Raspoređivanje prometa u web aplikaciji pomoću Kubernetes platforme

Gazdek, Leonardo

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Leonardo Gazdek

RASPOREĐIVANJE PROMETA U WEB APLIKACIJI POMOĆU KUBERNETES PLATFORME

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Leonardo Gazdek

Matični broj: 0016135816

Studij: Informacijsko i programsko inženjerstvo

RASPOREĐIVANJE PROMETA U WEB APLIKACIJI POMOĆU KUBERNETES PLATFORME

DIPLOMSKI RAD

Mentor :

Prof. dr. sc. Dragutin Kermek

Varaždin, srpanj 2023.

Leonardo Gazdek

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj se rad bavi tematikom distribuiranja web aplikacija na više poslužitelja i raspoređivanjem prometa između tih aplikacija i poslužitelja. Skup umreženih računala koja međusobno komuniciraju i orkestriraju aplikacije naziva se klasterom, a jedno računalo u klasteru uglavnom se naziva čvorom. Da bi se u radu mogao demonstrirati rad klastera prvo je potrebno svhatiti koncept kontejnera. Kontejner je standardna jedinica softvera koja pakira kod i sve njegove ovisnosti koja se može pokrenuti u bilo kojoj okolini koja podržava kontejnere. Kod distribucije softvera se u današnje vrijeme najčešće koriste kontejneri jer su lagani i ne trebaju mnogo resursa, a vrlo su fleksibilni i upravljivi. Klasteri uglavnom rade na način da se na čvorove ravnomjerno raspoređuju aplikacije u obliku kontejnera, a svaki kontejner može imati jednu ili više replika. Jedan čvor može pokretati više kontejnera iste aplikacije u svrhu automatskog oporavka, raspoređivanja opterećenja unutar jednog čvora, postepenih ažuriranja, itd. U ovom su radu obrađene dvije platforme za orkestriranje kontejnera: Docker Swarm i Kubernetes. Docker Swarm mnogo je jednostavniji za postaviti i nema toliko mogućnosti dok je Kubernetes jako moćan alat namijenjen za velike distribuirane sustave, ali zahtjeva i mnogo postavljanja ako se ne koristi uz jedan od upravljanih servisa od strane pružatelja usluga u oblaku. Oba alata koriste deklarativnu YAML sintaksu za postavljanje servisa. U radu je napravljena jednostavna aplikacija s korisničkom i poslužiteljskom stranom koja se isporučuje na Kubernetes klaster, dok se za Docker Swarm koristi mnogo jednostavniji primjer aplikacije koja samo ispisuje svoj identifikator. Na kraju rada, s obzirom na to da Kubernetes u lokalnoj postavi ne podržava servise tipa LoadBalancer, potrebno je postaviti MetalLB. MetalLB softverska je implementacija LoadBalancer servisa za Kubernetes. Postavljanjem MetalLB implementacije klaster postaje dostupan na internetu preko priključka 80.

Diplomski rad izrađen je u okviru Laboratorija za Web arhitekture, tehnologije, servise i sučelja.

Ključne riječi: Kubernetes, metrike, skaliranje, web, aplikacije, promet, kontejneri

Sadržaj

1. Uvod

U današnje vrijeme web su aplikacije nešto što ljudi koriste svakodnevno. Te web aplikacije moraju se pokretati na nekom udaljenom poslužitelju. Kod aplikacija koje nemaju velik promet, realizirati visoku dostupnost jednostavna je stvar. Dovoljno je samo pokrenuti aplikaciju na jednom poslužitelju i računati da će taj poslužitelj biti dovoljan da izdrži promet koji će proći kroz aplikaciju.

Kod velikih aplikacija, nažalost, stvari nisu baš tako jednostavne. Kako je internet polako postajao značajan dio svakodnevnice tako su i velike web usluge morale naučiti nositi se sa sve većim opterećenjem na njihove poslužitelje. Google je, kao jedan veliki internetski gigant, razvio Kubernetes koristeći znanje stečeno u prethodnih 15 godina. Kubernetes je osmišljen kao rješenje za orkestraciju kontejnera. Orkestrator je sustav koji isporučuje aplikacije i upravlja njima. Može rasporediti aplikacije po računalima i dinamički reagirati na promjene [1].

Kontejneri su standardne jedinice softvera koje pakiraju kod i sve njegove ovisnosti tako da se aplikacija može pokrenuti na bilo kojem računalu u bilo kojoj okolini. Kontejneri su napravljeni da budu lagani, samostalni izvršni paket softvera koji uključuje sve što je potrebno da se softver pokrene. U kontejneru se nalaze programski kod, distribucija operacijskog sustava, sistemski alati, sistemske biblioteke i postavke. Iz jedne slike kontejnera može nastati bilo koji broj kontejnera za pokretanje [2].

Kubernetes rješava problem pokretanja web aplikacija i usluga na više čvorova (računala) odjednom. Brine se o tome kako će se rasporediti promet, koliko će instanci aplikacije pokrenuti na kojem čvoru, itd. Temeljni koncept koji će se obraditi u ovom radu je raspoređivanje prometa (load balancing). Treba spomenuti koje su vrste raspoređivanja prometa, kako to radi Kubernetes i kada je prikladno koristiti Kubernetes za ovu zadaću.

2. Metode i tehnike rada

U ovom će radu biti korištena Kubernetes platforma za demonstraciju horizontalnog skaliranja dva osnovna web servisa, ali prije toga, važno je prvo obraditi sam koncept kontejnera. U radu će se prvenstveno pričati o Docker kontejnerima iako postoje mnogi drugi pokretači kontejnera. Kontejneri će se na Kubernetes klasteru pokretati koristeći pokretač kontejnera cri-o. Ono što je bitno je to da su kontejneri u skladu s OCI standardima koji će biti spomenuti kasnije u radu.

Za izradu aplikacije koja će se pakirati u kontejner koristiti će se Spring razvojni okvir, tj. projekt će biti inicijaliziran Spring Boot alatom. Kao baza podataka koristiti će se Amazon DynamoDB zbog svoje mogućnosti praktički beskonačnog skaliranja. Skaliranje nekakve relacijske baze poput PostgreSQL bilo bi preteško i preopširno za ovaj rad. Korisnička strana aplikacije biti će izrađena u biblioteci React koristeći Next.js razvojni okvir. Klijentski kod biti će uređivan u Visual Studio Code-u, dok će Java kod biti uređivan koristeći IntelliJ IDEA integriranu razvojnu okolinu. Za najam poslužitelja koristit će se Oracle Cloud platforma i njeni x86 poslužitelji koji su više nego dovoljni za pokretanje Kubernetes klastera u ovom radu. Demonstrirat će se i Docker Swarm kao jednostavnija alternativa Kubernetes klasteru. Ansible je jedan od alata koji mogu značajno olakšati inicijalno postavljanje poslužitelja pa će se on u radu koristiti za instalaciju Dockera na Swarm čvorove.

3. Kontejneri

Kontejner je standardna jedinica softvera koja pakira kod i sve njegove ovisnosti tako da se aplikacija može pokrenuti na bilo kojem računalu u bilo kojoj okolini. Kontejneri su napravljeni da budu lagani, samostalni izvršni paketi softvera koji uključuju sve što je potrebno da se softver pokrene. U kontejneru se nalaze programski kod, distribucija operacijskog sustava, sistemski alati, sistemske biblioteke i postavke. Iz jedne slike kontejnera može nastati bilo koji broj kontejnera za pokretanje [2].

Kontejneri dijele ljusku operacijskog sustava sa svojim domaćinom i to je ono što ih čini puno lakšim od pravih virtualnih strojeva. Svaki virtualni stroj mora imati svoju ljusku, dok ju kontejneri dijele.

Kontejneri izoliraju softver koji se u njima nalazi od ostatka sustava i njegovog okruženja [2]. To znači da softver unutar kontejnera ne može komunicirati sa sustavom domaćina osim preko jasno definiranih kanala (poput računalnih mreža). Ovo je iznimno bitno za sigurnost, a omogućava i jednostavnu enkapsulaciju svih ovisnosti bez potrebe da te ovisnosti budu instalirane na sustavu domaćina.

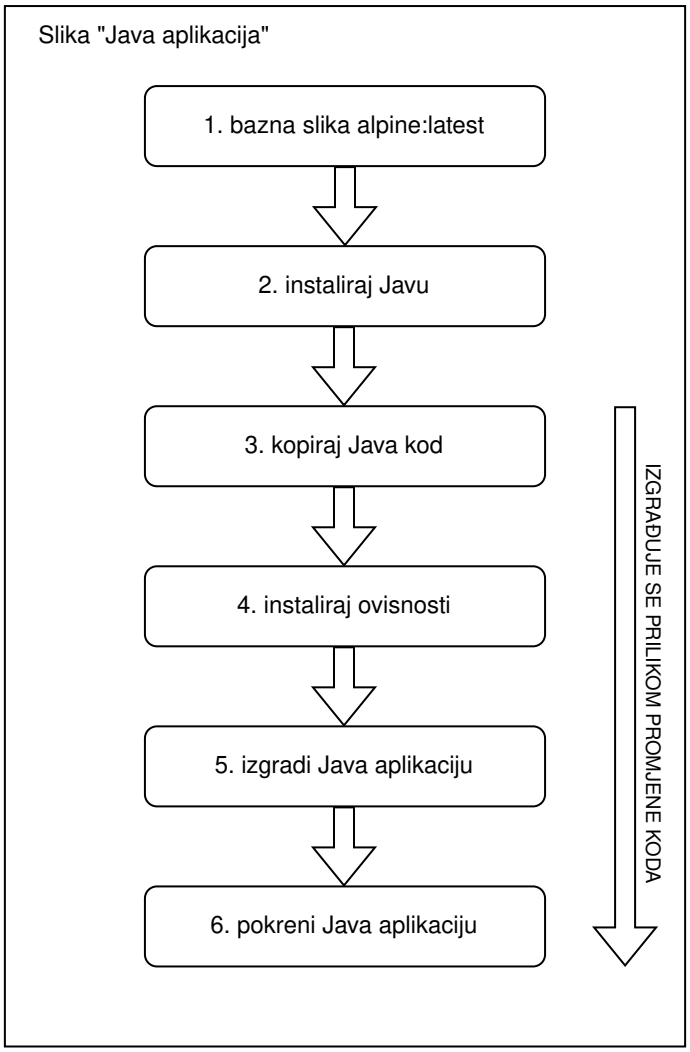
U današnje je vrijeme iznimno popularno pokretanje kontejnera u tzv. serverless okruženjima. Serverless ne znači da se kontejneri ne izvršavaju na poslužiteljima, već da se sam korisnik serverless usluga ne mora brinuti o infrastrukturi i najmu poslužiteljskih računala. To su usluge na koje se može prenijeti kontejner, a one će obaviti sve ostalo. Takve će usluge po potrebi pokrenuti još kontejnera kada je promet visok i smanjit će broj pokrenutih kontejnera kada nema potrebe da ih se pokreće toliko. Ovakvo skaliranje i raspoređivanje prometa radi i Kubernetes - jedina je razlika to što su upravljane usluge uglavnom zatvorene, posjedovane od strane neke velike korporacije, manje fleksibilne, ali i jeftinije. Kod ovakvih usluga uvijek postoji rizik od tzv. zaključavanja od strane pružatelja usluga. To znači da izrađeni softver koji se pokreće na takvoj upravljanoj usluzi radi samo na toj usluzi i ne može se migrirati nigdje drugdje. Kubernetes je otvorena platforma i to je čini vrlo privlačnom velikim korporacijama koje ne žele ovisiti o samo jednom pružatelju usluga.

3.1. Slike kontejnera

Slika je predložak za čitanje koji sadrži upute za kreiranje kontejnera. O slici se može razmišljati kao o nacrtu koji prikazuje što će biti u kontejneru kada se on pokrene [3].

Slika se sastoji od više naslaganih slojeva. Svaki od slojeva mijenja nešto u softverskom okruženju. Slika ima ovisnost o ljusci domaćinskog računala [3].

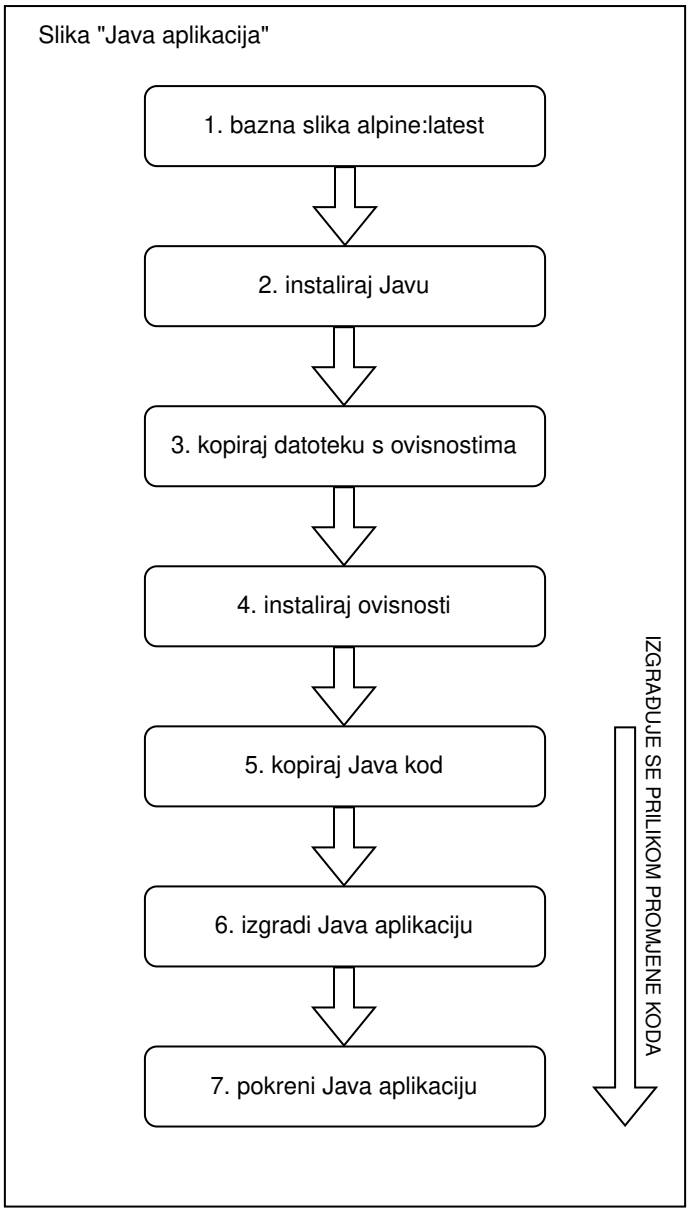
Činjenica da se slike sastoje od slojeva iznimno je pogodna kod njihove izgradnje jer se svi nepromijenjeni koraci izgradnje slike spremaju u predmemoriju. Zadana je sljedeća struktura slike kontejnera:



Slika 1: Slojevita struktura neefikasne slike kontejnera (Izvor: autorski rad)

Ovakva bi se slika za izgradnju kontejnera smatrala neefikasnom u okruženju gdje se konstantno razvija softver. Razlog tome je taj da bi se svaki puta kada bi programer promijenio kod ponovno izvršili koraci 3, 4, 5 i 6 zato što se kod mora ponovno kopirati kada se promijeni (korak 3). Svaki se korak nakon tog koraka također mora ponovno izvršiti. To znači da bi programer svaki puta kada promijeni kod trebao opet instalirati sve ovisnosti Java aplikacije, iako se one možda nisu promijenile.

Zadana je sljedeća struktura slike kontejnera:



Slika 2: Slojevita struktura efikasne slike kontejnera (Izvor: autorski rad)

Ovo se smatra puno efikasnijom slikom za izgradnju kontejnera nego što bi to bila ona iz prethodnog primjera. Sada će se, prilikom promijene koda, korak za instaliranje ovisnosti aplikacije povući iz predmemorije i neće se trebati ponovno izvršiti. Jedini koraci koji će se ponovno izvršiti su 5, 6 i 7.

3.1.1. Dockerfile

U današnje se vrijeme uglavnom koristi Dockerfile sintaksa za kreiranje slika kontejnera. Dockerfile je skripta koja opisuje korake koje Docker treba poduzeti za izgradnju nove slike. Te se datoteke distribuiraju zajedno sa softverom za koji autor želi da se stavi u sliku kontejnera, a kasnije i u kontejner. U ovom slučaju razvojni programer ne preuzima već gotovu izgrađenu sliku, već upute za izgradnju te slike. Uobičajena praksa je distribuirati Dockerfile uz softver u

sustavu za verzioniranje kao što je Git [4].

Kao primjer može se uzeti repozitorij autora knjige iz izvora i pomoću Dockerfile-a unutar repozitorija može se kreirati Docker slika kontejnera.

```
git clone https://github.com/dockerinaction/ch3_dockerfile.git
docker build -t dia_ch3/dockerfile:latest ch3_dockerfile
```

Dockerfile u tom repozitoriju ima sljedeći sadržaj:

```
FROM busybox:latest
MAINTAINER dia@allingeek.com
ADD demo.sh /demo/
WORKDIR /demo/
CMD ./demo.sh
```

Iz ovog Dockerfile-a vidljivo je da se koristi bazna slika busybox:latest što znači najnovija verzija slike busybox. Busybox kombinira male verzije čestih UNIX alata u jedan mali izvršni paket [5]. Održavatelj je definiran kao dia@allingeek.com - taj se podatak koristi samo kao informacija o tome tko je kreirao kontejner. Naredba ADD će kopirati datoteku demo.sh (izvršna skripta) u mapu /demo/ unutar kontejnera. Nakon toga, kontekst kontejnera pozicionira se u mapu /demo/ i pokreće skriptu demo.sh. CMD ili ENTRYPOINT uvijek je zadnja naredba u Dockerfile-u i to je jedina naredba koja ne ulazi u proces izgradnje kontejnera. Ta se naredba izvršava samo prilikom pokretanja kontejnera, a ne prilikom izgradnje.

Izvršna skripta demo.sh ima sljedeći sadržaj:

```
#!/bin/sh
echo This image was built from a Dockerfile
```

Iz sadržaja se vidi da će skripta samo ispisati tekst "This image was built from a Dockerfile".

Kod sustava baziranih na UNIX-u bitno je dati +x ovlast izvršnoj skripti jer se u Dockerfile datoteke kopiraju s istim ovlastima kao i na domaćinskom sustavu. Ako datoteka nema ovlasti za izvršavanje, u kontejneru se neće moći izvršiti ta skripta.

Izvršavanjem prethodno navedenih naredbi dobiva se sljedeći izlaz [4]:

```
% git clone https://github.com/dockerinaction/ch3_dockerfile.git
Cloning into 'ch3_dockerfile'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 7 (delta 0), reused 7 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
```

```
% docker build -t dia_ch3/dockerfile:latest ch3_dockerfile
[+] Building 7.4s (8/8) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 139B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/busybox:latest
 => [internal] load build context
 => => transferring context: 88B
 => [1/3] FROM docker.io/library/busybox:latest@sha256:...
 => => resolve docker.io/library/busybox:latest@sha256:...
 => => sha256:... 2.29kB / 2.29kB
 => => sha256:... 528B / 528B
 => => sha256:... 1.47kB / 1.47kB
 => => sha256:... 1.92MB / 1.92MB
 => => extracting sha256:...
 => [2/3] ADD demo.sh /demo/
 => [3/3] WORKDIR /demo/
 => exporting to image
 => => exporting layers
 => => writing image sha256:...
 => => naming to docker.io/dia_ch3/dockerfile:latest
```

Prilikom izgradnje kontejnera jasno se vide slojevi - oni su označeni s [1/3], [2/3] i [3/3].

Nakon izvršavanja ovih naredbi na računalu naredbom "docker images" može se vidjeti da je slika kontejnera stvarno izgrađena i spremna za korištenje.

```
% docker images
REPOSITORY           TAG       IMAGE ID       CREATED         SIZE
dia_ch3/dockerfile   latest    241c449c9a37   7 minutes ago   4.04MB
```

Za pokretanje kontejnera iz ovakve slike kontejnera koristi se naredba "docker run".

```
% docker run dia_ch3/dockerfile
This image was built from a Dockerfile
```

Naredba "echo" koju poziva skripta demo.sh unutar Dockerfile-a radi na način da ispiše tekst i odmah izađe iz programa. Zbog toga se pokrenut kontejner iz primjera odmah ugasi nakon što ispiše poruku.

Da se u kontejneru poziva skripta koja ne izlazi odmah nakon pokretanja, kontejner bi se nastavio pokretati u pozadini te bi njime upravljao Docker daemon.

Naredba "docker run" može se pozvati više puta i to će kreirati više kontejnera. Ta naredba također sadrži mnoge opcije u obliku parametara, a njima se može, na primjer, preslikati

pristup (port) iz kontejnera na domaćinsko računalo, spojiti mapa s domaćinskog računala na kontejner, itd.

3.2. Open Container Initiative (OCI) i Cloud Native Computing Foundation (CNCF)

Jedna od prvih inicijativa koje je dobila široki angažman industrije je Open Container Initiative (OCI). Kreiranju inicijative OCI doprinjelo je 36 suradnika među kojima su Docker, Red Hat, VMWare, IBM, Google i AWS. Svrha OCI je odvajanje standarda i formata kontejnera od same implementacije kontejnera kao što su npr. Docker ili Linux kontejneri (LXC) [6].

Cilj OCI specifikacije ima tri osnovna načela [6]:

- Stvaranje formalne specifikacije za formate slika kontejnera i okruženja u kojem se pokreću koji će biti kompatibilni sa svim sukladnim operacijskim sustavima i platformama.
- Prihvaćanje, održavanje i unapređenje projekata povezanih s OCI standardima. Gleda se da se utvrdi standardizirani skup radnji nad kontejnerima (pokretanje, izvršavanje, pauziranje...) i okruženja u kojem se pokreću.
- Harmoniziranje prijašnje spomenutog standarda s ostalim predloženim standardima.

Druga inicijativa koja je također široko prihvaćena u industriji je Cloud Native Comupting Foundation (CNCF). Ta je inicijativa i dalje usredotočena na rad s kontejnerima, ali se više odnosi na razinu dizajna aplikacije. Svrha je pružiti standardni skup alata i tehnologija za izgradnju, rad i upravljanje aplikacijama u oblaku [6].

CNCF se posebno usredotočuje na novu paradigmu mikroservisno orijentiranog razvoja aplikacija. Kao jedan od osnivača CNCF, Google je donirao Kubernetes projekt otvorenog koda kao prvi korak. Cilj je bio povećati interoperabilnost u ekosustavu i podržavati bolju integraciju s projektima [6].

3.2.1. Standardna specifikacija kontejnera

Ključan rezultat rada OCI je stvaranje i razvoj sveobuhvatne specifikacije kontejnera [6]. Specifikacija ima pet temeljnih načela kojih se moraju pridržavati sve implementacije kontejnera [6]:

- Kontejner mora imati standardne operacije za kreiranje, pokretanje i zaustavljanje kroz sve implementacije.
- Kontejner mora biti agnostičan na svoj sadržaj. To znači da tip aplikacije koji je sadržan u kontejneru ne utječe na standardne operacije ili na objavljivanje samog kontejnera.
- Kontejner mora biti agnostičan na infrastrukturu. Prenosivost je od ključne važnosti. Kontejneri moraju jednako funkcionirati na svim računalima neovisno radi li se o poslužitelju, osobnom računalu ili nekakvom trećem računalu.

- Kontejner mora biti dizajniran za automatizaciju. Ovaj je korak dosta nespecifičan, ali nalaže da kontejneri ne bi trebali zahtijevati tegobne ručne operacije da bi se kreirali, isporučili ili pokrenuli.
- Implementacija mora podržavati industrijsku isporuku. Ovaj korak također se odnosi na automatizaciju i nalaže da mora postojati jednostavan način za isporuku kontejnera.

3.3. Zašto su kontejneri bolja alternativa?

Glavne razlike između kontejnera i virtualnih strojeva odnose se na njihovu veličinu i prenosivost [7].

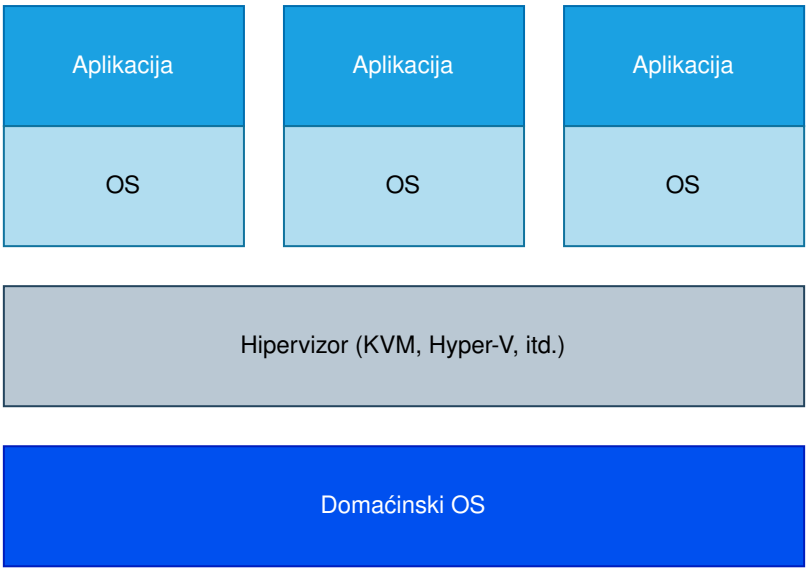
- Kontejneri se obično mjere u megabajtima. U sebe ne pakiraju ništa više nego aplikaciju i njene ovisnosti koje su potrebne za pokretanje aplikacije. Često se koriste za pakiranje jednostavnih funkcija (mikroservisa) koje izvršavaju specifične zadaće. Lagana priroda kontejnera i njihov dijeljeni operacijski sustav čine ih iznimno jednostavnim za prenošenje između okruženja.
- Virtualni strojevi se obično mjere u gigabajtima. Obično u sebi sadrže cijeli operacijski sustav što im dozvoljava da izvršavaju više intenzivnih funkcija odjednom. Povećani resursi dostupni virtualnim strojevima dozvoljavaju im da apstrahiraju, podijele, dupliciraju ili oponašaju cijele poslužitelje, operacijske sustave, računala, baze podataka i mreže.

Iznad tehničkih razlika, uspoređivanje kontejnera i virtualnih strojeva vodi do usporedbe IT praksa u nastajanju s tradicionalnim IT arhitekturama [7].

IT prakse u nastajanju moguće su zato što su poslovi podijeljeni u najmanje moguće upravljive jedinice - uglavnom funkcije ili mikroservisi. Ovakve male jedinice najbolje se mogu zapakirati u kontejnere što dozvoljava da više timova radi na zasebnim dijelovima aplikacije ili usluge bez da utječu na tuđi kod i rad [7].

Tradicionalne IT arhitekture monolitne su i sadrže svaki aspekt rada u jednom velikom tipu datoteka koji se ne može razdvajati i treba biti zapakiran kao cjelina unutar veće okoline što je često virtualni stroj. Nekad je bilo uobičajeno izgraditi i posluživati cijelu aplikaciju ili uslugu unutar virtualnog stroja, ali držanje cijelog koda i njegovih ovisnosti na jednom mjestu dovodilo je do prevelikih virtualnih strojeva koji su doživljavali kaskadni zastoj i nedostupnost prilikom ažuriranja [7].

Kontejneri također značajno olakšavaju proces skaliranja aplikacije. Kada su aplikacija ili usluga isporučeni na virtualnom stroju, nije baš lako samo duplicirati taj stroj u kratkom roku da bi se nastavio normalan rad. U takvoj situaciji treba pokrenuti još jedan virtualni stroj. S potpunom automatizacijom i unaprijed pripremljenim slikama virtualnih strojeva ovakav proces će trajati minimalno nekoliko minuta. Puno je lakše samo pokrenuti novi kontejner na bilo kojem od fizičkih poslužitelja. S obzirom da su slike kontejnera već izgrađene, pokretanje novog kontejnera u praksi često traje manje od sekundu. Također, u današnje vrijeme postoje



Slika 3: Arhitektura virtualiziranih aplikacija (Izvor: RedHat, 2020)

mnoge usluge na koje je samo potrebno prenijeti izgrađenu sliku kontejnera, a te će usluge dalje pokretati i gasiti kontejnere po potrebi s iznimno brzim vremenom odaziva. Prelazak IT industrije na kontejnere velikim je korporacijama značajno olakšao skaliranje svojih aplikacija i usluga. Uglavnom su sve velike usluge danas pogonjene kontejnerima.

