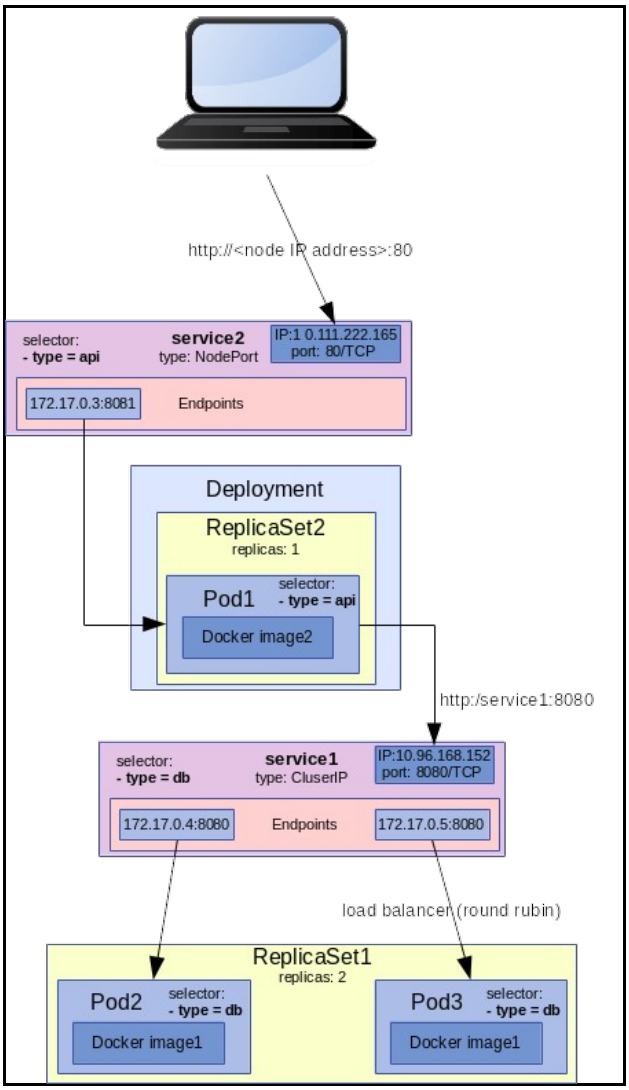
Kubernetes alapok

Contents

Kubernetes felépítése

Logikai épít?kockák

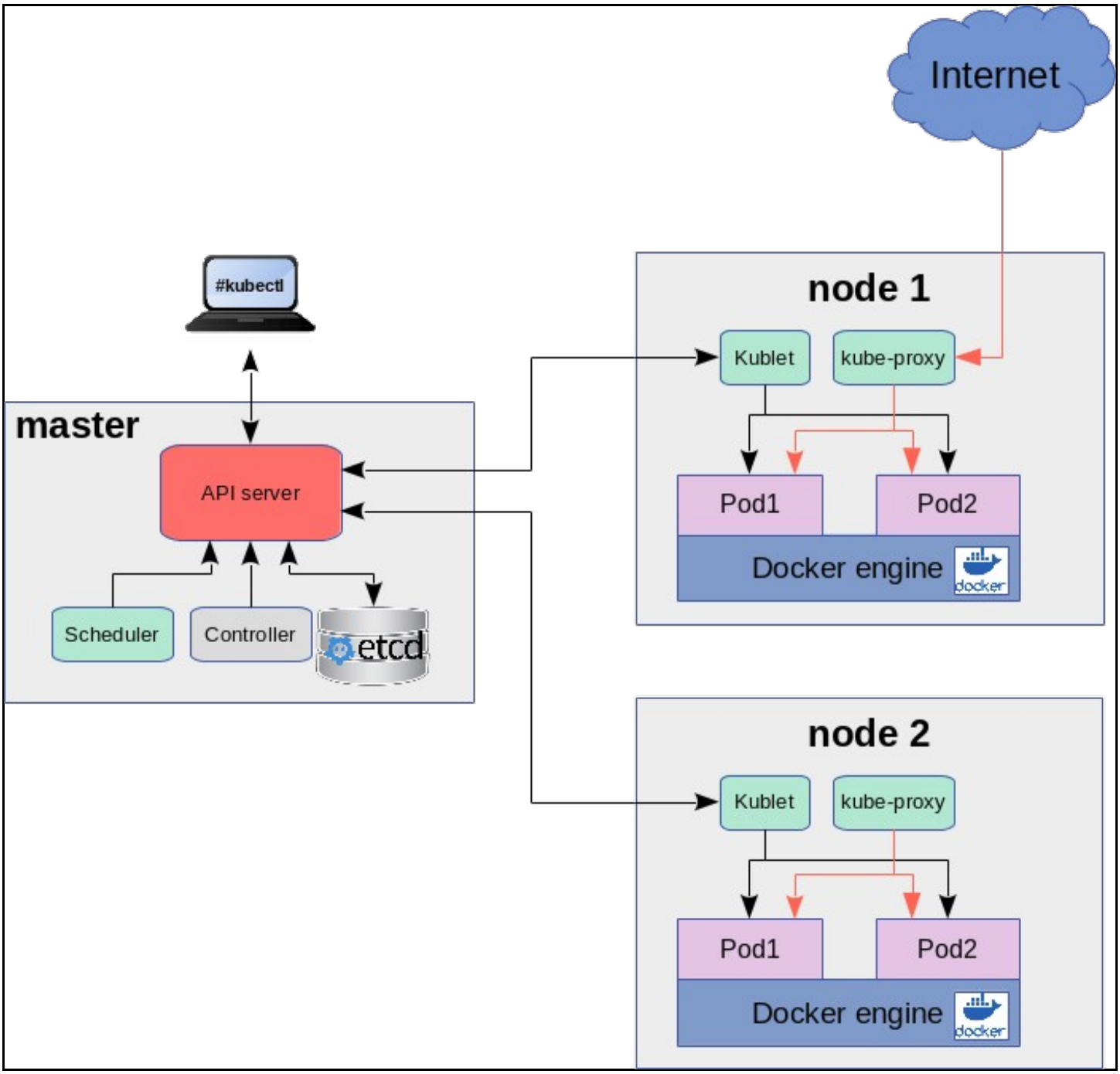
- Container:
- Pod:
- ReplicaSet:
- Deployment:
- Service:
- Endpoint:
- Namespace
- Network Policies

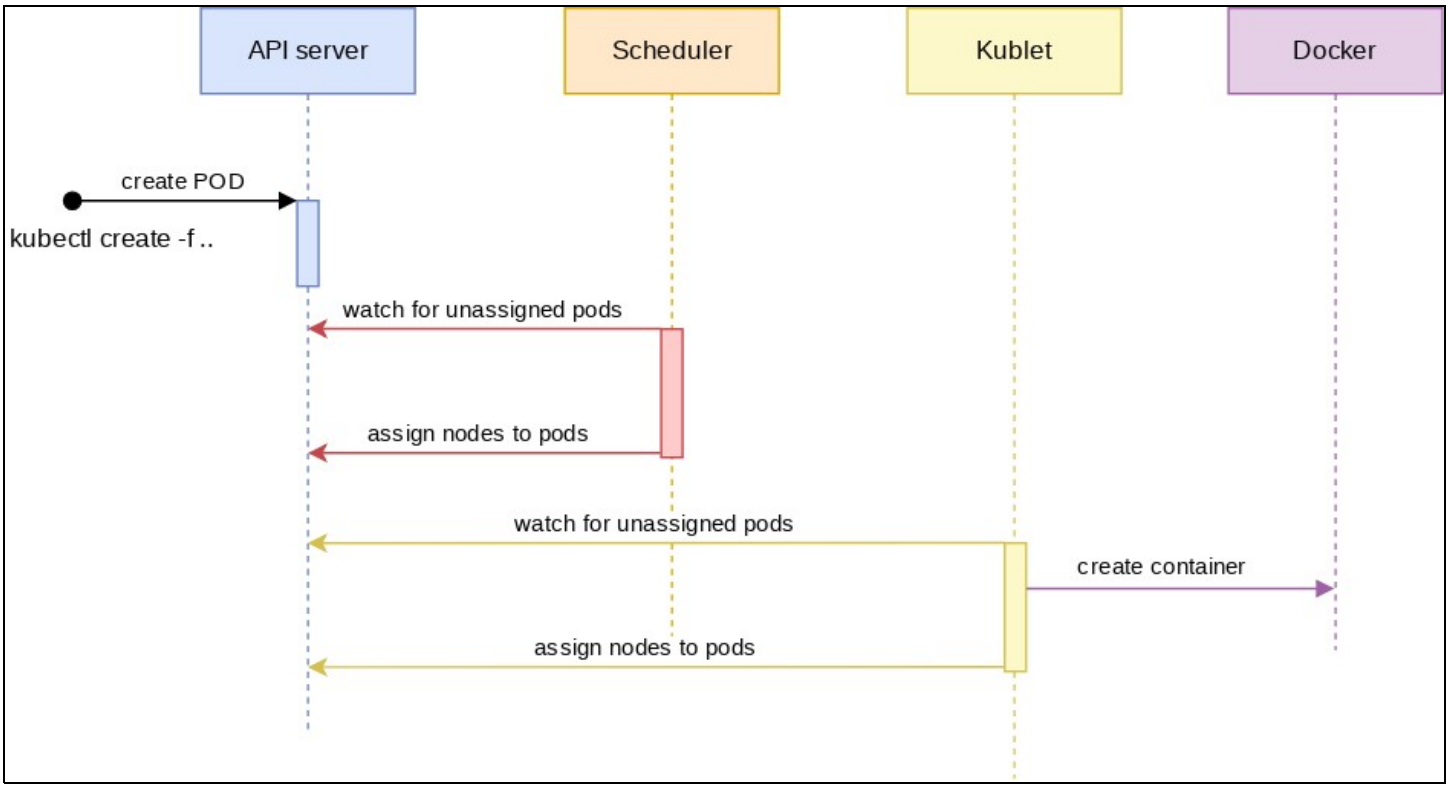


Namespace

https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

Infrastruktúra elemek


Internet
#kubectl
node 1
Kublet
kube-proxy
master
API server
Pod1
Pod2
Docker engine
docker
Scheduler
Controller
etcd
node 2
Kublet
kube-proxy
Pod1
Pod2
Docker engine
docker



Minikube Telepítés

kubectl

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stab
$ chmod +x ./kubectl
$ ln -s /home/adam/Programs/Kubernetes/kubectl /usr/local/bin/kubectl

$ kubectl version --output=yaml
```

Minikube

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 && chmod +x minikube
$ ln -s /home/adam/Programs/Kubernetes/minikube /usr/local/bin/minikube
```

kvm2 driver install

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/docker-machine-driver-kvm2 \
  && sudo install docker-machine-driver-kvm2 /usr/local/bin/

$ minikube version
```

Minikube indítása

```
$ minikube start --vm-driver=kvm2
Creating kvm VM (CPUs=2, Memory=2048MB, Disk=20000MB)
...
- docker
- kubernetes master
- kubernetes node: kublet
?  minikube v0.35.0 on linux (amd64)
? Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
```

Letölti a minikube.iso image-et a /home/adam/.minikube/cache/iso/ mappába, és innen létrehoz egy kvm vm-et:

```
# virsh list
 Id    Name                           State
----------------------------------------------------
 1     minikube                       running
```

Minikube parancsok

```
$ minikube dashboard
```

A minikube itt mutatja meg, hogy egy adott service hol érhet? el:

```
# minikube service <service-name> --url
```

Be lehet ssh-zni a vm-be mint a docker-machine ssh nál.

