

Napredni operativni sistemi

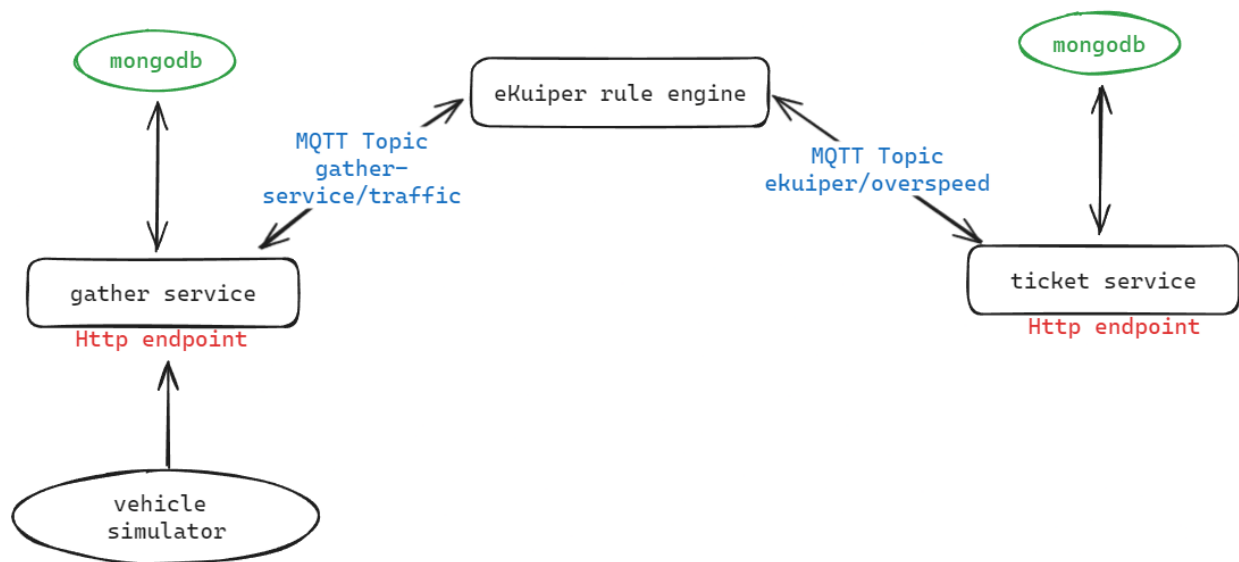
III Projekat

Student:

Vukadin Drašković 1613

Arhitektura sistema

Arhitektura sistema je prikazana na slici 1. Kratki opis arhitekture je sledeći: *vehicle simulator* je *python* skripta koja ima za cilj da HTTP POST metodom pošalje podatke *gather* servisu, koji će te podatke smestiti u svoju bazu podataka i pored toga će ih poslati na *gather-service/traffic* MQTT topic na analizu. Podaci koje sam koristio u projektu su preuzeti iz mog diplomskog rada, koji se može naći na mom *github* nalogu. Ekuiper servis preuzima podatke sa MQTT topic-a na koji je *gather* servis poslao podatke, i detektuje vozila koja se kreću brže nego što je dozvoljeno. Rezultate analize šalje na *ekuiper/overspeed* MQTT topic, odakle ih preuzima *ticket* servis, koji ih skladišti u svoju bazu podataka. *Gather* servis HTTP GET metodama omogućava klijentima pretragu podataka iz svoje baze podataka na osnovu *vehicleId*, *vehicleLane* i *timestepTime* parametara, dok *ticket* servis, pored pretrage po ovim parametrima, omogućava i pretragu po *vehicleSpeed* parametru.



Slika 1. Arhitektura sistema

Sve komponente, osim *vehicle* simulatora su kontejnerizovane, jer on predstavlja eksternu komponentu ovog sistema koja egzistira nezavisno od ostalih (konkretno, svaki podatak koji šalje *vehicle* simulator predstavlja jedno vozilo koje šalje podatke *gather* servisu).

Implementacija sistema

1. *Vehicle* simulator

Simulator vozila je predstavljen *python* skriptom **vehicles.py** koja čita red po red podataka iz **traffic.csv** fajla i svaki red šalje *gather* servisu pomoću HTTP POST metode. Prilikom slanja je stavljen određeni timeout, kako bi mogao da se prati rad sistema „korak po korak“. Takođe, kasnijem smanjenjem tog timeout-a ćemo demonstrirati skaliranje sistema kada se dođe do Kubernetesa. Na slici 2 je prikazan kod skripte.

```
1  import csv
2  import time
3  import requests
4
5  file_path = './traffic.csv'
6
7  def format_row(row: dict):
8      for key in row:
9          if key == 'vehicle_id' or key == 'vehicle_type' or key == 'vehicle_lane':
10             continue
11             row[key] = float(row[key])
12
13     return row
14
15  with open(file_path, 'r') as file:
16      csv_reader = csv.DictReader(file, delimiter=';')
17      vehicle_values = []
18      for row in csv_reader:
19          row = format_row(row)
20          requests.post("http://localhost:8000/", json=row)
21          time.sleep(5)
```

Slika 2. Kod vehicles.py skripte

2. *Gather* servis

Ovaj servis je implementiran u *node.js*-u, odnosno u *express framework*-u. Izvorni kod aplikacije se nalazi u *index.js* fajlu. On nudi HTTP POST *endpoint* preko koga će mu simulator slati podatke, koje će on smeštati u bazu podataka i slati na *MQTT Topic*.

Baza podataka koju servis koristi je *MongoDB*, i u tu svrhu je instaliran paket *mongoose* koji će na efikasan način obavljati komunikaciju sa bazom podataka.

