Izvorne Java aplikacije i arhitekture za oblak

Turković, Fran

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fran Turković

Izvorne Java aplikacije i arhitekture za oblak

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fran Turković

Matični broj: 46221/18–R

Studij: Baze podataka i baze znanja

Izvorne Java aplikacije i arhitekture za oblak

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, rujan 2024.

Fran Turković

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/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Tema ovog rada su izvorne Java aplikacije i arhitekture za oblak. Rad se sastoji od teorijskog i praktičnog dijela. Praktični dio je izvorna Java aplikacija za oblak napravljena pomoću modernih tehnologija, a teorijski je detaljno objašnjenje implementacije izvornih aplikacija za oblak te dobrih i loših praksa u procesu odluka i razvoja aplikacija. U radu su objašnjene sinkrone i asinkrone tehnike razvoja izvornih aplikacija za oblak te sama metodologija. Rad se fokusira na komunikaciji, odlukama izbora spremišta, automatizaciji, asinkronosti cijelog sustava i mikro servisnoj arhitekturi. Da bi se sve to realno prikazalo, napravljena je aplikacija za upoznavanje ljudi online pomoću Jakarta EE skupa specifikacija, Payara Micro Community poslužitelja, RabbitMQ posrednika poruka, MySQL baze podataka, Docker stroja za izradu slika i Kubernetes orkestratora kontejnera. Aplikacija je podijeljena u 5 različitih servisa od kojih svaki obavlja svoju specifičnu odgovornost: autentifikacijski servis, geolokacijski servis, servis čavrljanja, servis obavijesti i korisnički servis. Kako bi se servisi ujedinili u smislenu cjelinu napravljen je API pristupnik. Da bi se sve to prikazalo postoji i aplikacija korisničke strane. Arhitektura sustava prati najbolje prakse kako bi se ostvarila sva načela „Twelve-Factor“ metodologije.

Ključne riječi: Jakarta EE, mikro servisi, oblak, web, Java

Sadržaj

1. Uvod

Izvorne aplikacije za oblak danas sve više i više postaju standard pravilne arhitekture. Mnoge tvrtke prelaze s tradicionalnih monolitnih arhitektura na mikro servisnu, koja je osnova izvornih aplikacija za oblak, kada dođu do problema skaliranja monolitnih aplikacija [1]. Korisnički zahtjevi postaju sve veći, korisnici traže sve više i više značajki, porastom popularnosti i dostupnosti interneta aplikacije imaju porast broja korisnika i moraju obuhvatiti veće geografske lokacije. Svi ti problemi doveli su do velikih problema skaliranja i dostupnosti monolitnih aplikacija koje su centralizirane [1]. Kako ih skalirati i pokriti velik broj korisnika i geografskih lokacija? Kako skalirati samo određene dijelove aplikacija? Kako sinkronizirati velike aplikacije na kojima radi velik broj timova? Programeri su došli na ideju razbijanja aplikacije na manje dijelove, svaki sa svojom određenom odgovornošću, koji međusobno komuniciraju kako bi zajedno činili veliki sustav koji predstavlja aplikacija. To je upravo ono što mikro servisna arhitektura radi. Razdvajanjem aplikacije na manje smislene dijelove, arhitektura je riješila probleme skalabilnosti i dostupnosti aplikacije, no sada su se stvorili novi problemi. Kako implementirati komunikaciju između tih mikro servisa? Kako održati dosljednost podataka u cijelom sustavu? Kako napraviti ne blokirajući sustav? Kako korisnička strana sada komunicira sa serverskom? U ovom radu objašnjene su sve prepreke kod razvoja izvornih aplikacija za oblak i odgovoreno je na ova pitanja. Objašnjene su metodologije i najbolje i loše prakse koje uvelike mogu utjecati na performanse servisa i sustava u cjelini. Kako sve to ne bi ostalo samo na teoriji, napravljena je i aplikacija koja implementira mikro servisnu arhitekturu. Aplikacija prati „Twelve-Factor“ metodologiju, koja je današnji standard za razvoj izvornih aplikacija za oblak.

2. Izvorne aplikacije i arhitekture za oblak

Izvorne aplikacije za oblak su aplikacije koje su izgrađene putem arhitekture za oblak i budućnost su izgrađivanja skalabilnih aplikacija. Izvorne arhitekture i tehnologije za oblak su načela dizajniranja, konstruiranja i upravljanja radnim područjima koja su ugrađena u oblak i iskorištavaju sve prednosti modela računalstva u oblaku, poput skalabilnosti, razdvojenih odgovornosti servisa, velike dostupnosti i konstantne isporuke. Projekt „Cloud Native Computing Foundation“ daje svoju službenu definiciju:

„Izvorne tehnologije za oblak potiču organizacije da grade i isporučuju skalabilne aplikacije u modernim i dinamičkim okružjima kao što su javni, privatni i hibridni oblaci. Kontejneri, mreže servisa, mikro servisi, nepromjenjive infrastrukture i deklarativni API-i primjeri su ovog pristupa“. [2]

Iz gore navedenih objašnjenja može se zaključiti da su izvorne aplikacije za oblak zapravo aplikacije koje se sastoje od više servisa od kojih je svaki samostalan i radi na zasebnom sustavu te ti svi servisi putem određenog protokola vrše komunikaciju kako bi zadovoljili korisničke zahtjeve. Temelj ovog pristupa su brzina i agilnost. Poslovni sustavi se konstantno razvijaju, kreću od osnovnih poslovnih zahtjeva do strateških transformacija koje uvelike ubrzavaju brzinu i rast poslovanja. Neophodno je da aplikacija prati poslovni rast. U isto vrijeme, poslovne aplikacije postale su znatno kompleksnije kao i korisnički zahtjevi. Očekuje se brz odaziv, inovativne značajke i konstantan rad sustava 24/7. Problemi performansi, ponavljajuće greške i nesposobnost brzog otklanjanja problema se više ne toleriraju. Izvorne aplikacije i arhitekture za oblak su zbog ovih zahtjeva i dizajnirane kako bi pružile brze i lake promjene i nadogradnje sustava, veliku mogućnost skaliranja te elastičnost sustava.

