Pod: najmniejsza jednostka

Poziom: podstawy **Dotyczy: Kubernetes 1.37** **Czas: ~30 min**

Po przeczytaniu tej strony wiemy, czym pod różni się od kontenera, dlaczego traktujemy pody jako jednorazowe, jak działają sondy i dlaczego źle skonfigurowana sonda to najczęstsza przyczyna restartów w pętli.

Skróty użyte na tej stronie (7)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CPU	Central Processing Unit: procesor
HTTP	Hypertext Transfer Protocol: protokół stron WWW
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
QoS	Quality of Service: klasa ważności, np. który pod zostanie zabity pierwszy przy braku pamięci
TCP	Transmission Control Protocol: protokół sieciowy z potwierdzaniem dostarczenia
TLS	Transport Layer Security: szyfrowanie połączeń, np. w HTTPS

Fiszka

TYLKO ODCZYT

Sprawdzenie fazy poda: kubectl get pod -o jsonpath

```
kubectl get pod NazwaPoda -o jsonpath='{.status.phase}'
```

Przykład:

```
Running
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Sprawdzenie przyczyny ostatniego zakończenia kontenera: kubectl get pod -o jsonpath

```
kubectl get pod NazwaPoda -o jsonpath='{.status.containerStatuses[0].lastState}'
```

Wyciągamy wyłącznie informację o poprzednim, już nieaktualnym stanie kontenera — to tutaj widać `OOMKilled` albo kod wyjścia procesu, zanim `kubelet` go zrestartował.

Przykład:

```
{"terminated":{"exitCode":137,"reason":"OOMKilled","startedAt":"2026-09-23T07:58:10Z","finishedAt":"2026-09-23T07:58:12Z"}}
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podejrzenie przyczyny restartu w zdarzeniach poda: kubectl describe pod

```
kubectl describe pod NazwaPoda
```

