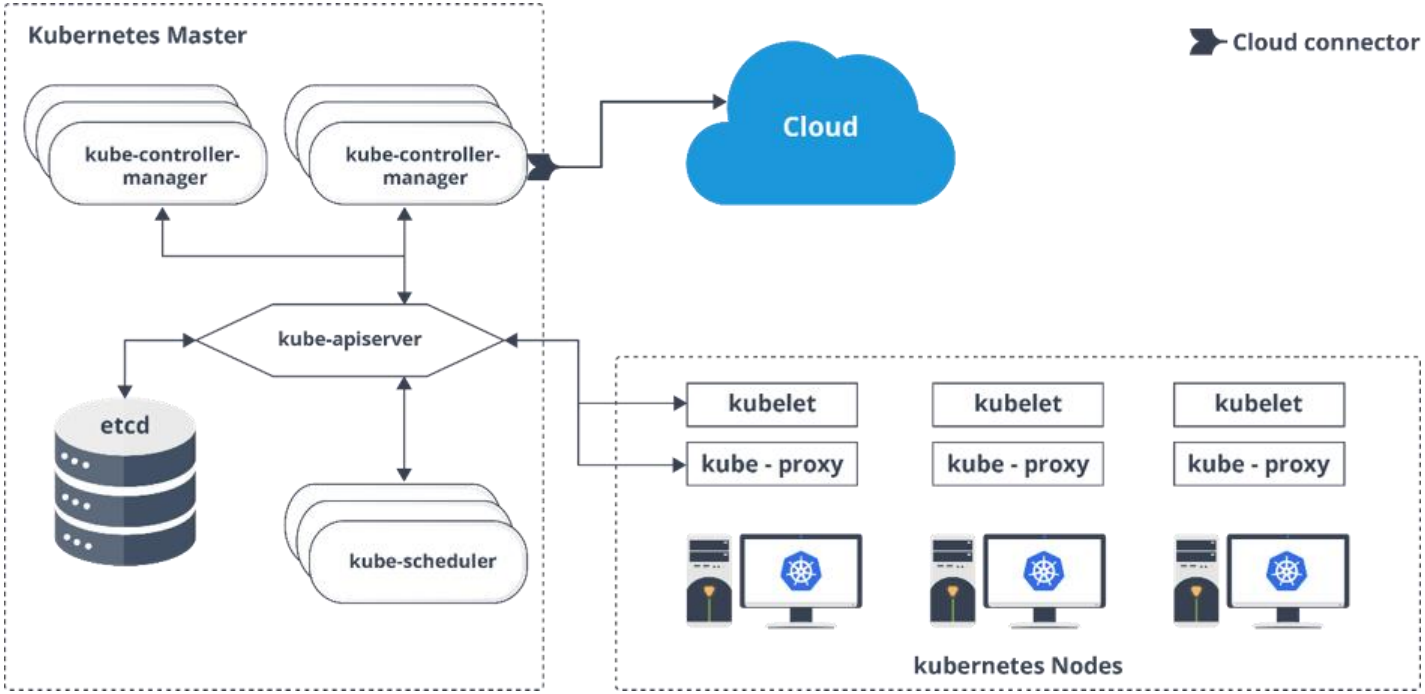
Kubernetes w praktyce

www.kubernetes.io



Strukruta Kubernetes

Control plane (master node):

- monitoruje stan klastra i reaguje na wydarzenia w klastrze
- rozplanowuje miejsce uruchomienia kontenerów (wybiera odpowiedni node itp.), steruje replikami, strategią RollingUpdate itp.

Data plane (worker nodes):

- działają tu aplikacje, którymi steruje Control plane
- aplikacje te zawarte są w Podach (najmniejszych jednostkach jakimi steruje Kubernetes)
- działa tu container runtime (od jakiegoś czasu domyślnie ContainerD) CRI – Container Runtime Interface

Kubernetes – Control Plane

W klastrze **HA** minimalna ilość Control Plane wynosi 3 osobne serwery.

Etcd – baza danych wewnątrz Contol plane, zapisuje stan klastra m.in.
jakie pody powinny działać w klastrze, informacje o nodach itp.
Etcd wymaga qourum do poprawnego działania w klastrze.
Quorum: (n/2)-1; n to liczba nodów

Apiserver – komunikuje się z nim użytkownik poprzez kubectl albo bezpośrednio za pomocą np. curl. Jest to API Rest.
Z apiserverem komunikuje się także każdy z nodów poprzez kubelet (kubelet występuje na każdym nodzie klastra).

Kubernetes – Control Plane

W klastrze **HA** minimalna ilość Control Plane wynosi 3 osobne serwery.

Controler Manager – kontroluje kilka innych kontrolerów ;)
np. node controller – kontroler stanu nodów
deployment controller – do kontroli obiektów typu deployment
Nieustannie sprawdza stan całego klastra i ma za zadanie doprowadzić stan klastra do oczekiwanego przez nas stanu.
Wykrywa również awarie poszczególnych nodów w klastrze.

Cloud controller manager – działa podobnie do Conroller Managera w środowiskach typu cloud.

Kubernetes – Control Plane

Kube Scheduler – przypisuje Pody do poszczególnych nodów na podstawie dostępności nodów, etykiet, warunków itp. Oczekuje na nowe zadania od apiservera.

Informacje do jakiego klastra jesteśmy podłączeni możemy uzyskać za pomocą komendy:
kubectl cluster-info
Kubernetes **control plane** is running at https://192.168.39.3:8443
KubeDNS is running at https://192.168.39.3:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

Kubernetes – Data Plane

Elementy Data Plane:

Kubelet – agent działający na każdym z nodów. Informuje o stanie podów do Control Plane. Zajmuje się pobraniem obrazu kontenera i uruchamianiem kontenerów, volumenów w Podach.

Kube-proxy – agent działający na każdym z nodów odpowiadający za komunikację sieciową noda z resztą klastra. Kieruje ruch z obiektów typu service do odpowiednich podów.

Container Runtime – odpowiada bezpośrednio za odpalanie kontenerów w podach. Pobiera również obrazy z repozytoriów. Może to być ContainerD, CRI-O itp.

Kubernetes - namespaces

Namespaces służą do logicznego podziału klastra.
Większość obiektów w Kubernetes przypisana jest do konkretnego namespace, standardowo do namespace ‘default’.

