Implementacija backenda socijalne mreže u mikroservisnoj arhitekturi nad platformom kubernetes

Degmečić, Filip

Sveučilište Jurja Dobrile u Puli
Fakultet informatike
Diplomski studij informatike

Filip Degmečić

Implementacija backenda socijalne mreže u mikroservisnoj arhitekturi nad platformom kubernetes

Diplomski rad

Pula, 2021.

Sveučilište Jurja Dobrile u Puli
Fakultet informatike
Diplomski studij informatike

Implementacija backenda socijalne mreže u mikroservisnoj arhitekturi nad platformom kubernetes

Diplomski rad

JMBAG: 0303054201, redovan student
Studijski smjer: Informatika
Predmet: Izrada informatičkih projekata
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor: doc. dr. sc. Nikola Tanković

Pula, 2021.

IZJAVA o korištenju autorskog djela

Ja, _____ dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom

_____

_____

koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 13. 09. 2021.

Potpis

_____

IZJAVA o korištenju autorskog djela

Ja, _____ dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom

_____

_____

koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 13. 09. 2021.

Potpis

_____

Sadržaj

1 Uvod

Ovaj rad bavit će se aktualnim pitanjem odabira arhitekture računalnog sustava. Cilj rada je istražiti prednosti i mane mikroservisne arhitekture te implementirati sustav uzevši u obzir nedostatke mikroservisne arhitekture. U implementaciji korišteni su brojni alati koji olakšavaju razvoj i isporuku sustava u mikroservisnoj arhitekturi. Za usporedbu prednosti i mana korišten je monolitni sustav koji predstavlja suprotnost temeljnim principima mikroservisne arhitekture. Monolitni sustav je usko povezan dok su komponente unutar mikroservisne arhitekture strukturalno odvojene.

Veliki broj organizacija poput Netflixa ili Amazona predvodi prelazak s monolitnih sustava na sustave mikroservisa. Nedavno izvješće [1] pokazuje da devedeset dva posto organizacija doživljava uspjeh koristeći mikroservise.

Iako mikroservisna arhitektura nije namijenjena za svaki sustav, postoji velika korist u poznavanju tehnologija korištenih kao temelj u mikroservisnoj arhitekturi. Te tehnologije moguće je primijeniti i na sustave ostalih arhitektura te im olakšati razvoj ili povećati učinkovitost.

U radu će biti predstavljene temeljne tehnologije i njihovi koncepti koji definiraju mikroservisnu arhitekturu. Implementacija programskog rješenja predstavit će prednosti i mane tih tehnologija.

2 Mikroservisi

Tradicionalno monolitna arhitektura godinama je bila prvi izbor za organizaciju i razvoj sustava. Takav sustav izgrađen je kao jedan veliki objekt, gdje su komponente usko povezane. Primjerice, monolit će sadržati jednu bazu podataka, jednu konfiguraciju te jednu implementaciju sustava. Ovaj tip arhitekture u početku omogućava jednostavnost pri razvoju, testiranju i održavanju, međutim, s vremenom i daljnjim razvojem postaje sve složeniji i nepristupačniji. [2]

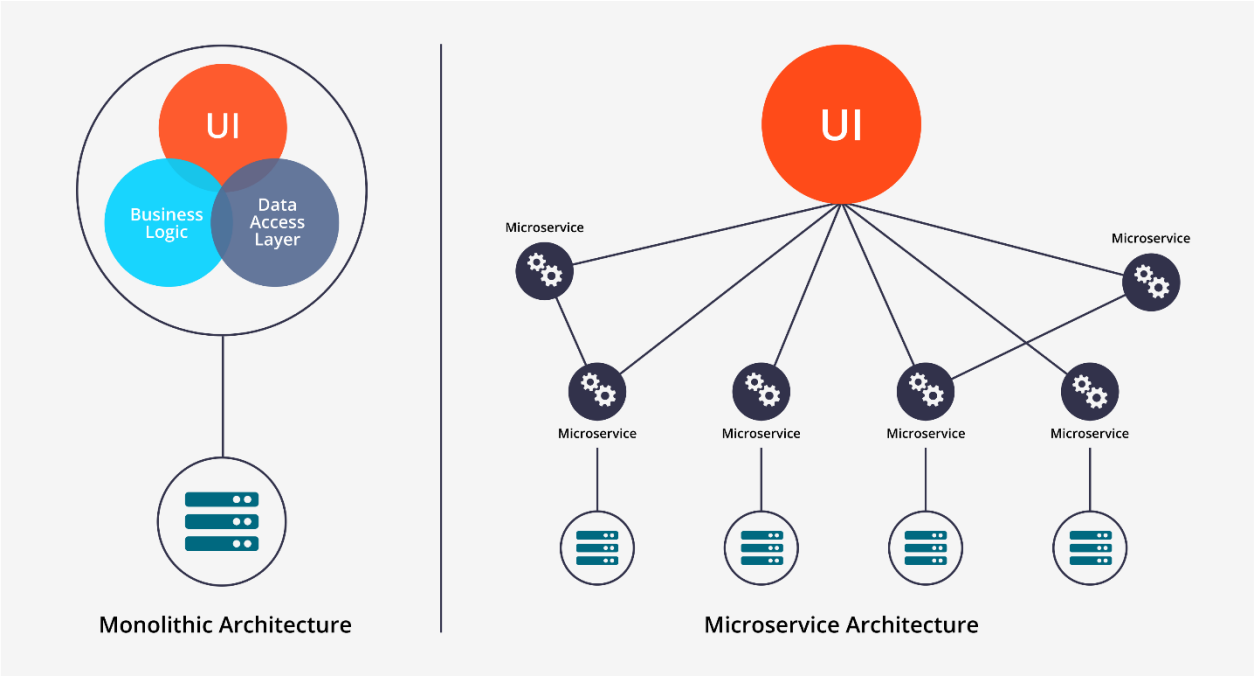
Ovakav tip arhitekture svoje prve nedostatke pokazuje kod skaliranja (engl. scaling) i dodavanja novih značajki (engl. feature). S obzirom na to da su ovakvi sustavi usko povezani, dodavanjem novih značajki postižu složenost koja utječe na druge dijelove sustava. S vremenom ih postaje nemoguće održavati.

Nadalje, skaliranje ovakvog tipa arhitekture predstavlja jedan od većih problema. Zbog spomenutog ograničenja uske povezanosti komponenata, skaliranje ovakvog sustava jedino je moguće kopiranjem cijelog sustava. Iako jednostavan, takav pristup je skup te nije učinkovit. Različiti će dijelovi sustava biti podvrgnuti različitim razinama stresa. Kopiranje cijelog sustava znači da je broj komponenata podvrgnut velikoj količini stresa jednak broju komponenata podvrgnutih manjoj količini stresa, što nije idealno korištenje resursa.

Današnji sustavi ujedno zahtijevaju brzo i efikasno implementiranje promjena što monolitna arhitektura otežava. Za integraciju novih promjena i isporuku tih promjena potrebno je ponovo pokretanje cijele aplikacije što može biti vrlo sporo i može ugroziti kontinuiranost rada aplikacije.

Nasuprot tome, kao nov način organiziranja arhitekture sustava javlja se mikroservisna arhitektura. Temeljni princip ove arhitekture je odvajanje sustava u više nezavisnih komponenti koje nisu usko povezane. Komponente unutar mikroservisa građene su na principu jedne odgovornosti (engl. single responsibility principle) [3]

gdje je svaka komponenta odgovorna za jedan dio funkcionalnosti koje sustav pruža, a međusobno komuniciraju sustavom poruka. Samim time svaka komponenta je zasebna cjelina vidljivo na slici 1 koja može obavljati rad samostalno, čak i u slučaju kada druge komponente u sustavu prestanu raditi.



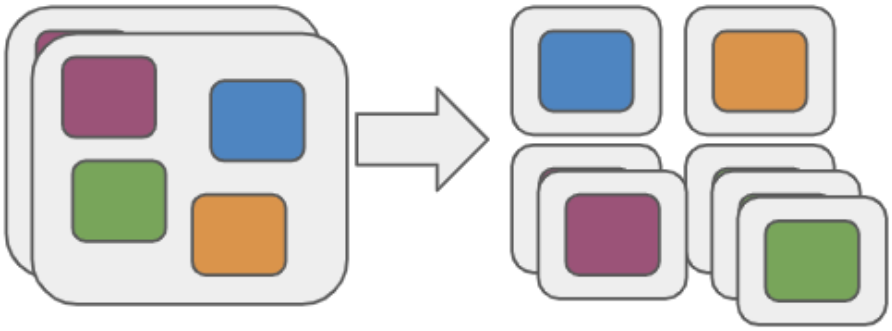
Slika 1: Monolitna i mikroservisna arhitektura

Jedna od glavnih prednosti polazi od spomenutog principa slabe povezanosti koji omogućava otklanjanje jedne točke otkaza. U slučaju kvara određenog dijela aplikacije ostatak aplikacije može nesmetano nastaviti raditi. Razvijanje i održavanje kontinuiranosti rada aplikacije time je uvelike olakšano. Određena komponenta može biti razvijana bez utjecaja na ostale komponente; može biti ugašena, promijenjena, i vraćena u pogon bez da korisnik primijeti. Ove značajke omogućuju brze iteracije pri dodavanju novih značajki ili otklanjanja kvarova.

Još jedna od prednosti ove arhitekture je mogućnost razvoja zasebnih komponenti koristeći različite tehnologije koje najbolje odgovaraju zahtjevu komponente. Primjerice, jedna komponenta može biti razvijena koristeći Javascript s bazom podataka MongoDB dok druga komponenta može koristiti Javu s bazom podataka MySQL. [4]

Ovaj princip osigurava laku promjenu na novu vrstu tehnologiju te omogućava da sustav prati inovacije u tehnologijama za razvoj. Takva prilagodba nije moguća u sustavima s monolitnom arhitekturom jer bi cijeli sustav zahtijevao promjenu na novu tehnologiju.

Mikroservisi također olakšavaju skaliranje aplikacije. Zahvaljujući podjeli na različite servise, jednostavno je skalirati određeni servis koji je pod opterećenjem. Takav način skaliranja drastično je isplativiji naspram skaliranja monolitnih aplikacija gdje je potrebno kopiranje cijelog sustava prikazano na slici 2.



