StatefulSet, DaemonSet, Job i CronJob

Poziom: podstawy **Dotyczy: Kubernetes 1.37, apps/v1, batch/v1** **Czas: ~30 min**

Po przeczytaniu tej strony wiemy, czym różnią się StatefulSet, DaemonSet, Job i CronJob od Deploymentu, kiedy po który sięgamy, i dlaczego baza danych postawiona jako StatefulSet nie jest pierwszym wyborem.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
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)

Fiszka

TYLKO ODCZYT

Wylistowanie podów StatefulSetu z ich stałymi nazwami: kubectl get pods -l

```
kubectl get pods -l app=baza-kolejki -o wide
```

Przykład:

```
NAME             READY   STATUS    RESTARTS   AGE   IP            NODE
baza-kolejki-0   1/1     Running   0          14d   10.244.1.20   k3s02
baza-kolejki-1   1/1     Running   0          14d   10.244.2.11   k3s03
```

```
baza-kolejki-2   1/1     Running   0          13d   10.244.1.22   k3s02
```

W przeciwieństwie do podów Deploymentu, nazwy tutaj są przewidywalne i stałe — kolejny numer porządkowy, nie losowy sufiks — i pozostają identyczne nawet po restarcie danego poda.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wylistowanie DaemonSetów i liczby węzłów, na których działają: kubectl get daemonset

```
kubectl get daemonset -n kube-system
```

Przykład:

```
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR
AGE
agent-logow     3         3         3       3            3           <none>
62d
kube-proxy      3         3         3       3            3           <none>
62d
```

Kolumna `DESIRED` równa liczbie węzłów w klastrze potwierdza, że DaemonSet trzyma się swojej zasady — dokładnie jeden pod na każdym pasującym węźle, niezależnie od tego, ile ich jest.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wylistowanie zadań i ich statusu: kubectl get jobs

```
kubectl get jobs -n produkcja
```

Przykład:

```
NAME                       STATUS     COMPLETIONS   DURATION   AGE
migracja-schematu-2026-09  Complete   1/1           14s        3h
```

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

□□ TYLKO ODCZYT

Wylistowanie zadań cyklicznych i czasu ostatniego uruchomienia: kubectl get cronjobs

```
kubectl get cronjobs -n produkcja
```

Przykład:

```
NAME             SCHEDULE      TIMEZONE   SUSPEND   ACTIVE   LAST SCHEDULE   AGE
kopia-zapasowa   0 2 * * *     Local      False     0        14h             90d
```

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

□□ ZMIENIA SYSTEM

Ręczne wywołanie CronJoba poza harmonogramem: kubectl create job --from

```
kubectl create job kopia-zapasowa-recznie --from=cronjob/kopia-zapasowa -n produkcja
```

Tworzymy jednorazowy `Job` na podstawie szablonu istniejącego `CronJob`, bez czekania na najbliższe uruchomienie zgodne z harmonogramem — przydatne przy teście zadania po zmianie jego definicji.

