Obiekty robocze

kubectl, pody, Deployment i pozostałe kontrolery.

- kubectl: narzędzie codzienne
- Pod: najmniejsza jednostka
- Deployment, ReplicaSet i aktualizacje bez przerwy
- StatefulSet, DaemonSet, Job i CronJob

kubectl: narzędzie codzienne

Poziom: podstawy **Dotyczy: kubectl 1.37, Kubernetes 1.35–1.37 (skew ±1 wersja)**

Czas: ~35 min

Komendy wykonujemy z CLI bash jako user mający dostęp do klastra przez skonfigurowany `~/.kube/config` — bez poprawnego kontekstu każde polecenie kończy się błędem połączenia albo `Unauthorized`.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
CA	Certificate Authority: urząd certyfikacji, który podpisuje certyfikaty
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
RBAC	Role-Based Access Control: uprawnienia nadawane rolom, a role ludziom
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Fiszka

TYLKO ODCZYT

Wylistowanie dostępnych kontekstów: kubectl config get-contexts

```
kubectl config get-contexts
```

Przykład:

```
CURRENT   NAME            CLUSTER      AUTHINFO        NAMESPACE
*         produkcja       k8s-prod     admin-prod      produkcja
          test            k8s-test     admin-test      default
          lokalny-kind    kind-dev     kind-dev        default
```

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

□□ TYLKO ODCZYT

Przełączenie aktywnego kontekstu: kubectl config use-context

```
kubectl config use-context test
```

Przykład:

```
Switched to context "test".
```

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

□□ ZMIENIA SYSTEM

Ustawienie domyślnej przestrzeni nazw dla bieżącego kontekstu: kubectl config set-context --current

```
kubectl config set-context --current --namespace=produkcja
```

Zmieniamy przestrzeń nazw zapisaną w bieżącym kontekście, żeby nie dopisywać `-n produkcja` do każdej kolejnej komendy w tej sesji terminala.

Przykład:

```
Context "produkcja" modified.
```

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

□□ TYLKO ODCZYT

Wylistowanie zasobów z dodatkowymi kolumnami: kubectl get -o wide

```
kubectl get pods -o wide
```

Dokładamy do standardowego wyniku adres IP poda, węzeł, na którym działa, oraz nazwę węzła nominowanego — informacje pomijane w domyślnym, wąskim widoku.

Przykład:

```
NAME                              READY   STATUS    RESTARTS   AGE   IP
NODE    NOMINATED NODE   READINESS GATES
katalog-produktow-6f9b7c9-4kxpl   1/1     Running   0          3h    10.244.1.12
k3s02   <none>           <none>
katalog-produktow-6f9b7c9-9wq2z   1/1     Running   0          3h    10.244.2.8
k3s03   <none>           <none>
```

> Nazwy węzłów w wynikach (`k3s01`, `k3s02`, `k3s03`, adresy 10.20.10.44, 10.20.10.67 i 10.20.10.68) pochodzą z laboratorium k3s Zielonej, bo na nim ćwiczymy `kubectl`; budowę klastra przez `kubeadm` opisujemy osobno jako sposób ogólny.

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

TYLKO ODCZYT

Pobranie pełnej definicji obiektu w formacie YAML: kubectl get -o yaml

```
kubectl get deployment katalog-produktow -o yaml
```

Zwracamy kompletny obiekt tak, jak faktycznie leży w `etcd` — łącznie z polami wypełnionymi domyślnie przez klaster, których nie było w naszym manifeście. Przydatne, gdy chcemy zobaczyć, co Kubernetes dopisał sam, albo skopiować istniejący obiekt jako punkt wyjścia do nowego manifestu.

Przykład:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: katalog-produktow
  namespace: produkcja
  generation: 4
  resourceVersion: "88213"
spec:
  replicas: 3
  selector:
    matchLabels:
```

```
      app: katalog-produktow
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
status:
  replicas: 3
  readyReplicas: 3
  availableReplicas: 3
```

YAML = YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach · **etcd** = etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wyciągnięcie pojedynczego pola: kubectl get -o jsonpath

```
kubectl get pods -l app=katalog-produktow -o jsonpath='{.items[*].status.podIP}'
```

Wyciągamy z wyniku dokładnie tę wartość, która nas interesuje, pomijając resztę — nieocenione w skryptach, gdzie dalej chcemy przetwarzać samą wartość, a nie sparsowany YAML.

Przykład:

```
10.244.1.12 10.244.2.8 10.244.1.15
```

YAML = YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Śledzenie zmian zasobów na bieżąco: kubectl get --watch

```
kubectl get pods --watch
```

Zamiast jednorazowego zrzutu stanu, `kubectl` utrzymuje otwarte połączenie i dopisuje nową linię za każdym razem, gdy stan któregoś poda się zmienia — wygodne przy obserwowaniu

przebiegu wdrożenia albo skalowania na żywo, bez odpytywania w pętli.

Przykład:

```
NAME                               READY   STATUS    RESTARTS   AGE
katalog-produktow-6f9b7c9-4kxpl    1/1     Running   0          3h
katalog-produktow-7d8f6b5-xk2mn    0/1     Pending   0          0s
katalog-produktow-7d8f6b5-xk2mn    0/1     ContainerCreating   0   1s
katalog-produktow-7d8f6b5-xk2mn    1/1     Running   0          4s
katalog-produktow-6f9b7c9-4kxpl    1/1     Terminating   0        3h
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Szczegółowy opis obiektu ze zdarzeniami: kubectl describe

```
kubectl describe pod katalog-produktow-6f9b7c9-4kxpl
```

W przeciwieństwie do `kubectl get`, `describe` dokłada sekcję `Events` na końcu — chronologiczny dziennik tego, co płaszczyzna sterowania i `kubelet` faktycznie zrobiły z tym obiektem. To pierwsze miejsce do sprawdzenia, gdy pod nie startuje, a `kubectl get` pokazuje tylko suchy status bez przyczyny.

