

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

**DIPLOMSKI RAD**

**SKALABILNE APLIKACIJE TEMELJENE NA  
KUBERNETES SUSTAVU**

**Mario Mileta**

**Split, rujan 2021.**



SVEUČILIŠTE U SPLITU  
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I  
BRODOGRADNJE



Diplomski studij: **Računarstvo**  
Smjer/Usmjerenje: /  
Oznaka programa: 250  
Akademska godina: 2020./2021.

Ime i prezime: Mario Mileta  
Broj indeksa: 758-2018

## **ZADATAK DIPLOMSKOG RADA**

Naslov: **Skalabilne aplikacije temeljene na Kubernetes sustavu**

Zadatak: Izučiti principe dizajna i izrade skalabilnih web aplikacija. Izučiti Kubernetes sustav, principe na kojima se temelji te mogućnosti i ograničenja sustava s posebnim osvrtom na prednosti koje uvodi pri skaliranju aplikacija. Na primjeru jednostavne web aplikacije demonstrirati prednosti Kubernetes sustava.

Rad predan:

Predsjednik

Odbora za diplomski rad:

Mentor:

prof. dr.sc. Sven Gotovac

izv. prof. dr. sc. Ljiljana Šerić



SVEUČILIŠTE U SPLITU  
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I  
BRODOGRADNJE



Diplomski studij: **Računarstvo**

Oznaka programa: 250

Ime i prezime: **MARIO MILETA**

Broj indeksa: 725-2018

## **PRIJAVA DIPLOMSKOG RADA**

Radni naslov: Skalabilne aplikacije temeljene na Kubernetes sustavu

Zadatak: Izučiti principe dizajna i izrade skalabilnih web aplikacija. Izučiti Kubernetes sustav, principe na kojima se temelji te mogućnosti i ograničenja sustava s posebnim osvrtom na prednosti koje uvodi pri skaliranju aplikacija. Na primjeru jednostavne web aplikacije demonstrirati prednosti Kubernetes sustava.

Mentor: izv. prof. dr. sc. Ljiljana Šerić

Predsjednik Odbora: Prof. dr. sc. Sven Gotovac

Split, 25.2.2021.

## **IZJAVA**

Ovom izjavom potvrđujem da sam diplomski rad s naslovom Skalabilne aplikacije temeljene na Kubernetes sustavu pod mentorstvom izv. prof. dr. sc. Ljiljana Šerić 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 u diplomskom radu citirao sam i povezao s korištenim bibliografskim jedinicama.

Student

Mario Mileta

# SADRŽAJ

1	UVOD .....	1
2	SKALABILNOST I VIRTUALIZACIJA .....	2
2.1	Horizontalno i vertikalno skaliranje .....	2
2.1.1	Vertikalno skaliranje .....	2
2.1.2	Horizontalno skaliranje .....	2
2.2	Monolitna i mikro-servis arhitektura.....	3
2.3	Virtualne mašine i kontejneri .....	4
2.3.1	Virtualna mašina .....	4
2.3.2	Kontejnerizacija .....	6
2.3.3	Usporedba kontejnera i virtualnih mašina .....	7
2.3.4	Docker .....	8
2.4	Mikroservisi, docker i Kubernetes .....	10
2.5	Kubernetes sustav .....	11
2.5.1	Arhitektura Kubernetesa .....	11
2.5.2	Kubernetes klaster.....	12
2.5.3	Pokretanje aplikacije unutar Kubernetesa.....	14
2.5.4	Zašto Kubernetes?.....	16
2.6	Glavni dijelovi Kubernetes sustava.....	17
2.6.1	Pod .....	17
2.6.2	Deployments .....	24
2.6.3	Services .....	26
2.6.4	Volumes .....	32
2.6.5	Secrets .....	38
3	DIZAJN I IMPLEMENTACIJA APLIKACIJE .....	40

3.1	Opis aplikacije.....	40
3.2	Jedan mikroservis – jedna baza podataka .....	41
3.2.1	Problemi prilikom upravljanja podataka kod mikro-servisa.....	41
3.3	Komunikacija između servisa .....	44
3.4	Arhitektura aplikacije.....	45
3.5	Spremanje komponentata u kontejnere .....	46
3.6	Skaffold .....	48
4	MIGRACIJA APLIKACIJE NA KUBERNETES KLASITER .....	50
4.1	Uspostavljanje klastera.....	50
4.2	Arhitektura aplikacije unutar klastera .....	52
4.3	Migracija aplikacije na klaster .....	56
4.4	Automatsko skaliranje.....	68
5	ZAKLJUČAK.....	72
	LITERATURA .....	73
	POPIS OZNAKA I KRATICA.....	74
	SAŽETAK .....	75
	SUMMARY .....	76

# 1 UVOD

Razvoj aplikacija u današnje vrijeme, osim zahtjeva za ispravnnošću, zahtjeva još i cijeli niz optimizacija. Kod razvoja software-a postoji više dimenzija u kojima možemo promatrati optimalnost, no neke od bitnih značajki optimalnog softvera bile bi da se aplikacija može jednostavno skalirati te je kod dobro organiziran. Takvu aplikaciju lakše je održavati te se novi developeri mogu brzo priključiti razvojnom timu. Nastoji se olakšati posao developerima ali također i sistem administratorima koji su zaduženi za smještanje (eng. deploy) aplikacije na server, praćenje da se sve izvršava u redu te održavanje sigurnosti aplikacije. Pristup razvoju aplikacije dosta ovisi o svrsi same aplikacije. Jedan pristup koji odgovara pojedinoj aplikaciji ne mora nužno i odgovarati nekoj drugoj.

U početku su se aplikacije razvijale kao monolitne. Takve aplikacije su bile sastavljene kao jedan veliki proces koji bi s vremenom toliko narastao da bi otežao daljnje održavanje i razvoj. Sistem bi imao spore razvojne cikluse te prilikom bilo koje promjene, koliko god ona mala bila, bi se cijeli sustav morao ponovno migrirati (eng. redeploy).

U današnje vrijeme težimo razvoju aplikacija koristeći mikro servise te mikro-servis arhitekturu. Kod takve arhitekture aplikacija je sastavljena od mnogo malih procesa (mikro servisa) gdje je svaki zadužen za određenu ulogu u sustavu te je kompletno neovisan o ostalim mikro servisima koji postoje unutar sustava. Jedina njihova interakcija leži u međusobnoj komunikaciji koristeći definirane API-je. Svaki servis se može samostalno razvijati i održavati te promjene na jednom servisu ne utječu na ostale. To bi značilo da prilikom male promjene ne moramo ponovno migrirati cijelu aplikaciju već samo njen dio.

Za najbolje rezultate prilikom migracije mikroservisa koristimo se određenom kontejner (container) tehnologijom kao npr. Docker. Pomoću nje mikro servisi se izoliraju u Linux kontejnere te ih vrtimo na izabranoj mašini koja ima određeno okruženje (eng. environment). Nakon što bi sve servise složili u zasebne kontejnere, migrirali bi ih kroz Kubernetes sustav. Kubernetes sustav bi tada samostalno motrio te održavao našu aplikaciju zdravom. Detalji te potrebni koraci koje moramo napraviti će biti opisani u okviru rada.

## **2 SKALABILNOST I VIRTUALIZACIJA**

### **2.1 Horizontalno i vertikalno skaliranje**

Skalabilnost u razvoju softvera odnosi se na sposobnost sustava da se prilagodi većim zahtjevima koje sustav ili aplikacija u određenom vremenu treba moći podnijeti, bilo u smislu performansi, održavanja, dostupnosti i slično. Pitanje je li sustav skalabilan, također se može postaviti kao "hoće li izdržati test vremena" ili će porastom korisnika doći do problema u korištenju sustava [1]. Često se može dogoditi da je pojedine aplikacije nemoguće skalirati. To svakako predstavlja problem te bi odmah u početku razvoja aplikacije bilo dobro razmisliti o načinima skaliranja. Postoje dva tipa skaliranja:

- Vertikalno skaliranje
- Horizontalno skaliranje

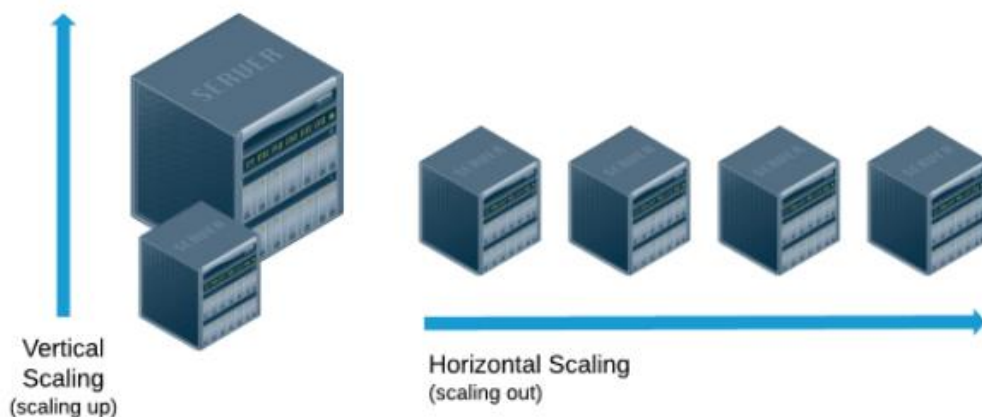
#### **2.1.1 Vertikalno skaliranje**

Vertikalno skaliranje uključuje poboljšanje servera dodavanjem određene procesorske snage, povećavanjem memorije ili migriranje aplikacije na novi i jači server koji odgovara novim zahtjevima sustava. Problem ovog načina skaliranja je ograničenost. S vremenom trošak hardvera odnosno potrebnih resursa za nastavak vertikalnog skaliranja postane prevelik a rezultati koje dobijemo povećanjem resursa su neučinkoviti. Trošak skaliranja može postati toliki da se na kraju ne isplati nastaviti skalirati sustav vertikalno. Vidimo da kod ovakvog pristupa postoji određeno ograničenje.

#### **2.1.2 Horizontalno skaliranje**

Kod horizontalnog skaliranja, umjesto povećavanja računalne snage servera, skaliramo tako što dodajemo više servera dok ne ispunimo zahtjeve sustava ili aplikacije. Na taj način imamo više instanci iste aplikacije te će se nadolazeće opterećenje raspodijeliti između njih.





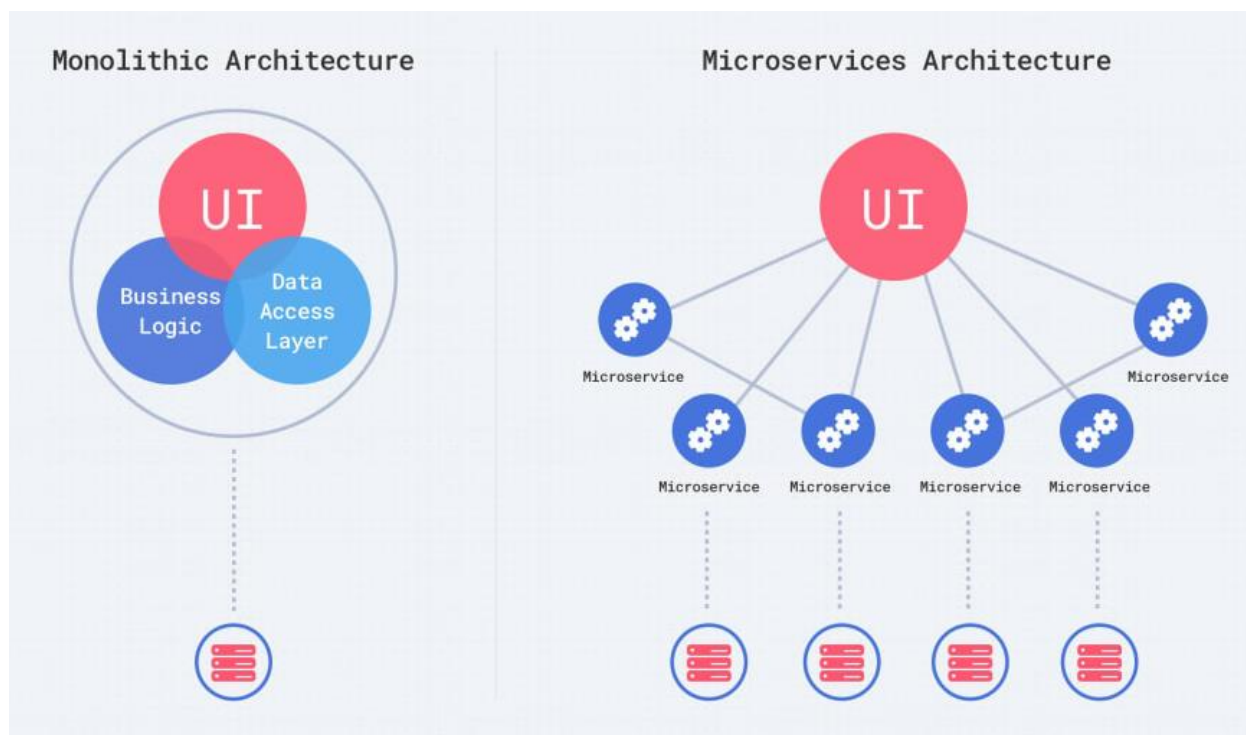
*Slika 2.1. Vertikalno i horizontalno skaliranje [2]*

## **2.2 Monolitna i mikro-servis arhitektura**

Razvoj aplikacija prateći monolitnu arhitekturu je bio tradicionalni pristup radu. Monolitna aplikacija je sagrađena kao jedna velika cjelina. Kod web aplikacije to bi značilo da su frontend, backend i baza podataka usko povezani i čine spomenutu cjelinu. Sve su funkcionalnosti na jednom mjestu te kodna baza (eng. code base) postaje iznimno velik. Ako želimo promijeniti neku sitnicu, to će utjecati na cijeli sustav koji će se onda ponovno morati testirati te sve skupa kasnije migrirati na produkcijski server. Takve aplikacije se najčešće mogu jednostavno vertikalno skalirati, no kako smo naveli prije, to ima svoja ograničenja [6]. Također, ako samo jedan dio monolitne aplikacije nije skalabilan, ona čitava postaje ne skalabilna. Tada se okrećemo drugim rješenjima odnosno mikro-servis arhitekturi.

Kod mikro-servis arhitekture naša aplikacija, umjesto da je sagrađena kao cjelina, razbijena je u mnogo servisa. Servisi su neovisni jedan o drugom te komuniciraju preko definiranih API-ja. Tada je svaki servis moguće zasebno horizontalno skalirati odnosno dodati više instanci tog servisa. Na taj način, umjesto da skaliramo cijelu aplikaciju zbog jednog malog dijela koji nam zahtjeva više resursa, možemo skalirati samo jedan njen dio. S obzirom da su mikro servisi međusobno neovisni,

svaki tim koji radi na pojedinom mikro servisu može birati u kojoj tehnologiji žele raditi te izabrati onu koja najbolje pristaje zahtjevima servisa. Također promjenom na nekom od servisa nije potrebno testirati cijelu aplikaciju ispočetka već samo servis na kojem smo radili promjene. Time uštedujemo i vrijeme i novac.



*Slika 2.2. Monolitna i mikroservis arhitektura [4]*

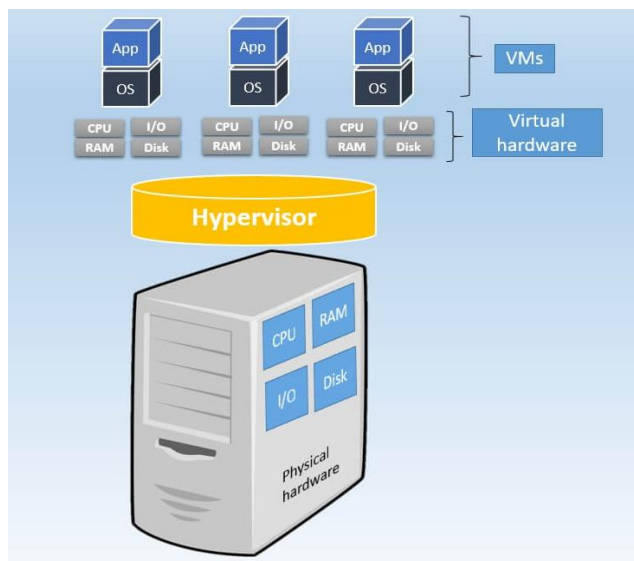
## **2.3 Virtualne mašine i kontejneri**

### **2.3.1 Virtualna mašina**

U početku glavni način i pristup izoliranju određenih komponenti bio je korištenje virtualizacije i virtualnih mašina. Virtualizacijom postizemo kompletnu izolaciju resursa kojima raspolažemo te ih na taj način možemo raspodijeliti i grupirati u određene komponente odnosno kreirati virtualne mašine [6]. Virtualna mašina (skraćeno VM) može svojevoljno raspolagati resursima kojima dobije dozvolu od svoje 'host' mašine. Kada kažemo host mašina, mislimo na fizički hardver kojim

raspolaze npr. naš laptop. Tada na tom laptopu možemo kreirati više VM-ova i svakom VM-u dodijeliti potrebne količine CPU-a, memorije, odnosno svega potrebnog da bi VM ispravno funkcionirao i bio u mogućnosti ispuniti tražene zadaće. S obzirom da su resursi dodijeljeni VM-u izolirani od ostatka, možemo instalirati i odgovarajući operacijski sustav te zapravo se može reći da dobijamo novi laptop unutar laptopa nad kojim vršimo virtualizaciju. Samo što laptop nije pravi već virtualni te ga nazivamo virtualnom mašinom. Tako možemo kreirati i imati više VM-ova na jednom laptopu. Sve ovisi o hardveru s kojim početno raspolazemo. Kada vrtimo npr. tri virtualne masine na našem računalu, imamo zapravo tri različita operacijska sustava koja dijele isti hardware. Razdvajanje fizičkih resursa u više virtualnih resursa koji onda mogu biti korišteni od strane određenog VM-a dobijamo pomoću hypervisora. Hypervisor je zapravo software koji se može vrtiti na našem operacijskom sustavu te služi za kreiranje i pokretanje VM-ova te postoje dva tipa:

- Tip 1 – koji se ponaša kao operacijski sustav i vrti se direktno na ‘host’ hardware-u
- Tip 2 – koji se vrti kao software na ‘host’ operacijskom sustavu kao i neki drugi programi



*Slika 2.1. Virtualna mašina [5]*

Na slici možemo vidjeti pojednostavljeni prikaz funkcioniranja virtualnih mašina. Uz pomoć navedene tehnologije, aplikaciju koju razvijamo moguće je razdvojiti u više izoliranih komponenti te svakoj komponenti dodijeliti zasebnu VM. Međutim kada imamo veći broj manjih komponenti, dodjeljivanjem VM-a svakoj komponenti bi predstavljalo gubitak resursa i smanjenje efikasnosti. To proizlazi iz razloga što svaka VM ima vlastiti operacijski sustav koji također zahtjeva njeno upravljanje što bi značilo određene ljudske resurse. Tada se okrećemo kontejner tehnologijama i kontejnerizaciji (eng. kontejnerization).

### 2.3.2 Kontejnerizacija

S vremenom umjesto korištenja virtualnih mašina, sve češće se koriste kontejner tehnologije za izolaciju pojedinih servisa. Uz pomoć Linux kontejner tehnologije moguće je više servisa vrtiti na istoj mašini, međusobno ih izolirati te im omogućiti drukčija okruženja. Slično kao i kod VM-ova samo s manjim troškom resursa. Utrošak resursa je smanjen jer kod kontejner tehnologije ne dodjeljujemo svakom malom servisu vlastiti operacijski sustav već postoji samo jedan – host OS. To je moguće uz dva glavna mehanizma:

- Linux Namespaces – omogućava da svaki process ima zasebni pogled na sistem što uključuje pojedine datoteke, procese koji se vrte, mrežu, hostname-ove itd...
- Linux Control Groups (cgroups) – koji limitira upotrebu resursa koju pojedini process može konzumirati a to uključuje CPU, memoriju itd...