Przykład:

```
job.batch/kopia-zapasowa-recznie created
```

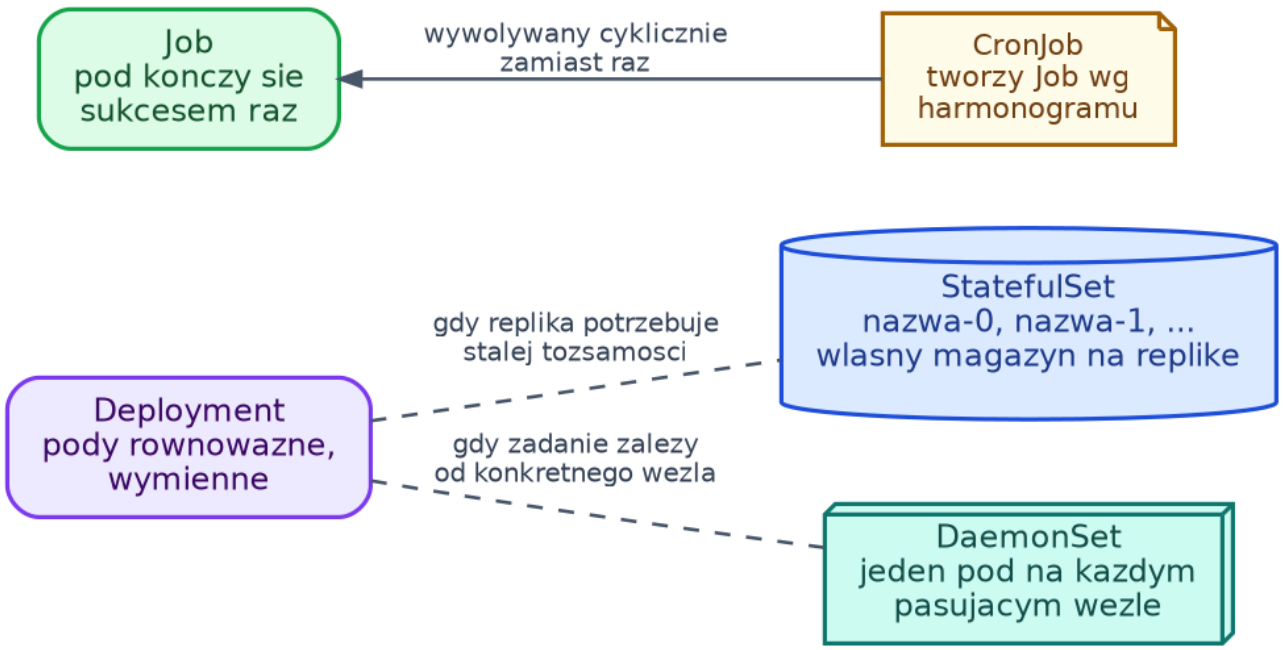
□□ **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

□□ Z ŻYCIA WZIĘTE

O 3:00 w nocy CronJob kopii zapasowej bazy kolejki nie kończy się przed uruchomieniem kolejnej instancji tego samego zadania, bo poprzednia noc miała wyjątkowo dużo danych do

przetworzenia. Rano Tomek widzi w `kubectl get jobs` dwa uruchomienia naraz, oba piszące do tego samego pliku docelowego, i jeden z plików kopii zapasowej okazuje się uszkodzony. Dodanie `concurrencyPolicy: Forbid` do definicji CronJoba rozwiązuje problem raz na zawsze, bez zmiany ani jednej linii samego skryptu kopii zapasowej.



StatefulSet i DaemonSet odchodzą od Deploymentu w dwóch różnych kierunkach, a CronJob tworzy Job cyklicznie zamiast jednorazowo (kliknięcie otwiera pełny rozmiar)

StatefulSet — stabilna tożsamość

Pojęcie: StatefulSet. Kontroler analogiczny do Deploymentu w tym, że utrzymuje zadeklarowaną liczbę replik jednego szablonu poda — ale w odróżnieniu od Deploymentu, dla którego wszystkie pody są równoważne i wymienne, StatefulSet nadaje każdej replice trwałą, numerowaną tożsamość, którą zachowuje przez cały cykl życia tej repliki, łącznie z restartami.

Stabilna tożsamość sieciowa oznacza, że pod nazywa się zawsze `nazwa-N` (`baza-kolejki-0`, `baza-kolejki-1`, ...), a każda z tych nazw ma stały wpis DNS wewnątrz klastra wskazujący akurat na tę konkretną replikę — nie na dowolną z puli, jak przy Deploymencie i zwykłym `Service`. Restart repliki numer 1 daje w efekcie nowy pod, z nowym adresem IP, ale wciąż pod nazwą `baza-kolejki-1` i wciąż z tym samym, przypisanym tylko jej, magazynem danych.

