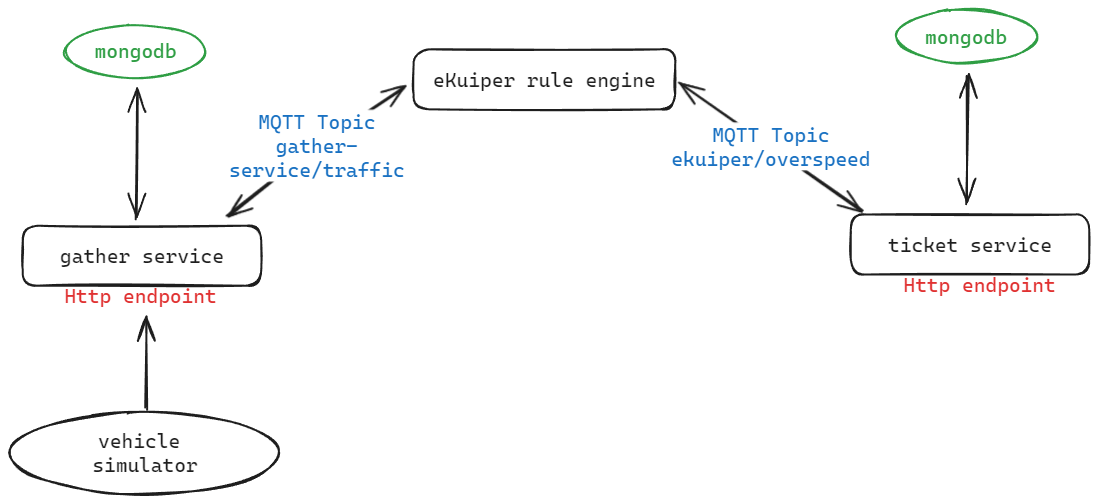
Napredni operativni sistemi  
III Projekat

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 pretage po ovim parametrima, omogućava i pretragu po *vehicleSpeed* parametru.



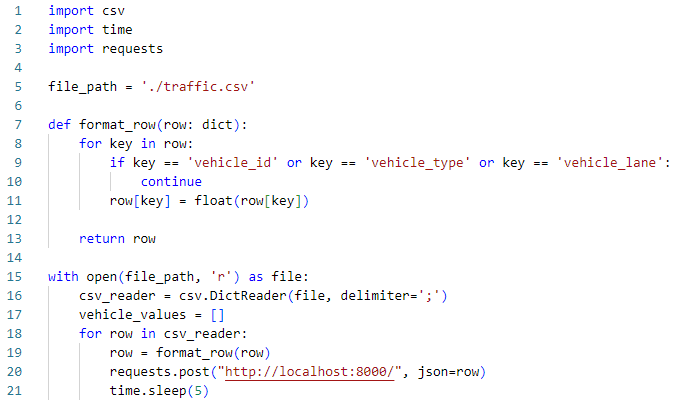
Slika 1. Arhitektura sistema

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

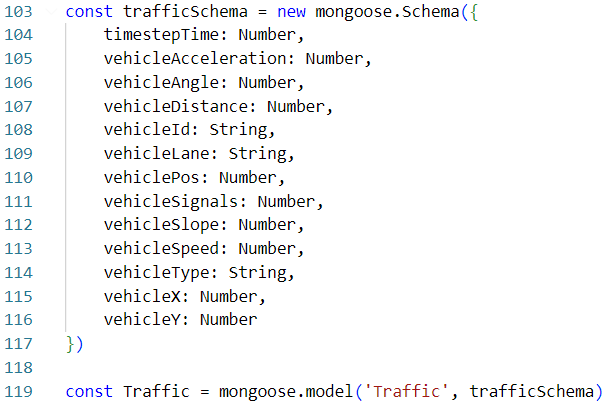
Simulator vozila je predstavljen *python* skriptom **vehicles.py** koja čita red po red podataka iz **traffic.csv** fajla i svaki red šalje *gather* servisu pomoću HTTP POST metode. Prilikom slanja je stavljen određeni timeout, kako bi mogao da se prati rad sistema „korak po korak“. Takođe, kasnijem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