Przykład:

```
Name:         katalog-produktow-6f9b7c9-4kxpl
Namespace:    produkcja
Node:         k3s02/10.20.10.67
Status:       Running
IP:           10.244.1.12
Containers:
  katalog-produktow:
    Image:         git.zielona.example/zielona/katalog-produktow:2.4.1
    State:          Running
      Started:      Wed, 23 Sep 2026 08:12:04 +0200
    Ready:          True
    Restart Count:  0

Events:
```

```
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  3h    default-scheduler  Successfully assigned produkcja/katalog-
produktow-6f9b7c9-4kxpl to k3s02
  Normal  Pulled     3h    kubelet            Container image
"git.zielona.example/zielona/katalog-produktow:2.4.1" already present on machine
  Normal  Created    3h    kubelet            Created container katalog-produktow
  Normal  Started    3h    kubelet            Started container katalog-produktow
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podgląd logów kontenera na bieżąco: kubectl logs -f

```
kubectl logs -f katalog-produktow-6f9b7c9-4kxpl
```

Przykład:

```
2026-09-23T08:12:05.014Z INFO  Server started on port 8080
2026-09-23T08:12:41.220Z INFO  GET /api/produkty 200 12ms
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podgląd logów poprzedniej instancji kontenera po restarcie: kubectl logs --previous

```
kubectl logs --previous katalog-produktow-6f9b7c9-4kxpl
```

Kontener, który padł i został zrestartowany przez `kubelet`, dostaje nowy proces, a jego stare logi normalnie zniknęłyby razem z nim — `--previous` pokazuje logi z tej poprzedniej, już nieistniejącej instancji, co jest jedynym sposobem zobaczenia, co doprowadziło do awarii, zamiast oglądać logi już czystego, świeżo wystartowanego procesu.

Przykład:

```
2026-09-23T07:58:12.501Z ERROR Out of memory: heap limit exceeded
2026-09-23T07:58:12.502Z FATAL Process terminated
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Ograniczenie logów do ostatniego okresu: kubectl logs --since

```
kubectl logs --since=10m katalog-produktow-6f9b7c9-4kxpl
```

Przykład:

```
2026-09-23T08:05:12.100Z INFO  GET /api/produkty 200 9ms
2026-09-23T08:09:47.331Z INFO  GET /api/produkty 200 11ms
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Ograniczenie logów do ostatnich N linii: kubectl logs --tail

```
kubectl logs --tail=50 katalog-produktow-6f9b7c9-4kxpl
```

Zwracamy wyłącznie ostatnie 50 linii zamiast pełnej historii od startu kontenera — praktyczne przy długo działających kontenerach, których pełne logi liczą się w gigabajtach.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wejście do powłoki działającego kontenera: kubectl exec -it

```
kubectl exec -it katalog-produktow-6f9b7c9-4kxpl -- bash
```

Odpowiednik `docker exec -it` (patrz strona Codzienne komendy), tylko przez `kube-apiserver` jako pośrednika — `kubectl` nie łączy się bezpośrednio z węzłem, żądanie przechodzi przez tę samą warstwę uwierzytelniania i autoryzacji, co każde inne polecenie `kubectl`.

Przykład:

```
root@katalog-produktow-6f9b7c9-4kxpl:/app#
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

□□ ZMIENIA SYSTEM

Zastosowanie manifestu deklaratywnie: kubectl apply -f

```
kubectl apply -f wdrozenie.yaml
```

Przykład:

```
deployment.apps/katalog-produktow configured
```

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

□□ TYLKO ODCZYT

Podgląd różnicy przed zastosowaniem: kubectl diff -f

```
kubectl diff -f wdrozenie.yaml
```

Pokazujemy dokładnie to, co `apply` by zmieniło, zanim faktycznie to zrobimy — bezcenne przed wprowadzeniem zmiany na produkcji, żeby nie odkryć niespodziewanej różnicy dopiero po fakcie.

Przykład:

```
diff -u -N /tmp/LIVE-123456789/apps.v1.Deployment.produkcja.katalog-produktow
/tmp/MERGED-123456789/apps.v1.Deployment.produkcja.katalog-produktow
--- /tmp/LIVE-123456789/apps.v1.Deployment.produkcja.katalog-produktow
+++ /tmp/MERGED-123456789/apps.v1.Deployment.produkcja.katalog-produktow
@@ -14,7 +14,7 @@
   template:
     spec:
       containers:
-      - image: git.zielona.example/zielona/katalog-produktow:2.4.0
+      - image: git.zielona.example/zielona/katalog-produktow:2.4.1
```

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

□□ OSTROŻNIE · NIEODWRACALNE

Usunięcie obiektu: kubectl delete

```
kubectl delete -f wdrozenie.yaml
```

Przykład:

```
deployment.apps "katalog-produktow" deleted
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Sprawdzenie postępu wdrożenia: kubectl rollout status

```
kubectl rollout status deployment/katalog-produktow
```

Przykład:

```
Waiting for deployment "katalog-produktow" rollout to finish: 2 out of 3 new replicas
have been updated...
deployment "katalog-produktow" successfully rolled out
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Cofnięcie wdrożenia do poprzedniej wersji: kubectl rollout undo

```
kubectl rollout undo deployment/katalog-produktow
```

Przykład:

```
deployment.apps/katalog-produktow rolled back
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Ręczne przeskalowanie: kubectl scale

```
kubectl scale deployment/katalog-produktow --replicas=5
```

Przykład:

```
deployment.apps/katalog-produktow scaled
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Przekierowanie portu z poda na maszynę lokalną: kubectl port-forward

```
kubectl port-forward pod/katalog-produktow-6f9b7c9-4kxpl 8080:8080
```

Otwieramy tunel przez `kube-apiserver` do konkretnego poda, dostępny lokalnie pod `localhost:8080` — wygodne przy diagnostyce usługi, która normalnie nie jest wystawiona na zewnątrz klastra, bez zmieniania niczego w konfiguracji sieciowej.

Przykład:

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

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Skopiowanie pliku między podem a hostem: kubectl cp

```
kubectl cp produkcja/katalog-produktow-6f9b7c9-4kxpl:/app/logs/error.log ./error.log
```

Przykład:

```
tar: Removing leading '/' from member names
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podgląd zużycia zasobów przez pody: kubectl top

```
kubectl top pods
```

Wymaga zainstalowanego `metrics-server` w klastrze — bez niego komenda kończy się błędem `error: Metrics API not available`.