#### 2.3.2.1 Linux Namespaces

Po defaultu Linux sistem ima jedan namespace te svi resursi pripadaju njemu. Međutim mi možemo kreirati dodatne namespace-ove te resurse raspodijeliti po potrebi. Tada proces koji se vrti unutar nekog od kreiranih namespace-ova vidi samo resurse koji su unutar istog namespacea [6]. Postoji više vrsta namespace-ova te proces zapravo ne pripada samo jednom namespace-u već određenoj grupi. Postoje:

- Mount (mnt)
- Process ID (pid)

- Network (net)
- Inter-process communication (ipc)
- UTS
- User ID (user)

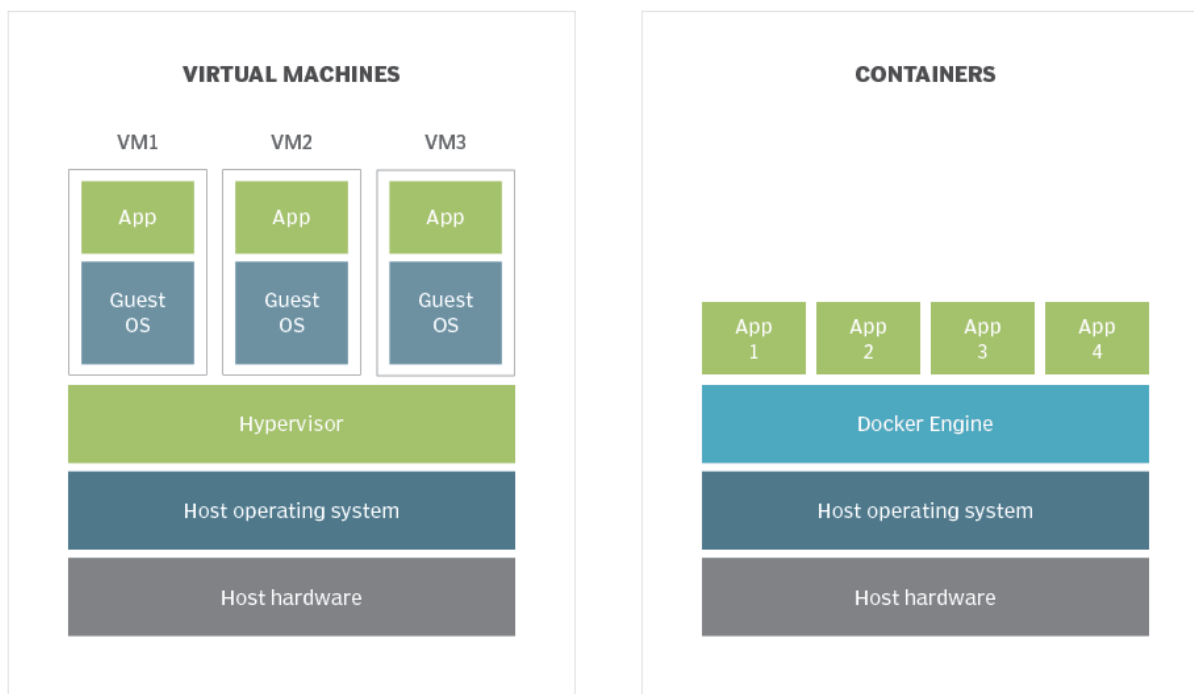
Svaki namespace služi za izolaciju pojedine grupe resursa koji su onda neovisni jedni o drugima.

#### 2.3.2.2 Linux Control Groups (cgroups)

Linux cgroups nam služe za limitiranje količine sistemskih resursa koje pojedini kontejner može koristiti. Na taj način proces ne može koristiti više od zadane količine CPU-a, memorije, propusnosti mreže i slično. Procesi nisu u stanju koristiti resurse koji su namijenjeni odnosno rezervirani za druge procese makar se oni vrtili na istoj mašini.

#### 2.3.3 Usporedba kontejnera i virtualnih mašina

Kao što smo naveli, krajnji cilj kontejnera i virtualnih mašina je isti – kompletno izolirati procese koji se vrte na određenom hardveru. Virtualne mašine su i danas u velikoj upotrebi međutim ovaj rad će biti fokusiran na kontejnere. Razlog tome je što svaka virtualna mašina zahtjeva svoj vlastiti operacijski sustav (na slici Guest OS) što je jako neisplativo. Ne želimo našim mikro-servisima, koji će ponekad biti zaduženi za jednu jednostavnu stvar, instalirati cijeli operacijski sustav. Za razliku od virtualnih mašina, kontejneri ne trebaju vlastiti OS kako bi uspješno izolirali pojedine procese. Time imamo manje troškove te su prikladniji za korištenje u slučaju naše aplikacije.



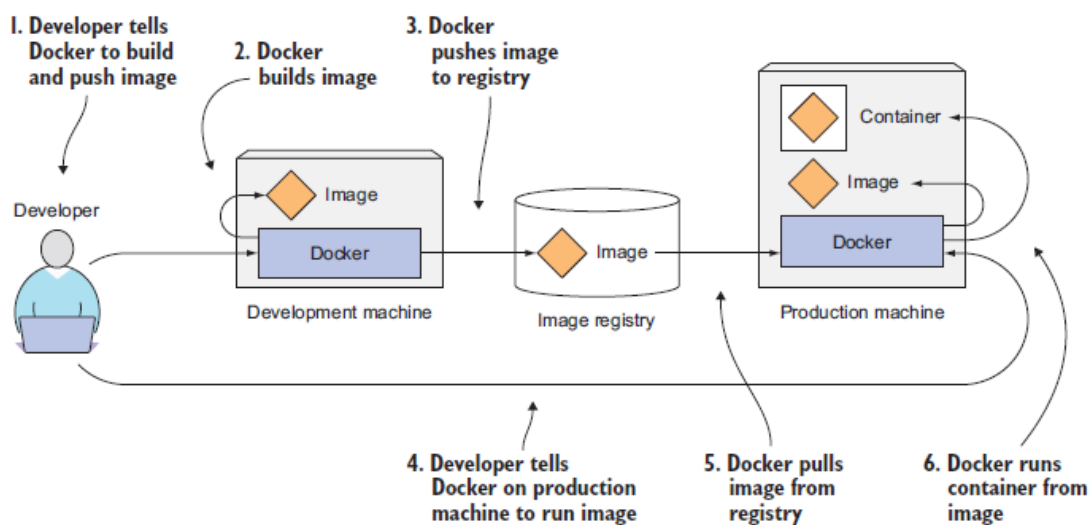
*Slika 2.2. Usporedba virtualne mašine i kontejnera [3]*

#### 2.3.4 Docker

Docker [6] je jedna od najpoznatijih kontejner tehnologija. Omogućuje jednostavno zapakiranje servisa ili biblioteka u kontejnere koji se jednostavno mogu pokretati na bilo kojem drugom računalu koje ima instaliran Docker. Prilikom razvijanja software često se zna dogoditi da određeni software ne radi ili se drukčije ponaša ovisno na kojoj mašini je pokrenut. Docker rješava taj problem. Kada zapakiramo aplikaciju pomoću Dockera, neovisno na kojoj mašini pokrećemo aplikaciju, ona vidi isti datotečni sustav kao i prije pakiranja, naravno, pod uvjetom da mašina ima instaliran Docker. Također operacijski sustav nije bitan, tako da možemo razvijati aplikaciju na OS-u koji je različit od onog koji je instaliran produkcijskoj mašini.

Docker također omogućava prebacivanje zapakiranih paketa na centralni repozitorij iz kojeg se onda paketi mogu povući i izvršavati na bilo kojoj mašini koja ima instaliran Docker. Imamo tri glavna koncepta uključena u taj proces:

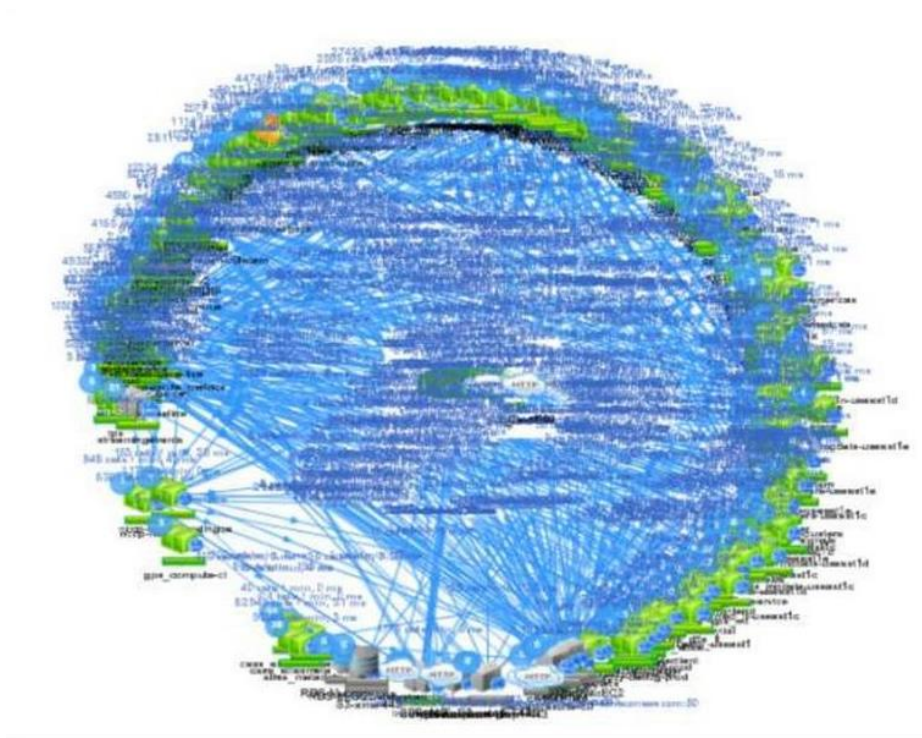
- Docker slike (Docker images) – ‘kontejner image’ sadrži (snapshot) nasu aplikaciju i informacije o okruženju. Također sadrži i cijeli datotečni sustav koji će aplikacija moći koristiti te ima informacije kako pokreniti izvršnu (exe) datoteku
- Registri – ‘Docker registry’ je repozitorij na kojem se nalaze Docker slike. Pomoću njega imamo jednostavan način za brzo dijeljenje Docker slika među ljudima i računalima. Kada napravimo sliku naše biblioteke ili aplikacije, možemo je pokrenuti na našem računalu ili uploadati na Docker registrar. Kasnije biblioteku možemo skinuti sa Docker registra na neko drugo računalo te od tamo pokrenuti. Postoje privatni i javni registri. Sa javnih registra svak može preuzeti Docker sliku i na svom računalu imati potreban zapakirani servis koji je spreman na korištenje
- Kontejners – bilo koji zapakirani program koji mozemo kreirati te pokrenuti uz pomoć Docker image-a. Pokrenuti kontejneri su procesi koji se vrte na host mašini koja vrti Docker, ali je kompletno izolirana od svih drugih pokrenutih procesa. Proces također ima limitirane resurse koje može koristiti, odnosno može pristupiti samo resursima koje smo mu prethodno alocirali.



Slika 2.3. Prikaz kreiranja Docker slike i pokrećanja Docker kontejnera [6]

## 2.4 Mikroservisi, docker i Kubernetes

Da bismo razumijeli zbog čega nam je potreban Kubernetes i zašto je nastao, morali smo se prvo upoznati sa mikroservisima i kontejnerizacijom. Kako su aplikacije u novije vrijeme sve veće te njihov broj mikroservisa znatno raste, nastaje problem pri održavanju tolikog broja komponenti. Najbolji primjer velikog broja mikroservisa koje treba održavati je kompanija Netflix.



*Slika 2.4. Netflix mikro servisi [8]*

Danas Netflix ima preko tisuću mikroservisa te možemo vidjeti na slici kako to sve zna zbunjujuće izgledati. Kada bi ovoliku količinu komponenti određeni broj timova manualno morao migrirati i održavati, svakako bi naišli na velik broj problema. Također osigurati da se servisi i nakon migriranja ponašaju kao jedna velika cjelina nije jednostavno. Tada se okrećemo drugim opcijama i rješenjima problema te se upoznajemo sa Kubernetesom.

Kubernetes [7] je softverski sustav koji nam omogućava jednostavno migriranje i održavanje velikog broja aplikacija ili servisa koje smo prethodno kontejnerizirali. Kako se servisi vrte u

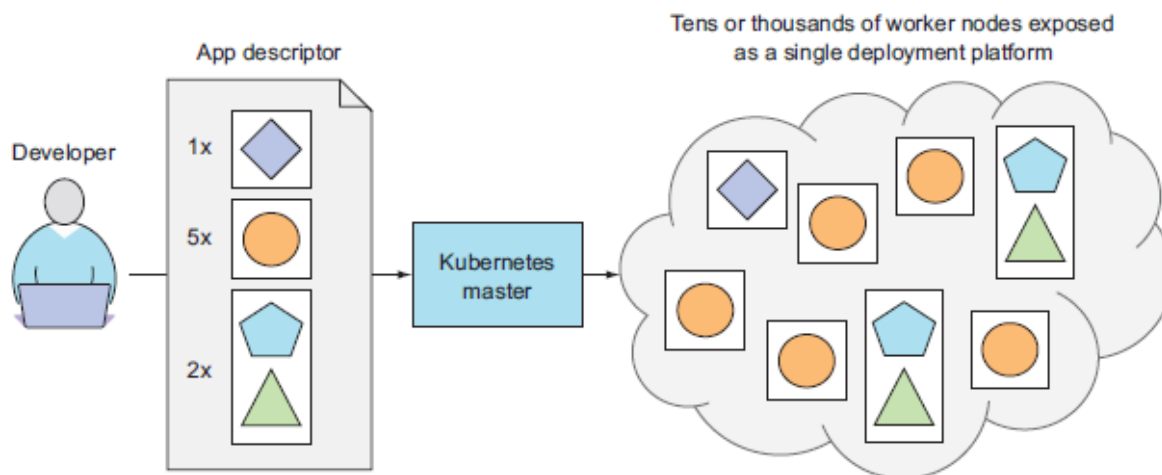


kontejneru, ne utječu međusobno jedni na druge te se mogu migrirati na isti server. Tako možemo mikro-servise i od drugih kompanija staviti na isti server bez ikakvih problema, čime cloud provideri mogu najbolje iskoristiti hardver kojim raspolaze. Migriranje aplikacije pomoću Kubernetesa je uvijek isto. Nije bitno da li nam se naš Kubernetes klaster sastoji od tisuća čvorova ili samo jednog. Veličina klastera ne predstavlja razliku tokom migracije. Više čvorova predstavlja samo povećan broj resursa koji će biti dostupni aplikaciji. Princip ostaje isti. Također koristeći Kubernetes dodajemo jedan dio abstrakcije infrastrukturi čime pojednostavljujemo razvoj, migriranje i održavanje aplikacije, kako za sistem administratore, tako i za developere. Zbog abstrakcije postoji određeni 'learning curve' međutim benefiti koje kasnije dobijemo svakako se isplate.

## **2.5 Kubernetes sustav**

### **2.5.1 Arhitektura Kubernetesa**

Kubernetes sustav [6] se sastoji od glavnog (master) čvora i proizvoljnog broja radnih čvorova. Developer ili sistem administrator je u komunikaciji sa glavnim čvorom kojem predaje svoj opis aplikacije odnosno listu servisa koje je potrebno migrirati. Tada Kubernetes sam vrši migraciju servisa na klaster koji je sačinjen od određenog broja radnih čvorova. Na posljatku developeru nije ni bitno na kojem čvoru završi pojedini servis. Također možemo specificirati da određeni servisi trebaju biti skupa pokrenuti i Kubernetes će njih deployati na isti radni čvor. Opet, developeru nije bitno na koji. S obzirom da nije bitno na kojem čvoru se vrti pojedini servis, Kubernetes je u mogućnosti premještati određene servise i međusobno ih kombinirati na pojedine čvorove čime može postići puno bolju iskoristivost resursa nego što bi to bilo moguće kod manualnog deployanja.



*Slika 2.5. Pojednostavljeni prikaz Kubernetes arhitekture [6]*

### 2.5.2 Kubernetes klaster

Kubernetes klaster sastoji se od više čvorova koje možemo podijeliti u dva tipa:

- Glavni čvor - sadrži Kubernetes Control Plane koji kontrolira i upravlja cijelim Kubernetes sustavom
- Radni čvorovi – čvorovi na kojima se pokreće aplikacija koju migriramo

#### 2.5.2.1 Kubernetes Control Plane

Control Plane je u biti ono što kontrolira cijeli klaster i čini ga funkcionalnim. Sastoji se od par komponenata koje se mogu vrtiti na jednom glavnom čvoru ili biti raspoređene među više čvorova. Moguće ih je i duplicirati kako bi osigurali visoku dostupnost. Te komponente su:

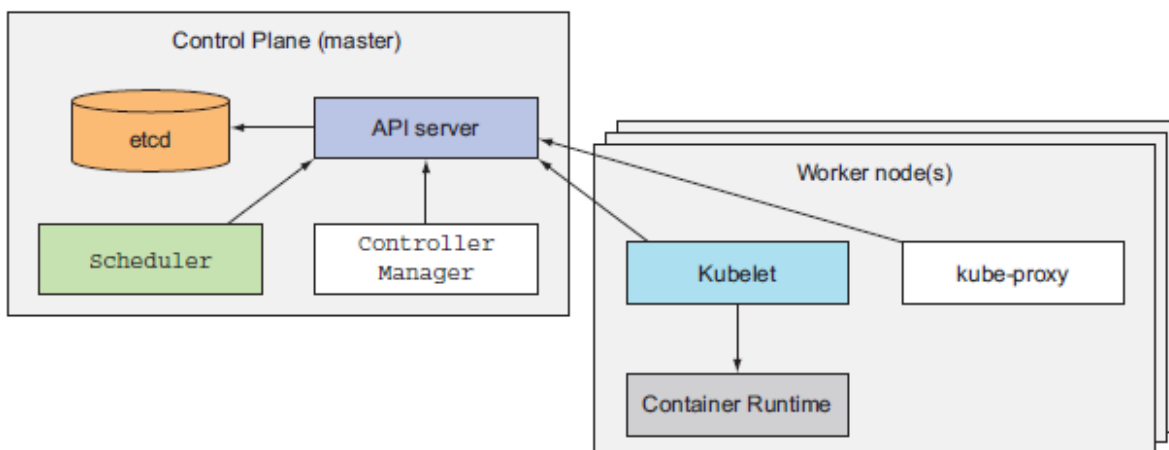
- Kubernetes API Server je komponenta preko koje komuniciramo sa Kubernetes API-jem i drugim komponentama Control Plane-a
- The Scheduler (raspoređivač) ima za zadatak rasporediti naše migrirane komponente. Zadužen je za dodjeljivanje radnih čvorova servisima

- The Controller Manager (kontroler) čija je odgovornost na klaster levelu odnosno zadužen je za repliciranje komponenti, motrenje radnih čvorova te poduzimanju određenih koraka u slučaju greške
- etcd koji predstavlja bazu podataka koja pouzdano sprema konfiguraciju klastera

### 2.5.2.2 Radni Čvorovi

Radni čvorovi predstavljaju mašine na kojima se vrti naša aplikacija ili servis koji smo prethodno smjestili u kontejner. Pokretanje, motrenje te pružanje potrebnih servisa našoj aplikaciji imaju za zadatak slijedeće komponente:

- Docker, rkt za kontejnerizaciju koji vrti našu kontejneriziranu aplikaciju
- Kubelet koji je u komunikaciji sa API serverom. Nalazi se na svakom čvoru te osigurava da su kontejneri pokrenuti.
- Kube-proxy koji raspoređuje promet odnosno opterećenje među komponentama pridržavajući se određenih mrežnih pravila

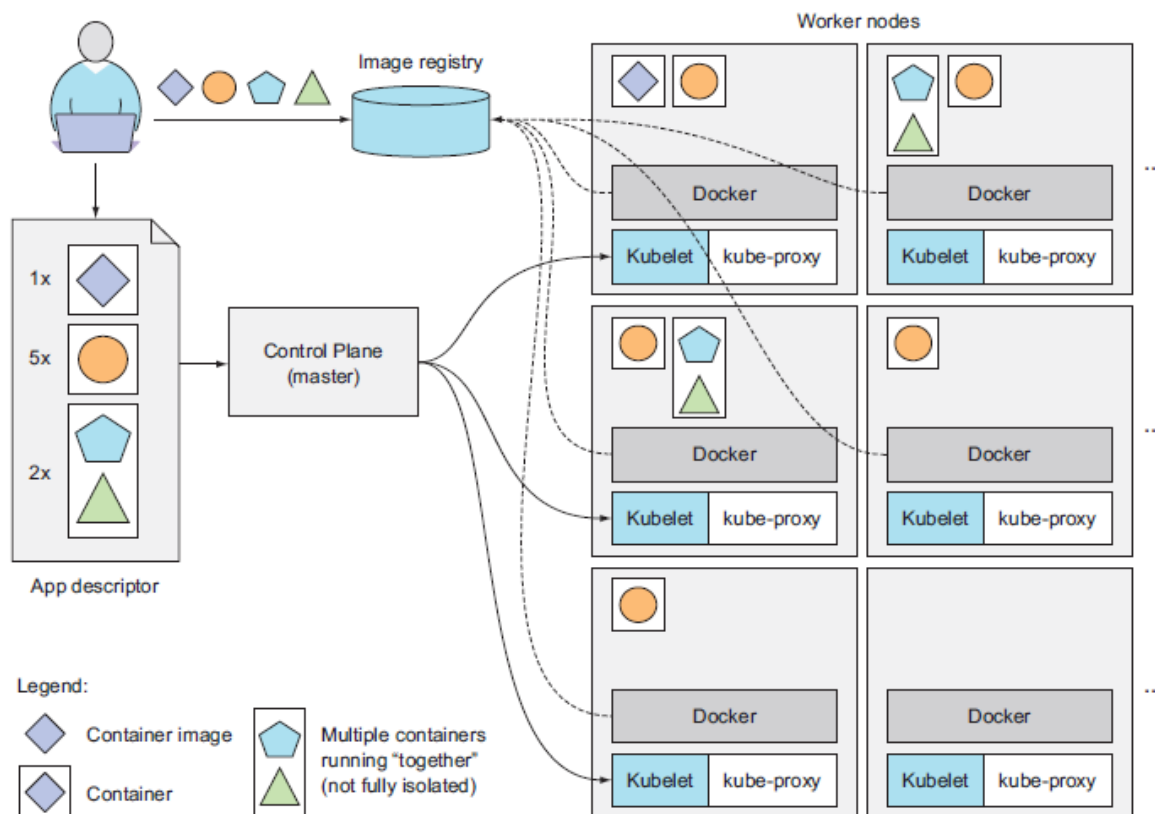


*Slika 2.6. Komponente glavnog i radnih čvorova [6]*

### 2.5.3 Pokretanje aplikacije unutar Kuberbetesa

Ukratko za migrirati aplikaciju pomoću Kuberbetesa, prvo moramo njene komponente zapakirati u kontejner te napraviti Docker sliku koju ćemo staviti na određeni registar slika. Nakon toga ćemo opisati našu aplikaciju Kuberbetes API serveru. Opis nam mora uključivati o pojedinim Docker slikama od kojih se naša aplikacija sastoji. Svaka slika odgovara jednoj komponenti. Također moramo reći Kuberbetesu kako su te komponente povezane te moraju li se određene komponente migrirati na isti čvor. Za svaku pojedinu komponentu možemo izabrati koliki broj kopija želimo imati.