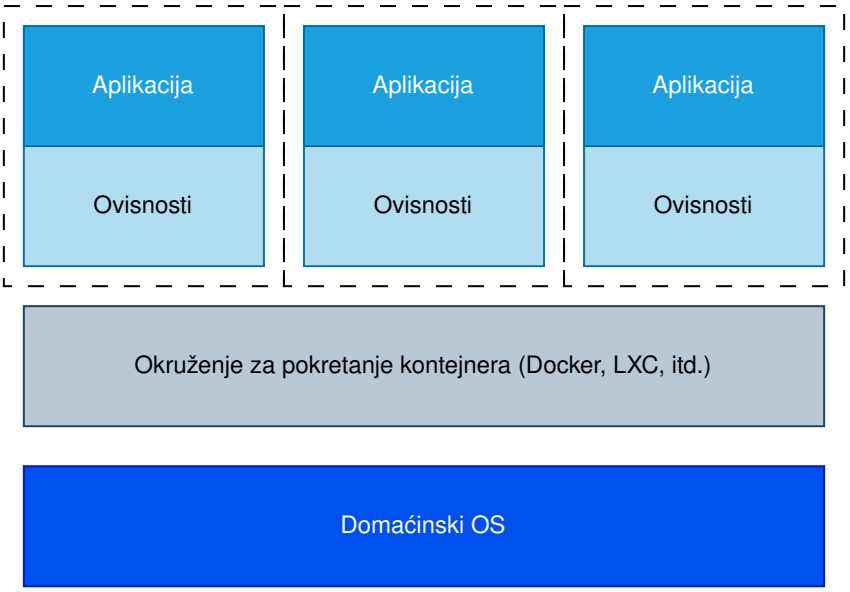
3.4. Podman ili Docker?

Jedna od glavnih prednosti Podmana je ta da je nastao mnogo kasnije od Dockera. Razvojni programeri Podmana gledali su načine da poboljšaju Dockerov dizajn iz potpuno drugačije perspektive. S obzirom na to da je Docker softver otvorenog koda, Podman dijeli neke dijelove koda s Dockerom i iskorištava nove standarde kao što je OCI. Podman surađuje sa zajednicom u dodavanju novih mogućnosti [8].

Tablica u nastavku uspoređuje mogućnosti Podmana i Dockera.

Tablica 1: Usporedba mogućnosti Podmana i Dockera

Mogućnost	Podman	Docker	Opis
Podržava sve OCI i Docker slike	✓	✓	Povlači i pokreće kontejnerske slike iz registara.
Pokreće OCI pokretače kontejnera	✓	✓	Pokreće runc, crun, Kata, gVisor i OCI kompatibilne pokretače.
Jednostavno naredbeno sučelje	✓	✓	Podman i Docker dijele isto naredbeno sučelje
Integracija sa systemd	✓	×	Podman podržava pokretanje systemd unutar kontejnera kao i mnoge systemd mogućnosti.



Slika 4: Arhitektura aplikacija u kontejnerima (Izvor: RedHat, 2020)

Tablica 1: Usporedba mogućnosti Podmana i Dockera

Fork/exec model	✓	×	Kontejner je dijete naredbe
Potpuno podržava korisnički imenski prostor	✓	×	Podman podržava pokretanje kontejnera u različitim korisničkim imenskim prostorima
Model klijent-poslužitelj	✓	✓	Docker je REST API daemon. Podman podržava REST API kroz systemd uslugu aktiviran utičnicom
Podržava docker-compose	✓	✓	Compose skripte rade na oba REST API-ja. Podman radi bez administratorskih privilegija.
Podržava docker-py	✓	✓	Docker-py Python funkcije rade na oba REST API-ja.
Ne zahtjeva daemon (daemonless)	✓	×	Podman radi kao tradicionalni alat u naredbenom retku, dok Docker zahtjeva nekoliko daemona pokrenutih s administratorskim pravima.
Podržava Kubernetes grupe	✓	×	Podman podržava pokretanje više kontejnera unutar jedne grupe.

Tablica 1: Usporedba mogućnosti Podmana i Dockera

Podržava Kubernetes YAML	✓	×	Podman može pokretati kontejnere i grupe na temelju Kubernetes YAML datoteka. Također može generirati Kubernetes YAML datoteke temeljene na trenutno pokrenutim kontejnerima.
Podržava Docker Swarm	×	✓	Podman vjeruje da je budućnost orkestriranja kontejnera Kubernetes i ne planira podržavati Swarm.
Prilagodljivi registri	✓	×	Podman dozvoljava konfiguraciju registra za ekspanziju kratkih imena. Docker je zaključan na docker.io kada se specificira kratko ime.
Prilagodljive zadane postavke	✓	×	Podman podržava prilagodbu svih zadanih postavki uključujući sigurnosne postavke, imenske prostore i volumene.
macOS podrška	✓	✓	Podman i Docker podržavaju pokretanje unutar macOS operacijskog sustava koristeći virtualni stroj koji pokreće Linux.
Windows podrška	✓	✓	Podman i Docker podržavaju pokretanje unutar Windows operacijskog sustava koristeći WSL2 ili virtualni stroj koji pokreće Linux.
Linux podrška	✓	✓	Podman i Docker su podržani na svim glavnim Linux distribucijama.
Kontejneri nisu zaustavljeni kod nadogradnje softvera	✓	×	Podman se ne treba nastaviti pokretati kada se kontejneri pokreću. Kod Dockera daemon nadgleda kontejnere i kada on prestane raditi, prestaju raditi i kontejneri.

Izvor: Walsh, 2023

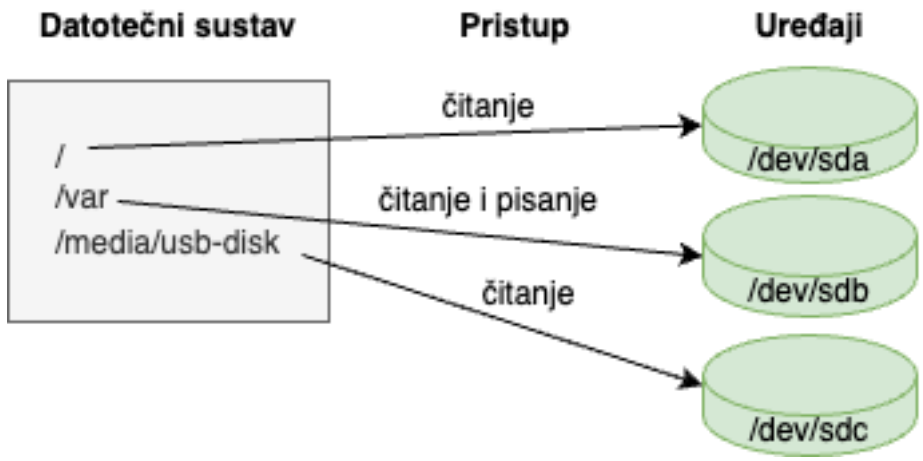
3.5. Montiranje datotečnog sustava

Ako se uzme u obzir, na primjer, pokretanje baze podataka unutar kontejnera, softver bi se zapakirao u kontejner, kontejner bi se pokrenuo i krenuo zapisivati podatke unutar konteksta kontejnera. Što ako treba ugasiti stari kontejner i pokrenuti novi? U tom bi se slučaju obrisali svi podaci iz baze podataka u kontejneru [4].

Kao rješenje ovog problema dosljednosti, u kontejnerima postoje razni načini za montiranje datotečnog sustava domaćinskog računala [4].

3.5.1. Stabla datoteka i točke montiranja

Za razliku od ostalih operacijskih sustava, Linux ujedinjuje svu pohranu u jedno stablo. Uređaji za pohranu poput particija na disku montirani su na određena mjesta u stablu datoteka. Ta se mjesta nazivaju točkama montiranja. Točka montiranja definira lokaciju u stablu, svojstva pristupa podacima na toj točki i izvor podataka montiranih na toj točki [4].



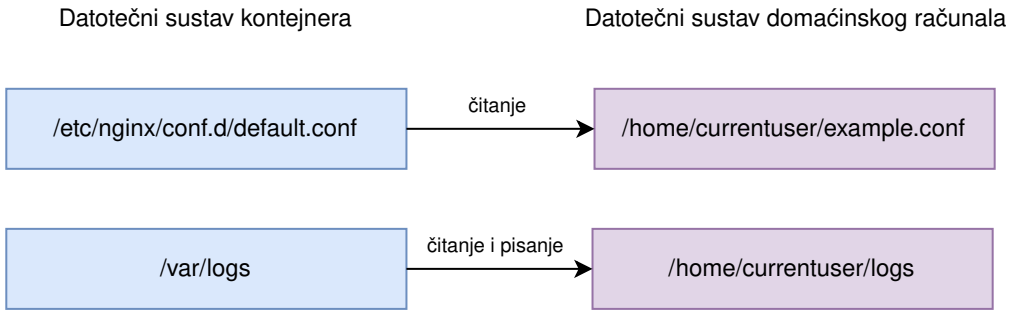
Slika 5: Uređaji za pohranu spojeni na stablo datotečnog sustava na svojim točkama montiranja (Izvor: Nickoloff i Kuenzli, 2019)

Točke montiranja omogućuju softveri i korisnicima korištenja stabla datoteka u Linux okruženju ne znajući točno kako se to stablo preslikava u određeje uređaje za pohranu. Ovo je osobito korisno u kontejnerskim okruženjima. Svaki spremnik ima nešto što se zove MNT imenski prostor i jedinstveni korijen stabla datoteka [4].

Dovoljno je razumjeti da je slika iz koje je kreiran kontejner montirana na taj kontejner u izvorištu njegovog stabla datotečnog sustava. Iz toga slijedi logika da se u kontejner mogu montirati razni uređaji na razne točke u stablu datotečnog sustava. Na ovaj način kontejneri mogu dobiti pristup datotečnom sustavu domaćinskog računala i mogu dijeliti pohranu s drugim kontejnerima [4].

3.5.1.1. Vezane točke montiranja

Vezane točke montiranja (bind mounts) koriste se za montiranje dijelova stabla datotečnog sustava na druge lokacije. Kada se radi s kontejnerima, vezane točke montiranja spajaju korisničko definiranu lokaciju na datotečnom sustavu domaćinskog računala na specifičnu točku u kontejnerovom stablu datotečnog sustava. Vezane točke montiranja korisne su kada domaćinsko računalo pruža datoteku ili mapu koju zahtjeva program unutar kontejnera ili kada kontejner proizvodi datoteke ili zapise koji se moraju obraditi od strane korisnika ili programa izvan kontejnera [4].

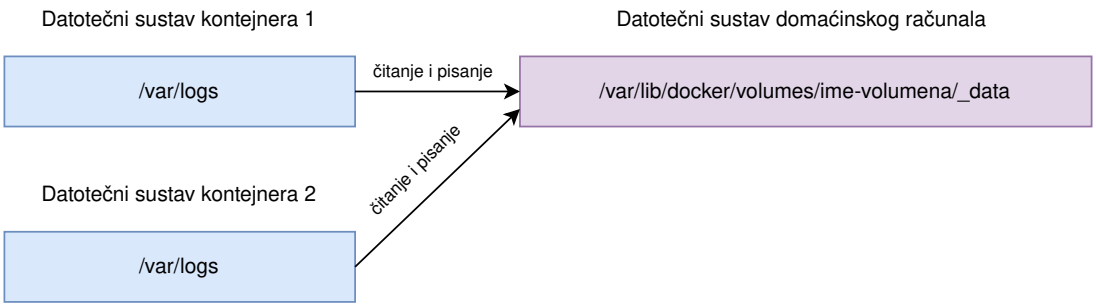


Slika 6: Vezane točke montiranja (Izvor: Nickoloff i Kuenzli, 2019)

3.5.2. Volumeni

Volumeni u Dockeru su imenovana stabla datotečnih sustava kojima upravlja Docker (ili neki drugi pokretač kontejnera). Oni mogu biti realizirani kao pohrana na disku domaćinskog računala, ali mogu biti i realizirani pomoću nešto egzotičnijeg rješenja kao što je cloud pohrana. Sve radnje nad Docker volumenima mogu se vršiti naredbom "docker volume". Korištenje volumena metoda je za razdvajanje pohrane iz specifičnih lokacija na datotečnom sustavu. Time se mogu izbjeći konflikti u lokacijama na datotečnom sustavu - svi se podaci nalaze u svom imenovanom volumenu. Dokle god su imena volumena drugačija, neće biti konflikata. [4]

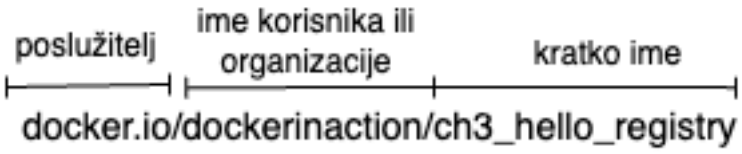
Standardno ponašanje Docker volumena je takvo da se koristi lokalni volumenski dodatak. To znači da je zadano ponašanje takvo da će Docker kreirati mapu na datotečnom sustavu domaćinskog računala kojom će upravljati Docker pokretač. Ovo ponašanje može se izmijeniti [4].



Slika 7: Volumeni (Izvor: Nickoloff i Kuenzli, 2019)

3.6. Repozitoriji slika

Imenovani repozitorij imenovani je spremnik slika kontejnera. Ime je slično URL-u. Ime repozitorija sastoji se od imena poslužitelja gdje se slika nalazi, korisničkog imena računa koji posjeduje sliku i kratkog imena [4].



Slika 8: Ime repozitorija (Izvor: Nickoloff i Kuenzli, 2019)

Isto kao što može postojati više verzija softvera, tako i repozitorij može posluživati više slika kontejnera. Svaka slika u repozitoriju jedinstveno je identificirana oznakom (tag). Na primjer, u gore navedeni repozitorij može se prenijeti nova verzija slike ch3_hello_registry i ta bi se verzija mogla označiti kao v2. Tada bi potpuno ime repozitorija glasilo [4]:

```
docker.io/dockerinaction/ch3_hello_registry:v2
```

Stara se verzija onda, na primjer, može označiti verzijom v1 i tada je uvijek moguć pristup prošloj verziji slike kontejnera [4].

3.6.1. Docker Hub

Docker Hub najveći je svjetski repozitorij slika kontejnera iz mnogih izvora kao što su razvojni programeri iz zajednice, projekti otvorenog koda i neovisni softverski prodavači. Korisnici dobivaju pristup besplatnim javnim repozitorijima za pohranu i dijeljenje slika kontejnera ili mogu izabrati plan pretplate za privatne repozitorije [9].

Neke od ključnih mogućnosti koje nudi Docker Hub su [9]:

- privatni repozitoriji: prenašanje i preuzimanje slika kontejnera
- automatizirana izgradnja: slike kontejnera automatski se izgrađuju iz GitHub-a ili Bitbucket-a i prenašaju se na Docker Hub
- timovi i organizacije: upravljanje pristupa privatnim repozitorijima
- službene slike kontejnera: preuzimanje i korištenje službenih slika kontejnera pružanih od strane Dockera
- izdavačke slike kontejnera: preuzimanje i korištenje visokokvalitetnih slika kontejnera pružanih od strane neovisnih prodavača
- pozivanje radnji nakon uspješnog prenašanja na repozitorij u svrhu integracije s drugim uslugama

4. Osnovna orkestracija kontejnera

Kubernetes je orkestrator aplikacija. Uglavnom služi za orkestriranje aplikacija u oblaku.

4.1. Što je orkestrator?

Orkestrator je sustav koji isporučuje aplikacije i upravlja njima. Može rasporediti aplikacije po računalima i dinamički reagirati na promjene. Na primjer, Kubernetes može [1]:

- isporučivati aplikacije
- dinamički skalirati aplikacije ovisno o prometu
- samoizliječiti se u slučaju greške
- ažurirati aplikacije bez zastoja u radu

Najbolja stvar kod Kubernetesa je ta što se sve to odrađuje bez potrebe za ljudskim nadgledanjem sustava. U početku je potrebno sve postaviti, ali jednom kada je sve postavljeno dovoljno je pustiti Kubernetes da radi svoje.

4.2. Umrežavanje kontejnera

U Dockeru, kao i ostalim pokretačima kontejnera, postoji koncept mreža.

Docker apstrahira mrežu domaćinskog računala i mreže kontejnera. Kontejner spojen na Docker mrežu dobit će jedinstvenu IP adresu do koje se može doći s drugih kontejnera na istoj mreži. Glavni problem ovog pristupa je taj što kontejner sam unutar sebe ne može znati koja je IP adresa drugih kontejnera s obzirom da ona nije unaprijed određena [4]. Adresu je moguće odrediti unaprijed, ali kontejneri sami po sebi ne mogu otkrivati IP adrese drugih kontejnera unutar mreže.

Docker tretira mreže kao objekte prvog reda. To znači da mreže nisu vezane uz kontejnere i da se mreže ne gase ili brišu kada to rade kontejneri [4].

U sljedećem jednostavnom primjeru će dva kontejnera komunicirati putem Docker mreže. Kao poslužitelj i klijent koristit će se netcat.

Prvo je potrebno kreirati docker mrežu naredbom docker network create:

```
% docker network create netcatPrimjer
a596460f080ee52db27b64982c249bd6367c2ae7bd25bfc49d50e4ce92f8a709
```

Nakon kreiranja mreže moguće je dohvatiti podatke o njoj naredbom docker network inspect:

```
% docker network inspect netcatPrimjer
[
    {
        "Name": "netcatPrimjer",
        "Id": "a596460f080ee52db27b64982c249bd6367c2ae7bd25bfc49d50e
        ↪  4ce92f8a709",
        "Created": "2023-08-20T18:36:20.894074466Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Vidljivo je da je ovo mreža kreirana "bridge" pokretačem. Bridge je zadani pokretač koji se koristi kada jedan kontejner treba komunicirati s drugim unutar mreže.

Nakon kreiranja mreže, potrebno je izgraditi dvije slike kontejnera koje će se pokretati - jedna koja će služiti kao netcat poslužitelj, a jedna kao klijent koji se spaja na poslužitelja.

Sadržaj datoteke Dockerfile-server:

```
FROM alpine:3.18.3
RUN apk add netcat-openbsd
CMD ["nc", "-l", "2399"]
```

Sada treba izgraditi sliku kontejnera pomoću naredbe *docker build*.

```
% docker build -f Dockerfile-server -t "ncatserver:latest" .
[+] Building 6.6s (6/6) FINISHED
 => [internal] load build definition from Dockerfile-server
  ↪  0.0s
 => => transferring dockerfile: 120B
  ↪  0.0s
 => [internal] load .dockerignore
  ↪  0.0s
 => => transferring context: 2B
  ↪  0.0s
 => [internal] load metadata for docker.io/library/alpine:3.18.3
  ↪  3.6s
 => [1/2] FROM docker.io/library/alpine:3.18.3@sha256:...
  ↪  1.2s
 => => resolve docker.io/library/alpine:3.18.3@sha256:...
  ↪  0.0s
 => => sha256:... 1.64kB / 1.64kB
  ↪  0.0s
 => => sha256:... 528B / 528B
  ↪  0.0s
 => => sha256:... 1.49kB / 1.49kB
  ↪  0.0s
 => => sha256:... 3.33MB / 3.33MB
  ↪  0.9s
 => => extracting sha256:...
  ↪  0.2s
 => [2/2] RUN apk add netcat-openbsd
  ↪  1.7s
 => exporting to image
  ↪  0.0s
 => => exporting layers
  ↪  0.0s
 => => writing image sha256:...
  ↪  0.0s
 => => naming to docker.io/library/ncatserver:latest
  ↪  0.0s
```

Pokretanje poslužitelja:

```
% docker run -d --network netcatPrimjer ncatserver:latest
```

Parametrom -d osigurava se da će se kontejner pokrenuti u "detached" načinu rada, tj. da njegovo pokretanje neće biti vezano uz trenutnu ljusku. Na taj se način postiže to da kontejner radi u pozadini, a korisnik može dalje pozivati naredbe.

Sada je potrebno saznati IP adresu poslužitelja unutar Docker mreže netcatPrimjer. To se može saznati naredbom *docker network inspect*.

```
% docker network inspect netcatPrimjer
[
    {
        "Name": "netcatPrimjer",
        "Id": "a596460f080ee52db27b64982c249bd6367c2ae7bd25bfc49d50e⌋
        ↪  4ce92f8a709",
        "Created": "2023-08-20T18:36:20.894074466Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "b202b04a2b86974c1fb06766460a2d8a405734dd44f90ab7b3dc65b⌋
            ↪  ccebf342e":
            ↪  {
                "Name": "stoic_greider",
                "EndpointID": "6ca4a10b2cc02bcd0ccc32b7317e12e6f65e1⌋
                ↪  8e746eef475faa4437681074eac",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
```

```
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Vidljivo je da je IP adresa netcat poslužitelja 172.18.0.2. Tu adresu treba unijeti u Dockerfile klijenta.

Sadržaj datoteke Dockerfile-client:

```
FROM alpine:3.18.3
RUN apk add netcat-openbsd
CMD ["/bin/sh", "-c", "echo test | nc 172.18.0.2 2399"]
```

U obje Docker slike koristi se bazna slika alpine (Alpine Linux), instalira se netcat i pokreće se netcat poslužitelj ili klijent po potrebi.

Sliku klijenta sada je potrebno izgraditi naredbom *docker build*.

```
% docker build -f Dockerfile-client -t "ncatclient:latest" .
[+] Building 0.9s (6/6) FINISHED
 => [internal] load build definition from Dockerfile-client
  ↪  0.0s
 => => transferring dockerfile: 151B
  ↪  0.0s
 => [internal] load .dockerignore
  ↪  0.0s
 => => transferring context: 2B
  ↪  0.0s
 => [internal] load metadata for docker.io/library/alpine:3.18.3
  ↪  0.9s
 => [1/2] FROM docker.io/library/alpine:3.18.3@sha256:...
  ↪  0.0s
 => CACHED [2/2] RUN apk add netcat-openbsd
  ↪  0.0s
 => exporting to image
  ↪  0.0s
 => => exporting layers
  ↪  0.0s
 => => writing image sha256:...
  ↪  0.0s
 => => naming to docker.io/library/ncatclient:latest
  ↪  0.0s
```

Iz ispisa naredbe vidljivo je da se slika pravilno izgradila. Slijedi pokretanje kontejnera iz te slike na mreži netcatPrimjer.

```
% docker run --network netcatPrimjer ncatclient:latest
```

Ako se sve izvršilo i umrežilo kako treba, u zapisima poslužiteljskog kontejnera trebala bi se vidjeti poruka koja je stigla s klijentskog kontejnera.

Za provjeravanje zapisa kontejnera prvo treba saznati kontejnerov identifikator, a to se radi naredbom docker ps:

```
% docker ps
CONTAINER ID   IMAGE               COMMAND        CREATED
 ↪  STATUS         PORTS                    NAMES
78638f705b5d   ncatserver:latest   "nc -l 2399"   3 minutes ago   Up
 ↪  3 minutes   0.0.0.0:2399->2399/tcp   charming_brahmagupta
```

Bez ikakvih parametara, naredba docker ps prikazuje sve kontejnere koji se trenutno pokreću na sustavu. Ako je potrebno vidjeti sve kontejnere, pa čak i one ugašene, potrebno je dodati parametar -a.

Za pregledavanje zapisa koristi se naredba docker logs:

```
% docker logs 78638f705b5d
test
```

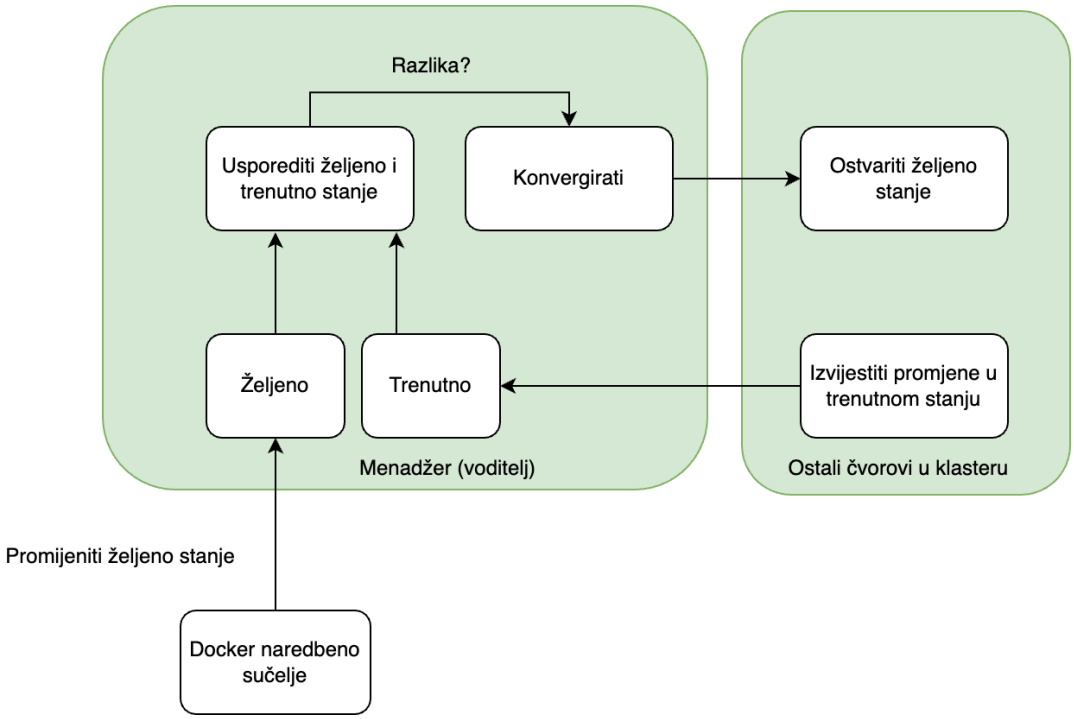
Naredba docker logs daje izlaz "test", a to je poruka koja je poslana s netcat klijenta prema poslužitelju. Kontejneri su uspješno umreženi unutar mreže "netcatPrimjer".

4.3. Osnovna orkestracija alatom Docker compose

Korištenje običnih Docker naredbi jednu za drugom može postati naporno, pogotovo kada se radi o kompleksnijem sustavu s mnogo servisa. Ovo se zove imperativni uzorak. Imperativni alati pozivaju naredbe pozvane od strane korisnika. Naredbe mogu primati specifične informacije ili opisivati promjene. Programski jezici i alati u naredbenom retku prate imperativni uzorak [4].

Prednost imperativnih alata je ta da dozvoljavaju korisnicima da koriste primitivne naredbe da bi opisali mnogo kompleksnije tokove i sustave, ali naredbe moraju biti pozvane točnim redoslijedom tako da imaju ekskluzivnu kontrolu nad trenutnim stanjem. Ako više korisnika mijenja stanje sustava u isto vrijeme, tada može doći do nepredvidivih konflikata [4].

Docker servisi deklarativna su apstrakcija. U servisima se deklarira kakvo mora biti njihovo stanje, a Docker se pobrine da stanje bude takvo [4].



Slika 9: Deklarativna petlja obrade (Izvor: Nickoloff i Kuenzli, 2019)

Docker compose deklarira se YAML sintaksom. Potrebno je kreirati datoteku docker-compose.yml i u njoj opisati servise. U prije spomenutom primjeru nalazila bi se dva servisa: jedan kao poslužitelj, a drugi kao klijent.