```
# minikube ssh  -
```

A minikube telepít? beállította a kubectl-t is, hogy a minikube-ban futó cluster-re csatlakozzon. Ha listázzuk a kubectl beállításait, láthatjuk a minikube VPN IP címét.

```
# kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /root/.minikube/ca.crt
    server: https://192.168.42.224:8443
  name: minikube
```

Nodes

Az összes node listázása.

```
# kubectl get nodes
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   11m   v1.13.4
```

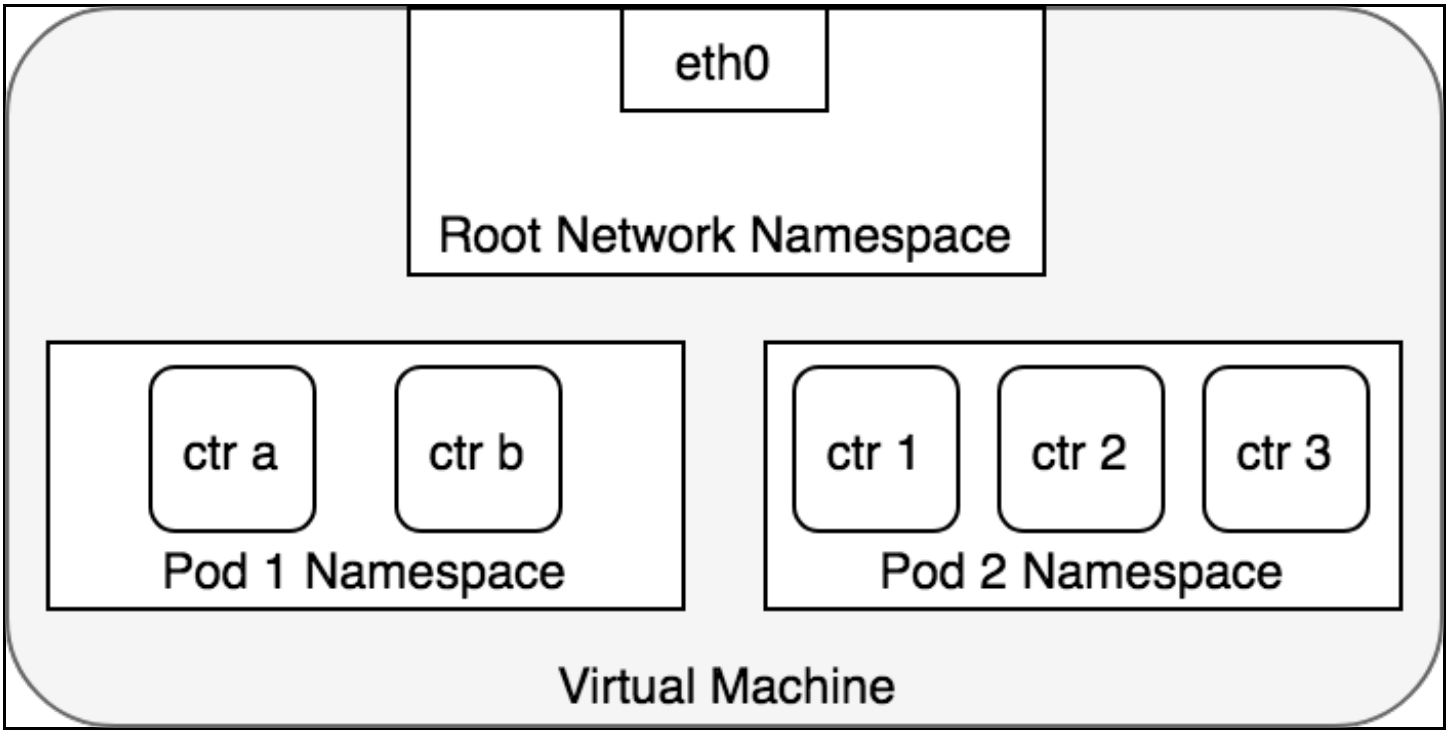
Megfelel a swarm-ban: docker node ls parancsnak.

Pod

A POD-ot úgy kell elképzelni, mint egy egyedi image futtató környezet, tehát nem pusztán egy plusz réteg a docker konténer körül. Ha messzir?l nézzük, akkor a pod, ha csak egy konténer van benne megfelel egy natív docker konténernek. A docker konténer a Linux virtualizációs eszközökkel teremet egy izolált Linux futtató környezetet az image-nek. Az izolációhoz eszközei:

- chroot: fájl rendszer izoláció
- chgroup: er?forrás izoláció (CPU, network, IO..)
- namespace: process izoláció, a névteren belül futó processzek egy saját sub-tree-t fognak csak látni az eredeti process fából, aminek úgy fogják látni, hogy a gyökere az 1-es process, ami a valóságban egy ága az eredeti process fának.

A POD ezen izolációs eszközökkel részben újra gombolja a natív docker konténert, részben meg meghagyja a natív docker funkciókat. Az egy POD-ban futó konténerek közös hálózati névtérbe kerülnek, vagyis osztoznak a hálózati er?forrásokon, viszont minden egyes konténer saját fájlrendszerrel rendelkezik, tehát a fájlrendszer izoláció megegyezik a natív docker konténer futtatással.



Ebb?l következik, hogy a localhost-on az egy POD-ba futó konténerek elérik egymást a megfelel? portokon, minden további hálózati beállítás nélkül tudnak a localhost-on kommunikálni egymással, viszont egymás fájlrendszerét nem tudják olvasni.

Az ipari standard szerint egy pod-ba csak egy konténert szokás rakni, így a valóságban általában elmosódik a POD és a natív docker konténer között a különbség, mivel a POD-ban futó egy szem konténer mind hálózati mind fájlrendszer szinten teljesen izolálva fut a külvilágtól.

Podok készítése

Imperative megközelítés

Konténert a legegyszer?bben a **kubectl run** paranccsal futtathatunk. Létre fog hozni egy pod-ot és benne el fogja indítani az image-et.

```
$ kubectl run db --image mongo
```

Deklaratív megközelítés

```
apiVersion: v1
kind: Pod
metadata:
  name: go-demo-2
  labels:
    app: myapp
spec:
  containers:
  - name: db
    image: mongo:3.3
```

```
# kubectl create -f pod-db.yaml
pod/go-demo-2 created

# kubectl get pod
NAME        READY   STATUS    RESTARTS   AGE
go-demo-2   1/1     Running   0          50s
```

Pod-ok kezelése

Pod-ok listázása

```
# kubectl get pods
NAME                  READY   STATUS              RESTARTS   AGE
db-7fdd878ff9-7v66g   0/1     ContainerCreating   0          66s
```

Belépés a pod-ban futó konténerekbe:

```
# kubectl exec -it <pod név> /bin/bash
```

Ha több konténer is van a pod-ban, akkor a -c vel meg kell adni a konténer nevét:

```
# kubectl exec -it -c db dbstack /bin/bash
```

Ha több konténer is van a pod-ban, akkor a pod neve után meg kell adni a konténer nevét.

```
# kubectl logs dbstack -c db
```

Minden kubernetes elem listázása:

```
# kubectl get all
NAME                      READY   STATUS    RESTARTS   AGE
pod/db-6b5c96c65f-9lxnb   1/1     Running   0          2m46s

NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/db   1/1     1            1           2m46s

NAME                            DESIRED   CURRENT   READY   AGE
replicaset.apps/db-6b5c96c65f   1         1         1       2m46s
```

Egy pod összes adatának listázása. Itt külön listában láthatjuk a POD-ban futó összes konténert. Látható, hogy az alábbi POD-ban két konténer is fut: **db** és **api** néven.

```
[root@adamDell2 ~]# kubectl describe pod dbstack
Name:               dbstack
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               minikube/192.168.122.228
Start Time:         Wed, 13 Mar 2019 21:51:19 +0100
Labels:             type=stack
Annotations:        <none>
Status:             Running
IP:                 172.17.0.5
Containers:
  db:
    Container ID:   docker://c371653e62b4bb1f8a7fb7de4d88452b6de623cf02caa41df9728df15d080481
    Image:          mongo:3.3
   ...
  api:
    Container ID:   docker://6b64c392a77347ce5e7b36ddfd80f2ef320a0e31d25985f813294d08cacf76b3
    Image:          vfarcic/go-demo-2
```

A **get** paranccsal is ugyan ezt a részletességet érhetjük el a **-o json** kapcsolóval.

```
# kubectl get pod go-demo-2 -o json
{
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
        "creationTimestamp": "2019-03-31T21:13:34Z",
```

```
        "labels": {
            "app": "myapp",
            "type": "example"
        },
        "name": "go-demo-2",
        "namespace": "default",
        "resourceVersion": "133741",
        "selfLink": "/api/v1/namespaces/default/pods/go-demo-2",
        "uid": "d7d61cac-53f9-11e9-bdbb-5254008eeeec"
    },
    "spec": {
        "containers": [
            {
                "image": "mongo:3.3",
                "imagePullPolicy": "IfNotPresent",
                "name": "db",
                "resources": {},
                "terminationMessagePath": "/dev/termination-log",
                "terminationMessagePolicy": "File",
                "volumeMounts": [
                    {
                        "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount",
                        "name": "default-token-zj4tp",
                        "readOnly": true
                    }
...
```

A **-f** -el meg lehet adni a leíró fájlt a describe parancsban. Ez minden Kubernetes elemre m?ködik. Ha nem adunk meg a formátumra paramétert, akkor egy szimpla felsorolást kapunk. A végén pedig egy esemény történet van az adott pod-ról.

```
# kubectl describe -f pod-db.yaml
Name:               db
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               minikube/192.168.122.228
Start Time:         Wed, 13 Mar 2019 20:49:30 +0100
Labels:             type=db
...
..
Events:
  Type    Reason          Age   From               Message
  ----    ------          ----  ----               -------
  Normal  Scheduled       21h   default-scheduler  Successfully assigned default/go-demo-2 to minikube
  Normal  Pulled          21h   kubelet, minikube  Container image "mongo:3.3" already present on machine
  Normal  Created         21h   kubelet, minikube  Created container
  Normal  Started         21h   kubelet, minikube  Started container
```

-o kapcsoló használata

Név	Leírás
-o custom-columns=<spec>	Print a table using a comma separated list of custom columns
-o custom-columns-file=<filename>	Print a table using the custom columns template in the <filename> file
-o json	Output a JSON formatted API object
-o jsonpath=<template>	Print the fields defined in a jsonpath expression
-o jsonpath-file=<filename>	Print the fields defined by the jsonpath expression in the <filename> file
-o name	Print only the resource name and nothing else
-o wide	Output in the plain-text format with any additional information, and for pods, the node name is included
-o yaml	Output a YAML formatted API object

A **-o json** kapcsolóval írhatjuk ki JSON -ban a pod infókat.

```
# kubectl get -f pod-db.yaml -o json
{
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
        "creationTimestamp": "2019-03-13T19:49:30Z",
        "labels": {
            "type": "db",
            "vendor": "MongoLabs"
        },
...
```

Részletes pod lista. A lényeg itt az IP cím.

```
# kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE     IP           NODE       NOMINATED NODE   READINESS GATES
db      1/1     Running   0          2m39s   172.17.0.2   minikube   <none>           <none>
```

els? pod a listáról:

```
# kubectl get pods -o name | tail -1
```

Sz?rés

Listázzuk ki JSON-ban a pod részleteket.