Nakon komunikacije sa API Serverom, raspoređivač (scheduler) raspoređuje kontejnere na dostupne radne čvorove ovisno o potrebnim resursima koje zahtjevaju. Tada Kubelet koji se nalazi na svakom čvoru naredi pokretanje kontejnera što uključuje povlačenje potrebnih slika iz registra i startanje samog kontejnera.



Slika 2.7. Prikaz pokretanja Docker slika unutar Kubernetes radnih čvorova [6]

Na slici možemo vidjeti prikaz cijelog procesa. Opis aplikacije sastoji se od četiri kontejnera koji su grupirani u tri 'Pod'-a. Termin Pod ćemo objasniti kasnije. Prva dva Pod-a sadrže po jedan kontejner dok zadnji ima dva. To znači da oba kontejnera moraju biti pokrenuta zajedno te nebi trebali biti izolirani jedan od drugoga. Također vidimo broj potrebnih kopija svakog 'Pod'-a. Nakon što Kubernetes primi opis, rasporedit će broj kopija svakog 'Pod'-a na dostupne radne čvorove. Kubelet će onda obavjestiti Docker da dohvati slike iz registra te pokrene kontejnere.

#### 2.5.3.1 Održavanje kontejnera

Kada se aplikacija uspješno pokrene, Kubernetes neprekidno provjerava da aplikacija odgovara svim zahtjevima koju su bili prethodno opisani. Tako ako smo specificirali da želimo dvije kopije nekog servisa, Kubernetes će se pobrinuti da to i bude tako. Ako jedna od njih prestane raditi,

automatski će biti ponovno pokrenuta. Također ako cijeli čvor prestane funkcionirati, Kubernetes će servise koji su bili pokrenuti na tom čvoru prebaciti na novi, zdravi čvor i tamo ih pokrenuti.

Po potrebi, Kubernetes može automatski povećavati i smanjivati broj kopija koje imamo. Odluku o tome donosi na temelju trenutne opterećenosti naših postojećih servisa poput iskorištenosti memorije, upita po sekundi, opterećenju procesora. Na taj način Kubernetes sam prilagođava broj kopija određenog servisa tako da u svakom trenutku imamo najbolju moguću iskoristivost resursa uz optimalno opterećenje.

U slučaju potrebe, Kubernetes je u mogućnosti premještati naše servise sa jednog čvora na drugi. Daljnji pronalazak servisa koji je premješten ne predstavlja problem jer će Kubernetes kontejnerima koji pružaju iste usluge dodijeliti jednu statičnu IP adresu te će ti servisi preko nje biti dostupni u cijelom klasteru. To je moguće uz pomoć env varijabli. Tada će se kube-proxy pobrinuti da zahtjevi prema tom servisu odnosno IP adresi budu podjednako raspoređeni među pokrenutim kontejnerima.

#### 2.5.4 Zašto Kubernetes?

Ako koristimo Kubernetes, sistem administratorima je posao dosta olakšan. Kako aplikacija pokrenuta u kontejneru već sadrži sve što joj je potrebno za ispravno funkcioniranje, sistem administratori ne moraju nista dodatno instalirati prilikom migriranja i pokretanja same aplikacije. Na bilo kojem čvoru gdje imamo postavljen Kubernetes, on može sam pokrenuti cijelu aplikaciju bez pomoći sistem administratora.

##### 2.5.4.1 Olakšava razvoj

Više nemoramo biti upoznati sa serverima koji čine naš klaster. Svi radne čvorove koji postoje možemo promatrati kao jednu zajedničku cjelinu koja sadrži potrebne resurse za našu aplikaciju. Developere najčešće ne zanima na kojem serveru se aplikacija vrti sve dok ima potrebne resurse, a to Kubernetes i omogućava.

Ponekad postoje situacije gdje nam je bitno na kakvom je serveru i kojem je hardveru pokrenuta naša aplikacija. Međutim za razliku od prije gdje su sistem administratori morali manualno birati čvorove koji odgovaraju zahtjevima aplikacije, sada ćemo samo spomenute zahtjeve proslijediti

Kubernetesu. Kubernetes sustav će biti taj koji će se pobrinuti da nam aplikacija bude na čvoru koji ispunjava sve zahtjeve.

#### 2.5.4.2 Bolja iskoristivost

Koristeći Kubernetes, dajemo njemu mogućnost izbora optimalnog čvora za našu aplikaciju prema opisu kojeg mu mi pružimo. Opis sadrži potrebne resurse za našu aplikaciju. Uz pomoć kontejnera i mogućnosti prebacivanja aplikacije sa jednog čvora na drugi, Kubernetes može kombinirati razne servise na isti čvor te na taj način postići maksimalnu iskoristivost resursa kojim raspolaže.

#### 2.5.4.3 Pouzdanost

Kubernetes neprekidno motri sve čvorove na kojima se dijelovi naše aplikacije nalaze te automatski ih premješta na drugi čvor u slučaju greške. Time su sistem administrator oslobođeni manualnog migriranja pojedinih komponenti aplikacije i omogućuje im da rade na otklanjanju greške sa samog čvora. Također ako raspolažemo s dovoljno resursa, sistem admini ne moraju žurno krenuti s otklanjanjem greške na čvoru (ponekad i po noći), već mogu mirno spavati i čekati svoje radno vrijeme, što prije nije bio slučaj.

#### 2.5.4.4 Automatsko skaliranje

Kao što smo već rekli, koristeći Kubernetes sistem administratori ne moraju konstantno motriti čvorove jer to čini sam Kubernetes. To je također istina što se tiče nadolazećeg opterećenja prema određenim komponentama aplikacije. Porastom opterećenja Kubernetes je sam u mogućnosti pokrenuti još jednu kopiju određenog servisa kako bi bolje podnio nadolazeći promet. Isto tako, s padom prometa može ugasi tu kopiju i vratiti se na početni broj. Drugim rječima može samostalno skalirati cijelu ili pojedine dijelove aplikacije.

## 2.6 Glavni dijelovi Kubernetes sustava

U prošlom poglavlju smo objasnili glavne koncepte kubernetesa i dobili pregled rada Kubernetes sustava. Sad ćemo malo detaljnije objasniti tipove objekata (još se nazivaju i resursima) unutar Kubernetesa. Počnimo s ‘Pod’-om jer su svi ostali objekti na neki način povezani sa njim.

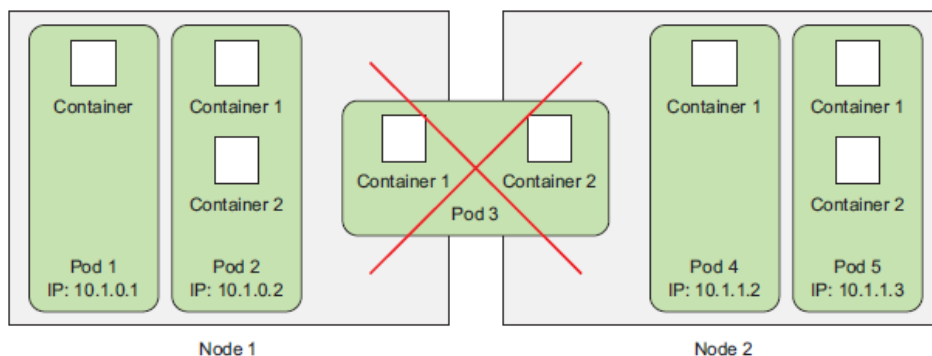
### 2.6.1 Pod

Termin ‘Pod’ smo prvi put spomenuli kad smo Kubernetesu slali opis naše aplikacije i kontejnera. Tada smo rekli da će Kubernetes pokrenuti naše kontejnere unutar Pod-a. Pod [6] nam služi za

grupaciju kontejnera te predstavlja najmanju migracijsku jedinicu Kubernetes sustava (deploy unit).

Možemo se zapitati zašto je uopće potrebno grupirati kontejnere. Zašto nemožemo jednostavno procese koji su povezani smjestiti u jedan kontejner. Iz razloga što su kontejneri su napravljeni s namjerom da se u njima vrti samo jedan izolirani process. Ako vrtimo više procesa unutar istog kontejnera, naša je zadaća te procese motriti, pratiti logove i slično. Također morali bi naći način za jednostavno ponovno pokretanje procesa unutar kontejnera ako dođe do greške i prestanka rada.

S obzirom da nije preporučljivo grupirati više procesa u jedan kontejner, trebamo način kako spojiti kontejnere kako bi se oni ponašali kao jedna cjelina. Tada nam pomažu Pod-ovi koji omogućuju da kontejnere koji sadrže međusobno povezane procese pokrenemo na jednom čvoru. Time im pružamo isto okruženje čime dobivamo privid kao da se vrte svi skupa unutar jednog kontejnera, a uspjeli smo ih međusobno izolirati.



*Slika 2.8. Kontejneri unutar Pod-a na radnom čvoru [6]*

Kako kontejneri koji se vrte u istom Pod-u dijele isti Network namespace (jedan od Linux namespace-ova) također dijele i istu IP adresu. To znači da moramo paziti da procesi koji se vrte u kontejnerima na istom Pod-u imaju različite port brojeve kako nebi došlo do konflikata.

Svi postojeći Pod-ovi unutar Kubernetes klastera nalaze se u zajedničkom mrežnom adresnom prostoru što znači da svaki pod može pristupiti IP adresi od drugog Pod-a. Time je komunikacija između Pod-ova pojednostavljena. Nije bitno na kojem se čvoru nalaze Pod-ovi jer mogu



međusobno komunicirati preko ‘flat NAT-less’ mreže, slično kao što i računala na istom LAN-u komuniciraju.

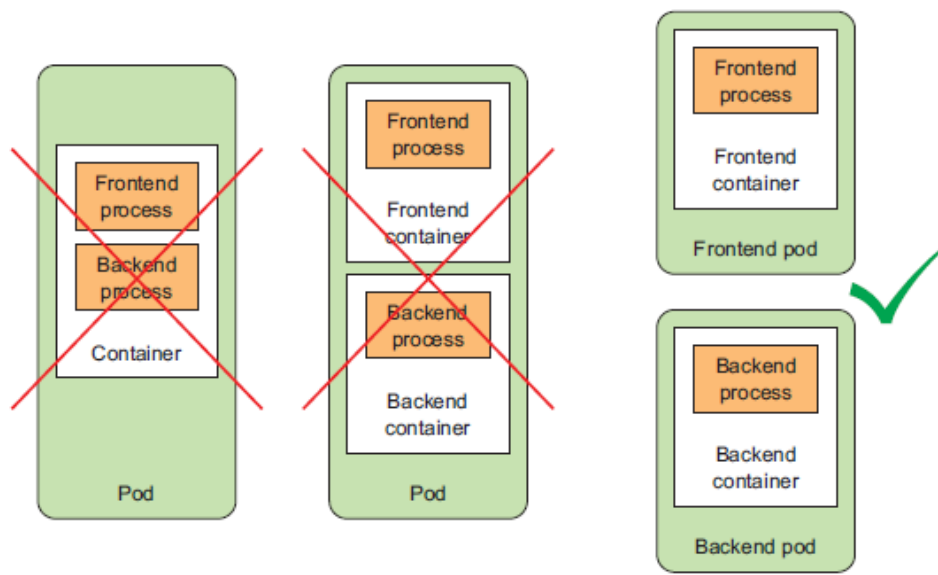
#### 2.6.1.1 Organiziranje Pod-ova

Pomaže ako Pod-ove promatramo kao odvojene mašine. S obzirom da ne zahtjevaju puno memorije možemo imati Pod-ova koliko želimo. Nakon što aplikaciju razdvojimo u zasebne servise nastojimo je također ograničiti u što više Pod-ova. Optimalno bi bilo kada bi svaki servis bio u zasebnom kontejneru i zasebnom Pod-u te stavljen u zajednički Pod samo kada je to potrebno. Uzmimo za primjer jednostavnu web aplikaciju. Svaka web aplikacija sastoji se od frontenda, backenda i baze podataka. Zbog više razloga bilo bi pogrešno sva tri dijela staviti u isti Pod. Najmanje što bi trebali napraviti je međusobno ih odvojiti kako bi Kubernetes mogao optimalno iskoristiti resurse kojima raspolazemo. Također problem dolazi kod skaliranja. Ako nam je baza podataka spora i ona predstavlja problem, ne trebaju nam dodatne kopije frontenda i backenda uz to čime bi došlo do nepotrebnog troška hardver resursa. Također svaki dio web aplikacije ima specifične zahtjeve skaliranja. Ako je poželjno da servis koji se vrti u kontejneru skalira zasebno, to je najčešće dobar indikator da pripada u jedan zasebni Pod.

Kod odlučivanja koje kontejnere staviti zajedno a koje odvojeno moramo razmotriti tri stvari:

- Moraju li kontejneri biti pokrenuti skupa ili se mogu nalaziti na različitim host-ovima
- Predstavljaju li jednu cjelinu?
- Moraju li biti skalirani zajedno?

Ako je odgovor na ova pitanja ne, tada bi bilo dobro kad bi kontejnere smjestili u različite Pod-ove. Generalno trebamo težiti tome da svakom kontejneru pripada zasebni Pod.



Slika 2.9. Ispravno organiziranje kontejnera i Pod-ova [6]

#### 2.6.1.2 Kreiranje Pod-ova

Pod-ove i ostale objekte koji pripadaju Kubernetes sustavu najčešće kreiramo uz pomoć JSON ili YAML datoteke. Nakon što napišemo YAML datoteku, objavu vršimo pomoću komande *kubectl apply -f pod.yaml*. Kasnije ćemo taj process automatizirati tako da nećemo morati pisati tu komandu repetitivno, već će to alat (Skaffold) činiti umjesto nas. Na taj način možemo svu konfiguraciju objekata možemo spremiti u određen sustav kontrole verzija poput Github-a te im kasnije jednostavno pristupiti. Objekte također možemo kreirati koristeći *kubectl* naredbu i komandnu liniju (*kubectl create*). Međutim u tom slučaju teško je pratiti koji su svi objekti kreirani i s kojom konfiguracijom te gubimo određene prednosti u odnosu na YAML datoteke.

Jednom kad kreiramo Pod ili bilo koji drugi objekt, Kubernetes će se pobrinuti ta kreirani objekt uvijek postoji unutar našeg klastera. Pod-ove zapravo u pravilu ne kreiramo direktno već kreiramo druge tipove resursa poput Deploymenta koji onda automatski kreiraju i održavaju Pod. Ako Pod kreiramo direktno i na čvoru gdje se nalazi dođe do greške, tada taj Pod neće biti automatski zamijenjen sa novim. Osim ako nije napravljen koristeći drugi resurs Kubernetes sustava poput Deploymenta. U slijedećem poglavlju ćemo malo detaljnije objasniti Deploymente.

```

apiVersion: apps/v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 8080
      protocol: TCP

```

*Slika 2.10. YAML datoteka za kreiranje Pod-a*

Na slici je prikazan jedan primjer YAML datoteke koja služi za kreiranje Pod-a. U YAML datoteci postoje dvije bitne cjeline koje možemo pronaći u svakoj YAML datoteci koja služi za kreiranje Kubernetes resursa:

- Metadata – uključuje ime, namespace, oznake i druge informacije vezane za sami Pod
- Spec – pobliže opisuje sadržaje Pod-a poput kontejnera i slike, ‘volumes’-a (koje ćemo također kasnije objasniti) i slično

U datoteci na slici govorimo Kubernetes API-ju kakav objekt nam je potreban. Želimo Pod s imenom nginx-pod. Unutar Pod-a ćemo imati jedan kontejner s imenom nginx te naziv slike preko koje ćemo ga dogvatiti. Također smo rekli da kontejner sluša na portu 8080.

### 2.6.1.3 Oznake Pod-ova

Kako aplikacija raste, tako raste i broj naših mikroservisa a samim time i potreban broj Pod-ova. Još kad uzmemo u obzir da ćemo imati više kopija Pod-ova istog mikroservisa i vrlo vjerojatno više verzija koje trenutno postoje, možemo zaključiti da ćemo vrlo brzo doći do velikog broja Pod-ova u našem klasteru. Radi jednostavnijeg snalaženja trebamo na neki način organizirati Pod-ove koji se nalaze u sustavu i to činimo koristeći oznake (labels).

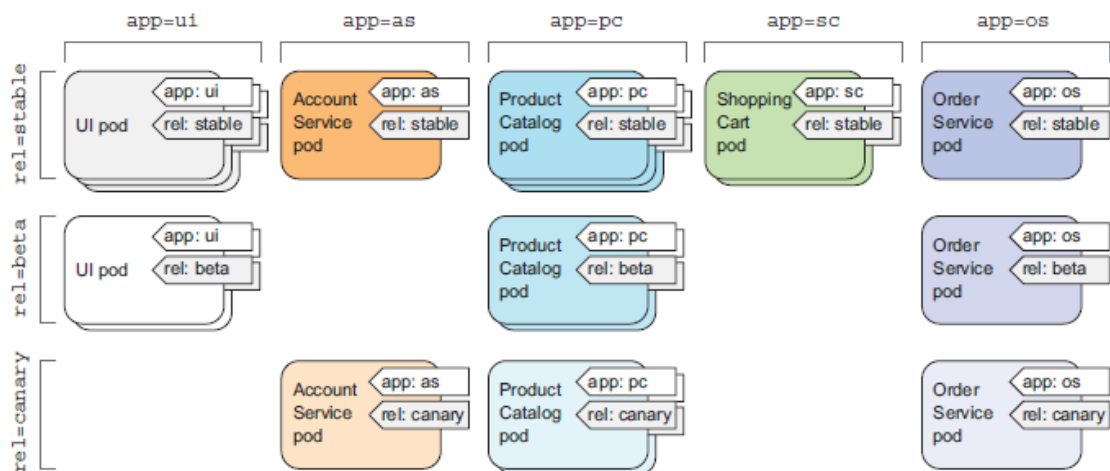
Oznake su dosta jednostavna a vrlo korisna svojstva Kubernetes sustava koja koristimo ne samo za označavanje Pod-ova već svih drugih resursa koji postoje unutar Kubernetesa. Oznaka predstavlja par ključ-vrijednost koju dodijelimo resursu kojeg onda kasnije možemo pronaći prema prethodno dodanoj oznaci. Određeni resurs može imati više oznaka koje mu najčešće dodjeljujemo prilikom kreiranja.

Dodavanjem npr. dvije oznake našim resursima možemo ih organizirati u dvije dimenzije – horizontalno i vertikalno. Na slici možemo vidjeti jednostavni YAML file gdje se kreira Kubernetes resurs koji će sadržavati dvije labele.

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: pod-with-two-labels
5    labels:
6      app: ui
7      rel: beta
8  spec:
9    containers:
10     - name: test-container
11       image: mmileta/test:0.0.1
12
```

*Slika 2.11. Pod s oznakama*

Daljnjom organizacijom i kreiranjem Pod-ova koristeći ove dvije oznake dobivamo mnogo bolju strukturu i organizaciju Kubernetes resursa u našem klasteru. Pronalaženje pojedinog resursa ili više njih koji sadrže istu oznaku sada postaje jednostavno (npr. selector app = ui, rel = prod). Grafički prikaz Pod-ova u našem klasteru možemo vidjeti na slici.



Slika 2.12. Prikaz dvodimenzionalne organizacije Pod-ova sa oznakama [6]

Oznake nisu korisne samo prilikom traženja Pod-ova unutar već i prilikom samog kreiranja. Ako Pod-u, koji vrti određeni mikroservis, potreban čvor koji ima ugrađen SSD umjesto HDD-a onda to također možemo postići koristeći oznake. Kao što smo rekli oznake možemo dodijeliti bilo kojem Kubernetes objektu, uključujući i čvorove. Čvorovima dodajemo oznake koje nam govore npr. o tipu hardvera koji čvor posjeduje i drugim informacijama koje mogu biti korisne prilikom kreiranja Pod-ova. Tada prilikom kreiranja Pod-a, u YAML datoteci Kubernetesu dajemo do znanja da nam neki Pod zahtjeva specifične hardver resurse. Bit je da i dalje ne govorimo na kojem čvoru mora biti pojedini Pod već samo o potrebnim resursima koje zahtjeva taj Pod. Na taj način nam je i dalje iskoristivost resursa maksimalna jer prepuštamo Kubernetesu odluku na kojem čvoru će pokrenuti određeni servis, a servis dobije potrebne resurse za pravilno funkcioniranje.

