Unaprijeđenje sigurnosti pomoću mikroservisa

Kokić, Paula

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:697587

Rights / Prava: Attribution-NonCommercial-ShareAlike 3.0 Unported/Imenovanje-Nekomercijalno-Dijeli pod istim uvjetima 3.0

Download date / Datum preuzimanja: **2024-03-28**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Paula Kokić

UNAPRIJEĐENJE SIGURNOSTI POMOĆU MIKROSERVISA

DIPLOMSKI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Paula Kokić

Matični broj: 45263/16-R

Studij: Baze podataka i baze znanja

UNAPRIJEĐENJE SIGURNOSTI POMOĆU MIKROSERVISA

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Tonimir Kišasondi

Varaždin, kolovoz 2018.

Paula Kokić

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristila drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autorica potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Rad će se baviti mikroservisnom arhitekturom kao konceptom za unaprijeđenje općenite stabilnosti, sigurnosti i raspoloživosti današnjih web aplikacija. Mikroservisi kao manje izolirane jedinice dolaze s mnogo prednosti u odnosu na klasične monolitne arhitekture, ponajprije u pogledu nezavisnog razvoja te jednostavnije i brže isporuke. Koncept kontejnera tehnički omogućuje realizaciju teorijskih zahtjeva mikroservisa te čini glavnu poveznicu između mikroservisne arhitekture i Dockera, danas najpopularnijeg sustava za kontejnerizaciju. Orkestracija mikroservisa će biti prikazana uz alat Kubernetes, koji omogućuje zaista vrlo jednostavno upravljanje web aplikacijom u pogledu raspoloživosti i skaliranja. U konačnici, rad prikazuje testove koji još jednom pokazuju i demonstriraju realan slučaj u radu s mikroservisima te dokazuju koliko je i nadzor jako bitna komponenta cijelog procesa. Praktični dio ovog rada koristit će gotove aplikacije te će se fokusirati na prikaz najboljih praksi u radu s mikroservisima.

Ključne riječi: mikroservisi, sigurnost, izolacija, raspoloživost, Docker, Kubernetes

Sadržaj

1. Uvod 1
2. Metode i tehnike rada 2
3. Mikroservisna arhitektura 3
   3.1. Povijest arhitekture softvera 3
   3.2. Što su mikroservisi? 4
   3.3. Prednosti i nedostaci 5
4. Docker 7
   4.1. Kontejneri 7
   4.2. Osnovni pojmovi 9
   4.3. Uobičajeni tijek rada 9
5. Kubernetes 11
   5.1. Organizacija sustava 11
   5.2. Osnovni objekti 11
   5.3. Arhitektura sustava 13
6. Isporuka mikroservisne aplikacije 15
   6.1. Postavljanje Kubernetes klastera 15
   6.2. Kontejnerizacija mikroservisa 17
   6.3. Konfiguracija ljuski i servisa 19
   6.4. Replikacija mikroservisa 24
   6.5. Balansiranje opterećenja 27
7. Nadzor mikroservisa 30
   7.1. Važnost pravilnog nadzora i vidljivosti 30
   7.2. Prometheus 31
   7.3. Postavljanje aplikacije *Sock Shop* 32
   7.4. Postavljanje nadzornih alata 36
   7.5. Test opterećenja 38
8. Zaključak 41
Popis literature 42
Popis slika 44

1. Uvod

Ovaj rad bavit će se jednom od trenutno vrlo aktualnih tema na području informacijskih tehnologija, a to su mikroservisi. S obzirom da je to poprilično široka tema, zadržat ću se samo na nekim aspektima njihove primjene, odnosno pokušati kroz primjer pokazati osnovne procese i neke od alata koji se mogu koristiti kako bi se povećala raspoloživost aplikacije te napravila što bolja izolacija različitih servisa. Prikazat će se implementacija i konfiguracija alata koji nam omogućuju lakšu isporuku i upravljanje mikroservisima, odnosno već poznatih tehnologija kao što su Docker i Kubernetes.

Najprije ću opisati alate koje će biti korišteni u ovom radu te nakon toga dati teorijsku podlogu koja je potrebna za shvaćanje navedenih koncepata: mikroservisna arhitektura te osnove isporuke i nadzora mikroservisnih aplikacija. U konačnici ću analizirati i prikazati konfiguraciju mikroservisne aplikacije za isporuku kroz jedan primjer te nadzor mikroservisa kroz drugi primjer, odnosno pokušati dati najbolje prakse za izolaciju, optimizaciju, raspoloživost i nadzor mikroservisa, čime se upravo postiže i bolja sveukupna sigurnost aplikacije.

Osim što je ova tema danas vrlo popularna i relativno nova, izabrala sam je zato jer je moj cilj u IT industriji uvijek više naginjao prema tehničko-hardverskoj strani, odnosno prema konfiguraciji računala, mreža poslužitelja (eng. *server*), sistemskoj administraciji i nadzoru aplikacija i korisnika. Iz tog razloga ovo područje mi se čini jako privlačnim i vrijednim istraživanja budući da se ovakvoj arhitekturi pridaje sve više pažnje te bih osobno željela naučiti puno više o samoj isporuci aplikacija i njihovom nadzoru te administraciji. Velike, monolitne aplikacije jako su nezgrapne za održavanje, isporuku novih verzija, nadzor i upravljanje sigurnosti te će biti zanimljivo vidjeti koliko nam mikroservisi mogu pomoći u izgradnji lakše održive, stabilnije i sigurnije aplikacije.

2. Metode i tehnike rada

Prilikom izrade ovog rada koristila sam nekoliko tehnologija i softverskih alata, uključujući:

- Microsoft Office Word 2013 – za izradu dokumenta rada
- DigitalOcean – platforma koja omogućuje pristup i korištenje udaljenih virtualnih računala (poslužitelja), a besplatan pristup do određene granice omogućio mi je GitHub Student Developer Pack
- Docker 17.03 – alat za kontejnerizaciju aplikacija
- Kubernetes 1.11 – alat za orkestraciju aplikacija
- Ubuntu 16.04 – operacijski sustav na Digital Ocean *droplet*-ima
- Prometheus – alat za nadzor web aplikacija i mikroservisa
- Grafana – alat za vizualizaciju podataka dobivenih iz Prometheusa
- Draw.io (https://www.draw.io/) – online alat korišten za izradu svih dijagrama
- Oracle VirtualBox – alat za virtualizaciju.

U prvom dijelu rada bavit ću se teorijskim osnovama i opisom samih tehnologija (Docker i Kubernetes alata), dok ću drugi dio rada posvetiti praktičnoj primjeni svega navedenog te pokušati prikazati koje su najbolje prakse prilikom korištenja navedenih alata.

3. Mikroservisna arhitektura

U ovom poglavlju bavit ću se opisom i arhitekturom mikroservisnih aplikacija te dati kratki pregled problema koje ista rješava. Kako bismo bolje razumjeli prednosti mikroservisne arhitekture, najprije ću dati kratak pregled najčešćih arhitektura koje su se dosada koristile.

3.1. Povijest arhitekture softvera

Razvoj softvera krenuo je s monolitnom (eng. *monolith*) arhitekturom – cijela aplikacija bila je sagrađena kao jedan veliki komad objekta. Takve aplikacije su jednostavne za razvoj i produkciju jer imaju samo jednu bazu podataka i jednu konfiguraciju. Međutim, vrlo brzo mogu postati jako velike i komplekse te najmanja promjena rezultira ponovnim dugotrajnim testiranjem i produkcijom. Kada se razvoju pridruži novi developer, isti mora proučiti kako cijela aplikacija funkcionira da bi se mogao pridružiti razvoju. Osim toga, kada poraste potreba za korištenjem aplikacije, jedini način na koji se ostvaruje skalabilnost je replikacija aplikacije, što rezultira dodatnim visokim troškovima.

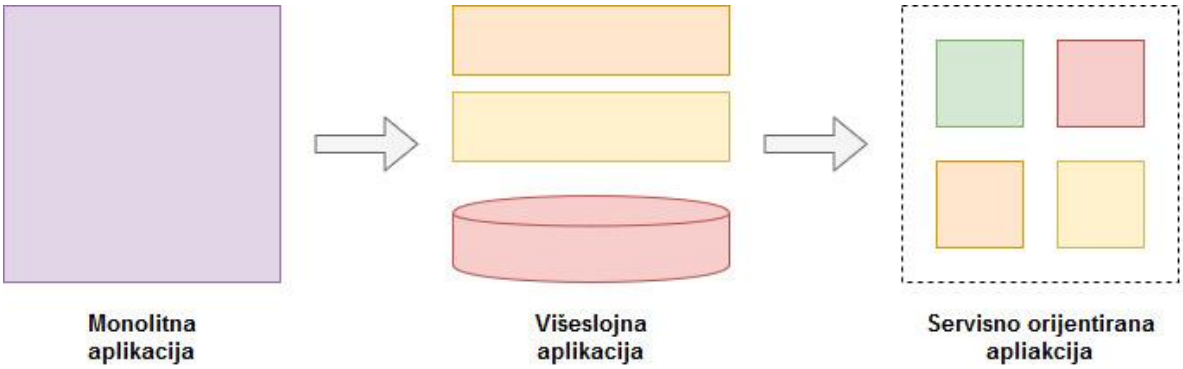
Loše strane monilitne arhitekture programeri su pokušali riješiti višeslojnom (eng. *multi-tier*) arhitekturom. Na taj način aplikacija se dijeli u slojeve, najčešće na podatkovni sloj (baza podataka), sloj poslovne logike i prezentacijski sloj (grafičko sučelje). Ovakva podjela pojednostavnila je proces skalabilnosti, budući se (ukoliko se ukaže potreba) baza podataka može replicirati zasebno, poslovni sloj se također može zasebno širiti ovisno o novim funkcionalnostima, a prezentacijski sloj za klijente može se napraviti da radi na više platformi. Međutim, i ova logika funkcionira s relativno malim aplikacijama – čim se opseg znatno poveća, aplikaciju je teže skalirati, održavati i dalje razvijati te se vraćamo na probleme monolitnih aplikacija.

Idući korak u razvoju bila je servisno-orijentirana (eng. *service-oriented*) arhitektura, pomoću koje se aplikacija pokušala odvojiti na manje dijelove (servise) koji su bili zaduženi za određeno područje (funkcionalnost) aplikacije (npr. autentifikacija, obavijesti, narudžbe). Ovakva ideja bila je vrlo efektivna, ali ponekad ipak previše kompleksna iz perspektive slojevitosti, konfiguracije, protokola, odnosno teška za održavanje i unaprjeđivanje. [1]

S napretkom tehnologije, točnije njezinom brzinom razvoja, mijenjaju se i zahtjevi za razvojem aplikacija. Današnji softver mora biti moguće brzo mijenjati, održavati i skalirati prema potrebama poslovanja. Javljaju se pojmovi poput kontinuirane integracije (eng. *continuous integration*) i kontinuirane isporuke (eng. *continuous delivery*) – CI/CD, a odnose se upravo na to: neprestani razvoj, nadogradnju i integraciju postojećeg softvera, s ciljem da

stabilna aplikacija, s novim funkcionalnostima može biti puštena u rad svaki dan – zahtjevi koje dosadašnje arhitekture nisu mogle zadovoljiti.

Na Slici 1. skiciran je arhitekturalni dizajn prethodno navedenih pristupa.



Slika 1. Dizajn različitih arhitektura kroz povijest (Prema: [1])

3.2. Što su mikroservisi?

„*Mikroservis* je nezavisno isporučiva komponenta vezanog područja koji podržava interoperabilnost putem komunikacije porukama. *Mikroservisna arhitektura* je stil programskog inženjerstva u kojem su visoko automatizirani i nadogradivi softverski sustavi sastavljeni od podjednako sposobnih mikroservisa.“ [2]

Ključna obilježja koja opisuju mikroservise su sljedeća:

- male veličine - termin „mikro“ ne odnosni se nužno na veličinu sevisa u pogledu linija kôda, već na njegovu funkcionalnost - svaki mikroservis slijedi princip jedne odgovornosti (eng. *single responsibility principle*);
- komunikacija porukama - komunikacija između mikroservisa najčešće se realizira putem HTTP protokola, koristeći RESTful API sučelja;
- vezani kontekstom – mikroservisi su visoko kohezivni te rade zajedno u sklopu jednog sustava;
- samostalan razvoj – svaki mikroservis se može razvijati neovisno o ostalima;
- nezavisna isporuka – svaki mikroservis se može isporučiti neovisno o ostalima;
- decentralizacija – nema središnjeg sustava, svi su jednako sposobni te svaki razvojni tim brine o svom mikroservisu zasebno i neovisno o ostalima;
- izgradnja i isporuka s automatiziranim procesima. [1]–[3]

3.3. Prednosti i nedostaci

Prednosti korištenja mikroservisa su brojne, a neke od njih uključuju:

- eliminacija jedne točne otkaza – kada se dogodi bilo kakva vrsta kvara, lakše ga je dijagnosticirati i otkloniti te je vrlo mala vjerojatnost da će zbog manje greške pasti cijeli sustav;
- aerodinamična orkestracija (eng. *streamlined orchestration*) – svaki mikroservis može imati vlastite automatizirane procese koji su manje kompleksni, a okolina može biti jednaka u razvoju i produkciji s efektivnijim postavkama;
- brže iteracije – svaki razvojni tim može pratiti svoj tempo razvoja mikroservisa, neovisno o ostalima, a prilikom isporuke opet se radi o produkciji samo jednog mikroservisa, a ne cijelog sustava;
- efektivna skalabilnost – svaki mikroservis može biti skaliran zasebno, prema potrebi;
- nezavisni razvoj i isporuka – svaki mikroservis je razvijen zasebno, što se odnosi i na npr. verzioniranje API sučelja, ali i na korištenje programskog jezika/tehnologije – svaki mikroservis može biti napisan u jeziku koji je naprikladniji za konkretnu funkcionalnost i ovisno o znanju razvojnog tima. [1], [3]

Bez obzira na velike prednosti koje ovakava arhitektura donosi prvenstveno razvoju izuzetno velikih softverskih sustava, postoje i nedostatci, odnosno nekoliko faktora na koje svakako treba pripaziti prilikom implementacije:

- kompleksna orkestracija – više servisa znači i više nadzora nad razvojem svakog od njih zasebno, prema tome posebno do izražaja dolaze tzv. DevOps koji konfiguriraju i podešavaju svaki mikroservis tijekom cijelog životnog ciklusa;
- unutar-servisna komunikacija – razdvojeni servisi trebaju imati učinkovit način međusobne komunikacije, a da pritom ne usporavaju rad cjelokupne aplikacije, što se potom odražava na korisničko iskustvo (eng. *user experience*);
- konzistentnost podataka – kao i kod svih drugih oblika distribuiranih sustava, bitno je podesiti dobru konfiguraciju usklađivanja baza podataka kako ne bi došlo do nekonzistentnosti (nepoklapanja) određenih podataka;
- održavanje visoke raspoloživosti – svaki servis mora se zasebno nadgledati i skalirati prema vlasitim potrebama kako bi cjelokupna aplikacija radila s najboljim performansama i imala visoku dostupnost / raspoloživost;

- testiranje – bez obzira na razdvajanje ovisnosti u razvoju, servisi su prije ili kasnije u nekoj mjeri ovisni jedni o drugima, pogotovo kad se radi o slanju/primanju određenih podataka, što definitivno otežava testiranje svakog servisa zasebno. [1]

4. Docker

U ovom poglavlju predstavit ću Docker i na koji način može pridonijeti razvoju i održavanju aplikacija mikroservisne arhitekture te što ih uopće povezuje. Uz primjer odabrane mikroservisne aplikacije prikazat ću ukratko sposobnosti ovog alata te kako nam isti može pomoći pri izgradnji i isporuci sigurnije i stabilnije mikroservisne aplikacije.

