Konfiguracja i magazyn

ConfigMap, Secret, wolumeny i PersistentVolumeClaim.

- ConfigMap i Secret
- Wolumeny, PV, PVC i StorageClass

ConfigMap i Secret

Poziom: średni **Dotyczy: Kubernetes 1.30+** **Czas: ~20 min**

ConfigMap i Secret oddzielają konfigurację i dane wrażliwe od obrazu kontenera — ta strona pokazuje, jak wstrzykujemy je do poda, czym Secret realnie jest, a czym nie jest, i jak zarządzamy sekretami spoza samego Kubernetesa.

Skróty użyte na tej stronie (11)

Skrót	Co znaczy
AKS	Azure Kubernetes Service: Kubernetes w chmurze Microsoftu, w którym płaszczyznę sterowania utrzymuje dostawca
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
AWS	Amazon Web Services: chmura Amazonu
DER	binarny, nieczytelny dla człowieka format zapisu certyfikatu
EKS	Elastic Kubernetes Service: Kubernetes w chmurze Amazon Web Services, w którym płaszczyznę sterowania utrzymuje dostawca
etcd	etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft
GitOps	Git Operations: sposób prowadzenia klastra, w którym pożądany stan jest zapisany w repozytorium Git, a agent w klastrze sam doprowadza rzeczywistość do tego stanu
KMS	Key Management Service: usługa chmurowa przechowująca i zarządzająca kluczami szyfrującymi
RBAC	Role-Based Access Control: uprawnienia nadawane rolom, a role ludziom
TLS	Transport Layer Security: szyfrowanie połączeń, np. w HTTPS
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Fiszka

ZMIENIA SYSTEM

Utworzenie ConfigMap z pliku

```
kubectl create configmap NazwaUslugi-config --from-file=SciezkaDoPliku -n
NazwaPrzestrzeni
```

Tworzymy ConfigMap, którego klucz to nazwa pliku, a wartość to jego zawartość — szybki sposób bez pisania YAML-a, przydatny przy jednorazowym imporcie istniejącego pliku konfiguracyjnego.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

PLIK KONFIGURACYJNY

Utworzenie ConfigMap deklaratywnie

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: NazwaUslugi-config
  namespace: NazwaPrzestrzeni
data:
  APP_ENV: produkcja
  LOG_LEVEL: info
  app.properties: |
    timeout=30
    retries=3
```

Trzymamy zarówno pojedyncze pary klucz-wartość, jak i cały plik konfiguracyjny jako jedną wartość wieloliniową — oba warianty mogą współistnieć w jednym obiekcie.

PLIK KONFIGURACYJNY

Wstrzyknięcie ConfigMap jako zmiennych środowiskowych: envFrom

```
apiVersion: v1
kind: Pod
metadata:
  name: NazwaPoda
  namespace: NazwaPrzestrzeni
spec:
  containers:
```

```
    - name: NazwaKontenera
      image: NazwaObrazu
      envFrom:
        - configMapRef:
            name: NazwaUslugi-config
```

Wstrzykujemy wszystkie pary klucz-wartość z ConfigMap jako zmienne środowiskowe kontenera naraz, bez wymieniania ich pojedynczo.

□□ PLIK KONFIGURACYJNY

Wstrzyknięcie ConfigMap jako wolumenu z plikami

```
apiVersion: v1
kind: Pod
metadata:
  name: NazwaPoda
  namespace: NazwaPrzestrzeni
spec:
  containers:
    - name: NazwaKontenera
      image: NazwaObrazu
      volumeMounts:
        - name: config
          mountPath: /etc/NazwaUslugi
  volumes:
    - name: config
      configMap:
        name: NazwaUslugi-config
```

Każdy klucz z ConfigMap staje się osobnym plikiem w katalogu `/etc/NazwaUslugi` wewnątrz kontenera — różnica w zachowaniu przy aktualizacji względem zmiennych środowiskowych jest opisana niżej.

□□ ZMIENIA SYSTEM

Utworzenie Secret z linii poleceń

```
kubectl create secret generic NazwaSekretu --from-literal=haslo=NieWpisujHaslaWDokumentacji -n NazwaPrzestrzeni
```

Tworzymy Secret bezpośrednio z linii poleceń, żeby wartość nie trafiła do żadnego pliku YAML w repozytorium — sprawa rozwinięta w sekcji o tym, czego nigdy nie trzymamy w repozytorium.

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

ZMIENIA SYSTEM

Utworzenie Secret z certyfikatem TLS

```
kubectl create secret tls NazwaSekretu --cert=SciezkaDoPliku --key=SciezkaDoPliku -n NazwaPrzestrzeni
```

Tworzymy sekret typu `kubernetes.io/tls`, dokładnie w formacie, jakiego oczekuje pole `tls.secretName` obiektu Ingress opisanego na stronie Ingress i Gateway API: wejście z zewnątrz.

TLS = Transport Layer Security: szyfrowanie połączeń, np. w HTTPS · **API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podejrzenie zawartości Secret: kubectl get secret

