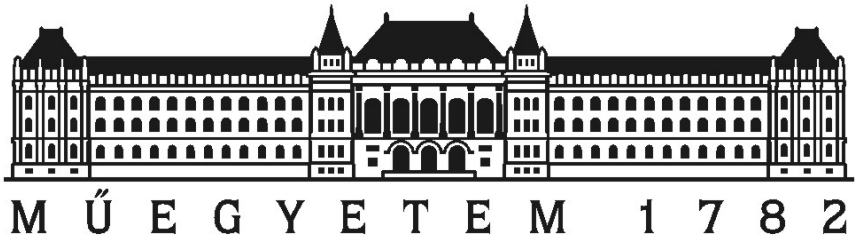


Háromrétegű architektúra megvalósítása docker konténerekben Kubernetes felett

Rába Tamás

CRLJQ8

Konzulens: Szeberényi Imre

Tartalomjegyzék

Megvalósítandó feladat leírása

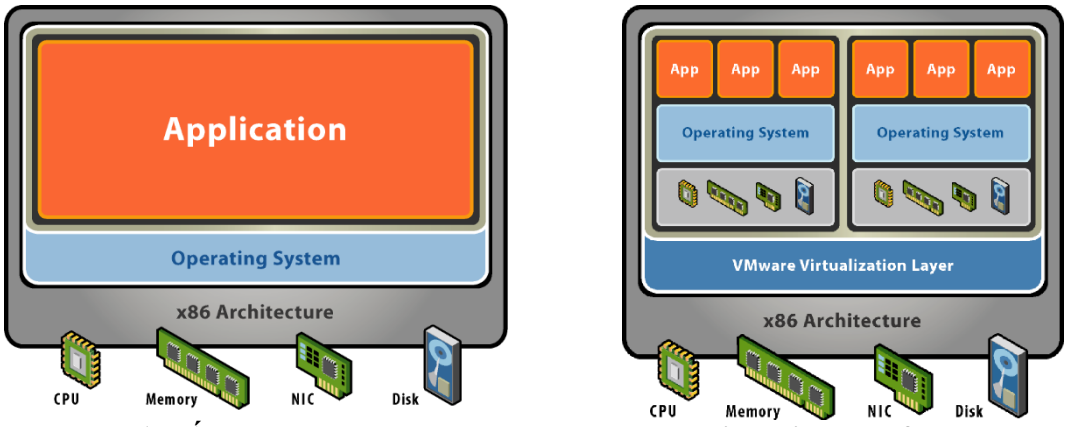
A félév során Python alapú MVC modellt (webserver, adatbázis és üzleti logika) követő konténerekben futó alkalmazást valósítottam meg. A webserverhez Python Flask könyvtárát használtam, az adatbázist pedig egy Redis konténer nyújtotta. Emellett a futtató alrendszert kiegészítettem úgy, hogy az alkalmazás bizonyos paramétereit (CPU, memória, stb..) képes legyen monitorozni. A futtató alrendszernek a Kubernetes-t választottam, mivel ez a legelterjedtebb konténer orchesztrációs megoldás. A monitorozáshoz a Kubernetes által biztosított metrikákat használtam. Ezeket a Grafana felületén megjelenítettem.

Virtualizációs technológiák

A munkám során egy egyszerű webalkalmazást valósítottam meg, melyhez különböző virtualizációs technológiákat használtam fel. Így virtuális gépeket és konténereket.

Virtuális gépek

Az alkalmazás-virtualizációra adott egyik megoldás a virtuális gépek bevezetése volt. Ebben az esetben több felhasználó futatthatja az alkalmazásait ugyanazon a fizikai számítógépen anélkül, hogy ezek az alkalmazások egymással interferálnának. Mivel az alkalmazások virtuális gépekben vannak izolálva [1]. Ennek egy következménye az is, hogy a régi egy alkalmazás per számítógép megoldásnál a virtuális gépek használata egy jobb erőforrás-kihasználtságot eredményez. A két megoldás különbségét az 1. Ábra szemlélteti.



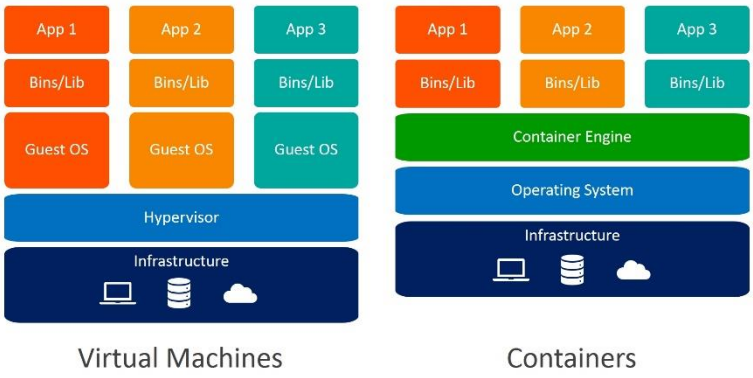
1. Ábra: Egy alkalmazás per számítógép vs Virtuális gépek futtatása

A VM-ek egy teljes értékű operációs rendszert (Operating System/OS) futtatnak, saját kernellel. Emiatt lehetőség van a gazda operációs rendszertől eltérő operációs rendszereket futtatni. A VM-ek egy úgynevezett VM Manager-ben (VMM) futnak. A munkám során kezdetben a VirtualBox VMM-et használtam.

