
2. Wymieniamy pięć składników płaszczyzny sterowania i dla każdego podajemy jedno zdanie o tym, co się stanie, jeśli akurat on ulegnie awarii.
3. Prześledzamy krok po kroku, co się dzieje od momentu wykonania `kubectl apply -f manifest.yaml` do momentu, gdy `kubectl get pods` pokazuje status `Running` — z wymienieniem, który składnik odpowiada za który krok.
4. Wyjaśniamy, dlaczego usunięcie `dockershim` w Kubernetes 1.24 nie unieważniło obrazów zbudowanych wcześniej poleceniem `docker build`.

Powiązane: Po co Kubernetes, skoro mamy Dockera · Obiekty, manifesty i przestrzenie nazw · Pod: najmniejsza jednostka · kubectl: narzędzie codzienne

Źródła: dokumentacja Kubernetes, rozdział „Kubernetes Components" (kubernetes.io/docs/concepts/overview/components/) · „Container Runtimes" (kubernetes.io/docs/setup/production-environment/container-runtimes/) · „Dockershim Removal FAQ" (kubernetes.io/blog/2022/02/17/dockershim-faq)

Obiekty, manifesty i przestrzenie nazw

Poziom: podstawy **Dotyczy: Kubernetes 1.37** **Czas: ~20 min**

Po przeczytaniu tej strony wiemy, z czego składa się manifest YAML, czym różnią się etykiety od adnotacji, co dzieli klaster na przestrzenie nazw, i jak samodzielnie sprawdzić dowolny zasób poleceniami `kubectl api-resources` i `kubectl explain`, zamiast szukać w internecie.

Skróty użyte na tej stronie (6)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CRD	Custom Resource Definition: własny typ obiektu dodany do Kubernetesa
JSON	JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami
RBAC	Role-Based Access Control: uprawnienia nadawane rolom, a role ludziom
URL	Uniform Resource Locator: adres strony lub zasobu
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Fiszka

TYLKO ODCZYT

Sprawdzenie wszystkich dostępnych rodzajów zasobów: kubectl api-resources

```
kubectl api-resources
```

Zwracamy pełną listę rodzajów obiektów (`Kind`), jakie rozumie ten konkretny klaster — łącznie z tymi dodanymi przez zainstalowane rozszerzenia (CRD) — wraz ze skróconą nazwą, grupą API i informacją, czy dany rodzaj jest przypisany do przestrzeni nazw.

Przykład:

```
NAME                  SHORTNAMES   APIVERSION                     NAMESPACED
KIND
namespaces            ns           v1                             false
Namespace
nodes                 no           v1                             false
Node
pods                  po           v1                             true         Pod
services              svc          v1                             true
Service
configmaps            cm           v1                             true
ConfigMap
deployments           deploy       apps/v1                        true
Deployment
replicasets           rs           apps/v1                        true
ReplicaSet
statefulsets          sts          apps/v1                        true
StatefulSet
daemonsets            ds           apps/v1                        true
DaemonSet
jobs                               batch/v1                       true         Job
cronjobs              cj           batch/v1                       true
CronJob
ingresses             ing          networking.k8s.io/v1           true
Ingress
networkpolicies       netpol       networking.k8s.io/v1           true
NetworkPolicy
```

CRD = Custom Resource Definition: własny typ obiektu dodany do Kubernetesa · **API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Sprawdzenie definicji pola zasobu: kubectl explain

```
kubectl explain pod.spec.containers
```

Wyciągamy z klastra dokumentację danego pola, wygenerowaną z tego samego schematu API, którego klaster faktycznie używa do walidacji — zawsze zgodną z zainstalowaną wersją, w przeciwieństwie do artykułu znalezionego w wyszukiwarce, który mógł zostać napisany dla innej wersji Kubernetesa.