Datoteka docker-compose.yml za netcat primjer ima sljedeći sadržaj:

```
version: "3.8"
services:
  server:
    build:
      context: ./
      dockerfile: Dockerfile-server
    networks:
      - netcatPrimjer
  client:
    build:
      context: ./
      dockerfile: Dockerfile-client
    networks:
      - netcatPrimjer
    depends_on:
      - server

networks:
  netcatPrimjer:
```

```
    driver: bridge
```

U compose datoteci vidi se da su deklarirana dva servisa, jedan za poslužiteljski kontejner, a jedan za klijentski. Atribut context označava mapu u kojoj se vrši izgradnja Docker slike, a atribut dockerfile definira koja će se Docker datoteka koristiti za izgradnju pojedinog servisa.

Na kraju compose datoteke deklarirana je mreža s bridge pokretačem koja služi za komunikaciju između kontejnera. Nažalost, prije same izgradnje kontejnera nemoguće je znati koju će IP adresu ta mreža poprimiti. Potrebno je prvo izgraditi kontejnere, a tek onda promijeniti IP adresu unutar Docker datoteka.

Također je deklarirano da klijentski servis ovisi o poslužiteljskom. To se deklarira atributom depends_on, a on omogućava da se prvo pokrene poslužiteljski kontejner, a tek onda klijentski.

Nakon deklariranja servisa u compose datoteci potrebno je pokrenuti te servise koristeći naredbu docker compose up.

```
% docker compose up -d
[+] Building 3.1s (8/8) FINISHED
 => [netcat-network-primjer-server internal] load build definition
  ↪  from Dockerfile-server                        0.0s
 => => transferring dockerfile: 115B
  ↪  0.0s
 => [netcat-network-primjer-client internal] load build definition
  ↪  from Dockerfile-client                        0.0s
 => => transferring dockerfile: 146B
  ↪  0.0s
 => [netcat-network-primjer-server internal] load .dockerignore
  ↪  0.0s
 => => transferring context: 2B
  ↪  0.0s
 => [netcat-network-primjer-client internal] load .dockerignore
  ↪  0.0s
 => => transferring context: 2B
  ↪  0.0s
 => [netcat-network-primjer-client internal] load metadata for
  ↪  docker.io/library/alpine:3.18.3                3.0s
 => [netcat-network-primjer-client 1/2] FROM
  ↪  docker.io/library/alpine:3.18.3@sha256:...  0.0s
 => CACHED [netcat-network-primjer-client 2/2] RUN apk add
  ↪  netcat-openbsd                                    0.0s
 => [netcat-network-primjer-server] exporting to image
  ↪  0.0s
 => => exporting layers
  ↪  0.0s
```

```
 => => writing image sha256:...                    0.0s
 => => writing image sha256:...                    0.0s
 => => naming to docker.io/library/netcat-network-primjer-client
  ↪  0.0s
 => => naming to docker.io/library/netcat-network-primjer-server
  ↪  0.0s

Use 'docker scan' to run Snyk tests against images to find
 ↪  vulnerabilities and learn how to fix them
[+] Running 3/3
 Network netcat-network-primjer_netcatPrimjer  Created
  ↪  0.1s
 Container netcat-network-primjer-server-1     Started
  ↪  0.5s
 Container netcat-network-primjer-client-1     Started
  ↪  0.6s

Attaching to netcat-network-primjer-client-1,
 ↪  netcat-network-primjer-server-1
netcat-network-primjer-server-1  | test
```

Nakon pokretanja servisa vidljiv je ispis "test" iz poslužiteljskog kontejnera što znači da su kontejneri umreženi.

Ako je potrebno ponovno izgraditi kontejnere, za to se koristi naredba docker compose build. To je naredba koju treba izvršavati prilikom promjene Docker datoteka ili datoteka u datotečnom sustavu kontejnera, a ako se mijenja samo compose datoteka i definicije servisa, tada nema potrebe za ponovnom izgradnjom kontejnera.

Za gašenje kontejnera koristi se naredba docker compose stop, a za potpuno brisanje svih resursa (servisa, mreža, slika...) koristi se naredba docker compose down.

Ranije je spomenuto da se ovakvim umrežavanjem kontejnera ne može unaprijed znati IP adresa određenog kontejnera. Ovo je velik problem u kompleksnijim sustavima, ali Docker compose ima rješenje za to. U mreži unutar Docker compose definicije moguće je koristiti samo imena servisa te će se ona prevesti u IP adrese. Ovim pristupom moguće je potpuno izbaciti bridge mrežu definiranu u compose datoteci.

Nova docker-compose.yml datoteka glasi:

```yaml
version: "3.8"
services:
  server:
    build:
      context: ./
      dockerfile: Dockerfile-server
```

```
  client:
    build:
      context: ./
      dockerfile: Dockerfile-client
    depends_on:
      - server
```

Jedina razlika je ta da su sada u compose datoteci potpuno maknute sve reference na mrežu.

Sada je potrebno urediti datoteku Dockerfile-client. Ta datoteka trenutno izgleda ovako:

```
FROM alpine:3.18.3
RUN apk add netcat-openbsd
CMD ["/bin/sh", "-c", "echo test | nc 172.18.0.2 2399"]
```

Potrebno je samo promijeniti IP adresu u ime servisa:

```
FROM alpine:3.18.3
RUN apk add netcat-openbsd
CMD ["/bin/sh", "-c", "echo test | nc server 2399"]
```

Docker će automatski prevesti adresu "server" u pravi IP adresu i na taj se način eliminira potreba za poznavanjem IP adrese unaprijed.

Sada je potrebno ponovno izgraditi kontejnere:

```
% docker compose build
[+] Building 3.2s (8/8) FINISHED
 => [netcat-network-primjer-server internal] load build definition
  ↪  from Dockerfile-server                        0.0s
 => => transferring dockerfile: 115B
  ↪  0.0s
 => [netcat-network-primjer-client internal] load build definition
  ↪  from Dockerfile-client                        0.0s
 => => transferring dockerfile: 142B
  ↪  0.0s
 => [netcat-network-primjer-server internal] load .dockerignore
  ↪  0.0s
 => => transferring context: 2B
  ↪  0.0s
 => [netcat-network-primjer-client internal] load .dockerignore
  ↪  0.0s
 => => transferring context: 2B
  ↪  0.0s
```

```
 => [netcat-network-primjer-client internal] load metadata for
  ↪  docker.io/library/alpine:3.18.3                3.1s
 => [netcat-network-primjer-client 1/2] FROM
  ↪  docker.io/library/alpine:3.18.3@sha256:...  0.0s
 => CACHED [netcat-network-primjer-client 2/2] RUN apk add
  ↪  netcat-openbsd                                    0.0s
 => [netcat-network-primjer-server] exporting to image
  ↪  0.0s
 => => exporting layers
  ↪  0.0s
 => => writing image sha256:...                   0.0s
 => => writing image sha256:...                   0.0s
 => => naming to docker.io/library/netcat-network-primjer-client
  ↪  0.0s
 => => naming to docker.io/library/netcat-network-primjer-server
```

Nakon izgradnje, potrebno je pokrenuti kontejnere naredbom docker compose up:

```
% docker compose up
[+] Running 3/2
 Network netcat-network-primjer_default     Created
 Container netcat-network-primjer-server-1  Created
 Container netcat-network-primjer-client-1  Created
Attaching to netcat-network-primjer-client-1, netcat-network-primjer-server-1
netcat-network-primjer-server-1  | test
```

U ispisu naredbe vidi se da je poslužiteljski kontejner primio poruku "test" i ispisao je.

4.4. Orkestracija koristeći Docker Swarm

Aplikacije se često isporučuju na više poslužitelja za bolju dostupnost i skalabilnost. Kada je aplikacija isporučena na više poslužitelja, redundantnost u isporuki aplikacije omogućava kapacitet za obradu zahtjeva čak i kada neki od poslužitelja ima greške u radu. Isporuka kroz više poslužitelja također omogućava aplikaciji da koristi više resursa nego što ih jedan poslužitelj može ponuditi [4].

Docker Swarm pruža sofisticiranu platformu za isporuku i upravljanje aplikacijama unutar kontejnera kroz više Docker poslužitelja. Dockerovi alati automatiziraju proces isporuke novog servisa ili promjena nad postojećim servisima [4].

Definicije servisa nalaze se u docker-compose.yml datoteci. U toj datoteci definiraju se slike kontejnera, naredbe, ograničenja resursa, broj replika, varijable [4]...

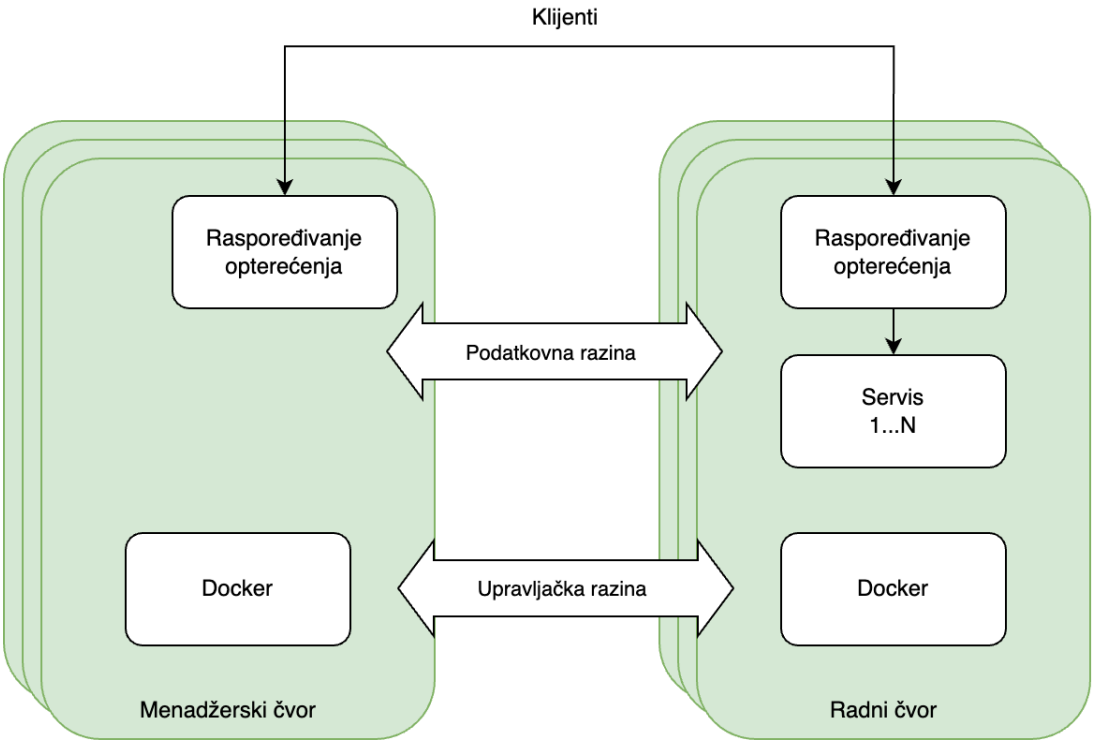
Jednom kada je aplikacija isporučena, Swarm nadgleda aplikaciju i brine se da se greške brzo otkriju i poprave [4].

4.4.1. Menadžerski i radni čvorovi

Kada se računalo pridruži Swarm klasteru, treba se definirati služi li to računalo kao radni čvor (worker) ili menadžerski čvor [4].

Menadžerski čvor sluša instrukcije za kreiranje, promjenu ili brisanje definicija za entitete kao što su Docker servisi, konfiguracije, mreže, itd. Menadžerski čvorovi naređuju radnim čvorovima da kreiraju kontejnere i volumene koje implementiraju instance Docker servisa. Menadžeri kontinuirano konvergiraju klaster u stanje koje je deklarirano u compose datoteci [4].

Upravljačka razina koja povezuje klastere prikazuje komunikaciju željenog stanja klastera i događaja vezanih za ostvarenje tog stanja. Klijenti Docker servisa mogu slati zahtjeve bilo kojem čvoru klastera na utičnici objavljenoj za tu uslugu. Swarmova će mreža usmjeriti zahtjev s čvora koji je primio zahtjev na čvor koji može podnijeti zahtjev. Swarm u sebi ima implementirano raspoređivanje opterećenja koje se brine da se promet rasporedi ravnomjerno za svaku objavljenu utičnicu, tj. servis [4].



Slika 10: Isporuka Swarm klastera (Izvor: Nickoloff i Kuenzli, 2019)

4.4.2. Isporuka Swarm klastera

Za isporuku Swarm klastera potrebno je prvo na svim poslužiteljima imati instaliran Docker.

U ovom će se primjeru koristiti Oracle E4 virtualne instance kao Swarm čvorovi.

Naivni način postavljanja ovakvog klastera bio bi takav da bi se ručno na svakom od čvorova instalirao Docker. Ovo postaje jako nepraktično kada se radi o većem broju čvorova. U praksi se nikada ručno ne postavljaju stotine čvorova, već se koriste razni alati, a najpopularniji od njih je RedHat Ansible.

4.4.2.1. RedHat Ansible

Ansible je softver za automatizaciju otvorenog koda koji se koristi kroz naredbeni redak. Može konfigurirati sustave, isporučiti softver i orkestrirati napredne tokove rada da bi podržavao razvoj softvera, ažuriranja sustava, itd [10].

Ansible je prvi put izdao Michael DeHaan 2012. kao mali hobi projekt i od tada je imao ogromni porast popularnosti s preko 17.000 GitHub zvjezdica i 1.410 jedinstvenih suradnika. GitHub zvjezdice način su praćenja napretka projekta, ali ti korisnici ne moraju nužno pridonositi kodu. Osim što je Ansible uspješan projekt otvorenog koda, uspješno ga koriste i velike tvrtke kao što su Apple i NASA. One se oslanjaju na Ansible za svoje potrebe upravljanja konfiguracijom [10].

Ansible koristi YAML datoteke kao glavni izvor informacija tijekom izvođenja. YAML je deklarativni podatkovni jezik koji se obično koristi za konfiguraciju. YAML datoteke jednostavne su za pisanje i većina razvojnih programera brzo se snalazi u njima nakon kratkog uvoda [10].

Ansible je u potpunosti napisan u Pythonu. DeHaan je izabrao Python za Ansible jer to znači da Ansible neće imati nikakve dodatne ovisnosti. Naime, danas svako Linux i Mac računalo unaprijed ima instaliran Python. Linux je operacijski sustav koji pogoni veliku većinu poslužitelja u današnje vrijeme, a to znači da takvi poslužitelji nemaju nikakvih dodatnih softverskih ovisnosti. Ansible pokreće sve naredbe putem SSH (secure shell) tako da nema potrebe za instaliranjem poslužiteljskog softvera. Ovo je velika prednost iz dva razloga [10]:

- Poslužitelji pokreću svoje aplikacije bez ikakvog dodatnog softvera u pozadini koji bi trošio resurse namijenjene za glavne aplikacije.
- Sve mogućnosti SSH dostupne su korištenjem Ansible alata. Ne treba izmišljati nikakav dodatan sustav autentifikacije, već se koriste standardni SSH mehanizmi poput SSH ključeva.

Instalacija Ansible alata iznimno je jednostavna i vrši se pomoću Pythonovog upravitelja paketima (pip):

```
% pip3 install ansible
```

Ako na računalu postoji instaliran Python 3, Ansible će biti uspješno instaliran i spreman za korištenje.

4.4.2.2. Postavljanje čvorova

Za instalaciju Docker softvera na menadžerskom i radnom čvoru koristit će se Ansible.

Prvo treba definirati inventar poslužitelja u datoteci hosts.yml:

```
oracleservers:
  hosts:
    master:
      ansible_host: 130.61.180.110
    worker:
      ansible_host: 138.2.141.229
  vars:
    ansible_user: ubuntu
```

U grupi oracleservers definirana su dva poslužitelja preko svoje IP adrese. Za autentifikaciju koristit će se SSH agent domaćinskog računala tako da ne treba definirati SSH ključeve u datotekama.

Za postavljanje Docker pokretaća na čvorovima koristit će se sljedeći Ansible playbook (playbook.yml):

```
---
- hosts: oracleservers
  become: true
  tasks:
    - name: Install aptitude
      apt:
        name: aptitude
        state: latest
        update_cache: true
    - name: Install required system packages
      apt:
        pkg:
          - apt-transport-https
          - ca-certificates
          - curl
          - software-properties-common
          - python3-pip
          - virtualenv
          - python3-setuptools
        state: latest
        update_cache: true
    - name: Add Docker GPG apt Key
```

```
      apt_key:
        url: https://download.docker.com/linux/ubuntu/gpg
        state: present
    - name: Add Docker Repository
      apt_repository:
        repo: deb https://download.docker.com/linux/ubuntu focal
↪   stable
        state: present
    - name: Update apt and install docker-ce
      apt:
        name: docker-ce
        state: latest
        update_cache: true
    - name: Install Docker Module for Python
      pip:
        name: docker
```

U navedenoj datoteci definirano je da se prvo instalira aptitude, instaliraju paketi potrebni za Docker, doda se Docker GPG ključ, doda se Docker deb repozitorij (s njega se preuzima deb paket za instalaciju Docker softvera), instalira se paket docker-ce (Docker bez grafičkog sučelja) i instalira se Docker Python modul.

U ovoj YAML datoteci zapravo je opisano stanje. Nije potrebno nizati naredbe, već je samo potrebno definirati što treba biti instalirano na čvorovima. Ansible operacije su idempotentne što znači da se mogu pozvati više puta, a Ansible će prepoznati trenutno stanje sustava i ažurirati samo stanja koja je potrebno ažurirati.

Sada samo treba pozvati ranije spomenut playbook naredbom ansible-playbook:

```
% ansible-playbook playbook.yml -i hosts.yml

PLAY [oracleservers]
↪   ***************************************************************

TASK [Gathering Facts]
↪   ***************************************************************
ok: [worker]
ok: [master]

TASK [Install aptitude]
↪   ***************************************************************
changed: [worker]
changed: [master]
```

```
TASK [Install required system packages]
 ↪  ************************************************************
changed: [worker]
changed: [master]

TASK [Add Docker GPG apt Key]
 ↪  ************************************************************
changed: [worker]
changed: [master]

TASK [Add Docker Repository]
 ↪  ************************************************************
changed: [worker]
changed: [master]

TASK [Update apt and install docker-ce]
 ↪  ************************************************************
changed: [worker]
changed: [master]

TASK [Install Docker Module for Python]
 ↪  ************************************************************
changed: [worker]
changed: [master]

PLAY RECAP
 ↪  ************************************************************
master                     : ok=7    changed=6    unreachable=0
 ↪  failed=0    skipped=0    rescued=0    ignored=0
worker                     : ok=7    changed=6    unreachable=0
 ↪  failed=0    skipped=0    rescued=0    ignored=0
```

U ispisu naredbe ansible-playbook vidi se da su sve operacije uspješno izvršeno na oba čvora. Na oba čvora piše failed=0 što znači da ne postoji naredba koja se nije uspjela izvršiti.

Nakon što je Ansible odradio svoju magiju lako je provjeriti imaju li oba čvora instaliran Docker:

```
ubuntu@diplomski-master:~$ docker -v
Docker version 24.0.5, build ced0996

ubuntu@diplomski-worker:~$ docker -v
Docker version 24.0.5, build ced0996
```

Vidljivo je da oba čvora odgovaraju na naredbu docker -v i time je potvrđeno da je

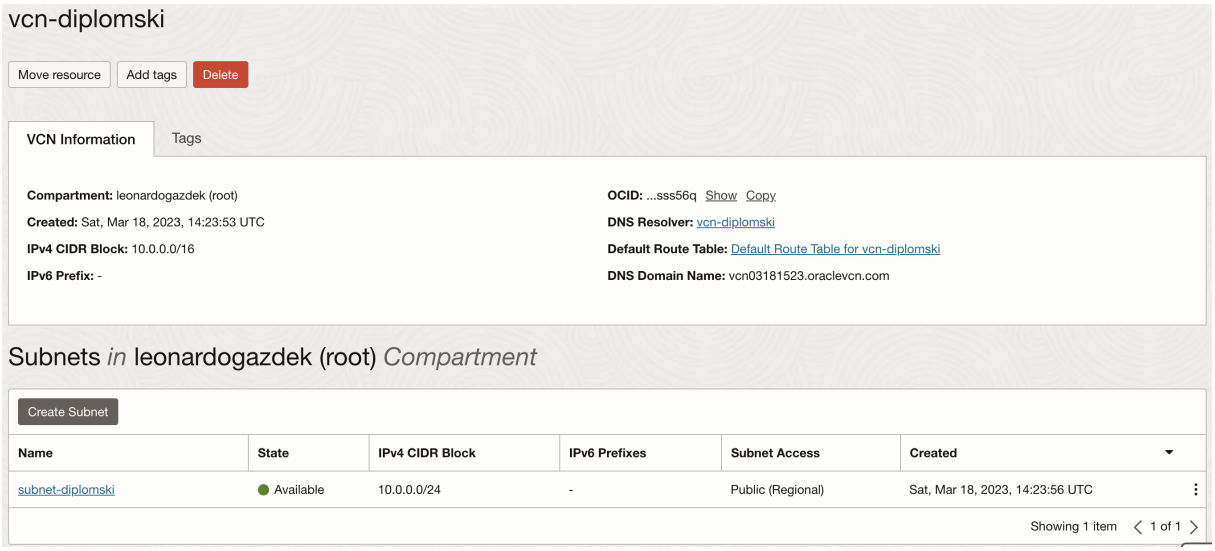
Docker instaliran ispravno.

4.4.3. Umrežavanje Swarm čvorova

Da bi čvorovi unutar Swarm klastera mogli komunicirati moraju biti spojeni na istu mrežu.

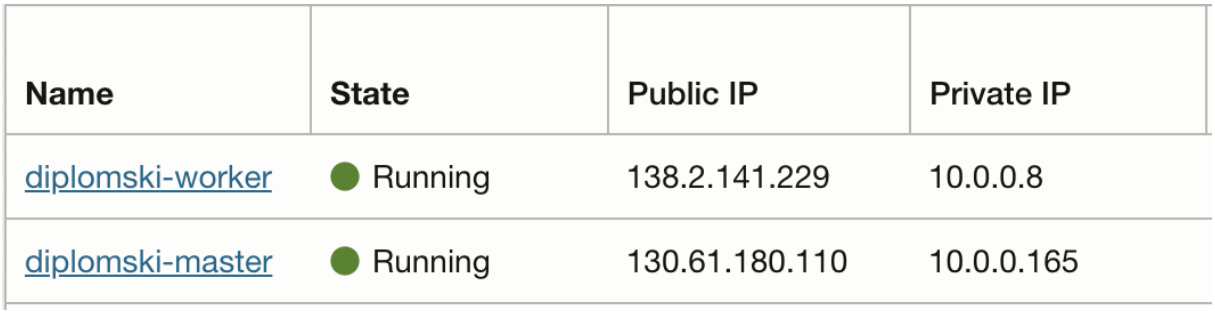
S obzirom na to da se za ovaj primjer koriste Oracle poslužitelji, te je poslužitelje potrebno spojiti na istu virtualnu mrežu u oblaku (virtual cloud network - VCN).

Za potrebe ovog rada kreirana je mreža pod imenom vcn-diplomski. Raspon IP adresa za tu mrežu je 10.0.0.0/16, a unutar te mreže postoji i podmreža (subnet) s rasponom adresa 10.0.0.0/24, a ta podmreža služi kao javna podmreža s internetskim pristupnikom.



Slika 11: Podaci o virtualnoj mreži u oblaku (Izvor: autorski rad)

Daljnjim proučavanjem Oracle Cloud Infrastructure konzole vidljivo je da su oba čvora dobila privatnu IP adresu unutar podmreže 10.0.0.0/24. Čvor koji će se koristiti kao menadžerski dobio je adresu 10.0.0.165, a radni je čvor dobio adresu 10.0.0.8.



Slika 12: Dodijeljene IP adrese (Izvor: autorski rad)

Svaka instanca poslužitelja ima primarnu virtualnu mrežnu karticu (VNIC) koja se kreira prilikom pokretanja instance i ne može se ukloniti. Moguće je dodati sekundarnu virtualnu mrežnu karticu postojećoj instanci [11].

Svaka virtualna mrežna kartica ima privatnu IP adresu iz pridružene podmreže. Moguće je ručno birati IP adresu ili Oracle može odabrati nasumičnu IP adresu unutar podmreže. Privatna se IP adresa ne mijenja tijekom životnog tijeka instance i ne može se ukloniti. Moguće je dodati sekundarne privatne IPv4 adrese ili sekundarne privatne IPv6 adrese [11].

Ako je virtualna mrežna kartica u javnoj podmreži, onda svaka privatna IP adresa može imati pridruženu javnu IPv4 ili IPv6 adresu po potrebi. U slučaju korištenja IPv4 adrese Oracle bira IP adresu, dok u slučaju IPv6 adrese korisnik može ručno birati IP adresu. Postoje dvije vrste javnih IP adresa: kratkotrajne i rezervirane. Kratkotrajne IP adrese postoje samo za vrijeme životnog tijeka privatne IP adrese kojoj su pridružene dok su rezervirane IP adrese trajne i postoje koliko god dugo korisnik to želi [11].

Nakon umrežavanja potrebno je spojiti čvorove u klaster. To se moglo odraditi direktno u Ansible playbook datoteci, ali dobro bi bilo pokazati ručni postupak koji je ionako dosta jednostavan, pogotovo kada se radi o samo dva čvora.

U menadžerskom čvoru prvo bi valjalo provjeriti ispravnost mreže i dodjelu IP adrese:

```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
↪  group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc
↪  pfifo_fast state UP group default qlen 1000
    link/ether 00:00:17:01:00:cb brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.165/24 metric 100 brd 10.0.0.255 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::200:17ff:fe01:cb/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
↪  noqueue state DOWN group default
    link/ether 02:42:fa:92:4c:52 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
```

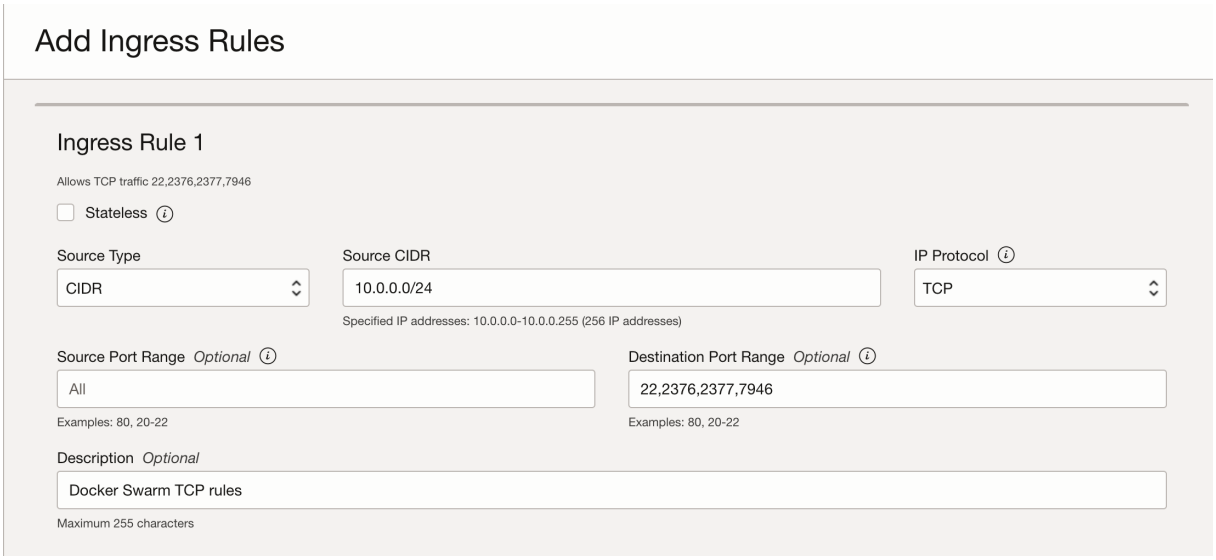
U ispisu ove naredbe vide se sva mrežna sučelja u menadžerskom čvoru. Onaj čvor koji je bitan je enp0s3 i odnosi se na virtualnu mrežnu karticu koja je spomenuta ranije. Vidi se da je ispravno dodijeljena IP adresa 10.0.0.165.