Da bi servis mogao da radi se *MongoDB*-jem, neophodno je da prvo kreira model, odnosno šemu podataka koji želi da koristi u radu sa bazom, i da ga kao takvog registruje u *MongoDB*. Taj deo je prikazan na slici 3.

```

103 const trafficSchema = new mongoose.Schema({
104   timestepTime: Number,
105   vehicleAcceleration: Number,
106   vehicleAngle: Number,
107   vehicleDistance: Number,
108   vehicleId: String,
109   vehicleLane: String,
110   vehiclePos: Number,
111   vehicleSignals: Number,
112   vehicleSlope: Number,
113   vehicleSpeed: Number,
114   vehicleType: String,
115   vehicleX: Number,
116   vehicleY: Number
117 })
118
119 const Traffic = mongoose.model('Traffic', trafficSchema)

```

Slika 3. *Traffic* model za rad sa *MongoDB*-jem

Takođe, zbog slanja podataka na *MQTT topic*, servis se mora registrovati kao *MQTT client*, i u tu svrhu je korišćen paket *mqtt* iz *npm*-a.

Na slici 4 se može videti implementacija HTTP POST metode na koju simulator šalje podatke, dok se na slici 5 vide HTTP GET metode koje nudi servis, konkretno za pretragu podataka po identifikacionom broju vozila, po ulici u kojoj su se vozila kretala i po trenutku slanja podataka. Servis osluškuje HTTP zahteve na portu 8000.

```

39 app.post('/', function (req, res) {
40   console.log(req.body)
41   const obj = new Traffic({
42     timestepTime: req.body.timestep_time,
43     vehicleAcceleration: req.body.vehicle_acceleration,
44     vehicleAngle: req.body.vehicle_angle,
45     vehicleDistance: req.body.vehicle_distance,
46     vehicleId: req.body.vehicle_id,
47     vehicleLane: req.body.vehicle_lane,
48     vehiclePos: req.body.vehicle_pos,
49     vehicleSignals: req.body.vehicle_signals,
50     vehicleSlope: req.body.vehicle_slope,
51     vehicleSpeed: req.body.vehicle_speed,
52     vehicleType: req.body.vehicle_type,
53     vehicleX: req.body.vehicle_x,
54     vehicleY: req.body.vehicle_y
55   })
56   obj.save()
57   .then(
58     () => console.log("Added to db"),
59     (err) => console.log(err)
60   )
61
62   mqttClient.publish(topic, JSON.stringify(req.body), { qos }, error => {
63     //console.log("monitoring-service sending: ", req.body)
64     if (error) {
65       console.error('ERROR: ', error)
66       res.sendStatus(400)
67     }
68     res.sendStatus(200)
69   })
70 });

```

Slika 4. HTTP POST metoda

```

72 app.get('/vehicleLane/:vehicleLane', (req, res) => {
73   Traffic.find({
74     vehicleLane: req.params.vehicleLane
75   }).then(data => {
76     res.send(data);
77   }).catch(err => console.log("Error occured, " + err));
78 })
79
80 app.get('/vehicleId/:vehicleId', (req, res) => {
81   Traffic.find({
82     vehicleId: req.params.vehicleId
83   }).then(data => {
84     res.send(data);
85   }).catch(err => console.log("Error occured, " + err));
86 })
87
88 app.get('/timestepTime/:timestepTime', (req, res) => {
89   Traffic.find({
90     timestepTime: req.params.timestepTime
91   }).then(data => {
92     res.send(data);
93   }).catch(err => console.log("Error occured, " + err));
94 })

```

Slika 5. HTTP GET metode koje nudi servis

3. Ticket servis

Ovaj servis je veoma sličan gather servisu, tako da ćemo u opisu njegove implementacije navesti samo razlike u odnosu na gather servis. Za razliku od gather servisa, drugačiji mu je model, ne smešta u bazu podataka sve podatke koje je poslao vozilo, već samo one koji su od interesa za tiket o prekoračenoj brzini, tj. koja je to brzina kojom se vozilo kretalo, koje je to vozilo, u kojoj ulici i u kom trenutku se kretalo. Model je prikazan na slici 6.

```

97 const ticketSchema = new mongoose.Schema({
98   timestepTime: Number,
99   vehicleId: String,
100   vehicleLane: String,
101   vehicleSpeed: Number
102 })
103
104 const Ticket = mongoose.model('Ticket', ticketSchema)

```

Slika 6. *Ticket* model korišćen u *ticket* servisu

Pored toga, za razliku od *gather* servisa, *ticket* servis treba da se pretplati na *MQTT topic* sa koga će preuzeti podatke u vozilima koja su prekoračila brzinu. *Subscribe*-ovanje na *ekuiper* topik i definisanje ponašanja servisa prilikom pristizanja poruke je prikazano na slici 7.

```

28 mqttClient.on('connect', () => {
29   console.log('Connected')
30   mqttClient.subscribe(topic, () => {
31     console.log(`Subscribed to topic '${topic}'`)
32   })
33 })
34
35 mqttClient.on('message', (topic, payload) => {
36   json = JSON.parse(payload.toString())
37   console.log(`Received Message from '${topic}': `, json.vehicle_speed)
38   const obj = new Ticket({
39     timestepTime: json.timestep_time,
40     vehicleId: json.vehicle_id,
41     vehicleLane: json.vehicle_lane,
42     vehicleSpeed: json.vehicle_speed
43   })
44   obj.save()
45     .then(
46       () => console.log("Added to db"),
47       (err) => console.log(err)
48     )
49 })

```

Slika 7. Postavljanje *MQTT* klijenta u *ticket* servisu

Servis osluškuje HTTP zahteve na portu 8001.

