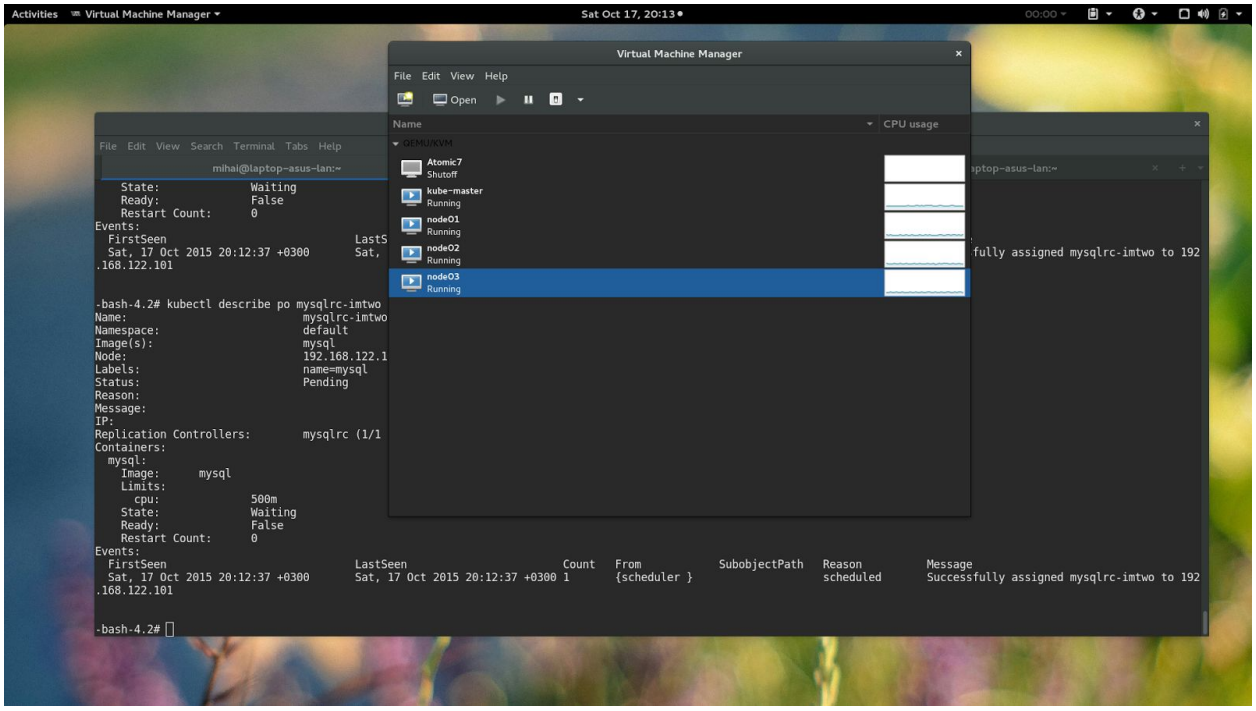


ProjectAtomic - Tutorial

12.11.2015

Mihai Csaky
SysOP Consulting

Introducere

În ultimul timp auzit tot mai des despre Docker, aplicaţii în cloud, Linux pentru Cloud (ProjectAtomic, CoreOS), Kubernetes. Dar,în afară de cei ce lucrează efectiv cu ele pentru firmele din IT, pentru utilizatorul de desktop, rămân o mare necunoscută.

Scop

1. Acest tutorial se dorește o introducere în ProjectAtomic, cu un exemplu clar de utilizare. Exemplul poate fi rulat pe orice laptop/desktop cu minim 6-8GB RAM și un procesor modern care suportă extensiile de virtualizare.
2. Partea a doua a tutorialului arată cum putem transforma o aplicație clasică web, scrisă clar pentru utilizarea pe un singur calculator, fără posibilități prea mari de scalare, într-o aplicație care scalează în cloud, folosind Docker.

Specificații

Avem nevoie de un calculator host, care să aibă liber 6-8GB RAM, procesor cu extensii de virtualizare. Exemplele au fost create pe un laptop ce rula Linux (fedora) și KVM cu virt-manager. Se poate folosi orice combinație de sistem de operare/soluție de virtulizare (VirtualBox etc), cu adaptările de rigoare.
Acces internet, de pe host și din rețeaua în care se află mașinile virtuale.

AtomicProject

CentOS Atomic se poate descărca de la adresa de mai jos:

https://wiki.centos.org/SpecialInterestGroup/Atomic/Download/

Eu am folosit imaginea ISO, pentru a o instala folosind virt-manager, evident se pot folosi și alte soluții sau se poate lucra direct în cloud, pe Amazon sau Google Cloud Platform. Pentru un data-center, atomic se poate instala direct pe bare-metal. Exemplele functionează și direct pe CentOS 7 (nu este nevoie de versiunea Atomic) sau direct pe orice altă distribuție de linux, dar versiunile pentru Cloud sunt gândite pentru o instalare rapidă, simplă și minimală, cu update incremetal și posibilități de roll-back. CoreOS este un alt bun candidat.

Concepte

Un cluster de calculatoare cu AtomicOS folosește Kubernetes pentru a orchestra și distribui automat resursele între nodurile care fac parte din cluster. Principala resursă distribuită automat se numește POD și constă într-o colecție de containere Docker. Unul dintre calculatoare trebuie desemnat master (atomic-master în exemplul nostru), iar celelalte sunt noduri (node01, node02,...) pe care rulează containerele Docker grupate in

POD-uri. Pentru orchestrare se mai foloseşte etcd iar pentru asigurarea comunicaţiei între containere vom folosi flanneld (flanneld creează o reţea virtuală suprapusă pe reţeaua docker, astfel încât două containere Docker aflate pe maşini virtuale diferite să poată comunica direct între ele).

Exemplul nu abordează partea de securitate, best-practices, criptare, redundanţă, partea de izolare, limitare şi control al resurselor, monitorizare sau partea de portal pentru accesul clienţilor. Este o abordare entry-level, dar un nivel mai sus decât o simplă rulare a unei aplicatii într-un container docker.

Etape

Configuraţia la care vrem să ajungem este un host master (atomic-master), care să coordoneze două noduri (node01, node02), iar un host (frontend) să asigure expunerea serviciilor web ale aplicaţiei în reţea (intern sau internet). Nodurile vor fi coordonate de către master, dar pe frontend nu vom configura kubernetes, va rula doar containerele Docker necesare pentru load-balancing/acces din exterior.

Pe atomic-master vom rula şi o bază de date MySQL, de aceea va avea alocat 2G de RAM.

În producţie vom rula baza de date pe un host dedicat, sau o putem rula ca şi pod în clusterul kubernetes. Fiecare soluţie are avantaje/dezavantaje, sau depinde de providerul de cloud pe care-l folosim.

I. Creare imagine de bază cu CentOS atomic

O imagine de bază, singurele configurări vor fi cele de la instalare şi asigurarea accesului cu ssh, pentru uşurinţa configurărilor. Opţional, se poate configura un utilizator cu drepturi sudo, pentru a mări securitatea, dar partea de securitate nu face obiectul acestui tutorial, deci nu este recomandată utilizarea acestei configuraţii în producţie.