Niektóre obiekty są globalne i nie są przypisane do konkretnego namespace, np. PersistentVolume

Osobny Namespace najczęściej przechowuje aplikacje przypisane do jakiegoś projektu.

Aby zobaczyć dostępne namespace w klastrze wydajemy polecenie:
kubectl get namespaces lub kubectl get ns

Kubernetes - namespaces

Tworzenie nowego namespace:

```
[student@base ~]$ kubectl create ns test-ns
namespace/test-ns created
```

Podstawowa deklaracja namespace:

```
apiVersion: v1
kind: Namespace
metadata:
  name: test-ns
```

Kubernetes - namespaces

W namespace kube-system działają wszystkie obiekty zarządzające pracą całego klastra:

```
[student@base ~]$ kubectl get all -n kube-system
NAME                                   READY   STATUS    RESTARTS   AGE
pod/coredns-74ff55c5b-z2hzk            1/1     Running   2          218d
pod/etcd-minikube                      1/1     Running   2          218d
pod/kube-apiserver-minikube            1/1     Running   2          218d
pod/kube-controller-manager-minikube   1/1     Running   2          218d
pod/kube-proxy-h75sn                   1/1     Running   2          218d
pod/kube-scheduler-minikube            1/1     Running   2          218d
pod/storage-provisioner                1/1     Running   4          218d

NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
service/kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   218d

NAME                        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
daemonset.apps/kube-proxy   1         1         1       1            1           kubernetes.io/os=linux   218d

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/coredns   1/1     1            1           218d

NAME                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/coredns-74ff55c5b    1         1         1       218d
```

Kubernetes - namespaces

Deklaracja poda należącego do konkretnego namespace odbywa się w sekcji metadata:

```
apiVersion: v1
kind: Pod
metadata:
    labels:
      run: test3
    name: test3
    namespace: test-ns
spec:
   containers:
      - image: nginx:1.16
        name: test3
        resources: {}
```

Kubernetes - namespaces

Ustawienie aktywnego namespace dla contextu:

kubectl config get-contexts
kubectl config set-context --current --namespace=dev

Dla danego namespace możemy również zastosować ograniczenie dostępności do zasobów serwera poprzez obiekt ResourceQuota:

```
apiVersion: v1
kind: ResourceQuota
metadata:
    name: qouta-serv
    namespace: dev
spec:
   hard:
      pods: "10"
      requests.cpu: "0.5"
      requests.memory: "1Gi"
      limits.cpu: "1"
      limits.memory: "2Gi"
```

Kubernetes - Pod

Pod to najmniejsza jednostka logiczna w Kubernetes.
Zazwyczaj zawiera jeden kontener, ale jeden Pod może również zawierać kilka kontenerów, np. Init Containers, aplikacja + sidecar.

Kontenery w podach mają ten sam interfejs sieciowy, mogą się komunikować po localhost.

Adres IP przypisany przez Kubernetes jest dla całego poda.
Wewnątrz poda kontenery mogą współdzielić również storage korzystając z tych samych wolumenów.

Kubernetes - Pod

Wdrożenie obiektu **Pod** – podejście imperatywne:

kubectl run app1 --image=httpd

Wyświetlanie aktualnie działających podów:
kubectl get pods

Wyświetlanie aktualnie działających podów z danego namespace,
w tym przypadku namespace o nazwie dev:
kubectl get pods -n dev

Kubernetes - Pod

Pod – podejście deklaratywne:

pod.yml:

```
apiVersion: v1
kind: Pod
metadata:
    name: app2
spec:
    containers:
        - name: app2-cont
          image: httpd
```

kubectl apply -f pod.yml

Kubernetes - Pod

Za pomocą polecenia kubectl describe pod <NAZWA_PODA> możemy prześledzić co działo się z danym Podem od momentu jego uruchomienia:

Informacja o stanie Poda:

[student@base ~]$ kubectl describe pod app1

Name: **app1**
Namespace: default
Priority: 0
Node: minikube/192.168.39.3
Start Time: Tue, 05 Oct 2021 21:12:54 +0200
Labels: run=app1
Annotations: <none>
Status: **Running**
IP: 172.17.0.11
IPs:
IP: 172.17.0.11

Uwaga! Status: Running nie zawsze oznacza, że Pod działa prawidłowo.

Kubernetes - Pod

Za pomocą polecenia kubectl describe pod <NAZWA_PODA> możemy prześledzić co działo się z danym Podem od momentu jego uruchomienia:

Informacja o kontenerach działających w obrębie Poda:

[student@base ~]$ kubectl describe pod app1

Containers:
  app1:
    Container ID: docker://dc45aae1d3263912247d62175e67a7f8b19687f2004a8ec414c...
    Image: httpd
    Image ID: docker-pullable://httpd@sha256:47523aeeb9214d3b20e3dec66de849...
    Port: <none>
    Host Port: <none>
    State: Running
...

Kubernetes - Pod

[student@base ~]$ kubectl describe pod app1

Za pomocą polecenia kubectl describe pod <NAZWA_PODA> możemy prześledzić co działo się z danym Podem od momentu jego uruchomienia:

Kronika życia Poda:

Events:

Type	Reason	Age	From	Message
Normal	Scheduled	13s	default-scheduler	Successfully assigned default/pod5 to poznan.domain1.local
Normal	Pulled	11s	kubelet	Container image "httpd" already present on machine
Normal	Created	11s	kubelet	Created container www
Normal	Started	11s	kubelet	Started container www

...

Kubernetes - Pod

Pod – dostęp do poda przez kubectl proxy:

[student@base ~]$ kubectl proxy
Starting to serve on 127.0.0.1:8001

Po uruchomieniu proxy Kubernetesa możemy w przeglądarce wyświetlić status naszej aplikacji odwołując się bezpośrednio przez API podając odpowiednią wersję API, namespace w którym dany pod był uruchomiony itp., np.:

http://localhost:8001/api/v1/namespaces/default/pods/http:app2:/proxy/

Kubectl proxy nadaje się do debugowania działania aplikacji, nie do stałego dostępu np. w środowisku produkcyjnym.