Prije nego što se Docker Swarm mreža može ostvariti potrebno je i otvoriti priključke koje koristi Docker Swarm unutar sigurnosne liste virtualne privatne mreže. Priključci koje koristi Docker Swarm su:

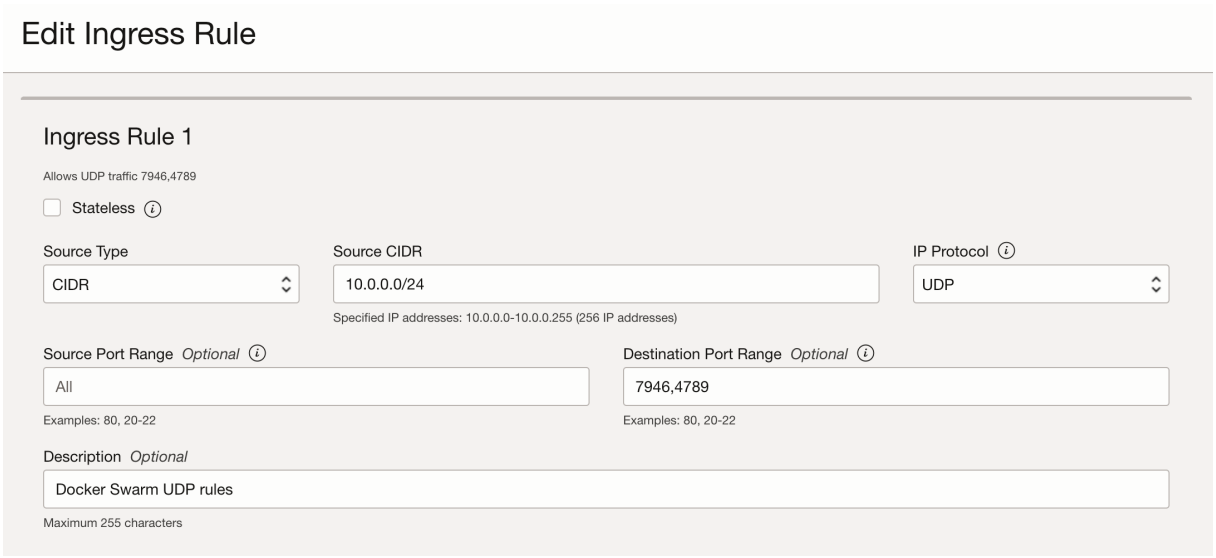
- priključak 2376 TCP za sigurnu Docker klijentsku komunikaciju
- priključak 2377 TCP za komunikaciju menadžerskih čvorova
- priključci 7946 TCP/UDP za otkrivanje mreže
- priključak 4789 UDP za preklapanje mrežnog prometa

Ovi se priključci otvaraju u postavkama virtualne mreže pod opcijom sigurnosnih lista (security lists). Potrebno je kreirati nova ulazna pravila na već postojećoj sigurnosnoj listi.

Ulazna pravila konfiguriraju se zasebno za TCP i UDP priključke.



Slika 13: Otvaranje TCP priključaka (Izvor: autorski rad)



Slika 14: Otvaranje UDP priključaka (Izvor: autorski rad)

Sada je potrebno inicijalizirati Docker Swarm klaster na toj IP adresi:

```
$ sudo docker swarm init --advertise-addr 10.0.0.165
Swarm initialized: current node (p6geprfysmwq1w6hhwz7pnz18) is now a
 ↪  manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-1zqjm4r93m14s8jc9bvs9qj196uzp⌋
     ↪  tba6o8gj60tor8yppojxq-c6nzjew9g7jaas5snh5x1cvuh
     ↪  10.0.0.165:2377

To add a manager to this swarm, run 'docker swarm join-token
 ↪  manager' and follow the instructions.
```

U ispisu naredbe prikazan je i žeton (token) koji će se koristiti na radnim čvorovima. Docker Swarm koristi žetone kao autentifikacijski sigurnosni mehanizam.

Sada je potrebno spojiti se na radni čvor i pozvati naredbu dobivenu u ispisu prošle naredbe.

```
$ sudo docker swarm join --token SWMTKN-1-1zqjm4r93m14s8jc9bvs9qj196⌋
 ↪  uzptba6o8gj60tor8yppojxq-c6nzjew9g7jaas5snh5x1cvuh
 ↪  10.0.0.165:2377
This node joined a swarm as a worker.
```

Ispis naredbe *docker swarm join* javlja da se radni čvor uspješno spojio na Docker Swarm klaster. To je moguće provjeriti i na menadžerskom čvoru:

```
$ sudo docker node list
ID                            HOSTNAME          MANAGER STATUS
p6geprfysmwq1w6hhwz7pnz18 *   diplomski-master  Leader
20tbop30p2jikbjrcqv3z703f     diplomski-worker
```

Rad klastera biti će demonstriran na primjeru jednostavne aplikacije koja prilikom inicijalizacije generira nasumični UUID. To znači da će svaka instanca aplikacije imati svoj jedinstven UUID i lako će se moći vidjeti identifikator aplikacije.

Prvo treba na menadžerskom čvoru postaviti Docker registar jer će se on koristiti za distribuciju Docker slika:

```
$ sudo docker service create --name registry --publish
 ↪  published=5000,target=5000 registry:2
```

U nastavku slijedi kod jednostavne Node.JS aplikacije koja prilikom pokretanja odredi nasumični identifikator te ga na zahtjevu pošalje u odgovoru:

```
import express from "express";
import crypto from "crypto";

const app = express();

const randomUUID = crypto.randomUUID();

app.get("/", (req, res) => {
  res.json({ uuid: randomUUID });
});

app.listen(3000, () => {
  console.log("Express app started on port 3000");
});
```

Dockerfile samo izgrađuje i pokreće aplikaciju.

U datoteci docker-compose.yml definiran je atribut replicas koji označava koliko će se replika kontejnera pokrenuti kroz klaster:

```
version: "3.8"
services:
  app:
    image: 127.0.0.1:5000/app
    build:
      context: ./
    deploy:
      replicas: 2
  nginx:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf:ro
```

Nginx konfiguracija je sljedeća:

```
server {
    listen 80;
    location / {
        proxy_pass http://app:3000;
    }
}
```

Nginx šalje sav promet na http://app:3000, a Docker će automatski raspoređivati promet

na servisu app kroz replike kontejnera. Ne treba postavljati nikakvo dodatno raspoređivanje prometa.

Dalje, treba objaviti sliku aplikacije na lokalni registar slika kontejnera tako da ju ostali čvorovi u klasteru mogu preuzeti.

```
$ sudo docker compose push
[+] Pushing 10/10
 ✓ nginx Skipped                                                    0.0s
 ✓ Pushing app: a066652ac2d7 Pushed                                 0.4s
 ✓ Pushing app: bb0c8cc823f5 Pushed                                10.4s
 ✓ Pushing app: afc96619c692 Pushed                                12.0s
 ✓ Pushing app: 66ad280b6bfd Pushed                                 0.3s
 ✓ Pushing app: ea88754363b3 Pushed                                 0.2s
 ✓ Pushing app: fabb6e7cc132 Pushed                                 1.0s
 ✓ Pushing app: 46e75cb9d01b Pushed                                 2.1s
 ✓ Pushing app: 525e899ff770 Pushed                                14.0s
 ✓ Pushing app: b2191e2be29d Pushed                                 7.2s
```

U ispisu naredbe vidi se da je slika nginx preskočena, a to je zato jer je ta slika već javno dostupna na Docker Hub repozitoriju.

Zadnji korak je isporučiti aplikaciju na klaster:

```
$ sudo docker stack deploy --compose-file docker-compose.yml swarm-app
Ignoring unsupported options: build

Creating network swarm-app_default
Creating service swarm-app_app
Creating service swarm-app_nginx
```

Odlaskom na IP adresu menadžerskog čvora u web pregledniku vidljiva je web stranica koja ispisuje jedinstveni identifikator kontejnera. Ako se aplikacija osvježi nekoliko puta jasno se vidi da se izmjenjuju dva identifikatora što znači da se promet raspoređuje između ta dva kontejnera. To je upravo ono što se i trebalo postići s ovim Swarm klasterom.

```
% curl http://130.61.180.110/
{"uuid":"71fe4572-d697-4fc4-83aa-9fd09beeb6f2"}

% curl http://130.61.180.110/
{"uuid":"389d3211-4ac7-42ec-9b0a-aa11a3634418"}
```

5. Kubernetes

Kubernetes je aplikacijski orkestrator. Ranije je već opisano što je točno orkestrator pa će se ovaj dio više fokusirati na stvari koje su specifične Kubernetes orkestratoru.

Najveća prednost Kubernetes platforme nad Docker Swarm-om je ta da Kubernetes ima značajno više mogućnosti i koncepata za iskoristiti. Neke od mogućnosti koje Kubernetes ima, a Swarm nema su [12]:

- automatski oporavak
- automatsko skaliranje
- mogućnost rada s većim i kompleksnijim sustavima i konfiguracijama
- veća i aktivnija zajednica

Razlog zašto se često bira Swarm umjesto Kubernetes platforme je taj što je Swarm manje kompleksan i brži za savladati i postaviti. Ako neka tvrtka nema potrebe za mogućnostima koje nudi Kubernetes, on ih može na kraju koštati više, a neće imati nikakve koristi od toga, a imati će svu dodatnu kompleksnost [12].

Uglavnom je za manje organizacije i manje kompleksne sustave prikladniji Swarm, dok je za veće organizacije i kompleksne sustave uglavnom prikladniji Kubernetes [12].

5.1. Gdje je nastao Kubernetes?

Amazon Web Servisi (AWS) promijenili su svijet kada su počeli pružati moderno računarstvo u oblaku svojim korisnicima. Od kada se to dogodilo, svi su morali nadoknaditi svoje tehnologije jer su zaostajali za Amazonom [1].

Jedna od tvrtki koja je pokušavala nadograditi svoje tehnologije je Google. Google je trebao način da apstrahira vrijednost AWS-a i da olakša potencijalnim klijentima da se prebace na njihovo rješenje u oblaku [1].

Google je, također, imao jako puno iskustva s kontejnerima na velikoj skali. Na primjer, velike Googleove aplikacije poput tražilice i Google Mail-a su se godinama pokretale na ekstremnoj skali koristeći kontejnerske tehnologije, čak davno prije nego što je Docker osmislio svoje jednostavne kontejnere. Da bi orkestrirali svoje kontejnere, zaposlenici Google-a osmislili su neke interne tehnologije koje su se zvale Borg i Omega [1].

Google je mnogo naučio iz tih internih sustava te je odlučio razviti Kubernetes kojeg je donirao CNCF inicijativi kao projekt otvorenog koda [1].

Od svojih početaka pa sve do danas, Kubernetes je postala najpopularnija tehnologija u oblaku na cijelom svijetu. Kao i mnoge moderne tehnologije u oblaku, Kubernetes je pisan u Go programskom jeziku kojeg je također razvio Google [1].

Za Kubernetes se često koristi i naziv K8s. K i soznačavaju prvo i zadnje slovo riječi Kubernetes, dok 8 predstavlja 8 znakova između [1].

5.2. Kubernetes i Docker

Docker i Kubernetes dobro rade zajedno od početka Kubernetesa. Docker ugrađuje aplikacije u slike kontejnera i može ih pokretati. Kubernetes ne može ni jedno ni drugo. Umjesto toga, sjedi na višoj razini i orkestrira radnje vezane uz kontejnere [1].

Ako, na primjer, postoji Kubernetes klaster s 10 čvorova za pokretanje aplikacija prvi je korak da razvojni timovi koriste Docker za pakiranje svojih aplikacija u slike kontejnera. Nakon toga, slike kontejnera šalju se Kubernetesu na pokretanje. Kubernetes vrši orkestraciju na visokoj razini i donosi odluke o tome koji će čvor pokretati koji kontejner, ali sam Kubernetes ne može pokretati i zaustavljati kontejnere. U prošlosti je svaki čvor Kubernetes klastera sadržavao kopiju Dockera koji bi se brinuo o radu s kontejnerima [1].

Izvana je sve izgledalo dobro. Međutim, kad se bolje pogleda, Docker je prenapuhan i pretežak za ono što Kubernetes treba. Kao rezultat, Kubernetes je počeo raditi na tome da se pokretač kontejnera učini modularnim kako bi korisnici mogli odabrati najprikladniji pokretač kontejnera za svoje potrebe. Kubernetes je 2016. godine predstavio sučelje za pokretače kontejnera koje je učinilo sloj pokretača modularnim. Kubernetes je 2020. godine proglasio Docker pokretač kontejnera zastarjelim što znači da će vjerojatno prestati raditi u jednoj od kasnijih inačica Kubernetes platforme [1].

U vrijeme pisanja rada, containerd je zamijenio Docker kao zadani pokretač kontejnera u većini Kubernetes klastera. Containerd je olakšana verzija Docker sučelja koja je optimizirana za rad s Kubernetes platformom. Rezultat toga je da će sve slike kontejnera kreirane Docker-om i dalje raditi u Kubernetesu, pogotovo zato jer obje implementacije prate OCI specifikaciju. Razvojni programeri mogu i dalje na svom računalu nastaviti koristiti Docker za izgradnju slika kontejnera, a to većina programera i radi s obzirom na to da je Docker daleko najpopularniji pokretač kontejnera [1].

5.3. Kontrolna razina

Kubernetes čvor kontrolne razine upravlja kolekcijom servisa sustava od kojih se sadrži kontrolna razina klastera. Često se koristi i naziv *glavni (master) čvor*, ali isto kao što je slučaj kod GitHub-a, taj se naziv polako pokušava izbaciti zbog političke korektnosti [1].

Najjednostavnija postava klastera sastoji se od jednog čvora kontrolne razine. Ovakva postava prikladna je samo za testne okoline. Za produkcijske okoline uglavnom je dobro imati nekoliko čvorova kontrolne razine da bi se postigla visoka raspoloživost sustava. Moguće je pokretati korisničke aplikacije na čvorovima kontrolne razine, ali to se ne preporuča jer bi glavna zadaća takvih čvorova trebala biti upravljanje klasterom, dok bi radni čvorovi trebali pokretati aplikacije [1].

Čvorovi kontrolne razine pokreću kontrolne servise klastera. Ovi su servisi mozak klastera u kojem se odvija sva kontrola i raspoređivanje zadataka u klasteru. Iza kulisa, u ove servise spadaju API poslužitelj, spremište klastera, servis za raspoređivanje i servisi jezgre [1].

API poslužitelj korisnička je strana prema kontrolnoj razini i sve instrukcije i komunikacije prolaze kroz njega. U zadanim postavkama poslužitelj poslužuje REST servis na utičnici 443 (HTTPS) [1].

5.4. Radni čvorovi

Na radnim se čvorovima pokreću korisničke aplikacije. Radni čvorovi nemaju podatke o ostalim čvorovima u klasteru niti mogu s njima direktno komunicirati [1].

Na visokoj razini, radni čvorovi rade sljedeće:

- promatraju API poslužitelj za nove radne zadatke
- izvršavaju radne zadatke
- odgovaraju kontrolnoj razini (preko API poslužitelja)

5.4.1. Glavne komponente radnog čvora

Postoje tri glavne komponente radnog čvora [1], a to su **Kubelet**, **pokretač kontejnera** i **kube-proxy**.

5.4.1.1. Kubelet

Kubelet je glavni Kubernetes agent i pokreće se na svakom radnom čvoru [1].

Kada se čvor pridruži klasteru, proces instalira kubelet koji je onda odgovoran za to da se registrira na klaster. Proces registrira procesor, memoriju i pohranu čvora u šire udruženje klastera [1].

Jedna od glavnih zadaća kubelet-a je da nadgleda API poslužitelj za nove radne zadatke. Izvršava zadatak svaki put kada ga vidi i odgovara natrag kontrolnoj razini [1].

Ako kubelet ne može izvršiti zadatak, tada odgovara natrag kontrolnoj razini i pušta kontrolnoj razini da odluči što treba dalje. Ako kubelet ne može izvršiti zadatak, njegova zadaća nije da pronađe čvor koji to može - to je zadaća upravljačke razine [1].

5.4.1.2. Pokretač kontejnera

Kubelet treba pokretač kontejnera da bi mogao izvršavati zadatke nad kontejnerima [1].

U starim danima Kubernetes je imao podršku za Docker. Nedavno se prebacio na modularni model sučelja pokretača kontejnera. Na visokoj razini, sučelje pokretača kontejnera

sakriva interni rad Kubernetes platforme i pruža jednostavno i dokumentirano sučelje za pokretače kontejnera treće strane [1].

Prilikom postavljanja klastera kasnije u radu će se koristiti pokretač kontejnera cri-o.

5.4.1.3. Kube-proxy

Zadnja komponenta radnih čvorova je kube-proxy. On se izvršava na svakom čvoru i odgovoran je za lokalnu mrežu u klasteru. Brine se da svaki čvor ima svoju jedinstvenu IP adresu i implementira lokalni iptables vatrozid ili IPVS pravila za upravljanje usmjeravanjem. Također se brine za raspoređivanje prometa u mreži grupe [1].

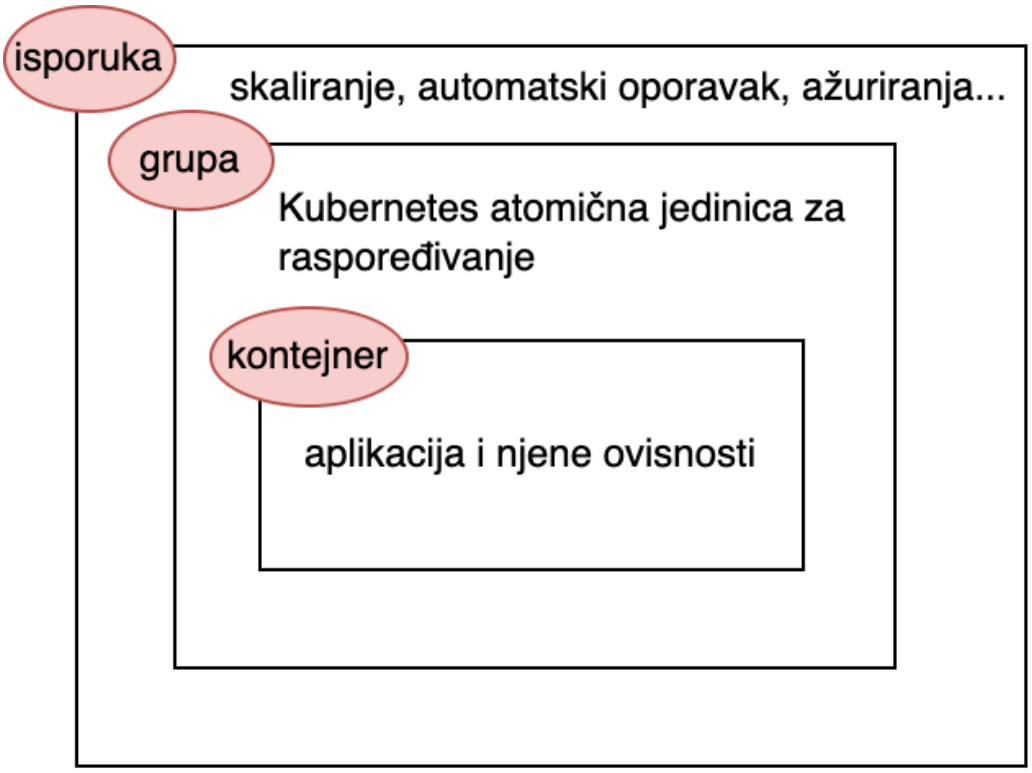
5.5. Pakiranje aplikacija za Kubernetes

Aplikacija treba zadovoljavati nekoliko uvjeta da bi se mogla pokretati na Kubernetes klasteru. Aplikacija mora biti [1]:

- pakirana u kontejner
- omotana u grupu
- isporučena deklarativnom datotekom manifestiranja

Moguće je pokretati i same grupe aplikacija nakon pakiranja aplikacije u kontejner i omatanja aplikacije u grupu, ali preferirani model je da se sve grupe stavljaju u isporuku (deployment). Isporuke omogućavaju skalabilnost, automatski oporavak i jednostavna ažuriranja za aplikacije. Isporuke se deklariraju u YAML datotekama manifestiranja, a u tim se datoteka deklarira koliko mora biti replika, kako se izvode ažuriranja, itd [1].

Nakon isporuke, grupama kontejnera se pristupa pomoću servisa koje također treba definirati i vezati uz isporuke. Postoji nekoliko vrsta servisa, ali najčešće korišten tip je ClusterIP koji omogućava komunikaciju servisa unutar mreže klastera.



Slika 15: Odnos kontejnera, grupe i isporuke (Izvor: Poulton i Joglekar, 2022)

5.6. Deklarativni model i željeno stanje

Koncept deklarativnog modela i željenog stanja su u srcu Kubernetes platforme [1].

U Kubernetes platformi, deklarativni model radi na sljedeći način [1]:

- Potrebno je deklarirati željeno stanje aplikacijskih mikroservisa u datoteci manifestiranja.
- Datoteku treba poslati API poslužitelju.
- Kubernetes sprema datoteku u spremište klastera kao aplikacijino željeno stanje.
- Kubernetes primjenjuje željeno stanje na klaster.
- Kontroler se brine da je promatrano stanje isto kao i željeno stanje.

Kao i u Ansible poglavlju, važno je razumjeti da se ovo razlikuje od imperativnog modela u kojem se izvršavaju naredbe jedna za drugom i ne uspoređuje se nikakvo stanje. Deklarativni model puno je jednostavniji i nudi svojstvo idempotentnosti što znači da se ista datoteka manifestiranja može primijeniti više puta bez da se kreira više resursa. Imperativne skripte također često nisu kompatibilne s različitim platformama, dok deklarativne sintakse uglavnom jesu [1].

Deklarativni model ostvaruje automatski oporavak, skaliranje i samodokumentaciju uz još mnogo ostalih benefita na način da se u klasteru opisuje kako stvari *trebaju izgledati*. Odgovarajući kontroler primjećuje razlike u stanju ako stanje počinje izgledati drugačije nego što je to željeno i poduzima odgovarajuće korake da se stanje izjednači [1].

Deklarativni model već je demonstriran na temelju docker compose yaml datoteka, a sličan takav model koristi se i u Kubernetes datotekama manifestiranja koje će biti prikazane prilikom isporuka aplikacija na klaster.

5.7. Grupe

U svijeti virtualizacije najmanja je jedinica raspoređivanja virtualni stroj. U Docker svijetu, najmanja je jedinica kontejner. U Kubernetes svijetu, najmanja jedinica raspoređivanja naziva se grupa (pod) [1].

Istina je da Kubernetes pokreće kontejnerizirane aplikacije. Unatoč tome, Kubernetes zahtjeva da se svi kontejneri pokreću unutar grupe [1].

5.7.1. Kontejneri i grupe

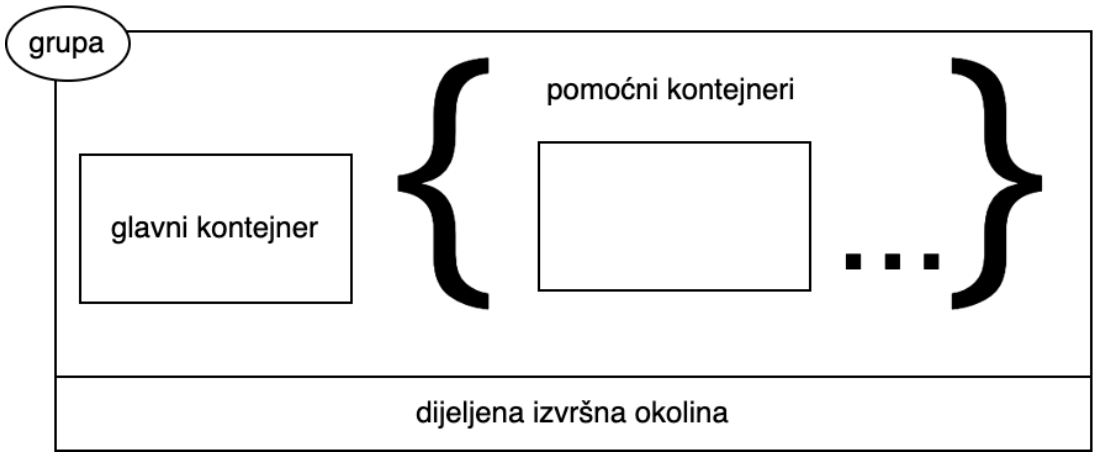
Prva stvar koju treba razumjeti je da riječ grupa (pod) dolazi iz termina *grupa kitova (pod of whales)*. Dockerov logotip prikazuje kita i zato se grupe zovu tako kako se zovu [1].

Najjednostavniji je model takav da se u svakoj grupi pokreće jedan kontejner. Iz tog se razloga često termini *grupa* i *kontejner* koriste sinonimno [1].

Postoje i napredniji scenariji gdje se u jednoj grupi pokreće više kontejnera. Primjeri grupa s više kontejnera su [1]:

- servisna mreža
- web kontejneri integrirani uz pomoćni kontejner koji dohvaća podatke
- općenito čvrsto povezani kontejneri

Na kraju krajeva, Kubernetes grupa konstrukt je za pokretanje jednog ili više kontejnera [1].



Slika 16: Kubernetes grupa (Izvor: Poulton i Joglekar, 2022)

5.7.2. Anatomija grupe

Na najvišoj razini, grupe su ograđena okruženja za pokretanje kontejnera. Same grupe ne pokreću aplikacije, to rade kontejneri, a grupa je samo tzv. *pješčana kutija* u kojoj se pokreće jedan ili više kontejnera. Grupe ograđuju područje operacijskog sustava, izgrađuju mrežu kreiraju imenske prostore u ljusci i pokreću jedan ili više kontejnera [1].

Ako se više kontejnera pokreće unutar grupe, oni svi dijele istu okolinu. Ovo uključuje mrežu, volumene, imenske prostore, dijeljenu memoriju i još neke druge stvari. To znači da svi kontejneri unutar grupe dijele i istu IP adresu, a to je IP adresa grupe. Ako dva kontejnera žele komunicirati unutar grupe, tada mogu koristiti *localhost* loopback adresu. Kada je cilj imati čvrsto spojene kontejnere koji dijele memoriju, pohranu, mrežu ili neki drugi resurs, tada je dobra ideja staviti te kontejnere unutar iste grupe. Ne preporuča se postaviti više kontejnera u istu grupu ako ne dijele nikakve resurse [1].

5.7.3. Grupe kao jedinice skaliranja

Grupe su, također, najmanja jedinica skaliranja unutar Kubernetes platforme. Prilikom skaliranja dodaju i brišu se grupe, a ne sami kontejneri. Grupe s više kontejnera koriste se samo u situacijama kada se i pomoćni kontejneri trebaju skalirati s glavnim kontejnerima u grupi [1].

Grupe su atomične jedinice što znači da se cijela grupa neće pokrenuti ako se jedan kontejner unutar grupe ne pokrene. Kontejneri unutar jedne grupe uvijek se pokreću na istom čvoru. Nemoguće je imati situaciju u kojoj se neki od kontejnera grupe pokreće na različitom čvoru [1].

5.8. Isporuke

Većinu vremena, grupe će biti isporučene indirektno pomoću kontrolera više razine. Primjeri takvih kontrolera su isporuke, DaemonSet-ovi i StatefulSet-ovi [1].

Kao primjer, isporuka je Kubernetes objekt više razine koji omata grupe i dodaje karakteristike kao što su automatski oporavak, skaliranje, ažuriranje bez prekida rada, itd [1].

Tu se radi o kontrolerima koji se pokreću kao petlje koje konstantno nadgledaju klaster i brinu se da je promatrano stanje jednako željenom stanju [1].

5.8.1. Servisni objekti i stabilno umrežavanje

Grupe nisu uvijek pouzdane i podložne su greškama u radu. Ako se njima upravlja pomoću kontrolera više razine, one će se zamijeniti kada naiđu na greške u radu. Problem je što zamjene dolaze s potpuno različitim IP adresama. Ovo se događa i kod operacija skaliranja i ažuriranja [1].