```
apiVersion: v1
kind: Pod
metadata:
  name: node-selector-labels
spec:
  nodeSelector:
    gpu: "true"
  containers:
  - name: nodeSelContainer
    image: mmileta/test:0.0.1
```

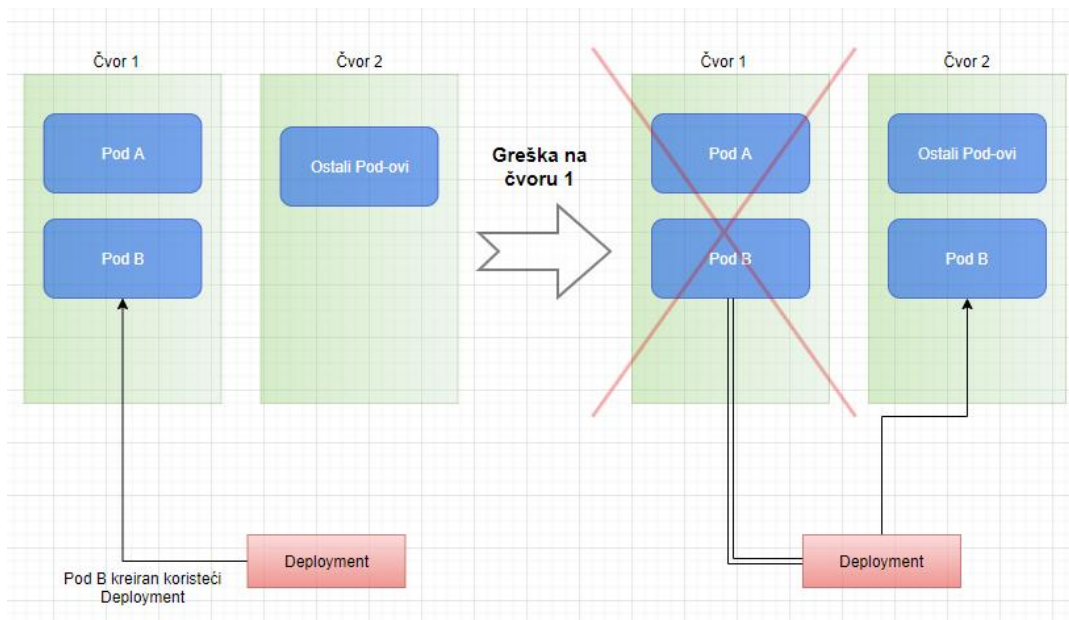
*Slika 2.13. YAML datoteka kreiranja Pod-a uz specifičan zahtjev prema čvoru*

Na slici vidimo primjer kreiranja Pod-a koji će biti migrirana na čvor u našem klasteru koji ima oznaku `gpu = true`.

### 2.6.2 Deployments

Kao što smo prije naveli, Pod-ove nikad nećemo direktno kreirati. Umjesto toga kreirat ćemo druge tipove resursa unutar Kubernetesa koji će se pobrinuti za održavanje i kreiranje pojedinog Pod-a. Jedan od tih resursa naziva se Deployment.

Glavna prednost korištenja Deploymenta je u tome što prilikom greške i prestanka rada Pod-a, on će biti automatski ponovno pokrenut. Nije bitno jeli došlo do greške u Pod-u ili na čvoru gdje se Pod vrti. Deployment će u oba slučaja ponovno pokrenuti Pod na jednom od zdravih radnih čvorova.



*Slika 2.14. Korištenje Deployments i događanje greške na radnom čvoru*

#### 2.6.2.1 Liveness probes

Kubernetes provjerava jesu li naši kontejneri i dalje u funkcionalnom stanju uz pomoć ‘liveness probe’. To radi na jedan od slijedeća tri mehanizma:

- HTTP GET request na IP pojedinog kontejnera te ako response koji dobije nazad predstavlja error (4xx ili 5xx kod) ili ne dobije response nazad uopće zaključit će da liveness probe nije prošao te će restartati kontejner
- TCP Socket gdje pokušava otvoriti TCP konekciju prema portu pojedinog kontejnera. Ako konekcija bude uspostavljena, liveness probe se smatra uspješom, a u suprotnom restarta kontejner
- Exec probe gdje unutar kontejnera pokrene proizvoljne komande i provjerava status izlazne naredbe. Ako je nula, liveness proba je prošla, a u suprotnom se kontejner restarta.

Uz Liveness probu postoji i Readiness proba. Pomoću nje određuje se jeli određeni kontejner spreman za primanje zahtjeva. Rediness proba na slične načine (HTTP Request, TCP Socket i exec) pokušava saznati da li je određeni kontejner u funkcionalnom stanju. U slučaju da nije spriječit će slanje zahtjeva tom kontejneru, sve dok pozitivno ne odgovori na Readiness probu.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: shopping-cart
spec:
  replicas: 1
  selector:
    matchLabels:
      app: shopping
  template:
    metadata:
      labels:
        app: shopping
    spec:
      containers:
        - name: users
          image: us.gcr.io/forward-emitter-321609/shopping-cart

```

*Slika 2.15. Kreiranje Pod-a koristeći resurs Deployment*

Kreiranjem Deploymenta, Kubernetes će se pobrinuti da je taj Pod uvijek pokrenut te da se u klasteru nalaz naveden broj kopija.

### 2.6.3 Services

U mikroservis arhitekturi Pod-ovi će najvjerojatnije trebati međusobno komunicirati koristeći HTTP protocol odnosno međusobno odgovarati na HTTP zahtjeve. Iz tog razloga svaki Pod mora znati kako pronaći IP adresu drugog Pod-a, ukoliko želi komunicirati s njime. Da ne koristimo Kubernetes, sistem administratori bi trebali migrirati određeni servis na neki server i javno dobivenu IP adresu manualno podijeliti s ostalim servisima koji žele komunicirati s tim servisom. Kod kubernetesa to ne funkcionira iz više razloga:

- Pod-ovi nisu trajni, odnosno svaki Pod može u svakom trenutku biti izbrisan sa čvora ili zamijenjen novim čime mu se IP adresa mijenja
- Kubernetes dodjeljuje IP adresu Pod-ovima tek nakon što im je raspoređivač dodijelio čvorove pa ne možemo znati IP adresu Pod-a unaprijed



- S obzirom da možemo imati više replika istog Pod-a, svaki Pod ima svoju IP adresu. To znači da bi mi trebali pratiti sve IP adrese Pod-ova koji se vrte a to sigurno ne želimo

Za rješenja navedenih problema Kubernetes pruža resurs koji se zove Service.

Koristeći resurs Service možemo napraviti jednu, statičnu i konstantnu krajnju točku preko koje možemo pristupiti grupi Pod-ova koji pružaju isti servis. Trebamo razlikovati Kubernetes Service resurs od naših mikro-servisa u aplikaciji. Svaki Service ima vlastitu IP adresu i port koji se ne mijenjaju sve dok Service postoji. Klijenti ili drugi Pod-ovi tada šalju zahtjeve na tu IP adresu i onda ih Service dalje proslijedi odgovarajućem Pod-u. Na taj način ne moramo znati točnu IP adresu Pod-a te se oni slobodno mogu premještati među radnim čvorovima u klasteru. Ima više tipova resurs Servicea:

1. ClusterIP – koristimo kada želimo napraviti Pod dostupnim samo unutar našeg klastera. U slučaju da ne navedemo tip Servisa prilikom kreiranja, Kubernetes će kreirati ClusterIP Service.
2. NodePort – uz pomoć NodePort-a, Pod postaje dostupan i izvan našeg klastera. Na svakom radnom čvoru otvori se port preko kojeg mu možemo pristupiti. Tada radni čvor proslijedi promet dobiven na tom port-u pojedinim Pod-ovima
3. LoadBalancer – koristeći Load Balancer možemo Pod izložiti vanjskom svijetu, odnosno učiniti ga dostupnim van klastera. Naš Pod tako postaje dostupan preko Load Balancera kojeg osiguraju cloud provideri. Load Balancer je zadužen za preusmjeravanje prometa prema određenom portu čvora. Klijenti se povežu na Load Balancer preko njegove IP adrese.
4. Ingress – način za izlaganje više Service-a preko jedinstvene IP adrese. To uspijeva koristeći određena pravila usmjeravanja koja mu mi prethodno opišemo.

Možemo uzeti za primjer jednostavno web aplikaciju koju smo prije spominjali. Pogledat ćemo komunikaciju između frontend-a i backend-a. Tip Service resursa nećemo specificirati pa će Kubernetes pretpostaviti da želimo ClusterIP Service što je i istina. Recimo da imamo više Pod-ova koji vrte frontend servis i više Pod-ova backend servisa. U tom slučaju za frontend ćemo napraviti tako da svaki HTTP zahtjev ide na Service koji ćemo prethodno napraviti. Taj Service će biti zadužen za prosljeđivanje zahtjeva prema ispravnom Pod-u. Prilikom kreiranja pojedinog

Service mi mu moramo dati to znanja kojoj grupi Pod-ova će on proslijeđivati zahtjeve. To postizemo koristeći oznake. Pod-ovima koji predstavljaju određen servis damo oznaku npr. user-servis i onda prilikom kreiranja Service resursa, kažemo mu da se zahtjevi proslijede Pod-ovima koji imaju tu oznaku.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-end-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  # tell the deployment how to find all the pods it should create (what pods the deployment will apply to)
  template:
    metadata:
      labels:
        app: frontend
    # tell the pod how to behave
    spec:
      containers:
        # name for logging purposes
        - name: frontend-container
          # arbitrary image from gke image repo
          image: us.gcr.io/forward-emitter-321609/frontend-service-image
# usually we have 1 to 1 relationship between services and deployments

---
# type is clusterIP service by default
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  ports:
    - name: frontend
      protocol: TCP
      port: 3000
      targetPort: 3000
```

*Slika 2.16. YAML datoteka za kreiranje Deploymenta i popratnog ClusterIP Service-a za frontend*

Na slici imamo primjer YAML datoteke koja kreira resurs Deployment koji će se pobrinuti da imamo tri replike našeg Pod-a. Unutar svakog Pod-a će se nalaziti kontejner, a unutar kontejnera ćemo imati frontend servis preuzet sa navedene Docker slike. Kubelet će dohvatiti tu sliku i pokreniti kontejner unutar Pod-a. Također Pod-ovima smo dali oznaku frontend. Nakon toga

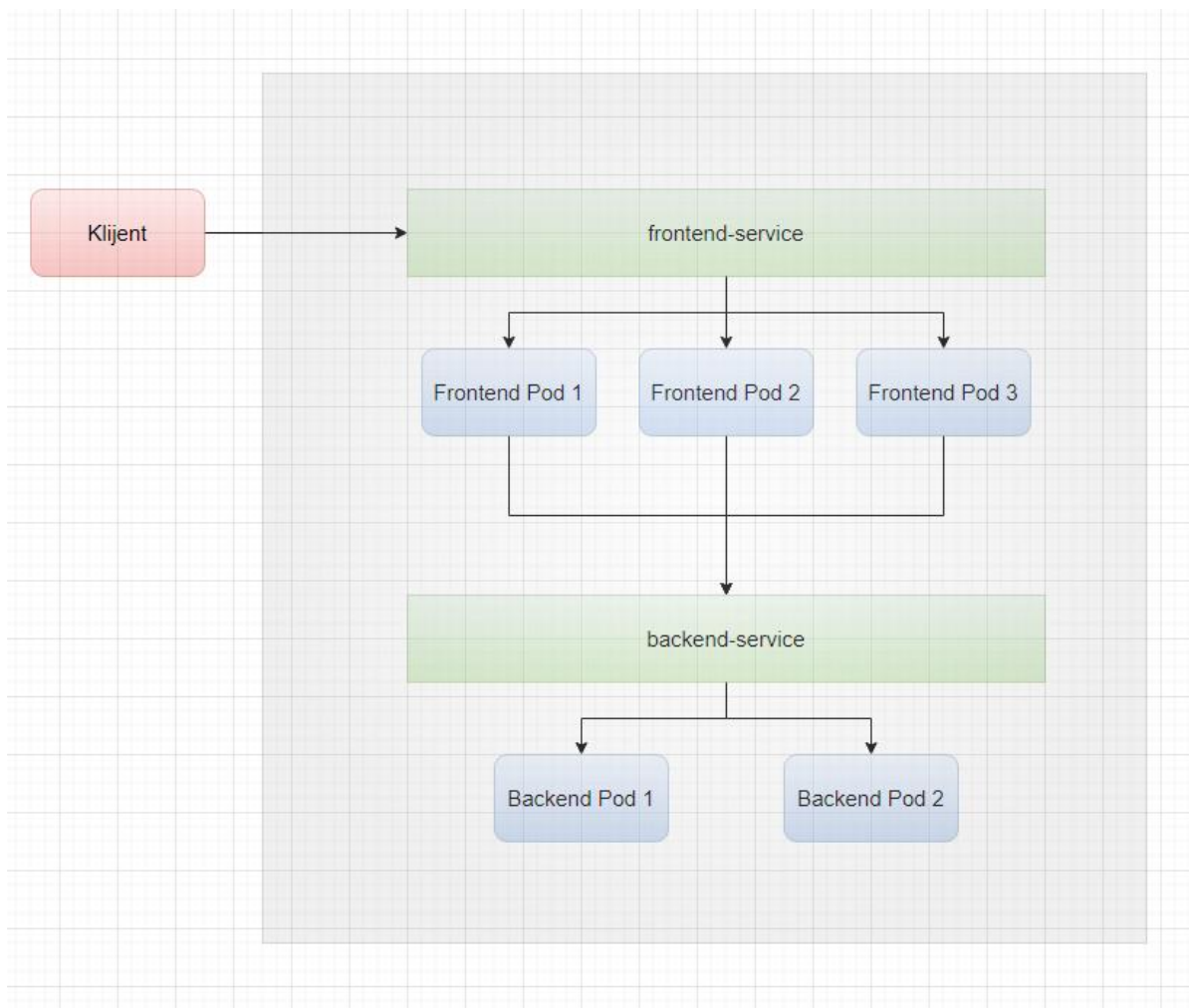
kreiramo resurs Service koji će nam služiti za proslijeđivanje komunikacije put frontend Pod-ova. Dajemo mu ime frontend-service i u polju selektor moramo napisati koju oznaku mora imati Pod kojem će Service proslijediti zahtjev. U ovom slučaju to je frontend. Service će biti dostupan na portu 3000 te će zahtjeve također proslijediti na port 3000 (targetPort). Kubernetes u sebi posjeduje lokalni DNS server preko kojega možemo određenim resursima poput Servica pristupiti preko njihovog imena umjesto direktno preko IP adrese koja mu kasnije bude dodijeljena. U ovom slučaju ime resursa je frontend-service.

Napravimo istu stvar i za backend. YAML datoteka će izgledati:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: back-end-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  # tell the deployment how to find all the pods it should create (what pods the deployment will apply to)
  template:
    metadata:
      labels:
        app: backend
    # tell the pod how to behave
    spec:
      containers:
        # name for logging purposes
        - name: backend-container
          # arbitrary image from gke image repo
          image: us.gcr.io/forward-emitter-321609/backend-service-image
# usually we have 1 to 1 relationship between services and deployments
---
# type is clusterIP service by default
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - name: backend
      protocol: TCP
      port: 3000
      targetPort: 3000
```

*Slika 2.17. YAML datoteka za kreiranje Deploymenta i popratnog ClusterIP Service-a za frontend*

Umjesto tri replike Pod-a za backend govorimo Deploymentu da su nam potrebne dvije te da im dajemo ime backend. Također kreiramo i resurs Service kojem dajemo ime backend-service. Servisu ćemo i pristupiti preko tog imena uz pomoć Kubernetes lokalnog DNS servera. U polje app unutar selektora govorimo Service-u koju oznaku će imati Pod-ovi kojima mora proslijediti zahtjev. Grafički prikaz komunikacije Pod-ova koje smo upravo opisali možemo pogledati na slijedećoj slici.



*Slika 2.18. Trenutni prikaz klastera*

Sada trebamo pogledati kako točno klijent uspijeva napraviti zahtjev na frontend servis. Da bi ga otkrili vanjskom svijetu odnosno da bude dostupan i van klastera možemo koristiti NodePort,

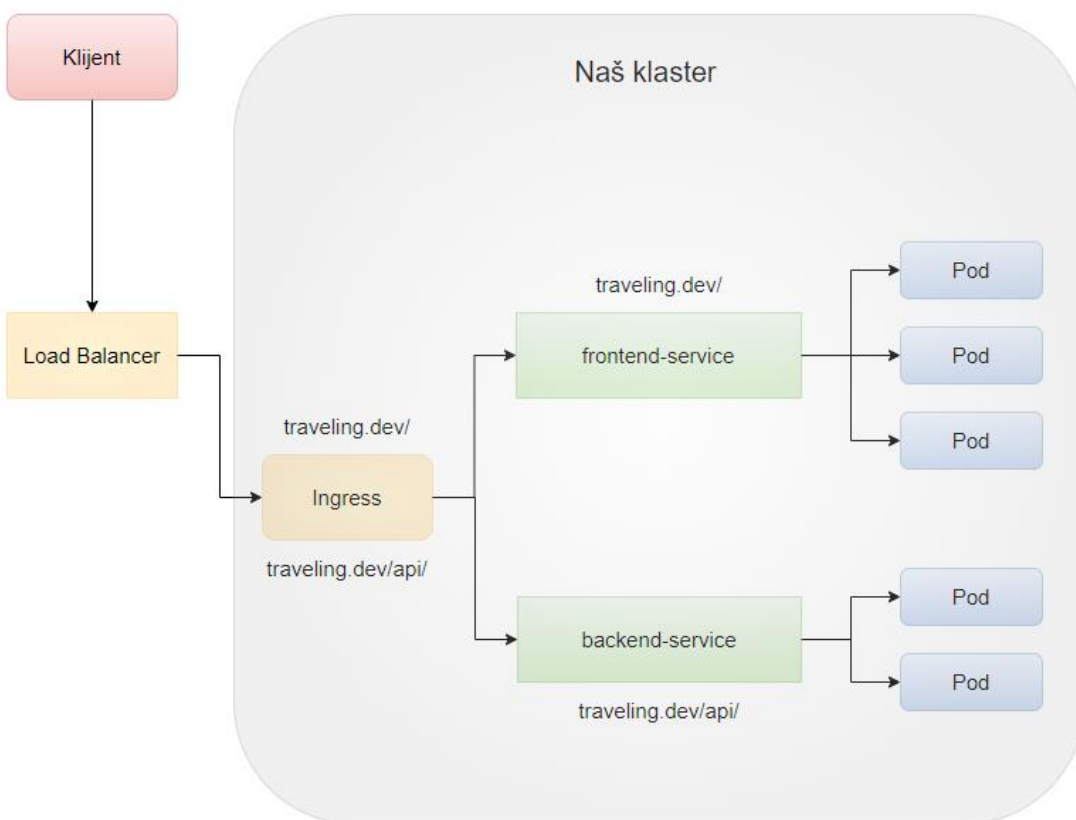
LoadBalancer ili Ingress Service. Kao što vidimo ima više načina za to napraviti. Mi ćemo objasniti jedan način koji je također bio korišten prilikom migriranja naše aplikacije na Kubernetes. Da naši Pod-ovi postanu dostupni i van klastera koristit ćemo kombinaciju Load Balancera i Ingress Servica. Load Balancer će nam služiti kako bi klaster i Pod-ovi unutar njega bili dostupni preko jedinstvene IP adrese. Tada će sav promet u naš klaster ići preko Load Balancera te će ga on usmjeriti na Ingress Service. Ingress Service ćemo konfigurirati tako da ovisno o izgledu i putanji zahtjeva se on preusmjeri na ispravan Pod. Load Balancer ćemo napraviti u našem Cloud Provideru te tako dobiti njegovu IP adresu. Ingress možemo napraviti putem YAML datoteke:

```
apiVersion: apps/v1
kind: Ingress
metadata:
  name: ingress-service
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-methods: "PUT, GET, POST, OPTIONS"
    nginx.ingress.kubernetes.io/cors-allow-credentials: "true"
spec:
  rules:
    - host: traveling.dev
      http:
        paths:
          - path: /api/(.*) # request that go to traveling.dev/api/ anything route to backend service
            backend:
              serviceName: backend-service
              servicePort: 3000
          - path: /?(.*) # request that don't have api prefix route to frontend service
            backend:
              serviceName: frontend-service
              servicePort: 3000
```

*Slika 2.19. YAML datoteka za kreiranje Ingress resursa*

Unutar datoteke vidimo kako smo naveli dvije putanje. Jedna koja nam predstavlja backend Service te jedna za frontend. Sada ako napravimo zahtjev na naš host odnosno traveling.dev iza kojeg se krije IP adresa Load Balancera (to prethodno moramo izkonfigurirati), zahtjev će ići prvo na Load Balancer koji će ga uputiti u Pod u kojem se nalazi Ingress Service. Tada će na temelju

putanje zahtjeva, Ingress odlučiti na koji Service će proslijediti zahtjev. Ako upišemo u browser url `traveling.dev` tada će Ingress taj zahtjev preusmjeriti na naš frontend Service te ćemo vidjeti HTML stranicu odnosno što god nam jedan frontend Pod-ova vrati. U slučaju da upišemo `traveling.dev/api/users` Ingress će taj zahtjev preusmjeriti na Pod u kojem se nalazi backend Service. Grafički bi to izgledalo ovako:



Slika 2.20. Grafički prikaz klastera nakon dodanog Inress objekta

#### 2.6.4 Volumes

Do sada smo vidjeli na koji način funkcioniraju Pod-ovi, kako se vrte kontejneri unutar njih te na koji način mogu međusobno komunicirati. S obzirom da se cijeli datotečni sustav nalazi unutar

kontejnera, ako dođe do greške i Pod se mora ponovno pokrenut, vidjet ćemo da novo pokrenuti kontejner neće vidjeti sve što je dosad bilo napisano u sustav datoteka. Vidjet će samo one datoteke koje su nastale iz slike koja je povučena iz registra slika. U nekim slučajevima želimo sačuvati određene datoteke koje pojedini servis generira. Tada koristimo resurs Volume.

