

**SVEUČILIŠTE U SPLITU
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I
BRODOGRADNJE**

DIPLOMSKI RAD

**IZRADA SKALABILNE CHAT APLIKACIJE
KOJA SE IZVRŠAVA NA KUBERNETES
SUSTAVU**

Ivo Kovačević

Split, rujan 2020.



Diplomski studij: **Računarstvo**

Smjer/Usmjerenje: -

Oznaka programa: 250

Akadska godina: 2019./2020.

Ime i prezime: **IVO KOVAČEVIĆ**

Broj indeksa: 744-2018

ZADATAK DIPLOMSKOG RADA

Naslov: **IZRADA SKALABILNE CHAT APLIKACIJE KOJA SE IZVRŠAVA NA KUBERNETES SUSTAVU**

Zadatak: Izučiti principe dizajna i izrade skalabilnih web aplikacija. Izučiti način korištenja Kubernetes sustava kao i prednosti koje donosi u radu sa skalabilnim aplikacijama. Izradom chat aplikacije demonstrirati način i prednosti korištenja Kubernetes sustava.

Prijava rada: 13.02.2020.

Rok za predaju rada: 15.09.2020.

Rad predan: 31.08.2020.

Predsjednik
Odbora za diplomski rad:

Mentor:

prof. dr. sc. Sven Gotovac

doc. dr. sc. Marin Bugarić

IZJAVA

Ovom izjavom potvrđujem da sam diplomski rad s naslovom (Izrada skalabilne chat aplikacije koja se izvršava na Kubernetes sustavu) pod mentorstvom (doc. dr. sc. Marin Bugarić) pisao samostalno, primijenivši znanja i vještine stečene tijekom studiranja na Fakultetu elektrotehnike, strojarstva i brodogradnje, kao i metodologiju znanstveno-istraživačkog rada, te uz korištenje literature koja je navedena u radu. Spoznaje, stavove, zaključke, teorije i zakonitosti drugih autora koje sam izravno ili parafrazirajući naveo/la u završnom radu citirao/la sam i povezao/la s korištenim bibliografskim jedinicama.

Student

Ivo Kovačević

SADRŽAJ

1.	UVOD	1
2.	MOTIVACIJA ZA UVOĐENJE KUBERNETES SUSTAVA.....	2
2.1	Prelazak na mikro-servis arhitekturu	2
2.2	Uvod u container tehnologiju	4
2.3	Potreba za Kubernetes sustavom	6
3.	UVOD U KUBERNETES SUSTAV	7
3.1	Pojednostavljeni prikaz rada Kubernetes sustava.....	7
3.2	Arhitektura Kubernetes sustava	7
3.3	Izvršavanje aplikacije na Kubernetes klasteru.....	9
3.4	Prednosti korištenja Kubernetes sustava	10
4.	PRVA APLIKACIJA NA KUBERNETES SUSTAVU	12
5.	TEMELJNI POJMOVI KUBERNETES SUSTAVA	19
5.1	Pods	19
5.1.1	Oznake Pod-ova	21
5.1.2	Objava Pod opisa kroz yaml datoteku	22
5.2	Deployments	23
5.2.1	Kontrole ispravnosti Pod-a	24
5.2.2	Objava Deployment opisa i demonstracija automatske zamjene neispravnih Pod-ova	25
5.3	Services.....	27
5.3.1	Vrste servisa	30
5.4	Volumes.....	33
5.4.1	Trajno skladište	35
5.5	ConfigMaps i Secrets.....	36
6.	AUTOMATSKO SKALIRANJE.....	39
6.1.1	Primjer automatskog skaliranja	40

7.	CHAT APLIKACIJA	44
7.1	Opis.....	44
7.2	Funkcionalnosti	45
7.3	Izvršavanje na Kubernetes klasteru	49
8.	ZAKLJUČAK	62
	LITERATURA.....	64
	POPIS OZNAKA I KRATICA	65
	SAŽETAK.....	66
	SUMMARY	67
	DODATAK A	68

1. UVOD

Novi trend razvijanja aplikacija su svakako mikro-servisi. Mikro-servis arhitektura se uveliko razlikuje od tradicionalnog razvoja aplikacija gdje je cijela aplikacija „zapakirana“ kao jedna velika cjelina. Takve aplikacije nazivaju se monolitne aplikacije.

Monolitan razvoj je i dalje vrlo raširen iako ima mnogo nedostataka. Samim tim što je aplikacija jedna velika cjelina gdje su svi „servisi“ strogo povezani, jako je teško nadograđivati i mijenjati aplikaciju pa su nove verzije aplikacija vrlo rijetke. Nakon što se takve aplikacije naprave, programeri aplikaciju prosljede Ops timu. Ops tim potom aplikaciju ručno migrira na „zdravi“ (engl. healthy) server. Ukoliko dođe do prekida rada servera, aplikaciju je potrebno opet ručno migrirati na drugi „zdravi“ server.

Mikro-servis arhitektura je skup više nezavisnih, samostalnih, odvojenih servisa. Svaki servis se razvija zasebno, najčešće od strane manjeg tima unutar organizacije te se potom migrira (engl. deploy) i održava neovisno o ostalim servisima. Ovakva arhitektura donosi mnoge prednosti u odnosu na tradicionalni razvoj. Glavna prednost je svakako mogućnost učestalijeg mijenjanja pojedinih komponenti ovisno o poslovnim zahtjevima.

Kod velikih aplikacija mikro-servis arhitektura može rezultirati velikim brojem zasebnih servisa. Kod kompanija kao što su Amazon i Netflix taj broj prelazi nekoliko tisuća. Upravljanje s nekoliko tisuća zasebnih mikro-servisa bi svakako bilo nemoguće da ne postoje alati koji nam olakšavaju i omogućavaju taj posao. Jedan od tih alata je i Kubernetes sustav koji će biti objašnjen u okviru ovog diplomskog rada. Rad će biti fokusiran na Kubernetes sustavu u oblaku (engl. cloud) radije nego na Kubernetes sustavu koji se nalazi na vlastitom klasteru s ograničenim hardverom.

2. MOTIVACIJA ZA UVOĐENJE KUBERNETES SUSTAVA

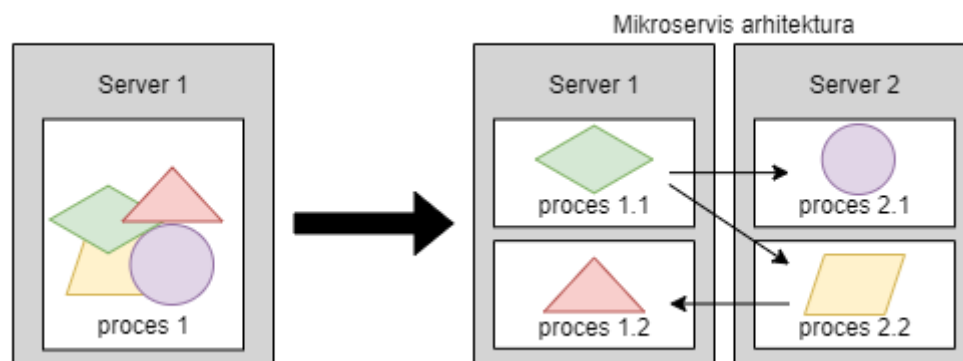
2.1 Prelazak na mikro-servis arhitekturu

Monolitne aplikacije sastoje se od više strogo povezanih aplikacija ili servisa koji se razvijaju, migriraju i održavaju kao cjelina. Takva aplikacija je u biti jedan proces koji se vrti unutar servera. Za svaku i najmanju promjenu neke od komponenti, cijela aplikacija se treba iznova migrirati na server (engl. redeploy).

Ovakva arhitektura predstavlja ograničenje u skaliranju aplikacije pri promjeni opterećenja. Postoje dva tipa skaliranja: vertikalno i horizontalno skaliranje. Vertikalno skaliranje podrazumijeva nadogradnju servera dodavanjem procesorske snage, memorije, povećanje frekvencije procesora itd. Horizontalno skaliranje odnosi se na dodavanje više servera koji su najčešće slabijih performansi nego kod vertikalnog skaliranja, ali je ključ u njihovom broju. Ti serveri posjeduju istu instancu aplikacije te se opterećenje raspoređuje između servera. Ukoliko je neki server u velikom opterećenju novi zahtjevi će se prebaciti na drugi, manje opterećen server.

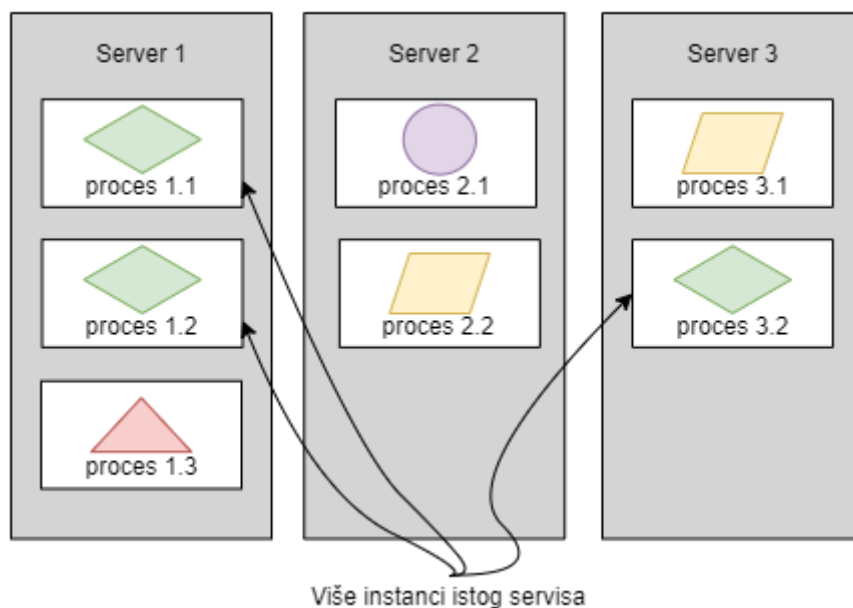
Monolitne aplikacije vrlo se lako vertikalno skaliraju. Zamislimo scenarij da posjedujemo Internet trgovinu koja prodaje kreme za sunčanje. Lako je zaključiti da će u periodu od jeseni do proljeća posjećenost naše Internet trgovine biti vrlo mala. Za takvo opterećenje će vrlo vjerojatno biti dovoljni prosječni serveri. Pri dolasku sunčanih ljetnih mjeseci, za očekivati je da će posjećenost naše stranice naglo porasti. Moguć je scenarij da naš prosječni server neće biti u mogućnosti podnijeti nagli porast opterećenja. Jedno od rješenja bi bilo vertikalno skalirati našu aplikaciju dodavanjem procesorske snage i memorije. Ukoliko se radi o manjoj Internet trgovini ovo će vrlo vjerojatno riješiti problem. U ovoj priči radi se o velikoj Internet trgovini koja drži monopol prodaje krema za sunčanje u cijelom svijetu. Može se pomisliti: „Pa zašto ne bismo dodali još, još i još procesorske snage i memorije?“. Naime, vertikalno skaliranje ima svoja ograničenja. Glavno ograničenje svakako bi bila cijena. Snažni i moćni hardver je vrlo skup. Drugo ograničenje je proizvodnja i dostupnost hardvera na tržištu. Iako danas na tržište vrlo brzo izlaze nove i bolje verzije hardvera opet ćemo doći do točke gdje više nećemo moći dobiti bolji server nego što ga imamo. Kao ograničenje svakako treba izdvojiti i rizik prestanka rada servera. Ukoliko imamo samo jedan server, prestankom rada tog servera usluga koju pružamo će biti nedostupna. Može se naslutiti da smo dosegli vrhunac vertikalnog skaliranja i da moramo pronaći neko drugo rješenje. To bi bilo horizontalno skaliranje. Međutim, monolitne aplikacije je najčešće vrlo teško horizontalno skalirati jer su komponente strogo povezane. Ukoliko samo jedan dio monolitne aplikacije nije skalabilan,

cijela aplikacije će biti neskalicabilna. Uvjet za horizontalno skaliranje je neovisnost komponenti. Da bi ispunili taj preduvjet monolitnu aplikaciju moramo „razbiti“ na više neovisnih servisa gdje svaki servis predstavlja jedan nezavisni proces (Slika 2.1.). Proces i mogu međusobno komunicirati. Komunikacija među procesima može se ostvariti velikim brojem protokola koji nisu strogo vezani za određeni programski jezik nego su implementirani gotovo u svakom programskom jeziku. To su HTTP, TCP, AMQP itd.



Slika 2.1. Prelazak s tradicionalnog razvoja na mikro-servis arhitekturu

Budući da je svaki mikro-servis neovisni proces, moguće ih je migrirati, razvijati i održavati neovisno o drugim mikro-servisima. Promjena na jednom mikro-servisu ne uzrokuje „redploy“ ostalih mikro-servisa. Posljedično tome, kao rezultat dobijemo mogućnost skaliranja svakog mikro-servisa zasebno, neovisno o drugim mikro-servisima (Slika 2.2.). Na taj način možemo skalirati samo one mikro-servise koji zahtijevaju više resursa, a ne cijelu aplikaciju.



Slika 2.2. Svaki mikro-servis može biti skaliran zasebno

2.2 Uvod u container tehnologiju

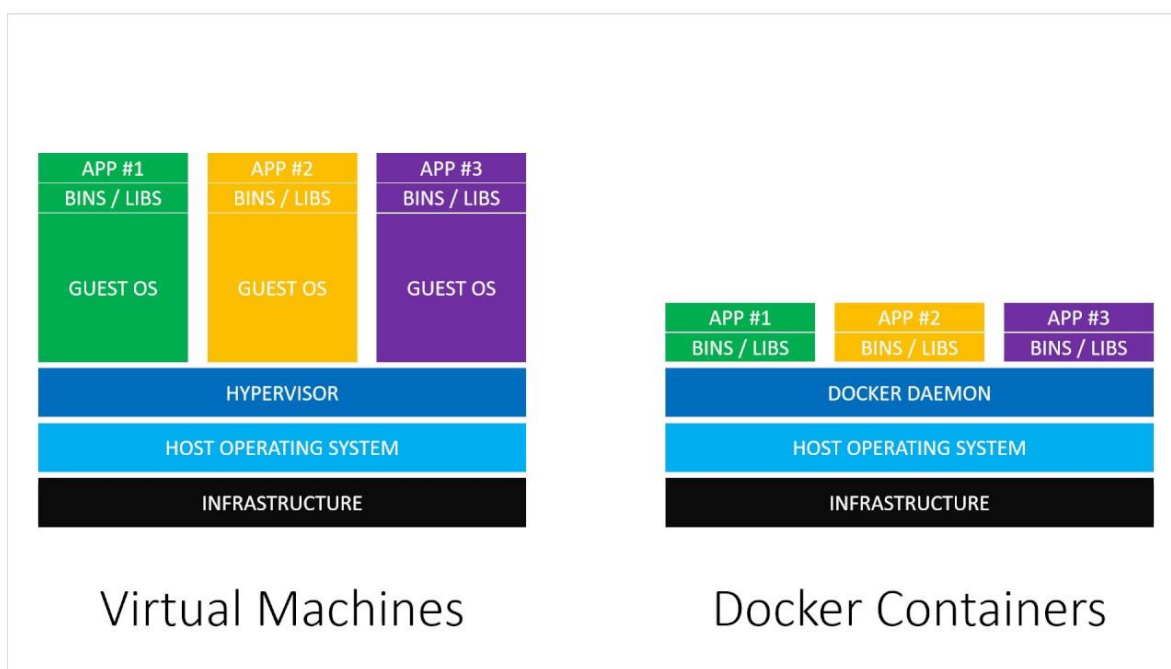
Kod migracije aplikacija na server, može se dogoditi da različite aplikacije koriste različite verzije biblioteka. Kako je već rečeno, kod mikro-servis arhitekture, najčešće se razvoj aplikacije radi unutar manjih timova od kojih svaki tim razvija jedan zasebni servis. To također može rezultirati korištenjem različitih verzija biblioteka između timova što će rezultirati brojnim problemima kod migracije na isti server.

Kako bi se riješio spomenuti problem, pribjegava se korištenju virtualizacijskih tehnologija kao što su virtualne mašine i „container“ tehnologije. Virtualizacijom se postiže neovisne, izolirane okoline pogodne za izvršavanje aplikacija. Kod aplikacija koje se sastoje od manjeg broja komponenti, moguće je svaku komponentu izolirati unutar vlastite virtualne mašine (VM). Svaka VM ima svoj operacijski sustav (OS). Kada se broj ovakvih komponenti počne povećavati, dodjeljivanjem VM svakoj komponenti predstavlja veliki gubitak hardverskih resursa iz razloga što VM imaju veliki „overhead“ budući da se unutar svake VM izvršava vlastiti OS. Osim hardverskih resursa, upravljanje (engl. manage) VM odvija se zasebno što predstavlja i veliki trošak ljudskih resursa.

Umjesto korištenje virtualnih mašina za osiguravanje izolirane cjeline za svaki mikro-servis koristi se Linux „container“ tehnologija. Container tehnologija ima znatno manji „overhead“ od virtualnih mašina jer se izvršavaju na operacijskom sustavu domaćina (engl. host) umjesto da svaki container ima vlastiti OS unutar kojeg se izvršava proces. Proces unutar containera je

također izoliran kao i kod VM. Korištenje container tehnologije naspram VM rezultira većim brojem aplikacija koje je moguće smjestiti na jedan server.

Na slici (Slika 2.3.) vidimo spomenute razlike dvaju tehnologija. Uočavamo da svaka VM ima svoj vlastiti OS koji je na slici označen kao „Guest OS“ dok se aplikacije kod container tehnologije izvršavaju na OS hosta označenog kao „Host OS“. To je ujedno i glavna prednost VM u odnosu na container tehnologiju jer se postiže potpuna izoliranost što rezultira i većom sigurnošću.

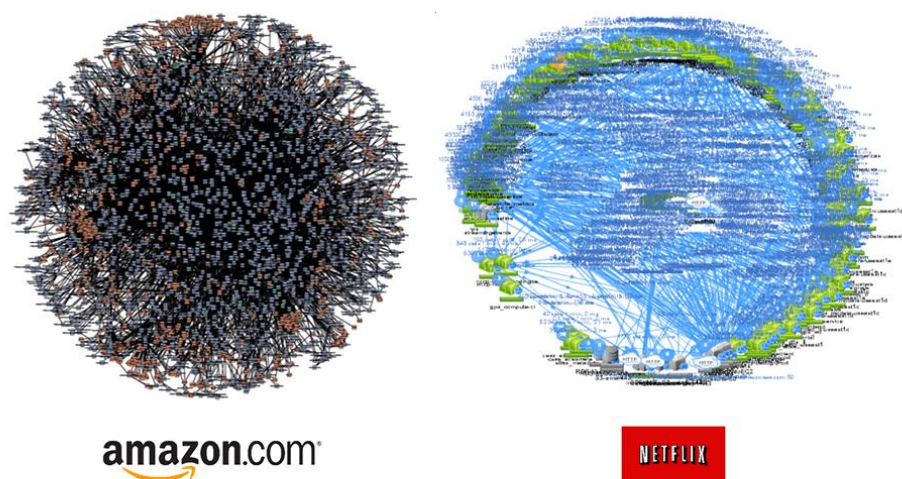


Slika 2.3. Usporedba VM i container tehnologije [1]

Najpopularnija container tehnologija je svakako Docker. Docker omogućava „pakiranje“ aplikacija, zavisnih biblioteka, pa čak i cijelog OS datotečnog sustava unutar jednog, jednostavnog paketa koji se lako prenosi na druge mašine. Kada se takav paket prenese na drugu mašinu, on vidi identični datotečni sustav kao i kada je zapakiran. To omogućava rad aplikacije čak i na kompletno različitom operacijskom sustavu od onoga na kojem je aplikacija razvijana. Na primjer, ukoliko razvijamo našu aplikaciju na Ubuntu Linux distribuciji, a server na kojem ćemo migrirati aplikaciju posjeduje Debian distribuciju, Docker tehnologija omogućava da zapakirani Docker container i dalje vjeruje kako se izvršava u Ubuntu. Jedini uvjet je da sve mašine između kojih se razmjenjuju Docker containeri imaju instaliran Docker.

2.3 Potreba za Kubernetes sustavom

Iako je monolitni razvoj puno brži, dugoročno gledano, svakako se isplati uložiti vrijeme u mikro-servis arhitekturu. Velika većina startup-ova u početku razvoja krene sa monolitnim pristupom te vrlo brzo naiđe na ograničenja. Mnoge velike kompanije su shvatile niz prednosti mikro-servis arhitekture te se odlučile prebaciti na istu. Arhitektura kompanija kao što su Amazon i Netflix se sastoji od nekoliko tisuća mikro-servisa koji međusobno komuniciraju (Slika 2.1.). Gotovo nemoguće je upravljati s nekoliko tisuća mikro-servisa koji komuniciraju i zajedno obavljaju neki složeniji posao. Uzročno tome, došlo je do potrebe za sustavom koji će olakšati razvoj aplikacije temeljene na mikro-servis arhitekturi.



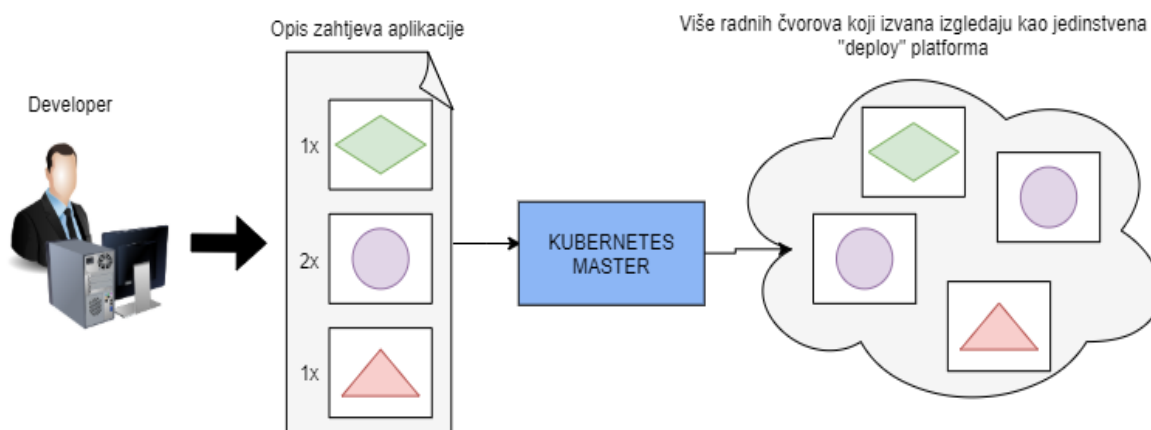
Slika 2.4. Mikro-servis arhitektura Amazon-a i Netflix-a [2]

Google kompanija je prva uvidjela ovaj problem te se odlučila na razvoj vlastitog, besplatnog i svima dostupnog rješenja koji su nazvali Kubernetes. Kubernetes je programsko rješenje koje omogućava jednostavno raspoređivanje i upravljanje container aplikacijama [3]. Zbog toga što su containeri izolirane cjeline, ne mogu utjecati na rad drugih aplikacija na istom serveru. To je vrlo važno svojstvo iz razloga što Cloud kompanije nastoje što bolje iskoristiti resurse, stavljajući različite aplikacije na isti server. Upravo na tom svojstvu se temelji i Kubernetes sustav. Migracija aplikacije na server kod Kubernetes sustava je vrlo jednostavna. Postupak je uvijek isti i ne razlikuje se o broju čvorova u klasteru. Dakle, isto je da li klaster ima jedan čvor ili 100 čvorova, što je uveliko doprinijelo prihvaćanju Kubernetes softvera i masivnom korištenju istog.