```
"status": {
        ...
```

```
        "hostIP": "192.168.122.228",
...
```

Sz?rni a **-o jsonpath** -al lehet.
Mindig úgy kell sz?rni, hogy a legfels?bb szint? elem elé teszünk egy .-ot. Majd megadjuk a path-t.

```
# kubectl get -f pod-db.yaml -o jsonpath="{.status.hostIP}"
192.168.122.228
```

Ha egy lista összes elemére akarjuk hogy illeszkedjen a keresés, akkor [*] -ot kell használni. Ha egy konkrét lista elemet akarunk, akkor azt adjuk meg így [1]
Pl. az összes konténer nevét így listázhatjuk:

```
# kubectl get -f pod-db.yaml -o jsonpath="{.spec.containers[*].name}"
db api
```

Címkék

A selector-ok címék alapján m?ködnek.

Címke hozzáadása egy node-hoz:

```
# kubectl get node
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   22d   v1.13.4

# kubectl label node minikube disktype=ssd
node/minikube labeled

# kubectl describe node minikube
Name:               minikube
Roles:              master
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    disktype=ssd
...

# kubectl get node --show-labels
NAME       STATUS   ROLES    AGE   VERSION   LABELS
minikube   Ready    master   22d   v1.13.4   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,disktype=ssd,..
```

Szintaxis:

```
# kubectl label <resource type: node, pod, ...> <resource név> <címke neve>=<címke értéke>
```

nodeSelector:

```
 labelName: labelvalue
```

Címke felrakása pod-ra:

```
# kubectl label pod go-demo-2 type=example
pod/go-demo-2 labeled

# kubectl get pod --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
go-demo-2   1/1     Running   1          23h   app=myapp,type=example
```

Címke törlése

```
# kubectl label pod go-demo-2 type-
pod/go-demo-2 labeled

# kubectl get pod --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
go-demo-2   1/1     Running   1          23h   app=myapp
```

Health check

- **readinessProbe**: ezt addig fogja futtatni amíg nem lesz egyszer sikeres, ez után fogja ready-re állítani a konténert.
- **livenessProbe**: Ezzel pedig a konténer egészségét fogja ellen?rizni

```
apiVersion: v1
kind: Pod
...
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    ...
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

Három változatuk van.

- command futtatása
- http rest hívás
- tcpSocket

Command

```
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
```

Http

- path
- port
- httpHeaders

```
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

Socket

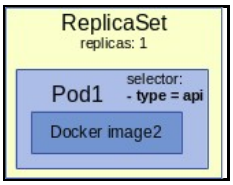
```
livenessProbe:
  tcpSocket:
    port: 8080
```

Név	Leírás
initialDelaySeconds	ennyi után fogja elkezdeni a próbálkozást
periodSeconds	Ilyen s?r?n fogja megpróbálni
timeoutSeconds	Ennyi id? után fogja feladni
failureTreshold	Ennyi negatív válasz után fogja feladni.

Második szint? kontrollerek

ReplicaSets

Áttekintés



Arra szolgál, hogy a benne definiált pod lehet?leg mindig annyi példányban fusson, amit defináltunk. Tehát ez továbbra is egy pod-ra vonatkozik, csak annak vezérelni tudjuk most már az életciklusát.

A ReplicaSet és a ReplicaController között a selector-ban van a különbség. A Controller-ben csak a = b-vel selector-okat lehet megadni, míg a Set-ben meg lehet adni összetett kifejezéseket is:

```
   selector:
     matchExpressions:
      - {key: app, operator: In, values: [soaktestrs, soaktestrs, soaktest]}
      - {key: teir, operator: NotIn, values: [production]}
```

A pod-hoz hasonlóan a dekleratív megközelítés szellemében, a replicaSet-et is yaml fájllal kell definiálni:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-rs
spec:
  replicas: 2
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
```

```
    containers:
    - name: my-nginx
      image: nginx
      ports:
      - containerPort: 80
```

A ReplicaSet leírásában 3 fontos rész van a spec szekción belül:

- **replicas**: megmondja, hogy hány példány kell hogy fusson a pod-ból
- **selector**: megmondja, hogy milyen címkékkel kell hogy a pod rendelkezzen, amire a replicaSet vonatkozik. Mivel a pod és a replicaSet csak lazán csatolt kapcsolatban van, a ReplicaSet csak annyit figyel, hogy minden fusson a replicas részben megadott számú pod, ami rendelkezik olyan címkékkel amiket a selector szekcióba megadtunk (itt tagadni is lehet)
- **template**: Itt definiáljuk a pod-ot. Ez a rész megegyezik a pod yaml fájlnál leírtakkal. A metadata részben adjuk meg a pod címkéit, a spec részbe pedig a konténereket.

Tip
Furcsa hogy ezt a két egymástól független entitást egy fájlban definiáljuk. Simán megcsinálhatjuk, hogy a tempalte szekcióban olyan pod-ot adunk meg, aminek nincs egy olyan címkéje sem, ami illeszkedne a selector részben megadott címkékre, így a replicaSet sosem tudna elindulni.

Selectorok és címkék

A ReplicaSet és majd a Service is a label-ek alapján találnak rá a Pod-okra. A selektorokonak két nagy családja van:

- egyenl?, nem egyenl? kifejezések: key=value illetve key!=value
- halmaz kifejezések: key in (v1, v2..), key notin (v1,v2..) illetve felírhatunk pusztán a kulcs létezésére ill nem létezésére feltételeket a kulcs értékét?l függetlenül:

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

A példában az utolsó két sor a kulcsra vonatkozik, hogy legyen olyan kulcs amit partition-nek hívnak ill ne legyen olyan kulcs amit partition-nek hívnak.

Egy kifejezésben vessz?vel lehet AND kapcsolatba hozni a kulcs kifejezéseket. A kubectl parancsokban a **-l** -el vagy hosszan **label** paraméterben kell megadni a kulcsokat:

```
# kubectl get pod -l app=myapp,type=example
NAME        READY   STATUS    RESTARTS   AGE
go-demo-2   1/1     Running   3          8d
```

Selector

A **Service** definíciójában illetve a ReplicaSet el?djében a **ReplicationController**-ben még a hagyományos **selector** megadást kell használni, a halmazos megadást itt nem szabad használni.

```
apiVersion: v1
kind: Service
metadata:
  name: db-svc
spec:
  type: ClusterIP
  ports:
    - port: 27017
  selector:
    type: db
    service: mongo-db
```

A selector-ok után fel kell sorolni a címéket, amik ÉS kapcsolatban vannak.

matchLabels

A **deployment**-ben ill. a **ReplicaSet**-ben már a **matchLabels** ill a vele ekvivalens **matchExpressions** -t kell a címke megadásnál használni. A matchLabels-ben meg lehet adni egyenl?ség és set alapú címke definíciókat is. Viszont a **selector** kulcsszó után kötelez? a **matchLabels** használata.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-db
spec:
  replicas: 1
  selector:
    matchLabels:
      type: db
      service: mongo-db
  template:
```

ReplicaSet kezelése

A **get** parancsban ReplicaSet esetén az **rs**-t kell megadni a **pod** helyett:

```
# kubectl get rs -o wide
NAME        DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES                          SELECTOR
go-demo-2   2         2         2       23s   db,api       mongo:3.3,vfarcic/go-demo-2   service=go-demo-2,type=backend
```

Ugyan úgy használhatjuk a **describe** parancsot is az rs-el mint a pod esetében:

```
# kubectl describe rs go-demo-2
Name:         go-demo-2
Namespace:    default
Selector:     service=go-demo-2,type=backend
Labels:       <none>
Annotations:  <none>
Replicas:     2 current / 2 desired
Pods Status:  2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  db=mongo
           language=go
           service=go-demo-2
           type=backend
  Containers:
   db:
    Image:        mongo:3.3
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
   api:
    Image:      vfarcic/go-demo-2
...
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  112s  replicaset-controller  Created pod: go-demo-2-6nq9h
  Normal  SuccessfulCreate  112s  replicaset-controller  Created pod: go-demo-2-5c2sk



# kubectl get pods --show-labels
NAME              READY   STATUS    RESTARTS   AGE     LABELS
go-demo-2-5c2sk   2/2     Running   0          5m41s   db=mongo,language=go,service=go-demo-2,type=backend
go-demo-2-6nq9h   2/2     Running   0          5m41s   db=mongo,language=go,service=go-demo-2,type=backend
```

A pod-ok és az RS között csak lazán csatolt kapcsolat van. Ha **--cascade=false** kapcsolóval töröljük az RS-t akkor a podokat meg fogja hagyni. És ha újra létrehozzuk az RS-t ezeket a pod-okat fogja felhasználni.

```
# kubectl delete -f rs/go-demo-2.yml --cascade=false
```

kubectl apply --> frissíti a konfigurációt. Rárak egy annotációt, és kés?bb az alapján dönti el, hogy mi válozott. Csak akkor lehet használni, ha eleve apply-al hoztuk létre, vagy create --save-confg kapcsolóval.

```
# kubectl create --save-config
```

kubectl edit: egyenl? azzal, mint ha describe -o yaml -el elmentenénk a konfigot, átírnánk, majd nyomnánk rá egy applay-t.

A -o yaml-el meg tudjuk szeretni az eredeti konfigurációs fájlt:

```
# kubectl get rs go-demo-2 -o yaml
```

ReplicationController

Legfels?bb szint? kontrollerek

Deployment

"zero-downtime deployment"

A Deployment egy legfels? szint? kubernetes vezérl?. A replicaSet-eket vezérli, nem közvetlenül a pod-okat. Deployment-el tudjuk frissíteni a futó pod-ok paramétereit, és ami a legfontosabb, hogy ezt úgy tehetjük meg (pl. átállni egy új image-re), hogy közben nem áll le a szolgáltatás, egyenként le tudja cserélni a replicaSet-hez tartozó pod-okat. Ezen felül rollBack funkciót is biztosít, a deployment-ben meghatározott számú vissza-állást el tudunk végezni (default 10).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  revisionHistoryLimit:5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge:1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
```

```
        - containerPort: 80
```

- apply
- create --save-config

A **--record** hatására a telepítéskor kiadott parancsot, és az eredeti leírófájlt be JSON formátumba be fogja tenni az **annotations** mez?be:

```
# kubectl get deployment nginx-deployment -o yaml
..
metadata:
  annotations:
    deployment.kubernetes.io/revision: "2"
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{"kubernetes.io/change-cause":"kubectl apply --filename=deploymen
    kubernetes.io/change-cause: kubectl apply --filename=deployment-nginx.yaml --record=true
  creationTimestamp: "2019-07-03T20:19:05Z"
```

https://www.mirantis.com/blog/kubernetes-replication-controller-replica-set-and-deployments-understanding-replication-options/

save/ apply ...

kubectl rollout status deploy

DeploymentConfig

```
.. OpenShift specifikus komonens, ahogy én értem, ez még azel?tt jött létre, hogy lett volna Deployment a Kubernetes-ben, ezért nem is az úja
```

- deploymentconfig ? replicationcontroller ? pod
- deployment ? replicaset ? pod

Statefulset

...

DaemonSet

...