Przykład:

```
NAME                              CPU(cores)   MEMORY(bytes)
katalog-produktow-6f9b7c9-4kxpl   12m          186Mi
katalog-produktow-6f9b7c9-9wq2z   9m           179Mi
```

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

□□ TYLKO ODCZYT

Wylistowanie zdarzeń klastra posortowanych czasem: kubectl get events --sort-by

```
kubectl get events --sort-by='.lastTimestamp' -n produkcja
```

Zdarzenia znikają domyślnie po godzinie, więc to polecenie ma sens uruchamiać zaraz po zaobserwowaniu problemu, nie następnego dnia.

Przykład:

```
LAST SEEN   TYPE      REASON             OBJECT
MESSAGE
2m          Warning   FailedScheduling   pod/katalog-produktow-7d8f6b5-xk2mn     0/3
nodes are available: 3 Insufficient memory.
45s         Normal    Scheduled          pod/katalog-produktow-7d8f6b5-xk2mn
Successfully assigned produkcja/katalog-produktow-7d8f6b5-xk2mn to k3s03
```

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

□□ TYLKO ODCZYT

Sprawdzenie, czy mamy uprawnienia do wykonania akcji: kubectl auth can-i

```
kubectl auth can-i delete deployments -n produkcja
```

Sprawdzamy uprawnienia RBAC bieżącego użytkownika bez faktycznego wykonywania akcji — przydatne przed skryptem automatyzującym coś na produkcji, żeby błąd braku uprawnień wykrył sam skrypt na starcie, a nie w połowie sekwencji.

Przykład:

```
yes
```

RBAC = Role-Based Access Control: uprawnienia nadawane rolom, a role ludziom · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Wygenerowanie szkieletu manifestu bez wysyłania go do klastra: --dry-run=client -o yaml

```
kubectl create deployment katalog-produktow --image=git.zielona.example/zielona/katalog-produktow:2.4.1 --dry-run=client -o yaml
```

Przykład:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: katalog-produktow
  name: katalog-produktow
spec:
  replicas: 1
  selector:
    matchLabels:
      app: katalog-produktow
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
```

```
      labels:
        app: katalog-produktow
    spec:
      containers:
      - image: git.zielona.example/zielona/katalog-produktow:2.4.1
        name: katalog-produktow
status: {}
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 16:30 Tomek wykonuje `kubectl delete -f wdrozenie.yaml`, żeby posprzątać po teście, i dopiero po komunikacie `deployment.apps "katalog-produktow" deleted` zauważa, że gwiazdka w `kubectl config get-contexts` stała przy kontekście `produkcja`, nie `test`. Przez chwilę usługa katalogu produktów jest niedostępna, zanim `kubectl apply -f wdrozenie.yaml` z tego samego pliku w repozytorium odtwarza Deployment i wszystkie trzy pody wracają do stanu `Running`. Od tego dnia Tomek zaczyna każdą sesję pracy na klastrze od sprawdzenia, który kontekst ma gwiazdkę.

Plik kubeconfig — klaster, użytkownik, kontekst

Pojęcie: kubeconfig. Plik YAML, domyślnie `~/.kube/config`, w którym `kubectl` trzyma wszystko, co potrzebne do połączenia się z jednym albo wieloma klastrami. Składa się z trzech osobnych list, które dopiero kontekst łączy w jedną całość:

```
apiVersion: v1
kind: Config
clusters:
  - name: k8s-prod
    cluster:
```

```
      server: https://10.20.10.44:6443
      certificate-authority-data: LS0tLS1CRUdJTi...
users:
  - name: admin-prod
    user:
      client-certificate-data: LS0tLS1CRUdJTi...
      client-key-data: LS0tLS1CRUdJTi...
contexts:
  - name: produkcja
    context:
      cluster: k8s-prod
      user: admin-prod
      namespace: produkcja
current-context: produkcja
```

Klaster to sam adres `kube-apiserver` i certyfikat CA potrzebny, żeby `kubectl` mógł zweryfikować, że łączy się z właściwym serwerem, a nie z podstawionym. **Użytkownik** to poświadczenia — certyfikat klienta, token albo wywołanie zewnętrznego programu uwierzytelniającego (np. wtyczki chmurowej). **Kontekst** łączy jeden klaster, jednego użytkownika i domyślną przestrzeń nazw w jeden nazwany zestaw, między którymi przełączamy się poleceniem `kubectl config use-context` zamiast ręcznie edytować plik za każdym razem.

Jedna typowa stacja robocza administratora ma w `kubeconfig` kilka kontekstów naraz — produkcję, środowisko testowe i lokalny klaster do eksperymentów postawiony narzędziem `kind` albo `minikube` (oba są aktualnie oficjalnie rekomendowanymi sposobami uruchomienia Kubernetesa na własnej maszynie do nauki i developmentu, bez ryzyka pomyłkowego wykonania czegoś na realnym klastrze). Zanim wykonamy cokolwiek destrukcyjnego, dobrym nawykiem jest sprawdzenie `kubectl config get-contexts` i upewnienie się, który kontekst ma gwiazdkę przy `CURRENT`.

Tryb deklaratywny kontra imperatywny w kubectl

`kubectl` udostępnia oba podejścia opisane na stronie Po co Kubernetes, skoro mamy Dockera i to od nas zależy, którego użyjemy w danej sytuacji.