```
kubectl get secret NazwaSekretu -n NazwaPrzestrzeni -o jsonpath='{.data.haslo}' | base64 -d
```

Odkodowujemy wartość z Secret z powrotem do postaci czytelnej — ta sama komenda pokazuje, dlaczego samo base64 nigdy nie było mechanizmem ochrony: dekodowanie nie wymaga żadnego klucza ani hasła.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Utworzenie sekretu do pobierania obrazów z prywatnego rejestru

```
kubectl create secret docker-registry NazwaSekretu --docker-server=NazwaHosta --docker-username=nazwa_usera --docker-password=NieWpisujHaslaWDokumentacji -n NazwaPrzestrzeni
```

Tworzymy sekret typu `kubernetes.io/dockerconfigjson`, który podpinamy pod pole `imagePullSecrets` poda, żeby kubelet mógł uwierzytelnić się w prywatnym rejestrze obrazów.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

PLIK KONFIGURACYJNY

Podpięcie imagePullSecrets do poda

```
apiVersion: v1
kind: Pod
metadata:
  name: NazwaPoda
  namespace: NazwaPrzestrzeni
spec:
  imagePullSecrets:
    - name: NazwaSekretu
  containers:
    - name: NazwaKontenera
      image: NazwaObrazu
```

Bez tego pola kubelet próbuje pobrać obraz z prywatnego rejestru anonimowo i kończy to błędem `ImagePullBackOff`, opisanym na stronie Diagnostyka: objaw, przyczyna, pierwsze kroki.

Z życia wzięte

Z ŻYCIA WZIĘTE

O 11:30 Tomek zmienia wartość `LOG_LEVEL` w ConfigMapie aplikacji katalogu produktów, żeby na chwilę włączyć bardziej szczegółowe logi, i czeka na efekt, patrząc na `kubectl logs -f`. Po pięciu minutach nic się nie zmienia, więc sprawdza manifest i widzi, że aplikacja czyta `LOG_LEVEL` jako zwykłą zmienną środowiskową przez `envFrom`, wstrzykiwaną raz przy starcie procesu, a nie z pliku w wolumenie, który aktualizowałby się sam. Dopiero po ręcznym

> usunięciu i odtworzeniu poda nowy proces startuje z nową wartością i szczegółowe logi faktycznie się pojawiają.

Czym jest ConfigMap i po co oddzielamy konfigurację od obrazu

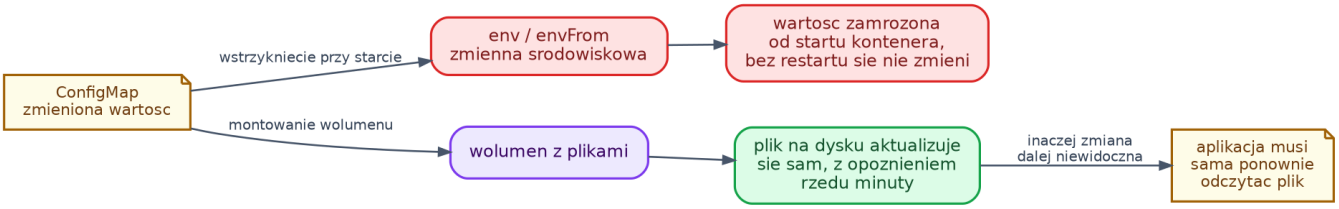
> **Pojęcie: ConfigMap.** Obiekt Kubernetesa przechowujący dowolne dane konfiguracyjne w postaci jawnego tekstu — par klucz-wartość albo całych plików — niezależnie od obrazu kontenera, który z nich korzysta.

Oddzielenie konfiguracji od obrazu ma jeden konkretny cel: ten sam obraz, zbudowany raz, trafia bez zmian na środowisko testowe i produkcyjne, różniąc się wyłącznie ConfigMapą podpiętą przy uruchomieniu. Bez tego rozdzielenia każda zmiana adresu bazy danych albo poziomu logowania wymagałaby przebudowania obrazu od nowa — a to jest dokładnie to, czego obraz kontenera jako niezmienny artefakt ma unikać.

Sposoby wstrzyknięcia i różnica przy aktualizacji

Trzy fiszki wyżej pokazują trzy drogi, którymi zawartość ConfigMap trafia do kontenera, i każda z nich zachowuje się inaczej przy zmianie danych po fakcie:

Sposób	Zachowanie przy aktualizacji ConfigMap
pojedyncza zmienna środowiskowa (`env` + `configMapKeyRef`)	wartość zamrożona na moment startu kontenera, nie odświeża się nigdy bez restartu poda
`envFrom`	tak samo jak wyżej — cały zestaw zmiennych zamrożony na starcie
wolumen z plikami	plik na dysku wewnątrz kontenera aktualizuje się automatycznie, z typowym opóźnieniem rzędu minuty, bez restartu poda



ConfigMap wstrzyknięty jako zmienna środowiskowa zamraża wartość na starcie kontenera, a wstrzyknięty jako wolumen aktualizuje się sam na dysku, choć aplikacja musi go i tak ponownie odczytać (kliknięcie otwiera pełny rozmiar)