Pokretanje sistema *Docker compose*-om

Kako bi uklonili zavisnosti našeg operativnog sistema, i postavka našeg računara od sistema koji implementiramo, najbolje bi bilo da se koristi platforma *Docker* koja ima za cilj da kontejnerizuje naš sistem i njene komponente, kako bi sistem mogao da se pokrene i na drugim računarima bez problema. *Docker* kontejner predstavlja izolovano okruženje u kome se izvršava naša aplikacija. Kreira se na osnovu *Docker* slike, koja se može preuzeti sa javnog *Docker Hub* repozitorijuma, ili je možemo samoinicijativno kreirati na osnovu *Dockerfile*-a, ukoliko želimo da kreiramo sliku za aplikaciju koju smo sami kreirali. Na osnovu prethodnog, moraćemo da kreiramo *Dockerfile* za naša dva servisa, dok ćemo za ostale komponente sistema direktno preuzimati slike iz javnog *Docker Hub* repozitorijuma. U svrhu pokretanja celog sistema odjednom, u izolovanom okruženju, za koje će se kreirati virtuelna mreža preko koje će moći da komuniciraju navođenjem naziva servisa, umesto navođenja njihove IP adrese, pokretanje sistema će biti prikazano *Docker compose*-om.

Dockerfile za *gather* servis je identičan *Dockerfile*-u za *ticket* servis. Naime, iz *node* slike sa javnog *Docker Hub* repozitorijuma, želimo da kreiramo sliku naše aplikacije. Da bi instalirali neophodne pakete za rad aplikacije, neophodno je da se iskopiraju **package.json** i **package-lock.json** fajlovi u radni direktorijum kontejnera koji će se kreirati, kako bi nakon toga mogli da upravo komandom *npm install* instaliramo sve neophodne pakete za rad naše aplikacije. Nakon toga, potrebno je da kopiramo ostale fajlove i foldere naše aplikacije u radni direktorijum kontejnera, da otvorimo port 8000 na korišćenje (kod *Dockerfile*-a za *ticket* servis će to biti port 8001), i izvršimo komandu *node index.js* kako bi se aplikacija pokrenula. Ovim je ukratko opisan *Dockerfile* koji je prikazan na slici 8.

```
1 FROM node
2 WORKDIR /usr/src/app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 8000
7 CMD ["node", "index.js"]
```

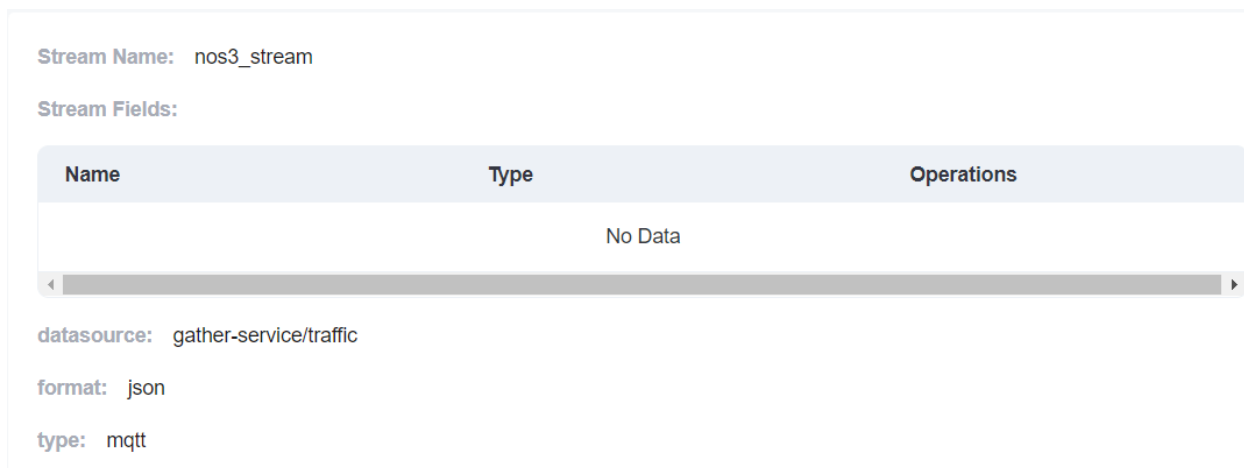
Slika 8. *Dockerfile* *gather* servisa

Sadržaj **docker-compose.yml** fajla predstavlja skup svih aplikacija, odnosno servisa koji čine naš sistem. Važno je napomenuti da se za *emqx* (kontejner koji predstavlja *MQTT* broker) koristi *healthcheck*, kako bi ostali servisi koji koriste ovaj broker bili pokrenuti nakon što se on pokrene, da ne bi došlo do greške prilikom podizanja tih servisa. Takođe, za bazu podataka je korišćen kontejner definisan *mongo* slikom, koji podrazumevano osluškuje zahteve na portu 27017. Kako u našem sistemu imamo 2 baze podataka, ne mogu obe osluškivati na portu 27017, tako da je to

prevaziđeno komandom *mongod --port 27018* na kome će osluškivati baza namenjena *ticket* servisu. Na kraju je potrebno navesti volume koji će se kreirati za potrebe *ekuiper* servisa.

Ekuiper manager servis se podiže isključivo kako bi se konfigurisao *ekuiper*, tj. kako bi se navelo pravilo po kome će se detektovati prekoračenje brzine kretanja vozila.