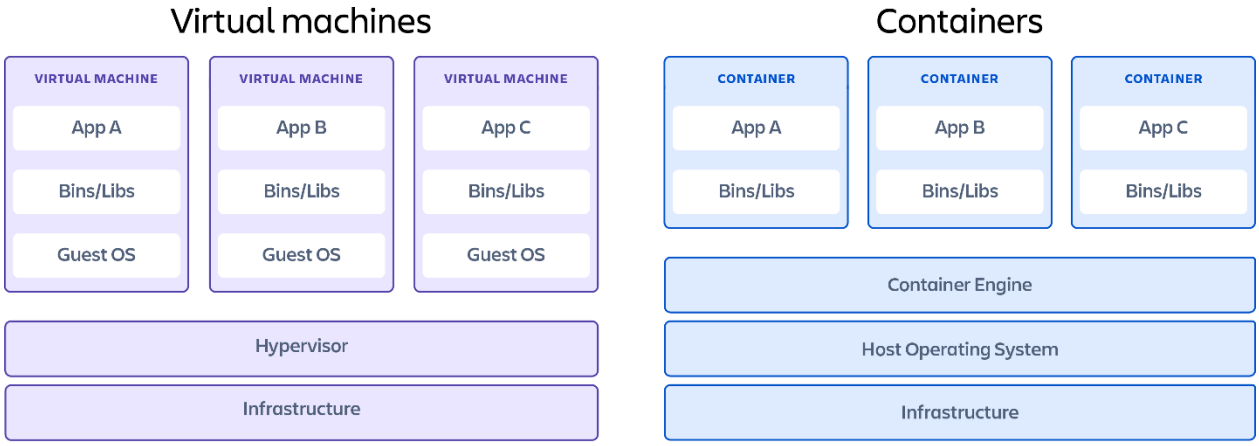
Slika 2: Skaliranje monolita i skaliranje mikroservisa

Iako posjeduju puno prednosti naspram monolitne arhitekture, mikroservisi dolaze s određenim preprekama i manama. Odvojenost servisa otežava i usporava komunikaciju između njih te zahtjeva orkestraciju zasebnih servisa. Potrebno je izgraditi pouzdan sustav komunikacije između mikroservisa te se pobrinuti da ne dođe do problema s konzistentnosti podataka.

3 Docker

Docker je alat često korišten u mikroservisnoj arhitekturi jer olakšava razvoj, održavanje i isporučivanje sustava. Svaka zasebna jedinica mikroservisne arhitekture zahtjeva određenu konfiguraciju, što može dovesti do prevelike složenosti pri razvoju i pokretanju sustava.

Docker omogućava isporuku sustava unutar kontejnera (engl. container), koji postaju zamjena virtualnim strojevima. Kontejneri i virtualne strojevi su slični procesi virtualizacije resursa; virtualni strojevi su potpune implementacije operacijskog sustava dok je kontejner samo virtualizacija softverskih slojeva iznad razine operacijskog sustava što je vidljivo na slici 3. [5] Virtualne mašine unutar sebe sadržavaju razne aplikacije potrebne za pokretanje sustava te se veličina zauzeća memorijskog prostora nalazi u gigabajtima. Kontejneri su naspram toga puno jednostavniji i lakši način virtualizacije koji uključuje samo osnovne resurse potrebne za pokretanje aplikacije.

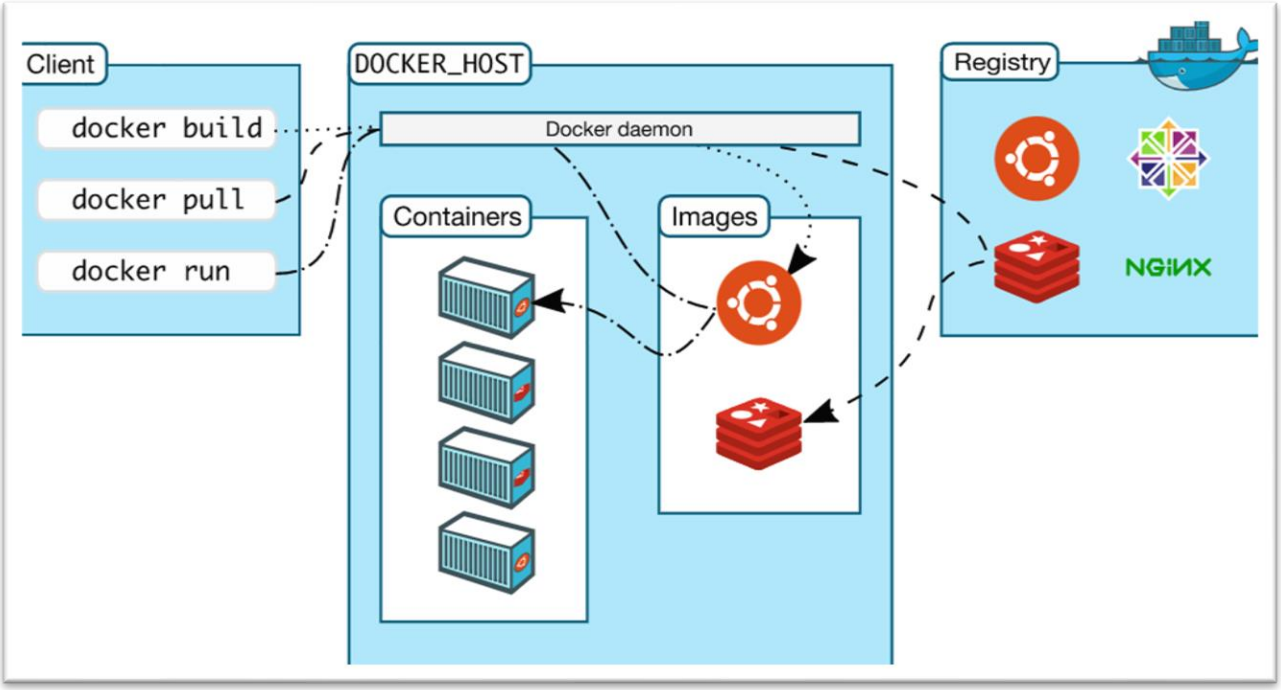


Slika 3: Usporedba virtualnih mašina i kontejnera [5]

Primarni zadatak kontejnera i virtualnih strojeva je upakirati sve potrebne elemente određenog sustava u izolirane cjeline koje mogu samostalno pokrenuti sustav. Taj proces izolacije omogućava da se sustavima pruže točno oni resursi koji su potrebni unutar tog sustava te se time eliminiraju vanjski elementi koji mogu utjecati na sustav.

3.1 Docker arhitektura

Slika 4 prikazuje osnovne komponente Docker sustava a to su: Docker klijent (engl. client), Docker domaćin (engl. host), Docker registar te Docker kontejneri i slike (engl. image), koji spadaju pod Dockerove objekte. [6] Na slici je također prikazan način na koji komponente arhitekture međusobno komuniciraju.



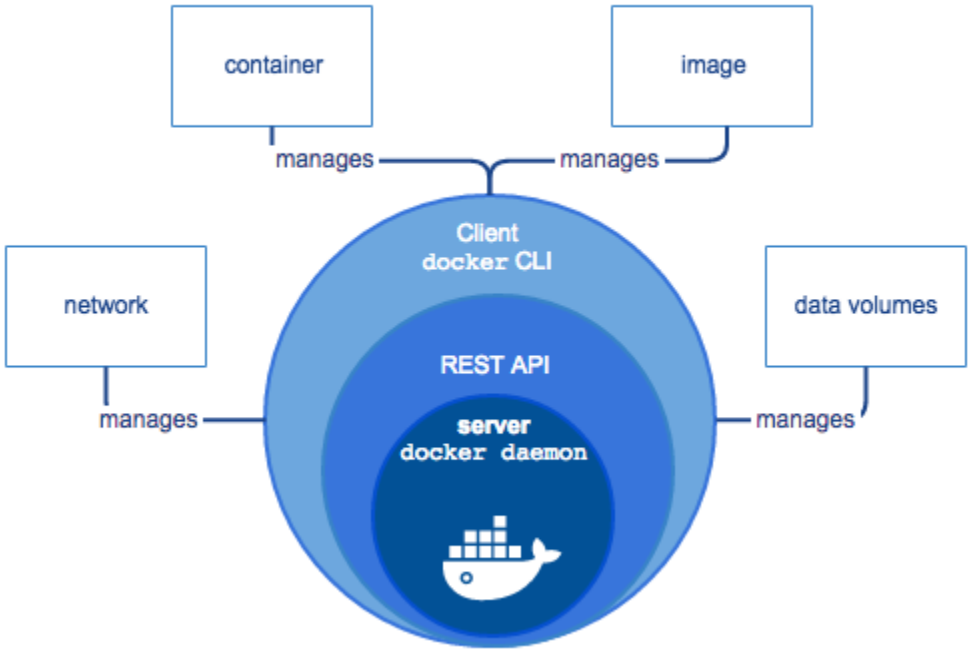
Slika 4: Docker arhitektura [6]

Docker klijent komunicira sa Dockerovim pozadinskim procesom (engl. daemon) koji je zadužen za gradnju i pokretanje Docker slika i kontejnera. Dockerov pozadinski proces ne mora se nalaziti na istom sustavu kao i Docker klijent te jedan klijent može komunicirati s više pozadinskih procesa odjednom.

Dockerov registar unutar sebe sadrži veliki broj Docker slika koje su spremne za korištenje i dohvaćanje putem Dockerovog pozadinskog procesa. Javni registar poznat je kao Docker Hub. On sadrži veliki broj službenih slika raznih organizacija ili slika koje su objavili drugi korisnici.

3.2 Docker pokretač

Docker pokretač (engl. engine) je klijent – server aplikacija koja se sastoji od tri komponente prikazane na slici 5.



Slika 5: Docker engine [7]

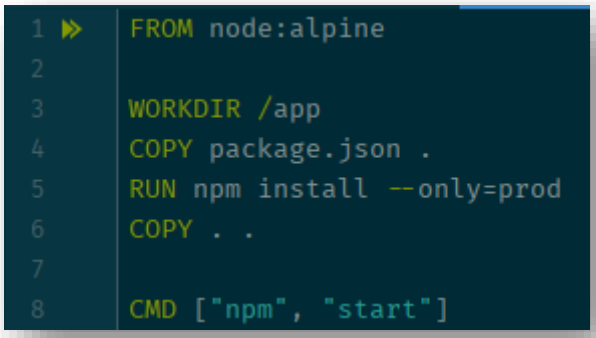
Unutrašnji sloj, *docker daemon*, izvršava se na domaćinskom sustavu kao pozadinski proces. Njegova glavna uloga je stvaranje i upravljanje Dockerovim objektima poput kontejnera, slika i mreža. [8] Sljedeći sloj, aplikacijsko programsko sučelje, služi za omogućavanje komunikacije i izdavanje naredbi Docker pozadinskom procesu od strane klijenta. Posljednja komponenta, Docker klijent, je komandno sučelje koje omogućava korisniku kontroliranje Docker pozadinskog procesa putem skripti ili izravnih naredbi.

3.3 Docker objekti

Tijekom korištenja Dockera, koriste se i stvaraju kontejneri, slike, volumeni i drugi oblici objekata. Docker slika je predložak koji je namijenjen samo za čitanje. Unutar njega su sadržane instrukcije za stvaranje već spomenutog kontejnera. Korisnici mogu koristiti

postojeće slike s javnih registara te ih prema potrebi izmijeniti i proširiti njihovu upotrebljivost. Mogu napraviti i vlastitu sliku koja je prilagođena njihovim potrebama. [6]

Korisnik je za izgradnju vlastite slike u mogućnosti napraviti *dockerfile* datoteku, koja u sebi sadrži naredbe namijenjene za Dockerov pozadinski proces. Jedan od *dockerfile-ova* korišten u implementaciji ovog rada, vidljiv je na slici 6.