Przykład:

```
GROUP:
KIND:       Pod
VERSION:    v1

FIELD: containers <[]Container>

DESCRIPTION:
    List of containers belonging to the pod. Containers cannot currently be
    added or removed. There must be at least one container in a Pod. Cannot
    be updated.
```

Zagłębienie się w konkretne pole zagnieżdżone (np. sondę gotowości) działa dokładnie tak samo:

```
kubectl explain pod.spec.containers.readinessProbe
```

API = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wylistowanie przestrzeni nazw: kubectl get namespaces

```
kubectl get namespaces
```

Przykład:

```
NAME              STATUS   AGE
default           Active   180d
kube-node-lease   Active   180d
kube-public       Active   180d
kube-system       Active   180d
produkcja         Active   94d
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wyfiltrowanie obiektów po etykiecie: kubectl get -l

```
kubectl get pods -l app=NazwaUslugi -n produkcja
```

Zwracamy tylko te pody, które mają etykietę `app` o wskazanej wartości — niezależnie od tego, jak się nazywają ani przez jaki obiekt zostały utworzone.

Przykład:

```
NAME                              READY   STATUS    RESTARTS   AGE
katalog-produktow-6f9b7c9-4kxpl   1/1     Running   0          3h12m
katalog-produktow-6f9b7c9-9wq2z   1/1     Running   0          3h12m
katalog-produktow-6f9b7c9-vh8mn   1/1     Running   0          2h58m
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Nadanie etykiety istniejącemu obiektowi: kubectl label

```
kubectl label pod NazwaPoda srodowisko=produkcja -n produkcja
```

Przykład:

```
pod/katalog-produktow-6f9b7c9-4kxpl labeled
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 11:05 Tomek dostaje pytanie, dlaczego usługa `katalog-produktow` w przestrzeni `produkcja` nie widzi trzech nowych podów, mimo że `kubectl get pods` pokazuje je jako `Running`.

> Sprawdza manifest usługi i widzi, że ktoś przy kopiowaniu manifestu z innej aplikacji zostawił selektor `app: stary-katalog`, podczas gdy nowe pody dostały etykietę `app: katalog-produktow` z aktualnego szablonu. Po poprawieniu selektora usługa od razu zaczyna kierować ruch do wszystkich trzech podów, bez żadnego restartu czy dodatkowej rejestracji.

Obiekt, spec i status

> **Pojęcie: obiekt Kubernetesa.** Trwały zapis w `etcd` opisujący jeden element stanu klastra — jeden pod, jeden Deployment, jedna usługa. Każdy obiekt ma dokładnie dwie części odpowiadające na dwa różne pytania: `spec` opisuje, jak ten obiekt ma wyglądać (piszemy ją my, w manifeście), a `status` opisuje, jak wygląda w rzeczywistości w tej chwili (wypełnia go sam Kubernetes, na podstawie tego, co faktycznie obserwuje). Cała pętla uzgadniania, opisana na stronie Po co Kubernetes, skoro mamy Dockera, sprowadza się do nieustannego porównywania tych dwóch pól i korygowania różnic.

Rozróżnienie jest wyraźne w praktyce: edytujemy wyłącznie `spec` (przez `kubectl apply` albo `kubectl edit`) — nigdy ręcznie nie ustawiamy `status`, bo i tak zostanie natychmiast nadpisany przez kontroler, który raportuje to, co faktycznie widzi.

Budowa manifestu YAML

Manifest to plik YAML (albo JSON, chociaż YAML jest w praktyce jedynym używanym formatem) opisujący jeden albo więcej obiektów do wysłania do `kube-apiserver`. Cztery pola są obowiązkowe w niemal każdym manifeście:

```
apiVersion: v1
kind: Pod
metadata:
  name: siec-testowy
  namespace: produkcja
  labels:
    app: siec-testowy
spec:
  containers:
    - name: siec-testowy
      image: nginx:1.27
      ports:
```

