UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
U NOVOM SADU

Prikaz funkcionalnosti Kubernetes alata na realnom primeru

Seminarski rad

- MASTER AKADEMSKE STUDIJE -

Student:
Ognjen Kuzmanović
E2 33/2021

Novi Sad, 2021. godine

SADRŽAJ:

1. UVOD

Pre pojave servisno-orijentisanih arhitektura, aplikacije su uglavnom bile zasnovane na monolitnoj arhitekturi što podrazumeva da je sav kod aplikacije na jednom mestu i da se kompajlira zajedno u jednu celinu. Monolitne aplikacije su jednostavnije za razvoj i postavljanje na server (engl. *deployment*) kako bi im korisnici mogli pristupiti. Ovakav vid arhitekture je dobar za manje aplikacije koje razvija manji tim ljudi. Međutim, sa povećanjem količine koda aplikacije, broja članova tima i ukupnog broja korisnika aplikacije, ovakve arhitekture kreću da ispoljavaju svoje mane [1].

- Kako novim, tako i starim članovima tima je sve teže da razumeju celu aplikaciju.
- Menjanje koda na jednom mestu može da prouzrokuje greške na drugom mestu.
- Kompajliranje aplikacije traje sve duže.
- Nije moguće vršiti skaliranje samo jednog dela aplikacije, pa tako u slučaju povećanog broja korisnika, cela aplikacija se mora ponovo instancirati, bez obzira što je samo jedan njen deo pod većim opterećenjem.
- U slučaju pojave novih tehnologija koje bi kvalitetnije rešavale neki problem u nekom delu aplikacije ili želje da se aplikacija prevede u novu tehnologiju, tranzicija na novu tehnologiju može da bude izazovna i neretko zahteva prevođenje kompletnog koda aplikacije što može da bude riskantno.

Sve ove mane monolitnih arhitektura dovode do otežanog, usporenog i neizvesnog daljnjeg razvoja aplikacije. Sa sve većim širenjem interneta i potpuno normalnom pojavom da izrazito veliki broj ljudi u svakom trenutku pristupa nekoj aplikaciji, bilo je jasno da monolitne aplikacije nemaju budućnost za podržavanje prevelikog opterećenja, pogotovo u kombinaciji sa navedenim manama. Zbog toga je nastala potreba za servisno-orijentisanim aplikacijama koje se sastoje iz više malih delova koji zaokružuje neku celinu i rade samo jedan posao koji im je poveren. Svaki servis se razvija nezavisno od strane jednog tima. Takođe, moguće je vršiti skaliranje samo jednog servisa koji je trenutno pod opterećenjem, umesto cele aplikacije, što u mnogome pobošljava performanse i odziv sistema. Početkom ovog milenijuma došlo je do velikog rasta broja aplikacije koje su zasnovane na mikroservisnoj arhitekturi koja predstavlja jednu implementaciju servisno-orijentisanih arhitektura. Česta je preksa da se mikroservisne aplikacije pakuju u kontejnere za koje se naknadno vrši postavljanje na servere. Iako u teoriji, benefiti koji pružaju mikroservisne aplikacije poput izolovanog postavljanja na server i lakšeg skaliranja zvuče primamljivo, u praksi ih nije bilo lako ostvariti do pojave alata poput *Kubernetesa*, pogotovo za aplikacije koje se sastoje iz više hiljada kontejnera koji su raspoređeni svuda po svetu, po raznim *data* centrima. Na inženjerima je ostavljena briga o tome koji kontejneri komuniciraju sa kojim, organizovanje skladišta podataka i njihovo uvezivanje sa kontejnerima, rukovanje sa kontejnerima koji su pali, skaliranje broja kontejnera u slučaju da su neki preopterećeni ili su premalo opterećeni itd. Jasno je da je bio potreban neki alat koji

bi na adekvatan način referencirao sve ove probleme i tu prazninu je popunio upravo *Kubernetes*.

Kubernetes je *open-source* alat kreiran od strane kompanije *Google* 2014. godine i služi za vršenje automatskog *deployment*-a, skaliranja i rukovanja kontejnerskim aplikacijama. On osigurava visoku dostupnost, skalabilnost i performantnost aplikacije, kao i oporavak od grešaka. *Kubernetes* je odmah na početku svog nastanka prepoznat kao kvalitetan alat za upravljanje kontejnerima i iz tog razloga je doživeo veliki uspeh. Pored njega, popularni alati su još *Docker Swarm* i *Apache Mesos*.

U drugom poglavlju ovog rada („*Arhitektura Kubernetesa*"), objašnjena je arhitektura na kojoj se zasniva *Kubernetes* alat, kao i njegove najvažnije komponente. U narednom, trećem poglavlju („*Kubegres operator*"), objašnjen je princip rada *Kubegres* operatora koji je neophodan za rad demo primera čija arhitektura je opisana u četvrtom poglavlju („*Arhitektura demo aplikacije*"). Fokus ovog rada je prvenstveno usmeren ka četvrtvom i petom poglavlju („*Prikaz funkconalnosti Kubernetes alata*") gde su u ovom poglavlju prikazane najvažnije funkcionalnosti *Kubernetes* alata na konkretnoj aplikaciji opisanoj u četvrtom poglavlju. U poslednjem poglavlju, dat je krajnji pregled celog rada i zaključci do kojih je autora rada došao.

2. ARHITEKTURA KUBERNETES-A

U osnovi, rad sa *Kubernetesom* podrazumeva rad sa klasterom koji se sastoji iz glavnih (engl. *master*) i radnih (engl. *worker*) čvorova.

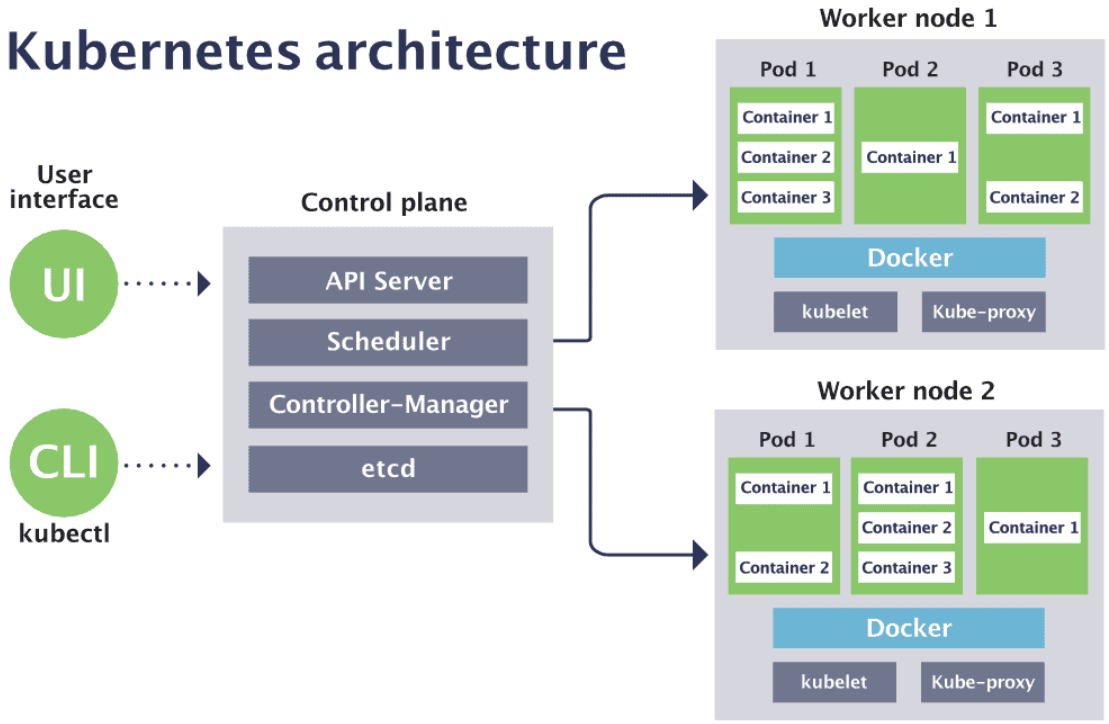
Radni čvorovi su čvorovi klastera u kojima se izvrša kontejnerizovana aplikacija i svaka aplikacija mora da ima makar jedan radni čvor. Svaki radni čvor može da ima jedan ili više podova, gde *pod* predstavlja najmanju *Kubernetes* jedinicu koja vrši apstrakciju nad jednim ili više kontejnera. Svaki radni čvor je podržan radom sledeća 3 procesa:

1. *kubelet* – osnovni proces u radnom čvoru koji vrši instanciranje podova na osnovu prosleđene konfiguracije i brine se o tome da se *pod* izvršava i da je u zdravom stanju
2. *kube-proxy* – na pametan način omogućava pristup podovima tako da je opterećenje na mrežu optimalno
3. *container-runtime* – softver koji je odgovoran za povlačenje slike kontejnera i za njegovo izvršavanje

Pored radnih čvorova, u *Kubernetes* klasteru se nalaze i glavni čvorovi koji su zaduženi za donošenje odluka o celom klasteru, rukovanje i interakciju sa radnim čvorovima. Ta interakcija se najpre odnosi na zakazivanje pokretanja podova, kontinuirani monitoring rada radnih čvorova i podova u njemu, rukovanje propadanjem radnih čvorova i instanciranje novih itd. Svaki glavni čvor se zasniva na radu sledeća 4 procesa:

1. *api-server* – svaki zahtev klijenta koji je upućen ka *Kubernetes* klasteru, prvo prolazi kroz *api server* koji vrši verifikaciju samog zahteva i njegovo prosleđivanje procesu
2. *kube-scheduler* – svaki put kada je potrebno pokrenuti izvršavanje novog poda, potrebno je uputiti zahtev *api serveru* koji će taj zahtev proslediti *kube scheduler*-u. Njegov zadatak je da na pametan način odluči na kom radnom čvoru bi trebalo pokrenuti *pod*, a samo njegovo pokretanje vrši *kubelet* proces u radnom čvoru. Odluka na kojem radnom čvoru će biti pokrenut *pod* najviše zavisi od resursa koji radni čvorovi imaju na raspolaganju, njihovog zauzeća, lokacije podataka, afiniteta čvora koji je moguće definisati kroz specifikaciju itd.
3. *controller-manager* – krucijalan proces zadužen za monitoring rada radnih čvorova i podova unutar njih. Ukoliko se desi da neki radni čvor ili *pod* propadne, ovaj proces bi trebalo da prepozna taj događaj i da izvrši ponovno zakazivanje pokretanja
4. *etcd* – svojevrstan mozak celog klastera koji je zadužen za ispraćanje svake promene koja se desila u klasteru i njeno zapisavanje u *key-value* bazu

Grafički prikaz glavnih i radnih čvorova, zajedno sa procesima unutar svakog od njih dat je na slici 1.



Slika 1. Grafički prikaz arhitekture Kubernetesa
(izvor: www.cncf.io)

2.1 Komponente Kubernetes-a

U nastavku su opisane najvažnije komponente koje su podržane *Kubernetesom*, a koje se koriste u praktičnom delu ovog rada.

2.1.1 Pod

Kao što je već nagovešteno u prethodnom poglavlju, svaki radni čvor sadrži jedan ili više podova, gde *pod* predstavlja najmanju *Kubernetes* jedinicu koja vrši apstrakciju nad jednim ili više kontejnera. *Pod* je obeležen IP adresom koja mu se ponovo dodeljuje svaki put kada *pod* propadne. *Pod* uglavnom sadrži jedan kontejner, ali može i više njih ukoliko neki kontejneri nemaju smisla da postoje nezavisno od nekog drugog kontejnera. Svi kontejneri unutar jednog poda dele iste resurse koji su podu dodeljeni. Podovi se smatraju i osnovnom jedinicom replikacije jer u slučaju većeg opterećenja na neki *pod*, oni se mogu skalirati u više kopija na koje će potom saobraćaj biti optimalno raspoređen.

2.1.2 Deployment

Deployment jeste komponenta koja definiše specifikaciju na osnovu koje se kreiraju podovi i u kojoj je navedeno željeno stanje delova aplikacija definisanih pomoću mikroservisa. Sa takvom ulogom, *deployment* predstavlja apstrakciju nad podovima kojom se najčešće i vrši njihovo upravljanje. U toj specifikaciji se nalaze informacije neophodne za pokretanje podova, kao što su lokacije slika kontejnera koji bi trebalo da se nalaze u podu i broj replika podova. Pomoću *deployment*-a se vrši pokretanja *stateless* aplikacija.

2.1.3 Servisi

Podovi nisu trajni resursi. Oni se kontinuirano kreiraju i uništavaju kako bi u svakom trenutku klaster bio u željenom i definisanom stanju. Nakon što se *pod* kreira, on dobija IP adresu, međutim, ta IP adresa je važeća samo za vreme života poda, a potom se i ona uklanja. Iz tog razloga su nastali servisi kao još jedna *Kubernetes* komponenta koja ima statičku IP adresu koja se ne menja i koja grupiše više podova u jednu celinu. Životni ciklus servisa nije povezan sa životnim ciklusom poda, tako da ako *pod* propadne, to neće imati uticaj na životni ciklus servisa. Pored grupisanja podova, servisi rade i balansiranje opterećenja (engl. *Load balancing*) podova time što koriste algoritme poput *Round-robin* algoritma koji svaki novi zahtev rutiraju na sledeći *pod* i tako u krug.

2.1.4 Ingress

Nije preporučljivo da servisi budu direktno izloženi ka internetu i da svako može direktno da im pristupi. Takođe, izloženost svakog pojedinačnog servisa ka internetu bi otežalo razvoj s obzirom da bi *frontend* inženjeri morali da brinu na kojim lokacijama se nalaze željeni mikroservisi. U slučaju dekomponovanja nekog mikroservisa na više njih ili postojanja više verzija jednog istog mikroservisa, a tranzicija na najnoviji nije još odrađena, razvoj *frontend*-a se dodatno otežava. Iz tog razloga postoji potreba za nekom komponentom koja će se nalazi na samom ulazu u klaster i kroz koju će prolaziti svi zahtevi, a ona bi zahteve rutirala na tačno određene servise na osnovu određenih pravila rutiranja. Takva komponenta se ogleda u *Ingress-u*. Pored toga što radi rutiranje, *Ingress* može da vrši početno i najgrublje balansiranje opterećenja na aplikaciju.

