

Core Concepts

Cluster architecture

Celem Kuberntes jest hostowanie aplikacji w formie zautomatyzowanych kontenerów, tak aby tworzyć pożądaną ilość instancji i łatwo się między nimi komunikować.

Kubernetes cluster składa się ze zbioru nodów (mogą być wirtualne lub fizyczne, w cloudzie lub on premise), które hostują aplikację w formie kontenerów. Są to tzw Worker Nodes, które łączą kontenery. Mamy też Master Node, który zarządza całym clustrem, monitoruje nody, zarządza tworzeniem i uzupełnianiem kontenerów. Master wykonuje swoją pracę przy pomocy tzw Control Plane Components.

Komponenty:

- ETCD Cluster - jest to bardzo ważny komponent, który zawiera informację o tym ile kontenerów zostało stworzony, o której godzinie itp. Przechowuje on informacje w formacie key-value.
- kube-scheduler - identyfikuje odpowiedni node, na którym trzeba umieścić kontener na podstawie np. tego ile kontenerów już istnieje na nodach, worker nodes capacity, czy innych ustawionych policy.
- Controllers (Node-Controller, Replication-Controller, Controller-Manager) - Node jest odpowiedzialny za nody, on tworzy nowe, zajmuje się sytuacjami, w których node jest niedostępny, albo zniszczony. Replication sprawdza, aby odpowiednia ilość kontenerów chodziła przez cały czas, czyli jak jeden się zepsuje to on zleca stworzenie nowego.
- kube-apiserver - główny komponent zarządzający Kuberntesem, jest odpowiedzialny za nadzorowanie wszystkich operacji w clustrze. Udostępnia Kubernetes API do zewnętrznych użytkowników, aby mogli zarządzać clustrem i go monitorować.

Zarządzające komponenty na Masterze też mogą być w formie kontenerów. Aby kontenery w ogóle działały potrzebujemy tzw container engine, wśród których najpopularniejszy jest Docker. Musi on być zatem zainstalowany na wszystkich nodach łącznie z Masterem.

Na każdym z Worker Nodów mamy następujące komponenty:

- kubelet, który zajmuje się komunikowaniem z Masterem, daje znać, że kontener jest gotowy do pracy, zwraca status noda i kontenerów przez raportowanie itp. Jest to agent, który pracuje na każdym node w clustrze, słucha poleceń od kube-apiserver i tworzy lub usuwa kontenery.
- Kube-proxy server odpowiada za umożliwianie lub blokowanie komunikacji pomiędzy kontenerami na tym samym lub na innym node. Ustawia on odpowiednie role na kontenerach.

ETCD

Jest to komponent przechowujący w bezpieczny sposób informacje o kontenerach. Informacje są przechowywane w postaci key-value.

Tradycyjne bazy danych (np. SQL) przechowują dane w postaci tabel. W ETCD mamy dwie kolumny: key i value. Dla każdego klucza mamy daną wartość. Jeśli chcemy dodać do bazy nową wartość to robimy to w ten sposób:

Put Key "Value"

Jeśli chcemy wyświetlić daną wartość to robimy:

Get Key

Klucze nie mogą być zduplikowane. Taki rodzaj przechowywania danych nie zastąpi nam tradycyjnej bazy, ale będzie dobry do przechowywania małych ilości danych, takich jak konfiguracje, pody, node, role itp, które muszą być często odczytywane.

Aby zainstalować ETCD trzeba pobrać jego binarkę, rozpakować ją i uruchomić serwis: `./etcd`. Po uruchomieniu serwis będzie nasłuchiwał na porcie 2379. Można teraz dodawać klientów do ETCD. Domyślny to ETCD control client, jest to command line klient. Jest on używany do przechowywania i

wyciągania danych w postaci key-value. Aby stworzyć nową wartość w bazie używamy komendy:

`./etcdctl set klucz wartość`

`./etcdctl get klucz` - wyświetli wartość danego klucza.

`./etcdctl` - wyświetli wszystkie dostępne komendy

ETCD w Kubernetesie

Zawsze gdy uruchamiamy komendę `kubectl get`, wyświetlane informacje pochodzą z ETCD server. Gdy chcemy coś zmienić na clustrze to dopiero wejdzie to w życie, gdy zmiana zostanie zrobiona w ETCD. W zależności od tego, jak tworzymy cluster, ETCD będzie stworzone inaczej. Jeśli tworzymy cluster ręcznie to trzeba samemu pobrać binarki ETCD, potem trzeba samemu je zainstalować i skonfigurować na Masterze. Bardzo ważne jest ustawienie parametru `advertise-client-urls`. Jest to adres, na którym ETCD nasłuchiwa. Trzeba podać adres IP, a port to domyślnie 2379, ten url trzeba skonfigurować na `kube-api` server.

Jeśli tworzymy cluster przy pomocy `kubeadm` tool, to ETCD zostanie automatycznie stworzone jako pod (nazwa to `etcd-master`) w `kube-system` namespace. W tym przypadku, aby wylistować informacje z bazy, trzeba komendę `etcdctl` uruchamiać bezpośrednio w podzie:

`Kubectl exec etcd-master -n kube-system etcdctl get --prefix -keys-only` - wyświetlenie wszystkich kluczy

Dane są przechowywane w specjalnej strukturze folderów. Root directory to `registry` i w nim mamy umieszczone kolejne foldery np. `pods`, `deployments`, `minions` itp.

Jeśli mamy środowisko high availability, to będzie tu kilka master nodów, czyli będzie tzw ETCD cluster. Aby każdy node wiedział o istnieniu innych trzeba ustawić parametr `initial-cluster`. Tu trzeba będzie sprecyzować wszystkie nody.

ETCDCTL jest to CLI tool, który ma dwie wersje: 2 i 3 (domyślna to 2). Mają one różne nazewnictwo komend, więc dobrze jest wybrać wersję, którą chcemy używać. W tym celu trzeba ustawić zmienną środowiskową `ETCDCTL_API`:

`export ETCDCTL_API=3`

Dodatkowo trzeba będzie ustawić ścieżki do certyfikatów. Ostateczna komenda tworząca konfigurację w podzie ETCD na Masterze to:

```
kubectl exec etcd-master -n kube-system -- sh -c "ETCDCTL_API=3 etcdctl get / --prefix --keys-only --limit=10 --cacert /etc/kubernetes/pki/etcd/ca.crt --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key"
```

Kube-API Server

Główny komponent do zarządzania całym clustrem. Wszystkie zmiany na clustrze przechodzą przez niego. Gdy uruchamiamy komendę `kubectl` to tak naprawdę wywołujemy zapytanie do Kube-API Servera. Server najpierw autentykuje zapytanie, sprawdza jego poprawność i na końcu dostarcza nam odpowiednie dane z ETCD clustra. Aby komunikować się z Kube-API serverem można używać API zamiast command line i tam wysyłać post requesty.

Przykład: Tworzenie poda

Najpierw odbywa się autentykacja użytkownika-> Sprawdzenie poprawności requestu-> Stworzenie poda i poinformowanie o tym użytkownika-> update ETCD clustra-> Potem scheduler sprawdza, że jest pod nieprzypisany do żadnego noda-> Przypisuje on go do noda i informuje użytkownika-> Informacje są znowu zaktualizowane w ETCD clustrze-> Następnie informacja ta dociera do kubleta w odpowiednim Worker Nodzie-> Kubelet tworzy poda na nodzie i prosi silnik kontenerowy (tu Docker) o deployment obrazu aplikacji-> Kubelet informuje o tych krokach Kube-API server, a on przesyła to do ETCD clustra.

Jeśli tworzymy cluster z `kubeadm` toola, to wszystkie te kroki będą automatyczne, ale jeśli ręcznie to trzeba to będzie samemu konfigurować. I Kube-API server będzie binarką, którą trzeba będzie skonfigurować na Masterze. W trakcie ręcznej konfiguracji w pliku `kube-apiserver.service` musimy

ustawić odpowiednie parametry. Certyfikaty, które służą do bezpiecznej komunikacji pomiędzy komponentami; -etcd-servers (tu precyzujeśmy, lokalizację servera ETCD, podajemy port i URL). Lokalizacja pliku to: `etc/systemd/system/kube-apiserver.service`

W przypadku deploymentu przy użyciu kubeadmin tool, Kube-API server będzie podem o nazwie kube-apiserver-master w kube-system namespace na Masterze. Tam wszystkie parametry będą w pliku: `etc/kubernetes/manifests/kube-apiserver.yaml`

Kube-API Server jest też do sprawdzenia jako proces działający bezpośrednio na Masterze, będą tam też jego opcje.

Kube Controller Manager

Zarządza on innymi kontrolerami w Kubernetesie. Kontrolery to procesy, które ciągle monitorują komponenty Kubernetesa i gdy coś jest nie tak, podejmują akcje, aby to naprawić. Wykorzystują one do swojej pracy Kube-API Server.

Node-Controller zajmuje się nodami. Sprawdza co 5 sekund stan noda i jeśli on nie odpowie to kontroler czeka 40 sekund i jeśli dalej nic się nie zmienia to status node jest zmieniany na Unreachable. Potem node ma 5 minut na naprawę, jeśli się nie uda, to wszystkie jego pody zostaną usunięte i stworzone na innym zdrowym node, jeśli pody są w replica secie.

Replication-Controller sprawdza status Replica Setów i dba o to, aby odpowiednia ilość podów była dostępna w sieci.

Jest też wiele innych kontrolerów działających w podobny sposób według swojej funkcji (Job-Controller, Namespace-Controller itp.).

Wszystkie kontrolery są spakowane w proces Kube-Controller-Manager. Jeśli to instalujemy, to inne kontrolery też będą zainstalowane. Aby to zrobić ręcznie, trzeba pobrać kube-controller-managera i uruchomić go jako service. W pliku:

`etc/systemd/system/kube-controller-manager.service` mamy opcje i parametry, które pozwalają modyfikować kontrolery i ich pracę. Można też tu włączać i wyłączać konkretne kontrolery (domyślnie wszystkie działają).

Jeśli tworzymy cluster przy pomocy kubeadmin tool to zostanie stworzony pod kube-controller-manager-master w kube-system namespace na Masterze. Plik z konfiguracjami to: `etc/kubernetes/manifests/kube-controller-manager.yaml`

Kube-Controller-Manager jest też widoczny jako proces działający na Masterze.

Kube Scheduler

Jego zadaniem jest schedulowanie podów do nodów. Jest on odpowiedzialny tylko za decydowanie, który pod ma iść, na który node. Samą pracę wykonuje kubelet. Scheduler decyduje o tym, gdzie umieścić poda na podstawie ilości podów już istniejących na nodach, zużyciu pamięci, rozmiarze nodów (CPU i RAM), czy wymaganaich dotyczących resourców. W drugim kroku scheduler wybiera node, który najlepiej pasuje do danego poda. Np. oblicza ile pamięci zostanie na node, gdy dodamy tam poda.

Pracę schedulera można customizować poprzez napisanie swoich wymagań.

Instalacja schedulera. Jeśli chcemy ręcznie to ściągamy binarkę, rozpakowujemy ją i uruchamiamy jako serwis. Jeśli robimy to przy pomocy kubeadmin tool to ustawienia będą w pliku. `etc/kubernetes/manifests/kube-scheduler.yaml`

Kube Scheduler to także proces działający na Masterze i tam też są jego opcje.

Kubelet

Jest to główny punkt kontaktu z Masterem, zajmuje się on ładowaniem lub usuwaniem kontenerów na polecenie Kube Schedulera i odpowiada za połączenie do clustra. Wysyła też raporty o stanie nodów i podów.

Kubelet rejestruje noda w Kubernetes clustrze. Jeśli otrzyma polecenie o stworzeniu poda, to wysyła prośbę do container engine (Docker) o stworzenie odpowiedniego poda na nodzie z odpowiedniego obrazu. Potem będzie on monitorował i raportował stan poda do Kube-API servera.

Jeśli chcemy zainstalować cluster przy pomocy kubeadmin tool, to kubelet nie będzie automatycznie zainstalowany. Trzeba zawsze ręcznie instalować kubelet na Worker nodach. Pobrać instalkę, rozpakować i uruchomić jako serwis. Jego proces wraz z opcjami można zobaczyć na Worker node:
`ps -aux | grep kubelet`

Kube Proxy

Jeśli mamy zdeployowane pod networking solution to wszystkie pody mogą się ze sobą komunikować w clustrze. Pod network to wirtualna sieć rozszerzona na wszystkie nody, a co za tym idzie pody.

Jeśli mamy dwa pody i na jednym jest web app, a na drugim baza danych to domyślnie mogą się ze sobą komunikować po adresie IP. Jednak dla bazy danych to IP może się zmieniać, więc lepiej jest stworzyć oddzielnie serwis tej bazy danych. Teraz aplikacja będzie się łączyć do bazy przez nazwę serwisu i następnie ruch pójdzie do poda bazy. Serwis nie jest podem, więc nie ma interfejsu, to tylko wirtualny komponent istniejący w pamięci Kuberntesa. Aby taki serwis był udostępniony dla wszystkich nodów, potrzebujemy Kube-Proxy.

Kube-Proxy to serwis działający na każdym node. Jego zadaniem jest sprawdzanie, czy nie powstał jakiś nowy serwis i jeśli tak, to jego zadaniem jest tworzenie network roli, która pozwoli dostać się przez serwis do backend poda.

Pierwszy sposób na osiągnięcie tego to użycie IPTABLES rules. Wtedy na każdym node będzie dodana tabela rutingu z okrešeniem, adresu IP serwisu i przypisanym do niego adresu IP backend poda.

Jeśli instalujemy ręcznie Kube Proxy, to trzeba pobrać instalkę, rozpakować i uruchomić jako serwis. Gdy używamy kubeadmin tool, to automatycznie zostaną stworzone dwa pody kube-proxy w kube-system namespace. Jest to stworzone jako deamon set, czyli na każdym node z clustra jest oddzielny pod.

Pody

Założenie aplikacja jest już stworzona i zbudowana na obrazach Dockerowych i jest dostępna w Docker repository np. w Docker Hub. Mamy też stworzony Kubernetes cluster, który będzie mógł pobrać obrazy z repozytorium.

W Kuberntesie naszym głównym celem jest stworzenie aplikacji w formie kontenerów umieszczonych na Worker Nodach w clustrze. Kubernetes nie tworzy jednak kontenerów bezpośrednio w clustrze, ale robi enkapsulację kontenerów na pody (obiekty Kubernetesowe). Pod to pojedyncza instancja aplikacji w Kuberntesie, jest to najmniejszy obiekt, jaki możemy tu stworzyć.

Jeśli nasza aplikacja będzie używana przez więcej użytkowników, to aby zwiększyć wydajność stworzymy dodatkowy pod tej samej aplikacji w clustrze (nie tworzymy dodatkowego kontenera w podzie). Jeśli trzeba będzie jeszcze powiększyć aplikację, to możemy dodać kolejny node z podami naszej aplikacji.

Pody najczęściej mają relację 1 do 1 z kontenerami. Jeśli chcemy to w podzie może być kilka kontenerów, np. dodatkowo może być Helper Container wspierający działającą aplikację. Wtedy oba kontenery będą się uruchamiać razem, ale i będą usuwane razem, wraz z usunięciem poda. Mogą one ze sobą się komunikować jako localhost, ponieważ są w tej samej sieci.

Jeśli chcemy mieć aplikację w Dockerze to uruchamiamy ją w kontenerze poprzez `docker run python-app`, jak trzeba będzie ją zwiększyć to dodajemy więcej kontenerów tą samą komendą. Jednak jeśli aplikacja się rozwinie i zmieni się architektura to np. możemy stworzyć sobie helper container do pomocy aplikacji: `docker run helper -link app1`. Będzie on w stosunku 1 do 1 z aplikacją, więc trzeba będzie ich stworzyć tyle, ile jest kontenerów aplikacji. Helper będzie musiał się też komunikować z aplikacją i móc z niej uzyskać dane. Dodatkowo trzeba byłoby tworzyć samemu połączenia sieciowe, aby kontenery ze sobą rozmawiały i trzeba byłoby też skonfigurować wspólny dysk lub volume dla nich. Trzeba byłoby też monitorować stan application kontenera i jeśli by padł, to trzeba byłoby też ręcznie usunąć odpowiedniego Helper kontenera. W przypadku podów i Kuberntesesa, to wszystko jest robione automatycznie. Trzeba tylko zdefiniować, jakie kontenery będzie miał pod, a one domyślnie będą mieć dostęp do tego samego storage, namespace itp. W przypadku używania Kuberntesesa, łatwiej jest rozwijać architektury i zwiększać aplikację.

Tworzenie podów. Odbywa się to przy pomocy kubectl.

`kubectl run nginx` - komenda ta, tworzy poda i uruchamia w nim odpowiedni Docker container (nazwa poda to tutaj nginx). Jeśli chcemy sprecyzować obraz to trzeba użyć dodatkowego parametru:

`kubectl run nginx --image nginx`

Image będzie pobrany z Docker Huba. Można też skonfigurować Kuberntesesa, aby pobierał obrazy z innego repozytorium.

`kubectl get nodes` - wyświetla listę nodów.

`kubectl get pods` - wyświetla listę podów na clustrze. Pokazana jest tu liczba kontenerów, status poda, liczba restartów i długość życia poda.

`kubectl explain pods --recursive` - wyświetla pomoc do tworzenia poda i tam mamy wszystkie możliwe parametry dla poda

`kubectl describe pod nazwa_poda` - wyświetlenie szczegółowej informacji o podzie. Będzie pokazane, jaki jest image, kiedy pod został stworzony, jakie ma labele, kontenery itp.

`kubectl delete pod nazwa_poda` - usuwanie poda

`kubectl edit pod nazwa_poda` - modyfikacja istniejącego poda, można zmieniać np. image

Należy pamiętać, że można edytować tylko określone sekcje dla istniejących już podów: spec.containers[*].image, spec.initContainers[*].image, spec.activeDeadlineSeconds i spec.tolerations. Jeśli chcemy np. zmienić limity i requesty cpu i memory to plik yaml nie będzie mógł być zapisany. Aby zmienić takie limity najlepiej wygenerować plik yaml istniejącego poda, edytować go, usunąć istniejącego poda i rekreować go z nowego pliku yaml. Jeśli pod jest częścią Deploymentu to jest łatwiej, bo wystarczy edytować deployment i on automatycznie zrekonstruuje pody po zmianach.

Pody i język YAML

Kubernetes używa plików YAML jako inputów do tworzenia obiektów takich jak pody, replica sety, serwisy itp. Plik YAML zawsze ma taką samą strukturę i jego cztery główne pola to (pola te są zawsze wymagane):

```
pod-definition.yml
apiVersion:
kind:
metadata:
spec:
```

`apiVersion` - wersja Kubernetes API użyta do stworzenia obiektu. Dla podów, serwisów może być v1, dla replica setu, czy deploymentu to np. apps/v1.

`kind` - tu określamy obiekt, jaki tworzymy, czyli np. Pod, Service, ReplicaSet itp.

metadata - tutaj sprecyzowujemy dane o tworzonym obiekcie, każda dana musi być przesunięta w bok tabem. Dane te są podane w postaci słownika. Dobrze jest dodawać labele do podów, bo jak mamy ich dużo to łatwiej jest je rozróżnić. W metadatacie możemy podawać tylko wartości oczekiwane przez Kubernetes, nie można tu sobie samemu stworzyć jakiegoś parametru. Ale np. już wewnątrz labels możemy wymyślać jakie chcemy.:

metadata:

```
name: myapp
```

labels:

```
app: myapp
```

spec - tu podajemy dodatkowe informacje o tworzonym obiekcie. Dla różnych obiektów, będą tu różne wartości, więc dobrze jest sprawdzić w dokumentacji. Np. dla poda będziemy tu mieć:

spec:

containers:

```
- name: nginx-container
```

```
image: nginx
```

Wartość containers to lista, bo możemy mieć kilka kontenerów w podzie. Myślnik określa to, że precyzujemy pierwszy obiekt, kolejny myślnik byłby drugim obiektem.

Po stworzeniu pliku pod-definition.yml uruchamiamy komendę:

kubectl create -f nazwa_pliku - stworzy ona odpowiednie obiekty na podstawie pliku yaml.

kubectl apply -f nazwa_pliku - jeśli stworzyliśmy jakiś deployment z pliku yaml, ale chcemy go zmodyfikować to po wprowadzeniu zmian uruchamiamy komendę apply.

Do stworzenia pliku yaml, wystarczy jakikolwiek edytor tekstu. Np. PyCharm od Pythona

Przy tworzeniu obiektów nie zawsze trzeba korzystać z plików yaml, bardzo dużo można uzyskać komendą *kubectl run*, można też jej użyć do generowania pliku yaml:

kubectl run nginx --image=nginxv - stworzenie poda o danym obrazie.

kubectl create deployment --image=nginx nginx - stworzenie odpowiedniego deploymentu.

kubectl run nginx --image=nginx --dry-run=client -o yaml - uruchomienie dry run nie wprowadzi żadnych zmian, ale jako output dostaniemy yaml file.

kubectl create deployment --image=nginx nginx --dry-run=client -o yaml > nginx-deployment.yaml - zapisanie wygenerowanego pliku yaml do pliku na dysku.

ReplicaSet

Replication Controllers - umożliwia nam posiadanie kilku instancji naszej aplikacji, tak, aby gdy główny pod się zepsuje, użytkownicy nie stracili dostępu do niej. Ustawiamy w nim ile działających podów musimy mieć i on monitoruje wszystkie i jeśli się któryś zepsuje to automatycznie uruchamia dodatkowego. Potrzebujemy też Replication Controllera, gdy chcemy mieć kilka podów i rozdzielać ruch na nie wszystkie. Replication Controller będzie wtedy tworzył nowe pody i nody w zależności od zaporzebowania.

Replica Set to nowsza wersja Replication Controllera, która jest najbardziej polecana.

Tworzenie Replication Controllera przy pomocy pliku yaml:

```
rc-definition.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: frontPOD
    spec:
      containers:
        - name: nginx-container
          image: nginx
```

Budowa pliku jest klasyczna, ale w sekcji spec musimy podać template. Jest to określenie poda, którego Replica Set będzie budował, gdy zajdzie taka potrzeba. W template trzeba podać metadatę i spec, z pliku do tworzenia poda. Dodatkowo w sekcji spec Replica Setu podajemy paramter replicas, który określi ile podów ma być na danym node.

Teraz możemy uruchomić ten plik komendą:

```
kubectl create -f plik
```

Najpierw zostanie stworzony Replication Controller, a potem od razu odpowiednia liczba podów.

kubectl get replicationcontroller - wyświetli wszystkie Replication Controller'y na nodzie. Widzimy też ile podów powinno być przez niego stworzone. Jak wylistujemy pody to zobaczymy, że ich nazwa zaczyna się od nazwy Replication Controller'a.

Tworzenie Replica Setu:

```
replicaset-definition.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
        - name: nginx-container
          image: nginx
  replicas: 3
  selector:
```

Plik yaml jest bardzo podobny do Replication Controllera, tylko jest inne apiVersion i wymagane jest pole selector. Pole to pomaga ustalić Replica Setowi, jakie pody do niego należą. Robimy to dlatego, że Replica Set może też zarządzać podami, które nie są stworzone w trakcie tworzenia Replica Setu. Pole selector jest też w Replication Controllerze, ale tam nie jest wymagane. Selector precyzujemy poprzez dodanie matchLabels, które szukają labeli dodanych na podach. Jeśli się jakieś będą zgadzać, to te pody będą brane pod uwagę przy tworzeniu Replica Setu.

```
  selector:
    matchLabels:
      type: front-end
```

kubectl get replicaset - wyświetlenie wszystkich replica setów.

kubectl delete replicaset nazwa - usunie ReplicaSet z wszystkimi jego podległymi podami

ReplicaSet jest w stanie monitorować istniejące wcześniej pody i jeśli coś się wydarzy z nimi to będzie tworzył nowe, tak aby zgadzała się odpowiednia ich ilość. ReplicaSet wie, które pody monitorować na podstawie Labeli dodanej na podzie i dodanej w selectorze w ReplicaSecie. Monitorowane będą tylko te pody, których labele się zgadzają z matchLabels.