Kolejność uruchamiania jest domyślnie sekwencyjna i uporządkowana — replika `0` musi osiągnąć stan gotowości, zanim StatefulSet w ogóle zacznie tworzyć replikę `1`, i tak dalej w górę. Tę samą kolejność, tylko odwróconą, StatefulSet zachowuje przy skalowaniu w dół i przy usuwaniu — replika o najwyższym numerze znika pierwsza. Ma to znaczenie dla aplikacji rozproszonych, w których kolejność dołączania węzłów do klastra (np. wybór pierwszego węzła jako inicjującego) jest częścią logiki działania.

`volumeClaimTemplates` to mechanizm, dzięki któremu każda replika dostaje własny, dedykowany magazyn trwały (temat rozwija część poświęcona magazynowaniu danych) — nie jeden wspólny wolumen dzielony przez wszystkie repliki, tylko po jednym na każdą, tworzony automatycznie przy pierwszym uruchomieniu tej repliki i zachowywany dla niej nawet po usunięciu i ponownym utworzeniu poda.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: baza-kolejki
  namespace: produkcja
spec:
  serviceName: baza-kolejki
  replicas: 3
  selector:
    matchLabels:
      app: baza-kolejki
  template:
    metadata:
      labels:
        app: baza-kolejki
    spec:
      containers:
        - name: baza-kolejki
          image: git.zielona.example/zielona/kolejka:3.8
          ports:
            - containerPort: 5672
          volumeMounts:
            - name: dane
              mountPath: /var/lib/kolejka
  volumeClaimTemplates:
    - metadata:
        name: dane
      spec:
```

```
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

Pole `serviceName` wskazuje na usługę bezgłową (headless Service), która nadaje poszczególnym replikom ich stałe wpisy DNS — sam mechanizm usług i DNS klastra rozwija część poświęcona sieci.

Dlaczego baza danych to nie jest pierwszy kandydat do Kubernetesa

StatefulSet rozwiązuje dwa konkretne problemy — stałą tożsamość sieciową i dedykowany magazyn per replika — ale nie rozwiązuje samej logiki replikacji, wyboru repliki nadrzędnej (primary) po awarii, ani bezpiecznego, spójnego tworzenia kopii zapasowych z uwzględnieniem stanu transakcji w toku. Te elementy dla każdego silnika bazodanowego wyglądają inaczej i StatefulSet o nich nic nie wie — dostarcza tylko szkielet, na którym ta logika musiałaby zostać dopisana ręcznie albo przez dedykowanego operatora (rozszerzenie API specyficzne dla danego silnika, temat spoza zakresu tej strony).

Zespoły stawiające bazę danych produkcyjną po raz pierwszy w Kubernetesie często odkrywają to dopiero przy pierwszej awarii repliki — StatefulSet sam z siebie nie wybierze nowego primary, nie poinformuje aplikacji o zmianie, nie upewni się, że failover nie skutkuje utratą ostatnich zapisanych transakcji. Stąd praktyczna reguła: bazę danych stawiamy w Kubernetesie przez dojrzałego, przetestowanego operatora danego silnika (jeśli taki istnieje i jest zaufany) albo — częściej i bezpieczniej — korzystamy z zarządzanej usługi bazodanowej poza klastrem, zostawiając Kubernetesowi to, w czym faktycznie jest dobry: bezstanowe warstwy aplikacyjne.

DaemonSet — jeden pod na węzeł

> **Pojęcie: DaemonSet.** Kontroler, który nie przyjmuje liczby replik jako parametru — zamiast tego utrzymuje dokładnie jeden pod na każdym węźle klastra pasującym do opcjonalnego selektora węzłów, automatycznie dokładając pod na każdym nowym węźle, który dołączy do klastra, i usuwając go, gdy węzeł odchodzi. Nie ma tu pojęcia „skalowania" w sensie Deploymentu — liczba podów jest bezpośrednią funkcją liczby węzłów, nie osobno ustawianą wartością.