Slika 6: Dockerfile za Docker sliku

Osnovna Docker slika navedena je putem naredbe FROM koja dohvaća postojeću sliku s javnog registra Docker hub. To je službena slika Dockerovog tima za održavanje i izgradnju Node.js okruženja. Nakon toga, naredba *WORKDIR* stvara direktorij pod nazivom *app*, te mijenja radni direktoriji iz korijena kontejnera u novonastali direktorij. Ulaskom u željeni direktorij, sljedećom naredbom COPY kopira se datoteka *„package.json“* koja u sebi sadrži sve potrebne zavisnosti nužne za pokretanje aplikacije. Naredba RUN instalira zavisnosti i pakete iz prethodne datoteke, a nakon toga već spomenuta naredba COPY ovaj puta korištena je za kopiranje svih postojećih datoteka iz korijena kontejnera u radni direktoriji.

Kao posljednji korak, potrebno je pokrenuti aplikaciju te se to obavlja koristeći naredbu CMD putem koje se postavlja željeni način za pokretanje kontejnera.

Dockerov pozadinski proces će na temelju svake instrukcije unutar *dockerfile*-a stvoriti jedan sloj slike. Ako je potrebno raditi izmjene na slici nastaloj od *dockerfile*-a, kada se promijeni određena linija naredbe, ponovno će se izgraditi samo taj specificirani sloj.

Ovo je jedan od glavnih razloga koji čini Docker slike bržim, manjim i lakšim naspram ostalih virtualizacijskih tehnologija. [9]

Koristeći Docker sliku, moguće je stvoriti jednu ili više instanci kontejnera. Naredbama, putem komandnog sučelja moguće je stvarati, pokretati, zaustavljati i brisati kontejnere. Kontejneri su poprilično dobro izolirani od domaćinskog sustava i drugih kontejnera te je razina izolacije u rukama korisnika.

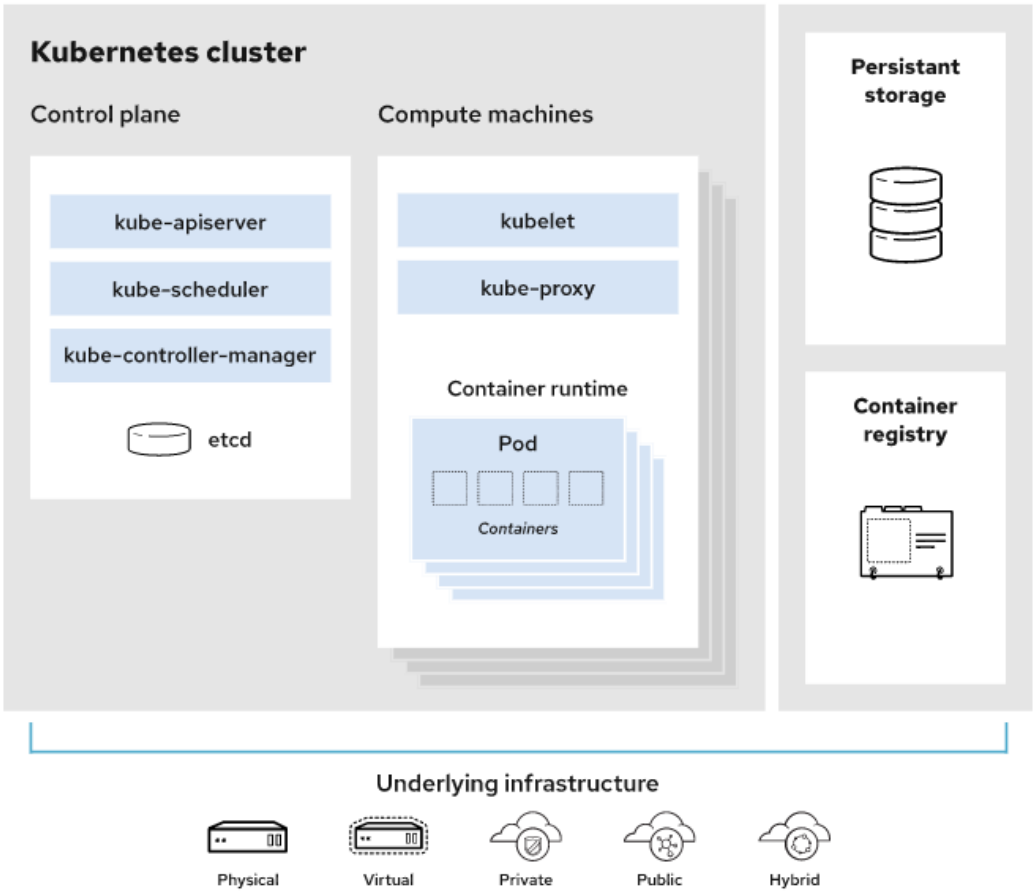
Komunikacija s vanjskim elementima izvodi se spajanjem na virtualnu mrežu ili putem IP adrese. Prilikom zaustavljanja kontejnera, sve promjene ili novonastale datoteke unutar kontejnera se brišu, ako se ne koristi trajno pohranjivanje podataka.

4 Kubernetes

Kubernetes, poznat pod kraticom K8s, besplatni je alat otvorenog koda koji se koristi za automatiziranu orkestraciju, isporuku i razvoj kontejneriziranih sustava. Igra veliku ulogu u rješavanju pojedinih prepreka u mikroservisnoj arhitekturi. Alat razvija Google, a korisnicima između brojnih funkcionalnosti pruža mogućnost jednostavnog skaliranja određenih dijelova sustava. Uz Kubernetes rješenje za orkestraciju kontejnera pružaju i Docker Swarm, AWS Fargate, Azure Container Instances i mnogi drugi.

4.1 Kubernetes arhitektura

Pokrenuti Kubernetes sustav zvan je grozd (engl. cluster). Grozd sadrži dva dijela prikazana na slici 7.



Slika 7: Kubernetes arhitektura [10]

Prvi dio, poznat kao kontrolna ravnina (engl. control plane), sadrži komponente koje kontroliraju grozd, dok drugi dio unutar sebe sadrži čvorove (engl. node). Svaki zasebni čvor unutar sebe sadrži Linux okolinu, koja može biti fizička ili virtualni stroj. U okviru čvorova nalaze se ljuske (engl. pod) koje sadržavaju već spomenute kontejnere.

4.1.1 Kontrolna ravnina

Kontrolna ravnina omogućava korisniku da putem komandnog sučelja izdaje naredbe pomoću kojih se kontroliraju čvorovi unutar grozda. Prikazano na slici 7 komponente unutar kontrolne ravnine su:

- *kube-apiserver* – Ovaj dio kontrolne ravnine zaslužan je izložiti Kubernetes API vanjskome svijetu. Kontrolira vanjske i unutrašnje zahtjeve koje dolaze grozdu.
- kube-scheduler – komponenta koja je zaslužna za praćenje novonastalih ljuski bez dodijeljenog čvora te izabire čvor gdje će se ljuska izvršavati. Uzima u obzir brojne faktore poput kolektivnog ograničenja i opterećenja hardverskih resursa kao što su CPU-a i memorija te globalnog zdravlja grozda.
- kube-controller-manager – odgovoran je za pokretanje kontrolnih procesa. Kontrolni procesi su korišteni kako bi se sustav doveo u željeno stanje. Određeni procesi mogu komunicirati s drugim komponentama te tako provode potrebne promjene. Neki od kontrolnih procesa su*: node controller, job controller, endpoints controller* i drugi.
- etcd – sadrži konfiguracijske podatke i informacije o stanju grozda, unutar ključ-vrijednost baze podataka [11]

4.1.2 Čvorovi

Unutar svakog čvora postoje dvije komponente i jedan proces koji je zaslužan za pokretanje neke od kontejnerskih tehnologija. Dvije komponente su.

- kubelet – zaslužan za pokretanje kontejnera unutar ljuske te komunikaciju između kontrolne ravnine i čvora. Kada kontrolna ravnina zahtjeva neku promjenu, kubelet ju izvršava.
- kube-proxy – održava mrežna pravila za svaki čvor i rukuje komunikacijom prema ili izvan grozda. [11]

4.2 Kubernetes objekti

Kubernetes objekti su entiteti unutar Kubernetes sustava, koji predstavljaju stanje grozda. Njihov zadatak je opisati koje su kontejnerizirane aplikacije pokrenute u grozdu i dostupne resurse za aplikacije. Sadrži police koje opisuju kako će se aplikacije ponašati. Neki od osnovnih Kubernetes objekata su: ljuske, usluge, isporuke, i imenski prostori.

4.2.1 Ljuska

Ljuska je najmanja jedinica Kubernetes sustava. [12] Predstavlja jednu instancu procesa koji se izvodi na grozdu. Ljuske sadrže jedan ili više kontejnera s podijeljenim memorijskim i mrežnim prostorom. Najčešće korištena tehnologija za kontejnere unutar ljuski je Docker. Svaka bi ljuska trebala sadržavati jednu instancu aplikacije u grozdu. Stvaranje ljuski se rijetko kada obavlja ručno. [13] Umjesto toga se koriste razni kontroleri. Ljuske su smatrane kao prolazni, kratkotrajni objekti te se njihov životni ciklus sastoji od nekoliko faza. Prilikom stvaranja, ljuskama je dodijeljen jedinstveni identifikacijski broj te je pripisan čvor gdje će ostati do gašenja. U slučaju pada čvora, ljuskama je zakazano vrijeme brisanja.

4.2.2 Usluga

Usluga (engl. service) je apstraktan način za pokretanje ljuske kao mrežne usluge. Koristi se zbog spomenutog životnog ciklusa ljuski. Ljuskama je pri nastajanju dodijeljen identifikacijski broj te se vrlo lako mogu ugasiti i napraviti nove. Uz jedinstveni identifikacijski broj svaka ljuska pri stvaranju dobiva i vlastitu IP adresu. Zbog ovog problema potrebne su usluge, koji zadaju police i pravila pomoću kojih će se komunicirati s ljuskama. Zbog svog prolaznog životnog vijeka, često se kreiraju nove ili gase stare ljuske te se koristeći usluge svakoj ljusci zadaje određena IP adresa. Neke od vrsta usluga su: ClusterIP, NodePort, LoadBalancer i ExternalName. [14]

4.2.3 Isporuka

Isporuka (engl. deployment) je Kubernetes objekt namijenjen za upravljanje jednom ili više ljuski. [16] Zadužen je za praćenje trenutnog broja ljuski te će u slučaju pada ili brisanja određene ljuske kreirati novu na njenom mjestu. Isporuku se također može koristiti kako

bi se ažurirao kod ili podatci unutar ljuske te ako je potrebno ljusku vratiti na prijašnju verziju.