Gdy chcemy skalować ReplicaSet, czyli np. mieć 6 instancji zamiast 3 to możemy:

- zmienić definicję ReplicaSetu modyfikując replicaset-definition.yaml, potem trzeba uruchomić komendę *kubectl replace -f replicaset-definition.yaml* aby wprowadzić zmiany
- uruchomić komendę *kubectl scale --replicas=6 -f replicaset-definition.yaml*

Deployments

Gdy chcemy zdeployować aplikację na produkcji i chcemy mieć wiele jej instancji (podów), dodatkowo, gdy pojawi się nowa wersja obrazu Dockerowego, to chcemy, aby się automatycznie aktualizowało, ale nie wszystkie na raz, aby nie wpłynęło to na użytkowników. Dodatkowo, jak jest jakiś błąd to chcemy w łatwy sposób cofnąć wprowadzone zmiany. Wszystko do możemy osiągnąć przy pomocy Deploymentów w Kubernetesie. Jest to obiekt wyższy w hierarchii niż ReplicaSet. Pozwalają one na upgrade instancji przy pomocy rolling updates, cofanie zmian, zatrzymywanie i wznawianie zmian, gdy jest to potrzebne.

Tworzenie deploymentu. Plik deployment-definition.yaml, jest prawie identyczny z plikiem do tworzenia Replica Setów, tylko kind jest inny.

```
deployment-definition.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
        - name: nginx-container
          image: nginx
  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

kubectl create -f deployment-definition.yaml - tworzenie deploymentu z pliku. Po uruchomieniu zostaną też stworzone wszystkie obiekty w nim zapisane, czyli ReplicaSet i pody. Ich nazwy będą się zaczynać od nazwy deploymentu

kubectl get deployments - wyświetlenie wszystkich deploymentów

kubectl get all - wyświetlenie wszystkich stworzonych obiektów

Namespaces

Namespace pozwala nam oddzielić jedne resourcy od drugich. Wewnątrz namespace, wszystkie pody mogą się ze sobą komunikować po swojej nazwie i mają tam określone resourcy i zasady pracy. Jeśli chcą się połączyć z podami z innego namespace, to muszą używać full name. Możemy np. rozdzielać resourcy prodowe od devowych, jeśli mamy wszystko na jednym clustrze. Wtedy można do nich dodać różne policy określające zasady pracy. Można też ograniczyć maksymalną ilość resourców dopuszczalną w namespace.

Na początku stworzenia clustra, mamy domyślny namespace o nazwie *default*, w którym tworzymy pody. Jest też automatycznie stworzony namespace systemowy, w którym Kubernetes sam tworzy resourcy potrzebne do networkingu i DNS. Namespace ten nazywa się *kube-system*. Jest też namespace *kube-public*, w którym mamy resourcy, które są używane przez wszystkich użytkowników.

Przykładowo, gdy web service chce się łączyć z bazą danych w tym samym namespace, musi użyć tylko nazwy bazy:

```
mysql.connect("database")
```

Jeśli jednak połączenie miałoby być do bazy w innym namespace, to trzeba zmodyfikować nazwę dodając docelowy namespace:

```
mysql.connect("database.namespace.svc.cluster.local")
```

Cluster.local to domyślna domena dla clustra Kubernetesowego, svc to subdomain dla serwisów.

kubectl get pods - wyświetlenie wszystkich podów, ale tylko w namespace *default*

kubectl get pods -n nazwa_namespace - wyświetlenie wszystkich podów z danego namespace.

Jeśli chcemy stworzyć poda z pliku yaml, ale w konkretnym namespsace, to trzeba to zdefiniować:

```
kubectl create -f pod-definition.yaml -n nazwa_namespace
```

Można też dodać parametr namespace w sekcji metadata pliku yaml do tworzenia poda.

```
kubectl get pods --all-namespaces
```

 - wyświetlenie wszystkich podów ze wszystkich namespace

Tworzenie namespace:

Można użyć pliku namespace-dev.yaml, należy pamiętać, że nie ma w nim sekcji spec:

```
namespace-dev.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: dev
```

Można też użyć komendy:

```
kubectl create namespace nazwa
```

kubectl config set-context \$(kubectl config current-context) --namespace=nazwa - ustawienie danego namespace jako głównego, na którym się znajdujemy. Polecenie to sprawdza obecny context i do niego przypisuje dany namespace.

Jeśli chcemy ustawić ograniczenia dla resourców w namespace, musimy stworzyć ResourceQuota, który jest osobnym obiektem i najlepiej zrobić to plikiem yaml.

```

Compute-quota.yaml

apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: dev

spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 5Gi
    limits.cpu: "10"
    limits.memory: 10Gi

```

Podajemy tu nazwę namespace, który chcemy ograniczać i w sekcji spec ograniczenia.
kubectl create -f compute-quota.yaml - wprowadzenie w życie ograniczeń z pliku yaml

Services

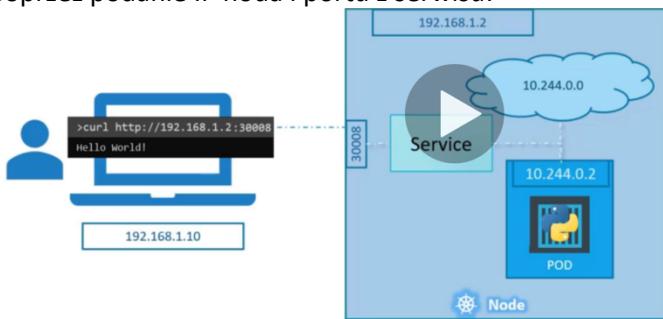
Serwisy umożliwiają połączenie pomiędzy różnymi komponentami wewnętrz i na zewnątrz aplikacji oraz łączenie aplikacji z innymi userami, czy aplikacjami. Założymy, że mamy grupę podów (frontend), grupę podów (backend) i grupę, która łączy się do zewnętrznego data source. To serwisy umożliwiają użytkownikom łączenie się do tych grup i umożliwiają także grupom łączenie się ze sobą.

Przykład:

Mamy node o adresie 192.168.1.2, w którym jest prywatna sieć 10.244.0.0 i w niej pod 10.244.0.2, na którym jest web server. Nasz komputer jest w sieci noda, więc ma adres 192.168.1.10, ale nie może bezpośrednio pingować poda po jego wewnętrznym IP.



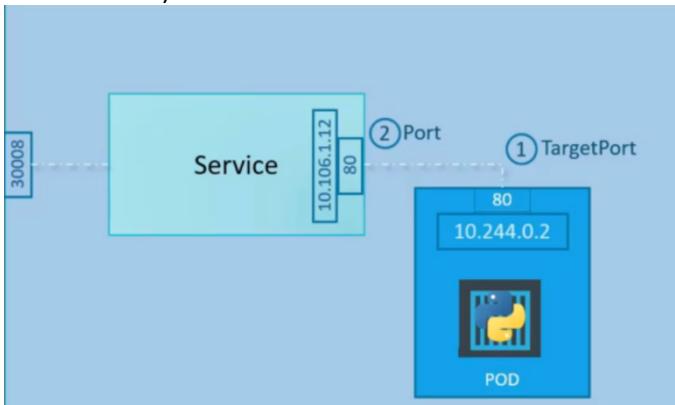
Można byłoby się zalogować SSH do noda i stamtąd curllem, lub przez GUI dostać się do strony internetowej, ale jeśli chcemy to zrobić z zewnątrz to właśnie service może nam pomóc. Jest to obiekt w nodzie, którego jednym z zadań jest nasłuchiwanie na danym porcie i przekazanie ruchu na tym samym porcie do poda. W związku z tym będziemy mogli się zalogować do web servera z zewnątrz poprzez podanie IP noda i portu z serwisu:



Jest to tak zwany NodePort service (sprawia, że pod jest dostępny na danym porcie w nodzie). Inne rodzaje serwisów to: Cluster IP (serwis tworzy virtualne IP wewnętrz clustra, aby umożliwić

komunikację pomiędzy serwisami) oraz LoadBalancer (tworzony jest load balancer, dobrym zastosowaniem jest to, gdy chcielibyśmy rozdzielać ruch pomiędzy różne pody we frontendzie).

NodePort - Mamy tu do czynienia z trzema portami. TargetPort, jest to port, po którym chcemy się połączyć do poda (tu jest to 80, bo mamy web server). Port, jest to port w serwisie, który będzie łączył się z podem. Każdy service ma też swoje wewnętrzne IP, które jest nazywane Cluster IP of the Service. NodePort, to port, do którego będziemy łączyć się z zewnątrz (tu 30008. Przedział tych portów to 30000-32767).



Tworzenie przy użyciu pliku yaml. W sekcji spec musimy podać typ serwisu, jaki chcemy stworzyć i w przypadku NodePort podajemy trzy porty (obowiązkową wartością jest port, bo nodePort może być przypisany losowo, a targetPort będzie zapisany z tą samą wartością co port domyślnie):

```
service-definition.yml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: NodePort
  ports:
    - targetPort: 80
      port: 80
      nodePort: 30008
  selector:
    app: myapp
    type: front-end
```

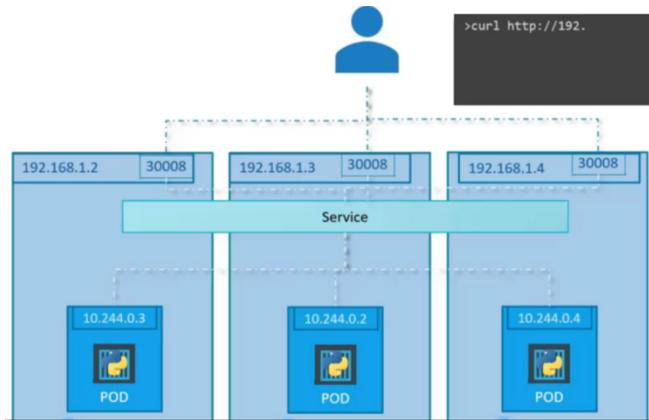
Można mieć wiele mappingów w jednym serwisie. Aby połączyć serwis z danym podem musimy użyć labeli, które mają pody i przypisać je w sekcji selector w serwisie. Dzięki temu Kubernetes będzie wiedziały, do jakich podów należy kierować ruch. W ten sposób możemy też wysyłać ruch do wielu podów, mających odpowiednie labele, wtedy będą one wybierane losowo i serwis będzie działał, jako load balancer.

`kubectl create -f service-definition.yml` - stworzy nam serwis

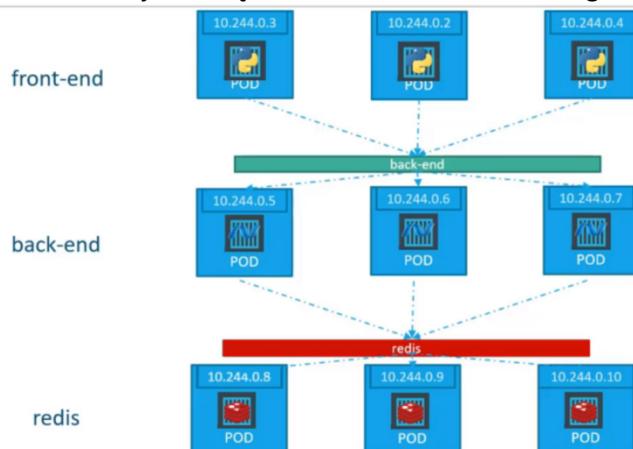
`kubectl get services` - wyświetlenie wszystkich serwisów. Przy wyświetaniu zobaczymy też wiele innych informacji, takich jest Cluster IP, typ, port itp.

`kubectl describe service nazwa` - wyświetlenie szczegółowych informacji o serwisie

Po stworzeniu takiego serwisu będziemy mogli się już połączyć z zewnątrz z naszym web serwerem. Jeśli mamy pody na kilku nodach w clustrze, to Kubernetes automatycznie tworzy serwis, który jest udostępniony na wszystkich nodach w clustrze i mapuje porty dla wszystkich podów z labelami. Dzięki temu będzie można łączyć się do web serwera używając adresów IP dowolnego noda.



Cluster IP - przykład, mamy aplikację podzieloną na części: frontend, backend, key-store (Redis) i te części muszą się komunikować ze sobą po kolej. Każdy z podów ma swoje IP, ale nie jest ono statyczne, bo jak pod padnie i nowy tam powstanie, to będzie już miał inny adres. Dodatkowo skąd pod z fronenda będzie wiedział, do którego poda z backenda się połączyć. Tu przydaje się ClusterIP service. Tworzy on wspólny interfejs dla grupy podów i pomaga się przez niego łączyć do tych podów. Każdy interfejs może być skalowany automatycznie przez zwiększanie, czy zmniejszanie ilości podów. Każdy serwis dostaje nazwę i adres IP i to właśnie do niego należy kierować połączenie sieciowe:



Tworzenie service przy pomocy pliku yaml odbywa się tak, jak dla poprzedniego serwisu, ale tutaj w portach podajemy TargetPort (port, na którym ruch ma dojść do podów) oraz Port (port, na którym nasłuchuje serwis). Selector pozwala nam precyzować, jakie pody będą połączone z tym serwisem przy pomocy labeli.

```
service-definition.yaml

apiVersion: v1
kind: Service
metadata:
  name: back-end

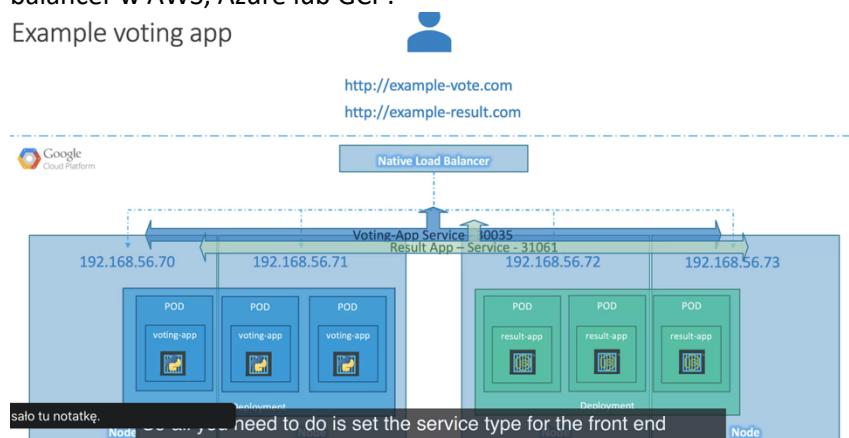
spec:
  type: ClusterIP
  ports:
    - targetPort: 80
      port: 80

  selector:
    app: myapp
    type: back-end
```

Aby dostać się do serwisu, należy połączyć się z nim przez nazwę lub ClusterIP.

LoadBalancer - przykład: mamy 4 nody i każda para odpowiada za inne pody (voting-app i result-app). Tworzymy serwisy typu NodePort, aby móc dostać się z zewnątrz do podów, ale jakiego URLa trzeba użyć? Mamy 4 nody, więc będą 4 różne adresy IP i 2 porty. Należy pamiętać, że jeśli mamy pody tylko na dwóch nodach, to i tak będą one dostępne przez adresy IP innych nodów, bo serwis jest zawsze deployowany na wszystkie nody w clustrze. Chcielibyśmy jednak, mieć tylko jeden URL do voting-app i jeden do result-app, więc musimy stworzyć load balancer. Możemy to zrobić tworząc cloudowy load balancer w AWS, Azure lub GCP:

Example voting app



W pliku yaml trzeba zmienić typ serwisu na LoadBalancer i dzięki temu osiągniemy pożądany rezultat. Działa to jednak tylko u znanych dostarczycieli chmury:

```
service-definition.yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: LoadBalancer
  ports:
    - targetPort: 80
      port: 80
      nodePort: 30008
```

Imperative vs Declarative approach

W IaaC mamy dwa podejścia do zarządzania infrastrukturą. Podejście imperative to podawanie po kolej komend mówiących, jak wykonać dane zadanie. Jeśli przy pierwszym uruchomieniu komend nie wszystkie się wykonają, to może powstać problem, bo nie wiadomo, czy puszczać wszystkie od początku, czy tylko część. Tworzymy wtedy dodatkowe elementy, jak health checki, żeby sprawdzić, czy poszczególne elementy się stworzyły. Podejście deklaratywne polega na ustaleniu końcowego celu, a system sam będzie pracował nad jego osiągnięciem, nie ważne jaką drogą. Przykład: w imperatywnym podajemy komendy, a w deklaratywnym gotowy plik yaml, mówiący, co chcemy mieć:

Imperative

1. Provision a VM by the name ‘web-server’
2. Install NGINX Software on it
3. Edit configuration file to use port ‘8080’
4. Edit configuration file to web path ‘/var/www/nginx’
5. Load web pages to ‘/var/www/nginx’ from GIT Repo - X
6. Start NGINX server

Declarative

```
VM Name: web-server
Database: nginx
Port: 8080
Path: /var/www/nginx
Code: GIT Repo - X
```

W Kubernetesie wszystkie używane komendy to imperative approach:

Imperative

```
> kubectl run --image=nginx nginx
> kubectl create deployment --image=nginx nginx
> kubectl expose deployment nginx --port 80
> kubectl edit deployment nginx
> kubectl scale deployment nginx --replicas=5
> kubectl set image deployment nginx nginx=nginx:1.18
> kubectl create -f nginx.yaml
> kubectl replace -f nginx.yaml
> kubectl delete -f nginx.yaml
```

Mówią one dokładnie, krok po kroku, co i jak ma być wykonane. Komendy te możemy podzielić na dwie grupy: tworzące (run, create, expose) i modyfikujące (edit, scale, set image). Pozwalają one w

szynki sposób wykonać potrzebne dla nas małe zmiany i są przydatne na egzaminach. Niestety komendy te mogą być skomplikowane i mieć problemy, gdy chcemy wykonać duże modyfikacje. Innym minusem jest to, że te komendy po uruchomieniu nie są prawie nigdzie zapisywane, więc inna osoba nie będzie wiedziała, jak został stworzony dany obiekt. W przypadku tworzenia obiektów użyjemy `kubectl create -f`, a w przypadku modyfikowania `kubectl edit obiekt`. Niestety po wprowadzaniu zmian komendą `edit`, zmiana ta też nie jest nigdzie zapisana, tylko będzie w serwerowej definicji obiektu, nasz yaml file, który tworzył ten obiekt, nie będzie zaktualizowany. Dlatego lepszym pomysłem jest wprowadzenie zmiany w pliku yaml i uruchomienie zmian komendą `kubectl replace -f`. Dzięki temu żadna ze zmian nie zostanie stracona i będą one śledzone. Jeśli chcemy całkowicie usunąć obiekty i je zredukować to dodajemy flagę `force`: `kubectl replace --force -f plik.yaml`. W tym podejściu musimy często sprawdzać aktualny stan obiektów i na tej podstawie podejmować decyzję, bo np. jeśli chcemy stworzyć obiekt, który już istnieje to będzie błąd, tak samo błąd będzie jeśli będziemy chcieć modyfikować obiekt nieistniejący.

W podejściu declarative mamy tylko komendę `kubectl apply`, gdyż ona wprowadza zmiany z plików yaml, które opisują końcowy stan deploymentu:

Declarative

> `kubectl apply -f nginx.yaml`

Plusem plików yaml jest to, że mamy wszystko zapisane w pliku i każdy ma do tego dostęp. Można dodać w GitHubie np. dodatkowe walidacje przed każdą zmianą, tak aby każdy wiedział co i jak jest stworzone. Tutaj do tworzenia używamy komendy `kubectl apply -f` która jest na tyle inteligentna, że jeśli obiekt już istnieje to będzie modyfikowany, a jak nie, to zostanie stworzony. Można też podać jako parametr ścieżkę do plików yaml zamiast pojedynczego pliku:

`kubectl apply -f /bin/scripts`. Przy wprowadzaniu modyfikacji używamy tej samej komendy po zmianie pliku yaml.

Na egzaminie jeśli potrzebujemy szybkiej zmiany, to lepiej jest stosować imperative, bo zaoszczędzi to nam czas. Do dużych modyfikacji, czy deploymentów lepiej użyć declarative. Imperative jest też przydatne w szybkim generowaniu plików yaml. Przydatne opcje i kilka przykładów:

`--dry-run=client` - po uruchomieniu komendy z tą flagą obiekt nie będzie utworzony, ale Kubernetes powie nam, czy obiekt może być stworzony i czy komenda jest okej.

`-o yaml` - dodanie na końcu tej opcji wyświetli gotowy plik yaml danego obiektu, więc będzie go można łatwo zapisać do pliku.

`kubectl run nginx --image=nginx --dry-run=client -o yaml > pod.yaml` - tworzenie pliku yaml dla poda.

`kubectl run nginx --image=nginx --labels=tier=frontend` - tworzenie poda z określonego image, z określzoną labelką.

`kubectl create deployment --image=nginx nginx --namespace=nazwa --dry-run=client -o yaml` - tworzenie pliku yaml dla deploymentu w odpowiednim namespace (można teraz ten plik modyfikować i np. dodać replicasety).

`kubectl expose pod redis --port=6379 --target-port=6379 --name redis-service --dry-run=client -o yaml` - tworzenie serwisu i automatycznie dodanie labelek danego poda jako selector. Pod będzie wystawiony na określonym porcie, a domyślny typ serwisu to ClusterIP.

`kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml` - tworzenie serwisu w inny sposób, ale tutaj nie możemy ustawić ręcznie selectora, jego wartość to będzie automatycznie `app=redis` na podstawie nazwy, więc jeśli pody mają inne labele, to nie zadziała. W tym przypadku lepiej jest modyfikować plik yaml.

`kubectl expose pod nginx --type=NodePort --port=80 --name=nginx-service --dry-run=client -o yaml` - w tym przypadku selector zadziała, ale nie można tak sprecyzować NodePort, więc i tak będzie nam potrzebny plik yaml.

`kubectl run nazwa_poda --image=obraz --port 8080` - stworzenie poda i wystawienie go na porcie 8080 w kontenerze.

kubectl apply command

Komenda korzysta z lokalnego pliku yaml do tworzenia obiektów. Bierze pod uwagę plik yaml, obecną konfigurację obiektu i ostatnią zaakceptowaną zmianę konfiguracji. Na tej podstawie jeśli obiekt nie istnieje, zostanie stworzony.

Po uruchomieniu komendy, gdy tworzymy nowy obiekt w obecnej konfiguracji obiektu zostanie dodana sekcja status opisująca to co dzieje się w tym momencie z obiektem. Gdy modyfikujemy istniejący obiekt, w trakcie działania komendy plik yaml jest przekonwertowany na .json (jest to ostatnia zaakceptowana konfiguracja, która pomaga nam śledzić, jakie pola zostają dodane lub usunięte w trakcie modyfikacji). Jeśli wprowadzamy jakieś zmiany to porównywany jest plik yaml z obecną konfiguracją i tam zostanie naniesiona zmiana. Potem nowa wersja obecnej konfiguracji jest zapisywana w pliku .json, jako ostatnia zaakceptowana konfiguracja.

Plik.json z ostatnią zaakceptowaną konfiguracją jest przechowywany bezpośrednio w aktualnej konfiguracji obiektu, w pamięci Kuberntesesa.