Sistem se pokreće izvršavanjem *docker compose up* komande u folderu u kome se nalazi **docker-compose.yml** fajl. Nakon prvog pokretanja sistema, neophodno je definisati pravilo u *ekuiper*-u za detektovanje prekoračenje brzine. Za to je potrebno otići u web pretraživač i otići na adresu *localhost:9082*, logovati se kredencijalima **admin** i **public**, i kreirati *stream* prikazan na slici 9 koji će preuzimati podatke sa *gather-service/traffic* MQTT topika.



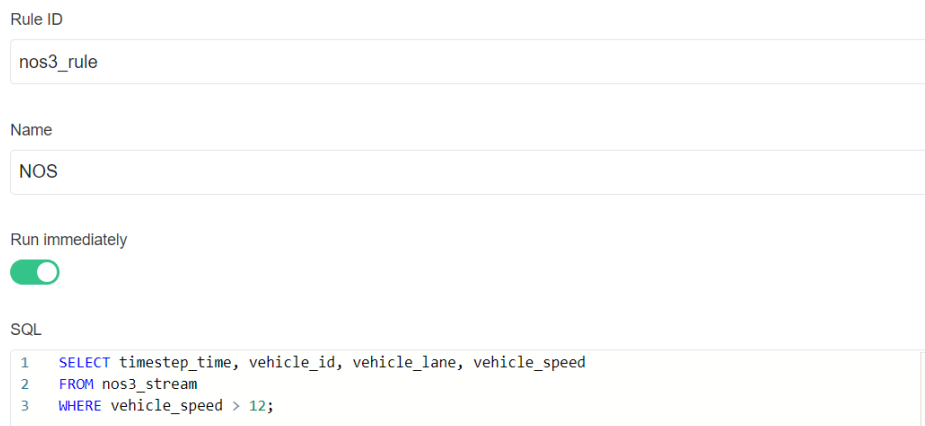
The screenshot shows the Ekuiper web interface for configuring a stream. At the top, 'Stream Name' is set to 'nos3_stream'. Below it, 'Stream Fields' is shown as an empty table with columns 'Name', 'Type', and 'Operations', displaying 'No Data'. At the bottom, the configuration is summarized: 'datasource: gather-service/traffic', 'format: json', and 'type: mqtt'.

Name	Type	Operations
No Data		

datasource: gather-service/traffic
format: json
type: mqtt

Slika 9. Ekuiper stream

Nakon toga je potrebno definisati pravilo prikazano na slici 10. Kako bi se opisalo šta će se desiti kada dođe do prekoračenja brzine, potrebno je definisati akciju koja će slati podatke na *mqtt://emqx:1883* broker, odnosno na *ekuiper/overspeed* topik (prikazano na slici 11).



The screenshot shows the Ekuiper web interface for configuring a rule. 'Rule ID' is 'nos3_rule' and 'Name' is 'NOS'. The 'Run immediately' toggle is turned on. The 'SQL' field contains a query to select data from 'nos3_stream' where 'vehicle_speed' is greater than 12.

Rule ID: nos3_rule
Name: NOS
Run immediately: ☒
SQL:
1 SELECT timestep_time, vehicle_id, vehicle_lane, vehicle_speed
2 FROM nos3_stream
3 WHERE vehicle_speed > 12;

Slika 10. Jednostavno pravilo za detektovanje prekoračenja brzine

View action

* Sink [Documentation](#)

mqtt

Resource ID [+ Add sink template](#)

Select

MQTT broker address ? MQTT topic ?

mqtt://emqx:1883 ekuiper/overspeed

Slika 11. Akcija prilikom ispunjenja uslova iz pravila

Sada, kada smo pokrenuli sistem, možemo pokrenuti simulator izvršenjem komande *python vehicle.py* u *vehicle-simulator* direktorijumu.

Za demonstraciju rada sistema je korišćen alat *Postman*. Za preuzimanje svih podataka sa servisa *gather* koje je poslalo vozilo sa identifikatorom *veh0*, prikazan je rezultat HTTP GET upita na slici 12. Za preuzimanje tiketa kod kojih se vozilo kretalo brzinom između 15 i 25 jedinica mere, izvršen je HTTP GET upit prikazan na slici 13.

http://localhost:8000/vehicleId/veh0

GET http://localhost:8000/vehicleId/veh0 Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (7) Test Results