To rozróżnienie ma bezpośrednie znaczenie praktyczne: aplikacja, która czyta konfigurację raz przy starcie procesu (typowe dla zmiennych środowiskowych) nie zobaczy zmiany nawet w wolumenie, dopóki proces się nie zrestartuje — mechanizm aktualizacji pliku na dysku nie pomaga, jeśli nikt tego pliku nie czyta ponownie. Aplikacje zaprojektowane do obserwowania zmian pliku konfiguracyjnego w locie (np. przez `inotify`) korzystają z wolumenu właśnie po to, żeby dostać nową konfigurację bez restartu.

Uważaj: to opóźnienie rzędu minuty przy aktualizacji przez wolumen wynika z okresu synchronizacji kubeletu, nie jest natychmiastowe — plan zakładający propagację zmiany „od razu” po `kubectl apply` na ConfigMapę bywa źródłem fałszywego alarmu, gdy ktoś sprawdza efekt sekundę po zmianie.

Czym jest Secret i czym nie jest

> **Pojęcie: Secret.** Obiekt Kubernetesa o tej samej strukturze co ConfigMap, przeznaczony do przechowywania danych wrażliwych — haseł, tokenów, kluczy prywatnych, certyfikatów. Różnica względem ConfigMap na poziomie samego API sprowadza się właściwie do jednego: wartości w polu `data` są zakodowane base64.

To jest najważniejsze zdanie tej strony: base64 to nie szyfrowanie. Base64 to odwracalne kodowanie binarnych danych do postaci tekstowej, bez żadnego klucza ani hasła po drugiej stronie — każdy, kto widzi zawartość obiektu Secret (przez `kubectl get secret -o yaml`, przez dostęp do etcd, przez backup etcd), odczytuje wartość jedną komendą `base64 -d`, dokładnie tak jak w fiszce wyżej. Kodowanie base64 istnieje wyłącznie po to, żeby dane binarne (np. certyfikat w formacie DER) dało się bezpiecznie zapisać w polu tekstowym JSON-a czy YAML-a, nie po to, żeby cokolwiek ukryć.

Szyfrowanie sekretów w etcd

> **Pojęcie: szyfrowanie danych w spoczynku (encryption at rest).** Mechanizm, w którym kube-apiserver szyfruje wartość obiektu Secret, zanim zapisze ją do etcd, i odszyfrowuje ją z powrotem tylko dla uprawnionego żądania przez API.

Bez tego mechanizmu Secret leży w bazie etcd w postaci zakodowanej wyłącznie base64 — czyli praktycznie jawnym tekstem dla każdego, kto dostanie dostęp do plików etcd albo do jego kopii zapasowej, opisanej na stronie Utrzymanie klastra, z pominięciem całego RBAC-u Kubernetesa. Szyfrowanie sekretów w etcd nie jest w Kubernetesie domyślnie włączone — wymaga jawnej konfiguracji `EncryptionConfiguration` na kube-apiserverze, wskazującej dostawcę szyfrowania (najprostszy to `aescbc`, zalecany produkcyjnie to zewnętrzny dostawca KMS przez `KMSv2`, w którym klucz szyfrujący nigdy nie trafia na dysk API servera). Zarządzane usługi Kubernetesa w chmurze (EKS, GKE, AKS) często oferują tę opcję jako włączaną funkcję, ale również nie zawsze domyślnie aktywną — sprawdzenie tego ustawienia jest jedną z pierwszych rzeczy, które warto zweryfikować w każdym nowym klastrze, niezależnie od tego, kto go postawił.

Zewnętrzne zarządzanie sekretami

Trzymanie sekretów bezpośrednio jako obiekty Secret w Kubernetesie ma dwa praktyczne ograniczenia: sam plik YAML z sekretem nie nadaje się do repozytorium Git (nawet zakodowany base64, z przyczyn wyżej), a rotacja hasła wymaga ręcznej aktualizacji obiektu w każdym klastrze z osobna. Dwa popularne narzędzia rozwiązują to inaczej, każde inny problem.

> **Pojęcie: External Secrets Operator.** Kontroler, który synchronizuje sekrety z zewnętrznego magazynu (HashiCorp Vault, AWS Secrets Manager, Azure Key Vault i inne) do obiektów Secret wewnątrz klastra, na podstawie obiektu `ExternalSecret` wskazującego, którego wpisu z zewnętrznego magazynu użyć. Rozwiązuje problem miejsca prawdy: hasło żyje i jest rotowane w jednym centralnym magazynie sekretów, a Kubernetes dostaje tylko jego aktualną kopię, odświeżaną automatycznie w ustalonym interwale.