Slika 2. Kod vehicles.py skripte

1. *Gather* servis

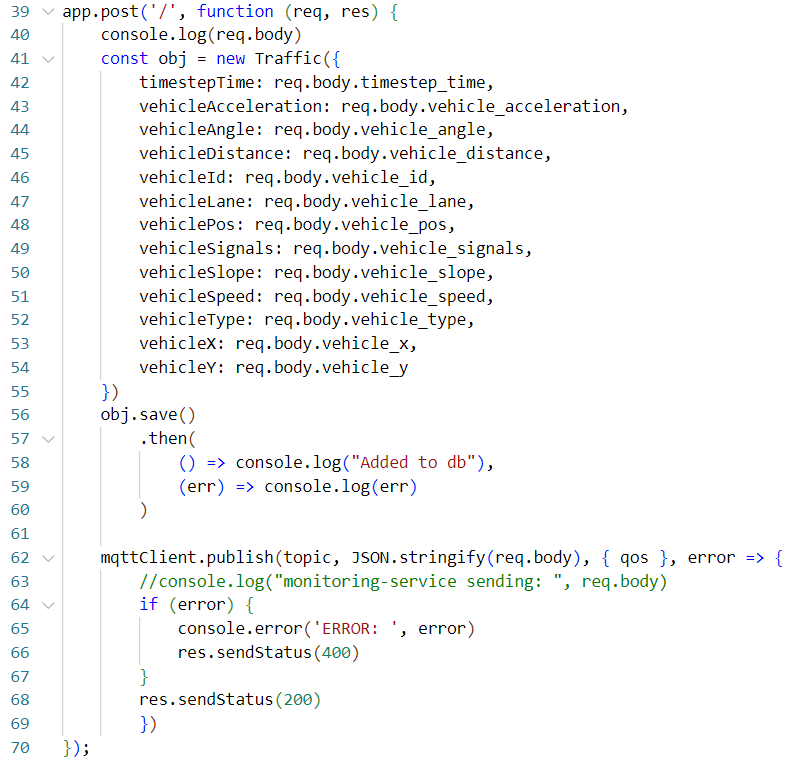
Ovaj servis je implementiran u *node.js*-u, odnosno u *express framework*-u. Izvorni kod aplikacije se nalazi u *index.js* fajlu. On nudi HTTP POST *endpoint* preko koga će mu simulator slati podatke, koje će on smeštati u bazu podataka i slati na *MQTT Topic*.  
Baza podataka koju servis koristi je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



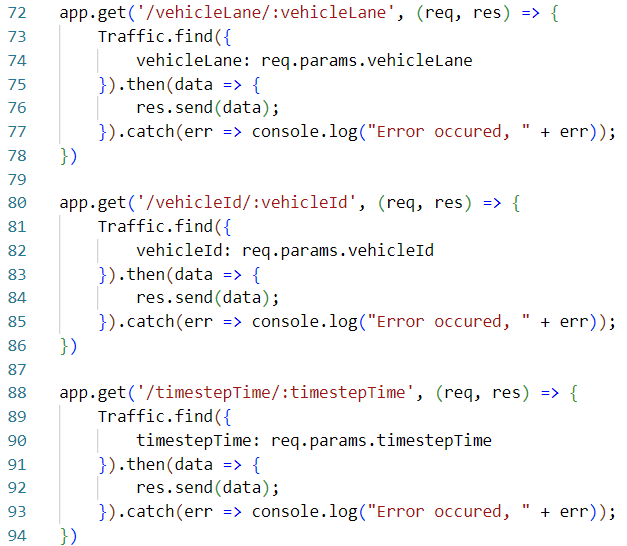
Slika 3. *Traffic* model za rad sa *MongoDB*-jem

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, konrektno 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.



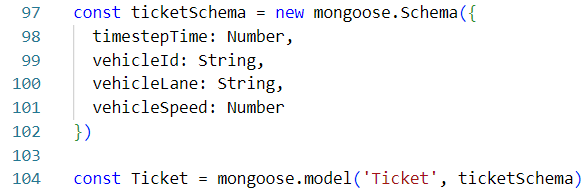
Slika 4. HTTP POST metoda



Slika 5. HTTP GET metode koje nudi servis

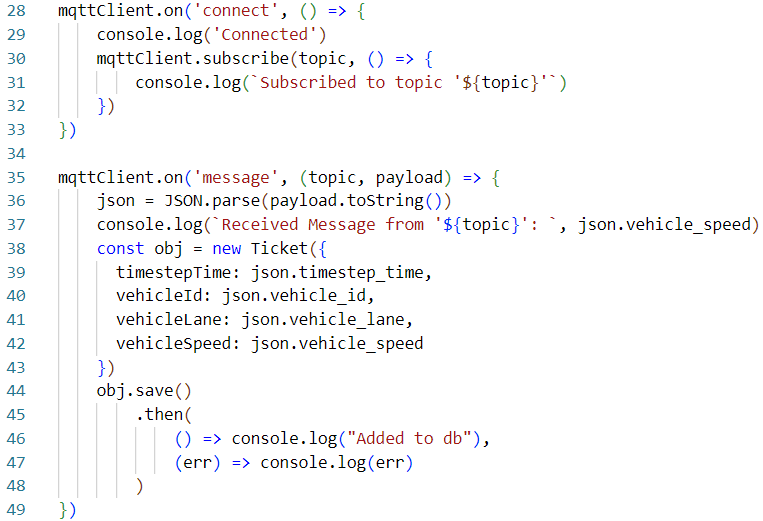
1. Ticket servis

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 poslalao 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.



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.



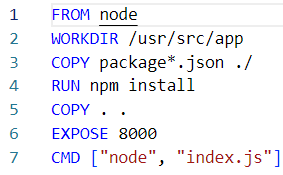
Slika 7. Postavljanje *MQTT* klijenta u *ticket* servisu

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 serivsa, 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.