4.2.4 Imenski prostor

Imenski prostor (engl. namespace) omogućava nastanak više virtualnih grozdova na istom fizičkom grozdu. Korisni su za organizacije ili projekte u kojem veliki broj ljudi dijeli jedan Kubernetes grozd. Prilikom pokretanja Kubernetes sustava postoje četiri zadana imenska prostora:

- default – zadani imenski prostor za sve objekte bez određenog imenskog prostora
- kube-system – imenski prostor za objekte stvorene od strane Kubernetes sustava
- kube-public – nastaje automatski i mogu ga koristiti svi korisnici (čak i ako nisu autorizirani). Unutar njega se nalaze resursi koji su dostupni i vidljivi kroz cijeli grozd
- kube-node-lease – sadrži objekt koji omogućava kubeletu da šalje signale o trenutnom stanju čvora kako bi kontrolna ravnina mogla otkriti pad čvorova [16]

5 Implementacija programskog rješenja

Za demonstraciju navedenih koncepata i tehnologija napravljena je implementacija socijalne mreže u mikroservisnoj arhitekturi. Implementiran je sustav unutar kojeg je svaka funkcionalnost odvojena u poseban servis, sa zasebnom bazom podataka koristeći koncept koji se zove baza podataka po usluzi (engl. database per service).

Ovaj koncept uklanja zavisnost jednog servisa o drugom te je svaki servis u mogućnosti obavljati rad bez da ovisi o bazi podataka drugog servisa. U slučaju pada jednog od servisa, drugi servisi mogu neometano nastaviti rad, što je jedna od velikih prednosti mikroservisne arhitekture.

Kroz implementaciju će biti prikazano kreiranje dockerovih slika, njihova orkestracija putem sustava kubernetes i komuniciranje između servisa putem NATS streaming server-a. Za implementaciju backend servisa korišten je programski jezik Typescript u okviru Express, dok je za frontend korišten javascript u okviru React. Implementirana je MongoDB baza podataka.

5.1 Arhitektura sustava

Sustav se sastoji od šest mikroservisa i podrške za komunikaciju servisa putem poruka. Servisi komuniciraju putem NATS streaming servera koji omogućava slanje poruka između servisa. Određeni servis ima potrebu komunicirati s drugim servisima te mu poslati potrebne informacije za njegov samostalni rad. Servisi koji se nalaze u sustavu su:

- servis za autorizaciju
- servis za objave
- servis za komentare
- servis za pratitelje
- servis za pretragu
- servis za vremensku crtu

5.2 Orkestracija mikroservisa

Za početak je potrebno kreirati Kubernetes objekt koji će stvoriti ljuske za svaki servis. Za to se koristi objekt Kubernetes isporuka prikazan na slici 8 pomoću kojeg se stvara instanca servisa za objave. Konfiguracija za objekt se piše u obliku YAML datoteke.

Objektu za isporuku potrebna je Docker slika koja će biti korištena u stvaranju kontejnera unutar ljuske. Docker slika korištena u kreiranju ove isporuke nastala je putem *dockerfile*-a spomenutog u poglavlju za Docker te je nakon toga pogurana na javni registar Docker slika.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: posts-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: posts
  template:
    metadata:
      labels:
        app: posts
    spec:
      containers:
        - name: posts
          image: fdegmecic/posts
```

Slika 8: Kubernetes isporuka za objave

Prva naredba odnosi se na izbor API verzije Kubernetesa te omogućava odabir određenih objekata unutar tog grupiranja. Ova verzija je potrebna jer se objekt za isporuku nalazi u grupiranju pod *apps/v1*. Sljedeća linija određuje tip objekta i stavlja ga na *Deployment*. *Metadata* određuje koji će se meta podatci zadati ovom objektu. Primjerice, u ovom slučaju zadano je ime *post-depl*.

Spec označava dio gdje se opisuju obilježja ljuske. Za početak definiran je broj ljuski, koji je u ovom primjeru određen na jednu ljusku. Zatim se koristi *template* koji zadaje predložak za izgradnju ove ljuske. Unutar tog predloška određeno je ime kontejnera i Docker slika koju će koristiti za njegovu izgradnju.

Objekti za isporuku za ostale servise nastaju na srodan način te koristeći njih lako se pokreće, zaustavlja ili povećava broj instanci svakog servisa.

Za prikaz svih nastalih Kubernetes usluga, koristi se naredba *kubectl get deployment*. Rezultat naredbe vidljiv je na slici 9.

```
E:\Code\Diplomski\fipubook-dev>kubectl get deployment
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
auth-depl             1/1     1            1           21d
auth-mongo-depl       1/1     1            1           21d
client-depl           1/1     1            1           21d
comments-depl         1/1     1            1           21d
comments-mongo-depl   1/1     1            1           21d
followers-depl        1/1     1            1           21d
followers-mongo-depl  1/1     1            1           21d
nats-depl             1/1     1            1           21d
posts-depl            1/1     1            1           21d
posts-mongo-depl      1/1     1            1           21d
search-depl           1/1     1            1           21d
search-mongo-depl     1/1     1            1           21d
timeline-depl         1/1     1            1           21d
timeline-mongo-depl   1/1     1            1           21d
```

Slika 9: Lista kubernetes isporuka

Naredbom *kubectl get pods* dohvaća se lista ljuski koje nastaju nakon pokretanja Kubernetes usluge što je vidljivo na slici 10.

```
E:\Code\Diplomski\fipubook-dev>kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
auth-depl-58fbd76569-6jq54            1/1     Running   0          15h
auth-mongo-depl-567d8bc8b9-6hfgf      1/1     Running   0          15h
client-depl-76df496d8-qbdcb           1/1     Running   0          15h
comments-depl-5f9dff684d-w94fq        1/1     Running   0          15h
comments-mongo-depl-74745785f7-2xnck  1/1     Running   0          15h
followers-depl-5ff7cc7754-m4jvh       1/1     Running   0          15h
followers-mongo-depl-8c4b7cf85-bkxg6  1/1     Running   0          15h
nats-depl-9c5ddd786-bmgv8             1/1     Running   0          15h
posts-depl-b59896fdd-c6nqt            1/1     Running   0          15h
posts-mongo-depl-6cdf58487c-44p7q     1/1     Running   0          15h
search-depl-75f5f995f8-r6gtb          1/1     Running   0          15h
search-mongo-depl-5f4c9b9fcc-r87hz    1/1     Running   0          15h
timeline-depl-785b6c5497-qbjjx        1/1     Running   0          15h
timeline-mongo-depl-8664865cc5-h7q9c  1/1     Running   0          15h
```

Slika 10: Lista ljusaka

Sljedeći objekt koji je potreban kako bi ljuske mogle komunicirati s vanjskim elementima je Kubernetes usluga prikazana na slici 11.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: posts-srv
spec:
  selector:
    app: posts
  ports:
    - name: posts
      protocol: TCP
      port: 3000
      targetPort: 3000
```

Slika 11: Kubernetes usluga za objave

Kao i kod prethodnog objekta potrebna je YAML datoteka unutar koje se specificiraju postavke Kubernetes usluge. Pod *kind* se definira *Service* te slično kao kod isporuke postavljaju se meta podatci s nazivom *posts-srv*.

Koristeći *spec*, označava se *selector* pomoću kojeg se opisuje naziv ljuske kojoj će sav promet biti preusmjeren. U ovome slučaju to je naziv prethodno navedene ljuske *posts.*

Ako nije izričito određen, Kubernetes kao zadani tip usluge postavlja prethodno navedeni *ClusterIP. ClusterIP* zadaje IP adresu servisu koja je dostupna samo unutar grozda. Usluge za ostale servise nastaju na jednak način.

Koristeći naredbu *kubectl get services* dohvaćaju se sve nastale usluge vidljive na slici 12.

```
E:\Code\Diplomski\fipubook-dev>kubectl get services
NAME                  TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)             AGE
auth-mongo-srv        ClusterIP   10.20.13.253   <none>        27017/TCP           21d
auth-srv              ClusterIP   10.20.12.191   <none>        3000/TCP            21d
client-srv            ClusterIP   10.20.0.79     <none>        3000/TCP            21d
comments-mongo-srv    ClusterIP   10.20.6.119    <none>        27017/TCP           21d
comments-srv          ClusterIP   10.20.13.45    <none>        3000/TCP            21d
followers-mongo-srv   ClusterIP   10.20.9.96     <none>        27017/TCP           21d
followers-srv         ClusterIP   10.20.10.74    <none>        3000/TCP            21d
kubernetes            ClusterIP   10.20.0.1      <none>        443/TCP             21d
nats-srv              ClusterIP   10.20.10.169   <none>        4222/TCP,8222/TCP   21d
posts-mongo-srv       ClusterIP   10.20.5.81     <none>        27017/TCP           21d
posts-srv             ClusterIP   10.20.2.41     <none>        3000/TCP            21d
search-mongo-srv      ClusterIP   10.20.9.111    <none>        27017/TCP           21d
search-srv            ClusterIP   10.20.2.68     <none>        3000/TCP            21d
timeline-mongo-srv    ClusterIP   10.20.9.14     <none>        27017/TCP           21d
timeline-srv          ClusterIP   10.20.2.177    <none>        3000/TCP            21d
```

Slika 12: Lista Kubernetes usluga

5.2.1 Balanser opterećenja

Glavna zadaća balansera opterećenja (engl. load balancer) je poslati promet prema jednoj ljusci u grozdu. [17] Kako bi promet došao do ciljane ljuske osim balansera potreban je i *Ingress-Controller.* To je ljuska koja će sadržavati pravila za usmjeravanje zahtjeva prema kubernetes *ClusterIP* uslugama koje će zahtjev proslijediti instanci aplikacije unutar ljuske.

Ingress-Controller će pogledati putanju dolazećeg zahtjeva te na temelju toga odlučiti kojoj ljusci poslati zahtjev. Vidljivo na slici 13, prikazana su pravila unutar YAML datoteke

Ingress-Controller-a za usmjeravanje zahtjeva od Kubernetes usluga za autorizaciju i objave.