2.1.5 StatefulSet

U sekciji 2.1.2. je rečeno da se *deployment* koristi za *stateless* aplikacije, tj. aplikacije koje ne pamte podatke, već svaki novi zahtev obrađuju na osnovu novoprosleđenih podataka. *Stateless* aplikacije se često oslanjaju na *stateful* aplikacija poput baza podataka koje imaju zadatak da pamte prethodne podatke i da omoguće rukovanje nad njima. Zbog nepostojanja te potrebe da se pamte prethodni podaci, *stateless* aplikacije se pomoću *Kubernetesa* mogu lako instacirati i replicirati. Svaki novoinstacirani *pod* je identičan svim ostalim podovima nastalih pomoću određenog *deployment*-a. S druge strane, rukovanje *stateful* aplikacijama nije tako lako i zahteva uvođenje dodatnih koncepata i komponenti. *Kubernetes* komponenta zadužena za rukovanje *stateful* aplikacijama jeste *StatefulSet*. Umesto dodeljivanje nasumičnog imena podovima kao što to radi *deployment*, *StatefulSet* komponenta obraća pažnju na njihova imena i dodeljuje ih tako da postoji rastući niz brojeva. Konkretnije rečeno, prvo se instacira *pod* koji na kraju svog imena ima broj 0. Nakon što se on instacira, kreće instaciranje novog poda koji će na kraju imena imati broj 1 itd. Bitno je primetiti, da instaciranje novog poda ne kreće dok se ne završi instaciranje poda koji u svom imenu ima broj za jedan manji. U obrnutom redosledu od instaciranja, vrši se brisanje podova. Na taj način se osigurava da svaki *pod* uvek ima isti identifikator bez obzira na njegov životni ciklus. Ovakvo imenovanje pomaže u slučaju kreiranja klastera baza podataka koji sadrži jednu primarnu instancu koja je odgovorna i za čitanje i za pisanje, kao i više replika koje služe samo za čitanje i koje vrše kontinuiranu sinhronizaciju podataka počevši od primarne baze podataka. Primarna instanca enkapsulirana u *pod* bi uvek na kraju svog imena imala broj 0 i

sve ostale komponente bi mogle da se oslone na tu činjenicu. Detaljnije o klasteru baza podataka je diskutovano u trećem poglavlju.

2.1.6 Secret

Tajna (engl. *Secret*) je komponenta pomoću koje se čuvaju osetljive informacije poput lozinke, ključeva, sertifikata itd. Umesto stavljanja ovih osetljivih informacija u kod aplikacije ili specifikaciju poda, one se stavljaju u *secret* komponentu i time smanjuje mogućnost njihovog izlaganja neželjenim korisnicima. Pre nego što se sačuvaju, osetljive informacije se uglavnom konvertuju u *base64* format kako bi se omogućilo inkorporiranje binarnih podataka.

2.1.7 ConfigMap

Konfiguraciona mapa (engl. *config map*) je komponenta koja omogućava razdvajanje konfiguracionih promenljivih od koda aplikacije i korisna je u slučajevima kada više različitih podova koriste iste ili slične konfiguracione detalje jer oni ne moraju da se menjaju u samom kodu ili na više mesta mesta prilikom specifikacije podova. U slučaju postajanja osetljivih informacije, ipak bi se trebalo osloniti na *secret* komponentu iz bezbednosnih razloga.

2.1.8 Skladišta podataka

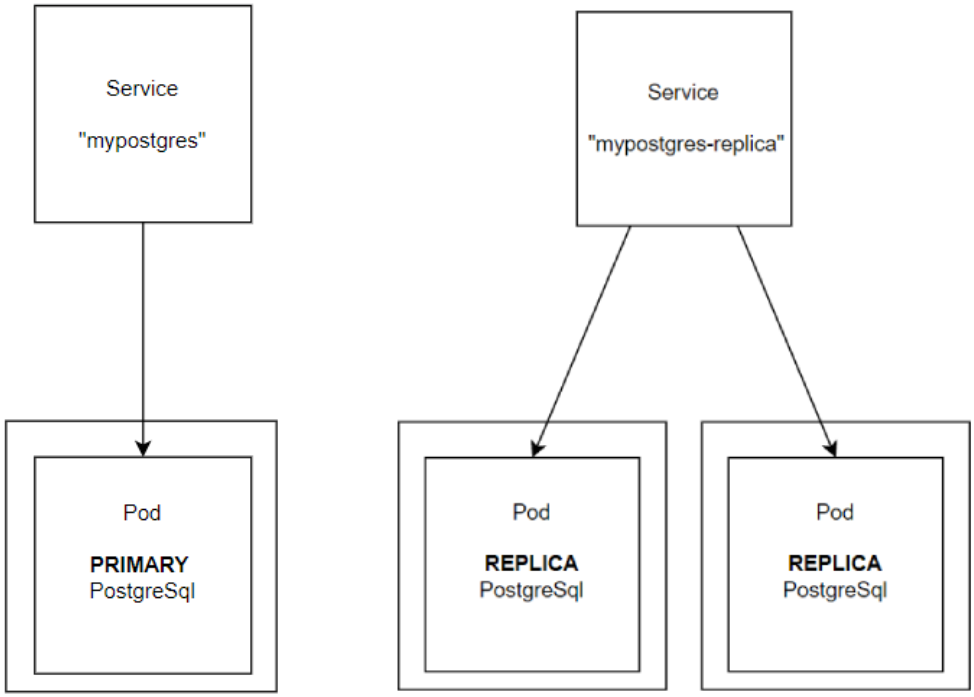
Svaki put kada *pod* propadne, svi podaci koji su generisani u kontejneru koji on apstrahuje nestaju. Ovo može da bude problem ukoliko novi podovi imaju potrebu da se oslone na podatke generisane od strane prethodnih podova. *Kubernetes* uvodi dve komponente koje omogućavaju prevazilaženje ovog problema, a to su *PersistentVolume (PV)* i *PersistentVolumeClaim (PVC)*. *PV* je skladište podataka koje je obezbeđeno od strane administratora klastera, a *PVC* je zahtev za dobijanje skladišta podataka od strane onog ko vrši sam razvoj klastera.