```
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: NazwaSekretu
  namespace: NazwaPrzestrzeni
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: vault-backend
    kind: ClusterSecretStore
```

```
  target:
    name: NazwaSekretu
  data:
    - secretKey: haslo
      remoteRef:
        key: sklep/produkcja/baza
        property: haslo
```

> **Pojęcie: Sealed Secrets.** Mechanizm od Bitnami, który rozwiązuje inny problem — możliwość trzymania sekretu w repozytorium Git obok reszty manifestów. Sekret jawny szyfrujemy narzędziem `kubeseal` kluczem publicznym kontrolera działającego w klastrze, zapisujemy zaszyfrowany wynik jako obiekt `SealedSecret` (bezpieczny do trzymania w repozytorium, bo odszyfrować go potrafi tylko kontroler z odpowiadającym mu kluczem prywatnym), a kontroler w klastrze sam odszyfrowuje go z powrotem do zwykłego obiektu Secret.

Różnica między tymi dwoma podejściami sprowadza się do tego, gdzie leży źródło prawdy: przy External Secrets Operator jest nim zewnętrzny magazyn sekretów poza Gitem, a Sealed Secrets pozwala trzymać zaszyfrowaną wersję sekretu w tym samym repozytorium Git, w którym trzymamy resztę konfiguracji GitOps.

Czego nigdy nie trzymamy w repozytorium

- jawnego obiektu `Secret` w YAML-u, nawet zakodowanego base64 — dekodowanie nie wymaga żadnego sekretu
- hasła ani tokenu wpisanego wprost w `ConfigMap`, choćby tymczasowo „do testów”
- pliku `kubeconfig` z wbudowanym tokenem dostępu do klastra produkcyjnego
- klucza prywatnego certyfikatu TLS w żadnej postaci, nawet w gałęzi roboczej

Częste błędy

Traktowanie base64 jako ochrony. Sekret wklejony do publicznego repozytorium „bo i tak jest zakodowany” jest w praktyce jawnym hasłem — dekodowanie to jedna komenda bez żadnego klucza.

Brak włączonego szyfrowania w etcd na klastrze on-premise. Domyślna instalacja `kubeadm` nie włącza szyfrowania sekretów w etcd automatycznie — warto to sprawdzić explicité, zamiast zakładać, że Kubernetes robi to sam z siebie.

ConfigMap zamiast Secret dla wartości wrażliwej. ConfigMap nie ma żadnego mechanizmu kontroli dostępu innego niż zwykły RBAC na poziomie obiektu i nie jest traktowany przez narzędzia audytujące jako miejsce, w którym szuka się poświadczeń — hasło wpisane tam z przyzwyczajenia umyka przeglądom bezpieczeństwa skupionym na obiektach `Secret`.

□□ Ćwiczenia

> Ćwiczenia zakładają dostęp do klastra testowego z uprawnieniami do tworzenia obiektów w wybranej przestrzeni nazw.

Ćwiczenie 1 — dekodowanie Secret

```
kubectl create secret generic test-secret --from-literal=klucz=wartosc-testowa -n
NazwaPrzestrzeni
kubectl get secret test-secret -n NazwaPrzestrzeni -o jsonpath='{.data.klucz}'
kubectl get secret test-secret -n NazwaPrzestrzeni -o jsonpath='{.data.klucz}' | base64 -d
```

Tworzymy testowy Secret i odkodowujemy jego wartość bez żadnych dodatkowych uprawnień poza zwykłym odczytem obiektu.

Odpowiedź

Sama komenda `base64 -d` w pełni odtwarza oryginalną wartość — potwierdza to, że jedyną ochroną Secret jest RBAC kontrolujący dostęp do samego obiektu, a nie treść pola `data`.

Pytania kontrolne

1. Dlaczego aplikacja czytająca konfigurację ze zmiennej środowiskowej nie zauważy zmiany w ConfigMap bez restartu, mimo że plik zamontowany jako wolumen zaktualizowałby się sam?

Odpowiedź

Zmienne środowiskowe są wstrzykiwane raz, w momencie startu procesu w kontenerze, i system operacyjny nie ma mechanizmu ich odświeżenia w działającym procesie — plik na dysku aktualizuje kubelet niezależnie od procesu aplikacji, ale to i tak aplikacja musi go ponownie odczytać, żeby zobaczyć zmianę.

2. Jaki problem rozwiązuje Sealed Secrets, którego nie rozwiązuje samo szyfrowanie sekretów w etcd?

Odpowiedź

Szyfrowanie w etcd chroni dane już zapisane w klastrze przed odczytem z samego etcd, ale nie pozwala bezpiecznie trzymać sekretu w repozytorium Git przed jego wdrożeniem. Sealed Secrets rozwiązuje właśnie ten drugi problem — zaszyfrowana wersja w Gicie jest bezużyteczna bez klucza prywatnego kontrolera w konkretnym klastrze.

Powiązane: Ingress i Gateway API: wejście z zewnątrz · RBAC i bezpieczeństwo klastra · Diagnostyka: objaw, przyczyna, pierwsze kroki · Utrzymanie klastra

Źródła: dokumentacja kubernetes.io, rozdziały „Secrets", „ConfigMaps" i „Encrypting Confidential Data at Rest" · dokumentacja external-secrets.io i Bitnami Sealed Secrets (github.com/bitnami-labs/sealed-secrets)

Wolumeny, PV, PVC i StorageClass

Poziom: zaawansowany **Dotyczy: Kubernetes 1.30+** **Czas: ~25 min**

Kubernetes rozdziela wolumen fizycznie istniejący w infrastrukturze od jego żądania przez aplikację — ta strona pokazuje, po co to rozdzielenie istnieje i jak z niego korzystamy krok po kroku, od `emptyDir` *do dynamicznego udostępniania przez StorageClass.*

Skróty użyte na tej stronie (8)

Skrót	Co znaczy
CNI	Container Network Interface: wtyczka, która daje sieć podom w Kubernetesie
CSI	Container Storage Interface: wtyczka, która daje dyski podom w Kubernetesie
HDD	Hard Disk Drive: tradycyjny dysk twardy z wirującymi talerzami i głowicą, wolniejszy od SSD, ale tańszy za gigabajt, typowo używany jako warstwa pojemnościowa
NFS	Network File System: udostępnianie katalogów po sieci w świecie Linuksa
PV	Physical Volume: dysk lub partycja oddana pod LVM; w Kubernetesie PersistentVolume, trwały wolumen
PVC	PersistentVolumeClaim: zamówienie trwałego dysku przez aplikację w Kubernetesie
REST	sposób budowania API przez zwykłe żądania HTTP (GET, POST) zamiast osobnego protokołu
SSD	Solid State Drive: dysk półprzewodnikowy

Fiszka

PLIK KONFIGURACYJNY

Utworzenie poda z emptyDir

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: NazwaPoda
spec:
  containers:
    - name: NazwaKontenera
      image: NazwaObrazu
      volumeMounts:
        - name: cache
          mountPath: /cache
  volumes:
    - name: cache
      emptyDir: {}
```