Kubernetes - Pod

Pod – dostęp do poda przez forwardowanie portu:

Innym sposobem na dostęp do aplikacji wewnątrz Poda
jest przekierowanie portów:

```
[student@katowice ~]$ kubectl get po
NAME   READY   STATUS    RESTARTS   AGE
pod5   1/1     Running   0          11m

[student@katowice ~]$ kubectl port-forward pod5 9090:90
Forwarding from 127.0.0.1:9090 -> 90
Forwarding from [::1]:9090 -> 90
```

Kubectl port-forward nadaje się do debugowania działania aplikacji,
nie do stałego dostępu np. w środowisku produkcyjnym.

Kubernetes - Pod

Pod – generowanie yamla za pomocą podejścia imperatywnego.

kubectl run app-v1 --image=httpd --dry-run=server -o yaml > app-v1.yml

W wyniku powyższego polecenia wygenerowany zostanie plik yaml
z deklaracją aplikacji app-v1 bazującej na obrazie httpd.
Aplikacji app-v1 nie zostanie uruchomiona w Podzie dopóki
nie użyjemy trybu deklaratywnego:

kubectl apply -f app-v1.yml

Usuwanie podów:

- podejście imperatywne:
kubectl delete pod app-v1.yml

- podejście deklaratywne:
kubectl delete -f app-v1.yml

Kubernetes - Pod

Pod – wejście do działającego poda i uruchomienie powłoki bash:.

kubectl exec -it <NAZWA_PODA> -- bash

Gdy w podzie działają 2 lub więcej kontenerów, zawsze po wejściu do poda zostajemy przekierowani do pierwszego zdefiniowanego kontenera.

Aby podłączyć się do drugiego kontenera w podzie musimy użyć flagi -c:

kubectl exec -it <NAZWA_PODA> -c <NAZWA_KONTENERA> -- bash

Kubernetes - Pod

Pod – przydzielanie zasobów serwera.

W definicji Poda możemy podać żądania oraz limity górne zasobów.
Określamy to w sekcji spec: na poziomie konkretnego kontenera:

```
apiVersion: v1
kind: Pod
metadata:
    name: pod5
spec:
    containers:
      - name: pod1-cont
        image: httpd
        resources:
          requests:
            cpu: 100m
            memory: 200Mi
          limits:
            cpu: 200m
            memory: 500Mi
```

Kubernetes - Pod

Pod – przydzielanie zasobów serwera.

W sekcji resources:

- requests określa zasoby jakie pod żada na starcie
  Gdy w klastrze nie ma odpowiedniej ilości wolnych zasobów, Pod będzie działał w stanie Pending i będzie oczekiwał, aż żadane zasoby zostaną zwolnione.

- limits określa górną granicę zasobów jakie mogą być przydzielone dla danego kontenera. Jeśli w Podzie znajdują się 2 kontenery i dla każdego kontenera ustawione są górne granice dostępności do zasobów, to są one sumowane.

Kubernetes - Pod

Pod – przydzielanie zasobów serwera – przykład:

```
apiVersion: v1
kind: Pod
metadata:
...
spec:
    containers:
        - name: www
          image: httpd
          resources:
             requests:
                cpu: 1
                memory: 1Gi
             limits:
                cpu: 2
                memory: 4Gi.
```

Kubernetes - Pod

Pod – przekazywanie argumentów.

Działający w podzie kontener może przyjąć argumenty sprecyzowane w sekcji containers nadpisujące ENTRYPOINT i CMD z oryginalnego obrazu:

- ENTRYPOINT → COMMAND
- CMD → ARGS

```
containers:
    - name: app1
      image: app:1.0
      command: [ „docker-entrypoint.sh” ]
      args: [ „sleep”, „20” ]
```

Kubernetes - Pod

Pod – przekazywanie argumentów.

Zmienne systemowe możemy przekazać za pomocą env w sekcji containers. Przyjmuje postać klucz → wartość:

```
containers:
   - name: app1
     image: httpd
     env:
        - name: URL
          value: „http://mikroserwis.pl”
```

Kubernetes - Pod

Pod – przekazywanie argumentów.

Zmienne systemowe możemy przekazać za pomocą obiektu ConfigMap. Po zdefiniowaniu ConfigMap odwołujemy się do niego w następujący sposób:

```
containers:
    - name: app1
      image: httpd
      env:
          - name: URL
            value_from:
                configMapKeyRef:
```

Kubernetes - Pod

Pod – przekazywanie argumentów.

Zmienne systemowe, które nie powinny być przekazywane w sposób jawny, możemy umieścić w obiecie Secret.
Zmienna taka będzie kodowana algorytmem base64.
Do Sekretów odwołujemy się w następujący sposób:

```
containers:
   - name: app1
     image: httpd
     env:
        - name: URL
          value_from:
             secretKeyRef:
```

Kubernetes - Pod

Pod – opcje schedulera.

Podczas rozmieszczania podów na klastrze Kubernetes wybiera nody, który mają najwięcej do zaoferowania (BestEffort)
Możemy jednak wykorzystać pewne argumenty,
które pozwolą na pewną kontrolę w ich rozmieszczeniu.

Niektóre z nich to:

- **nodeName**: Pod zostanie umieszczony na nodzie z odpowiednią nazwą

- **nodeSelector**: Pod zostanie umieszczony na nodach z odpowiednią etykietą

Kubernetes - Pod

Pod – opcje scheduler.

Podczas rozmieszczania Podów na klastrze Kubernetes
możemy wykorzystać argumenty, które pozwolą na pewną kontrolę
w ich rozmieszczeniu.