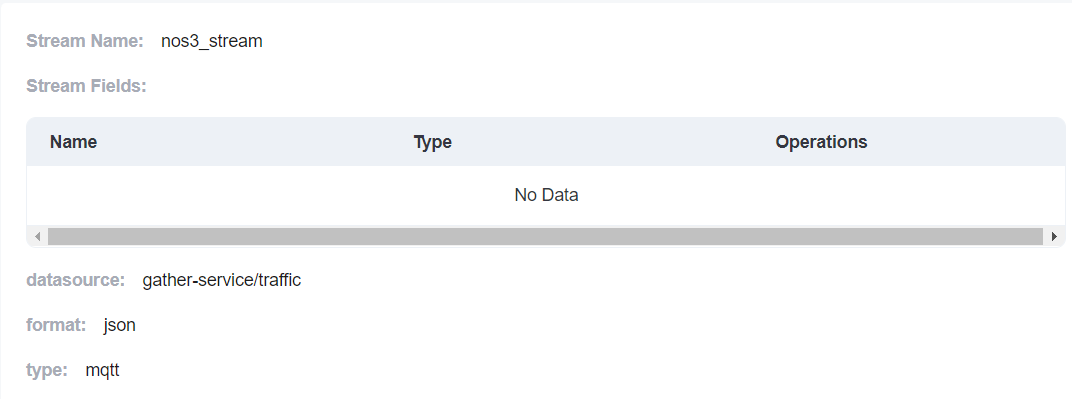


Slika 8. Dockerfile *gather* servisa

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.

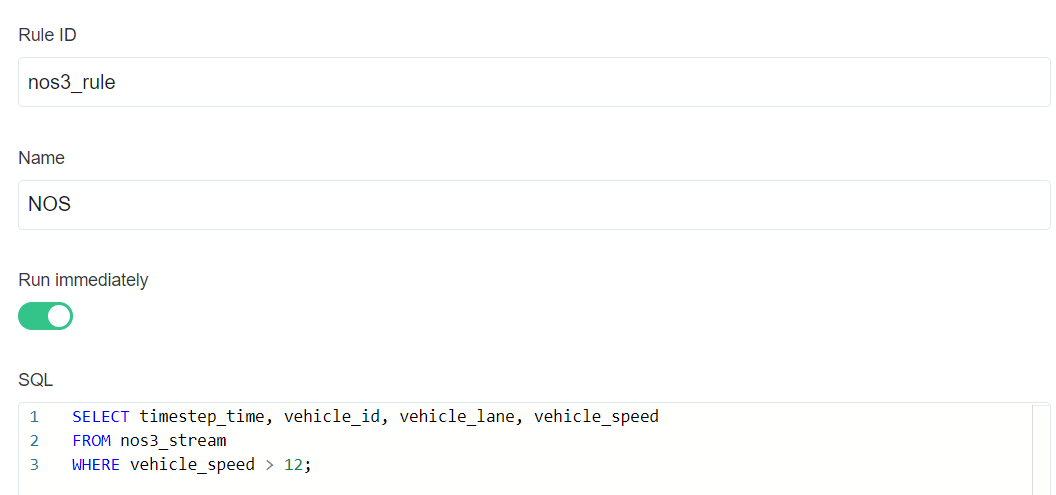
*Ekuiper manager* servis se podiže isključivo kako bi se konfigurisao *ekuiper*, tj. kako bi se navelo pravilo po kome će se detektovati prekoračenje brzine kretanja vozila.

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 detekotvanje 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.

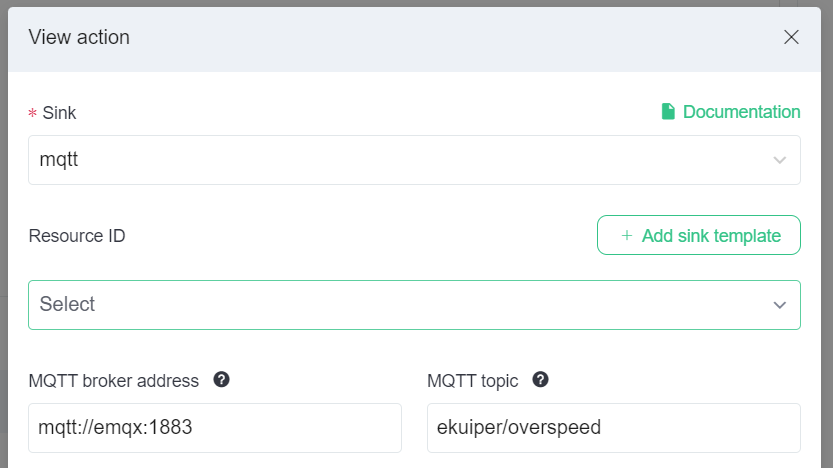


Slika 9. Ekuiper stream

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).