Konténerek

Az alkalmazások virtualizálására adott másik megoldás a konténer alapú virtualizáció. A konténerek úgynevezett pehelysúlyú virtuális környezetek, ahol az esetek többségében egy konténerben csak egy alkalmazás fut. A konténerek esetén a virtuális környezet nem

rendelkezik saját kernellel, hanem a gazda OS saját kernelét használja (lásd 2. Ábra) [2]. Emiatt ebben az esetben a VM-ektől eltérő módon a virtuális környezet típusa meg kell, hogy egyezzen a gazda OS-sel. A legelterjedtebb konténer virtualizációs megoldás a Docker vagy az LXC. A munkám során én az előbbit használtam.



2. Ábra: VM-ek és konténerek közötti különbség

Kubernetes

Kubernetes egy olyan nyílt forráskódú felhő-orchestrációs rendszer, amely lehetővé teszi a Docker konténerek indítását, leállítását és egyéb menedzselését [3]. A Kubernetes számos erőforrást definiál, ezek közül csak a számomra legfontosabbakat érintem a dokumentációm során, melyekkel a példa alkalmazásomat meg tudtam valósítani egy Kubernetes környezetben.

Namespace

Kubernetesben különböző névtereket definiálhatunk, amelyek segítségével futtatott alkalmazásokat el tudjuk szeparálni.

Pod

A Pod legegyszerűbb Kubernetes által menedzselhető elem, amely virtualizációs környezetet futtat, egy vagy több konténer együttes futtatásáért felel. Az egy podban futó konténereket a Kubernetes mindig egy fizikai gépre ütemezi.

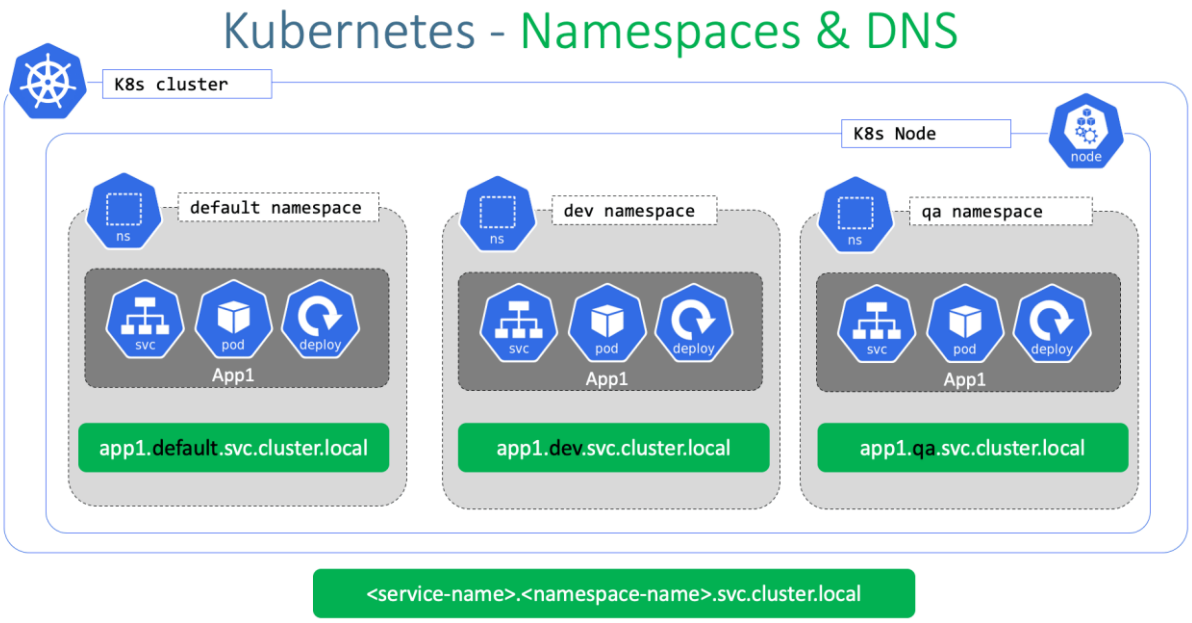
ReplicaSet és Deployment

Az azonos típusú Podok létrehozását és egyéb menedzselését a ReplicaSet teszi lehetővé. A ReplicaSet-eket használva a Kubernetes garantálja, hogy felhasználó definiált számú Pod az adott típusból mindig jelen legyen a rendszerben. Azonban a Deployment erőforrás számos hasznos kiegészítést ad a ReplicaSet-ekhez emiatt a valós környzetekben érdemes a ReplicaSet-ek helyett a Deployment erőforrást használni.

Service

A Service erőforrás hozzáférést biztosít a Kubernetesben futtatott hálózati alkalmazások számára. Egy Service komponens egy vagy több azonos típusú Podot rejt egy közös hozzáférési pont mögé. A Kubernetesben minden Pod egy saját IP címmel rendelkezik, amit induláskor kap, azonban a Pod ha valamilyen hiba miatt újraindul, akkor egy teljesen új IP címet fog kapni és így a Service használata nélkül minden hoozá csatlakozó alkalmazást újra kellene konfigurálni.

A Kubernetes rendszer rendelkezik egy DNS (Domain Name System) szolgáltatással, amely lehetővé teszi a Service-ek név szerinti elérését ezzel segítve az alkalmazáshalmazok telepítését és konfigurálását különböző Kubernetes rendszerekbe. A Kubernetes rendszer felépítését a 3. Ábra szemlélteti.



3. Ábra: Kubernetes logikai felépítése [4]

Monitorozó rendszerek

A monitorozó rendszerek segítségével vizsgálhatjuk rendszerünk számos paraméterét és bizonyos paraméterértékek alapján triggereket válthatunk ki, amelyek valamilyen módon befolyásolhatják a rendszert. Például autoskálázás bizonyos erőforrás paraméterek alapján, DDoS támadás felismerése.