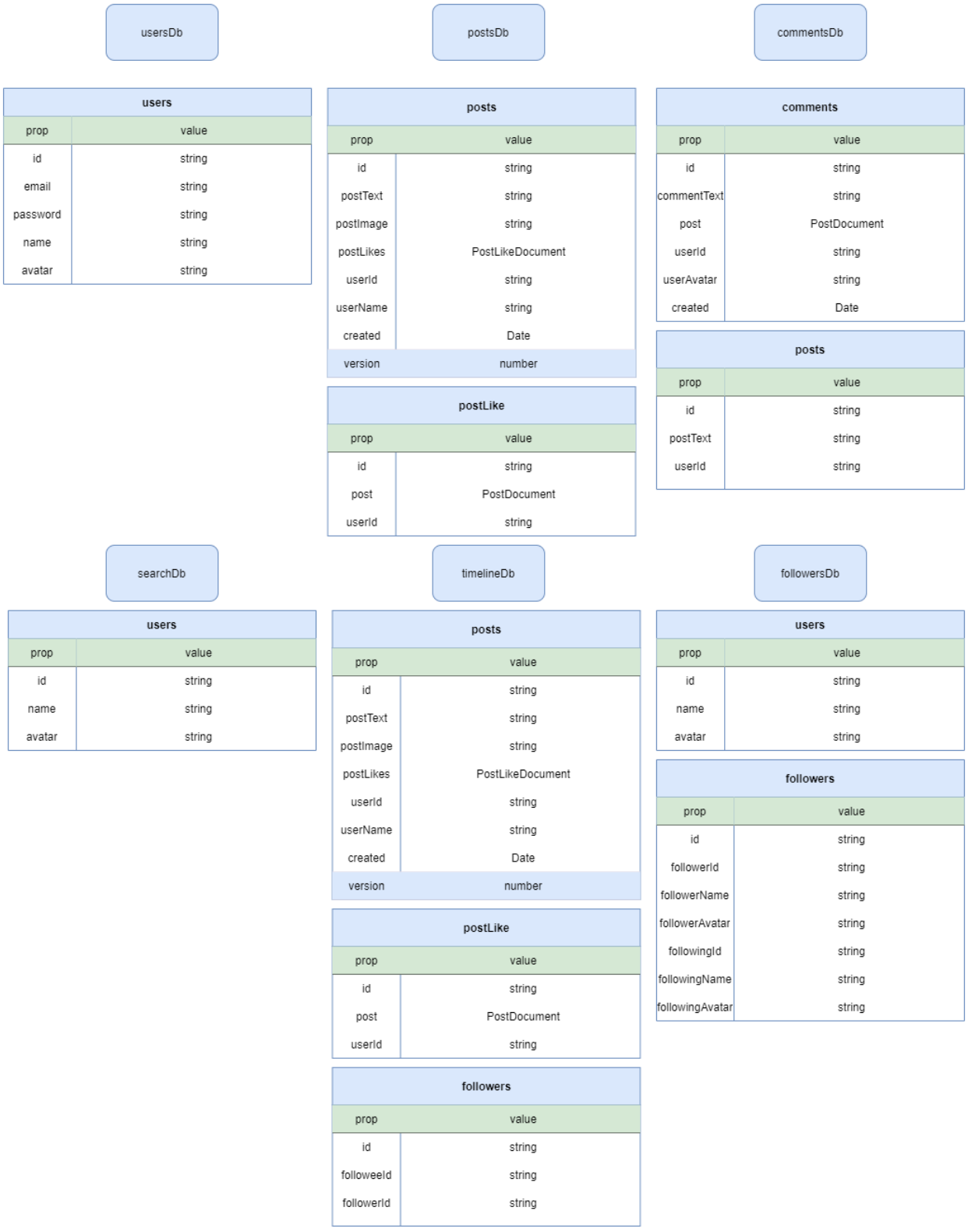
Slika 13: YAML datoteka Ingress-Controller-a

Također na slici, vidljive su putanje zahtjeva na temelju kojeg će se odlučiti kojoj Kubernetes usluzi pripada zahtjev. Na ovaj način definirana su pravila i za ostatak Kubernetes usluga unutar grozda.

5.3 Baza podataka

Na slici 14 prikazana je arhitektura baze podataka gdje svaki zasebni servis ima jednu bazu podataka, često sadržavajući podatke iz drugih servisa. To je neophodno kako bi svaki servis mogao obavljati rad samostalno.

Servisu za komentare potrebno je imati dostupne sve objave, kako bi znao vezati komentar za određenu objavu. U slučaju da servis za objave prestane raditi, servis za komentare mora moći napraviti novi komentar. Za izgradnju novog komentara prvo mora provjeriti postoji li određena objava za koju servis za komentare pokušava povezati određeni komentar. Može to napraviti jer su mu dostupne sve objave i ne mora komunicirati sa servisom za objave kako bi došao do tih podataka.



Slika 14: Baze podataka svih servisa

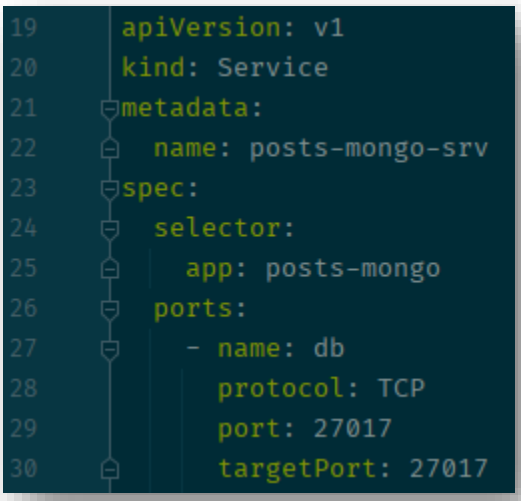
Najsloženija baza podataka nalazi se u servisu za prikaz vremenske crte korisnika. Ona ispisuje objave svih osoba koje korisnik trenutno prati. Da bi to bilo moguće, potrebno je dohvatiti sve objave korisnika te filtrirati samo one objave osoba koje korisnik prati.

Kako bi baza podataka bila dostupna unutar grozda, potrebno je napraviti ljusku putem Kubernetes isporuke i način da ostali elementi čvora komuniciraju s njom. Prvo nastaje YAML datoteka za stvaranje isporuke, putem koje će nastati ljuska baze podataka. Datoteka je prikazana na slici 15.

Slika 15: Kubernetes isporuka baze podataka

Slično Kubernetes isporuci korištenoj za stvaranje instanci servisa, nastaje YAML datoteka za izgradnju ljuske unutar koje će biti baza podataka određenog servisa. Značajna razlika u isporukama jedino se nalazi u imenovanju i korištenju druge Docker slike.

Svaka baza podataka također mora biti dostupna svom pripadajućem servisu te je zbog toga potrebna Kubernetes usluga koja će to omogućiti. Kubernetes usluga za bazu podataka prikazana je na slici 16.



Slika 16: Kubernetes usluga baze podataka

Na identičan način nastaju baze podataka i za ostale servise.

Važna značajka baze podataka za vremensku crtu je da objave sadrže polje pod nazivom verzija, koje je zaslužno za praćenje trenutne verzije određene objave unutar baze podataka. Ovo polje postoji zbog jedne od glavnih prepreka korištenja koncepta baze podataka po servisu, što će biti objašnjeno u sljedećem dijelu rada.

5.4 NATS streaming server

NATS streaming server omogućava komunikaciju između servisa. To obavlja putem poruka poznatih kao događaji (engl. event), unutar kojih jedan servis može poslati drugome određene podatke.

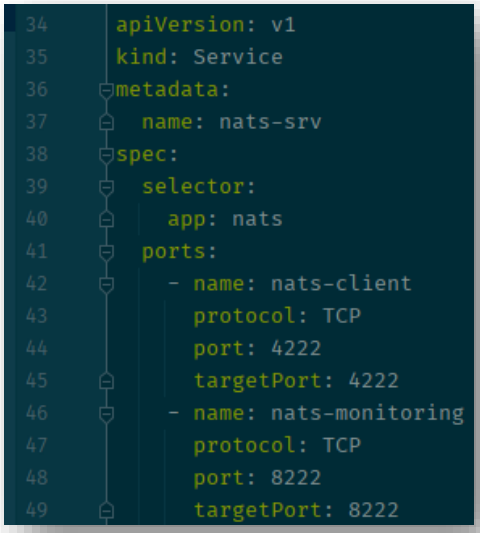
Za početak potrebno je napraviti Kubernetes usluge i isporuke. Slika 17 prikazuje YAML datoteku potrebnu za izgradnju usluge koja se koristi za stvaranje ljuske koja u sebi sadrži kontejner *NATS streaming-a.* Datoteka u sebi sadrži slične postavke kao do sada