Przykład:

```
Events:
  Type     Reason     Age                From     Message
  ----     ------     ----               ----     -------
  Warning  Unhealthy  4m (x3 over 6m)    kubelet  Liveness probe failed: HTTP probe failed with statuscode: 503
  Normal   Killing    4m                 kubelet  Container katalog-produktow failed liveness probe, will be restarted
  Normal   Pulled     3m59s              kubelet  Container image "git.zielona.example/zielona/katalog-produktow:2.4.1" already present on machine
  Normal   Created    3m59s              kubelet  Created container katalog-produktow
  Normal   Started    3m58s              kubelet  Started container katalog-produktow
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 7:15 Tomek dostaje alert, że katalog-produktow restartuje się w kółko na klastrze produkcyjnym, mimo że wieczorem wszystko działało poprawnie. W `kubectl describe pod` widzi serię zdarzeń `Liveness probe failed`, tuż po sobie, jedno po drugim, od momentu ostatniego wdrożenia z nową, cięższą wersją obrazu. Aplikacja po prostu rozgrzewa się dłużej niż wcześniej, a sonda żywotności bez żadnej sondy startowej zabija ją, zanim zdąży skończyć rozruch. Po dodaniu `startupProbe` z rozsądnym `failureThreshold` restarty ustają.

Pod, nie kontener

Pojęcie: pod. Najmniejsza jednostka, jaką Kubernetes umieszcza na węźle i jaką planista przydziela do konkretnej maszyny — nigdy pojedynczy kontener z osobna. Pod to koperta wokół jednego lub kilku kontenerów, które mają współdzielić ten sam adres IP, tę samą przestrzeń nazw sieciową jądra i, opcjonalnie, te same wolumeny — Kubernetes zarządza cyklem życia poda jako całości, nie poszczególnych kontenerów wewnątrz niego z osobna.

Różnica wobec pojedynczego kontenera Dockera (opisanego na stronie Obraz, kontener, warstwa, wolumen: cztery pojęcia) jest fundamentalna, nie kosmetyczna: `docker run` uruchamia jeden izolowany proces z własną, oddzielną siecią. Pod z dwoma kontenerami uruchamia dwa procesy, które widzą się nawzajem przez `localhost`, tak jakby działały na tej samej maszynie — bo z perspektywy sieciowej dzielą dokładnie tę samą przestrzeń nazw jądra. Najprostszy, najczęstszy przypadek to pod z dokładnie jednym kontenerem — i wtedy różnica jest niewidoczna w praktyce, ale mechanizm pod spodem jest wciąż ten sam: Kubernetes zawsze operuje na podach, kontener wewnątrz to szczegół implementacji tego konkretnego poda.

Dlaczego pody są jednorazowe

Pod raz utworzony ma stały adres IP tylko do momentu, aż przestanie istnieć — restart kontenera wewnątrz poda nie zmienia adresu IP poda (`kubelet` po prostu podnosi nowy proces w tym samym podzie), ale usunięcie i ponowne utworzenie całego poda (bo na przykład węzeł padł, albo trwa aktualizacja przez Deployment) tworzy nowy obiekt z zupełnie nowym adresem IP i zwykle nową nazwą wygenerowaną z losowym sufiksem.

Z tego wynika reguła, którą warto zapamiętać: nie tworzymy podów ręcznie na produkcji i nie polegamy na ich tożsamości. Pod utworzony bezpośrednio (bez kontrolera nad nim) nie ma nikogo, kto by go odtworzył po awarii węzła — znika bezpowrotnie, dokładnie tak samo jak zapisywalna warstwa kontenera dockerowego znika po `docker rm`. Dlatego w praktyce pody prawie zawsze tworzone są pośrednio, przez kontroler wyższego poziomu — Deployment, StatefulSet, Job — który sam decyduje, kiedy stary pod zastąpić nowym, i to jemu, nie pojedynczym podom, powierzamy trwałość działania aplikacji. Kontrolery te rozwija strona Deployment, ReplicaSet i aktualizacje bez przerwy.

Wiele kontenerów w jednym podzie

Wzorzec sidecar

> **Pojęcie: sidecar.** Dodatkowy kontener w tym samym podzie co kontener główny, uzupełniający jego funkcjonalność bez ingerencji w jego kod — na przykład kontener wysyłający logi głównej aplikacji do zewnętrznego systemu, kontener terminujący TLS przed ruchem trafiającym do aplikacji, albo kontener synchronizujący pliki konfiguracyjne z zewnętrznego źródła do wspólnego wolumenu. Sidecar dzieli z kontenerem głównym tę samą sieć (może się z nim komunikować przez `localhost`) i, jeśli manifest tak deklaruje, te same wolumeny — ale ma własny obraz, własny cykl życia startu i własne żądania oraz limity zasobów.

```
apiVersion: v1
kind: Pod
metadata:
  name: aplikacja-z-logerem
  namespace: produkcja
spec:
  containers:
    - name: aplikacja
      image: git.zielona.example/zielona/katalog-produktow:2.4.1
      volumeMounts:
        - name: logi
          mountPath: /var/log/app
    - name: log-shipper
```

```
      image: git.zielona.example/zielona/fluent-bit:2.2
      volumeMounts:
        - name: logi
          mountPath: /var/log/app
          readOnly: true
  volumes:
    - name: logi
      emptyDir: {}
```

Oba kontenery widzą ten sam katalog `/var/log/app` przez wspólny wolumen `emptyDir` (wolumen istniejący tylko przez czas życia poda, znikający razem z nim) — aplikacja zapisuje logi, sidecar je odczytuje i przekazuje dalej, bez modyfikowania samej aplikacji o logikę wysyłki.

Kontenery inicjalizujące

> **Pojęcie: kontener inicjalizujący (init container).** Kontener uruchamiany przed kontenerami głównymi poda, do samego końca, po czym kończący działanie — dopiero gdy wszystkie kontenery inicjalizujące zakończą się powodzeniem (kod wyjścia 0), `kubelet` uruchamia właściwe kontenery poda. Kilka kontenerów inicjalizujących uruchamia się po kolei, jeden po drugim, nie równolegle — w przeciwieństwie do kontenerów głównych, które startują wszystkie naraz.

Typowe zastosowanie to poczekanie na dostępność zależności (np. bazy danych) przed startem aplikacji, pobranie i rozpakowanie pliku konfiguracyjnego do wspólnego wolumenu, albo wykonanie migracji schematu bazy danych, zanim aplikacja zacznie do niej pisać.