Kao primjer može se koristiti mikroservisna aplikacija s mnogo grupa koja obrađuje

videozapise. Kako će to raditi ako drugi dijelovi aplikacije trebaju koristiti servis za obradu, ali ne mogu znati gdje će taj servis biti u mreži? Upravo ovdje u igru ulaze servisi. Servisi pružaju pouzdano umrežavanje za skup grupa [1].

Servisi su punopravni objekti u Kubernetes API-ju, baš kao grupe i isporuke. Imaju korisničku stranu koja se sastoji od stabilnog DNS imena, IP adrese i priključka. S druge strane, oni raspoređuju promet kroz dinamički skup grupa. Kako grupe dolaze i odlaze, servisi to primjećuju, automatski se ažuriraju i nastavljaju pružati stabilnu krajnju točku umrežavanja [1].

Isto vrijedi ako se broj grupa skalira gore ili dolje. Nove se grupe neprimjetno dodaju u servis i obrađuju promet. Ugašene grupe se neprimjetno miču iz grupe i njima se promet više ne šalje [1].

Servisi rade na TCP i UDP slojevima što znači da nemaju aplikacijsku inteligenciju. Oni ne mogu pružati aplikacijsko usmjeravanje putanja. Za to služe ulazni kontroleri koji razumiju HTTP protokol i pružaju usmjeravanje na temelju putanja [1].

6. Isporuka aplikacija na Kubernetes klaster

Za isporuku aplikacije na Kubernetes klaster prvo je potrebno imati aplikaciju. U sljedećim će se poglavljima obraditi izrada jednostavne aplikacije koja prati stavke koje korisnik treba izvršiti (to-do lista). Svaka od tih aplikacija će se trebati zapakirati u kontejner i prenijeti na Docker Hub da bi ju čvorovi klastera mogli preuzeti.

Aplikacija će se sastojati od korisničke i poslužiteljske strane. Korisnička će strana biti napravljena pomoću razvojnog okvira Next.js i koristit će novi koncept poslužiteljskih komponenti. Next.js razvojni je okvir za izradu React aplikacija. On projektu daje određenu strukturu, mogućnosti usmjeravanja, prikaz aplikacije na poslužiteljskoj strani (server-side rendering)... Jedna od glavnih prednosti Next.js razvojnog okvira upravo je mogućnost prikazivanja aplikacija na poslužiteljskoj strani. Naime, u današnje vrijeme, većina se korisničkih web aplikacija piše u JavaScriptu, a to znači da se HTML sadržaj web stranice ne može izračunati prije nego što se u pregledniku izvrši JavaScript kod. Posluživanje na poslužiteljskoj strani znači da će razvojni okvir unaprijed, prilikom izgradnje aplikacije, izračunati HTML sadržaj aplikacije na temelju JavaScript koda. Na taj se način aplikacija brže učitava i razni web pretraživači lakše mogu pregledati sadržaj web stranice.

Poslužiteljska će strana biti napravljena u Javi te će koristiti Spring kao razvojni okvir. Za bazu podataka koristiti će se Amazon DynamoDB zbog svoje jednostavnosti. Još jedna prednost te baze je što je potpuno upravljana i ne mora se pokretati na Kubernetes klasteru, već Amazon potpuno upravlja isporukom baze u oblaku.

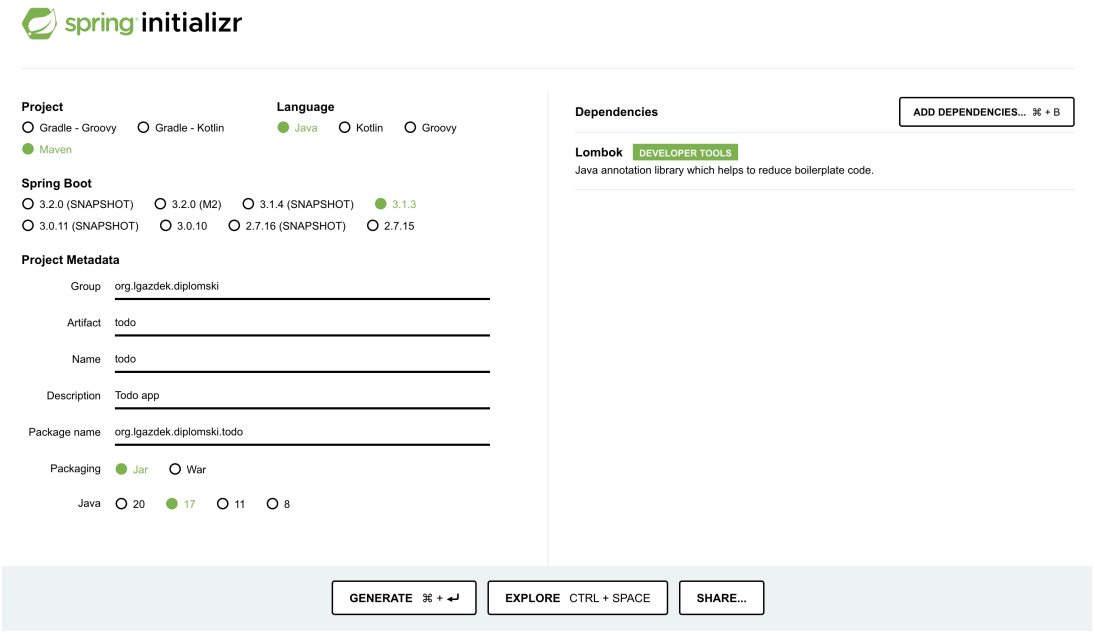
6.1. Poslužiteljska Spring aplikacija

Poslužiteljska je aplikacija, kao što je već ranije navedeno, izrađena koristeći programski jezik Java i Spring razvojni okvir. Za upravljanje izgradnjom i paketima koristi se Apache Maven, a za samu inicijalizaciju aplikacije koristi se Spring Boot.

6.1.1. Kreiranje Spring Boot projekta

Spring Boot olakšava kreiranje samostalne Spring aplikacije koja se može pokrenuti bez ikakvih dodatnih koraka. Spring Boot može čak i ugraditi servlet u aplikaciju tako da se razvojni programer ne mora zamarati s njegovim postavljanjem. Dovoljno je pokrenuti Spring aplikaciju kao standardnu jar izvršnu datoteku i automatski će se pokrenuti i Apache Tomcat poslužitelj skupa s aplikacijom.

Za kreiranje aplikacije koristeći Spring Boot potrebno je otići na https://start.spring.io. Za ovaj projekt, tamo je potrebno odabrati Maven za upravljanje projektom, Javu kao jezik i Spring Boot verziju 3.1.3. Nakon toga treba popuniti osnovne podatke o aplikaciji, odabrati Java verziju, odabrati pakete (u slučaju ove aplikacije, samo Lombok) i pritisnuti gumb za generiranje.



Slika 17: Spring Boot inicijalizacija (Izvor: autorski rad)

6.1.2. Postavljanje Swagger korisničkog sučelja

Swagger korisničko sučelje dozvoljava bilo kome, neovisno o tome radi li se o razvojnim programerima ili korisnicima, korištenje i interakciju s krajnjim točkama aplikacijskog programskog sučelja (API) [13]. Swagger korisničko sučelje generira se na temelju OpenAPI specifikacije [13] koja se djelomično automatski generira iz Java klasa, a djelomično se definira anotacijama u Java kodu.

Za postavljanje Swagger sučelja u Spring projektu dovoljno je dodati samo jednu Maven ovisnost:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.1.0</version>
</dependency>
```

Također ne bi bilo loše postaviti kontekstnu putanju uz još neke druge postavke za cijelu Spring aplikaciju. To se može napraviti u datoteci application.properties:

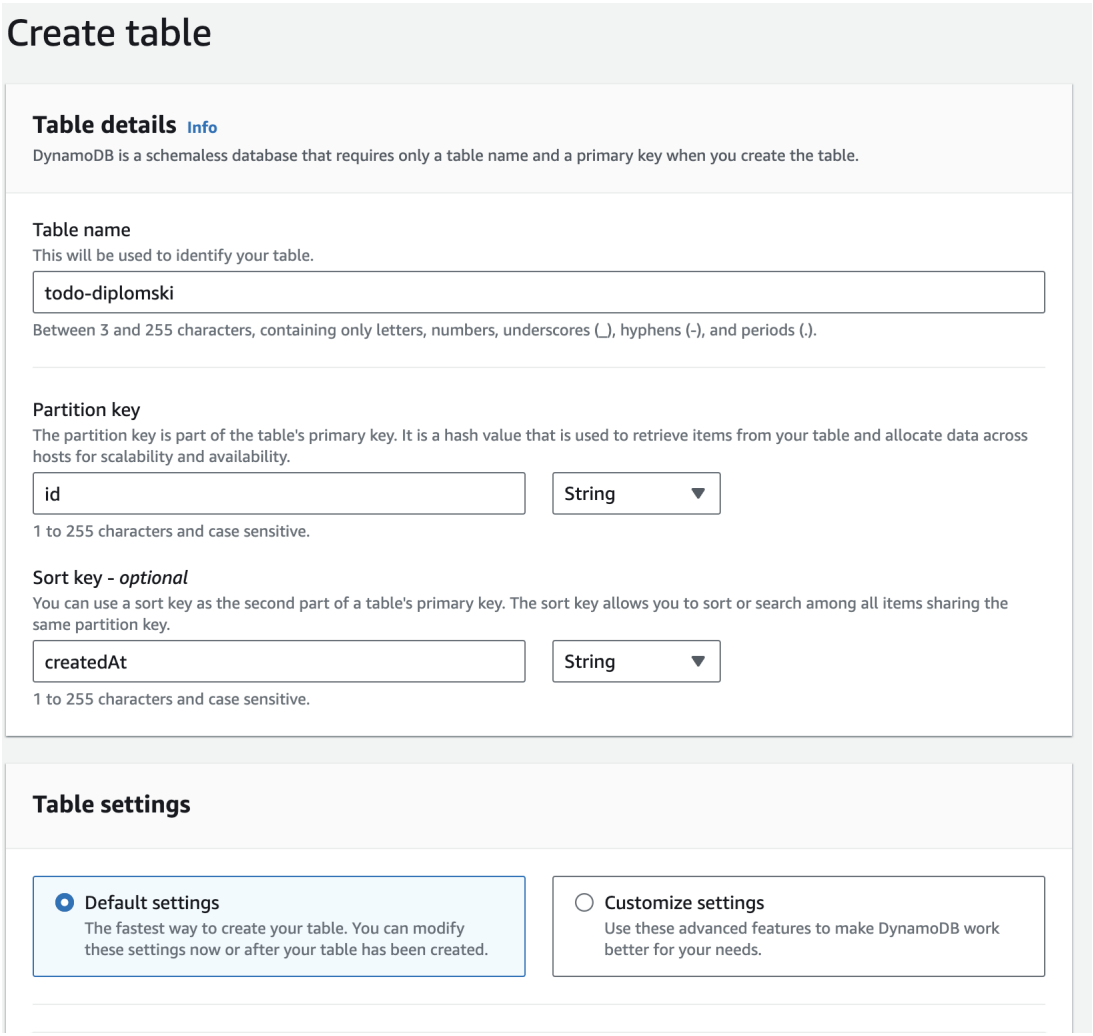
```
server.address=0.0.0.0
server.port=8888
server.servlet.context-path=/todo-api
server.forward-headers-strategy=framework
```

Swagger korisničko sučelje sada je dostupno na ruti /todo-api/swagger-ui.html. Ono će uvelike olakšati razvoj korisničke aplikacije jer se iz OpenAPI specifikacije mogu generirati tipovi

i klijent za TypeScript što znači da programer korisničke strane aplikacije ne treba napamet znati koja se krajnja točka nalazi na kojoj putanji i koje parametre prima.

6.1.3. Konfiguranje Amazon DynamoDB baze podataka

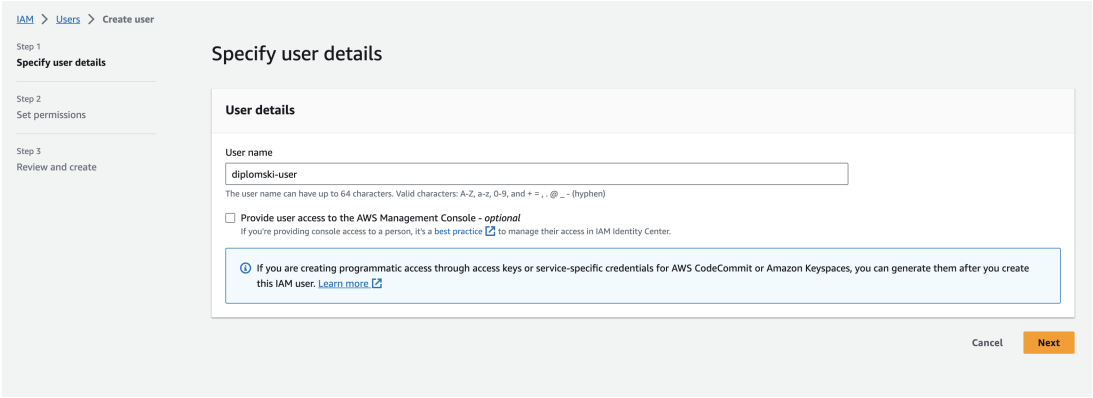
Kreiranje tablice u DynamoDB bazi podataka jako je jednostavno. Potrebno je samo otići u AWS konzolu, kliknuti na gumb za kreiranje tablice i odrediti ključ particije i ključ za sortiranje. DynamoDB baza je podataka koja radi na principu ključ-vrijednost (key-value) te su ključevi jedini podaci koji moraju biti dostupni za svaki redak u bazi podataka. Svi ostali podaci mogu biti varijabilni i nema definiranog oblika.



Slika 18: Kreiranje DynamoDB tablice (Izvor: autorski rad)

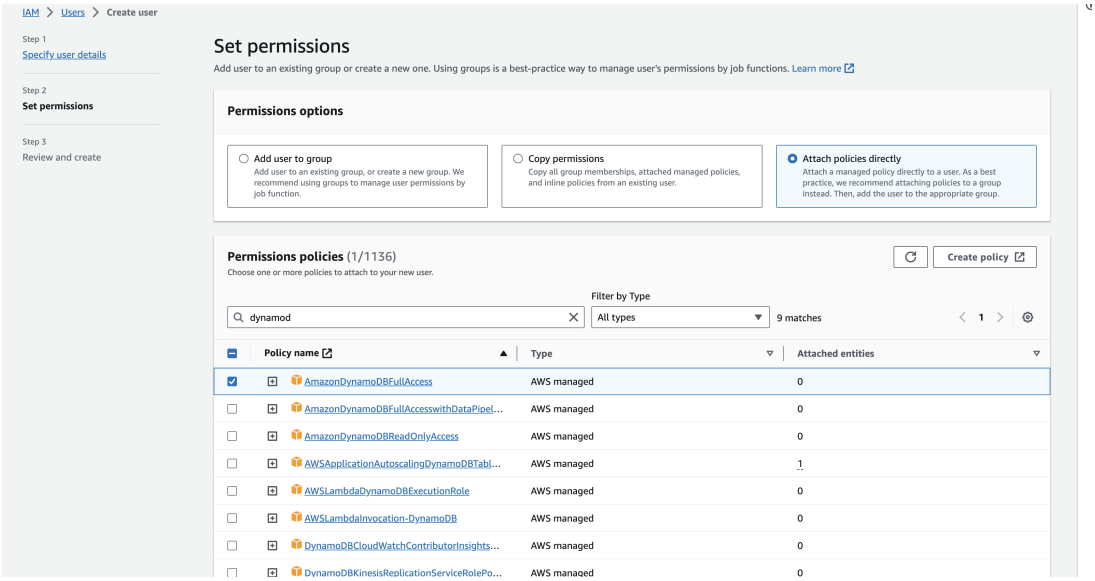
Nakon kreiranja tablice treba joj omogućiti pristup izvana. DynamoDB nije kao klasična baza podataka koja sluša na svom posebnom priključku, već radi koristeći HTTP. Potrebno je kreirati korisnika u Identity and Access Management (IAM) konzoli, pridružiti tom korisniku prava pristupa na DynamoDB tablicu i pridružiti mu pristupni ključ za autentifikaciju. Korisnik kreiran u IAM konzoli naziva se IAM korisnik. Ljudski korisnici ili AWS resursi predstavljaju se preko IAM korisnika da bi mogli vršiti radnje nad resursima i servisima unutar AWS računa.

IAM korisnik sastoji se od imena i vjerodajnica preko kojih se prijavljuje [14]. Za DynamoDB korisnika koristiti će se pristupni ključ kao vjerodajnica. Za autorizaciju potrebno je kreirati IAM korisnika u AWS konzoli.



Slika 19: Kreiranje IAM korisnika (Izvor: autorski rad)

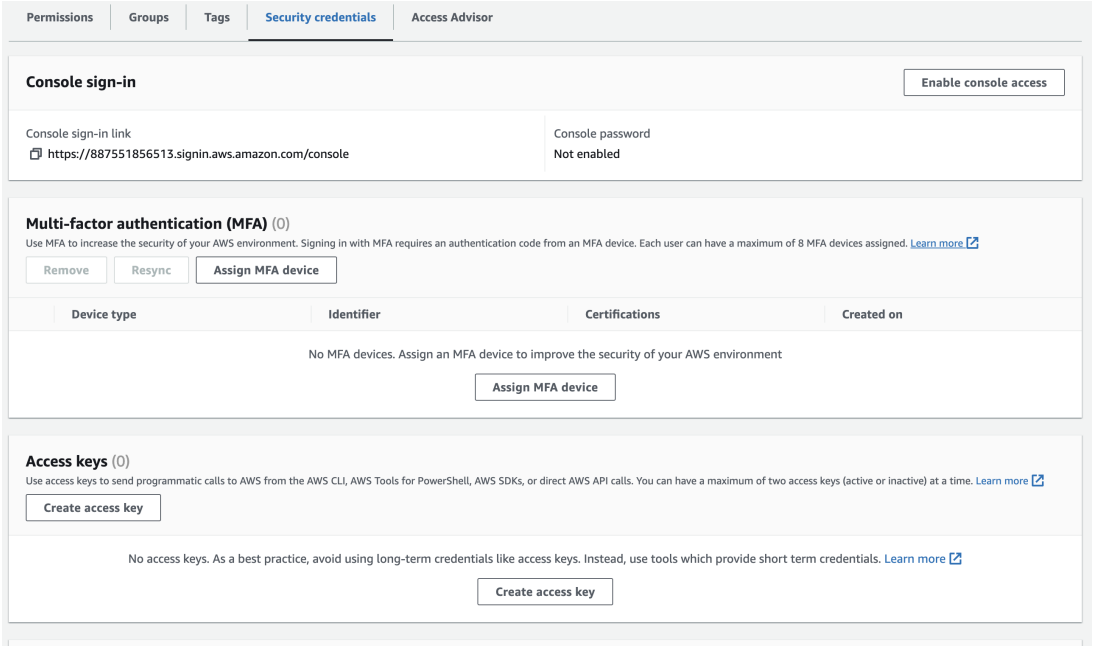
U sljedećem koraku kreiranja korisnika njemu treba pridružiti prava pristupa na DynamoDB resurse unutar AWS računa. Inače je dobro definirati finija prava pristupa, ali za sada će poslužiti pravilo *AmazonDynamoDBFullAccess*.



Slika 20: Dozvola pristupa DynamoDB resursima (Izvor: autorski rad)

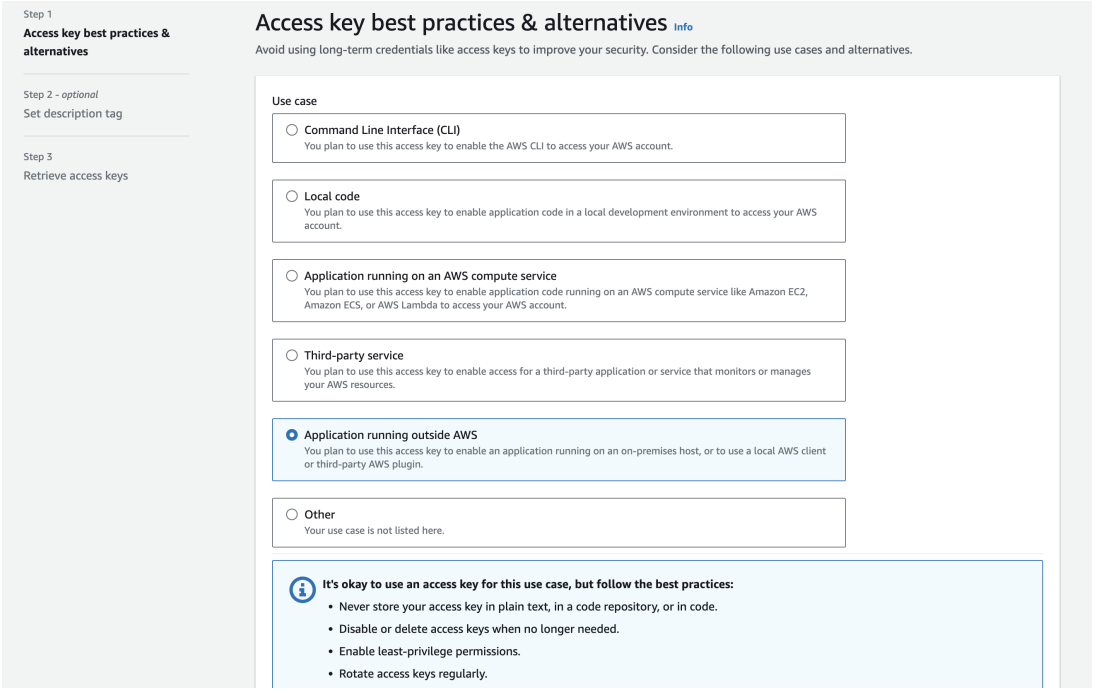
Korisniku sada treba pridružiti pristupne ključeve koji će se koristiti za autorizaciju nad DynamoDB HTTP pristupnim točkama. Ove ključeve treba držati u tajnosti jer ih napadači mogu lako iskoristiti za kreaciju resursa na tuđem AWS računu.

U postavkama korisnika pod karticom *Security credentials* treba odabrati opciju *Create access key*.



Slika 21: Sigurnosne vjerodajnice (Izvor: autorski rad)

U prvom koraku kreacije ključa treba odabrati koja će biti svrha tog ključa. Odabir bilo koje opcije unutar ovog koraka nema nikakvog utjecaja na konačno ponašanje ključa, već samo služi da bi AWS korisniku mogao ispisati neke dodatne informacije. U slici ispod te su informacije vidljive na dnu i govore korisniku kako je u redu koristiti ključ za svrhu pristupa aplikacija izvan AWS-a.



Slika 22: Biranje svrhe ključa (Izvor: autorski rad)

Nakon odabira svrhe ključa kreirat će se pristupni ključ i njegov tajni podatak. Tajni se

podatak ne može naknadno pogledati tako da ga je potrebno spremiti na sigurno mjesto u ovom trenutnu. Ovi će se ključevi učitavati u Spring aplikaciju pomoću varijabli okoline (environment variables).

Prilikom razvoja praktično je spremiti varijable okoline u datoteku (uglavnom naziva .env) koja će se kasnije učitavati u postavkama Spring profila.

Datoteka .env sadrži sve tajne i promjenjive podatke koji se ne žele "tvrdo kodirati".

```
ACCESS_KEY=(skriveno)
SECRET_ACCESS_KEY=(skriveno)
REGION=eu-central-1
DYNAMODB_TODO_TABLE_NAME=todo-diplomski
```

Dalje, u datoteci zadanog Spring profila *application.yml* treba učitati .env datoteku ako ona postoji.

```
spring:
  config:
    import: optional:file:.env[.properties]

aws:
  accessKey: ${ACCESS_KEY}
  secretAccessKey: ${SECRET_ACCESS_KEY}
  region: ${REGION}
  dynamoDb:
    todoTableName: ${DYNAMODB_TODO_TABLE_NAME}
```

Atribut import unutar konteksta spring->config zadaje da se učitavaju varijable iz .env datoteke ako ona postoji. Ako ne postoji, učitati će se varijable iz okoline.

Ovakve postavke sada je moguće čitati direktno u kodu. Spring nudi moćan uzorak injekcija ovisnosti i taj će se uzorak koristiti za učitavanje DynamoDB klijenta u aplikaciji.

Potrebno je postaviti Spring Bean za DynamoDB klijent koji će se kasnije injektirati u komponente aplikacije. U ovom će se Beanu učitati postavke iz datoteke zadanog profila.

```
@Configuration
public class DynamoDbConfiguration {
    @Value("${aws.accessKey}")
    private String accessKey;
    @Value("${aws.secretAccessKey}")
    private String secretAccessKey;
    @Value("${aws.region}")
    private String region;
```

```
    @Bean
    public DynamoDbClient dynamoDbClient() {
        StaticCredentialsProvider staticCredentialsProvider = StaticCredentialsP
            AwsBasicCredentials.create(accessKey, secretAccessKey));
        return DynamoDbClient
            .builder()
            .credentialsProvider(staticCredentialsProvider)
            .region(Region.of(region))
            .httpClientBuilder(ApacheHttpClient.builder())
            .build();
    }
}
```

Sada je moguće injektirati DynamoDbClient u bilo koju komponentu aplikacije pomoću konstruktora.

6.1.4. Operacije kreiranja, čitanje, ažuriranja i brisanja nad bazom podataka

Nakon postavljenog DynamoDB klijenta, aplikacija ga je spremna koristiti.

Potrebno je napraviti REST kontroler za resurse aplikacije i u njega injektirati DynamoDbClient. Također treba učitati ime tablice nad kojom se vrše operacije.

```
@RestController
@RequiredArgsConstructor
public class TodoController {
    private final DynamoDbClient dynamoDbClient;

    @Value("${aws.dynamoDb.todoTableName}")
    private String todoTableName;
}
```

@RequiredArgsConstructor lombok je anotacija koja će prevesti gornji kod u nešto nalik sljedećem:

```
@RestController
public class TodoController {
    private final DynamoDbClient dynamoDbClient;

    @Value("${aws.dynamoDb.todoTableName}")
    private String todoTableName;
```

```
    public TodoController(DynamoDbClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }
}
```

Na taj se način postiže injekcija ovisnosti pomoću konstruktora.

Sada se može kreirati operacija za dohvaćanje resursa na ruti GET /api/todo.

```
@Operation(summary = "Gets all todo items")
@GetMapping("/api/todo")
public List<TodoResponseDto> getTodos() {
    ScanRequest scanRequest = ScanRequest
        .builder()
        .tableName(todoTableName)
        .consistentRead(true)
        .build();
    ScanResponse data = dynamoDbClient.scan(scanRequest);
    return
     ↪  todoResponseMapper.toDtoCollection(data.items()).stream().sorted((a,
     ↪  b) -> {
        // kod za sortiranje...
    }).toList();
}
```

Mapiranja transportnih objekata koristeći mapstruct neće biti objašnjena u ovom radu. Cilj je demonstrirati što radi aplikacija i ne ići u previše detalja.

Operacija getTodos vrši DynamoDB scan operaciju nad bazom podataka i vraća te sortirane podatke u odgovoru zahtjeva.

Operacija za kreiranje resursa koristi DynamoDB putItem operaciju nad bazom podataka i nalazi se na ruti POST /api/todo.

```
@Operation(summary = "Create a todo item")
@PostMapping("/api/todo")
public TodoResponseDto createTodo(@RequestBody @Valid TodoRequestDto
 ↪  reqDto) {
    Map<String, AttributeValue> insertItem =
     ↪  todoRequestMapper.toModel(reqDto);
    PutItemRequest putItemRequest = PutItemRequest
        .builder()
        .tableName(todoTableName)
        .item(insertItem)
        .build();
```

```
    dynamoDbClient.putItem(putItemRequest);
    return todoResponseMapper.toDto(insertItem);
}
```

DynamoDB operacija putItem služi za kreiranje redaka u DynamoDB bazi podataka. Operacija createTodo vrši DynamoDB putItem operaciju nad bazom i vraća kreirani objekt u odgovoru zahtjeva.

Operacija za brisanje resursa koristi DynamoDB deleteItem operaciju nad bazom podataka i nalazi se na ruti DELETE /api/todo/id.