Niektóre z nich to:

- **schedulerName**: możemy wybrać innego schedulera niż standardowy
- **affinity lub anti-affinity**: można użyć do preferowanych lub żądanych nodów
- **taints i tolerations**: z jakiegoś powodu Pod nie powinien być umieszczany na nodach oznaczonych za pomocą taints

Kubernetes – Replica Set

ReplicaSet – jest jedną z głównych zalet Kubernetes. Określa ilość replik danego obiektu jaka ma działać w obrębie klastra.
ReplicaSet może zwiększać lub zmniejszać ilość replik danego poda nawet jeśli ten pod był wcześniej zdefiniowany i jest uruchomiony.
Wówczas ReplicaSet rozpoznaje działające pody za pomocą etykiet w sekcji spec:

```
selector:
    matchLabels:
        app: web
```

Nazwa etykiety musi być zgodna z etykietą określającą specyfikę poda w sekcji template:

```
template:
    metadata:
        labels:
            app: web
```

Dzięki temu ReplicaSet wie, które pody ma replikować

Kubernetes – Replica Set

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
    name: rs-web1
    labels:
        app: web1
spec:
    replicas: 3
    selector:
        matchLabels:
            app: web1
    template:
        metadata:
            labels:
                app: web1
        spec:
            containers:
                - name: web1-cont
                  image: nginx
                  ports:
                      - containerPort: 80
```

Kubernetes – Replica Set

Ilość replik danego poda określamy w sekcji **spec:** za pomocą zmiennej **replicas:**

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
    name: rs-web1
    labels:
        app: web1
spec:
    replicas: 3
```

Kubernetes – Replica Set

Gdy określimy ilość replik danego Poda, w momencie awarii jednego z nich lub zabicia procesu na którym działa Pod, **ReplicaSet** automatycznie powołuje do życia nowego Poda, tak by ilość Podów zgadzała się z naszą deklaracją.

Nowe Pody uruchamiane przez ReplicaSet startują standardowo od razu od momentu zwiększenia ilości Podów lub w razie podmniany Poda.
Można to zmienić wprowadzając pewne opóźnienie za pomocą opcji **minReadySeconds** w sekcji spec w obiekcie **ReplicaSet**.

Aby zamknąć wszystkie Pody należące do **ReplicaSet** musimy najpierw usunąć replikę.

Kubernetes – Deployment

Deployment – to jeden z najważniejszych obiektów Kubernetesa. Wdraża całościowo aplikację wraz z podami, replikacją, strategią RollingUpdate itp.

Najważniejsze mechanizmy działające w **Deployment** to:

- skalowanie aplikacji w górę, w dół

- bieżące aktualizacje aplikacji

- powrót do poprzedniej wersji aplikacji

Obiekt **Deployment** ma złożoną budowę:
zawiera w sobie **ReplicaSet**, który z kolei zawiera **pody**.

Kubernetes – Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: webapp-deployment
spec:
    selector:
       matchLabels:
            name: moj-web
    replicas: 3
    strategy:
       type: RollingUpdate
    template:
      metadata:
          labels:
             name: moj-web
      spec:
         containers:
            - name: webapp-cont
              image: httpd
```

Kubernetes – Deployment

Deployment – wdrożenie

```
[student@base ~]$ kubectl apply -f deployment.yml

[student@base ~]$ kubectl get deploy
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
webapp-deployment   2/2     2            2           32s

[student@base ~]$ kubectl get rs
NAME                       DESIRED   CURRENT   READY   AGE
webapp-deployment-b7f5c7bf7   2         2         2       36s

[student@base ~]$ kubectl get po
NAME                          READY   STATUS    RESTARTS   AGE

webapp-deployment-b7f5c7bf7-2b52q   1/1     Running   0          42s
webapp-deployment-b7f5c7bf7-vm998   1/1     Running   0          42s
```

Kubernetes – Deployment

Deployment – strategia aktualizacji, to jedna z wyróżniających cech deploymentu. Pozwala aktualizować aplikację uruchomioną w podach za pomocą **strategy**:

- **RollingUpdate** – jest to domyślna strategia. Pozwala stopniowo wprowadzać nowe Pody ze zaktualizowaną aplikacją, jednocześnie kasować Pody z poprzednią wersją aplikacji. W trakcie updatowania aplikacji w klastrze działają równocześnie Pody z dotychczasową wersją aplikacji, jak i pody powstałe z nowszą wersją.

- **Recreate** – przy aktualizacji do nowej wersji aplikacji wszystkie Pody z dotychczasową wersją aplikacji zostają skasowane i gdy już nie pozostanie żaden Pod – tworzone są od nowa nowe Pody z nowszą wersją aplikacji

Kubernetes – Deployment

Deployment – strategia aktualizacji typu **RollingUpdate**
pozwala określić ile Podów maksymalnie zostanie skasowanych jednocześnie podczas aktualizacji za pomocą zmiennej: **maxUnavailable** oraz ile Podów maksymalnie może powstać jednocześnie (zmienna **maxSurge**). Np:

```
strategy:
        rollingUpdate:
                maxSurge: 25%
                maxUnavailable: 25%
        type: RollingUpdate
```

W tym przypadku 25% aktualnych Podów może zostać skasowanych i tak samo 25% nowych Podów może zostać uruchomionych.

maxSurge i **maxUnavailable** przyjmuje wartości procentowe lub liczbowe.

Kubernetes – Deployment

Deployment – strategia aktualizacji typu **Recreate**
polega na całkowitym skasowaniu dotychczas działających Podów z daną aplikacją oraz uruchomienie nowej ilości Podów zgodnie z podaną ilością replik w zmiennej replicas:

```
strategy:
        rollingUpdate:
        type: Recreate
```

Strategia **Recreate** powoduje chwilową niedostępność aplikacji bądź usługi.

Kubernetes – Deployment

Deployment – podczas aktualizacji deploymentu powstaje zupełnie nowy obiekt ReplicaSet. Stara wersja ReplicaSet nie zostaje całkowicie usunięta, tylko tymczasowo zatrzymana.
Dzięki temu jesteśmy w stanie zrobić tzw. rollback do poprzedniej wersji aplikacji.

Ostatnia konfiguracja deploymentu przechowywana jest w adnotacjach. Możemy to zobaczyć przy zrzuceniu deploymentu np. do formatu yaml:

kubectl get deployment <NAZWA> -o yaml

```
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},
       "name":"webapp-deployment","namespace":"default"},
        "spec":{"replicas":2,"selector":{"matchLabels":{"name":"webapp"}},"strategy":{"typ
```

Kubernetes – Deployment

Deployment – podczas aktualizacji deploymentu powstaje historia naszych poprawek, którą możemy zobaczyć za pomocą polecenia **kubectl rollout history**:

```
[student@base ~]kubectl rollout history deployment webapp-deploymentml
deployment.apps/webapp-deployment
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