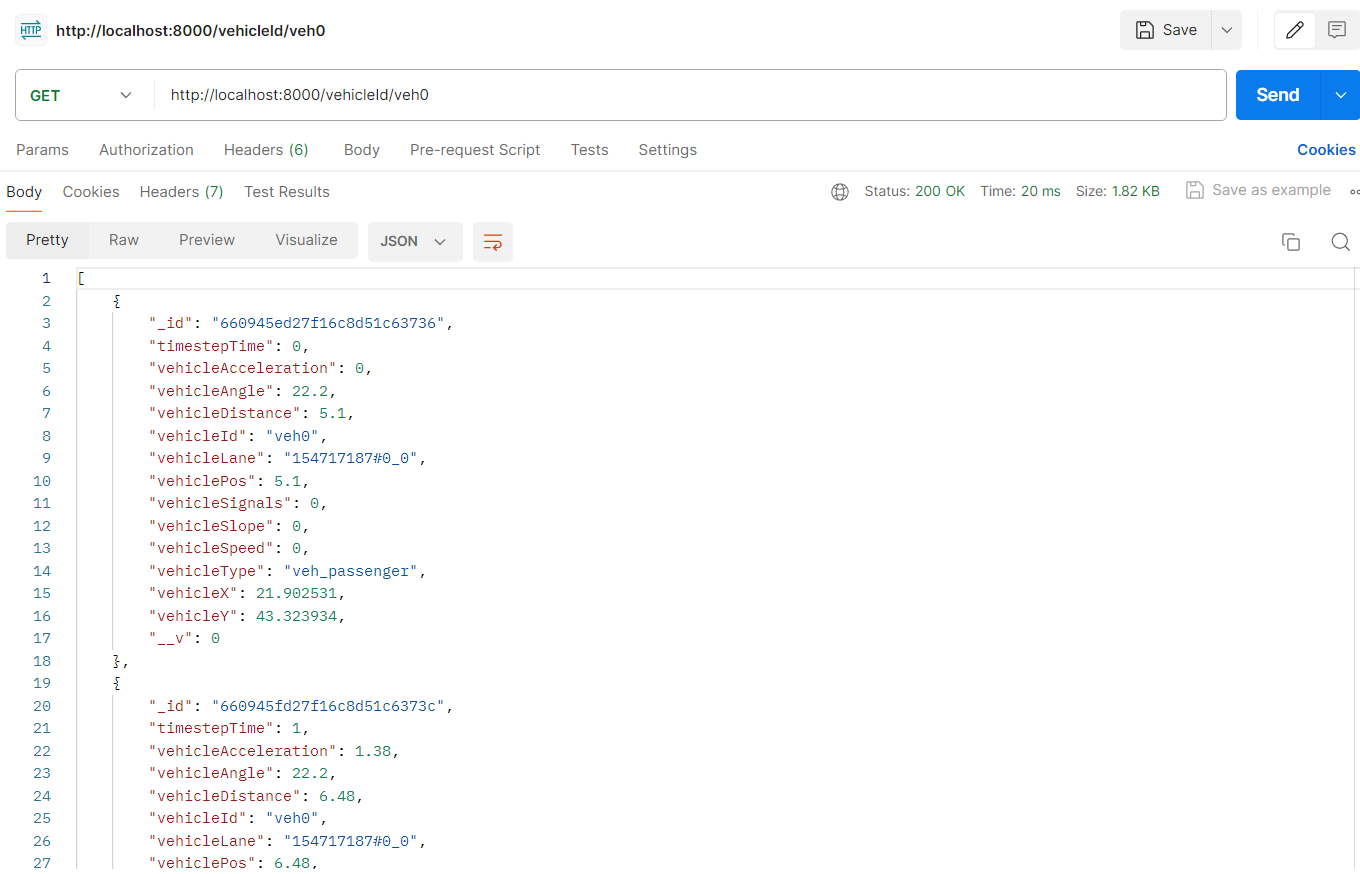
Slika 10. Jednostavno pravilo za detektovanje prekoračenja brzine

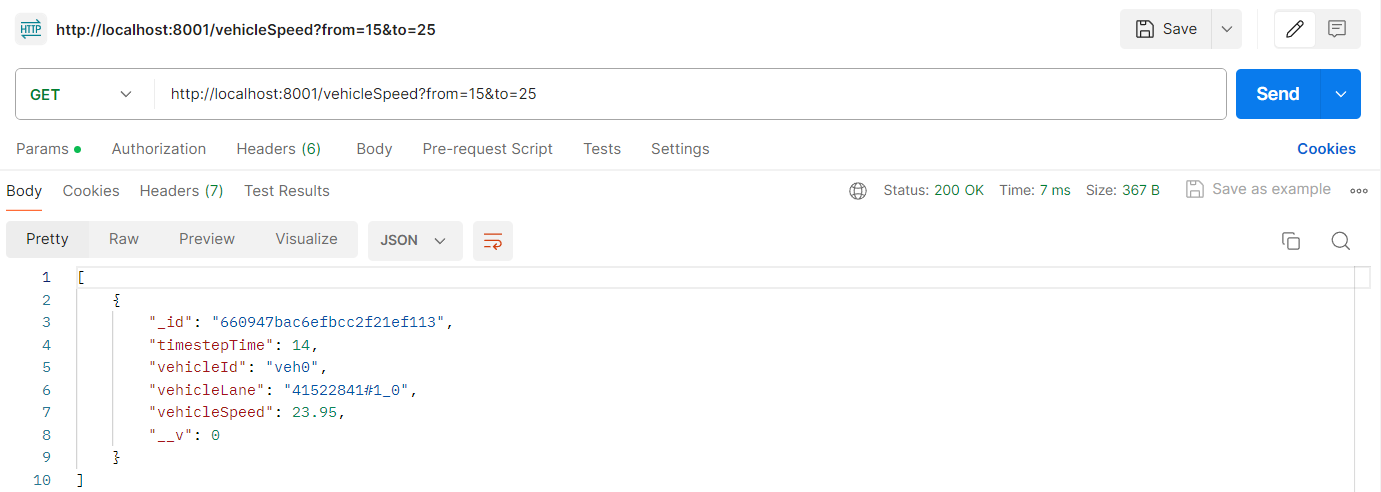


Slika 11. Akcija prilikom ispunjenja uslova iz pravila

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.