Neke od vrlo popularnih tvrtki poput Netfilx-a, Uber-a i WeChat-a koriste baš ovaj pristup pri izgradnji aplikacije. Poznato je da Netflix koristi oko šesto servisa u produkciji i da isporučuje nove verzije oko sto puta na dan, Uber ima oko tisuću servisa i obavlja isporuku nekoliko tisuća puta na tjedan te WeChat ima oko tri tisuće servisa i obavlja isporuku oko tisuću puta na dan. Vidi se koliko su zapravo velike te aplikacije i koliko različitih servisa se treba izgrađivati, testirati i konstanto verzionirati kako bi cijeli sustav ispravno radio bez ikakvog vremenskog zastoja. Stari monolitni pristup izgradnji aplikacija, koje su ovolikih razmjera,

gotovo je nemoguć. Vidi se da je njihov pristup izgradnja velikog sustava od puno neovisnih servisa. Ovaj pristup im omogućuje da brzo odgovaraju na zahtjeve tržišta. Time što su servisi neovisni i slabo povezani omogućava se lako dodavanje novih i brisanje starih servisa. Isto tako dobivaju puno i kod isporuke aplikacije, budući da ne moraju konstanto isporučivati cijelu aplikaciju, već konstantno nadograđuju male dijelove sustava. Kako bi skalirali sustav, identificiraju samo one servise koje je potrebno skalirati i obave operacije skaliranja samo nad njima. [3]

Postoji šest glavnih principa izgradnje izvornih aplikacija za oblak:

1. Oblak
2. Moderni dizajn
3. Mikro servisi
4. Kontejneri
5. Usluge podrške
6. Automatizacija

2.1. Oblak

Oblak je fizičko računalo koje netko pruža na korištenje putem Interneta. Oblak može biti bilo koje računalo (stolno, laptop) koje ima pristup Internetu i koje je izloženo putem određenih internetskih protokola korisnicima na korištenje. Dizajniran da postoji u dinamičkom i virtualiziranom online okružju, oblak koristi računalnu infrastrukturu platforme kao usluge (eng. *Platform as a Service*). Koristeći široko prihvaćen DevOps koncept tretiranja servera „Kućni ljubimci protiv stoke“, može se reći da je glavni koncept tretiranja servera u oblaku poput stoke. U tradicionalnim centrima podataka, serveri su tretirani kao kućni ljubimci: fizička računala, koja imaju svoje ime, i brine se o njima kao i o kućnim ljubimcima. Skaliranje se obavlja na način da se doda još resursa tim računalima (skaliranje prema gore). Ako se server „razboli“, radimo sve da on „ozdravi“. Ako neki od servera ne radi, korisnici to mogu primijetiti. Tretiranje servera poput stoke je drugačije. Svaka instanca je virtualno računalo ili kontejner. Oni su identični i dobivaju sistemsko ime poput: „servis-1“, „servis-2“, „servis-3“ i tako dalje. Skaliranje se obavlja na način da se dodaje još instanci (skaliranje prema van). Kada jedna od instanci više ne radi, korisnici to ne mogu primijetiti. Model stoke implementira nepromjenjivu infrastrukturu što znači da se serveri ne popravljaju i ne modificiraju. Ako jedan padne ili ga je potrebno modificirati, on se uništava i na njegovo mjesto dolazi novi. Cijeli proces je omogućen

automatizacijom. Izvorne aplikacije za oblak implementiraju tretiranje servera poput stoke. Servisi su pokrenuti iako se infrastruktura skalira i mijenja bez obzira na kojem su računalu pokrenuti. Platforme poput Azure-a omogućavaju ovakve elastične infrastrukture koje imaju mogućnosti automatskog skaliranja, samopopravljanja i konstantnog praćenja stanja.

2.2. Moderni dizajn

Postavljaju se pitanja kako dizajnirati izvorne aplikacije za oblak, kako im izgleda arhitektura, koje principe, uzorke i najbolje prakse koristiti. Široko prihvaćena metodologija za izgradnju izvornih aplikacija za oblak je „Twelve-Factor App“. Ona objašnjava skup principa i praksi koje programeri trebaju pratiti kako bi izgradili aplikacije optimizirane za moderna okružja u oblaku. Posebno se gleda prenosivost između različitih okružja i deklarativna automatizacija. Iako je metodologija primjenjiva na bilo kakvu web aplikaciju, mnogi stručnjaci tvrde da je „Twelve-Factor“ metodologija solidan temelj za izgradnju izvornih aplikacija za oblak. Sistemi koji koriste ovu metodologiju imaju brzo isporučivanje i skaliranje te dodavanje novih značajki kako bi odgovorili na zahtjeve tržišta i klijenata.

Sljedeća tablica objašnjava „Twelve-Factor“ metodologiju [4]:

Tablica 1. "Twelve-Factor" metodologija

Faktor	Objašnjenje
1- Baza koda	Jedinstvena baza koda za svaki mikro servis, pohranjena u svoj repozitorij. Praćena sustavom za verzioniranje koda, te s mogućnosti postavljanja na različita okružja.
2- Ovisnosti	Svaki mikro servis izolira i pakira samo svoje ovisnosti, zagovara mijenjanje servisa bez utjecaja na cijeli sustav.
3- Konfiguracije	Konfiguracije se izdvajaju iz mikro servisa i eksternaliziraju kroz alat za upravljanje konfiguracijama izvan samog koda. Tako se identična isporuka može propagirati kroz sva okružja s ispravnom konfiguracijom.

4- Usluge podrške	Pomoćni resursi (skladišta podataka, predmemorija, posrednici poruka) trebaju biti izloženi putem adresabilnog URL-a. Time se odvaja resurs od aplikacije što omogućuje da budu zamjenjivi.
5- Izgradi, izdaj, pokreni	Svako izdanje treba imati jasnu podjelu između faza izgradnje, izdaje i pokretanja. Svako izdanje bi trebalo biti identificirano s jedinstvenim ID-em i imati mogućnost vraćanja na staro. Moderni CI/CD sustavi omogućavaju ovaj princip.
6- Procesi	Svaki mikro servis treba biti pokrenut u svojem procesu, izoliran od ostalih pokrenutih servisa. Stanje aplikacije (podaci) su eksternalizirani u neki servis za pohranu podataka (baza podataka, na primjer)
7- Vezivanje portova	Svaki mikro servis treba biti samostalan sa svojim sučeljima i funkcionalnostima izloženim prema svojem priključku (eng. *port*). Ovime se dobiva izolacija od ostalih servisa.
8- Konkurencija	Kada se dogodi potreba za skaliranjem aplikacije, skaliranje prema van (horizontalno) preko više identičnih procesa (kopija, replika) je bolja opcija od skaliranja prema gore (vertikalno). Izgradite aplikaciju da bude konkurentna tako da se može skalirati horizontalno u okružju oblaka, tj. da je moguće pokrenuti više instanca aplikacije istovremeno.
9- Zamjenjivost	Instance aplikacije trebaju biti zamjenjive. Mogućnost brzog pokretanja za mogućnost povećavanja skalabilnosti i elegantnog gašenja kako bi sistem ostao u ispravnom stanju. Docker kontejneri zajedno s orkestratorom inherentno ispunjavaju ovaj princip.
10- Razvojni/Produkcijski paritet	Što manja razlika između razvojnog i produkcijskog okruženja. Iskorištavanje kontejnera može uvelike pridonijeti tako što promovira isto izvršno okružje.

11- Dnevnik	Izrada dnevnika od svih događaja koji su se dogodili u mikro servisu (eng. *logs*). Obrađivanje agregatorom događaja.
12- Administratorski procesi	Pokretanje administratorskih/upravljačkih procesa, poput čišćenja podataka ili pokretanja migracija, kao jednokratnih procesa. Korištenje neovisnih alata za pokretanje ovih procesa u produkcijskom okruženju, posebno od procesa aplikacije.