3. KUBEGRES OPERATOR

U sekciji 2.1.5. su nagovešteni problemi koje *StatefulSet* pokušava da reši. U suštini, za *stateful* aplikacije nije tako lako vršiti *deployment*, kao za *stateless* aplikacije. Pogotovo uzevši u obzir da svaka *stateful* aplikacija ima neku sebi svojstvenu problematiku pred sobom kao što su načini na koji se vrši skaliranje, ažuriranje, konfiguracija itd. *Stateful* aplikacije često zahtevaju posedovanje domenski-orijentisano znanje koje imaju eksperti iz neke konkretne oblasti. Kako ovakva znanja nema veliki broj ljudi osmišljen je koncept *Kubernetes* operatora. Pored brojnih automatizacija različitih procesa koji *Kubernetes* operatori mogu ostvariti, oni omogućavaju i olakšan rad sa konkretnim *stateful* aplikacijama jer je sva logika o tome kako bi trebalo rukovati nekom konkretnom *stateful* aplikacijom ugrađena u njih. Oni se ne razvijaju od strane *Kubernetes* tima, već od strane eksternih organizacija ili pojedinaca. Kao tipičan primer *stateful* aplikacije, često se navode baze podataka zbog njihove potrebe da vode računa i o prethodnim podacima koji su došli u system. U praktičnom delu ovog rada, kao *stateful* aplikacija korišćena je *PostgreSQL* baza podataka. Za rukovanje njom, postoji nekoliko različitih operatora, razvijenih od strane različitih kompanija, a u praktičnom delu ovog rada je korišćen *Kubegres* operator.

Kubregres operator je *open-source* operator prvi put objavljen u aprilu 2021. Godine, te spade u vrlo mlade alate i još uvek se aktivno razvija, ali je uzet za potrebe ovog rada zbog njegove jednostavnosti i lakoće podešavanja i korišćenja. *Kubegres* omogućava *deployment* jednog ili više klastera *PostgreSQL* instanci sa podržanom replikacijom podataka između instanci i rukovanjem propadanjem instanci, te ne postoji potreba da održavalac *Kubernetes* klastera brine o tome.

Rad *Kubegres* operatora biće objašnjen na primeru podizanja 3 instance *PostgreSQL* baze podataka. Nakon što se pozove kreiranje klastera *PostgreSQL* baza podataka, kreiraće se *pod* sa nazivom *mypostgres-1-0* i u ovom podu se nalazi primarna baza podataka u koju se može pisati, kao i čitati iz nje. Nakon nje, kreiraća se *pod* sa nazivom *mypostgres-2-0*, a na samom kraju i pod sa nazivom *mypostgres-3-0*. Ova 2 poda predstavljaju replike i iz njih se može samo čitati. *Kubegres* operator je zadužen da vrši kontinuiranu sinhronizaciju podataka između primarne baze podataka i replike. Pa tako, kada se neki podatak upiše u primarnu bazu podataka, on se prosleđuje prvoj replici, a potom ona prosleđuje podatak drugoj replici, tako da će u jednom trenutnku sve instance baze podataka imati isti stanje podataka. Svaki od podova je povezan sa konkretnom *StatefulSet* komponentom koje su se takođe kreirale u pozadini i iz kojih nastaju podovi. Takođe, kreiraju se i 2 servisa – *mypostgres* i *mypostgres-replica* preko kojih je omogućena komunikacija između klijenata i baza podataka. Na prvi servis je povezana samo primarna baza podataka tako da je preko njega omogućeno i čitanje i upis podataka, dok su na drugi servis povezane replike, tako da je omogućava samo čitanje. Grafički prikaz dobijene arhitekture se može videti na slici 2. *Kubegres* se takođe stara da budu kreirane i *PV* i *PVC* komponente kako bi se omogućila trajnost podataka čak i kada se ceo klaster ugasi.



Slika 2. Grafički prikaz Kubegres klastera baza podataka
(izvor: www.kubegres.io/doc)

Kubegres operator vrši kontinuirani monitoring rada baza podataka i ukoliko se desi da primarna baza podataka propadne iz bilo kojeg razloga, pokreće se vraćanje klastera u željeno stanje kroz 2 koraka. U prvom koraku, *Kubegres* provera da li postoji neka replika i ukoliko postoji, ona se promoviše u primarnu bazu. Tokom ovog procesa, pisanje u bazu podataka će biti privremeno onemogućeno, ali s obzirom na postojanje jedne aktivne replike na nju će biti prebačen sav teret čitanja, ali ono će i dalje biti omogućeno i radiće bez prekida pod uslovom da klijent adekvatno koristi postajanje 2 različita servisa opisanih u prethodnom paragrafu. Nakon što se završi promovisanje replike u primarnu bazu podataka, pokreće se kreiranje nove replike kako bi se stanje sistema vratilo na prethodno tj. jedna primarna baza i 2 replike. S druge strane, u slučaju pada replike, samo će se instancirati nova replika. Ono što je vrlo korisno je to da *Kubegres* vodi računa o tome gde se nalaze sve instance baze podataka i trudi se da instance budu što ravnomernije raspoređene po svim radnim čvorovima kako bi se omogućila visoka dostupnost baze podataka, čak i slučaju da neki radni čvor propadne.

4. ARHITEKTURA DEMO APLIKACIJE

Za potrebe praktičnog dela ovog rada, napravljena je jednostavna *Go* aplikacija za naručivanje hrane iz nekog restorana. U svojoj osnovi, ona se sastoji iz 2 mikroservisa – *consumer-service* i *order-servis*, koji su oslonjeni na *PostgreSQL* bazu podataka. Kompletan kod aplikacije se može naći na linku [7].