3. UVOD U KUBERNETES SUSTAV

3.1 Pojednostavljeni prikaz rada Kubernetes sustava

Najveći nivo apstrakcije Kubernetes sustava izgleda kao na slici (Slika 3.1.). Sustav se sastoji od glavnog čvora (Kubernetes master) i radnih čvorova. Broj radnih čvorova nije ni na koji način ograničen. Može se kretati od samo jednog čvora pa do preko tisuću radnih čvorova, ovisno o potrebi. Developer pri migraciji aplikacije na server specificira koliko instanci pojedine komponente aplikacije želi, sve ostalo je zadaća Kubernetes sustava. Kubernetes potom, na osnovu specificiranih zahtjeva, komponente aplikacije migrira na klaster radnih čvorova. Nebitno je na koji će radni čvor Kubernetes smjestiti komponentu aplikacije. Bitno je samo da su ispunjeni uvjeti koje je developer specificirao. Kubernetes sustav će samostalno zaključiti koji je radni čvor najpogodniji za izvršavanje pojedine komponente. Također, developer može naznačiti ukoliko je potrebno da se neke komponente nalaze na istom čvoru te će sukladno tome Kubernetes navedene komponente rasporediti na isti radni čvor. Ukoliko nije posebno naglašeno, Kubernetes će imati potpunu slobodu rasporeda komponenti na dostupne radne čvorove. Komponente će i dalje moći međusobno komunicirati iako se nalaze nasumično raspoređene na klasteru radnih čvorova.

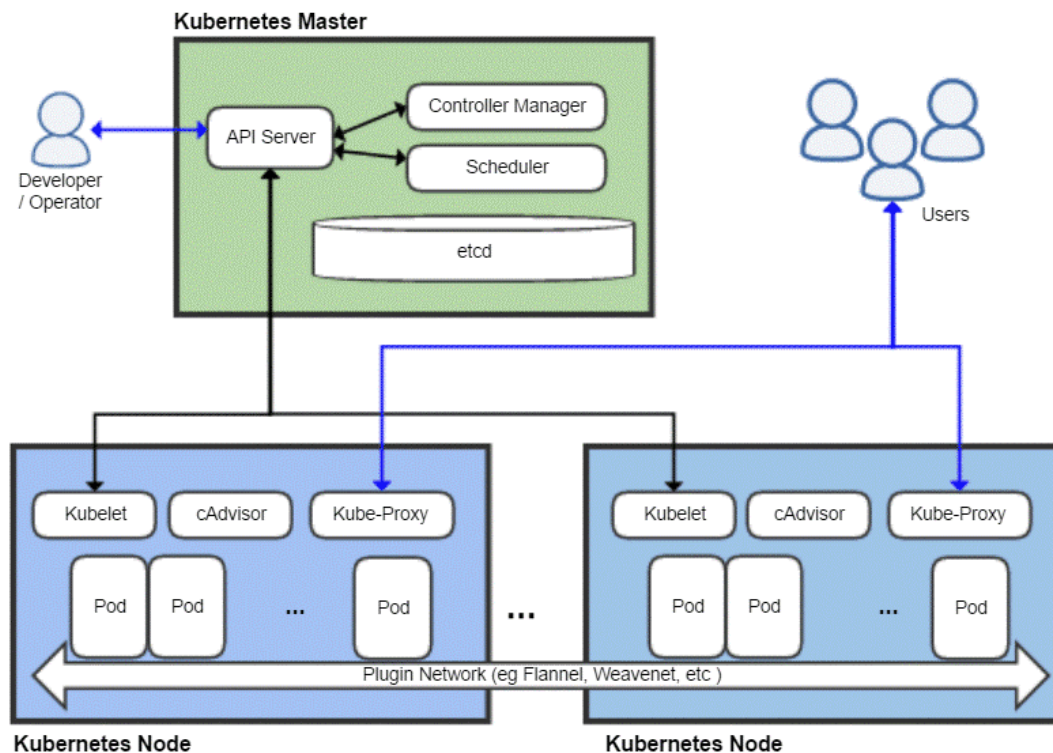


Slika 3.1. Proces „deploy-a“ na Kubernetes cluster

3.2 Arhitektura Kubernetes sustava

Arhitektura kubernetes sustava se sastoji od puno čvorova (engl. nodes). Dva su glavna tipa čvorova:

1. Master čvor – sadrži Kubernetes upravljačku ploču koja upravlja cijelim sustavom.
2. Radni čvorovi – čvorovi na kojima se izvršava aplikacija koju „deploy-amo“.



Slika 3.2. Arhitektura Kubernetes sustava - glavni čvor i radni čvorovi [4]

Master čvor sastoji se od sljedećih komponenti:

- **API server** – korisnici (developeri, sistem administratori i drugi...) upravljaju Kubernetes klasterom komunicirajući s API serverom. U prethodnom poglavlju (Slika 3.1.) prikazano je da korisnik opis zahtjeva aplikacije šalje master čvoru. Komunikacija između korisnika i master čvora se odvija upravo preko API servera.
- **Kontroler** (engl. Controller manager) – kontrolira cjelokupni klaster. Zadužen je za replikaciju komponenti, praćenje radnih čvorova, poduzima radnje u slučaju prestanka rada pojedinih čvorova i slično.

- Raspoređivač (engl. Scheduler) – Zadužen je za raspoređivanje komponenti aplikacije na radne čvorove kako bi resursi bili što bolje iskorišteni i kako bi bili zadovoljeni svi specificirani zahtjevi.
- etcd – pouzdana baza podataka koja sprema konfiguraciju i stanje klastera.

Radni čvor sastoji se od sljedećih komponenti:

- Kubelet – komunicira s API serverom i upravlja komponentama radnog čvora na kojem se nalazi.
- Kube-Proxy – zadužen je za optimalni raspored opterećenja između radnih čvorova.
- Docker – zadužen za dohvaćanje „container image-a“ i pokretanje container-a.

3.3 Izvršavanje aplikacije na Kubernetes klasteru

Nakon što smo objasnili arhitekturu Kubernetes sustava, možemo detaljnije objasniti proces migriranja i izvršavanja aplikacije na Kubernetes klasteru nego što li je to objašnjeno u poglavlju 3.1.

Prvi korak migriranja aplikacije na Kubernetes kluster (u nastavku „deploy“) je „pakiranje“ aplikacije u jednu ili više „container“ slika (engl. container images). Te se slike potom objave na registru slika (engl. image registry). Zatim developer objavi opis aplikacije na Kubernetes API server (Slika 3.1.). U opisu aplikacije specificirano je koliko treba biti replika pojedine komponente, koje komponente se moraju nalaziti na istom radnom čvoru (ukoliko ima takvih komponenti), od koje container slike će nastati pojedina komponenta itd.

API server obradi zahtjev te posao preuzima Raspoređivač (engl. Scheduler). Raspoređivač raspoređuje containere na slobodne radne čvorove uzimajući u obzir opterećenje radnog čvora, zahtijevane resurse pojedinog containera i sl. Nakon što Raspoređivač rasporedi gdje će ići koji container, Kubelet radnog čvora na kojem je container smješten naredi Dockeru da preuzme container sliku s registra slika te pokrene container na osnovu preuzete slike. Na slici (Slika 3.2.) ostao je neobjašnjen termin „Pod“. Pod sadrži jedan ili više pokrenutih slika – containera. Pod će detaljno biti objašnjen u nadolazećim poglavljima.

Nakon što je aplikacija uspješno pokrenuta, Kubernetes sustav će i dalje kontinuirano nadgledati rad naše aplikacije vodeći računa da je broj svakog containera jednak broju kojeg je developer specificirao u opisu aplikacije. Ukoliko neki od containera prestane s radom Kubernetes će automatski restartirati problematični container. Slično tome, ukoliko neki od radnih čvorova prestane s radom, Kubernetes će odabrati drugi radni čvor pogodan za premještanje svih containera koji se nalaze na problematičnom radnom čvoru.

Svaka aplikacija ima periode kada posjećenost raste ili pada. Na već spomenutom primjeru Internet stranice krema za sunčanje, porast prometa će biti u ljetnim mjesecima, dok će u zimskom periodu promet biti vrlo mali. Koristeći Kubernetes sustav, ne moramo se brinuti da će dolaskom ljetnih mjeseci, naša aplikacija prestati s radom zbog velike posjećenosti. Isto tako, u zimskim mjesecima će Kubernetes sustav zaključiti kako imamo previše resursa nego što nam u tom trenutku treba te će automatski skalirati aplikaciju da ne trebamo plaćati dodatne, nepotrebne resurse. Na taj način ćemo uštedjeti novac, a isto tako osigurati da aplikacija radi i u periodima nagle velike posjećenosti. Kada potreba za pojedinim servisom pređe dozvoljenu specificiranu vrijednost, Kubernetes će automatski za nas napraviti još jednu kopiju kritičnog servisa.

3.4 Prednosti korištenja Kubernetes sustava

Osim već spomenutog automatskog skaliranja u prethodnom poglavlju, Kubernetes sustav donosi i niz drugih prednosti. Automatskim skaliranjem postiže se ušteda novca reduciranjem nepotrebnih resursa i konzistentni rad naše aplikacije čak i u periodima povećane posjećenosti. Kubernetes sustav vrši konstantno nadgledanje cjelokupnog klastera. Ukoliko neki od čvorova ili containera prestane s radom, automatski će se poduzeti potrebna radnja. Dok nije bilo Kubernetes sustava, sistem administratori su morali konstantno nadgledati opterećenje pojedinih komponenti aplikacije i sukladno tome vršiti radnje koje bi rezultirale novim brojem instanci pojedine komponente, premještanjem svih komponenti na drugi radni čvor itd. Ovo možda i ne zvuči toliko strašno dok vam u 2 sata ujutro ne zazvoni mobitel da aplikacija nije dostupna u srcu sezone i da ste izgubili nekoliko tisuća ili milijuna potencijalne zarade. Kubernetes sustav omogućava sistem administratorima da mirno spavaju noću tako što automatski zamjenjuje pokvarene instance novim instancama. Ujutro, kada se naspavaju, u okviru radnog vremena mogu proučiti zbog čega je nastao problem i raditi na otklanjanju problema.

Koristeći Kubernetes sustav posao migriranja aplikacije na server više ne treba raditi Ops tim. Iz razloga što su aplikacije zapakirane u containere (engl. containerized), posjeduju sve potrebne biblioteke i okolinu za njihovo izvršavanje. Nije potrebno instalirati ništa dodatno na servere. Sukladno tome, programeri mogu samostalno aplikaciju migrirati na klaster. Svi radni čvorovi predstavljeni su kao jedna cjelina te nije potrebno imati informaciju o karakteristikama servera za izvršavanje aplikacije. Radni čvorovi Kubernetes klastera ne moraju biti svi isti. Određen broj čvorova npr. može imati SSD diskove, dok drugi posjeduju HDD diskove. Ukoliko je potrebno, može se npr. specificirati zahtjev da se određena komponenta

izvršava samo na čvorovima koji posjeduju SSD disk. Kubernetes sustav će samostalno rasporediti tu komponentu na čvor koji posjeduje SSD disk bez da mu mi točno navedemo koji je to čvor. S ovim smo postigli da ne trebamo voditi brigu na koji će se čvor pojedina komponenta smjestiti sve dok su zadovoljeni zahtjevi koje smo naveli u opisu aplikacije. Bez Kubernetes sustava, sistem administrator bi samostalno morao odabrati radni čvor koji posjeduje SSD disk od svih mogućih čvorova sa SSD diskom i na njega smjestiti komponentu.

Ljudi, za razliku od računala, su vrlo loši u pronalaženju optimalnih kombinacija pogotovo kad je broj takvih kombinacija velik. Zbog toga možemo biti sigurni da će Kubernetes sustav bolje i optimalnije rasporediti komponente aplikacije na klaster nego što bi to napravili ljudi. Kod odabira svakog radnog čvora uzimaju se u obzir razni parametri kao što su opterećenost radnog čvora, ispravnost rada, zahtijevani resursi i sl.

4. PRVA APLIKACIJA NA KUBERNETES SUSTAVU

Prije nego što se objasne temeljni pojmovi Kubernetes sustava, svakako bi bilo dobro prvo na jednostavnom primjeru vidjeti kako izgleda proces izvršavanja aplikacije na Kubernetes-u.

Mnogi autori literatura o Kubernetes sustavu slično pristupaju objašnjavanju istog navodeći sličnost s vožnjom auta. Prvo naučite voziti auto, a tek potom krenete učiti što se nalazi ispod haube i kako promijeniti ulje.

U ovom poglavlju cilj će biti napraviti „deploy“ jednostavne NodeJs aplikacije na Kubernetes klaster. Kompletan kod aplikacije može se naći na Github linku:

<https://github.com/ikovac/kubernetes-hello-world>.

Aplikacija se sastoji od svega par linija koda:

```
const express = require('express');

const app = express();

app.use('/', (req, res) => {
  res.send('Hello world!');
});

app.listen(3000, () => {
  console.log('App is listening on port 3000');
});
```

Sve što aplikacija radi je kreira server koji sluša sav dolazni promet na portu 3000 i odgovara sa „Hello world!“.

U poglavlju 3 već smo opisali kako izgleda ovaj proces, stoga će biti mnogo sličnosti ali s dodanim praktičnim primjerima. Kako je već rečeno prije nego što započnemo sa samim migriranjem na server, moramo aplikaciju zapakirati u Docker container. Da bi to napravili moramo kreirati Docker sliku.

Unutar projekta kreirajmo datoteku naziva Dockerfile sa sljedećim naredbama:

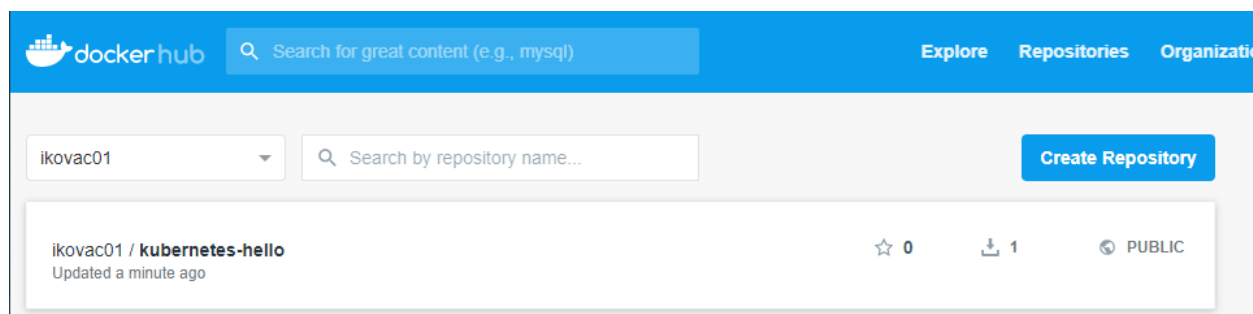
```
FROM node:12.17.0
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY ./ ./

EXPOSE 3000
CMD ["node", "index.js"]
```

S ovim kažemo Dockeru da kreira sliku na temelju Nodejs slike. Time ćemo imati mogućnost izvršavanja NodeJs aplikacija bez da samostalno instaliramo NodeJs. Potom odaberemo radni direktoriji i kopiramo package.json datoteku unutar containera te instaliramo sve biblioteke. Naredbom „EXPOSE 3000“ naznačimo koji PORT koristimo unutar Docker containera. Naredba ništa ne radi nego služi samo kao smjernica. Na posljetku pokrenemo NodeJs aplikaciju sa naredbom „node index.js“ kao što bi to napravili i da izvršavamo aplikaciju izvan Docker-a. Da bi kreirali Docker sliku na temelju datoteke Dockerfile moramo izvršiti sljedeću naredbu:

docker build -t image-name .

Za image-name odabrao sam ikovac01/kubernetes-hello, gdje ikovac01 označava ime mog korisničkog računa na Docker Hub-u, a kubernetes-hello je ime same slike. Sliku je potrebno potom objaviti na registar slika – Docker Hub. To se postiže komandom: *docker push ikovac01/kubernetes-hello*. Ukoliko vam objava slike nije uspjela, morate se prijaviti na Docker Hub račun sa naredbom *docker login*. Pri uspješnoj objavi, Docker slika se treba pojaviti na Docker Hub-u (Slika 4.1.).



Slika 4.1. Objava slike na Docker Hub

Prethodnim korakom našu aplikaciju smo zapakirali unutar Docker slike i to objavili na registar slika. Kubernetes će dohvatiti našu sliku i na temelju nje kreirati Docker container unutar kojeg će se izvršavati naša aplikacija.

Sljedeći korak je kreiranje Kubernetes klastera. Da bi mogli kreirati klaster prvo moramo imati račun na GKE. Iako se usluga plaća, dostupan je probni period od jedne godine koji je potpuno besplatan i bit će korišten u svrhu ovog diplomskog rada.

Kreiranje klastera je vrlo jednostavno. Stisnemo dugme „Create cluster“ i odaberemo ime našeg klastera (Slika 4.2.).

Free trial status: \$295.92 credit and 274 days remaining - with a full account, you'll get unlimited access to all of Google Cloud Platform.

Google Cloud Platform My First Project Search products and resources

Create a Kubernetes cluster ADD NODE POOL REMOVE NODE POOL

Cluster basics

Cluster basics

The new cluster will be created with the name, version, and in the location you specify here. After the cluster is created, name and location can't be changed.

Cluster set-up guides
MY FIRST CLUSTER

Name
hello-world

Location type
☒ Zonal
☐ Regional

Zone
us-central1-c

☐ Specify node locations

Master version

Choose Release Channel to get automatic GKE upgrades as new versions are ready. Choose a static version to upgrade manually in the future. [Learn more.](#)

☐ Release channel
☒ Static version

Static version
1.14.10-gke.36 (default)

Slika 4.2. Izrada klastera

Izrada klastera će potrajati neko vrijeme dok se ne kreiraju svi radni čvorovi koji posjeduju vlastite IP adrese. Klaster će se sastojati od 3 radna čvora, ukoliko niste promijenili standardne postavke pri kreiranju klastera. Nakon što se klaster kreira, možemo se iz komandne linije računala spojiti na novo-kreirani klaster. To se radi pritiskom na dugme „Connect“ i kopiranjem danog linka u komadnu liniju. Da bi se uvjerali kako klaster stvarno sadrži 3 radna čvora možemo izvršiti naredbu „*kubectl get nodes*“ (Slika 4.3.). Kao rezultat dobijemo listu 3 radna čvora s nazivom, statusom i ostalim specifikacijama.

```
PS D:\Documents\nodejs\kubernetes-hello-world> kubectl get nodes
NAME                                STATUS    ROLES    AGE    VERSION
gke-hello-world-default-pool-d60a8448-7lsj    Ready    <none>    19m    v1.14.10-gke.36
gke-hello-world-default-pool-d60a8448-cl1h    Ready    <none>    19m    v1.14.10-gke.36
gke-hello-world-default-pool-d60a8448-gxnb    Ready    <none>    19m    v1.14.10-gke.36
PS D:\Documents\nodejs\kubernetes-hello-world>
```

Slika 4.3. Lista radnih čvorova klastera

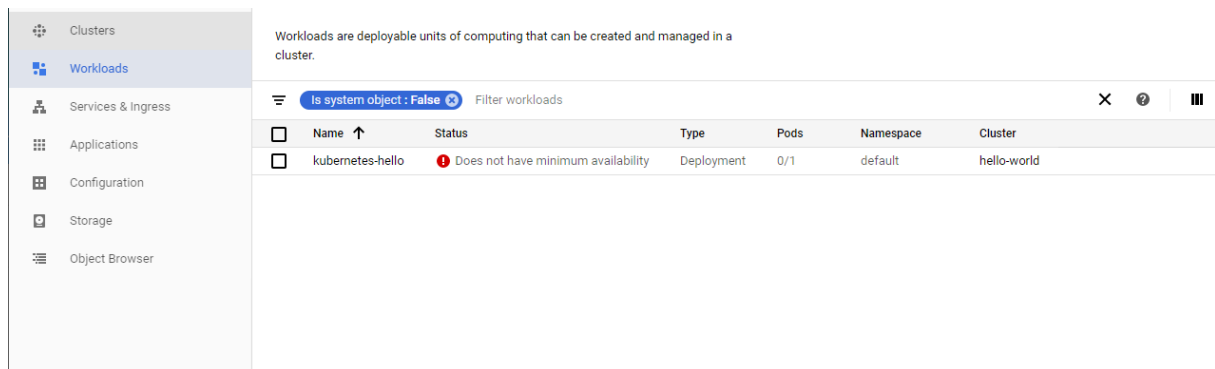
Sada konačno možemo objaviti našu aplikaciju na Kubernetes klaster. U prethodnim poglavljima rečeno je da developer mora objaviti opis aplikacije Kubernetes glavnom čvoru.

Opis aplikacije se piše u yaml datotekama. Za objaviti aplikaciju kreirat ćemo datoteku Deployment.yaml sa sljedećim kodom:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kubernetes-hello
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kubernetes-hello
  template:
    metadata:
      labels:
        app: kubernetes-hello
    spec:
      containers:
        - image: ikovac01/kubernetes-hello
          name: kubernetes-hello
```

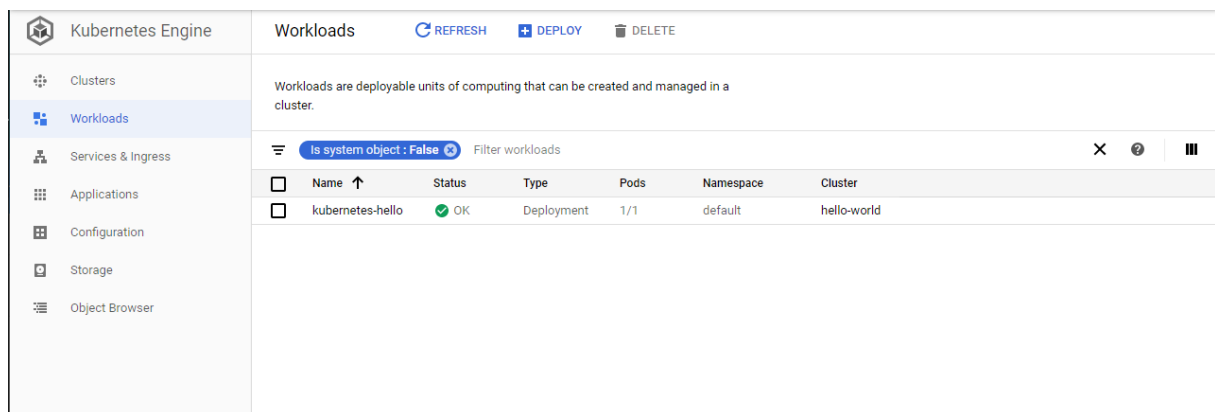
U opisu se nalazi specificirano na osnovu koje Docker slike želimo kreirati aplikaciju i koliko replika te aplikacije želimo. Iako sama datoteka može izgledati zbunjujuće to je doista sve što smo naveli. Kao vrstu opisa koju šaljemo glavnom čvoru stavili smo „Deployment“. Rečeno je da Kubernetes u slučaju prestanka rada neke komponente kreira novu komponentu. To upravo radi Deployment resurs. Ukoliko trenutni broj replika ne odgovara specificiranom broju (u ovom slučaju samo jedna replika) onda Deployment poduzme radnje kako bi se postigao specificirani broj replika. Iz razloga što se naša aplikacija može sastojati od velikog broja servisa sa selektorom smo odabrali koju komponentu želimo staviti u nadzor ovog Deployment-a. Odabrali smo selektor „kubernetes-hello“ jer je to oznaka (engl. label) koju smo dodijelili toj komponenti. Deployment tip resursa će biti objašnjen detaljnije u narednim poglavljima.

Objava na glavni čvor se vrši naredbom „*kubectl apply -f ./Deployment.yaml*“. Primjetit ćemo da se pojavila crvena oznaka jer jedna replika ove aplikacije još nije spremna (Slika 4.4.). Imamo 0/1 pokrenutih Pod-ova. Pod će također detaljnije biti objašnjen kasnije. Za sada to zamislimo kao kutiju u kojoj se izvršava jedan ili više containera.



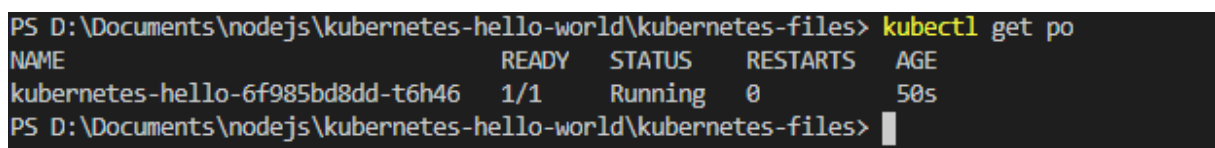
Slika 4.4. Replika je u procesu stvaranja

Nakon što Kubernetes dohvati objavljenu Docker sliku i kreira Docker container iz iste trebali bi dobiti zelenu oznaku koja govori kako je sve prošlo uspješno (Slika 4.5.).



Slika 4.5. Replika je kreirana

Ovo možemo provjeriti i ako dohvatimo sve Pod-ove na kreiranom klasteru naredbom „*kubectl get po*“ (Slika 4.6.) što je skraćeno od „*kubectl get pods*“.



Slika 4.6. Lista svih Pod-ova

Završetkom ovog koraka aplikacija je spreman i izvršava se na Kubernetes klasteru.

Kako pristupiti aplikaciji i uvjeriti se da stvarno radi?

Pri kreiranju Pod-a, svaki dobije vlastitu IP adresu. Ta IP adresa je interna adresa i nije joj moguće pristupiti iz vanjskog svijeta. Da bi pristupili aplikaciji moramo nekako izložiti pristup aplikaciji i izvan klastera. To se radi pomoću novog Kubernetes resursa nazvanog „Service“ koji će također biti detaljnije objašnjen kasnije.

Da bi kreirali Service, kreirajmo novu datoteku service.yaml sa sljedećim kodom:

```
apiVersion: v1
kind: Service
metadata:
  name: kubernetes-hello-service
spec:
  type: LoadBalancer
  ports:
  - port: 3000
    targetPort: 3000
  selector:
    app: kubernetes-hello
```

Service možemo nazvati kako hoćemo. U primjeru je nazvan kubernetes-hello-service. Sljedeće što moramo napraviti je specificirati tip servisa. LoadBalancer je tip servisa koji će biti dostupan izvan klastera jer će imati vanjsku IP adresu na koju ćemo moći pristupiti aplikaciji. Uz IP adresu moramo specificirati i na kojem portu želimo da aplikacija bude dostupna. Odabran je port 3000. Ovo može biti bilo koji drugi slobodni i dostupni port samo je bitno da kao „targetPort“ bude navedeno 3000 jer je to port na kojem aplikacija sluša dolazeće zahtjeve. Naposljetku, sa selektorom kažemo Kubernetes-u koju točno komponentu aplikacije želimo izložiti da bude javno dostupna. Naša aplikacija ima samo jednu komponentu kojoj smo dodijelili oznaku „kubernetes-hello“.

Nakon što objavimo servis na glavni čvor već poznatom naredbom „*kubectl apply -f ./service.yaml*“ u listi servisa moći ćemo vidjeti novo-kreirani servis (Slika 4.7.).

Kubernetes Engine

Clusters

Workloads

Services & Ingress

Applications

Configuration

Storage

Object Browser

Services & Ingress

REFRESH

CREATE INGRESS

DELETE

SERVICES

INGRESS

Services are sets of Pods with a network endpoint that can be used for discovery and load balancing. Ingresses are collections of rules for routing external HTTP(S) traffic to Services.

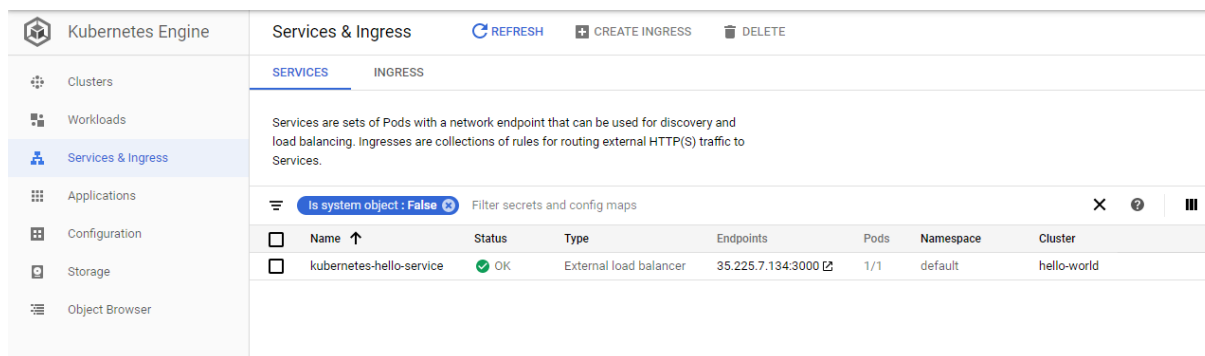
Is system object: False

Filter secrets and config maps

<input type="checkbox"/>	Name	Status	Type	Endpoints	Pods	Namespace	Cluster
<input type="checkbox"/>	kubernetes-hello-service	Creating Service Endpoints	External load balancer		1/1	default	hello-world

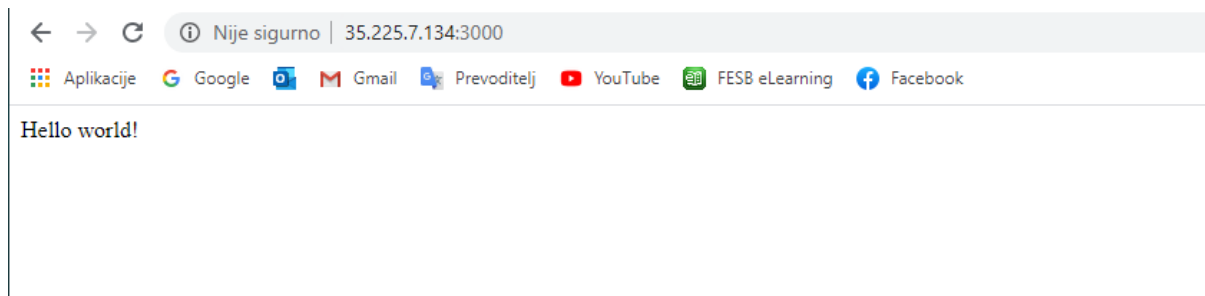
Slika 4.7. Servis je kreiran, ali još nije spreman

Aplikacija još neće biti dostupna jer proces dodjele javne IP adrese traje neko vrijeme. Nakon što se dodijeli javna IP adresa možemo pristupiti servisu (Slika 4.8.).



Slika 4.8. Javna IP adresa je dodijeljena servisu

Ovim korakom aplikacija je u potpunosti spremna i možemo joj pristupiti. Ukoliko odemo na dobivenu vanjsku IP adresu na port 3000 vidjet ćemo „Hello world!“ kao rezultat (Slika 4.9.).



Slika 4.9. Aplikacija je dostupna izvan klastera

Bilo tko u svijetu može pristupiti aplikaciji na dobivenoj IP adresi. Aplikacija je napokon objavljena i radi.

Iako ovaj proces možda u početku izgleda kompliciran, u biti je vrlo jednostavan. Ukoliko napravimo sažetak, sve što smo napravili je zapakirali aplikaciju u Docker sliku i tu sliku objavili na registar slika. Potom smo glavnom čvoru poslali opis aplikacije sa navedenim imenom Docker slike i koliko replika želimo i na kraju smo kreirali servis da naša aplikacija bude dostupna i izvan klastera. Aplikacija je sada u rukama Kubernetes sustava koji će biti zadužen da uvijek imamo željeni broj replika i u slučaju prestanka rada kreirati nove replike. Moguće je i naknadno mijenjati broj replika. Samo uredite datoteku tako što promijenite broj iz 1 u novi željeni broj replika i spremite promjene. Kubernetes sustav će potom vidjeti da željeni broj replika nije jednak postojećem stanju te će kreirati nove replike.

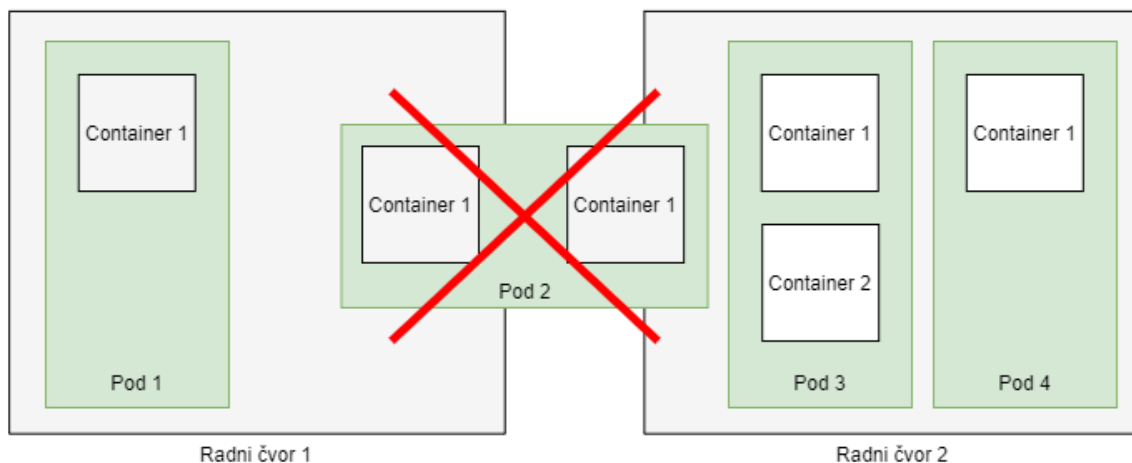
5. TEMELJNI POJMOVI KUBERNETES SUSTAVA

U prethodnom poglavlju spomenute su neke osnovne Kubernetes komponente koje još nisu detaljno objašnjene. U ovom poglavlju detaljnije ćemo objasniti najvažnije Kubernetes komponente koje je nužno poznavati ukoliko želimo da se naša aplikacija uspješno izvršava na Kubernetes sustavu. Započet ćemo sa komponentom koja se naziva Pod.

5.1 Pods

Container je Docker slika koja se izvršava. Predstavlja proces tj. program u izvođenju. Zamisao containera je da izoliraju jedan zaseban proces. Iako se unutar containera može nalaziti više procesa, to nije preporučljivo. Svaki container bi trebao sadržavati najviše jedan proces (osim ukoliko proces ne pokreće vlastite pod-procese). Ukoliko se unutar containera nagomila više procesa, odgovornost programera je da osigura ponovno pokretanje procesa u slučaju grešaka.

Iz razloga što ne bismo smjeli grupirati više procesa unutar jednog containera, mora postojati neka komponenta koja će osigurati viši nivo konstrukcije. Upravo ta komponenta se naziva Pod. Pod-ovi omogućavaju povezivanje različitih containera u jedinstvenu cjelinu. Pod je najmanja migracijska jedinica (engl. deploy unit) Kubernetes sustava. Umjesto da se containeri samostalno migriraju, migracija se uvijek radi na Pod-u koji sadrži containere. Preporučljivo je da svaki Pod ima samo jedan container, iako postoje iznimke kada je potrebno da Pod sadržava više containera. Naknadno će biti objašnjeno kada je to slučaj. Ukoliko Pod sadrži više containera pravilo je da se svi containeri uvijek izvršavaju na istom radnom čvoru (Slika 2.1.).



Slika 5.1. Svi containeri Pod-a moraju se izvršavati na istom radnom čvoru

Pod omogućava da se srodni procesi pokreću zajedno pružajući istu okolinu kao i da se pokreću skupa unutar istog containera, uz prednost što su procesi izolirani jer se svaki izvršava unutar vlastitog containera.

Svaki Pod posjeduje vlastitu IP adresu koju dijele svi containeri unutar Pod-a. Containeri unutar Pod-a mogu komunicirati preko localhost-a. Također, svi containeri dijele i isti port prostor. Posljedično tome, moramo paziti da containeri unutar Pod-a ne dijele isti port broj. Ovo se samo odnosi na containere koji se nalaze u istom Pod-u. Containeri u različitim Pod-ovima mogu bez problema dijeliti isti port broj.

Komunikacija između Pod-ova je također moguća. Čak i u slučajevima kada se Pod-ovi ne nalaze na istom radnom čvoru oni i dalje mogu komunicirati. Razlog tome je upravo što svaki Pod posjeduje vlastitu IP adresu koja je dohvatljiva bilo gdje u klasteru.

Pod-ove zamislimo kao zasebne mašine gdje se u svakoj mašini nalazi zasebna komponenta aplikacije. Pod-ovi su vrlo učinkoviti s obzirom na hardverske resurse. Zbog toga se ne moramo misliti da li nešto odvojiti u različite Pod-ove ili smjestiti sve u jedan Pod. Na primjer, ukoliko imamo aplikaciju koja se sastoji od front-end i back-end komponente, uvijek bi trebali te dvije komponente smjestiti u različite Pod-ove. Iako nas ništa ne sprječava da sve smjestimo u isti Pod, to svakako nije poželjno i preporučljivo. Razlog više je i skaliranje. Razne komponente aplikacije će vrlo vjerojatno imati različite potrebe za skaliranjem. Zamislimo situaciju da od ukupnog prometa samo 10% zahtjeva ide na back-end. Vrlo vjerojatno ćemo imati potrebu za većim brojem front-end Pod-ova nego back-end Pod-ova. Da se ove dvije komponente nalaze unutar istog Pod-a, ne bismo ih mogli skalirati odvojeno

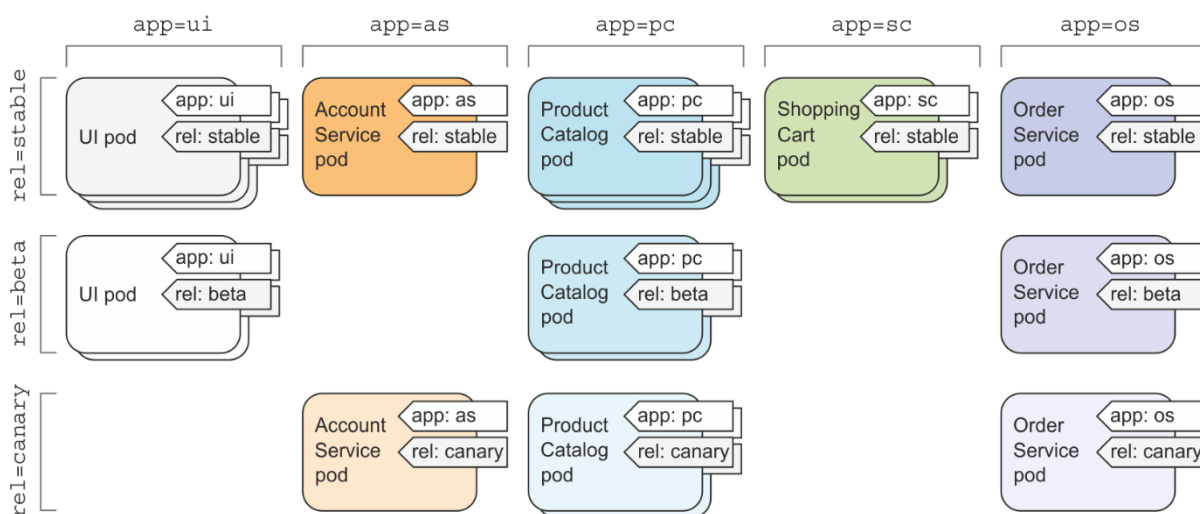
nego kao cjelinu. Slično je i da imamo web server i bazu podataka. Ove dvije komponente aplikacije također ima smisla odvojiti u različite Pod-ove.

Zbog svega navedenog, vidimo da ima smisla odvajati različite komponente aplikacije u različite Pod-ove, nekada ima smisla staviti različite komponente u isti Pod. Komponente bi trebalo staviti u isti Pod samo i samo ako su usko povezane. Na primjer, ukoliko imamo primarni container i sekundarni container koji zajedno rade istu zadaću. Zamislimo da imamo web server koji poslužuje sadržaj iz određenog direktorija i sekundarni proces koji sa interneta dohvaća resurse i sprema ih u folder iz kojeg web server poslužuje sadržaj.

Na ovom primjeru dva containera dijele isti datotečni sustav. Sekundarni container sprema u datoteku iz koje primarni container čita. Samo u ovakvim i sličnim situacijama trebalo bi različite containere staviti u isti Pod.

5.1.1 Oznake Pod-ova

Kod mikroservis arhitektura uobičajeno je da imamo veliki broj različitih servisa. To rezultira velikim brojem Pod-ova. Kako bi s Pod-ovima bilo lakše raditi koristimo oznake (engl. labels) s kojima Pod-ove grupiramo u različite kategorije (Slika 5.2.). Oznaka se sastoji od para ključ-vrijednost (engl. key-value) koja se pridodaje ne samo Pod-ovima nego i ostalim resursima Kubernetes sustava. Na primjer, radnom čvoru možemo dodijeliti oznaku disk=SSD ukoliko radni čvor posjeduje SSD disk. Ova tehnika predstavlja vrlo moćan alat Kubernetes sustava jer znatno olakšava upravljanje resursima. Kod migracije na klaster, ukoliko želimo da se neki Pod izvršava na radnom čvoru sa SSD diskom, samo ćemo dodati tu oznaku u selektor.



Slika 5.2. Organizacija Pod-ova pomoću oznaka [3]

Dodavanjem oznaka kao na slici (Slika 5.2.) postizemo organizaciju Pod-ova u dvije dimenzije. App se odnosi na vrstu komponente aplikacije. Na primjer, korisničko sučelje će za ključ app imati vrijednost ui, kupovna košarica će imati svoju oznaku sc itd. Svaka komponenta može imati i više različitih verzija. U procesu izrade softvera svaka komponenta prolazi kroz više ciklusa. Kada programer isprogramira komponentu, komponenta najprije mora proći testiranje prije nego što se stavi u produkciju. Oznakama možemo označiti i je li komponenta pripada produkcijskoj, testnoj ili nekoj drugoj verziji. Ukoliko želimo korisnicima prikazati samo produkcijsku verziju aplikacije, prilikom kreiranja Pod-a kao selektor ćemo staviti samo produkcijsku oznaku verzije. Oznake se mogu dodavati i mijenjati kod već postojećih Pod-ova. U nadolazećim poglavljima vidjet ćemo zašto je ovo važno.

5.1.2 Objava Pod opisa kroz yaml datoteku

Kao što smo već vidjeli, s Kubernetes glavnim čvorom komuniciramo šaljući zahtjeve u obliku yaml datoteka. Yaml datoteke sadrže opis resursa kojeg želimo kreirati na Kubernetes klaster. Ukoliko želimo kreirati Pod resurs, trebamo poslati yaml datoteku glavnom čvoru koja kao tip resursa sadrži „Pod“.

Jednostavan primjer pod.yaml datoteke koja kreira Pod je sljedeći:

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-hello
spec:
  nodeSelector:
    gpu: "true"
  containers:
  - image: ikovac01/kubernetes-hello
    name: kubernetes-hello
```

Ova datoteka će kreirati resurs tipa Pod unutar kojeg će se pokretati container koji nastaje iz Docker slike ikovac01/kubernetes-hello. Slika je javno dostupna na Docker registru [slika](https://hub.docker.com/r/ikovac01/kubernetes-hello). Oznakom gpu: „true“ smo specificirali da želimo odabrati radni čvor koji posjeduje grafičku karticu. Ovim primjerom vidimo koliko je jednostavno pomoću oznaka (engl. labels) selektirati željene resurse.

Datoteku šaljemo glavnom čvoru već viđenom naredbom: „*kubectl apply -f pod.yaml*“.

Ukoliko je sve prošlo u redu, Pod će biti kreiran na klasteru. Naredbom „*kubectl get pods*“ se mogu ispisati svi Pod-ovi na klasteru i vidjeti je li novo-kreirani Pod na listi.

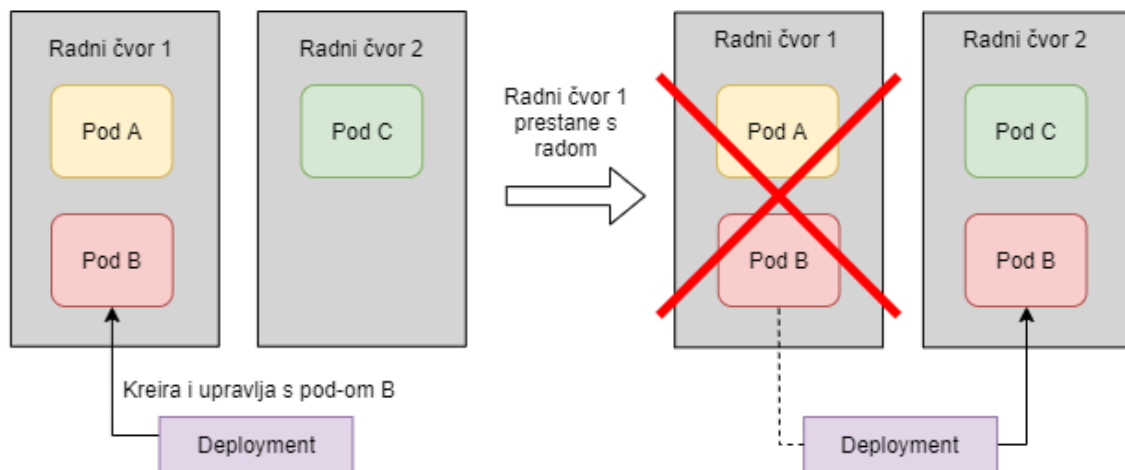
5.2 Deployments

U prethodnom poglavlju je pokazano kako se može kreirati Pod resurs na Kubernetes klasteru. U praksi se vjerojatno ovaj proces nikada neće samostalno raditi. Kada Pod resurs kreiramo na spomenuti način, Kubernetes odabere radni čvor kojem će dodijeliti novo-stvoreni Pod. Ukoliko Pod prestane s radom ili se iz nekog razloga izbriše iz klastera neće biti stvoren novi Pod. Na primjer, ukoliko radni čvor na kojem se nalazi Pod prestane s radom, Pod koji je na ovaj način stvoren i nalazi se na ovom radnom čvoru je izgubljen i neće biti ponovno kreiran od strane Kubernetes-a.

Ukoliko želimo da Kubernetes sustav vodi računa o automatskom ponovnom pokretanju neispravnih Pod-ova moramo kreirati resurs Deployment koji će potom kreirati potreban broj Pod-ova. Deployment smo već kreirali u poglavlju 4.

Zadaća Deployment resursa je prvo da kreira potreban broj Pod-ova u skladu sa specifikacijom, a potom da prati status tih Pod-ova. Ukoliko container unutar Pod-a prestane s radom, Pod će ponovno pokrenuti neispravni container. Slično tome, ukoliko Pod prestane s radom (npr. zbog memory leak-a) zadaća Deployment resursa je da uvidi problematičnost Pod-a i kreira novi Pod kojim će zamijeniti problematični Pod. Ovim načinom iskorišteni su svi potencijali Kubernetes sustava i u pravilu ne bih trebalo nikad samostalno kreirati Pod-ove nego taj posao prepustiti Deployment resursu.

Slika 5.3. opisuje razliku između Pod-a koji je stvoren ručno i onoga koji je stvoren od strane Deployment-a. Primjećujemo da prilikom prestanka rada radnog čvora 1, Deployment uoči da trenutno ne postoji dovoljan broj Pod-a B koji je u nadležnosti tog Deployment-a te stvori novi Pod B na nekom od ispravnih radnih čvorova. Pod A koji se nalazio na radnom čvoru 1 će biti izgubljen i neće biti stvoren novi isti takav Pod. Razlog tome je što je Pod A ručno stvoren i nije u nadležnosti niti jednog Deployment resursa.



Slika 5.3. Prilikom prestanka rada radnog čvora, Deployment resurs kreira novi Pod

5.2.1 Kontrole ispravnosti Pod-a

Dvije su glavne tehnike kojima se provjerava ispravnost Pod-ova. Nazivaju se „probe života i spremnosti (engl. liveness i readiness probes).

Liveness probe zadužena je za praćenje ispravnosti Pod-a tokom njegovog rada. Provjera ispravnosti Pod-a se odvija periodički u postavljenim vremenskim intervalima. Za svaki container unutar Pod-a može biti postavljena zasebna liveness proba. Više je načina na koji se liveness proba može izvršiti a najpoznatiji je onaj putem HTTP GET zahtjeva. Ova proba uključuje slanje HTTP GET zahtjeva na URL koji se sastoji od IP adrese containera, porta containera i putanje koje specificiramo u yaml opisu. Ukoliko kao odgovor dobijemo statusni kod 2xx ili 3xx, proba se smatra uspješnom. U protivnom, proba se smatra neuspješnom i kreće postupak stvaranja novog containera koji će zamijeniti neispravan container. Naravno da se novi container ne stvara odmah nakon prve neuspjele probe, nego se treba dogoditi određen broj neuspjelih probi i tek onda kreće spomenuti postupak.