Paşi de urmat:

- descărcaţi centos atomic (versiunea preferată pentru tipul de instalare folosit)
- instalaţi şi configuraţi sistemul de operare (un disk de 10-20GB este suficient), setaţi parola pentru root, creaţi opţional un utilizator pentru sudo (în exemplele mele, voi folosi direct acces root)
- configuraţi posibilitatea de acces remote, cu ssh, de preferinţă cu chei
-

  de exemplu, dacă maşina virtuală are IP-ul 192.168.122.144, de pe host executaţi comanda:

```
~$ ssh-copy-id root@192.168.122.144
```

II. Configurare Kubernetes Master (host atomic-master)

Pentru atomic-master dorim un IP fix: 192.168.122.10 (libvirt foloseşte implicit 192.168.122.0/24 pentru maşinile virtuale, IP-ul se alege în funţie de soluţia de virtualizare folosită).

exemplu:

```
$vi /etc/libvirt/qemu/networks/default.xml
<dhcp>
        <range start='192.168.122.100' end='192.168.122.254'/>
        <host mac="52:54:00:e3:7b:90" name="atomic-master.cloud"
ip="192.168.122.10" />
</dhcp>
```

Registry Docker Local - pentru a păstra imaginile docker la dispoziţia sistemului, vom crea un registru docker local, în care putem pune direct propriile imagini. Dacă o imagine nu este în registrul local, va fi descărcată din Docker HUB.

Pe atomic-master, rulăm:

```
$ mkdir -p /var/lib/local-registry
$ chcon -Rvt svirt_sandbox_file_t /var/lib/local-registry
```

Creăm imaginea docker:

```
$ docker create -p 5000:5000 \
        -v /var/lib/local-registry:/srv/registry \
        -e STANDALONE=false \
        -e MIRROR_SOURCE=https://registry-1.docker.io \
        -e MIRROR_SOURCE_INDEX=https://index.docker.io \
        -e STORAGE_PATH=/srv/registry \
        --name=local-registry registry
```

Ne asigurăm că imaginea Docker creată porneşte automat la pornirea maşinii virtuale, folosind un fişier pentru systemd:

```
vi /etc/systemd/system/local-registry.service
```

cu conținutul:

```
[Unit]
Description=Local Docker Mirror registry cache
Requires=docker.service
After=docker.service
[Service]
Restart=on-failure
RestartSec=10
ExecStart=/usr/bin/docker start -a %p
ExecStop=-/usr/bin/docker stop -t 2 %p
[Install]
WantedBy=multi-user.target
```

serviciul mai trebuie activat și pornit:

```
$systemctl daemon-reload
$systemctl enable local-registry
$systemctl start local-registry
```

verificăm dacă totul este în ordine:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
03ca68d7bc4d        registry            "docker-registry"   38 seconds ago
Up 9 seconds        0.0.0.0:5000->5000/tcp   local-registry
```

În acest moment, ar trebui să avem o mașină virtuală, cu numele atomic-master, cu IP-ul 192.168.122.10, pe care rulează un registru Docker, expunând rețelei interne portul 5000/tcp.

Configurăm etcd

etcd este un serviciu folosit de kubernetes, dar și de scripturile din containerele Docker. etcd poate fi instalat în cluster, accesul poate fi securizat. Noi vom folosi forma cea mai simplă, etcd instalat pe master, cu acces liber din rețeaua internă.

editățm $vi /etc/etcd/etcd.conf

şi modificăm fişierul pentru ca etcd să asculte pe toate ip-urile, înlocuind localhost cu 0.0.0.0

Un mod mai simplu de a ne atinge scopul, este rularea comenzii:

$sed -i 's/localhost/0.0.0.0/g' /etc/etcd/etcd.conf

Configurare Kubernetes pe master
editățm fişierele din /etc/kubernetes:

$vi /etc/kubernetes/config

```
# Comma seperated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd_servers=http://192.168.122.10:2379"
# How the replication controller and scheduler find the kube-apiserver
KUBE_MASTER="--master=http://192.168.122.10:8080"
```

$vi /etc/kubernetes/apiserver

```
# The address on the local server to listen to.
KUBE_API_ADDRESS="--address=0.0.0.0"
```

Ştergem ServiceAccount din parametrul KUBE_ADMISSION_CONTROL:

```
KUBE_ADMISSION_CONTROL="--admission_control=NamespaceLifecycle,N
amespaceExists,LimitRanger,SecurityContextDeny,ResourceQuota"
```

activăm serviciile:

$systemctl enable etcd kube-apiserver kube-controller-manager kube-scheduler

pornim serviciile:

$systemctl start etcd kube-apiserver kube-controller-manager kube-scheduler

verificăm:

$ ps aux | grep kube

```
kube        2685  8.0  1.5  46972 28288 ?        Ssl  11:03   0:01
/usr/bin/kube-apiserver --logtostderr=true --v=0
--etcd_servers=http://127.0.0.1:2379 --address=0.0.0.0
--allow_privileged=false --service-cluster-ip-range=10.254.0.0/16
--admission_control=NamespaceLifecycle,NamespaceExists,LimitRanger,Se
curityContextDeny,ResourceQuota
kube        2686  0.6  0.5  26944 10452 ?        Ssl  11:03   0:00
/usr/bin/kube-controller-manager --logtostderr=true --v=0
--master=http://192.168.122.10:8080
kube        2687  0.3  0.3  16220  7108 ?        Ssl  11:03   0:00
/usr/bin/kube-scheduler --logtostderr=true --v=0
--master=http://192.168.122.10:8080
```

În acest moment ar trebui să avem un host master complet configurat, care aşteaptă conexiuni de la nodurile din cluster şi instrucţiuni pentru a le coordona. În etapa următoare configurăm două noduri, prin clonarea imaginii iniţiale a maşinii virtuale.

III. Configurare Nod Kubernets

Clonăm imaginile pentru noduri şi le dăm câte un IP fix:

```
$ vi /etc/libvirt/qemu/networks/default.xml
<dhcp>
        <range start='192.168.122.100' end='192.168.122.254'/>
        <host mac="52:54:00:e3:7b:90" name="atomic-master.cloud"
ip="192.168.122.10" />
        <host mac="52:54:00:e3:7b:92" name="frontend.cloud"
ip="192.168.122.20" />
        <host mac="52:54:00:91:ec:15" name="node01.cloud"
ip="192.168.122.101" />
  <host mac="52:54:00:e1:ef:92" name="node01.cloud" ip="192.168.122.102"
/>

</dhcp>
```

în exemplu am alocat un IP fix şi pentru host-ul frontend: 192.168.122.20

IV. Configurăm docker, kubernetes şi flanneld pe fiecare nod

Docker trebuie să utilizeze docker-registry cache:

```
$vi /etc/sysconfig/docker
OPTIONS='--registry-mirror=http://192.168.122.10:5000 --selinux-enabled'
```

Flannel overlay:

```
$vi /etc/sysconfig/flanneld
# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://192.168.122.10:2379"
# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/atomic01/network"
```

activăm serviciul:

```
$systemctl enable flanneld
```

setăm numele hostname:

```
 $vi /etc/hostname
kube01.cloud
```

configurăm nodul   kubelet:

```
$vi /etc/kubernetes/config
# How the controller-manager, scheduler, and proxy find the apiserver
KUBE_MASTER="--master=http://192.168.122.10:8080"
$vi /etc/kubernetes/kubelet
# kubernetes kubelet (minion) config
# The address for the info server to serve on (set to 0.0.0.0 or "" for all
interfaces)
KUBELET_ADDRESS="--address=192.168.122.101"
# The port for the info server to serve on
# KUBELET_PORT="--port=10250"
# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname_override=192.168.122.101"
# location of the api-server
KUBELET_API_SERVER="--api_servers=http://192.168.122.10:8080"
# Add your own!
KUBELET_ARGS=""
```

activăm serviciile:

```
$systemctl enable flanneld kubelet kube-proxy
```

reboot:

```
$systemctl reboot
```

Repetăm pentru fiecare nod din rețea. Folosim doar două noduri, deci mai facem un host prin clonare, numit node02, cu setările respective.

Verificăm setările, pe atomic-master, nodurile trebuie să fie vizibile:

```
$ssh root@192.168.122.10
$kubectl get no
NAME              LABELS                                  STATUS
192.168.122.101   kubernetes.io/hostname=192.168.122.101   Ready
192.168.122.102   kubernetes.io/hostname=192.168.122.102   Ready
```

Pe fiecare nod, inclusiv frontend, trebuie să configurăm docker astfel încât să folosească flanneld:

```
$ mkdir -p /etc/systemd/system/docker.service.d/
$  vi /etc/systemd/system/docker.service.d/10-flanneld-network.conf
[Unit]
After=flanneld.service
Requires=flanneld.service
[Service]
EnvironmentFile=/run/flannel/subnet.env
ExecStartPre=-/usr/sbin/ip link del docker0
ExecStart=
ExecStart=/usr/bin/docker -d \
        --bip=${FLANNEL_SUBNET} \
        --mtu=${FLANNEL_MTU} \
        $OPTIONS \
        $DOCKER_STORAGE_OPTIONS \
        $DOCKER_NETWORK_OPTIONS \
        $INSECURE_REGISTRY
```

V. Configurare frontend

Acest host va avea flanneld şi docker configurat să-l folosească. Clonaţi imaginea originală, asiguraţi-vă că foloseşte IP-ul desemnat: 192.168.122.20

Docker trebuie să folosească docker-registry cache:

```
$vi /etc/sysconfig/docker
```

```
OPTIONS='--registry-mirror=http://192.168.122.10:5000 --selinux-enabled'
```

Flannel overlay:

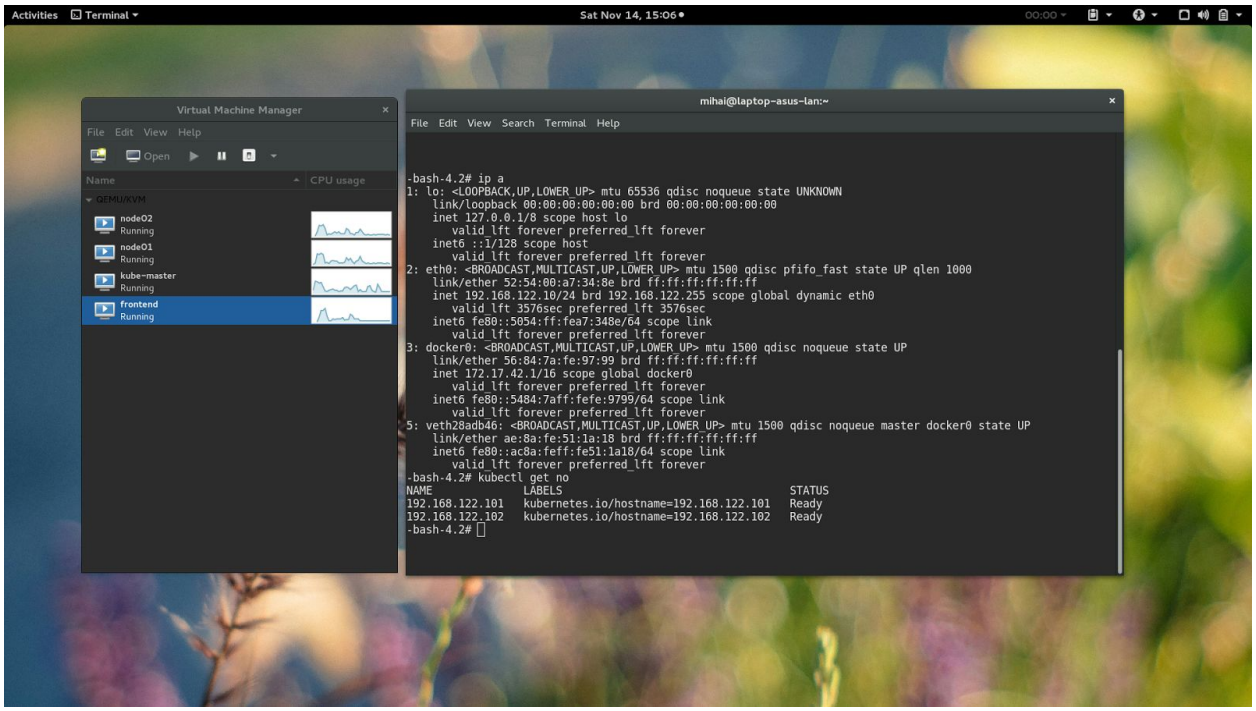
```
$vi /etc/sysconfig/flanneld
# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://192.168.122.10:2379"
# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/atomic01/network"
```

activați serviciul:

```
$systemctl enable flanneld
$ mkdir -p /etc/systemd/system/docker.service.d/
$  vi /etc/systemd/system/docker.service.d/10-flanneld-network.conf
[Unit]
After=flanneld.service
Requires=flanneld.service
[Service]
EnvironmentFile=/run/flannel/subnet.env
ExecStartPre=-/usr/sbin/ip link del docker0
ExecStart=
ExecStart=/usr/bin/docker -d \
        --bip=${FLANNEL_SUBNET} \
        --mtu=${FLANNEL_MTU} \
        $OPTIONS \
        $DOCKER_STORAGE_OPTIONS \
        $DOCKER_NETWORK_OPTIONS \
        $INSECURE_REGISTRY
```

În acest punct trebuie să putem rula clusterul ce constă în 4 imagini virtuale, un host master (192.168.122.10), două noduri pentru pod-uri (192.168.122.101 și 192.168.122.102) și un host pentru expunerea aplicației către exterior (frontend, 192.168.122.20). Pe frontend nu configurăm kubernetes, dar configurăm flanneld pentru a permite comunicația între contaienrele Docker de pe frontend și cele de pe nodurile clusterului.

VI. Instalare Mysql (mariadb)



Clusterul nostru virtual arată astfel: un master (192.168.122.10), două noduri (192.168.122.101, 192.168.122.102) şi un frontend (192.168.122.20).

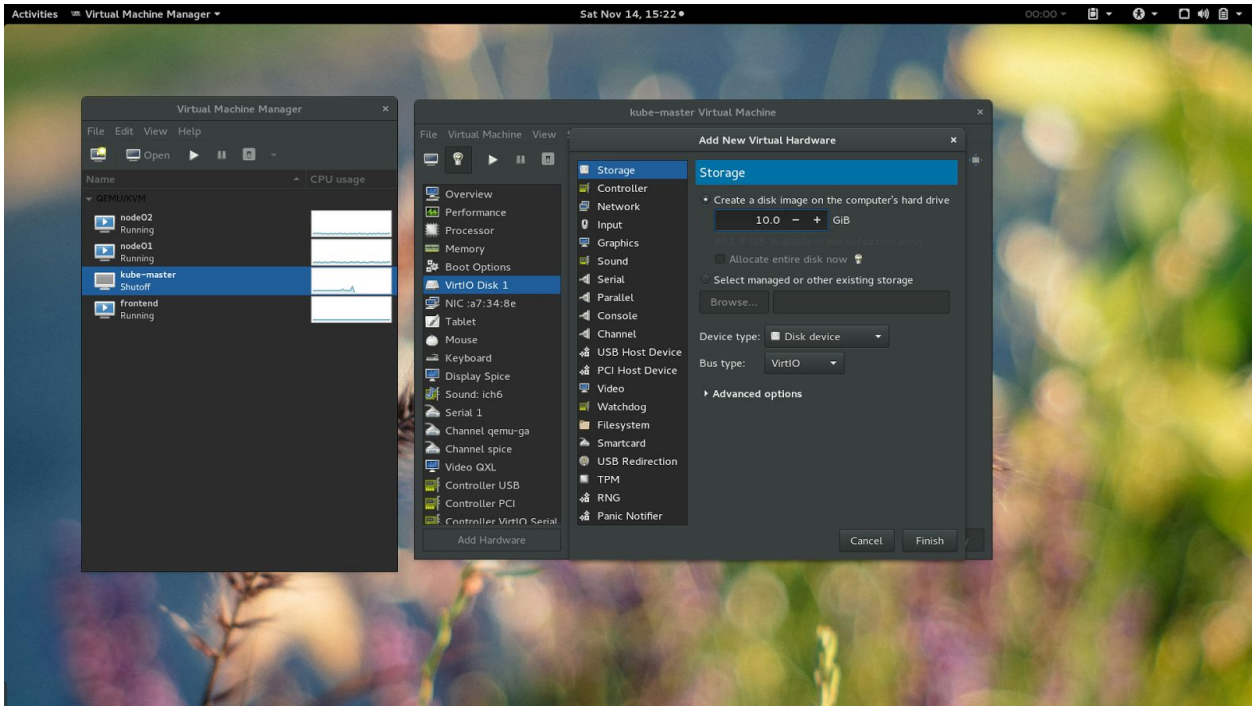
Verificăm dacă masterul şi nodurile sunt conectate, rulând pe master comanda:

```
-bash-4.2# kubectl get no
NAME              LABELS                                   STATUS
192.168.122.101   kubernetes.io/hostname=192.168.122.101   Ready
192.168.122.102   kubernetes.io/hostname=192.168.122.102   Ready
```

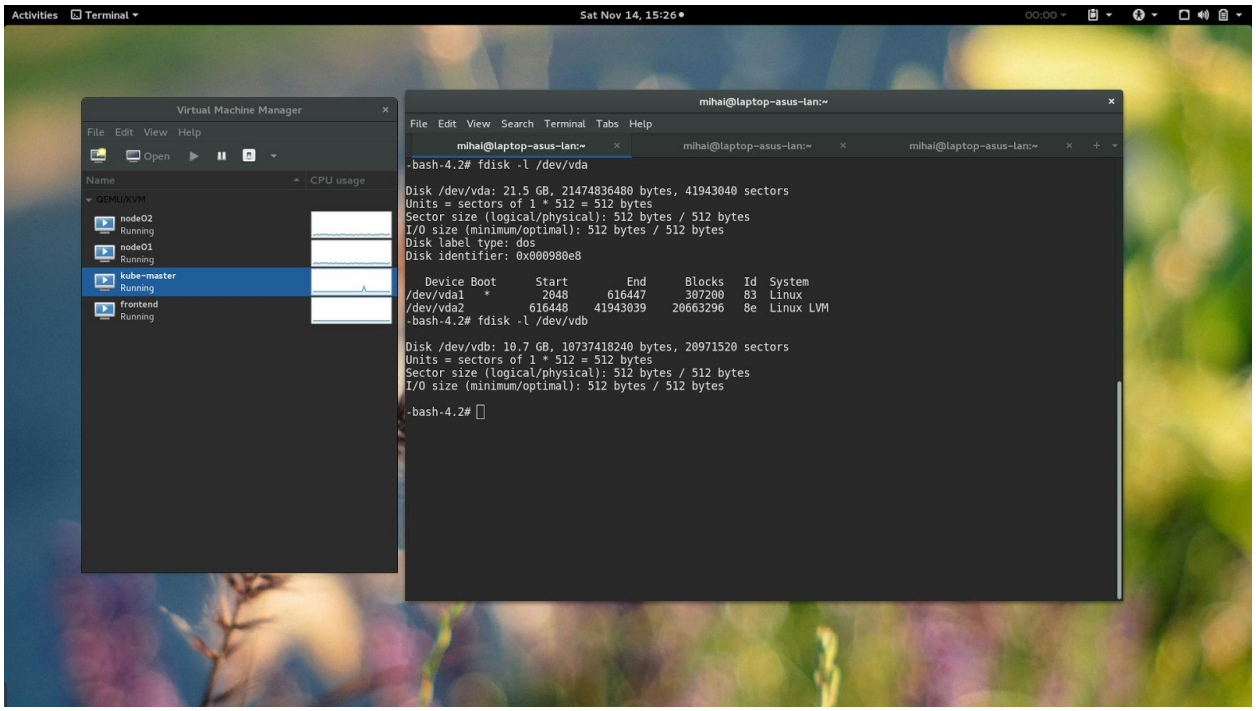
Avem nevoie de persistenţă, deci prima aplicaţie pe care o vom instala este o bază de date MySQL. Avem multe variante de instalare la dispoziţie, de la un host complet separat de cluster până la o integrare totală, în care baza de date să fie controlată de kubernetes. Fiind doar un demo, am ales o soluţie de compromis, baza de date va rula pe master, nu va fi controlată de kubernetes, dar va expune portul 3306 ca şi serviciu disponibil aplicaţiilor din cloud.

Pe master vom crea un disk suplimentar, de 10GB, unde vor fi stocate fişierele bazei de date. Fiindcă folosim Atomic ca şi distribuţie, nu putem instala aplicaţii noi pe host, MySQL

va rula ca și instanță docker.



În qemu, discul 2 se numește vdb:



Creăm o partiţie primară şi o formatăm XFS.

```
-bash-4.2# mkfs.xfs -L data /dev/vdb1

meta-data=/dev/vdb1              isize=256    agcount=4, agsize=655296 blks
         =                       sectsz=512   attr=2, projid32bit=1
         =                       crc=0        finobt=0
data     =                       bsize=4096   blocks=2621184, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0 ftype=0
log      =internal log           bsize=4096   blocks=2560, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
```

Creăm un director /var/lib/mysql accesibil din containerele docker (de aceea modificăm contextul), şi montăm permanent partiţia nou creată în acest director:

```
-bash-4.2# mkdir /var/lib/mysql

-bash-4.2# vi /etc/fstab

/dev/vdb1     /var/lib/mysql               xfs   nobarrier,noatime,nodiratime   1  1

-bash-4.2# mount -a
```

verificăm:

```
-bash-4.2# df -h
Filesystem           Size  Used Avail Use% Mounted on
/dev/mapper/cah-root  3.0G  957M  2.1G  32% /
devtmpfs              904M     0  904M   0% /dev
tmpfs                 921M     0  921M   0% /dev/shm
tmpfs                 921M  424K  920M   1% /run
tmpfs                 921M     0  921M   0% /sys/fs/cgroup
/dev/vda1             283M   70M  195M  27% /boot
/dev/vdb1              10G   33M   10G   1% /var/lib/mysql
```

-bash-4.2# chcon -Rvt svirt_sandbox_file_t /var/lib/mysql/

changing security context of ‘/var/lib/mysql/’

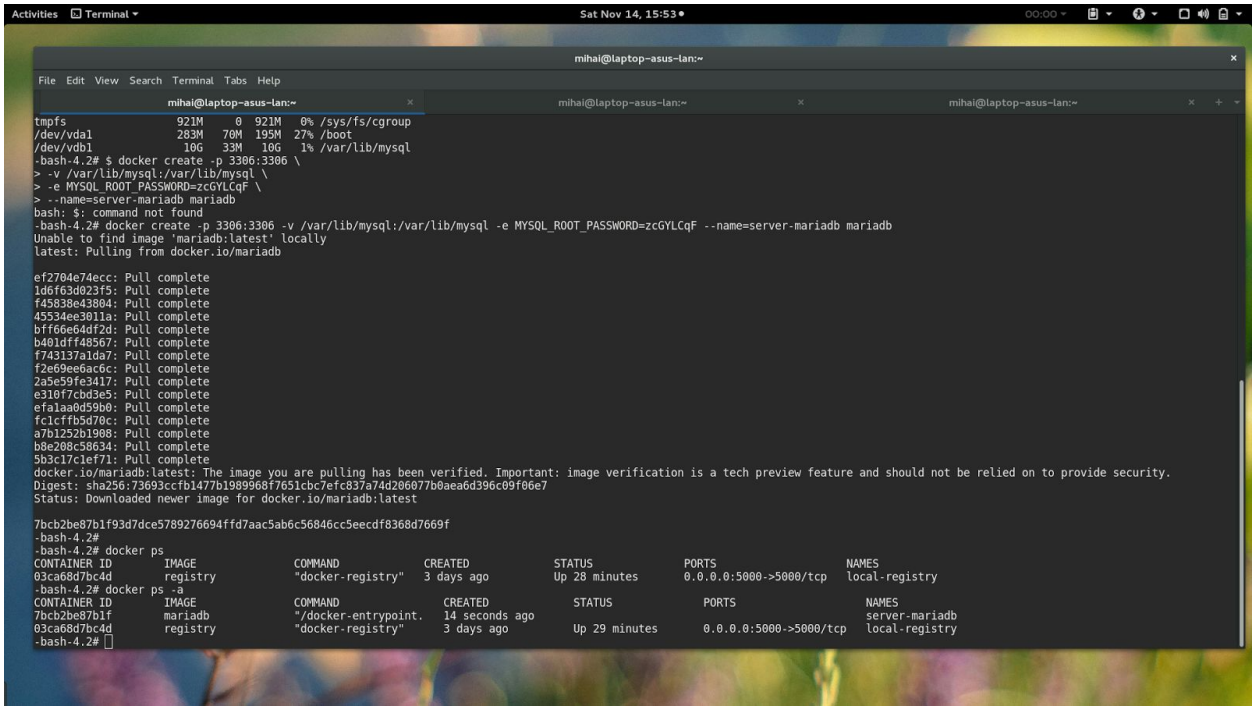
Am folosit ultima variantă de MariaDB, a cărei documentaţie spune că se rulează astfel:

```
docker run --name some-mariadb -v /my/own/datadir:/var/lib/mysql -e
MYSQL_ROOT_PASSWORD=my-secret-pw -d mariadb:tag
```

Creăm o imagine docker:

```
-bash-4.2# docker create -p 3306:3306 -v /var/lib/mysql:/var/lib/mysql -e
MYSQL_ROOT_PASSWORD=zcGYLCqF --name=server-mariadb mariadb
Unable to find image 'mariadb:latest' locally
latest: Pulling from docker.io/mariadb
```

Imaginea este descărcată, dar nu rulează:



Facem setările pentru a porni imaginea la pornirea calculatorului:

vi /etc/systemd/system/server-mariadb.service

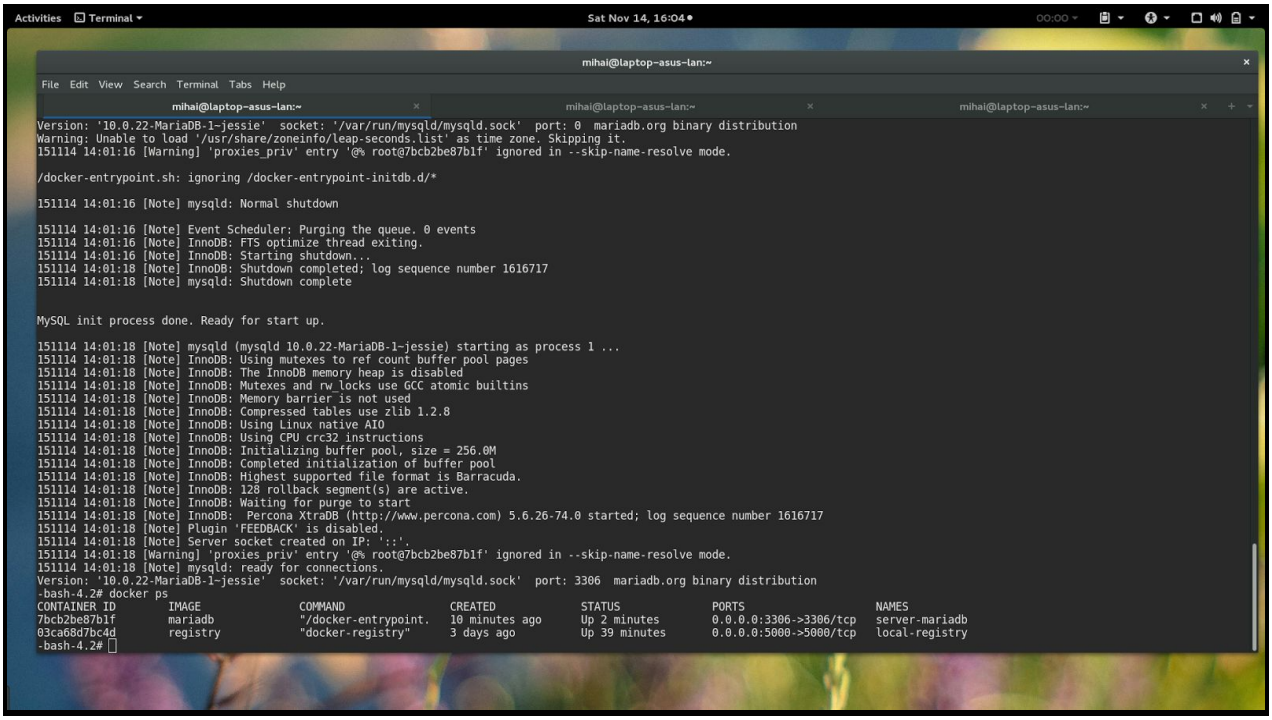
```
[Unit]
Description=Local Docker MariaDB server
Requires=docker.service
After=docker.service
[Service]
Restart=on-failure
RestartSec=10
ExecStart=/usr/bin/docker start -a %p
ExecStop=-/usr/bin/docker stop -t 2 %p
[Install]
WantedBy=multi-user.target
```

serviciul mai trebuie activat și pornit:

```
$systemctl daemon-reload
$systemctl enable local-registry
$systemctl start local-registry
```

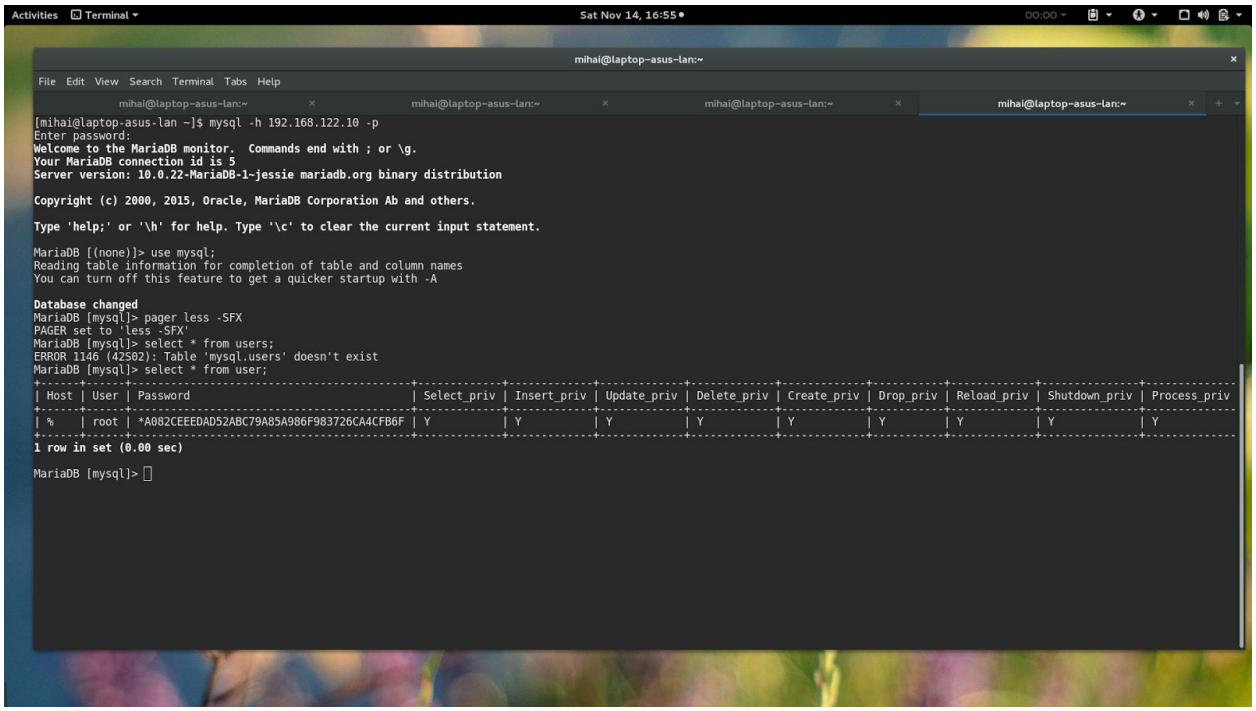
verificăm dacă totul este în ordine:

```
-bash-4.2# docker logs server-mariadb
```



Imaginea a pornit, logurile arată că totul este ok, iar directorul /var/lib/mysql este populat cu fișierele bazei de date.

Serviciul MySQL este disponibil și din rețeaua externă clusterului, putem să ne conectăm direct la 192.168.122.10 portul 3306.



VII. Kubernetes și Mysql

Baza de date este gestionată în afara clusterului Kubernetes, dar dorim o integrare, astfel ca orice pod kubernetes să poată folosi baza de date. Pentru aceasta kubernetes are o facilitate interesantă numită servicii externe:

http://kubernetes.io/v1.0/docs/user-guide/services.html#services-without-selectors

Pe hostul master, creăm un fișier, mysql-service.yml

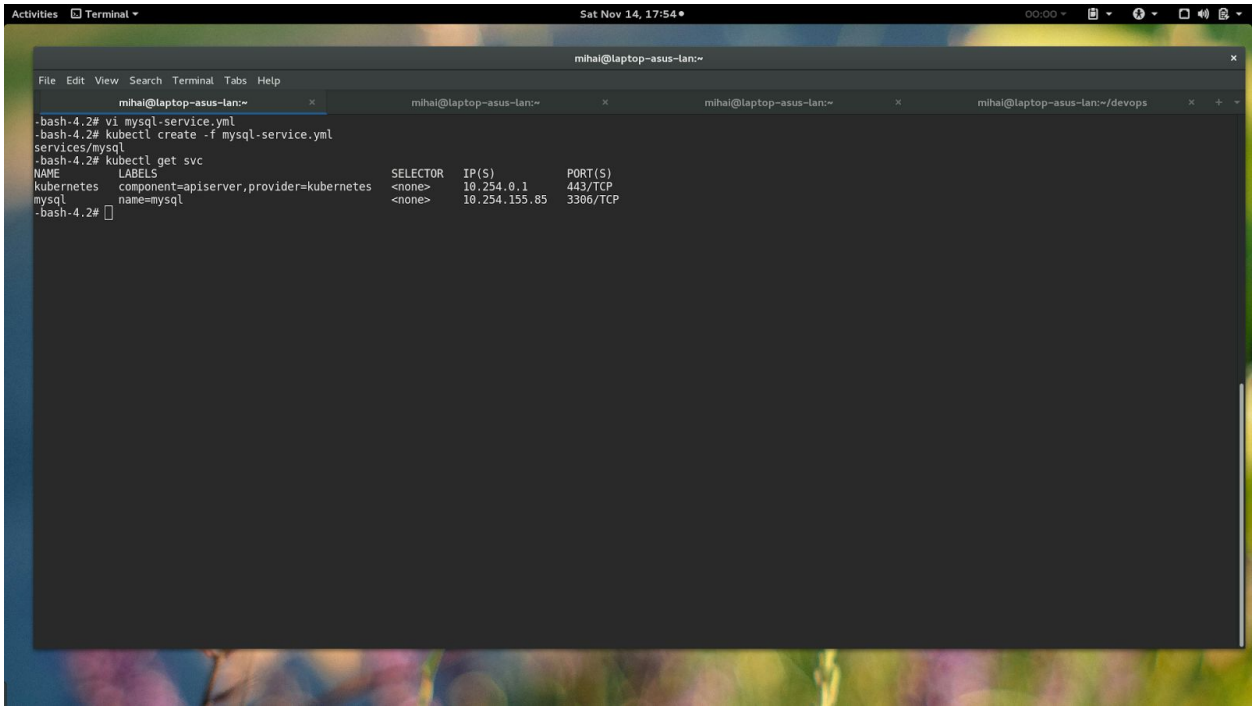
vi mysql-service.yml

cu următorul conținut:

```
apiVersion: v1
Kind: Service
metadata:
 name: mysql
 labels:
  name: mysql
spec:
  ports:
  - port: 3306
```

Apoi rulăm comanda:

```
kubectl create -f mysql-service.yml
services/mysql
-bash-4.2# kubectl get svc
NAME         LABELS                                    SELECTOR   IP(S)          PORT(S)
kubernetes   component=apiserver,provider=kubernetes   <none>     10.254.0.1     443/TCP
mysql        name=mysql
```



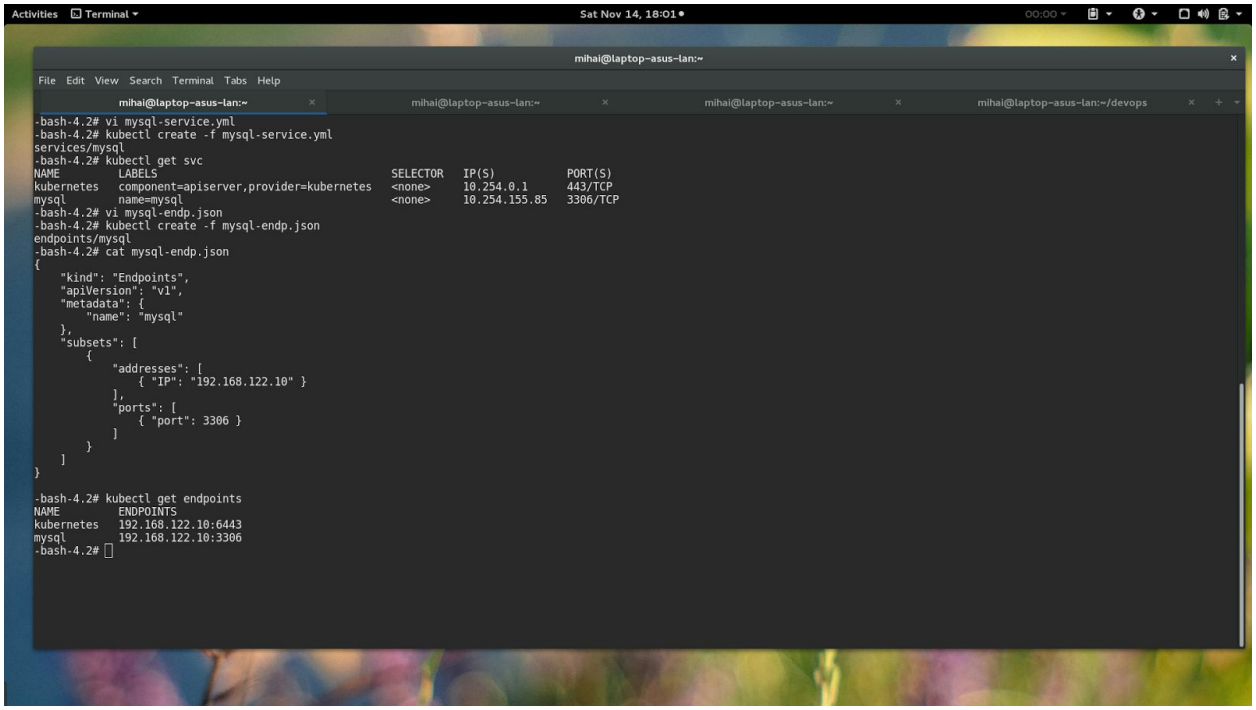
Astfel am definit un serviciu în kubernetes, dar el nu poate fi încă folosit până nu îl conectăm la baza de date reală, de aceea vom defini și un endpoint:

```
-bash-4.2# vi mysql-endp.json

{
  "kind": "Endpoints",
  "apiVersion": "v1",
```

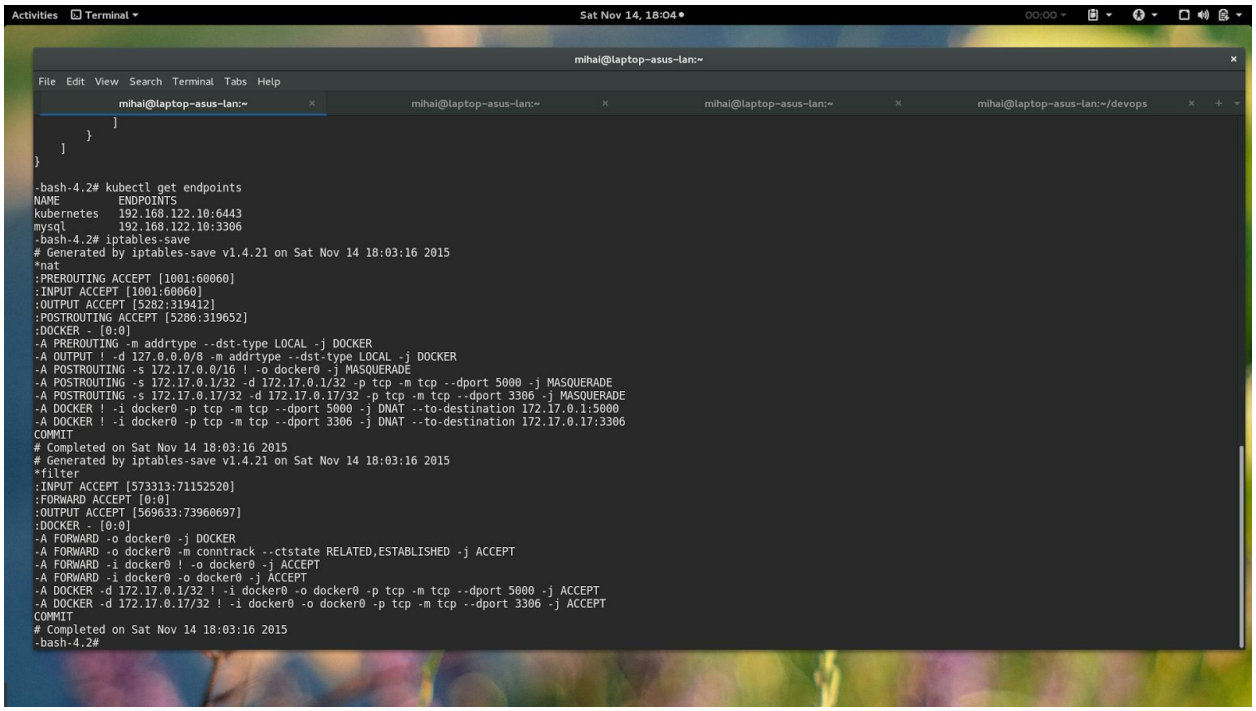
```
  "metadata": {
     "name": "mysql"
  },
  "subsets": [
    {
        "addresses": [
            { "IP": "192.168.122.10" }
        ],
        "ports": [
            { "port": 3306 }
        ]
    }
  ]
}
```

```
-bash-4.2# kubectl create -f mysql-endp.json
endpoints/mysql
```

Acum baza de date mysql rulează în afara controlului clusterului kubernetes, dar este accesibilă din interiorul clusterului, pe un IP al rețelei definite la services: 10.254.155.85



Din curiozitate, putem rula comanda iptables-save pentru a vizualiza cum a realizat kubernetes accesul la aceste servicii setând reguli în firewall.



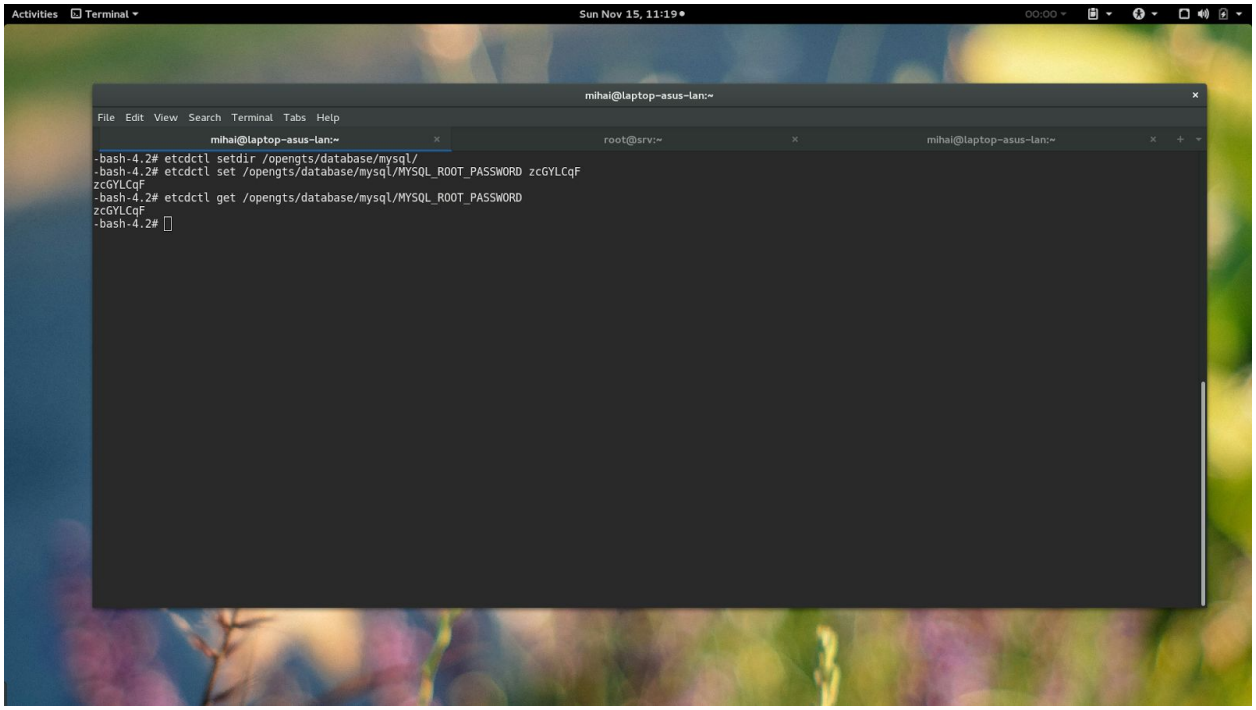
Din nefericire, în acest punct un reboot a dat sistemul peste cap, iar soluția (temporară), până aflu și cauza, a fost să dezactivez IPV6:

```
-bash-4.2# vi /etc/sysconfig/network

# Created by anaconda

NETWORKING_IPV6=no

IPV6INIT=no
```

VIII. OpenGTS în cloud

Pentru a rula OpenGTS în cloud, am pregătit o imagine docker. Dar înainte de a crea resursele, ne folosim de etcd pentru face accesibilă parola bazei de date către imaginile docker. Parola MYSQL_ROOT_PASSWORD a fost aleasă aleator la crearea imaginii docker pentru serverul MySQL.