Typowe zastosowania to zadania, które z natury muszą działać na każdej maszynie z osobna, nie gdziekolwiek w klastrze: agent zbierający logi z każdego węzła (np. Fluent Bit czytający pliki logów

kontenerów bezpośrednio z dysku węzła), agent monitoringu odczytujący metryki sprzętowe i systemowe węzła (np. `node-exporter`), oraz wtyczka sieciowa (CNI), która musi skonfigurować interfejsy sieciowe lokalnie na każdej maszynie, żeby pody na niej mogły w ogóle komunikować się siecią.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: agent-logow
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app: agent-logow
  template:
    metadata:
      labels:
        app: agent-logow
    spec:
      containers:
        - name: agent-logow
          image: git.zielona.example/zielona/fluent-bit:2.2
          volumeMounts:
            - name: logi-kontenerow
              mountPath: /var/log/containers
              readOnly: true
      volumes:
        - name: logi-kontenerow
          hostPath:
            path: /var/log/containers
```

Wolumen typu `hostPath` w tym przykładzie podpina katalog `/var/log/containers` bezpośrednio z hosta — DaemonSet z natury działa blisko systemu operacyjnego węzła, więc taki dostęp jest tu uzasadniony, w przeciwieństwie do zwykłej aplikacji biznesowej, dla której `hostPath` byłby nietypowym i ryzykownym wyborem.

Job — zadanie jednorazowe

> **Pojęcie: Job.** Kontroler tworzący jeden lub więcej podów w celu wykonania zadania, które ma się zakończyć — w przeciwieństwie do Deploymentu, którego pody z założenia działają bez końca. Job pilnuje, żeby zadeklarowana liczba podów zakończyła się sukcesem (kod wyjścia 0), i uznaje się za ukończony (`Complete`) dopiero wtedy.

`backoffLimit` określa, ile razy Job spróbuje ponownie uruchomić pod, jeśli poprzednia próba zakończyła się niepowodzeniem, zanim uzna całe zadanie za ostatecznie nieudane (`Failed`). **`activeDeadlineSeconds`** to twardy limit czasu na całe zadanie, liczony od jego startu — po przekroczeniu Job zabija wszystkie swoje pody i kończy się jako `Failed`, niezależnie od tego, ile prób z `backoffLimit` jeszcze zostało; zabezpiecza przed zadaniem, które zawiesiło się bez końca zamiast zakończyć się błędem, który `backoffLimit` mógłby wyłapać.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: migracja-schematu
  namespace: produkcja
spec:
  backoffLimit: 3
  activeDeadlineSeconds: 300
  template:
    spec:
      restartPolicy: OnFailure
      containers:
        - name: migracja-schematu
          image: git.zielona.example/zielona/migracje:1.14
          command: ["./migruj.sh"]
```

Zwracamy uwagę na `restartPolicy: OnFailure` — Job nigdy nie działa z `restartPolicy: Always` (opisanym na stronie Pod: najmniejsza jednostka), bo pod, który z definicji ma nie kończyć działania, nigdy nie osiągnąłby stanu `Complete`.

CronJob — zadanie cykliczne

> **Pojęcie: CronJob.** Kontroler tworzący nowy obiekt `Job` automatycznie, zgodnie z harmonogramem zapisanym w polu `schedule`, dokładnie tą samą składnią, którą zna systemowy `crontab` (rozwiniętą na stronie Zadania cykliczne: cron i crontab) — pięć pól oddzielonych spacją, minuta godzina dzień-miesiąca miesiąc dzień-tygodnia.

`concurrencyPolicy` rozstrzyga, co zrobić, jeśli poprzednie uruchomienie jeszcze trwa w momencie, gdy harmonogram każe uruchomić kolejne. `Allow` (domyślne) pozwala obu działać jednocześnie. `Forbid` pomija nowe uruchomienie, jeśli poprzednie wciąż trwa — właściwy wybór dla zadań, których dwie równoległe kopie mogłyby sobie nawzajem przeszkadzać (np. dwie kopie tworzące kopię zapasową tej samej bazy naraz). `Replace` zabija wciąż trwające poprzednie uruchomienie i od razu startuje nowe.