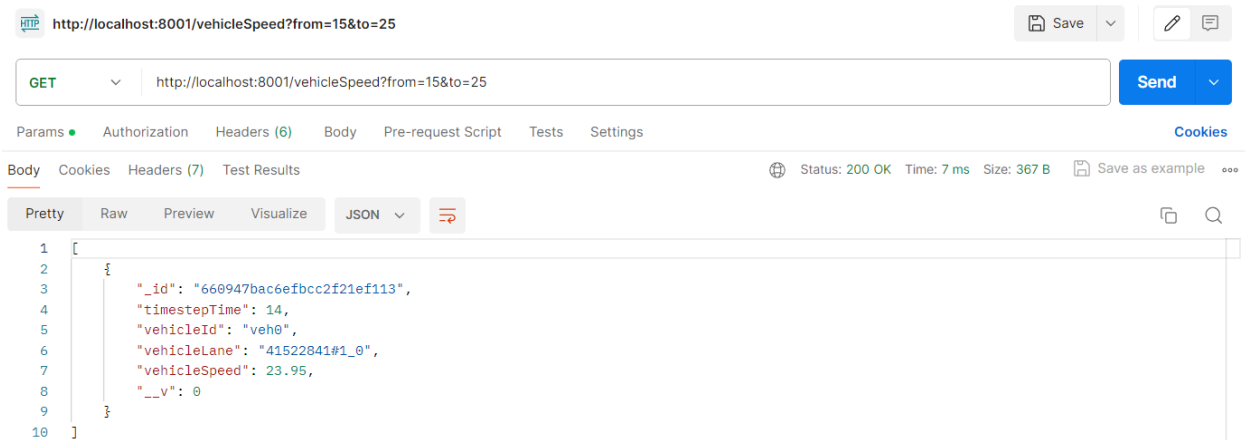
Status: 200 OK Time: 20 ms Size: 1.82 KB Save as example

```

1 {
2   {
3     "_id": "660945ed27f16c8d51c63736",
4     "timestamp": 0,
5     "vehicleAcceleration": 0,
6     "vehicleAngle": 22.2,
7     "vehicleDistance": 5.1,
8     "vehicleId": "veh0",
9     "vehicleLane": "154717187#0_0",
10    "vehiclePos": 5.1,
11    "vehicleSignals": 0,
12    "vehicleSlope": 0,
13    "vehicleSpeed": 0,
14    "vehicleType": "veh_passenger",
15    "vehicleX": 21.982531,
16    "vehicleY": 43.323934,
17    "__v": 0
18  },
19  {
20    "_id": "660945fd27f16c8d51c6373c",
21    "timestamp": 1,
22    "vehicleAcceleration": 1.38,
23    "vehicleAngle": 22.2,
24    "vehicleDistance": 6.48,
25    "vehicleId": "veh0",
26    "vehicleLane": "154717187#0_0",
27    "vehiclePos": 6.48,

```

Slika 12. Izvršenje HTTP GET upita nad *gather* servisom



Slika 13. Izvršenje HTTP GET upita nad ticket servisom

Pokretanje i upravljanje sistema *Kubernetes*-om

Kubernetes je platforma otvorenog koda namenjena automatizovanom upravljanju kontejnerizovanih aplikacija i skaliranju istih. Pod je najmanja komponenta unutar Kubernetes klastera, i predstavlja jedinicu izvršavanja kontejnerizovane/ih aplikacija, zato što se unutar jednog poda može izvršavati više kontejnera. Kako bi pod mogao da komunicira sa drugim kontejnerima, neophodno je da se za pod kreira servis. Servis može biti interni (*default type ClusterIP*), ukoliko ne treba da bude izložen spoljašnjoj sredini, i može biti eksterni (*type LoadBalancer*) ukoliko kontejner treba da omogući da sa njim komuniciraju aplikacije koje se nalaze van klastera. Važno je napomenuti da i ClusterIP tip servisa radi balansiranje opterećenja određenog Deployment-a, iako postoji poseban tip servisa koji se zove *LoadBalancer*. Svaki pod se u kubernetesu kreira pomoću *Deployment*-a, koji predstavlja apstrakciju poda. U Deployment-u navodimo iz koje slike želimo da pokrenemo kontejnere unutar poda, kao i koliko replika podova želimo. Takođe, ukoliko aplikacija koristi neki volume za smeštanje podataka, neophodno je kreirati i *PersistentVolumeClaim* na osnovu koga će se iz klastera zahtevati određeni deo memorije, kako bi aplikacija mogla da funkcioniše.

U *kubernetes* folderu projekta se nalaze svi .yaml fajlovi (ekstenzija koja se koristi kod pisanja *Deployment*, *Service* i *PersistentVolumeClaim* fajlova), i u svakom fajlu se nalazi definisan *PersistentVolumeClaim* ukoliko je to zahtev kontejnera, a nakon toga idu potpisi *Deploymenta* i servisa. Servisi koji su kreirani kao eksterni servisi su *ticket* servis, *gather* servis i *ekvipier manager* servis koji služi za konfigurisanje pravila za detektovanje prekoračenja u brzini.

Da bismo koristili kubernetes lokalno, na našem računaru, neophodno je da instaliramo *minikube* koji će kreirati virtuelni kubernetes klaster na našem računaru, na kome ćemo moći da izvršavamo podove i servise. Pored njega, neophodno je instalirati i *kubectl* paket koji će nam omogućiti upravljanje kubernetes klasterom. Komandom *minikube start* pokrećemo naš kubernetes klaster. Nakon toga, kako bismo mogli da koristimo lokalne Docker slike u našem Kubernetes klasteru, neophodno je da uradimo sledeće: da izvršimo komande *minikube docker-env* i *@FOR /f "tokens=*" %i IN ('minikube -p minikube docker-env --shell cmd') DO @%i*, kako bi smo postavili neophodne promenljive okruženja u našu terminal sesiju iz našeg Docker-a koje će minikube koristiti. Nakon toga komandom *minikube image ls --format table* se možemo uveriti da ih nema naše slike za *ticket* i *gather* servis (prikazano na slici 14). Izvršavanjem komande *docker build -t nos3-gather-service -f .\Microservices\gather-service\Dockfile .\Microservices\gather-service* iz korenskog direktorijuma projekta se blduje slika za *gather* servis, a analogno njoj je potrebno izvršiti komandu i za kreiranje slike za *ticket* servis. Sada se možemo, sa slike 15, uveriti da se u našem minikube klasteru nalaze slike za podizanje *ticket* i *gather* servisa.