```
@Operation(summary = "Delete a todo item")
@DeleteMapping("/api/todo/{id}")
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void deleteTodo(@PathVariable String id) {
    DeleteItemRequest deleteItemRequest = DeleteItemRequest
        .builder()
        .tableName(todoTableName)
        .key(Map.of("id", AttributeValue.fromS(id)))
        .build();
    dynamoDbClient.deleteItem(deleteItemRequest);
}
```

DynamoDB operacija deleteItem služi za brisanje redaka u DynamoDB bazi podataka. Operacija deleteTodo vrši DynamoDB deleteItem operaciju nad bazom i ne vraća ništa osim HTTP koda 204.

Operacija za ažuriranje resursa koristi DynamoDB updateItem operaciju nad bazom i nalazi se na ruti PATCH /api/todo/id.

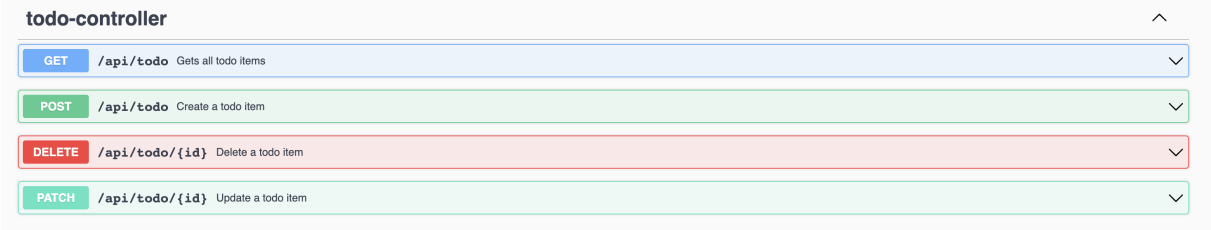
```
@Operation(summary = "Update a todo item")
@PatchMapping("/api/todo/{id}")
public void updateTodo(@PathVariable String id, @RequestBody @Valid
↪   TodoRequestDto reqDto) {
    Map<String, AttributeValueUpdate> updateItem =
    ↪   todoRequestMapper.toUpdateModel(reqDto);
    UpdateItemRequest updateItemRequest = UpdateItemRequest
        .builder()
        .tableName(todoTableName)
        .key(Map.of("id", AttributeValue.fromS(id)))
        .attributeUpdates(updateItem)
        .build();

    dynamoDbClient.updateItem(updateItemRequest);
}
```

DynamoDB operacija updateItem služi za ažuriranje redaka u DynamoDB bazi podataka. Operacija updateTodo vrši DynamoDB updateItem operaciju nad bazom i ne vraća ništa.

Treba primijetiti da svaka od API operacija ima anotaciju @Operation koja definira opis rute za Swagger i OpenAPI definiciju.

Pokretanjem Swagger korisničkog sučelja mogu se isprobati operacije novokreiranog kontrolera.



Slika 23: Pregled kontrolera u Swagger korisničkom sučelju (Izvor: autorski rad)

6.2. Korisnička aplikacija

Korisnička je aplikacija, kao što je već ranije navedeno, izrađena koristeći programski jezik TypeScript i Next.js razvojni okvir. Za inicijalizaciju Next.js projekta koristi se create-next-app.

6.2.1. Kreiranje aplikacije i API klijenta

Za pokretanje skripte create-next-app koristi se naredba u nastavku.

```
% npx create-next-app
```

Prilikom kreiranja aplikacije potrebno je ispuniti neke podatke, a to su, na primjer, ime projekta, koristi li se JavaScript ili TypeScript, koristi li se Tailwind CSS, itd.

Ranije u radu spomenuto je Swagger korisničko sučelje koje se usko veže uz OpenAPI specifikaciju. Za TypeScript projekte postoji paket swagger-typescript-api. Taj paket može generirati klijent za pristup resursima unutar kontrolera dostupnih u poslužiteljskoj aplikaciji. Taj se klijent generira na temelju OpenAPI specifikacije.

Postavljanje je jako jednostavno:

```
% npm install swagger-typescript-api -D
```

Ova naredba instalira swagger-typescript-api. Parametar -D označava da se paket instalira kao razvojna ovisnost, a to znači da je ta ovisnost potrebna samo kod izgradnje aplikacije. Paket swagger-typescript-api nije potreban u produkcijskoj okolini kada je aplikacija već izgrađena.

Nakon instalacije, u package.json datoteku potrebno je dodati sljedeću npm skriptu:

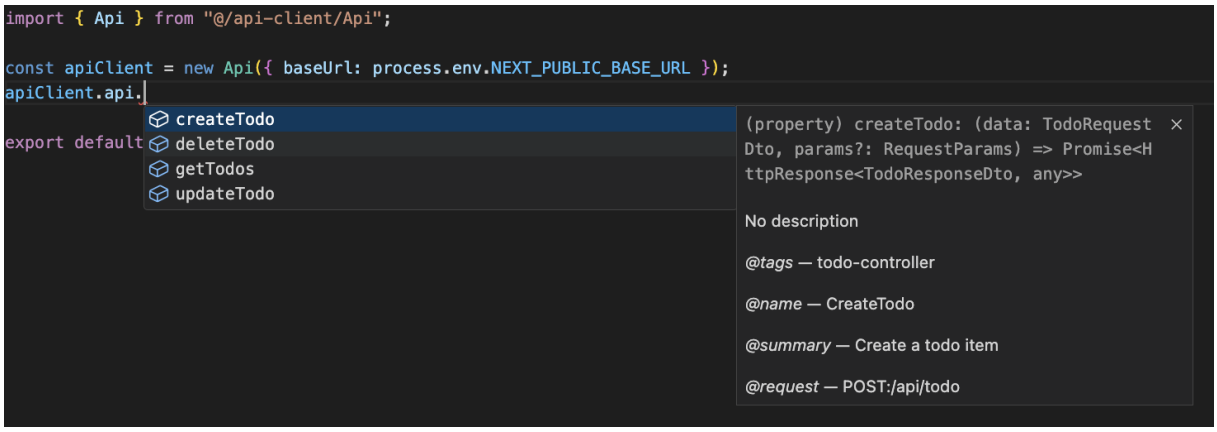
```
"generate-types": "swagger-typescript-api -p
↪  http://localhost:8888/todo-api/v3/api-docs -o ./api-client"
```

Ta će skripta pozvati swagger-typescript-api nad lokalno pokrenutom poslužiteljskom aplikacijom. Na temelju pokrenute aplikacije generirat će sve tipove i klijent za pristup resursima u kontroleru. Klijent će se spremiti u mapi api-client. Skripta se poziva naredbom:

```
% npm run generate-types
```

Nakon generacije tipova, TypeScript može prepoznati zahtjeve koji su loše formirani i sam uređivač koda (Visual Studio Code u ovom slučaju) može davati preporuke koda.

Sljedeća slika prikazuje korištenje generiranog API klijenta u jeziku TypeScript i automatsko popunjavanje koda od strane uređivača koda.



Slika 24: Automatsko popunjavanje koda koristeći API klijent (Izvor: autorski rad)

6.2.2. Komponente aplikacije

Sve kreće od početne stranice app/page.tsx. Ta stranica ponaša se kao poslužiteljska komponenta i dohvaća listu zadataka s poslužiteljske strane (server side rendering). Nakon što dohvati listu zadataka, prikazuje komponentu za kreiranje novog zadatka i komponentu za ispis zadataka.

```
async function getTodos() {
  const req = await apiClient.api.getTodos({
    format: "json",
    cache: "no-cache",
  });
  return req.data;
}

export default async function Home() {
```

```
  try {
    var todos = await getTodos();
  } catch (err) {
    console.log(err);
    return (
      <main className={styles.main}>
        <div className={styles.errorContainer}>
          <p style={{ flex: 1 }}>Nemoguće učitati</p>
        </div>
      </main>
    );
  }

  return (
    <main className={styles.main}>
      <NewTodoForm />
      <TodosTable todos={todos} />
    </main>
  );
}
```

Jako je važno u pozivanju API klijenta navesti parameter *cache: "no-cache"*. U protivnom se zahtjevi agresivno spremaju u predmemoriju i korisnici ne dobivaju svježe podatke.

Komponenta NewTodoForm služi kao obrazac za kreiranje novog zadatka i glasi:

```
const initialState = {
  content: "",
  dueDate: "",
};

function formDataReducer(
  state: typeof initialState,
  { set, value }: { set: string; value: string }
) {
  return {
    ...state,
    [set]: value,
  };
}

export default function NewTodoForm() {
  const router = useRouter();
```

```
const [formData, dispatchFormData] = useReducer(
  formDataReducer,
  initialState
);

async function createTodo(e: FormEvent<HTMLFormElement>) {
  e.preventDefault();
  await apiClient.api.createTodo({
    content: formData.content,
    dueDate:
      formData.dueDate.length > 0
        ? new Date(formData.dueDate).toISOString()
        : undefined,
  });
  startTransition(() => {
    router.refresh();
  });

  return false;
}

return (
  <div className={styles.newTodoFormContainer}>
    <form className={styles.newTodoForm} onSubmit={createTodo}>
      <input
        placeholder="Stavka..."
        value={formData.content}
        onChange={(e) =>
          dispatchFormData({ set: "content", value: e.currentTarget.value })
        }
        className={styles.input}
        style={{ flex: 2 }}
      />
      <input
        placeholder="Obaviti do..."
        type="datetime-local"
        value={formData.dueDate}
        onChange={(e) =>
          dispatchFormData({
            set: "dueDate",
            value: e.currentTarget.value,
          })
        }
```

```
          className={styles.input}
          style={{ flex: 1 }}
        />
        <input
          type="submit"
          value="OK"
          className={styles.input}
          style={{ flex: 0, minWidth: "100px" }}
        />
      </form>
    </div>
  );
}
```

NewTodoForm poprilično je jednostavna komponenta koja ažurira svoje podatke koristeći obrazac reduciranja. Kod obrasca reduciranja postoje trenutno stanje i radnje nad stanjima (tzv. reduceri) koje definiraju prijelaze stanje. U ovom se kodu u radnje šalju ime objekta u obrascu te njegova vrijednost, a radnja ažurira stanje na temelju podataka poslanih u radnji.

Prilikom slanja forme šalje se zahtjev na createTodo resurs i na taj se način kreira novi zadatak.

Komponenta TodosTable služi za prikazivanje tablice i pruža opcije uređivanja. Ta je komponenta nešto kompleksnija i sadrži nekoliko podkomponenti.

```
interface EditingState {
  id?: string;
  content?: string;
  dueDate?: string;
}

const initialEditingState: EditingState = {
  id: undefined,
  content: undefined,
  dueDate: undefined,
};

function editingDataReducer(
  state: EditingState,
  { set, value }: { set: string; value?: string | EditingState }
) {
  if (set === "batch") {
    return {
      ...(value as EditingState),
```

```
    };
  }
  if (set === "reset") {
    return {
      ...initialEditingState,
    };
  }
  return {
    ...state,
    [set]: value,
  };
}

export default function TodosTable({ todos }: { todos: TodoResponseDto[] }) {
  const [editingData, dispatchEditingData] = useReducer(
    editingDataReducer,
    initialEditingState
  );

  return (
    <table className={styles.todosTable}>
      <thead>
        <tr>
          <th>Stavka</th>
          <th>Kreirana</th>
          <th>Obaviti do</th>
          <th style={{ textAlign: "right" }}>Radnje</th>
        </tr>
      </thead>
      <tbody>
        {todos?.map((todo) => (
          <tr key={todo.id}>
            {editingData.id !== todo.id && (
              <ReadOnlyTableRow
                todo={todo}
                dispatchEditingData={dispatchEditingData}
              />
            )}
            {editingData.id === todo.id && (
              <EditingTableRow
                todo={todo}
                editingData={editingData}
                dispatchEditingData={dispatchEditingData}
```

```
              />
            ))}
          </tr>
        ))}
      </tbody>
    </table>
  );
}

function ReadOnlyTableRow({
  todo,
  dispatchEditingData,
}: {
  todo: TodoResponseDto;
  dispatchEditingData: Dispatch<{
    set: string;
    value: string | EditingState | undefined;
  }>;
}) {
  const router = useRouter();
  function editTodo(todo: TodoResponseDto) {
    dispatchEditingData({
      set: "batch",
      value: {
        id: todo.id,
        content: todo.content,
        dueDate: convertToDateTimeLocal(
          todo.dueDate ? new Date(todo.dueDate) : undefined
        ),
      },
    });
  }
  async function deleteTodo(id: string) {
    await apiClient.api.deleteTodo(id);
    startTransition(() => {
      router.refresh();
    });
  }
  return (
    <>
      <td>{todo.content}</td>
      <td>{new Date(todo.createdAt).toLocaleString("hr-HR")}</td>
      <td>
```

```
        {todo.dueDate ? new Date(todo.dueDate).toLocaleString("hr-HR") : "-"}
      </td>
      <td style={{ textAlign: "right" }}>
        <Image
          src={trashIco}
          className={styles.icon}
          alt="Obriši"
          width={25}
          height={25}
          onClick={() => {
            deleteTodo(todo.id);
          }}
        />
        <Image
          src={editIco}
          className={styles.icon}
          alt="Uredi"
          width={25}
          height={25}
          onClick={() => {
            editTodo(todo);
          }}
        />
      </td>
    </>
  );
}

function EditingTableRow({
  todo,
  editingData,
  dispatchEditingData,
}: {
  todo: TodoResponseDto;
  editingData: EditingState;
  dispatchEditingData: Dispatch<{
    set: string;
    value: string | EditingState | undefined;
  }>;
}) {
  const router = useRouter();
  async function updateTodo() {
    await apiClient.api.updateTodo(editingData.id!, {
```

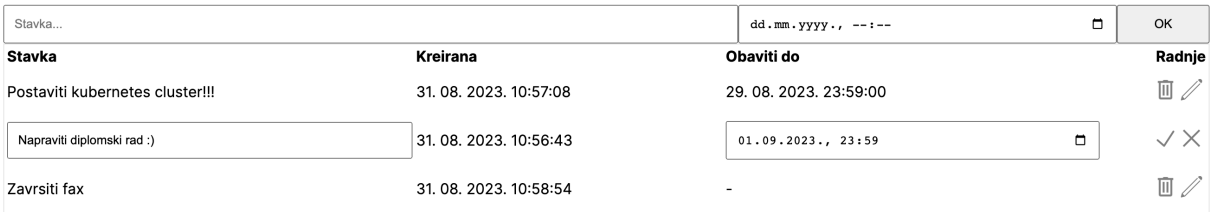
```
      content: editingData.content!,
      dueDate:
        editingData.dueDate && editingData.dueDate.length > 0
          ? new Date(editingData.dueDate).toISOString()
          : (null as unknown as undefined),
    });
    dispatchEditingData({ set: "reset", value: undefined });
    startTransition(() => {
      router.refresh();
    });
  }
  return (
    <>
      <td>
        <input
          className={styles.editInput}
          value={editingData.content}
          onChange={(e) =>
            dispatchEditingData({
              set: "content",
              value: e.currentTarget.value,
            })
          }
        />
      </td>
      <td>{new Date(todo.createdAt).toLocaleString("hr-HR")}</td>
      <td>
        <input
          className={styles.editInput}
          value={editingData.dueDate}
          type="datetime-local"
          onChange={(e) =>
            dispatchEditingData({
              set: "dueDate",
              value: e.currentTarget.value,
            })
          }
        />
      </td>
      <td style={{ textAlign: "right" }}>
        <Image
          src={tickIco}
          className={styles.icon}
```

```
            alt="OK"
            width={25}
            height={25}
            onClick={updateTodo}
          />
          <Image
            src={closeIco}
            className={styles.icon}
            alt="OK"
            width={25}
            height={25}
            onClick={() => {
              dispatchEditingData({ set: "reset", value: undefined });
            }}
          />
        </td>
      </>
    );
}
```

TodosTable komponenta također koristi obrazac reduciranja, ali ovog se puta taj obrazac koristi samo za praćenje retka koji se trenutno uređuje. Radnja dispatchEditingData kao parametar set prima ime polja koje se postavlja i vrijednost, ali također može primiti i vrijednosti "batch" i "reset". Vrijednost "batch" služi za postavljanje svih vrijednosti odjednom umjesto da se vrijednosti postavljaju jedna po jedna, a "reset" služi za resetiranje na početno stanje. Početno stanje bi u ovoj komponenti bilo stanje gdje se niti jedan redak ne uređuje. Klikom na gumb za prestanak uređivanja poziva se dispatchEditingData sa "set" parametrom s vrijednošću "reset" i na taj se način odustaje od uređivanja.

TodosTable poziva dvije podkomponente, a to su ReadOnlyTableRow i EditingTableRow. Te se komponente izmjenjuju za svaki redak ovisno o tome uređuje li se redak ili ne.

U sljedećoj se slici vidi konačan izgled aplikacije. Na vrhu je obrazac za kreiranje novog zadatka, a svaki se redak može brisati ili uređivati.



Slika 25: Izgled korisničke aplikacije (Izvor: autorski rad)

6.3. Kreiranje Docker slika kontejnera za aplikacije

Za isporuku aplikacija na Kubernetes klaster prvo je potrebno te aplikacije izgraditi u obliku slike kontejnera.

Dockerfile za poslužiteljsku aplikaciju glasi:

```
FROM maven:3.8.3-openjdk-17 AS maven_build
WORKDIR /build
COPY . .
RUN mvn clean package

FROM amazoncorretto:17-alpine3.17-full
RUN mkdir -p /opt/todo
COPY --from=maven_build /build/target/*.jar /opt/todo/app.jar
ENTRYPOINT ["sh", "-c", "java ${JAVA_OPTS} -jar /opt/todo/app.jar"]
```

Dockerfile za poslužiteljsku aplikaciju sastoji se od dva stadija - jedan za izgradnju aplikacije, a drugi za pokretanje te aplikacije. Razlog tome je taj što za izgradnju treba programski kod, a za pokretanje ne treba. Na ovaj se način postiže to da se aplikacija izgradi, ali u konačnoj će slici kontejnera biti samo .jar datoteka bez programskog koda, maven-a, itd.

U stadiju izgradnje prvo se koristi bazna slika s maven verzijom 3.8.3 i OpenJDK verzijom 17. Nakon toga se kopiraju sve datoteke u kontekst izgradnje te se pokreće naredba *mvn clean package* za izgradnju aplikacije. Drugi stadij koristi baznu sliku koja sadrži samo OpenJDK 17. U tu se sliku kopira .jar datoteka iz prošlog stadija i nakon toga se aplikacija pokreće.

Dockerfile za korisničku aplikaciju glasi:

```
FROM node:16-alpine AS build
ARG base_url=http://localhost:8888/todo-api
WORKDIR /build
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM node:16-alpine
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm i --production
COPY --from=build /build/.next /app/.next
COPY --from=build /build/public /app/public
ENTRYPOINT ["npm", "run", "start"]
```

Dockerfile za korisničku aplikaciju se također sastoji od dva stadija od kojih jedan služi za izgradnju, a drugi za pokretanje, baš kao i kod poslužiteljske aplikacije.

U stadiju izgradnje koristi se bazna slika koja sadrži Node.js verziju 16. U stadiju izgradnje se prvo definira parametar base_url, a njega treba definirati zato jer Next.js unaprijed izgrađuje HTML stranice prilikom izgradnje aplikacije. To znači da će aplikacija prilikom izgradnje poslati zahtjev na poslužiteljsku aplikaciju i spremiti predmemoriranu verziju stranice u konačni paket.

Dalje je potrebno kopirati datoteke package.json i package-lock.json zbog efikasne izgradnje. Naime, da se odmah kopiraju sve datoteke i onda se poziva *npm ci*, tada bi se korak za instaliranje ovisnosti pozivao svaki puta kada se izmjeni kod. To nije efikasno i nema potrebe za time te je zato dobro staviti taj korak prije kopiranja koda. Nakon kopiranja poziva se *npm ci* i pokreće instalaciju ovisnosti.

Nakon instaliranja ovisnosti kopiraju se sve datoteke i izgrađuje se aplikacija. Aplikacija će se izgraditi u mapi .next.

U drugom stadiju također se kopiraju package.json i package-lock.json. Nakon toga se instaliraju ovisnosti, ali samo one koje su potrebne za produkciju. To znači da se neće instalirati swagger-typescript-api jer on više ne treba. Tipovi su odavno generirani.

Nakon instalacije ovisnosti kopiraju se mape .next i public iz stadija izgradnje. Te su dvije mape važne jer mapa public sadrži statičke resurse poput medija, a .next mapa sadrži izgrađenu Next.js aplikaciju koju Node.js konzumira. Nakon kopiranja treba pokrenuti aplikaciju, a to se radi ulaznom točkom *npm run start*.

Sada slike treba izgraditi:

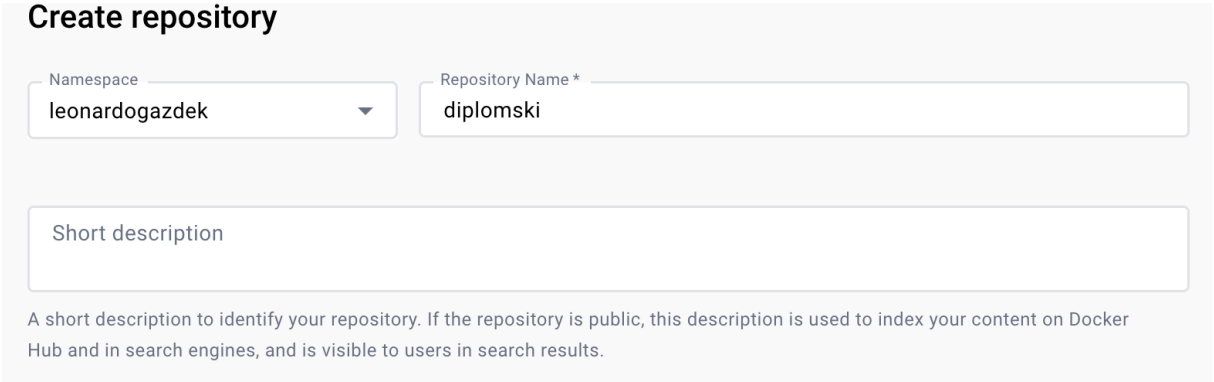
```
% docker build -t leonardogazdek/diplomski:frontend-x86v4
↪  --platform=linux/amd64 .

% docker build -t leonardogazdek/diplomski:backend-x86
↪  --platform=linux/amd64 .
```

Slike moraju imati tag u oblilu leonardogazdek/diplomski:*. Razlog tome je taj da je korisničko ime korisnika unutar Docker Hub platforme *leonardogazdek*, a ime repozitorija je *diplomski*. Potrebno je specificirati platformu linux/amd64 jer se slike izgrađuju na Apple M1 arm64 procesoru, a čvorovi u klasteru imaju amd64 arhitekturu procesora i nisu kompatibilni s amd64 slikama.

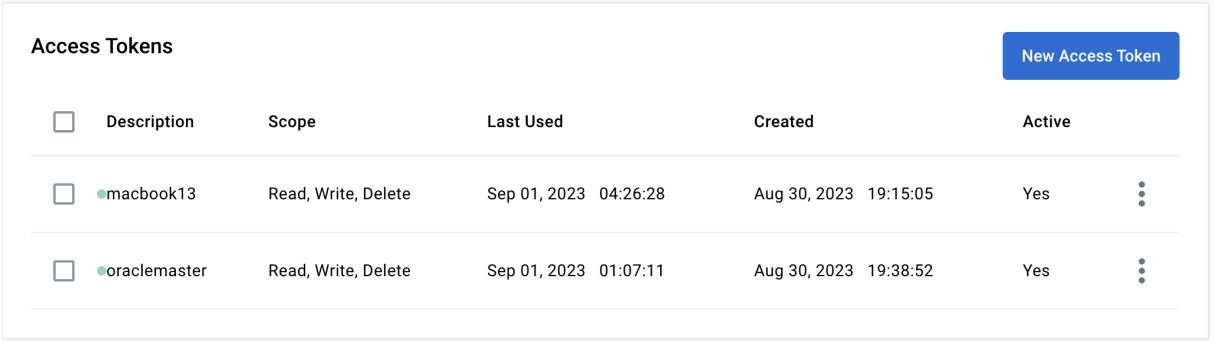
6.4. Prijenos slika na privatni Docker Hub repozitorij

Da bi Kubernetes mogao preuzeti kontejnere, njih je prvo potrebno prenijeti na Docker Hub. Prvo unutar platforme Docker Hub treba kreirati repozitorij pod imenom diplomski.



Slika 26: Kreiranje Docker Hub repozitorija (Izvor: autorski rad)

Nakon toga, u korisničkom računu treba kreirati dva pristupna žetona. Žetoni su potrebni jer se radi o privatnom repozitoriju i računalo se može autentificirati na Docker Hub koristeći pristupne žetone. Jedan će se koristiti za pristup lokalnog računala, a drugi za pristup Kubernetes klastera.



Slika 27: Docker Hub pristupni žetoni (Izvor: autorski rad)

Sada je na lokalnom računalu potrebno pozvati naredbu *docker login* s korisničkim imenom.

```
% docker login -u leonardogazdek
```

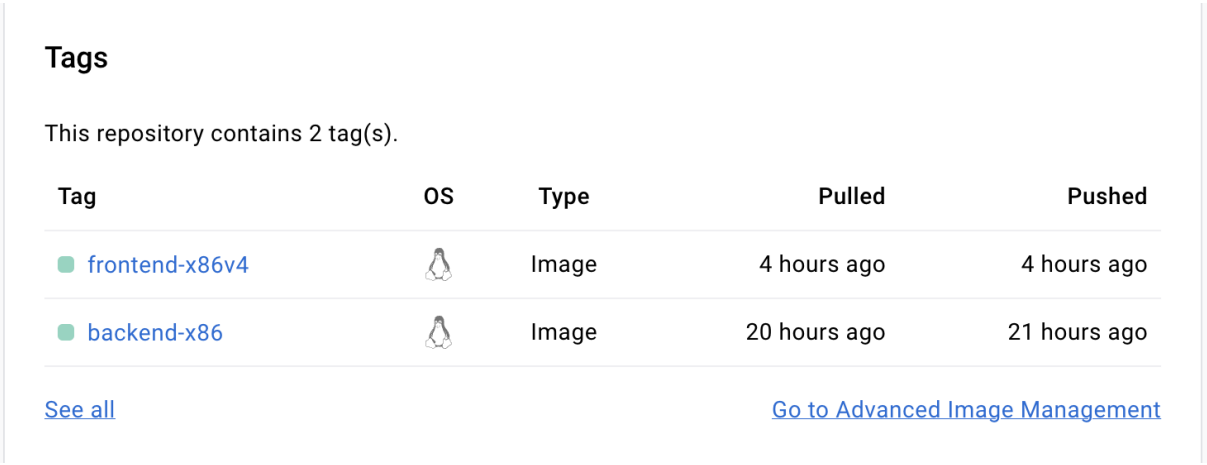
Nakon unosa naredbe, konzola pita za lozinku. Pristupni žetoni i lozinke su sinonimi u Docker svijetu tako da je pod lozinku dovoljno unijeti pristupni žeton. Sada je računalo spremno za slanje slika kontejnera na Docker Hub repozitorij.

Za slanje slika kontejnera poslužiteljske i klijentske aplikacije na Docker Hub potrebno je pozvati sljedeće naredbe:

```
% docker push leonardogazdek/diplomski:backend-x86
```

```
% docker push leonardogazdek/diplomski:frontend-x86v4
```

Sada se slike mogu vidjeti unutar Docker Hub repozitorija.



Slika 28: Slike unutar Docker Hub repozitorija (Izvor: autorski rad)

6.5. Kreiranje Kubernetes klastera

Prije samog postavljanja klastera potrebno se pobrinuti da su svi čvorovi klastera unutar iste podmreže i da mogu komunicirati. Ovo je već omogućeno pomoću virtualnih privatnih mreža u ranijim dijelovima ovog rada.

Prije početka ikakve instalacije trebalo bi otvoriti svu komunikaciju na podmreži na svim čvorovima klastera. To se može napraviti u datoteci /etc/iptables/rules.v4 dodavanjem sljedećeg pravila:

```
-A INPUT -s 10.0.0.0/24 -j ACCEPT
```

Za primjenu pravila potrebno je koristiti iptables-restore.