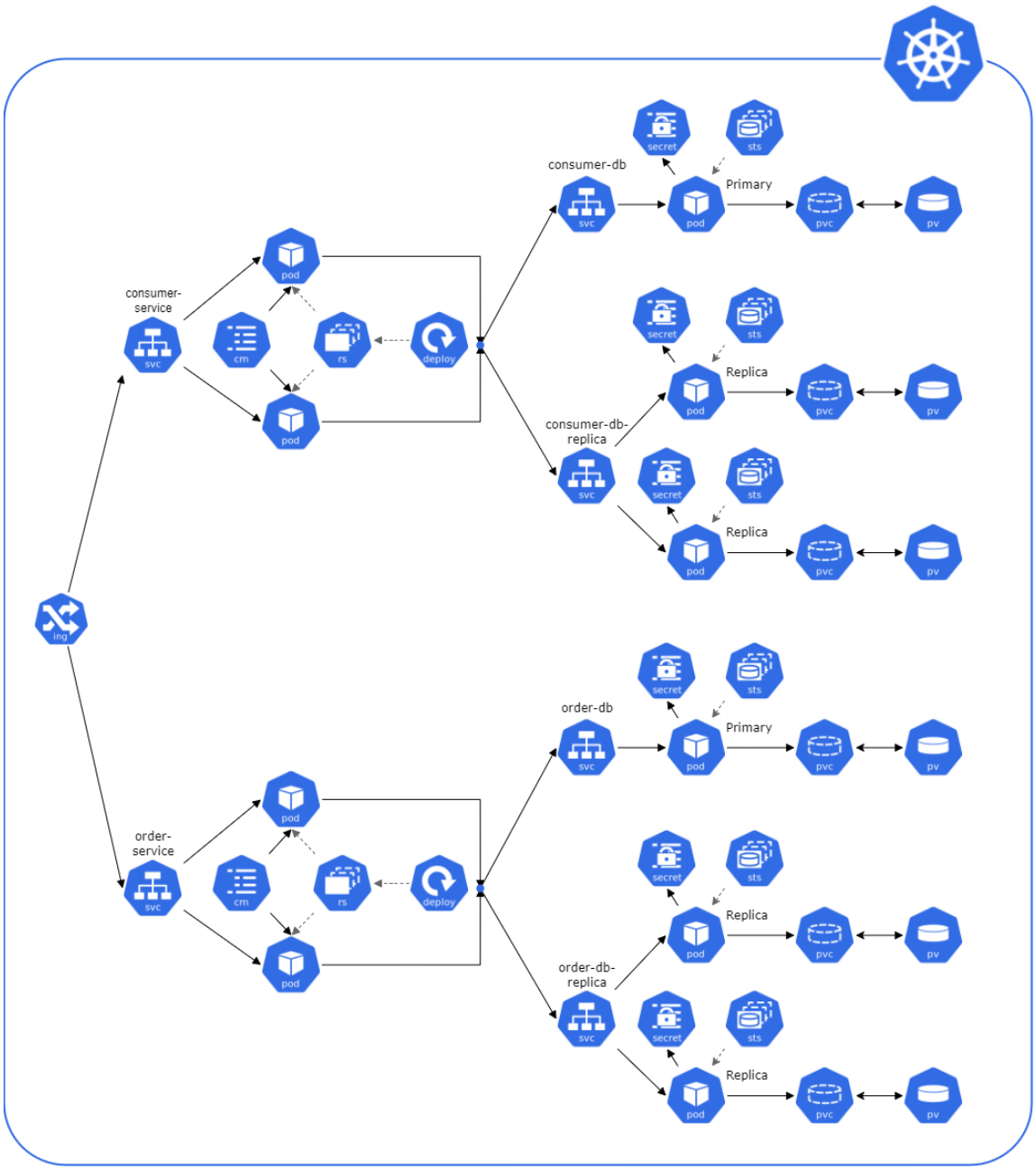
Consumer mikroservis omogućava kreiranja korisnika aplikacije i verifikaciju postojanja nekog korisnika na osnovu njegovog identifikacionog broja. U osnovi, rad ovog mikroservisa je zasnovan na pozivu 4 *REST endpoint*-a:

- GET /hello – vraća *„Hello World"* string na osnovu kojeg se potvrđuje da je mikroservis pokrenut i da se aktivno izvršava
- GET /crash – vrši obaranje celog mikroservisa i služi za demonstraciju rukovanja greškama od strane *Kubernetes* alata
- POST / - prosleđuje mu se informacije o novom korisniku u *JSON* formatu i na osnovu njih se kreira novi korisnik
- GET /verify/:consumerId – vrši verifikaciju postojanja korisnika sa identifikacionom oznakom *consumerId* navedenom u *endpoint*-u

Pored *consumer* mikroservisa, postoji i *order* mikroservis koji se koristi za kreiranje i ažuriranje narudžbina koje sadrže identifikacionu oznaku korisnika koji poručuje hranu zajedno sa imenima obroka koje poručuje i njihovom količinom. Takođe, rad ovog mikroservisa je podržan pozivom 4 *REST endpoint*-a:

- GET /hello – slično kao i kod *consumer* mikroservisa, za proveru izvršavanja mikroservisa
- GET /crash – nasilno obaranje mikroservisa
- POST / - za kreiranje nove narudžbine
- PUT /:orderId/:status – služi za ažuriranje statusa narudžbine sa identifikacionom oznakom *orderID* navedenom u *endpoint*-u

Oba mikroservisa je potrebno instancirati u 2 replike i omogućiti komunikaciju sa *PostgreSQL* bazama podataka. Za potrebe baza podataka, kreirana su 2 njihova klastera – jedan za komunikaciju sa *consumer* mikroservisom, a drugi za komunikaciju sa *order* mikroservisom. Oba klastera sadrže po 3 instance *PostgreSQL* baze podataka i to jedne primarne i dve replike. Detaljan grafički prikaz *Kubernetes* arhitektura ove aplikacije je dat na slici 3.



Slika 3. Grafički prikaz Kubernetes arhitekture demo aplikacije

Na samom ulazu u *Kubernetes* klaster se nalazi *Ingress* komponenta koja ima ulogu da vrši rutiranje ka *consumer* i *order* servisima u zavisnosti od *endpoint*-a ka kojem je HTTP zahtev upućen, a na slici 4 je prikazan izgled *ingress.yml* fajla u kojem je definisana njegova specifikacija. Kao *Ingress controller* je korišćen *Nginx*. Postoje 2 tipa putanja – one koje počinju sa */api/consumer* i za koje se vrši rutiranje na *consumer-service* koji se nalazi na portu 9090 i */api/order* za koje se vrši rutiranje na *order-service* koji se takođe nalazi na portu 9090, ali na drugoj IP adresi.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
    - http:
        paths:
          - path: /api/consumer
            pathType: Prefix
            backend:
              service:
                name: consumer-service
                port:
                  number: 9090
          - path: /api/order
            pathType: Prefix
            backend:
              service:
                name: order-service
                port:
                  number: 9090
```

Slika 4. ingress.yml fajl u kojem je opisana Ingress komponenta

Oba servisa su *LoadBalancer* tipa što podrazumeva da su odgovorni za ravnomerno raspoređivanja tereta na po 2 poda koji on grupišu. Specifikacija *consumer-service* komponente je data na slici 5 i gotovo je identična specifikaciji *order-service* komponente. Suština je da je *spec.type* definisan kao *LoadBalancer*, da je port na kojem se servis izvršava 9090, a da zahteve koji su upućeni ka njemu, usmerava na podove koji se nalaze na portu 9000.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: consumer-service
spec:
  type: LoadBalancer
  selector:
    app: consumer-app
  ports:
    - protocol: TCP
      port: 9090
      targetPort: 9000
```