Prometheus

A Prometheus egy nyílt forráskódú montírozó rendszer, amely idősorokat tárol és ezek alapján jelzéseket generál [5]. A Prometheus jól illeszkedik egy microservice alapú környezetbe, ahol különböző metrikákat tud gyűjteni a futtatott szolgáltatásokról. Továbbá egy lekérdező nyelvet is definiál, ez a PromQL, amivel a különböző adatokat tudjuk lekérni az adatbázisból. Mivel a Prometheus webes felülete nem rendelkezik felhasználóbarát funkciókkal, ezért a tárolt adatokat a munka során a Grafana felületén jelenítettem meg.

Grafana

A Grafana egy nyílt forráskódú interaktív adatvizualizáló platform, amely segítségével felhasználók különböző adatokat jeleníthetnek meg különböző grafikonokon [6].

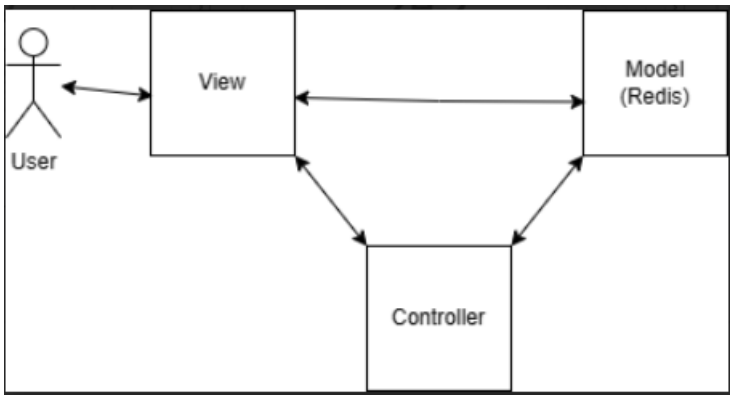
A munkám során a KubePrometheus programcsomagot használtam a megvalósított rendszer monitorozására.

A megvalósított alkalmazás

A félév során egy egyszerű háromrétegű alkalmazást valósítottam meg Python alapon Docker és Kubernetes környezetben.

Az alkalmazásban az én könyveimet tudom tárolni. A könyvek megtekintésére és hozzáadására egy webes interfészt hoztam létre.

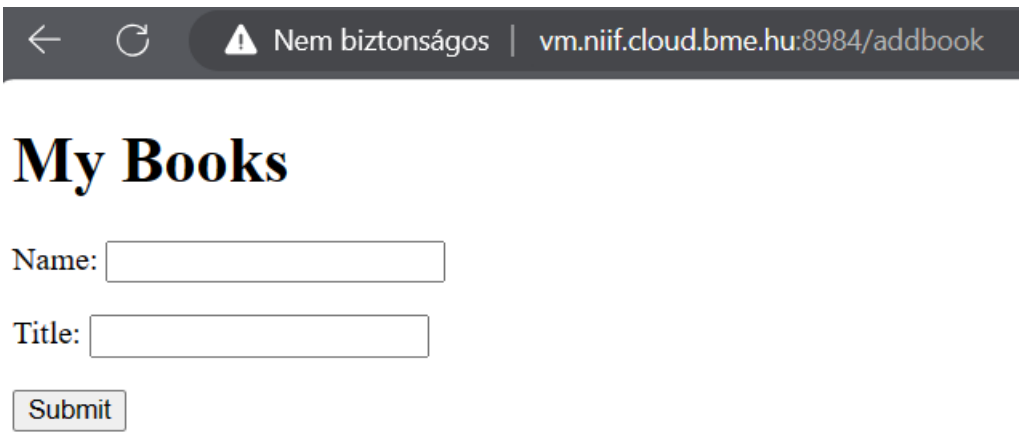
A feladat megvalósításához az MVC modellt követtem. A View komponens biztosítja a webes felületet, a Controller a bemeneti adatokat átalakítja az adatbázisban való tároláshoz. Míg a modellt egy Redis adatbázisra képeztem le. A megvalósított szolgáltatás kapcsolatait a 4. Ábra mutatja be.



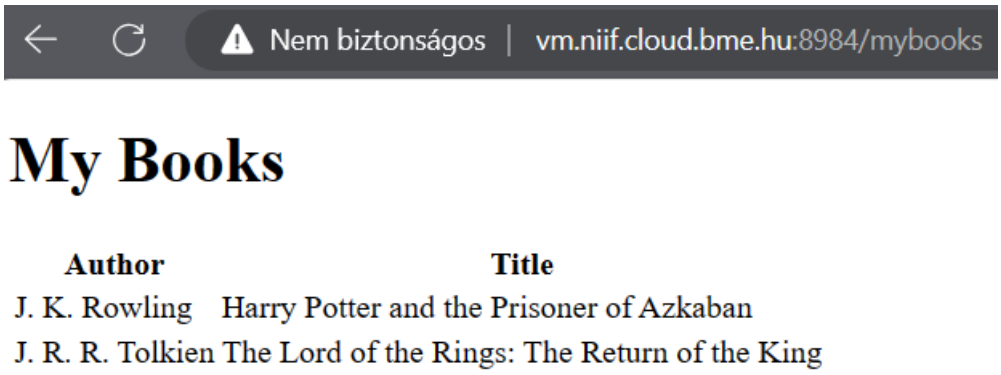
4. Ábra: A megvalósított alkalmazás architektúrája

A View és Controller elemeket a Python flask webes könyvtárával hoztam létre.