Polecenia imperatywne (`kubectl run`, `kubectl create deployment`, `kubectl scale`, `kubectl set image`) mówią klastrowi wprost, co zrobić, w tej chwili, i nie zostawiają po sobie żadnego pliku opisującego wynik. `kubectl run` tworzy dokładnie jeden pod i od wydania Kubernetes 1.18 nie potrafi już nic więcej — wcześniejsze mechanizmy generatorów, które pozwalały tą samą komendą

stworzyć też Deployment czy Job, zostały usunięte. Traktujemy go jako narzędzie do szybkiego testu albo debugowania, nie jako sposób wdrażania czegokolwiek na stałe.

`kubectl apply -f` działa deklaratywnie — porównuje trzy stany naraz: to, co jest zapisane w pliku teraz, to, co było zapisane przy poprzednim `apply` (zachowane w adnotacji `last-applied-configuration`, patrz strona Obiekty, manifesty i przestrzenie nazw), i to, co faktycznie jest aktualnie w klastrze — po czym oblicza trójstronny scalony wynik (three-way merge) i wysyła tylko różnicę. Dzięki temu `apply` nie nadpisuje pól, które ktoś (albo jakiś kontroler) zmienił ręcznie poza naszym manifestem, o ile sami tych pól nie dotykamy.

Na produkcji trzymamy się wyłącznie `apply` na plikach trzymanych w repozytorium, z dwóch powodów naraz. Po pierwsze manifest w repozytorium jest źródłem prawdy, które da się przejrzeć w code review, zanim trafi do klastra — polecenie imperatywne wpisane ręcznie w terminalu nie zostawia takiego śladu. Po drugie `apply` jest idempotentne — wykonanie tej samej komendy dwa razy z rzędu daje ten sam efekt (drugie wykonanie nic nie zmienia), podczas gdy powtórzenie `kubectl create` na już istniejącym obiekcie kończy się błędem `AlreadyExists`.

Generowanie szkieletu manifestu zamiast pisania od zera

Praktyczny kompromis między obydwoma trybami: używamy polecenia imperatywnego z flagą `--dry-run=client -o yaml`, żeby wygenerować poprawny składniowo szkielet manifestu bez wysyłania go do klastra (`client` oznacza, że walidacja i generowanie odbywają się wyłącznie lokalnie w `kubectl`, bez kontaktu z `kube-apiserver` — odmiennie od `--dry-run=server`, który wysyła żądanie do serwera i przechodzi pełną walidację łącznie z kontrolerami przyjęcia, ale bez trwałego zapisu). Przekierowujemy wynik do pliku, dopisujemy ręcznie to, czego generator nie umie wygenerować (limity zasobów, sondy, zmienne środowiskowe), i dopiero ten plik trafia do `kubectl apply` oraz do repozytorium.

```
kubectl create deployment katalog-produktow --image=git.zielona.example/zielona/katalog-produktow:2.4.1 --dry-run=client -o yaml > wdrozenie.yaml
```

Częste błędy

Wykonywanie `kubectl delete` bez sprawdzenia bieżącego kontekstu. Polecenie usuwające coś na klastrze testowym w zamierzeniu, wykonane przypadkiem na kontekście `produkcja` (bo ktoś zapomniał przełączyć), nie pyta drugi raz o potwierdzenie.

Sprawdzanie logów po restarcie kontenera bez `--previous`. Świeżo zrestartowany kontener ma czyste logi od nowego startu — przyczyna poprzedniej awarii jest widoczna wyłącznie w logach poprzedniej instancji.

Edytowanie obiektu bezpośrednio poleceniem `kubectl edit` na produkcji. Zmiana wchodzi w życie natychmiast, ale znika przy najbliższym `kubectl apply -f` z manifestu w repozytorium, który jej nie zawiera — źródło niespójności trudnej do zdiagnozowania później.

Poleganie na `kubectl run` do tworzenia czegokolwiek trwałego. Tworzy gołego poda, bez ReplicaSetu i bez mechanizmu samoleczenia po awarii — jeśli węzeł padnie, ten pod nie wstanie nigdzie indziej.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Wypisujemy wszystkie konteksty zapisane w naszym `~/.kube/config` i wskazujemy, który jest aktualnie aktywny.
2. Generujemy poleceniem `kubectl create deployment ... --dry-run=client -o yaml` szkielet Deploymentu dla obrazu `nginx:1.27` i zapisujemy go do pliku, nie wysyłając niczego do klastra.
3. Wyjaśniamy różnicę między `kubectl logs` a `kubectl logs --previous` i podajemy sytuację, w której to drugie jest jedynym sposobem zdiagnozowania awarii.
4. Opisujemy, co dokładnie robi `kubectl apply -f` inaczej niż `kubectl replace -f`, i dlaczego to pierwsze jest bezpieczniejsze przy współpracy wielu osób nad tym samym klastrem.

Powiązane: Po co Kubernetes, skoro mamy Dockera · Obiekty, manifesty i przestrzenie nazw · Deployment, ReplicaSet i aktualizacje bez przerwy · Codzienne komendy

Źródła: dokumentacja Kubernetes, „kubectl Reference Docs" (kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/) · „kubectl Cheat Sheet" (kubernetes.io/docs/reference/kubectl/cheatsheet/) · „Object Management Using kubectl" (kubernetes.io/docs/concepts/overview/working-with-objects/object-management/)