Docker kao alat je predstavljen na konferenciji *Python Developers Conference 2013*. godine, od strane Solomona Hykesa, osnivača tvrtke dotCloud. Jedna od najpreciznijih definicija jest:

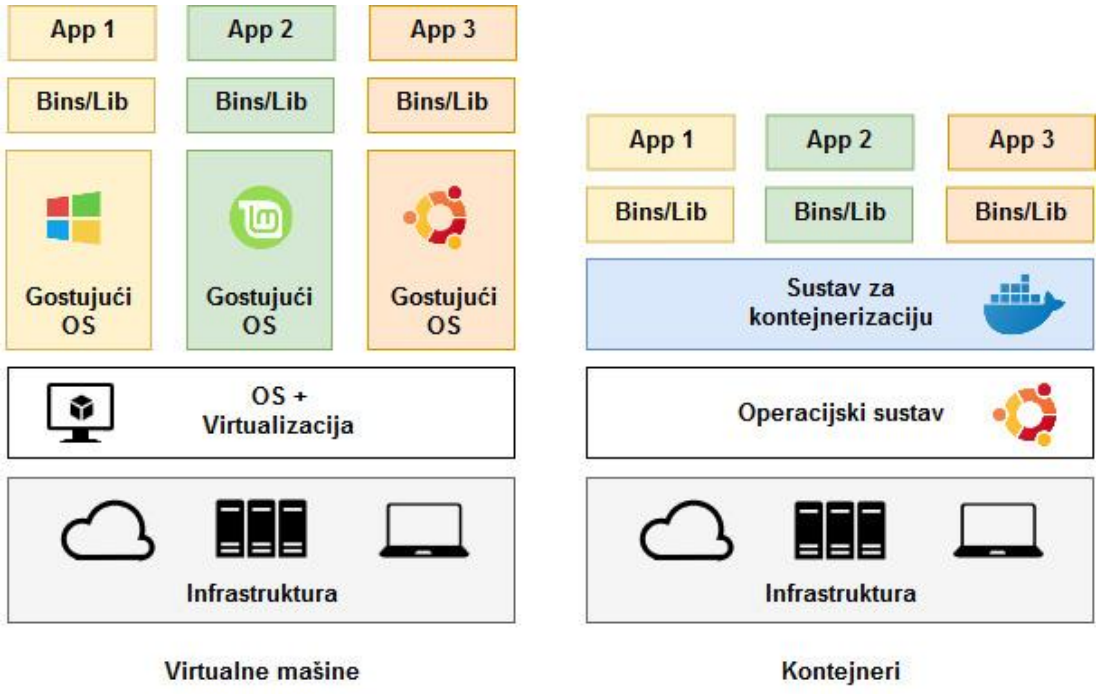
„Docker je alat koji obećava jednostavnu enkapsulaciju procesa kreiranja distribuiranog artefakta bilo koje aplikacije, skalabilne isporuke iste u bilo kakvo okruženje, zadržavajući aerodinamičnost radnog procesa i responzivnost agilnih softverskih organizacija.“ [4] Kako bismo mogli razumjeti na koji način Docker predstavlja rješenje za isporuku aplikacija mikroservisne arhitekture, krenut ćemo upravo od opisa relativno nove tehnologije koja omogućava sve ono što je opisano u teoriji mikroservisa – kontejnera.

4.1. Kontejneri

Kao što je već spomenuto, jedna od temeljnih prednosti je neovisna isporuka, odnosno samostalnost isporuke svakog pojedinog mikroservisa, što istovremeno vuče za sobom i neovisnu skalabilnost servisa. Međutim, svaki od servisa razvija se zasebno, što znači da ima drugačije konfiguracije, zahtjeve od operacijskog sustava i drugačije ovisnosti (eng. *dependencies*). To može postati poprilično nezgodno za konfiguraciju ako se izvodi na jednom fizičkom računalu - postavlja se pitanje kako zapravo omogućiti sigurnu isporuku svakog pojedinog servisa. Očito je da je potrebna veća sposobnost njihove međusobne izolacije. Dakle, jedna mogućnost je da svaki mikroservis isporučujemo na zasebnom fizičkom poslužitelju. Recimo da u vašoj tvrtci radite samo s 200 mikroservisa i neka postoje barem 3 instance od svakog – to je već 600 instanci, odnosno 600 fizičkih poslužitelja, i to samo za produkciju. Sigurno osim produkcije trebate i druga okruženja: testna, razvojna, integracijska, itd – već smo prešli 1000 fizičkih poslužitelja. Nije potrebno ni spominjati koliko je to skupo, ali i teško za održavati. Druga opcija mogu biti virtualne mašine (eng. *virtual machines*, VM), ali čak i u tom slučaju ta brojka je prevelika i koštat će previše. Osim toga, developeri sigurno trebaju imati instance određenih mikroservisa lokalno na vlastitom računalu, što definitivno isključuje ovu metodu jer prosječno računalo u najboljem slučaju može pokrenuti maksimalno 5-10 instanci virtualnih mašina istovremeno. [2]

Ovaj problem riješen je kontejnerima (eng. *containers*) – instancama korisničkog prostora koje omogućuju izolirano okruženje unutar jednog domaćina (eng. *host*), odnosno operacijskog sustava. Programima koji se izvršavaju unutar kontejnera isti izgleda kao obično računalo. Procesi koji se izvršavaju unutar jednog kontejnera mogu pristupiti i koristiti samo one resurse koji su dodijeljeni tom kontejneru i ne mogu vidjeti ništa van istog. Korištenje kontejnera može se nazivati kontejnerizacija (eng. *containerization*), odnosno virtualizacija na razini operacijskog sustava (eng. *operating-system-level virtualization*). [5]

Kontejner se vrlo često povezuje ili čak miješa s pojmom virtualna mašina, kojeg sam zapravo već spomenula – zato je važno da ovdje odmah razložimo osnovne razlike. Virtualne mašine pokreću se unutar programa za virtualizaciju i predstavljaju zapravo nove, zasebne (virtualne) operacijske sustave unutar domaćeg. Prema tome, imaju svoje biblioteke, programe i sve ostalo te zauzimaju i po nekoliko gigabajta memorijskog prostora. S druge strane, kontejneri su zapravo samo izolirana okruženja, unutar domaćeg operacijskog sustava (dakle, ne postoji još jedan sloj operacijskog sustava), ali sadrže vlastite biblioteke za određene procese i aplikacije, što ih memorijski čini puno manjima naspram virtualnih mašina. Kontejnere se ponekad naziva „vrlo laganim virtualnim mašinama“. [6] U nastavku na Slici 2 nalazi se skica koja prikazuje glavne tehničke razlike između virtualnih mašina i kontejnera.



Slika 2. Razlike između virtualnih mašina i kontejnera (Prema: [6])

Zbog malih hardverskih zahtjeva, moguće je imati i stotine kontejnera unutar jednog operacijskog sustava. Upravo zbog te činjenice, kontejneri su idealna tehnologija za rad s mikroservisima – svaki mikroservis možemo izolirati u vlastiti kontejner bez opasnosti da će doći do kolizije u bibliotekama ili zavisnostima drugih mikroservisa.

Najpoznatiji alat za kontejnerizaciju je Docker – koji će se koristiti i u ovom radu. Međutim, osim Dockera, postoji još nekoliko alata koji se koriste sličnim tehologijama: Singularity, Lmctfy, Turbo, itd.

4.2. Osnovni pojmovi

Prije nego krenemo s korištenjem Dockera, objasnit ću osnovnu terminologiju koja će se koristiti u nastavku i nužna je za razumijevanje radnih procesa s Dockerom:

- Docker klijent (eng. *client*) – naredba koja se koristi za kontrolu većine procesa unutar Dockera i za uspostavu kontakta s udaljenim poslužiteljima
- Docker poslužitelj (eng. *server*) – naredba koja se koristi u servisnom (eng. *daemon*) načinu rada: moguća je isporuka, pokretanje i gašenje kontejnera
- Docker slika (eng. *image*) – sastoji se od jednog ili vise slojeva datotečnog sustava i važnih metapodataka koji predstavljaju sve datoteke koje su potrebne kako bi se pokrenula Dockerizirana (eng. *Dockerized*) aplikacija
- Docker kontejner (eng. *container*) – Linux kontejner koji je instanciran iz Docker slike; moguće je kreirati više kontejnera iz iste slike.
- Osnovni domaćin (eng. *atomic host*) – manji, fino podešeni operacijski sustav (npr. CoreOS ili Project Atomic), koji podržava kontejnerizaciju i osnovne nadogradnje operacijskog sustava. [4]

Docker se zasniva na Linux kontejnerima. Prema tome, najčešće se koriste Linux serveri. U ovom radu koristit će se računala s Linux Ubuntu 16.04 64-bitnim operacijskim sustavom.

4.3. Uobičajeni tijek rada

Kontejnerizacija bilo kakvog softvera počinje s Docker slikom. Docker slika (eng. *image*) izgrađuje se na temelju tekstualne datoteke koja se zove *Dockerfile*. ***Dockerfile*** definira što će se nalaziti u okruženju našeg kontejnera – sadrži skup instrukcija kako izgraditi sliku i zapravo prvi je korak u procesu kontejniziranja aplikacije. Svaka instrukcija piše se u zasebnom redu i predstavlja jedan sloj u izgradnji slike.

Prvi red gotovo svakog *Dockerfilea* počinje naredbom FROM. FROM započinje izgradnju nove slike na temelju (bazi) neke druge, postojeće slike. To je najčešće slika koja predstavlja instaliranu inačicu laganog Linuxa s okruženjem koje nam je potrebno, možda

Apache serverom, Javom ili nečim sličnim. Nakon izgrađene baze, koja je najčešće „najdeblja“, slijede koraci kojima prilagođujemo svoje okruženje ili dodajemo svoj softver – vršimo kopiranje određenih podataka, postavljamo varijable okoline, itd. Naposljetku se otvaraju određeni portovi na kojima će aplikacija raditi te pokreće sam softver. Cilj je da slika bude što manja veličinom i da se što brže da izgraditi. Prema tome, važno je da instrukcije pišemo redom, odnosno slažemo slojeve, od onih memorijski najvećih, i za koje su najmanje šanse da će se mijenjati u budućnosti (operacijski sustav, okruženje, kompajleri, itd.), te da na kraju stavljamo najlakše slojeve, tj. one koji će se češće mijenjati. Razlog tomu jest što Docker gradi lokalni *cache* i kada gradi sliku iz *Dockerfilea*, prvo provjerava postoji li određeni sloj već izgrađen i ako da, koristi ga, te na njega slaže druge slojeve. Upravo to omogućuje Dockeru da funkcionira toliko brzo i efikasno – nanovo se izgrađuju samo oni slojevi koji su mijenjani ili nisu postojali prije. [4]

Nakon napisanog *Dockerfile*-a, naredbom *docker build* iz *Dockerfile*-a izgrađuje se Docker **slika**. Docker slika se potom najčešće pohranjuje u odabrani online (*cloud*) repozitorij. Docker po zadanome koristi Docker Hub, ali mogu se koristiti i drugi servisi. Nakon registracije na Docker Hub, moguće je imati javne i jedan privatni repozitorij besplatno. Naredbom *docker login* možemo se prijaviti na svoj račun te nakon toga spremiti sliku na vlastiti repozitorij slično kao u ostalim sustavima za verzioniranje – naredbom *docker push* <*image-tag*>. Kada je slika jednom na repozitoriju, lakše je možemo dohvatiti na bilo kojem drugom računalu s Dockerom, naredbom *docker pull* <*image-tag*>. Pregled lokalno pohranjenih slika možemo vidjeti s *docker image ls*.

Docker slika postaje **kontejner** naredbom *docker run* <*image-tag*>. Može se reći da je kontejner slika u izvršavanju. *Run* naredba zapravo obuhvaća dvije naredbe: *create* (koja stvara kontejner) i *start* (koja pokreće kontejner). Kontejner se počinje izvršavanjem naredbe koja je posljednja napisana u *Dockerfile*-u, tj. pokrenuta kao RUN ili ENTRYPOINT. To je najčešće naredba koja pokreće određenu aplikaciju ili servis. Kontejner se izvršava koliko god se izvršava i proces unutar njega. Kada je proces gotov i kontejner se gasi. Kontejner se može ponovno pokrenuti naredbom *docker container start* <*hash*>, zaustaviti s *docker container stop* <*hash*> ili prisilno terminirati s *docker container kill* <*hash*>. [4]

Važno je napomenuti da svi objekti (tj. slike i kontejneri) prilikom kreiranja dobiju unikatni id (hash vrijednost), prema kojem se identificiraju. Kako bi se lakše rukovalo s objektima, preporučljivo je slikama dodavati oznake (eng. *tag*), a kontejnerima prepoznatljive nazive (eng. *name*).

Svi navedeni postupci kasnije će biti prikazani i demonstrirani u praktičnom dijelu rada.

5. Kubernetes

Kubernetes (često prikazan kraticom kao K8s) je besplatan, *open-source* alat za orkestraciju i razvoj kontejneriziranih aplikacija. Razvijen je od strane Google-a 2014. godine i predstavlja čitavu platformu za automatiziranu produkciju, skaliranje i upravljanje kontejniziranim aplikacijama smještenih na velikom broju virtualnih ili fizičkih računala. Konačni cilj je omogućiti izgradnju i produkciju pouzdanog i skalabilnog distribuiranog softvera. [7]

Osim Kubernetesa, postoje i drugi orkestracijski alati, kao npr. Azure Container, Marathon, Amazon ECS, Docker Swarm ili Google Container Engine. U nastavku najprije ću objasniti kako je Kubernetes sustav organiziran i način na koji radi, a kasnije ću prikazati njegovu primjenu na odabranom primjeru.

5.1. Organizacija sustava

U Kubernetes sustavu, računala su organizirana u klaster. Klaster (eng. *cluster*) predstavlja skup virtualnih ili fizičkih računala, odnosno skup procesorskih, memorijskih i mrežnih resursa, koji funkcioniraju kao jedno računalo, najčešće unutar iste LAN mreže. Svaki Kubernetes klaster sastoji se od minimalno jednog računala koji ima ulogu glavnog čvora (eng. *master node*, radi lakšeg snalaženja, u daljnjem tekstu - master) te jednog ili više čvorova (eng. *node*). Kao korisnik (sistem administrator) praktički nikad nećete komunicirati s čvorovima, već uvijek s Kubernetes masterom, a master onda raspodjeljuje zadatke na čvorove. Čvorovi su samo računalna snaga (infrastruktura), a Kubernetes se brine da se ona racionalno iskoristi prema zadanim zahtjevima. [8]

Za pristup Kubernetes masteru postoji Kubernetes API, kojim se može upravljati putem komandne linije (CLI – Kubernetes klijent, *kubectl*) ili putem grafičkog sučelja (UI).