2.3. Mikro servisi

Izvorna arhitektura za oblak implementira mikro servisnu arhitekturu kako bi ispunila principe koji objedinjuju dizajn izvorne aplikacije za oblak. Izgrađeni kao distribuirani skup malih, neovisnih servisa koji imaju međusobnu interakciju, mikro servisi imaju sljedeće karakteristike:

- Svaki servis implementira specifičnu poslovnu logiku unutar većeg konteksta domene
- Svaki servis je izgrađen autonomno i može biti postavljen samostalno
- Svaki servis je samostalan i učahuruje tehnologiju spremanja podataka, ovisnosti i platformu u kojoj je programiran
- Svaki servis je pokrenut u svom procesu i komunicira s ostalima putem osnovnih protokola komunikacije poput HTTP/HTTPS-a, RPC-a, WebSocket-a ili AMQP-a
- Zajedno sastavljeni formiraju aplikaciju

Mikro servisna arhitektura je nadogradnja na monolitnu. Monolitna arhitektura je napravljena od slojeva podataka, servisa i web-a koji svi rade u istom procesu. Obično koristi jednu relacijsku bazu podataka. Mikro servisi, nasuprot monolitnoj arhitekturi, odvajaju funkcionalnosti u samostalne servise, svaki sa svojom poslovnom logikom, stanjem i podacima. Svaki mikro servis ima svoje skladište podataka. Ova arhitektura pruža agilnost. Svaki mikro servis ima autonomni životni ciklus i može rasti neovisno i često se može isporučiti nova verzija. Ne mora se čekati kvartalno ili polugodišnje izdanje kako bi se mogle postaviti nove značajke ili ažuriranja. Može se ažurirati mali dio produkcijske aplikacije s puno manje rizika da će se cijeli sistem srušiti. To ažuriranje se može napraviti bez ponovne isporuke cijele aplikacije na poslužitelj. Svaki mikro servis se može skalirati neovisno o ostalima. Umjesto da se cijeli sustav skalira kao jedna jedinica, skaliraju se samo oni servisi koji trebaju više

procesorske snage ili radne memorije da bi zadovoljili nivo performansi i ostale slične zahtjeve. Ovakvo skaliranje pruža veću kontrolu nad sustavom i smanjuje troškove ažuriranja s obzirom da se skaliraju dijelovi aplikacije, a ne cijeli sustav.

Mikro servisi se mogu izgraditi pomoću svake moderne programske platforme (.Net, Jakarta, Node.js itd.). Iako mikro servisne arhitekture pružaju veliko povećanje agilnosti i brzine, one svejedno predstavljaju pojedine prepreke poput komunikacije, distribuiranih podataka unutar mikro servisnog sustava i sigurnosti.

2.3.1. Komunikacija

Kako će aplikacija korisničke strane razgovarati s mikro servisima serverske strane? Hoće li biti dopuštena direktna komunikacija ili će se napraviti apstrakcija poput API pristupnika i fasade koja pruža fleksibilnost, kontrolu i sigurnost? Kako će servisi razgovarati jedni s drugima? Hoće li se koristiti direktna HTTP komunikacija između servisa ili će servisi razgovarati preko posrednika poruka i redova poruka?

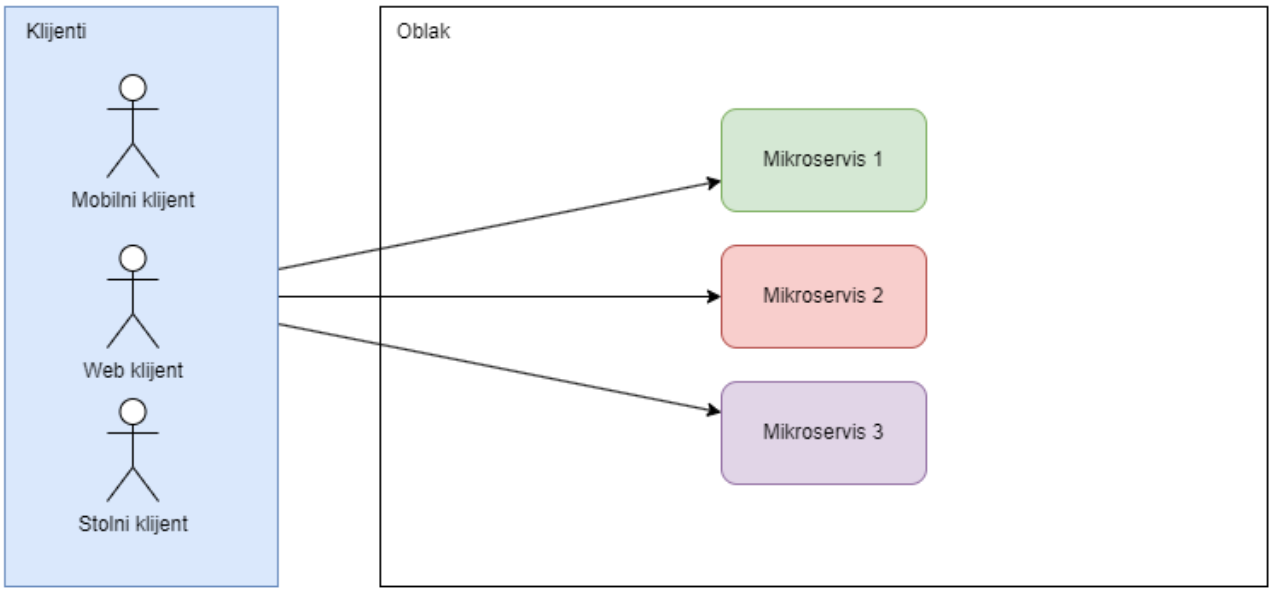
Odgovori na ova pitanja u monolitnim aplikacijama su jednostavni. Moduli kodova se izvršavaju zajedno unutar istog izvršnog prostora na serveru. Ovaj pristup može imati bolje performanse jer se sve izvršava zajedno u zajedničkoj memoriji, ali rezultira u usko vezanoj strukturi koda koju je teško održavati, razvijati i skalirati. Mikro servisna arhitektura rješava ove probleme razdvajanjem koda u zasebne neovisne servise od koji svaki radi u svom okružju u svom virtualnom računalu koji su upravljani od strane orkestralnog alata (npr. Kubernetes-a), no sada nastaje problem vanjske i unutarnje komunikacije između servisa. Servisi moraju komunicirati putem nekog mrežnog protokola, što dodaje kompleksnost sustavu:

- Zagušenja mreže, latencija i greške
- Otpornost je obavezna (npr. ponavljanje neuspješnih zahtjeva)
- Neki zahtjevi trebaju biti idempotentni*
- Svaki mikro servis mora imati implementiranu autorizaciju i autentifikaciju
- Svaka poruka mora biti serijalizirana i deserijalizirana što može biti skupo u vidu korištenja procesora i radne memorije
- Enkripcija i dekripcija poruka postaje važna

*idempotencija – poziv određene operacije uvijek vraća isti očekivani rezultat

2.3.1.1. Direktna komunikacija

U izvornoj arhitekturi za oblak, aplikacije korisničke strane (mobilne, web i stolne aplikacije) zahtijevaju komunikacijski kanal do mikro servisa serverske strane. Da bi stvari bile jednostavne, klijenti korisničke strane mogu direktno razgovarati sa servisima serverske strane.