A View komponens két oldal elérését teszi lehetővé, ezek az addbook (5. Ábra) és mybooks (6. Ábra) oldalak, melyeket két külön elérési úton definiáltam. Az addbook oldalon tudok új könyvet felvenni, amely egy Submit gomb megnyomására a könyv adatait továbbítja a Controllernek, amely az adatokat JSON formátumban egy egyedi azonosítóval a Redis adatbázisba eltárolja. Sikeres tárolás esetén az addbook oldal átirányítódik a mybooks felületre, ahol megtekinthetem a felvitt könyveimet és azok adatait.



5. Ábra: A könyvek felvételéhez használt felület



6. Ábra: A könyvek listázásához használt felület

Az alkalmazás Dockerben történő futtatása a saját VM-ben

A megvalósított alkalmazás Docker-ben való futtatását egy VM-ben valósítottam meg, ehhez létrehoztam egy üres VM-et amelyre az Ubuntu 20.04-es szerver verzióját telepítettem. Ezt követően a vm-be feltelepítettem a Docker csomagot [7] a következő parancsokkal:

```
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

echo \
  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-
buildx-plugin docker-compose-plugin
```

A Docker konténerek futtatásához úgynevezett Docker Image-ekre van szükség, amelyeket Dockerfile-ok segítségével tudunk testreszabni. A Controller és View komponenseket a Python 3.9 Debian alapú konténerében futtattam, ehhez telepítettem a Redis és flask függőségeket és hozzáadtam a megvalósított kódot. A konténerek belépési pontja a Python alkalmazás futtattása.

A Redis adatbázis futtatásához a Redis által nyújtott Docker Image-et használtam módosítás nélkül.

Mivel a megvalósított szolgáltatáshoz a View elemen keresztül férünk hozzá, ezért elég csak a View-t futtató konténer portját nyilvánossá tenni. Viszont tesztelési megfontolásból a Redis hozzáférési portját is nyilvánossá tettem. A futatott konténerek és azok eléréséhez szükséges portok a 7. Ábrán láthatóak.

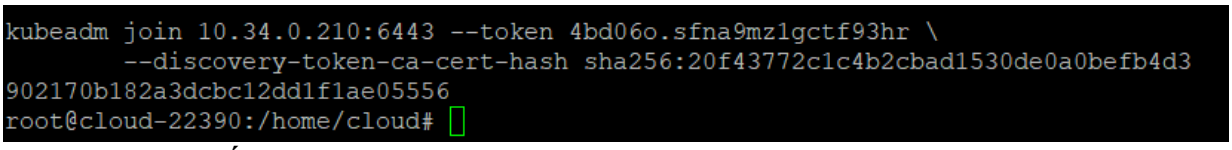
7. Ábra: Az alkalmazás komponensek konténerben való futása

Megvalósított alkalmazás Kubernetesben való futtatása

A Kubernetes telepítése

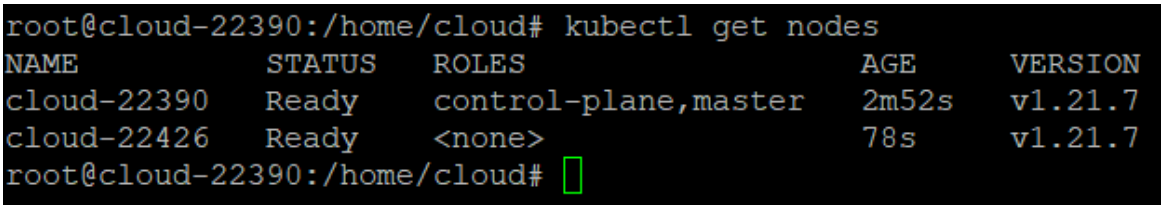
A megvalósított alkalmazást integráltam egy Kubernetes rendszerbe, melynek első lépéseként a konzulensemtől kapott Circle felhőben futó két VM-re feltelepítettem egy Kubernetes cluster-t. Első lépésként telepítettem a Docker virtualizációs környzetet. Ezt követően a Kubernetes szolgáltatásait, melyek a Kubelet, Kubeadm és Kubectl komponensek [8]. A telepítés követően az egyik VM-et kineveztem Controller node-nak a másikat pedig Worker-nek.

A Kubernetes node-ok egy kliens-szerver alapon csatlakoznak egymáshoz, ahol a Controller képviseli a szervert, a Worker-ek pedig a kliensek. Ennek megfelelően a Controller inicializálásakor a kubeadm init parancsot adtam meg, amely egy csatlakozási paranccsal tér vissza, amit a Worker node-on adtam ki. A csatlozási parancs a 8. Ábrán látható.



8. Ábra: A Controller node által generált csatlakozási parancs

Ennek hatására a Worker és a Controller összecsatlakozott, amit a kubectl get nodes paranccsal ellenőrizhetünk, ami a 9. Ábrán látható.