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



Slika 13. Izvršenje HTTP GET upita nad ticket servisom

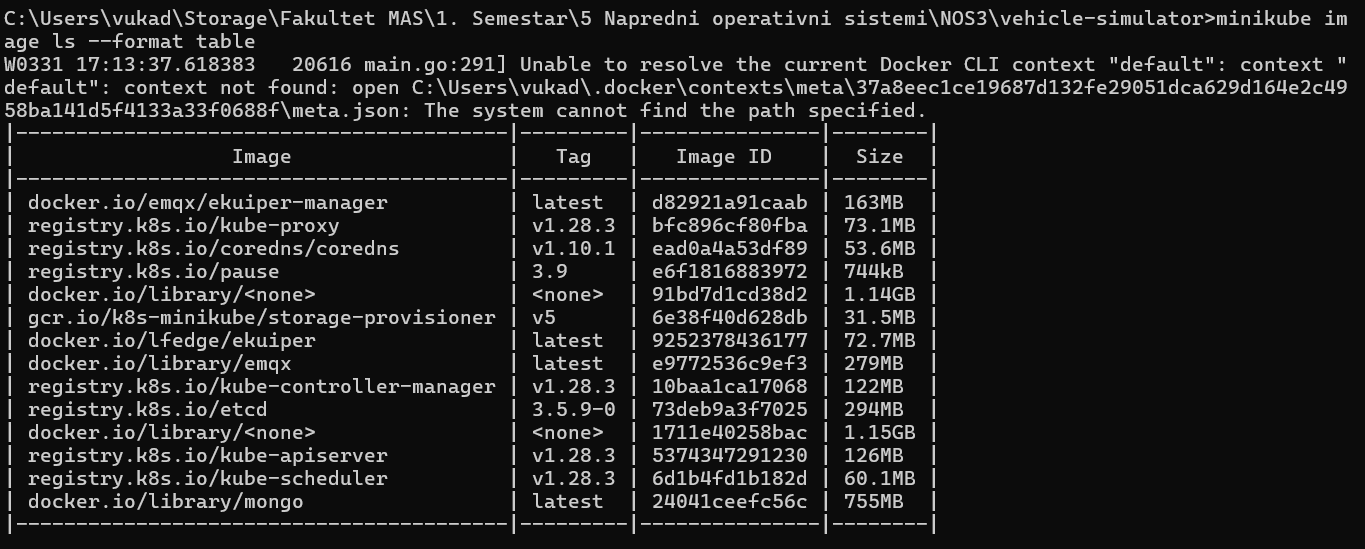
Pokretanje i upravljanje sistema *Kubernetes*-om

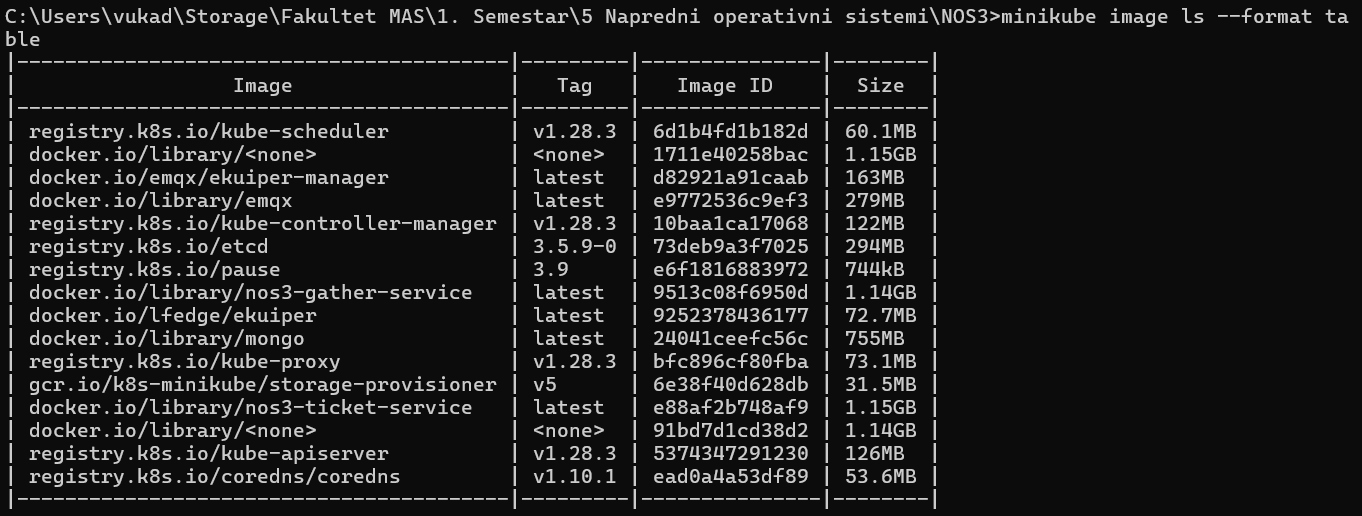
Kubernetes je platforma otvorenog koda namenjena automatizovanom upravljanju kontejnerizovanih aplikacija i skaliranju istih. Pod je najmanja komponenta unutar Kubernetes klastera, i predstavlja jedinicu izvršavanja kontejnerizovane/ih aplikacija, zato što se unutar jednog poda može izvršavati više kontejnera. 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 spoljašnoj 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 ili StatefulSet-a, koji predstavljaju apstrakciju poda. U Deployment-u navodimo iz koje slike želimo da pokrenemo kontejnere unutar poda, kao i koliko replika podova želimo. Razlika izmedju Deploymenta i StatefulSet-a je ta što StatefulSet ima svoje interno stanje, a Deployment ne. 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 funckioniše.