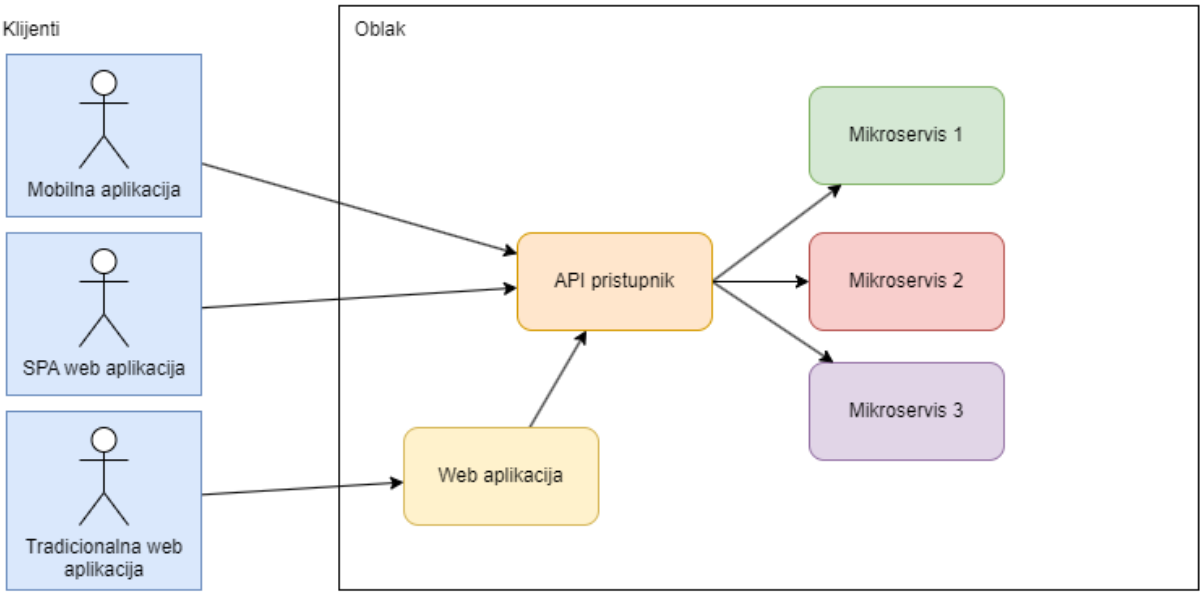
Slika 1. Direktna komunikacija između klijentske i serverske strane

U ovom pristupu svaki mikro servis ima javnu krajnju točku kojoj mogu pristupiti klijenti. U produkciji se postavi usmjerivač opterećenja (eng. *load balancer*) ispred mikro servisa koji podjednako preusmjerava promet. Iako je jednostavna za implementaciju, direktna klijentska komunikacija je dopustiva samo za jednostavne mikro servisne aplikacije. Ovaj pristup usko veže korisničku stranu s mikro servisima serverske strane što stvara dodane probleme poput:

- Klijenti su osjetljivi na promjene na serverskoj strani
- Širi prostor hakerskih napada kada su servisi serverske strane direktno izloženi
- Dupliciranje međusektorskih problema u svakom mikro servisu (autentifikacija, predmemoriranje, zagušivanje zbog prevelikog prometa)
- Previše kompleksan klijentski kod – klijet mora pratiti više krajnjih točaka i posebno upravljati greškama za svaku

2.3.1.2. API pristupnik

Da bi se problemi direktne komunikacije izbjegli, široko prihvaćen uzorak dizajna je da se implementira API pristupnik (eng. *API gateway*) kao fasada između klijenata korisničke strane i servisa serverske strane.

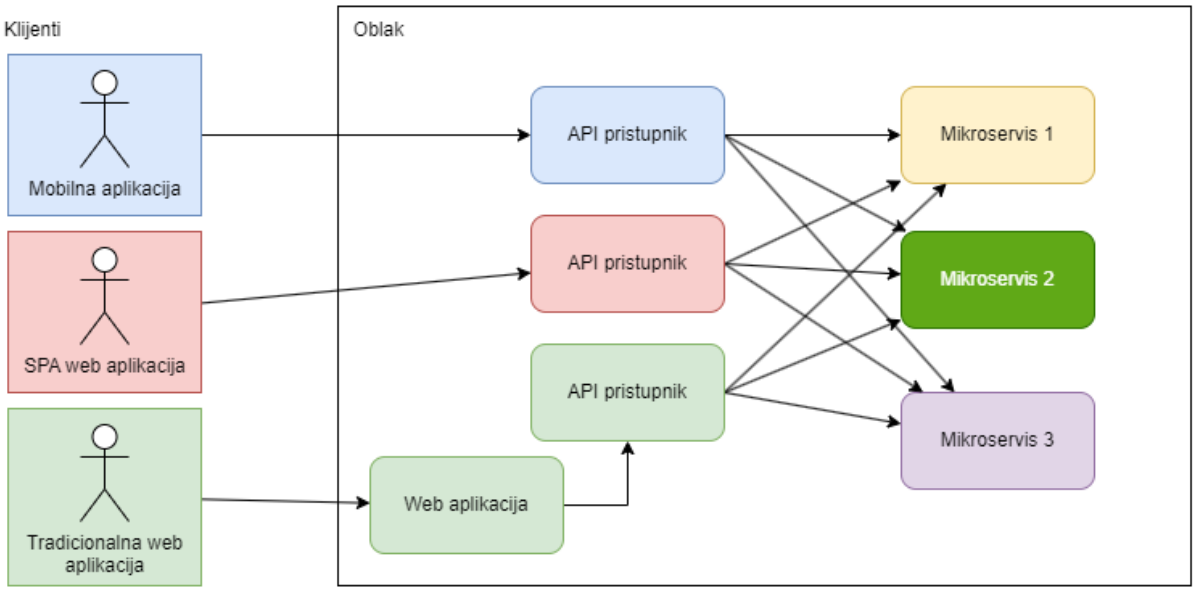


Slika 2. Komunikacija preko API pristupnika

Vidi se kako API pristupnik radi apstrakciju nad servisima serverske strane. Implementiran kao web API, ponaša se kao obrnuti posrednik (eng. *reverse proxy*), tako što preusmjerava promet na unutarnje mikro servise. API pristupnik izolira klijenta od unutarnjih mikro servisa, što znači da se skaliranje servisa može obaviti neovisno o API pristupniku i neke promjene implementacije servisa ne utječu na API pristupnik. Ako se promijeni neki od mikro servisa i to će utjecati na implementaciju API pristupnika, prilagodbe se naprave na API pristupniku i time se ne utječe na klijenta. Isto tako je prva crta obrane za međusektorske probleme poput autentifikacije, predmemorije, otpornosti, mjerenja prometa i zagušivanja zbog prevelikog prometa. Mnogi od ovih problema se mogu prebaciti na API pristupnik i tako osloboditi teret s mikro servisa. Dalje treba paziti da pristupnik bude brz i jednostavan. Obično se poslovna logika ne odvija u pristupniku. Kompleksan pristupnik stvara veliku šansu da će se desiti gušenja prometa i time postati monolit sam po sebi.