Tworzymy wolumen istniejący tylko przez cały czas życia poda, dzielony między kontenerami tego samego poda — znika bezpowrotnie razem z podem.

□□ TYLKO ODCZYT

Wylistowanie PersistentVolume w klastrze: kubectl get pv

```
kubectl get pv
```

Sprawdzamy, jakie woluminy fizyczne są zarejestrowane w klastrze, jaki mają rozmiar, tryb dostępu i status — kolumna `STATUS` mówi, czy wolumen jest wolny, czy już przypisany do jakiegoś żądania.

Przykład:

```
NAME                CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                     STORAGECLASS   AGE
pv-baza-danych-01   20Gi       RWO            Retain           Bound    produkcja/baza-
danych-pvc   nfs-baza      90d
```

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

□□ PLIK KONFIGURACYJNY

Utworzenie PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: NazwaUslugi-pvc
  namespace: NazwaPrzestrzeni
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: NazwaKlasy
  resources:
    requests:
      storage: 10Gi
```

Zgłaszamy żądanie przestrzeni dyskowej o określonym rozmiarze i trybie dostępu, nie dotykając w ogóle tego, gdzie fizycznie ten dysk leży — o to dbają PersistentVolume i StorageClass opisane niżej.

□□ PLIK KONFIGURACYJNY

Podpięcie PVC do poda

```
apiVersion: v1
kind: Pod
metadata:
  name: NazwaPoda
  namespace: NazwaPrzestrzeni
spec:
  containers:
    - name: NazwaKontenera
      image: NazwaObrazu
      volumeMounts:
        - name: dane
          mountPath: /var/lib/dane
  volumes:
    - name: dane
      persistentVolumeClaim:
        claimName: NazwaUslugi-pvc
```

Montujemy przestrzeń z PVC pod konkretną ścieżką wewnątrz kontenera — z punktu widzenia aplikacji wygląda to jak zwykły katalog na dysku.

PVC = PersistentVolumeClaim: zamówienie trwałego dysku przez aplikację w Kubernetesie

TYLKO ODCZYT

Wylistowanie dostępnych klas magazynowania: kubectl get storageclass

```
kubectl get storageclass
```

Sprawdzamy, jakie klasy magazynowania są zdefiniowane w klastrze i która z nich jest domyślna — domyślną poznajemy po adnotacji widocznej w nazwie z gwiazdką w tym poleceniu.

Przykład:

```
NAME                 PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE   AGE
standard (default)   ebs.csi.aws.com         Delete          WaitForFirstConsumer   120d
nfs-baza             nfs.csi.k8s.io          Retain          Immediate              120d
```

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Rozszerzenie wolumenu: kubectl edit / kubectl patch

```
kubectl patch pvc NazwaUslugi-pvc -n NazwaPrzestrzeni --type merge -p
'{"spec":{"resources":{"requests":{"storage":"20Gi"}}}}'
```

Zwiększamy zadeklarowany rozmiar PVC — czy operacja faktycznie powiększy dysk bez przestoju, zależy od tego, czy wybrana StorageClass wspiera rozszerzanie, opisane niżej.

PVC = PersistentVolumeClaim: zamówienie trwałego dysku przez aplikację w Kubernetesie · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Sprawdzenie statusu PVC po rozszerzeniu