Slika 5. deo consumer-deployment.yml fajla u kojem je definisan consumer-service

U *consumer-deployment.yml* fajlu (slika 6) definisan je *deployment* na osnovu koga se kreiraju 2 identična poda. U pozadini, *Kubernetes* će kreirati *ReplicaSet* koji će referencirati ove podove.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: consumer-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: consumer-app
  template:
    metadata:
      labels:
        app: consumer-app
    spec:
      containers:
      - name: consumer-app
        image: ognjenkuzmanovic/consumer-app:v2
        imagePullPolicy: Always
        ports:
        - containerPort: 9000
        env:
          - name: DB_PRIMARY
            valueFrom:
              configMapKeyRef:
                name: consumer-db-configmap
                key: db_primary_host
          - name: DB_REPLICA
            valueFrom:
              configMapKeyRef:
                name: consumer-db-configmap
                key: db_replica_host
          - name: DB_USER
            valueFrom:
              secretKeyRef:
                name: consumer-db-secret
                key: user
          - name: DB_PASSWORD
            valueFrom:
              secretKeyRef:
                name: consumer-db-secret
                key: superUserPassword
          - name: DB_NAME
            valueFrom:
              configMapKeyRef:
                name: consumer-db-configmap
                key: db_name
          - name: DB_PORT
            valueFrom:
              configMapKeyRef:
                name: consumer-db-configmap
                key: db_port
          - name: PORT
            value: "9000"
```

Slika 6. consumer-deployment.yml fajl

Kontejneri unutar podova se kreiraju na osnovu slike *ognjenkuzmanovic/consumer-app:v2* koja je okačena na *Docker Hub* registar slika. Podovi se pokreću na portu 9000 i ovu informaciju koristi *consumer-service* za prosleđivanje zahteva ka podovima. Definisano je i nekoliko promenljivih okruženja (engl. *environment variable*):

- DB_PRIMARY – *hostname* servisa koji je povezan sa primarnom instancom *PostgreSQL* baze podataka i preko kojeg aplikacija može da vrši upis i čitanje iz baze
- DB_REPLICA – *hostname* servisa koji je povezan sa replikama baze podataka
- DB_USER – korisnično ime vlasnika baze podataka
- DB_PASSWORD – lozinka za pristup bazama podataka
- DB_NAME – ime baze podataka nad kojom podovi vrše upite
- DB_PORT – port na kome se nalaze baze podataka
- PORT – port na kojem je potrebno izvšiti pokretanje *consumer* mikroservisa

DB_USER i DB_PASSWORD promenljive okruženja se dobija iz *consumer-db-secret* komponente (Slika 7) koja je takođe napravljena za potrebe čuvanja osetljivih informacija baze podataka. Ostale promenljive okruženja se dobijaju iz konfiguracione mape *consumer-db-configmap* (Slika 8). Gotovo identična arhitektura je kreirana za potrebe *order* mikroservisa.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: order-db-secret
  namespace: default
type: Opaque
stringData:
  user: postgres
  superUserPassword: a
  replicationUserPassword: a
```

Slika 7. consumer-db-secret.yml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: consumer-db-configmap
data:
  db_primary_host: consumer-db
  db_replica_host: consumer-db-
replica
  db_name: postgres
  db_port: "5432"
```

Slika 8. consumer-db-configmap.yml

Poslednja stvar o kojoj bi trebalo diskutovati vezana je za kreiranje klastera baza podataka pomoću *Kubegres* operatora. Detaljne instrukcije kako se vrši njegovo pokretanje date su u narednom poglavlju, a njegovo specifikacija je data na slici 9. Za vrstu objektu (*Kind*) je naveden *Kubegres*, broj instanci baza podataka je 3 (jedna primarna i dve replika). Po potrebi, moguće je promeniti inicijalan broj instanci, a preporuka je da postoje barem 3, kako bi savladavanje pada bilo koje instance prošlo bez da to korisnik oseti. Veličina svakog skladišta podataka kojoj svaka instanca ima pristup je 100MB, a za promenljive okruženja su navedene POSTGRES_PASSWORD i POSTGRES_REPLICATION_PASSWORD koje se dobijaju iz *secret* komponente (Slika 7).

```yaml
apiVersion: kubegres.reactive-tech.io/v1
kind: Kubegres
metadata:
  name: consumer-db
  namespace: default

spec:

   replicas: 3
   image: postgres:latest

   database:
      size: 100Mi

   env:
      - name: POSTGRES_PASSWORD
        valueFrom:
           secretKeyRef:
              name: consumer-db-secret
              key: superUserPassword

      - name: POSTGRES_REPLICATION_PASSWORD
        valueFrom:
           secretKeyRef:
              name: consumer-db-secret
              key: replicationUserPassword
```

Slika 9. consumer-db.yml

5. PRIKAZ FUNKCONALNOSTI KUBERNETES ALATA

Za najrealističniji prikaz funkcionalnosti *Kubernetes* alata neophodno je njegov klaster kreirati na distribuiranim resursima raspoređenih svuda po svetu u različitim *data* centrima. S obzirom da autor ovog rada nema pristup takvim resursima, klaster je kreiran lokalno pomoću alata koji se često koristi u ove svrhe, a naziva se *minikube*. Prateći instalaciono uputstvo na *Minkube* zvaničnom vebsajtu (*https://minikube.sigs.k8s.io/docs/start/*), on se može lako instalirati na lokalnu mašinu, a sa njim dolazi i *kubectl*, alat komandne linije (engl. *Command line tool*) koji se koristi za komunikaciju sa *api server* procesom u glavnom čvoru, a posledično omogućava komunikaciju sa *Kubernetes* klasterom. *Kubernetes* klaster korišćen u praktičnom delu ovog rada, na kojem su i prikazane funkcionalnosti *Kubernetes* alata, može se kreirati pozivom komande ispod. Ova komanda će kreirati klaster sa 3 čvora, od kojih će jedan biti glavni čvor, a druga dva čvora radni čvorovi.

```
minikube start --nodes=3 --driver=hyperv
```

Nakon toga, potrebno je eksplicitno omogućiti rad sa *Ingress* kontrolerom pozivom komande ispod.

```
minikube addons enable ingress
```

Pozivom komande ispod, za sve servise i *Ingress* komponente u klasteru se dobijaju URL putanje pomoću kojih im se može pristupiti. S obzirom da se *Ingress* komponenta nalazi na samom ulazu u klaster i da je zadužena za rutiranje eksternog saobraćaja ka internim servisima, njegova URL putanja će biti korišćena kao URL osnova za komunikaciju sa celokupnom veb aplikacijom za naručivanje hrane iz restorana.

```
minikube service list
```

Za potrebe vizuelnog prikaza svih komponenti u klasteru i njihovog zdravlja, u nastavku će biti korišćen *Kubernetes dashboard* veb klijent. Pristup njemu se omogućava pozivom komande ispod. Nakon što ova komanda generiše URL za pristup *Kubernetes* kontrolnom panelu, potrebno je u bilo kojem internet pretraživaču otići na njega.

```
minikube dashboard --url
```

Pre pokretanja samih komponenti klastera, potrebno je uraditi instalaciju *Kubegres* operatora pozivom komande ispod. Ova komanda će kreirati *kubegres-system* imenski prostor unutar *Kubernetes* klastera i u njemu će vršiti instalaciju svih neophodnih *Kubegres* komponenti.

```
kubectl apply -f https://raw.githubusercontent.com/reactive-
tech/kubegres/v1.15/kubegres.yaml
```

5.1 Pokretanje aplikacije

Nakon što su sve prethodne stvari uzete u obzir i uspešno izvršene, može se pristupiti pokretanju svih komponenti u klasteru. Kako pokretanje svih komponenti može da bude zamorno ukoliko se to radi ručno, za potrebe praktičnog dela ovog rada napravljene su dve *bash* skripte – *deploy.sh* i *restart.sh*. Skripta *deploy.sh* (Slika 10) je odgovorna za ponovno kreiranje slika *consumer* i *order* servisa, kao i za njihovo registrovanje na *Docker Hub* repozitorijumu slika.