Primjer liveness probe prikazan je sljedećim yaml opisom:

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-hello
spec:
  containers:
  - image: ikovac01/kubernetes-hello
    name: kubernetes-hello
    livenessProbe:
      httpGet:
```

```
path: /  
port: 3000
```

Ovim opisom kreiramo novi Pod koji nastaje iz slike `ikovac01/kubernetes-hello`. Jedino što se razlikuje od prijašnje kreacije Pod-a je dodana liveness proba pod „`livenessProbe`“ sekcijom. Liveness proba će poslati GET zahtjev na putanju „`path: /`“ i na port 3000. Dakle GET zahtjev će se u konačnici poslati na url: „`http://[CONTAINER_IP]:3000/`“. Nakon što kreiramo Pod, ukoliko dođe do bilo kakve pogreške servera i odgovor na liveness probu ne bude uspješan Kubernetes će automatski pokrenuti stvaranje novog containera kojim će zamijeniti postojeći problematični container.

Osim spomenutih specifikacija, za liveness probu se mogu konfigurirati i period slanja zahtjeva, vrijeme odgode (vrijeme koje prođe od stvaranja Pod-a do prve liveness probe), vrijeme u kojem container mora dati odgovor na probu, koliko najviše puta proba smije biti neuspješna prije kreiranje novog containera itd.

Za razliku liveness probi koje su zadužene za provjeru ispravnosti Pod-a tokom rada Pod-a, postoje i readiness probe. Readiness probe su zadužene za provjeru je li container unutar Pod-a ispravno započeo sa svojim radom.

Readiness probe sprječavaju slanje zahtjeva containeru koji još nije spreman za rad. Na primjer, ukoliko imamo jedan Pod web server komponente i želimo dodati još jedan Pod. Novo-stvorenom Pod-u neće biti prosljeđivani zahtjevi sve dok container unutar Pod-a uspješno ne odgovori na readiness probu. U slučaju da ne postoji readiness proba, promet bi bio ravnomjerno raspoređen između dva Pod-a te bi novo-stvoreni Pod odgovorio sa server pogreškom ili bi isteklo vrijeme u kojem je potrebno dobiti odgovor servera. Tek nakon što bi container unutar novo-stvorenog Pod započeo s radom, sve bi uspješno radilo.

Liveness i Readiness probe je uvijek dobro staviti u yaml opis jer predstavljaju vrlo koristan i moćan alat Kubernetes sustava.

5.2.2 Objava Deployment opisa i demonstracija automatske zamjene neispravnih Pod-ova
Deployment resurs ćemo objaviti na isti način kao i u poglavlju 4. Objavit ćemo `deployment.yml` datoteku s naredbom „`kubectl apply -f ./deployment.yml`“.

Sadržaj `deployment.yml` datoteke je sljedeći:

```
apiVersion: apps/v1
```

```

kind: Deployment
metadata:
  name: kubernetes-hello
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubernetes-hello
  template:
    metadata:
      labels:
        app: kubernetes-hello
    spec:
      containers:
      - image: ikovac01/kubernetes-hello
        name: kubernetes-hello

```

Sadržaj je u potpunosti isti kao i u poglavlju 4 osim što je broj replika povećaj na 3.

Ukoliko ispišemo sve Pod-ove na klasteru naredbom: „*kubectl get po*“ dobit ćemo ispis kao na slici ispod (Slika 5.4.).

```

PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files> kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
kubernetes-hello-6f985bd8dd-4f67c   1/1     Running   0           48s
kubernetes-hello-6f985bd8dd-fplw2   1/1     Running   0           48s
kubernetes-hello-6f985bd8dd-nkg6t   1/1     Running   0           48s
PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files>

```

Slika 5.4. Ispis Pod-ova na klasteru

Demonstracije radi, ručno ćemo izbrisati jedan Pod kako bi se uvjerali da Deployment resurs automatski kreira novi Pod kako bi specificirani broj Pod-ova (u ovom primjeru je 3) bio jednak stvarnom broju Pod-ova.

Pod ćemo izbrisati komandom „*kubectl delete pod [POD_NAME]*“ kao što je prikazano na slici ispod (Slika 5.5.).

```

PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files> kubectl delete pod kubernetes-hello-6f985bd8dd-4f67c
pod "kubernetes-hello-6f985bd8dd-4f67c" deleted
PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files> kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
kubernetes-hello-6f985bd8dd-fplw2   1/1     Running   0           119s
kubernetes-hello-6f985bd8dd-nkg6t   1/1     Running   0           119s
kubernetes-hello-6f985bd8dd-wpvtk   1/1     Running   0           37s
PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files>

```

Slika 5.5. Ispis Pod-ova nakon brisanja jednog Pod-a

Za očekivati je bilo da ćemo nakon brisanja Pod-a imati samo 2 Pod-a. To se nije dogodilo zbog već spomenutog Deployment resursa koji je odmah kreirao novi Pod. Proces kreiranja novog poda se odvio toliko brzo da nismo uspjeli niti ispisat stanje na klasteru kada smo imali samo 2 Pod-a.

5.3 Services

Kao što smo već vidjeli u poglavlju 4 Service resurs nam omogućava pristup skupu Pod-ova za koje je taj Service zadužen. U ovom dijelu ćemo detaljnije objasniti zbog čega uopće postoji potreba za Service resursima.

U mikro-servis arhitekturi postoje servisi koji moraju međusobno komunicirati kako bi zajedno izvršili neki zadatak. Ukoliko ne govorimo o Kubernetes sustavu, da bi ostvarili komunikaciji između dvaju servisa, sistem administrator bi vjerojatno trebao migrirati taj servis na neki server i javno dobivenu IP adresu podijeliti s ostalim servisima koji žele komunicirati s tim servisom. Međutim u Kubernetes sustavu ovo ne bi bilo funkcionalno. Sljedeći su razlozi zašto je to tako:

- U svakom trenutku novi Pod-ovi se mogu stvarati i uklanjati zbog promjene broja instanci Pod-a, zbog prestanka rada određenog Pod-a, zbog premještanja Pod-a na neki drugi radni čvor i slično.
- IP adresa se dodjeljuje Pod-u nakon što je smješten na odgovarajući radni čvor i prije nego što je sami Pod kreiran. Posljedično tome, klijent ne može unaprijed znati IP adresu koju će novo-stvoreni Pod imati te nema nikoju informaciju o tome kako pristupiti Pod-u.
- Horizontalnim skaliranjem se povećava broj Pod-ova od kojih svaki ima svoju vlastitu i jedinstvenu IP adresu. Klijent se ne bi trebao zamarati koliko je tih Pod-ova i koje su im IP adrese. Umjesto da klijenti prate listu svih Pod-ova koji pružaju isti servis i traže njihove IP adrese lakše bi bilo da tom servisu mogu pristupiti preko jedinstvene IP adrese.

Service resurs upravo rješava sve ove probleme. Kreiranjem Service resursa omogućuje se pristup određenom Pod-u ili određenoj grupi Pod-ova koja predstavlja neku komponentu aplikacije. Service resurs prilikom kreiranja dobije jedinstvenu IP adresu koja se više ne mijenja te je Service-u moguće pristupiti neovisno o tome jesu li u međuvremenu stvoreni ili uklonjeni neki Pod-ovi. Dakle Service-i pružaju jedinstvenu IP adresu i port koji se nikada ne mijenjaju dok Service postoji. Klijent može poslati zahtjev na IP adresu i port Service-a koji

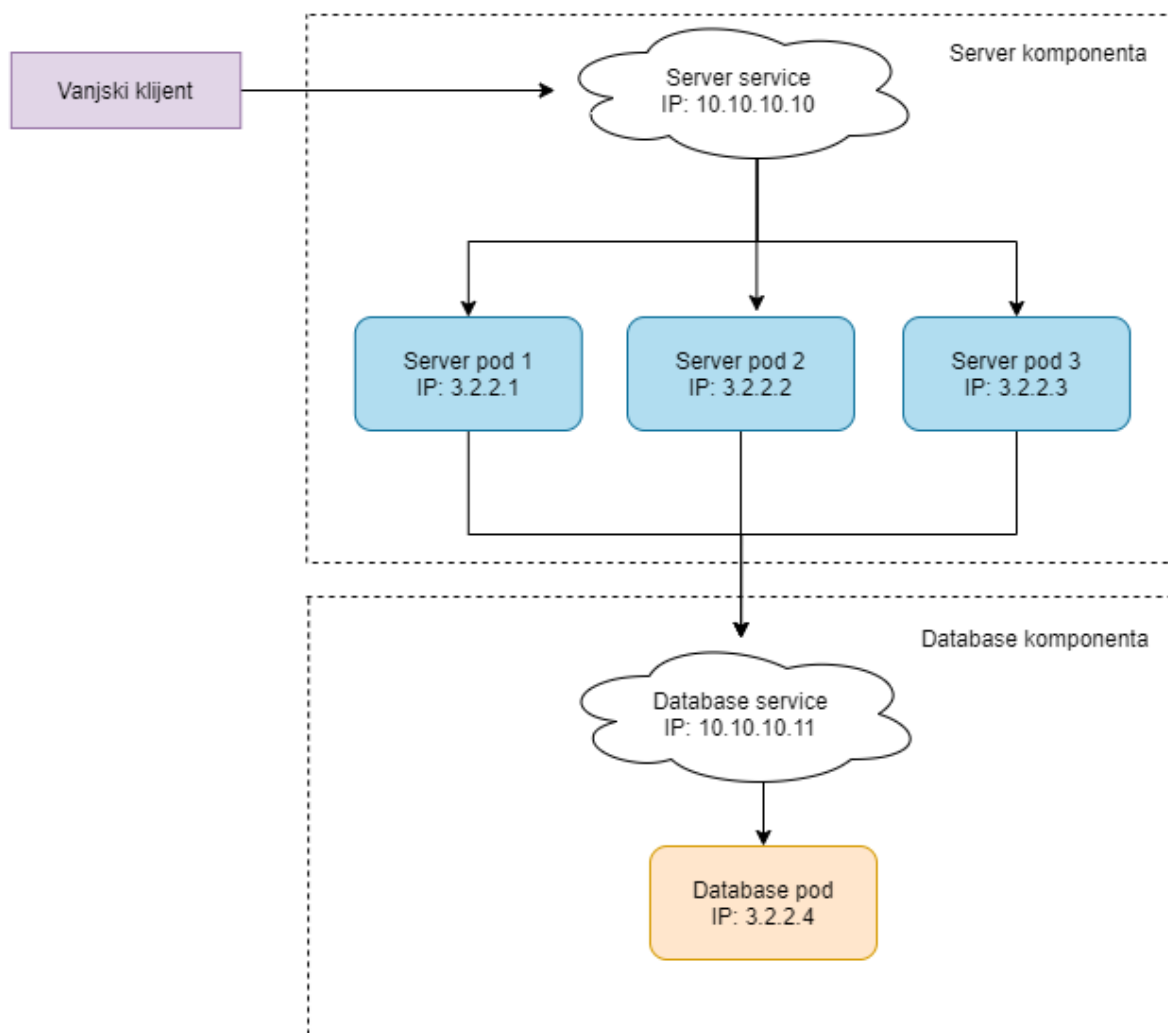
će se potom proslijediti na neki od Pod-ova koji spadaju pod taj Service. Može biti više raznih Service-a na klasteru. Obično za svaku komponentu aplikacije (svaki mikro-servis) postoji jedan Service resurs preko kojeg se pristupa toj komponenti.

Na ovaj način prilikom pristupa komponenti, nije potrebno poznavati lokaciju pojedinog Pod-a koji se mogu slobodno premještati na radne čvorove klastera.

Service može biti i interni i eksterni. To znači da ukoliko je Service interni, može biti dohvatljiv samo unutar klastera dok eksterni Service može biti dohvatljiv i izvan klastera.

Na primjeru dva tipa servisa ćemo pokazati ulogu Service resursa. Neka imamo prvu komponentu koja predstavlja web server i drugu komponentu koja predstavlja bazu podataka. Web server prilikom klijentskog zahtjeva radi upit na bazu podataka i klijentu vraća odgovor. Vanjski klijent mora imati mogućnost spajanja samo na web server. Također klijent se ne bi trebao zamarati detaljima da li postoji samo jedan ili više instanci tog web servera. Klijentu je bitno samo da poznaje jedinstvenu IP adresu i port na koji može poslati svoj zahtjev da dobije željeni odgovor.

Komponenta baze podataka mora također biti dohvatljiva ali ne i izvan klastera. Web server će morati komunicirati sa komponentom baze podataka da bi klijentu dao željeni odgovor. Web server mora poznavati jedinstvenu IP adresu Service-a baze podataka. Slika ispod (Slika 5.6.) pokazuje ovaj primjer. Ukoliko dođe do potrebe horizontalnog skaliranja web servera i dodaju se novi web server Pod-ovi, IP adresa web server Service-a će ostati nepromijenjena te će klijent i dalje pristupati web serveru na isti način kao što je i prije pristupao.



Slika 5.6. Vanjski i unutrašnji klijenti Pod-ovima pristupaju preko Service resursa

Unutar Kubernetes klastera nije potrebno raditi s IP adresama Service-a. Kubernetes posjeduje lokalni DNS server koji nam omogućava da pojedinim servisima pristupamo na nama lakši i jednostavni način. Ljudi su vrlo loši u pamćenju brojeva kao što je IP adresa i mnogo bolje pamte riječi. Kubernetes nam omogućava da unutar klastera servisima pristupamo preko njihovog imena. Tako ćemo u gornjem primjeru servisu baze podataka moći pristupiti i preko imena servisa umjesto IP adrese 10.10.10.11. Ovo ćemo detaljnije vidjeti u narednim poglavljima na primjeru Chat aplikacije koja će biti obrađena u sklopu ovog diplomskog rada.

Ukoliko naročito nije specificirano, Service će nasumično proslijediti zahtjev nekom od Pod-ova. Može se specificirati da Service prosljeđuje zahtjev uvijek istom Pod-u na koji je prosljedio prvi zahtjev tog klijenta. To se radi na temelju IP adrese klijenta. Ako je odabran ovaj način, nakon što se nasumično odabere Pod kod prvog zahtjeva, svi budući zahtjevi ovog

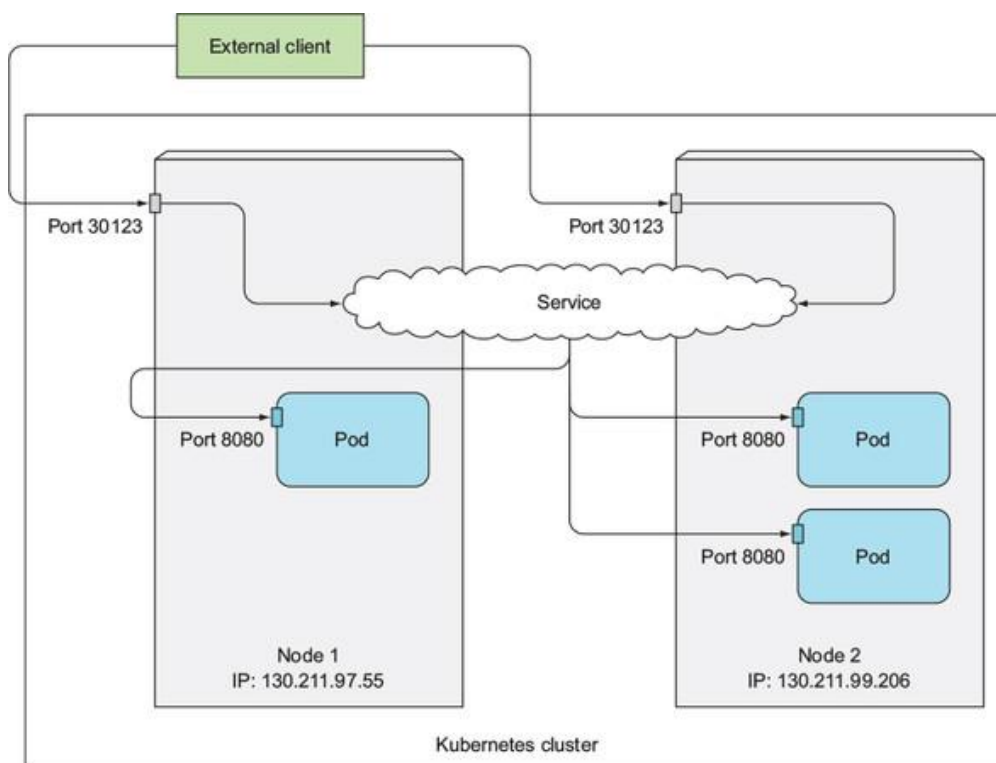
klijenta će se također slati na isti Pod. Service resursi ne rade na HTTP razini te će biti moguće prosljeđivati zahtjeve samo na temelju IP adrese a ne recimo i informacije u kolačićima (engl. cookies). Da bi ovo postigli morat ćemo koristiti neke druge tipove Service-a kao Ingress servis koji će biti objašnjen naknadno.

5.3.1 Vrste servisa

Više je vrsta Service resursa:

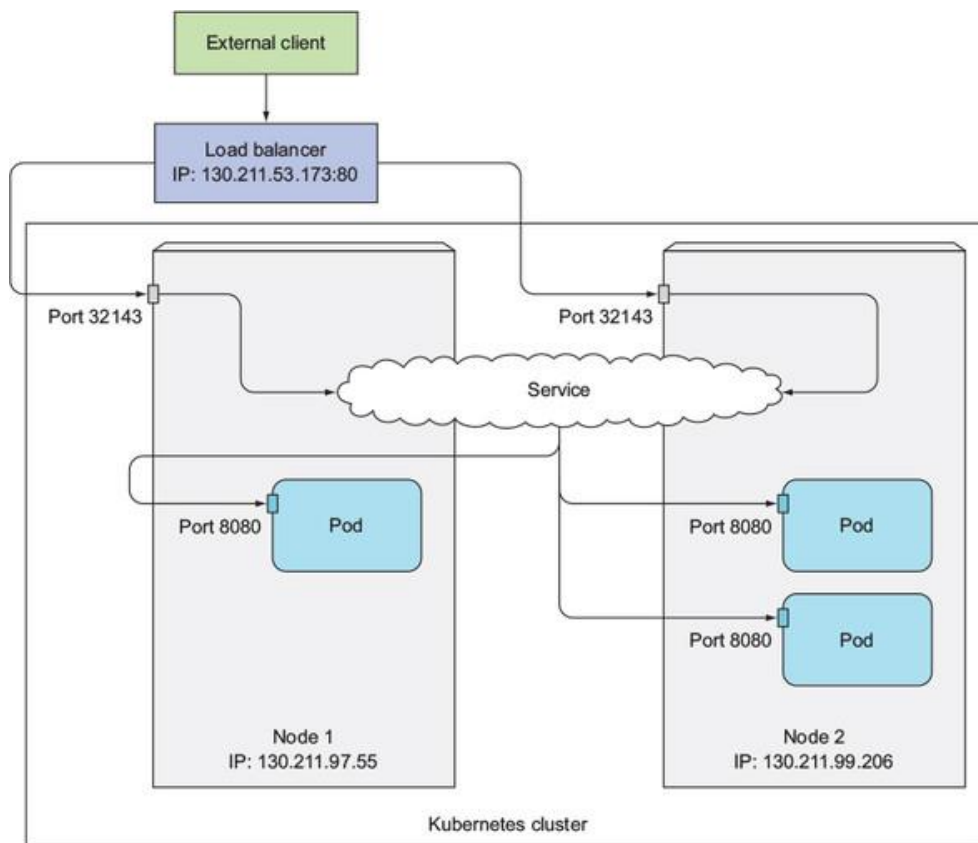
1. ClusterIP - ovakav tip servisa je dostupan samo unutar klastera. Ovo je zadani tip servisa te će servis biti ovog tipa ukoliko nikoji drugi tip servisa nije specificiran.
2. NodePort - na svakom radnom čvoru izlaže broj porta servisa. Servisu će se moći pristupiti na <NodeIp>:<NodePort> te će servis biti eksterni, dakle moći će mu se pristupiti i izvan klastera.
3. LoadBalancer - ovakav tip servisa je također dostupan i izvan klastera. LoadBalancer dobije jedinstvenu IP adresu preko koje klijent pristupa servisu. Zahtjev se potom prosljeđuje radnim čvorovima na specificiranom portu. Ovaj tip servisa je u stvari NodePort tip servisa ali je na neki način odvojen od same klaster arhitekture.
4. Ingress – mehanizam koji omogućava izlaganje više servisa preko jedinstvene IP adrese. Radi na HTTP razini koja omogućava još više konfiguracijskih opcija nego standardni tip Service resursa.

NodePort je najlakše razumijeti iz slike (Slika 5.7.). Klijent se spaja na prikazani servis preko IP adrese nekog od radnih čvorova te porta radnog čvora koji je namijenjen tom servisu. U ovom slučaju klijent može specificirati IP adresu radnog čvora 1 ili radnog čvora 2. Iako je za zaključiti da će se zahtjev proslijediti nekom od Pod-ova na radnom čvoru kojeg klijent specificira, to nije tako. Nakon što klijent navede IP adresu i port na radnom čvoru, zahtjevi će se ravnomjerno raspoređivati između Pod-ova. Nije važno koju IP adresu radnog čvora klijent odabere, sve dok je odabrani čvor ispravan. Ukoliko se dogodi da radni čvor kojeg je klijent odabrao prestane s radom, neće biti moguće slanje zahtjeva radnom čvoru.



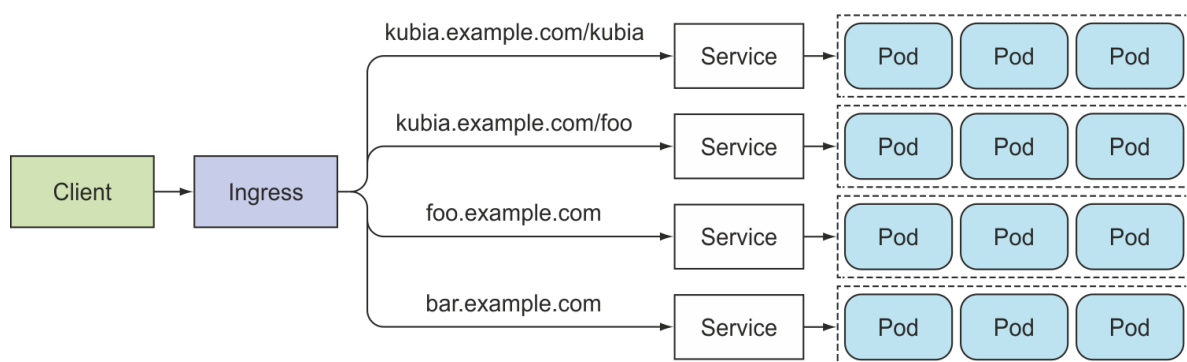
Slika 5.7. Eksterni klijent se povezuje s NodePort servisom preko IP adrese radnog čvora 1 ili preko IP adrese radnog čvora 2 [3]

LoadBalancer tip servisa je odvojen od klaster arhitekture (Slika 5.8.). Klijent pristupa servisu tako što navede IP adresu LoadBalancer-a koji potom zahtjev prosljeđuje na neki od dostupnih radnih čvorova. LoadBalancer tehnika kao podlogu koristi NodePort tehniku, ali i rješava problem NodePort tehnike kada neki od radnih čvorova prestane s radom. Tada smo rekli da klijent neće moći pristupiti servisu ukoliko koristi IP adresu radnog čvora koji je prestao s radom. LoadBalancer to rješava tako da postoji viši sloj koji je odvojen od NodePort tehnike. Taj viši sloj prosljeđuje zahtjev na onaj čvor koji je u tom trenutku ispravan ili ukoliko su svi ispravni onda ravnomjerno raspoređuje zahtjeve između radnih čvorova.



Slika 5.8. Eksterni klijent pristupa LoadBalancer servisu [3]

Još je jedan tip servisa, a to je **Ingress** servis. Ingress servis radi na HTTP razini te znatno proširuje mogućnosti Service resursa. Ingress resurs može raditi preusmjeravanje prometa ne samo na osnovu IP adrese već i na osnovu imena domene i na osnovu putanje (engl. path) (Slika 5.9.).



Slika 5.9. Više servisa može biti izloženo kroz jedan jedini Ingress [3]

Korištenjem LoadBalancer tipa, moramo za svaki servis osigurati vlastitu javnu IP adresu. Ingress zahtjeva samo jednu IP adresu. To je moguće iz razloga što Ingress radi na već spomenutoj HTTP razini pa je moguće raditi usmjeravanje i na osnovu host polja te na osnovu putanje. S ovim je moguće postići da imamo na primjer jednu domenu: „http://example.com“

te ukoliko je putanja '/' zahtjev preusmjeriti na servis koji izvršava klijentsku aplikaciju, a ako je putanja 'api' (<http://example.com/api>) zahtjev preusmjeriti na API web server.

Osim usmjeravanja prometa na osnovu putanje, Ingress posjeduje još i mnogo drugih mogućnosti. Jedna od njih je svakako postavljanje TLS prometa. Umjesto da programeri unutar same aplikacije postavljaju TLS certifikate, to se jednostavno postiže u postavkama Ingress-a. Promet između klijenta i Ingress kontrolera je TLS enkriptiran, dok nije enkriptiran promet između Ingress-a i servisa koji se svakako nalazi unutar klastera. Koristeći TLS na razini Ingress-a smo osigurali da programeri ne trebaju mijenjati aplikaciju postavljajući TLS certifikate. To nije dobro iz više razloga a jedan od njih je da mijenjanje TLS certifikate (npr. zbog isteka istog) uzrokuje rekompajliranje cijele aplikacije.

5.4 Volumes

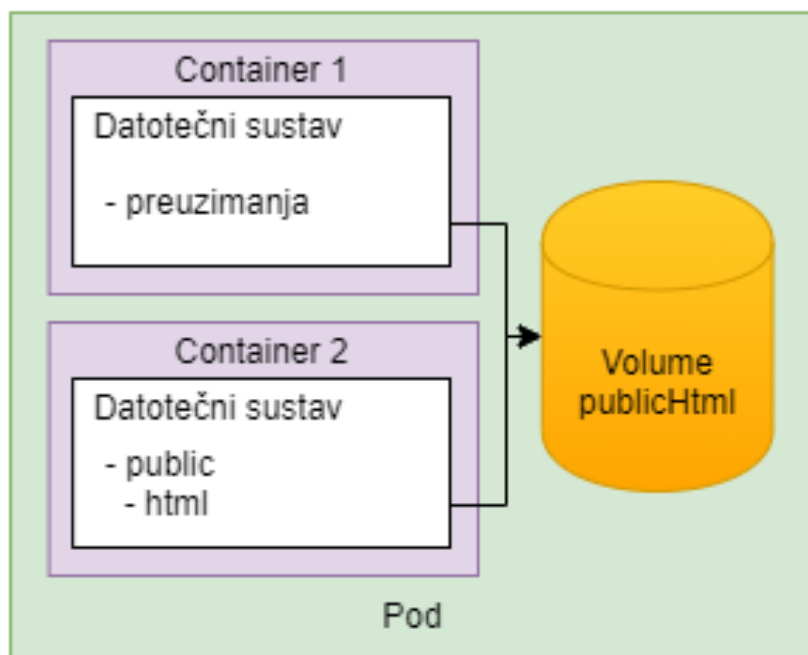
Procesi unutar Pod-a dijele iste resurse kao što su CPU, RAM, mrežna sučelja i druge. Međutim, procesi ne dijele isti disk prostor. Već smo rekli da svaki container u Pod-u posjeduje vlastiti izolirani datotečni sustav iz razloga što datotečni sustav nastaje na osnovu Docker slike. Prilikom kreiranja novih containera iz iste Docker slike, svi će imati isti početni datotečni sustav. Container unutar Pod-a se može ponovno pokrenuti uslijed prestanka rada ili ukoliko liveness probe otkrije neispravnost u radu containera. Prilikom ponovnog pokretanja containera cijeli datotečni sustav u tom trenutku će se izgubiti. Novi container će vidjeti samo one datoteke koje nastaju iz same Docker slike, a ne i one koje je sam proces unutar containera izgenerirao (ukoliko postoje).

Kubernetes sustav omogućava rješavanjem ovog problema koristeći Volumes resurs.

Volumes resurs se ne kreira zasebno kao što je bio slučaj sa Pod, Deployment i Service resursima već se specificiraju u opisu Pod-a i imaju isti životni vijek kao Pod-ovi. To znači da će prilikom kreiranja Pod-a biti kreiran i Volume resurs. Isto tako ukoliko se Pod uništi biti će uništen i Volume resurs. Prilikom ponovnog pokretanja containera zbog neispravnost, novo-stvoreni container će vidjeti sve datoteke spremljene od strane prethodnog containera jer se nalaze u istom Pod-u. Volume se stvara na razini Pod-a i zajednički je za sve containere unutar Pod-a. Da bi containeri mogli koristiti Volume Pod-a mora se u opisu točno specificirati koji containeri imaju pristup Volume resursu.

U poglavlju 5.1 rečeno je da jedino kod srodnih procesa ima smisla u isti Pod stavljati više containera. Naveden je primjer sa web serverom kao primarnim procesom i sekundarnim procesom koji dohvaća sadržaj koji će web server posluživati. Da bi ovo bilo moguće,

primarni i sekundarni proces moraju dijeliti isti datotečni sustav. Ovo se postiže korištenjem Volume resursa (Slika 5.10.). Na slici možemo vidjeti da unutar istog Pod-a postoje dva containera. Container 1 zaslužan je za dohvaćanje sadržaja koji će Container 2 posluživati. Container 1 dohvaćeni sadržaj spremi u vlastiti datotečni sustav u mapu „preuzimanja“ koja je povezana sa zajedničkim Volume resursom. Container 2 poslužuje sadržaj iz vlastite mape public/html koja je spojena na istu mapu kao i mapa „preuzimanja“ containera 1. Na ovaj način container 2 će vidjeti sve datoteke koje container 1 preuzme.



Slika 5.10. Korištenje istog disk prostora između dva različita containera unutar Pod-a

Primjer yaml opisa ovakvog Pod-a bi bio:

```
apiVersion: v1
kind: Pod
metadata:
  name: content-generator
spec:
  containers:
  - image: html-generator
    name: html-generator
    volumeMounts:
    - name: html
      mountPath: /preuzimanja
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    - name: html
```

```
    mountPath: /usr/share/nginx/html
    readOnly: true
  ports:
  - containerPort: 80
  volumes:
  - name: html
    emptyDir: {}
```

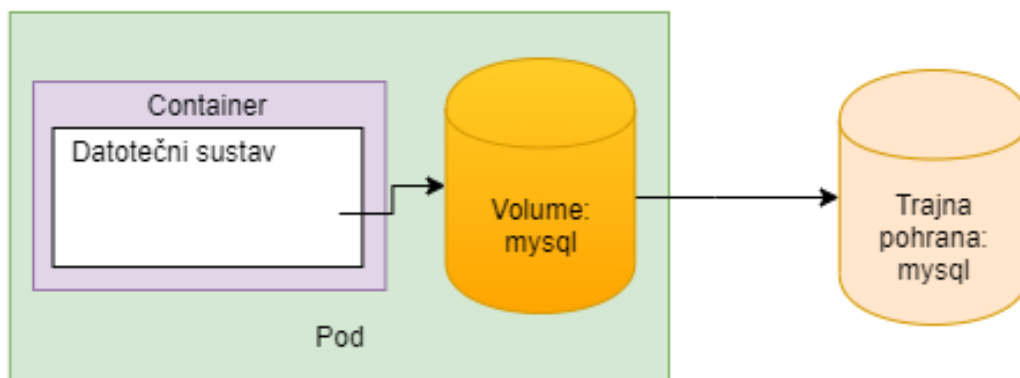
Docker slika „html-generator“ je izmišljena za potrebe ovog primjera i vjerojatno ne postoji ali samo ime bi trebalo otkriti što ovaj container radi. Ovim opisom kreirali smo Pod s dva containera – html-generator container i web-server container. Na dnu datoteke kreiran je prazni Volume naziva html. Taj smo Volume specificirali da koristimo u oba containera. Sa ključnom riječi „volumeMounts“ smo povezali datotečni sustav unutar containera sa Volume datotečnim sustavom koji je zajednički za oba containera. Postigli smo da sve što html-generator kreira u direktoriju /preuzimanja bude dostupno web-server containeru u direktoriju /usr/share/nginx/html.

5.4.1 Trajno skladište

Životni vijem Volume resursa koji smo kreirali u prethodnom primjeru isto je kao i životni vijek Pod-a u kojem se nalazi. To znači da ćemo prilikom ponovnog kreiranja Pod-a ili premještanja Pod-a na drugi radni čvor izgubiti sve dotadašnje podatke iz datotečnog sustava. Na primjer, ukoliko se radi o Pod-u koji izvršava bazu podataka, prestankom rada tog Pod-a svi podaci iz baze će biti trajno izbrisani. Ovo je pogotovo nepraktično u procesu testiranja kada se učestalije stvaraju i uklanjaju Pod-ovi.

Kubernetes sustav ima rješenje i za ovaj problem korištenjem trajne pohrane. Korištenjem Google-ovog Kubernetes sustava imamo mogućnost kreiranja trajnog disk prostora željene veličine.

Korištenje trajne pohrane je vrlo slično kao i korištenje običnog Volume resursa. U opisu Pod yaml datoteke moramo specificirati kojem containeru dozvoljavamo pristup trajnoj pohrani (koju smo prethodno ručno kreirali na GKE) i koji direktoriji povezujemo. Prilikom kreiranja novog Pod-a, Volume unutar Pod-a neće biti prazan, nego će se napuniti sadržajem trajne pohrane. Isto tako, kada container unutar Pod-a napravi izmjenu unutar svog datotečnog sustava, ta će se promjena odraziti i na trajnu pohranu (Slika 5.11.). Ovim načinom postigli smo da će sav prethodni sadržaj baze podataka biti dostupan i nakon ponovnog kreiranja Pod-a.

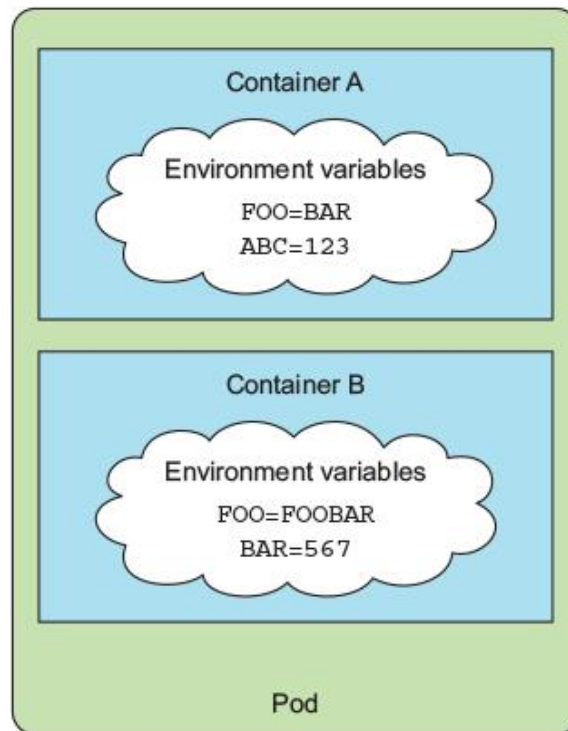


Slika 5.11. Logika korištenja trajne pohrane

5.5 ConfigMaps i Secrets

Skoro sve aplikacije koje se koriste u produkciji zahtijevaju određene konfiguracijske parametre. Ukoliko se radi o bazi podataka, konfiguracijske parametri će vjerojatno biti ime baze, ime korisnika, lozinka korisnika itd. Za web server parametri mogu biti port, TSL certifikat itd. Konfiguracijske parametre nikako ne bi smjeli „zapakirati“ unutar same aplikacije. Samo ime „konfiguracijski“ naglašava da se radi o parametrima koji se mogu mijenjati ovisno o različitim verzijama aplikacije, okolini, da li se radi o produkcijskoj ili testnoj aplikaciji itd. Iako se konfiguracijski parametri mogu „zapakirati“ u samu Docker sliku prilikom kreiranja slike, ovo svakako nije preporučljivo i Kubernetes sustav pruža bolji način korištenja istih. Razlog više je i taj što je velika većina Docker slika javno objavljena na Docker registar slika te će svi moći vidjeti konfiguracijske parametre što svakako nije poželjno ukoliko se radi o osjetljivim informacijama kao što su korisničko ime i lozinka.

Kubernetes sustav nam omogućava da svakom containeru unutar Pod-a dodijelimo vlastite konfiguracijske parametre tj. okolinske varijable (engl. environment variables) (Slika 5.12.).



Slika 5.12. Svaki container može imati svoje parametre [3]

Postoji više načina kako je ovo moguće napraviti, a u sklopu ovog diplomskog bit će objašnjen samo najpreporučljiviji način.

Prvi korak je kreiranje ConfigMap resursa specificirajući yaml opisnu datoteku.

```
apiVersion: v1
data:
  PORT: 3000
kind: ConfigMap
metadata:
  name: my-config
```

Uočavamo da je opis ConfigMap resursa prilično jednostavan. Najvažniji dio datoteke je polje „data“ koje sadrži okolinske varijable. Tu se mogu specificirati sve potrebne varijable, a u ovom primjeru dana je samo jedna varijabla „port“ sa vrijednošću 3000. Datoteku je nakon kreiranja potrebno objaviti na Kubernetes glavni čvor sa već dobro poznatom naredbom: „*kubectl apply -f ./ime-datoteke.yaml*“.

Da bi koristili okolinsku varijablu PORT iz kreiranog ConfigMap resursa u opisu containera moramo dodati sljedeći dio koda:

```
spec:
  containers:
  - image: some-image
```

```
envFrom:  
- prefix: CONFIG_  
  configMapRef:  
    name: my-config
```

Novi dio koda je samo ovaj dio od „envFrom“. Na taj način kažemo containeru odakle da uzme okolinske varijable. Budući da smo ConfigMap resurs nazvali „my-config“ pod „configMapRef“ sekcijom smo stavili isto to ime. Polje „prefix“ je opcionalno i koristeći ovo polje sve okolinske varijable iz ConfigMap resursa će dobiti prefiks CONFIG_. Sada ćemo moći koristiti varijablu PORT unutar containera tako da joj pristupimo preko imena CONFIG_PORT. Ukoliko se radi o NodeJs aplikaciji, okolinskoj varijabli pristupamo na način: process.env.CONFIG_PORT.

Korištenjem ConfigMap resursa možemo mijenjati vrijednosti okolinskim varijabli bez potrebe za ponovnim pokretanjem aplikacije tj. svih containera i Pod-ova. Na ovaj način znatno je olakšan i ubrzan proces testiranja i samog razvoja aplikacije, a isto tako okolinske varijable su zaštićene od neovlaštenog pristupa.

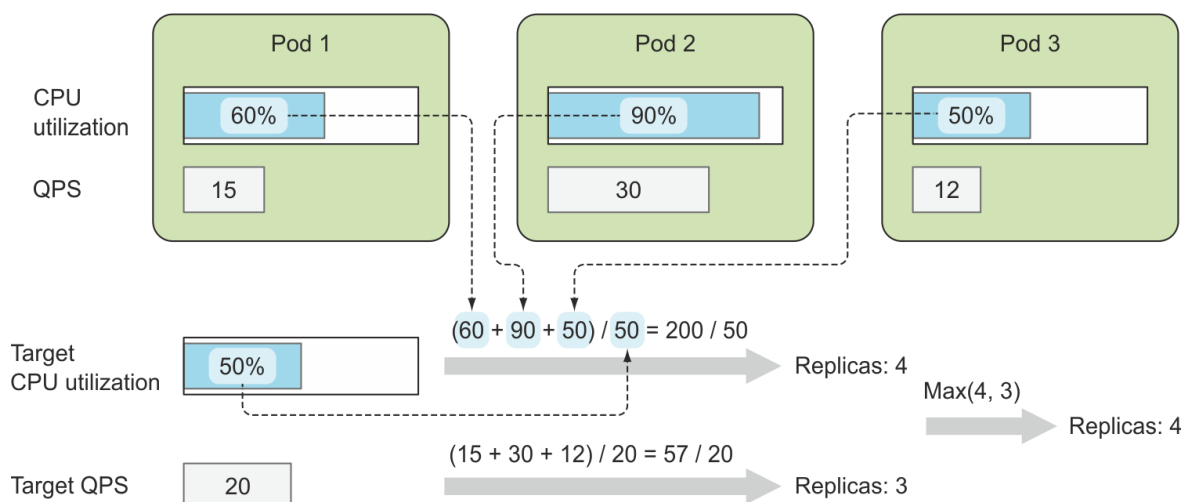
Ukoliko je potrebno veća doza sigurnosti okolinskih varijabli postoji resurs koji se naziva Secrets. Secret resurs je vrlo sličan ConfigMap resursu te se kreira i koristi na potpuno identičan način s dodatnim naglaskom na sigurnost. Tako se ovaj resurs kreira na radnom čvoru Pod-a koji ga koristi i nikad se ne zapisuje na disk nego se izvršava u radnoj memoriji. Također, Secret resurs je za razliku od ConfigMap resursa enkriptiran.

6. AUTOMATSKO SKALIRANJE

Najjednostavniji način skaliranja Pod-ova u Kubernetes sustavu podrazumijeva ručno mijenjanje broja replika u yaml datoteci. Međutim, u praksi nije baš zgodno skaliranje raditi ručno. Razlog tome je što bi unaprijed trebali znati kada će biti nagle promjene u opterećenju i sukladno tome smanjiti ili povećati broj Pod-ova. Napomene radi, osim Pod-ova mogu se skalirati i radni čvorovi, ali to neće biti obrađeno u sklopu ovog diplomskog rada iz razloga što koristimo GKE gdje je za besplatno verziju broj čvorova ograničen.

Ručno skaliranje ne može riješiti nagle i nepredvidljive skokove u promjeni opterećenja. Kubernetes sustav omogućava automatsko skaliranje na osnovu postavljenih metrika kao što su korištenje CPU-a, memorije, diska itd. Kod automatskog skaliranja Kubernetes sustav samostalno povećava i smanjuje broj replika Pod-ova.

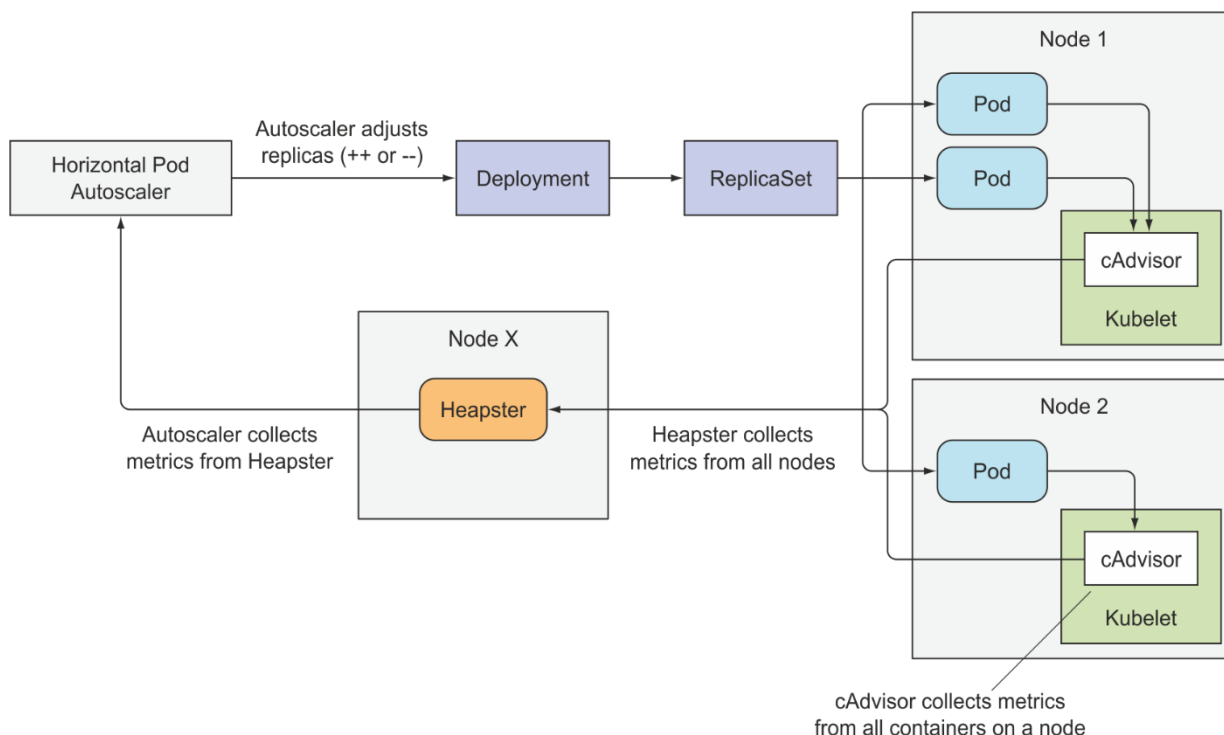
Slika 6.1. pokazuje proces određivanja broja potrebnih Pod-ova. Na primjer, uzmimo dvije metrike za praćenje opterećenja Pod-ova – CPU i QPS. Cilj nam je da CPU opterećenje bude na 50% i da QPS bude na 20. Broj potrebnih Pod-ova ćemo dobiti iz formule prikazane na slici gdje se dijeli zbroj opterećenja svih Pod-ova i željeno opterećenje. Postupak se ponovi za sve metrike koje se prate i uzme se najveći dobiveni broj od svih metrika. U konkretnom primjeru, dobit ćemo da je broj Pod-ova potrebno povećati na 4 jer 3 Pod-a nisu dovoljna da bi se zadovoljilo željeno opterećenje.



Slika 6.1. Računanje broja potrebnih Pod-ova na osnovu dvije metrike [3]

Računanje trenutnog opterećenja događa se u komponenti Kubernetes sustava koja se naziva cAdvisor. cAdvisor je pokrenut na svim radnim čvorovima i periodički prikuplja informacije o opterećenju Pod-ova. Svi prikupljeni podatci šalju se komponenti Heapster koja provodi

opisane računske operacije kako bi se utvrdilo ima li potrebe za skaliranjem. Rezultati se potom šalju HPA komponenti (engl. Horizontal Pod Autoscaler) koja provodi skaliranje (Slika 6.2.).



Slika 6.2. Proces automatskog skaliranja [3]

Bitno je naglasiti kako se skaliranje neće dogoditi trenutno nego će trebati neko vrijeme. To se vrijeme razlikuje i kod povećanja i smanjivanja broja replika Pod-a. Vrijeme potrebno za povećanje replika Pod-a će biti znatno manje nego vrijeme potrebno za smanjivanje broja Pod-ova. Razlog tome je što je važnije što prije odgovoriti na povećanje opterećenja novim replikama kako se ne bi odbacio ni jedan zahtjev. Smanjivanje broja Pod-ova nije toliko hitan postupak i događa se tek kada je HPA komponenta potpuno sigurno da neće doći do ponovnog naglog porasta opterećenja. Kada se zaključi da su trenutni resursi nepotrebni, smanji se broj Pod-ova.

Spomenuta vremena se mogu konfigurirati ukoliko je potrebno, ali Kubernetes sustav to već jako dobro obavlja. Također se prati i broj prethodnih skaliranja te se na temelju svih prikupljenih podataka donose naprednije odluke o skaliranju nego što je to objašnjeno.

6.1.1 Primjer automatskog skaliranja

U poglavlju 4 demonstriran je proces migriranja aplikacije na Kubernetes sustav. Primjer ćemo proširiti s auto-skaliranjem.

Koristit ćemo istu kubernetes-hello Docker sliku jedino ćemo napraviti malu promjenu u

Deployment resursu.

Novi Deployment resurs će sadržavati sljedeći kod:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kubernetes-hello
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubernetes-hello
  template:
    metadata:
      labels:
        app: kubernetes-hello
    spec:
      containers:
      - image: ikovac01/kubernetes-hello
        name: kubernetes-hello
        resources:
          requests:
            cpu: 100m
```

Jedino što je novo je „resource“ sekcija na kraju datoteke. Primjer automatskog skaliranja bit će demonstriran na CPU metrici. Da bi to bilo moguće, moramo specificirati koliko svaki Pod zahtjeva CPU resursa. 100m se odnosi na 100 milicores procesora. Milicores su posebna metrika Kubernetes sustava gdje je svaka procesorska jezgra predstavljena kao 1000 milicore-a. Dakle ukoliko imamo procesor s 4 jezgre ukupno ćemo imati 4000m. U navedenom primjeru koristimo 100m što je jednako korištenju 10% jedne jezgre procesora.

