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/)

Revision #33
Created 2026-09-25 10:32:49 UTC by Tomasz Konieczny
Updated 2026-10-26 14:16:51 UTC by Tomasz Konieczny