```
apiVersion: v1
kind: Pod
metadata:
  name: aplikacja-z-oczekiwaniem
  namespace: produkcja
spec:
  initContainers:
    - name: czekaj-na-baze
      image: busybox:1.36
      command:
        - sh
        - -c
        - "until nc -z baza-danych 5432; do echo czekamy na baze; sleep 2; done"
```

```
  containers:
    - name: aplikacja
      image: git.zielona.example/zielona/katalog-produktow:2.4.1
```

Kontener inicjalizujący, który kończy się niepowodzeniem, jest przez `kubelet` uruchamiany ponownie zgodnie z `restartPolicy` poda — kontenery główne w ogóle nie startują, dopóki ten warunek nie zostanie spełniony.

Cykl życia poda i fazy

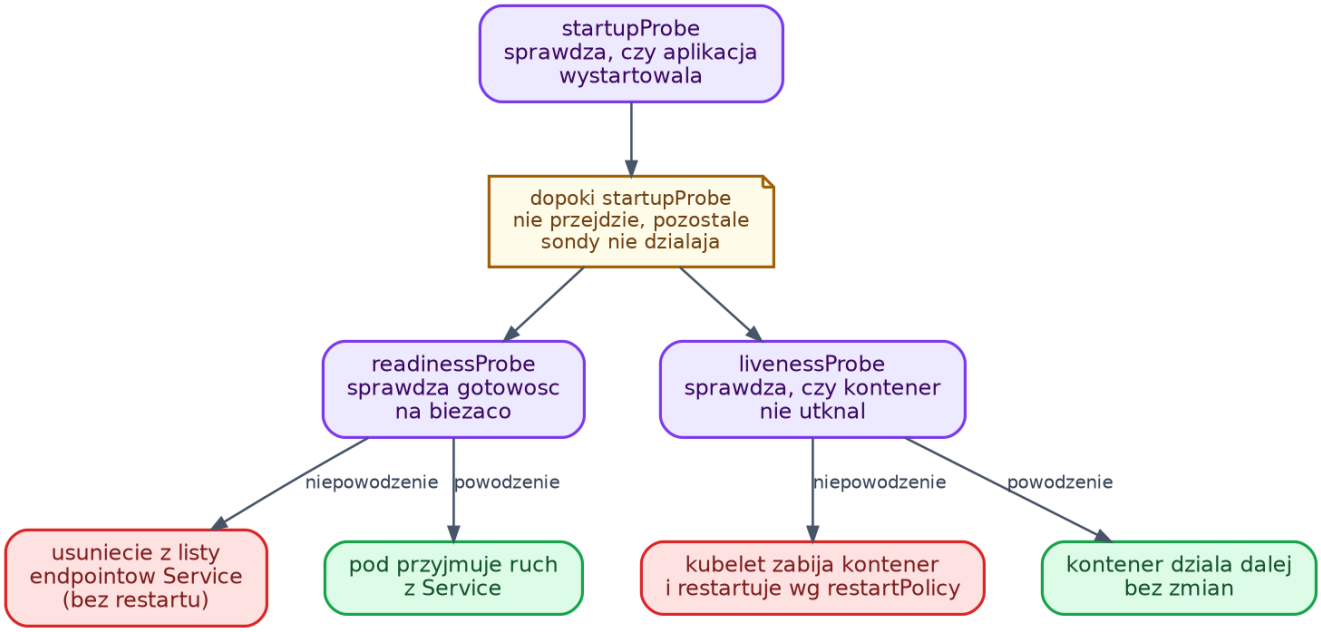
> **Pojęcie: faza poda (phase).** Ogólny, uproszczony wskaźnik tego, na jakim etapie swojego cyklu życia znajduje się pod, widoczny w polu `status.phase`. Kubernetes rozróżnia pięć faz:

Faza	Znaczenie
`Pending`	pod przyjęty przez `kube-apiserver`, ale co najmniej jeden z jego kontenerów jeszcze nie działa — najczęściej czeka na przydział węzła przez planistę albo na pobranie obrazu
`Running`	pod został przydzielony do węzła i przynajmniej jeden z jego kontenerów działa albo jest w trakcie startu lub restartu
`Succeeded`	wszystkie kontenery poda zakończyły działanie z kodem wyjścia 0 i nie zostaną ponownie uruchomione — typowe dla podów tworzonych przez `Job`
`Failed`	przynajmniej jeden kontener zakończył działanie z niezerowym kodem wyjścia, a `restartPolicy` nie przewiduje ponownego uruchomienia
`Unknown`	stan poda nie mógł zostać ustalony, zwykle wskutek utraty łączności z węzłem, na którym działa

Faza to informacja zgrubna — `Running` nie oznacza „gotowy do przyjmowania ruchu", tylko „proces działa". To rozróżnienie jest właśnie tym, po co istnieją sondy.

Sondy: startowa, gotowości i żywotności