2.3.1.3. Backend for Frontend

Veliki sustavi obično imaju više različitih API pristupnika podijeljenih prema vrsti klijenta (mobilni, web, stolni) ili prema funkcionalnostima serverske strane. „Backend for Frontend“ uzorak pruža način kako implementirati ovaj pristup:



Slika 3. "Backend for Frontend" uzorak

Vidi se da je dolazni promet poslan specifičnom API pristupniku baziranom na vrsti klijentske aplikacije. Ovaj pristup ima smisla kada zahtjevi i mogućnosti elektroničkih uređaja dosta variraju, na primjer performanse i prikaz na ekrane različitih veličina. Obično mobilne aplikacije imaju manje zahtjeve od web i stolnih. Tako da se može svaki pristupnik optimizirati da direktno odgovara na zahtjeve uređaja za koji je implementiran.

2.3.1.4. Komunikacija u stvarnom vremenu

Zadnji od principa komunikacije između klijentske i serverske strane je komunikacija u stvarnom vremenu. Komunikacija u stvarnom vremenu ili komunikacija guranja poruka (eng. *push communicatio*n) obično se odvijaju preko HTTP protokola. Mnoge aplikacije zahtijevaju ažuriranje sadržaja u stvarnom vremenu, posebno aplikacije poput društvenih mreža, online kladionica, komunikacijske aplikacije itd. Sa standardnom HTTP komunikacijom klijent nikako ne može znati kada se ažuriralo stanje na servisima serverske strane. Klijent mora konstantno ispitivati servise serverske strane da li ima promjena. Da bi se ti problemi izbjegli, uvodi se koncept komunikacije u stvarnom vremenu u kojoj server može direktno dostaviti promjene

koje se dogode u servisima serverske strane. Sustavi stvarnog vremena su karakterizirani kao sustavi s velikim protokom podataka i velikim brojem usporednih veza. Tehnologije poput WebSocket-a i Server-Side Event-a se implementiraju kako bi se ostvarila komunikacija u stvarnom vremenu.

Dalje treba odgovoriti na pitanja komunikacije između samih mikro servisa. Komunikacija između servisa je dosta osjetljiva i može dovesti do velikih zagušenja. Dobar princip je napraviti što je manje moguće komunikacije. Svejedno, izbjegavanje komunikacije nije uvijek moguće jer neki servisi ovise jedni o drugima kod izvršavanja zahtjeva. Postoji više široko prihvaćenih principa komunikacije između servisa, a onaj koji se primjenjuje ovisi o vrsti komunikacije:

- Upit – kada zahtjev na drugi mikro servis zahtjeva i odgovor, na primjer, „Dohvati mi sve transakcije ovog korisnika“
- Komanda – kada poziv jednog servisa na drugi zahtijeva izvršavanje neke komande, ali ne treba odgovor, na primjer, „Pošalji ovu narudžbu“
- Događaj – kada mikro servis, koji je izdavač, izdaje događaj da se stanje promijenilo ili se neka akcija dogodila. Ostali mikro servisi, koji su pretplatnici, mogu reagirati na taj događaj. Izdavač i pretplatnik međusobno ne znaju za sebe.

Mikro servisna arhitektura često koristi kombinaciju ovih vrsta komunikacija, posebno kada se izvršavaju naredbe koje se provlače kroz više servisa.

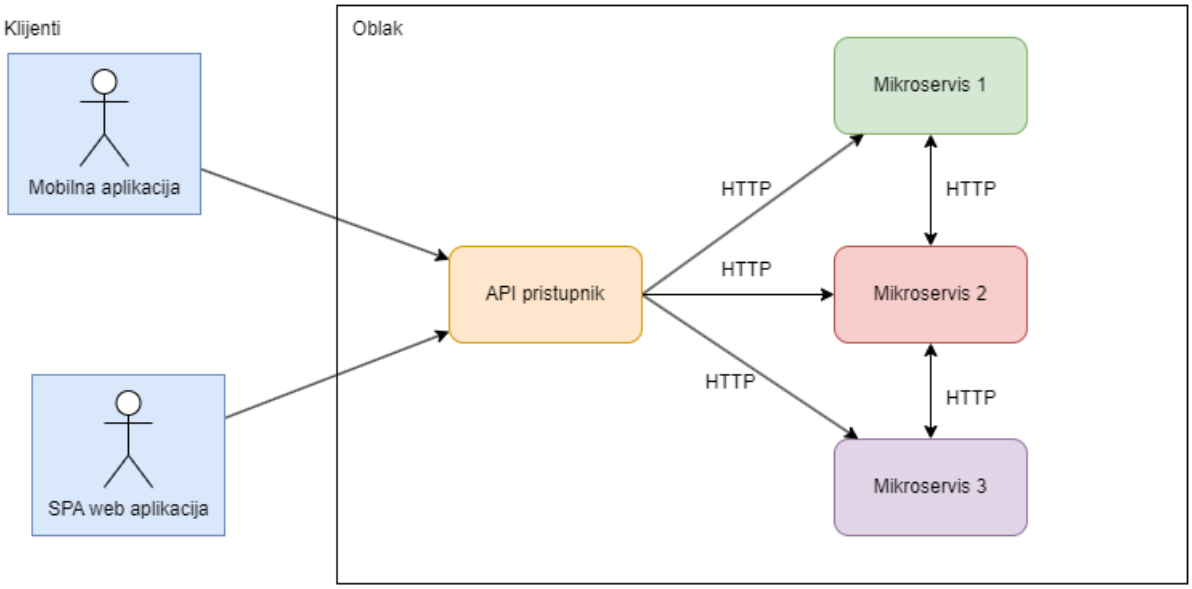
2.3.1.5. Upiti

Često se događa da jedan mikro servis mora poslati upit nekog drugom, zahtijevajući neposredan odgovor kako bi završio operaciju. Postoji više načina implementacije upita između mikro servisa:

- Komunikacija zahtjev/odgovor
- Uzorak materijaliziranog pogleda
- Uzorak agregatora servisa
- Komunikacija zahtjev/odgovor s redom

Komunikacija zahtjev/odgovor

Jedan od načina kako ostvariti ovu komunikaciju je da servis pošalje direktan HTTP zahtjev drugom servisu:



Slika 4. HTTP komunikacija zahtjev/odgovor

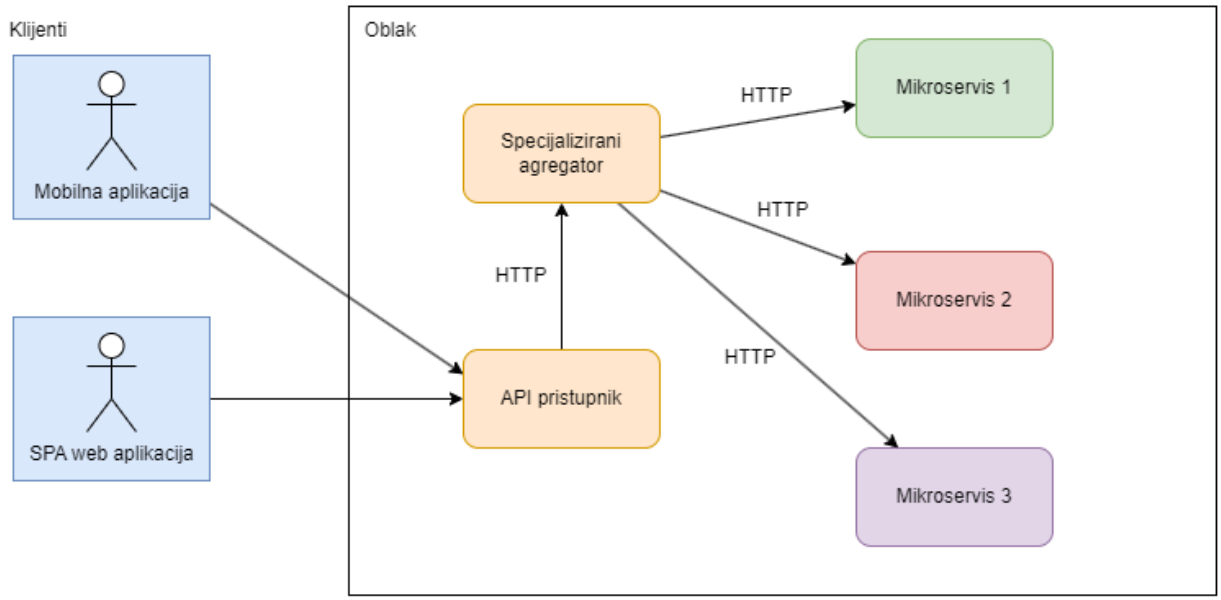
Dok je ovu vrstu komunikacije vrlo jednostavno implementirati, treba voditi računa da se što manje koristi. Kao prvo, ovakvi pozivi su sinkroni i blokiraju operacije sve dok se rezultat ne vrati od drugog servisa ili zahtjev istekne. Servise koji su nekada bili samostalni, neovisni, koji mogu samostalno evoluirati i isporučivati se, sada postaju ovisni jedni o drugima. Kako ovisnost između servisa raste, arhitekturne prednosti se smanjuju. Izvršavanje nefrekventnog zahtjeva koji obavlja jedan HTTP poziv nekad može biti prihvatljivo u nekim sustavima. Ako se pogleda sustav poput Netflix-a sa šesto i više servisa, ovakvi HTTP pozivi, posebno ako je više servisa potrebno za obavljanje operacije, su apsolutno neprihvatljivi. Oni mogu povećati latenciju i time negativno utjecati na performanse, skalabilnost i dostupnost sistema. U najgorem slučaju, duže serije HTTP zahtjeva mogu dovesti do dubokih i kompleksnih lanaca sinkronih poziva koji uvelike guše sustav.

Uzorak materijaliziranog pogleda

Jedan od popularnih načina kako razbiti ovisnosti mikro servisa. U ovom uzorku, mikro servis sprema svoju lokalnu i denormaliziranu kopiju podataka nekog drugog servisa. Bez da ispituje drugi servis da mu dohvati podatke, direktno ima svoju kopiju tih podataka drugog servisa. Ovime se razbija nepotrebna ovisnost servisa i dobiva se na pouzdanosti i vremenu odaziva sustava. Uzorak je dalje objašnjen u poglavlju 2.3.3.1 Upiti kroz više servisa.

Uzorak agregatora servisa

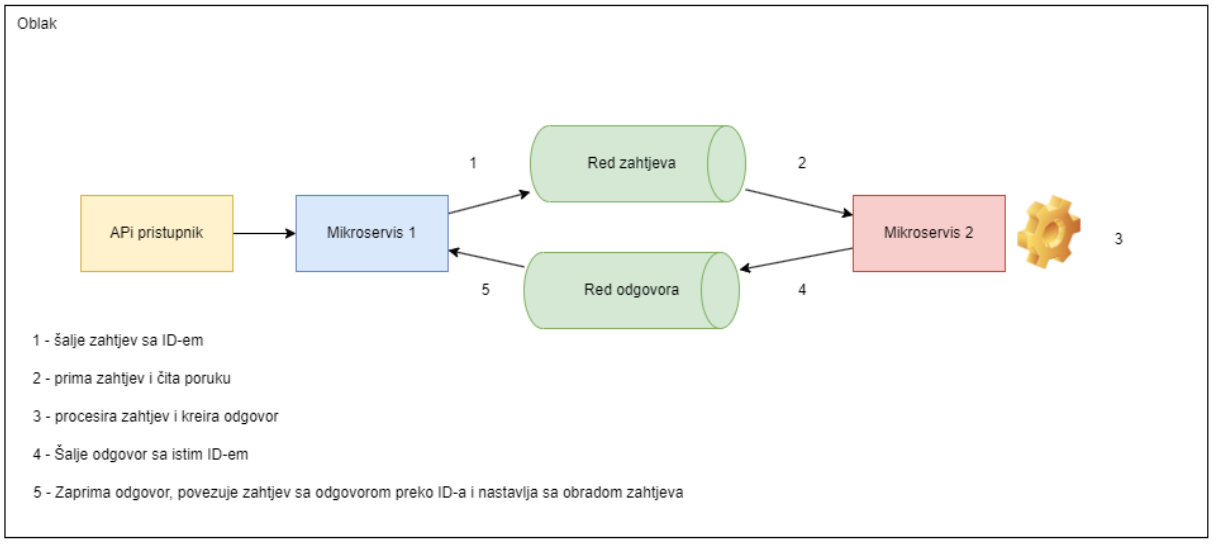
Ovaj uzorak izolira operaciju koja zahtijeva više servisa tako što centralizira logiku u jedan specijalizirani mikro servis. Taj specijalizirani servis, tj. agregator, se ponaša kao orkestrator koji upravlja tijekom operacija. Problem je što se mora napraviti specijalizirani agregator posebno za svaku funkciju koja poziva više servisa. Taj agregator poziva sve potrebne servise redoslijedno i agregira podatke svih poziva te vraća odgovor klijentu. Iako koristi direktnu HTTP komunikaciju, svejedno razbija ovisnosti između servisa.



Slika 5. Uzorak agregatora servisa

Komunikacija zahtjev/odgovor s redom

Još jedan od način kako razbiti ovisnosti sinkronih HTTP zahtjeva je komunikacija zahtjev/odgovor ali s korištenjem mehanizma reda. Komunikacija korištenjem reda je uvijek jednosmjerna, u kojoj pošiljatelj šalje poruke, a primatelj ih prima. U ovom uzorku, implementirana su oba reda, i red HTTP zahtjeva i red HTTP odgovora:

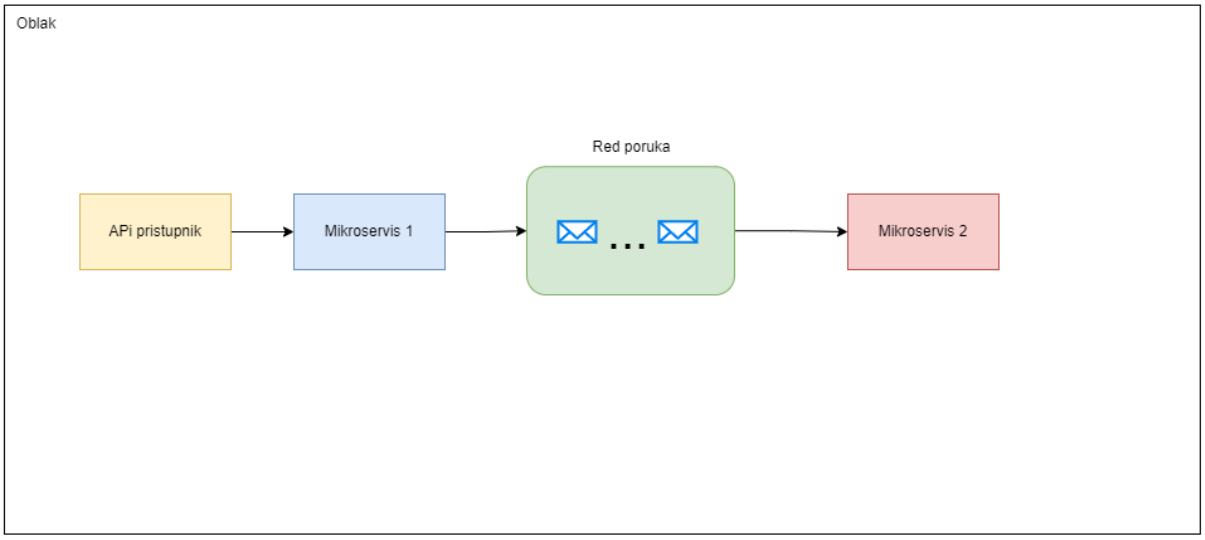


Slika 6. Komunikacija zahtjev/odgovor s redom poruka

Vidi se da pošiljatelj šalje HTTP zahtjev s upitom i korelacijskim jedinstvenim ID-om prema redu zahtjeva. Primatelj preuzima zahtjev iz reda, čita poruku, procesira zahtjev i šalje odgovor s istim korelacijskim ID-em u red odgovora. Pošiljatelj preuzima odgovor iz reda odgovora, pronalazi prvobitni zahtjev prema ID-u i nastavlja ga procesirati.

2.3.1.6. Komande

Mikro servis često treba neki drugi mikro servis da obavi određenu operaciju. Jedan servis, pošiljatelj, šalje poruku drugom servisu, primatelju, naređujući mu da odradi nešto. Vrlo često pošiljatelj ne zahtijeva odgovor, već može poslati komandu i zaboraviti na nju. Ukoliko je odgovor potreban, primatelj šalje odgovor kao posebnu poruku pošiljatelju na drugom kanalu. Najbolja praksa je da se komanda šalje asinkrono u neki red poruka, podržanog od nekog jednostavnog posrednika poruka.

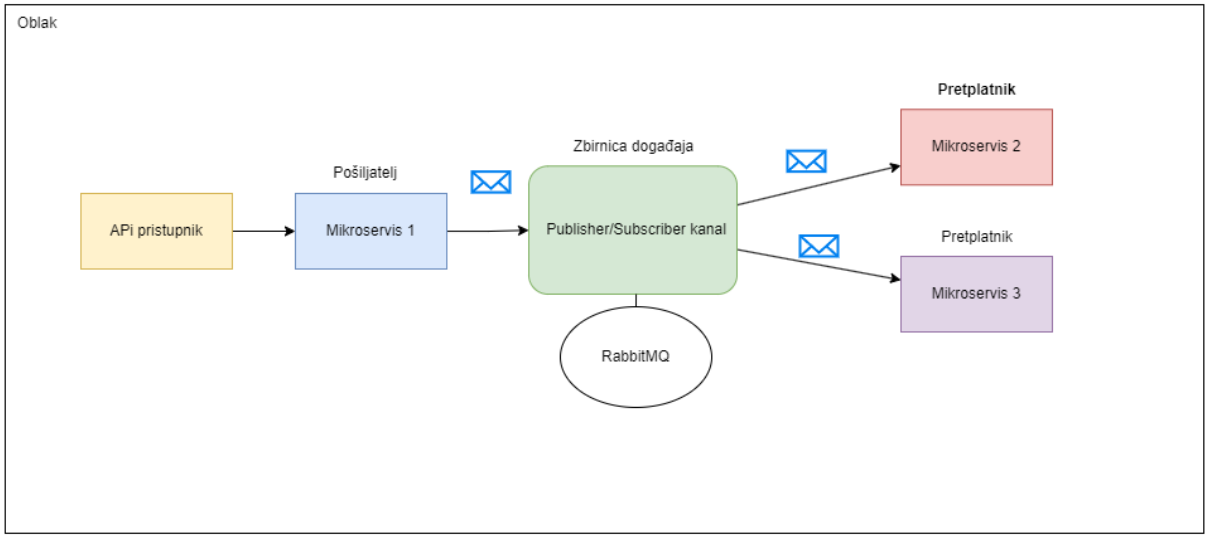


Slika 7. Komunikacija preko komandi i reda poruka

Vidi se da red poruka razbija ovisnost servisa. Red poruka je neposredni konstrukt kroz koji pošiljatelj i primatelj šalju poruke. Redovi poruka implementiraju asinkroni, od točke do točke, uzorak poruka. Pošiljatelj zna gdje komanda treba biti poslana i prema tome prilagođava rutu poruke. Red poruka garantira da je poruka procesuirana od točno jednog primatelja koji čita poruke s kanala. Pošiljatelj i primatelj se mogu skalirati neovisno jedan o drugome.

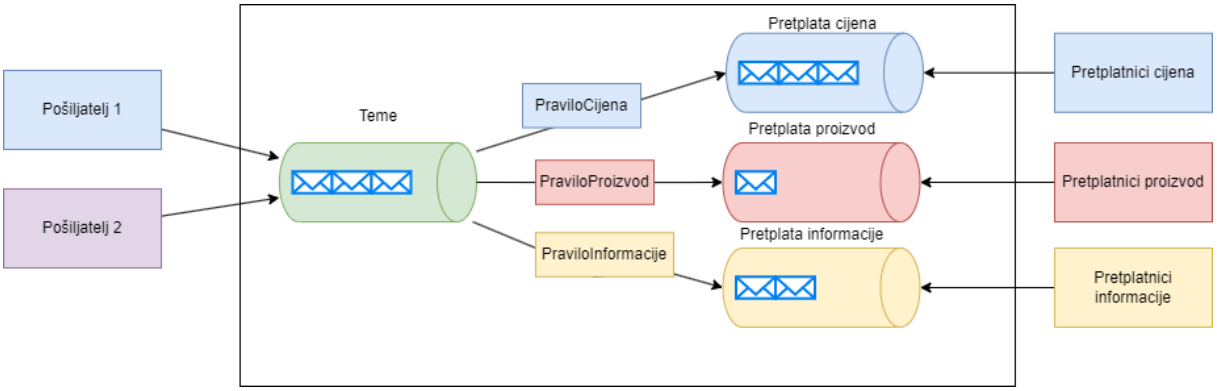
2.3.1.7. Događaji

Redovi poruka su odlični mehanizmi kada pošiljatelj treba poslati asinkronu poruku primatelju. Što ako postoji slučaj da više različitih primatelja želi primiti tu poruku? Ako se napravi novi red poruka za svakog primatelja to bi negativno utjecalo na skaliranje i sa širenjem sustava, samim sustavom bi eventualno bilo nemoguće upravljati. Kako bi se riješio ovaj problem uvodi se novi koncept zvan događaj. Jedan mikro servis objavi da se dogodio događaj, a ostali ako su zainteresirani reagiraju na njega. Ovaj koncept je poznat kao arhitektura vođena događajima. Arhitektura se implementira tako da kad se dogodi promjena stanja sustava, određeni mikro servis šalje događaj posredniku poruka. Time čini događaj dostupnim ostalim mikro servisima. Zainteresirani mikro servisi su obaviješteni o događaju tako što se prvobitno pretplate u posredniku poruka. Obično se koristi "Publisher/Subscriber“ uzorak dizajna da se implementira arhitektura vođena događajima.