9. Ábra: A kubernetes node-ok listája

Végül egy virtuális pod hálózatot telepítettem, amelyet a flannel biztosít. Ezt követően a Kubernetes rednszer már működőképes, a 10. Ábrán szemléltetem a főbb névtereket és a főbb szolgáltatásokhoz tartozó podokat.

```
root@cloud-22390:/home/cloud# kubectl get namespaces
NAME              STATUS   AGE
default           Active   4m21s
kube-flannel      Active   23s
kube-node-lease   Active   4m22s
kube-public       Active   4m22s
kube-system       Active   4m22s
root@cloud-22390:/home/cloud# kubectl get pods -n kube-flannel
NAME                    READY   STATUS    RESTARTS   AGE
kube-flannel-ds-9pbxf   1/1     Running   0          59s
kube-flannel-ds-fqlwj   1/1     Running   0          59s
root@cloud-22390:/home/cloud# kubectl get pods -n kube-system
NAME                                     READY   STATUS    RESTARTS   AGE
coredns-558bd4d5db-66w2g                 1/1     Running   0          5m10s
coredns-558bd4d5db-78sxt                 1/1     Running   0          5m10s
etcd-cloud-22390                         1/1     Running   7          5m17s
kube-apiserver-cloud-22390               1/1     Running   7          5m18s
kube-controller-manager-cloud-22390      1/1     Running   3          5m18s
kube-proxy-q52p5                         1/1     Running   0          3m52s
kube-proxy-xp9l6                         1/1     Running   0          5m10s
kube-scheduler-cloud-22390               1/1     Running   7          5m18s
root@cloud-22390:/home/cloud#
```

10. Ábra: A Kubernetes főbb komponensei

Az alkalmazás Kubernetesben való futattása

A megvalósított webszolgáltatás a Kubernetesben való futtatásához létrehoztam különböző szolgáltatásleíró (yaml) fájlokat.

A komponensekhez tartozó yaml fájlokban definiáltam a Deployment és Service elemeket. A Deploymentek esetén megadtam, hogy melyik Docker Imaget indítsa el a Kubernetes és azt is, hogy mely porton kommunikál a futtatott alkalmazás. A Service-ek esetén pedig kiajánlottam a konténerek portjait. A View Service esetén a belső portot kívülről elérhetővé tettem, úgy hogy a Service típusának a LoadBalancer-t választottam. A 11. Ábra szemlélteti a komponenseim leíróit.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: view
spec:
  selector:
    matchLabels:
      app: view
  template:
    metadata:
      labels:
        app: view
    spec:
      containers:
      - name: view
        image: tamasraba/view
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: view
spec:
  selector:
    app: view
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 30362
  type: LoadBalancer
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: controller
spec:
  selector:
    matchLabels:
      app: controller
  template:
    metadata:
      labels:
        app: controller
    spec:
      containers:
      - name: controller
        image: tamasraba/controller
        ports:
        - containerPort: 8090
---
apiVersion: v1
kind: Service
metadata:
  name: controller
spec:
  selector:
    app: controller
  ports:
    - protocol: TCP
      port: 8090
      targetPort: 8090
  type: LoadBalancer
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:6.2.6
        ports:
        - containerPort: 6379
---
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  selector:
    app: redis
  ports:
    - protocol: TCP
      port: 6379
      targetPort: 6379
```

11. Ábra: Az alkalmazás komponensek leírói (View, Controller, Redis)

A Yaml fájlokban leírt komponenseket a kubectl apply parancs segítségével példányosítottam, lekérve a Deploymenteket, Service-ket és Pod-okat látható, hogy az alkalmazások elindultak Kubernetes környezetben, ezt a 12. Ábra mutatja.

```
root@cloud-22390:/home/cloud/controller# kubectl get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
controller   1/1     1            1           12s
redis        1/1     1            1           46s
view         1/1     1            1           27s
root@cloud-22390:/home/cloud/controller# kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
controller-849c557b59-gbxrw   1/1     Running   0          30s
redis-6db6859ddc-jl22r        1/1     Running   0          64s
view-6f696888bf-qpxpq         1/1     Running   0          45s
root@cloud-22390:/home/cloud/controller# kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
controller   LoadBalancer   10.110.83.151   <pending>     8090:31402/TCP   38s
kubernetes   ClusterIP      10.96.0.1       <none>        443/TCP          30m
redis        ClusterIP      10.99.134.150   <none>        6379/TCP         72s
view         LoadBalancer   10.109.8.221    <pending>     8080:30362/TCP   53s
root@cloud-22390:/home/cloud/controller#
```

12. Ábra: A futó alkalmazások és végpontjaik a Kubernetes rendszerben

A Circle-ben való futtatás anomáliái

Az alkalmazáshoz való hozzáféréskor problémába ütköztem, mivel a Controller node-on kiajánlott View alkalmazáshoz tartozó porton keresztül nem értem el az alkalmazás webes felületét. Emiatt arra következtettem, hogy a VM-ben definált tűzfal szabályok nem engedik a csatlakozást. Így a tűzfalon engedélyeztem az összes porthoz való csatlakozást a két gépen a másik gép IP címével. Ezt követően már elértem az alkalmazást.

Számos esetben találkoztam azzal a problémával, hogy a VM-et nem tudtam felélesztени csak újraindítani. Viszont az újraindításkor a Kubernetes node-okat újra kell inicializálni. Erre készítettem egy script-et, amely reseteli a Controller node-ot, újraindítja azt, frissíti a hozzáféréshez szükséges config fájlt és telepíti a pod-ok virtuális alhálózatát. A script visszatér az előzőekben látott csatlakozási paranccsal, amit a Worker node-on a reset után kell kiadni. A script a 13. Ábrán látható.