> **Pojęcie: sonda (probe).** Mechanizm, którym `kubelet` sam, aktywnie, sprawdza stan kontenera z zewnątrz — wykonując żądanie HTTP, otwierając połączenie TCP albo uruchamiając komendę wewnątrz kontenera — zamiast biernie ufać, że skoro proces działa, to znaczy, że działa poprawnie. Kubernetes rozróżnia trzy rodzaje sond, każdą o innym przeznaczeniu i innej konsekwencji niepowodzenia.



Sonda startowa blokuje pozostałe sondy do czasu rozruchu, sonda gotowości steruje tylko ruchem z Service, a sonda żywotności decyduje o zabiciu i restarcie kontenera (kliknięcie otwiera pełny rozmiar)

Sonda startowa (`startupProbe`) sprawdza, czy aplikacja w ogóle skończyła się uruchamiać — przydatna przy aplikacjach z długim, zmiennym czasem startu (np. rozgrzewanie cache, wczytywanie dużego modelu). Dopóki sonda startowa nie przejdzie pomyślnie, sondy gotowości i żywotności w ogóle nie są wykonywane — dzięki temu długi, legalny czas rozruchu nie zostaje pomylony z zawieszeniem się aplikacji.

Sonda gotowości (`readinessProbe`) sprawdza, czy kontener jest w danej chwili gotowy do obsługi ruchu. Niepowodzenie tej sondy nie restartuje kontenera — usuwa go jedynie z listy endpointów usługi (patrz mechanizm `kube-proxy` opisany na stronie Architektura klastra: płaszczyzna sterowania i węzły), więc ruch przestaje na niego trafiać, dopóki sonda znów nie zacznie przechodzić. Typowe zastosowanie: aplikacja, która na chwilę traci połączenie z bazą danych, powinna przestać dostawać ruch, ale nie musi być za to restartowana.

Sonda żywotności (`livenessProbe`) sprawdza, czy kontener w ogóle jeszcze działa poprawnie, czy utknął w stanie, z którego sam się nie wybudzi (zakleszczenie, wyciek pamięci prowadzący do

zawieszenia, wątek główny zablokowany). Niepowodzenie tej sondy kończy się zabiciem kontenera i jego ponownym uruchomieniem przez `kubelet`, zgodnie z `restartPolicy`.