Networking

https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/

Pod szint? kommunikáció

A pod hálózati szempontból nem egy plusz réteg a konténer körül, nem úgy kell elképzelni, mint ha egy konténerben futtatnánk egy másik konténert, sokkal inkább úgy, hogy a POD egy interfésze a konténernek, újra csomagolja a docker konténert és csinál bel?le egy "kubernetes konténert". A POD IP címe megegyezik a POD belsejében futó konténer IP címével, amit a Kubernetes oszt ki véletlenszer?en minden egyes POD-nak. A POD-nak akkor is csak egy IP címe van, ha több konténer fut benne. ... TODO rajz...

TODO: Alapértelmezetten minden pod minden er?forrással kommunikál. Ezt a network policy-val lehet sz?kíteni. TODO: namespace: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

Konténerek egy pod-ban

... TODO rajz...

Ha egy POD belsejében több konténer van, akkor azok osztoznak a hálózati névtérben, tehát ez nem egyenl? azzal, mint ha a lokális gépünkön két konténert elindítanánk egymás mellett, mert ott mind a két konténernek saját hálózati névtere lenne, és alap esetben nem tudnak egymással kommunikálni. Tehát ebb?l is látszik, hogy a POD a konténer újra csomagolása a hagyományos docker-hez képest, vagyis ha több konténer van egy POD-ban, akkor az olyan mint ha a két image-et egy konténerben "csomagoltuk" volna. A közös POD-ban futó konténerek a localhost-on úgy látják egymást, de ugyan azokon a portokon nem hallgatózhat két konténer, mert akkor port ütközés lesz. A POD tudni fogja a közös névtér ellenére, hogy melyik konténer melyik porton figyel, és a küls? kéréseket a megfelel? konténernek fogja továbbítani.

```
apiVersion: v1
kind: Pod
metadata:
  name: go-demo-2
  labels:
    app: myapp
spec:
  containers:
   - name: my-nginx
     image: nginx
     ports:
      - containerPort: 80
   - name: nettest
     image: amouat/network-utils
     command: [ "bin/bash" ]
```

```
    args: ["-c", "while true; do echo hello; sleep 10;done"]
```

A network-utils image-b?l készül? konténernek végtelen ciklusban adtunk munkát, hogyne álljon le, ahogy létrejön.

```
# kubectl create -f pod-db.yaml
pod/go-demo-2 created
```

Lépjünk be a nettsest konténerbe:

```
# kubectl exec -it -c nettest go-demo-2 /bin/sh
#
```

Majd a localhost-on nézzük meg mit találunk a 80-as porton:

```
# curl localhost:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

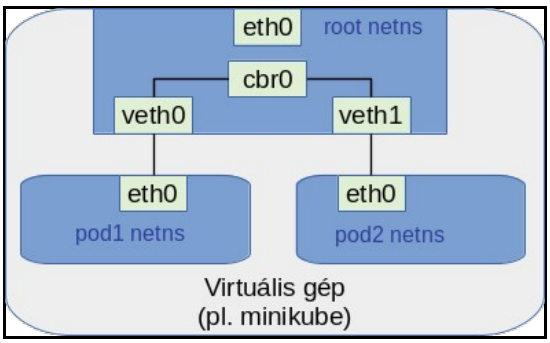
Láthatjuk, hogy annak ellenére választ kapott, hogy valójában a másik konténer hallgatózik a 80-as porton.

Pod-ok közötti kommunikáció

Egy Kubernetes cluster-ben minden pod automatikusan tud minden másik POD-al kommunikálni a POD IP címén keresztül. Ez igaz a node-okra is. Minden node elér minden POD-ot és minden POD eléri az összes node-ot a node IP címén annak ellenére, hogy a pod-ok és a node-ok más hálózatban vannak.

Note
Ez a koncepció megfelel a **docker swarm** overlay hálózatának, amit a swarm-ban implicit definiálni kell, két swarm service csak akkor látja egymást, ha ugyan azon az overlay hálózaton vannak. (swarm-ban a service egy replicaSet-nek felel meg, nem egyenl? a Kubernetes service objektummal)

Linux-ban a névterekben lév? virtuális hálózatokat össze lehet kötni a host root hálózatával úgynevezett **Veth** (Virtual Ethernet Device) hálózati csatolókkal. Minden pod és a host root hálózata között létrejön egy Veth pár. A Veth párok pedig hálózati híddal (bridge) vannak összekötve a root hálózatban. Ezeket a hálózati beállításokat automatikusan létrehozza a Kubernetes egy cluster-en belül. Ezért éri el minden pod a host gép hálózati interfészeit és viszont.



A veth interfészek a root hálózati névtérben Linux Ethernet bridge-el vannak összekötve (Layer 2 kapcsolat). Ezért tud minden pod az összes többi pod-al kommunikálni. A Pod-to-pod kommunikáció node-ok között is m?ködik, de az már hálózatfügg? hogy pontosan hogyan van megvalósítva. Pl. AWS-en erre több lehet?ség is van.

Hozzunk létre egy nginx pod-ot a kubectl run paranccsal (A run parancs a pod-on kívül létre fog hozni egy ReplicaSet-et és egy Deploymenet-et is.).

```
# kubectl run webserver --image nginx
deployment.apps/webserver created
```

Keressük meg az IP címét:

```
# kubectl get pod -o wide
NAME                         READY   STATUS    RESTARTS   AGE    IP           NODE       NOMINATED NODE   READINESS GATES
webserver-786f555565-gv4m6   1/1     Running   0          103s   172.17.0.7   minikube   <none>           <none>
```

Hozzunk létre szintén a run paranccsal egy pod-ot amiben a network-utils hálózat tesztel? konténer fog futni. Ezt interaktív üzemmódban fogjuk elindítani, ahogy létrejön a pod a konténer belsejében futó sh shell-hez fogunk kapcsolódni. Ehhez a kubernetes itt is alapértelmezetten létre fog hozni egy deployment-et és egy replicaSet-et. De ez most nem fontos.

```
$ kubectl run -it netpod --image amouat/network-utils -- sh
#
```

Keressük meg a pod IP címét.

```
# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:06
          inet addr:172.17.0.6
```

Most a netpod belsejéb?l kérjük le a webserver pod index.html oldalát:

```
# curl 172.17.0.7
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

Láthatjuk hogy a két pod valóban tud egymással kommunikálni.

Most ssh-val lépjünk be a minikube node-ra majd onnan szintén kérjük le ezt az oldalt:

```
# minikube ssh
                         _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___   (_) | |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| |/ , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)

$

$ curl 172.17.0.7
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

Látható hogy a node is elérte a pod-ban futó konténert.

Network policy

https://kubernetes.io/docs/concepts/services-networking/network-policies/

A network policy segítségével lehet korlátozni a pod-ok közötti kommunikációt, ami alapesetben minden pod között adott.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

DNS feloldás

A service nevére minden POD-ban m?ködik a DNS feloldás. A pod nevére ugyan ez már nem m?ködik.

Korábban létrehoztunk egy **http** nev? service-t, ami a 80 portot nyitja ki a run=webserver címkével rendelkez? pod felé:

```
# kubectl get svc
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
http            NodePort    10.109.91.148   <none>        80:31997/TCP     29h
```

Indítsuk el interaktív üzemmódban a korábban már használt network util csomagot tartalmazó konténert:

```
# kubectl run -it netpod --image amouat/network-utils -- sh
#
```

Majd futtassunk egy dig-et a http nev? service-re. Vagy akár curl-al kérjük el a http:80-as porton elérhet? tartalmat:

```
# nslookup http
Server:         10.96.0.10
Address:        10.96.0.10#53

Name:   http.default.svc.cluster.local
Address: 10.109.91.148

# curl http
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

Service

Bevezet?

Láthattuk, hogy egy Kubernetes cluster-en belül az összes pod tud kommunikálni egymással minden további hálózati komponens beiktatása nélkül, amennyiben tudják egymás IP címét. Ez a kommunikáció még akkor is lehetséges, ha a pod-ok külön node-on vannak. Azonban a POD IP címét korántsem tekinthetjük állandónak. A pod ahányszor újra létrejön egy ReplicaSet-ben mindig más és más dinamikus IP címet fog kapni a Kubernetes-t?l, ezért a POD IP címére nem igazán lehet építeni. Ráadásul, ha egy ReplicaSet-hez több POD példány is tartozik, akkor azt szeretnénk, hogy a ReplicaSet pod-jai load-balanceolva kapják meg a csomagokat, egyáltalán nem fontos, hogy pontosan melyik POD szolgálja ki a kérést az identikits pod-ok közül. A fenti két problémára szolgál megoldással a service. Minden service kap egy virtuális IP címet mikor létrejön, ami a service élete során már nem változik. A service-ekhez pod-okat lehet rendelni. A service-hez rendelt pod-ok elérhet?k a service virtuális IP címén keresztül, a service elfedi el?lünk a POD dinamikusan változó IP címét, hiába változik meg a POD címe, azt a service leköveti, és meg fogja találni a megfelel? pod-ot. Ha egy service-hez több POD is tartozik, akkor azokhoz load-balance-olva fogja eljuttatni a kéréseket. A load-balancing egy serivce-ben kétféle Linux komponenssel is végrehajtható, ezt a service definiálásakor kell megadni:

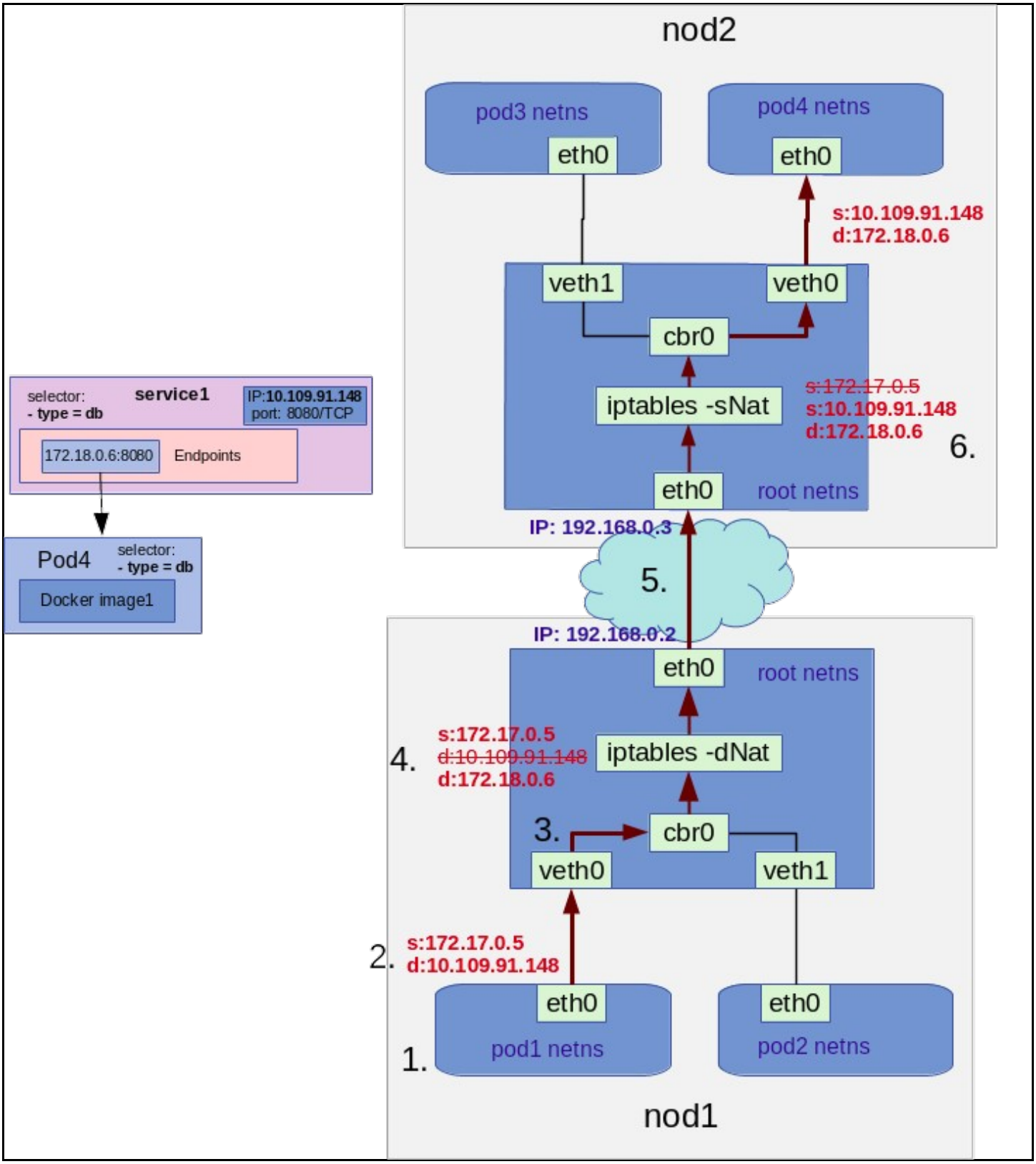
- iptables:
- IPVS (IP Virtual Server): szintén része a Linux kernelnek

Ha egy service mögötti pod kommunikálni akar a külvilággal, akkor az üzeneteket a service-nek fogja elküldeni, aki a destination IP-t ki fogja cserélni annak a POD-nak az IP címére, akit eredetileg ...