```
Live object configuration
apiVersion: v1
kind: Pod

metadata:
  name: myapp-pod
  annotations:
    kubectl.kubernetes.io/last-applied-configuration:
      {"apiVersion": "v1", "kind": "Pod", "metadata": {"annotations": {}, "labels": {"run": "myapp-pod"}, "spec": {"containers": [{"image": "nginx:1.18", "name": "nginx-container"}]}}, "type": "front-end-service"
  labels:
    app: myapp
    type: front-end-service
spec:
```

Należy pamiętać, że pole to w metadacie jest dodawane tylko przy użyciu komendy *kubectl apply*, a więc przy używaniu podejścia declarative. Komendy *kubectl create* i *kubectl replace* nie dodają tego pola.

Scheduling

Manual scheduling

Każdy pod ma w pliku yaml w sekcji spec pole nodeName, które domyślnie nie jest ustawione. My go nigdy nie dodajemy, ale Kubernetes robi to automatycznie. Scheduler sprawdza wszystkie pody, czy jest w nich obecne to pole, jeśli nie, to taki pod będzie musiał być zascheduleowany. Następnie uruchamiany jest algorytm schedulingu i scheduler dobiera odpowiedni node dla tego poda. Na końcu przypisuje nazwę noda w pliku yaml do poda.

Jeśli nie mamy schedulera to trzeba ręcznie przypisać wartość nodeName w pliku yaml, podczas tworzenia poda.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    nodeName: node01
```

Bez tego pody będą w stanie Pending. Wartość tę można przypisać ręcznie tylko podczas tworzenia. Jeśli chcemy przypisać node do istniejącego poda trzeba stworzyć binding object i wysłać request do binding API poda. Plik yaml binding objectu:

```
Pod-bind-definition.yaml
apiVersion: v1
kind: Binding
metadata:
  name: nginx
target:
  apiVersion: v1
  kind: Node
  name: node02
```

W tym pliku precyzujemy nazwę noda. Następnie wysyłamy post request do binding API:

```
curl --header "Content-Type:application/json" --request POST --data '{"apiVersion":"v1", "kind": "Binding" ... }' http://$SERVER/api/v1/namespaces/default/pods/$PODNAME/binding/
```

Tutaj wartość data to binding object przekonwertowany do formatu .json

Labels and Selectors

Labele i selectory są używane do grupowania obiektów, tak aby je móc potem filtrować. Labele to parametry danego obiektu (np. rodzaj aplikacji, funkcja itp.), a selectory filtrują te obiekty.

Labele precyzujemy w sekcji metadata pliku yaml, można ich dodać tyle ile się chce, są one w formacie key:value.

```
pod-definition.yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end

spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
      - containerPort: 8080
```

`kubectl get pods --selector app=App1` - wyświetlenie podów z labelą app o wartości App1

`kubectl get pods -l app=App1` - taki sam efekt, jak w komendzie powyżej

`kubectl get pods -l app=App1,env=dev,function=Front-end` - wyświetlenie obiektów mających kilka labeli (ustawienie filtra na kilka labeli)

`kubectl get pods --show-labels` - wyświetlenie wszystkich podów z ich labelami

`kubectl get all--selector app=App1` - wyświetlenie wszystkich obiektów z labelą app o wartości App1

Jeśli chcemy stworzyć Replicaset to najpierw precyzujemy labele ReplicaSetu w jego sekcji metadata, a potem dodatkowo w sekcji spec->template->metadata->labels precyzujemy labele podów, które będą w RepilcaSecie. Aby połączyć pody z ReplicaSetem dodajemy dodatkowe pole matchLabels w sekcji spec->selector, aby wybrać, na podstawie których labeli będziemy przyłączać pody do ReplicaSetu. Aby ReplicaSet się stworzył, te labele muszą się zgadzać.

```
replicaset-definition.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      app: App1
  template:
    metadata:
      labels:
        app: App1
        function: Front-end
    spec:
```

W przypadku tworzeniu serwisu, w sekcji spec->selector ustawiamy też labele, tak aby serwis był przypisany do danych podów z ReplicaSetu.

```
service-definition.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: App1
  ports:
    - protocol: TCP
```

Dodatkowym miejscem na ustawianie informacji na temat obiektu są Annotations. Ustawiamy je np. w sekcji metadata->annotations dla ReplicaSetu. Pozwalają one przechowywać dane informacyjne na temat obiektów.

```
replicaset-definition.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
  annotations:
    buildversion: 1.34
spec:
  replicas: 3
  selector:
```

Taints and Tolerations

Są one używane do ustawiania restrykcji dla podów i nodów i do kontrolowania relacji między nimi (czy dany pod może iść na dany node). Domyślnie scheduler będzie rozdysponowywać pody do nodów, tak aby na każdym była równa ilość. Jeśli chcemy, aby na danym node były tylko określone pody to dodajemy mu Taint. Domyślnie pody nie mają żadnej tolerancji, więc żaden domyślny pod nie będzie zaschedulowany na tym node. Aby pod mógł być umieszczony na tym node, trzeba mu dodać Toleration na dany Taint, który ma node. W ten sposób oddzielony jest np. master node, który ma nadany domyślnie Taint, tak aby żaden pod nie był na nim stworzony.

kubectl describe node kubemaster | grep Taint - wyświetlenie tego Taintu na masterze.

kubectl taint nodes nazwa_noda key=value:taint-effect - ustawienie taint dla noda. Jest on w postaci key:value. Taint-effect określa co się stanie z podem, gdy nie będzie miał tolerancji na taint. Są 3 opcje: NoSchedule (nie zostanie umieszczony na tym node, jeśli nie będzie miał miejsca na innych to będzie w statusie Pending), PreferNoSchedule (system postara się nie umieszczać poda na tym nodzie, ale nie jest to gwarantowane) i NoExecute (pod nie będzie umieszczony na node, a jeśli już na nodzie są inne pody, które nie tolerują Taint, to będą z niego eksmitowane, czyli zkilowane).

Jeśli chcemy dodać Toleration na podzie to ustawiamy to w pliku yaml.

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - name: nginx-container
      image: nginx
  tolerations:
    - key: "app"
      operator: "Equal"
      value: blue
      effect: NoSchedule
```

Należy podać 4 wartości dla poda: key, operator, value i effect bazując na Taint. Przykład z obrazka to app=blue:NoSchedule. Należy pamiętać o znaku cudzysłowu.

Należy pamiętać o tym, że jeśli dany pod ma jako jedyny Toleration na dany Taint, to wcale nie znaczy, że zostanie on umieszczony na nodzie z Taintem. Scheduler stara się rozdzielać pody po równo i może się zdarzyć, że będzie on dodany do innego noda. Jeśli chcemy sprecyzować, aby dany pod szedł tylko do określonego noda to trzeba ustawić Node Affinity.

Node Selectors

Używamy tego, aby kierować określone pody na określone nody i bardziej kontrolować proces umieszczania podów na nody. Node Selector ustawiamy w pliku yaml.

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - name: data-processor
      image: data-processor
  nodeSelector:
    size: Large
```

Size: Large to key:value pair, która jest ustawiona jako label na node. Scheduler używa tych labeli do przypisania podów do nodów. Aby ten sposób zadziałał, nody muszą mieć labela przed tworzeniem podów.

`kubectl label nodes nazwa_noda key=value` - określenie labeli dla noda

Node Selector to prosty sposób precyzowania podów i ma on swoje ograniczenia. Jeśli chcemy np. aby pody były umieszczone na Large or Medium node, lub nie na Small node to Node Selector nie poradzi sobie z takim wyrażeniem logicznym. Trzeba tutaj użyć Node Affinity.

Node Affinity

Podobnie jak Node Selector sprawia, że pody zostają umieszczone na odpowiednich nodach. Node Affinity jest też ustawiana w pliku yaml poda.

```
pod-definition.yml
apiVersion:
kind:
metadata:
  name: myapp-pod
spec:
  containers:
    - name: data-processor
      image: data-processor
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: size
              operator: In
              values:
                - Large
```

W sekci spec->affinity->nodeAffinity->requiredDuringSchedulingIgnoredDuringExecution->nodeSelectorTerms ustawiamy key:value pair, na podstawie którego nastąpi rozmieszczanie podów. Mamy tu dodatkowo operator In, który sprawia, że pod zostanie umieszczony na nodzie, który ma jakąkolwiek wartość labeli size z podanych niżej w values, np. poniżej Large moglibyśmy mieć też Medium. Inne możliwe wartości operatora to NotIn, wtedy podajemy te, na których ma go nie być; Exists (wtedy tylko sprawdzane jest istnienie dalej labeli na nodzie).

Zachowanie podów, w przypadku, gdy żaden node nie odpowiada ich kryteriom lub, gdy labele zostaną zmienione określa długi paramter pod nodeAffinity:

-requiredDuringSchedulingIgnoredDuringExecution - pod musi być umieszczony na nodzie z odpowiednią labelą, jeśli takiego nie ma, to pod nie będzie zaschedulowany. W przypadku zmiany labeli po uruchomieniu, nie wpłynie to na pracę podów, bo jest Ignored w drugiej części.

-requiredDuringSchedulingIgnoredDuringExecution - pod będzie umieszczony na nodzie z odpowiednimi labelami, jeśli to możliwe, jeśli nie to na innym dostępnym. W przypadku zmiany labeli po uruchomieniu, nie wpłynie to na pracę podów, bo jest Ignored w drugiej części.

Aby dodać labele do istniejących już nodów, można je np. edytować. Można też dodać flagę --show-labels przy get żeby zobaczyć istniejące już labele.

Jeśli chcemy aby konkretne pody były umieszczony tylko i wyłącznie na określonym nodzie i aby inne pody nie były umieszczone na tym nodzie musimy użyć kombinacji Taints and Toleration i Node Affinity. Pierwsze zablokuje inne pody przed umieszczeniem na nodzie z Taintem, a Node Affinity sprawi, że pody będą szukały tylko i wyłącznie noda z określonymi labelami.

Resource Requirements and Limits

Każdy node ma swoje CPU, memory i dysk, z których korzysta. Pody działające w node konsumują część tych resourców. Scheduler w trakcie umieszczania podów na nody, sprawdza dostępne zasoby i dodaje pody tam, gdzie będą miały miejsce. Jeśli nie ma już miejsca na żadnym z nodów, to pod nie zostanie zaschedulowany, tylko będzie w stanie Pending. W eventach znajdziemy informacje na ten temat. Aby ustawić domyślne limity pamięci i CPU dla wszystkich podów w namespace trzeba stworzyć LimitRange na podstawie plików yaml.

```

apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
spec:
  limits:
    - default:
        memory: 512Mi
  defaultRequest:
    memory: 256Mi
  type: Container

```

```

apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
    - default:
        cpu: 1
  defaultRequest:
    cpu: 0.5
  type: Container

```

Domyślnie pod requestuje dla kontenera 0.5 CPU i 256MB pamięci przy tworzeniu. Te wartości można zmieniać w pliku yaml dla poda w sekcji spec->containers->resources.

`pod-definition.yaml`

```

apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
      ports:
        - containerPort: 8080
      resources:
        requests:
          memory: "1Gi"
          cpu: 1

```

Dla CPU przycyjemy wartości liczbowe bez jednostek. Możemy je też podawać z literką m (mili), więc 1=1000m. Najmniejsza wartość, jaką możemy podać to 1m. 1 CPU oznacza 1 AWS VCPU.

Dla pamięci Mi to Mebibyte, różnica od MB, polega na tym, że 1MB to 1000kB, a 1Mi to 1024kB. Można też podać wartość w GB, wtedy będzie to sama litera G.

W świecie Dockera kontenery nie mają żadnych ograniczeń i mogą konsumować tyle resourców ile chcą, dlatego w Kubernetesie jest ograniczenie, że każdy pod domyślnie może używać maksymalnie 1CPU i 512Mi. Jeśli chcemy to zmienić, to dodajemy to w pliku yaml dla poda pod sekcją resources. Należy pamiętać, że limity są ustawiane dla każdego poda oddzielnie.

```

resources:
  requests:
    memory: "1Gi"
    cpu: 1
  limits:
    memory: "2Gi"
    cpu: 2

```

Jeśli pody będą przekraczać limity, to w przypadku CPU będzie to zablokowane i nie dostaną one więcej, ale w przypadku pamięci nie ma blokady i jeśli przekraczają limit zostaną usunięte.

DaemonSets

Działają jak ReplicaSet, z tą różnicą, że tworzą po jednej replice naszego poda na każdym nodzie w clustrze. Jeśli jakiś nowy node powstanie, to automatycznie zostanie tam też stworzona kopia. Kubernetes używa NodeAffinity, aby na każdym nodzie był jeden pod. Zastosowaniem tego może być tworzenie podów do monitorowania lub zbierania logów, wtedy chcemy, aby było to robione automatycznie na każdym clustrze. Przykładowo możemy tak zdeployować kube-proxy, aby był na każdym nodzie.

Tworzenie DaemonSetu jest identyczne, jak ReplicaSetu, też trzeba podać labele, które będą określać pody do niego należące, a definicja poda jest określona w template.

```

daemon-set-definition.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-daemon
spec:
  selector:
    matchLabels:
      app: monitoring-agent
  template:
    metadata:
      labels:
        app: monitoring-agent
    spec:
      containers:
        - name: monitoring-agent
          image: monitoring-agent

```

`kubectl get daemonsets --all-namespaces` - wyświetlenie wszystkich deamon setów ze wszystkich namespace

`kubectl describe daemonset nazwa` - wyświetlenie większej ilości informacji o daemonsecie

Static pods

Kubelet dostaje informacje z api-servera o tym, jakie pody załadować na nody. Decyzje o tym co ładować podejmuje scheduler i przesyła ją do api-servera. Jeśli jednak nie byłoby Kubernetes clustra, a więc nie byłoby api-servera, to kubelet może sam zarządzać nodem, bo umie tworzyć pody, a kontenerami się zajmie Docker.

Tworzenie podów będzie jednak ciężkie, bo nie mamy kube-api servera, więc nie można uruchamiać komend. Trzeba wtedy stworzyć samemu pliki yaml i wrzucić je do odpowiednio skonfigurowanego directory (aby je skonfigurować trzeba je podać jako opcję w trakcie uruchamiania serwisu), z którego kubelet będzie odczytywał podowe pliki yaml.

Kubelet będzie nadzorował status poda i zrestartuje go, jeśli będzie to potrzebne, dodatkowo jak wprowadzimy zmiany w pliku yaml, to zrestartuje pody aby wprowadzić zmiany. Jak usuniemy plik yaml, to kubelet usunie poda. Stworzone w ten sposób pody to Static Pods, można tak tworzyć jedynie pody.

Konfigurowanie directory dla plików yaml:

-ustawienie go jako opcji podczas tworzenia serwisu

```
kubelet.service
ExecStart=/usr/local/bin/kubelet \\
--container-runtime=remote \\
--container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
--pod-manifest-path=/etc/Kubernetes/manifests \\
--kubeconfig=/var/lib/kubelet/kubeconfig \\
--network-plugin=cni \\
--register-node=true \\
--v=2
```

-można też podać ścieżkę do pliku konfiguracyjnego w serwisie i potem w pliku podać odpowiednią ścieżkę (ta metoda jest używana przez kubeadmin tool)

```
kubelet.service
ExecStart=/usr/local/bin/kubelet \\
--container-runtime=remote \\
--container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
--config=kubeconfig.yaml \\
--kubeconfig=/var/lib/kubelet/kubeconfig \\
--network-plugin=cni \\
--register-node=true \\
--v=2
```

kubeconfig.yaml
staticPodPath: /etc/kubernetes/manifest

Aby wyświetlić statyczne pody używamy komendy *docker ps*, bo nie mamy kube-api.

Gdy mamy już stworzony cluster to możemy tworzyć zarówno pody przy pomocy kube-api, a także statyczne. Kubie-api będzie w stanie wyświetlić obraz tego poda w terminalu, ale nie będziemy mogli go edytować, ani usuwać, tak jak to się robi z innymi przy pomocy komend. Trzeba będzie modyfikować plik yaml. Do nazwy statycznego poda będzie dodana automatycznie nazwa node.

Zastosowanie. Jako, że Static pods są niezależne od controlplane, można ich użyć do stworzenia właśnie controlplane jako pod na nodzie. Najpierw trzeba będzie wtedy zainstalować kubelet na nodach i potem stworzyć pliki yaml dla każdego komponentu controlplane np. apiserver.yaml, etcd.yaml, controll manager.yaml itp. I umieścić je w directory do plików yaml. Dzięki temu nie trzeba ściągać binarek i konfigurować serwisów, bo kubelet się zajmie restartami, gdy coś się zepsuje. W ten sposób kubeadmin tool ustawia Kuberntes.

Różnice z DaemonSet. Daemon set jest tworzony przez kube-api i raczej używany do monitoringu i zbierania logów, a StaticPod przez kubelet i można nim stworzyć controlplane. Oba te rodzaje podów sa jednak ignorowane przez Kube-Scheduler.

Jak potrzebujemy się dostać na inny node i tam pousuwać Static Pods to najpierw trzeba się przełączyć na inny node: *ssh nazwa_node* potem sprawdzić lokalizację plików yaml: *ps aux | grep kubelet | grep config* w pliku konfiguracyjnym będzie ścieżka. Na końcu usuwamy pliki yaml statycznych podów.

Multiple Schedulers

Scheduler umieszcza nody na podach i bierze pod uwagę różne aspekty takie jak Taints and Tolarations, Node Affinity itp. Jeśli jednak chcemy stworzyć własny algorytm wybierania to też możemy to zrobić, dodatkowo możemy stworzyć własny scheduler i wtedy będzie on obsługiwał wybraną aplikację, a reszta podów będzie schedulowana defaultowym schedulerem.

Ręcznie Scheduler tworzymy poprzez pobranie binarki i uruchomienie jej jako serwis. Możemy jej też użyć, żeby stworzyć drugi własny scheduler, ale trzeba zmienić parametr scheduler-name i podać nową wartość. Domyślny scheduler ma nazwę *default-scheduler*.

```
kube-scheduler.service
ExecStart=/usr/local/bin/kube-scheduler \\
--config=/etc/kubernetes/config/kube-scheduler.yaml \\
--scheduler-name=default-scheduler
```

```
my-custom-scheduler.service
ExecStart=/usr/local/bin/kube-scheduler \\
--config=/etc/kubernetes/config/kube-scheduler.yaml \\
--scheduler-name=my-custom-scheduler
```

Teraz przy tworzeniu podów lub deploymentów mamy opcje sprecyzowania, który scheduler ma się nimi zajmować i tam ustawiamy nową nazwę.

Przy pomocy kubeadmin tool, scheduler jest deployowany jako pod. Aby stworzyć drugi własny, trzeba skopiować plik yaml schedulera z folderu *etc/kubernetes/manifests* i w nim dodać pole `--scheduler-name`. Ważnym polem jest też `--leader-elect`. Jest ono używane gdy mamy wiele kopii schedulera na kilu master nodach. W tym przypadku tylko jeden może być aktywny w danym czasie, ta opcja wybiera lidera, który będzie głównym schedulerem. Dodatkowo należy dodać opcję `--lock-object-name`, która pozwala rozróżnić customowego schedulera od defaultowego w trakcie wybierania lidera.

```
kube-scheduler.service
ExecStart=/usr/local/bin/kube-scheduler \\
--config=/etc/kubernetes/config/kube-scheduler.yaml \\
--scheduler-name=default-scheduler
```

```
my-custom-scheduler.service
ExecStart=/usr/local/bin/kube-scheduler \\
--config=/etc/kubernetes/config/kube-scheduler.yaml \\
--scheduler-name=my-custom-scheduler
```

Natomiast w naszym przypadku, gdy mamy wiele schedulerów na jednym node, to musimy tą opcję zmienić na `false`. Dodatkowo trzeba dodać nazwę schedulera `--scheduler-name`, oraz zmienić port w sekcji `command` (ten port jest używany przez `readiness` i `liveness probe`, więc jest już zajęty. W ćwiczeniu podaliśmy `10282` i potem niżej w `readiness` i `liveness probe` też trzeba go zmienić). W ćwiczeniu dodatkowo wyłączyliśmy `https` przez dodanie `--secure-port=0` i w `readiness` i `liveness probe` zmieniliśmy wartość na `http`. Po stworzeniu schedulera, pojawi się nowy pod w `kube-system` namespace.

Teraz przy tworzeniu poda musimy sprecyzować nazwę nowego schedulera.

```
pod-definition.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
  schedulerName: my-custom-scheduler
```

Jeśli scheduler nie jest dobrze skonfigurowany to pod będzie w stanie Pending.

Aby przeglądać informacje w nodzie można wyświetlić eventy:

`kubectl get events`

Lub wyświetlić logi z danego poda, w tym przypadku z poda nowego schedulera:
kubectl logs nazwa_schedulera --name-space=kube-system

Logging and Monitoring

Monitoring Cluster Components

Chcemy monitorować metryki na nodach np. ilość nodów na clustrze, ile z nich jest healthy, CPU i memory usage itp. Dodatkowo te same metryki na poziomie podów. Potrzebujemy do tego serwera, który będzie zbierał logi i je przechowywał. W Kubernetesie jest to Metrics Server (jest tylko jeden na dany cluster). Pobiera on metryki z podów i nodów, agreguje i przechowuje je w pamięci. Należy pamiętać, że jest to In-Memory solution, więc nie przechowuje metryk na dysku, przez to nie można oglądać historycznych danych, tylko aktualne z ostatniego czasu. Jak chcemy mieć wcześniejsze, to trzeba użyć np. Datadoga lub innego zewnętrznego toola. W podach działa kubelet, który zbiera i udostępnia dane do Metrics Servera. Robi to przy pomocy cAdvisora.

Aby zainstalować Metrics Server, musimy go pobrać z gita, lub użyć minikube. Po pobraniu należy też zainstalować jego komponenty.

```
git clone https://github.com/kubernetes-incubator/metrics-server.git  
  
kubectl create -f deploy/1.8+/  
  
clusterrolebinding "metrics-server:system:auth-delegator" created  
rolebinding "metrics-server-auth-reader" created  
apiservice "v1beta1.metrics.k8s.io" created  
serviceaccount "metrics-server" created  
deployment "metrics-server" created  
service "metrics-server" created  
clusterrole "system:metrics-server" created  
clusterrolebinding "system:metrics-server" created
```

Komponenty te to różne pody i serwisy, odpowiadające za pracę Metrics Servera. Po zainstalowaniu dane są już zbierane i aby je wyświetlić można użyć następujących komend:

kubectl top node - wyświetlenie zużycia CPU i memory na nodach

kubectl top pod - wyświetlenie zużycia CPU i memory na podach

Managing Application Logs

Logi w kontenerze Dockerowym wyświetlamy następującą komendą, flaga *f* pokazuje logi na żywo na ekranie: *docker logs -f nazwa_kontenera*

Analogiczna komenda może być użyta w Kubernetesie: *kubernetes logs -f nazwa_poda*. Komenda ta pokazuje logi z kontenera w podzie, jeśli jednak mamy więcej niż jeden kontener to komenda ta przestanie działać i trzeba będzie sprecyzować dodatkowo nazwę kontenera: *kubernetes logs -f nazwa_poda nazwa_kontenera*

Application Lifecycle Management

Rolling Updates and Rollbacks

Gdy tworzymy aplikację to jej pierwsze wdrożenie (Rollout) będzie jej pierwszą wersją (Revision 1). Gdy chcemy załadować jakiś update to tworzymy nowy Rollout i przez to tworzy nam się Revision 2. Dzięki temu jest nam łatwiej śledzić zmiany w aplikacji i łatwiej cofnąć się do poprzednich ustawień, jeśli to będzie potrzebne.

kubectl rollout status deployment/nazwa_deploymentu - wyświetlenie statusu rolloutu dla naszego deploymentu

kubectl rollout history deployment/nazwa_deploymentu - wyświetlenie historii rolloutów i pokazanie wszystkich revision

Są dwie strategie updatów:

-Recreate - wszystkie pody ze starą wersją aplikacji są usuwane i tworzone są nowe z nową wersją. Minusem tego jest to, że mamy downtime pomiędzy usunięciem starych, a stworzeniem nowych. Replicaset jest skalowany do 0 a potem do określonej liczby podów.