```
kubectl get pvc NazwaUslugi-pvc -n NazwaPrzestrzeni -o jsonpath='{.status.conditions}'
```

Sprawdzamy, czy operacja rozszerzenia zakończyła się (`FileSystemResizePending` oznacza, że dysk urósł, ale system plików wewnątrz poda jeszcze czeka na zmianę rozmiaru).

PVC = PersistentVolumeClaim: zamówienie trwałego dysku przez aplikację w Kubernetesie · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 12:15 druga replika bazy kolejki nie chce wystartować na klastrze produkcyjnym i wisi w fazie `Pending` mimo wolnych zasobów na wszystkich węzłach. Tomek sprawdza zdarzenia poda i widzi komunikat o niemożności zamontowania wolumenu, bo PVC ma tryb dostępu `ReadWriteOnce`, a druga replika trafiła planistą na inny węzeł niż pierwsza. Po zmianie magazynu na obsługujący `ReadWriteMany` obie repliki montują ten sam wolumen jednocześnie, bez żadnego dalszego błędu.

Czym jest wolumen w Kubernetesie

Pojęcie: wolumen (volume). Katalog z danymi dostępny wewnątrz jednego lub kilku kontenerów danego poda, o czasie życia i pochodzeniu zależnym od tego, jakiego typu wolumenu użyjemy. To jest inne pojęcie niż wolumen Dockera: wolumen Dockera żyje niezależnie od kontenera na tym samym hoście i pozostaje po jego usunięciu, dopóki ktoś go jawnie nie usunie. Wolumen Kubernetesa w wersji podstawowej (`emptyDir`) żyje dokładnie tak długo jak pod, do którego jest przypięty, i znika razem z nim — trwałość danych ponad cykl życia jednego poda wymaga świadomego sięgnięcia po PersistentVolume, opisany niżej, a nie jest domyślną cechą samego pojęcia wolumenu.

emptyDir

Pojęcie: emptyDir. Pusty katalog tworzony w momencie startu poda i usuwany bezpowrotnie w momencie jego usunięcia — restart kontenera wewnątrz tego samego poda nie kasuje `emptyDir`, ale usunięcie całego poda tak. Typowe zastosowanie to przestrzeń robocza dzielona między kilkoma kontenerami tego samego poda (np. kontener główny i sidecar czytający jego logi) albo bufor na dane tymczasowe, których utrata nie ma żadnych konsekwencji.

hostPath

Pojęcie: hostPath. Wolumen wskazujący bezpośrednio na katalog istniejący na dysku węzła, na którym akurat działa pod. Dane w `hostPath` przeżywają usunięcie poda, ale tylko dopóty, dopóki kolejny pod tej samej aplikacji trafi na ten sam węzeł — scheduler nie ma żadnego obowiązku umieszczać kolejnych replik zawsze na tym samym węźle, więc pod zrestartowany na innym węźle widzi pusty katalog, mimo że dane fizycznie leżą gdzie indziej. `hostPath` na produkcji jest zwykle błędem z dwóch powodów naraz: łamie założenie, że pod może wylądować na dowolnym węźle bez utraty danych, i otwiera furtkę bezpieczeństwa — kontener z dostępem do `hostPath` wskazującego na wrażliwy katalog systemowy węzła (np. `/etc` albo gniazdo dockerowe) potrafi wpłynąć na cały węzeł, nie tylko na siebie.

PersistentVolume i PersistentVolumeClaim

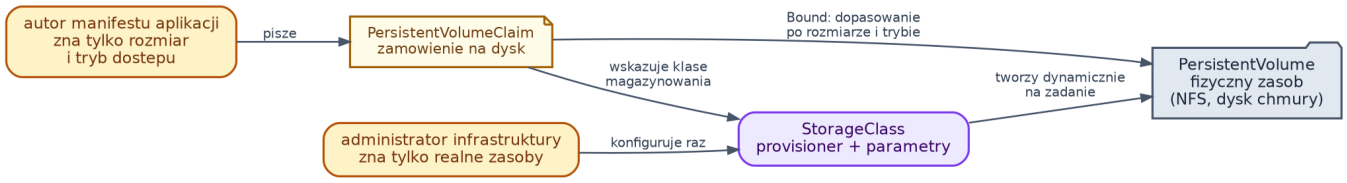
To jest sedno tej strony: Kubernetes celowo rozdziela dwie role, które w prostszych systemach są jedną rzeczą.

Pojęcie: PersistentVolume (PV). Obiekt reprezentujący fizycznie istniejącą przestrzeń dyskową — dysk w chmurze, zasób NFS, wolumen CSI — zarejestrowaną w klastrze jako zasób dostępny do użycia. PV tworzy zwykle administrator klastra (albo, w wariancie dynamicznym opisanym niżej, tworzy go automatycznie kontroler StorageClass) i istnieje niezależnie od jakiegokolwiek konkretnego poda czy aplikacji.

Pojęcie: PersistentVolumeClaim (PVC). Obiekt reprezentujący żądanie przestrzeni dyskowej zgłoszone przez aplikację — ile miejsca potrzebuje, w jakim trybie dostępu,

> ewentualnie z jakiej klasy magazynowania. PVC nie jest samym dyskiem, tylko zamówieniem na dysk.

Rozdzielenie tych dwóch ról ma konkretny cel organizacyjny: osoba pisząca manifest aplikacji (PVC) nie musi wiedzieć nic o infrastrukturze pod spodem — nie zna adresu serwera NFS, nie zna identyfikatora woluminu w chmurze, nie wie, czy dysk jest SSD czy HDD. Zna tylko to, ile miejsca potrzebuje i jak bardzo potrzebuje go współdzielić. Administrator infrastruktury po drugiej stronie zarządza realnymi zasobami (PV) i nie musi wiedzieć nic o konkretnych aplikacjach, które z nich skorzystają. Kubernetes w środku dopasowuje jedno do drugiego (`Bound`) na podstawie zgodności rozmiaru, trybu dostępu i klasy magazynowania — dokładnie tak jak zamówienie w sklepie (PVC) dopasowuje się do towaru na półce (PV) bez potrzeby, żeby klient wiedział, z której hurtowni towar przyjechał.



Autor manifestu pisze tylko PVC, administrator konfiguruje tylko StorageClass, a Kubernetes dopasowuje PVC do PV automatycznie (kliknięcie otwiera pełny rozmiar)

StorageClass i dynamiczne udostępnianie

> **Pojęcie: StorageClass.** Obiekt opisujący jeden rodzaj przestrzeni dyskowej dostępny w danej infrastrukturze wraz z parametrami jej tworzenia — i, co ważniejsze, wskazujący sterownik (provisioner), który potrafi taki dysk utworzyć na żądanie, automatycznie, bez ręcznej pracy administratora.

Bez StorageClass administrator musiałby ręcznie tworzyć obiekt PV za każdym razem, gdy aplikacja potrzebuje nowego kawałka przestrzeni dyskowej — podejście, które działa przy kilku aplikacjach, i przestaje skalować się przy setkach. Z dynamicznym udostępnianiem PVC odwołujący się do danej StorageClass sam wyzwala utworzenie odpowiadającego mu PV w tle, w momencie zgłoszenia żądania — administrator konfiguruje StorageClass raz, a każdy kolejny PVC obsługuje się sam.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: NazwaKlasy
provisioner: nfs.csi.k8s.io
reclaimPolicy: Retain
allowVolumeExpansion: true
parameters:
  server: NazwaSerweraNFS
  share: /export/dane
```