```
$ sudo iptables-restore < /etc/iptables/rules.v4
```

Kubernetesova mreža uvelike ovisi o premoštenom prometu (bridged traffic). Treba konfigurirati iptables da dozvoljava premošteni promet pozivanjem sljedećih naredbi na svim čvorovima.

```
$ cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

$ sudo modprobe overlay
$ sudo modprobe br_netfilter

$ cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
```

```
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF

sudo sysctl --system
```

Naredba *sudo sysctl –system* primjenjuje nova sysctl pravila bez potrebe za resetiranjem čvora.

U Linux sustavu treba isključiti swap. Swap je memorija diska koja se koristi u svrhu radne memorije. Kubernetes ne podržava swap jer se na njemu značajno gube performanse.

```
$ sudo swapoff -a
$ (crontab -l 2>/dev/null; echo "@reboot /sbin/swapoff -a") | crontab
↪  - || true
```

Prva naredba isključuje swap u trenutku pozivanja, a druga se brine da se swap isključi i prilikom ponovnog pokretanja čvora.

Sada treba instalirati cri-o pokretač kontejnera. Prvo treba podesiti crio konfiguraciju koja učitava određene module ljuske.

```
$ cat <<EOF | sudo tee /etc/modules-load.d/crio.conf
overlay
br_netfilter
EOF

$ sudo sysctl --system
```

Prije preuzimanja cri-o paketa potrebno je dodati njihove repozitorije u datoteku sources.list.d. Apt upravitelj paketa koristi tu datoteku kao listu izvora.

```
$ cat <<EOF | sudo tee
↪  /etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list
deb https://download.opensuse.org/repositories/devel:/kubic:/libcont⌟
↪  ainers:/stable/xUbuntu_20.04/
↪  /
EOF
$ cat <<EOF | sudo tee /etc/apt/sources.list.d/devel:kubic:libcontai⌟
↪  ners:stable:cri-o:1.23.list
deb http://download.opensuse.org/repositories/devel:/kubic:/libconta⌟
↪  iners:/stable:/cri-o:/1.23/xUbuntu_20.04/
↪  /
EOF
```

Za korištenje repozitorija potrebno je dodati i gpg ključeve. Apt upravitelj paketa koristi ove ključeve da potvrdi potpise paketa koji se preuzimaju.

```
$ curl -L https://download.opensuse.org/repositories/devel:kubic:lib⌟
↪  containers:stable:cri-o:$VERSION/$OS/Release.key | sudo apt-key
↪  --keyring /etc/apt/trusted.gpg.d/libcontainers.gpg add -
$ curl -L https://download.opensuse.org/repositories/devel:/kubic:/l⌟
↪  ibcontainers:/stable/$OS/Release.key | sudo apt-key --keyring
↪  /etc/apt/trusted.gpg.d/libcontainers.gpg add -
```

Sada se konačno može instalirati cri-o.

```
$ sudo apt-get update
$ sudo apt-get install cri-o cri-o-runc cri-tools -y
```

Nakon cri-o instalacije potrebno je ponovno učitati systemd i uključiti cri-o.

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable crio --now
```

Sada su čvorovi spremni za instalaciju Kubernetes platforme. Potrebno je instalirati pakete kubeadm, kubelet i kubectl.

Kubernetes ima nekoliko ovisnosti koje prvo treba instalirati:

```
$ sudo apt-get update
$ sudo apt-get install -y apt-transport-https ca-certificates curl
```

Potrebno je kreirati mapu /etc/apt/keyrings, u nju dodati preuzeti gpg ključ i dodati službeni Kubernetes repozitorij u listu izvora.

```
$ sudo mkdir -p /etc/apt/keyrings
$ echo "deb [signed-by=/etc/apt/keyrings/kubernetes.gpg]
↪  https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee
↪  /etc/apt/sources.list.d/kubernetes.list
$ curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg |
↪  sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes.gpg
$ sudo apt-get update -y
```

Sada je moguće instalirati Kubernetes komponente. Iznimno je važno "zaključati" verzije ovih Kubernetes komponenti naredbom *apt-mark hold* jer nepodudaranje verzija komponenti nije podržano.

```
$ sudo apt-get install -y kubelet kubeadm kubectl
$ sudo apt-mark hold kubelet kubeadm kubectl
```

Nakon instalacije komponenti moguće je inicijalizirati klaster. Ova će naredba kreirati klaster te će vezati kontrolnu razinu uz privatnu IP adresu glavnog čvora.

```
$ sudo kubeadm init --apiserver-advertise-address=10.0.0.238
 ↪  --apiserver-cert-extra-sans=10.0.0.238
 ↪  --pod-network-cidr=192.168.0.0/16 --node-name $(hostname -s)
[init] Using Kubernetes version: v1.28.1
...
Your Kubernetes control-plane has initialized successfully!
...
To start using your cluster, you need to run the following as a
 ↪  regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
...
Then you can join any number of worker nodes by running the following
 ↪  on each as root:

kubeadm join 10.0.0.238:6443 --token <zeton>
 ↪  --discovery-token-ca-cert-hash sha256:<sazetak>
```

Ispis ove naredbe iznimno je važan jer se u njemu nalazi žeton za pridruživanje Kubernetes klasteru. Za korištenje alata kubectl potrebno je pozvati naredbe iz ispisa prethodne naredbe:

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Za pridruživanje radnog čvora na klaster potrebno je pozvati naredbu *kubeadm join* koja je ispisana prilikom inicijalizacije klastera. Tu je naredbu, naravno, potrebno pozvati na radnom čvoru.

```
$ sudo kubeadm join 10.0.0.238:6443 --token <zeton>
 ↪  --discovery-token-ca-cert-hash sha256:<sazetak>
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n
 ↪  kube-system get cm kubeadm-config -o yaml'
```

```
[kubelet-start] Writing kubelet configuration to file
↪  "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file
↪  "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS
↪  Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response
↪  was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join
↪  the cluster.
```

Klaster bi sada trebao biti potpuno inicijaliziran. To je moguće provjeriti naredbom *kubectl get nodes* na glavnom čvoru.

```
$ kubectl get nodes
NAME                     STATUS   ROLES           AGE   VERSION
instance-20230831-0647   Ready    control-plane   23h   v1.28.1
w1                       Ready    <none>          23h   v1.28.1
```

Kubernetes ne dolazi s mrežnim sučeljem (CNI - container network interface). Mrežno se sučelje treba naknadno instalirati. Za ovaj je klaster odabrano mrežno sučelje Calico.

Za instalaciju Calico sučelja potrebno je preuzeti Calico manifest datoteku i istu primijeniti na klaster koristeći naredbu *kubectl apply*.

```
$ curl https://raw.githubusercontent.com/projectcalico/calico/v3.26.┘
↪  1/manifests/calico.yaml
↪  -O
$ kubectl apply -f calico.yaml
```

Nakon instaliranja Calico sučelja moguće je vidjeti pokrenute grupe naredbom *kubectl get pods -n kube-system*. Calico grupe nalaze se u imenskom prostoru kube-system.

```
$ kubectl get pods -n kube-system
NAME                                READY   STATUS    RESTARTS   AGE
calico-kube-controllers-...         1/1     Running   6          22h
calico-node-...                     1/1     Running   2          22h
calico-node-...                     1/1     Running   3          22h
coredns-5dd5756b68-dgx4s            1/1     Running   0          19h
coredns-5dd5756b68-vm29x            1/1     Running   0          19h
```

```
etcd-instance-...                    1/1     Running   3          23h
kube-apiserver-instance-...          1/1     Running   3          23h
kube-controller-manager-instance-...1/1     Running   5          23h
kube-proxy-9t6d9                     1/1     Running   2          21h
kube-proxy-g6wln                     1/1     Running   1          21h
kube-scheduler-instance-...          1/1     Running   5          23h
```

Izlaz gornje naredbe potvrđuje da je Calico uspješno instaliran i aktivan. Klaster je sada spreman za rad i isporuku aplikacija.

6.6. Isporučivanje aplikacija na klaster

Prije kreiranja isporuka aplikacija potrebno je kreirati tajni registar kontejnera pod nazivom *regcred*. Pomoću tajnog registra Kubernetes klaster komunicirati će s privatnim Docker Hub repozitorijom. U idućoj naredbi moguće je koristiti prethodno generirani pristupni žeton kao lozinku.

```
$ kubectl create secret docker-registry regcred
 ↪  --docker-username=leonardogazdek
 ↪  --docker-password=<pristupni_zeton>
 ↪  --docker-email=leonardogazdek@gmail.com
```

Nakon kreiranja tajnog registra moguće je kreirati Kubernetes isporuku koja sadrži kontejner poslužiteljske aplikacije. Za tu isporuku potrebno je kreirati i servis tipa ClusterIP koji se veže na virtualne IP adrese Kubernetes klastera i na taj način omogućava umrežavanje.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: diplomski-todo
spec:
  selector:
    matchLabels:
      app: diplomski-todo
  replicas: 2
  template:
    metadata:
      labels:
        app: diplomski-todo
    spec:
      containers:
        - name: diplomski
          image: leonardogazdek/diplomski:backend-x86
```

```
          envFrom:
            - configMapRef:
                name: backend-env
          resources:
            requests:
              memory: "256Mi"
              cpu: "150m"
            limits:
              memory: "768Mi"
              cpu: "500m"
          ports:
            - containerPort: 8888
          imagePullPolicy: Always
      imagePullSecrets:
          - name: regcred

---
apiVersion: v1
kind: Service
metadata:
  name: diplomski-todo-service
spec:
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 9000
      targetPort: 8888
  selector:
    app: diplomski-todo
```

Gornja datoteka manifestiranja definira isporuku pod imenom diplomski-todo koja sadrži aplikaciju diplomski-todo. Aplikacija koristi sliku kontejnera *leonardogazdek/diplomski:backend-x86* iz tajnog registra *regcred*. Isporuka će kreirati dvije replike i koristi varijable okoline iz ConfigMap resursa backend-env.

Resurs backend-env trenutno ne postoji pa ga treba i kreirati:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: backend-env
  namespace: default
data:
```

```
  ACCESS_KEY: (skriveno)
  SECRET_ACCESS_KEY: (skriveno)
  REGION: eu-central-1
  DYNAMODB_TODO_TABLE_NAME: todo-diplomski
```

Varijable definirane u backend-env resursu biti će dostupne Spring aplikaciji unutar kontejnera.

Nakon kreiranja datoteka manifestiranja potrebno ih je primjeniti na klaster:

```
$ kubectl apply -f backend-env.yml
$ kubectl apply -f backend-deployment.yml
```

Sada su u ispisu naredbe *kubectl get pods* vidljive nove grupe:

```
$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
diplomski-todo-69bf76bc48-2grrz         1/1     Running   3          22h
diplomski-todo-69bf76bc48-m8npw         1/1     Running   3          22h
```

Vidljivo je da postoje dvije replike grupa diplomski-todo. Poslužiteljska aplikacija sada je službeno pokrenuta što se može i potvrditi naredbom *kubectl logs*.

```
$ kubectl logs diplomski-todo-69bf76bc48-2grrz

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.1.3)

2023-08-31T08:11:02.377Z  INFO 1 --- [           main]
 ↪  o.l.diplomski.todo.TodoApplication       : Starting
 ↪  TodoApplication v0.0.1-SNAPSHOT using Java 17.0.8 with PID 1
 ↪  (/opt/todo/app.jar started by root in /)
2023-08-31T08:11:02.380Z  INFO 1 --- [           main]
 ↪  o.l.diplomski.todo.TodoApplication       : No active profile set,
 ↪  falling back to 1 default profile: "default"
2023-08-31T08:11:09.779Z  INFO 1 --- [           main]
 ↪  o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized
 ↪  with port(s): 8888 (http)
2023-08-31T08:11:09.791Z  INFO 1 --- [           main]
 ↪  o.apache.catalina.core.StandardService   : Starting service
 ↪  [Tomcat]
```

```
2023-08-31T08:11:09.792Z  INFO 1 --- [           main]
 ↪  o.apache.catalina.core.StandardEngine    : Starting Servlet
 ↪  engine: [Apache Tomcat/10.1.12]
2023-08-31T08:11:10.594Z  INFO 1 --- [           main]
 ↪  o.a.c.c.C.[.[localhost].[/todo-api]      : Initializing Spring
 ↪  embedded WebApplicationContext
2023-08-31T08:11:10.596Z  INFO 1 --- [           main]
 ↪  w.s.c.ServletWebServerApplicationContext : Root
 ↪  WebApplicationContext: initialization completed in 7895 ms
2023-08-31T08:11:18.491Z  INFO 1 --- [           main]
 ↪  o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on
 ↪  port(s): 8888 (http) with context path '/todo-api'
2023-08-31T08:11:18.591Z  INFO 1 --- [           main]
 ↪  o.l.diplomski.todo.TodoApplication       : Started
 ↪  TodoApplication in 20.085 seconds (process running for 28.18)
2023-08-31T08:30:02.621Z  INFO 1 --- [0.0-8888-exec-1]
 ↪  o.a.c.c.C.[.[localhost].[/todo-api]      : Initializing Spring
 ↪  DispatcherServlet 'dispatcherServlet'
2023-08-31T08:30:02.621Z  INFO 1 --- [0.0-8888-exec-1]
 ↪  o.s.web.servlet.DispatcherServlet        : Initializing Servlet
 ↪  'dispatcherServlet'
2023-08-31T08:30:02.623Z  INFO 1 --- [0.0-8888-exec-1]
 ↪  o.s.web.servlet.DispatcherServlet        : Completed
 ↪  initialization in 1 ms
2023-08-31T08:30:04.785Z  INFO 1 --- [0.0-8888-exec-8]
 ↪  o.springdoc.api.AbstractOpenApiResource  : Init duration for
 ↪  springdoc-openapi is: 1105 ms
```

Sada treba kreirati isporuku koja sadrži kontejner korisničke aplikacije. Za tu isporuku je također potrebno kreirati i servis tipa ClusterIP koji se veže na virtualne IP adrese Kubernetes klastera i omogućava umrežavanje.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: diplomski-todo-frontend
spec:
  selector:
    matchLabels:
      app: diplomski-todo-frontend
  replicas: 2
  template:
    metadata:
```

```
      labels:
        app: diplomski-todo-frontend
    spec:
      containers:
        - name: diplomski-frontend
          image: leonardogazdek/diplomski:frontend-x86v4
          envFrom:
            - configMapRef:
                name: frontend-env-4
          resources:
            requests:
              memory: "256Mi"
              cpu: "150m"
            limits:
              memory: "768Mi"
              cpu: "400m"
          ports:
            - containerPort: 3000
          imagePullPolicy: Always
      imagePullSecrets:
          - name: regcred

---
apiVersion: v1
kind: Service
metadata:
  name: diplomski-todo-frontend-service
spec:
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 9001
      targetPort: 3000
  selector:
    app: diplomski-todo-frontend
```

Gornja datoteka manifestiranja definira isporuku pod imenom diplomski-todo-frontend koja sadrži aplikaciju diplomski-todo-frontend. Aplikacija koristi sliku kontejnera *leonardogazdek/diplomski:frontend-x86v4*iz tajnog registra *regcred*. Isporuka će kreirati dvije replike i koristi varijable okoline iz ConfigMap resursa frontend-env.

Resurs frontend-env trenutno ne postoji pa ga treba i kreirati:

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: frontend-env-4
  namespace: default
data:
  NEXT_PUBLIC_BASE_URL: http://141.147.10.108.nip.io/todo-api
```

Servis nip.io usmjerava promet u IP adresu definiranu u poddomeni. Razlog zašto nije definirana samo IP adresa je taj da Nginx Ingress kontroler koji će se koristiti za pristup klasteru ne podržava korištenje IP adresa bez domene. Za ovakav probni klaster ne isplati se kupovati domenu pa je servis nip.io dobra alternativa.

IP adresa 141.147.10.108 javna je adresa radnog čvora na kojem će se pokretati servis tipa Ingress koji dozvoljava promet izvana.

Varijabla NEXT_PUBLIC_BASE_URL biti će dostupna unutar Next.js aplikacije.

Nakon kreiranja datoteka manifestiranja potrebno ih je primjeniti na klaster:

```
$ kubectl apply -f frontend-env.yml
$ kubectl apply -f frontend-deployment.yml
```

Sada su u ispisu naredbe *kubectl get pods* vidljive nove grupe:

```
$ kubectl get pods
diplomski-todo-69bf76bc48-2grrz          1/1     Running   3          22h
diplomski-todo-69bf76bc48-m8npw          1/1     Running   3          22h
diplomski-todo-frontend-649b68dc4d-jshn2 1/1     Running   0          5h41m
diplomski-todo-frontend-649b68dc4d-s5jq  1/1     Running   0          5h41m
```

Vidljivo je da su se dodale dvije replike grupa diplomski-todo-frontend. Korisnička aplikacija sada je službeno pokrenuta što se može i potvrditi naredbom *kubectl logs*.

```
$ kubectl logs diplomski-todo-frontend-649b68dc4d-jshn2

> diplomski-todo-frontend@0.1.0 start
> next start

- ready started server on [::]:3000, url: http://localhost:3000
```

Za kreiranje Ingress servisa potrebno je instalirati Ingress kontroler. Jedan od popularnih je ingress-nginx. Prvo je potrebno instalirati upravitelj paketa za Kubernetes - helm.

```
$ wget https://get.helm.sh/helm-v3.12.3-linux-amd64.tar.gz
$ tar -zxvf helm-v3.0.0-linux-amd64.tar.gz
$ mv linux-amd64/helm /usr/local/bin/helm
```

Nakon instalacije helm upravitelja paketa, pomoću njega treba instalirati ingress-nginx s tipom servisa NodePort:

```
$ helm upgrade --install ingress-nginx ingress-nginx \
  --repo https://kubernetes.github.io/ingress-nginx \
  --namespace ingress-nginx --create-namespace
  --set controller.service.type=NodePort
```

Razlog zbog kojeg se koristi NodePort je taj da je servis tipa LoadBalancer nepodržan u standardnoj instalaciji Kubernetes platforme. Velike platforme u oblaku poput AWS, Google Cloud, itd. podržavaju svoje implementacije LoadBalancer servisa. Kada se klaster postavlja na lokalnom poslužitelju potrebno je koristiti softversko rješenje poput MetalLB ili treba koristiti NodePort.

Sada je moguće saznati priključak na kojem sluša ingress-nginx.

```
$ kubectl get services --namespace ingress-nginx
NAME                                 ...   PORT(S)
ingress-nginx-controller             ...   80:31032/TCP,443:31620/TCP
ingress-nginx-controller-admission   ...   443/TCP
```

Radi se o priključku 31032. Sada treba modificirati frontend-env.yml:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: frontend-env-4
  namespace: default
data:
  NEXT_PUBLIC_BASE_URL: http://141.147.10.108.nip.io:31032/todo-api
```

Na IP adresu u konfiguraciji dodan je priključak 31032. Sada je potrebno konfigurirati ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: diplomski-todo-ingress
spec:
  rules:
  - host: 141.147.10.108.nip.io
    http:
      paths:
      - path: /todo-api
        pathType: Prefix
```

```
        backend:
          service:
            name: diplomski-todo-service
            port:
              number: 9000
      - path: /
        pathType: Prefix
        backend:
          service:
            name: diplomski-todo-frontend-service
            port:
              number: 9001
  ingressClassName: nginx
```

Ovaj ingress definira dvije putanje:

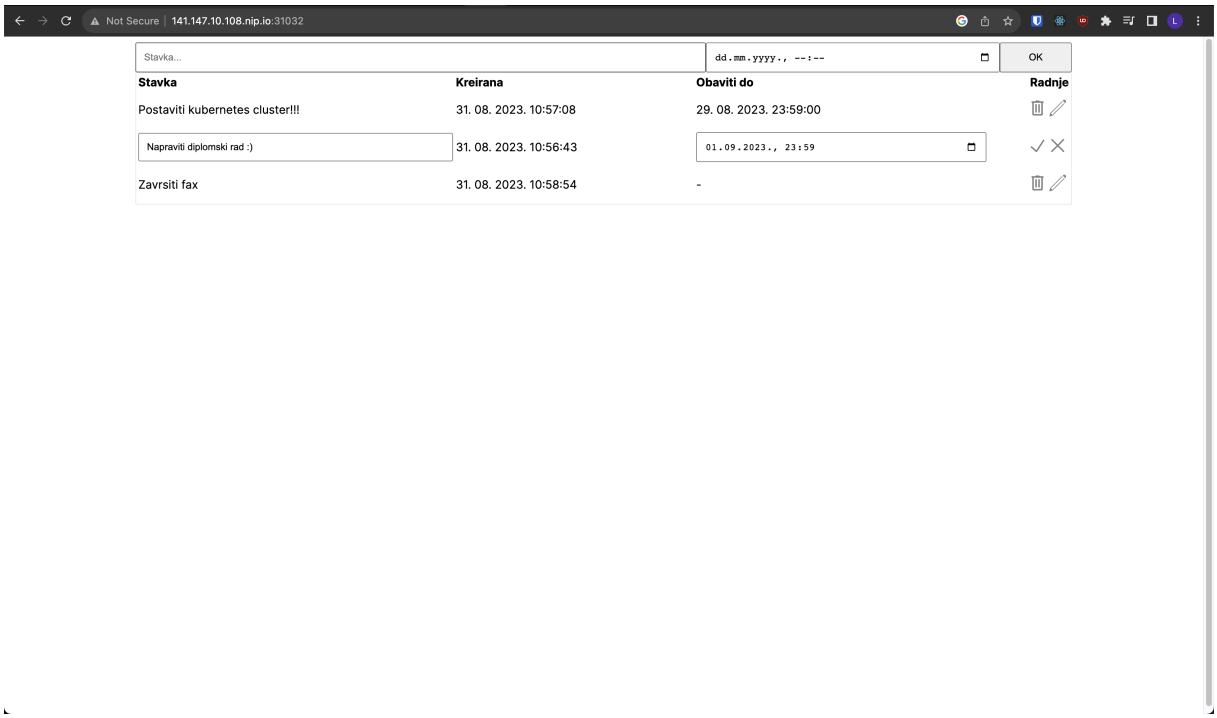
- / za korisničku aplikaciju
- /todo-api za poslužiteljsku aplikaciju

Ingress se veže na priključke definirane u servisima. To su 9000 za poslužiteljsku aplikaciju i 9001 za korisničku aplikaciju.

Sada treba primjeniti tu datoteku manifestiranja:

```
$ kubectl apply -f ingress.yml
```

Aplikacija je službeno dostupna na javnoj mreži nakon postavljanja ingress servisa. Taj će ingress, čak i kao NodePort, raspoređivati promet ravnomjerno kroz grupe u servisima.



Slika 29: Javno dostupna isporuka aplikacije (Izvor: autorski rad)

6.7. MetalLB konfiguracija

Velik problem NodePort konfiguracije je taj što se priključak ne može birati. Do sad se aplikaciji pristupalo na adresi http://141.147.10.108.nip.io:31032/, a standardni HTTP priključak je 80. Koristeći MetalLB moguće je konfigurirati da ingress bude dostupan na lokalnoj IP adresi unutar mreže. Adresa ne mora biti vezana uz fizički čvor jer se prevodi preko NAT protokola.

MetalLB softverska je implementacija LoadBalancer servisa za Kubernetes. Za MetalLB prvo treba uključiti strogi ARP u kube-proxy servisu:

```
$ kubectl get configmap kube-proxy -n kube-system -o yaml | \
sed -e "s/strictARP: false/strictARP: true/" | \
kubectl apply -f - -n kube-system
```

Nakon toga potrebno je instalirati MetalLB pomoću datoteke manifestiranja:

```
$ kubectl apply -f https://raw.githubusercontent.com/metallb/metallb
↪  /v0.13.10/config/manifests/metallb-native.yaml
```

U prethodnim koracima postavljeno je da ingress-nginx koristi tip servisa NodePort. Potrebno ga je konfigurirati da koristi tip servisa LoadBalancer.

```
$ helm upgrade --install ingress-nginx ingress-nginx   --repo
↪  https://kubernetes.github.io/ingress-nginx   --namespace
↪  ingress-nginx --create-namespace --set
↪  controller.service.type=LoadBalancer
```

Za integraciju s Calico mrežnim sučeljem potrebno je definirati podmrežu za IP adrese raspoređivača prometa unutar BGP konfiguracije:

```
$ calicoctl patch BGPConfig default --patch '{"spec":
↪  {"serviceLoadBalancerIPs": [{"cidr": "10.0.0.0/24"}]}}'
```

Nakon toga je potrebno kreirati servis tipa IPAddressPool pomoću datoteke manifestiranja. IP adrese u rasponu moraju pripadati istoj podmreži. Kao primjer će se koristiti raspon IP adresa 10.0.0.15-10.0.0.20.

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: production
  namespace: metallb-system
spec:
  addresses:
  - 10.0.0.15-10.0.0.20

$ kubectl apply -f addresspool.yaml
```

Potrebno je saznati IP adresu objekta tipa LoadBalancer:

```
$ kubectl get services -n ingress-nginx
NAME                                 TYPE          ...     EXTERNAL-IP
ingress-nginx-controller             LoadBalancer  ...     10.0.0.15
ingress-nginx-controller-admission   ClusterIP     ...     <none>
```

Sada je moguće pristupiti aplikaciji na privatnoj adresi http://10.0.0.15. Priključak će uvijek biti 80. Vidljivo je da curl odgovara na zahtjeve na adresi http://10.0.0.15 što znači da aplikacija radi.

```
$ curl --header "Host: 141.147.10.108.nip.io" http://10.0.0.15
<!DOCTYPE html><html lang="en">...
```

Sada samo treba postaviti obrnuti proxy koji će preusmjeravati intenetski promet na http://10.0.0.15. Na radni čvor se može instalirati nginx.

```
$ sudo apt install nginx
```

U datoteci /etc/nginx/sites-available/default potrebno je promijeniti *location* blok i u njemu definirati obrnuti proxy. Također je potrebno proslijediti i **host** zaglavlje.

```
location / {
    proxy_pass http://10.0.0.15;
    proxy_set_header Host            $host;
}
```

Ponovnim pokretanjem nginx servisa učitavaju se nove postavke.

```
$ sudo systemctl restart nginx
```

Sada je aplikacija dostupna na priključku 80, tj. na adresi http://141.147.10.108.nip.io.

Slika 30: Javno dostupna isporuka aplikacije na priključku 80 (Izvor: autorski rad)

U idealnoj bi postavi izvan klastera, ali na istoj mreži, postojao još jedan poslužitelj koji bi samo usmjeravao promet na klaster. U ovoj postavi to radi radni čvor klastera koji na sebi ima instaliran nginx kao obrnuti proxy.

6.8. Demonstracija raspoređivanja opterećenja

Raspoređivanje opterećenja prometa trebalo bi biti vidljivo nakon postavljanja servisa za korisničku aplikaciju, poslužiteljsku aplikaciju i MetalLB.

Servisi u klasteru namijenjeni su da ravnomjerno raspoređuju promet kroz grupe kontejnera unutar njih. S obzirom na to da su obje aplikacije postavljene da imaju dvije replike, promet bi se trebao ravnomjerno raspoređivati kroz te dvije replike.

Raspoređivanje opterećenja biti će demonstrirano na poslužiteljskoj aplikaciji. U svaku rutu kontrolera stavit će se naredba za ispis zahtjeva. Na taj će se način moći pratiti svaki puta kada određena grupa kontejnera obradi zahtjev.

Za ispis zahtjeva koristit će se Simple Logging Facade for Java (SJF4J). Lombok pruža anotaciju @Slf4j koja kreira atribut *log* kao instancu klase *org.slf4j.Logger*.

U svakoj od ruta na kontroleru ispisat će se kada stigne zahtjev.

```
@RestController
@RequiredArgsConstructor
@Slf4j
public class TodoController {
    // ...