-Rolling Update - pody sa usuwane i tworzone z nową wersją jeden po drugim, tak aby nie było downtime. Jest to domyślna strategia. Gdy uruchamiamy upgrade, tworzony jest nowy analogiczny Replicaset do istniejącego i w nim tworzone są nowe pody, a ze starego są one odpowiednio usuwane. Możemy tu dodatkowo ustawić parametry mówiące o tym, ile procentowo podów może być usunięte w danym czasie.

Aby stworzyć update najlepiej jest wprowadzić odpowiednie zmiany do pliku yaml i uruchomić *kubectl apply -f plik.yaml*.

Jeśli chcemy cofnąć nasze zmiany i wrócić do poprzedniego revision naszej aplikacji, trzeba użyć komendy:

kubectl rollout undo deployment/nazwa_deploymentu. W tym przypadku nowy Replicaset będzie zniszczony, a stary będzie przywrócony i w nim będą tworzone pody.

Commands

docker run ubuntu - stworzenie kontenera z obrazu ubuntu

docker ps - wyświetlenie działających kontenerów (są one procesami)

docker ps -a - wyświetlenie wszystkich kontenerów.

Jeśli stworzymy taki prosty kontener to po chwili będzie on zastopowany, ponieważ kontenery nie są stworzone do hostowania systemu operacyjnego, ale tylko do wykonania określonego zadania (hostowanie bazy danych, części aplikacji webowej, wykonanie analizy). Po skończeniu zadania kontener jest wyłączany. Działa do momentu, gdy jest w nim uruchomiony proces. Aby w kontenerze działał jakiś proces, trzeba w obrazie dockerowym ustawić komendę, która będzie uruchamiana przy tworzeniu kontenera, np. w obrazie nginx, ta komenda to nginx tworząca ten proces. W naszym przypadku: *docker run ubuntu*. Uruchomiona komenda to *bash*, czyli uruchomienie shella i czekanie na input z terminala. Jak inputu nie ma (a domyślnie terminal nie jest dodany do basha), to komenda się kończy i kontener pada. Jeśli chcemy dodać jakąś komendę do uruchamianego kontenera to podajemy ją w komendzie tworzenia:

docker run ubuntu sleep 5

W tym przypadku komenda ta nadpisze wszystkie komendy sprecyzowane w Dockerfile i wykona się tylko ona.

Aby ta komenda była uruchamiana domyślnie przy każdym tworzeniu kontenera musimy zrobić swój własny Dockerfile i tam umieścić komendę:

FROM Ubuntu

CMD sleep 5

W obrazie komendy można podawać w postaci shell lub json array:

```
command parameter - sleep 5
["command", "parameter"] - ["sleep", "5"]
```

Teraz komendą `docker build -t nowy_image .` tworzymy nowy obraz dockerowy.

Jeśli chcielibyśmy móc sami podawać parametry do wykonywanej przy starcie komendy (np. żeby kontener nie spał zawsze 5 sekund, tylko np. 10) to zamiast polecenia `CMD` trzeba użyć polecenia `ENTRYPOINT` w Dockerfile:

```
FROM Ubuntu
ENTRYPOINT ["sleep"]
```

Teraz można użyć stworzyć obraz przy pomocy `docker run nazwa_obrazu 10`.

Jeśli chcemy natomiast podawać sami parametr, ale też mieć domyślną wartość parametru to trzeba użyć `CMD` i `ENTRYPOINT` razem.

```
FROM Ubuntu
ENTRYPOINT ["sleep"]
CMD ["5"]
```

Jest tak, dlatego, że wartość z `CMD` może być zawsze nadpisana jak podamy sami parametr.

Jeśli chcemy jednak nadpisać wartość `ENTRYPOINTU` z obrazu to trzeba dodać dodatkową flagę:

```
docker run --entrypoint nowa_komenda nazwa_obrazu parametr
```

Przykład: `docker run --entrypoint sleep2.0 ubuntu-sleeper 10`

Commands and Arguments

Jeśli teraz chcemy stworzyć poda na podstawie obrazu stworzonego w poprzednim wykładzie (`ubuntu-sleeper` śpiącego przez 5 sekund), to używamy do tego pliku yaml. Jeśli chcemy podać jakieś argumenty wejściowe do tworzonego poda, potem przejdą one do kontenera dockerowego, czyli nadpiszą domyślną wartość ustawioną wcześniej w komendzie `CMD`. Argumenty dodajemy w sekcji `spec->containers->args`. Argument podajemy w postaci macierzy. Jeśli natomiast chcemy nadpisać komendę `ENTRYPOINT`, czyli podać inne polecenie na wejście (np. `sleep2.0`), to trzeba to podać w sekcji `spec->containers->command` w postaci macierzy.

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      command: ["sleep2.0"]
      args: ["10"]
```

Inny sposób podania komendy z argumentem

```
spec:
  containers:
    - command:
      - sleep
      - "4800"
```

Environment Variables in Applications

Aby ustawić zmienne środowiskowe w Kubernetesie musimy dodać sekcję `spec->containers->env` w pliku yaml. Env to macierz, więc każdy element zaczyna się od myślnika i ma swoją nazwę i wartość.

```
containers:
- name: simple-webapp-color
  image: simple-webapp-color
  ports:
    - containerPort: 8080
  env:
    - name: APP_COLOR
      value: pink
```

To samo osiągnęlibyśmy w Dockerze uruchamiając:
docker run -r APP_COLOR=pink simple-webapp-color

Są trzy różne rodzaje zmiennych środowiskowych. Zwykłe key-value, ConfigMap i Secret. W ten sposób trzeba je definiować:

```
env:
- name: APP_COLOR
  value: pink

env:
- name: APP_COLOR
  valueFrom:
    configMapKeyRef:

env:
- name: APP_COLOR
  valueFrom:
    secretKeyRef:
```

Configuring ConfigMaps in Applications

Gdy mamy wiele plików yaml dla podów i wiele zmiennych, zarządzanie tym może być trudne. W takim przypadku dobrze jest użyć ConfigMapy. Należy podać wszystkie zmienne środowiskowe w formacie key-value do niej, a w plikach yaml podać tylko odnośnik do tego, aby kubernetes mógł wczytać tą mapę do definicji poda. Dzięki temu pliki yaml są prostsze i łatwiej jest obsługiwać zmienne.

kubectl create configmap nazwa --from-literal=APP_COLOR=blue - tworzenie ConfigMap i dodanie do niej wartości zmiennej w jednej komendzie. Potrzebna jest tu flaga --from-literal. Jeśli chcemy dodać kolejną zmienną, to trzeba jeszcze raz podać tę flagę z wartościami zmiennej.

Drugim sposobem jest podanie w komendzie ścieżki do pliku z ConfigMapą.

kubectl create configmap app-config --from-file=app_config.properties

ConfigMap może być też stworzony przy pomocy pliku yaml. W nim zamiast sekcji spec mamy sekcję data i tam podajemy wartości zmiennych. Można tworzyć tyle różnych ConfigMap, ile chcemy.

```
config-map.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_MODE: prod
```

kubectl get configmaps - wyświetlenie wszystkich configmap

Aby sprecyzować ConfigMap w definicji poda należy dodać sekcję spec->containers->envFrom->configMapRef ta sekcja to lista więc możemy podać tyle map, ile chcemy i wszystkie zmienne, w nich się znajdująco zostaną dodane do poda.

```
pod-definition.yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
      ports:
        - containerPort: 8080
      envFrom:
        - configMapRef:
            name: app-config
```

Są też inne sposoby na dodawanie ConfigMapy:

The diagram shows three horizontal sections representing different methods to add a ConfigMap named 'app-config' to a pod:

- ENV**: Shows the 'envFrom' section of the pod spec with a single entry: '- configMapRef: name: app-config'.
- SINGLE ENV**: Shows the 'env' section of the pod spec with a single environment variable: '- name: APP_COLOR valueFrom: configMapKeyRef: name: app-config key: APP_COLOR'.
- VOLUME**: Shows the 'volumes' section of the pod spec with a volume named 'app-config-volume' that has a 'configMap' named 'app-config' attached.

Configure Secrets in Applications

Secrets są używane do przechowywania wrażliwych danych takich jak hasła. Działają one na tej samej zasadzie co ConfigMap, ale są przechowywane w zakodowanej formie.

kubectl create secret generic nazwa_secretu --from-literal=DB_Host=mysql - tworzenie secretu DB_Host o wartości mysql. Flaga from-literal jest używana do precyzowania secretów w postaci key-value. Jeśli trzeba byłoby dodać kilka takich par, to trzeba flagę from-literal zastosować wiele razy.

Jeśli mamy wiele secretów do wprowadzenia to lepiej użyć do tego wprowadzania z pliku:

```
kubectl create secret generic nazwa_secretu --from-file=app_secret.properties
```

Można też stworzyć secret przy pomocy pliku yaml, ale w nim trzeba już wartości secretów zaszyfrować. Możemy to zrobić przy pomocy komendy:

`echo -n 'wartość_secretu' | base64` - jako output dostaniemy zaszyfrowaną wartość secretu. Następnie tworzymy plik yaml:

```
secret-data.yaml
```

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: bXlzcWw=
  DB_User: cm9vdA==
  DB_Password: cGFzd3Jk
```

`kubectl get secrets` - wyświetli wszystkie secrety. Aby wyświetlić wartości secretów należy dodać flagę `-o yaml`:

```
kubectl get secrets -o yaml
```

Aby odkodować zaszyfrowane wartości należy użyć komendy:

```
echo -n 'zaszyfrowana_wartość' | base64 --decode
```

Pobieranie wartości secretów do poda ustawiamy w sekcji spec-> containers->envFrom:

```
pod-definition.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
    - containerPort: 8080
    envFrom:
      - secretRef:
          name: app-secret
```

Inne sposoby na dodawanie secretów do poda to zmienna, lub plik w volume:

```
envFrom:
- secretRef:
  name: app-config
```

ENV

SINGLE ENV

```
env:
- name: DB_Password
  valueFrom:
    secretKeyRef:
      name: app-secret
      key: DB_Password
```



```
volumes:
- name: app-secret-volume
  secret:
    secretName: app-secret
```

VOLUME

Jeśli stworzymy secrety jako pliki w volume, każdy atrybut (login, hasło itp) zostanie stworzony jako oddzielny plik, a jego zawartością będzie wartość secretu:

```
▶ ls /opt/app-secret-volumes
DB_Host      DB_Password  DB_User

▶ cat /opt/app-secret-volumes/DB_Password
paswrd
```

W kwestii bezpieczeństwa secretów, każdy kto ma dostęp do base64 może je odkodować, więc aby zwiększyć bezpieczeństwo należy nie dodawać plików yaml secretów do źródłowych repozytoriów kodu, włączyć Encryption at Rest, tak aby były przechowywane zaszyfrowane w ETCD. Kubernetes sam w sobie też stara się zwiększyć bezpieczeństwo, bo secret jest tylko wysyłany do noda, gdy jest taki request, Kubelet przechowuje secrety w tmpf, więc nie jest on zapisywany na dysku, a jeśli pod potrzebujący secretu jest usunięty, to usunięta jest także lokalna kopia danych z secretu.

Multi Container PODs

Podzielenie aplikacji na kontenery, czyli mniejsze ilości kodu ułatwia nam skalowanie i modyfikacje poszczególnych części kodu. Czasami może być jednak potrzeba pracy dwóch agentów jednocześnie np. web server i log agent. Można wtedy stworzyć multi container pod, po to aby mieć te serwisy razem i będą one wtedy dzielić tę samą sieć i ten sam storage. Aby dodać drugi kontener musimy go określić w sekcji spec->containers w pliku yaml. Drugi kontener należy dodać od myślnika, bo containers to array:

```
pod-definition.yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
    - name: simple-webapp
      image: simple-webapp
      ports:
        - containerPort: 8080
    - name: log-agent
      image: log-agent
```

Są trzy główne rodzaje tworzenia multi-container podów:

- sidecar
- adapter
- ambassador

InitContainers

Jeśli mamy multi-container pod, to w nim w każdym kontenerze uruchamiany jest proces, który musi działać w sposób ciągły, aby podtrzymywać kontener przy życiu. Jeśli zostanie on zatrzymany, kontener napotka błąd i zostanie zrestartowany. Czasami chcemy jednak, aby przed uruchomieniem właściwego procesu wykonane zostały jednorazowo jakieś czynności np. pobranie kodu z repo, poczekanie aż

zewnętrzny serwis lub baza danych będą gotowe. Wtedy należy użyć initContainera. Taki kontener jest konfigurowany tak samo jak zwykły w podzie, ale jest w osobnej sekcji initContainers:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
    - name: myapp-container
      image: busybox:1.28
      command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
    - name: init-my-service
      image: busybox
      command: ['sh', '-c', 'git clone <some-repository-that-will-be-used-by-application> ; done;']
```

W trakcie tworzenia poda uruchamiany jest najpierw initContainer i proces w nim zawarty musi się zakończyć, dopiero potem właściwe kontenery będą uruchamiane. Można mieć też wiele initContainerów, wtedy będą uruchamiane one na początku tworzenia poda jeden po drugim. Jeśli któryś z nich będzie miał fail, pod zostanie zrestartowany.

Self Healing Applications

W Kubernetesie mamy samo naprawiające się aplikacje, dzięki działaniu ReplicaSetu i Replication Controllera. Replication Controller rekruje pody, gdy mają one crash i dba o to, aby działała odpowiednia ich liczba.

Dodatkowe sprawdzenie zdrowia poda może być przeprowadzone dzięki Liveness i Readiness Probe.

Cluster Maintenance

OS Upgrades

W trakcie przeprowadzania maintenance np. instalowania nowej wersji systemu operacyjnego, trzeba będzie wyłączać po kolej node'y. W zależności od tego, jak stworzyliśmy pody, aplikacja przestanie działać lub nie. Jeśli mamy repliki naszego poda na innych nodach to aplikacja będzie działać ciągle. Jeśli dany node po wyłączeniu wstanie sam szybko to Kubelet odtworzy pody, ale jeśli będzie wyłączony przez 5 minut to jego pody zostaną usunięte. Jeśli usunięte pody były częścią ReplicaSetu, to zostaną odtworzone na innym node. Czas czekania na powtórne wstanie poda to pod-eviction timeout i ustawia się go w kube-controller-managerze. Domyślna wartość to właśnie 5 minut.

`kube-controller-manager --pod-eviction-timeout=5m0s`

Jeśli natomiast node będzie down więcej niż 5 minut i wstanie potem, to będzie pusty. Wszystkie pody, które na nim były są stracone (chyba, że są w ReplicaSetie to zostaną odtworzone na innym node). Bezpieczniejszym sposobem jest zrobienie drain node. Wtedy przed wyłączeniem go wszystkie jego pody zostają usunięte i zrekonwertowane na innym node. Node stanie się też non-schedulable, czyli żaden pod na nim nie powstanie do momentu zdobycia tej restrykcji. Drain nie zadziała, jeśli mamy DaemonSet na nodzie. Trzeba dodać flagę `--ignore-daemonsets`.

`kubectl drain node_name`

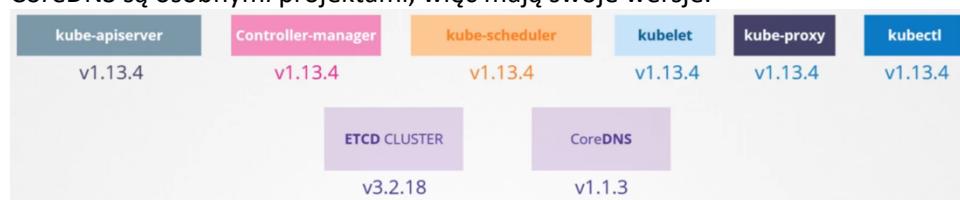
`kubectl uncordon node_name` - zdobycie restrykcji z noda, tak że mogą już na nim powstać pody.

`Kubectl cordon node_name` - ustawienie poda jako non-schedulable, ta komenda w przeciwieństwie do drain nie usunie istniejących już na nim podów.

Kubernetes Software Versions

Przy instalacji Kuberntesesa, instalujemy jego określona wersję. Możemy ją sprawdzić uruchamiając `kubectl get nodes` i zobaczyć kolumnę Version. `Kubectl version --short` pokaż nam też dokładnie wersję Clienta i Servera. Wersja release Kuberntesesa składa się z 3 części np. v1.11.3. v1 to major, 11 to minor a 3 to patch. Minor version jest wypuszczane co kilka miesięcy z nowymi funkcjonalnościami, a patche częściej żeby naprawić bugi. Przed wypuszczeniem wersji minor, najpierw testy odbywają się w fazie alpha i potem beta, zanim są dodane dla klientów.

Każdą wersję Kuberntesesa można pobrać z jego GitHub repo. Ma ona już wszystkie controlplane komponenty zainstalowane. Nie wszystkie komponenty jednak mają tą samą wersję. ETCD cluster i CoreDNS są osobnymi projektami, więc mają swoje wersje.



Cluster Upgrade Process

Główne komponenty clustra mogą mieć różne wersje, ale nic nie może mieć wersji wyższej od kube-apiservera, bo to główny komponent, z którym się każdy komunikuje. Controller-manager i kube-scheduler mogą mieć wersję o 1 niższą od kube-apiservera, a kubelet i kube-proxy o 2. Wyjątkiem jest kubectl, który może mieć wersję o 1 niższą lub wyższą niż kube-apiserver. Tylko 3 najnowsze wersje Kuberntesesa są wspierane. Upgrade robimy o jedną wersję w góre, nie można z 1.10 przejść na 1.13, tylko trzeba jedna po drugiej. Proces upgrade różni się w zależności od tego jak cluster jest zdeployowany. Jeśli mamy go w Google providerze (Azure, AWS, GCP) to najczęściej dostępny jest one-click upgrade. Jak instalowaliśmy go przy pomocy kubeadm to używamy `kubectl upgrade plan` i `kubectl upgrade apply`. Jeśli instalowaliśmy ręcznie, to ręcznie też robimy upgrade.

W przypadku kubeadm najpierw robimy upgrade master nodów. W jego trakcie wszystkie najważniejsze komponenty controlplane będą chwilowo wyłączone, więc wszystkie zadania zarządzania nie będą wtedy działać i nie będzie można zrobić żadnej zmiany na clustrze. Obecna konfiguracja i aplikacje nie będą jednak wyłączone i to co było przed startem upgrade będzie działać. Następnym krokiem jest upgrade worker nodów. Można to robić na wszystkich nodach na raz, ale wtedy wszystkie aplikacje się wyłączą i będzie potrzebny downtime. Druga strategia to upgrade nodów jeden po drugim, wtedy pody z danego noda są rekreatowane na innych nodach, aby użytkownicy mogli cały czas korzystać z aplikacji. Trzecia strategia (często używana w cloudzie, gdzie łatwo można dodać nowe nody) to dodanie nowych worker nodów do clustra i przeniesienie aplikacji na nie, następnie usunięcie starych.

`kubeadm upgrade plan` - ta komenda da nam przydatne informacje o upgrade, będzie pokazana obecna wersja, najnowsza stabilna wersja, pokazane będą także obecne wersje komponentów clustra. Jest też informacja, że po zakończeniu procesu trzeba będzie ręcznie zupgradować Kubelet. Na koniec będzie podana komenda, którą trzeba użyć do rozpoczęcia upgrade.

```
▶ kubeadm upgrade plan

[preflight] Running pre-flight checks.
[upgrade] Making sure the cluster is healthy:
[upgrade/config] Making sure the configuration is correct:
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: v1.11.8
[upgrade/versions] kubeadm version: v1.11.3
[upgrade/versions] Latest stable version: v1.13.4
[upgrade/versions] Latest version in the v1.11 series: v1.11.8

Components that must be upgraded manually after you have
upgraded the control plane with 'kubeadm upgrade apply':
COMPONENT      CURRENT      AVAILABLE
Kubelet        3 x v1.11.3   v1.13.4

Upgrade to the latest stable version:

COMPONENT      CURRENT      AVAILABLE
API Server     v1.11.8      v1.13.4
Controller Manager  v1.11.8  v1.13.4
Scheduler       v1.11.8      v1.13.4
Kube Proxy      v1.11.8      v1.13.4
CoreDNS         1.1.3        1.1.3
Etcfd          3.2.18       N/A

You can now apply the upgrade by executing the following command:

    kubeadm upgrade apply v1.13.4

Note: Before you can perform this upgrade, you have to update kubeadm to v1.13.4
```

Master node:

Na początku musimy zaktualizować kubeadm i dopiero potem cluster. Aby sprawdzić jakie wersje są dostępne:

`apt update`

`apt-cache madison kubeadm`

Potem robimy update kubeadm:

`apt-mark unhold kubeadm && \`

`apt-get update && apt-get install -y kubeadm=1.20.x-00 && \`

`apt-mark hold kubeadm`

Trzeba przenieść wszystkie aplikacje z controlplane:

`kubectl drain controlplane`

Potem uruchamiamy główny upgrade:

`kubeadm upgrade apply v1.20.x`

Po zakończeniu, jak uruchomimy `kubectl get nodes` to dalej będzie stara wersja, bo w tej komendzie jest pokazywana wersja Kubeleta, którego trzeba ręcznie zupgradować. Jeśli clustr jest zainstalowany przy pomocy kubeadm to mamy kubelet też na masterze i wszystkie komponenty jako pody. Wtedy do upgrade uruchamiamy komendę na masterze najpierw (w ten sam sposób, jeśli trzeba, robimy upgrade kubectl):

`apt-mark unhold kubelet kubectl && \`

`apt-get update && apt-get install -y kubelet=1.20.x-00 kubectl=1.20.x-00 && \`

```
apt-mark hold kubelet kubectl
```

Następnie trzeba zrestartować kubelet service:

```
systemctl daemon-reload
```

```
systemctl restart kubelet
```

Na końcu możemy odblokować master node:

```
kubectl uncordon controlplane
```

Worker node:

Najpierw trzeba przenieść aplikacje z danego noda na inny:

kubectl drain node-1 (tą komendę trzeba uruchomić na controlplane node, bo to jest komenda związana z kubectl. Wszystkie kubectl muszą być uruchomione na controlplane, a kubeadm mogą być na worker nodach)

Przechodzimy na node01:

```
ssh node01
```

Następnie tą samą komendą upgradujemy kubeadm, kubectl i Kubelet na worker nodzie:

```
apt-mark unhold kubeadm && \
```

```
apt-get update && apt-get install -y kubeadm=1.20.x-00 && \
```

```
apt-mark hold kubeadm
```

```
apt-mark unhold kubelet kubectl && \
```

```
apt-get update && apt-get install -y kubelet=1.20.x-00 kubectl=1.20.x-00 && \
```

```
apt-mark hold kubelet kubectl
```

Następnie trzeba zaktualizować konfigurację noda:

```
kubeadm upgrade node (lub z config --kubelet-version v1.20.x)
```

Na końcu robimy restart Kubeleta:

```
systemctl daemon-reload
```

```
systemctl restart kubelet
```

Trzeba też odblokować ten node, żeby pody mogły się na nim tworzyć (to już robimy z master node):

```
kubectl uncordon node-1
```

Teraz cały proces trzeba powtórzyć na wszystkich innych nodach.

Mogemy też uruchomić sobie dokumentację Kuberntesa, która pokazuje jak zupgradować się z konkretnej wersji i podążać za instrukcjami z niej.

[Upgrading kubeadm clusters | Kubernetes](#)

Komendy mogą być trochę inne w zależności od edycji Linuxa, ale logika jest ta sama, wszystko będzie w dokumentacji.

Backup and Restore methods

W Kuberntesie należy tworzyć backup konfiguracji wszystkich resourców (pliki yaml), ETCD Clustra (tu są przechowywane wszystkie informacje o clustrze i jego obiektach) oraz trwałych volume, które są dodane do aplikacji.

Konfiguracja resourców (pliki yaml) jest dużym ułatwieniem, gdy trzeba zrekonstruować cluster, bo wystarczy uruchomić po kolei wszystkie pliki. Dobrze jest przechowywać ich kopie w repozytorium kodu np. GitHub. Jeśli ktoś jednak tworzył resourcy drogą imperative (bez plików yaml) to można dostać konfigurację resourców poprzez zapytanie kube-apiservera. Tam możemy znaleźć też konfigurację wszystkich resourców. Jeśli chcemy zapisać konfigurację wszystkiego:

```
kubectl get all --all-namespaces -o yaml > all-deploy.services.yaml
```

Mogemy też zbackupować cały ETCD cluster, bo w nim są przechowywane informacje o resourcach. ETCD jest konfigurowane na master node. W jego konfiguracji parametr *--data-dir* mówi nam gdzie

dane są przechowywane. Można stworzyć backup tego directory przy pomocy toola do backupów. Dodatkowo mamy wbudowaną możliwość robienia snapshotów ETCD.

etcdctl snapshot save nazwa - robienie snapshota w obecnym directory.

etcdctl snapshot status nazwa - sprawdzenie statusu danego snapshota

Przy robieniu snapshota trzeba też podać odpowiednie flagi do połączenia się z ETCD serverem, jeśli jest włączony TLS (poniżej).

Aby zrestorować taki snapshot trzeba najpierw zastopować kube-apiserver (trzeba będzie zrestartować etcd, a server na nim polega):

service kube-apiserver stop

etcdctl snapshot restore nazwa --data-dir ścieżka - restorowanie danego snapshota etcd. Jedyną wymaganą flagą jest data dir, bo restorujemy do nowego miejsca, ale na tym samym serwerze. Przy restorze tworzona jest nowa konfiguracja clustra i członkowie etcd są konfigurowani jako nowi członkowie clustra. Dzięki temu nowy członek nie dołączy przypadkiem do starego clustra. Podana w komendzie ścieżka do --data-dir to nowa lokalizacja informacji o clustrze zapisywanych przez etcd. Musimy teraz skonfigurować nowy etcd i podać mu paramter --data-dir=ścieżka:

W pliku */etc/kubernetes/manifests/etcd.yaml* zmieniamy *hostPath* dla volume o nazwie *etcd-data* ze starej wartości na nową z komendy. Dzięki tej zmianie */var/lib/etcd* w kontenerze pointuje do nowej ścieżki etcd. Po tej zmianie pod ETCD jest automatycznie zrekreowany, bo jest to statyczny pod z lokalizacji */etc/kubernetes/manifests*, spowoduje to też restart kube-controller-managera i kube-schedulera. Jeśli pod ETCD będzie miał problemy ze wstaniem automatycznie to trzeba go jeszcze raz usunąć tym razem ręcznie.

Na końcu trzeba przeładować service daemon, zrestartować etcd oraz na koniec uruchomić kube-api server:

systemctl daemon-reload

service etcd restart

service kube-apiserver start

Jeśli baza danych ETCD ma włączony TLS, to do wszystkich komend z etcd trzeba ustawić cacert (weryfikacja certyfikatu TLS przy pomocy CA bundle), cert (secure client certificate), endpoint (domyślny to 127.0.0.1:2379) i key (plik z TLS key).