Sprzątanie po zakończonych zadaniach kontrolują `successfulJobsHistoryLimit` i `failedJobsHistoryLimit` — bez nich CronJob działający raz na godzinę zostawiałby po sobie setki zakończonych obiektów `Job` (i ich podów) w ciągu kilku tygodni, zaśmiecając wynik `kubectl get jobs`.

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: kopia-zapasowa
  namespace: produkcja
spec:
  schedule: "0 2 * * *"
  concurrencyPolicy: Forbid
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 3
  jobTemplate:
    spec:
      backoffLimit: 2
      activeDeadlineSeconds: 1800
      template:
        spec:
          restartPolicy: OnFailure
          containers:
            - name: kopia-zapasowa
              image: git.zielona.example/zielona/backup-tool:4.1
              command: ["./zrob-kopie.sh"]
```

Ten CronJob uruchamia zadanie kopii zapasowej codziennie o 2:00 w nocy, odrzuca kolejne uruchomienie, jeśli poprzednie jeszcze trwa, i trzyma historię trzech ostatnich udanych oraz trzech ostatnich nieudanych uruchomień do wglądu, kasując starsze automatycznie.

Częste błędy

Stawianie bazy danych jako StatefulSet bez operatora ani planu na failover. StatefulSet daje stały magazyn i stałą nazwę repliki, ale nie zdecyduje samodzielnie, która replika ma zostać nowym primary po awarii — bez tej logiki dopisanej osobno klaster bazy danych po awarii repliki głównej po prostu przestaje przyjmować zapisy.

Ustawienie `hostPath` w zwykłym Deploymencie na wzór DaemonSetu. DaemonSet ma uzasadniony powód sięgać po zasoby konkretnego hosta — zwykła aplikacja biznesowa przeniesiona na inny węzeł traci wtedy dostęp do danych, które zostały na poprzednim.

Brak `concurrencyPolicy: Forbid` przy zadaniu, które nie jest bezpieczne do równoległego uruchomienia. Dwie kopie tego samego zadania kopii zapasowej działające jednocześnie mogą wzajemnie się blokować na tym samym pliku docelowym albo, gorzej, obie zakończyć się pozornym sukcesem, zapisując niekompletne dane.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Wyjaśniamy, dlaczego pody StatefulSetu mają przewidywalne, numerowane nazwy, podczas gdy pody Deploymentu dostają nazwy z losowym sufiksem.
2. Piszemy manifest DaemonSetu instalującego agenta monitoringu na każdym węźle klastra, z wyjątkiem węzłów sterujących (podpowiedź: `tolerations` / `nodeSelector`, temat sygnalizowany, nie rozwinięty na tej stronie).
3. Dla zadania importu danych, które nie powinno nigdy działać w dwóch kopiach naraz i nie powinno trwać dłużej niż 15 minut, dobieramy wartości `concurrencyPolicy` i `activeDeadlineSeconds`.
4. Uzasadniamy, dlaczego StatefulSet sam w sobie nie jest wystarczającym powodem, żeby przenieść produkcyjną bazę danych do Kubernetesa.

Powiązane: Deployment, ReplicaSet i aktualizacje bez przerwy · Pod: najmniejsza jednostka · Zadania cykliczne: cron i crontab · Obiekty, manifesty i przestrzenie nazw

Źródła: dokumentacja Kubernetes, „StatefulSets" (kubernetes.io/docs/concepts/workloads/controllers/statefulset/) · „DaemonSet" (kubernetes.io/docs/concepts/workloads/controllers/daemonset/) · „Jobs" i „CronJob" (kubernetes.io/docs/concepts/workloads/controllers/job/, kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/)

Revision #33
Created 2026-09-25 10:32:59 UTC by Tomasz Konieczny
Updated 2026-10-26 14:17:01 UTC by Tomasz Konieczny