```
echo -e "deploying consumer-service..."
cd consumer-service
docker build -t ognjenkuzmanovic/consumer-app:v2 .
docker push ognjenkuzmanovic/consumer-app:v2

echo -e "deploying order-service..."
cd ../order-service
docker build -t ognjenkuzmanovic/order-app:v2 .
docker push ognjenkuzmanovic/order-app:v2
```

Slika 10. deploy.sh

```
kubectl delete all --all
kubectl delete kubegres consumer-db
kubectl delete kubegres order-db
kubectl delete -f https://raw.githubusercontent.com/reactive-
tech/kubegres/v1.15/kubegres.yaml

cd kubernetes

kubectl apply -f ingress.yml
kubectl apply -f https://raw.githubusercontent.com/reactive-
tech/kubegres/v1.15/kubegres.yaml

cd consumer
kubectl apply -f consumer-db-configmap.yml
kubectl apply -f consumer-db-secret.yml
kubectl apply -f consumer-db.yml
kubectl apply -f consumer-deployment.yml

cd ../order
kubectl apply -f order-db-configmap.yml
kubectl apply -f order-db-secret.yml
kubectl apply -f order-db.yml
kubectl apply -f order-deployment.yml
```

Slika 11. restart.sh

Skripta *restart.sh* je prikazana na slici 11 i ona je odgovorna za brisanje svih komponenti trenutno prisutnih u klasteru i ponovno pokretanje svih komponenti opisanih u prethodnom poglavlju i to pomoću komande:

```
kubectl apply -f [naziv yml fajla]
```

Kontinuiranim posmatranjem *Kubernetes* kontrolnog panela tokom pokretanja klastera može se uočiti da se *consumer* i *order* aplikacije kao predstavnici *stateless* aplikacija pokreću nezavisno od baza podataka tj. *stateless* aplikacija. Uže gledano, pokretanje klastera baza podataka funkcioniše po principu objašnjenom u poglavlju 3, a svodi se na pokretanje prve (glavne) instance baze podataka, potom jedne replike i na kraju druge replike koje će kontinuiranom sinhronizacijom preuzimati podatke iz glavne instance. Nakon što su svi podovi pokrenuti, aplikacija je spremna za rad i može se koristiti slanjem HTTP zahteva na URL adresu *Ingress* kontrolera.

5.2 Skaliranje podova

Ukoliko se u toku rada aplikacije dođe do zaključka da je inicijalno definisani broj replika nedovoljan da odgovori na potrebe korisnika, skaliranje *stateless* aplikacija je izuzetno lako uraditi i postiže se pozivom komande ispod.

```
kubectl scale –-replicas=[broj replika] –f [naziv yml fajla]
```

Postavljanjem broja replika na 3 za *consumer* podove, na već postojeća 2 će se dodati još jedan koji će *consumer-service* servis prepoznati, a potom će vršiti rutiranje saobraćaja i na njega u skladu sa algoritmom rutiranja. Po potrebi, broj podova se može smanjiti pozivom iste komande. Skaliranje instanci baza podakata unutar *Kubegres* klastera, bi trebalo da funkcioniše dosta slično pozivom komande ispod, ali u trenutku pisanja ovog rada njeno ponašanje je bilo nestabilno, pa bi je trebalo uzeti za rezervom. Prilikom opisa *Kubegres* operatora u poglavlju 3 rečeno je da je ovo mlad alat koji je još uvek na začetku svog razvoja, tako da ni sve funkcije nisu u potpunosti razrađene. Takođe, broj ljudi koji koriste ovaj alat, kao i sama dokumentacija su vrlo oskudni. Ovo za posledicu ima da se problemi vezani za ovaj alat teže i sporije rešavaju.

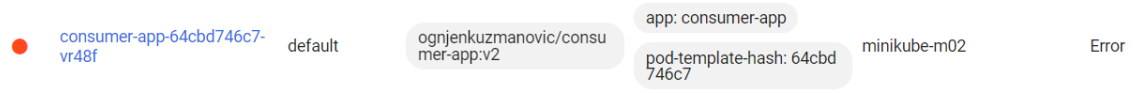
```
kubectl scale sts [naziv stateful set-a] --replicas [broj replika]
```

Poprilično zgodna *Kubernetes* komponenta kada je u pitanju skaliranje jeste *HorizontalPodAutoscaler* (HPA) koja uz pomoć *Kubernetes Metrics Server*-a vrši konstantan monitoring zauzeća računarskih resursa od strane podova i radnih čvorova i automatski vrši instanciranje novih podova u slučaju povećanog opterećenja ili obaranje starih podova ukoliko je njihovo opterećenje zanemarljivo. Ova komponenta nije deo praktičnog dela ovog rada, ali prema dokumentaciji na zvaničnom *Kubernetes* vebsajtu [6], sve što je potrebno uraditi jeste pozvati komandu ispod. HPA kontroler će potom, ažiriranjem *deployment* komponente, podizati i spuštati broj replika tako da održi prosečno zauzeće procesora od 50% na svakom podu. Sa *min* i *max* atributima se specifikuje minimalan i maksimalan broj replika koje je moguće imatu u klasteru.

```
kubectl autoscale deployment [deployment] --cpu-percent=50 --min=1 --max=10
```

5.3 Reakcija na pad podova

Za demonstraciju reakcije *Kubernetesa* na pad *stateless* aplikacija, u oba mikroservisa aplikacije je dodat *GET /crash endpoint* koji bi trebalo da pokrene obaranje poda jedan sekund nakon što je HTTP zahtev upućen na njega. Ubrzo nakon što dođe do obaranja poda, u *Kubernetes* kontrolnom panelu se pojavljuje crvena tačka (Slika 12) pored poda koji je obradio zahtev, a koja sugeriše da je došlo do njegovog obaranja. Odmah nakon što se *pod* obori, *Kubernetes* klaster pomoću *kubelet* i *controller-manager* procesa to prepoznaje i automatski se pokreće novi *pod* kako bi se željeno stanje aktivnih podova vratilo na 2. S obzirom da pored poda koji se obara, postoji još jedan *pod*, aplikacija nastavlja nesmetano da radi sa tim dodatkom da se na njega sada rutira sav saobraćaj, dok se oboreni pod ne vrati u normalno stanje. Nesmetani rad aplikacije dok je jedan *pod* u otkazu se može proveriti slanjem HTTP zahteva na *GET /hello endpoint*. Kao odgovor bi trebalo da se dobije *„Hello World“* string umesto odgovora da je servis nedostupan koji se dobija kada se oba poda obore.



Slika 12. Grafički prikaz oborenog poda

Pad *stateless* aplikacija, konkretnije onih za koje je zadužen *Kubegres* operator ide u dva pravca – pad glavne instace, kao i pad replike. U slučaju da replika iznenadno prekine svoje izvršavanje iz bilo kojeg razloga, biće kreirana nova instanca koja će uraditi sinhronizaciju podataka sa replikom pre nje po brojevnoj nomenklaturi. U slučaju pada glavne instance, ukoliko postoji neka replika, ona biva automatski promovisana u glavnu instancu, a potom se kreira nova instanca koja će dobiti ulogu replike. Tokom periodu promovisanje replike u glavnu instancu, privremeno se onemogućava pisanje u bazu zbog toga što je za pisanje zadužena glavna instanca, ali dokle god postoji još jedna slobodna replika, čitanje će i dalje funkcionisati. Zbog mogućnosti pada glavne instance, i privremenog prekida operacija upisa, autori *Kubegres* operatora preporučuju da se u klijenta baze podataka ugradi logika za ponovno slanje zahteva za upis podataka u bazu ukoliko prvi zahtev ne uspe.

5.4 Održavanje podataka

Nakon što se pošalje HTTP zahtev na *endpoint POST /api/consumer* zajedno sa podacima o novom korisniku u JSON formatu, taj zahtev biva obrađen i podaci se upisuju u glavnu bazu podataka, a potom replike vrši sinhronizaciju kako bi ispratile novu promenu u stanju podataka. Ažurnost replika se može postići ulaskom u kontejner poda replike pomoću komande:

```
kubectl exec consumer-db-2-0 -it  -- bash
```

Pozivom komandi ispod za ulazak u *PostgreSQL* terminal i izvršavanje *select* SQL naredbe za dobijanje svih korisnika, data teza se može i proveriti jer će se novo dodati podaci naći i u replici.