5.2. Osnovni objekti

Najmanja gradivna jedinica u Kubernetes sustavu je **ljuska** (eng. *pod*). Ljuska je najjednostavniji objekt i osnovna jedinica isporuke. Predstavlja određeni proces u izvođenju te služi kao omotač oko jednog ili više kontejnera, a izvršava se na čvoru unutar klastera. Najčešći alat za kontejnerizaciju je Docker, koji se koristi i u ovom radu. Kontejneri unutar ljuske dijele određene resurse, kao što su dijeljeni memorijski prostor za pohranu podataka, istu IP adresu u mreži te informacije o tome kako se neki kontejner pokreće, koje portove koristi itd. Ljuske imaju životni ciklus – kreiraju se i umiru. Kreira ih korisnik ili kontroler prema

rasporedu, te traje dok god se u njemu izvršava proces, tj. dok ne dođe do terminiranja procesa, dok se ljuska ne obriše, dok ljuska ima dovoljno resursa za izvršavanje ili dok se čvor ne sruši. Ljuske se same po sebi ne mogu oporavljati niti ponovno pokrenuti, već to čine različiti kontroleri koji upravljaju životnim ciklusima ljuski. [8]

Servis (eng. *service*) je apstraktni objekt koji predstavlja logičnu skupinu ljuski i pravila kako im se pristupa. To je zapravo sinonim za mikroservise o kojima govorimo. Činjenica je da ćemo sigurno imati nekoliko ljuski (duplikata) određenog softvera, apstrahiranog kao (mikro)servis pod određenim portom. Važno je napomenuti da ljuske nisu vezane za čvor, već nakon završetka životnog ciklusa (prestanak rada procesa, terminiranje ili bilo kakva nezgoda), Kubernetes sustav može istu ljusku ponovno podignuti na nekom drugom čvoru – u tom slučaju ljuska bi imala drugačiju IP adresu. Međutim, upravo povezivanje ljuski u servis omogućuje nam da uvijek imaju istu IP adresu, neovisno na kojem se računalu nalaze te njihov broj. Na taj način možemo mijenjati i skalirati broj ljuski unutar servisa, čak i mijenjati interni broj porta, bez da se o tome brinu drugi mikroservisi – oni uvijek komuniciraju na razini servisa, a ne na razini ljuske. Servis prepoznaje svoje ljuske prema oznaci (eng. *label*), što ćemo vidjeti i kasnije na samom primjeru. [8]

Nositelj podataka (eng. *volume*) je dijeljeni memorijski prostor za pohranu podataka unutar ljuske. Naime, podaci koji su zapisani unutar kontejnera su prolazni – ako se kontejner sruši, on se može ponovno pokrenuti, ali svi podaci su izgubljeni jer se kontejner uvijek pokreće iz početnog (čistog) stanja. Drugo, kada se koristi više kontejnera unutar jedne ljuske, vrlo je velika vjerojatnost da ta dva kontejnera trebaju koristiti, tj. dijeliti isti skup podataka, a to se postiže upravo ovim Kubernetes objektima. Dakle, nositelji podataka žive koliko i same ljuske, i jedna ljuska može imati više definiranih nositelja podataka paralelno. Hijerarhijski gledano, proces unutar kontejnera vidi sustav datoteka počevši od Docker predloška kao korijen hijerarhije. Zatim se nositelji montiraju na određenim putanjama unutar predloška. [8]

Imenski prostor (eng. *namespace*) je naziv za virtualni klaster unutar istog fizičkog klastera (koji može imati više virtualnih). Imenski prostori namijenjeni su okolinama s većim brojem računala, korisnika i projekata, kako bi se omogućilo bolje definiranje imena resursa (potrebno je imati unikatna imena za pojedine resurse/objekte unutar imenskog prostora). U Kubernetes sustavu prema zadanom postoje tri imenska prostora: *default* (zadani imenski prostor za sve objekte bez drugog imenskog prostora), *kube-system* (namijenjen za objekte kreirane od strane Kubernetes sustava) i *kube-public* (automatski kreiran, dostupan svim korisnicima i vidljiv kroz cijeli klaster). [8]

Postoje još brojni drugi objekti unutar sustava, ali nećemo ih posebno opisivati. Ostali objekti koji se budu spominjali u radu bit će opisani u trenutku korištenja.

5.3. Arhitektura sustava

Kubernetes klaster čine čvorovi i jedan (ali može i više) mastera, na kojima se izvršavaju osnovni Kubernetes procesi te egzistiraju određeni Kubernetes objekti.

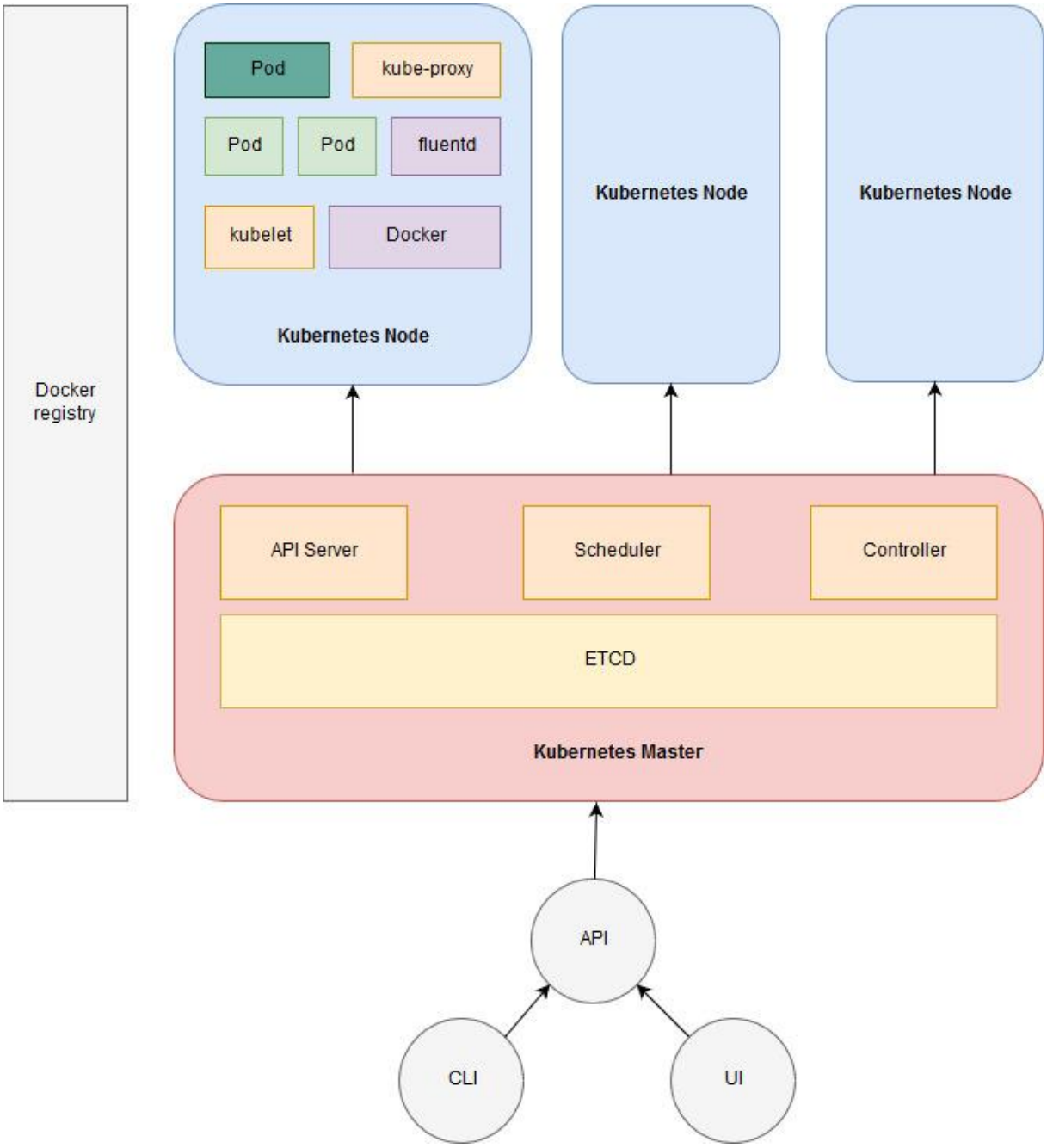
Kubernetes master sastoji se od četiri glavna dijela:

- API server (*kube-apiserver*) – proces koji se brine o validaciji i konfiguraciji podataka za API objekte koji uključuju ljuske, servise, replikacijske kontrolere, itd. Pruža REST operacije za trenutno stanje klastera, preko kojeg komuniciraju sve ostale komponente;
- kontroler (*kube-controller-manager*) – proces koji sadrži temeljne kontrolne petlje za provjeravanje stanja sustava. Dobiva informacije o trenutnom stanju sustava preko API servera i uspoređuje ih sa željenim stanjem te po potrebi radi promjene kako bi sustav došao u željeno stanje. Postoji nekoliko kontrolera unutar Kubernetes sustava: *replication controller, endpoints controller, namespace controller* i *serviceaccounts controller*;
- raspoređivač (*kube-scheduler*) – proces koji nadgleda i brine se o raspoloživosti, performansama i kapacitetu svih resursa koji su na raspolaganju u klasteru, prema zadanim pravilima. Brine se o rasporedu zadataka po raspoloživim čvorovima;
- ETCD – predstavlja temeljni gradivni blok Kubernetes sustava i sadrži informacije o željenom stanju sustava. [8]

Kubernetes čvor sastoji se od sljedećih dijelova:

- kubelet – temeljni Kubenetes proces za komunikaciju s masterom, osigurava zdravlje kontejnera unutar ljuski;
- kube-proxy – proces koji osigurava mrežne veze između čvorova;
- fluentd – proces koji se upravlja bilješkama povijesti sustava (eng. *log*);
- ljuska (jedna ili više);
- Docker instanca. [8]

Slika 3 prikazuje osnovnu arhitekturu Kubernetes sustava. Kompletan proces i scenarije korištenja objasnit ću kroz primjenu u sljedećem poglavlju.



Slika 3. Arhitektura Kubernetes sustava

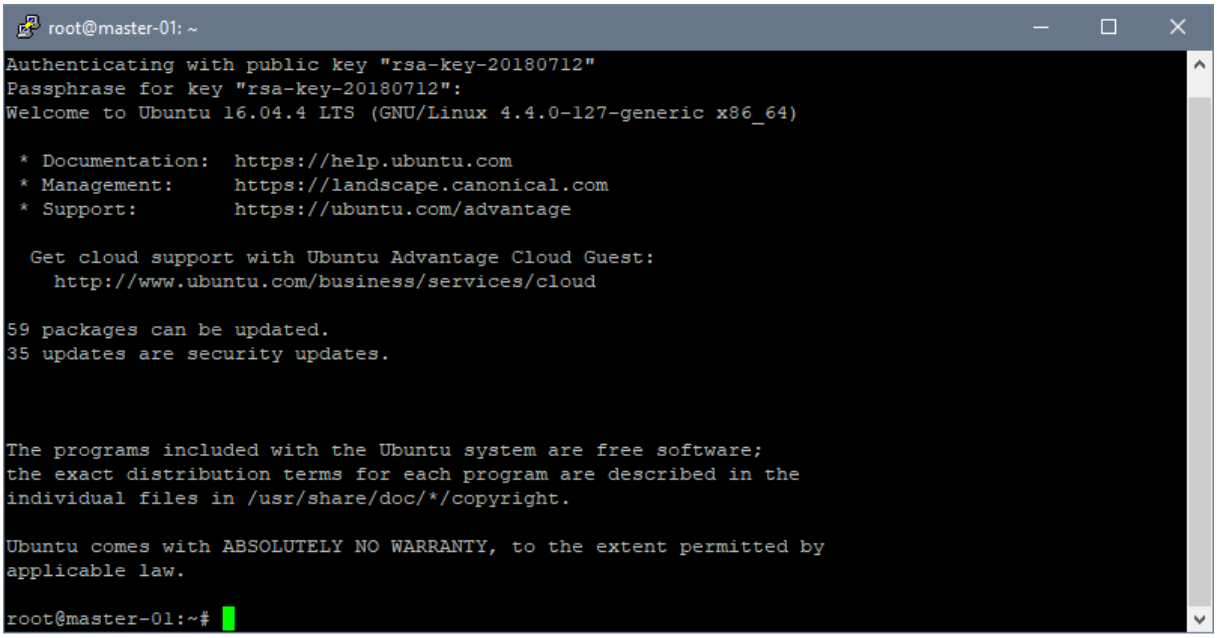
6. Isporuka mikroservisne aplikacije

U ovom poglavlju pokazat ćemo navedene tehnologije u njihovoj primjeni. Kao prvi primjer softvera mikroservisne arhitekture za analizu uzet ću jednostavan primjer s GitHub repozitorija (https://github.com/janakiramm/todo-app). Radi se o jednostavnoj „to-do“ aplikaciji koja se sastoji od dva dijela: prvi je baza podataka (MongoDB) kao jedan mikroservis te web aplikacija (NodeJS) kao drugi mikroservis.

6.1. Postavljanje Kubernetes klastera

Kao što je već spomenuto, za demonstraciju navedenih tehnologija koristit ću računala s platforme Digital Ocean. Koristit ću četiri računala (u Digital Ocean kontekstu: *droplet*-a), s jednakim specifikacijama: 1 CPU jezgra, 2GB RAM, 50GB SSD, s Ubuntu 16.04.4 64-bit operacijskih sustavom, lociranih u centru u Frankfurtu. Ta četiri računala činit će jedan Kubernetes klaster (jedan master i tri čvora).

Prilikom kreiranja *dropleta* na Digital Oceanu, poželjno je radi bolje sigurnosti dodati SSH javni ključ, te se prilikom prijave autorizirati samo još s parafrazom. Nakon kreiranja, slijedi konfiguracija svakog od njih. SSH vezom se spajam najprije na master (budući da radim u Windows okruženju, koristim alat Putty), kao što je vidljivo na Slici 4.