```
kubeadm reset -f
kubeadm init --pod-network-cidr=10.244.0.0/16

rm $HOME/.kube/config
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
kubectl apply -f \
https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel.yml
```

13. Ábra: A Controller node reseteléséhez használt script

A Kubernetes rendszer monitorozása

A megvalósított alkalmazás erőforrásainak monitorozására a Grafana által monitorozó felületet használtam. Ehhez a kube-prometheus [9] programcsomagot telepítettem. Viszont mivel nem a legújabb Kubernetes verzióval dolgoztam, ezért kikerestem, hogy mely verziójú kube-prometheus kompatibilis az én kubernetes verzómmal. Erre a 0.9-es verziót találtam a legalkalmasabbnak, mivel ez a legutolsó verzió, amely kompatibilis a 1.21-es Kubernetes-sel. A kube-prometheus kompatibilitási mátrixa a 14. Ábrán látható.

kube-prometheus stack	Kubernetes 1.18	Kubernetes 1.19	Kubernetes 1.20	Kubernetes 1.21	Kubernetes 1.22
release-0.6	✗	✔	✗	✗	✗
release-0.7	✗	✔	✔	✗	✗
release-0.8	✗	✗	✔	✔	✗
release-0.9	✗	✗	✗	✔	✔
HEAD	✗	✗	✗	✔	✔

14. Ábra: A kube-prometheus kompatibilitási mátrix [9]

A telepítéshez a következő parancsokat használtam:

```
git clone https://github.com/prometheus-operator/kube-prometheus.git
cd kube-prometheus/manifests/setup
kubectl apply -f ./
cd ..
kubectl apply -f ./
```

A 15. Ábrán láthatóak a kube-prometheus komponensei.

```
root@cloud-22390:/home/cloud/kube-prometheus/manifests# kubectl get pods -n monitoring
NAME                                   READY   STATUS    RESTARTS   AGE
alertmanager-main-0                    2/2     Running   0          24s
alertmanager-main-1                    2/2     Running   0          24s
alertmanager-main-2                    2/2     Running   0          24s
blackbox-exporter-6798fb5bb4-qw7cv     3/3     Running   0          24s
grafana-7476b4c65b-bn6s7               0/1     Running   0          23s
kube-state-metrics-74964b6cd4-vctnz    3/3     Running   0          23s
node-exporter-4p6hn                    2/2     Running   0          23s
node-exporter-brlxr                    2/2     Running   0          23s
prometheus-adapter-8587b9cf9b-7zmgz    1/1     Running   0          22s
prometheus-adapter-8587b9cf9b-n8kjt    1/1     Running   0          22s
prometheus-k8s-0                       2/2     Running   0          21s
prometheus-k8s-1                       2/2     Running   0          21s
prometheus-operator-75d9b475d9-wd28d   2/2     Running   0          35s
root@cloud-22390:/home/cloud/kube-prometheus/manifests#
```

15. Ábra: A kube-prometheus podjai

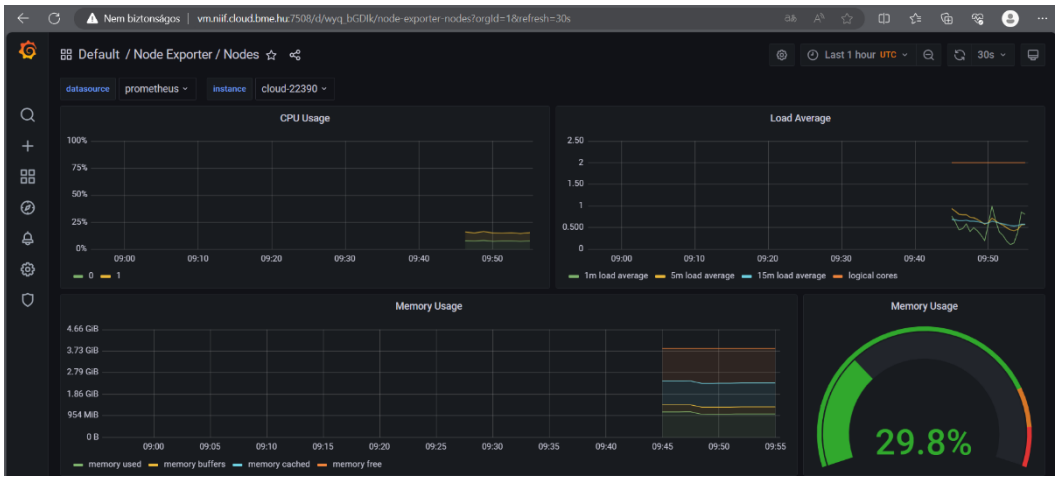
Eredetileg a Grafana Service nem érhető el kívülről, ehhez a manifests/grafana-service.yaml-t módosítottam úgy, hogy a Service típusa LoadBalancer lett. Így már kivülről is elérhető a Grafana webes felülete, 32000-es porton. Ezt az alábbi képen piros aláhúzással jelöltem a 16. Ábrán.