```
psql –U postgres
\c

SELECT * FROM consumers;
```

S obzirom da se za svaki *pod stateless* aplikacije kreira *PersistentVolume* komponente zajedno sa *PersistentVolumeClaim* komponente, bez obzira na to da li je neka instanca baze oborena ili čitav *Kubernetes* klaster spušten, svi podaci će ostati očuvani i biti na raspolaganju prilikom svakog narednog pokretanja.

5.5 Postepeno ažiriranje aplikacije

Vrlo zgodna funkcionalnost *Kubernetesa* je i postepen prelazak na noviju verziju aplikacije. Ukoliko bi se u toku rada celog klastera pozvala komanda ispod, slika *consumer-app deployment* komponente bi bila postepeno vraćena na verziju v1. Pod postepenim se misli na to da bi svi podovi sa verzijom v2 i dalje bili živi, međutim, krenulo bi se instanciranjem jednog poda sa verzijom v1. Nakon što se on kreira, jedan pod sa verzijom v2 se gasi, i tako bi se podovi naizmenično podizali i spuštali sve dok se ne pogase svi stari podovi sa verzijom v2 i ne pokrenu svi novi podovi sa starijom verzijom v1 na koju želimo da pređemo.

```
kubectl set image deployments/consumer-app \
consumer-app=ognjenkuzmanovic/consumer-app:v1
```

Praćenje ovih promena u realnom vremenu se postiže pozivom komande:

```
kubectl rollout status deployments/consumer-app
```

6. ZAKLJUČAK

Iako u teoriji mikroservisne aplikacije imaju brojne prednosti naspram monolitnih aplikacija, njihove prednosti nije tako lako ostvariti bez posedovanja adekvatnog alata. Kao jedan od poznatijih alata se istakao *Kubernetes* kreiran od strane kompanije *Google*. Ovaj alat omogućava olakšano rukovanje mikroservisnim aplikacijama distribuiranim svuda po svetu time što omogućava njihovo skaliranje, visoku performantnosti i dostupnost, rukovanje greškama, tranziciju na nove verzije aplikacije itd.

Motivacija za pisanje ovog rada je bila želja autora da prikaže najvažnije funkcionalnosti *Kubernetes* alata na jednom realnom i konkretnom primeru mikroservisne aplikacije koja se pakovala u *Docker* slike. Reč je o jednostavnoj aplikaciji za naručivanje hrane iz restorana koja se sastoji iz dva mikroservisa – *consumer* i *order*. *Consumer* mikroservis je zadužen za rukovanje korisnicima aplikacije, a *order* za kreiranje narudžbina i ažuriranje njivog statusa. Za svaki od ova dva mikroservisa je napravljen *deployment* fajl u kojem je definisam broj njihovih replika koji bi trebalo da bude održavan u svakom trenutku zajedno sa *Docker* slikom na osnovu koje će oni nastati i parametrima neophodnim za konekciju sa bazom podataka. S obzirom da je svaka baza podataka *stateful* aplikacija koja bi trebala da čuva podatke koji su joj povereni, *Kubernetes* je razvio komponente poput *StatefulSet* koji bi trebalo da pomognu u tome. Svaka *stateful* aplikacija uvodi problematiku za sebe kada je reč o njihovom održavanje, te različiti entuzijasti i organizacije kreiraju takozvane operatore za specifične *stateful* aplikacije. Operator za upravljanjem klasterom *PostgreSQL* baza podataka korišćen u ovom radu se naziva *Kubegres* operator. Iako je reč o dosta mladom alatu koji je još uvek u razvoju, prednost mu je što je lak za upotrebu i konfigurisanje za razliku od ostalih operatora. Ovaj operator nudi mogućnost lakog kreiranje klastera *PostgreSQL* instanci baza podataka koji se sastoji iz glavne instance, koja je odgovorna za čitanje i pisanje u bazu, i replika iz kojih se može vršiti samo čitanje. *Kubegres* operator se stara za to da postoji sinhronizacija podataka između glavne instance i replika kako bi u jednom trenutku svi dosegli isto stanje podataka. Takođe, ovaj operator se stara i za to da sve instance budu ravnomerno raspoređene po svim radnim čvorovima kako bi se povećala otpornost na njihove iznenadne otkaze.

Najvažnije funkcionalnosti *Kubernetes* alata koje su prikazane na konkretnoj aplikacije su skaliranje i spuštanje broja replika podova, reakcija na iznenadne padove podova, održavanje skladišta podataka tako da podaci budu nezavisni od trenutnog životnog ciklusa aplikacije, kao i mogućnost postepenog prelaska na noviju verziju aplikacije.

Ono što je autor uočio prilikom pripreme demo aplikacije i pisanja ovog rada je da pisanje *Kubernetes* specifikacionih fajlova nije teško, međutim potrebno je izvojiti određeno vreme za upoznavanje sa alatom, njegovim različitim komponentama i brojnih konfigurisanjem koje se može ostvariti. Najveći izazov za autora je bilo omogućavanje adekvatnog rada baza podataka tj. *stateful* aplikacija. Kao dominatna rešenja su se izdvajala korišćenja opširnih *PostgreSQL* operatora i pisanje kompleksnih *bash* skripti. Međutim, kao najbolje rešenje se izdvojilo korišćenje *Kubegres* operatora, iako nije toliko poznat i rasprostranjen, ali i poprilično oskudan u dokumentaciji.

LITERATURA

[1] https://microservices.io/patterns/monolithic.html - [Pristupano decembra 2021.]

[2] https://kubernetes.io/docs - [Pristupano decembra 2021.]

[3] https://medium.com/the-programmer/kubernetes-fundamentals-for-absolute-beginners-architecture-components-1f7cda8ea536 - [Pristupano decembra 2021.]

[4] https://medium.datadriveninvestor.com/overview-of-kubernetes-components-ca037b46d79d - [Pristupano decembra 2021.]

[5] https://www.kubegres.io/doc/ - [Pristupano decembra 2021.]

[6] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/ - [Pristupano januara 2021.]

[7] Link do projekta: https://github.com/ogikuzma/kubernetes-demo