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

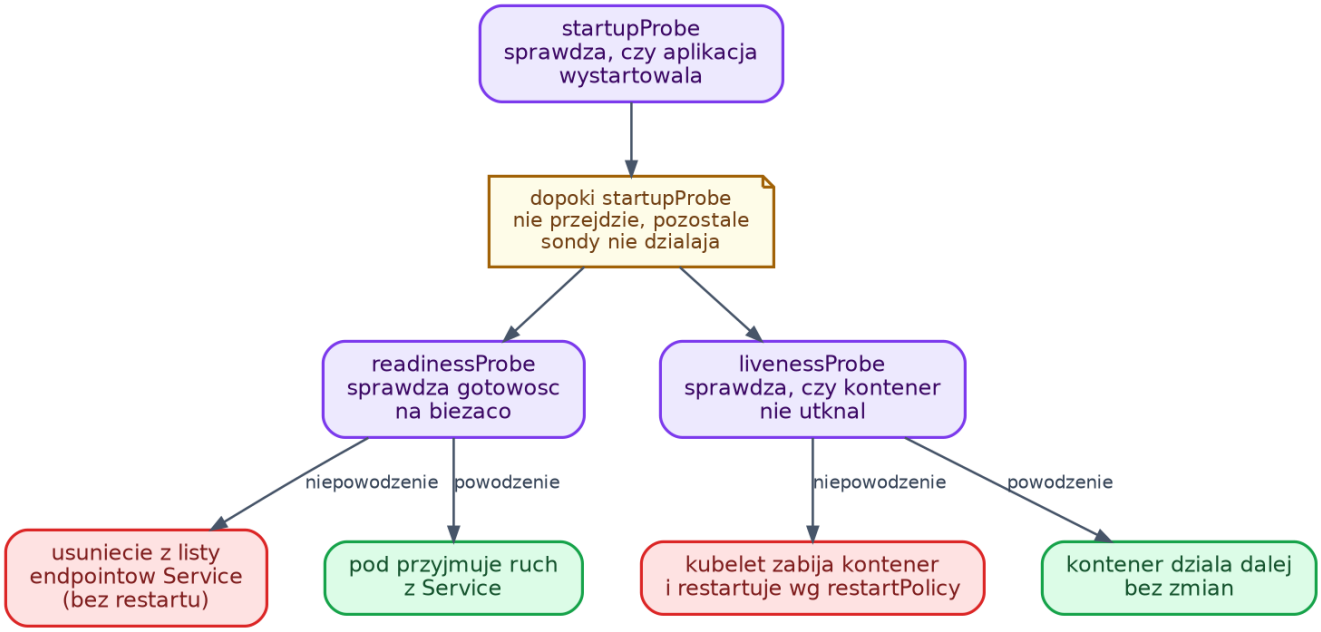
> **Pojęcie: faza poda (phase).** Ogólny, uproszczony wskaźnik tego, na jakim etapie swojego cyklu życia znajduje się pod, widoczny w polu `status.phase`. Kubernetes rozróżnia pięć faz:

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/)

Deployment, ReplicaSet i aktualizacje bez przerwy

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

Po przeczytaniu tej strony wiemy, jak Deployment, ReplicaSet i Pod są ze sobą powiązane, jak przebiega aktualizacja bez przerwy w działaniu krok po kroku i dlaczego bez sond gotowości ta sama aktualizacja oznacza przerwę.

Skróty użyte na tej stronie (2)

Skrót	Co znaczy
HPA	HorizontalPodAutoscaler: obiekt Kubernetesa, który sam zwiększa albo zmniejsza liczbę podów Deploymentu, gdy zużycie procesora lub pamięci odbiega od celu
PDB	Pluggable Database: baza wpinana do kontenera CDB w Oracle

Fiszka

TYLKO ODCZYT

Sprawdzenie ReplicaSetów należących do wdrożenia: kubectl get rs -l

```
kubectl get rs -l app=katalog-produktow
```

Przykład:

```
NAME                          DESIRED   CURRENT   READY   AGE
katalog-produktow-6f9b7c9     0         0         0       4d
katalog-produktow-7d8f6b5     3         3         3       12m
```

Stary ReplicaSet (`6f9b7c9`) nie zniknął — Deployment zachowuje go z zerową liczbą replik jako historię, do której można wrócić poleceniem `rollout undo`.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

□□ ZMIENIA SYSTEM

Podmiana obrazu w działającym wdrożeniu: kubectl set image

```
kubectl set image deployment/katalog-produktow katalog-
produktow=git.zielona.example/zielona/katalog-produktow:2.5.0
```

Przykład:

```
deployment.apps/katalog-produktow image updated
```

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

□□ ZMIENIA SYSTEM

Sprawdzenie historii wersji wdrożenia: kubectl rollout history

```
kubectl rollout history deployment/katalog-produktow
```

Przykład:

```
deployment.apps/katalog-produktow
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=wdrozenie.yaml
2         kubectl set image deployment/katalog-produktow katalog-
produktow=git.zielona.example/zielona/katalog-produktow:2.4.1
3         kubectl set image deployment/katalog-produktow katalog-
produktow=git.zielona.example/zielona/katalog-produktow:2.5.0
```

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

□□ ZMIENIA SYSTEM

Cofnięcie do konkretnej wcześniejszej rewizji: kubectl rollout undo --to-revision

```
kubectl rollout undo deployment/katalog-produktow --to-revision=2
```

Przykład:

```
deployment.apps/katalog-produktow rolled back
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Skonfigurowanie automatycznego skalowania: kubectl autoscale

```
kubectl autoscale deployment/katalog-produktow --min=3 --max=10 --cpu-percent=70
```

Przykład:

```
horizontalpodautoscaler.autoscaling/katalog-produktow autoscaled
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 13:00 zespół sprzedaży zgłasza, że strona katalogu produktów przez kilka sekund po każdym wdrożeniu zwraca błędy 502. Tomek sprawdza manifest Deploymentu i widzi, że nikt nie dodał `readinessProbe` do kontenera, więc Deployment uznaje nowy pod za gotowy natychmiast po starcie procesu, zanim aplikacja zdąży połączyć się z bazą danych. Po dopisaniu sondy gotowości kolejne wdrożenie przechodzi bez ani jednego zgłoszonego błędu, bo ruch trafia do nowego poda dopiero, gdy ten faktycznie odpowiada.