Nakon što objavimo sve potrebne resurse kao i u poglavlju 4 možemo započeti sa auto-skaliranjem. HPA resurs zadužen za automatsko skaliranje najjednostavnije je kreirati sljedećom naredbom: „*kubectl autoscale deployment kubernetes-hello --cpu-percent=30 --min=1 --max=5*“, gdje „kubernetes-hello“ predstavlja naziv Deployment resursa. Na ovaj način smo rekli Kubernetes sustavu da želimo Deployment resurs automatski skalirati na minimalno 1 Pod, a maksimalno na 5 Pod-ova ciljajući na željeno CPU opterećenje od 30%. Opterećenje od 30% se ne odnosi na CPU opterećenje radnog čvora na kojem se nalazi Pod. Odnosi se na 30% od traženog CPU resursa Pod-a, tj. od 100m CPU-a traženog od strane Pod-a skaliranje će se dogoditi ukoliko opterećenje na Pod-u pređe 30m što je 30 posto od 100m.

Naredbom „*kubectl get hpa*“ možemo provjeriti da je HPA resurs uistinu kreiran (Slika 6.3.). Resursu će trebati neko vrijeme da prepozna Deployment resurs i započne s normalnim radom.

```
PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files> kubectl get hpa
NAME                REFERENCE                TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
kubernetes-hello    Deployment/kubernetes-hello  <unknown>/30%    1         5         0         12s
PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files>
```

Slika 6.3. HPA resurs je kreiran, ali nije još spreman

Nakon što započne s radom u TARGETS dijelu će se prikazati trenutno opterećenje i željeno opterećenje (Slika 6.4.). Trenutno opterećenje je 0% jer još ne pristupamo našoj aplikaciji. Za očekivati je da će se broj replika smanjiti na 1 jer opterećenje ispod željenog. Međutim, to se neće dogoditi u isti tren zbog prethodno opisanog razloga. HPA će otpustiti resurse tek kad bude siguran da su resursi uistinu višak.

```
PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files> kubectl get hpa
NAME                REFERENCE                TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
kubernetes-hello    Deployment/kubernetes-hello  0%/30%    1         5         3         77s
PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files>
```

Slika 6.4. HPA resurs je započeo s radom

Jednog kada HPA bude siguran da resursi više ne trebaju smanjit će broj Pod-ova na samo jedan Pod (Slika 6.5.).

```
PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files> kubectl get hpa
NAME                REFERENCE                TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
kubernetes-hello    Deployment/kubernetes-hello  0%/30%    1         5         1         7m27s
PS D:\Documents\nodejs\kubernetes-hello-world\kubernetes-files>
```

Slika 6.5. Automatsko skaliranje na 1 Pod zbog viška resursa.

Na ovaj način smo postigli da se uslijed automatskog skaliranja smanjuje broj trenutnih Pod-ova na klasteru. Još se nismo uvjerali da radi i obrnuti postupak, tj. povećanje broja Pod-ova pri porastu opterećenja. Da bismo to postigli moramo povećati opterećenje pristupanjem aplikaciji.

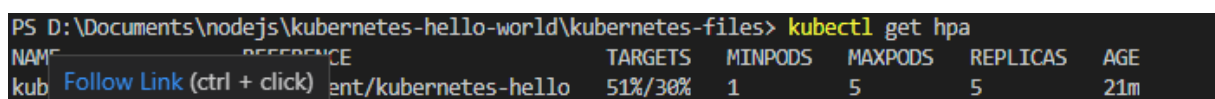
Aplikaciji možemo pristupiti putem pretraživača na [http://\[IP-SERVISA\]:3000](http://[IP-SERVISA]:3000) ali to neće biti dovoljno jer aplikacija može podnijeti to opterećenje i sa samo jednim Pod-om.

Koristit ćemo „busybox“ Docker sliku unutar koje ćemo kreirati petlju koja će slati zahtjeve našoj aplikaciji. To ćemo postići naredbom:

```
kubectl run -it --rm --restart=Never loadgenerator --image=busybox -- sh -c "while true; do
wget -O - -q http://kubernetes-hello-service.default:3000; done".
```

Na ovaj način kreirat ćemo Pod unutar kojeg će se izvršavati busybox Docker container. Unutar containera izvršavat će se navedena while petlja koja će slati zahtjeve na `http://kubernetes-hello-service.default:3000`. Već smo rekli da Kubernetes sustav omogućava komponentama unutar klastera da međusobno komuniciraju pružajući vlastiti DNS. Da bi pristupili servisu unutar klastera koristimo `[ime-servisa].[namespace]:[PORT]`, gdje je ime servisa `kubernetes-hello-service`, ime namespace-a je `default` i port je `3000`.

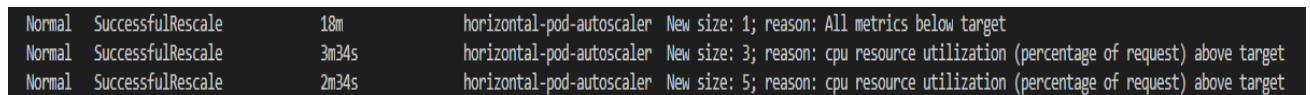
Nakon nekoliko sekundi broj replika će se početi povećavati (Slika 6.6.).



NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
kub	Follow Link (ctrl + click) ent/kubernetes-hello	51%/30%	1	5	5	21m

Slika 6.6. Broj replika Pod-a se povećao na 5

S naredbom „`kubectl describe hpa`“ možemo dobiti detaljnije informacije zašto je došlo do promjene broja Pod-ova. Na dnu ispisa trebali bi dobiti sličan ispis kao na slici ispod (Slika 6.7.). Vidimo da se na početku broj replika s 3 smanjio na 1 zbog toga što je opterećenje bilo premalo. Kasnije, prilikom porasta opterećenja, broj replika se postepeno počeo povećavati. Prvo se povećao na 3 replike a potom i na 5 replika.



Normal	SuccessfulRescale	18m	horizontal-pod-autoscaler	New size: 1; reason: All metrics below target
Normal	SuccessfulRescale	3m34s	horizontal-pod-autoscaler	New size: 3; reason: cpu resource utilization (percentage of request) above target
Normal	SuccessfulRescale	2m34s	horizontal-pod-autoscaler	New size: 5; reason: cpu resource utilization (percentage of request) above target

Slika 6.7. Detaljniji opis uzroka skaliranja

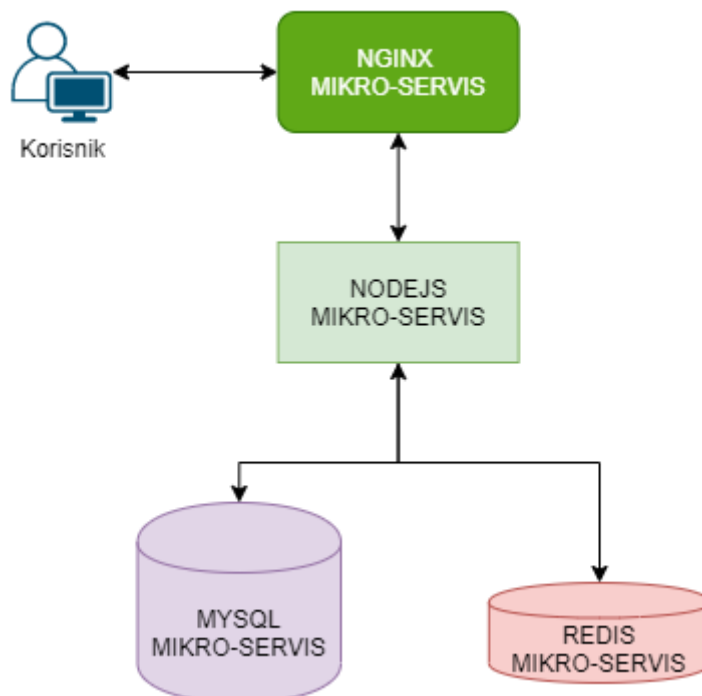
S ovim smo uspjeli automatski skalirati našu aplikaciju. Cijeli postupak je u biti vrlo jednostavan i sastoji se od samo jedne naredbe s kojom kreiramo HPA resurs. Kubernetes svojom jednostavnošću i moći predstavlja jedan od najboljih alata za automatsko skaliranje što smo se uvjerali i obrađenim primjerom.

7. CHAT APLIKACIJA

7.1 Opis

Sve dosadašnje demonstracije Kubernetes sustava rađene su na jednostavnim primjerima kako bi se što bolje razumjela svaka komponenta. U ovom poglavlju na Kubernetes klaster ćemo migrirati malo kompleksniju funkcionalnu Chat aplikaciju. Chat aplikacija će biti podijeljena u više mikro-servisa gdje je svaki mikro-servis jedna samostalna neovisna cjelina.

Arhitektura aplikacije prikazana je na slici ispod (Slika 7.1.). Nginx mikro-servis je web server koji poslužuje klijentsku Angular aplikaciju tj. „front-end“. Korišten je Nginx web server jer je isti veoma brz kod posluživanja statičkih resursa za razliku od NodeJs-a koji ima neke druge prednosti. NodeJs mikro-servis sadrži REST API server i Socket server. Socket je korišten za komunikaciju u stvarnom vremenu (engl. real-time). Za bazu podataka korištena je MYSQL baza podataka, a za spremanje sesija korišten je Redis. Redis je izrazito brza baza podataka iz razloga što se izvršava u RAM memoriji. Iz tog razloga koristi se i za „caching“. Svaka spomenuta komponenta izvršava se u izoliranom Docker containeru te komponente međusobno komuniciraju.

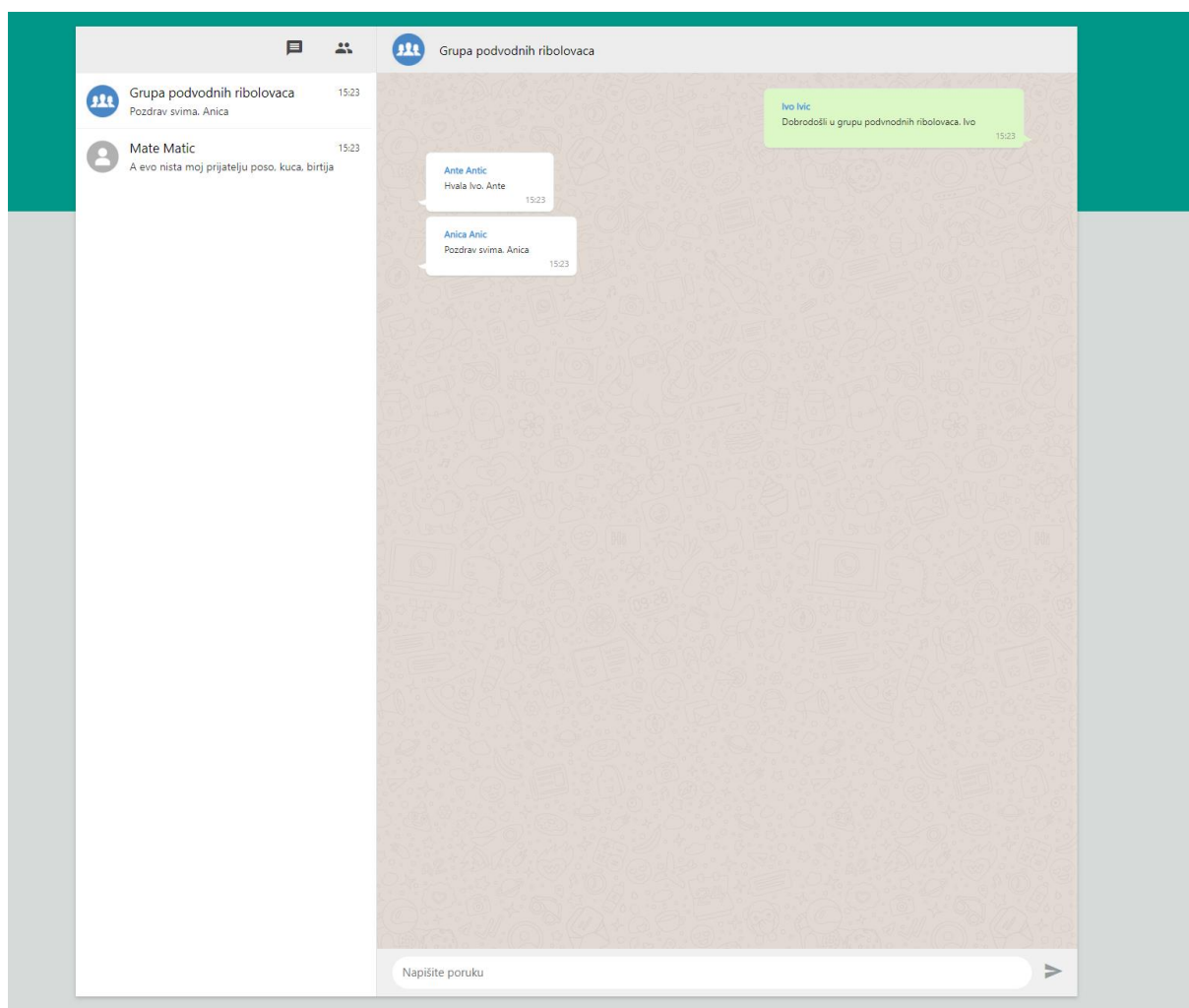


Slika 7.1. Arhitektura Chat aplikacije

Iako je aplikacija podijeljena u više mikro-servisa, komponente su i dalje dosta robusne. Za potrebe ovog diplomskog rada namjerno je napravljena ovakva arhitektura kako bi se fokus

stavio na sami Kubernetes sustav, a ne na razvoj aplikacije. Da se radi o produkcijskog chat aplikaciji arhitektura bi zasigurno bila sačinjena od većeg broja mikro-servisa gdje bi svaki mikro-servis radio samo jedan zadatak. Na primjer, NodeJs komponenta sastoji se više funkcionalnosti kao što su praćenje je li osoba na vezi, prosljeđivanje poruka, kreiranje novih poruka i grupa itd. Kod prave mikro-servis arhitektura svaka navedena funkcionalnost čini jedan mikro-servis.

Izgled Chat aplikacije će biti kao na slici ispod (Slika 7.2.). Dizajn aplikacije napravljen je po uzoru na web.whatsapp.com aplikaciju.



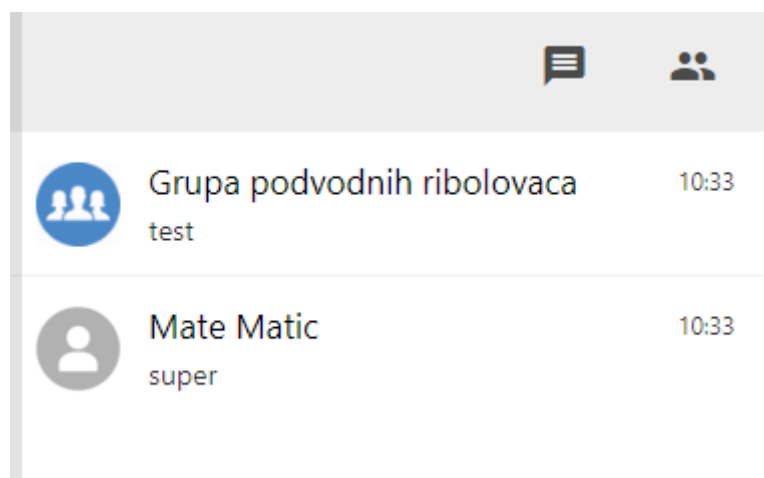
Slika 7.2. Izgled Chat aplikacije

7.2 Funkcionalnosti

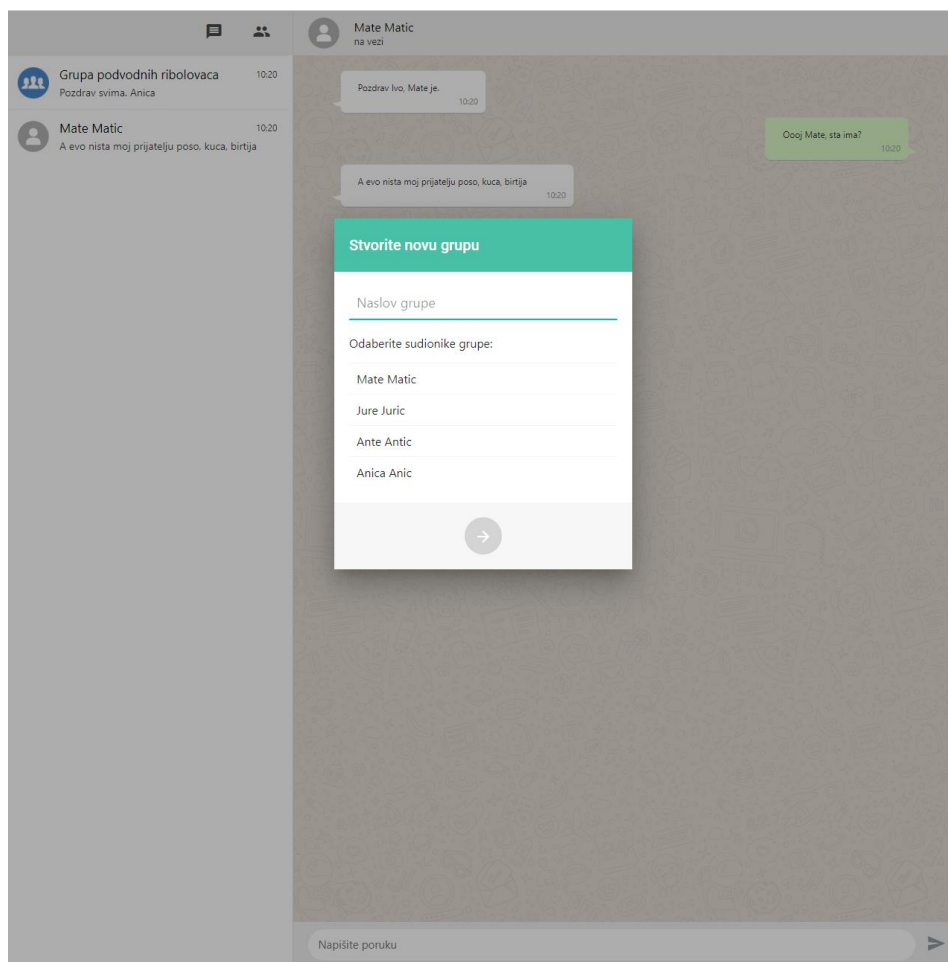
Chat aplikacija će imati sljedeće funkcionalnosti:

- Slanje pojedinačnih poruka drugim korisnicima (Slika 7.3.).
- Slanje grupnih poruka skupini korisnika (Slika 7.3.) (Slika 7.4.).

- Prikaz statusa „na vezi“ kada korisnik koristi aplikaciju (Slika 7.5.).
- Prikaz vremena poruke u skraćenom formatu (HH:mm) ukoliko je poruka poslana tog dana ili punog formata (MM.dd.YYYY, HH:mm) ukoliko je poruka starija od trenutnog datuma.
- Lista svih razgovora korisnika takva da se prikazuje samo zadnja poruka razgovora (Slika 7.3.). Lista je sortirana od najnovijeg razgovora prema najstarijem.
- Automatsko „skrolanje“ na dno razgovora pri dolasku i slanju novih poruka u prozoru razgovora (Slika 7.6.).
- Mogućnost pregleda svih sudionika grupe (Slika 7.7.).



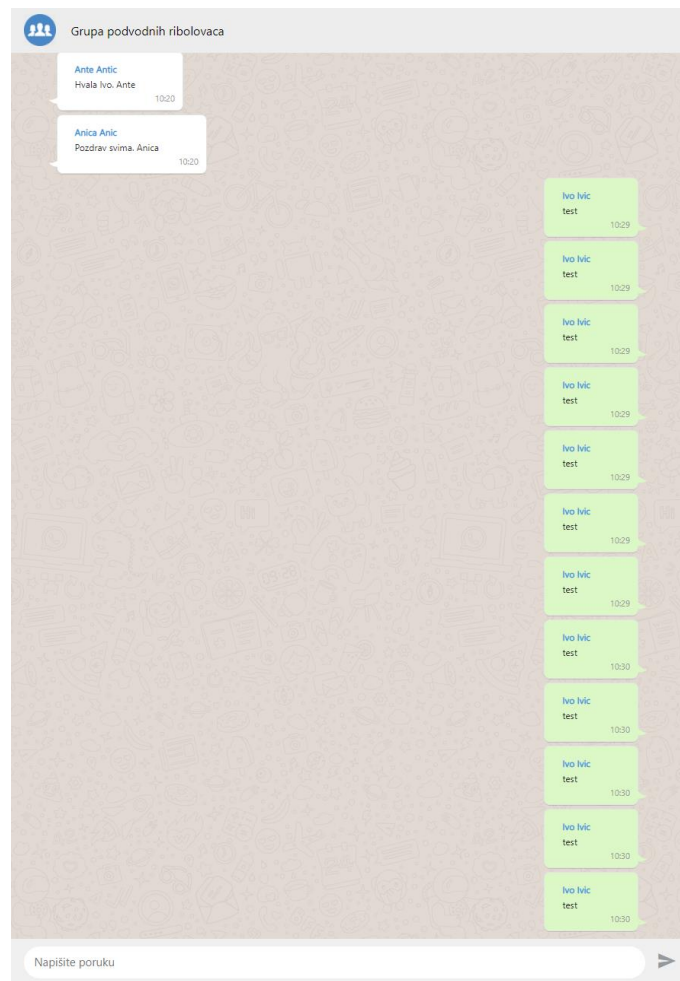
Slika 7.3. Lista zadnjih razgovora korisnika te ikone za kreiranje novog pojedinačnog ili grupnog razgovora



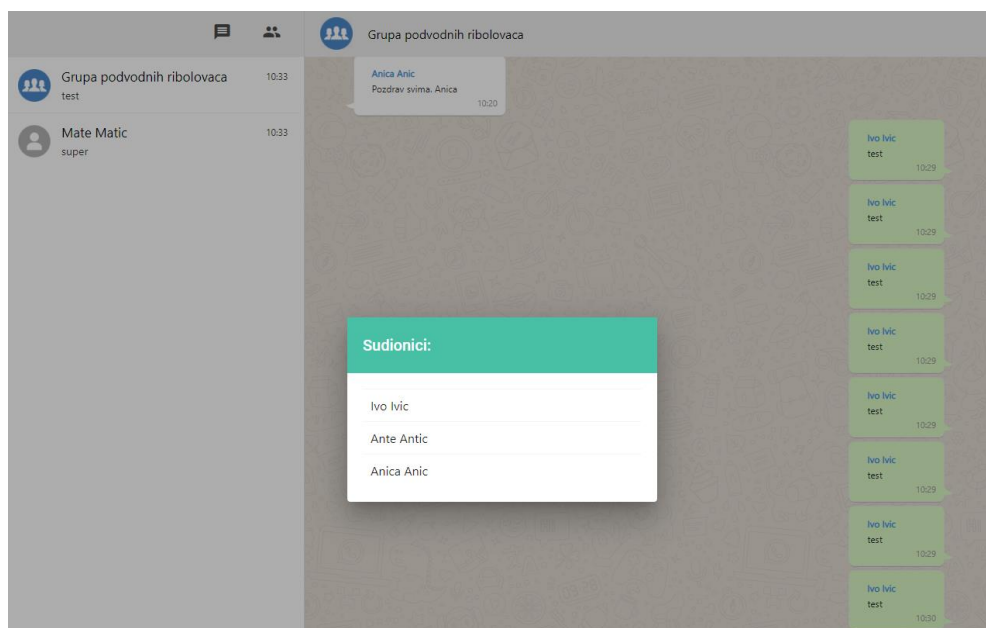
Slika 7.4. Kreiranje grupnog razgovora, odabir sudionika i unošenje imena grupe



Slika 7.5. Status „na vezi“ kada je osoba trenutno aktivna



Slika 7.6. Automatsko „skrolanje“ na dno razgovora



Slika 7.7. Pregled sudionika grupe

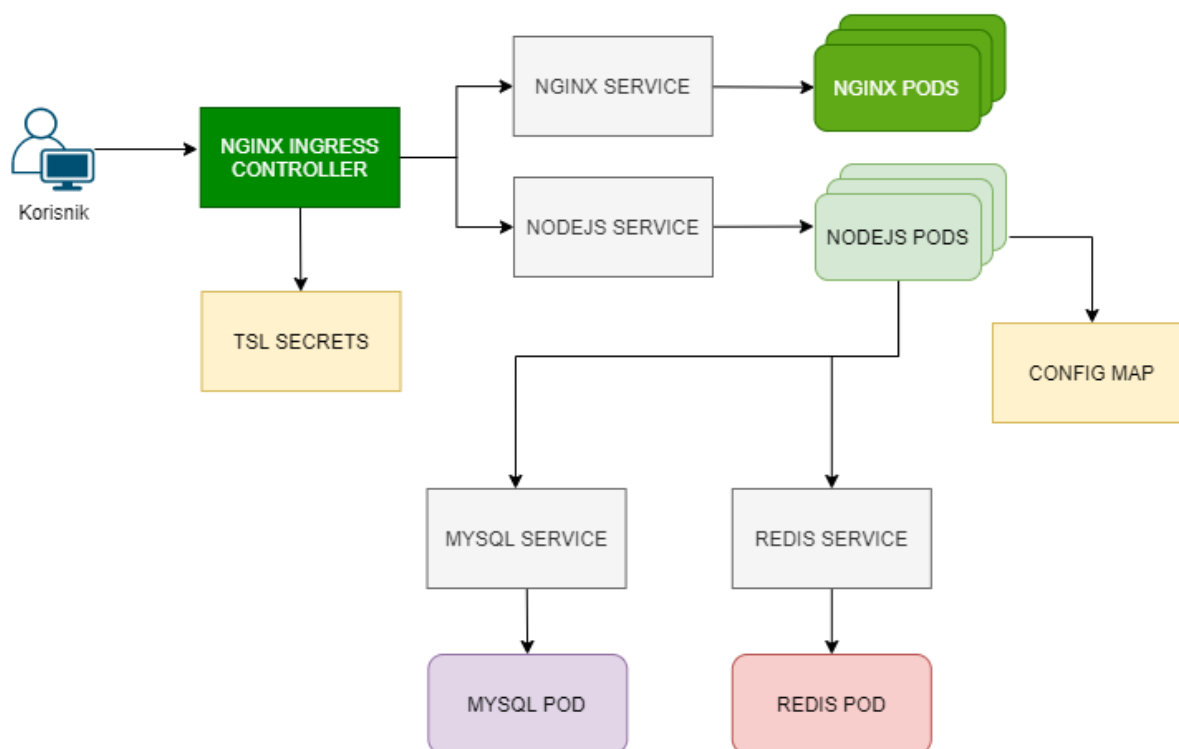
7.3 Izvršavanje na Kubernetes klasteru

Kako bi se Chat aplikacija izvršavala na Kubernetes klasteru moramo poslati opis aplikacije Kubernetes glavnom čvoru. Sav opis se može specificirati u jednu yaml datoteku ili razbiti u više yaml datoteka. U sklopu ovog diplomskog rada, opis svake komponente će biti vlastita yaml datoteka radi preglednosti i u svrhu lakšeg objašnjavanja.