```
        - containerPort: 80
```

`apiVersion` wskazuje, do której grupy API i w jakiej wersji dojrzałości należy dany rodzaj obiektu — omówione niżej.

`kind` to nazwa rodzaju obiektu — `Pod`, `Deployment`, `Service`, dokładnie taka, jaką widzimy w kolumnie `KIND` wyniku `kubectl api-resources`.

`metadata` niesie tożsamość obiektu — nazwę (unikalną w obrębie przestrzeni nazw i rodzaju zasobu), przestrzeń nazw, etykiety i adnotacje, opisane niżej.

`spec` niesie właściwą treść — to, co jest specyficzne dla danego rodzaju obiektu i różni się między `Pod`, `Deployment` czy `Service`. To jedyna sekcja, której zawartość trzeba sprawdzać osobno dla każdego rodzaju zasobu, najwygodniej poleceniem `kubectl explain`.

Grupy API i wersjonowanie

> **Pojęcie: grupa API.** Sposób organizacji rodzajów zasobów w Kubernetesowym API — zamiast jednej płaskiej listy wszystkich możliwych obiektów, API jest podzielone na grupy, z których każda wersjonuje się niezależnie od pozostałych. Podstawowe, najstarsze zasoby (`Pod`, `Service`, `ConfigMap`, `Namespace`) należą do tak zwanej grupy rdzeniowej (core), zapisywanej bez nazwy grupy — stąd `apiVersion: v1`, a nie `apiVersion: core/v1`. Nowsze rodzaje zasobów mają jawną nazwę grupy w `apiVersion`: `apps/v1` dla kontrolerów opisujących obciążenia (`Deployment`, `StatefulSet`, `DaemonSet`, `ReplicaSet`), `batch/v1` dla zadań (`Job`, `CronJob`), `networking.k8s.io/v1` dla `Ingress` i `NetworkPolicy`.

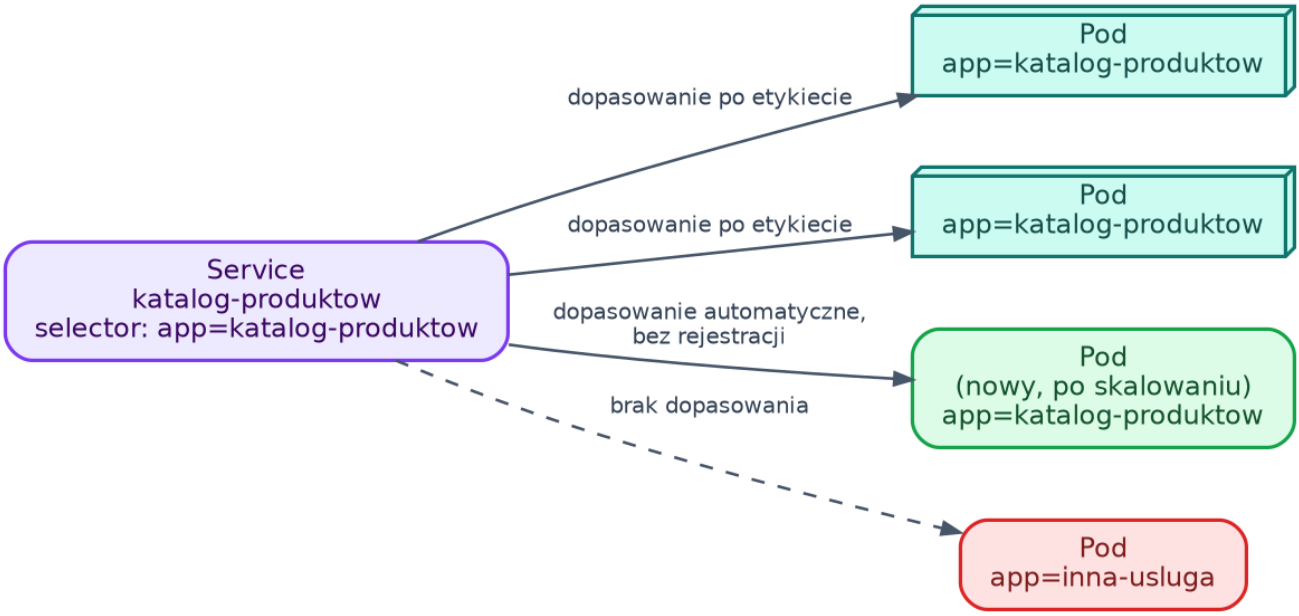