Slika 4. SSH spajanje na master čvor

Najprije provodim potrebna ažuriranja na Linux operacijskom sustavu:

```
export MASTER_IP=207.154.253.160
apt-get update && apt-get upgrade -y
```

Nakon toga preuzimam potrebne pakete za instalaciju Kubernetesa:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key
add -
cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update -y
```

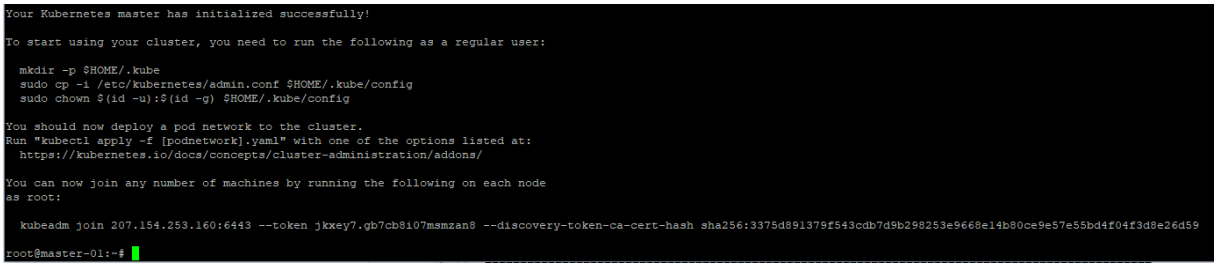
Zatim provodim instalaciju Dockera i svih komponenti Kubernetes sustava:

```
apt-get install -y docker.io
apt-get install -y kubelet kubeadm kubectl kubernetes-cni
```

Nakon uspješne instalacije, možemo inicijalizirati Kubernetes klaster:

```
kubeadm init --pod-network-cidr=10.244.0.0/16  --apiserver-advertise-
address $MASTER_IP
```

Nakon uspješne inicijalizacije, prikazana je poruka o uspješnosti kao na Slici 5.



Slika 5. Rezultat *kubeadm init* naredbe

Vrlo je važno spremiti navedeni token i discovery-token na sigurno mjesto jer ih se ne može naknadno ispisati, a prijeko su potrebni za spajanje na kreirani klaster. Međutim, prije nego krenemo u dodavanje novih čvorova klasteru, najprije dodajem još Flannel – servis koji je neophodan za pravilno konfiguriranje i korištenje mrežnih postavki za Kubernetes (umjesto njega može se koristiti npr. Calico ili neki drugi okvir, ali Flannel se najviše koristi uz Kubernetes):

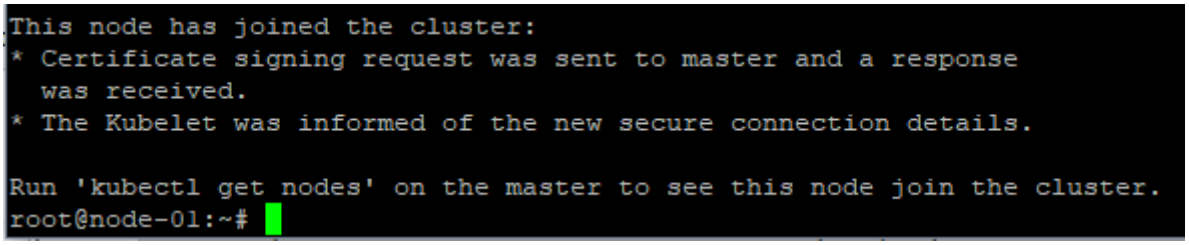
```
curl -sSL
"https://github.com/coreos/flannel/blob/master/Documentation/kube-
flannel.yml?raw=true" | kubectl --namespace=kube-system create -f -
```

Isto tako, dodajem i Kubernetes kontrolnu ploču (eng. *dashboard*), odnosno grafičko sučelje za Kubernetes:

```
kubectl create -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy
/recommended/kubernetes-dashboard.yaml --namespace=kube-system
```

S uspješnom instalacijom kontrolne ploče završili smo s master čvorom, te se SSH vezom spajamo na ostale čvorove i provodimo sličan postupak. Provodimo ažuriranja operacijskog sustava, paketa te instalaciju Dockera i Kubernetesa. Nakon toga, potrebno je samo još izvršiti naredbu za dodavanje čvora postojećem klasteru. To radimo naredbom *kubeadm join* te specificiranjem IP adrese master čvora te (kao što je navedeno i u uputama na Slici 4.) Važno je pripaziti na sljedeće: token koji je generiran naredbom *kubeadm init* validan je 24 sata od kreiranja, što znači da nećemo moći s istim tokenom dodati nove čvorove nakon isteka tog vremena. Za upravljanje tokenima možemo koristiti *kubeadm token generate* (za generiranje samog tokena), *kubeadm token create <token>* (za dodavanje generiranog tokena), *kubeadm token delete <token>* (za brisanje tokena) i *kubeadm token list* (za prikaz svih dodanih tokena). S druge strane, generirani discovery-token nije moguće ponovno generirati, a niti prikazati stari. Ukoliko ga nismo spremili, i dalje se možemo spojiti na master, ali veza nije u potpunosti sigurna i ne preporuča se. Ukoliko smo u mogućnosti, možemo napraviti *kubeadm reset* i resetirati sve što je učinjeno naredbama *kubeadm init* i *kubeadm join*, te krenuti ispočetka s inicijalizacijom klastera.

Ako smo uspješno generirali/spremili sve potrebne tokene, nakon provedbe naredbe *kubeadm join*, trebali bismo vidjeti poruku o uspješnosti kao što je prikazano na Slici 6.



Slika 6. Čvor je dodan Kubernetes klasteru

6.2. Kontejnerizacija mikroservisa

Prvi korak jest pakiranje NodeJS aplikacije u vlastiti kontejner. Dockerfile bi mogao izgledati ovako:

```
FROM node:slim
LABEL maintainer = "jani@janakiram.com"
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY ./app/ ./
RUN npm install
CMD ["node", "app.js"]
```

Dakle, baza ove slike jest *node-slim* – najmanje Linux okruženje s već instaliranim NodeJS-om. Nakon toga se (opcionalno) postavlja oznaka autora slike. Naredbom RUN možemo izvršiti bilo koju drugu naredbu unutar Linuxa, pa prema tome kreiramo potrebnu mapu za pohranu aplikacijskog koda, te sa WOKRDIR se pozicioniramo u isti folder. Naredbom COPY kopiramo sadržaj direktorija *app* u trenutni direktorij. Nakon toga s RUN pokrećemo instalaciju npm paketa, odnosno svih ovisnosti (eng. *dependency*). I u konačnici, naredbom CMD pokrećemo samo aplikaciju. Svaki napisani red predstavlja jednu instrukciju i jedan sloj pri izgradnji kontejnera. Kontejner izgrađujemo naredbom *docker build*, kao što je prikazano na Slici 7.

```
root@master-01:~/todo-app# docker build -t acerinth/todo-app .
Sending build context to Docker daemon 8.252 MB
Step 1/7 : FROM node:slim
slim: Pulling from library/node
5bba3ecb4cd6: Pull complete
196b8e3c919d: Pull complete
7d083412657b: Pull complete
66713c19e320: Pull complete
34a7a8e7f117: Pull complete
Digest: sha256:7a71fcfbe8d721ceb620f89fd6abf7f3a586346f40de5293357713b535b6b664
Status: Downloaded newer image for node:slim
 ---> 441869a4d9e2
Step 2/7 : LABEL maintainer = "jani@janakiram.com"
 ---> Running in 8dd4e7a2979e
 ---> 8b8f14507627
Removing intermediate container 8dd4e7a2979e
Step 3/7 : RUN mkdir -p /usr/src/app
 ---> Running in 2ac355abc676
 ---> 8e061d9d072d
Removing intermediate container 2ac355abc676
Step 4/7 : WORKDIR /usr/src/app
 ---> d813b1b6e8bb
Removing intermediate container d2f96e68c791
Step 5/7 : COPY ./app/ ./
 ---> 9697bba3f1a0
Removing intermediate container efd023ade9a4
Step 6/7 : RUN npm install
 ---> Running in 850ad30c6fb4
npm notice created a lockfile as package-lock.json. You should commit this file.
audited 158 packages in 2.086s
found 36 vulnerabilities (12 low, 15 moderate, 9 high)
  run `npm audit fix` to fix them, or `npm audit` for details
 ---> 481ab96ac3be
Removing intermediate container 850ad30c6fb4
Step 7/7 : CMD node app.js
 ---> Running in dc638ca80265
 ---> 9ce841c5143e
Removing intermediate container dc638ca80265
Successfully built 9ce841c5143e
```

Slika 7. Izgradnja slike

Koristila sam pritom zastavicu –t kako bih odmah dodala oznaku (eng. *tag*) slici, da je mogu lakše pratiti i odmah postaviti na Docker Hub. Dakle, ovo što možemo vidjeti na slici jest kako Docker, tj. po kojim koracima izvršava naš Dockerfile. Prvo povlači postojeću sliku node-slim s Docker Huba, jer je nema lokalno. Nakon toga redom kreira slojeve i dodaje na postojeće

slike. U konačnici vidimo da je slika uspješno izgrađena, što možemo i provjeriti naredbom *docker images*, kao što vidimo na Slici 8.

```
root@master-01:~/todo-app# docker images
REPOSITORY                                TAG                 IMAGE ID            CREATED             SIZE
acerinth/todo-app                         latest              9ce841c5143e        4 minutes ago       188 MB
node                                      slim                441869a4d9e2        16 hours ago        183 MB
acerinth/discovery-image                  v1.0                22cbe77f7e8b        5 days ago          145 MB
k8s.gcr.io/kube-proxy-amd64               v1.11.1             d5c25579d0ff        4 weeks ago         97.8 MB
k8s.gcr.io/kube-controller-manager-amd64  v1.11.1             52096ee87d0e        4 weeks ago         155 MB
k8s.gcr.io/kube-scheduler-amd64           v1.11.1             272b3a60cd68        4 weeks ago         56.8 MB
k8s.gcr.io/kube-apiserver-amd64           v1.11.1             816332bd9d11        4 weeks ago         187 MB
k8s.gcr.io/coredns                        1.1.3               b3b94275d97c        2 months ago        45.6 MB
k8s.gcr.io/etcd-amd64                     3.2.18              b8df3b177be2        4 months ago        219 MB
k8s.gcr.io/kubernetes-dashboard-amd64     v1.8.3              0c60bcf89900        6 months ago        102 MB
quay.io/coreos/flannel                    v0.10.0-amd64       f0fad859c909        6 months ago        44.6 MB
k8s.gcr.io/pause                          3.1                 da86e6ba6ca1        7 months ago        742 kB
```

Slika 8. Docker images

Idući je korak postavljanje kontejnera na Docker Hub, kako bi ga kasnije Kubernetes čvor mogao povući. Ako smo uspješno prijavljeni s vlastitim Docker Hub računom, možemo napraviti *docker push*, kao što je prikazano na Slici 9.

```
root@master-01:~/todo-app# docker push acerinth/todo-app
The push refers to a repository [docker.io/acerinth/todo-app]
7e102a24f08c: Pushed
e6cb8eecf739: Pushed
431b77a67fd5: Pushed
c16360204299: Mounted from library/node
1cae17cf0b05: Mounted from library/node
7c6ab5fb7059: Mounted from library/node
ed271eac23e9: Mounted from library/node
156ff16f37e4: Mounted from library/node
latest: digest: sha256:1fbfe78943ea34e19a4210fc44a58bdbbe6dcb17ab66be39494c473194822c06 size: 1996
root@master-01:~/todo-app#
```

Slika 9. Docker push

6.3. Konfiguracija ljuski i servisa

Svaki Kubernetes objekt opisan je YAML datotekom. Prvi objekt koji ćemo kreirati jest ljuska za bazu podataka. Db-pod.yaml može izgledati ovako:

```
apiVersion: v1
kind: Pod
metadata:
  name: db
  labels:
    name: mongo
    app: todoapp
spec:
      containers:
      - image: mongo
        name: mongo
        ports:
```

```
      - name: mongo
        containerPort: 27017
      volumeMounts:
        - name: mongo-storage
          mountPath: /data/db
    volumes:
      - name: mongo-storage
        hostPath:
          path: /data/db
```

Ovdje je bitno naglasiti nekoliko stavki. Prva je svostvo *Kind*. *Kind* označava vrstu Kubernetes objekta – to može biti *Pod, Service, Deployment, Job, ReplicaSet*, itd... U ovom slučaju ručno radimo *Pod*. *Metadata* sadrži naziv ljuske (*name*) i ono što je važno su oznake (*labels*) – prema tim oznakama Kubernetes zna koje ljuske će pripadati kojem servisu, odnosno kojoj aplikaciji. U *spec* svojstvu definiramo kontejner. Kao što vidimo, kontejner možemo definirati i kreirati na licu mjesta kroz Kubernetes – budući da nemamo potrebu za ikakvom specifičnom slikom, koristimo onu najobičniju za MongoDB (*mongo*), te važno je specificirati na kojem portu leži aplikacija. *VolumeMounts* predstavlja spremnik podataka o kojem je također već bilo riječi, te se tu specificira putanja do datoteke gdje će sve biti spremljeno. Kako bismo kreirali ljusku, izvršavamo *kubectl create* na Kubernetes masteru, kao što vidimo na Slici 10.

```
root@master-01:~/todo-app/kubernetes# kubectl create -f db-pod.yaml
pod/db created
```

Slika 10. Kubectl create

Ljuska samo izvršava zadani proces unutar kontejnera. Ono što treba još napraviti kako bi sve funkcioniralo jest napraviti servis:

```
apiVersion: v1
kind: Service
metadata:
  name: db
  labels:
    name: mongo
    app: todoapp
spec:
  selector:
    name: mongo
  type: ClusterIP
  ports:
    - name: db
```

```
    port: 27017
    targetPort: 27017
```

Ovdje vidimo kako za Kind postavljamo Service. Servis prepoznaje sve ljuske u istom imenskom prostoru koji se podudaraju prema odabranom selektoru, u ovom slučaju to je name: mongo. Type označava tip IP adrese – ClusterIP znači da je IP adresa servisa dostupna samo unutar klastera, što je logično budući da baza ne smije biti dostupna vanjskom svijetu. I na kraju, postavljaju se portovi – targetPort označava port na ljusci, koji se mapira na odabrani port na servisu. Naredbom *kubectl create* kreiramo servis.

Slično ovome, kreiramo još dvije YAML datoteke: za web frontend ljusku i servis. Ovako će izgledati datoteka za opis ljuske:

```
apiVersion: v1
kind: Pod
metadata:
  name: web
  labels:
    name: web
    app: todoapp
spec:
  containers:
    - image: acerinth/todo-app
      name: myweb
      ports:
        - containerPort: 3000
```

Više manje sve je isto kao u prethodnom primjeru, samo što ovdje za sliku kontejnera postavljamo vlasititi repozitorij s Docker Huba. Aplikacija radi na portu 3000 pa je tako i postavljeno. Nadalje, specificirmo i YAML datoteku za web servis:

```
apiVersion: v1
kind: Service
metadata:
  name: web
  labels:
    name: web
    app: todoapp
spec:
  selector:
    name: web
  type: NodePort
  ports:
```

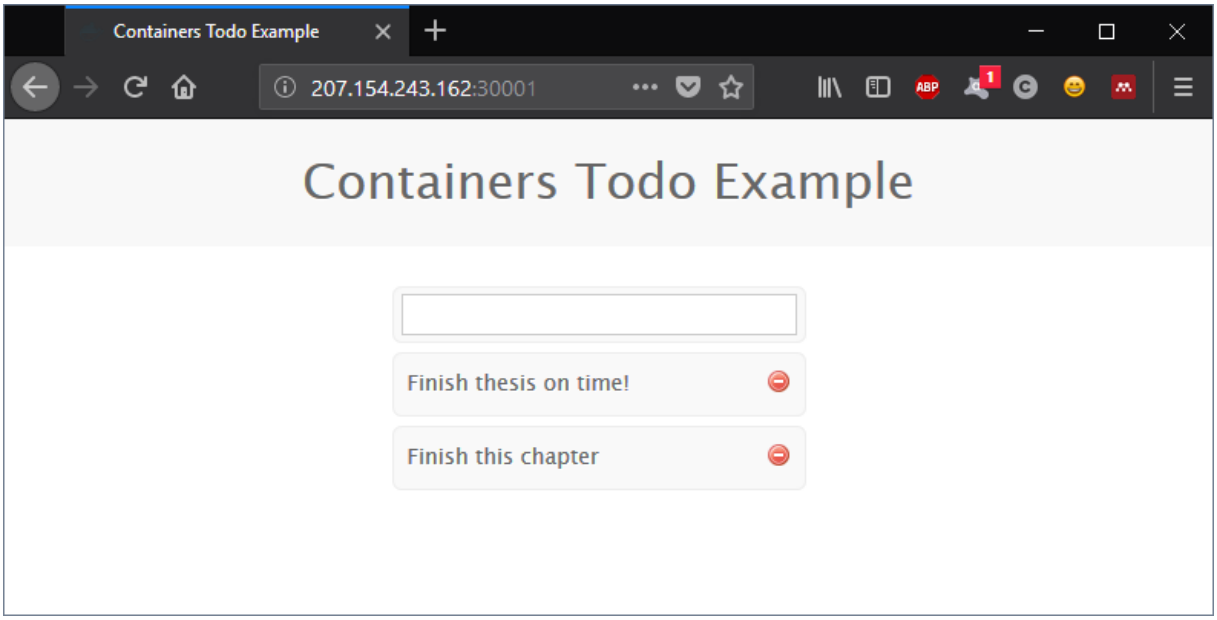
```
- name: http
  port: 3000
  targetPort: 3000
  nodePort: 30001
  protocol: TCP
```

Ovdje je jedna stvar bitna za naglasiti – type jest *NodePort*. To znači da će aplikacija biti dostupna na svakom čvoru pod portom definiranim kao nodePort (30001). Kreiramo još ova dva objekta i sada možemo provjeriti da je sve u redu naredbama *kubectl get* (Slika 11).