Cijeli kod aplikacije i yaml datoteka dostupan je na GitHub linku:

<https://github.com/ikovac/chatApp-on-Kubernetes>. U kubernetes-deploy grani (engl. branch) se nalazi sav kod potreban za izvršavanje aplikacije na Kubernetes klasteru.

Kada aplikaciju uspješno migriramo na Kubernetes sustav, arhitektura će izgledati kao na slici ispod (Slika 7.8.).



Slika 7.8. Komponente Chat aplikacije na Kubernetes sustavu

Započet ćemo sa kreiranjem novog klastera na GKE koji ćemo nazvati „chat-app“ (Slika 7.9.).

Google Cloud Platform My First Project Search products and resources

Create a Kubernetes cluster + ADD NODE POOL REMOVE NODE POOL

Cluster basics

The new cluster will be created with the name, version, and in the location you specify here. After the cluster is created, name and location can't be changed.

Cluster set-up guides
MY FIRST CLUSTER

NAME
chat-app

Location type
☒ Zonal
☐ Regional

Zone
us-central1-c

☐ Specify default node locations
Current default: us-central1-c

Master version
Choose Release Channel to get automatic GKE upgrades as new versions are ready. Choose a static version to upgrade manually in the future. [Learn more.](#)

☐ Release channel
☒ Static version

Static version
1.14.10-gke.36 (default)

CREATE CANCEL Equivalent REST or command line

Slika 7.9. Kreiranje klastera na GKE

Kreiranje klastera će potrajati neko vrijeme. Kada klaster bude spreman za rad pojavit će se zelena kvačica (Slika 7.10.).

<input type="checkbox"/> Name ^	Location	Cluster size	Total cores	Total memory	Notifications	Labels
<input checked="" type="checkbox"/> chat-app	us-central1-c	3	3 vCPUs	11.25 GB		Connect  

Slika 7.10. Klaster je uspješno kreiran

Sada se možemo spojiti na novo-kreirani klaster klikom na dugme „Connect“ te kopiranjem dobivene naredbe u terminal na našem računalu (Slika 7.11.).

Connect to the cluster

You can connect to your cluster via command-line or using a dashboard.

Command-line access

Configure [kubectl](#) command line access by running the following command:

```
$ gcloud container clusters get-credentials chat-app --zone us-central1-c --project balmy-rhino-269709
```

Copy

Run in Cloud Shell

Cloud Console dashboard

You can view the workloads running in your cluster in the Cloud Console [Workloads dashboard](#).

Open Workloads dashboard

OK

Slika 7.11. Spajanje na klaster

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> gcloud container clusters get-credentials chat-app --zone us-central1-c --project balmy-rhino-269709
Fetching cluster endpoint and auth data.
kubeconfig entry generated for chat-app.
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files>
```

Slika 7.12. Kopiranje dobivene naredbe u terminal

Nakon kreiranja i spajanja na klaster kreće migriranje svih komponenti na klaster. U folderu `kubernetes-files` se nalaze sve yaml opisne datoteke koje ćemo poslati glavnom čvoru. Započet ćemo sa kreiranjem Ingress kontrolera. Koristit ćemo Ingress kontroler baziran na Nginx-u. Kako ovakav Ingress kontroler nije dio Google Kubernetes sustava moramo ga kreirati kroz opisne datoteke.

Za kreiranje Ingress kontrolera u terminal ćemo unijeti sljedeće naredbe:

1. `kubectl create clusterrolebinding cluster-admin-binding --clusterrole cluster-admin --user $(gcloud config get-value account)`

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl create clusterrolebinding cluster-admin-binding --clusterrole cluster-admin --user $(gcloud config get-value account)
clusterrolebinding.rbac.authorization.k8s.io/cluster-admin-binding created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files>
```

Slika 7.13. Kreiranje autorizacijskih klaster rola

2. `kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.30.0/deploy/static/mandatory.yaml`

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.30.0/deploy/static/mandatory.yaml
namespace/ingress-nginx created
configmap/nginx-configuration created
configmap/tcp-services created
configmap/udp-services created
serviceaccount/nginx-ingress-serviceaccount created
clusterrole.rbac.authorization.k8s.io/nginx-ingress-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-role created
rolebinding.rbac.authorization.k8s.io/nginx-ingress-role-nisa-binding created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress-clusterrole-nisa-binding created
deployment.apps/nginx-ingress-controller created
limitrange/ingress-nginx created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> █
```

Slika 7.14. Kreiranje Nginx Ingress kontrolera i pripadajućih komponenti

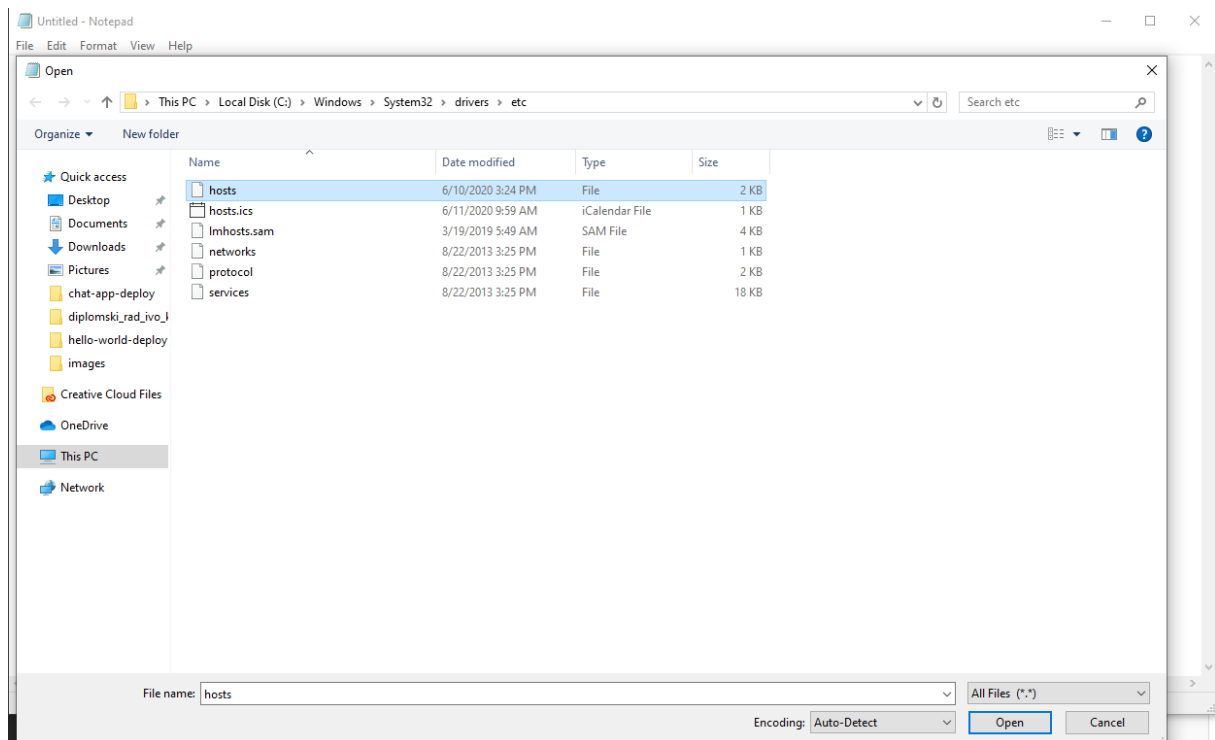
3. `https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.30.0/deploy/static/provider/cloud-generic.yaml`

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.30.0/deploy/static/provider/cloud-generic.yaml
service/ingress-nginx created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> █
```

Slika 7.15. Kreiranje Ingress servisa preko kojeg ćemo pristupiti Ingress kontroleru


Sada je Ingress kontroler kreiran. Međutim, Ingress kontroler još nije konfiguriran potrebama naše Chat aplikacije. To će biti učinjeno kasnije u posljednjem koraku nakon što kreiramo ostale komponente.


Prije nego što krenemo na sami proces stvaranja ostalih komponenti, uredit ćemo hosts datoteku na našem računalu kako bi napravili lokalni DNS server koji će promet sa željenih domena preusmjeravati na IP adresu kreiranog Ingress servisa. Da bi to napravili otvorimo Notepad ili neki drugi editor po želji s administratorskim privilegijama (Slika 7.16.).





Slika 7.16. Otvaranje host datoteke iz programa Notepad


U Services sekciji na GKE bi trebali vidjeti IP adresu stvorenog Ingress servisa. Tu adresu ćemo kopirati i dodijeliti domenama koje ćemo stvoriti.


Kubernetes Engine


Clusters


Workloads

Services & Ingress


Applications


Configuration


Storage

Object Browser

Services & Ingress

REFRESH



CREATE INGRESS

DELETE



SERVICES

INGRESS

Services are sets of Pods with a network endpoint that can be used for discovery and load balancing. Ingresses are collections of rules for routing external HTTP(S) traffic to Services.

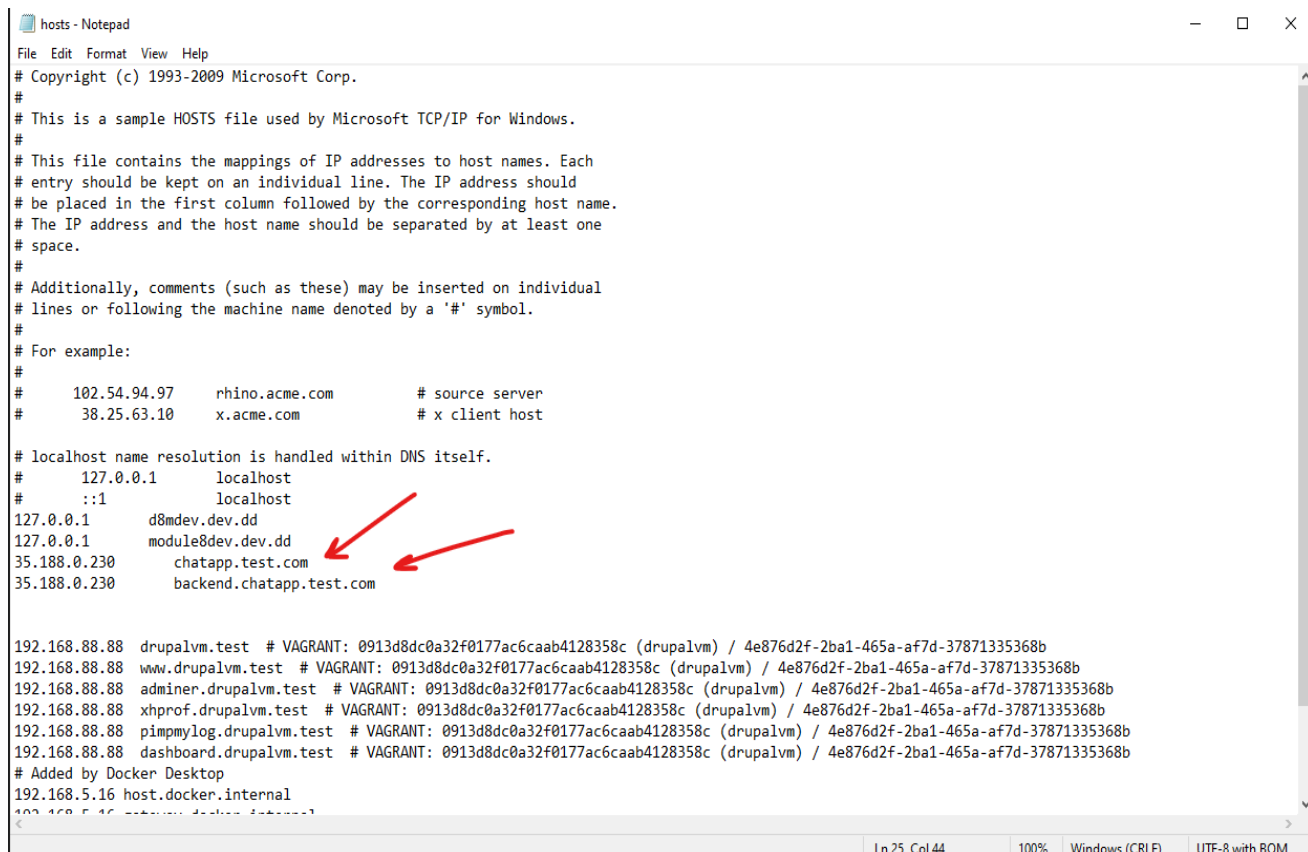
Is system object : False 

Filter secrets and config maps

<input type="checkbox"/>	Name ↑	Status	Type	Endpoints	Pods	Namespace	Cluster
<input type="checkbox"/>	ingress-nginx	 OK	External load balancer	35.188.0.230:80 	1/1	ingress-nginx	chat-app

Slika 7.17. Ingress servis je dobio javnu IP adresu

Sada u datoteci hosts možemo dodati željene domene. Korisnička aplikacija nalazit će se na domeni chatapp.test.com dok će se Nodejs server nalaziti na domeni backend.chatapp.test.com (Slika 7.18.). Mogu se odabrati bilo koje druge dvije domene po želji, samo treba voditi računa da se te iste domene kasnije unesu u konfiguraciju Ingress kontrolera.



```
hosts - Notepad
File Edit Format View Help
# Copyright (c) 1993-2009 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#       102.54.94.97     rhino.acme.com           # source server
#       38.25.63.10     x.acme.com               # x client host

# localhost name resolution is handled within DNS itself.
#       127.0.0.1       localhost
#       ::1             localhost
127.0.0.1       d8mdev.dev.dd
127.0.0.1       module8dev.dev.dd
35.188.0.230    chatapp.test.com
35.188.0.230    backend.chatapp.test.com

192.168.88.88   drupalvm.test # VAGRANT: 0913d8dc0a32f0177ac6caab4128358c (drupalvm) / 4e876d2f-2ba1-465a-af7d-37871335368b
192.168.88.88   www.drupalvm.test # VAGRANT: 0913d8dc0a32f0177ac6caab4128358c (drupalvm) / 4e876d2f-2ba1-465a-af7d-37871335368b
192.168.88.88   adminer.drupalvm.test # VAGRANT: 0913d8dc0a32f0177ac6caab4128358c (drupalvm) / 4e876d2f-2ba1-465a-af7d-37871335368b
192.168.88.88   xhprof.drupalvm.test # VAGRANT: 0913d8dc0a32f0177ac6caab4128358c (drupalvm) / 4e876d2f-2ba1-465a-af7d-37871335368b
192.168.88.88   pimpleylog.drupalvm.test # VAGRANT: 0913d8dc0a32f0177ac6caab4128358c (drupalvm) / 4e876d2f-2ba1-465a-af7d-37871335368b
192.168.88.88   dashboard.drupalvm.test # VAGRANT: 0913d8dc0a32f0177ac6caab4128358c (drupalvm) / 4e876d2f-2ba1-465a-af7d-37871335368b
# Added by Docker Desktop
192.168.5.16    host.docker.internal
192.168.5.16    gateway.docker.internal
```

Slika 7.18. Dodavanje domena u hosts datoteku

Nakon što smo dodali željene domene, možemo spremiti datoteku i izaći iz iste.

Kada u Internet pretraživač upišemo neku od ovih domena, preusmjerit će nas na IP adresu Ingress servisa tj. na sami Ingress servis koji će potom napraviti preusmjeravanje na onaj servis za koji je promet namijenjen. Ako promet dolazi s domene chatapp.test.com biti će preusmjeren na „front-end“ Angular aplikaciju, a ukoliko je domena backend.chatapp.test.com biti će preusmjeren na NodeJs „back-end“.

Već su objašnjene razne prednosti korištenja Ingress kontrolera a jedna od njih je i mogućnost dodavanja TLS certifikata. Da bi omogućili HTTPS promet prvo ćemo kreirati Secret resurs s TLS certifikatima a kasnije u konfiguraciji Ingress kontrolera povezati kreirane Secret resurse s domenama.

Da bi kreirali Secret resurs za prethodno generirane certifikate koristit ćemo sljedeće naredbe:

```
kubectl create secret tls tls-secret --cert=tls.cert --key=tls.key
kubectl create secret tls tls-secret2 --cert=tls2.cert --key=tls2.key
```

Naredbe će kreirati dva Secret resursa za dva certifikata. Jedan certifikat ćemo koristiti za „front-end“ komponentu aplikacije, a drugi za „back-end“.

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl create secret tls tls-secret --cert=tls.cert --key=tls.key
secret/tls-secret created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl create secret tls tls-secret2 --cert=tls2.cert --key=tls2.key
secret/tls-secret2 created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> █
```

Slika 7.19. Kreiranje Secret resursa za TSL certifikate

Za sada smo samo kreirali Secret resurse za generirane certifikate, ali još nigdje nismo iskoristili kreirane Secret resurse. To ćemo učiniti kasnije prilikom konfiguracije Ingress kontrolera.

Sljedeći korak će biti kreiranje ConfigMap resursa. Ovaj resurs sadrži konfiguracijske parametre koji će biti dostupni u NodeJs Pod-ovima kao okolinske varijable.

Kod svih opisnih datoteka moguće je pronaći u DODATAK A poglavlju.

Ovaj resurs ćemo kreirati sa već dobro poznatom naredbom „*kubectl apply -f ./ime-datoteke.yaml*“ gdje je ime datoteke u našem primjeru config-map.yaml.

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f ./config-map.yaml
configmap/my-config created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> █
```

Slika 7.20. Objava ConfigMap resursa

Slika ispod (Slika 7.21.) prikazuje sljedeći korak objave svih ostalih komponenti osim Ingress konfiguracije što će biti učinjeno nakon.

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f ./mysql-deployment.yaml
service/mysql created
deployment.apps/mysql created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f ./redis-deployment.yaml
service/redis created
deployment.apps/redis created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f ./nodejs-deployment.yaml
deployment.apps/nodejs created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f ./nodejs-service.yaml
service/nodejs-service created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f ./nginx-deployment.yaml
deployment.apps/nginx created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f ./nginx-service.yaml
service/nginx-service created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> █
```

Slika 7.21. Objava Mysql, Redis, NodeJs i Nginx mikro-servisa

Primjećujemo da smo Mysql i Redis mikro-servis objavili s jednom opisnom datotekom dok smo NodeJs i Nginx mikro-servis objavili sa dvije opisne datoteke (jednu za Deployment resurs a drugu za Service resurs). To je napravljeno na taj način samo da se pokaže na primjeru da nije bitno hoćemo li opis aplikacije objaviti kroz jednu ili više yaml datoteka.

Sada možemo konačno objaviti i konfiguraciju Ingress kontrolera za potrebe naše aplikacije.

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl apply -f .\ingress.yaml
ingress.networking.k8s.io/chatapp-ingress-frontend created
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> |
```

Slika 7.22. Objava opisa Ingress konfiguracije

Kod Ingress opisne datoteke kao i ostalih opisnih datoteka kao što je već rečeno se nalazi u poglavlju DODATAK A. Ukoliko pogledamo kôd ove datoteke možemo vidjeti da sadrži pravila usmjeravanja za dvije domene koje smo odredili u hosts datoteci. Osim pravila usmjeravanja svakoj domeni smo dodali i TSL certifikat kako bi se mogao koristiti HTTPS protokol.

Ovdje ćemo zastati na trenutak i pogledati na GKE što smo sve do sad kreirali. Na slici ispod (Slika 7.23.) vidimo sve kreirane Servis i Ingress resurse.

Kubernetes Engine

Clusters

Workloads

Services & Ingress

Applications

Configuration

Storage

Object Browser

Services & Ingress

REFRESH

CREATE INGRESS

DELETE

SERVICES

INGRESS

Services are sets of Pods with a network endpoint that can be used for discovery and load balancing. Ingresses are collections of rules for routing external HTTP(S) traffic to Services.

Is system object : False

Filter secrets and config maps

<input type="checkbox"/>	Name ↑	Status	Type	Endpoints	Pods	Namespace	Cluster
<input type="checkbox"/>	chatapp-ingress-frontend	Creating ing...	Ingress	chatapp.test.com/ backend.chatapp.test.com/ 	0/0	default	chat-app
<input type="checkbox"/>	ingress-nginx	OK	External load balancer	35.188.0.230:80	1/1	ingress-nginx	chat-app
<input type="checkbox"/>	mysql	OK	Cluster IP	None	1/1	default	chat-app
<input type="checkbox"/>	nginx-service	OK	Node Port	10.20.7.245:80 TCP	1/1	default	chat-app
<input type="checkbox"/>	nodejs-service	OK	Node Port	10.20.10.7:80 TCP	1/1	default	chat-app
<input type="checkbox"/>	redis	OK	Cluster IP	None	1/1	default	chat-app

Slika 7.23. Kreirani Service resursi na GKE

Također pod Workloads sekcijom vidimo sve kreirane Deployment resurse odnosno Pod-ove (Slika 7.24.).