Da bismo pokrenuli podove u našem klasteru, neophodno je da u *kubernetes* folderu izvršimo komandu prikazanu na slici 16.

```
C:\Users\vukad\Storage\Fakultet MAS\1. Semestar\5 Napredni operativni sistemi\NOS3\vehicle-simulator>minikube image ls --format table
W0331 17:13:37.618383 20616 main.go:291] Unable to resolve the current Docker CLI context "default": context "
default": context not found: open C:\Users\vukad\.docker\contexts\meta\37a8eec1ce19687d132fe29051dca629d164e2c49
58ba141d5f4133a33f0688f\meta.json: The system cannot find the path specified.
```

Image	Tag	Image ID	Size
docker.io/emqx/ekuiper-manager	latest	d82921a91caab	163MB
registry.k8s.io/kube-proxy	v1.28.3	bfc896cf80fba	73.1MB
registry.k8s.io/coredns/coredns	v1.10.1	ead0a4a53df89	53.6MB
registry.k8s.io/pause	3.9	e6f1816883972	744kB
docker.io/library/<none>	<none>	91bd7d1cd38d2	1.14GB
gcr.io/k8s-minikube/storage-provisioner	v5	6e38f40d628db	31.5MB
docker.io/lfedge/ekuiper	latest	9252378436177	72.7MB
docker.io/library/emqx	latest	e9772536c9ef3	279MB
registry.k8s.io/kube-controller-manager	v1.28.3	10baa1ca17068	122MB
registry.k8s.io/etcd	3.5.9-0	73deb9a3f7025	294MB
docker.io/library/<none>	<none>	1711e40258bac	1.15GB
registry.k8s.io/kube-apiserver	v1.28.3	5374347291230	126MB
registry.k8s.io/kube-scheduler	v1.28.3	6d1b4fd1b182d	60.1MB
docker.io/library/mongo	latest	24041ceefc56c	755MB

Slika 14. Izlaz komande *minikube image ls* pre build-ovanja slika

```
C:\Users\vukad\Storage\Fakultet MAS\1. Semestar\5 Napredni operativni sistemi\NOS3>minikube image ls --format table
```

Image	Tag	Image ID	Size
registry.k8s.io/kube-scheduler	v1.28.3	6d1b4fd1b182d	60.1MB
docker.io/library/<none>	<none>	1711e40258bac	1.15GB
docker.io/emqx/ekuiper-manager	latest	d82921a91caab	163MB
docker.io/library/emqx	latest	e9772536c9ef3	279MB
registry.k8s.io/kube-controller-manager	v1.28.3	10baa1ca17068	122MB
registry.k8s.io/etcd	3.5.9-0	73deb9a3f7025	294MB
registry.k8s.io/pause	3.9	e6f1816883972	744kB
docker.io/library/nos3-gather-service	latest	9513c08f6950d	1.14GB
docker.io/lfedge/ekuiper	latest	9252378436177	72.7MB
docker.io/library/mongo	latest	24041ceefc56c	755MB
registry.k8s.io/kube-proxy	v1.28.3	bfc896cf80fba	73.1MB
gcr.io/k8s-minikube/storage-provisioner	v5	6e38f40d628db	31.5MB
docker.io/library/nos3-ticket-service	latest	e88af2b748af9	1.15GB
docker.io/library/<none>	<none>	91bd7d1cd38d2	1.14GB
registry.k8s.io/kube-apiserver	v1.28.3	5374347291230	126MB
registry.k8s.io/coredns/coredns	v1.10.1	ead0a4a53df89	53.6MB

Slika 15. Izlaz komande *minikube image ls* nakon build-ovanja slika

```
C:\Users\vukad\Storage\Fakultet MAS\1. Semestar\5 Napredni operativni sistemi\NOS3\kubernetes>kubectl apply -f .
deployment.apps/ekuiper-manager-deployment created
service/ekuiper-manager created
persistentvolumeclaim/ekuiper-data-pvc created
persistentvolumeclaim/ekuiper-log-pvc created
deployment.apps/ekuiper-deployment created
service/ekuiper created
deployment.apps/emqx-deployment created
service/emqx created
persistentvolumeclaim/gather-mongodb-data-pvc created
deployment.apps/gather-mongodb-deployment created
service/gather-mongodb created
deployment.apps/gather-deployment created
service/gather created
persistentvolumeclaim/ticket-mongodb-data-pvc created
deployment.apps/ticket-mongodb-deployment created
service/ticket-mongodb created
deployment.apps/ticket-deployment created
service/ticket created
```

Slika 16. Izvršavanje komande za pokretanje podova, servisa i kreiranje *PersistentVolumeClaim*-ova