```
--endpoints=https://127.0.0.1:2379 \
--cacert=/etc/etcd/ca.crt \
--cert=/etc/etcd/etcd-server.crt \
--key=/etc/etcd/etcd-server.key
```

Etdctl to command line dla etcd. Aby używać go do tworzenia backupów trzeba ustawić zmienną odpowiadającą wersji etcd (np. *export ETCDCTL_API=3*). Można to sprawdzić komendą *etcdctl version* na master node. U nas wartość zmiennej to *ETCDCTL_API=3*, bo wersja to v3. Inna opcja na sprawdzenie wersji ETCD to uruchomienie logów na jego podzie:

kubectl logs etcd-controlplane -n kube-system

Lub sprawdzenie image, jaki ma ETCD pod:

kubectl describe pod etcd-controlplane -n kube-system

W ETCD clustrze parametr *--listen-client-urls* określa adres IP, po którym można się łączyć do ETCD servera, parametr *--listen-peer-urls* podaje adres IP dla innych nodów, aby dołączyły do clustra (dla worker nodów, bo etcd jest tylko na masterze).

Security

Kubernetes Security Primitives

Domyślnie dostęp do hostów (nodów) w Kubernetesie jest ograniczony. Root jest zablokowany, tak samo jak autentykacja hasłem. Można przez SSH Key się za to autentykować.

Pierwsza linia bezpieczeństwa to kontrolowanie dostępu do kube-apiservera, ponieważ on jest centrum zarządzania i dowodzenia w clustrze. Musimy tutaj zająć się dwoma ważnymi tematami: autentykacją (która ma dostęp) i autoryzacją (co może robić). Autentykować się można przez credentiale, tokeny, certyfikaty, LDAP, czy Service Account dla maszyn. Za autoryzację odpowiada RBAC (nadawanie określonym grupom określonych privilege), Webhook Mode, Node Authorization, czy ABAC Authorization.

Połączenia sieciowe pomiędzy wszystkimi komponentami w Kubernetesie są zabezpieczone przy pomocy TLS encryption. Należy tutaj ustawić odpowiednie certyfikaty, aby komponenty mogły ze sobą rozmawiać. Domyślnie w clustrze wszystkie pody mogą ze sobą rozmawiać, chyba że ograniczymy ten ruch przy pomocy Network Policies.

Authentication

Bezpieczeństwo aplikacji względem end-userów jest zarządzane przez same aplikacje (ktoś kto ją tworzył, stworzył też security do niej).

W przypadku użytkowników z dostępem administratora lub developera, Kubernetes nie zarządza tożsamością samemu, ale polega na credentialach zapisanych w pliku, tokenach, certyfikatach, czy LDAP.

W przypadku aplikacji łączących się do innych aplikacji mamy Service Account. One pozwalają im się zautentykować.

`kubectl create serviceaccount nazwa` - tworzenie nowego Service Account

`kubectl get serviceaccount` - wyświetlenie Service Account

Dostęp dla użytkowników jest zarządzany przez kube-apiserver, bez względu na to czy łączymy się przez kubectl czy API (np. curl <https://kube-server-ip:6443/>). Możliwości do autentykacji użytkownika:

- Statyczny plik z hasłem

Trzeba stworzyć plik .csv z trzema kolumnami: hasło, username i userID (może być też czwarta kolumna z nazwą grupy). Następnie skonfigurować ten plik w kube-apiserverze flagą --basic-auth-file (przy ręcznej instalacji Kuberntesa). Po konfiguracji trzeba zrestartować kube-apiserver. Jeśli instalowaliśmy Kubernetes przy pomocy kubeadm tool, to trzeba w pliku yaml dodać komendę --basic-auth-file.

```
/etc/kubernetes/manifests/kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
    - command:
        - kube-apiserver
        - --authorization-mode=Node,RBAC
        - --advertise-address=172.17.0.107
        - --allow-privileged=true
        - --enable-admission-plugins=NodeRestriction
        - --enable-bootstrap-token-auth=true
        - --basic-auth-file=user-details.csv
      image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
      name: kube-apiserver
```

Jeśli chcemy się autentykować w API, to trzeba sprecyzować username i hasło w curlu.

```
curl -v -k https://master-node-ip:6443/api/v1/pods -u "user1:password123"
```

-Statyczny plik z tokenem

Też tworzymy plik .csv, ale mamy w nim kolumny: token, username, userID i grupa. Należy też ten plik skonfigurować analogicznie, ale przy użyciu flagi --token-auth-file. Jeśli chcemy się autentykować przy pomocy API to trzeba token dodać do curla.

```
curl -v -k https://master-node-ip:6443/api/v1/pods --header "Authorization: Bearer KpjCVbI7rCFAHYPkBzRb7gu1cUc4B"
```

-Certyfikat

-3rd party application (LDAP, czy Kerberos)

Autentykacja przy pomocy hasła lub tokenu nie jest bezpieczna i w nowszych wersjach Kuberntesa nie jest już wspierana.

TLS Basics

Certyfikaty zapewniają, że połączenie pomiędzy dwoma hostami jest bezpieczne, zaszyfrowane (jeśli nie ma enkrypcji, to łatwo z hakować credentiali) i że host do którego się łączymy, jest tym, za kogo się podaje. Enkrypcja odbywa się przy pomocy encryption key, który jest ciągiem losowych znaków i zmienia odpowiednio wartość credentiali, tak że nie da się ich odczytać. Kopią klucza jest też wysyłana do celu połączenia, aby mógł odszyfrować hasło. Jeśli ten sam klucz jest użyty do enkrypcji i dekrypcji i wysyłany jest tą samą siecią (symetryczna enkrypcja) to łatwo go przechwycić i odszyfrować. Jest także bezpieczniejsza asymetryczna enkrypcja. Mamy tutaj parę Private Key i Public Key. Public Key robi enkrypcję i może być wysyłany do sieci, aby odszyfrować dane trzeba użyć jednak Private Key, który nie jest nigdzie udostępniany.

Przykład: Łączenie się po SSH przy pomocy kluczy.

`ssh-keygen` - generuje Public Key i Private Key do dostępu SSH. Dostęp do servera jest całkowicie zablokowany oprócz odszyfrowania go przez nasz klucz. Klucz ustawiamy w pliku `~/.ssh/authorized_keys`.

```
cat ~/.ssh/authorized_keys
ssh-rsa AAAAB3NzaC1yc...KhtUBfoTzlBqR
V1NThvOo4opzEwRQo1mWx user1
```

Przy logowaniu podajemy lokalizację klucza SSH w SSH Map.

```
ssh -i id_ras user1@server1
Successfully Logged In!
```

Możemy w ten sposób szyfrować wiele serwerów przy pomocy tej samej pary kluczy. Jeśli drugi użytkownik będzie chciał mieć dostęp, to musi wygenerować swoje klucze i Public Key musi być dodany do pliku `authorized_keys` na serwerach.

Przykład: Zwiększanie bezpieczeństwa symetrycznej enkrypcji.

Można używać tej enkrypcji w bezpieczniejszy sposób, poprzez dodatkowe użycie asymetrycznej enkrypcji do bezpiecznej przesyłania klucza do endpointu. Generujemy Public Key i Private Key (oba są w stanie zaszyfrować coś, ale tylko ten drugi może służyć do odszyfrowania, nie można odszyfrować tym kluczem, którym się zaszyfrowało. Dlatego lepiej szyfrować przy pomocy Public Key, bo tylko Private może odszyfrować) na serwerze przy pomocy komend:

```
openssl genrsa -out my-bank.key 1024
```

```
openssl rsa -in my-bank.key -pubout > mybank.pem
```

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCB
iQKBgQDkwLgQAgAN1HpEoLUaqKYiYJI
k9wetzotW2/w4nsGhonuWGrTd7+823xd
8FDH+WJLqXsTDkrpKNG3sh67dHRGGipK
cEXfZnzT5yDyK/jA6uQvAzl+I4xNNqtw
KDC03uoLpnMEsayPhNtexosfScu1KXe0
L6/nTkn9Gc/YoUWzgQIDAQAB
-----END PUBLIC KEY-----
```

Gdy użytkownik loguje się do strony internetowej po https dostaje Public Key od endpointa. Ma on ze sobą symetryczny klucz i następuje enkrypcja symetrycznego klucza przy pomocy Public Key od endpointa. Symetryczny klucz jest bezpieczny i w ten sposób zaszyfrowany jest wysyłany do endpointa. Endpoint używa Private Key do odszyfrowania Public Key i dzięki temu dostaje symetryczny klucz do odszyfrowania haseł. Haker nie jest w stanie tu nic przechwycić.

Przykład: Używamy https, ale haker odtworzył stronę banku i w jakiś sposób sprawił, że nasze requesty idą do jego serwera, a nie do naszego banku.

Jest dodatkowa opcja, która pozwala sprawdzić, czy Public Key który otrzymujemy od endpointa to prawdziwy klucz z naszego banku, czy nie. Wysyłany jest certyfikat, który ma Public Key w sobie. W certyfikacie znajdziemy informacje o tym, dla kogo został wystawiony (dzięki temu zobaczymy, czy nie dla hakera), Public Key, lokalizację serwera itp.

```
Certificate:
Data:
    Serial Number: 420327018966204255
Signature Algorithm: sha256WithRSAEncryption
Issuer: CN=kubernetes
Validity
    Not After : Feb  9 13:41:28 2020 GMT
    Subject: CN=my-bank.com
X509v3 Subject Alternative Name:
    DNS:mybank.com, DNS:i-bank.com,
    DNS:we-bank.com,
Subject Public Key Info:
    00:b9:b0:55:24:fb:a4:ef:77:73:
    7c:9b
```

Jeśli łączymy się do jakiegoś serwera w przeglądarce, to certyfikat musi odpowiadać temu co wpiszemy w URLu. Jeśli dany endpoint pozwala na łączenie się z nimi przy pomocy innych URL, to musi je sprecyzować w sekcji Alternative Name w certyfikacie. Oczywiście każdy może wygenerować taki certyfikat, nawet haker, ale najważniejszym elementem certyfikatu jest podpis. Jeśli certyfikat jest podpisany przez osobę, która go wystawia, to może to nie być bezpieczne, bo wygląda jak ruch hakerski. Najczęściej certyfikaty są walidowane przez przeglądarki, które mają specjalne mechanizmy, jeśli zobaczą podejrzany certyfikat, to o tym powiedzą. Aby stworzyć legitny certyfikat, który jest podpisany przez kogoś zaufanego należy użyć CA (Certificate Authority - np. Symantec, Digicert, Comodo, GlobalSign). Składany jest tam Certificate Signing Request (CSR) przy pomocy wygenerowanego wcześniej klucza i nazwy domeny serwera:

```
openssl req -new -key my-bank.key -out my-bank.csr -subj "/C=US/ST=CA/O=MyOrg, Inc./CN=mydomain.com" - tutaj my-bank.csr to ten CSR, który powinien być wysyłany do CA do podpisania. Tak on wygląda:
```

```
-----BEGIN CERTIFICATE REQUEST-----
MIICjDCAXQCAQAwRzELMAkGA1UEBhMCVWmxCzAJBgNVBAgMAMBRQwEgYDVQQK
DATNeU9yZywgSW5JlEVMBMGa1UEAwMbX1kb21haW4uY29tMTIBIjANBgkqhkJG
9w0BAQEFAAOCQAQ8AMIIBCgKCAQEAp8xohAksHxvj+/-pRKCC2sqx7021nuD49Kp4
WDOnDbwEeXNviY-SuQjptxmuVr/orIpU7Mhk/fkbIICLT4jrX-Bq4MwfFcwl1
n8T0S9A7aLfwKL4rxJGF1U9DAdzr-qGLHXF1C8obLpUWjkTerHpwG++/k2UDkuPJE
V0mQ16Fe/3jWGaMN1nkY/eNyYn+a27NFmd1wQUzs9t5uFPpZbwG81mjhDvVIobA8
yHNFRDNt6gKqvZtv+vGTaM0Lfg jedGne2uQ7/Bbq22rSsXgfLM9uHmSpNT57Tjs9
QOb14FZoOnphhSqLe1V/cGAjFCzrIx5887Hxzduw+eRTQIDAQABoAwDQVJ
KoZIhvcNAQELBQADggEBABtY/tvjpFp4u1UtcI2f13TFbtYziwAyob7U2sNrjzn
uEe4K2+fosUl1CjXk7EUT4sgGjVtaoqJqrFihwQ1SLCViRgTwkLBdvtvgViWnNQ
mDJeP5YY92xtAKZt52wsj8MeUwTUjn6eDuz5Nhp0KuiWMf9loxFYrgAgi2xlo
Fkse6Zr6zaB/cNm6daW8m6qVs9hKpud1iagD3g4MeuLLPK7VnxFTMoS1fkLUui
O1F8dq2CW/ByrYMHUmONCAkKaag1FwY2Vm551HY6srwnCPhszBcr17M5Bzf70E
rgK3Pf06cAhF17WpeuUz/0e4U12r6YF+Hhk7IDKnLeI=
-----END CERTIFICATE REQUEST-----
```

CA sprawdza szczegóły, podpisuje i odsyła certyfikat. Haker nie mógłby tego zrobić, bo nie udałaby się walidacja po stronie CA. Aby przeglądarka wiedziała, że ten certyfikat jest podpisany przez dany CA i że ten CA jest prawdziwy, a nie fejkowy, jest dodatkowa walidacja w przeglądarce. Wszystkie CA mają zestaw Private i Public Keys. Certyfikaty są podpisane przez Private Key danego CA, natomiast wszystkie Public Key, wszystkich CA są wbudowane w przeglądarkę, po to, aby mogła ona właśnie sprawdzić, czy to jest prawdziwy CA.

CA pomagają nam walidować strony publiczne, ale nie zrobią tego w przypadku prywatnych stron wewnętrz organizacji np. wewnętrzny email. Do takich przypadków można mieć własne wewnętrzne

CA. Większość firm, które udostępniają publiczne CA oferują też prywatne CA, które można zdeployować u siebie na serwerze. Dzięki temu można mieć Public Key swojego wewnętrznego CA zainstalowane w przeglądarce w pracy.

Podsumowanie:

Administrator używa Public Key do zaszyfrowania dostępu SSH do Servera (tylko użytkownik z odpowiednim Private Key ma dostęp). Serwer używa pary kluczy do zaszyfrowania ruchu https, ale najpierw wysyła CSR do CA o walidację certyfikatu. CA podpisuje certyfikat swoim Private Key, a Public Key jest wbudowany w przeglądarki. Podpisany certyfikat jest odesłany do serwera i tam skonfigurowany. Gdy użytkownik chce się dostać do aplikacji webowej. Serwer wysyła certyfikat, przeglądarka użytkownika odszyfrowuje go przy pomocy Public Key od CA i sprawdza, czy na pewnołączymy się do odpowiedniej żądanej strony. Odnajduje też dzięki temu Public Key serwera. Następnie generowany jest symetryczny klucz, który będzie użyty do komunikacji. Klucz ten staje się zaszyfrowany przy pomocy Public Key serwera i wysłany w ten sposób do serwera. Tam jest odszyfrowany i następuje rozpoczęcie połączenia. Wtedy można bezpiecznie używać hasła.

Przykład: Haker ma credentiale usera i podaje się za niego. Co serwer może zrobić, aby sprawdzić, czy jest to naprawdę ten dobry użytkownik? Serwer może poprosić o certyfikat od usera, a więc musi on wygenerować parę kluczy i podpisany certyfikat przez CA. Ten certyfikat jest wysyłany do serwera, aby sprawdzić jego tożsamość.

Wszystkie te procesy dzieją się automatycznie i są znane jako Public Key Infrastructure (PKI).

Najczęściej certyfikaty z Public Key są w rozszerzeniu .crt lub .pem.

Certificate (Public Key)

*.crt * .pem

server.crt
server.pem
client.crt
client.pem

Private Key mają najczęściej rozszerzenia .key lub -key.pem

Private Key

*.key *-key.pem

server.key
server-key.pem
client.key
client-key.pem

TLS in Kubernetes

Certificate z Public Key i Private Key serwera będziemy nazywać Server Certificates. Certyfikat podpisany przez CA będziemy nazywać Root Certificate. Certyfikat wymagany przez server od usera do autentykacji to Client Certificate.

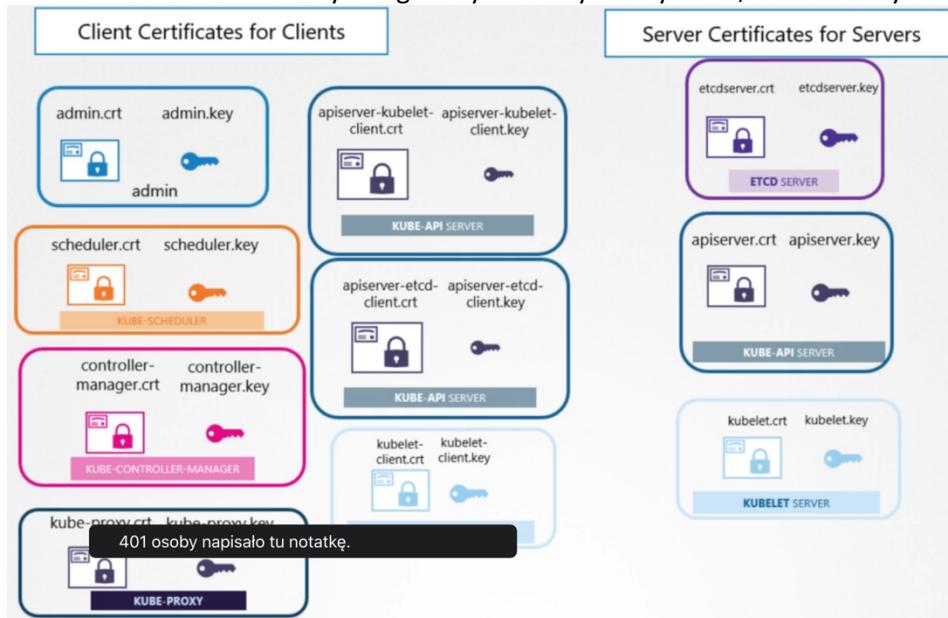
W Kubernetesie komunikacja pomiędzy nodami oraz z użytkownikami musi być zaszyfrowana. Np. administrator łączący się do kubectl lub przez API musi ustanowić bezpieczne połączenie przez TLS. Komunikacja wewnętrz clustra pomiędzy komponentami też musi być bezpieczna. Podsumowując, wszystko na serwerze musi używać Server Certificate, a userzy Client Certificate, aby potwierdzić kim są.

Server Components w clustrze:

- Kube-apiserver wystawia https service, który jest używany przez inne komponenty i użytkowników do zarządzania clustrem, dlatego jest tu wygenerowany certyfikat i para kluczy. Nazywają się apiserver.crt i apiserver.key.
- ETCD cluster też potrzebuje certyfikatu i pary kluczy, bo przechowuje on wszystkie informacje o clustrze. Obiekty te to etcdserver.crt i etcdserver.key
- Kubelet na worker nodach też wystawia https endpoint, z którym rozmawia kube-apiserver, aby zarządzać worker nodami, dlatego tu też mamy certyfikat i parę kluczy. Obiekty te to kubelet.crt i kubelet.key.

Client Componets w clustrze:

- użytkownicy, administratorzy, którzy łączą się do clustra przez kubectl lub rest api, też potrzebują certyfikatu i pary kluczy. Obiekty te to admin.crt i admin.key.
- kube-scheduler rozmawia z kube-api serverem, aby móc zaschedulować pody. Jego obiekty to scheduler.crt i scheduler.key
- kube-controller-manager to kolejny klient łączący się z kube-apiserverem. Ma on controller-manager.crt i controller-manager.key
- kube-proxy potrzebuje pary kluczy i certyfikatu żeby autentykować się do kube-apiservera. Te obiekty to kube-proxy.crt i kube-proxy.key
- kube-apiserver czasami też łączy się do ETCD servera i wtedy jest dla niego klientem. Może on jednak używać tych samych kluczy i certyfikatu, jak w przypadku, gdy inni łączą się do niego. Jeśli chcemy możemy też stworzyć osobny certyfikat i klucze do tego. Identyczna sytuacja jest z połączeniem kube-apiservera do kubeleta na worker nodach. Połączenie to występuje, aby zbierać informacje o podach na nodzie. Tu też można używać głównych kluczy i certyfikatu, albo stworzyć nowe osobne.



Kubernetes wymaga posiadania przynajmniej jednego CA do podpisania i walidacji tych certyfikatów. Można mieć oczywiście 2, wtedy jeden najczęściej jest wyłączny dla ETCD. Jak wiemy, CA ma swój zestaw certyfikatu i parę kluczy. Obiekty te nazwiemy ca.crt i ca.key.