Google Cloud Platform My First Project Search products and resources

Kubernetes Engine Workloads REFRESH DEPLOY DELETE

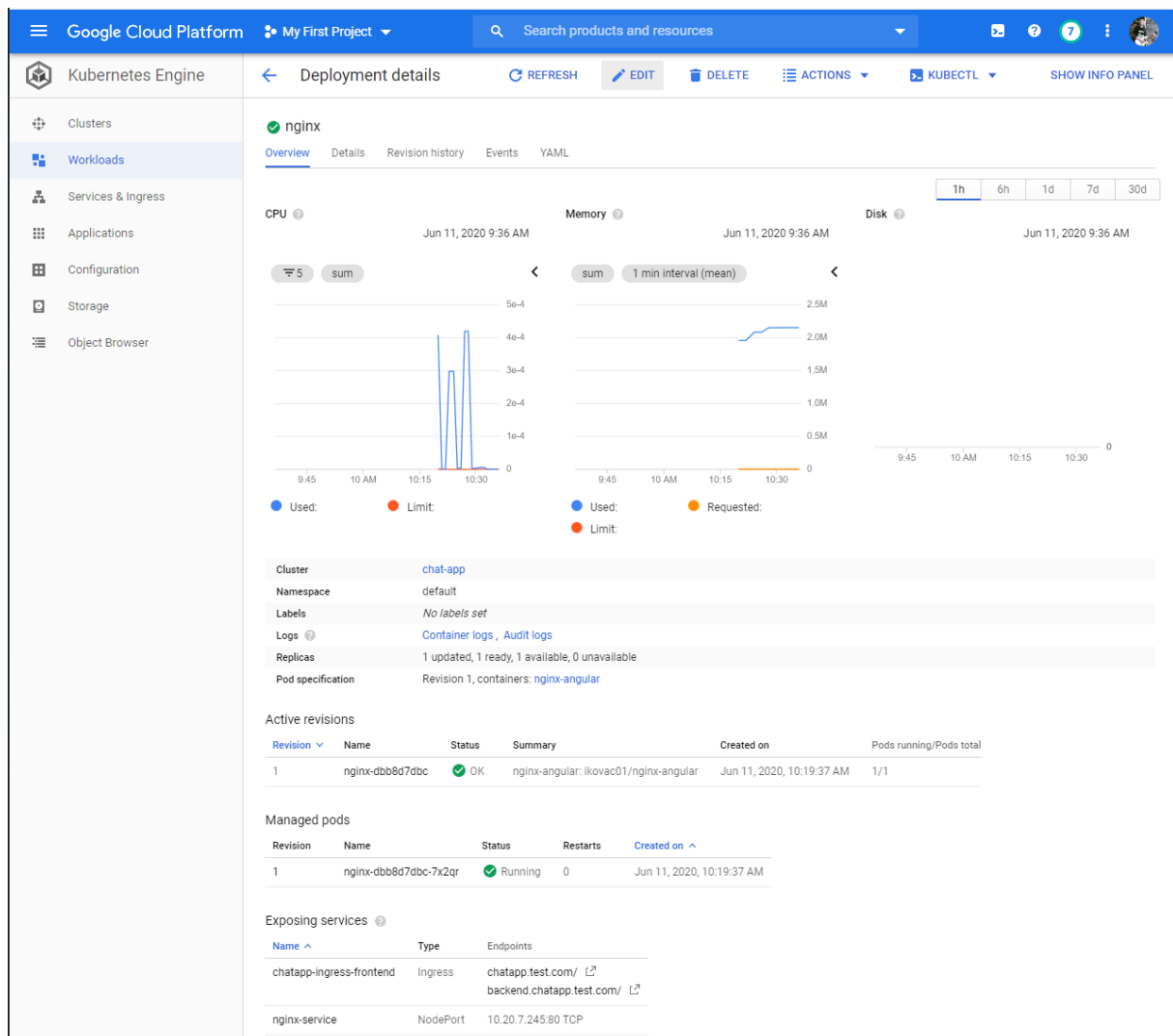
Workloads are deployable units of computing that can be created and managed in a cluster.

Is system object : False Filter workloads

<input type="checkbox"/>	Name ↑	Status	Type	Pods	Namespace	Cluster
<input type="checkbox"/>	mysql	OK	Deployment	1/1	default	chat-app
<input type="checkbox"/>	nginx	OK	Deployment	1/1	default	chat-app
<input type="checkbox"/>	nginx-ingress-controller	OK	Deployment	1/1	ingress-nginx	chat-app
<input type="checkbox"/>	nodejs	OK	Deployment	1/1	default	chat-app
<input type="checkbox"/>	redis	OK	Deployment	1/1	default	chat-app

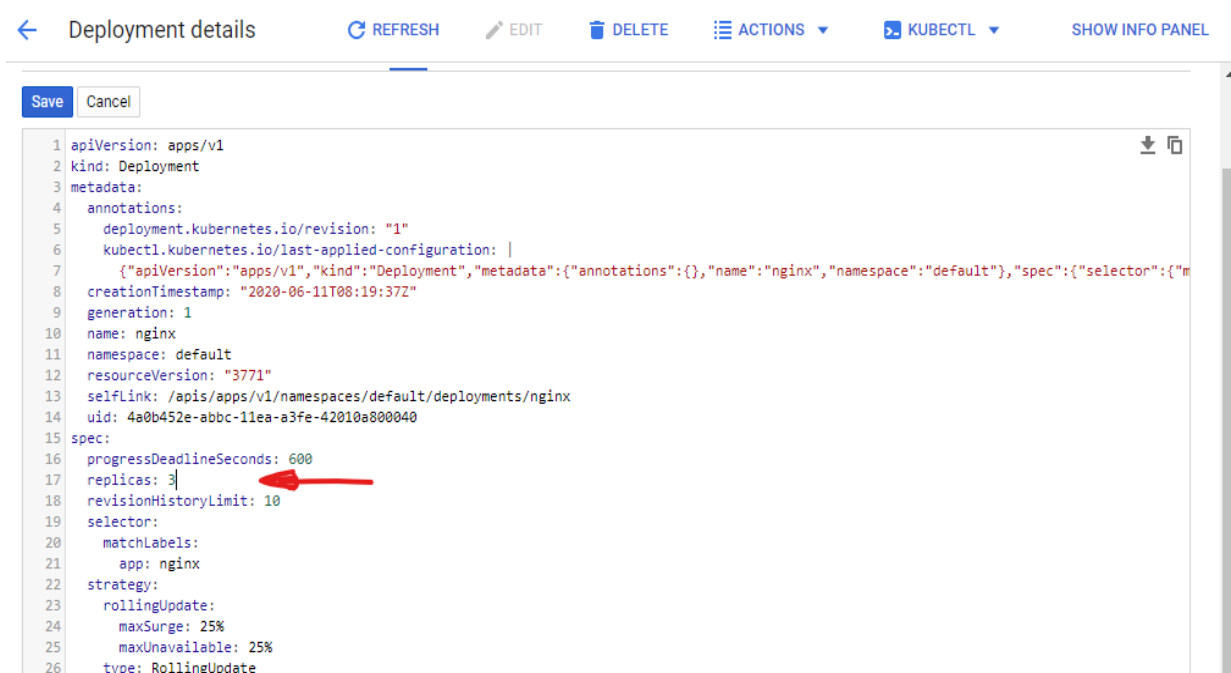
Slika 7.24. Kreirani Deployment resursi na GKE

Primjećujemo da za svaki mikro-servis imamo po jedan Pod jer smo u opisnim datotekama mikro-servisa specificirali da želimo točno jednu repliku. Za svaki mikro-servis stavljena je jedna replika kako bi sada mogli demonstrirati proces skaliranja komponenti aplikacije. Prvo što ćemo napraviti je ručno skalirati Nginx komponentu aplikacije tako što kliknemo na nginx Deployment resurs i uredimo datoteku pritiskom na dugme „Edit“ (Slika 7.25.). Uređivanje se može napraviti i iz komandne linije, ali putem Internet pretraživača je znatno preglednije.



Slika 7.25. Uređivanje Nginx Deployment opisne datoteke

Pod sekcijom „replicas“ mijenjamo željeni broj replika. U ovom primjeru, promijenit ćemo broj replika s 1 na 3 (Slika 7.26.).



Slika 7.26. Ručno skaliranje Pod-ova Nginx komponente

Možemo vidjeti kako se broj replika povećao na 3 jer su kreirana 2 nova Pod-a (Slika 7.27.).

Kubernetes Engine

Clusters

Workloads

Services & Ingress

Applications

Configuration

Storage

Object Browser

Workloads

REFRESH

DEPLOY

DELETE

Workloads are deployable units of computing that can be created and managed in a cluster.

Is system object : False

Filter workloads

<input type="checkbox"/>	Name	Status	Type	Pods	Namespace	Cluster
<input type="checkbox"/>	mysql	OK	Deployment	1/1	default	chat-app
<input type="checkbox"/>	nginx	OK	Deployment	3/3	default	chat-app
<input type="checkbox"/>	nginx-ingress-controller	OK	Deployment	1/1	ingress-nginx	chat-app
<input type="checkbox"/>	nodejs	OK	Deployment	1/1	default	chat-app
<input type="checkbox"/>	redis	OK	Deployment	1/1	default	chat-app

Slika 7.27. Kreirani su novi Pod-ovi kako bi se ispunio uvjet od 3 replike

Na ovaj način uspješno smo proveli ručno skaliranje. Ručno skaliranje i nije baš zanimljivo i korisno jer podrazumijeva čovjekovu intervenciju. U stvarnim aplikacijama nagle promjene prometa se mogu pojaviti preko noći dok svi radnici firme spavaju te je u tim trenutcima nemoguće ručno skalirati aplikaciju.

Sada ćemo pokazati i automatsko skaliranje željenog resursa. U primjeru ćemo koristiti Nginx Deployment resurs kojeg smo već ručno skalirali. Da bi omogućili automatsko skaliranje moramo kreirati HPA resurs (Slika 7.28.). Kreiranje istog je vrlo jednostavno te moramo unijeti samo jednu naredbu kao što je prikazano na slici. Kreirali smo HPA resurs koji će biti

zadužen za automatsko skaliranje te smo specificirali da želimo imati minimalno 1 repliku i maksimalno 5 replika. Također smo specificirali da želimo da nam opterećenje procesora svakog Pod-a bude na 30%.

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl autoscale deployment nginx --cpu-percent=30 --min=1 --max=5
horizontalpodautoscaler.autoscaling/nginx autoscaled
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files>
```

Slika 7.28. Kreiranje HPA resursa

S naredbom „*kubectl get hpa*“ možemo dohvatiti novo-stvoreni HPA resurs. HPA resursu će trebati neko vrijeme dok ne započne s normalnim radom te će pod stvarnim opterećenjem prikazivati „unknown“ (Slika 7.29.).

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl get hpa
NAME      REFERENCE          TARGETS      MINPODS  MAXPODS  REPLICAS  AGE
nginx     Deployment/nginx    <unknown>/30%  1         5         3          24s
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files>
```

Slika 7.29. HPA resursu treba neko vrijeme dok ne započne s normalnim radom

Nakon što je HPA započeo s normalnim radom prepoznat će trenutno opterećenje komponente za koju je zadužen.

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl get hpa
NAME      REFERENCE          TARGETS      MINPODS  MAXPODS  REPLICAS  AGE
nginx     Deployment/nginx    0%/30%       1         5         3          57s
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files>
```

Slika 7.30. HPA resurs je započeo s normalnim radom

Primjećujemo da je trenutno opterećenje komponente 0% a traženi iznos je 30%. Razlog tome je što trenutno nemamo nikakav promet na aplikaciji. Nakon nekog vremena HPA će shvatiti da imamo višak resursa (3 replike) te će započeti sa skaliranjem broja replika. U ovom primjeru će smanjiti broj replika na samo jednu repliku jer je procijenio da mu je to dovoljno za normalni rad pri trenutnom opterećenju (Slika 7.31.).

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files> kubectl get hpa
NAME      REFERENCE          TARGETS      MINPODS  MAXPODS  REPLICAS  AGE
nginx     Deployment/nginx    0%/30%       1         5         1          5m52s
PS D:\Documents\nodejs\chatApp-on-Kubernetes\kubernetes-files>
```

Slika 7.31. Smanjivanje broja replika s 3 na 1

Uvjerili smo se da skaliranje na manji broj replika ispravno radi. Bitno je uočiti kako je za skaliranje na manji broj replika trebalo čak 5 minuta. Već smo spomenuli kako će HPA smanjiti broj replika tek kad se uvjeri da te replike stvarno neće trebati i da je to vrijeme puno duže nego vrijeme potrebno da se poveća broj replika. Povećanje broja replika bi se trebalo dogoditi skoro pa istog trenutka kada dođe do porasta prometa. Da bi se uvjerili da je to stvarno tako povećat ćemo promet na aplikaciji. To ćemo učiniti tako što ćemo pokrenuti „busybox“ Docker container unutar klastera koji će slati zahtjeve na Nginx komponentu. Naredba koju ćemo izvršiti je sljedeća: „`kubectl run -it --rm --restart=Never loadgenerator --image=busybox -- sh -c "while true; do wget -O - -q http://nginx-service.default; done"`“. Kreirali smo container unutar kojeg ćemo stvoriti while petlju. Unutar while petlje slati ćemo zahtjeve nginx servisu. Ukoliko sada dohvatimo HPA resurs uočavamo povećanje broja replika s jedne replike na 2 (Slika 7.32.). Razlog tome je povećanje prometa na komponenti te vidimo da je trenutno opterećenje svake replike 24% što je u granicama od 30% opterećenja.

```
PS D:\Documents\nodejs\chatApp-on-Kubernetes> kubectl get hpa
NAME      REFERENCE          TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
nginx     Deployment/nginx    24%/30%   1         5         2          8m30s
PS D:\Documents\nodejs\chatApp-on-Kubernetes>
```

Slika 7.32. Povećanje broja replika uslijed porasta prometa

S ovim smo završili migriranje aplikacije na Kubernetes klaster gdje se uspješno izvršava i automatski skalira ovisno o prometu.

8. ZAKLJUČAK

U sklopu ovog diplomskog rada objašnjeni su glavni razlozi zašto, gdje i kada koristiti mikro-servis arhitekturu i Kubernetes sustav. Iako je monolitni razvoj znatno brži, dobra odluka je na početku odvojiti malo više vremena i dizajnirati kvalitetnu mikro-servis arhitekturu koja će kasnije biti vrlo lako proširiva novim funkcionalnostima te izrazito lako skalabilna. Želja svakog kreatora softvera je da isti uspije i ima puno prometa. Tada dolazi do promišljanja o skaliranju i poboljšanju performansi aplikacije.

Monolitne aplikacije su najčešće ograničene samo na skupo vertikalno skaliranje kod kojeg je najveća mana što imamo SAMO jedan ali izrazito moćan stroj. Problem se javlja kada nam taj jedan server prestane s radom što nemamo drugi tako moćan server ili ukoliko imamo što ćemo morati ručno migrirati aplikaciju na novi server. U tom procesu migriranja na novi server, aplikacija će neko vrijeme biti nedostupna što može rezultirati značajnim gubicima novca, ali i korisnika zbog lošeg iskustva. Zaključujemo da je mnogo bolje imati više slabijih servera gdje će prilikom prestanka rada jednog servera, aplikacija moći nesmetano raditi na ostalim zdravim serverima dok se ne riješi problem problematičnog servera. Naravno i dalje bi ručno trebali migrirati sve komponente aplikacije sa problematičnog servera na ostale zdrave servere da nema sustava kao što je Kubernetes.

Kubernetes osim što znatno olakšava upravljanje velikim brojem mikro-servisa ima puno drugih prednosti. Kubernetes sustavu kroz opisne datoteke šaljemo željenu specifikaciju aplikacije. Kubernetes vodi brigu da trenutno stanje klastera uvijek odgovara željenom specificiranom stanju. To znači da ukoliko neka komponenta prestane s radom, Kubernetes će je zamijeniti novom, zdravom komponentom. Također, ukoliko neki radni čvor prestane s radom, Kubernetes će sve komponente koje se nalaze na tom radnom čvoru migrirati na drugi zdravi radni čvor. Zbog svega navedenog, razvojni tim može mirno spavati noću, bez brige da će aplikacija preko noći prestati s radom.

U radu su navedene i pojašnjene glavne komponente koje tvore Kubernetes sustav kao i resursi koje je potrebno poznavati kako bi aplikaciju uspješno migrirali na Kubernetes sustav. Iz primjera u diplomskom radu može se vidjeti kako se cijeli proces migriranja aplikacije na Kubernetes klaster svodi na objavljivanje yaml opisnih datoteka Kubernetes glavnom čvoru. Cijeli proces je izrazito jednostavan. U procesu učenja Kubernetes sustava najteži dio je upoznavanje samih komponenti sustava jer ih ima puno. Razlog tome je što je Kubernetes

sustav izrazito moćan te sadrži puno funkcionalnosti. Glavni resursi s kojima smo se upoznali su Pod-ovi, Deployment, Service, Volume, ConfigMap i Secret resursi.

Osim upoznavanja sa samim komponentama spomenuta je vrlo važna funkcionalnost Kubernetes sustava, a to je automatsko skaliranje. Vidjeli smo da određenu komponentu aplikacije možemo vrlo jednostavno skalirati sa samo jednom naredbom. Iako ovaj proces vjerojatno zvuči puno teži, koristeći Kubernetes sustav, on se sveo na samo jednu naredbu.

Zbog svih navedenih i objašnjenih funkcionalnosti Kubernetes sustava zaključujemo kako ima smisla koristiti isti. Osim samih funkcionalnosti razvojne timove fascinira i lakoća korištenja i implementacija Kubernetes sustava. I za kraj ostaje još samo jednom pokušati dočarati benefit mirnog spavanja noću bez brige da će nam u 3 sata ujutro zvoniti mobitel jer je aplikacija prestala s radom. Ukoliko su svi odmorni i naspavani, unutar radnog vremena će moći još bolje obavljati svoj posao i posvetiti se rješavanju problema zbog kojih se dogodila pogreška u radu aplikacije.

LITERATURA

[1] „Dive Into Docker“, s interneta,

https://diveintodocker.com/?utm_source=nj&utm_medium=youtube&utm_campaign=virtual-machines-vs-docker-containers, 26. svibnja 2020.

[2] „10 companies that implemented the microservice architecture and paved the way for others“, s interneta, <https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/>, 27. svibnja 2020.

[3] Marko Lukša, „Kubernetes in Action“, Manning, NY, 2018.

[4] „What is Kubernetes?“, s interneta, <https://linuxhint.com/what-is-kubernetes/>, 28. svibnja 2020.

POPIS OZNAKA I KRATICA

AMQP – Advanced Message Queueing Protocol

API – Application Programming Interface

CPU – Central Processing Unit

GKE – Google Kubernetes Engine

HDD – Hard Disk Drive

HPA – Horizontal Pod Autoscaler

HTTP - HyperText Transfer Protocol

itd – i tako dalje

Ops – Information-technology operations

OS – Operacijski Sustav

QPS – Queries Per Second

REST – REpresentational State Transfer

sl – slično

SSD - Solid-state Drive

TCP – Transmission Control Protocol

VM – Virtualna Mašina

SAŽETAK

Cilj ovog diplomskog rada bio je objasniti sve prednosti mikro-servis arhitekture u odnosu na monolitni razvoj. Iako je monolitni pristup znatno jednostavniji i brži skriva mnoge nedostatke. Mikro-servis arhitektura omogućava razvoj svake komponente aplikacije zasebno unutar manjih timova u firmi. Također omogućava i zasebno skaliranje svake komponente.

Skoro sve velike kompanije uvidjele su prednosti mikro-servis arhitekture te prešle na istu. Međutim, to može rezultirati velikim brojem nezavisnih i različitih mikro-servisa. Raditi s tako velikim brojem različitih komponenti je izrazito teško. Ovakav razvoj bi bio gotovo nemoguć bez alata kao što je Kubernetes sustav.

Kubernetes sustav omogućava jednostavno upravljanje sa stotinama, tisućama ili više različitih mikro-servisa. Osim samog upravljanja različitim mikro-servisima, Kubernetes sustav uvijek vodi računa da je aplikacija uvijek dostupna. Ukoliko dođe do prekida rada neke komponente, neispravna komponenta se zamijeni novom, ispravnom komponentom.

Kubernetes omogućuje i automatsko skaliranje što predstavlja vrlo moćan i poželjan alat svake kompanije. Prilikom porasta prometa nije rijetkost da aplikacije prestanu s radom jer ne mogu podnijeti novo opterećenje. Uz Kubernetes sustav nema ovakvih briga. Ukoliko dođe do povećanog opterećenja Kubernetes sustav radi automatsko skaliranje tako što kreira nove replike određene komponente kako bi svi dolazeći zahtjevi bili obrađeni.

Zbog svih navedenih prednosti, Kubernetes sustav bio inspiracija za temu diplomskog rada.

Ključne riječi – Kubernetes, Chat aplikacija, Docker, skalabilne aplikacije, mikro-servis arhitektura

SUMMARY

The main goal of this graduate thesis is to explain all the advantages of microservice architecture over to monolithic development. Although the monolithic development is much simpler and faster, it has many disadvantages. Each component in microservice architecture is developed independently by smaller teams. Also, each component can be scaled individually depending on needs.

Almost all large companies switched to microservice architecture because of all benefits that microservice architecture provides. However, microservice architecture can result in a large number of independent microservices. Managing those microservice is extremely hard and would be impossible without some helping tool such as Kubernetes.

The Kubernetes system makes it easy to manage hundreds, thousands or more different microservices. In addition to the management, the Kubernetes system takes care that the application is always running. If some component stops working properly, it will be replaced with new, working component.

Also, Kubernetes provides autoscaling tool out of the box. It isn't a rare thing that application stops working due to load increase. With the Kubernetes system there are no such worries. If the increase in load occurs, the Kubernetes system will automatically scale specific component by creating new replicas so that all incoming requests are processed.

Title – Deploying a scalable chat application to the Kubernetes cluster

Keywords - Kubernetes, Chat application, Docker, scalable applications, microservice architecture

DODATAK A

config-map.yaml datoteka:

```
apiVersion: v1
data:
  NODEJS_DB_HOST: mysql
  NODEJS_DB_USER: root
  NODEJS_DB_PASSWORD: admin
  NODEJS_DB: chatapp
  NODEJS_JWT_SECRET: my_hard_to_guess_secret
  NODEJS_REDIS_HOST: redis
  NODEJS_CLIENT_HOST: https://chatapp.test.com
kind: ConfigMap
metadata:
  name: my-config
```

mysql-deployment.yaml datoteka:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
    - port: 3306
  selector:
    app: mysql
  clusterIP: None
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - image: mysql:5.7
```

```
name: mysql
env:
- name: MYSQL_ROOT_PASSWORD
  value: admin
- name: MYSQL_USER
  value: ivo
- name: MYSQL_PASSWORD
  value: admin
- name: MYSQL_DATABASE
  value: chatapp
```

redis-deployment.yaml datoteka:

```
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  ports:
  - port: 6379
  selector:
    app: redis
  clusterIP: None
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  selector:
    matchLabels:
      app: redis
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - image: redis
        name: redis
```

nodejs-deployment.yaml datoteka:

```
apiVersion: apps/v1
```

```

kind: Deployment
metadata:
  name: nodejs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nodejs
  template:
    metadata:
      labels:
        app: nodejs
    spec:
      containers:
      - image: ikovac01/nodejs
        name: nodejs
        resources:
          requests:
            cpu: 100m
        envFrom:
        - configMapRef:
            name: my-config

```

nodejs-service.yaml datoteka:

```

apiVersion: v1
kind: Service
metadata:
  name: nodejs-service
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 3000
  selector:
    app: nodejs

```

nginx-deployment.yaml datoteka:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx

```

```

template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - image: ikovac01/nginx-angular
        name: nginx-angular
        resources:
          requests:
            cpu: 100m

```

nginx-service.yaml datoteka:

```

apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: LoadBalancer
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
  selector:
    app: nginx

```

ingress.yaml datoteka:

```

apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: chatapp-ingress-frontend
spec:
  tls:
    - hosts:
        - chatapp.test.com
      secretName: tls-secret
    - hosts:
        - backend.chatapp.test.com
      secretName: tls-secret2
  rules:
    - host: chatapp.test.com
      http:
        paths:
          - path: /
            backend:

```

```
    serviceName: nginx-service
    servicePort: 80
- host: backend.chatapp.test.com
  http:
    paths:
    - path: /
      backend:
        serviceName: nodejs-service
        servicePort: 80
```