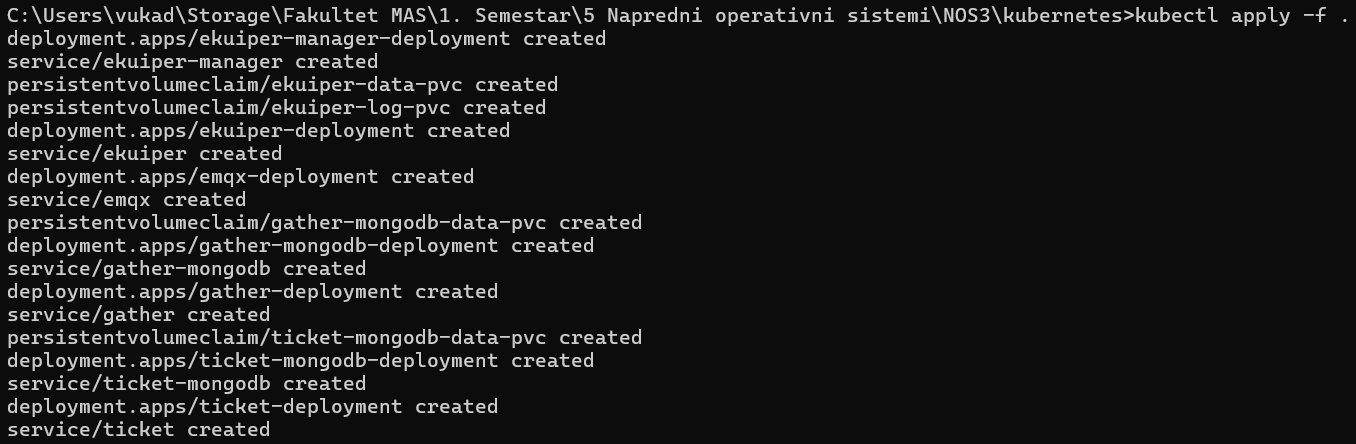
U *kubernetes* folderu projekta se nalaze svi .yaml fajlovi (ekstenzija koja se koristi kod pisanja *Deployment*, *Service* i *PersistenVolumeClaim* fajlova), i u svakom fajlu se nalazi definisan *PersistenVolumeClaim* ukoliko je to zahtev kontejnera, a nakon toga idu potpisi *Deploymenta* i servisa. Servisi koji su kreirani kao eksterni servisi su *ticket* servis, gather servis i *ekuiper* *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\Dockerfile .\Microservices\gather-service* iz korenskog direktorijuma projekta se bilduje 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.

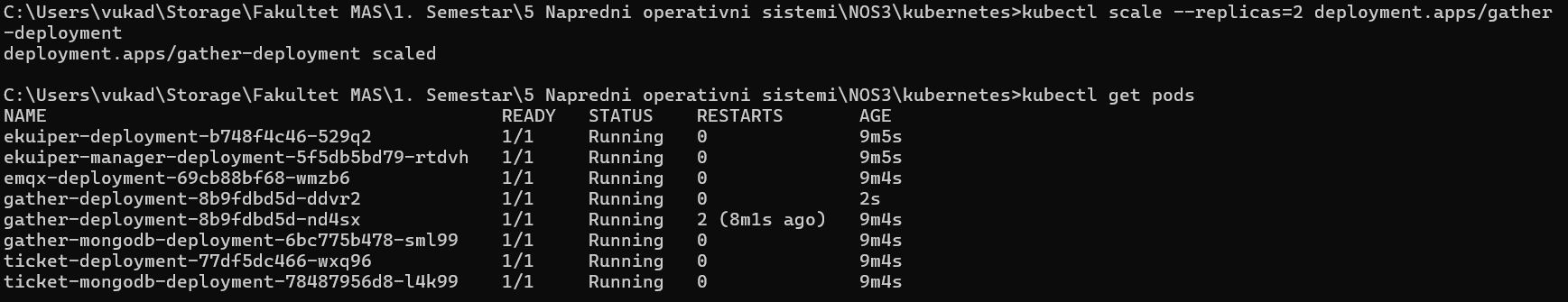
Da bismo pokrenuli podove u našem klasteru, neophodno je da u *kubernetes* folderu izvršimo komandu prikazanu na slici 16.