Jak widać deployment został jeden raz zaktualizowany (stąd 2 wersje: starsza i nowsza).
W razie niepoprawnego działania nowej wersji szybko możemy powrócić do poprzedniej – komenda **kubectl rollout undo**:

```
[student@base ~]$ kubectl rollout undo deployment webapp-deployment
deployment.apps/webapp-deployment rolled back
```

Kubernetes – Deployment

Deployment – podczas aktualizacji deploymentu możemy „nagrywać” jakie polecenie jest odapowiedzialne za dokonanie zmian w Deploymencie. Robimy to za pomocą opcji --**record=true**
Widoczne to będzie w statusie **CHANGE-CAUSE**

kubectl apply -f deployment.yml --record=true

[student@base ~]kubectl rollout history deployment webapp-deploymentml
deployment.apps/webapp-deployment
REVISION **CHANGE-CAUSE**
1 <none>
2 **kubectl apply -f deployment --record=true**

Kubernetes – Deployment

Deployment – historia aktualizacji:

Status aktualizacji możemy sprawdzić za pomocą
polecenia **kubectl rollout status**:

kubectl rollout status deployment/web-deploy-nginx
deployment "web-deploy-nginx" successfully rolled out

Co się działo podczas konkretnej aktualizacji (tu oznaczonej nr 2)
możemy zbadać za pomocą opcji **--revision=**:

kubectl rollout history deployment/web-deploy-nginx --revision=2

Kubernetes – Deployment

Deployment – przykładowy wynik sprawdzenia jaki efekt miała konkretna aktualizacja:

```
deployment.apps/web-deploy-nginx with revision #2
Pod Template:
  Labels:    name=webapp
pod-template-hash=6777b84d68
  Containers:
     webapp-cont:
     Image:        nginx:1.19
     Port:    80
     Host Port:    <none>
     Environment:       <none>
     Mounts:       <none>
  Volumes:<none>
```

Kubernetes – Deployment

Deployment – tylko niektóre obiekty można poddać aktualizacji za pomocą RollingUpdate. Historia aktualizacji będzie dostępna dla:

* deployments
* daemonsets
* statefulsets

Kubernetes – Deployment

W specyfikacji kontenerów wchodzących w skład
np. Deployment, Pod możemy posłużyć się dwoma zmiennymi:

readinessProbe – określa czy aplikacja jest już gotowa po starcie kontenera. Może potrzebować nieco czasu do pełnej funkcjonalności,
mimo że proces kontenera jest widoczny dla Kubernetesa

livenessProbe – określa czy aplikacja działająca w Deploymencie nadal jest dostępna. Może się bowiem okazać, że proces kontenera widoczny jest w systemie jako prawidłowy (również z punktu widzenia Kubernetes), ale sama aplikacja przestała działać prawidłowo

W przypadku gdy **readinessProbe** lub **livenessProbe** wykaże, że aplikacja nie odpowiada w sposób prawidłowy Kubernetes oddziałuje na Poda zgodnie z przewidzianą polityką restartu (restartPolicy).

Kubernetes – Deployment

Dostępne polityki restartu Poda w ramach Deploymentu (i nie tylko):

restartPolicy:

- **Always** : domyślna polityka restartu Podów.
  Jeśli nie zdecydujemy inaczej, działające Pody będą restartowan
  nawet po poprawnie wykonanej pracy.

- **OnFailure** : Pody zostaną zrestartowane tylko i wyłącznie wtedy,
  gdy działanie Poda zakończy się błędem.

- **Never** : Pody nie zostaną nigdy zrestartowane, niezależnie od tego
  czy zakończą pracę poprawnie czy z błędem.

RestartPolicy deklaruje się w specyfikacji Podów

```
metadata:
...
spec:
    restartPolicy: OnFailure
```

Kubernetes – Job

Obiekt **Job** tworzy jeden lub więcej podów.
Zazwyczaj **Job** wykonuje jedno zadanie (może się wykonywać kilka razy).

Znajduje zastosowanie w:
- analizie danych
- skomplikowanych, długotrwałych zadaniach
- przetwarzaniu danych

Jeśli Job składa się z kilku Podów to zakończenie działania obiektu Job następuje, gdy wykonają się poprawnie wszystkie Pody wchodzące w skład Joba.

Obiekt Job nie może mieć ustawionej polityki restartPolicy na Always. Domyślną politykę musimy zmienić na Never lub OnFailure.

Kubernetes – Job

Przydatne opcje w ramach konfiguracji obiektu Job:

- Za pomocą opcji backoffLimit możemy określić ile razy maksymalnie kontener w ramach Joba może się restartować z powodu błędu. Jeśli Podów jest więcej w Jobie niż 1 to jest to limit dla jednego Poda.

- **parallelism** – określa maskymalną liczbę podów działających w obiekcie Job jednocześnie

- **completions** – określa ile podów musi zakończyć się pomyślnie w ramach Joba, żeby obiekt Job mógł zostać uznany za zakończony prawidłowo

- **activeDeadlineSeconds** – określa ile maksymalnie czasu może się wykonywać dany Job. Czas podajemy w sekundach. Jeśli wszystkie Pody w ramach Joba nie zakończą się w tym czasie, Job zostaje uznany za niepoprawny.

Kubernetes – Job

Przykładowa deklaracja obiektu Job:

```
apiVersion: batch/v1
kind: Job
metadata:
    name: pi
spec:
    template:
      spec:
        containers:
          - name: pi
            image: perl
            command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
        restartPolicy: Never
    backoffLimit: 4.
```

Kubernetes – CronJobs

Obiekt **CronJobs** jest podobny do serwisu crond działającego w systemach Linux.

Wykonuje cyklicznie określone zadania zgodnie ze specyficznym dla Crona formatem zapisu czasu (podpowiedź w /etc/crontab).

Wykorzystywany do różnych zadań, np.:
- czyszczenie tymczasowych danych
- wysyłka powiadomień (maili)
- tworzenie backupów
- generowanie raportów itp..

Kubernetes – CronJobs

Obiekt **CronJobs** – przydatne opcje w sekcji spec:

- **concurrencyPolicy** – określa czy jeśli pierwsze zadanie się nie zakończyło to może startować zadanie drugie. Domyślnie opcja ta jest dozwolona, ustawiona na Allow

- **startingDeadlineSecond** – określa jak długo od czasu startu może się wykonywać dane zadanie (określone w sekundach). Jeśli zadanie przekroczy nałożony limit czasowy, wystartuje ponownie.