```
root@cloud-22390:/home/cloud/kube-prometheus/manifests# kubectl get services -n monitoring
NAME                    TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
alertmanager-main       ClusterIP      10.109.43.208    <none>        9093/TCP                     84s
alertmanager-operated   ClusterIP      None             <none>        9093/TCP,9094/TCP,9094/UDP   83s
blackbox-exporter       ClusterIP      10.105.75.235    <none>        9115/TCP,19115/TCP           83s
grafana                 LoadBalancer   10.109.181.255   <pending>     3000:32000/TCP               82s
kube-state-metrics      ClusterIP      None             <none>        8443/TCP,9443/TCP            82s
node-exporter           ClusterIP      None             <none>        9100/TCP                     82s
prometheus-adapter      ClusterIP      10.97.62.6       <none>        443/TCP                      81s
prometheus-k8s          ClusterIP      10.102.60.182    <none>        9090/TCP                     81s
prometheus-operated     ClusterIP      None             <none>        9090/TCP                     80s
prometheus-operator     ClusterIP      None             <none>        8443/TCP                     94s
root@cloud-22390:/home/cloud/kube-prometheus/manifests#
```

16. Ábra: A kube-prometheus által használt portok

Eredtileg VM-ek a 2GB memóriával rendelkeztek, viszont ez kevés volt a kube-rometheus számára és a prometheus-k8s podok pending állapotban ragadtak és a logok alpaján arra következtettem, hogy kevés a számomra elérhető memória. Kérvényeztem 4GB memóriát a VM-eimhez és ezt követően már el tudott indulni a monitorozó rendszer.

A 17. és 18. Ábrán látható, hogy a Grafana rendszer sikeresen megjeleníti a Prometheus által gyűjtött adatokat, továbbá az alkalmazásom komponenseit.



17. Ábra: A Grafana felülete (A Controller node erőforrás-használata)

18. Ábra: Az alkalmazásom komponensei a Grafana felületén

Terhelés-monitorozás:

A monitorozó rendszer és az alkalmazás tesztelésére egy HTTP terhelés generátort telepítettem, mellyel az alkalmazásom View komponens mybooks végpontját terheltem meg. A HTTP terhelés generátornak a Hey [10], Go nyelven írt nyílt forráskódú programot választottam. A Hey segítségével a felhasználó képes konkurens munkameneteket szimulálni egy HTTP szerver felé, ahol megválasztható a konkurenciaszint és a terhelés ideje.

A Hey program telepítéséhez a Go programozási nyelvhez tartozó csomagokat kellett installálnom, melyet a következő parancsokkal tettem meg:

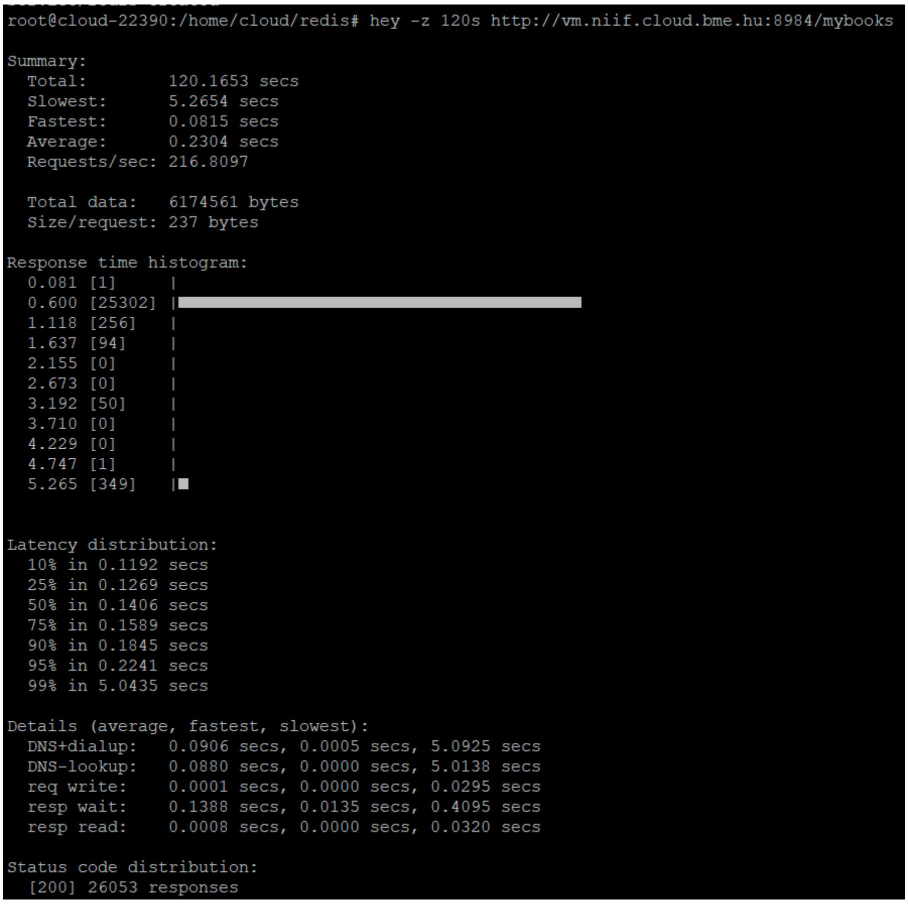
```
sudo add-apt-repository ppa:longsleep/golang-backports -y
sudo apt update -y
sudo apt install golang-go -y
```

A Hey ezt követően tudtam telepíteni a következő lépésekkel:

```
git clone https://github.com/rakyll/hey.git
cd hey
make
cp bin/hey_liny_amd64 /usr/bin/hey
```

Az alkalmazásom terhelésére 50-es konkurenciaszintet választottam és a terhelést 2 perc hosszúságúra állítottam.

Az ábrán látható, hogy a kérések 95%-a 0,22 másodperc alatt végzett, a medián pedig 1,4 másodperc az általam definiált terhelésre, ahogyan az a 19. Ábrán látható.

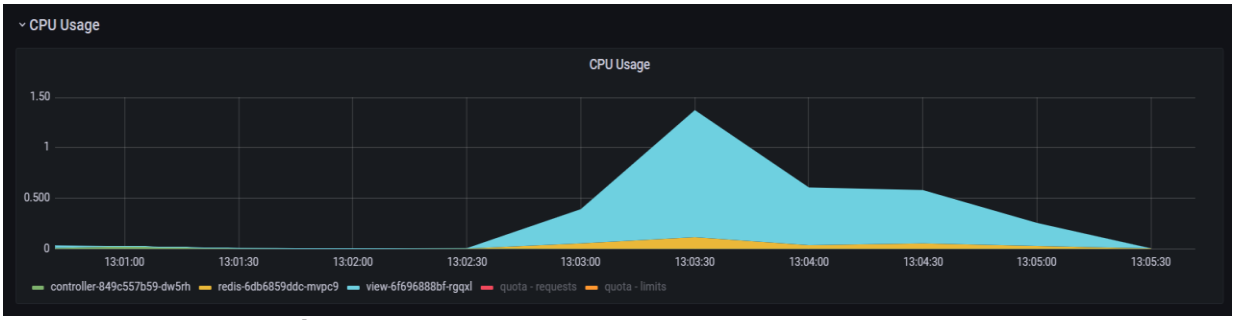