TLS in Kubernetes - Certificate Creation

Do stworzenia certyfikatów można użyć różnych tooli np. EasyRSS, OpenSSL, CFSSL itp. My użyjemy OpenSSL.

Najpierw tworzymy klucze dla naszego CA:

```
openssl genrsa -out ca.key 2048
```

Następnie tworzymy Certificate Signing Request:

`openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr` - CSR to już gotowy certyfikat, ale jeszcze bez podpisu. W sekcji CA precyzujemy dla kogo jest ten certyfikat.

Ostatni krok to podpisanie certyfikatu, w którym musimy sprecyzować stworzony wcześniej CSR:

`openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt` - używamy tutaj ca.key żeby podpisać certyfikat dla CA, ale jest to specjalnie, bo CA jako zaufane źródło może podpisywać swoje certyfikaty. W późniejszych CSR też będziemy używać ca.key do podpisów.

Tworzenie certyfikatów dla klientów:

Admin user:

`openssl genrsa -out admin.key 2048`

`openssl req -new -key admin.key -subj "/CN=kube-admin"` -out admin.csr - nazwę użytkownika musimy podać taką, z jaką kubectl będzie się autentykował, możemy ją zobaczyć w logach.

`openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -out admin.crt` - tym razem precyzujemy certyfikat CA i CA key, bo nim podpiszemy certyfikat. To sprawi, walidację certyfikatu w clustrze. Będzie on zapisany w admin.crt. Proces ten jest identyczny do tworzenia nowego użytkownika (można porównać, że certyfikat to zwalidowane userID, a klucz to jak hasło).

Aby odróżnić innych użytkowników od admina, trzeba będzie sprecyzować grupę użytkownika w CSR.

```
▶ openssl req -new -key admin.key -subj \
  "/CN=kube-admin/O=system:masters" -out admin.csr
```

Tutaj np. mamy admina z dodaniem grupy nadającej admin privileges.

W ten sam sposób generujemy certyfikaty dla innych komponentów tylko z innymi nazwami:

Scheduler: system:kube-scheduler

Controller-Manager: system:kube-controller-manager

Kube Proxy: kube-proxy

Tak wygenerowany certyfikat możemy użyć do autentykacji w rest api:

```
curl https://kube-apiserver:6443/api/v1/pods \
  --key admin.key --cert admin.crt
  --cacert ca.crt
```

Inną możliwością jest umieszczenie wszystkich tych parametrów w pliku kube-config.yaml (tu precyzujemy endpoint i certyfikaty do użycia).

```
kube-config.yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority: ca.crt
    server: https://kube-apiserver:6443
    name: kubernetes
kind: Config
users:
- name: kubernetes-admin
  user:
    client-certificate: admin.crt
    client-key: admin.key
```

Należy pamiętać, że przy konfiguracji jakiegokolwiek komponentu potrzebna też będzie kopia ca.crt, czyli root certificate od CA, do walidacji całego procesu.

Tworzenie certyfikatów dla Serwerów:

ETCD cluster: Jeśli ETCD jest zdeployowany na kilku serwerach to trzeba tworzyć osobne certyfikaty i pary kluczy dla każdego z serwerów. Wszystkie je trzeba też skonfigurować przy starcie serwera ETCD w pliku yaml. Peer to te dodatkowe.

```
▶ cat etcd.yaml
- etcd
  - --advertise-client-urls=https://127.0.0.1:2379
  - --key-file=/path-to-certs/etcdserver.key
  - --cert-file=/path-to-certs/etcdserver.crt
  - --client-cert-auth=true
  - --data-dir=/var/lib/etcd
  - --initial-advertise-peer-urls=https://127.0.0.1:2380
  - --initial-cluster=master=https://127.0.0.1:2380
  - --listen-client-urls=https://127.0.0.1:2379
  - --listen-peer-urls=https://127.0.0.1:2380
  - --name=master
  - --peer-cert-file=/path-to-certs/etcdpeer1.crt
  - --peer-client-cert-auth=true
  - --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
  - --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
  - --snapshot-count=10000
  - --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

Proces tworzenia certyfikatów jest taki sam, jak dla klientów, tylko inna nazwa.

Dla ETCD nazwa to: etcd-server

Kube-apiserver: nazwa to kube-apiserver

Jako, że jest to najpopularniejszy komponent w clustrze, jego nazwa ma też kilka aliasów: kubernetes, kubernetes.default, kubernetes.default.svc, kubernetes.default.svc.cluster.local. Czasami jest też wołany po adresie IP (adresie IP hosta, na którym jest kube-apiserver lub adresie IP poda, na którym jest kube-apiserver): np. 10.96.0.1, 172.17.0.87. Te wszystkie nazwy muszą być podane w certyfikacie. Tworzymy certyfikaty tak jak poprzednio:

`openssl genrsa -out apiserver.key 2048`

`openssl req -new -key apiserver.key -subj "/CN=kube-apiserver" -out apiserver.csr -config openssl.cnf` - Aby dodatkowo sprecyzować wszystkie nazwy musimy stworzyć openssl config file, wpisać tam wszystkie nazwy i dać go jako parametr w CSR:

```
openssl.cnf
[req]
req_extensions = v3_req
[ v3_req ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation,
subjectAltName = @alt_names
[alt_names]
DNS.1 = kubernetes
DNS.2 = kubernetes.default
DNS.3 = kubernetes.default.svc
DNS.4 = kubernetes.default.svc.cluster.local
IP.1 = 10.96.0.1
IP.2 = 172.17.0.87
```

Na końcu podpisujemy certyfikat:

`openssl x509 -req -in apiserver.csr -CA ca.crt -CAkey ca.key -out apiserver.crt`

Należy też sprecyzować miejsce tych certyfikatów, bo jak pamiętamy z poprzedniej lekcji, te certyfikaty będą też użyte jak kube-apiserver będzie klientem u innych serwerów (jeśli mamy do tego inne certyfikaty, to też trzeba je tam sprecyzować). Umieszczamy to w pliku executable, który jest konfiguracją kube-apiservera.

```

ExecStart=/usr/local/bin/kube-apiserver \\
--advertise-address=${INTERNAL_IP} \\
--allow-privileged=true \\
--apiserver-count=3 \\
--authorization-mode=Node,RBAC \\
--bind-address=0.0.0.0 \\
--enable-swagger-ui=true \\
--etcd-cafile=/var/lib/kubernetes/ca.pem \\
--etcd-certfile=/var/lib/kubernetes/apiserver-etcd-client.crt \\
--etcd-keyfile=/var/lib/kubernetes/apiserver-etcd-client.key \\
--etcd-servers=https://127.0.0.1:2379 \\
--event-ttl=1h \\
--kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
--kubelet-client-certificate=/var/lib/kubernetes/apiserver-kubelet-client.crt \\
--kubelet-client-key=/var/lib/kubernetes/apiserver-kubelet-client.key \\
--kubelet-https=true \\
--runtime-config=api/all \\
--service-account-key-file=/var/lib/kubernetes/service-account.pem \\
--service-cluster-ip-range=10.32.0.0/24 \\
--service-node-port-range=30000-32767 \\
--client-ca-file=/var/lib/kubernetes/ca.pem \\
--tls-cert-file=/var/lib/kubernetes/apiserver.crt \\
--tls-private-key-file=/var/lib/kubernetes/apiserver.key \\
--v=2

```

Kubelet server:

Jest to https na każdym node. Tutaj też potrzebujemy certyfikatu i pary kluczy, dla wszystkich nodów w clustrze. Nazwy certyfikatów będą nazwane tak samo jak nazwy nodów np. node01, node02, node03. Jak już będą stworzone to trzeba je ustawić w pliku kubelet-config.yaml. Trzeba to zrobić na każdym z nodów.

```

kubernetes-config.yaml (node01)
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  x509:
    clientCAFile: "/var/lib/kubernetes/ca.pem"
authorization:
  mode: Webhook
clusterDomain: "cluster.local"
clusterDNS:
  - "10.32.0.10"
podCIDR: "${POD_CIDR}"
resolvConf: "/run/systemd/resolve/resolv.conf"
runtimeRequestTimeout: "15m"
tlsCertFile: "/var/lib/kubelet/kubelet-node01.crt"
tlsPrivateKeyFile: "/var/lib/kubelet/kubelet-
node01.key"

```

Musimy tu też stworzyć zestaw Client Certificates do łączenia się z kube-apiserverem. Tutaj nazwy certyfikatów też będą połączone z nazwami nodów, ale w inny sposób: system:node:node01, system:node:node02, system:node:node03.

Dodatkowo, każdy z nodów musi mieć odpowiednie przywileje, więc należy je dodać do grupy system:nodes. Ostatecznie CN będzie miał postać:

CN=system:node:node-1/O=system:nodes

Po stworzeniu, te certyfikaty muszą być ustawione w kube-config.files tak samo jak te serwerowe.

View Certificate Details

Jeśli chcemy zrobić health check wszystkich certyfikatów w Kubernetesie, to w zależności od tego, jak cluster był stworzony są różne podejścia. Jeśli cluster był tworzony ręcznie, to wszystkie certyfikaty są stworzone ręcznie (tak jak w poprzedniej lekcji) i musimy zobaczyć w opcjach, komponentu, gdzie są ustawione:

```
▶ cat /etc/systemd/system/kube-apiserver.service
[Service]
ExecStart=/usr/local/bin/kube-apiserver \
--advertise-address=172.17.0.32 \
--allow-privileged=true \
--apiserver-count=3 \
--authorization-mode=Node,RBAC \
--bind-address=0.0.0.0 \
--client-ca-file=/var/lib/kubernetes/ca.pem \
--enable-swagger-ui=true \
--etcd-cafile=/var/lib/kubernetes/ca.pem \
--etcd-certfile=/var/lib/kubernetes/kubernetes.pem \
--etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \
--event-ttl=1h \
--kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \
--kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \
--kubelet-https=true \
--service-node-port-range=30000-32767 \
--tls-cert-file=/var/lib/kubernetes/kubernetes.pem \
--tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem
--v=2
```

Jeśli używamy kubeadm to certyfikaty są tworzone automatycznie i informacje o nich możemy znaleźć edytując pody komponentów:

```
▶ cat /etc/kubernetes/manifests/kube-apiserver.yaml
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.32
    - --allow-privileged=true
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --disable-admission-plugins=PersistentVolumeLabel
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --insecure-port=0
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,HostId
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
    - --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key
    - --requestheader-allowed-names=front-proxy-client
```

Żeby przeprowadzić health check, dobrze jest najpierw mieć wszystkie ścieżki do certyfikatów. Dobrze jest stworzyć sobie excela i sprawdzić w nim ścieżki certyfikatów, CN name, ALT names, Organization, Issuera i Expiration Date.

Ścieżki certyfikatów mamy w pliku yaml komponentów np. kube-apiservera.

```
▶ cat /etc/kubernetes/manifests/kube-apiserver.yaml
spec:
  containers:
    - command:
      - kube-apiserver
      - --authorization-mode=Node,RBAC
      - --advertise-address=172.17.0.32
      - --allow-privileged=true
      - --client-ca-file=/etc/kubernetes/pki/ca.crt
      - --disable-admission-plugins=PersistentVolumeLabel
      - --enable-admission-plugins=NodeRestriction
      - --enable-bootstrap-token-auth=true
      - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
      - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
      - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
      - --etcd-servers=https://127.0.0.1:2379
      - --insecure-port=0
      - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
      - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
      - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
      - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
      - --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key
      - --secure-port=6443
      - --service-account-key-file=/etc/kubernetes/pki/sa.pub
      - --service-cluster-ip-range=10.96.0.0/12
      - --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
      - --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
```

Jeśli chcemy obejrzeć szczegóły danego certyfikatu to musimy go podać jako input w komendzie:
`openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout`

```
▶ openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout
Certificate:
Data:
  Version: 3 (0x2)
  Serial Number: 3147495682089747350 (0x2bae26a58f090396)
  Signature Algorithm: sha256WithRSAEncryption
  Issuer: CN=kubernetes
  Validity
    Not Before: Feb 11 05:39:19 2019 GMT
    Not After : Feb 11 05:39:20 2020 GMT
  Subject: CN=kube-apiserver
  Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
      Public-Key: (2048 bit)
        Modulus:
          00:d9:69:38:80:68:b7:2e:9e:25:00:e8:fd:01:
        Exponent: 65537 (0x10001)
  X509v3 extensions:
    X509v3 Key Usage: critical
      Digital Signature, Key Encipherment
    X509v3 Extended Key Usage:
      TLS Web Server Authentication
    X509v3 Subject Alternative Name:
      DNS:master, DNS:kubernetes, DNS:kubernetes.default,
      DNS:kubernetes.default.svc, DNS:kubernetes.default.svc.cluster.local, IP
      Address:10.96.0.1, IP Address:172.17.0.27
```

Możemy tu zobaczyć CN name, ALT names, w sekcji Validity mamy expiration date i Issuera (to powinno być CA, który wystawiło certyfikat). Wszystkie wymagania dotyczące certyfikatów znajdziemy w dokumentacji Kuberntes-a.

Jeśli napotkamy na jakieś błędy dobrze jest przejrzeć logi. Jeśli tworzyliśmy cluster sami ręcznie to dobrze jest przejrzeć systemowe logi danego serwisu, który nie działa np.: `journalctl -u etcd.service -1`

```
▶ journalctl -u etcd.service -l
2019-02-13 02:53:28.144631 I | etcdmain: etcd Version: 3.2.18
2019-02-13 02:53:28.144680 I | etcdmain: Git SHA: eddf599c6
2019-02-13 02:53:28.144684 I | etcdmain: Go Version: go1.8.7
2019-02-13 02:53:28.144688 I | etcdmain: Go OS/Arch: linux/amd64
2019-02-13 02:53:28.144692 I | etcdmain: setting maximum number of CPUs to 4, total number of available CPUs is 4
2019-02-13 02:53:28.144734 N | etcdmain: the server is already initialized as member before, starting as etcd
member...
2019-02-13 02:53:28.146625 I | etcdserver: name = master
2019-02-13 02:53:28.146637 I | etcdserver: data dir = /var/lib/etcd
2019-02-13 02:53:28.146642 I | etcdserver: member dir = /var/lib/etcd/member
2019-02-13 02:53:28.146645 I | etcdserver: heartbeat = 100ms
2019-02-13 02:53:28.146648 I | etcdserver: election = 1000ms
2019-02-13 02:53:28.146651 I | etcdserver: snapshot count = 10000
2019-02-13 02:53:28.146677 I | etcdserver: advertise client URLs = 2019-02-13 02:53:28.185353 I | etcdserver/api:
enabled capabilities for version 3.2
2019-02-13 02:53:28.185588 I | embed: ClientTLS: cert = /etc/kubernetes/pki/etcd/server.crt, key =
/etc/kubernetes/pki/etcd/server.key, ca = , trusted-ca = /etc/kubernetes/pki/etcd/old-ca.crt, client-cert-auth =
true
2019-02-13 02:53:30.080007 I | embed: ready to serve client requests
2019-02-13 02:53:0.0800130 I | etcdserver: published {Name:master ClientURLs:[https://127.0.0.1:2379]} to cluster
c9be114fc2da2776
2019-02-13 02:53:30.080281 I | embed: serving client requests on 127.0.0.1:2379
WARNING: 2019/02/13 02:53:30 Failed to dial 127.0.0.1:2379: connection error: desc = "transport: authentication
handshake failed: remote error: tls: bad certificate"; please retry.
```

Jeśli tworzyliśmy Kuberntesa przy pomocy kubeadm to trzeba przejrzeć logi podów komponentów:

```
▶ kubectl logs etcd-master
2019-02-13 02:53:28.144631 I | etcdmain: etcd Version: 3.2.18
2019-02-13 02:53:28.144680 I | etcdmain: Git SHA: eddf599c6
2019-02-13 02:53:28.144684 I | etcdmain: Go Version: go1.8.7
2019-02-13 02:53:28.144688 I | etcdmain: Go OS/Arch: linux/amd64
2019-02-13 02:53:28.144692 I | etcdmain: setting maximum number of CPUs to 4, total number of available CPUs is 4
2019-02-13 02:53:28.144734 N | etcdmain: the server is already initialized as member before, starting as etcd
member...
2019-02-13 02:53:28.146625 I | etcdserver: name = master
2019-02-13 02:53:28.146637 I | etcdserver: data dir = /var/lib/etcd
2019-02-13 02:53:28.146642 I | etcdserver: member dir = /var/lib/etcd/member
2019-02-13 02:53:28.146645 I | etcdserver: heartbeat = 100ms
2019-02-13 02:53:28.146648 I | etcdserver: election = 1000ms
2019-02-13 02:53:28.146651 I | etcdserver: snapshot count = 10000
2019-02-13 02:53:28.146677 I | etcdserver: advertise client URLs = 2019-02-13 02:53:28.185353 I | etcdserver/api:
enabled capabilities for version 3.2
2019-02-13 02:53:28.185588 I | embed: ClientTLS: cert = /etc/kubernetes/pki/etcd/server.crt, key =
/etc/kubernetes/pki/etcd/server.key, ca = , trusted-ca = /etc/kubernetes/pki/etcd/old-ca.crt, client-cert-auth =
true
2019-02-13 02:53:30.080007 I | embed: ready to serve client requests
2019-02-13 02:53:30.0800130 I | etcdserver: published {Name:master ClientURLs:[https://127.0.0.1:2379]} to cluster
c9be114fc2da2776
2019-02-13 02:53:30.080281 I | embed: serving client requests on 127.0.0.1:2379
WARNING: 2019/02/13 02:53:30 Failed to dial 127.0.0.1:2379: connection error: desc = "transport: authentication
handshake failed: remote error: tls: bad certificate"; please retry.
```

Jeśli kube-apiserver nie działa, to wtedy kubectl też nie zadziała, ale możemy sprawdzić logi bezpośrednio w Dockerze:

`docker ps -a` - wyświetlenie wszystkich procesów wraz z ich ID:

CONTAINER ID	STATUS	NAMES
23482a0925b	Up 12 minutes	k8s_kube-apiserver_kube-apiserver-master_kube-system_8758a3d10776bb527e
b9bf77348c96	Up 18 minutes	k8s_etcd_etcd-master_kube-system_2cc1c8a24b68ab9b46bca47e153e74c6_0
87fc69913973	Up 18 minutes	k8s_POD_etcd-master_kube-system_2cc1c8a24b68ab9b46bca47e153e74c6_0

Następnie uruchomienie logów kontenera:

`docker logs 87fc`

```
▶ docker logs 87fc
2019-02-13 02:53:28.144631 I | etcdmain: etcd Version: 3.2.18
2019-02-13 02:53:28.144680 I | etcdmain: Git SHA: eddf599c6
2019-02-13 02:53:28.144684 I | etcdmain: Go Version: go1.8.7
2019-02-13 02:53:28.144688 I | etcdmain: Go OS/Arch: linux/amd64
2019-02-13 02:53:28.144692 I | etcdmain: setting maximum number of CPUs to 4, total number of available CPUs is 4
2019-02-13 02:53:28.144734 N | etcdmain: the server is already initialized as member before, starting as etcd
member...
2019-02-13 02:53:28.146625 I | etcdserver: name = master
2019-02-13 02:53:28.146637 I | etcdserver: data dir = /var/lib/etcd
2019-02-13 02:53:28.146642 I | etcdserver: member dir = /var/lib/etcd/member
2019-02-13 02:53:28.146645 I | etcdserver: heartbeat = 100ms
2019-02-13 02:53:28.146648 I | etcdserver: election = 1000ms
2019-02-13 02:53:28.146651 I | etcdserver: snapshot count = 10000
2019-02-13 02:53:28.146677 I | etcdserver: advertise client URLs = 2019-02-13 02:53:28.185353 I | etcdserver/api:
enabled capabilities for version 3.2
2019-02-13 02:53:28.185588 I | embed: ClientTLS: cert = /etc/kubernetes/pki/etcd/server.crt, key =
/etc/kubernetes/pki/etcd/server.key, ca = , trusted-ca = /etc/kubernetes/pki/etcd/old-ca.crt, client-cert-auth =
true
```

Certificate API

Jeśli np. mamy nowego administratora, który przychodzi do pracy i trzeba mu stworzyć parę kluczy i certyfikat. Nowy admin generuje swój Private Key, tworzy CSR i wysyła go do istniejącego już admina. Następnie główny admin wysyła ten CSR do CA serwera i tam on jest podpisywany przy pomocy Private Keya CA i root certificatu. Następnie odesłany do głównego admina i on to przesyła do drugiego

admina. Teraz drugi admin może już mieć dostęp do clustra, do momentu aż certyfikat wygaśnie, potem trzeba będzie ten proces powtórzyć.

Tam gdzie umieściliśmy pliki ca.crt i ca.key, główne pliki to podpisywania innych certyfikatów, to miejsce nazywamy CA Server. Jeśli np. mamy te pliki na master node (tak jest domyślnie, jak używamy kubeadm), to master staje się CA Serverem.

Jeśli mamy dużo użytkowników i certyfikatów do ogarniania, to ręcznie możemy nie dać rady. Kubernetes ma wbudowany tool Certificate API, który może to robić automatycznie. Administrator po otrzymaniu CSR nie musi się logować CA Servera, ale tworzy CertificateSigningRequest Objetct. Ten plik mogą przeglądać inni administratorzy i może on być sprawdzony i zaakceptowany przy pomocy kubectl. Następnie stworzony certyfikat można udostępnić użytkownikom.

Przykład:

Użytkownik tworzy klucz: `openssl genrsa -out jane.key 2048`

Potem tworzy CSR i wysyła go do admina: `openssl req -new -key jane.key -subj "/CN=jane" -out jane.csr` Następnie admin tworzy CertificateSigningRequest Objetct. Jest to tworzone przy pomocy pliku yaml. W sekcji groups ustawiamy, w jakich grupach ma być użytkownik, w usages do czego ma służyć konto (jak to dla użytkownika to np. client auth), w polu request precyzujemy CSR wysłany przez usera. CSR nie jest precyzowany jednak jako tekst, ale musi być zakodowany przy pomocy base64 (`cat jane.csr | base64`), tak skonwertowany tekst umieszczamy w polu request. W nowych wersjach Kuberntesera wymagane jest też pole: signerName i to w tym przypadku będzie `kubernetes.io/kube-apiserver-client`.

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: akshay
spec:
  groups:
    - system:authenticated
  request: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRWFVRY
  signerName: kubernetes.io/kube-apiserver-client
  usages:
    - client auth
```

Teraz musimy uruchomić komendę `kubectl apply -f csr.yaml`, aby stworzyć obiekt csr.