```
root@master-01:~/todo-app/kubernetes# kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
db        1/1       Running   0          56m
web       1/1       Running   0          56m
root@master-01:~/todo-app/kubernetes# kubectl get services
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
db           ClusterIP   10.99.199.198   <none>        27017/TCP        56m
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP          22h
web          NodePort    10.99.236.172   <none>        3000:30001/TCP   55m
```

Slika 11. Kubectl get pods i kubectl get services

Dakle, ono što sad imamo jest isporučena aplikacija. Možemo uzeti IP adresu bilo kojeg čvora u klasteru i poslati zahtjev prema portu 30001 – trebali bismo dobiti frontend naše web aplikacije (tj. web mikroservis), skupa s funkcionalnošću spremanja spremljenih zadataka (za koje je zadužen drugi mikroservis), kao što je vidljivo na Slici 12.



Slika 12. To-do aplikacija u izvođenju

Bitno je napomenuti sljedeće: jako je važno kojim redoslijedom pokrećemo ljuske, odnosno kontejnere unutar njih. Treba krenuti od mikroservisa koji su neovisni o drugima, odnosno o kojima drugi ovise – najčešće su to baze podataka, poslužitelji za otkrivanje drugih servisa (eng. *discovery server*), konfiguracijski serveri (eng. *configuration server*), ostali backend servisi pa tek onda web frontend. U suprotnom, velike su šanse da se ljuska za npr. web frontend neće ni pokrenuti, odnosno da će se kontejner, a s njime i ljuska, stalno rušiti, jer aplikacija ne može pronaći određeni servis. Isto se može dogoditi i ako nisu dobro konfigurirani portovi, tako da kod kompleksnijih aplikacija vrijedi malo više vremena uložiti u arhitekturu i pravilno konfigurirati sve mikroservise. U svakom slučaju, Kubernetes je opremljen s dovoljno naredbi koje nam mogu pomoći u otkrivanju grešaka i dijagnosticiranju kvarova. Jedna od njih je i *kubectl log*, s kojom možemo provjeriti log zapise određene ljuske, kao što je prikazano na Slici 13.

```
root@master-01:~/todo-app/kubernetes# kubectl log web
log is DEPRECATED and will be removed in a future version. Use logs instead.
Express server listening on port 3000
GET / 200 54.721 ms - 658
GET /stylesheets/screen.css 200 5.498 ms - 6395
GET /javascripts/ga.js 200 1.703 ms - 323
POST /create 302 39.458 ms - 46
GET / 200 8.757 ms - 899
GET /stylesheets/screen.css 304 1.302 ms - -
GET /javascripts/ga.js 304 1.087 ms - -
GET /images/delete.png 200 1.666 ms - 715
POST /create 302 4.899 ms - 46
GET / 200 5.676 ms - 1148
GET /stylesheets/screen.css 304 1.759 ms - -
GET /javascripts/ga.js 304 1.900 ms - -
GET /images/delete.png 304 0.797 ms - -
GET /destroy/5b76a02742a4190100ab1005 302 8.872 ms - 46
GET / 200 3.953 ms - 907
GET /javascripts/ga.js 304 1.475 ms - -
GET /stylesheets/screen.css 304 1.600 ms - -
GET /images/delete.png 304 0.900 ms - -
GET /destroy/5b76a02f42a4190100ab1006 302 5.071 ms - 46
GET / 200 3.328 ms - 658
```

Slika 13. Kubectl log

6.4. Replikacija mikroservisa

Pod pojmom replikacija (eng. *replication*) podrazumijevamo povećanje broja istog mikroservisa (ljuske). Replicirati mikroservise želimo iz više razloga:

- reduncancija (eng. *redundancy*) – više servisa znači da možemo tolerirati ako jedan zakaže, jer će drugi i dalje raditi;
- skaliranje (eng. *scale*) – više servisa znači da isti mogu prihvatiti i obraditi više korisničkih zahtjeva;
- usitnjavanje (eng. *sharding*) – različiti mikroservisi mogu podržati različite dijelove obrade paralelno. [7]

Jednostavna replikacija mikroservisa je jedna od najvećih prednosti općenito mikroservisne arhitekture, ali i Kubernetes sustava. Ukoliko dolazi do pojačanog prometa prema određenom servisu, sve što treba napraviti jest povećati broj ljuski određenog servisa. Prvo nam treba objekt koji će se brinuti o replikacijama ljuske – *ReplicaSet* ili *ReplicationController*. U ovom primjeru možemo vidjeti YAML datoteku za ReplicaSet:

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: web
  labels:
    name: web
    app: todoapp
spec:
  replicas: 2
  template:
    metadata:
      labels:
        name: web
    spec:
      containers:
      - name: web
        image: acerinth/todo-app
        ports:
        - containerPort: 3000
```

Dakle, u *specs* dijelu specificiramo željeno stanje replika (duplikata). U *template* svojstvu zapravo definiramo objekt ljuske na isti način kao i do sada. Obrisat ćemo web ljusku koju smo ručno kreirali s *kubectl delete*, i potom kreirati ReplicaSet. Kubernetes prema zadanom stanju odmah počinje kreirati dvije ljuske web servisa, kao što je prikazano na Slici 14.

```
root@master-01:~/todo-app/kubernetes# kubectl delete pod web
pod "web" deleted
root@master-01:~/todo-app/kubernetes# kubectl create -f web-rs.yaml
replicaset.extensions/web created
root@master-01:~/todo-app/kubernetes# kubectl get pods
NAME        READY     STATUS              RESTARTS   AGE
db          1/1       Running             0          1d
web-7jj58   0/1       ContainerCreating   0          10s
web-sqg4h   0/1       ContainerCreating   0          10s
```

Slika 14. Replica set

Nakon par sekundi, ljuske su kreirane i pokrenute. Međutim, ovdje sad možemo vidjeti snagu Kubernetesa – što se dogodi kad namjerno terminiramo jednu od ljuski? Kubernetes će automatski prema zadanom stanju u ReplicaSetu početi kreirati novu, kao što možemo vidjeti na Slici 15.

```
root@master-01:~/todo-app/kubernetes# kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
db          1/1       Running   0          2d
web-7jj58   1/1       Running   0          2h
web-sqg4h   1/1       Running   0          2h
root@master-01:~/todo-app/kubernetes# kubectl delete pod web-7jj58
pod "web-7jj58" deleted
root@master-01:~/todo-app/kubernetes# kubectl get pods
NAME        READY     STATUS              RESTARTS   AGE
db          1/1       Running             0          2d
web-7jj58   1/1       Terminating         0          2h
web-sgsc4   0/1       ContainerCreating   0          24s
web-sqg4h   1/1       Running             0          2h
root@master-01:~/todo-app/kubernetes# kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
db          1/1       Running   0          2d
web-sgsc4   1/1       Running   0          41s
web-sqg4h   1/1       Running   0          2h
root@master-01:~/todo-app/kubernetes#
```

Slika 15. Automatsko kreiranje nove ljuske

Naravno, prema potrebi uvijek možemo dinamički skalirati i replicirati ljuske. Možemo mijenjati direktne postavke unutar YAML datoteke ReplicaSeta i ponovno kreirati cijeli objekt, ali možemo to učiniti i preko komandne linije uz pomoć komande *kubectl scale*, kao što je vidljivo na Slici 16. Kubernetes automatski uspoređuje trenutno stanje (2 replike) i željeno stanje (5 replika) i prema tome automatski kreira dodatne ljuske.

```
root@master-01:~/todo-app/kubernetes# kubectl get replicaset
NAME      DESIRED   CURRENT   READY     AGE
web       2         2         2         2h
root@master-01:~/todo-app/kubernetes# kubectl scale rs/web --replicas=5
replicaset.extensions/web scaled
root@master-01:~/todo-app/kubernetes# kubectl get pods
NAME        READY     STATUS              RESTARTS   AGE
db          1/1       Running             0          2d
web-2ctzk   0/1       ContainerCreating   0          6s
web-ndx58   0/1       ContainerCreating   0          6s
web-sgsc4   1/1       Running             0          8m
web-sqg4h   1/1       Running             0          2h
web-zw2bb   0/1       ContainerCreating   0          6s
root@master-01:~/todo-app/kubernetes#
```

Slika 16. Povećavanje broja replikacija

Ovdje ću još jednom napomenuti jednu stvar – cilj cijelog sustava jest da nismo ovisni o zasebnim računalima (čvorovima) i da se ne brinemo o tome gdje se točno (na kojem čvoru) nalazi koji servis. Jer, kao što vidimo, na ovim ispisima to nije ni prikazano – dakle, nije bitno na kojem računalu se izvode koje ljuske (servisi). Računala služe samo kao resursi, a Kubernetes sustav raspoređuje servise prema čvorovima i sam se čak brine o njihovom statusu, zdravlju te da se sve izvodi i da je dostupno u onakvom stanju kakvog ste opisali. Međutim, naravno da možemo doći i do specifičnih podataka o tome što se događa s pojedinim objektom. Za to nam služi naredba *kubectl describe*, koju možemo vidjeti na Slici 17.

Činjenice koje možemo izvući iz ovog ispisa o ljusci, osim osnovih svojstava kao što su naziv, imenski prostor, vrijeme kreiranja i sl, jest i koji objekt koji kontrolira ovu ljusku (ReplicaSet), detalje o kontejnerima koji se izvršavaju unutar ljuske te status u kojemu se ljuska nalazi. Isto tako vidimo i na kojem konkretno čvoru se ova ljuska izvodi: node-01.

```
root@master-01:~# kubectl describe pod web-2ctzk
Name:               web-2ctzk
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               node-01/207.154.243.162
Start Time:         Sun, 19 Aug 2018 10:22:50 +0000
Labels:             name=web
Annotations:        <none>
Status:             Running
IP:                 10.244.1.48
Controlled By:      ReplicaSet/web
Containers:
  web:
    Container ID:   docker://9c0314558e2c525c249fb4239777ec5da202d0050d23fa7f041835b5df08
    Image:          acerinth/todo-app
    Image ID:       docker-pullable://acerinth/todo-app@sha256:1fbfe78943ea34e19a4210fc44
    Port:           3000/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Sun, 19 Aug 2018 10:23:04 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-kj9w8 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-kj9w8:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-kj9w8
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:          <none>
root@master-01:~#
```

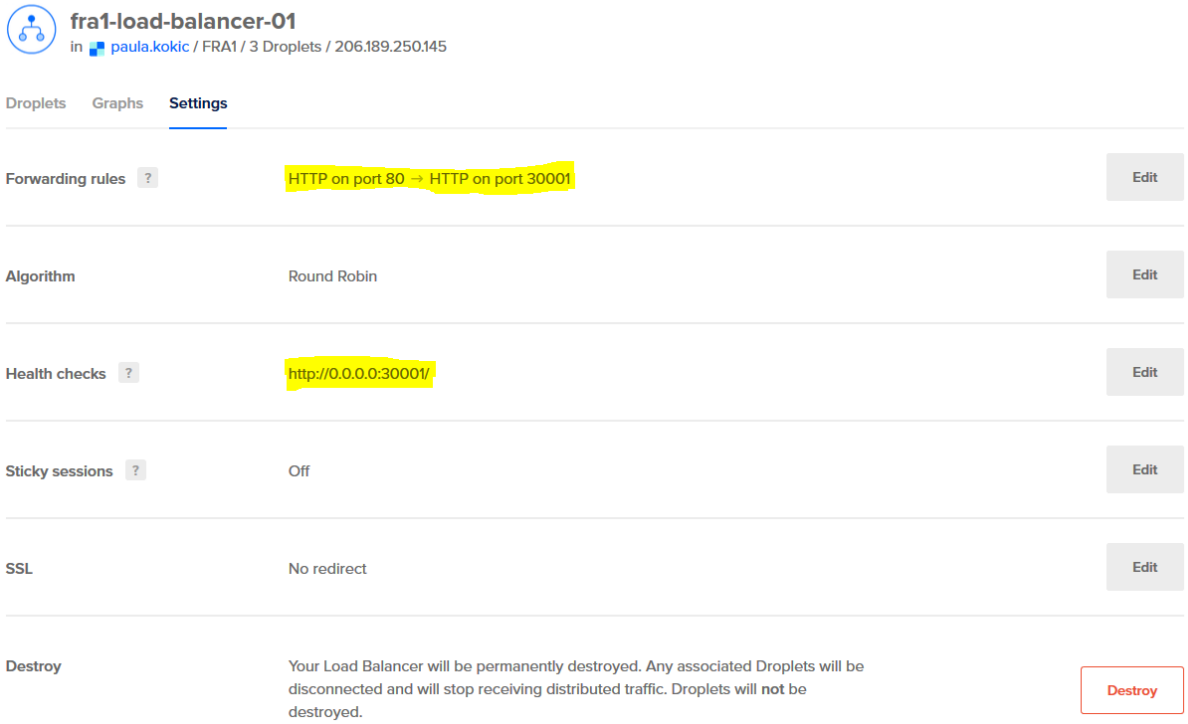
Slika 17. Kubectl describe

6.5. Balansiranje opterećenja

Posljednja vrlo bitna stvar koja je preostala za konfigurirati jest balansiranje opterećenja. Servis za balansiranje opterećenja (eng. *Load Balancer*) služi za ravnomjerno preusmjeravanje prometa prema svim čvorovima unutar odabrane mreže. Dostupan je na većini cloud servisa poput Digital Oceana.

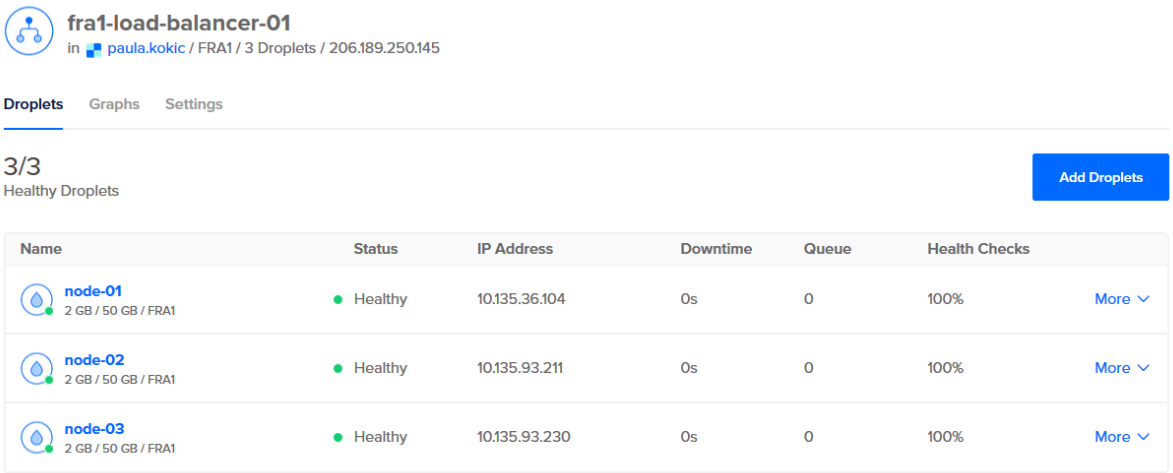
Prilikom kreiranja *Load Balancer*-a na Digital Oceanu, nakon dodavanja čvorova na koje će *Load Balancer* usmjeravati promet, najvažnije je pravilno konfigurirati pravila za prosljeđivanje. Ono što želimo jest da kad netko posjeti IP adresu (obični port 80), da ga

preusmjeri na jednog od naših čvorova na port 30001, gdje se izvršava naša aplikacija. Isto tako, budući da Digital Ocean provjerava stanje svakog od naših čvorova, moramo podesiti i *Health Checks*, tako da pokazuje na port 30001. Postavke *Load Balancer*-a možemo vidjeti na Slici 18. Žutom bojom su osjenčane bitne promjene.