Kontroler i pętla uzgadniania w praktyce

Deployment to konkretne, praktyczne zastosowanie pętli uzgadniania opisanej ogólnie na stronie Po co Kubernetes, skoro mamy Dockera — a właściwie dwóch osobnych pętli działających jedna nad drugą, każda odpowiedzialna za inny poziom szczegółowości.

Deployment → ReplicaSet → Pod

> **Pojęcie: ReplicaSet.** Kontroler o jednym, wąskim zadaniu — utrzymanie dokładnie zadeklarowanej liczby podów pasujących do danego selektora etykiet, niezależnie od tego, dlaczego akurat jednego brakuje. ReplicaSet sam z siebie nic nie wie o wersjach, aktualizacjach ani historii — widzi tylko liczbę `replicas` i selektor.

> **Pojęcie: Deployment.** Warstwa nad ReplicaSetem, odpowiedzialna za to, czego ReplicaSet nie potrafi — zarządzanie zmianą wersji w czasie. Deployment nie tworzy podów bezpośrednio — tworzy i zarządza ReplicaSetami, a to one tworzą właściwe pody. Przy każdej zmianie `spec.template` (czyli szablonu poda — najczęściej zmiana obrazu) Deployment nie modyfikuje istniejącego ReplicaSetu, tylko tworzy nowy, z nowym szablonem, i stopniowo przesuwa liczbę replik ze starego ReplicaSetu na nowy, zgodnie ze strategią aktualizacji opisaną niżej. Stary ReplicaSet zostaje w klastrze wyzerowany do `replicas: 0`, a nie usunięty — to właśnie on stanowi historię, do której wraca `kubectl rollout undo`.

Trzy poziomy razem: Deployment decyduje o strategii i wersji, ReplicaSet pilnuje liczby kopii dla konkretnej wersji, Pod to pojedyncza uruchomiona kopia. Zmiana obrazu przechodzi przez wszystkie trzy poziomy: zmieniamy `spec` Deploymentu → Deployment tworzy nowy ReplicaSet → nowy ReplicaSet tworzy nowe pody, a stary ReplicaSet usuwa swoje, w tempie kontrolowanym przez strategię.

Strategia aktualizacji: RollingUpdate i Recreate

> **Pojęcie: strategia aktualizacji.** Sposób, w jaki Deployment przechodzi od starych podów do nowych, gdy zmienia się `spec.template`.

`RollingUpdate` (strategia domyślna) wymienia pody stopniowo, w małych krokach, tak żeby w każdej chwili trwania aktualizacji działała jakaś liczba instancji zdolnych obsłużyć ruch — nigdy zero. Dwa parametry kontrolują tempo tej wymiany:

- `maxSurge` — o ile więcej podów ponad zadeklarowane `replicas` wolno chwilowo utworzyć w trakcie aktualizacji. Wartość `1` przy trzech replikach oznacza, że w szczytowym momencie może działać cztery pody naraz (trzy stare plus jeden nowy, zanim jeden stary

zostanie usunięty).
- **`maxUnavailable`** — ile podów spośród zadeklarowanych `replicas` wolno chwilowo mieć niedostępnych. Wartość `0` oznacza, że nowy pod musi osiągnąć gotowość, zanim jakikolwiek stary zostanie usunięty — bezpieczniejsze, ale wolniejsze; wyższa wartość przyspiesza aktualizację kosztem chwilowo mniejszej liczby dostępnych instancji.

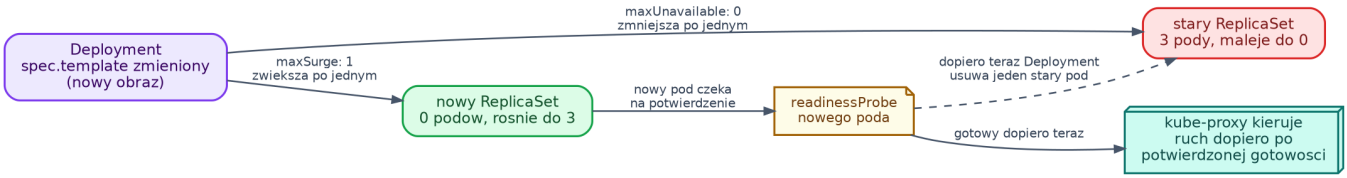
`Recreate` usuwa wszystkie stare pody naraz, i dopiero po ich całkowitym zniknięciu tworzy nowe. Prostsze i szybsze, ale oznacza świadomą przerwę w działaniu na czas między usunięciem starych a wystartowaniem nowych. Sięgamy po nią, gdy stare i nowe wersje aplikacji fizycznie nie mogą działać jednocześnie — najczęściej dlatego, że obie łączą się z tym samym zasobem (np. tą samą tabelą bazy danych) w sposób niekompatybilny między wersjami, i chwilowe współistnienie starych i nowych podów groziłoby uszkodzeniem danych.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: katalog-produktow
  namespace: produkcja
  labels:
    app: katalog-produktow
spec:
  replicas: 3
  selector:
    matchLabels:
      app: katalog-produktow
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: katalog-produktow
    spec:
      containers:
        - name: katalog-produktow
          image: git.zielona.example/zielona/katalog-produktow:2.4.1
          ports:
            - containerPort: 8080
          readinessProbe:
```