```
root@cloud-22390:/home/cloud/redis# hey -z 120s http://vm.niif.cloud.bme.hu:8984/mybooks

Summary:
  Total:        120.1653 secs
  Slowest:      5.2654 secs
  Fastest:      0.0815 secs
  Average:      0.2304 secs
  Requests/sec: 216.8097

  Total data:   6174561 bytes
  Size/request: 237 bytes

Response time histogram:
  0.081 [1]     |
  0.600 [25302] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  1.118 [256]   |
  1.637 [94]    |
  2.155 [0]     |
  2.673 [0]     |
  3.192 [50]    |
  3.710 [0]     |
  4.229 [0]     |
  4.747 [1]     |
  5.265 [349]   |■


Latency distribution:
  10% in 0.1192 secs
  25% in 0.1269 secs
  50% in 0.1406 secs
  75% in 0.1589 secs
  90% in 0.1845 secs
  95% in 0.2241 secs
  99% in 5.0435 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0906 secs, 0.0005 secs, 5.0925 secs
  DNS-lookup:   0.0880 secs, 0.0000 secs, 5.0138 secs
  req write:    0.0001 secs, 0.0000 secs, 0.0295 secs
  resp wait:    0.1388 secs, 0.0135 secs, 0.4095 secs
  resp read:    0.0008 secs, 0.0000 secs, 0.0320 secs

Status code distribution:
  [200] 26053 responses
```

19. Ábra: A Hey kimenete a terhelést követően

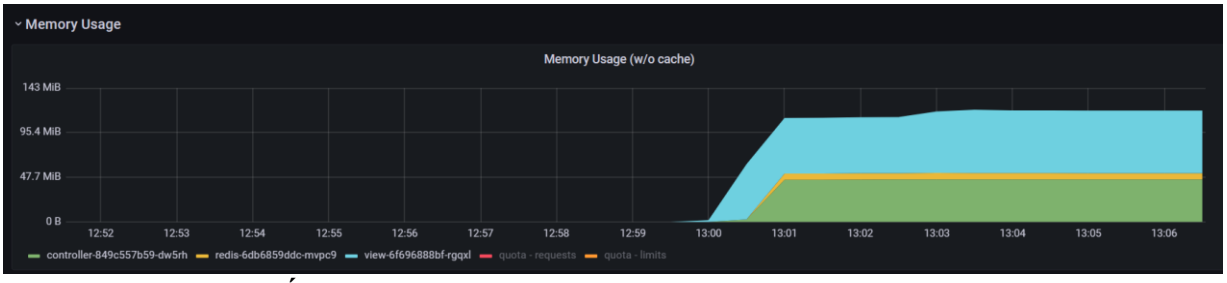
A terhelés alatt használt CPU erőforrások a 20. Ábrán láthatóak. Megfigyelhetjük, hogy a terhelés főként a View (kék szín) komponensre hatott és kis részben a Redis komponens CPU felhasználtsága is megnövekedett. Viszont mivel csak kiolvastunk az adatbázisból azért a

Controller komponens nem használt számítási erőforrásokat, ezért az ábrán sem jelenik meg hozzá tartozó erőforráshasználat.



20. Ábra: A CPU erőforrások vizsgálata a terhelés közben

Látható a 21. Ábrán, hogy az alkalmazások indításakor egy alap memóriamennyiséget foglalnak maguknak a programok. Mivel a terhelésmérést a bevitt könyvek lekérdezésére és nem pedig bevitelére végeztem, ezért a memória-felhasználás nem változott a terhelés során.



21. Ábra: A memória erőforrás vizsgálata a terhelés közben

Hivatkozásjegyzék:

1. SAP, A brief Intro of virtualization and vmware esxi, 2014, https://blogs.sap.com/2014/05/28/a-brief-intro-of-virtualization-and-vmware-esxi/

2. WeaveWorks, Docker vs Virtual Machines (VMs) : A Practical Guide to Docker Containers and VMs, 2020, https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms

3. Kubernetes Documentation, https://kubernetes.io/docs/

4. StackSimplify, Kubernetes Namespaces - Imperative using kubectl, https://stacksimplify.com/azure-aks/azure-kubernetes-service-namespaces-imperative/

5. Prometheus Documentation, https://prometheus.io/docs/introduction/overview/

6. RedHat What is Grafana https://www.redhat.com/en/topics/data-services/what-is-grafana

7. Docker, Install Docker Engine on Ubuntu, https://docs.docker.com/engine/install/ubuntu/

8. PhoenxNap, How to Install Kubernetes on Ubuntu 20.04, https://phoenixnap.com/kb/install-kubernetes-on-ubuntu

9. kube-prometheus, https://github.com/prometheus-operator/kube-prometheus

10. Hey, https://github.com/rakyll/hey