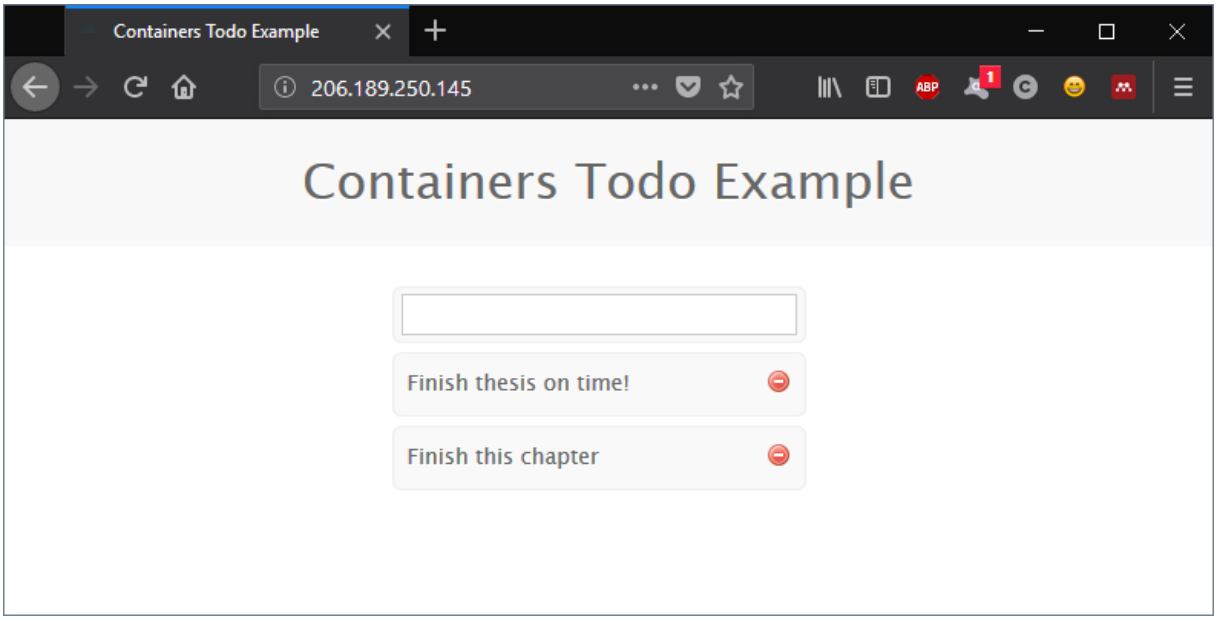
Slika 18. Postavke Load Balancer-a na Digital Oceanu

Ukoliko je sve u redu, nakon nekoliko minuta, dok se izvrše sve provjere, status svakog čvora trebao bi biti *Healthy*, kako je prikazano na Slici 19.



Slika 19. Load balancer - pregled čvorova

Kada je *Load Balancer* pravilno konfiguriran, slobodno možemo pristupiti našoj aplikaciji preko njegove IP adrese. Servis se sam brine o tome ne koji čvor će poslati zahtjev. Aplikaciju u izvođenju preko Load Balancera možemo vidjeti na Slici 20.



Slika 20. Aplikacija pokrenuta preko IP adrese Load Balancer-a

7. Nadzor mikroservisa

U ovom poglavlju, samo ću dati kratak pregled problema i potencijalnih rješenja još jednog vrlo važnog područja kada govorimo o web aplikacijama – nadzor. Dobro je poznato da projekt ne završava isporukom web aplikacije, već se nastavlja kroz održavanje i nadzor - važno je osigurati da aplikacija ima dobar odziv, da je raspoloživa i dostupna prema zahtjevima korisnika.

7.1. Važnost pravilnog nadzora i vidljivosti

Vrlo je bitno naglasiti koliko se zapravo kontejnizirane aplikacije razlikuju od klasičnih, monolitnih aplikacija i kako treba s velikim oprezom i pažnjom pristupiti ovom području. Kontejneri su vrlo dinamični, nepredviljivi, koriste vrlo različite tehnologije i programske jezike te uobičajeni alati za nadzor monolitnih aplikacija ovdje neće uopće biti od koristi. I to ne samo radi različitosti tehnologija, već i zbog same mikroservisne arhitekture koju ne podržavaju. S druge strane, ni slučajno se ne smijemo osloniti na ugrađene procese za nadzor unutar Dockera jer to nije ni približno dovoljno informacija. Slično tome, Kubernetes nije stvoren za nadzor – istina je da nam može dati određene informacije o čvorovima i resursima, ali te informacije služe ponajviše kako bi se ljuske optimalno rasporedile prema čvorovima te i dalje – nisu dovoljne. Trebaju nam noviji alati, specijalizirani za nadzor mikroservisa i općenito kontejniziranih aplikacija – zašto?

Osim onog klasičnog razloga, koji stoji i za monolitne aplikacije, a to je da pravilan nadzor i vidljivost svih procesa i resursa u određenom trenutku omogućuje lakše održavanje aplikacije zdravom, brzom i efektivnom, uz najmanji mogući trošak - postoji još nekoliko razloga. Jedan od njih jest različitost korištenih tehnologija. Koliko god različitost bila pozitivna strana radi lakše implementacije, toliko sada postaje pravi izazov sve uskladiti, a još više nadzirati – krenuvši od izbora programskog jezika, sustava za upravljanje bazom podataka, sustava za kontejnerizaciju, sustava za orkestraciju do samog hardvera i operacijskog sustava. Postavlja se pitanje kako učinkovito prikupljati podatke o toliko različitih tehnologija? Recimo da se negdje dogodi greška i sustav ne radi očekivano – kako ćemo odrediti izvor problema ako nemamo dovoljno informacija o tome što se i gdje točno događa? Da li je problem u programskom kôdu, u kontejneru, u komunikaciji između mikroservisa ili nečem četvrtom? Otkrivanje pogreške bez adekvatnog nadzora i dovoljno informacija može trajati satima. Još jedan razlog je upravo – skalabilnost. Kubernetes replicira mikroservise prema onome kako smo definirali. Ali kako ćemo znati što definirati, ako nemamo dovoljno informacija o tome koliko zahtjeva u koje vrijeme pristiže na određeni mikroservis, kolika je latencija, imamo li

dovoljno računalne i memorijske snage da povećamo kapacitet mikroservisa, itd.? Sve je to vrlo važno i treba uzeti u obzir prilikom odabira alata za nadzor. [9] U ovom radu pogledat ćemo neke osnove sustava Prometheus kao alata za nadzor mikroservisa.

7.2. Prometheus

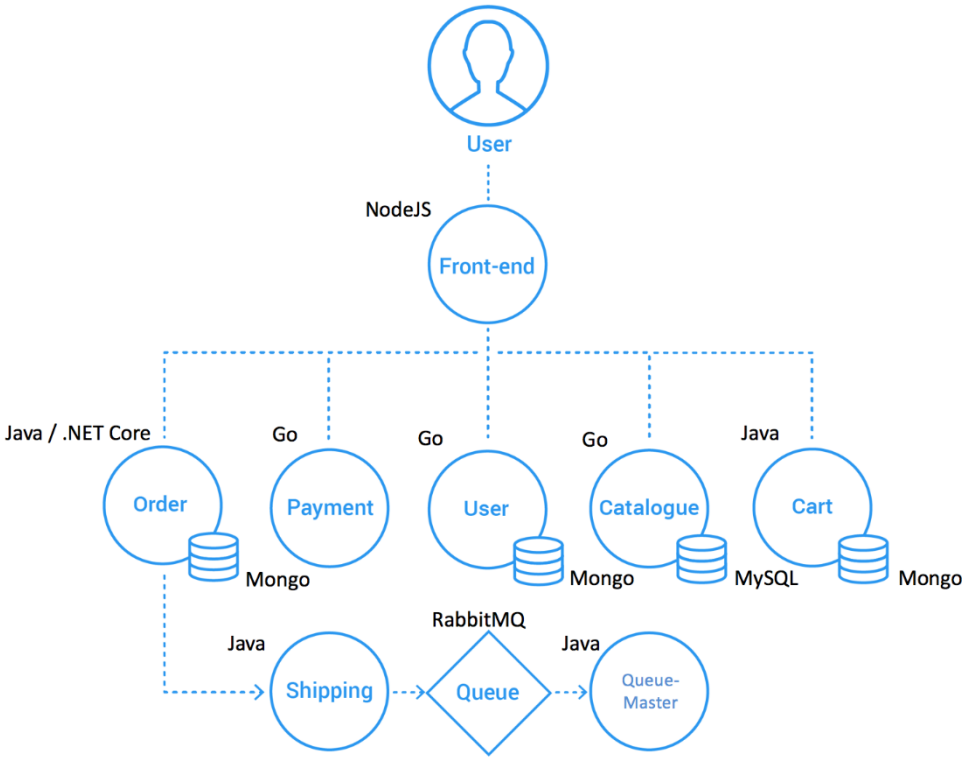
Prometheus je softver otvorenog koda za nadzor (eng. *monitoring*) i uzbunu (eng. *alerting*), prvi put korišten 2012. godine u *SoundCloud* projektu. [10] Sustav skuplja podatke i mjere servisa koji su pokrenuti i pohranjuje ih u vremensku bazu podataka. Radi s multidimenzionalnim modelom podataka, ima fleksibilan upitni jezik (eng. *query language*) i nekoliko varijanti za vizualizaciju podataka, a ja ću koristiti Grafanu. [11]

Najveći dio Prometheus sustava čine izvoznici (eng. *exporters*) – to su nadogradnje koje omogućuju nadgledanje i skupljanje podataka za specifične tehnologije i alate, web servere, baze podataka, pa čak i same infrastrukture. Neke od najčešćih izvoznika koji se koriste jesu:

- *Node_exporter* – skuplja podatke o infrastrukturi, uključujući korištenje CPU-a, memorije, diska, I/O mreže, itd.;
- *Blackbox_exporter* – skuplja podatke od mrežnih protokola kao što su HTTP i HTTPS, kako bi se dobila raspoloživost krajnje točke (eng. endpoint), vrijeme odziva i sl.;
- *Mysqld_exporter* – skuplja podatke o MySQL serveru, kao što su broj izvršenih upita, prosječno vrijeme izvršavanja upita, itd.;
- *Rabbitmq_exporter* – skuplja podatke o RabbitMQ sustavu za razmjenu poruka, uključujući broj objavljenih poruka, broj poruka spremnih za dostavu, veličinu svih poruka u redu čekanja i sl.;
- *Nginx_vts_exporter* – skuplja podatke o Ngnix web serveru, koristeći njegov VTS modul, uključujući broj otvorenih konekcija, broj poslanih odgovora grupiranih prema kodu, ukupnu veličinu poslanih ili primljenih zahtjeva u bajtovima, itd. [11]

7.3. Postavljanje aplikacije *Sock Shop*

Za demonstraciju nadzora koristit ću jednu veću aplikaciju, koja je svojevrsna web-trgovina za čarape (*Sock Shop*). Gotova konfiguracija za isporuku dostupna je na GitHub linku: https://github.com/microservices-demo/microservices-demo . Kako bismo lakše pratili što se sve događa, u nastavku se nalazi arhitektura aplikacije na Slici 21.



Slika 21. Arhitektura aplikacije *Sock Shop* (Izvor: [13])

Kao što možemo vidjeti na Slici 21., aplikacija se sastoji od niza mikroservisa, pisanih u različitim tehnologijama. Krajnji korisnik vrši interakciju samo s web frontend-om (grafičkim sučeljem) koje je napisano u NodeJS-u, a front-end dalje komunicira s ostalim servisima. Payment, User i Catalogue su pisani u Go-u, a Order, Shipping, Cart i Queue-Master u Javi. Korištene baze podataka su Mongo i MySQL.

Prije nego krenemo na nadzor, ukratko ću prikazati kako sam ovu aplikaciju pokrenula na vlastitom Kubernetes klasteru. Usput ću pokazati još nekoliko poteza koje je dobro primjenjivati prilikom isporuke većih i ozbiljnijih aplikacija.

Nakon klasičnog *git clone* cijelog repozitorija, prvo što radimo jest kreiranje novog imenskog prostora. To će nam omogućiti lakše snalaženje u klasteru i daljnju manipulaciju ljuskama, servisima, isporukama i sl. Naredba je *kubectl create namespace*, kao što možemo vidjeti na Slici 22.

```
root@master-01:~/microservices-demo/deploy/kubernetes# kubectl create namespace sock-shop
namespace/sock-shop created
```

Slika 22. Kubectl create namespace

Nakon što smo kreirali imenski prostor, možemo kreirati i sve ostale objekte koji su opisani u jednoj YAML datoteci – *complete-demo.yaml*. Ovdje idemo korak naprijed i umjesto da se ljuske kreiraju ručno, one su kreirane od strane objekta koji se zove isporuka (eng. *Deployment*). *Deployment* se brine o cijelom životnom ciklusu jedne ljuske, njegovim replikama i uz to omogućava isporuku nove verzije servisa bez ikakvih stanki u radu (eng. *downtime*). U nastavku je primjer specifikacije za *Deployment* iz navedene YAML datoteke:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: carts-db
  labels:
    name: carts-db
  namespace: sock-shop
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: carts-db
    spec:
      containers:
      - name: carts-db
        image: mongo
        ports:
        - name: mongo
          containerPort: 27017
        securityContext:
          capabilities:
            drop:
              - all
            add:
              - CHOWN
              - SETGID
              - SETUID
          readOnlyRootFilesystem: true
```

```
    volumeMounts:
    - mountPath: /tmp
      name: tmp-volume
  volumes:
    - name: tmp-volume
      emptyDir:
        medium: Memory
  nodeSelector:
     beta.kubernetes.io/os: linux
```

Ovaj *Deployment* opisuje servis koji će pokretati bazu podataka (MongoDB) za servis *Cart*, odnosno manipulaciju košaricom. Možemo primjetiti da se uglavnom koriste ista svojstva, odnosno da ovaj opis podosta podsjeća na *ReplicaSet*. Svi kontejneri su već definirani i nalaze se na DockerHubu, tako da je sljedeći korak kreiranje svih Kuberentes objekata navedenih u ovoj YAML datoteci. Kreiranje *Deploymenta* i servisa vidimo na Slici 23:

```
root@master-01:~/microservices-demo/deploy/kubernetes# kubectl apply -f complete-demo.yaml
deployment.extensions/carts-db created
service/carts-db created
deployment.extensions/carts created
service/carts created
deployment.extensions/catalogue-db created
service/catalogue-db created
deployment.extensions/catalogue created
service/catalogue created
deployment.extensions/front-end created
service/front-end created
deployment.extensions/orders-db created
service/orders-db created
deployment.extensions/orders created
service/orders created
deployment.extensions/payment created
service/payment created
deployment.extensions/queue-master created
service/queue-master created
deployment.extensions/rabbitmq created
service/rabbitmq created
deployment.extensions/shipping created
service/shipping created
deployment.extensions/user-db created
service/user-db created
deployment.extensions/user created
service/user created
```

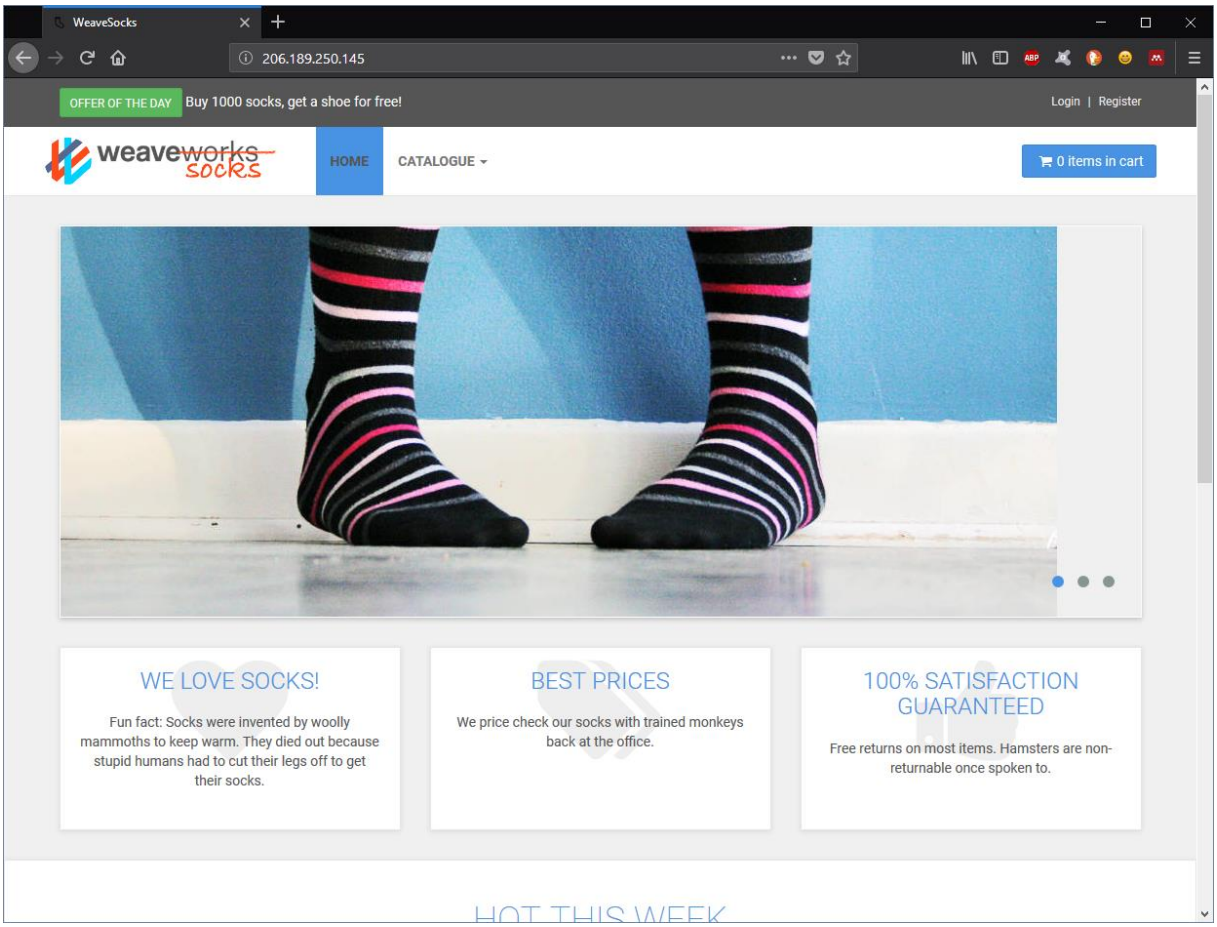
Slika 23. Kreiranje Kubernetes objekata

Ako je kreiranje uspješno prošlo, nakon nekoliko minuta možemo vidjeti da su sve ljuske i servisi uspješno pokrenuti, kao što je vidljivo na Slici 24.

```
root@master-01:~/microservices-demo/deploy/kubernetes# kubectl get pods --namespace=sock-shop
NAME                            READY     STATUS    RESTARTS   AGE
carts-6dfdcd59f8-cq7hp          1/1       Running   0          7m
carts-db-6c9b649b49-p44sp       1/1       Running   0          7m
catalogue-7d7f9f87f-bzjtn       1/1       Running   0          7m
catalogue-db-745c877d4f-jlrdr   1/1       Running   0          7m
front-end-6f779bdb68-kz4dm      1/1       Running   0          7m
orders-5c4f477565-bw9bl         1/1       Running   0          7m
orders-db-db498cfb9-pshht       1/1       Running   0          7m
payment-5df6dc6bcc-wcplc        1/1       Running   0          7m
queue-master-787b68b7fd-tccjz   1/1       Running   0          7m
rabbitmq-86fcc47fc-xxxlv        1/1       Running   0          7m
shipping-64f8c7558c-z955f       1/1       Running   0          7m
user-7848fb86db-6cwvt           1/1       Running   0          7m
user-db-586b8566b4-htckg        1/1       Running   0          7m
```

Slika 24. Ljuske *Sock shopa*

U YAML datoteci vidimo da je *front-end* servis izložen na portu 30001, kao *NodePort*. Prema tome, ne moramo dodatno konfigurirati *Load Balancer* na DigitalOceanu, već samo pristupiti njegovoj IP adresi i pronaći *Sock Shop* aplikaciju, kao što vidimo na Slici 25.



Slika 25. Pristup web aplikaciji putem IP adrese Load Balancera

Budući da imamo isporučenu aplikaciju, spremni smo za daljnje korake nadzora i testiranja.

7.4. Postavljanje nadzornih alata

Cijela *Sock Shop* mikroservisna aplikacija dolazi skupa s različitim varijantama za isporuku, ali i vizualizaciju i testiranje. Potrebno je izvršiti samo još nekoliko naredbi, a prva od njih će instalirati Prometheus, tj. kreirati potrebne Kubernetes objekte kako bi se isti mogao izvršavati u našem Kubernetes klasteru. Kreiranje servisa i *deploymenta* za Prometheus vidimo na Slici 26.

```
root@master-01:~/microservices-demo/deploy/kubernetes# kubectl create -f manifests-monitoring
namespace/monitoring created
configmap/prometheus-alertrules created
configmap/prometheus-configmap created
clusterrole.rbac.authorization.k8s.io/prometheus created
clusterrolebinding.rbac.authorization.k8s.io/prometheus created
deployment.extensions/prometheus-deployment created
daemonset.extensions/node-directory-size-metrics created
deployment.extensions/kube-state-metrics-deployment created
service/kube-state-metrics created
serviceaccount/prometheus created
service/prometheus created
```

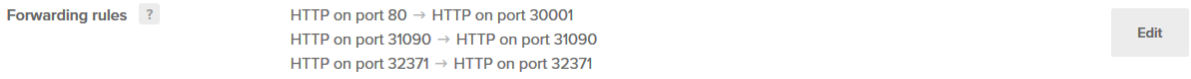
Slika 26. Instalacija Prometheusa

Nakon toga, praktički istu stvar radimo i za Grafanu. Kako bismo provjerili da je sve u redu, možemo pozvati *kubectl get* i provjeriti statuse ljuski i servisa, kao što je vidljivo na Slici 27.