navedene Kubernetes usluge s glavnom razlikom u Docker slici koja je korištena te dodatnim argumentima koji su se koristili pri izgradnji slike.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nats-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nats
  template:
    metadata:
      labels:
        app: nats
    spec:
      containers:
        - name: nats
          image: nats-streaming:0.17.0
          args: [
              '-p',
              '4222',
              '-m',
              '8222',
              '-hbi',
              '5s',
              '-hbt',
              '5s',
              '-hbf',
              '2',
              '-SD',
              '-cid',
              'fipubook',
          ]
```

Slika 17: Kubernetes isporuka NATS streaming server-a

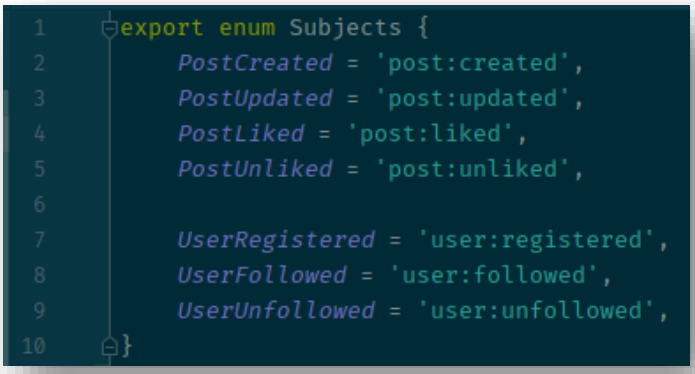
Slika 18 prikazuje YAML datoteku potrebnu za izgradnju Kubernetes usluge koja će izložiti NATS streaming ostatku grozda.



Slika 18: Kubernetes usluga NATS streaming server-a

Ključna razlika nalazi se u postojanju dva *ClusterIP-a* koji zadaje dvije IP adrese za ovu uslugu.

Nakon uspješnog stvaranja ljuski, potrebno je napraviti kanale putem kojih se definira koji će servisi dobivati koje događaje. Kanali za svaku vrstu događaja spremaju se u *enum* prikazan na slici 19.

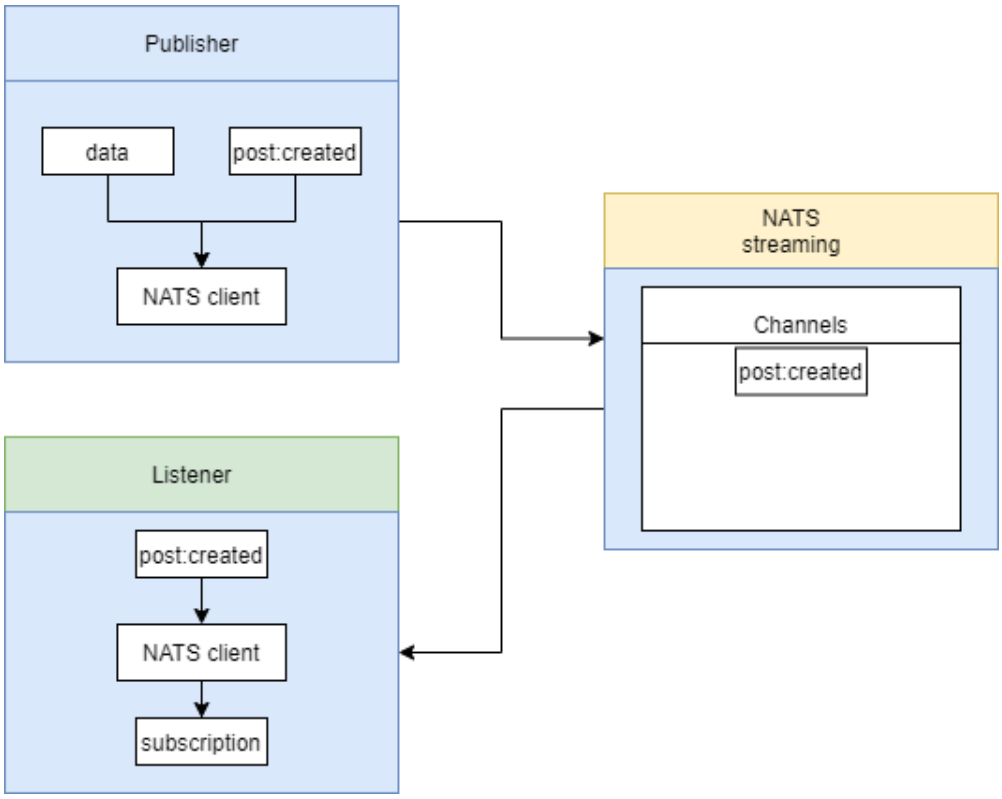


Slika 19: Kanali događaja

Kako bi svi servisi pratili istu konvenciju nazivanja kanala, enum je spremljen u npm paket koji će biti dostupan svim servisima kao zavisnost. Ostatak implementacije NATS

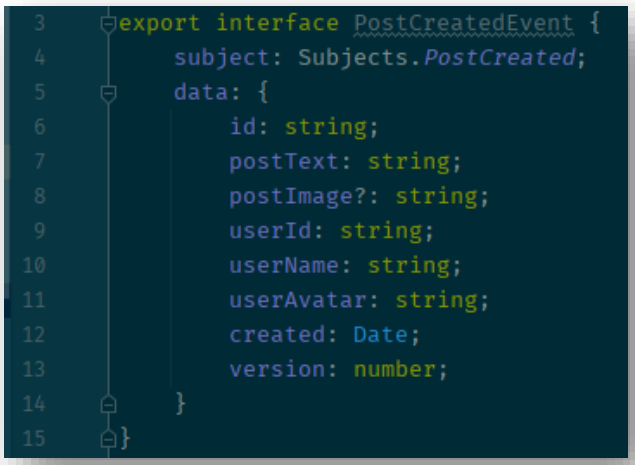
streaming-a također je spremljen u npm paket kako bi se smanjila razina dupliciranog koda i osigurala točna implementacija u svakom servisu.

Sljedeći korak u implementaciji NATS streaming-a je izgraditi izdavača (engl. publisher) i slušatelja (engl. listener) što je vidljivo na slici 20. Izdavač je zaslužan za slanje događaja određenom kanalu s podatcima koje će slušatelj primiti.



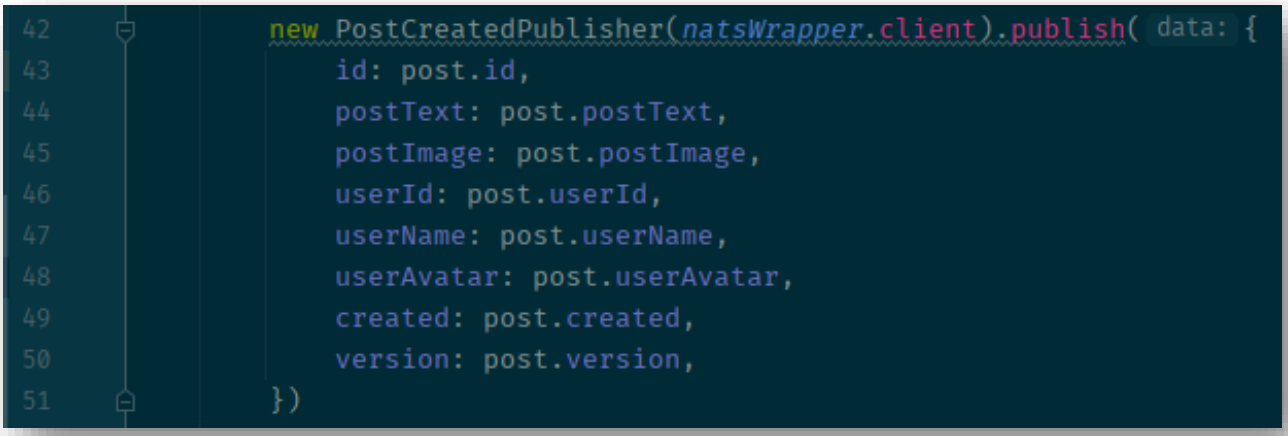
Slika 20: Izdavači i slušatelji događaja

Kako bi svaki izdavač i slušatelj slali točne podatke putem točnih kanala, pri njihovoj implementaciji korišteno je sučelje vidljivo na slici 21 (engl. interface) koje se nalazi u zajedničkom npm paketu.



Slika 21: Sučelje događaja

Nakon definiranja podataka koji će se slati određenim događajem, potrebno je implementirati izdavača koji će taj događaj poslati. Izdavač u primjeru na slici 22 koristi se za slanje događaja o nastanku novih objava. Mjesto implementacije izdavača nalazi se unutar servisa za objave, a događaj se šalje nakon nastanka nove objave. Prethodna varijabla definira koje će podatke izdavač poslati.



Slika 22: Izdavač događaja

Glavni cilj slušatelja je pretplatiti se na određeni kanal te u slučaju događaja obraditi zaprimljene podatke. Kao što je vidljivo na slici 23, slušatelju je definiran kanal PostCreated te će se putem NATS klijenta pretplatiti na sve događaje koji sadrže u sebi definirani kanal.

U sljedećem primjeru slušatelj se nalazi unutar servisa za komentare, kojemu je potrebno imati pristup svim postojećim objavama. Kako bi se to osiguralo, unutar servisa za komentare dodaje se slušatelj, koji je sposoban primiti podatke poslane događajem nakon nastanka objave.

```
export class PostCreatedListener extends Listener<PostCreatedEvent> {
    readonly subject = Subjects.PostCreated;
    queueGroupName = queueGroupName;

    async onMessage(data: PostCreatedEvent["data"], msg: Message) {
        const {id, postText, userId} = data;
        const post = Post.build( attributes: {
            id, postText, userId
        });
        await post.save();