Kubernetes Volume je jedna komponenta Pod-a i definiramo je prilikom kreiranja samog Pod-a. Dostupan je svim kontejnerima u Pod-u ali ga moramo montirati (eng. mount) u svaki kontejner koji zahtjeva pristup. Postoji više tipova Kubernetes Volume-a:

- emptyDir – kao što sam naziv nalaže, emptyDir nam pruža jedan prazni direktorij za pohranu prolaznih podataka
- hostPath – koristi se za pristup direktorijima koji se nalaze na radnom čvoru gdje i Pod
- gitRepo – Volume koji inicijaliziramo sadržajem određenog Git repozitorija
- nfs – network file sharing, specificira se prilikom kreiranja Pod-a
- gcePersistentDisk (Google Compute Engine Persistent Disk), awsElasticBlockStore (Amazon Web Services Elastic Block Store Volume), azureDisk (Microsoft Azure Disk Volume) – koriste se prilikom specificiranja određenog skališta cloud providera
- configMap, secret, downwardAPI – specijalni tipovi Volume-a koji se koriste za otkrivanje određenih Kubernetes resursa i informacija o klasteru Pod-u
- persistentVolumeClaim (PVC) – način za dinamičko osiguranje trajnih podataka.

Vidimo da svaki tip Volume-a služi određenoj svrsi. Da idemo detaljno objašnjavati svaki tip predugo bi trajalo kako ih ima previše. Proći ćemo one koji omogućuju trajno skladištenje podataka koje će biti dostupno svim Pod-ovima unutar klastera.

S obzirom da koristimo GKE – Google Kubernetes Engine, trebamo koristiti trajni disk za pohranu podataka od Googla – GCE – Google Compute Engine kao naš temeljni mehanizam. Da nam se klaster nalazi negdje drugdje koristili bi drukčiji tip trajnog diska kao Volume. To ovisi o cloud provideru. Prvo sa komandom trebamo kreirati trajni disk sa određenom količinom GB kako bi on bio spreman za pohranu podataka – *gcloud compute disks create --size=1GiB --zone=europe-west1-b mongodb*. Nakon toga moramo ga iskoristiti kao Volume unutar našeg Pod-a. YAML datoteka za kreiranje Pod-a koji ima specificiran Volume izgleda:

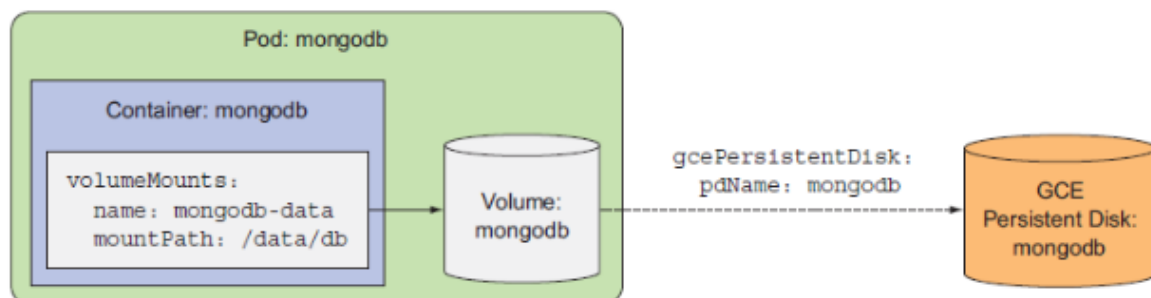
```

apiVersion: apps\v1
kind: Pod
metadata:
  name: database
spec:
  volumes:
  - name: mongodb
    gcePersistentDisk:
      pdName: mongodb
      fsType: ext4
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb
      mountPath: /data/db
    ports:
    - containerPort: 27017
      protocol: TCP

```

Slika 2.21. YAML datoteka za kreiranje Pod-a sa Volume resurs-om

Pod će imati samo jedan kontejner i jedan Volume koji će sadržavati prethodno kreirani GCE disk. Polje mountPath ovisi o bazi podataka te je sada postavljeno na /data/db jer tu MongoDB sprema podatke.



Slika 2.22. Prikaz referenciranja GCE trajnog diska [6]



Ovakav pristup bi funkcionirao međutim nije najbolji jer sada mi kao developeri moramo znati s kakvom infrastrukturom raspolažemo. Također aplikacija nije jednostavno prenosiva na druge cloud providere jer smo manualno i direktno naredili da se koristi trajni disk od GCE-a. Ideja Kubernetesa je da nam to lako omogući te da se mi ne moramo brinuti o resursima s kojim raspolažemo i koje koristimo. Iz tog razloga Kubernetes je uveo dva nova resursa a to su PersistentVolume (PV) i PersistentVolumeClaim (PVC). Koristeći ta dva resursa umjesto manualnog kreiranja trajnog diska možemo samo poslati zahtjeve Kubernetesu i na taj način osigurati potrebne resurse. U zahtjevu se mora nalaziti koliko nam je prostora potrebno. Ideja je bila da se developeri ne zamaraju infrastrukturom i resursima koje je potrebno kreirati u klasteru. Tada bi to bio zadatak sistem administratora. Oni bi napravili Kubernetes resurs zvan PersistentVolume unutar klastera koji bi nam služio za trajnu pohranu podataka. Također oni bi bili ti koji su upoznati sa detaljima skladištenja podataka koji znaju ovisiti o pružateljima usluga u oblaku. Mi kao developeri bi onda umjesto kreiranja Volume-a morali kreirati resurs PersistentVolumeClaim koristeći YAML file.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb
spec:
  resources:
    requests:
      storage: 100Mi
  accessModes:
    - ReadWriteOnce
```

*Slika 2.23. YAML datoteka za kreiranje PVC resursa*

Unutar YAML datoteke specificiramo koliko želimo imati prostora. Tada prilikom kreiranja Pod-a specificiramo prethodno napravljeni PersistentVolumeClaim i dolazimo do rezultata istog kao prije.

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: database
spec:
  volumes:
  - name: mongodb-pvc
    persistentVolumeClaim:
      claimName: mongodb
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb
      mountPath: /data/db
    ports:
    - containerPort: 27017
      protocol: TCP
```

*Slika 2.24. YAML datoteka za kreiranje Pod-a s PVC resursom*

PersistentVolumeClaim pregleda napravljene PersistentVolume u našem klasteru te kad nađe odgovarajućeg poveže ga sa Pod-om odnosno resurs PersistentVolume bude 'Claim-an' te postane nedostupan drugim Pod-ovima. Postoje tri 'Access Mod-a' PVC-a.

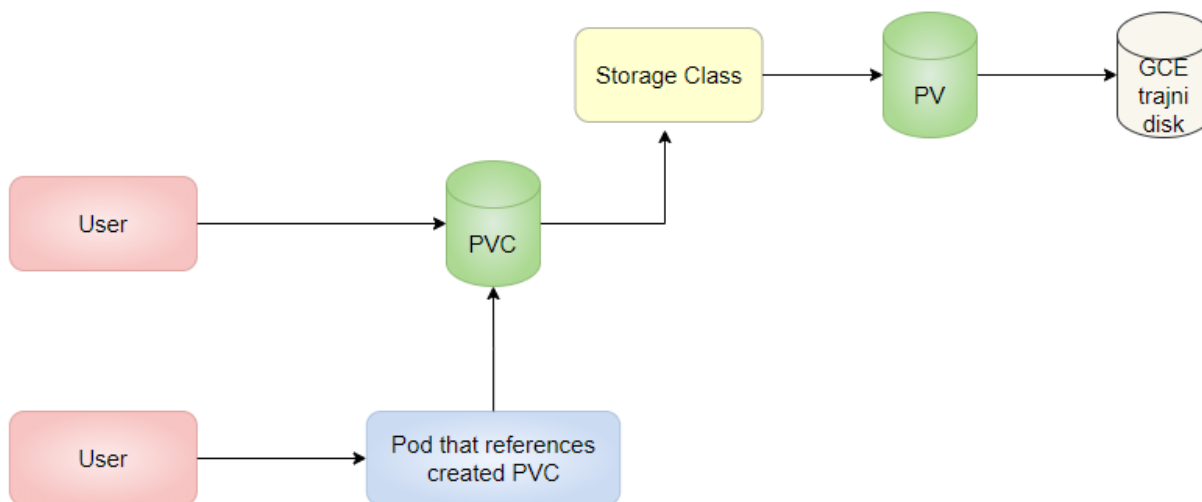
- ReadWriteOnce – volume možemo montirati kao 'read-write' od strane jednog čvora
- ReadOnlyMany – volume možemo montirati kao 'read' od strane jednog čvora
- ReadwriteMany – volume možemo montirati kao 'read-write' od strane više čvorova
- ReadWriteOncePod – volume možemo montirati kao 'read' od strane jednog Pod-a

S obzirom da mi u našem Pod-u referenciramo resurs PersistentVolumeClaim koji se veže za određeni PersistentVolume koji predstavlja trajno skladište podataka, prilikom ponovnog pokretanja Pod-a, Pod će i dalje vidjeti iste podatke jer će referencirati isti PersistentVolumeClaim koji je povezan sa istim PersistentVolume-om. Kada nam PersistentVolumeClaim više nije

potreban možemo ga izbrisati i omogućiti vraćanje resursa. Nakon brisanja PersistentVolumeClaim-a postoje tri opcije koje klaster može učiniti s oslobođenim PersistentVolume-om:

- Retain – Retain opcijom, PersistentVolume i podaci ostaju sačuvani nakon brisanja PersistentVolumeClaim-a, međutim PersistentVolume je i dalje ne dostupan te se ne može 'claim'-ati od strane drugog PVC-a
- Delete – uz Delete opciju uz brisanje PVC-a izbriše se i PV
- Recycle – PV može biti claim-am ponovno

Ovakav način kreiranja trajnog skladišta za naše podatke je prihvatljiviji za developere međutim i dalje sistem administratori moraju manualno kreirati PersistentVolume resurs. Iz tog razloga Kubernetes je uveo još jedan, novi resurs zvan StorageClass. Možemo definirati više StorageClass-ova međutim u svakom klasteru postoji jedan zadani StorageClass koji je spreman za korištenje. Uz pomoć StorageClass-a moguće je automatski i dinamički generirati potrebne PersistentVolume. Svaki put kada kreiramo Pod koji referencira PersistentVolumeClaim, StorageClass će pobrinuti za kreiranje potrebnih PersistentVolume-a. Na taj način sistem administratori i klaster administratori se ne moraju brinuti o opskrbljivanju developera određenim brojem PersistentVolume-a sa potrebnim specifikacijama. Također developeri ne moraju voditi računa ni o čemu osim o pisanju ispravnih YAML datoteka. Potrebno je napraviti YAML datoteku koja će kreirati PersistentVolumeClaim i YAML datoteku za Pod koji će je referencirati. Grafički prikaz trenutnog načina trajnog skladištenja podataka:



*Slika 2.25. Pojednostavljeni prikaz trajnog skladištenja podataka*

### 2.6.5 Secrets

Za pohranu osjetljivih podataka u Kubernetes sustavu koristimo Kubernetes resurs zvan Secret. Unutar tog resursa možemo pohraniti razne podatke. U našem slučaju to su bili određeni konfiguracijski podaci npr. JWT Key. Tim podacima možemo pristupiti prilikom kreacije Pod-a te ih spremati u env varijable. Kubernetes Secret najčešće nećemo kreirati pomoću YAML datoteka jer ne želimo da unutar neke datoteke se nalaze naši osjetljivi podaci. U tom slučaju kreiramo Secret koristeći komandnu liniju. Primjer kreiranja objekta koristeći komandnu liniju:

```
$ kubectl create secret generic our-jwt-secret --from-literal=jwtsecret=ourSecretValueGoesHere
secret/our-jwt-secret created
```