```
root@master-01:~# kubectl get pods --namespace=monitoring
NAME                                        READY     STATUS      RESTARTS   AGE
grafana-core-6df8b9687-828t6                1/1       Running     0          14h
grafana-import-dashboards-q4t6z             0/1       Completed   0          14h
kube-state-metrics-deployment-58c7bbdc-2c4lt   1/1       Running     0          15h
node-directory-size-metrics-7xcw4           2/2       Running     0          15h
node-directory-size-metrics-pj7t8           2/2       Running     0          15h
node-directory-size-metrics-s95dz           2/2       Running     0          15h
prometheus-deployment-6488dfc64c-jgjzh      1/1       Running     0          10h
root@master-01:~# kubectl get services --namespace=monitoring
NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
grafana              LoadBalancer   10.103.171.59    <pending>     80:32371/TCP     14h
kube-state-metrics   ClusterIP      10.103.50.8      <none>        8080/TCP         15h
prometheus           NodePort       10.103.125.208   <none>        9090:31090/TCP   15h
root@master-01:~#
```

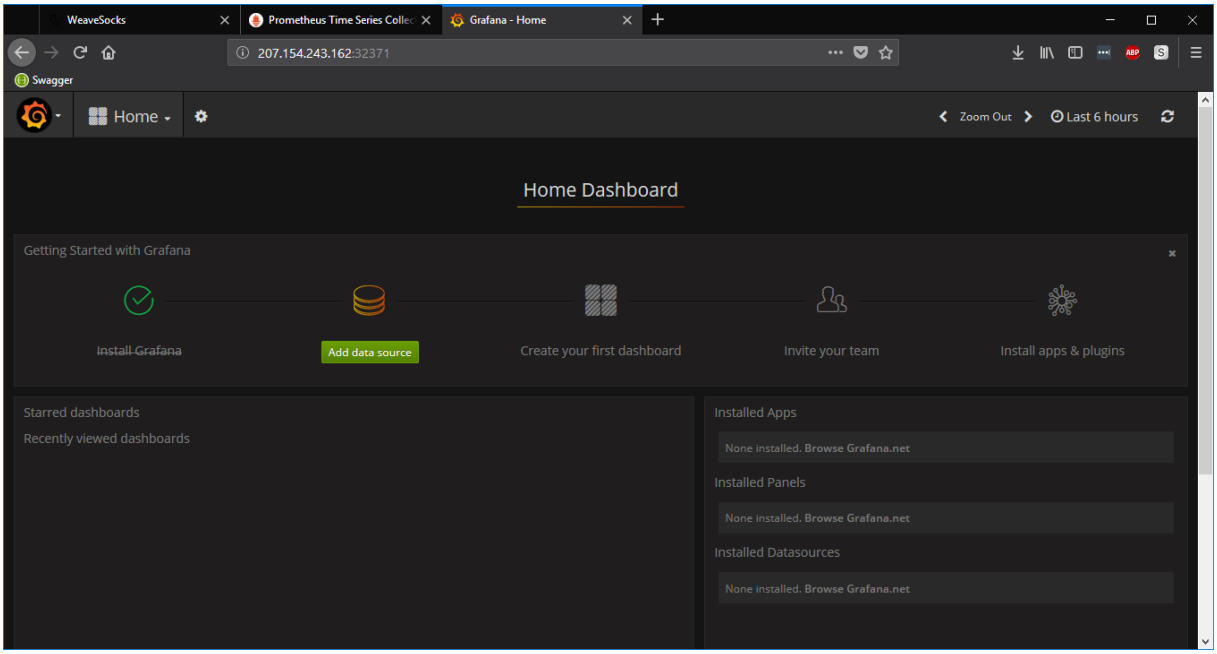
Slika 27. Ljuske i servisi Prometheusa i Grafane

S ovom naredbom smo upravo dobili i ispis portova na kojima se izvršavaju ovi servisi. Prema tome, možemo uzeti IP adresu bilo kojeg čvora i na portu 31090 dobiti Prometheus servis, odnosno na portu 32371 dobiti Grafanu. Još bolje, možemo konfigurirati naš *Load Balancer*, tj. dodati pravilo za prosljeđivanje, tako da sve možemo obavljati preko njegove IP adrese, kao što vidimo na Slici 28.



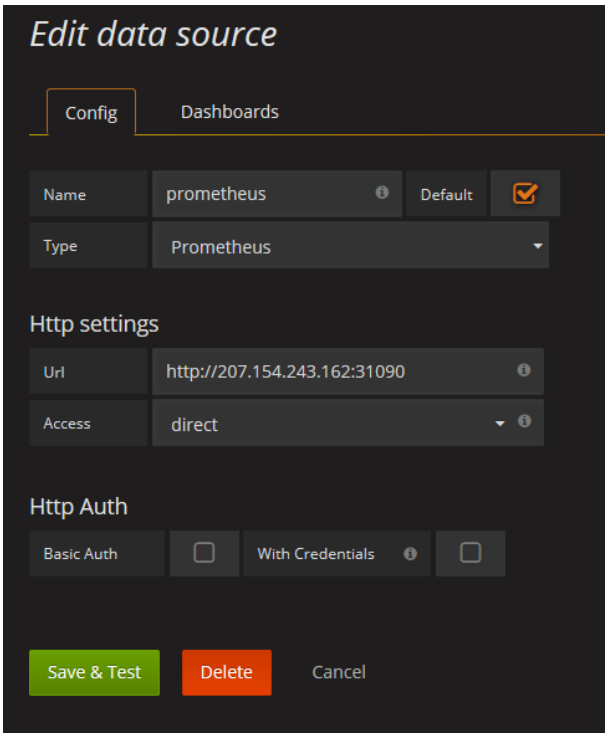
Slika 28. Prosljeđivanje na Load Balanceru

Grafanu možemo pokrenuti na portu 32371, koristiti uobičajene podatke za prijavu kao administrator te krenuti s konfiguracijom. Dočeka nas prikaz kao na Slici 29.



Slika 29. Grafana - prvo pokretanje

Slijedi dodavanje izvora podataka, odnosno povezivanje s Prometheus sustavom. Klikom na *Add data source* otvara se nova stranica s mogućnošću konfiguriranja. Dodat ćemo novi izvor tipa Prometheus, nazvati ga istim nazivom i postaviti IP adresu čvora na kojem se nalazi ljuska, kao što je prikazano na Slici 30.



Slika 30. Dodavanje izvora podataka za Grafanu

Ukoliko nakon klika na *Save & Test* dobijemo zelenu potvrdu da je sve u redu, znači da je Grafana uspješno prepoznala izvor podataka.

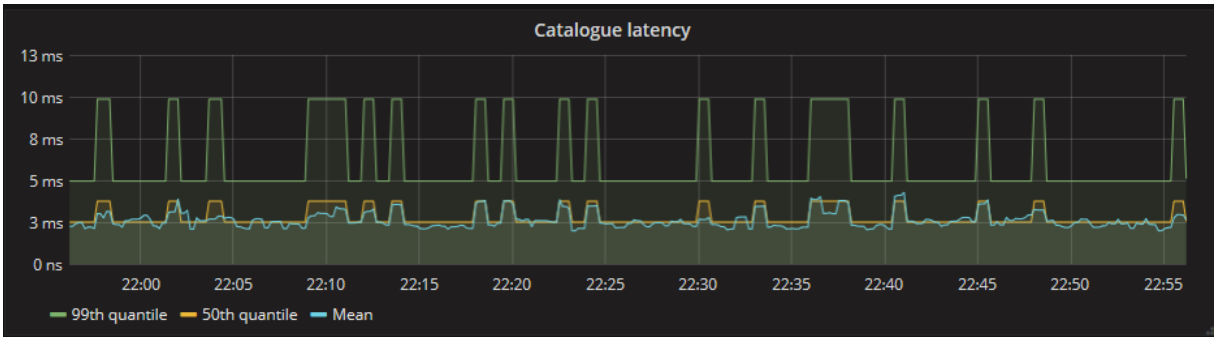
Sve postavke za Prometheus nalaze se u *prometheus.yaml* datoteci, koja se smješta u `/env/prometheus` direktorij. U našem slučaju, Prometheus će se naravno izvoditi u vlastitoj ljusci, tako da ćemo cijelu datoteku prosljeđivati kroz Kubernetes objekt *ConfigMap*. *ConfigMap* omogućuje nam razdvajanje konfiguracijskih parametara ljuske od samog sadržaja slika, tj. kontejnera, kako bi se održala portabilnost kontejniziranih aplikacija. [8] Sve što je potrebno napraviti jest spremiti *prometheus.yaml* datoteku unutar npr. *prometheus-configmap.yaml* i prilikom kreiranja *Deploymenta*, montirati isti *configmap* (služeći se oznakama), na nositelj podataka koji će se nalaziti na `/etc/prometheus` putanji.

Kada govorimo o Grafani, osim izvora podataka, naravno potrebno je konfigurirati i grafove koje će nam prikazivati kontrolna ploča (eng. *dashboard*). Ako u pogledu *Dashboards* vidimo već složene kontrolne ploče, znači da smo dobro uvezli podatke iz projekta (tijekom postavljanja nazdornih alata u potpoglavlju 7.1.).

7.5. Test opterećenja

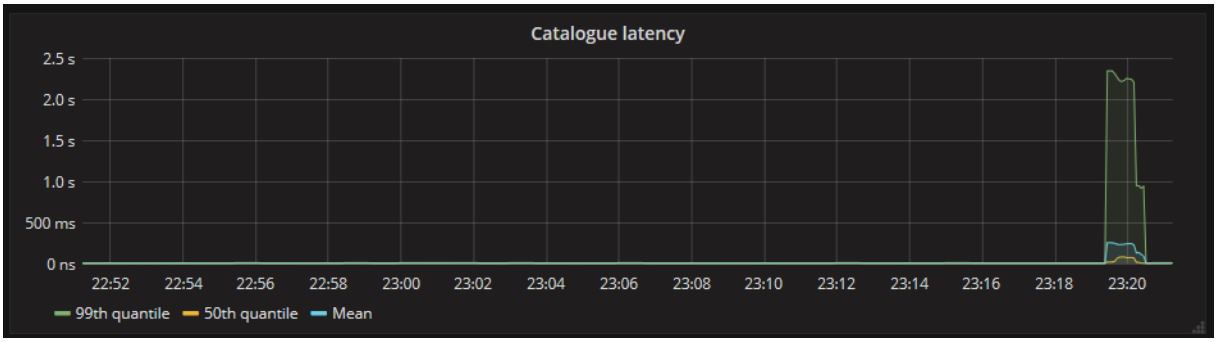
U nastavku ću prikazati jednostavan primjer testa opterećenja (eng. *load test*), koji je također dostupan u sklopu *microservices-demo* repozitorija na GitHubu (https://github.com/microservices-demo/load-test). Testovi se izvode s virtualnog Ubutnu računala (izvan klastera) te napadaju *front-end* aplikacije. Za prikaz koristit ćemo gotovu kontrolnu ploču koja prikazuje latencije pojedinih mikrosevisa, a radi jednostavnosti, fokusirat ćemo se na jedan mikroservis – *catalogue*. Latencija (eng. *latency*) predstavlja vrijeme potrebno da određeni web servis primi i obradi određeni zahtjev za nekim objektom. [12]

Prvo ćemo pogledati kako izgleda graf latencije za mikroservis *catalogue*, s jednom ljuskom i bez ikakvog velikog prometa, na Slici 31.



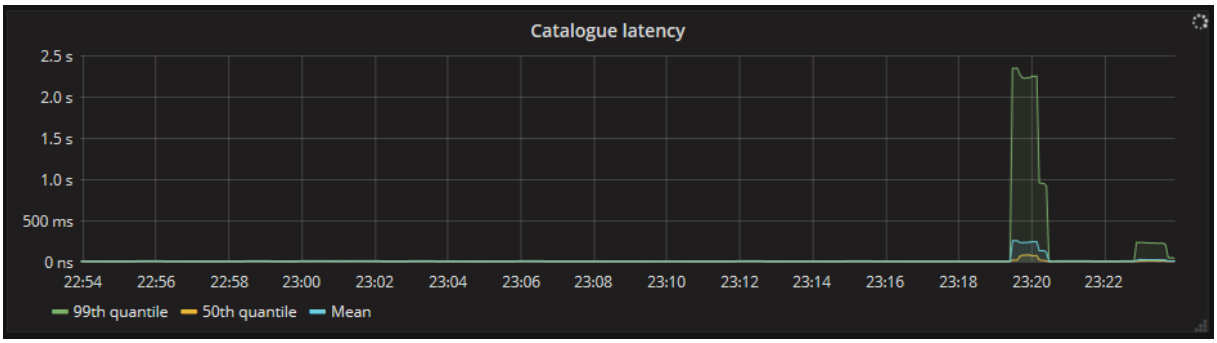
Slika 31. Catalogue latency - prije testova, 1 ljuska

Iz navedenog grafa možemo očitati da je prosječna latencija iznosi oko 4ms, a najveća do 10ms. Sada ćemo vidjeti što se dogodi kada pustimo test opterećenja s 10 klijenata i ukupno 100 zahtjeva. Rezultat je prikazan na Slici 32.



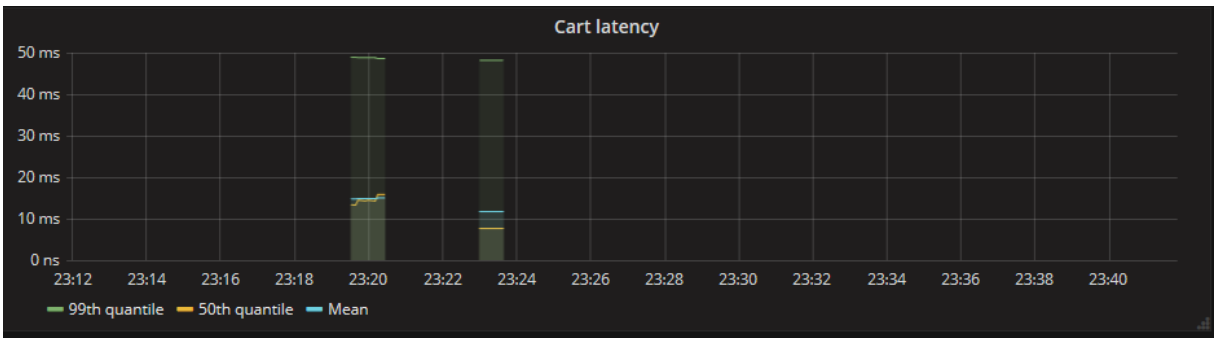
Slika 32. Catalogue latency - test opterećenja na 1 ljusku

Dakle, vidimo da se latencija drastično povećala, skoro do 2.5 sekunde. Sada ćemo skalirati *catalogue deployment* na 5 ljuski pa ponovno pokrenuti test. Rezultati se nalaze na Slici 33.

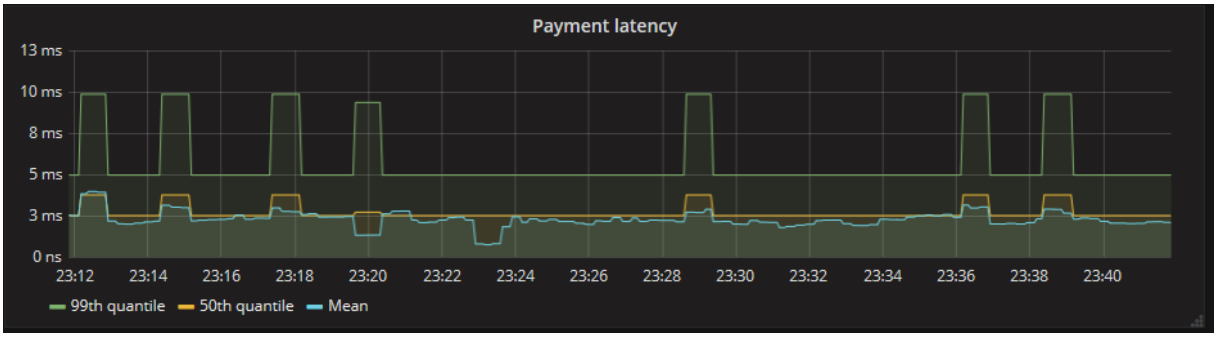


Slika 33. Catalogue latency graf - test opterećenja na 5 ljuski

Prema grafu iznad, možemo vidjeti da se latencija znatno poboljšala. Dakle, povećanje broja ljuski bila je dobra ideja. Budući da se testovi generiraju nasumične zahtjeve, ne možemo biti 100% sigurni da se generirao identičan omjer zahtjeva prema catalogue servisu u oba slučaja, pa ću za usporedbu ispod priložiti i grafove od druga dva mikroservisa: *cart* i *payment*, koji se nalaze na Slikama 34. i 35.



Slika 34. Cart latency graf



Slika 35. Payment latency graf

Prema posljednja dva grafa, vidimo da je omjer opterećenosti otprilike bio jednak u oba servisa, a njihove ljuske nismo skalirati. Payment se općenito dobro drži cijelo vrijeme, dok možemo primjetiti da je latencija servisa *cart* dosta porasla, pa bi npr. u realnoj situaciji morali pripaziti i na to. Upravo ovaj primjer pokazuje koliko nam izolacija zapravo pomaže – da imamo monolitnu aplikaciju, teško bismo mogli izvršiti ovako detaljne analize. Osim toga, definitivno bismo morali replicirati cijelu aplikaciju kako bismo zadovoljili zahtjevima korisnika, koristeći pritom puno više resursa nego što nam zapravo treba.

Ovo je vrlo jednostavan primjer, ali pokazuje koliko nam dobar nadzor u pravo vrijeme može zapravo pomoći u održavanju, sigurnosti i raspoloživosti web aplikacija.

U ovom poglavlju samo smo zagrebli još jedno zapravo veliko i vrlo važno područje web aplikacija, a posebno mikroservisa. Prema tome, nikako ga ne treba zanemariti, već prilikom rada s ozbiljnim aplikacijama, posvetiti dovoljno vremena i truda u analizu, odabir pravih alata i podataka za nadzor, kako bi se spriječile potencijalne nepoželjne situacije.

8. Zaključak

U ovom radu bavila sam se mikroservisima – relativno novoj arhitekturi koja iz godine u godinu uzima sve veći zamah i sve je popularnija. Mikroservisi su manje, izolirane jedinice softvera koje su sposobne same izvršavati određeni zadatak te u suradnji s drugim servisima čine veću aplikaciju. Glavno obilježje mikroservisne arhitekture jest međusobna izolacija – mogućnost samostalnog razvoja i isporuke. Koncept kontejnera kao laganog omotača koji čini okolinu jednog procesa temelj je realizacije cijele strukture oko mikroservisne arhitekture i temeljna poveznica s ostalim tehnologijama i alatima.

Pokazala sam osnovne procese kroz koje prolazi svaka mikroservisna aplikacija nakon završetka faze developmenta, a to su kontejnerizacija, isporuka i nadzor. Postoji nekoliko tehnologija koje možemo kombinirati, a u ovom radu koristila sam Docker kao sustav za kontejnerizaciju, Kubernetes kao sustav za isporuku i orkestraciju, Prometheus kao sustav za nadzor i prikupljanje podataka te Grafanu kao alat za vizualizaciju dobivenih podataka.

Nadam se da sam uspjela prikazati koliko upravo korištenje mikroservisa može pridonijeti sigurnosti i stabilnosti cijele aplikacije. Činjenica jest da je puno tehnologija u pitanju, bezbroj kombinacija i mogućnosti, što povlači da je potrebno mnogo (pred)znanja o svemu kako bi sve skupa imalo ikakvog smisla. Međutim, osobno smatram kako je definitivno vrijedno truda i vremena jer se naknadno može uštediti puno vremena. Naime, ovo je samo temelj i početna točka procesa koji danas također hvataju sve veći zamah, a to su automatizacija, kontinuirana integracija i kontinuirana isporuka. Nadam se da ću nastaviti razvijati i vlastitu IT karijeru u tom području te se uskoro naći na jednoj od tzv. *DevOps* radnih pozicija.

Popis literature

[1] I. Dwyer, “Microservices: Patterns for Bulding Modern Applications,” 2015.

[2] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*, First Edit. Sebastopol, CA 95472: O’Reilly Media, Inc., 2016.

[3] F. Harrison, “Getting started with Microservices,” *Methods Ecol. Evol.*, vol. 2, no. 1, pp. 1–10, 2011.

[4] K. Matthias and S. P. Kane, *Docker: Up & Running: Shipping Reliable Containers in Production*. 2015.

[5] Wikipedia, “Operating-system-level virtualization,” 2018. [Online]. Available: https://en.wikipedia.org/wiki/Operating-system-level_virtualization. [Accessed: 08-Jul-2018].

[6] P. Brey, “Containers vs. Virtual Machines (VMs): What’s the Difference?,” 2018. [Online]. Available: https://blog.netapp.com/blogs/containers-vs-vms/. [Accessed: 08-Jul-2018].

[7] K. Hightower, B. Burns, and J. Beda, *Kubernetes Up and Running; Dive into the Future of Infrastructure*. 2016.

[8] “Kubernetes Documentation,” 2018. [Online]. Available: https://kubernetes.io/docs/home/.

[9] Instana, “Application Performance Management in a Containarized World,” 2018.

[10] P. Authors, “Prometheus Documentation,” 2018. [Online]. Available: https://prometheus.io/docs/introduction/overview/.

[11] M. Mudrinić, “How To Install Prometheus on Ubuntu 16.04,” 2017. [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-install-prometheus-on-ubuntu-16-04.

[12] J. Bixby, “Latency 101: What is latency and why is it such a big deal?,” 2012. [Online]. Available: http://www.webperformancetoday.com/2012/04/02/latency-101-what-is-latency-and-why-is-it-such-a-big-deal/.

[13] Á. Sándor, A. Giurgiu, and V. Lal, “Sock Shop Design,” 2018. [Online]. Available: https://github.com/microservices-demo/microservices-demo/blob/master/internal-docs/design.md. [Accessed: 20-Aug-2018].

Popis slika

Slika 1. Dizajn različitih arhitektura kroz povijest (Prema: [1]) 4
Slika 2. Razlike između virtualnih mašina i kontejnera (Prema: [6]) 8
Slika 3. Arhitektura Kubernetes sustava 14
Slika 4. SSH spajanje na master čvor 15
Slika 5. Rezultat *kubeadm init* naredbe 16
Slika 6. Čvor je dodan Kubernetes klasteru 17
Slika 7. Izgradnja slike 18
Slika 8. Docker images 19
Slika 9. Docker push 19
Slika 10. Kubectl create 20
Slika 11. Kubectl get pods i kubectl get services 22
Slika 12. To-do aplikacija u izvođenju 22
Slika 13. Kubectl log 23
Slika 14. Replica set 25
Slika 15. Automatsko kreiranje nove ljuske 25
Slika 16. Povećavanje broja replikacija 26
Slika 17. Kubectl describe 27
Slika 18. Postavke Load Balancer-a na Digital Oceanu 28
Slika 19. Load balancer - pregled čvorova 28
Slika 20. Aplikacija pokrenuta preko IP adrese Load Balancer-a 29
Slika 21. Arhitektura aplikacije *Sock Shop* (Izvor: [13]) 32
Slika 22. Kubectl create namespace 33
Slika 23. Kreiranje Kubernetes objekata 34
Slika 24. Ljuske *Sock shopa* 35
Slika 25. Pristup web aplikaciji putem IP adrese Load Balancera 35
Slika 26. Instalacija Prometheusa 36
Slika 27. Ljuske i servisi Prometheusa i Grafane 36
Slika 28. Prosljeđivanje na Load Balanceru 36
Slika 29. Grafana - prvo pokretanje 37
Slika 30. Dodavanje izvora podataka za Grafanu 37
Slika 31. Catalogue latency - prije testova, 1 ljuska 38
Slika 32. Catalogue latency - test opterećenja na 1 ljusku 39
Slika 33. Catalogue latency graf - test opterećenja na 5 ljuski 39
Slika 34. Cart latency graf 39
Slika 35. Payment latency graf 40