- Pod to Service: Ha egy RepliaSet pod-jaihoz van service definiálva, akkor a pod-ok úgy látják a teljes kommunikáció alatt, hogy ?k a service virtuális interfészével kommunikálnak.
- Service to Pod:

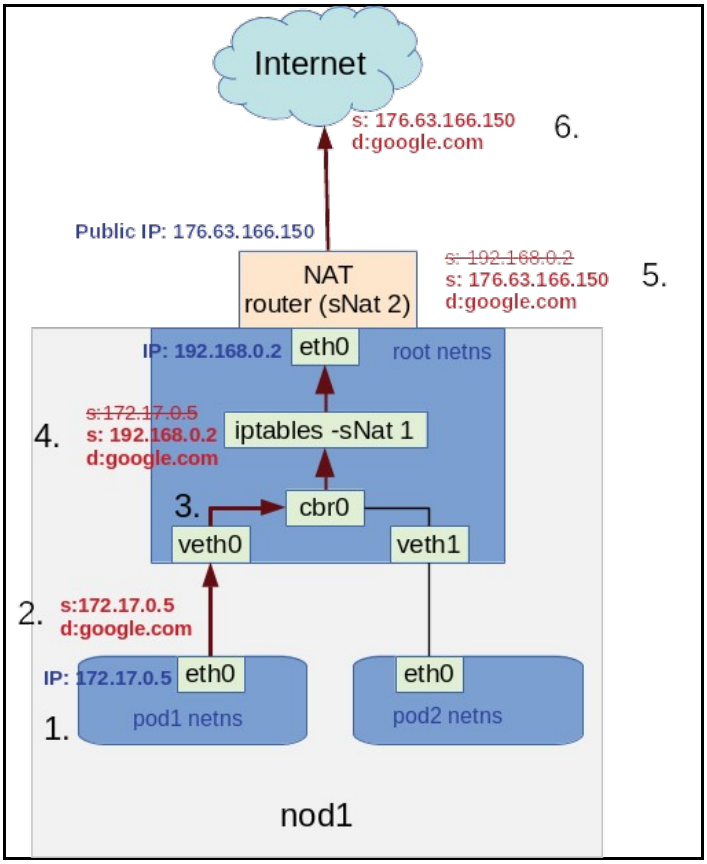
Pod to Service kommunikáció

Egy konkrét példán keresztül fogjuk bemutatni, hogy egy pod hogyan tud üzenetet küldeni egy service-nek ami továbbküldi a csomagot a hozzá rendelt pod-nak. Tételezzük föl, hogy létezik egy **service1** nev? szolgáltatás, amire a **pod4** nev? pod csatlakozik (selectorokon keresztül). A **service1** IP virtuális IP címe **10.109.91.148**. A mögötte található POD IP címe: **172.18.0.6**. A példában a **pod1** nev? pod (IP: 172.17.0.5) fog csomagot küldeni a **service1**-nek. A **pod1** úgy látja, hogy ? kizárólag a **service1**-el kommunikál. De a pod4, aki végül meg fogja kapni a csomagokat, ? is úgy fogja látni, hogy kizárólag a service1-t?l kap csomagokat, csak vele áll kapcsolatban.



1. A pod1 csomagot akar küldeni a service1-nek. Ebben a forrás IP a POD1 saját címe lesz, a cél IP pedig a service1 virtuális IP címe, ami nem változik.
2. A POD-ban lév? eth0 interfész fogja megkapni a csomagot. Mivel nem neki szól továbbítani fogja a Virtuális Ethernet eszköz párjának, ami már a node root névterében van.
3. A **cbr0** hálózati híd fogja megkapni a csomagot. Mivel a cél cím (a **service1** címe) nem neki szól (ARP protokoll) ezért továbbítani fogja a csomagot a root névtér **eth0** alapértelmezett átjárója felé mivel ? nem tud a service létezésér?l.
4. Az **eth0** nem kapja meg azonnal a csomagot. A service létrehozásakor a kube-proxy létrehozott egy iptables filtert a **node1**-ben. Itt az IPtables szabály hatására a cél IP, ami eddig a service1 virtuális IP címe volt le lesz cserélve a **service1** szolgáltatáshoz tartozó **pod4**-es pod IP címére dNat szabályokkal, majd a csomag továbbítva lesz az **eth0**-nak.
5. Az eth0 a megfelel? virtuális hálózaton tudni fogja, hogy melyik pod melyik node-on található, így továbbítani fogja a csomagot a **node2**-re.
6. A node2-ön az eth0 interfész továbbítani fogja a csomagot az ottani **cbr0** hídnak. Azonban a kube-proxy itt is létrehozott egy IpTables szabályt. Az itten szabály ezúttal a source címet fogja lecserélni a **service1** szolgáltatás virtuális IP címére sNat szabályokkal, hogy a **pod4** is úgy lássa, hogy ? kizárólag a **serve1** szolgáltatással kommunikál.

Pod to Internet kommunikáció



- Egress ? pod-ok "publikus" internet elérése
- Ingress ? pod-ok elérése az internetr?l

Service definiálása

A **containerPort** csak informatív jelleg? a konténer definícióban. Ett?l függetlenül bármilyen porot-ot expose-álni lehet kés?bb, olyat is ami itt nincs megadva. Ha az expose parancsban nem adunk meg **--target**-port-ot akkor a **-containerPort**-al megadott portot fogja megosztani.

```
    containers:
    - name: my-nginx
      image: nginx
      ports:
      - containerPort: 80
```

Az expose paranccsal lehet egy Pod vagy ReplicSet vagy Deployment-hez service-t készíteni implicit (tehát nem dekleratív módon). Meg kell mondani, hogy mib?l indulunk ki. Mi most egy replicaSet-b?l, ezért az rs után megadtuk a ReplicaSet nevét. Utána meg kell adni a service nevét, a portot, amit ki akarunk nyitni, valamint a típust.

```
# kubectl expose rs nginx-rs --name=http --port=80 --type=NodePort --target-port=80
```

Típusok:

- NodePort: a külvilágból is elérhet? lesz, minden egyes node-on --> clientIP típust is létrehoz, így a többi node is eléri ezt automatikusan
- ClusterIP: csak a cluster-en belülr?l
- ExternalName: ehhez cloud szolgáltató kell, pl aws
- LoadBalancer

Háromféle port érdekes a **NodePort** típusú service esetetén, amib?l az expose paranccsal csak kett?t lehet megadni:

- port: ezen a porton érhet? el a service a cluster-en belül. Ennek nincs köze ahhoz hogy a cluster-n kivül hol érhet? el a szolgáltatás. Ha nem adjuk meg, akkor ugyan az lesz mint a target-port
- target-port: ez azt mondja meg, hogy a pod-on/container-en belül hol hallgatózik a service. Ha nem adjuk meg külön, akkor a -containerPort -ból szedni.
- nodePort: na ezt nem lehet itt megadni. Azt mondja meg, hogy a cluster-en kívül hol lesz elérhet? a szolgálatás. Ha nem adjuk meg akkor random választ.

```
# kubectl get svc http -o wide
NAME    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE    SELECTOR
http    NodePort    10.98.187.168   <none>        80:32730/TCP   8m3s   run=my-nginx
```

Az itt mutatott IP cím a bels? virtuális cím ami csak a Kubernetes cluster-en belül érvényes. A konténernek sincs ilyen 10.X.X.X-es tartományú IP címe, ez egy Kubernetes bels? dolog.

Láthatjuk, hogy a service bels? IP címe 80 (ezt adtuk meg a --port -nál, bármi lehetett volna, szabadon választható) és ehhez rendelte hozzá a küls? portot: 32730 Ez a port az összes node publikus IP -én elérhet?.

```
# kubectl get node -o wide
NAME       STATUS   ROLES    AGE    VERSION   INTERNAL-IP       EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION   CONTAINER-RUNTIME
minikube   Ready    master   7d1h   v1.13.4   192.168.122.228   <none>        Buildroot 2018.05   4.15.0           docker://18.6.2
                                              ^^^^^^
```

Az expose autómatikusan legyártja a service definíciót a ReplicaSet-b?l, amib?l kiindultunk. A végerdmény az alábbi:

```
# kubectl get svc http -o yaml
apiVersion: v1
kind: Service
metadata:
  name: http
spec:
  ports:
  - nodePort: 32730
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: my-nginx
  type: NodePort
```

A service és a ReplicaSet semmilyen kapcsolatban nincsenek egymással. Mind a kett? a selector-okkal találják meg a POD-okat.

A pod-ok úgynevezett Endpoint-okon keresztül ajálják ki a portokat a service-ek számára. Rövidítése ep. Tehát itt a 80-as port megegyezik a target-port-nál megadott értékkel.

1. kubectl get ep

NAME ENDPOINTS AGE http 172.17.0.7:80 23m

A service-ek neve bekerül a DNS-be, és feloldja a Kubernetes a konténerek számára a bels? virtuális IP címmel. Ha van egy ClusterIP típusú service-ünk a db-svc névvel, akkor a cluster-ben minden egyes konténerben ezt fel fogja oldani a service IP címére: 10.111.222.165. A service meg a felsorolt Enpoints-okon éri el a konténert.

```
# kubectl describe svc db-svc
Name:              db-svc
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          service=mongo-db,type=db
Type:              ClusterIP
IP:                10.111.222.165
Port:              <unset>  27017/TCP
TargetPort:        27017/TCP
Endpoints:         172.17.0.2:27017
```

A service round-rubin módon választ a végpontok között. De csak azoknak küld csomagot, amiben a livenessProbe. sikeresen futott le. Korábban userpase-el ment a kommunikácó, most már iptabels szabályokkal, ami sokkal gyorsabb, viszont nem tudja deketálni ha egy pod kiesett. Erre kell a livenessProbe

Lépjünk be egy harmadik konténerbe, amire nincs rákötve a service:

1. kubectl exec -it rs-db-api-m6hkf /bin/sh

/ # nslookup db-svc Name: db-svc Address 1: 10.111.222.165 db-svc.default.svc.cluster.local

/ # nc -vz db-svc 27017 db-svc (10.111.222.165:27017) open

Publikus port lekrédezése: PORT=$(kubectl get svc svc-api -o jsonpath="{.spec.ports[0].nodePort}"

Tehát nem kell a -o json ahhoz hogy tudjunk sz?rni.