*Slika 2.26. Kreiranje Secret resursa koristeći komandnu liniju*

Sada unutar klastera imamo kreiran Secret s određenom vrijednošću te prilikom kreiranja Pod-a i kontejnera, prethodno kreirani Secret referenciramo na slijedeći način:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secret-pod-example-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: secret-pod-example
  template:
    metadata:
      labels:
        app: secret-pod-example
    spec:
      containers:
        - name: secret-pod-example
          image: us.gcr.io/forward-emitter-321609/secret-example
          env:
            - name: JWT_KEY
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: jwtsecret
```

*Slika 2.27. YAML datoteka za kreiranje Pod-a koji resurs Secret sprema u env varijablu*

### 3 DIZAJN I IMPLEMENTACIJA APLIKACIJE

Nakon pokazivanja kako mikro-servisi, Docker odnosno kontejner tehnologija i Kubernetes zajedno funkcioniraju, red je došao na aplikaciju gdje ćemo vidjeti to i u praksi. Napraviti ćemo web aplikaciju koja će se sastojati od više mikro-servisa te će oni međusobno komunicirati putem događaja (Events). Svaki servis biti će izoliran od drugih te zapakiran u Docker kontejner koristeći Docker. Nakon toga će svi biti servisi biti migrirani na Kubernetes Engine (GKE).

#### 3.1 Opis aplikacije

Web aplikacija 'Travel Memories' će omogućiti korisnicima da nakon registracije imaju jedinstveno mjesto gdje mogu pohranjivati svoje slike sa putovanja. U sebi će također imati ugrađen dio za razgovor (chat) preko kojeg će moći sa drugim korisnicima razmjenjivati dojmove i preporuke sa određenih mjesta koje su posjetili. Sukladno tome glavne tri funkcionalnosti aplikacije će biti:

- Mogućnost registracije i prijava korisnika
- Kreacija i pohrana novih uspomena sa putovanja
- Razmjenjivanje poruka u chat-u sa drugim korisnicima

Prilikom izgradnje aplikacije koristeći mikro-servise potrebno je dosta vremena provesti o arhitekturi cijele aplikacije. Razvoj pomoću mikro-servisa se razlikuje od standardnog pristupa kojeg koristimo kod izgradnje monolitne aplikacije. Najveća razlika nalazi se u pohrani podataka te načinima na kojima pristupamo podacima iz baze podataka. Umjesto da imamo jednu bazu podataka gdje ćemo sve potrebne podatke spremati, kod mikro servisa mi ćemo imati jednu bazu podataka po mikro-servisu. To bi značilo da u našem slučaju ćemo imati tri različite baze podataka. Jedna će biti zadužena za pohranu raznih podataka o korisnicima te će se koristiti prilikom registracije i prijavljivanja. Druga će pohranjivati razne slike i uspomene o pojedinim putovanjima korisnika. Treća će biti zadužena za pohranu razmijenjenih poruka u chat-u.

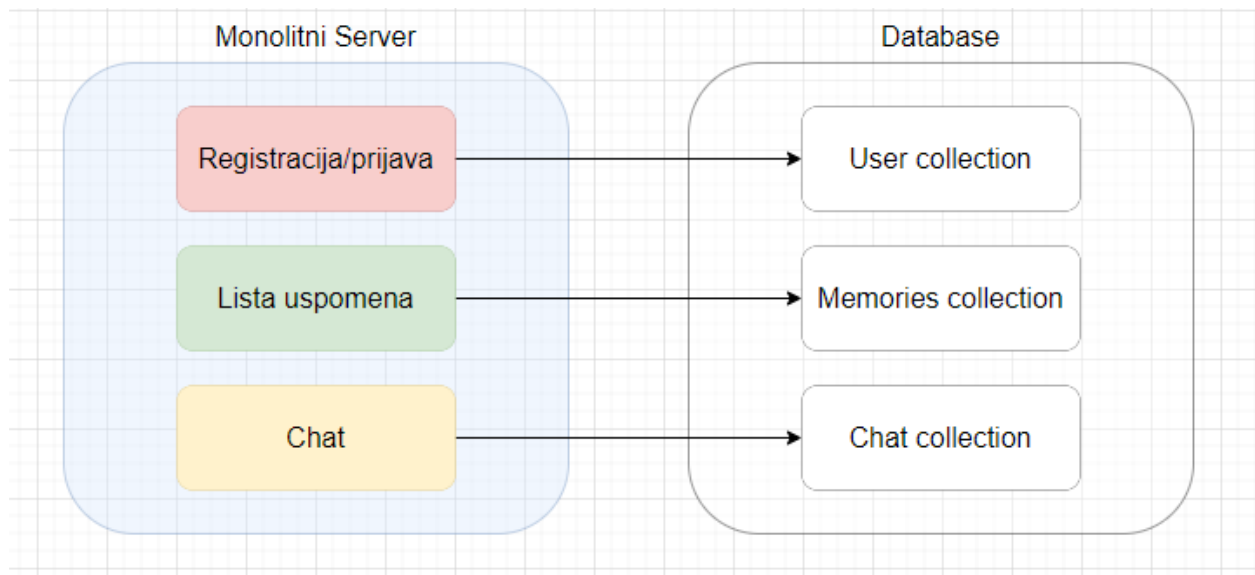
### 3.2 Jedan mikroservis – jedna baza podataka

Ima više pristupa i opcija kada razvijamo aplikaciju pomoću mikroservisa. Jedan od najraširenijih i prihvatljivijih je Database-Per-Service (jedna baza podataka po servisu). Ima više razloga zbog kojih želimo imati zasebnu bazu podataka za svaki servis a neki su:

- Neovisnost servisa jedni o drugima – recimo na primjer da imamo dva servisa koja dijele bazu podataka odnosno imamo više servisa koji koriste jednu bazu podataka. U tom slučaju ako dođe do greške u toj bazi podataka te ona postane ne funkcionalna na određeno vrijeme, tada svi naši servisi koji su se spajali na nju također neće raditi. Na taj način ne možemo postići kompletnu neovisnost mikroservisa. Zato ćemo svakom servisu dati zasebnu bazu podataka tako da u slučaju greške na određenoj bazi, samo jedan servis koji je spojen na nju prestane raditi. Još jedan razlog zašto to želimo je pojednostavljeno skaliranje jer tada možemo zasebno skalirati bazu podataka po potrebama servisa.
- Promjena scheme kod baze podataka – kako mikroservise mogu razvijati odvojeni timovi u različitim tehnologijama, može se dogoditi da neki od timova napravi promjenu koja mu odgovara na bazi podataka bez da to prethodno komunicira sa drugim timovima. U tom slučaju svi servisi koji su se spajali na tu bazu će početi dobivati greške unutar koda te će morati uložiti određeno vrijeme za njihovo otklanjanje.
- Različiti tipovi baza podataka – s obzirom da je svaki mikro-servis zadužen za obavljanje jedne zadaće u sustavu, može se dogoditi da za njeno obavljanje bi bila savršena 'nosql' baza podataka, dok bi nekom drugom servisu bila prikladnija 'sql' baza. U tom slučaju ne želimo mikroservise vezati za određenu bazu podataka, već njihovim timovima dati mogućnost biranja tipa baze podataka i tehnologija koje će koristiti.

#### 3.2.1 Problemi prilikom upravljanja podataka kod mikro-servisa

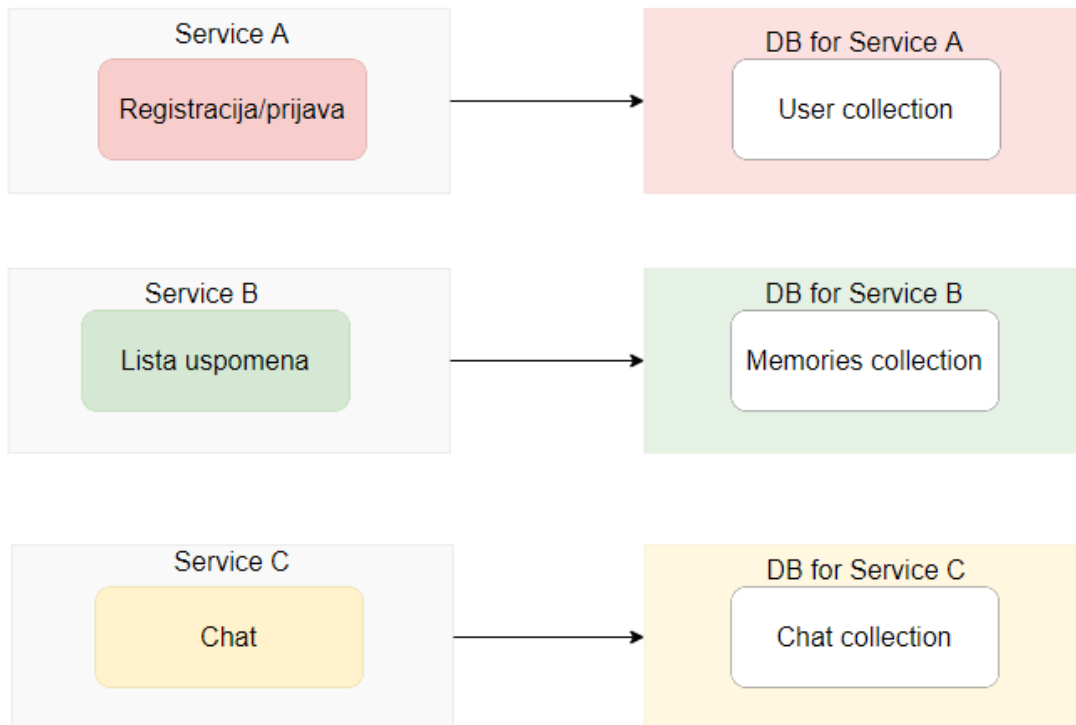
Prelaskom sa monolitne na mikro-servis arhitekturu javljaju se određeni problemi prilikom dohvaćanja podataka. Kada bi našu aplikaciju radili kao monolitnu tada bi imali jednu bazu podataka gdje bi se nalazili svi potrebni podaci.



*Slika 3.1. Prikaz strukture podataka kod monolitne aplikacije*

Ovakvim pristupom, na primjer dohvaćanje poruka koje pripadaju određenim korisnicima nije problem jer nam se svi podaci nalaze u istoj bazi podataka. Pogledajmo sada strukturu baze podataka kod mikro-servis arhitekture.





*Slika 3.2. Prikaz strukture podataka kod mikro-servis arhitekture*

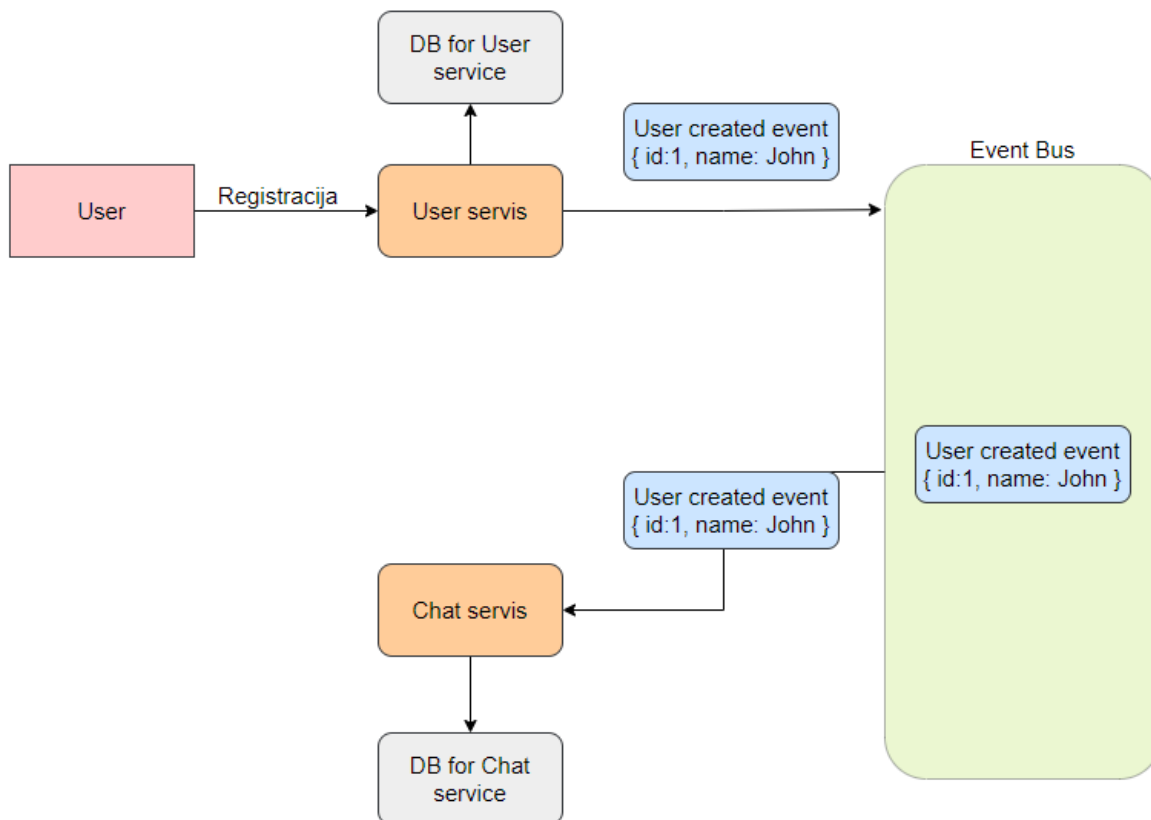
Sad uzmimo isti primjer – dohvaćanje poruka koje pripadaju određenom korisniku. Na slici je servis C zadužen za to te da bi mogao dohvatiti sve poruke koje pripadaju nekom korisniku on mora ili imati korisnikove podatke unutar svoje baze ili otići po njih u bazu servisa A (servis koji je zadužen za prijavu i registraciju korisnika). Međutim rekli smo da servisi moraju biti međusobno neovisni tako da dohvaćanje podataka iz druge baze podataka koja ne pripada servisu C nije dobar pristup. Postoji više pristupa rješavanju ovog problema kod mikro-servisa. Mi ćemo pogledati jedan koji će također biti korišten u razvoju naše aplikacije.

Da bi servis C mogao ispravno funkcionirati, mora imati pristup podacima o korisnicima. Iz tog razloga ćemo sve podatke koje bi pojedini servis mogao zatrebati također spremati i u bazu podataka tog servisa. Time ćemo duplicirati podatke jer će se podaci o korisniku nalaziti u bazi podataka servisa A i servisa C. To nije toliko problem jer je u današnje vrijeme skladištenje podataka postalo iznimno dostupno te pružatelji usluga u oblaku naplaćuju oko 50 lipa mjesečno za 1GB prostora. Time smo riješili problem međusobne ovisnosti servisa te u ovom slučaju ako se

na bazi podataka servisa A dogodi greška, korisnici se jedno vrijeme neće moći registrirati u sustav, međutim već postojeći korisnici (u slučaju da su prijavljeni) će moći koristiti sustav neometano.

### **3.3 Komunikacija između servisa**

Da bi nam Chat servis uspješno funkcionirao, mora znati koji korisnici postoje u sustavu. S obzirom da je User servis zadužen za registraciju korisnika prilikom koje prikupljamo željene podatke, moramo na neki način obavijestiti Chat servis da imamo novog korisnika u sustavu. Odnosno mora postojati određena komunikacija između dva navedena servisa. Komunikaciju ćemo postići koristeći event-ove (događaji). Kada se neki event dogodi unutar našeg sustava (npr. registracija korisnika), User servis će to objaviti. Tad ćemo unutar našeg sustava imati također Event-bus. Event-bus će biti servis koji će biti zadužen za primanje svih event-ova u našoj aplikaciji. Nakon što primi event, Event-bus će proslijediti isti event svim servisima koji postoje i koji su pretplaćeni (subscribed) na njega.



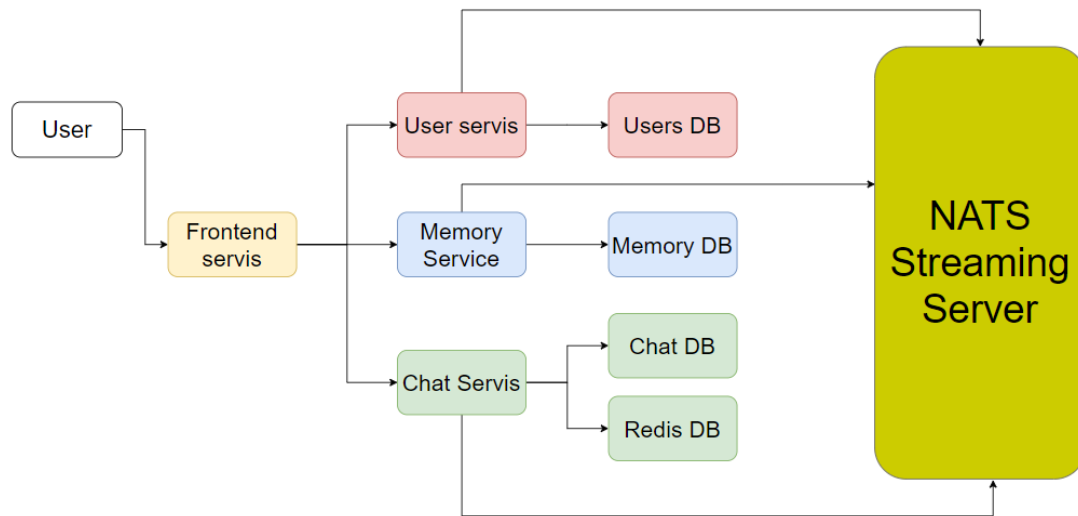
*Slika 3.3. Prikaz komunikacije između servisa*

Na slici vidimo način na koji će naši servisi komunicirati. Kada se određeni korisnik registrira, User servis će prije objavljivanja event-a spremiti korisnika u svoju bazu. Nakon toga event će primiti event-bus i proslijediti ga Chat servisu koji će samo spremiti samo one podatke korisnika koji su mu potrebni. Na isti način će funkcionirati i komunikacija između ostalih servisa.

### 3.4 Arhitektura aplikacije

Nakon rješavanja problema koji dolaze prilikom prelaska na mikro-servis arhitekturu, pogledat ćemo kako će cijela aplikacija sa svim servisima funkcionirati. Aplikacija će se sastojati od frontend dijela koji će predstavljati jednu cjelinu dok ćemo backend rastaviti u tri servisa (User, Memory, Chat) od kojih će svaki imati zasebnu bazu podataka. Također Chat servis ćemo spajati

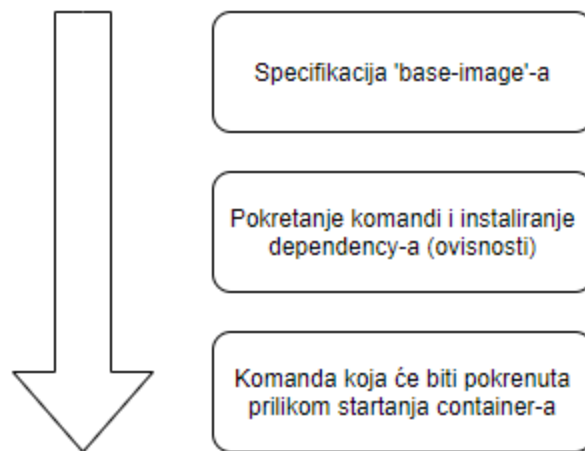
na Redis bazu kako bi mogli pratiti sesije korisnika. Redis je brza baza podataka jer se izvršava u RAM memoriji pa se može koristiti i za 'caching' podataka. Uz sve to imati ćemo i jedan servis gdje ćemo pokrenuti NATS Streaming Server. NATS će nam služiti kao Event-bus prilikom komunikacije između servisa. Kako će to sve izgledati možemo pogledati na slijedećoj slici.



*Slika 3.4. Mikroservis arhitektura naše aplikacije*

### 3.5 Spremanje komponenata u kontejnere

Nakon što smo odlučili o arhitekturi i dizajnu aplikacije, prije deployanja na Kubernetes potrebno je sve komponente aplikacije smjestiti u Docker kontejnere. S obzirom da ćemo aplikaciju deployati na Google Cloud, također ćemo koristiti i njihov registar slika. Za kreiranje Docker slike potrebno je napisati Dockerfile. Prema Dockerfile-u, Docker Client može sagrađiti sliku koju će kasnije koristiti prilikom pokretanja kontejnera. Dockerfile uvijek se sastoji od istih odjeljaka a to su specificiranje bazne slike, instaliranja ovisnosti te pokretanja komande koju želimo izvršiti nakon što se kontejner kreira.



*Slika 3.5. Dijelovi u Dockerfile-u*

S obzirom da su u našem slučaju Dockerfile-ovi jako slični pokazat ćemo primjer jednoga:

```
# specificiranje bazne slike
FROM node:alpine

# instaliranje dependency-a i priprema naše slike
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .

# komanda koju želimo pokrenuti nakon što se container starta
CMD ["npm", "start"]
```

*Slika 3.6. Primer Dockerfile-a*

Na slici možemo vidjeti od čega sve će nam se sastojati Dockerfile. S obzirom da su nam servisi napisani u Nodejs-u koristiti ćemo node:alpine kao baznu sliku. Kopirati naše datoteke unutar

radnog direktorija kontejnera te instalirati ‘dependency-e’ (zavisnosti). Na kraju moramo samo reći Docker-u koju komandu da izvrši prilikom pokretanja kontejnera. Uz pomoć ove datoteke će Docker napraviti odgovarajuću sliku pomoću kojeg će kasnije biti u mogućnosti startati naš kontejner. Drugim riječima uspješno smo ‘kontejnerizirali’ jedan od naših servisa.

Kako su nam i ostali servisi napisani u Nodejs-u i na sličan način, isti proces ćemo ponoviti i za njih te smo time uspješno smjestili naše komponente unutar kontejnera. Sve slike će nam biti smještene unutar kontejner registra na Google Cloud-u.

### **3.6 Skaffold**

Skaffold je alat koji će nam pomoći u razvoju aplikacije i komponentata koje su u kontejneru i deployanje na Kubernetes klaster. Recimo da imamo određeni Pod u kojem se vrti naš servis, da bismo napravili promjene u kodu tog servisa moramo uvijek proći kroz iste korake:

1. Napraviti potrebne promjene u kodu
2. Ponovno napraviti sliku
3. Gurnuti je na registar slika
4. Restartati Pod koji pokreće tu sliku

U početku ovo ne zvuči toliko strašno međutim kad se radi s više servisa proces s vremenom postane jako repetitivan i vremenski zahtjevan. Cijeli process ćemo automatizirati koristeći alat Skaffold tako da u budućnosti ne radimo repetitivne radnje. Nakon instalacije Skaffolda na našoj lokalnoj mašini koju koristimo za razvoj, moramo napisati određenu konfiguraciju kako bi scaffold znao kad je potrebno ponovno napraviti Docker sliku i restartati Pod.

```

apiVersion: skaffold/v2alpha3
kind: Config
deploy:
  kubectl:
    # govorimo skaffoldu gdje nam se nalaze yaml-deployment datoteke kako bi ih znao ponovno objaviti u slučaju promjene
    # umjesto nas će pokrenuti komandu kubectl apply za sve yaml datoteke
    manifests:
      - ./Infra/k8s/*
build:
  # local:
  # push: false
  googleCloudBuild:
    projectId: forward-emitter-321609
artifacts: # niz gdje imamo sve slike u Google cloud registru, kad se nešto promjeni unutar naših datoteka skaffold će update-ati sliku
  | | | # i ponovno pokrenuti deployment za koji je ta slika povezana
  - image: us.gcr.io/forward-emitter-321609/user-service
    context: UserService
    docker:
      dockerfile: Dockerfile
    sync:
      manual:
        - src: "src/**/*.ts"
          dest: .
  - image: us.gcr.io/forward-emitter-321609/memory-service
    context: MemoryService
    docker:
      dockerfile: Dockerfile
    sync:
      manual:
        - src: "src/**/*.ts"
          dest: .
  - image: us.gcr.io/forward-emitter-321609/chat-service
    context: ChatService
    docker:
      dockerfile: Dockerfile
    sync:
      manual:
        - src: "src/**/*.ts"
          dest: .

```

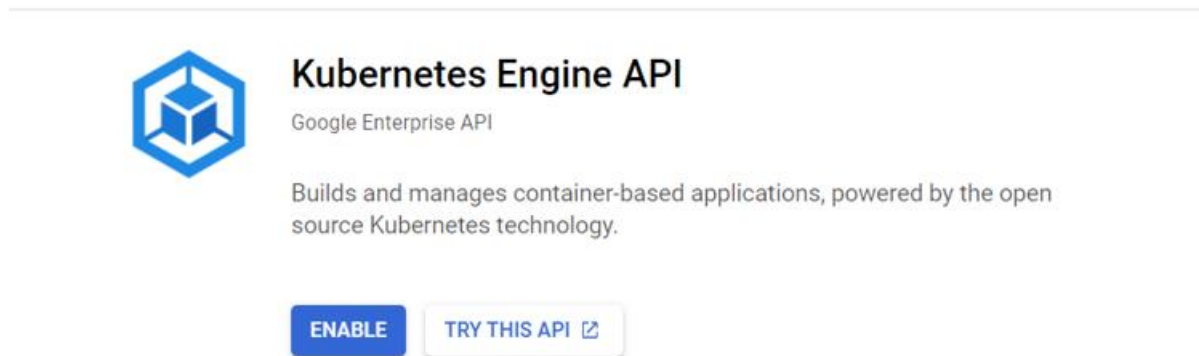
*Slika 3.7. Skaffold konfiguracija*

Na slici možemo vidjeti dijelove skaffold konfiguracije koju je potrebno napisati kako nebi morali manualno restartati Pod-ove prilikom malih promjena u kodu. Na taj način skaffold prati promjene u datotekama i pravovremeno reagira na njih. Time olakšavamo i ubrzavamo razvoj naše aplikacije.

## 4 MIGRACIJA APLIKACIJE NA KUBERNETES KLASTER

### 4.1 Uspostavljanje klastera

Nakon što smo uspješno smjestili sve dijelove aplikacije u kontejnere možemo krenuti s migracijom na Kubernetes kluster. Prvo što moramo učiniti je kreirati jedan. Otić ćemo na stranicu Google Clouda te omogućiti korištenje Kubernetes API-ja.



*Slika 4.1. Kubernetes API*

Proces traje par minuta te nakon što završi pojavit će nam se botun s oznakom ‘Create’ preko kojega ćemo kreirati naš klaster. Na slici možemo vidjeti naš nedavno kreirani klaster:



Kubernetes clusters

CREATE

DEPLOY

REFRESH

DELETE

Filter

Enter property name or value

<input type="checkbox"/>	Status	Name ↑	Location	Number of nodes	Total vCPUs	Total memory	Notifications	Labels
<input type="checkbox"/>	✓	standard-cluster-traveling	europe-central2-a	2	4	8 GB	—	⋮

*Slika 4.2. Kreirani klaster*

Nakon uspješno napravljenog klastera potrebno se spojiti na isti. Sučelje je vrlo intuitivno te nam daje komandu koju je potrebno pokrenuti da bi se uspješno spojili na upravo kreirani klaster.

## 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 autopilot-cluster-traveling --region europe-central2 --project forward-emitter-321609
```

[RUN IN CLOUD SHELL](#)

### Cloud Console dashboard

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

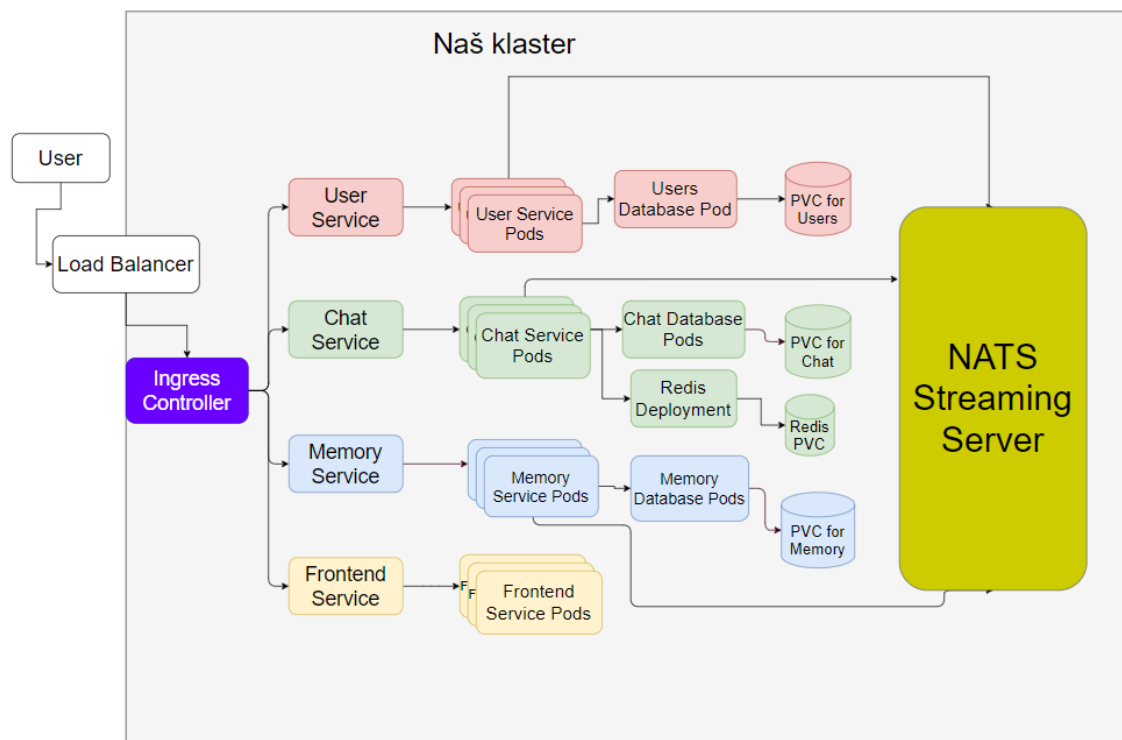
[OPEN WORKLOADS DASHBOARD](#)

OK

*Slika 4.3. Spajanje na klaster*

## 4.2 Arhitektura aplikacije unutar klastera

Pogledajmo još jednom arhitekturu naše aplikacije, ali ovaj put unutar Kubernetes klastera kako bi vidjeli koje sve komponente trebamo napraviti.