```
            httpGet:
              path: /gotowosc
              port: 8080
            periodSeconds: 5
            failureThreshold: 3
          resources:
            requests:
              cpu: "250m"
              memory: "256Mi"
            limits:
              cpu: "500m"
              memory: "512Mi"
```

Przebieg aktualizacji krok po kroku

Zakładamy powyższy Deployment (trzy repliki, `maxSurge: 1`, `maxUnavailable: 0`) i wykonujemy `kubectl set image` na nową wersję obrazu.



Deployment zwiększa nowy ReplicaSet i zmniejsza stary na przemian, a kube-proxy kieruje ruch do nowego poda dopiero po potwierdzeniu jego gotowości (kliknięcie otwiera pełny rozmiar)

1. Deployment tworzy nowy ReplicaSet z nowym szablonem poda, na razie z `replicas: 0`.
2. Ponieważ `maxSurge: 1` pozwala na jednego poda ponad docelową liczbę, Deployment podnosi nowy ReplicaSet do `replicas: 1`, tworząc czwarty pod (trzy stare wciąż działają, jeden nowy startuje).
3. Nowy pod przechodzi przez fazę `Pending` i `ContainerCreating`, po czym `kubelet` uruchamia kontener i zaczyna wykonywać sondę gotowości (patrz strona Pod: najmniejsza jednostka).
4. Dopiero gdy sonda gotowości nowego poda zacznie przechodzić — nie wcześniej — Deployment uznaje go za dostępnego i, ponieważ `maxUnavailable: 0` nie pozwala zejść poniżej trzech dostępnych podów, dopiero teraz zmniejsza stary ReplicaSet o jeden, usuwając jeden stary pod.

5. Sekwencja powtarza się: nowy ReplicaSet rośnie o kolejnego poda, czekamy na jego gotowość, stary ReplicaSet maleje o jednego — aż stary ReplicaSet osiągnie `replicas: 0`, a nowy `replicas: 3`.
6. `kubectl rollout status deployment/katalog-produktow` raportuje `successfully rolled out` w momencie, gdy nowy ReplicaSet ma tyle gotowych replik, ile wynosi `spec.replicas` całego Deploymentu.

W żadnym momencie tej sekwencji liczba dostępnych, gotowych podów nie spadła poniżej trzech — to jest dokładnie to, co rozumiemy pod pojęciem aktualizacji bez przerwy w działaniu, i to jest bezpośrednia zasługa kombinacji strategii `RollingUpdate` z działającą sondą gotowości.

Dlaczego aktualizacja bez sond gotowości oznacza przerwę

Bez zdefiniowanej `readinessProbe` Deployment ma tylko jedną informację o nowym podzie: czy jego kontener w ogóle wystartował (faza `Running`) — nie ma żadnego sposobu dowiedzieć się, czy aplikacja wewnątrz faktycznie skończyła się inicjalizować i jest zdolna odpowiadać na żądania. W tej sytuacji Deployment uznaje nowy pod za gotowy natychmiast po starcie kontenera i od razu zaczyna kierować do niego ruch przez `kube-proxy`, jednocześnie usuwając stary pod z rotacji — mimo że aplikacja wewnątrz może przez kolejne kilka sekund jeszcze się rozgrzewać, wczytywać konfigurację albo łączyć z bazą danych.

Efekt widoczny z zewnątrz to fala błędów `502`/`503` w oknie kilku sekund na każdym kroku aktualizacji, mnożona przez liczbę replik — coś, co formalnie nie jest przerwą w działaniu całej usługi (część replik cały czas coś odpowiada), ale w praktyce jest dokładnie tym samym doświadczeniem dla klienta, co krótka niedostępność. Dodanie sondy gotowości nie zmienia ani jednej linii kodu aplikacji — zmienia wyłącznie to, w którym dokładnie momencie Deployment uzna nowy pod za wart ruchu.

Skalowanie ręczne i automatyczne

Skalowanie ręczne to zmiana pola `spec.replicas`, wykonana poleceniem `kubectl scale` albo edycją manifestu i `kubectl apply` — kontroler ReplicaSet dogania nową wartość dokładnie tym samym mechanizmem uzgadniania, który dogania ją po awarii poda.

> **Pojęcie: HorizontalPodAutoscaler (HPA).** Osobny obiekt, który sam, cyklicznie, odczytuje bieżące zużycie zasobów podów (przez `metrics-server`, ten sam mechanizm, z

> którego korzysta `kubectl top`) i sam zmienia `spec.replicas` Deploymentu w reakcji na obciążenie, w granicach `minReplicas` i `maxReplicas`, bez naszego udziału. Polecenie `kubectl autoscale` z fiszki tworzy dokładnie taki obiekt. HPA rozwiązuje dokładnie ten sam problem skalowania, który opisaliśmy na stronie Po co Kubernetes, skoro mamy Dockera jako jeden z powodów istnienia orkiestracji — z tą różnicą, że decyzję „ile replik teraz potrzeba" podejmuje automat obserwujący metryki, a nie administrator budzony w nocy alertem.

PodDisruptionBudget

> **Pojęcie: PodDisruptionBudget (PDB).** Obiekt deklarujący, ile podów danej aplikacji wolno jednocześnie wyłączyć w ramach zaplanowanej, dobrowolnej operacji — takiej jak opróżnianie węzła przed jego aktualizacją (`kubectl drain`, temat części poświęconej eksploatacji klastra) — bez naruszania minimalnej liczby działających instancji. PDB nie chroni przed awarią (nagłym, niezaplanowanym zniknięciem poda) — chroni wyłącznie przed operacjami, które ktoś świadomie inicjuje i które w innym wypadku mogłyby jednocześnie zabrać zbyt wiele replik naraz, np. wyłączając dwa węzły z rzędu, na których akurat leżały wszystkie trzy repliki jednej usługi.

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: katalog-produktow
  namespace: produkcja
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: katalog-produktow
```

Ten PDB mówi: niezależnie od tego, ile operacji utrzymaniowych prowadzimy jednocześnie, przynajmniej dwa pody z etykietą `app: katalog-produktow` muszą pozostać dostępne w każdej chwili — `kubectl drain` na węźle, który naruszyłby ten warunek, zablokuje się i poczeka, zamiast opróżnić węzeł bezwarunkowo.