Po stworzeniu tego pliku, administratorzy mogą przeglądać wszystkie CSR komendą: `kubectl get csr`. Następnie po walidacji zaakceptować go komendą: `kubectl certificate approve nazwa_csr`. Jeśli coś jest nie tak, to możemy odrzucić ten certyfikat (`kubectl certificate deny nazwa_csr`). Jeśli dodatkowo trzeba sprawdzić jego zawartość, czy wszystko się zgadza to trzeba go edytować: `kubectl edit csr nazwa_csr`. Jeśli chcemy usunąć ten zły csr, to używamy komendy `kubectl delete csr nazwa_csr`. Podpisanie certyfikatu odbywa się automatycznie i można go udostępnić userowi. Jeśli chcemy go obejrzeć to najlepiej w formacie yaml.

```
▶ kubectl get csr jane -o yaml
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  creationTimestamp: 2019-02-13T16:36:43Z
  name: new-user
spec:
  groups:
    - system:masters
    - system:authenticated
usages:
  - digital signature
  - key encipherment
  - server auth
  username: kubernetes-admin
status:
  certificate:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURDakNDQWZLZ0F3SUJBZ01VRmwy
Q2wxYXoxaW1M3JNVisreFRYQUowU3dnnd0RRWUpLb1pJaHZjTkFRRUwKQ1FBd0ZURVRN
QkVHQTFVRUF4TUthM1ZpWlhdVpYumxjekFlRncweE9UQXlNVE14TmpNeU1EqmFGd1dn
Y0ZFeD12ajNuSXY3eFdDS1NIRm5sU041c0t5Z0VxUkwzTfM5V29Ge1hHZDdwCmlEZ2FO
MVRMFBXTVhjN09FVnVjSwC1Yk4weEVHTkVwRU5tdU1BN1ZWeHVjS1h6aG91dDY0MED1
MGU0YXFkWV1KwmVMbjBvRTFCY3dod2xic0I1ND0KLS0tLS1FTkQgQ0VSVE1GSUNBVEUT
LS0tLQo=
  conditions:
    - lastUpdateTime: 2019-02-13T16:37:21Z
      message: This CSR was approved by kubectl certificate approve.
      reason: KubectlApprove
      type: Approved
```

Podany jest on tutaj jednak w sposób zakodowany więc trzeba go jeszcze odkodować: `echo "LS0..Qo=" | base64 --decode`

Wszystkie te automatyczne operacje na certyfikatach są przeprowadzane przez Controller-Manager. Ma on w sobie kontrolery nazwane CSR-APPROVING i CSR-SIGNING. Wiemy, że aby podpisywać certyfikaty musimy mieć ca.crt i ca.key, więc trzeba je też skonfigurować w opcjach Controller-Managera, aby on mógł to robić automatycznie:

```
▶ cat /etc/kubernetes/manifests/kube-controller-manager.yaml
spec:
  containers:
    - command:
      - kube-controller-manager
      - --address=127.0.0.1
      - --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt
      - --cluster-signing-key-file=/etc/kubernetes/pki/ca.key
      - --controllers=*,bootstrapsigner,tokencleaner
      - --kubeconfig=/etc/kubernetes/controller-manager.conf
      - --leader-elect=true
      - --root-ca-file=/etc/kubernetes/pki/ca.crt
      - --service-account-private-key-file=/etc/kubernetes/pki/sa.key
      - --use-service-account-credentials=true
```

KubeConfig

Wiemy, że można używać curla, aby połączyć się z rest api Kuberntesa i np. wyświetlić pody, przy okazji precyzując certyfikaty i klucze.

```
curl https://my-kube-playground:6443/api/v1/pods \
--key admin.key
--cert admin.crt
--cacert ca.crt
```

Ten request jest potem autentykowany przez kube-apiserver.

Można to samo oczywiście uzyskać używając kubectl i podając te flagi, ale robiąc to za każdym razem byłoby to bardzo uciążliwe. Dlatego, aby nie podawać za każdym razem tych parametrów, można je sprecyzować w KubeConfigFile i podać ten plik we fladze --kubeconfig.

```
kubectl get pods  
--kubeconfig config
```

Domyślnie ten plik będzie szukany w `$HOME/.kube/config`, więc jeśli stworzymy ten plik w tej lokalizacji, to nie trzeba będzie podawać ścieżki do niego za każdym razem. Dzięki temu działa nam np. `kubectl get pods`.

KubeConfig file ma specyficzny format. Ma 3 sekcje: clusters (różne clustery Kuberntesa, do których trzeba mieć dostęp np. prod, dev, aws, google), users (konta użytkowników, którzy mają mieć dostęp do tych clustrów) i contexts (połączenie clustra z userem, mówi nam, który użytkownik będzie się łączyć, do którego clustra np. Admin@Prod, Dev User@google). Tu nie tworzy się nowych użytkowników, ale bierzemy już istniejących z ich privilegami, dzięki temu nie trzeba precyzować certyfikatów użytkowników za każdym razem, gdy puszczaemy kubectl. W naszym przypadku następujące wpisy pojawią się w KubeConfig:

```
--server my-kube-playground:6443  
--client-key admin.key  
--client-certificate admin.crt  
--certificate-authority ca.crt
```

Server będzie w sekcji Clusters, klucze i certyfikaty do sekcji Users, potem tworzymy Context Admin@My-Kube-Playground do sprecyzowania połączenia.

```
apiVersion: v1  
kind: Config  
  
clusters:  
- name: my-kube-playground  
  cluster:  
    certificate-authority: ca.crt  
    server: https://my-kube-playground:6443  
  
contexts:  
- name: my-kube-admin@my-kube-playground  
  context:  
    cluster: my-kube-playground  
    user: my-kube-admin  
  
users:  
- name: my-kube-admin  
  user:  
    client-certificate: admin.crt  
    client-key: admin.key
```

Tak wygląda prawdziwy CubeConfig z wymaganymi parametrami w poszczególnych sekcjach. Każda sekcja to macierz, więc można dodawać wiele clustrów, userów i contextów, wszystkie te które mamy należy tam dodać. Dobrą praktyką jest podawać pełne ścieżki do certyfikatów, a nie same nazwy. Inną opcją na podanie certyfikatu jest wpisanie jego zakodowanej treści:

```
clusters:
- name: production
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJU
                                SUZJQ0FURSBRSRZFVRVNULS0tLS0KTU1J
                                Q1dEQ0NBVUFQVFbd0V6RVJNQThQTFV
                                RUF3d0libVYzTFhWelpYSXdnZ0VpTUEw
                                R0NTcUdTTSWIzRFFFQgpBUVVBQTRJQkR3
                                QXdnZ0VLQW9JQkFRRE8wV0pXK0RYc0FK
                                U0lyanBObzV2Uk1CcGxuemcrNnhj0StV
                                VndrS2kwCkxmQzI3dCsxZUVuT041TXVx
                                OTl0ZXZtTUVPbnJ
```

Po stworzeniu tego pliku nie trzeba uruchamiać żadnej dodatkowej komendy (`kubectl apply -f`), tylko od razu Kubernetes będzie czytać ten plik. Jeśli chcemy, aby któryś z contextów był domyślny, trzeba dodać pole `current-context`:

```
apiVersion: v1
kind: Config
current-context: dev-user@google
clusters:
- name: my-kube-playground (valu
```

`kubectl config view` - obejrzenie obecnego ConfigFile. Jeśli chcemy zobaczyć inny ConfigFile, nie defaultowy, to trzeba to sprecyzować w komendzie: `kubectl config view --kubeconfig=ścieżka_do_pliku`

`kubectl config use-context dany_context` - zmiana obecnego contextu.

Inne modyfikacje pliku ConfigFile robimy przy pomocy opcji w komendzie `kubectl config`:

```
▶ kubectl config -h
Available Commands:
  current-context Displays the current-context
  delete-cluster Delete the specified cluster from the kubeconfig
  delete-context Delete the specified context from the kubeconfig
  get-clusters  Display clusters defined in the kubeconfig
  get-contexts  Describe one or many contexts
  rename-context Renames a context from the kubeconfig file.
  set           Sets an individual value in a kubeconfig file
  set-cluster   Sets a cluster entry in kubeconfig
  set-context   Sets a context entry in kubeconfig
  set-credentials Sets a user entry in kubeconfig
  unset         Unsets an individual value in a kubeconfig file
  use-context   Sets the current-context in a kubeconfig file
  view          Display merged kubeconfig settings or a specified kubeconfig file
```

Jeśli mamy wiele namespace, to możemy też w ConfigFile w contextach ustawić dany namespace tak, aby w trakcie zmiany, ustawić ten namespace jako domyślny:

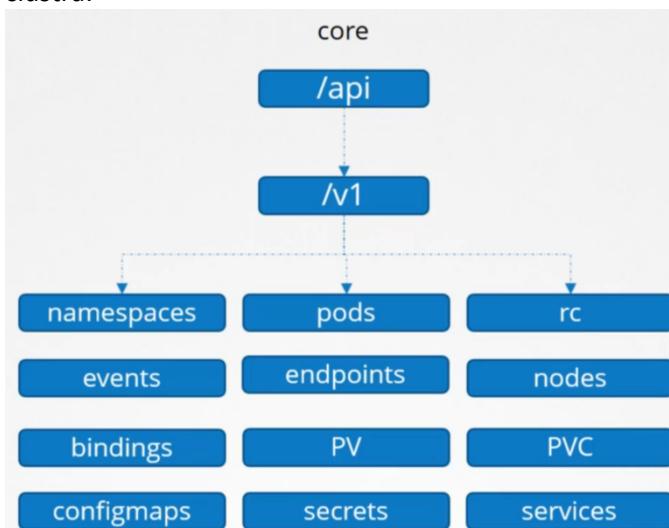
```
contexts:
- name: admin@production
  context:
    cluster: production
    user: admin
    namespace: finance
```

API Groups

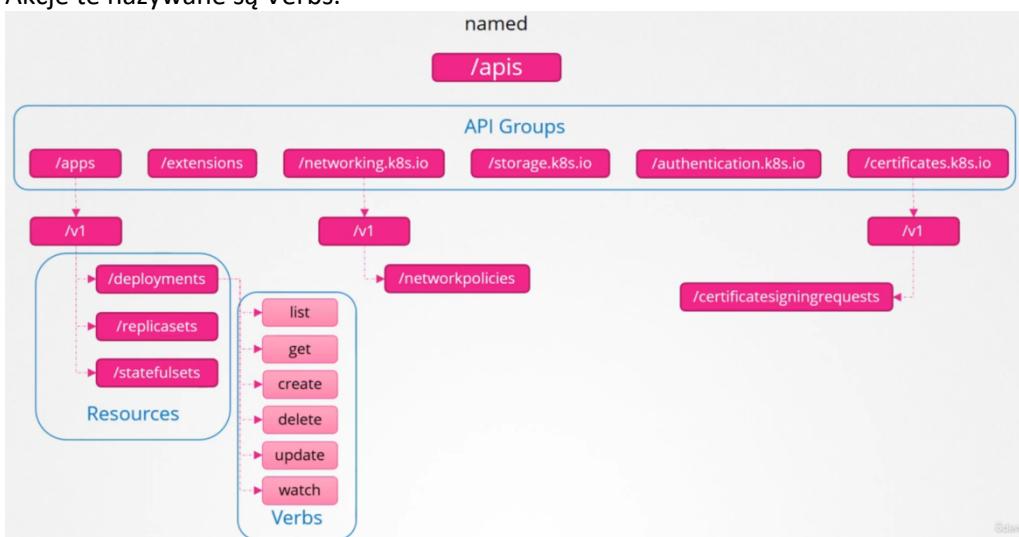
Wiemy, że możemy się komunikować z kube-apiserverem poprzez kubectl lub rest api. Np. `curl https://kube-master:6443/version` lub `curl https://kube-master:6443/api/v1/pods`

Kubernetes API jest pogrupowane na wiele API Groups, w zależności od ich przeznaczenia: /metrics, /healthz (do monitorowania zdrowia clustra), /version (do oglądania wersji clustra), /api (odpowiedzialne za funkcjonowanie clustra), /apis ((odpowiedzialne za funkcjonowanie clustra)), /logs (do integracji z 3rd-party logging applications) itp. Rozróżnienie na grupy pozwala nam łatwiej nadawać przywileje użytkownikom w trakcie autoryzacji.

Grupa API jest podzielona na dwie: /api - core group, /apis - named group. W API mamy główne funkcje clustra:



W APIS mamy nowsze funkcjonalności Kuryneresa, bardziej podzielone na podgrupy. Jest tu podział na API groups i resources wewnętrz takiej grupy. Każdy resource ma zestaw akcji połączonych ze sobą. Akcje te nazywane są Verbs.



Przeglądając dokumentację danego obiektu możemy zobaczyć do jakiej grupy należy. Możemy też wylistować dostępne grupy w Kubernetesie poprzez api call:

```
▶ curl http://localhost:6443 -k
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/healthz",
    "/logs",
    "/metrics",
    "/openapi/v2",
    "/swagger-2.0.0.json",
```

Jeśli chcemy wyświetlić tylko named groups, to trzeba jeszcze dodać *grep name*.

Jeśli chcemy się łączyć z Kubernetesem, należy pamiętać, że trzeba też podać certyfikaty i klucze do autentykacji w poleceniu. Bez nich większość API group nie zadziała.

```
curl http://localhost:6443 -k --key admin.key --cert admin.crt --cacert ca.crt
```

Inną opcją jest uruchomienie najpierw klienta kubectl proxy: *kubectl proxy*. Komenda ta uruchamia proxy service lokalnie na porcie 8001 i używa credentiali i certyfikatów z lokalnego KubeConfig file, aby połączyć się z clustrem. Można się połączyć do proxy przez rest api i proxy wyśle naszą prośbę do kube-apiservera (w poleceniu mamy localhost:8001 zamiast np kube-master:6443).

```
▶ curl http://localhost:8001 -k
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/healthz",
    "/logs",
    "/metrics",
    "/openapi/v2",
    "/swagger-2.0.0.json",
```

Tutaj należy nie mieszać pojęć kube proxy i kubectl proxy. Kube proxy jest używane do zezwolenia na połaczenie pomiędzy podami i serwisami pomiędzy różnymi nodami w clustrze. Kubectl proxy to http proxy service client stworzony przez kubectl do łączenia się z kube-apiserverem.

Authorization

Autoryzacja określa co określeni użytkownicy lub aplikacje mogą robić wewnątrz clustra. Chcemy aby np. developerzy mogli tworzyć aplikacje, ale nie usuwać obiektów sieciowych, tak samo z Service Accounts dla aplikacji. Innym sposobem podziału może być nadanie przywilejów użytkownikom tylko wewnątrz ich namespace.

Są różne rodzaje autoryzacji: Node Authorization, Attribute-Based Authorization, Role-Based Authorization i Webhook:

-W trakcie pracy użytkownicy często łączą się do kube-apiservera, tak samo kubelety z nodów, aby zapisać stan podów, czy odczytać inne informacje. Wszystkie te prośby są obsługiwane przez Node Authorizer. Wcześniej było mówione, że kubelety w certyfikacie muszą mieć nazwę *system:node:node01*, to właśnie dlatego, że requesty od użytkowników z tą nazwą są obsługiwane przez Node Authorizer. On też nadaje odpowiednie przywileje w tej sytuacji.

-Attribute-Based Authorization ma miejsce, jeśli zewnętrzny user lub grupa userów chce mieć dostęp do clustra i nadajemy im odpowiednie przywileje. Jest to robione przez stworzenie pliku policy i w formacie json, tam sprecyzowane są przywileje np.

```
{"kind": "Policy", "spec": {"user": "dev-user", "namespace": "*", "resource": "pods", "apiGroup": "*"}}
```

Taki plik wysyła się do kube-apiservera. W ten sam sposób trzeba stworzyć policy dla każdego użytkownika lub grupy w tym samym pliku. Za każdym razem, gdy trzeba wprowadzić zmianę w bezpieczeństwie, trzeba modyfikować ten plik i zrestartować kube-apiserver. Przez to pliki konfiguracyjne do Attribute-Based Authorization są trudne do zarządzania.

-W RBAC jest łatwiej, bo zamiast dopisywać zbioru przywilejów do użytkownika definiujemy rolę. Np. tworzymy rolę dla developerów i potem wszystkich developerów przypisujemy do niej. W przypadku zmiany dostępów, wystarczy tylko zmodyfikować rolę i będzie to uwzględnione od razu u wszystkich użytkowników.

-Jeśli chcemy zarządzać autoryzacją zewnętrznie (nie w Kubernetesie). Jest np. tool Open Policy Agent, który pomaga przy autoryzacji. Kubernetes może wysyłać do niego zapytania z informacjami o użytkowniku, aby nadać przywileje.

Są dwie dodatkowe możliwości autentykacji w Kubernetesie: AlwaysAllow (zawsze nadaje przywilej bez sprawdzania autoryzacji) i AlwaysDeny (nigdy nie nadaje przywileju).

Sposoby autoryzacji są ustalone w kube-apiserverze flagą --authorization-mode. Jeśli tego nie ustawimy to domyślnie jest AlwaysAllow. Można też ustawić wiele trybów na raz po przecinku:

```
ExecStart=/usr/local/bin/kube-apiserver \
--advertise-address=${INTERNAL_IP} \
--allow-privileged=true \
--apiserver-count=3 \
--authorization-mode=Node,RBAC,Webhook \
--bind-address=0.0.0.0 \
--enable-swagger-ui=true \
--etcd-cafile=/var/lib/kubernetes/ca.pem \
--etcd-certfile=/var/lib/kubernetes/apiserver-etcd-client.crt \
--etcd-keyfile=/var/lib/kubernetes/apiserver-etcd-client.key \
--etcd-servers=https://127.0.0.1:2379 \
--event-ttl=1h \
--kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \
--kubelet-client-certificate=/var/lib/kubernetes/apiserver-etcd-client.crt \
--kubelet-client-key=/var/lib/kubernetes/apiserver-etcd-client.key \
--service-node-port-range=30000-32767 \
--client-ca-file=/var/lib/kubernetes/ca.pem \
--tls-cert-file=/var/lib/kubernetes/apiserver.crt \
--tls-private-key-file=/var/lib/kubernetes/apiserver.key \
--v=2
```

Jeśli mamy ustawionych kilka trybów, to każdy request usera jest autoryzowany przez wszystkie ustawione tryby. Jeśli pierwszy go odrzuci to idzie do drugiego, aż w końcu któryś zaakceptuje.

Role Based Access Control

Rolę tworzymy jako obiekt przez stworzenie pliku yaml.

```
developer-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["list", "get", "create", "update", "delete"]
```

W sekcji rules tworzymy dostępy. Każda rule ma 3 sekcje: apiGroups (dla core group - /api, możemy zostawić tą sekcję pustą, ale dla innych trzeba to sprecyzować), resources (resourcy, do których ma

być dostęp) i verb (określone przywileje). Można do każdej roli dodać wiele ruli. Jeśli chcemy sprecyzować dostęp tylko do określonych podów to możemy dodać kolejną sekcję resourceNames i tam podać ich nazwy:

```
rules:  
- apiGroups: [""]  
  resources: ["pods"]  
  verbs: ["get", "create", "update"]  
  resourceNames: ["blue", "orange"]
```

Następnie tworzymy rolę przy pomocy: `kubectl create -f developer-role.yaml`

Następnym krokiem jest połączenie użytkownika z tą rolą. W tym celu musimy stworzyć inny obiekt nazwany RoleBinding.

`devuser-developer-binding.yaml`

```
apiVersion: rbac.authorization.k8s.io/v1  
kind: RoleBinding  
metadata:  
  name: devuser-developer-binding  
subjects:  
- kind: User  
  name: dev-user  
  apiGroup: rbac.authorization.k8s.io  
roleRef:  
  kind: Role  
  name: developer  
  apiGroup: rbac.authorization.k8s.io
```

W sekcji subjects precyzujeśmy szczegóły użytkownika (lub grupy). W sekcji roleRef ustawiamy odpowiednią rolę. RoleBinding tworzymy komendą `kubectl create -f nazwa.yaml`. Należy pamiętać, że Role i RoleBinding działają tylko wewnątrz danego namespace. Jeśli chcemy, aby użytkownik miał też privilege w innych namespace, to trzeba je podać w pliku yaml przy tworzeniu usera.

`kubectl get roles` - wyświetlenie roli

`kubectl get rolebindings` - wyświetlenie rolebindings

`kubectl describe role developer` - wyświetlenie większej ilości informacji o danej roli (analogicznie można wyświetlić info o RoleBinding).

Jeśli chcemy samemu sprawdzić, czy mamy dostęp do danej czynności to możemy uruchomić komendę:

`kubectl auth can-i create deployments lub delete nodes` - sprawdzimy tak, czy mamy dany privilege.

Jeśli chcemy sprawdzić jako admin, czy dany user ma dany przywilej to możemy uruchomić:

`kubectl auth can-i create deployments --as dev-user`

`kubectl auth can-i create deployments --as dev-user -n nazwa_namespace` - tutaj sprawdzenie, czy dany użytkownik ma dane dostępny w konkretnym namespace.

Cluster Roles and Role Bindings

Zwykłe Role i Role Bindings obejmują tylko resourcy wewnątrz namespace. Jeśli chcemy nadać rolę np. dla noda (jest to cluster scoped resource) to trzeba użyć Cluster Roles. Cluster Roles i Cluster Role Bindings są takie same jak w poprzedniej lekcji, ale przypisujemy je do resourców cluster scoped. Przykładowo Cluster Admin może tworzyć, usuwać i zarządzać nodami. Przykładowy plik yaml do stworzenia takiej roli to:

```
cluster-admin-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-administrator
rules:
- apiGroups: []
  resources: ["nodes"]
  verbs: ["list", "get", "create", "delete"]
```

Następnie, aby połączyć danego użytkownika z tą rolą tworzymy Cluster Role Binding:

```
cluster-admin-role-binding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-role-binding
subjects:
- kind: User
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-administrator
  apiGroup: rbac.authorization.k8s.io
```

Detalie użytkownika precyzujemy w sekcji subjects. Oba powyższe obiekty tworzymy komendą *kubectl create -f plik.yaml*

Można też tworzyć Cluster Role i Cluster Role Binding do obiektów, które są grupowane w namespacach (namespaced objects) takich jak pody, replicasety, service itp. Wtedy użytkownik będzie miał dane przywileje do wszystkich obiektów danego typu w każdym namespace (np. dostęp do wszystkich podów w clustrze).

kubectl api-resources - wyświetlenie api-resourców.

Service Accounts

Są dwa rodzaje kont w Kubernetesie: user account (używane przez ludzi np. admin lub developer account) i service account (używane przez programy np. konto dla aplikacji do interakcji z clustrem. Przykładowo monitorująca aplikacja musi pobierać Kubernetes api dla metryk, lub konto dla Jenkinsa będzie deployować aplikacje).

Przykład mamy aplikację, która się chce łączyć z kube-api i zapytać o listę podów w clustrze i potem ją wyświetlić na stronie internetowej. Aby złożyć zapytanie, musi ona być zautentykowana przy pomocy Service Account.

kubectl create serviceaccount nazwa - tworzenie service account.

Po stworzeniu Service Account, jego token zostanie automatycznie stworzony. Ten token zostanie potem użyty do autentykacji w kube-api.

kubectl describe serviceaccount nazwa - więcej informacji o service account zostanie wyświetlone, między innymi token. Zawartość tokenu ten jest przechowywana w secret objecie, a w opisie poda mamy tylko jego nazwę. Żeby zobaczyć pełną zawartość tokena należy uruchomić komendę:
kubectl describe secret nazwa

Token ten można użyć do autentykacji przykładowo:

Curl <https://IP:6443/api> -insecure --header "Authorization: Bearer zawartość_tokenu". W Kubernetes Dashboardzie trzeba wkleić token do odpowiedniego pola w celu autentykacji. Odpowiednie przywileje do Service Account możemy dodać przy pomocy RBAC.