Nem muszáj hogy selector-a legyen egy szolgáltatásnak. Készíthetünk serlector nélküli service-t, ami automatikusan egyik pod-ra sem fog ezért rákerülni, majd készíthetünk hozzá manuálisan egy endpoint-ot:

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

Endpoints

Az endpoint-ok teremtenek kapcsolatot a service és a hozzá tartozó pod példányok között. Minden egyes pod-hoz ami illeszkedik a service selector-ára a service létrehozásakor automatikusan létrejön egy endpoint. Az endpoint tartalmazza a POD IP címét amivel a service-hez csatlakozik. Ez az IP cím a konténerben lév? eth0 interfész IP címe. Egy service-hez legalább egy darab Enpoints Kubernetes objektum tartozik, ami leírja az összes végpontot / pod. Endpoints objektumot manuálisan is létre lehet hozni vagy módosítani, de tipikusan ezt a service kezeli.

Ha megnézzük az ep definícióját, akkor ott fel van sorolva az összes pod aki részt vesz a service-ben. Látható, hogy két pod van jelenleg . Az is ki van listázva, hogy mi a bels? IP címük és hogy melyik node-on vannak.

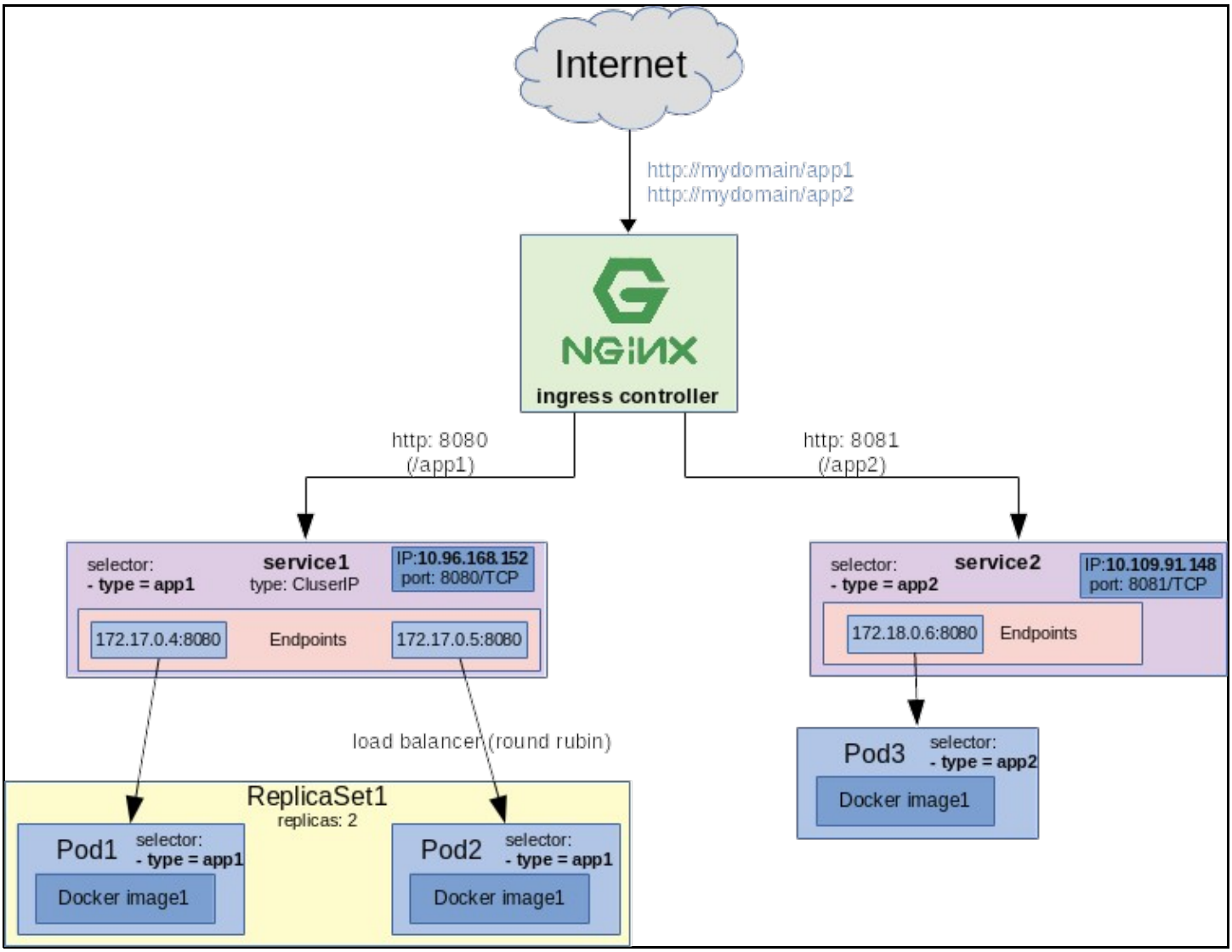
```
# kubectl get ep http -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  creationTimestamp: "2019-03-17T19:49:12Z"
  name: http
  namespace: default
  resourceVersion: "84822"
  selfLink: /api/v1/namespaces/default/endpoints/http
  uid: bd23a709-48ed-11e9-8d39-5254008eeeec
subsets:
- addresses:
  - ip: 172.17.0.7
    nodeName: minikube
    targetRef:
      kind: Pod
      name: nginx-rs-wtzmm
      namespace: default
      resourceVersion: "73593"
      uid: 1820549d-48d9-11e9-8d39-5254008eeeec
  notReadyAddresses:
  - ip: 172.17.0.8
    nodeName: minikube
    targetRef:
      kind: Pod
      name: nginx-rs-jkvx7
      namespace: default
      resourceVersion: "84820"
      uid: 1c547797-48ee-11e9-8d39-5254008eeeec
  ports:
  - port: 80

...
```

Ingress

A service-ekkel önmagukban az a baj, hogy egy Kubernetes klaszterben egy porton csak egy service hallgatózhat. nodePort esetén ezen a porton bármelyik node-ról elérhet? a service. Ahhoz egy ugyanazon a porton több szolgáltatást is elérhessünk a klaszeren kívülr?l szükségünk van az úgynevezett **ingress controller**-e. Az ingress komponens végezhet SSL terminálást, név alapú virtuális hosting-ot, és load-balanc-olást. Tipikusan HTTP és HTTPS protokollokat használ. Az ingress beékel?dik a külvilág és Kubernetes servcie-ek közé. Az ingress objektum nem része a Kubernetes-nek, a Kubernetes csak egy API-t biztosít. Számtalan ingress implementáció érhet? el, pl:

- HAproxy
- nginx
- Traefik (https://docs.traefik.io/user-guide/kubernetes/) ami Docker swarm-hoz is biztosít Layer 7-es sticky session load balancer-t.

Note
Swarm -ban Ingress-nek egy kicsit mást hívnak. Swarm-ban két f? hálózati típus van. Az egyik az **overly** hálózat, ahol az overly hálózatra kapcsolódó konténerek (swarm service) elérik egymást. A másik az **Ingress** hálózat, ahova publikálhatunk swarm szolgáltatásokat megadott portokon. A swarm klaszter bármelyik node-ján a megadott porton elérhet? lesz a szolgáltatás. Ha több konténer is van a szolgáltatásban, akkor az ingress load-balance-olni fog. Tehát a Kubernetes service keveréke a swarm overlay és swarm ingress hálózatnak, de a Kubernetes ingress komponensnek semmi köze a swarm ingress hálózathoz.



Az ingress kontroller is egy pod-ként fog futni a Kubernetes klaszterben. Ha több node is van a klaszerben, akkor minden egyes node-ra fell kell telepíteni. Ugyan pod-ként fut, de az ingress egy sepciális pod a **kube-system** névtérben (tehát a natúr get pod parancs nem listázza ki). Az ingress pod az ingress implementáció telepítése közben jön létre. Létre jön minden egyes node-on (virtuális gépen) egy külön ethernet interfész az ingress számára:

```
eth0      Link encap:Ethernet  HWaddr 52:54:00:8E:EE:EC
          inet addr:192.168.122.228  Bcast:192.168.122.255  Mask:255.255.255.0

eth1      Link encap:Ethernet  HWaddr 52:54:00:54:0F:3C
          inet addr:192.168.42.224  Bcast:192.168.42.255  Mask:255.255.255.0
```

Az els? interfész lett az ingress számára létrehozva. A **minikube ip** parancs a második (.224) IP címet fogja visszaadni, míg az ingress példányokhoz az els? IP cím (.228) lesz feltüntetve. Az ingress kontroller kizárólag a saját interfészén fog hallgatózni, jelen esetbe a 228-on.

Installálás minikube-ba

```
# minikube addons list
...
- ingress: disabled
...
```

```
# minikube addons enable ingress
?  ingress was successfully enabled
```

Ahhoz hogy listázni tudjuk az nginx pud-ját meg kell adni implicit a -n kapcsolóval a **kube-system** névteret. Alap esetben minden a default névtérbe kerül és ha külön nem adjuk meg, akkor mindig a default tartalmát listázza.

```
# kubectl get pods -n kube-system
NAME                                    READY   STATUS              RESTARTS   AGE
...
nginx-ingress-controller-7c66d668b-wnqsk   0/1     ContainerCreating   0          3s
```

Ingress definiálása

Az Ingress objektum formátuma az alábbi. Bármilyen ingress implementációt is választunk, a dekleratív leíró fájl szabványos kell legyen. Az egyetlen olyan elem, ahol az implementáció specifikus paramétereket adhatjuk meg, az a **annotations** szekció. Ha az adott implementációnak vannak egyedi beállításai, akkor azokat itt kell megadni. Láthatjuk hogy itt két nginx-es beállítást is használtunk.

Az ingress kontroller http path és http hostnév alapján is tud route-olni vagy a kett?t együtt is használhatjuk. Az alábbi példában nem adtuk meg hostnevet. A -http -> paths listában kell felsorolni az URL path-okat, amire az adott service-t kötni szeretnénk. Majd a backend szekcióban meg kell adni az adott service kubernetes nevét és bels? portájt. Vagyis nodePort típusú service esetén nem azt a portot kell megadni, ahol a node-on el lehet érni, és nem is azt a portot ahol a pod hallgatózik, hanem a service saját portját. Nekünk van egy http nev? service-ünk, ami a 80-as porton hallgatózik. A node-on kívülr?l a 31997-es porton érhet? el. Az ingress kontrollerben a /app1-es path-t erre a service-re akarjuk rákötni. A http service-ünkhöz egy nginx-es image-et tartalmazó pod van kötve.

```
# kubectl get svc
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
http             NodePort    10.109.91.148   <none>        80:31997/TCP     5d6h
```

Warning
Az ingress kontroller ugyan azt a path-t fogja továbbítani az alkalmazásnak, amit megadunk a routingban. Tehát nem fogja automatikusan a / (root)-ra irányítani a kérést. A lenti példában tehát az nginx szerver a /app1-et kapná meg, amilyen URL alapértelmezetten nem létezik.