```
apiVersion: v1
kind: Pod
metadata:
  name: aplikacja-z-sondami
  namespace: produkcja
spec:
  containers:
    - name: aplikacja
      image: git.zielona.example/zielona/katalog-produktow:2.4.1
      ports:
        - containerPort: 8080
      startupProbe:
        httpGet:
          path: /zdrowie
          port: 8080
        failureThreshold: 30
        periodSeconds: 2
      readinessProbe:
        httpGet:
          path: /gotowosc
          port: 8080
        periodSeconds: 5
        failureThreshold: 3
      livenessProbe:
        httpGet:
          path: /zdrowie
          port: 8080
        periodSeconds: 10
        failureThreshold: 3
```

Najczęstsza przyczyna restartów w pętli (`CrashLoopBackOff`) to źle dobrana sonda żywotności — zbyt krótki `periodSeconds` albo zbyt niski `failureThreshold` w stosunku do rzeczywistego czasu odpowiedzi aplikacji pod obciążeniem sprawia, że `kubelet` zabija zdrowy, tylko chwilowo wolniejszy kontener, ten wstaje od nowa, znowu jest wolny podczas rozgrzewki, znowu zostaje zabity — i tak w kółko. Rozwiązaniem niemal zawsze jest dodanie albo wydłużenie sondy startowej, żeby dać aplikacji spokojnie skończyć rozruch, zanim zacznie ją oceniać sonda żywotności.

Żądania i limity zasobów, klasy QoS, OOMKilled

> **Pojęcie: żądanie zasobów (`requests`).** Ilość CPU i pamięci, jaką kontener deklaruje jako potrzebną do normalnej pracy — planista używa tej wartości, decydując, czy dany węzeł ma jeszcze dość wolnych zasobów, żeby przyjąć ten pod. **Pojęcie: limit zasobów (`limits`).** Górna granica, powyżej której kontenerowi nie wolno się rozrosnąć — CPU powyżej limitu zostaje jedynie throttlowane (kontener zwalnia, ale działa dalej), pamięć powyżej limitu kończy się natychmiastowym zabiciem procesu przez jądro (`OOMKilled`, kod wyjścia 137) — w przeciwieństwie do CPU, pamięci nie da się „przydusić", więc jedyną reakcją jądra jest zabicie procesu.

```
apiVersion: v1
kind: Pod
metadata:
  name: aplikacja-z-limitami
  namespace: produkcja
spec:
  containers:
    - name: aplikacja
      image: git.zielona.example/zielona/katalog-produktow:2.4.1
      resources:
        requests:
          cpu: "250m"
          memory: "256Mi"
        limits:
          cpu: "500m"
          memory: "512Mi"
```

> **Pojęcie: klasa QoS (Quality of Service).** Kategoria przydzielana podowi automatycznie przez Kubernetesa na podstawie relacji między `requests` a `limits` wszystkich jego kontenerów, decydująca, który pod zostanie poświęcony jako pierwszy, gdy węzłowi zabraknie pamięci.

Klasa	Warunek	Kolejność przy braku zasobów węzła

`Guaranteed`	`requests` równe `limits` dla CPU i pamięci, we wszystkich kontenerach	usuwany jako ostatni
`Burstable`	ustawione `requests`, ale różne od `limits` (albo brak `limits`)	usuwany w drugiej kolejności
`BestEffort`	brak jakichkolwiek `requests` i `limits`	usuwany jako pierwszy

Pod z powyższego przykładu (`requests` niższe niż `limits`) trafia do klasy `Burstable` — może chwilowo skorzystać z większej ilości zasobów, jeśli węzeł ma je akurat wolne, ale przy presji pamięciowej padnie wcześniej niż pod skonfigurowany jako `Guaranteed`.

restartPolicy

> **Pojęcie: `restartPolicy`.** Pole na poziomie całego poda, określające, co `kubelet` ma zrobić, gdy kontener wewnątrz poda zakończy działanie. `Always` (wartość domyślna) restartuje kontener niezależnie od kodu wyjścia — jedyny sensowny wybór dla poda pod kontrolą Deploymentu, który z założenia ma działać bez końca. `OnFailure` restartuje wyłącznie po zakończeniu z niezerowym kodem wyjścia, zostawiając pod w spokoju po zakończeniu sukcesem (kod 0) — używane przez `Job`, patrz strona StatefulSet, DaemonSet, Job i CronJob. `Never` nie restartuje kontenera wcale, niezależnie od wyniku — rzadziej spotykane, głównie przy jednorazowych zadaniach diagnostycznych.

Nasz przykład: kolorowy nginx

Ten sam wzorzec kontenera inicjalizującego, wolumenu `emptyDir` i sond widzimy razem, na jednym konkretnym przykładzie, na stronie k3s w laboratorium: pierwszy deploy, nginx w kolorach. Kontener inicjalizujący `generator-koloru` odczytuje własną nazwę przez Downward API (dokładnie ten mechanizm z pola `fieldRef`, o którym mowa wyżej przy `restartPolicy` i sondach), liczy z niej prosty skrót i zapisuje gotową stronę `index.html` do wspólnego wolumenu, z którego czyta ją już tylko główny kontener nginksa. Sonda gotowości i sonda żywotności pilnują tego samego kontenera nginx dokładnie tak, jak opisano wyżej: dopóki nie odpowie poprawnie na żądanie HTTP do strony głównej, Service nie kieruje do niego ruchu.