        msg.ack();
    }
}
```

Slika 23: Slušatelj događaja

Nasljeđivanjem Listener klase koja se sastoji od generičkog tipa event, osigurava se konzistentnost kanala slušatelja te podataka koje će taj slušatelj primiti.

Nakon zaprimanja događaja, slušatelj će spremiti pristigle podatke u bazu podataka i tako osigurati servisu pristup objavama. Na kraju funkcije, događaj je potrebno priznati (engl. acknowledge) kako bi NATS klijent znao da je događaj zaprimljen i nije došlo do nikakvih problema unutar slušatelja. U slučaju da događaj nije priznat, izdavač će nakon određenog vremenskog perioda ponovno poslati događaj.

Vidljivo na slici 24 prikazani su razni događaji koje određeni servisi šalju i primaju.

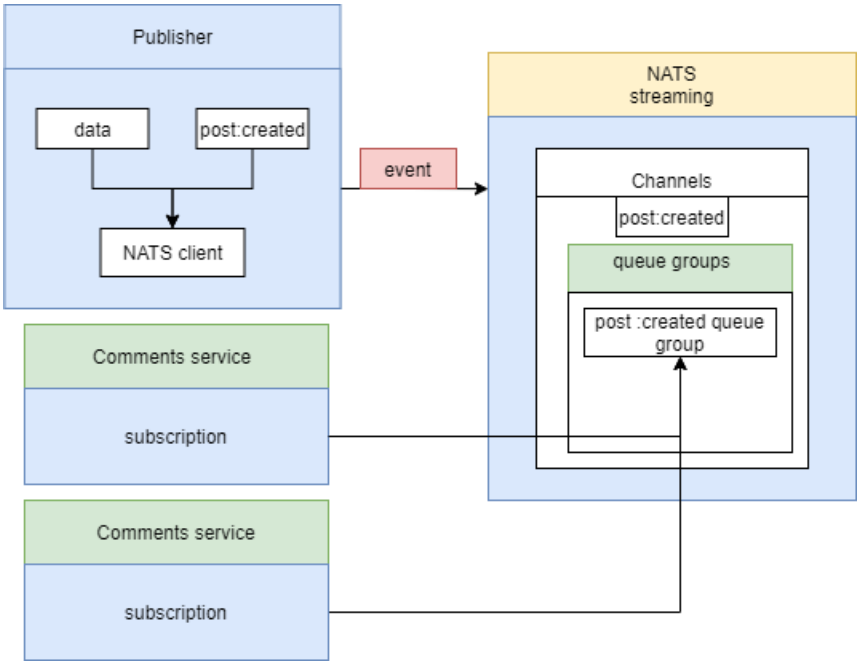
```
[followers] Message received: user:registered / followers-service
[auth] Event published to subject user:registered
[posts] Event published to subject post:created
[comments] Message received: post:created / comments-service
[timeline] Message received: post:created / timeline-service
```

Slika 24: Događaji

5.4.1 Konzistentnost podataka

U slučaju nastanka novih instanci potrebno je odrediti koja će instanca dobiti određeni događaj. To je potrebno kako ne bi došlo do dupliciranja podataka, gdje bi više slušatelja unutar svake instance spremilo događaj u bazu podataka ili obradilo podatke na neki način. Kako bi se takvo dupliciranje izbjeglo implementira se funkcionalnost NATS streaming-a koja se zove grupa čekanja (engl,. queue group).

Grupa čekanja nastaje unutar kanala. Svaki kanal može sadržavati jednu ili više grupa čekanja. Prateći nastanak grupe čekanja, svaka zasebna instanca pridružuje se grupi čekanja putem pretplata, što je vidljivo na slici 25.

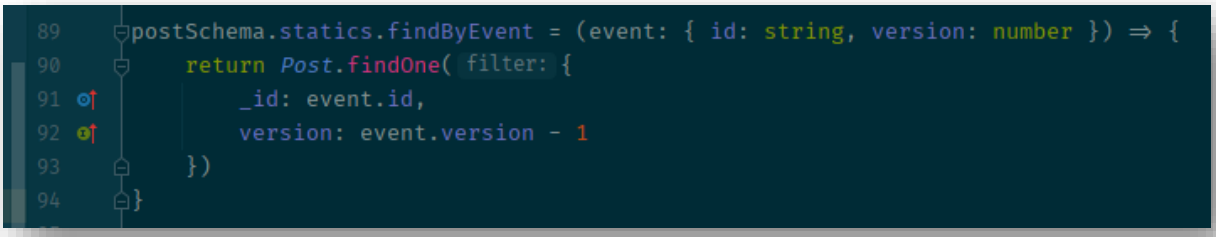


Slika 25: Grupe čekanja

Nakon primanja događaja u određenom kanalu, NATS klijent će nasumično odabrati samo jednu instancu kojoj će poslati događaj. Druge usluge također mogu sadržavati slušatelje koji će pratiti događaje na određenom kanalu bez pretplaćivanja na grupu čekanja te će primati događaje kao i prije implementacije grupe čekanja.

Osim problema s dupliciranjem podataka, još jedna prepreka pojavljuje se kod ažuriranja postojećih podataka u bazama podataka drugih servisa putem događaja. Problem nastaje kada određeni servis pošalje nekoliko događaja odjednom, te servis koji sadrži slušatelje za taj kanal primi događaje krivim redom. Problem također može nastati kada postoji više instanci servisa.

Rješenje problema nalazi se u implementiranju jednostavnog sustava verzioniranja svakog zapisa u bazi. Svaki zapis u bazi sadržavat će polje gdje će biti zapisana trenutna verzija zapisa. Ako se verzija sadržana unutar događaja ne podudara s verzijom unutar baze podataka, poruka će biti odbijena. Poruka nakon određenog vremena bit će poslana opet što bi trebalo dati slušatelju dovoljno vremena da događaj s točnom verzijom primi i potvrdi. Implementacija sustava verzioniranja vidljiva je na slici 26.



Slika 26: Sustav verzioniranja

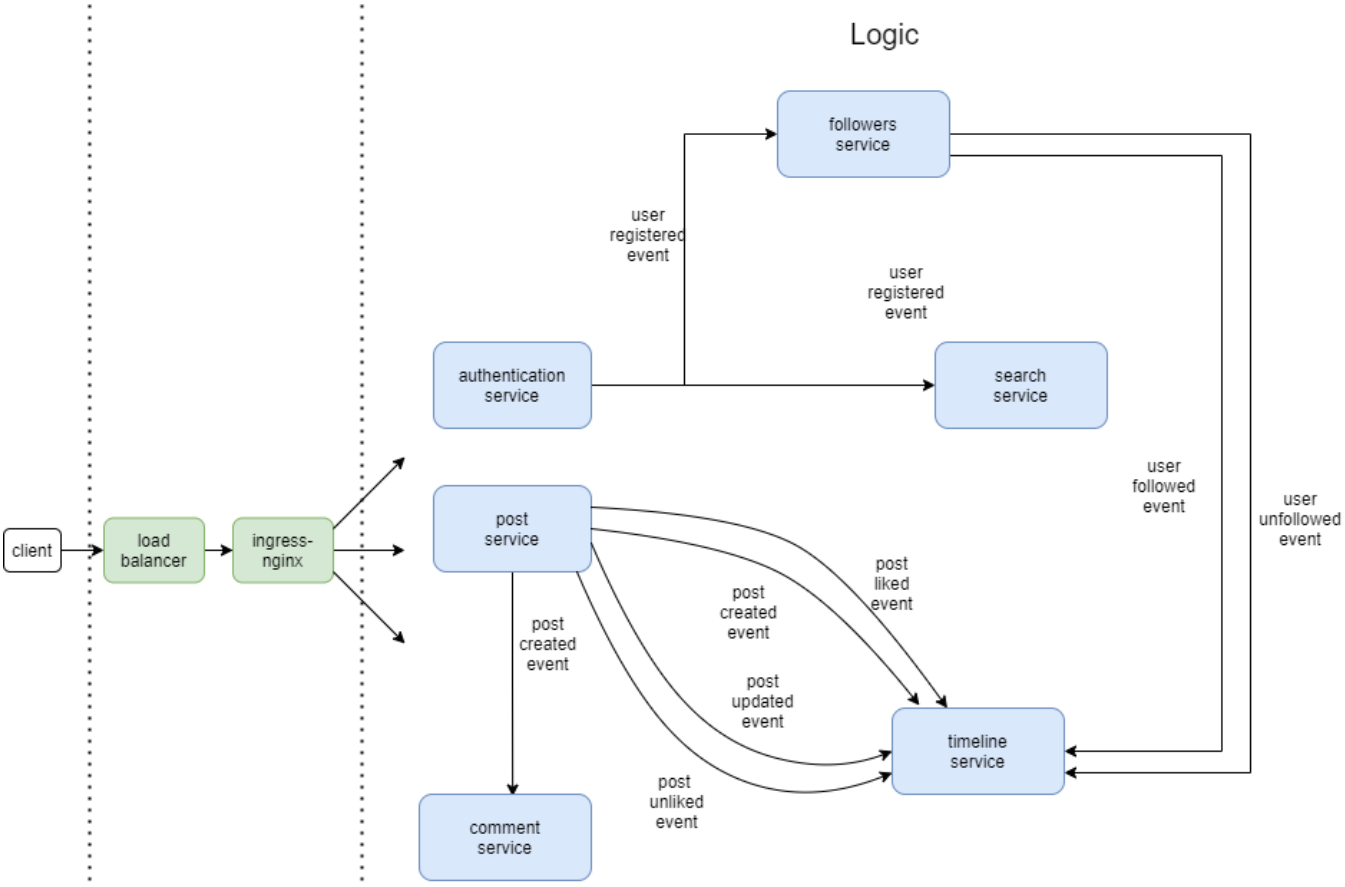
Pomoću ove funkcije prije svakog ažuriranja zapisa u bazi, osigurava se da je verzija unutar događaja točna.

NATS streaming također posjeduje mogućnost spremanja svih događaja u memoriju. Događaji također mogu biti spremljeni na bazu podataka poput MySQL ili PostgreSQL.

Ova funkcionalnost je korisna u slučaju kada određena instanca aplikacije prestane raditi. Pri povratku u rad, NATS je sposoban poslati sve događaje koje je ta instanca propustila.

5.5 Implementacija servisa

Nakon svih konfiguracija preostalo je implementirati svaki servis. Kao što je spomenuto servisi su implementirani u koristeći Express. Servisi su sposobni djelovati samostalno bez zavisnosti o drugim servisima. Servisi i način na koji komuniciraju s ostalim komponentama sustava vidljivo je na slici 27.

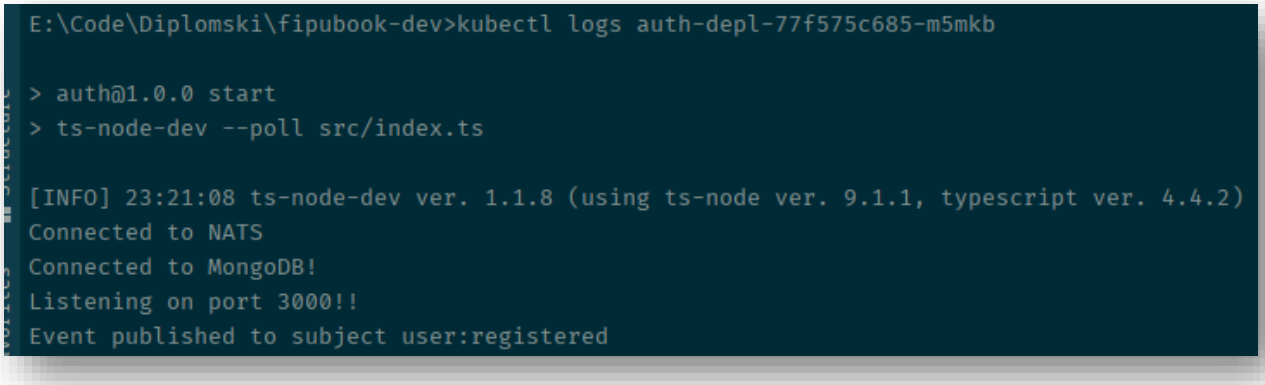


Slika 27: Arhitektura sustava

- *Authentication service*, servis koji se koristi za prijavljivanje, registriranje novih korisnika te postavljanje JWT tokena za autorizaciju. Komunicira događaj *user registered* prema servisu za pratitelje i servisu za pretragu.

- Post service, servis koji je zaslužan za stvaranje i ažuriranje objava. Također omogućava korisnicima da označe objave koje im se sviđaju ili ne sviđaju. Za sve svoje funkcionalnosti šalje događaj prema servisu za vremensku crtu.
- Comment service, servis koji se koristi za stvaranje komentara, potrebno mu je znanje o određenoj objavi kako bi mogao napraviti novi komentar.
- Followers service, zaslužan je za ažuriranje korisnikovih pratitelja i osoba koje on prati. Svoje događaje također šalje servisu za vremensku crtu. Potrebni su mu postojeći korisnici.
- Search service, koristi se za pretragu drugih korisnika. Za samostalan rad, potrebno mu je znanje postojećih korisnika.
- Timeline service, servis koji sadrži sve objave te osobe koje korisnik prati, te pomoću tih podataka korisniku dohvaća njegovu vremensku crtu.

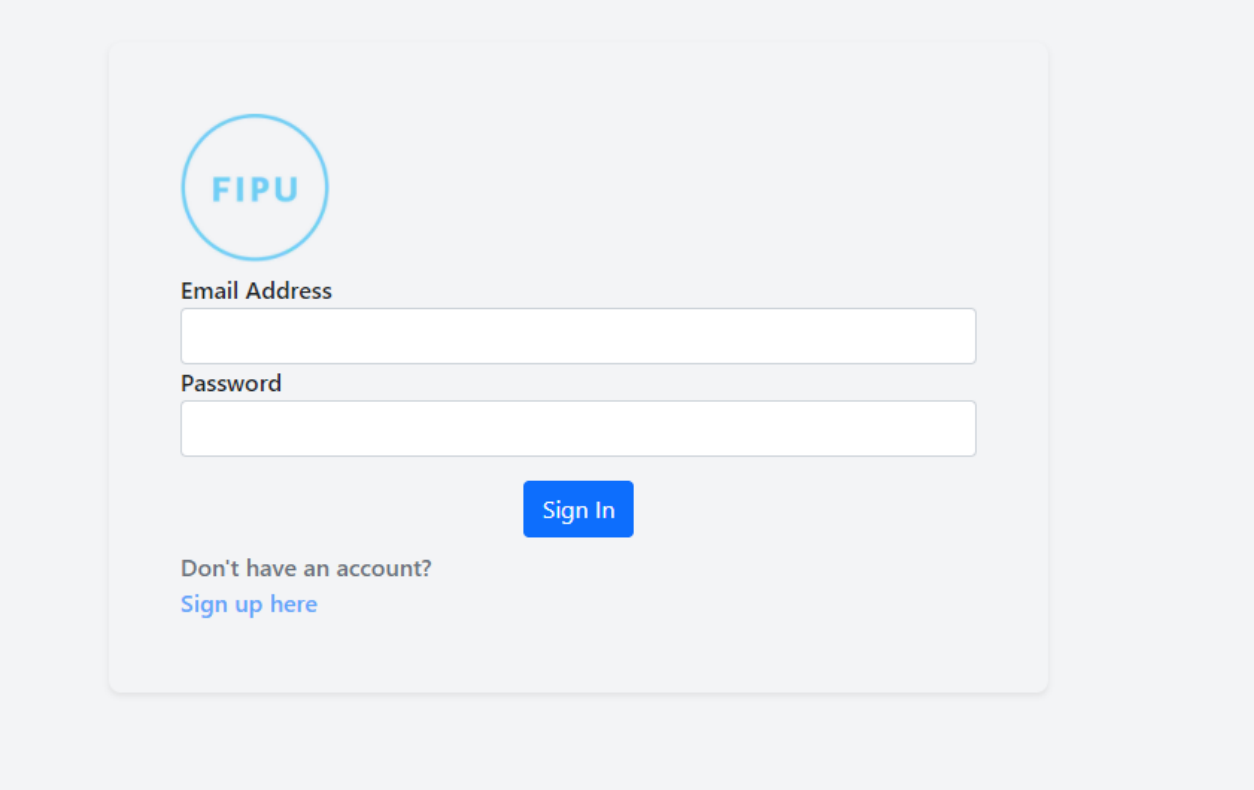
Nakon pokretanja svih ljusaka, u svrhe pronalaska ili otklanjanja greške na pojedinoj ljusci naredbom *kubectl logs <pod>* dohvaća se ispis unutar ljuske pojedinog kontejnera. Jedinstveni naziv određene ljuske dohvaća se prethodno navedenom naredbom *kubectl get pods.* Slika 28 prikazuje ispis nakon izvršene naredbe



Slika 28: Ispis evidencije

5.6 Klijent

Kao posljednji korak programskog rješenja, implementirano je korisničko sučelje. Slika 29 prikazuje klasični prozor za prijavu koji dočeka korisnike pri otvaranju aplikacije.

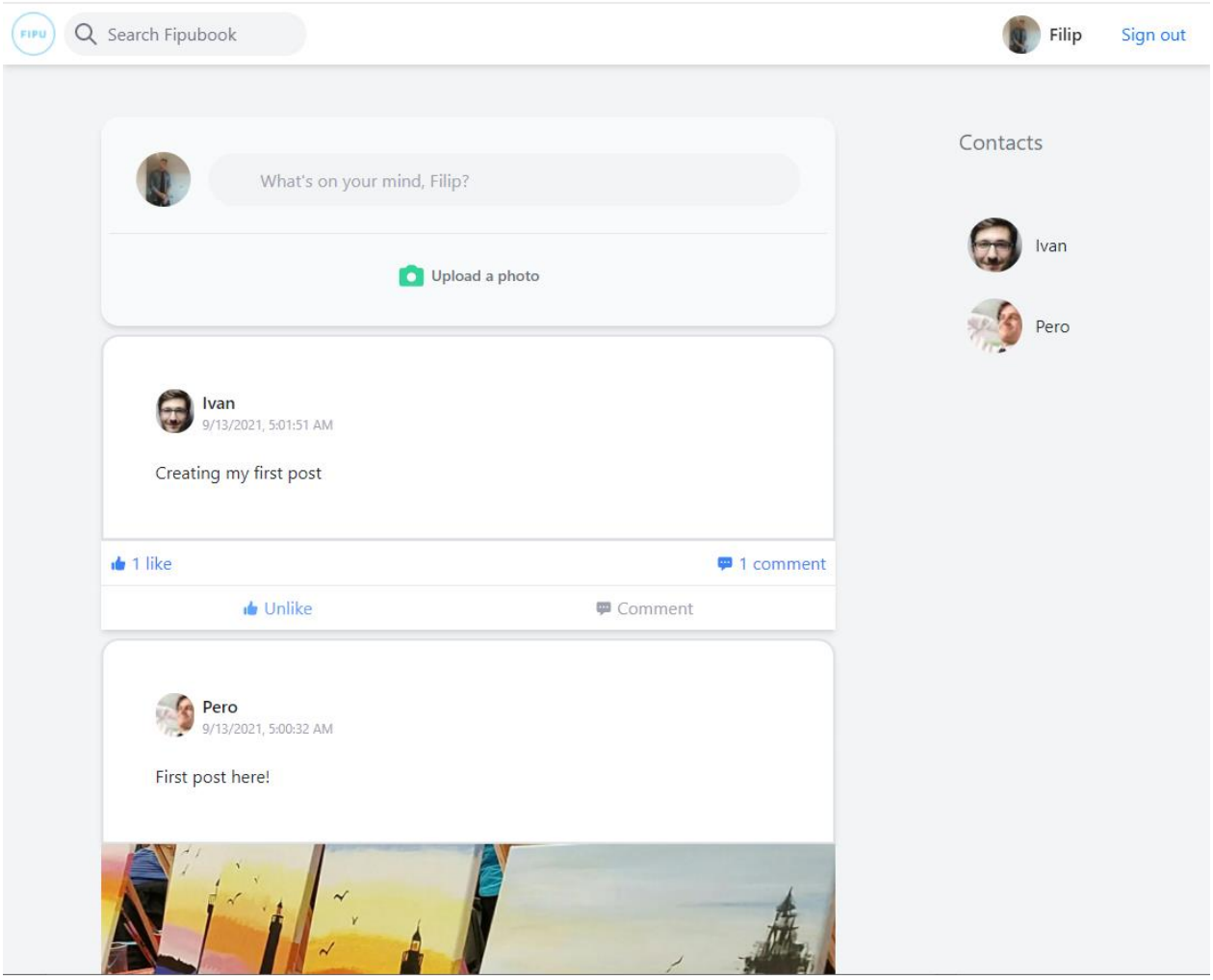


Slika 29: Korisničko sučelje za prijavu

Nakon uspješne prijave, korisnik je preusmjeren na glavni zaslon aplikacije, koji sadržava vremensku crtu korisnika vidljivu na slici 30. Ovdje korisnik može praviti nove objave koje sadrže tekst ili sliku. Također može komentirati na druge objave te označiti da mu se određena objava sviđa ili ne sviđa.

S desne strane zaslona prikazane su osobe koje korisnik prati. Odlaskom na korisnički profil zasebne osobe, korisnik može pratiti ili prestati pratiti druge korisnike.