Slika 4.4. Prikaz arhitekture aplikacije unutar klastera

Kako bi našem klasteru mogli pristupiti iz vanjskog svijeta potreban nam je Load Balancer. On pripada pružateljima usluga u oblaku (u ovom slučaju Google Cloud) i ne nalazi se u našem klasteru. Load Balancer-u će biti dodijeljena jedinstvena IP adresa preko koje ćemo moći pristupiti našem klasteru. Kad želimo poslati zahtjev prema nekom od Pod-ova, šaljemo ga prema IP adresi našeg Load Balancera. Nakon toga će Load Balancer proslijediti zahtjev Ingress Controller-u kojem ćemo mi dati određena pravila rutiranja prema kojima će on dalje znati gdje proslijediti koji zahtjev. Možemo prvo kreirati Load Balancer i Ingress Controller. To ćemo napraviti koristeći projekt ingress-nginx. Na njihovoj stranici možemo naći upute kako instalirati oboje u naš klaster koji se nalazi na GKE. Moramo pokrenuti komandu:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.0.0/deploy/static/provider/cloud/deploy.yaml
```

Vidimo da komanda počinje sa `kubectl apply` i ima putanju prema YAML datoteci. Unutar datoteke se nalaze razni Kubernetes objekti koji ćemo kreirati u našem klasteru pomoću komandne linije i `kubectl` naredbe. 'Output' naredbe možemo vidjeti na sljedećoj slici.

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.48.1/deploy/static/provider/cloud/deploy.yaml
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
configmap/ingress-nginx-controller created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
service/ingress-nginx-controller-admission created
service/ingress-nginx-controller created
Warning: Autopilot increased resource requests for Deployment ingress-nginx/ingress-nginx-controller to meet requirements. See http://g.co/gke/autopilot-resources.
deployment.apps/ingress-nginx-controller created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-admission created
serviceaccount/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
Warning: Autopilot set default resource requests for Job ingress-nginx/ingress-nginx-admission-create, as resource requests were not specified. See http://g.co/gke/autopilot-defaults.
job.batch/ingress-nginx-admission-create created
Warning: Autopilot set default resource requests for Job ingress-nginx/ingress-nginx-admission-patch, as resource requests were not specified. See http://g.co/gke/autopilot-defaults.
job.batch/ingress-nginx-admission-patch created
```

*Slika 4.5. Output naredbe*

Sada imamo kreirani Load Balancer sa jedinstvenom IP adresom i unutar našeg klastera imamo `ingress-nginx-controller`.

Google Cloud Platform Traveling Search products and

Network services

Load balancing

Cloud DNS

Cloud CDN

Cloud NAT

Traffic Director

Service Directory

Cloud Domains

Load balancer details EDIT DELETE

a9f99f86e36144ff5a688c9d8320b7b2

Frontend

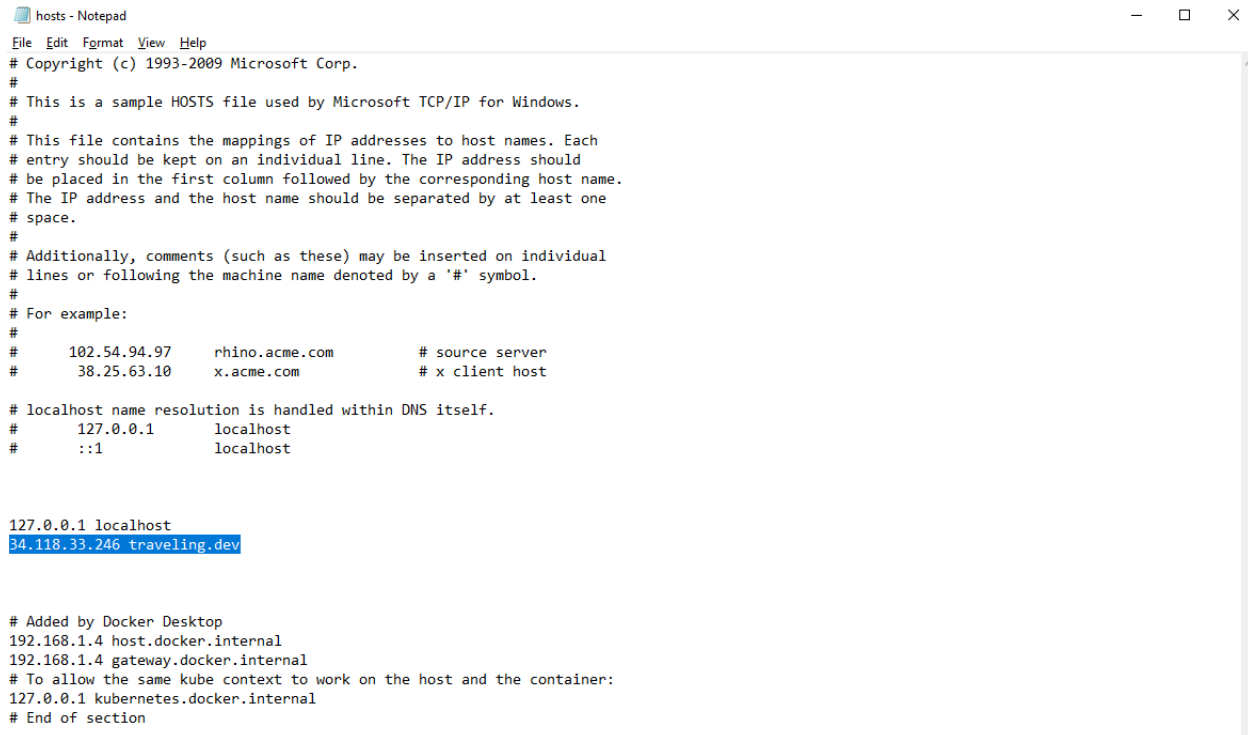
Protocol ↑	IP:Port	Network Tier ?
TCP	34.118.33.246:80-443	Premium

Backend

Name	Region	Health check
a9f99f86e36144ff5a688c9d8320b7b2	europe-central2	<a href="#">a9f99f86e36144ff5a688c9d8320b7b2</a>

Slika 4.6. Prikaz Load Balancera i IP adrese

Umjesto da pamtimo IP adresu Load Balancera možemo na Windowsu unutar host datoteke navesti IP adresu Load Balancera i povezati je sa željenim nazivom domene. Time smo napravili lokalni DNS server koji će promet sa navedene domene preusmjeravati na IP adresu kreiranog Load Balancera.



```
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 localhost
34.118.33.246 traveling.dev

# Added by Docker Desktop
192.168.1.4 host.docker.internal
192.168.1.4 gateway.docker.internal
# To allow the same kube context to work on the host and the container:
127.0.0.1 kubernetes.docker.internal
# End of section
```

*Slika 4.7. Mapiranje IP adrese u određenu domenu*

Sada kada u browseru upišemo `traveling.dev` bit ćemo preusmjereni na navedenu IP adresu. Da bi rutiranje bilo uspješno, moramo također napisati konfiguraciju za ingress kako bi on znao gdje proslijediti koji zahtjev. Napisana pravila možemo vidjeti na slijedećoj slici.

```

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-service
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-methods: "PUT, GET, POST, OPTIONS"
    nginx.ingress.kubernetes.io/cors-allow-credentials: "true"
    nginx.ingress.kubernetes.io/configuration-snippet: |
      more_set_headers "Access-Control-Allow-Origin: $http_origin";
spec:
  rules:
    - host: traveling.dev
      http:
        paths:
          - path: /api/users/(?.*)
            backend:
              serviceName: users-srv
              servicePort: 3000
          - path: /api/memories/(?.*)
            backend:
              serviceName: memory-srv
              servicePort: 3000
          - path: /api/chat/(?.*)
            backend:
              serviceName: chat-srv
              servicePort: 3000
          - path: /?(.*)
            backend:
              serviceName: client-srv
              servicePort: 3000

```

*Slika 4.8. Pravila ingress servisa*

U polje host stavljamo ime naše domene koja je mapirana sa IP adresom Load Balancera. Sada vidimo da ovisno o ruti, odnosno 'path'-u pojedinog zahtjeva, zahtjev će završiti na ispravom servisu koji će ga usmjeriti prema određenom Pod-u.

### 4.3 Migracija aplikacije na klaster

Kada smo s time završili možemo kreirati potrebne Kubernetes servise preko kojih ćemo moći pristupiti našim Pod-ovima. Prikaz kreiranja Service-a možemo vidjeti na slijedećoj slici.

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

*Slika 4.9. Prikaz keriranja User servisa*

Kreiranje Kubernetes Service-a nije komplicirano. Trebamo specificirati naziv Service-a preko kojeg ćemo mu moći pristupiti te na kojem portu će servis biti dostupan. Na slici se nalazi primjer UserService-a te je postupak jako sličan i za Memory, Chat i frontend komponentu pa ga nećemo ponavljati.

Slijedeće možemo kreirati odgovarajući Deployment. Pokazat ćemo također samo kreiranje User Deploymenta jer ćemo na jako sličan način kreirati i ostale potrebne Deploymente (frontend, Chat, Memory).

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: users-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: users
  template:
    metadata:
      labels:
        app: users
    spec:
      containers:
        - name: users
          image: us.gcr.io/forward-emitter-321609/user-service
          env:
            - name: NATS_CLIENT_ID
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: NATS_URL
              value: "http://nats-srv:4222"
            - name: NATS_CLUSTER_ID
              value: traveling-dev
            - name: JWT_KEY
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: JWT_KEY

```

*Slika 4.10. Prikaz kreiranja User Deploymenta*

Prilikom kreiranja Deploymenta moramo navesti sliku koja će biti pokrenuta unutar kontejnera. S obzirom da koristimo Google Cloud-ov registar slika, putanja do slike će nam uvijek biti oblika *us.gcr.io/{project\_id}/{service\_name}*. Također unutar Deploymenta možemo naći određene env varijable. NATS env varijable služe prilikom spajanja na NATS Streaming server kako bi komponente uspješno mogle objavljivati event-ove i primiti ih od drugih komponenata.



JWT\_KEY env varijabla se izvlači iz Kubernetes Secret objekta te služi prilikom prijave i registracija korisnika. Navedeni resurt Secret ćemo kreirati koristeći komandnu liniju.

```
$ k create secret generic jwt-secret --from-literal=JWT_KEY=asdf
secret/jwt-secret created
```

*Slika 4.11. Kreiranje secret objekta unutar klastera*

Na isti način kreiramo i Deploymente za Memory i Chat komponentu. Slijedeće što nam je potrebno su baze podataka. Prvo ćemo kreirati potrebni PVC koji će biti uzet od strane jednog od Deploymenta.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc-users
spec:
  resources:
    requests:
      storage: 100Mi
  accessModes:
    - ReadWriteOnce
```

*Slika 4.12. Kreiranje PersistentVolumeClaim-a*

Nakon uspješno kreiranog PVC-a možemo kreirati Deployment u kojem će se vrtiti naša baza podataka. Za pristup navedenom Deploymentu također ćemo koristiti Service. Kreiranje i jednog i drugog možemo vidjeti na slijedećoj slici.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: users-mongo-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: users-mongo
  template:
    metadata:
      labels:
        app: users-mongo
    spec:
      containers:
        - name: users-mongo
          image: mongo
          volumeMounts:
            - name: mongodb-pvc-users
              mountPath: /data/db
          ports:
            - containerPort: 27017
              protocol: TCP
          volumes:
            - name: mongodb-pvc-users
              persistentVolumeClaim:
                claimName: mongodb-pvc-users
---
apiVersion: v1
kind: Service
metadata:
  name: users-mongo-srv
spec:
  selector:
    app: users-mongo
  ports:
    - name: db
      protocol: TCP
      port: 27017
      targetPort: 27017

```

*Slika 4.13. Kreiranje Deploymenta za bazu podataka i pripadajućeg Servica*

Možemo primijetiti da se unutar YAML datoteke nalazi dio gdje uzimamo ('claim'-amo) nedavno napravljeni PVC kako bi mogli trajno skladištiti podatke. Za sliku pišemo 'mongo' jer je to naziv slike koju treba pokrenuti kako bi pristupili MongoDB bazi podataka. MongoDB baza se vrti na portu 27017 pa tu vrijednost stavljamo u kontejnerPort polje i u naš Service kao port i targetPort.

Na ovaj način smo uspješno kreirali User Service preko kojeg ćemo pristupati Pod-ovima, User Deployment koji će nam kreirati Pod-ove te ih održavati zdravima te Deployment za bazu podataka sa pripadajućim PVC-om. Sve navedene Kubernetes resurse moramo kreirati za Chat i za Memory

komponentu. Cijeli prethodno objašnjeni proces ponoviti još dva puta. S obzirom da je jako sličan nećemo ga ovdje pokazivati.

Slijedeće što nam je ostalo je kreirati Kubernetes Service i Deployment za NATS komponentu preko koje će naši servisi moći komunicirati. Prikaz YAML datoteke možemo vidjeti na slijedećoj slici.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nats-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nats
  template:
    metadata:
      labels:
        app: nats
    spec:
      containers:
        - name: nats
          image: nats-streaming:0.17.0
          args:
            [
              "-p",
              "4222",
              "-m",
              "8222",
              "-hbi",
              "5s",
              "-hbt",
              "5s",
              "-hbf",
              "2",
              "-SD",
              "-cid",
              "traveling-dev",
            ]
---
apiVersion: v1
kind: Service
metadata:
  name: nats-srv
spec:
  selector:
    app: nats
  ports:
    - name: client
      protocol: TCP
      port: 4222
      targetPort: 4222
    - name: monitoring
      protocol: TCP
      port: 8222
      targetPort: 8222
```

*Slika 4.14. YAML datoteka za NATS*

Možemo vidjeti da je YAML datoteka za kreiranje NATS Deploymenta jako slična prethodno kreiranim Deploymentima. Ima par različitih sitnica koje nećemo objašnjavati jer su izvan opsega rada.

Još nam je ostala samo jedna YAML datoteka za napisati, a to je Redis. Kao što smo rekli Redis koristimo za održavanje i praćenje sesija korisnika kod Chat servisa. Kao i prije moramo kreirati Service koji će ići uz odgovarajući Deployment te YAML datoteku možemo vidjeti na slijedećoj slici. S obzirom da se radi o bazi podataka moramo kreirati još jedan PersistentVolumeClaim koji će redis Deployment preuzeti. Persistent Volume Claim se radi na isti način kao što je prije opisan.

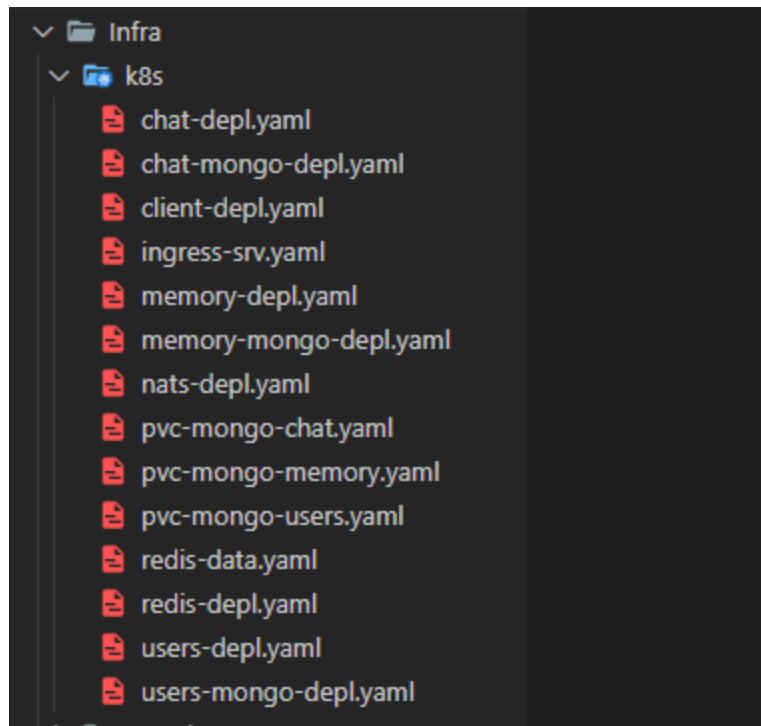
```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-depl
spec:
  selector:
    matchLabels:
      app: redis
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 6379
          # data volume where redis writes data
          volumeMounts:
            - name: data
              mountPath: /data
              readOnly: false
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: redis-data
---
apiVersion: v1
kind: Service
metadata:
  name: redis-srv
spec:
  selector:
    app: redis
  ports:
    - name: redis
      protocol: TCP
      port: 6379

```

*Slika 4.15. Prikaz YAML datoteke za Redis*

Time smo završili sa pisanjem svih YAML datoteka potrebnih za uspešno funkcioniranje naše aplikacije. Treba ih biti četrnaest:



*Slika 4.16. Sve YAML datoteke*

Sada ćemo pomoću prethodno objašnjenog alata Skaffold -a dati Kubernetes-u opis aplikacije kroz YAML datoteke kako bi on znao dalje kreirati potrebne resurse. Alat Skaffold pokrećemo preko komandne linije naredbom Skaffold dev.

```

$ scaffold dev
Listing files to watch...
- us.gcr.io/forward-emitter-321609/user-service
- us.gcr.io/forward-emitter-321609/memory-service
- us.gcr.io/forward-emitter-321609/chat-service
- us.gcr.io/forward-emitter-321609/client
Generating tags...
- us.gcr.io/forward-emitter-321609/user-service -> us.gcr.io/forward-emitter-321609/user-service:latest
- us.gcr.io/forward-emitter-321609/memory-service -> us.gcr.io/forward-emitter-321609/memory-service:latest
- us.gcr.io/forward-emitter-321609/chat-service -> us.gcr.io/forward-emitter-321609/chat-service:latest
- us.gcr.io/forward-emitter-321609/client -> us.gcr.io/forward-emitter-321609/client:e2ec460-dirty
Some taggers failed. Rerun with -vdebug for errors.
Checking cache...
- us.gcr.io/forward-emitter-321609/user-service: Found Remotely
- us.gcr.io/forward-emitter-321609/memory-service: Found Remotely
- us.gcr.io/forward-emitter-321609/chat-service: Found Remotely
- us.gcr.io/forward-emitter-321609/client: Found Remotely
Starting test...
Tags used in deployment:
- us.gcr.io/forward-emitter-321609/user-service -> us.gcr.io/forward-emitter-321609/user-service:latest@sha256:8df7396b8a89149a4676f5cb488b39dd9790ea0bb17c9a11ec23e5a1405e5e4
- us.gcr.io/forward-emitter-321609/memory-service -> us.gcr.io/forward-emitter-321609/memory-service:latest@sha256:a9e6288a86763d9f09ff0ae3e907b1fa47a8bfff4f76aad1e8bc710172fb053
- us.gcr.io/forward-emitter-321609/chat-service -> us.gcr.io/forward-emitter-321609/chat-service:latest@sha256:e0f041489e30b25b241342cba493ba278b3acedf53254599bc414ac556d40b0e
- us.gcr.io/forward-emitter-321609/client -> us.gcr.io/forward-emitter-321609/client:e2ec460-dirty@sha256:f01a02bee23a8341da2a49daca771dhead3e203a7c81bd09702f0fb09eb006c5
Starting deploy...
- deployment.apps/chat-depl created
- service/chat-srv created
- deployment.apps/chat-mongo-depl created
- service/chat-mongo-srv created
- deployment.apps/client-depl created
- service/client-srv created
- Warning: extensions/v1beta1 Ingress is deprecated in v1.14+, unavailable in v1.22+; use networking.k8s.io/v1 Ingress
- ingress.extensions/ingress-service created
- deployment.apps/memory-depl created
- service/memory-srv created
- deployment.apps/memory-mongo-depl created
- service/memory-mongo-srv created
- deployment.apps/nats-depl created
- service/nats-srv created
- persistentvolumeclaim/mongodb-pvc-chat created
- persistentvolumeclaim/mongodb-pvc-memory created
- persistentvolumeclaim/mongodb-pvc-users created
- persistentvolumeclaim/redis-data created
- deployment.apps/redis-depl created
- service/redis-srv created
- deployment.apps/users-depl created
- service/users-srv created
- deployment.apps/users-mongo-depl created
- service/users-mongo-srv created

```

Slika 4.17. Pokrećanje naredbe scaffold dev

S time smo završili migriranje naše aplikacije na Kubernetes klaster te samo ostaje vidjeti da li su uistinu sve komponente uspješno kreirane. Prvo pogledajmo Kubernetes Service resurse.