Nasz przykład: kolorowy nginx

Dokładnie tę sekwencję RollingUpdate widzimy krok po kroku, gołym okiem, na stronie k3s w laboratorium: pierwszy deploy, nginx w kolorach: trzy repliki prostego nginksa, z których każda losuje inny kolor tła na podstawie własnej nazwy poda. Skalowanie `kubectl scale --replicas=5` dokłada nowe pody z nowymi, wcześniej nieobecnymi kolorami, samoleczenie po `kubectl delete pod` podmienia usunięty pod na nowy z innym kolorem, a `kubectl set image` uruchamia dokładnie tę wymianę starego ReplicaSetu na nowy, opisaną wyżej, tyle że tutaj każdy krok widać jako zmianę koloru zamiast suchego licznika replik. Fragment Deploymentu z tamtej strony:

```
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
```

Częste błędy

Aktualizacja bez sondy gotowości, tłumaczona brakiem czasu na jej dodanie. Krótkoterminowa oszczędność czasu kosztuje falę błędów przy każdej kolejnej aktualizacji, aż do momentu, w którym ktoś w końcu dopisze sondę.

Ustawienie `maxUnavailable` na wartość większą niż zero na usłudze o niskim ruchu i małej liczbie replik. Przy dwóch replikach i `maxUnavailable: 1` aktualizacja chwilowo zostawia tylko jedną działającą kopię — jeśli akurat ona ulegnie awarii w tym oknie, usługa jest chwilowo całkowicie niedostępna, mimo formalnie deklarowanej wysokiej dostępności.

Traktowanie usunięcia starego ReplicaSetu jako usunięcia historii wdrożeń. Stary ReplicaSet wyzerowany do `replicas: 0` wciąż istnieje i jest tym, do czego wraca `kubectl rollout undo` — dopiero przekroczenie limitu `revisionHistoryLimit` (domyślnie 10) powoduje realne usuwanie najstarszych wpisów historii.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Wyjaśniamy, dlaczego Deployment tworzy nowy ReplicaSet zamiast modyfikować istniejący, gdy zmieniamy obraz w `spec.template`.

2. Dla Deploymentu z czterema replikami dobieramy `maxSurge` i `maxUnavailable` tak, żeby aktualizacja nigdy nie zeszła poniżej czterech dostępnych podów, i uzasadniamy wybór strategii `Recreate` albo `RollingUpdate` dla aplikacji, która podczas aktualizacji zmienia niekompatybilnie format danych w wspólnej bazie.
3. Opisujemy krok po kroku, co się dzieje z licznikiem starego i nowego ReplicaSetu w trakcie aktualizacji z `maxSurge: 1`, `maxUnavailable: 0`, przy trzech replikach.
4. Piszemy `PodDisruptionBudget` gwarantujący, że przy dowolnej operacji utrzymaniowej co najmniej 80% replik usługi `katalog-produktow` pozostaje dostępne.

Powiązane: Po co Kubernetes, skoro mamy Dockera · Pod: najmniejsza jednostka · StatefulSet, DaemonSet, Job i CronJob · kubectl: narzędzie codzienne

Źródła: dokumentacja Kubernetes, „Deployments" (kubernetes.io/docs/concepts/workloads/controllers/deployment/) · „Horizontal Pod Autoscaling" (kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/) · „Specifying a Disruption Budget for your Application" (kubernetes.io/docs/tasks/run-application/configure-pdb/)

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
```

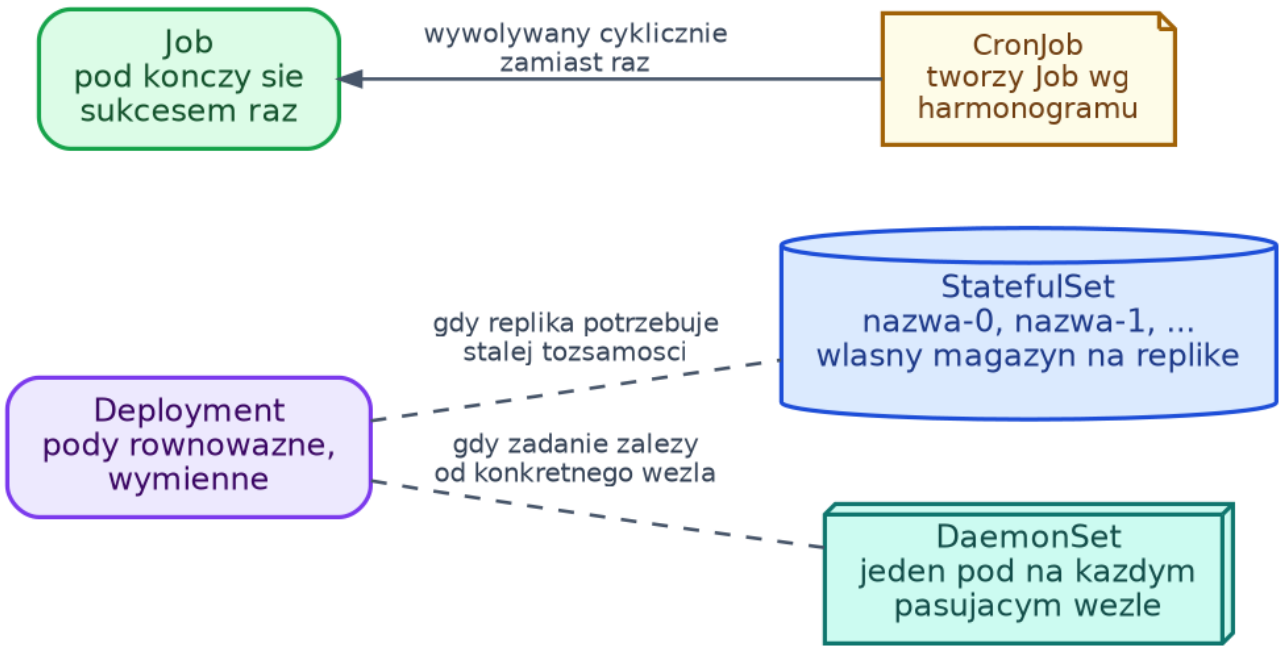
□□ **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

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/)