> (gdzie `NazwaSerweraNFS` to serwer udostępniający zasób NFS dla klastra; nie wskazujemy tu `bkp01`, bo to serwer kopii, a nie magazyn danych aplikacji: kopie z klastrów trafiają na niego przez restic i rest-server, z kontem REST nazwanym jak repozytorium, według strony Serwery kopii bkp01 i bkp02: restic, kopia krzyżowa i niezmienność)

Tryby dostępu

> **Pojęcie: tryb dostępu (access mode).** Deklaracja tego, ilu węzłom naraz i w jakim trybie wolno zamontować dany wolumen — nie jest to uprawnienie systemu plików w rozumieniu `chmod`, tylko ograniczenie na poziomie samej infrastruktury magazynowej.

Tryb	Co oznacza	Konsekwencja dla skalowania
`ReadWriteOnce` (RWO)	odczyt i zapis z jednego węzła naraz	wiele podów na tym samym węźle mogą go współdzielić, ale przeniesienie na inny węzeł wymaga odmontowania z poprzedniego — częsta przyczyna, dla której pod utyka w `Pending` przy próbie przeniesienia
`ReadOnlyMany` (ROX)	odczyt z wielu węzłów naraz, bez zapisu	dobre do danych współdzielonych i niezmiennych, np. statycznych zasobów serwowanych przez wiele replik
`ReadWriteMany` (RWX)	odczyt i zapis z wielu węzłów naraz	wymaga systemu plików faktycznie wspierającego dostęp współbieżny (NFS, CephFS) — większość dysków blokowych w chmurze tego nie oferuje

Tryb	Co oznacza	Konsekwencja dla skalowania
`ReadWriteOncePod` (RWOP)	odczyt i zapis, ale z dokładnie jednego poda w całym klastrze	ściślejsza wersja RWO, wprowadzona żeby wykluczyć nawet przypadkowe współdzielenie między dwoma podami na tym samym węźle — istotne dla aplikacji, które same zakładają wyłączność na zapis (część silników baz danych)

Uważaj: aplikacja skalowana do kilku replik z wolumenem w trybie `ReadWriteOnce` nie skaluje się poziomo tak, jak można by się spodziewać — druga i kolejne repliki na innym węźle utykają w `Pending`, dopóki nie trafią na ten sam węzeł co pierwsza, albo dopóki aplikacja nie przejdzie na magazyn wspierający `ReadWriteMany`.

Polityka odzyskiwania

> **Pojęcie: polityka odzyskiwania (reclaim policy).** Decyzja o tym, co dzieje się z PersistentVolume w momencie, gdy powiązany z nim PVC zostaje usunięty. `Delete` usuwa też fizyczny wolumen w infrastrukturze pod spodem — wygodne dla danych efemerycznych, niebezpieczne dla danych, których utrata boli. `Retain` zostawia wolumen i jego dane nietknięte, oznaczając go jako `Released` — wymaga ręcznej decyzji administratora, co dalej (ponowne podłączenie do nowego PVC albo ręczne usunięcie), ale nigdy nie kasuje danych automatycznie. Domyślna polityka zależy od tego, czy PV powstał ręcznie (`Retain`) czy dynamicznie przez StorageClass (najczęściej `Delete`, chyba że StorageClass jawnie ustawia inaczej) — warto to sprawdzić świadomie dla każdej klasy magazynowania używanej do danych, których nie da się łatwo odtworzyć.

