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.

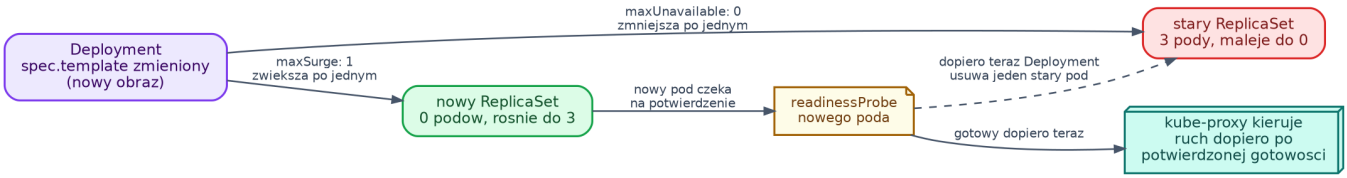
`Recreate` usuwa wszystkie stare pody naraz, i dopiero po ich całkowitym zniknięciu tworzy nowe. Prostsze i szybsze, ale oznacza świadomą przerwę w działaniu na czas między usunięciem starych a wystartowaniem nowych. Sięgamy po nią, gdy stare i nowe wersje aplikacji fizycznie nie mogą działać jednocześnie — 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/)

Revision #34
Created 2026-09-25 10:32:58 UTC by Tomasz Konieczny
Updated 2026-10-26 14:16:59 UTC by Tomasz Konieczny