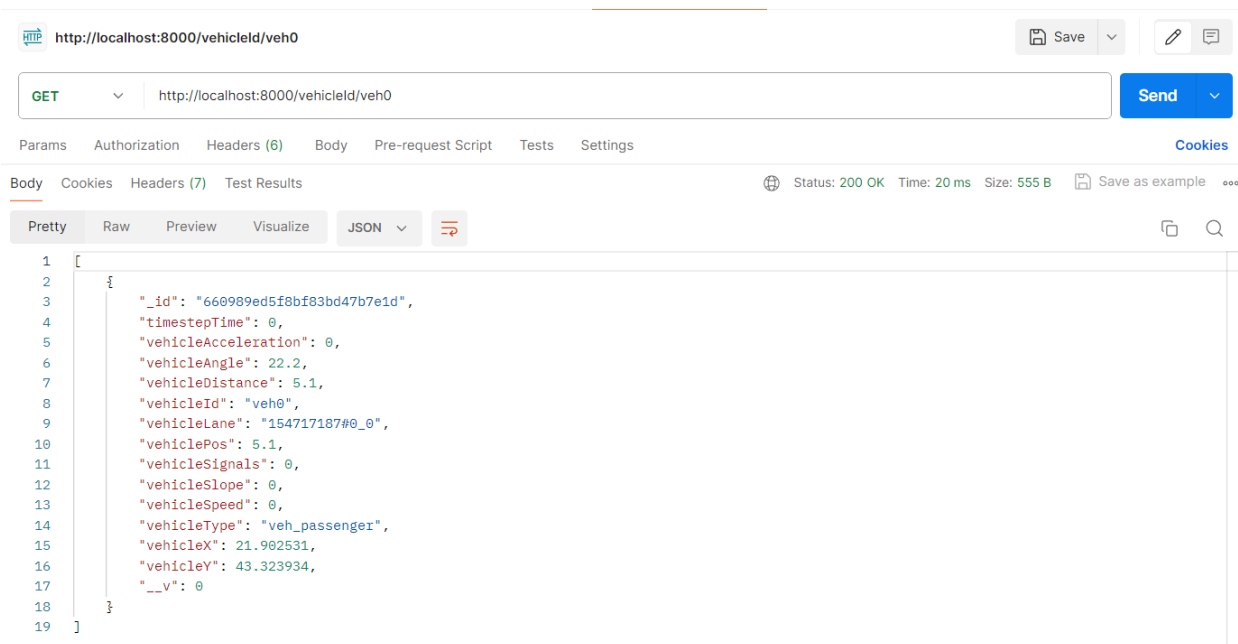
Da bismo omogućili pristup servisima unutar klastera iz spoljne sredine, neophodno je da u nekom posebnom terminalu izvršimo komandu *minikube tunnel*, kako bi *minikube* omogućio saobraćaj između klastera i spoljnog sveta. Komandom *kubectl get pods* se možemo uveriti da su naši podovi uspešno pokrenuti. Ukoliko želimo da manuelno skaliramo neki pod, to ćemo uraditi komandom *kubectl scale --replicas=<broj replika> <šta želimo da skaliramo>*. Na slici 17 je prikazano skaliranje *gather* servisa u sa jedne na dve instance.

```
C:\Users\vukad\Storage\Fakultet MAS\1. Semestar\5 Napredni operativni sistemi\NOS3\kubernetes>kubectl scale --replicas=2 deployment.apps/gather-deployment
deployment.apps/gather-deployment scaled

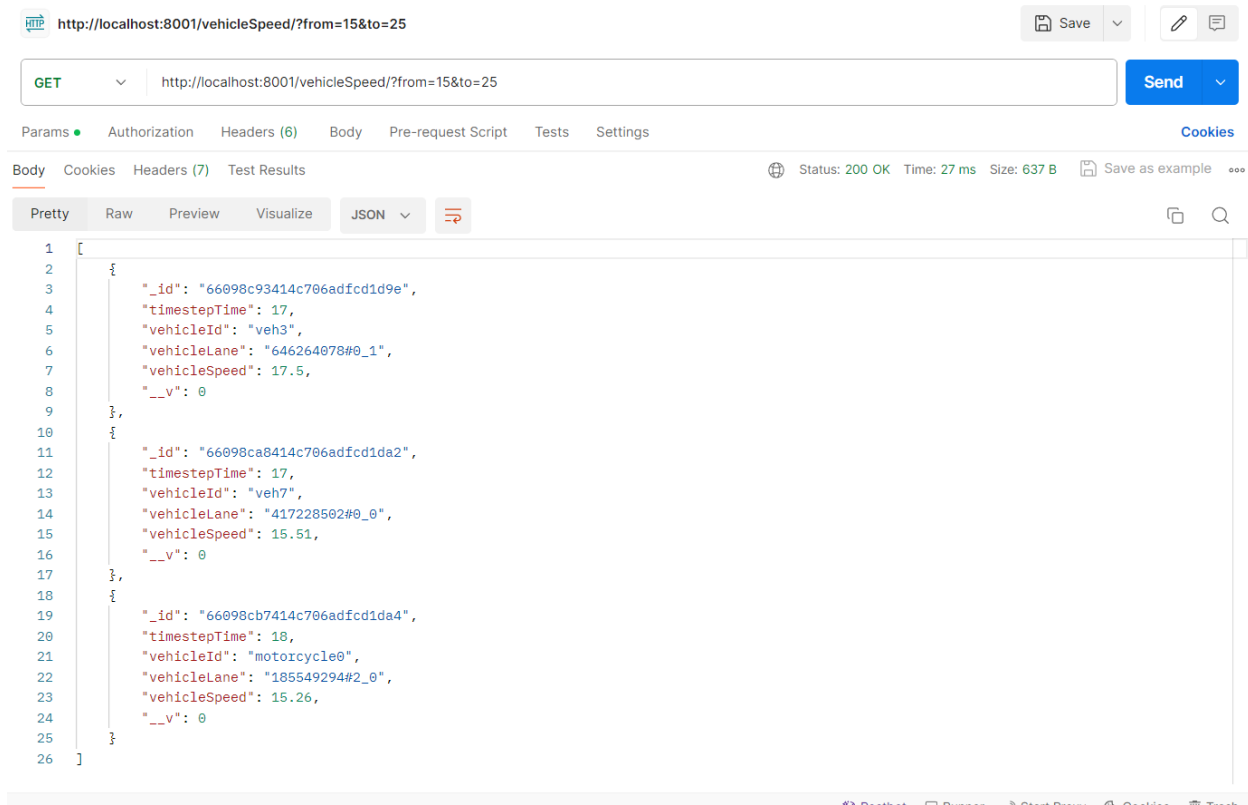
C:\Users\vukad\Storage\Fakultet MAS\1. Semestar\5 Napredni operativni sistemi\NOS3\kubernetes>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
ekuiper-deployment-b748f4c46-529q2  1/1     Running   0           9m5s
ekuiper-manager-deployment-5f5db5bd79-rtvvh  1/1     Running   0           9m5s
emqx-deployment-69cb88bf69-mmzb6        1/1     Running   0           9m4s
gather-deployment-8b9fddb5d-ddvp2        1/1     Running   0           2s
gather-deployment-8b9fddb5d-nd4sx        1/1     Running   2 (8m1s ago)  9m4s
gather-mongodb-deployment-6bc775b478-sm199  1/1     Running   0           9m4s
ticket-deployment-77df5dc466-wxq96       1/1     Running   0           9m4s
ticket-mongodb-deployment-78487956d8-l4k99  1/1     Running   0           9m4s
```