Rozszerzenie wolumenu

Rozszerzenie PVC, pokazane w Fiszce, działa tylko wtedy, gdy StorageClass ma ustawione `allowVolumeExpansion: true` — bez tego pola żądanie zwiększenia rozmiaru PVC zostaje odrzucone od razu. Po udanym rozszerzeniu na poziomie infrastruktury dysk często wymaga jeszcze rozszerzenia systemu plików wewnątrz niego, co przy nowoczesnych sterownikach CSI dzieje się automatycznie przy najbliższym zamontowaniu (status `FileSystemResizePending` sygnalizuje właśnie ten pośredni stan) — samo powiększenie dysku po stronie infrastruktury nie oznacza jeszcze, że aplikacja widzi więcej wolnego miejsca.

CSI w skrócie

> **Pojęcie: CSI (Container Storage Interface).** Standardowy interfejs, przez który Kubernetes komunikuje się z dowolnym systemem magazynowania — dyskiem w chmurze, macierzą on-premise, systemem rozproszonym — bez potrzeby wbudowywania kodu obsługi każdego z nich bezpośrednio w Kubernetesa. Sterownik CSI to zewnętrzny komponent, instalowany osobno dla każdego rodzaju magazynu, który tłumaczy uniwersalne żądania Kubernetesa (utwórz wolumen, zamontuj, rozszerz) na konkretne operacje danej infrastruktury. To dokładnie ten sam mechanizm rozszerzalności co CNI dla sieci, opisany na stronie Sieć klastra i polityki sieciowe, tylko dla magazynu danych.

Częste błędy

Oczekiwanie, że hostPath przetrwa przeniesienie poda na inny węzeł. Dane fizycznie zostają na starym węźle — nowy pod na innym węźle widzi pusty katalog, mimo że nic nie zgłasza błędu.

PVC w trybie ReadWriteOnce podpięty do aplikacji skalowanej poziomo na wielu węzłach. Pierwsza replika startuje poprawnie, każda kolejna na innym węźle utyka w `Pending` z komunikatem o niemożności zamontowania wolumenu, opisanym na stronie Diagnostyka: objaw, przyczyna, pierwsze kroki.

Reclaim policy Delete na danych, których nie da się odtworzyć. Usunięcie PVC w ramach porządkowania środowiska kasuje bezpowrotnie także fizyczny dysk — warto świadomie ustawiać `Retain` dla każdej klasy magazynowania obsługującej dane produkcyjne, zamiast polegać na domyślnym zachowaniu sterownika.

□□ Ćwiczenia

> Ćwiczenia zakładają klaster testowy z zainstalowanym sterownikiem CSI i co najmniej jedną StorageClass wspierającą dynamiczne udostępnianie.

Ćwiczenie 1 — cykl życia PVC i PV

```
kubectl apply -f pvc-test.yaml
kubectl get pvc,pv
kubectl delete pvc test-pvc
kubectl get pv
```

Tworzymy PVC, obserwujemy automatycznie utworzony PV i sprawdzamy, co się z nim dzieje po usunięciu PVC, w zależności od ustawionej polityki odzyskiwania.

Odpowiedź

Przy `reclaimPolicy: Delete` PV znika razem z PVC, a fizyczny dysk w infrastrukturze zostaje skasowany. Przy `reclaimPolicy: Retain` PV zostaje w stanie `Released`, dane pozostają nietknięte, ale nowy PVC nie podłączy się do niego automatycznie bez ręcznej interwencji.

Pytania kontrolne

1. Dlaczego administrator infrastruktury i autor manifestu aplikacji nie muszą znać nawzajem swoich szczegółów, mimo że oboje zajmują się tym samym dyskiem?

Odpowiedź

Bo PV i PVC to dwie osobne warstwy abstrakcji — PVC opisuje wyłącznie potrzebę (rozmiar, tryb dostępu), a PV opisuje wyłącznie ofertę (konkretny zasób infrastruktury). Kubernetes dopasowuje jedno do drugiego, więc żadna ze stron nie musi znać szczegółów drugiej.

2. Dlaczego aplikacja z trzema replikami i wolumenem ReadWriteOnce nie skaluje się tak, jak oczekiwano?

Odpowiedź

ReadWriteOnce pozwala na zapis z jednego węzła naraz — druga i trzecia replika na innych węzłach nie mogą zamontować tego samego wolumenu i utykają w stanie `Pending`, dopóki nie trafią na ten sam węzeł co pierwsza replika albo dopóki magazyn nie zostanie zmieniony na wspierający ReadWriteMany.

Powiązane: Sieć klastra i polityki sieciowe · Diagnostyka: objaw, przyczyna, pierwsze kroki

Źródła: dokumentacja kubernetes.io, rozdziały „Volumes", „Persistent Volumes", „Storage Classes" i „Volume Snapshots" · dokumentacja Container Storage Interface (kubernetes-csi.github.io)