Kubernetes – CronJobs

Przykładowa deklaracja biektu **CronJobs**:

```
apiVersion: batch/v1
kind: CronJob
metadata:
   name: hello
spec:
   schedule: "*/1 * * * *"
   jobTemplate:
      spec:
        template:
          spec:
             containers:
                - name: hello
                  image: busybox
                  command: [‘sh’, ‘-c’, ‘echo Hello world’]
        restartPolicy: OnFailure
```

Kubernetes – DaemonSet

Obiekt **DaemonSet** odpowiada za równomierny rozkład Podów na każdym nodzie w klastrze.

Deployment rozmieszcza Pody na nodach na zasadzie BestEffort. Ilość wolnego procesora oraz pamięci RAM decyduje ile Podów zostanie przypisane do danego noda.

DaemonSet przypisuje jednego Poda do każdego noda bez względu na ilość wolnych zasobów (jest to zachowanie domyślne – można to zmienić).

Gdy w klastrze pojawi się nowy node, **DaemonSet** od razu przydzieli mu swojego Poda.

Kubernetes – DaemonSet

Najczęstsze zastosowania obiektu **DaemonSet:**

- agregacja logów

- monitoring stanu nodów

- monitoring aplikacji działających na poszczególnych nodach

- wykrywanie nieporządanych wydarzeń na nodach

Kubernetes – DaemonSet

Przykładowa definicja **DaemonSet**:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
     name: daemon1
     labels:
        app: demon
spec:
    selector:
        matchLabels:
            app: demon
    template:
       metadata:
           labels:
             app: demon
    spec:
        containers:
          - name: daemon1-cont
            image: httpd
```

Kubernetes – DaemonSet

Obiekt **DaemonSet** posiada strategię updateStrategy:

- **RollingUpdate** – opcja domyślna, podobnie jak w Deployment.

- **OnDelete** – zmiana np. obrazu kontenera następuje jeśli
  Pod wchodzący w skład DaemonSet zostanie skasowany.
  Może to prowadzić do sytuacji, że w obrębie DaemonSet
  będziemy mieć na jednym nodzie starszą wersję aplikacji,
  a na drugim nodzie nowszą wersję.

Kubernetes – DaemonSet

Obiekt **DaemonSet** z dwoma obrazami w kontenerach:

```
[student@katowice ~]$ kubectl get ds
NAME      DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR
daemon1   2         2         2       1            2           <none>          11m

[student@katowice ~]$ kubectl get po
NAME              READY   STATUS    RESTARTS   AGE
daemon1-k8d8c     1/1     Running   0          2m47s
daemon1-ltrgz     1/1     Running   0          6m54s


[student@katowice ~]$ kubectl describe pod daemon1-k8d8c | grep image
  Normal  Pulled     3m6s  kubelet            Container image "httpd" already present on machine

[student@katowice ~]$ kubectl describe pod daemon1-ltrgz | grep image
  Normal  Pulled     7m26s  kubelet            Container image "nginx" already present on machine
```

Kubernetes – Service

Obiekt **Service** pozwala na dostęp do aplikacji działającej w Podach. Po skonfigurowaniu np. obiektu Deployment udostępnianie usługi robimy za pomocą polecenia **kubectl expose** np:

kubectl expose deployment aplikacja:1.0

Każdy Pod działający w klastrze dostaje IP wewnętrzne w klastrze i nie ma do niego bezpośredniego dostępu z zewnątrz klastra.

Dodatkowo Pody mogą ulec zniszczeniu i na ich miejsce powstaje nowy Pod z innym adresem IP, więc odwoływanie się do konkretnego adresu IP Poda nie jest dobrym pomysłem.

Obiekt **Service** automatycznie przekierowuje połączenie do poszczególnych Podów mając jeden stały adres IP.

Kubernetes – Service

Obiekt **Service** umieszczany jest w ramach konkretnego namespace. W momencie pojawienia się nowego Poda Service zauważa go automatycznie i może również do niego od razu kierować połączenie.

W momencie powoływania obiektu Service następuje także wpis do DNS klastra, dzięki temu możemy komunikować się pomiędzy mikroserwisami w klastrze.

Wpisy w DNS są tworzone wg schematu:
<nazwa_service>.<nazwa_namespace>.svc.cluster.local

Rodzaj obiektu Service:
- ClusterIP
- NodePort
- LoadBalancer
- ExternalName

Kubernetes – Service

Obiekt **Service** typu **CluserIP** jest podstawowym i domyślnym servisem dla aplikacji w klastrze. Udostępnia aplikację wewnątrz klastra.

Service kieruje połączenie do poszczególnych Podów aplikacji za pomocą sekcji selector:
Na podstawie etykiet obiekt **Service** wie, które Pody wchodzą w skład aplikacji (np. Deploymentu).

ClusterIP:
- domyślny typ Service
- otrzymuje własny adres IP
- dostępny tylko wewnątrz klastra

Kubernetes – Service

Definicja ClusterIP:

```
apiVersion: "v1"
kind: Service
metadata:
    name: cluster-IP1
spec:
    ports:
      - port: 80
        targetPort: 8080
        protocol: TCP
    type: ClusterIP
    selector:
      app: www1    #pody z etykietą www1
```

Kubernetes – Service

NodePort:

- pozwala na udostępnienie aplikacji na zewnątrz

- udostępnia port, który dostępny jest na adresie IP poszczególnych nodów

- przypisany port jest dostępny na każdym nodzie

- przypomina ‘docker run -p’

Wraz z **NodePort** automatycznie tworzy się obiekt typu **CluserIP**

Kubernetes – Service

Definicja **NodePort**:

```
apiVersion: "v1"
kind: Service
metadata:
    name: nodeport1
spec:
    ports:
      - port 80
        targetPort: 8080
        nodePort: 30080
    type: NodePort
    selector:
      app: www1
```

Kubernetes – Service

LoadBalancer:

- przewidziany dla rozwiązań chmurowych

- integruje się z chmurowym Load Balancerem

- pod spodem Service LoadBalancer wykorzystuje typ NodePort
  i kieruje ruch do poszczególnych nodów

- zazwyczaj otrzymujemy publiczny adres IP z chmury

Kubernetes – Service

ExternalName:

- najmniej popularny

- mapuje Service na wpis DNS

- akceptuje tylko format nazwy DNS zgodny z IPv4

- nie wspiera bezpośrednio adresów IP

- nie wykorzystuje etykiet i selectorów

- najczęściej wykorzystywany w migracji systemu, np. bazy danych

Kubernetes – Service

Wdrożenie obiektów Service:

```
[student@katowice ~]$ kubectl expose pod pod1
service/pod1 exposed

[student@katowice ~]$ kubectl expose pod pod2 --type=NodePort
service/pod2 exposed

[student@katowice ~]$ kubectl expose pod pod3 --type=LoadBalancer
service/pod3 exposed

[student@katowice ~]$ kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP      10.96.0.1       <none>        443/TCP        4d23h
pod1         ClusterIP      10.108.24.131   <none>        80/TCP         28s
pod2         NodePort       10.97.253.188   <none>        80:30154/TCP   18s
pod3         LoadBalancer   10.104.99.191   <pending>     80:32397/TCP   7s
```

Kubernetes – StatefulSet

W odróżnieniu od obiektów typu DaemonSet lub Deployment
StatefulSet jest obiektem dla aplikacji stanowych.

W obiekcie StatefulSet każdy zdefiniowany Pod wykonuje się
w odpowiedniej kolejności. Od pierwszego do ostatniego.
Po prawidłowym uruchomieniu pierwszego Poda dopiero startuje
Pod nr 2.
Pody numerowane są od 0.

Każdy Pod w obiekcie StatefulSet ma stałą nazwę. Podczas odtwarzania
Poda po awarii, tworzony jest jego zastępnik o dokładnie
takiej samej nazwie jak Pod zepsuty.

Kubernetes – StatefulSet

Do działania **StatefulSet** niezbędny jest **Headless Service**.

Headless Service związany jest tylko i wyłącznie z obiektem **StatefulSet**.
Nie ma stałego adresu IP, Load Balancing też nie jest obsługiwany, połączenia odbywają się tylko po adresie DNS poszczególnych Podów.

Headless Service można poznać po określeniu **ClusterIP: None** w definicji serwisu.

Do aplikacji **StatefulSet** odwołujemy się po konkretnej nazwie Poda:

<nazwa_poda>.<nazwa_service>.<nazwa_namespace>.svc.cluster.local
np.:
redis-0.redis.default.svc.cluster.local

Kubernetes – StatefulSet

Deklaracja **Headless Service**:

```
apiVersion: "v1"
kind: Service
metadata:
    name: redis-service
spec:
    clusterIP: None
    ports:
      - port: 6379
        targetPort: 6379
        name: client
    selector:
       app: redis
```

Kubernetes – StatefulSet

StatefulSet – przykładowa deklaracja:

```
apiVersion: "apps/v1"
kind: StatefulSet
metadata:
    name: redis
    labels:
      app: redis
spec:
   serviceName: redis-service #obiekt Headless Service
   replicas: 5
   template:
     metadata:
        labels:
          app: redis
     spec:
        containers:
          - name: redis-cont
            image: redis:5.0.1-alpine
    selector:
      matchLabels:
        app: redis
```

Kubernetes – ConfigMap

Kubernetes to środowisko rozproszone, dlatego poszczególne obiekty muszą być dostępne w obrębie całego klastra. Również dotyczy to plików konfiguracyjnych.

Obiekt **ConfigMap** jest wykorzystywany do centralnego przechowywania konfiguracji oraz podmontowywania w działających obiektach.

Dzięki **ConfigMap** możemy przechowywać konfigurację tak, by mogła być dostępna na wszystkich nodach w klastrze Kubernetesa.

Podmontowywanie jakichkolwiek plików z poszczególnego noda jest niewskazane.

Jeden obiekt **ConfigMap** może być wykorzystywany wielokrotnie.

Kubernetes – ConfigMap

Przykład obiektu **ConfigMap**:

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: dane1
data:
    app: httpd
    version: 2.4
```

Taką **ConfigMap** można podmontować do poda.

ConfigMap może również przechowywać cały plik konfiguracyjny danej aplikacji czy serwisu:

```
        data: |
            zmienna1=wartosc1
            zmienna2=wartosc2
```

Kubernetes – Secrets

Obiekt **Secret** służy do przekazywania konfiguracji,
która nie powinna być widoczna jako czysty tekst.

Zazwyczaj w **Secrets** znajdują się:
- tokeny logowania
- loginy
- hasła
- klucze itp.

Dane przechowywane w **Secrets** kodowane są base64.
Nie gwarantuje to wysokiego poziomu zabezpieczeń.

Kubernetes – Secrets

Budowa obiektu **Secrets:**

```
apiVersion: v1
kind: Secret
metadata:
     name: tajne
     labels:
        app: mysql
type: Opaque
data:
    MYSQL_ROOT_PASSWORD:  laskLJdsi45==
    MYSQL_DATABASE:  lshDJFslkdfj3ii!!!!!
```

Kubernetes – Volumeny

W Kubernetesie domyślnie wszystkie uruchamiane Pody
są efemeryczne, tzn. nie zapisują trwale danych wygenerowanych
przez aplikacje, które wchodzą w skład Podów.

Wraz z zakończeniem działania Poda nie mamy dostępu do danych.

Kubernetes pozwala na zdefiniowanie w obrębie Poda volumenu,
który zostanie podpięty pod konkretny kontener.

Jeden volumen może być podmontowany pod kilka kontenerów
działających wewnątrz Poda.
Może również być współdzielony przez kilka Podów w klastrze.

Poszczególny Pod może podmontować kilka volumenów.

Kubernetes – Volumeny

W obrębie klastra Kubernetesa możemy podłączyć szereg dostępnych techonologii storage takich jak:
- Ceph
- NFS
- GlusterFS
- różnego rodzaju rozwiązania stosowane w środowiskach chmurowych

CSI – Container Storage Interface stanowi interfejs ułatwiający podłączenie się do różnorodnych rozwiązań dostępu do pamięci masowych. Aktualnie jest w fazie beta.

Dzięki CSI sterowniki do obsługi dysków zostaną oddzielone od samego Kubernetesa. Aktualnie sterowniki te są wbudowane w Kubernetesa.