W obrębie każdej grupy poszczególne rodzaje zasobów przechodzą przez poziomy dojrzałości. **Alpha** (`v1alpha1`) oznacza funkcję eksperymentalną, wyłączoną domyślnie, mogącą zniknąć albo zmienić się bez ostrzeżenia między wydaniami — nie używamy jej na produkcji. **Beta** (`v1beta1`, `v1beta2`) oznacza funkcję włączoną domyślnie i testowaną w praktyce, ale ze schematem, który wciąż może się zmienić przed osiągnięciem stabilności. **Stable** (`v1`) oznacza zamrożony schemat, wspierany przez wiele kolejnych wydań i nieusuwany bez wieloletniego okresu przejściowego. Dobra praktyka to pisanie manifestów zawsze pod najnowszą wersję stabilną danego rodzaju zasobu, chyba że konkretna potrzebna funkcja istnieje jeszcze wyłącznie w wersji beta.

Etykiety i selektory

> **Pojęcie: etykieta (label).** Para klucz-wartość dopięta do metadanych dowolnego obiektu, służąca do jego identyfikacji i grupowania — nie do opisu w wolnej formie, tylko do

maszynowego dopasowywania. `app: katalog-produktow`, `srodowisko: produkcja`, `wersja: 2.4.1` to typowe etykiety.

Pojęcie: selektor (selector). Zapytanie o etykiety, którym jeden obiekt wskazuje zbiór innych obiektów pasujących do wzorca — zamiast wymieniać je po nazwie. To selektory, a nie nazwy, spinają Kubernetesa w jeden system: obiekt `Service` nie wie i nie musi wiedzieć, jak nazywają się konkretne pody, które ma obsługiwać — wie tylko, że interesują go pody z etykietą `app: katalog-produktow`, i za każdym razem, gdy pojawi się nowy pod z tą etykietą (bo Deployment go utworzył podczas skalowania albo aktualizacji), automatycznie zostaje objęty tym samym selektorem, bez żadnej dodatkowej rejestracji.



Service dopasowuje pody wyłącznie po etykiecie selektora, obejmuje automatycznie nowy pod z tą samą etykietą i pomija pod z inną etykietą (kliknięcie otwiera pełny rozmiar)