A "**nginx.ingress.kubernetes.io/rewrite-target:** /" paraméterrel mondhatjuk meg, hogy minden alkalmazáshoz a /-ra irányítsa a kérést a path paraméterben megadott érték helyett.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-inggress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /app1
        backend:
          serviceName: http
          servicePort: 31997
```

```
# kubectl create -f demo-ingress.yml
ingress.extensions/demo-inggress created
```

```
# kubectl get ingress
NAME            HOSTS   ADDRESS           PORTS   AGE
demo-inggress   *       192.168.122.228   80      6m59s
```

- path alapú routolás
- domain név alapú routolás
- default rout:

```
curl -I -H "Host: mydomain.com" \
"http://http://192.168.122.224/app2/"
```

Ingress konfigurációs fájl

Az ingress kontroller figyeli az API szerveren történt változásokat. Ha új ingress objektumot jegyez be ott a kliens, akkor átkonfigurálja a load-balancer-t jelen esetben az nginx szervert. A változásokat beírja a **/etc/nginx/nginx.conf** fájlba.

Ha belépünk az nginx pod-ba interaktív módban (bash-t futtatva) akkor megnézhetjük a fenti ingress definíció hatására létrejött konfigurációt.

```
# kubectl exec -it -n kube-system nginx-ingress-controller-7c66d668b-wnqsk /bin/sh
$
$ cat nginx.conf

...
        ## start server _
        server {
                server_name _ ;

                listen 80 default_server reuseport backlog=511;
```

```
location ~* "^/app1\/?(?<baseuri>.*)" {

        set $namespace      "default";
        set $ingress_name   "demo-inggress";
        set $service_name   "http";
        set $service_port   "80";
        set $location_path  "/app1";

        ...
        rewrite "(?i)/app1/(.*)" /$1 break;
        rewrite "(?i)/app1$" / break;
        proxy_pass http://upstream_balancer;

}
```

Compare to swarm

A Deployment, ReplicaSet és Service Kubernetes komponenseknek megvan az ekvivalens párja Docker Swarm-ban. A Docker swarm megoldása ráadásul sokkal felhasználó barátabb, a leíró fájl sokkal tömörebb és egyszer?bb. Azonban a szabványosított Ingress API megoldás teljesen hiányzok a swarm-ból, Swarm-ban minden gyártó (Ngnix, Traefic, Docker flow proxy) egyedi módon oldotta meg. Ezzel ellentétben a Kubernetes-ben egy teljesen szabványosított megoldás van, tehát itt egyértelm?en a Kubrnetes a nyer?.

Adding entries to Pod /etc/hosts

https://kubernetes.io/docs/concepts/services-networking/add-entries-to-pod-etc-hosts-with-host-aliases/

```
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  restartPolicy: Never
  hostAliases:
  - ip: "127.0.0.1"
    hostnames:
    - "foo.local"
    - "bar.local"
```

Security

Kubernetes secret

dockerconfigjson

https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/

A **dockerconfigjson** típusú secret docker privát repozitorikhoz való csatlakozáshoz szükséges adatokat tárol:

- csatlakozási URL
- username
- password
- email cím

A dockerconfigjson típusú secret-et a **create secret docker-registry** paranccsal hozhatjuk létre.

```
# kubectl create secret docker-registry <secret-name> --docker-server=<your-registry-server> --docker-username=<your-name> --docker-password=
```

Az elkészült yaml fájl az alábbi kritériumoknak kell megfeleljen:

- type: kubernetes.io/dockerconfigjson
- A data-ban egy darab mez? lehet, aminek a kulcsa: .dockerconfigjson
- A .dockerconfigjson mez? értéke Base64-elt JSON-ban tartalmazza a csatlakozási információkat (user, pass, email, URL)

```
apiVersion: v1
kind: Secret
type: kubernetes.io/dockerconfigjson
metadata:
  ...
data:
  .dockerconfigjson: eyJodHRwczovL2luZGV4L ... J0QUl6RTIifX0=
```

A JSON tartalma az alábbi:

```
{"auths":{"your.private.registry.example.com":{"username":"janedoe","password":"xxxxxxxxxxx","email":"jdoe@example.com","auth":"c3R...zE2"}}}
```

ServiceAccount-okban illetve a deployment leírókba be lehet hivatkozni a docker-repository secret-eket a **imagePullSecrets** szekcióban:

```
apiVersion: v1
kind: ServiceAccount
imagePullSecrets:
- name: pull-secret-example1
- name: pull-secret-example2
...
```

....!!! hova kerül ez mountolásra a pod-ban???....

ServiceAccount

Minden névtérnek van egy default service-caccount-ja, aminek a neve **default**. Ha külön nem adunk meg service-account-ot, akkor ezt fogja használni minden deployment.

A service account-ot tartalmazza azokat a certifikációkat, amivel a Kubernetes azonosítja az adott account-ot. A **kubectl create serviceacount** paranccsal létrejön az account és hozzá le is generálja a Kubernetes a szükséges secret-eket.

```
# kubectl create serviceaccount prometheus-kube-state-metrics -n mynamespace
```

Ez jött bel?le létre:

```
apiVersion: v1
imagePullSecrets:
- name: prometheus-kube-state-metrics-dockercfg-rngd8
kind: ServiceAccount
metadata:
  labels:
    app: prometheus
    chart: prometheus-8.14.0
    component: kube-state-metrics
    heritage: Tiller
    release: prometheus
  name: prometheus-kube-state-metrics
  namespace: mynamespace
secrets:
- name: prometheus-kube-state-metrics-token-t4d6r
- name: prometheus-kube-state-metrics-dockercfg-rngd8
```

imagePullSecrets: Az itt megadott secret-ben lév? ceritifkációt fogja használni az image repository-hoz való csatlakozáshoz minden olyan Deployment vagy DemonSet, ami ezzel a service-account-al fut. **secrets**: Az ebben a szekcióban megadott secret-ek

Készíthetünk service-account-ot úgy is, hogy létrehozunk egy 'üres' serviceAccount definíciót. Ekkor a Kubernetes ugyan úgy létre fogja hozzá hozni a token-eket.
prometheus-kube-state-metrics.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus-kube-state-metrics
  namespace: mynamespace

 # kubectl apply -f prometheus-kube-state-metrics.yaml
```

ServiceAccount secret-ek

Minden service-account-hoz létrejön autómatikusan két secret:

- <service-account név>-dockercfg-<azonosító>
- <service-account név>-token-<azonosító>

A dockercfg-ben van az összes olyan docker registry elérhet?sége, ahonnan le tudja húzni az image-t az a Deployment ami ezt a serviceAccount-ot használja. Nézzünk bele ebbe a docker-es secret-be:

```
# kubectl get secret prometheus-kube-state-metrics-dockercfg-rngd8 -n mynamespace -o yaml
---------------------------
apiVersion: v1
data:
  .dockercfg: eyIxNzIuMzAuMS4xOjUwMDAiO....
kind: Secret
metadata:
  annotations:
    kubernetes.io/service-account.name: prometheus-kube-state-metrics
    kubernetes.io/service-account.uid: d3615a80-acb9-11e9-ab6f-525400efb4ec
    openshift.io/token-secret.name: prometheus-kube-state-metrics-token-w9qjx
  name: prometheus-kube-state-metrics-dockercfg-rngd8
  namespace: mynamespace
type: kubernetes.io/dockercfg
```

Ha Base64-el dekódoljuk a tartalmat, ezt találjuk benne.

```
{"172.30.1.1:5000":{"username":"serviceaccount","password":"eyJhbGciOiJSUz...1lYwAtg","email":"serviceaccount@example.org","auth":"c2Vydml...
```

A token-es secret-ben pedig a ca, a cert és a token van:

```
# kubectl get secret prometheus-kube-state-metrics-token-t4d6r -n mynamespace -o yaml
-------------------------------------
apiVersion: v1
data:
```

```
  ca.crt: LS0tLS1CRUdJTi...
  namespace: bXluYW1lc3BhY2U=
  service-ca.crt: LS0tLS1CRU...
  token: ZXlKaGJHY2l..
kind: Secret
metadata:
  name: prometheus-kube-state-metrics-token-t4d6r
  namespace: mynamespace
type: kubernetes.io/service-account-token
```

Volumes

A volume a konténer számára egy kívülr?l felcsatolt fájl vagy mappa. Ez lehet hálózati megosztás vagy a host (node) gépr?l megosztott mappa, ahol a pod fut. A felcsatolás lehet permanes, dinamikus vagy a pod életciklusához kötött. A Kubernetes számtalan volume típust támogat: https://kubernetes.io/docs/concepts/storage/volumes/

Nagy különbség a docker swarm-hoz képest, hogy a fenti linken felsorolt volume-okat a Kubernetes natívan támogatja, ezek minden node-on out of the box elérhet?ek. Ezzel ellentétben swarm-ban a volume plugin-ek nagy részét minden swarm node-on külön-külön telepíteni kell, ami elég kényelmetlen.

Pod szint? volume

A legegyszer?bb volume definíció a pod szinten definiált volume. Ezeket a pod/replicaSet/Deployment yaml fájlokban kell megadni a konténer definíciójában. Ekkor a volume életciklusa meg fog egyezni a pod életciklusával, mikor a pod megsz?nik, a volume is meg fog sz?nni (a mappa tartalma, ami fel volt csatolva a konténerbe nem feltétlen fog törl?dni a volume megsz?nésekor, ez a volume típusától függ)

A pod szint? volume definíciónak két része van. A POD **.spec.volumes** szekciójában kell definiálni a volume forrását (pl hostPath típus esetén a hostVM-en a source mappa/fájl/socket helyét), és a POD **.spec.containers[].volumeMounts** szekciójában kell megadni a konténer számára a mount pontot.

A **.spec.volumes** szekcióban a POD-ban futó összes konténer számára pod-globálisan definiálhatunk volume-okat, majd minden egyes konténer **.spec.containers[].volumeMounts** szekciójában felcsatolhatjuk a pod szinten definiált volume-ot egy konkrét konténerbe. Mivel általában egy pod-ban csak egy konténer van, ezért ezzel a két szint? definícióval nem sokra megyünk.

A volumes listában fel kell sorolni minden egyes volume-ont. A volume definíciója a névvel kezd?dik (ezzel fogunk rá hivatkozni a mount pont megadásánál). Ezt követi a típus megadása.

```
spec:
  ...
  volumes:
  - name: myFileMount
    hostPath:
      path: /hostVM/somefile
      type: File
```

```
....
spec:
  containers:
  - name: xxx
    ...
    volumeMounts:
    - mountPath: /home/mydirectory
      name: myFileMount
```

Note
Swarm-os megállapítás ....

A pod szinten definiálható volume típusok száma limitált, a https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes linken felsorolt volume típusok közül nem mindegyik használható pod szint? definícióiban. Pl az nfs volume-hot csak PersistentVolume definícióval tudjuk használni.

Warning
A pod szint? volume definíció er?sen ellenjavallott, és produkciós környezetben sosem használják. Éles helyzetben mindig a **StorageClass**-t vagy a **PersistentVolumes**-ot használjuk

hostPath

Ez a legegyszer?bb volume típus. Arról a virtuális gépr?l (node-ról) mountol fel egy mappát/fájlt, ahol aktuálisan a pod fut. Vagyis ha a pod átkerül egy másik node-ra, minden a hostPath mappába írt adat el fog veszni. Ezért ezt csak olyan alkalmazások esetében szabad használni (vagy még ott sem) amiknek munkaterületre van szüksége írás intenzív m?veletekhez, de nem baj, ha a mappa tartalma elveszik. (Az írás intenzív konténereknek mindig szükséges egy felcsatolt volume, ugyanis a konténerek írható rétegének a módosítgatása nagyon rossz hatásfokú)

A volume neve csak kisbet?kb?l és "-"jelb?l állhat. Nézzük egy példát.