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

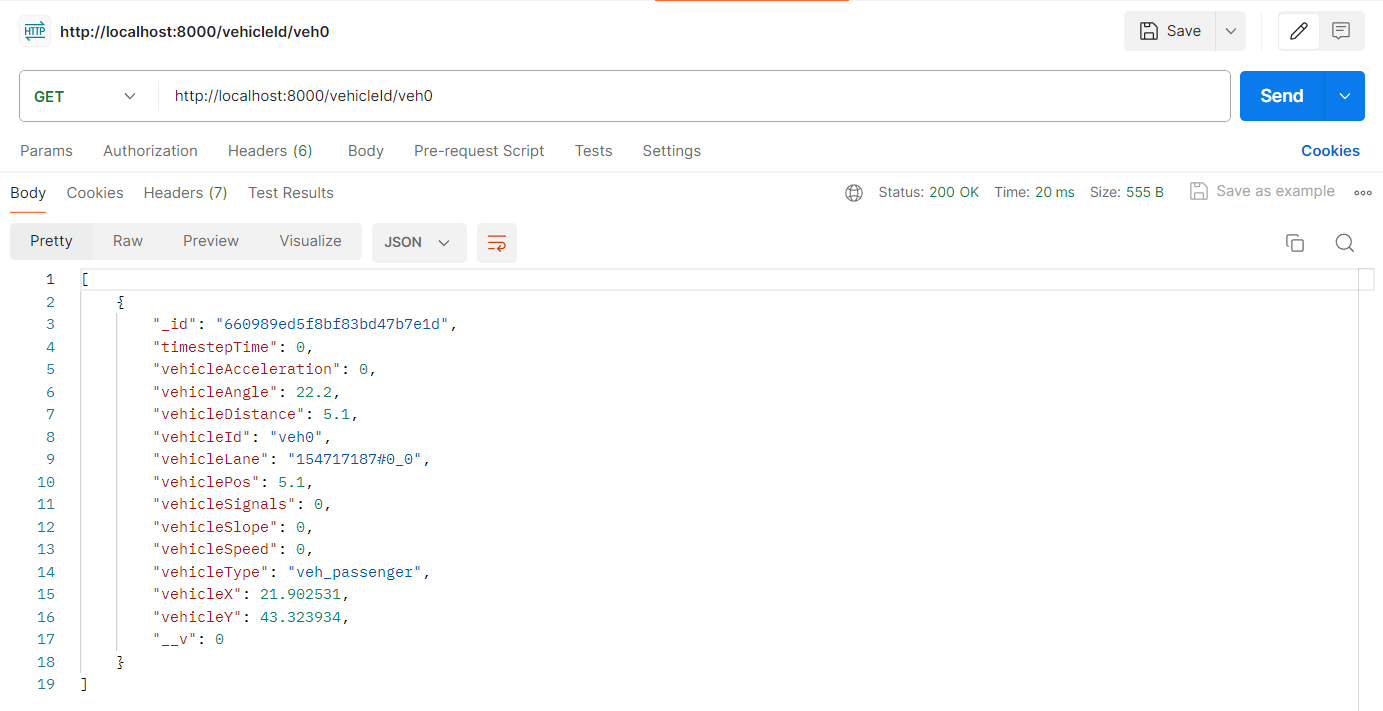
Slika 15. Izlaz komande *minikube image ls* nakon *build*-ovanja slika

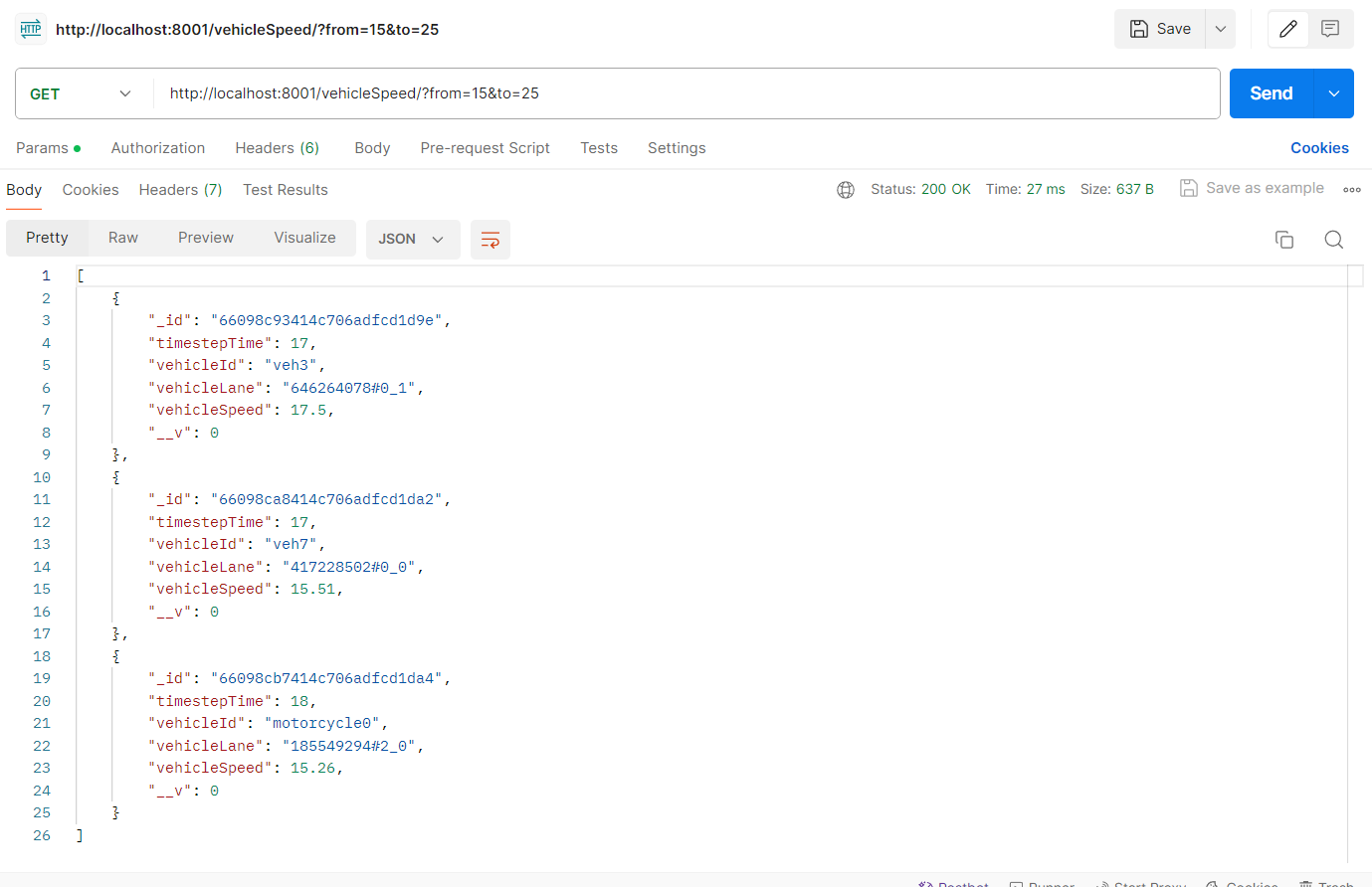
Slika 16. Izvršavanje komande za pokretanje podova, servisa i kreiranje *PersistenVolumeClaim*-ova