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.

Filter	Is system object: False	Filter services and ingresses						
<input type="checkbox"/> Name ↑	Status	Type	Endpoints	Pods	Namespace	Clusters		
<input type="checkbox"/> chat-mongo-srv	OK	Cluster IP	10.68.0.231	1/1	default	standard-cluster-traveling		
<input type="checkbox"/> chat-srv	OK	Cluster IP	10.68.7.38	1/1	default	standard-cluster-traveling		
<input type="checkbox"/> client-srv	OK	Cluster IP	10.68.12.235	1/1	default	standard-cluster-traveling		
<input type="checkbox"/> ingress-nginx-controller	OK	External load balancer	34.118.33.246.80	1/1	ingress-nginx	standard-cluster-traveling		
<input type="checkbox"/> ingress-nginx-controller-admission	OK	Cluster IP	10.68.2.25	1/1	ingress-nginx	standard-cluster-traveling		
<input type="checkbox"/> memory-mongo-srv	OK	Cluster IP	10.68.10.125	1/1	default	standard-cluster-traveling		
<input type="checkbox"/> memory-srv	OK	Cluster IP	10.68.15.54	1/1	default	standard-cluster-traveling		
<input type="checkbox"/> nats-srv	OK	Cluster IP	10.68.3.185	1/1	default	standard-cluster-traveling		
<input type="checkbox"/> redis	OK	Cluster IP	10.68.8.225	1/1	default	standard-cluster-traveling		
<input type="checkbox"/> redis-srv	OK	Cluster IP	10.68.4.177	1/1	default	standard-cluster-traveling		
<input type="checkbox"/> users-mongo-srv	OK	Cluster IP	10.68.12.62	1/1	default	standard-cluster-traveling		
<input type="checkbox"/> users-srv	OK	Cluster IP	10.68.4.75	1/1	default	standard-cluster-traveling		

Slika 4.18. Kreirani Kubernetes Service resursa

Vidimo da su svi servisi uspješno kreirani. Pogledamo li i Deploymente, vidjet ćemo istu stvar.

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

Filter

Is system object : False

Filter workloads

<input type="checkbox"/>	Name ↑	Status	Type	Pods	Namespace	Cluster
<input type="checkbox"/>	chat-depl	✓ OK	Deployment	1/1	default	standard-cluster-traveling
<input type="checkbox"/>	chat-mongo-depl	✓ OK	Deployment	1/1	default	standard-cluster-traveling
<input type="checkbox"/>	client-depl	✓ OK	Deployment	1/1	default	standard-cluster-traveling
<input type="checkbox"/>	ingress-nginx-admission-create	✓ OK	Job	0/1	ingress-nginx	standard-cluster-traveling
<input type="checkbox"/>	ingress-nginx-admission-patch	✓ OK	Job	0/1	ingress-nginx	standard-cluster-traveling
<input type="checkbox"/>	ingress-nginx-controller	✓ OK	Deployment	1/1	ingress-nginx	standard-cluster-traveling
<input type="checkbox"/>	memory-depl	✓ OK	Deployment	1/1	default	standard-cluster-traveling
<input type="checkbox"/>	memory-mongo-depl	✓ OK	Deployment	1/1	default	standard-cluster-traveling
<input type="checkbox"/>	nats-depl	✓ OK	Deployment	1/1	default	standard-cluster-traveling
<input type="checkbox"/>	redis-depl	✓ OK	Deployment	1/1	default	standard-cluster-traveling
<input type="checkbox"/>	users-depl	✓ OK	Deployment	1/1	default	standard-cluster-traveling
<input type="checkbox"/>	users-mongo-depl	✓ OK	Deployment	1/1	default	standard-cluster-traveling

Slika 4.19. Kreirani Kubernetes Deployment resursi

S time smo završili migriranje naše aplikacije na Kubernetes klaster. Sada je red na Kubernetesu da Pod-ove drži zdravima te ih ponovno pokrene u slučaju greške. Hoće li to učiniti možemo provjeriti na način da manualno izbrišemo određeni Pod. Pod-ove možemo izbrisati naredbom `kubectl delete pod [pod-id]`. Prije brisanja pogledajmo trenutno stanje Pod-ova u klasteru naredbom `kubectl get pods`.



```
$ k get pods
```

NAME	READY	STATUS	RESTARTS	AGE
chat-depl-64c9d9c7c7-82cjb	1/1	Running	0	42m
chat-mongo-depl-cd4fcf57f-mj9mk	1/1	Running	0	42m
client-depl-888cc9bbb-hkbfb	1/1	Running	0	42m
memory-depl-5f5d894b58-hw9l9	1/1	Running	0	22m
memory-mongo-depl-6c7dd7554d-zfx4q	1/1	Running	0	42m
nats-depl-6d98494475-24v5h	1/1	Running	0	42m
redis-depl-666496f59-z85ws	1/1	Running	0	42m
users-depl-6f8ddc579c-kntr9	1/1	Running	0	42m
users-mongo-depl-554df48f89-vdrn9	1/1	Running	0	42m

Slika 4.20. Trenutno stanje Pod-ova u klasteru

Uzmimo ID jednog od Pod-ova te izvršimo naredbu za brisanje. Pogledajmo stanje Pod-ova u klasteru nakon pokrenute naredbe:

```
Karlo@DESKTOP-9KD04Q2 MINGW64 /d/Users/Karlo/Desktop/Programming
$ k delete pod memory-depl-5f5d894b58-hw9l9
pod "memory-depl-5f5d894b58-hw9l9" deleted

Karlo@DESKTOP-9KD04Q2 MINGW64 /d/Users/Karlo/Desktop/Programming
$ k get pods
```

NAME	READY	STATUS	RESTARTS	AGE
chat-depl-64c9d9c7c7-82cjb	1/1	Running	0	48m
chat-mongo-depl-cd4fcf57f-mj9mk	1/1	Running	0	48m
client-depl-888cc9bbb-hkbfb	1/1	Running	0	48m
memory-depl-5f5d894b58-8tctj	1/1	Running	0	5s
memory-mongo-depl-6c7dd7554d-zfx4q	1/1	Running	0	48m
nats-depl-6d98494475-24v5h	1/1	Running	0	48m
redis-depl-666496f59-z85ws	1/1	Running	0	48m
users-depl-6f8ddc579c-kntr9	1/1	Running	0	48m
users-mongo-depl-554df48f89-vdrn9	1/1	Running	0	48m

Slika 4.21. Stanje Pod-ova u klasteru nakon brisanja

Vidimo da smo naredbom uspješno izbrisali Pod. Također možemo vidjeti da je novi Pod istog trenutka kreiran od strane Deploymenta te da mu je starost pet sekundi. Ovime smo pokazali da Kubernetes neprestano kontrolira broj Pod-ova u našem klasteru te osigurava da nam aplikacija uvijek bude u funkcionalnom stanju.

Također postoji opcija i automatskog skaliranja gdje će Kubernetes automatski povećati broj Pod-ova određene komponente ukoliko dođe do povećanog prometa.

## 4.4 Automatsko skaliranje

Jedna od prednosti migriranja aplikacije na ovaj način zasigurno je mogućnost automatskog skaliranja koje nam pruža Kubernetes. Trenutno smo u YAML datotekama naveli da želimo jednu repliku našeg kontejnera. Prilikom razvijanja aplikacije više od toga nam i nije potrebno. Međutim jednom kad aplikacija bude u produkciji, teško je predvidjeti kada će doći do povećanog opterećenja na nekom od Pod-ova. Kubernetes resurs HorizontalPodAutoscaler (HPA) nam nudi dinamičko povećanje i smanjenje broja Pod-ova ovisno o određenim metrikama pojedinog Pod-a. Kubernetes sustav to radi na način da neprestano provjerava metrike Pod-ova u klasteru i na temelju opterećenja donosi odluku o smanjenju odnosno povećanju broja replika. Najčešće korištena metrika je iskorištenost CPU-a te ćemo na takvom primjeru pokazati automatsko skaliranje u praksi. Prvo je potrebno kreirati resurs HPA. HPA možemo kreirati pomoću YAML datoteke i preko sučelja. Ni jedan način nije pretjerano kompliciran no kako smo sve prijašnje resurse kreirali pomoću YAML datoteke tako ćemo napraviti i za HPA resurs.

Prilikom kreacije potrebno je odabrati minimalni i maksimalni broj replika koje želimo unutar sustava. Zatim se moramo odlučiti za željenu metriku po kojoj ćemo skalirati pojedini servis te postotak nakon kojeg želimo povećati broj replika kako bi smanjili opterećenje. U ovom slučaju za metriku ćemo uzeti opterećenost CPU-u. Također moramo odabrati Deployment na koji će se odnositi kreirani HPA resurs. Recimo da to bude naš memory-deployment. YAML datoteka za navedeni HPA će izgledati:

```

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: memory-depl
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: memory-depl
  minReplicas: 1
  maxReplicas: 5
  targetCPUUtilizationPercentage: 50

```

*Slika 4.22. YAML datoteka za HPA resurs*

Na slici govorimo HPA da u slučaju prelaska 50% CPU opterećenja pokrene novu repliku servisa. Maksimalan broj replika želimo da je pet, a minimalan jedna. S obzirom da ćemo pokazati automatsko skaliranje preko opterećenosti CPU-a, moramo specificirati koliko svaki Pod zahtjeva resursa. To ćemo napraviti unutar YAML datoteke.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: memory-depl
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: memories
  template:
    metadata:
      labels:
        app: memories
    spec:
      containers:
        - name: memories
          image: us.gcr.io/forward-emitter-321609/memory-service
          resources:
            requests:
              cpu: "100m"

```

*Slika 4.23. YAML datoteka sa specificiranim resursom*

Napravili smo malu promjenu unutar naše YAML datoteke. Stavili smo početni broj replika na tri te smo specificirali da Pod zahtjeva 100m gdje m označava milicores procesora. Milicores je određena metrika Kubernetes sustava gdje 1000 Milicore-a odgovara jednoj procesorskoj jezgri. To znači da bi na procesoru s četiri jezgre mogli pokrenuti oko 40 Pod-ova od kojih svaki zahtjeva 100m. Koristeći alat skaffold uz komandu *skaffold-dev* ćemo kreirati HPA resurs i primijeniti promjene na Deploymentu.

Prvo će se Memory Deployment pobrinuti da imamo tri replike Memory Pod-a. Tada moramo pričekati određeno vrijeme da HPA resurs ugasi nepotrebne replike Pod-ova kako ne bismo trošili resurse koji nam nisu potrebni.

```
$ k get hpa
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
memory-depl	Deployment/memory-depl	1%/50%	1	5	1	8m26s

Slika 4.24. Trenutno opterećenje i broj replika

Polje TARGETS na slici označava trenutno opterećenje Pod-a (1%) i dopušteno opterećenje (50%). Sada ćemo simulirati nagli porast zahtjeva prema našem servisu. To ćemo učiniti pokretanjem koamnde *kubectrl run -it --rm --restart=Never loadgenerator --image=busybox -- sh -c "while true; do wget -O - -q http://memory-srv.default:3000; done"*.

Pomoću komande ćemo kreirati Pod koji će pokrenuti busybox Docker kontejner. Unutar kontejnera će se izvršavati petlja koja će slati zahtjeve prema Memory Service-u koji će ih proslijediti Memory Pod-u. Pogledamo li sada HPA resurs možemo vidjeti da je povećao brojeve replika kako bi uspješno obradio sve dolazeće zahtjeve:

```
$ k get hpa
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
memory-depl	Deployment/memory-depl	238%/50%	1	5	5	18m

Slika 4.25. Automatsko povećanje broja replika

Naredbom *kubectl get pods* ćemo provjeriti Pod-ove koji se nalaze unutar klastera. Možemo primjetiti da imamo četiri novo kreirana *memory-depl* Pod-a u stanju *READY* stara pet minuta. Ti Pod-ovi su bili pokrenuti od strane HPA resursa kako bi naš servis izdržao nadolazeće opterećenje.

```
$ k get pods
```

NAME	READY	STATUS	RESTARTS	AGE
chat-depl-64c9d9c7c7-82cjb	1/1	Running	0	31m
chat-mongo-depl-cd4fcf57f-mj9mk	1/1	Running	0	31m
client-depl-888cc9bbb-hkbfb	1/1	Running	0	31m
memory-depl-5f5d894b58-f1fd1	1/1	Running	0	5m32s
memory-depl-5f5d894b58-h5q27	1/1	Running	0	5m32s
memory-depl-5f5d894b58-hw919	1/1	Running	0	11m
memory-depl-5f5d894b58-kfkj9	1/1	Running	0	5m32s
memory-depl-5f5d894b58-z6nbw	1/1	Running	0	5m32s
memory-mongo-depl-6c7dd7554d-zfx4q	1/1	Running	0	31m
nats-depl-6d98494475-24v5h	1/1	Running	0	31m
redis-depl-666496f59-z85ws	1/1	Running	0	31m
users-depl-6f8ddc579c-kntr9	1/1	Running	0	31m
users-mongo-depl-554df48f89-vdrn9	1/1	Running	0	31m

*Slika 4.26. Pod-ovi nakon naglog porasta opterećenja*

Time smo se uvjerali da naš HPA resurs zaista radi. Ponovit ćemo radnju i za ostale servise odnosno kreirat ćemo im odgovarajuće HPA resurse. S obzirom da je proces isti nećemo ga pokazivati. Na ovaj način Kubernetes će se automatski pobrinuti da sav nadolazeći promet prema našim servisima bude uspješno obrađen te smo s time gotovi s migracijom svih servisa na klaster.

## 5 ZAKLJUČAK

Tokom razvoja aplikacije dalo se uočiti da je mikroservis arhitektura kompliciranija od monolitne. Potrebno je paziti na razne stvari kako bi se najbolje iskoristila prednost koju mikroservisi pružaju. Time je cijeli razvoj aplikacije u početku teži te treba odvojiti nešto vremena za smišljanje kompletne arhitekture aplikacije. Međutim prednosti koje donosi ovakav razvoj sigurno nadmašuju njene nedostatke. Skaliranje servisa mnogo je jednostavnije od skaliranja cijele monolitne aplikacije, koju je ponekad i ne moguće skalirati. Testiranje postaje jednostavnije te se pronađene greške u aplikaciji često mogu pripisati određenom servisu i krenuti odmah na njihovo uklanjanje.

Kada koristimo mikroservis arhitekturu najčešće se i koristi određena kontejner tehnologija uz čiju pomoć je moguće kompletno izolirati servise. To je vrlo bitan detalj kod cijelog razvoja jer je cilj napraviti aplikaciju od mnogo dijelova gdje se greška na jednom servisu ne propagira na sve druge. Recimo da postoji servis zadužen za slanje notifikacija. Sigurno ne želimo da nam cijela aplikacija stane sa radom ako nam se na tom servisu dogodi greška te korisniku trenutno ne dolaze notifikacije. Kako se broj servisa unutar aplikacije povećava treba nam i alat za njihovo upravljanje te smo koristili Kubernetes.

Korištenje Kubernetesa znatno olakšava posao sistem administratorima. Vidjeli smo kako kroz slanje jednostavne specifikacije Kubernetes sustavu možemo kreirati sve potrebne resurse u njemu. Nakon toga Kubernetes je taj koji je zadužen za motrenje naše aplikacije i svih servisa. Ukoliko se dogodi greška na jednom od servisa Kubernetes će se pobrinuti da ga ponovno pokrene. U slučaju da radni čvor prestane s radom, Kubernetes će sve komponente migrirati na drugi, zdravi čvor. Također pobrinuti će se da stanje u klasteru uvijek odgovara poslanim specifikacijama. Po potrebi, može automatski skalirati aplikaciju odnosno u nekom trenutku povećati broj Pod-ova određene komponente. Time sistem administratori mogu noći provesti spavajući bez da su oni zaduženi za dežuranje i motrenje cijele aplikacije.

Kubernetes zasigurno zahtjeva određeno vrijeme kako bi naučio. Postoji mnogo resursa koje je potrebno savladati. Međutim kad se to prođe, proces migriranja aplikacije na Kubernetes je jednostavan te prednosti koje s time donosi zasigurno su vrijedne prethodno uloženog vremena.

## LITERATURA

- [1] „Horizontal vs Vertical Scaling“ s interneta <https://touchstonesecurity.com/horizontal-vs-vertical-scaling-what-you-need-to-know/> , 13. rujna 2021.
- [2] “Scaling Horizontally vs. Scaling Vertically“ s interneta <https://www.section.io/blog/scaling-horizontally-vs-vertically/> 13. rujna 2021.
- [3] „Why use containers vs. VMs in a modern enterprise?“ s interneta <https://searchitoperations.techtarget.com/tip/Why-use-containers-vs-VMs-in-a-modern-enterprise> , 13. rujna 2021.
- [4] „Microservices vs. Monolith Architecture“ s interneta [https://dev.to/alex\\_barashkov/microservices-vs-monolith-architecture-411m](https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-411m) , 13. rujna 2021.
- [5] „Physical Servers vs. Virtual Machines: Key Differences and Similarities“ s interneta <https://www.nakivo.com/blog/physical-servers-vs-virtual-machines-key-differences-similarities/> , 13. rujna 2021.
- [6] Marko Lukša, “Kubernetes in Action”, Manning, NY, 2018.
- [7] Kubernetes koncepti i dokumentacija s interneta, <https://kubernetes.io/docs/concepts> , 13. rujna 2021.
- [8] „Service Mesh in Kubernetes - Getting Started“ s interneta, <https://www.programmingwithwolfgang.com/service-mesh-kubernetes-getting-started/> 13. rujna 2021

## **POPIS OZNAKA I KRATICA**

API – Application Programming Interface

CPU – Central Processing Unit

GKE – Google Kubernetes Engine

HDD – Hard Disk Drive

HTTP – HyperText Transfer Protocol

OS – Operacijski sustav

PV – PersistentVolume

PVC – PersistentVolumeClaim

SSD – Solid State Drive

TCP – Transmission Control Protocol

VM – Virtualna mašina



## SAŽETAK

Kroz ovaj diplomski rad izučili smo i opisali cijeli proces razvoja aplikacije koristeći mikroservis arhitekturu. Istakli smo prednosti ovakve arhitekture u odnosu s monolitnom, posebno s aspekta skaliranja aplikacije. Iako je monolitni pristup brži, ima brojna ograničenja prilikom skaliranja što u budućnosti može predstavljati problem. Mogućnost skaliranja je jako bitan faktor prilikom razvoja web aplikacija. U svakom trenutku želimo korisnicima pružiti najbolje moguće iskustvo prilikom interakcije sa našim sustavom, a praćenje velikog broja mikroservisa bez alata poput Kubernetesa bi bio iznimno težak pa čak i nemoguć zadatak.

U radu smo demonstrirali prednosti Kubernetes sustava koji nam omogućuje upravljanje s velikim brojem različitih mikroservisa. Kubernetes sustavu smo predali opis servisa u obliku YAML datoteka, a Kubernetes se sam brine da klaster uvijek odgovara opisu. U slučaju prestanka rada servisa, Kubernetes će ga ponovno pokrenuti te prebaciti na drugi radni čvor ukoliko je potrebno. Također, Kubernetes je u mogućnosti sam skalirati cijelu aplikaciju. Neprestano motri opterećenost naših servisa te u slučaju povećanog prometa može samostalno pokrenuti novu kopiju određenog servisa. Sve se odvija automatski. Time se ne moramo brinuti da će se naglim porastom prometa srušiti cijela aplikacija.

**Ključne riječi** – Kubernetes, Docker, skalabilnost, mikroservis arhitektura

## SUMMARY

Through this thesis we have studied and described the whole process of application development using microservice architecture. We highlighted the advantages of using this architecture over monolithic, especially from the aspect of application scaling. Although the monolithic approach is faster, it has a number of limitations when scaling which could cause some problem in the future. The ability to scale is a very important factor when developing web applications. At all times, we want to provide users with the best possible experience when interacting with our system, and monitoring a large number of microservices without tools like Kubernetes would be an extremely difficult and even impossible task.

In this paper, we have demonstrated the advantages of the Kubernetes system, which allows us to manage a large number of different microservices. We have submitted a description of the service in the form of YAML files to the Kubernetes system, and Kubernetes itself takes care that the cluster always matches the description. In the event of a service failure, Kubernetes will restart it and switch to another work node if necessary. Also, Kubernetes is able to scale the entire application itself. It constantly monitors the load of our services and in case of increased traffic can independently launch a new copy of a particular service. Everything happens automatically. This way, we don't have to worry that a sudden increase in traffic will crash the entire app.

**Title** – Scalable applications based on the Kubernetes system

**Keywords** – Kubernetes, Docker, scalability, microservice architecture