```
apiVersion: v1
kind: Service
metadata:
  name: katalog-produktow
  namespace: produkcja
spec:
  selector:
    app: katalog-produktow
  ports:
```

```
  - port: 80
    targetPort: 80
```

Ten sam mechanizm łączy Deployment z jego ReplicaSetem, a ReplicaSet z jego podami — Deployment nie trzyma listy nazw trzech konkretnych podów, które ma pilnować, tylko selektor `matchLabels`, dopasowujący dowolny pod noszący odpowiednią etykietę. Gdyby Kubernetes identyfikował powiązania po nazwach, każda operacja skalowania czy aktualizacji wymagałaby ręcznego przepisywania list nazw we wszystkich powiązanych obiektach — selektory eliminują tę zależność całkowicie.

Adnotacje

> **Pojęcie: adnotacja (annotation).** Para klucz-wartość dopięta do metadanych obiektu, tak jak etykieta, ale nieużywana do selekcji — adnotacje nie da się dopasować selektorem. Służą do przechowywania dowolnych dodatkowych informacji, które komuś się przydają, ale nie mają służyć do grupowania obiektów: opis ostatniej zmiany, adres osoby odpowiedzialnej, konfiguracja narzędzia trzeciej strony (np. kontrolera Ingress, który czyta adnotacje, żeby dowiedzieć się, jak ma skonfigurować reverse proxy dla danej trasy), albo — jak w przypadku `kubectl apply` — pełna treść ostatnio zastosowanego manifestu, zapisana automatycznie pod adnotacją `kubectl.kubernetes.io/last-applied-configuration`, żeby kolejne `apply` mogło policzyć różnicę.

Rozróżnienie w praktyce: jeśli kiedykolwiek napiszemy do tej wartości selektor (`kubectl get pods -l ...`), to była etykieta. Jeśli wartość jest zbyt długa, zbyt zmienna albo z natury nienadająca się do dopasowywania (adres URL, znacznik czasu, pełny JSON), to adnotacja.

Przestrzenie nazw

> **Pojęcie: przestrzeń nazw (namespace).** Mechanizm dzielący pojedynczy klaster na logicznie odseparowane obszary — nazwy obiektów muszą być unikalne w obrębie jednej przestrzeni nazw, ale ten sam `Pod` o nazwie `siec-testowy` może istnieć jednocześnie w przestrzeni `produkcja` i w przestrzeni `test`, bez konfliktu. Typowe zastosowanie to rozdzielenie środowisk (`dev`, `test`, `produkcja`) albo zespołów korzystających z jednego wspólnego klastra.

Klaster startuje z czterema wbudowanymi przestrzeniami: `default` (dla obiektów utworzonych bez podania przestrzeni jawnie), `kube-system` (składniki płaszczyzny sterowania i dodatki, jak `coredns`), `kube-public` (dane odczytywalne przez każdego, nawet nieuwierzytelnionego), `kube-node-lease`

(obiekty dzierżawy używane do wykrywania, czy węzeł wciąż żyje).

Nie każdy rodzaj obiektu mieści się w przestrzeni nazw — kolumna `NAMESPACED` w wyniku `kubectl api-resources` mówi wprost, który jest który. Obiekty przypisane do przestrzeni (`Pod`, `Deployment`, `Service`, `ConfigMap`) są w niej zamknięte — dwie różne przestrzenie nie widzą nawzajem swoich podów przez proste `kubectl get pods` bez podania przestrzeni. Obiekty globalne dla całego klastra (`Node`, `Namespace` sam w sobie, `PersistentVolume`, `ClusterRole`) nie należą do żadnej przestrzeni — istnieją raz, dla całego klastra, i przestrzenie nazw nie mają na nie wpływu.

Warto podkreślić, że przestrzeń nazw sama w sobie nie jest granicą bezpieczeństwa ani izolacją sieciową — domyślnie pod w przestrzeni `test` może swobodnie łączyć się siecią z podem w przestrzeni `produkcja`, chyba że jawnie to ograniczymy regułami opisanymi w części poświęconej sieci klastra. Przestrzeń nazw porządkuje przestrzeń nazewniczą i ułatwia zarządzanie (limity zasobów, RBAC per przestrzeń), ale nie izoluje ruchu sieciowego sama z siebie.

Częste błędy

Ręczna edycja pola `status` obiektu. Nie wywoła żadnego trwałego efektu — kontroler przy najbliższym przebiegu pętli uzgadniania nadpisze je stanem, jaki faktycznie obserwuje.

Zakładanie, że dwa obiekty o tej samej nazwie w różnych przestrzeniach nazw to ten sam obiekt. `kubectl get pod siec-testowy` bez podania przestrzeni odpytuje `default` (albo bieżącą przestrzeń z kontekstu) i może zwrócić „not found", mimo że pod o tej nazwie istnieje — tylko w innej przestrzeni.

Mylenie etykiety z adnotacją przy projektowaniu manifestu. Wrzucenie informacji, po której nigdy nie będziemy filtrować (np. pełny opis zmiany), do etykiety zaśmieca wynik `kubectl get --show-labels` bez żadnej korzyści — adnotacja jest właściwym miejscem na taką treść.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Sprawdzamy poleceniem `kubectl explain`, jakie pola przyjmuje `spec` obiektu `Deployment` na poziomie `strategy`, bez zaglądania do dokumentacji w przeglądarce.
2. Wyjaśniamy, dlaczego `Service` odwołuje się do podów przez selektor etykiet, a nie przez listę ich nazw, i co by się zepsuło, gdyby było odwrotnie.
3. Tworzymy dwa pody o identycznej nazwie w dwóch różnych przestrzeniach nazw i sprawdzamy, że współistnieją bez konfliktu.
4. Podajemy przykład informacji, którą umieścilibyśmy w adnotacji, a nie w etykiecie, i uzasadniamy wybór.

Powiązane: Po co Kubernetes, skoro mamy Dockera · Architektura klastra: płaszczyzna sterowania i węzły · kubectl: narzędzie codzienne · Deployment, ReplicaSet i aktualizacje bez przerwy

Źródła: dokumentacja Kubernetes, rozdział „Understanding Kubernetes Objects" i „Working with Kubernetes Objects" (kubernetes.io/docs/concepts/overview/working-with-objects/) · „Kubernetes API Concepts" (kubernetes.io/docs/reference/using-api/api-concepts/)