    @Operation(summary = "Gets all todo items")
    @GetMapping("/api/todo")
    public List<TodoResponseDto> getTodos() {
        log.info("Primljen zahtjev getTodos");
        // ostatak koda...
    }

    @Operation(summary = "Create a todo item")
    @PostMapping("/api/todo")
    public TodoResponseDto createTodo(@RequestBody @Valid
    ↪  TodoRequestDto reqDto) {
        log.info("Primljen zahtjev createTodo");
        // ostatak koda...
    }

    @Operation(summary = "Delete a todo item")
    @DeleteMapping("/api/todo/{id}")
    @ResponseStatus(value = HttpStatus.NO_CONTENT)
    public void deleteTodo(@PathVariable String id) {
        log.info("Primljen zahtjev deleteTodo");
        // ostatak koda...
    }

    @Operation(summary = "Update a todo item")
    @PatchMapping("/api/todo/{id}")
    public void updateTodo(@PathVariable String id, @RequestBody
    ↪  @Valid TodoRequestDto reqDto) {
        log.info("Primljen zahtjev updateTodo");
        // ostatak koda...
    }
}
```

Za slanje zahtjeva koristi se jednostavna Node.js skripta:

```
import { Api } from "api-client/Api";
```

```
const apiClient = new Api();

async function saljiZahtjeve() {
  const noviTodo = (
    await apiClient.api.createTodo(
      {
        content: "Testna stavka",
        dueDate: new Date().toISOString(),
      },
      { format: "json" }
    )
  ).data;
  await apiClient.api.deleteTodo(noviTodo.id);
}

for (let i = 0; i < 100; i++) {
  saljiZahtjeve();
}
```

Navedena skripta istovremeno kreira 100 stavki u aplikaciji, a potom ih obriše. To znači da skripta sveukupno šalje 200 zahtjeva. Nakon pozivanja skripte očekivano je da će se u zapisima obje replike poslužiteljskog servisa vidjeti 100 zahtjeva što bi značilo da se promet ravnomjerno raspoređuje.

Treba izgraditi i pozvati skriptu:

```
% npm run build
% npm run start
```

Sada treba saznati identifikatore svih grupa kontejnera koje se pokreću na klasteru:

```
$ kubectl get pods -n default
NAME                                    READY   STATUS    RESTARTS
 ↪  AGE
diplomski-todo-774b56d45-jtr8l          1/1     Running   0
 ↪  10m
diplomski-todo-774b56d45-xjkmf          1/1     Running   0
 ↪  10m
diplomski-todo-frontend-6f47ccfc95-qs7t4 1/1    Running   2
 ↪  3d3h
diplomski-todo-frontend-6f47ccfc95-vr8wb 1/1    Running   2
 ↪  3d3h
```

U zapisima prve grupe kontejnera vidljivo je točno 100 zahtjeva:

```
$ kubectl logs diplomski-todo-774b56d45-jtr8l

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.1.3)

2023-09-08T16:28:30.937Z  INFO 1 --- [           main]
 ↪  o.l.diplomski.todo.TodoApplication       : Starting
 ↪  TodoApplication v0.0.1-SNAPSHOT using Java 17.0.8 with PID 1
 ↪  (/opt/todo/app.jar started by root in /)
2023-09-08T16:28:30.941Z  INFO 1 --- [           main]
 ↪  o.l.diplomski.todo.TodoApplication       : No active profile
 ↪  set, falling back to 1 default profile: "default"
2023-09-08T16:28:38.942Z  INFO 1 --- [           main]
 ↪  o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized
 ↪  with port(s): 8888 (http)
2023-09-08T16:28:38.960Z  INFO 1 --- [           main]
 ↪  o.apache.catalina.core.StandardService   : Starting service
 ↪  [Tomcat]
2023-09-08T16:28:38.961Z  INFO 1 --- [           main]
 ↪  o.apache.catalina.core.StandardEngine    : Starting Servlet
 ↪  engine: [Apache Tomcat/10.1.12]
2023-09-08T16:28:39.858Z  INFO 1 --- [           main]
 ↪  o.a.c.c.C.[.[localhost].[/todo-api]      : Initializing Spring
 ↪  embedded WebApplicationContext
2023-09-08T16:28:39.860Z  INFO 1 --- [           main]
 ↪  w.s.c.ServletWebServerApplicationContext : Root
 ↪  WebApplicationContext: initialization completed in 8602 ms
2023-09-08T16:28:49.038Z  INFO 1 --- [           main]
 ↪  o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on
 ↪  port(s): 8888 (http) with context path '/todo-api'
2023-09-08T16:28:49.141Z  INFO 1 --- [           main]
 ↪  o.l.diplomski.todo.TodoApplication       : Started
 ↪  TodoApplication in 21.404 seconds (process running for 24.372)
2023-09-08T16:28:49.745Z  INFO 1 --- [0.0-8888-exec-1]
 ↪  o.a.c.c.C.[.[localhost].[/todo-api]      : Initializing Spring
 ↪  DispatcherServlet 'dispatcherServlet'
```

```
2023-09-08T16:28:49.745Z  INFO 1 --- [0.0-8888-exec-1]
↪  o.s.web.servlet.DispatcherServlet        : Initializing Servlet
↪  'dispatcherServlet'
2023-09-08T16:28:49.746Z  INFO 1 --- [0.0-8888-exec-1]
↪  o.s.web.servlet.DispatcherServlet        : Completed
↪  initialization in 1 ms
2023-09-08T16:29:24.955Z  INFO 1 --- [.0-8888-exec-29]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.037Z  INFO 1 --- [.0-8888-exec-48]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.037Z  INFO 1 --- [.0-8888-exec-44]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.038Z  INFO 1 --- [.0-8888-exec-28]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.038Z  INFO 1 --- [.0-8888-exec-32]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.039Z  INFO 1 --- [0.0-8888-exec-4]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.055Z  INFO 1 --- [0.0-8888-exec-3]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.039Z  INFO 1 --- [.0-8888-exec-23]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.039Z  INFO 1 --- [.0-8888-exec-20]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.039Z  INFO 1 --- [.0-8888-exec-24]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.039Z  INFO 1 --- [.0-8888-exec-33]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.039Z  INFO 1 --- [.0-8888-exec-17]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
```

```
2023-09-08T16:29:25.039Z  INFO 1 --- [0.0-8888-exec-6]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.039Z  INFO 1 --- [.0-8888-exec-25]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.039Z  INFO 1 --- [.0-8888-exec-18]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.040Z  INFO 1 --- [.0-8888-exec-45]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.057Z  INFO 1 --- [0.0-8888-exec-5]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.040Z  INFO 1 --- [.0-8888-exec-46]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.040Z  INFO 1 --- [.0-8888-exec-19]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.058Z  INFO 1 --- [.0-8888-exec-43]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.040Z  INFO 1 --- [0.0-8888-exec-2]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.041Z  INFO 1 --- [.0-8888-exec-30]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.041Z  INFO 1 --- [.0-8888-exec-14]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.041Z  INFO 1 --- [.0-8888-exec-42]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.059Z  INFO 1 --- [0.0-8888-exec-1]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.059Z  INFO 1 --- [.0-8888-exec-31]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
```

```
2023-09-08T16:29:25.042Z  INFO 1 --- [.0-8888-exec-35]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.042Z  INFO 1 --- [.0-8888-exec-40]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.042Z  INFO 1 --- [.0-8888-exec-26]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.061Z  INFO 1 --- [.0-8888-exec-15]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.141Z  INFO 1 --- [.0-8888-exec-12]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.149Z  INFO 1 --- [.0-8888-exec-50]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.151Z  INFO 1 --- [.0-8888-exec-34]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.042Z  INFO 1 --- [.0-8888-exec-37]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.042Z  INFO 1 --- [.0-8888-exec-41]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.042Z  INFO 1 --- [.0-8888-exec-22]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.053Z  INFO 1 --- [.0-8888-exec-21]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.053Z  INFO 1 --- [.0-8888-exec-39]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.054Z  INFO 1 --- [.0-8888-exec-49]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.055Z  INFO 1 --- [.0-8888-exec-38]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
```

```
2023-09-08T16:29:25.138Z  INFO 1 --- [.0-8888-exec-16]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.042Z  INFO 1 --- [.0-8888-exec-13]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.145Z  INFO 1 --- [.0-8888-exec-47]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.146Z  INFO 1 --- [0.0-8888-exec-9]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.146Z  INFO 1 --- [0.0-8888-exec-8]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.147Z  INFO 1 --- [.0-8888-exec-10]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.148Z  INFO 1 --- [.0-8888-exec-36]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.148Z  INFO 1 --- [.0-8888-exec-27]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.149Z  INFO 1 --- [0.0-8888-exec-7]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.150Z  INFO 1 --- [.0-8888-exec-11]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:26.652Z  INFO 1 --- [0.0-8888-exec-2]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:27.141Z  INFO 1 --- [.0-8888-exec-50]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:27.151Z  INFO 1 --- [.0-8888-exec-39]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:27.355Z  INFO 1 --- [.0-8888-exec-15]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
```

```
2023-09-08T16:29:27.840Z  INFO 1 --- [.0-8888-exec-44]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:27.847Z  INFO 1 --- [.0-8888-exec-38]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:27.946Z  INFO 1 --- [0.0-8888-exec-1]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.264Z  INFO 1 --- [.0-8888-exec-42]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.341Z  INFO 1 --- [.0-8888-exec-23]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.342Z  INFO 1 --- [0.0-8888-exec-4]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.356Z  INFO 1 --- [.0-8888-exec-48]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.456Z  INFO 1 --- [.0-8888-exec-45]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.556Z  INFO 1 --- [0.0-8888-exec-2]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.559Z  INFO 1 --- [.0-8888-exec-36]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.560Z  INFO 1 --- [.0-8888-exec-35]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.648Z  INFO 1 --- [.0-8888-exec-10]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.664Z  INFO 1 --- [.0-8888-exec-37]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.748Z  INFO 1 --- [.0-8888-exec-44]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
```

```
2023-09-08T16:29:28.839Z  INFO 1 --- [.0-8888-exec-23]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.751Z  INFO 1 --- [0.0-8888-exec-7]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.752Z  INFO 1 --- [.0-8888-exec-43]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.847Z  INFO 1 --- [.0-8888-exec-34]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.847Z  INFO 1 --- [.0-8888-exec-38]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.848Z  INFO 1 --- [.0-8888-exec-33]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.749Z  INFO 1 --- [.0-8888-exec-12]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.544Z  INFO 1 --- [.0-8888-exec-27]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.544Z  INFO 1 --- [.0-8888-exec-11]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.551Z  INFO 1 --- [0.0-8888-exec-5]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.547Z  INFO 1 --- [.0-8888-exec-49]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.554Z  INFO 1 --- [.0-8888-exec-26]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.558Z  INFO 1 --- [.0-8888-exec-48]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.558Z  INFO 1 --- [.0-8888-exec-41]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
```

```
2023-09-08T16:29:33.563Z  INFO 1 --- [0.0-8888-exec-4]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.564Z  INFO 1 --- [.0-8888-exec-29]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.560Z  INFO 1 --- [.0-8888-exec-21]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.641Z  INFO 1 --- [.0-8888-exec-30]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.647Z  INFO 1 --- [.0-8888-exec-36]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.560Z  INFO 1 --- [0.0-8888-exec-6]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.643Z  INFO 1 --- [.0-8888-exec-14]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.653Z  INFO 1 --- [.0-8888-exec-15]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.659Z  INFO 1 --- [.0-8888-exec-31]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.751Z  INFO 1 --- [.0-8888-exec-26]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.754Z  INFO 1 --- [.0-8888-exec-50]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.754Z  INFO 1 --- [.0-8888-exec-17]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.666Z  INFO 1 --- [.0-8888-exec-40]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.755Z  INFO 1 --- [.0-8888-exec-35]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
```

```
2023-09-08T16:29:33.756Z  INFO 1 --- [.0-8888-exec-28]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.764Z  INFO 1 --- [0.0-8888-exec-8]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.841Z  INFO 1 --- [.0-8888-exec-19]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.845Z  INFO 1 --- [.0-8888-exec-18]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
```

U zapisima druge grupe kontejnera također je vidljivo točno 100 zahtjeva:

```
$ kubectl logs diplomski-todo-774b56d45-xjkmf


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.1.3)

2023-09-08T16:28:28.256Z  INFO 1 --- [           main]
↪  o.l.diplomski.todo.TodoApplication       : Starting
↪  TodoApplication v0.0.1-SNAPSHOT using Java 17.0.8 with PID 1
↪  (/opt/todo/app.jar started by root in /)
2023-09-08T16:28:28.339Z  INFO 1 --- [           main]
↪  o.l.diplomski.todo.TodoApplication       : No active profile
↪  set, falling back to 1 default profile: "default"
2023-09-08T16:28:36.255Z  INFO 1 --- [           main]
↪  o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized
↪  with port(s): 8888 (http)
2023-09-08T16:28:36.348Z  INFO 1 --- [           main]
↪  o.apache.catalina.core.StandardService   : Starting service
↪  [Tomcat]
2023-09-08T16:28:36.348Z  INFO 1 --- [           main]
↪  o.apache.catalina.core.StandardEngine    : Starting Servlet
↪  engine: [Apache Tomcat/10.1.12]
2023-09-08T16:28:37.162Z  INFO 1 --- [           main]
↪  o.a.c.c.C.[.[localhost].[/todo-api]      : Initializing Spring
↪  embedded WebApplicationContext
```

```
2023-09-08T16:28:37.163Z  INFO 1 --- [           main]
 ↪  w.s.c.ServletWebServerApplicationContext : Root
 ↪  WebApplicationContext: initialization completed in 8502 ms
2023-09-08T16:28:46.559Z  INFO 1 --- [           main]
 ↪  o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on
 ↪  port(s): 8888 (http) with context path '/todo-api'
2023-09-08T16:28:46.738Z  INFO 1 --- [           main]
 ↪  o.l.diplomski.todo.TodoApplication       : Started
 ↪  TodoApplication in 21.514 seconds (process running for 24.366)
2023-09-08T16:29:24.057Z  INFO 1 --- [.0-8888-exec-11]
 ↪  o.a.c.c.C.[.[localhost].[/todo-api]      : Initializing Spring
 ↪  DispatcherServlet 'dispatcherServlet'
2023-09-08T16:29:24.057Z  INFO 1 --- [.0-8888-exec-11]
 ↪  o.s.web.servlet.DispatcherServlet        : Initializing Servlet
 ↪  'dispatcherServlet'
2023-09-08T16:29:24.059Z  INFO 1 --- [.0-8888-exec-11]
 ↪  o.s.web.servlet.DispatcherServlet        : Completed
 ↪  initialization in 1 ms
2023-09-08T16:29:25.765Z  INFO 1 --- [.0-8888-exec-47]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  createTodo
2023-09-08T16:29:25.839Z  INFO 1 --- [.0-8888-exec-45]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  createTodo
2023-09-08T16:29:25.839Z  INFO 1 --- [.0-8888-exec-12]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  createTodo
2023-09-08T16:29:25.839Z  INFO 1 --- [0.0-8888-exec-4]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  createTodo
2023-09-08T16:29:25.840Z  INFO 1 --- [.0-8888-exec-41]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  createTodo
2023-09-08T16:29:25.840Z  INFO 1 --- [0.0-8888-exec-8]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  createTodo
2023-09-08T16:29:25.840Z  INFO 1 --- [.0-8888-exec-25]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  createTodo
2023-09-08T16:29:25.841Z  INFO 1 --- [.0-8888-exec-23]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  createTodo
```

```
2023-09-08T16:29:25.841Z  INFO 1 --- [.0-8888-exec-32]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.842Z  INFO 1 --- [.0-8888-exec-38]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.842Z  INFO 1 --- [.0-8888-exec-42]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.842Z  INFO 1 --- [0.0-8888-exec-3]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.842Z  INFO 1 --- [0.0-8888-exec-9]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.842Z  INFO 1 --- [0.0-8888-exec-5]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.842Z  INFO 1 --- [.0-8888-exec-19]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.845Z  INFO 1 --- [.0-8888-exec-24]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.846Z  INFO 1 --- [0.0-8888-exec-1]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.847Z  INFO 1 --- [.0-8888-exec-22]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.842Z  INFO 1 --- [.0-8888-exec-13]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.842Z  INFO 1 --- [.0-8888-exec-43]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.853Z  INFO 1 --- [.0-8888-exec-15]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.853Z  INFO 1 --- [.0-8888-exec-34]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
```

```
2023-09-08T16:29:25.853Z  INFO 1 --- [.0-8888-exec-39]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.852Z  INFO 1 --- [.0-8888-exec-48]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.854Z  INFO 1 --- [.0-8888-exec-17]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.854Z  INFO 1 --- [.0-8888-exec-14]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.854Z  INFO 1 --- [.0-8888-exec-30]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.856Z  INFO 1 --- [.0-8888-exec-33]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.765Z  INFO 1 --- [.0-8888-exec-37]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.849Z  INFO 1 --- [.0-8888-exec-29]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.849Z  INFO 1 --- [.0-8888-exec-36]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.849Z  INFO 1 --- [.0-8888-exec-50]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.851Z  INFO 1 --- [.0-8888-exec-10]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.851Z  INFO 1 --- [.0-8888-exec-27]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.851Z  INFO 1 --- [.0-8888-exec-31]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.851Z  INFO 1 --- [.0-8888-exec-49]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
```

```
2023-09-08T16:29:25.852Z  INFO 1 --- [0.0-8888-exec-2]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.852Z  INFO 1 --- [.0-8888-exec-16]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.852Z  INFO 1 --- [0.0-8888-exec-7]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.852Z  INFO 1 --- [.0-8888-exec-35]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.852Z  INFO 1 --- [.0-8888-exec-44]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.846Z  INFO 1 --- [.0-8888-exec-21]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.852Z  INFO 1 --- [.0-8888-exec-28]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.849Z  INFO 1 --- [.0-8888-exec-26]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.857Z  INFO 1 --- [.0-8888-exec-11]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.857Z  INFO 1 --- [.0-8888-exec-18]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.858Z  INFO 1 --- [0.0-8888-exec-6]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.858Z  INFO 1 --- [.0-8888-exec-46]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.859Z  INFO 1 --- [.0-8888-exec-20]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
2023-09-08T16:29:25.942Z  INFO 1 --- [.0-8888-exec-40]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  createTodo
```

```
2023-09-08T16:29:27.253Z  INFO 1 --- [.0-8888-exec-52]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:27.339Z  INFO 1 --- [.0-8888-exec-51]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:27.558Z  INFO 1 --- [.0-8888-exec-53]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:27.651Z  INFO 1 --- [.0-8888-exec-54]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:27.653Z  INFO 1 --- [.0-8888-exec-55]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.443Z  INFO 1 --- [.0-8888-exec-57]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.444Z  INFO 1 --- [.0-8888-exec-59]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.449Z  INFO 1 --- [.0-8888-exec-58]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.445Z  INFO 1 --- [.0-8888-exec-60]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.447Z  INFO 1 --- [.0-8888-exec-61]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.455Z  INFO 1 --- [.0-8888-exec-62]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.458Z  INFO 1 --- [.0-8888-exec-63]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.541Z  INFO 1 --- [.0-8888-exec-65]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.546Z  INFO 1 --- [.0-8888-exec-56]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
```

```
2023-09-08T16:29:28.547Z  INFO 1 --- [.0-8888-exec-64]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.742Z  INFO 1 --- [.0-8888-exec-66]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:28.752Z  INFO 1 --- [.0-8888-exec-67]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:29.143Z  INFO 1 --- [.0-8888-exec-68]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:29.241Z  INFO 1 --- [.0-8888-exec-70]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:29.243Z  INFO 1 --- [.0-8888-exec-69]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:30.643Z  INFO 1 --- [.0-8888-exec-71]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:30.647Z  INFO 1 --- [.0-8888-exec-72]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:30.648Z  INFO 1 --- [.0-8888-exec-73]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:30.649Z  INFO 1 --- [.0-8888-exec-74]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:30.741Z  INFO 1 --- [.0-8888-exec-75]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.555Z  INFO 1 --- [.0-8888-exec-57]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.556Z  INFO 1 --- [.0-8888-exec-64]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.561Z  INFO 1 --- [.0-8888-exec-58]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
```

```
2023-09-08T16:29:33.558Z  INFO 1 --- [.0-8888-exec-48]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.639Z  INFO 1 --- [.0-8888-exec-70]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.640Z  INFO 1 --- [.0-8888-exec-51]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.644Z  INFO 1 --- [.0-8888-exec-60]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.651Z  INFO 1 --- [.0-8888-exec-74]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.558Z  INFO 1 --- [.0-8888-exec-73]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.752Z  INFO 1 --- [.0-8888-exec-61]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.651Z  INFO 1 --- [.0-8888-exec-62]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.745Z  INFO 1 --- [.0-8888-exec-66]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.657Z  INFO 1 --- [.0-8888-exec-60]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.657Z  INFO 1 --- [.0-8888-exec-67]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.660Z  INFO 1 --- [.0-8888-exec-59]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.661Z  INFO 1 --- [.0-8888-exec-72]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
2023-09-08T16:29:33.662Z  INFO 1 --- [.0-8888-exec-46]
↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
↪  deleteTodo
```

```
2023-09-08T16:29:33.663Z  INFO 1 --- [.0-8888-exec-75]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  deleteTodo
2023-09-08T16:29:33.664Z  INFO 1 --- [.0-8888-exec-71]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  deleteTodo
2023-09-08T16:29:33.742Z  INFO 1 --- [.0-8888-exec-68]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  deleteTodo
2023-09-08T16:29:33.743Z  INFO 1 --- [.0-8888-exec-40]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  deleteTodo
2023-09-08T16:29:33.744Z  INFO 1 --- [.0-8888-exec-65]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  deleteTodo
2023-09-08T16:29:33.747Z  INFO 1 --- [.0-8888-exec-63]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  deleteTodo
2023-09-08T16:29:33.748Z  INFO 1 --- [.0-8888-exec-16]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  deleteTodo
2023-09-08T16:29:33.748Z  INFO 1 --- [.0-8888-exec-13]
 ↪  o.l.d.todo.controller.TodoController     : Primljen zahtjev
 ↪  deleteTodo
```

Na poslužiteljsku aplikaciju bilo je poslano točno 200 zahtjeva. U prvu grupu kontejnera pristiglo je 100 zahtjeva, a u drugu grupu kontejnera također je pristiglo 100 zahtjeva. Može se zaključiti da se promet ravnomjerno raspoređuje.

7. Zaključak

Nakon postavljanja Kubernetes klastera, aplikacija je konačno dostupna javnosti i zbrinuta je od strane Kubernetes platforme. Mrežni promet u aplikaciji ravnomjerno će se raspoređivati kroz grupe kontejnera u klasteru pomoću servisa i ingress kontrolera. Ako se jedna od grupa kontejnera sruši, Kubernetes će automatski pokrenuti novu grupu kontejnera i na taj način se postiže visoka dostupnost aplikacija unutar klastera.

U radu je demonstrirano i postavljanje Docker Swarm klastera. Postavljanje takvog klastera značajno je jednostavnije od postavljanja Kubernetes klastera, ali Docker Swarm sam po sebi nema toliko mogućnosti koliko ih ima Kubernetes i upravo je zato prikladniji za manje kompleksne aplikacije i sustave.

Postavljanje Docker Swarm klastera značajno olakšava i alat Ansible koji deklarativnim jezikom definira stanje koje mora biti zadovoljno na poslužitelju. U stanju je dovoljno detaljno definirati kako izgleda stanje poslužitelja koji na sebi ima instaliran Docker.

Kubernetes je platforma koja je uglavnom prikladna za velike i kompleksne aplikacije s naprednim zahtjevima. Kubernetes također nudi mnogo veću ekstenzibilnost i modularnost. Na primjer, u Kubernetesu je moguće birati mrežno sučelje, pokretač kontejnera, itd., dok to nije moguće koristeći Docker Swarm.

Nažalost, postavljanje Kubernetes klastera na lokalnim čvorovima dodatno otežava i činjenica da treba instalirati i upravljati svojim pružateljem LoadBalancer servisa. U ovom se radu za tu svrhu koristi MetalLB, a njega koriste i mnoge velike korporacije uz mnogo uspjeha.

Kada se klaster postavlja na nekom od upravljanih servisa kao što su Amazon EKS, Azure AKS, itd., tada je postavljanje značajno jednostavnije jer ne treba previše razmišljati o umrežavanju i postavljanju pružatelja LoadBalancer servisa, već je samo potrebno razmišljati o isporuci aplikacija i servisa na Kubernetes klaster. Upravo zato je velika većina isporuka Kubernetes aplikacija u današnje vrijeme pogonjena takvim upravljanim servisima.

Popis literature

[1] N. Poulton i P. Joglekar, *The Kubernetes Book*. Nigel Poulton, 2022., ISBN: 9798402153776. adresa: https://books.google.hr/books?id=zJExzwEACAAJ.

[2] Docker. „What is a Container? | Docker.” (2023.), adresa: https://web.archive.org/web/20230713183457/https://www.docker.com/resources/what-container/ (pogledano 13. 7. 2023.).

[3] J. Schmitt. „Docker image vs container: What are the differences?” (2023.), adresa: https://web.archive.org/web/20230713223335/https://circleci.com/blog/docker-image-vs-container/ (pogledano 13. 7. 2023.).

[4] J. Nickoloff i S. Kuenzli, *Docker in Action, Second Edition*. Manning, 2019., ISBN: 9781638351740. adresa: https://books.google.hr/books?id=qzozEAAAQBAJ.

[5] DockerHub. „busybox - Official Image | Docker Hub.” (2023.), adresa: https://web.archive.org/web/20230714133146/https://hub.docker.com/_/busybox (pogledano 14. 7. 2023.).

[6] J. Baier, *Getting Started with Kubernetes - Second Edition*. Packt Publishing, 2017., ISBN: 9781787283367. adresa: https://books.google.hr/books?id=eAbuswEACAAJ.

[7] RedHat. „Containers vs VMs.” (2020.), adresa: http://web.archive.org/web/20230714011605/https://www.redhat.com/en/topics/containers/containers-vs-vms (pogledano 13. 7. 2023.).

[8] D. Walsh, *Podman in Action: Secure, rootless containers for Kubernetes, microservices, and more*. Manning, 2023., ISBN: 9781638351832. adresa: https://books.google.hr/books?id=C16mEAAAQBAJ.

[9] Docker. „What is Docker Hub? | Docker.” (2023.), adresa: https://web.archive.org/web/20230715200449/https://www.docker.com/products/docker-hub/ (pogledano 15. 7. 2023.).

[10] M. Heap, *Ansible: From Beginner to Pro* (For professionals by professionals). Apress, 2016., ISBN: 9781484216590. adresa: https://books.google.hr/books?id=D-wmDQAAQBAJ.

[11] Oracle. „Networking Overview.” (2023.), adresa: https://web.archive.org/web/20230822152624/https://docs.oracle.com/en-us/iaas/Content/Network/Concepts/overview.htm (pogledano 22. 8. 2023.).

[12] V. Farcic, *The DevOps 2.3 Toolkit*. Packt Publishing, 2018., ISBN: 9781789135503. adresa: https://books.google.hr/books?id=LJvFuwEACAAJ.

[13] Swagger. „REST API Documentation Tool | Swagger UI.” (2023.), adresa: https://web.archive.org/web/20230831231716/https://swagger.io/tools/swagger-ui/ (pogledano 31. 8. 2023.).

[14] AWS. „IAM users - AWS Identity and Access Management.” (2023.), adresa: https://web.archive.org/web/20230908144505/https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html (pogledano 8. 9. 2023.).

Popis slika

Popis tablica

1. Prilozi

- https://github.com/leonardogazdek/diplomski-todo-app - izvorni kod poslužiteljske aplikacije
- https://github.com/leonardogazdek/diplomski-todo-app-frontend - izvorni kod korisničke aplikacije
- https://github.com/leonardogazdek/diplomski-todo-kubernetes-manifest - Kubernetes datoteke manifestiranja
- https://github.com/leonardogazdek/diplomski-todo-loadbalance-test - skripta za demonstriranje raspoređivanja opterećenja
- http://141.147.10.108.nip.io/ - adresa na kojoj je moguće isprobati isporučenu aplikaciju