Slika 17. Skaliranje gather servisa sa jedne na 2 instance

Sada ćemo pokrenuti simulator vozila kako bismo se utvrdili da sistem radi, izvršavanjem HTTP GET upita nad *gather* i *ticket* servisima(prikazano na slikama 18 i 19).



Slika 18. Prikaz rada *gather* servisa



Slika 19. Prikaz rada *ticket* servisa

U svrhu automatskog skaliranja podova u našem sistemu, koristi se posebna komponenta *HorizontalPodAutoscaler*. Ova radi po sledećoj jednačini: $d = \text{ceil}[a * (c / t)]$, gde je d broj replika koji treba da postoji u našem sistemu, a predstavlja trenutni broj replika, c trenutnu metriku poda i t predstavlja ciljanu metriku (metrika može predstavljati opterećenje procesora prikazano u procentima ili zauzeće memorije). U našem sistemu smo kreirali komponente za autoskaliranje za *MQTT* broker, *ekuiper* servis, kao i *gather* i *ticket* servise. Ove komponente se nalaze u posebnom folderu *kubernetes-autoscalers*. Komandom *kubectl apply -f* . ćemo pokrenuti sve autoskalere unutar našeg klastera. Broj podova u klasteru pre pokretnja autoskalera možemo videti na slici 20.

```

C:\Users\vukad\Storage\Fakultet MAS\1. Semestar\5 Napredni operativni sistemi\NOS3\kubernetes-autoscalers>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
ekuiper-deployment-b748f4c46-529q2  1/1     Running   0           42m
ekuiper-manager-deployment-5f5db5bd79-rtdvh  1/1     Running   0           42m
emqx-deployment-69cb88bf68-wmzb6       1/1     Running   0           42m
gather-deployment-8b9fdbd5d-nd4sx       1/1     Running   2 (41m ago)  42m
gather-mongodb-deployment-6bc775b478-sml99  1/1     Running   0           42m
ticket-deployment-77df5dc466-fctwq      1/1     Running   0           18m
ticket-mongodb-deployment-78487956d8-l4k99  1/1     Running   0           42m

```

Slika 20. Pre pokretanja autoskalera

```
C:\Users\vukad\Storage\Fakultet MAS\1. Semestar\5 Napredni operativni sistemi\NOS3\kubernetes-autoscalers>kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
ekuiper-deployment-77df847576-qxq88	1/1	Running	0	8m17s
ekuiper-manager-deployment-5f5db5bd79-rtdvh	1/1	Running	0	87m
emqx-deployment-558d7c88f8-4nbzs	1/1	Running	0	51s
emqx-deployment-558d7c88f8-5nxgr	1/1	Running	0	8m17s
emqx-deployment-558d7c88f8-dlmbz	1/1	Running	0	51s
gather-deployment-6c589f7f84-2qrxn	1/1	Running	0	8m16s
gather-mongodb-deployment-6bc775b478-sml99	1/1	Running	0	87m
ticket-deployment-6d54c7749b-4wj6t	1/1	Running	0	11m
ticket-mongodb-deployment-78487956d8-l4k99	1/1	Running	0	87m

Slika 21. Posle pokretanja autoskalera

```
C:\Users\vukad\Storage\Fakultet MAS\1. Semestar\5 Napredni operativni sistemi\NOS3\kubernetes-autoscalers>kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
ekuiper-deployment-77df847576-qxq88	1/1	Running	0	16m
ekuiper-manager-deployment-5f5db5bd79-rtdvh	1/1	Running	0	95m
emqx-deployment-558d7c88f8-2vlld	1/1	Running	0	34s
emqx-deployment-558d7c88f8-5nxgr	1/1	Running	0	16m
emqx-deployment-558d7c88f8-cfkvg	1/1	Running	1 (22s ago)	2m19s
emqx-deployment-558d7c88f8-cpcng	1/1	Running	0	2m19s
emqx-deployment-558d7c88f8-xbjw9	1/1	Running	0	34s
gather-deployment-6c589f7f84-2qrxn	1/1	Running	0	16m
gather-deployment-6c589f7f84-dxxkq	1/1	Running	0	4s
gather-mongodb-deployment-6bc775b478-sml99	1/1	Running	0	95m
ticket-deployment-6d54c7749b-4wj6t	1/1	Running	0	19m
ticket-mongodb-deployment-78487956d8-l4k99	1/1	Running	0	95m

Slika 22. Nakon izmene tajmauta u simulatoru na 0.2 sekunde

Na slici 21 možemo videti da se nakon pokretanja autoskalera, odmah skalirao MQTT EMQX broker na 3 replike. Kako bismo izazvali autoskaliranje i *gather* servisa, smanjićemo *timeout* u simulatoru na 0.2 sekunde. Rezultat skaliranja nakon promene je prikazan na slici 22.