Jeśli mamy naszą aplikację hostowaną w Kubernetesie (nie jest to 3-rd party połączenie). To możemy autentykację przeprowadzić poprzez mountowanie secretu z service tokenem jako volume wewnątrz poda z aplikacją. Możemy zobaczyć, że dla każdego namespace domyślnie jest stworzony Service Account, o nazwie default. Gdy jakikolwiek pod jest stworzony w ns, token z domyślnego Service Account jest montowany jako volume w tym podzie. Możemy to zobaczyć robiąc describe na dowolnym podzie:

```
▶ kubectl describe pod my-kubernetes-dashboard
Name:           my-kubernetes-dashboard
Namespace:      default
Annotations:    <none>
Status:         Running
IP:            10.244.0.15
Containers:
  nginx:
    Image:        my-kubernetes-dashboard
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from default-token-j4hkv (ro)
Conditions:
  Type          Status
Volumes:
  default-token-j4hkv:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-j4hkv
    Optional:   false
```

Jeśli wewnątrz poda wejdziemy do ścieżki, gdzie zamontowany jest token, zobaczymy, że zawiera on 3 pliki:

```
▶ kubectl exec -it my-kubernetes-dashboard ls /var/run/secrets/kubernetes.io/serviceaccount
ca.crt  namespace  token
```

Plik z tokenem nazywa się token. Jest on potrzebny do dostawania się do kube-api. Należy pamiętać, że domyślny service account ma dużo restrykcji, ma tylko dostęp do podstawowych kube-api queries. Jeśli chcielibyśmy, aby pod używał innego Service account, to trzeba zmodyfikować plik yaml do tworzenia poda i dodać pole serviceAccountName. Nie można zmienić Service Account istniejącego poda, trzeba go zredukować. Jeśli jest to deployment to po zmianach, automatycznie pody się odtworzą.

```
pod-definition.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard
  serviceAccountName: dashboard-sa
```

Jeśli nie chcemy, żeby dla danego poda Service Account był automatycznie mountowany należy dodać opcję automountServiceAccountToken - false:

```
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard
  automountServiceAccountToken: false
```

Image Security

Gdy tworzymy pody precyzujemy w pliku yaml nazwę image np. image: nginx. Nazwa ta odpowiada Dockerowym nazwom imagów. Tutaj nginx to image lub nazwa repozytorium. Tak naprawdę gdy precyzujemy nginx to jest to library/nginx. Pierwsza część to użytkownik lub service account, więc jeśli tego nie podamy, to będzie domyślnie library. Library to nazwa domyślnego konta, gdzie oficjalne image dockerowe są przechowywane. Jeśli mielibyśmy swoje repozytorium, usera i image to można precyzować to w analogiczny sposób.

Jeśli w trakcie podawania image nie sprecyzujemy gdzie obraz ma być przechowywany to będzie to domyślnie Docker Hub, którego DNS to docker.io, więc nasz image ma tak naprawdę postać: docker.io/library/nginx. Inne znane repozytoria to np gcr.io. Są one darmowe i każdy ma do nich dostęp. Jeśli budujemy swoją własną aplikację, to dobrze jest użyć prywatnego repozytorium, tak by nikt tam nie miał dostępu. Np. AWS, czy Azure oferują takie repozytoria.

Aby zwiększyć bezpieczeństwo repozytorium, można wymagać autentykacji, przy pomocy loginu i hasła w trakcie logowania. Najpierw trzeba będzie się zalogować do swojego repozytorium:

docker login private-registry.io

Teraz można uruchomić aplikację używając prywatnego image:

docker run private-registry.io/apps/internal-app

Jeśli chcemy to samo zrobić w Kubernetesie należy w pliku yaml dla image podać pełną ścieżkę do prywatnego image:

Image: private-registry.io/apps/internal-app

Należy jednak pamiętać, że pozostaje nam autentykacja do repozytorium. Wiemy, że image są pobierane przez Docker runtime na worker nodach.

Pierwszym krokiem, jaki musimy zrobić to stworzyć secret o typie docker-registry (jest to wbudowany typ przygotowany do przechowywania dockerowych credentiali) i w nim podać credentiali:

```
kubectl create secret docker-registry regcred \
--docker-server= private-registry.io \
--docker-username= registry-user \
--docker-password= registry-password \
--docker-email= registry-user@org.com
```

Następnie w pliku yaml poda ustawiamy opcję imagePullSecret w sekcji spec, aby pod skorzystał z credentiali:

nginx-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: private-registry.io/apps/internal-app
  imagePullSecrets:
  - name: regcred
```

Security Contexts

W trakcie uruchamiania dockerowego kontenera mamy możliwość podania wielu zabezpieczeń np. ID usera, możliwości Linuxa, które mogą być dodane lub zabrane z kontenera itp. W Kubernetesie możemy te ustawienia zrobić na poziomie poda i wtedy będą zaaplikowane do wszystkich jego kontenerów. Jeśli ustawimy jednocześnie w podzie i kontenerze, to ustawienia z kontenera nadpiszą te z poda.

Aby ustawić Security Context dla poda, trzeba w pliku yaml dodać opcję securityContext w sekcji spec. Tam możemy dodawać odpowiednie zabezpieczenia np. runAsUser pozwala nam ustawić ID usera dla poda:

```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  securityContext:
    runAsUser: 1000

  containers:
    - name: ubuntu
      image: ubuntu
      command: ["sleep", "3600"]
```

Jeśli chcemy to samo ustawić wewnątrz kontenera, należy to przenieść bezpośrednio do precyzowania kontenera. Jeśli chcemy dodać jakieś możliwości Linuxa to też w opcji securityContext:

```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
    - name: ubuntu
      image: ubuntu
      command: ["sleep", "3600"]
      securityContext:
        runAsUser: 1000
      capabilities:
        add: ["MAC_ADMIN"]
```

Aby sprawdzić jaki użytkownik wykonuje polecenia w kontenerze trzeba uruchomić: *kubectl exec ubuntu-sleeper – whoami*

Network Policy

Przykład: Mamy web server, który jest front endem dla użytkowników i app server jako back-end api i serwer bazy danych. Użytkownik wysyła zapytanie do web servera na porcie 80, następnie web server wysyła prośbę do app servera po porcie 5000, który wyciąga dane z bazy danych po porcie 3306 i wysyła je przez front-end do klienta. Mamy tu dwa rodzaje ruchu ingress i egress. Ingress to ruch przychodzący od klienta do web servera. Egress to request wysłany przez web server do app servera. Natomiast dla app servera, ingress to request od web severa, a egress to zapytanie do bazy danych (ruch przychodzący i wychodzący). Dla bazy danych mamy tylko ingress od app servera. Zwracanie danych do klienta nie ma dużego znaczenia.

W Kubernetesie mamy zestaw nodów i w każdym z nich pody, serwisy itp. Każdy z tych obiektów ma swój oddzielnny adres IP. Domyslnie wszystkie pody powinny się ze sobą komunikować bez ustawiania ruli. Domyslnie wszystkie te obiekty są w jednej Virtual Network i mogą się do siebie zwracać po nazwach. Domyslnie Kubernetes ma ustawioną "All Allow" rulę. Teraz wracając do przykładu na początek, każdy z serwerów jest podem i tworzymy serwisy, aby pody mogły się ze sobą komunikować,

oraz aby mogły się komunikować z end-userem. Domyślnie jest to wszystko pozwierane, ale co w przypadku gdybyśmy chcieli zablokować ruch np. z web servera do database servera. Trzeba wtedy zaimplementować Network Policy.

Network Policy to obiekt Kuberntesesa, który przypisujemy do jednego lub więcej podów. Możemy w nim ustawać network rule. Aby połączyć pod i Network Policy musimy użyć Labeli i Selectorów. Labele podów muszą się zgadzać z polem podSelector w Network Policy. Następnie budujemy rule. W policyTypes ustawiamy, czy Policy ma pozwalać na ruch ingress, egress, czy oba. Przykładowy plik yaml:

```
policyTypes:
- Ingress
ingress:
- from:
  - podSelector:
    matchLabels:
      name: api-pod
  ports:
  - protocol: TCP
    port: 3306
```

App server (app pod) jest precyzowany przy pomocy labeli i selectorów. Podajemy też port połączenia. Pełny plik yaml dla Network Policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
      matchLabels:
        name: api-pod
    ports:
    - protocol: TCP
      port: 3306
```

w sekcji spec najpierw podajemy podSelector, czyli te pody, które będą miały dodaną tą Network Policy. Potem ustawiamy rodzaj policy i w rulach ustawiamy jaki ruch będzie zezwolony przez policy i po jakim porcie, tutaj mamy ingress z app poda po porcie 3306. W tym przypadku nie trzeba ustawiać egress, ponieważ, gdy app pod zrobi zapytanie w bazie danych, odpowiedź od bazy jest automatycznie puszczana, więc nie trzeba tu dodatkowych rul. Potrzebne by to było, gdyby z bazy wychodził jakiś ruch.

`kubectl create -f policy.yaml` - stworzy Network Policy.

Network Policy jest wprowadzone przez Netowrk Solution zaimplementowane w Kubernetesie. W związku z tym, nie wszystkie rozwiązania wspierają Network Policy. Netowrk Policy może być użyte z Kube-router, Calico, Romana i Weave-net, natomiast Flannel ich nie wspiera. Pomimo niewspierania, możemy jednak tworzyć policy, ale nie będą one po prostu wdrażane automatycznie.

Developing Network Policies

Mając ten sam przykład co wyżej.

Jeśli chcielibyśmy dodatkowo dodać kolejną restrykcję na wybieranie źródła ruchu możemy np. dodać namespaceSelector, który pozwoli tylko na ruch przychodzący z określonego namesapce (należy pamiętać, że najpierw labele należy nałożyć na namespace, a dopiero potem ustawić to w Network Policy).

Jeśli natomiast chcielibyśmy zezwolić na ruch zewnętrz clustra, możemy też określić konkretny adres IP, z którego będzie mógł przychodzić ruch:

```
- podSelector:  
    matchLabels:  
        name: api-pod  
  
    namespaceSelector:  
        matchLabels:  
            name: prod  
  
- ipBlock:  
    cidr: 192.168.5.10/32
```

Należy pamiętać, że każdą ingress rulę podajemy od myślnika. W przykładzie powyżej mamy 2: jedną na pod i namespace, a drugą na adres IP.

Jeśli chcemy dodać egress rule, to dodajemy najpierw Egress w policyTypes i dodatkowo opcję egress i w niej precyzujeśmy szczegółowo rolę:

```
egress:  
- to:  
    - ipBlock:  
        cidr: 192.168.5.10/32  
  
    ports:  
        - protocol: TCP  
          port: 80
```

W przypadku egress mamy opcję to zamiast from, a reszta działa tak samo.

Storage

Introduction to Docker Storage

W Dockerze mamy Storage Drivers i Volume Drivers.

Jak Dockerze przechowuje dane w lokalnym systemie plików. Jeśli zainstalujemy Docker, to stworzy on taką strukturę folderów: `/var/lib/docker` będzie tam wiele folderów (aufs, containers, image, volumes itp.). To właśnie tu domyślnie Docker przechowuje dane (pliki związane z image, contenerami itp.).

W jaki sposób Docker przechowuje dane? Należy pamiętać, że podczas budowy image, Docker buduje go w warstwowej architekturze. Każda nowa linia w Dockerfile tworzy nową warstwę (layer) w image Dockerowym ze zmianami z poprzednich warstw:

```
Dockerfile
FROM Ubuntu

RUN apt-get update && apt-get -y install python

RUN pip install flask flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask
run

docker build Dockerfile -t mmumshad/my-custom-app
```

Np. tutaj pierwsza warstwa to domyślny Ubuntu, potem instalujemy paczki apt, to będzie druga warstwa itd. Jakoże każda warstwa przechowuje tylko swoje wprowadzone zmiany, każda ma inny rozmiar w MB. Zaletami takiej architektury, jest to, że gdybyśmy np. teraz chcieli zbudować drugi image używając Dockerfile, który ma taki sam początek, ale dwa ostatnie kroki inne, wszystkie powtarzające się instrukcje nie byłyby powtarzane, ale byłyby użyte z cache. Dzięki temu operacja ta jest o wiele szybsza i zajmie mniej miejsca na dysku. Przydaje się to, gdy chcemy np. zmodyfikować kod aplikacji i szybko go przetestować.

Wszystkie warstwy ze zdjęcia są użyte do stworzenia ostatecznego Docker image, więc są to Image Layers. Po zakończeniu builda, nie można modyfikować ich zawartości, chyba, że będziemy chcieli stworzyć nowy obraz. Jeśli teraz na podstawie tego obrazu stworzymy contener (`docker run image`) zostanie stworzona kolejna 6 warstwa (Container Layer) i będzie można ją modyfikować. W niej będą też przechowywane pliki stworzone przez container takie jak logi, pliki chwilowe, czy każde pliki zmodyfikowane przez użytkownika. Ta warstwa istnieje tak długo, jak istnieje container. Jak on padnie, to wszystkie zmiany przepadną.

Jeśli chcielibyśmy w kontenerze zmodyfikować pliki, które są stworzone w domyślnym image (np. kod aplikacji), to gdy zaczniemy je modyfikować, Docker stworzy ich kopię w Container Layer (należy pamiętać, że Image Layer jest tylko Read Only) i będziemy tak naprawdę modyfikować tę kopię. Nazywa się to Copy-On-Write mechanism.

Co w przypadku, gdy chcemy zapisać gdzieś na stałe zmiany z naszego containera, tak aby, po jego usunięciu, one nie przepadły. Wtedy można np. dodać Persistent Volume. Aby to zrobić musimy najpierw stworzyć volume:

`docker volume create nazwa`.

Stworzy to folder `data_volume` w `/var/lib/docker/volumes`. Następnie można zamontować ten volume w trakcie tworzenia containera i będzie on dostępny w modyfikalnej warstwie:

```
docker run -v data_volume:/var/lib/mysql mysql
```

Po dwukropku precyzujeśmy lokalizację na kontenerze, gdzie pliki będą zapisane. Dzięki temu zapisując zmiany w kontenerze w tej lokalizacji, będą one tak naprawdę zapisane w volume. Jeśli nie stworzylibyśmy volume przez tworzeniem contenera, ale użyli flagi -v to Docker automatycznie stworzy nam pożądaną przez nas volume.

Ta operacja to Volume Mounting.

Jeśli mielibyśmy jakiś zewnętrzny storage na Docker hoście i chcielibyśmy go podpiąć do kontenera zamiast domyślnej lokalizacji (/var/lib/docker), to w trakcie tworzenia containera musimy podać pełną ścieżkę do tego storage:

```
docker run -v /data/mysql:/var/lib/mysql mysql
```

Ta operacja to Bind Mounting.

Używanie flagi -v jest klasycznym podejściem. Nowa opcja to --mount, wtedy możemy podać dodatkowo więcej parametrów naszego volume:

```
docker run \
--mount type=bind,source=/data/mysql,target=/var/lib/mysql mysql
```

Storage Drivers są odpowiedzialne za wszystkie te operacje (tworzenie warstw, przenoszenie plików pomiędzy nimi itp.) oraz pomagają zarządzać storage w image i containerach. Przykładowe Storage Drivers: AUFS, ZFS, BTRFS, Device Mapper, Overlay, Overlay2. Użycie różnych Storage Drivers zależy od systemu operacyjnego naszego image. Np. dla Ubuntu jest AUFS.

Volume Drivers in Docker

Volume Drivers pomagają zarządzać volumami. Domyślny nazywa się Local. Pomaga on stworzyć volume na Docker hoście i przechowywać jego dane w /var/lib/docker/volumes. Są też inne drivery pomagające tworzyć volume w 3-rd party: Azure File Storage, Convoy, DigitalOcean Block Storage, Flocker, gce-docker, GlusterFS, NetApp, RexRay, Vmware vSphere Storage. Niektóre z tych driverów wspierają różnych storage providerów np. RexRay może być użyty do tworzenia storage w Aws EBS, S3, Google Persistent Desk itd.

Gdy tworzymy container możemy w komendzie sprecyzować jaki volume driver chcemy użyć, np. tu tworzymy volume w AWS EBS:

```
docker run -it \
--name mysql
--volume-driver rexray/ebs
--mount src=ebs-vol,target=/var/lib/mysql
mysql
```

Jak container zostanie usunięty, to dane będą zapisane w chmurze AWS.

Container Storage Interface (CSI)

W przeszłości Kubernetes używał tylko Dockera jako silnika do containerów i cała integracja pomiędzy Kubernetesem i Dockerem była w kodzie źródłowym. Teraz doszły też silniki rkt i cri-o, więc wprowadzono rozszerzenia. Dzięki temu stworzono też Container Runtime Interface. Określa on jak Kubernetes będzie się komunikował z silnikiem containerowym. Pozwala to na łatwą komunikację z nowymi silnikami w przyszłości.

Analogicznie mamy Container Network Interface do integracji z różnymi Network Solutions.

Tak samo Container Storage Interface jest stworzony do wspierania wielu rozwiązań do storage (np. Portworx, AWS EBS, Azure Disk, Dell EMC, GlusterFS itp.). CSI jest stworzony w uniwersalnym standardzie, pozwala to na pracę każdemu orchestration tool (np. Kubernetes, Cloud Foundry i Mesos) na pracę z każdym storage.

CSI tworzy zbiór RPC (remote procedure call), które będą wywoływanie przez orchestratora. Następnie te procedury mają zostać wykonane przez storage driver, dodatkowo ma on zwrócić informację o ukończeniu (przykładowe procedury to CreateVolume, DeleteVolume itp.).

Volumes

Volume jest dodany do containera po to, aby dane w nim przetwarzane nie zostały stracone, jak container będzie usunięty. W Kubernetesie pody mają podobną naturę do containerów, więc też potrzebują volume.

Przykładowo mamy poda, który generuje losowy numer od 1 do 100 i zapisuje go w pliku /opt/number.out. Aby go zatrzymać po usunięciu poda, używamy volume. W volume musimy ustawić rodzaj storage (w tym przypadku będzie to storage bezpośrednio na hoście).

```
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
    - image: alpine
      name: alpine
      command: ["/bin/sh", "-c"]
      args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
      volumeMounts:
        - mountPath: /opt
          name: data-volume

  volumes:
    - name: data-volume
      hostPath:
        path: /data
        type: Directory
```

W spec->volumes tworzymy volume i podajemy rodzaj storage (Directory i ścieżka to /data bezpośrednio na hoście). Następnie w spec->containers dodajemy opcję volumeMounts i tam podajemy gdzie w kontenerze zamontować volume (tu /opt, bo tam będzie zapisany plik).

Takie rozwiązanie ze storage (bezpośrednio na hoście) działa dobrze dla jednego noda, ale nie jest rekommendowane dla wielu nodów, ponieważ, jak użyjemy /data to Kubernetes będzie oczekiwał, że to directory będzie takie same na wszystkich nodach w clustrze. Jako, że będą to inne serwery, to pliki też będą najprawdopodobniej inne.

Innymi opcjami na stworzenie storage są np. NFS, GLusterFS, SCALEIO, AWS EBS, Azure Disk or File, Google Persistent Disk itp. Przykładowo jeśli chcielibyśmy tu zamontować AWS EBS storage to trzeba byłoby zmienić trochę sekcję volumes:

```
volumes:
  - name: data-volume
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

Persistent Volumes

W poprzedniej lekcji ustawialiśmy wszystkie parametry dotyczące volume bezpośrednio w pliku yaml poda. W przypadku gdy mamy dużo różnych podów może to być pracochnonne. Lepiej jest

skonfigurować jeden duży storage i pozwolić użytkownikom pobierać jego mniejsze części do podów. W tym przypadku pomaga nam Persistent Volume.

Jest to zbiór storage volumów skonfigurowanych przez administratora, po to, aby użytkownicy go używali, w trakcie deploymentu aplikacji. Użytkownicy wybierają części dużego volume przy pomocy Persistent Volume Claims.

pv-definition.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-voll
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  hostPath:
    path: /tmp/data
```

Lub (zamiast hostPath):

```
awsElasticBlockStore:
  volumeID: <volume-id>
  fsType: ext4
```

Plik yaml do tworzenia Persistent Volume. Ustawiamy w nim accessMode (może być ReadWriteOnce, ReadOnlyMany i ReadWriteMany) określający, jak volume powinien być zamontowany na hoście. Następnie w storage ustawiamy rozmiar miejsca zarezerwowany dla tego volume. Dodatkowo ustawiamy też rodzaj storage (bezpośrednio na hoście, czy np. z chmury).

Kubectl create -f pv-definition.yaml - tworzy PersistentVolume.

Persistent Volume Claims

PVC jest tworzony po to, aby uruchomić korzystanie z Persistent Volume na nodzie. PVC jest też osobnym obiektem w Kubernetesie. PVC jest łączone z Persistent Volume na podstawie ustawień w volume. Każdy PVC jest połączony tylko z pojedynczym Persistent Volume. Kubernetes będzie próbował znaleźć taki volume, aby zaspokoić potrzebę miejsca z PVC. Można też wprowadzać inne prośby w PVC jak AccessMode, StorageMode itp. Jeśli natomiast chcemy użyć konkretnego Persistent Volume, możemy użyć Label i Selectorów, aby go połączyć z PVC.

Należy pamiętać, że może się zdarzyć tak, że będziemy potrzebować mniej miejsca, ale dostaniemy duży volume. Miejsce, którego nie użyjemy nie będzie wtedy mogło być przypisane do innego PVC, bo pomiędzy PVC i Persistent Volume jest relacja 1:1.

pvc-definition.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

W pliku yaml PVC, w sekcji spec ustawiamy AccessMode i pożądane wolne miejsce.

`kubectl create -f plik.yaml` - stworzy PVC. Jeśli nie znajdzie on wolnego Persistent Volume, to będzie w stanie Pending dopóki jakiś się nie zwolni.

`kubectl get persistentvolumeclaims` - wyświetlenie wszystkich PVC

Po stworzeniu PVC, Kubernetes przegląda wszystkie Persistent Volume i dołącza ten odpowiedni. Teraz w komendzie get zobaczymy więcej szczegółów.

`Kubectl delete persistentvolumeclaim nazwa` - usunie PVC, a to co się stanie z Persistent Volume zależy od opcji, którą ustawimy w pliku yaml Persistent Volume:

persistentVolumeReclaimPolicy: Retain

Domyślna wartość to Retain, więc po usunięciu PVC, Persistent Volume zostanie, chyba, że ktoś go ręcznie usunie. Należy jednak pamiętać, że nie będzie on możliwy do użycia przez inne PVC. Gdy natomiast ustawimy Delete, to po usunięciu PVC, Persistent Volume też będzie usunięty. Ostatnia opcja to Recycle, wtedy, po usunięciu PVC, Persistent Volume będzie wyczyszczony i przygotowany do ponownego użycia.

Po stworzeniu PVC, należy umieścić odnośnik do niego w pliku yaml poda, tak aby pod korzystał z tego volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
        - mountPath: "/var/www/html"
          name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

W ten sam sposób możemy umieścić PVC w ReplicaSecie i Deploymencie. Należy pamiętać, że jeśli PVC jest użyte przez poda, nie będzie można go usunąć.

Storage Class

Tworzymy Persistent Volume w cloudzie GCP:

```
pv-definition.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-voll
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 500Mi
  gcePersistentDisk:
    pdName: pd-disk
    fsType: ext4
```

Należy jednak pamiętać, że najpierw musimy mieć stworzony dysk na GCP:

```
gcloud beta compute disks create \
  --size 1GB
  --region us-east1
  pd-disk
```

Taki proces nazywamy Static Provisioning of Volumes.

Żeby taki proces zautomatyzować musimy użyć Storage Classes. Możemy tutaj ustawić tzw Provisioner np. Google Storage, który będzie mógł automatycznie stworzyć pożądanego dysku i podłączyć go do pody. Taki proces to Dynamic Provisioning of Volumes.

W tym przypadku musimy stworzyć plik yaml dla Storage Class (jako provisioner możemy też użyć innych dostawców np. AWS EBS, Azure Disk, Azure File, NFS, Flocker itp.):

```
sc-definition.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: google-storage
provisioner: kubernetes.io/gce-pd
```

Teraz należy zmodyfikować nasze PVC i dodać tam Storage Class:

pvc-definition.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: google-storage
  resources:
    requests:
      storage: 500Mi
```

Ten krok pozwoli nam zautomatyzowanie tworzenia dysku w chmurze i Persistent Volume w Kubernetesie.

W trakcie tworzenia Storage Class możemy też sprecyzować inne parametry, takie jak type (rodzaj dysku), replication-type itp. Parametry te są jednak różne dla różnych provisionerów.

sc-definition.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: google-storage
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard [ pd-standard | pd-ssd ]
  replication-type: none [ none | regional-pd ]
```

Inną przydatną opcją jest VolumeBindingMode (to nie jest parametr chyba). Jeśli ustawimy go na WaitForFirstConsumer to PVC będzie w stanie Pending dopóki nie będzie poda, który będzie chciał konsumować storage. Jeśli ustawimy to na Immediate, to niezależnie od tego powstanie dysk i PV gotowy dla jakiegokolwiek poda.