```
initContainers:
  - name: generator-koloru
    image: busybox:1.36
    env:
```

```
      - name: POD_NAME
        valueFrom:
          fieldRef:
            fieldPath: metadata.name
    volumeMounts:
      - name: strona
        mountPath: /strona
containers:
  - name: nginx
    image: nginx:1.29.8
    readinessProbe:
      httpGet:
        path: /
        port: 80
      periodSeconds: 5
```

Częste błędy

Ustawienie zbyt agresywnej sondy żywotności bez sondy startowej. Aplikacja z długim, legalnym czasem rozruchu zostaje zabita, zanim zdąży wystartować, i wpada w `CrashLoopBackOff` mimo że w rzeczywistości działałaby poprawnie, gdyby dać jej więcej czasu.

Brak `requests` i `limits` „bo na razie działa". Pod trafia do klasy `BestEffort` i zostanie zabity jako pierwszy przy jakimkolwiek niedoborze pamięci na węźle — nawet jeśli sam w sobie nigdy nie przekroczył rozsądnego zużycia.

Mylenie sondy gotowości z sondą żywotności przy diagnozowaniu problemu. Pod, który dostał `Unhealthy` od sondy gotowości, nie zostanie zrestartowany — zniknie tylko z listy endpointów usługi. Szukanie przyczyny restartów w logach sondy gotowości jest szukaniem w niewłaściwym miejscu.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Wyjaśniamy, dlaczego pod z dwoma kontenerami współdzielącymi ten sam wolumen `emptyDir` to nadal jeden obiekt Kubernetesa, a nie dwa.

2. Piszemy manifest poda z jednym kontenerem inicjalizującym, który czeka na dostępność usługi sieciowej, zanim wystartuje kontener główny.
3. Opisujemy różnicę w konsekwencjach niepowodzenia między sondą gotowości a sondą żywotności, na przykładzie aplikacji, która na chwilę traci połączenie z bazą danych.
4. Dobieramy klasę QoS (Guaranteed, Burstable, BestEffort) dla trzech scenariuszy: usługa krytyczna dla działania firmy, usługa pomocnicza tolerująca chwilowe spowolnienie, jednorazowe zadanie diagnostyczne.

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

Źródła: dokumentacja Kubernetes, „Pods" i „Pod Lifecycle" (kubernetes.io/docs/concepts/workloads/pods/) · „Configure Liveness, Readiness and Startup Probes" (kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/) · „Resource Management for Pods and Containers" (kubernetes.io/docs/concepts/configuration/manage-resources-containers/)

Revision #35
Created 2026-09-25 10:32:57 UTC by Tomasz Konieczny
Updated 2026-10-26 14:16:57 UTC by Tomasz Konieczny