Lépjünk be a minikube VM-be és hozzunk benne létre egy fájlt a /home/docker mappában (mert a docker user ezt tudja írni). Kés?bb ezt a mappát fogjuk felcsatolni a pod-ba.

```
# minikube ssh
                         _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___    (_) | |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\  | | , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || | | |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)

$
$ echo "This is the message!" > /home/docker/demo.txt
```

Hozzunk létre egy pod-ot. Ebbe fel fogjuk csatolni a /home/docker mappát a VM-r?l.

```
apiVersion: v1
kind: Pod
metadata:
  name: netpod2
  labels:
    run: netpod2
spec:
  containers:
  - args: ["sh", "-c", "while true; do echo hello; sleep 10;done"]
    image: amouat/network-utils
    name: netpod2
    volumeMounts:
    - mountPath: /home
      name: my-file-mount
  volumes:
  - name: my-file-mount
    hostPath:
      path: /home/docker
      type: Directory
```

Hozzuk létre a pod-ot.

```
# kubectl create -f pod-network-util.yaml
```

Majd nézzük meg mi van a /home mappába. És láss csudát, ott van a demo.txt fájl.

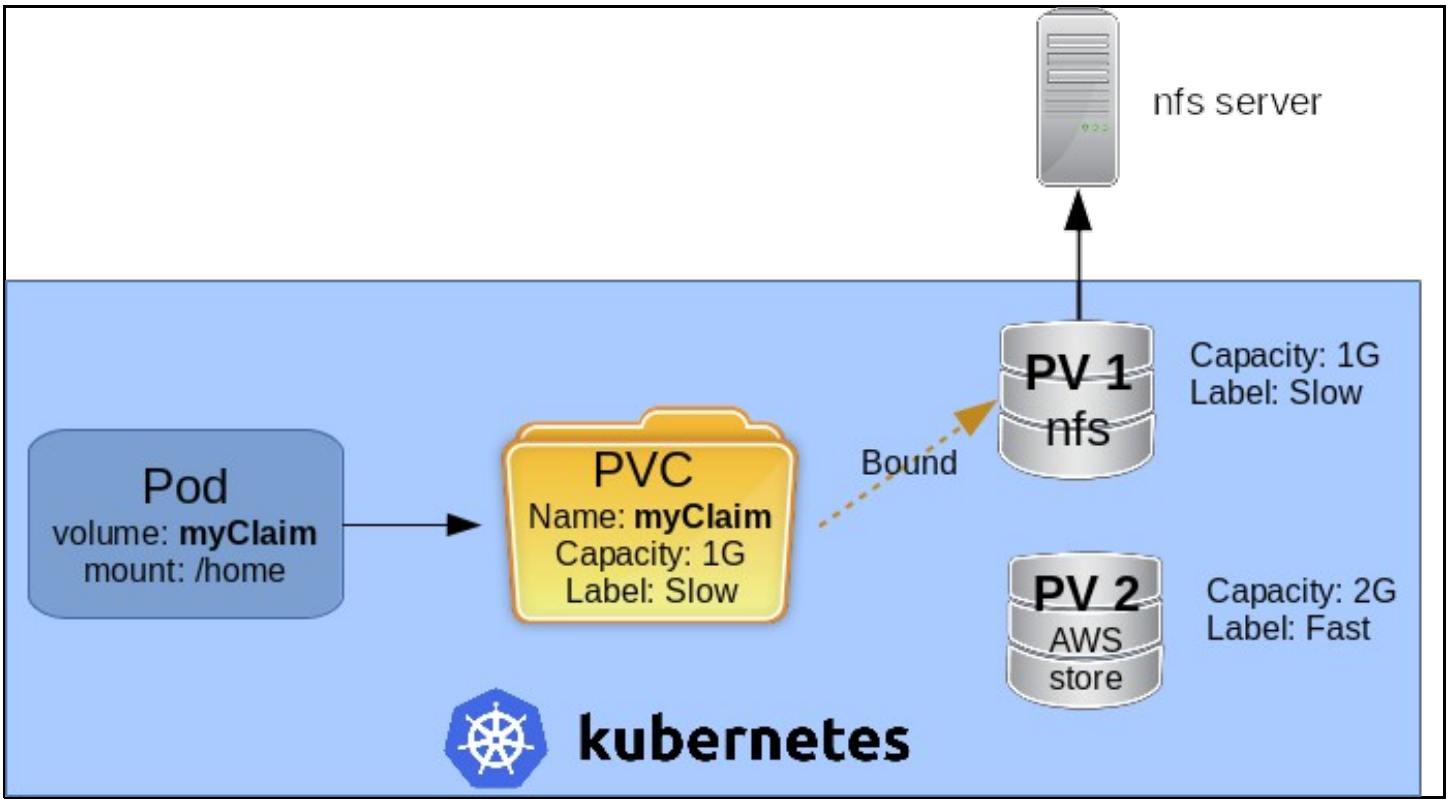
```
# kubectl exec -it netpod2 cat /home/demo.txt
This is the message!
```

emptyDir

Persistent volumes

https://kubernetes.io/docs/concepts/storage/persistent-volumes/#lifecycle-of-a-volume-and-claim

A PeristentVolume-okat (PV) arra használjuk, hogy szétválasszuk egymástól a Volume és a pod definícióját. Ezáltal a pod független lesz a volume fizikai megvalósításától, a pod-ban csak hivatkozunk egy Perzisztens Volume-ra, pod szinten nem jelenek meg a volume részletei (hogy milyen típus, hogy milyen autentikációval lehet hozzá csatlakozni, és hogy pontosan hol van, stb...) ugyanis az kizárólag a Kubernetes adminisztrátorára tartozik, a klaszter felhasználóinak, akik a pod-okat definiálják már nem szabad rálátni a volume definícióra. A pod-okban nem hivatkozunk közvetlenül a PV-kre (PersitentVolume). A pod és a volume közé bevezettek még egy absztrakció szintet, az úgynevezett PersitentVolumeClaime-et (PVC). Ezeket az Igényeket már a felhasználók definiálják, és a pod-ban az igény nevére hivatkozunk. Azonban az igényben sem hivatkozunk egy konkrét PV-re. Az elérhet? PV-k rejtve maradnak a pod-ok számára. A PVC-ben csak leírjuk a pod igényeit, pl a tárhely igényt, címkéken keresztül további paramétereket ... és a Kubernetes fog keresni egy az igényhez passzoló PV-t és hozzá fogja kötni az igényhez (Bound). Tehát a PV és a PVC között csak lazán csatolt kapcsolat van. Egy PV csak egy Igényhez (PVC-hez) köthet?, de gy PVC-re bármennyi pod hivatkozhat.



- ReadWriteOnce - the volume can be mounted as read-write by a single node
- ReadOnlyMany - the volume can be mounted read-only by many nodes
- ReadWriteMany - the volume can be mounted as read-write by many nodes

NFS

A Kubernetes az NFS volume-okat out of the box támogatja (ellentétben a Swarm-al, ahol külön fel kell telepíteni minden egyeses node-ra az NFS driver-et). Az NFS a legegyszer?bben bemutatható Volume típus, mivel minden Linuxon elérhet? "gyárilag", ezért ezen keresztül mutatjuk be a PV és PVC kezelését.

Az NFS szerver telepítésér?l itt olvashatunk b?vebben: https://wiki.berki.org/index.php/Docker_volume_orchestration#NFS_mount_on_the_VM
Listázzuk ki az anyagépen az elérhet? NSF megosztásokat a **showmount** paranccsal:

```
# showmount -e 192.168.42.1
Export list for 192.168.42.1:
/home/adam/dockerStore
```

A /home/adam/dockerStore -t fogjuk felcsatolni egy pod-ba mint PV.

Készítsük el a alábbi PV definíciót:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mynfspv
  labels:
    release: stable
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: slow
  nfs:
    server: 192.168.42.1
    path: "/home/adam/dockerStore/kubernetes"
```

```
# kubectl create -f pv-nfs.yaml
persistentvolume/nfs-example created
```

```
# kubectl get pv
NAME    CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
nfs     1Mi        RWX            Retain           Available                                   50s
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

```
  name: myclaim
spec:
  accessModes:
    - ReadWriteMany
  volumeMode: Filesystem
  resources:
    requests:
      storage: 1Mi
  storageClassName: slow
  selector:
    matchLabels:
      release: stable

# kubectl get pvc
NAME      STATUS   VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS   AGE
myclaim   Bound    mynfspv   1Mi        RWX            slow           22h



apiVersion: v1
kind: Pod
metadata:
  name: mypodnfs
  labels:
    run: netpod
spec:
  containers:
  - args: ["sh", "-c", "while true; do echo hello; sleep 10;done"]
    image: amouat/network-utils
    name: netpod2
    volumeMounts:
    - mountPath: /home
      name: my-file-mount
  volumes:
  - name: my-file-mount
    persistentVolumeClaim:
      claimName: myclaim

# kubectl describe pod netpod-pv
Name:               netpod-pv
...
    Mounts:
      /home from my-file-mount (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-zj4tp (ro)
...
Volumes:
  my-file-mount:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  myclaim
    ReadOnly:   false
...




ls /home/adam/dockerStore/kubernetes/
example.txt



# kubectl exec -it netpod-pv ls /home
example.txt
```

Dynamic provisioning

https://kubernetes.io/docs/concepts/storage/storage-classes/

A PV objektumok használata esetén a Kubernetes adminisztrátornak el?re el kell készíteni el?re az összes fizikai storage-ot és a hozzá tartozó Persistent Volume Kubernetes objektumokat. Van rá lehet?ség, hogy a Volume-okat dinamikusan hozza létre a Kubernetes. Erre szolgálnak a StorageClass objektumok. Ez nem csak azt jelenti, hogy egy meglév? storage-hoz a PV objektumot dinamikusan hozza létre a Kubernetes, hanem azt, hogy a fizikai storage-ot is akkor fogja létrehozni amikor a POD megigényli azt. Pl. egy NFS esetében ez azt jelentené, hogy a StorageClass-ban definiált méret? NFS megosztást is dinamikusan hozná létre a Kubernetes, majd az újonnan legyártott NFS megosztáshoz dinamikusan elkészítené a PV objektumot is. (Az NFS nem a legjobb példa, mert az NFS driver pont nem támogatja a dinamikus m?ködést) Csak olyan storage driver-t tudunk dinamikusan használni, ami kifejezetten támogatja, hogy a Kubernetes adminisztrálja a storage-ok létrehozását és törlését. Ezen driver-ek a hagyományos, statikus storage driver-ek listájával ellentétben már elég sz?k:

- quay.io/kubernetes_incubator/nfs-provisioner : https://github.com/kubernetes-incubator/external-storage/tree/master/nfs
- ...

Dynamic provisioning can be enabled on a cluster such that all claims are dynamically provisioned if no storage class is specified. A cluster administrator can enable this behavior by:

Marking one StorageClass object as default; Making sure that the DefaultStorageClass admission controller is enabled on the API server.

.. példa ...

ConfigMap