U gornjem desnom kutu nalazi se prozor za pretragu koja omogućava korisniku pronalazak drugih korisnika.



Slika 30: Korisničko sučelje vremenske crte

6 Zaključak

Tijekom ovog rada razmotrene su razne tehnologije i koncepti potrebni za uspješno implementiranje sustava u mikroservisnoj arhitekturi.

Nakon usporedbe monolita i mikroservisne arhitekture koji predstavljaju dvije suprotnosti u razvoju sustava demonstrirani su alati koji su korišteni kako bi se olakšao razvoj komponenata mikroservisa.

Prvi od alata istražen u radu je Docker. Predstavljeni su teorijski koncepti koji su potrebni za razumijevanje njegova rada. Nakon toga praktično je prikazano korištenje kontejnerskih tehnologija u razvoju mikroservisne arhitekture. Objašnjeno je na koji način izoliranje sustava doprinosi njegovoj sigurnosti i učinkovitosti.

Sljedeći alat Kubernetes korišten je za orkestraciju komponenata unutar mikroservisne arhitekture. Proučena je njegova uloga te način na koji se može upravljati i definirati granicu onoga što se nalazi u sustavu. Također je predstavljena njegova povezanost s alatom Docker. Kubernetes olakšava horizontalno skaliranje sustava jednostavnim procesom povećanja instanci određene komponente.

Jedan od temeljnih izazova mikroservisa je komunikacija između komponenata. Ovaj problem riješen je implementacijom NATS streaming server-a. NATS putem događaja šalje informacije između komponenata te im omogućava samostalan rad bez zavisnosti na druge komponente. NATS dolazi s preprekama čije je rješenje prikazano u implementaciji rada.

Programsko rješenje izvedeno je prateći temeljna obilježja mikroservisne arhitekture. Predstavljene su razne prepreke koje se susreću u razvoj ovakvog sustava te način na koji se one mogu suzbiti. Osim prepreka pokazane su prednosti koje implementacija ove arhitekture donosi.

Implementacija pokazuje kako je moguće izolirati razvoj svake komponente te omogućiti joj rad bez vanjskih zavisnosti. Svaka komponenta unutar sustava može obavljati rad samostalno. U slučaju kvara komponente korisnik će nesmetano koristiti ostatak sustava.

Projekt s izvornim kodom može se pronaći na javnom GitHub repozitoriju (link: https://github.com/fdegmecic/FipubookMicroservices)

Literatura

[1] O'Reilly, O'Reilly's Microservices Adoption in 2020 Report Finds that 92% of Organizations are Experiencing Success with Microservices, URL https://www.businesswire.com/news/home/20200716005101/en/O%E2%80%99Reilly%E2%80%99s-Microservices-Adoption-in-2020-Report-Finds-that-92-of-Organizations-are-Experiencing-Success-with-Microservices

[2] Siraj ul Haq, Introduction to Monolithic Architecture and MicroServices Architecture, 2018, URL https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63

[3] Lokesh Gupta, Microservices – Definition, Principles and Benefits, 2020, URL

https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/

[4] Google Cloud, Introduction to microservices, URL https://cloud.google.com/architecture/microservices-architecture-introduction

[5] Ian Buchanan, Containers vs Virtual Machines, URL https://www.atlassian.com/continuous-delivery/microservices/containers-vs-vms

[6] Docker, Docker overview,URL https://docs.docker.com/get-started/overview/

[7] George Studenko, Docker containers, what are they, how to create and use them,

2019, URL https://www.georgestudenko.com/software-craftsmanship/docker-containers-what-are-they-how-to-create-and-use-them/

[8] Imen Gharsalli, Deep dive into Docker architecture, 2020, URL https://faun.pub/deep-dive-into-docker-architecture-ddb343594056

[9] Emmanuel Christian, How Containerized Applications Increase Speed & Efficiency, 2020, URL https://scoutapm.com/blog/how-containerized-applications-work

[10] Red hat, Introduction to Kubernetes architecture, URL https://www.redhat.com/en/topics/containers/kubernetes-architecture

[11] Kubernetes, Kubernetes Components, URL https://kubernetes.io/docs/concepts/overview/components/

[12] Vmware, What are Kubernetes Pods?, URL https://www.vmware.com/topics/glossary/content/kubernetes-pods

[13] Google Kubernetes Engine, Pod, URL https://cloud.google.com/kubernetes-engine/docs/concepts/pod

[14] Kubernetes, Service, URL https://kubernetes.io/docs/concepts/services-networking/service/

[15] Vmware, What is a Kubernetes Deployment? ,URL https://www.vmware.com/topics/glossary/content/kubernetes-deployment

[16] Kubernetes, Namespaces ,URL https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[17] Avinetworks, Kubernetes Load Balancer, URL, https://avinetworks.com/glossary/kubernetes-load-balancer/

Popis slika

Sažetak

Sve veći broj organizacija prisvaja mikroservisnu arhitekturu kao odabrani način organiziranja razvoja sustava. Mikroservisna arhitektura donosi veliki broj prednosti, ali i određene mane naspram monolitne arhitekture koje su prikazane u ovom radu. Koristeći Docker, svaki servis zapakiran je u kontejner, koji osigurava izolaciju sustava od vanjskih elemenata. Jedna od glavnih prepreka, orkestriranje kontejnerima riješeno je koristeći platformu Kubernetes. Implementacija programskog rješenja koristi princip baza podataka po usluzi te tako omogućava nesmetani rad sustava u slučaju pada zasebnog dijela sustava. Komponente su u stanju zasebno nastaviti rad te nemaju nikakve vanjske zavisnosti.

Ključne riječi: mikroservisna arhitektura, Docker, Kubernetes

Abstract

Increasing number of organisations are adopting a microservice architecture as the desired way of organising their system development. Microservice architecture brings a lot of advantages compared to monolith architecture, but also many drawbacks which are displayed in this thesis. Using Docker, every service is packaged into a container, which provides the system isolation from outside elements. One of the main obstacles, orchestrating containers is resolved by using the Kubernetes platform. This implementation uses database per service principle as a way to offer unobstructed performance in case of a crash in a specific system component. Components are able to operate uninterruptedly and have no outside dependencies

Keywords: microservice architecture, Docker, Kubernetes