```
-bash-4.2# etcdctl setdir /opengts/database/mysql/
-bash-4.2# etcdctl set /opengts/database/mysql/MYSQL_ROOT_PASSWORD zcGYLCqF
zcGYLCqF
-bash-4.2# etcdctl get /opengts/database/mysql/MYSQL_ROOT_PASSWORD
zcGYLCqF
```



Imaginea docker pentru aplicaţia OpeGTS se află în registrul Docker, la adresa:

https://hub.docker.com/r/mcsaky/opengts-cloud/

şi este generată automat din repo-ul github:

https://github.com/mihaics/opengts-cloud
de latfel, în github se află şi fişierele de care avem nevoie, gts_account.sh:

```
#!/bin/bash
etcdctl set /opengts/cartrack/MYSQL_DBNAME dbcartrack
etcdctl set /opengts/cartrack/MYSQL_DBUSER usrcartrack
```

```
etcdctl set /opengts/cartrack/MYSQL_DBPASSWORD $( < /dev/urandom tr -dc _A-Z-a-z-0-9 | head -c${2:-10};echo;)
etcdctl set /opengts/cartrack/SYSADMIN_PASSWORD $( < /dev/urandom tr -dc _A-Z-a-z-0-9 | head -c${2:-10};echo;)
etcdctl set /opengts/cartrack/CREATE_DATABASE true

kubectl create -f opengts-rc-cartrack.yml
```

Acest script generează cheile care conţin parole (random), în etcd. Parolele vor fi citite de scripturile care iniţializează aplicaţia, de aceea accesul la etcd trebuie securizat.

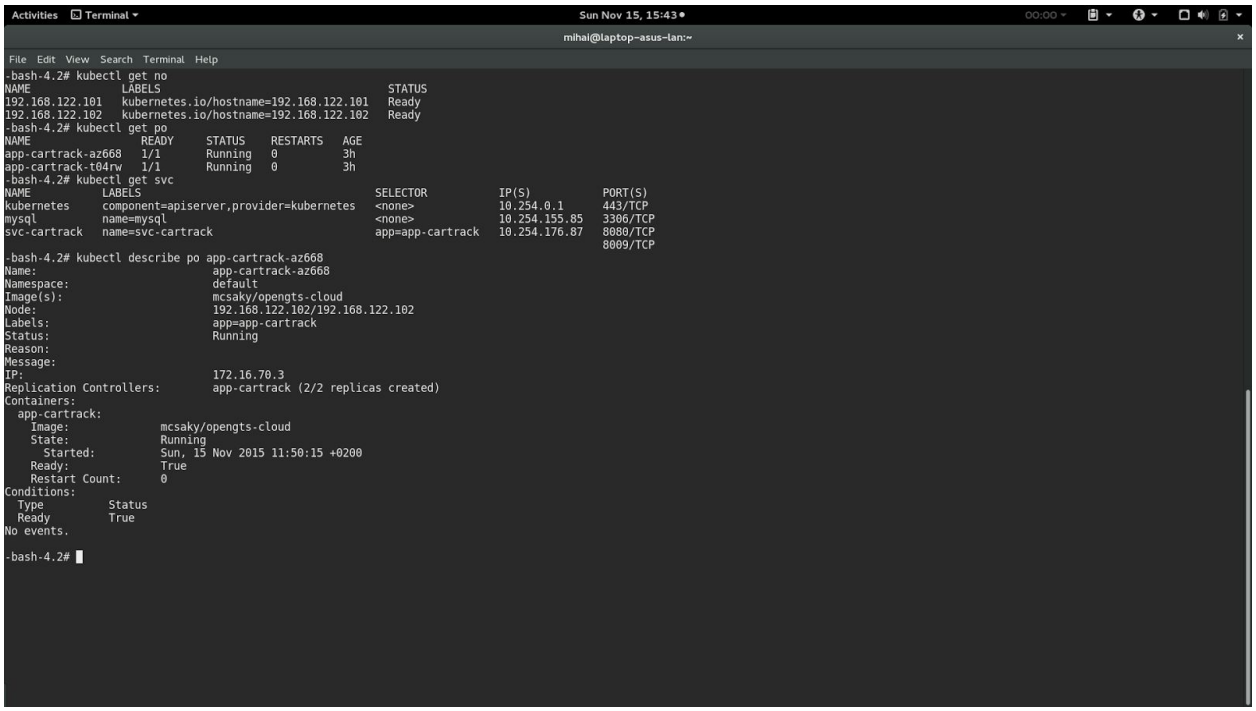
Ultima linie din script kubectl create -f opengts-rc-cartrack.yml generează pod-urile care rulează efectiv aplicaţia. Din acest punct, kubernetes se va ocupa de tot, va descărca imaginile necesare din docker hub, va distribui aplicaţia pe noduri şi va crea serviciile necesare accesării aplicaţiei. În funcţie de resurse, va trebui să aşteptăm poate şi peste 10 minute, dar în final vom avea două înstanţe ale aplicaţiei configurate şi rulând pe noduri. Continutul fişierului opengts-rc-cartrack.yml este:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: app-cartrack
  labels:
    name: app-cartrack
spec:
  replicas: 2
  selector:
    app: app-cartrack
  template:
    metadata:
      labels:
        app: app-cartrack
    spec:
      containers:
        - image: mcsaky/opengts-cloud
          name: app-cartrack
          env:
            - name: ETCD_SRV_ADDR
              value: 192.168.122.10
            - name: OPENGTS_CLIENT_ID
```

```
          value: cartrack
        ports:
        - containerPort: 8080
          name: http-cartrack
        - containerPort: 8009
          name: ajp-cartrack
-------------------------
apiVersion: v1
kind: Service
metadata:
  name: svc-cartrack
  labels:
    name: svc-cartrack
spec:
  ports:
  - port: 8080
    targetPort: 8080
    protocol: TCP
    name: http-cartrack
  - port: 8009
    targetPort: 8009
    protocol: TCP
    name: ajp-cartrack
  selector:
    app: app-cartrack
  type: NodePort
```

și definește resursele necesare rulării aplicației.

rezultatele se pot consulta pe master:



IX. Frontend

Aplicația noastră rulează în cluster, este distribuită automat și este accesibilă celorlalte aplicații în cluster, dar nu este accesibilă în exterior.

Pentru un cluster mic, cu puține aplicatii, cum este acesta, am putea rezolva simplu accesul din exterior cu ajutorul unor reguli de port-forward în firewall. Aceasta nu este însa o soluție elegantă, am ales să folosesc un load-balancer cu nginx și generarea automată a configurației cu confd.

De ce generare automată: kubernetes ne garantează că aplicația noastră va rula, că va reporni automat dacă se întâmplă ceva, că vor rula întotdeauna câte instanțe avem nevoie, dar nu ne garantează un IP fix al pod-ului și nici pe ce nod va rula. confd va trebui să determine orice schimbare în cluster și să anunțe serverul proxy nginx de aceste schimbări.