Kubernetes – Volumeny

Kubernetes oferuje Persistent Volume jako obiekt, który może być skojarzony z zewnętrznym źródłem pamięci masowej.

Pody mogą korzystać z Persistent Volume za pomocą obiektu pośredniczącego Persistent Volume Claim.

Dane zgromadzone w volumenie zostają zachowane i mogą być wykorzystane przez następne Pody.

ConfigMap oraz Secret również można podmontować w obrębie działającego Poda. Zawierają one najczęściej dane konfiguracyjne niekodowane (ConfigMap) oraz kodowane base64 (Secret).

Kubernetes – Volumeny

EmptyDir jest przykładem wykorzytania volumenów do zapewnienia wymiany informacji wewnątrz Poda przez kilka kontenerów.

Wewnątrz kontenera tworzony jest katalog, który nie montuje żadnego zewnętrznego źródła. Wraz z zakończeniem pracy Poda volumen również jest usuwany.

```
spec:
    containers:
    - image: httpd
      name: web1
      volumeMounts:
        - mountPath: /var/www/html
          name: web1-volume
    volumes:
        - name: web1-volume
          emptyDir: {}
```

Kubernetes – Volumeny

HostPath montuje zasób, który jest dostępny w obrębie konkretnego noda w klastrze. Może to być zasób typu dysk, katalog, socket itp.
Dany zasób musi istnieć na nodzie, chyba że zastosujemy opcje: **DirectoryOrCreate** lub **FileOrCreate** (stworzy odpowiednio katalog lub plik).

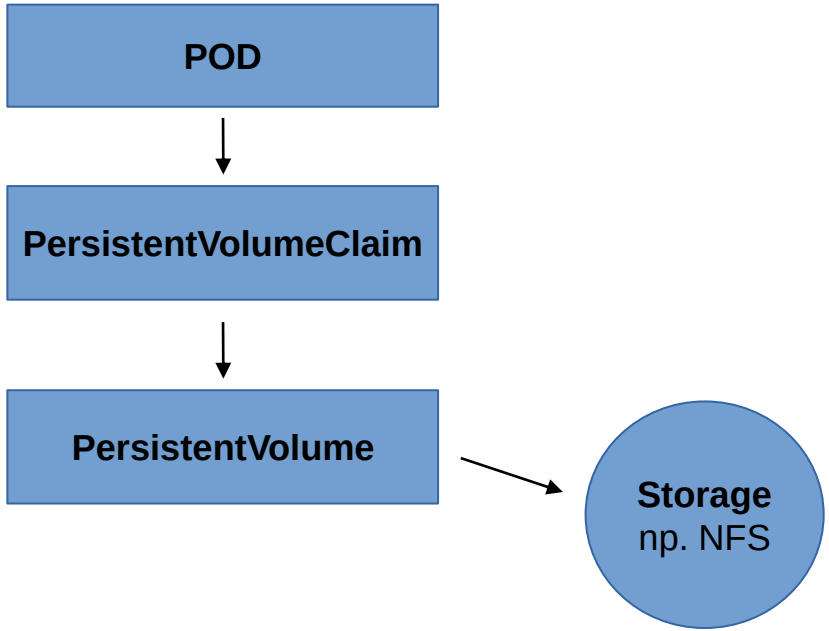
```
spec:
    containers:
    - image: httpd
      name: web1
      volumeMounts:
         - mountPath: /var/www/html
           name: vol1
    volumes:
        - name: vol1
          hostPath:
      path: /data
```

Kubernetes – Volumeny

```
claimName: claim1
```

```
metadata:
    name: claim1
spec:
    accesMode:
    volumeName: pv1
```

```
metadata:
    name: pv1
spec:
    accessMode:
```



Kubernetes – Volumeny

Persistent Volume to warstwa abstrakcji używana do dłuższego zatrzymania danych nawet po zabiciu Poda. Persistent Volume nie działają w obrębie konkretnego namespace.

```
kind: PersistentVolume
apiVersion: v1
metadata:
    name: 10Gpv01
    labels:
        type: local
spec:
    capacity:
        storage: 10Gi
    accessModes:
        - ReadWriteOnce
    nfs:
 server: base
        path: "/exampleshare"
```

Kubernetes – Volumeny

Persistent Volume będzie dostępne dla Poda po zdefiniowaniu manifestu **Persistent Volume Claim.**
Persistent Volume Claim działają w obrębie konkretnego namespace.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
   name: myclaim
spec:
   accessModes:
      - ReadWriteOnce
   resources:
      requests:
            storage: 8Gi
```

Kubernetes – Volumeny

Wywołane **Persistent Volume Claim** w Podzie**:**

```
kind: Pod
metadata:
    name: pod1
spec:
   containers:
      - name: kont1
        image: nginx
   volumes:
      - name: test-volume
        persistentVolumeClaim:
            claimName: myclaim
```

Pole **claimName** musi być zgodne z nazwą PersistentVolumeClaim

Kubernetes – Volumeny

W Kubernetesie od wersji 1.4 dostępny jest także Dynamic Provisioning mogący żądać zewnętrznego wcześniej skonfigurowanego zasobu. Odbywa się to za pomocą obiektu **Storage Class** (często używanym w środowiskach chmurowych), np. w Google Cloud:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: fast
provisioner: kubernetes.io/gce-pd
parameters:
    type: pd-ssd
```

Aby skorzystać z przestrzeni dyskowej zdefiniowanej w **StorageClass** możemy się posłużyć **PersistentVolumeClaim**.

Kubernetes – Volumeny

Żądania **PersistentVolumeClaim** jesteśmy w stanie ograniczyć za pomocą **Resource Quota**.
W poniższym przykładzie ograniczamy dostęp do zasobów dla 10 PersistentVolumeClaim
i dostępność zasobów na poziomie 500MB:

```
apiVersion: v1
kind: ResourceQuota
metadata:
    name: storagequota
spec:
   hard:
      persistentvolumeclaims: "10"
      requests.storage: "500Mi"
```

Zapraszamy do współpracy

ALTKOM AKADEMIA
ul. Chłodna 51,
00-867 Warszawa
Telefon: (+48 22) 460 99 99,
warszawa@altkom.pl