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.

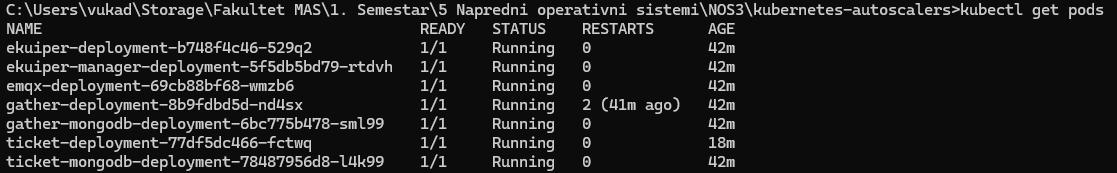
Slika 17. Skaliranje gather servisa sa jedne na 2 instance

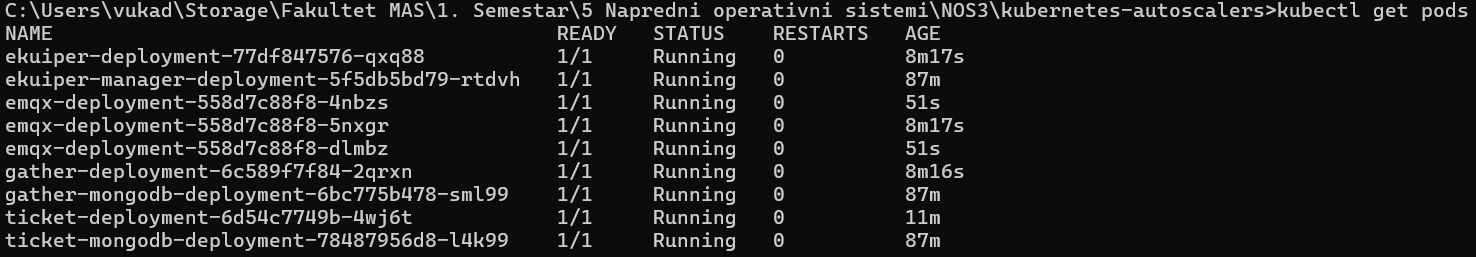
Sada ćemo pokrenuti simulator vozila kako bismo se utvrdili da sistem radi, izvršavanjem HTTP GET upita nad *gather* i *ticket* servisima(prikazano na slikama 18 i 19).

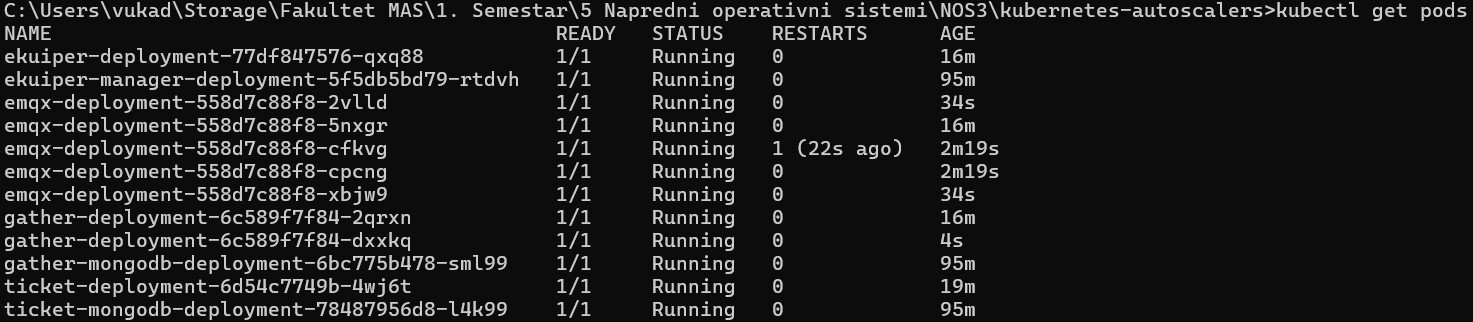
Slika 18. Prikaz rada *gather* servisa

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=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* predstavla 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.

Slika 20. Pre pokretanja autoskalera

Slika 21. Posle pokretanja autoskalera

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.