Slika 8. Arhitektura vođena događajima

Vidi se da zbirnica događaja stoji na sredini komunikacijskog kanala. To je prilagođena klasa koja učahuruje posrednika poruka i razbija njegovu ovisnost od aplikacije. To znači da se može iskoristiti bilo koji drugi posrednik poruka. Mikro servisi koji su pretplatnici izvršavaju operacije događaja neovisno jedan o drugome. Kada se određeni događaj pojavi u posredniku poruka, oni reagiraju na njega. Kod ovog pristupa, miče se od reda poruka prema tehnologiji tema (eng. *topic*). Tema je slična kao red poruka, ali podržava „jedan prema više“ uzorak dizajna poruka. Jedan mikro servis objavi poruku, a više mikro servisa koji su pretplaćeni na tu temu reagiraju na tu poruku.



Slika 9. Komunikacija porukama pomoću tema

Vidi se da pošiljatelj šalje poruku u teme. Na kraju, pretplatnik primi poruku iz pretplate. Između toga, teme preusmjeravaju poruke prema pretplatama bazirano na pravilima. Pravila se ponašaju kao filteri koji šalju specifičnu poruku na točno određenu pretplatu, tj. na točno određeni red poruka. Sustav se može izgraditi s više različitih tema. Korisnik šalje poruku u

određenu temu s određenim pravilom, tema prema pravilu preusmjerava poruku prema određenom redu poruka. Klijenti koji su se pretplatili na taj red poruka, dobivaju tu poruku.

2.3.1.8. gRPC

gRPC je moderni, programski okvir visokih performansi koji je evoluirao iz starog RPC protokola (eng. *Remote Call Procedure protocol*). Na razini aplikacije, gRPC struji poruke između klijentske strane i servisa serverske strane. Tipična gRPC klijentska aplikacija razotkriva lokalnu funkciju koja implementira poslovnu logiku. „Ispod haube“, lokalna funkcija poziva drugu funkciju na udaljenom računalu. Za što se prvo čini da je lokalni poziv funkcije, zapravo je transparentni poziv izvan procesa na neki udaljeni servis. gRPC radi apstrakciju nad komunikacijom od točke do točke, serijalizaciju i izvršavanje operacija između računala. U izvornim aplikacijama za oblak, programeri često rade s različitim programskim jezicima, programskim okvirima i tehnologijama. Ta interoperabilnost komplicira komunikaciju porukama i izradu cjevovoda potrebnog za komunikaciju kroz različite tehnologije. gRPC pruža univerzalni horizontalni sloj koji radi apstrakciju nad ovim problemima. Programeri programiraju u svojim osnovnim platformama i fokusiraju se na poslovnu logiku, dok gRPC upravlja cjevovodima komunikacije. gRPC je dostupan u većini popularnih programskih jezika/tehnologija, poput Java, JavaScript, C#, Go, Swift i NodeJS. [5]

gRPc koristi HTTP/2 protokol, ali je kompatibilan s HTTP/1.1. Neke od naprednih značajki koje sadrži:

- Binarni prijenos podataka – bolji od HTTP/1.1 koji je baziran na tekstu
- Podrška za multipleksiranje (eng. *multiplexing*) za slanje više paralelnih zahtjeva putem iste veze – HTTP/1.1 ima podršku za samo jedan zahtjev/odgovor
- Dvosmjerna komunikacija (eng. *full-duplex*) za slanje klijentskih zahtjeva i serverskih odgovora istovremeno
- Ugrađeno strujanje omogućava zahtjeve i odgovore da asinkrono šalju velike skupine podataka
- Kompresija zaglavlja koja smanjuje trošenje internetskih resursa

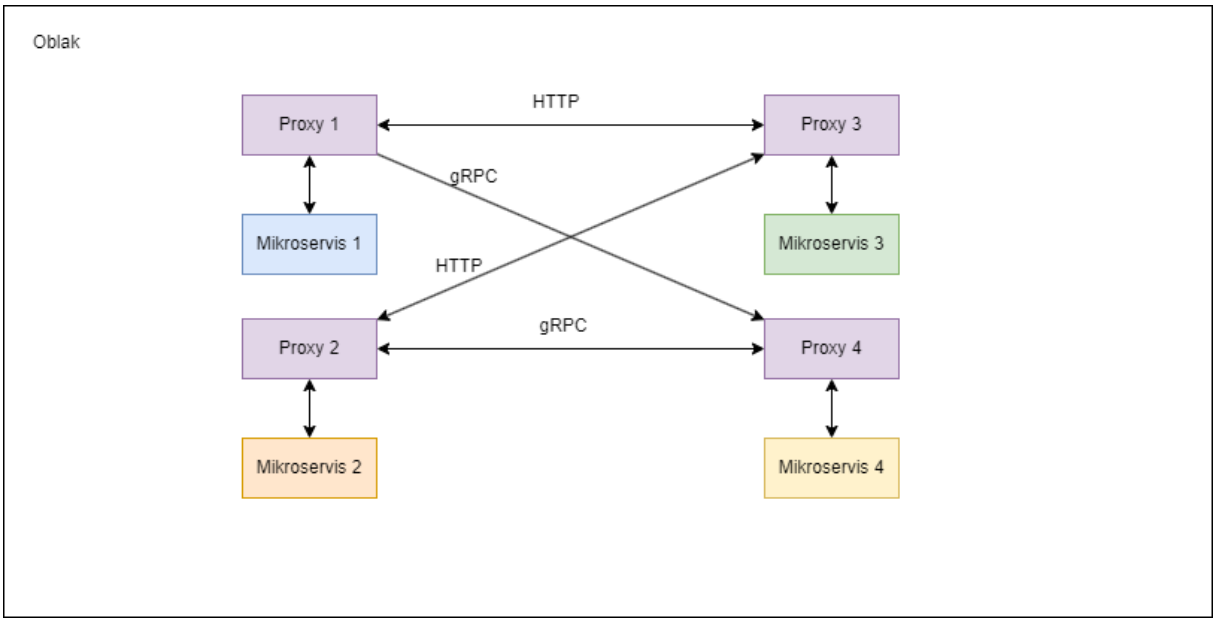
gRPC je jednostavan programski okvir visokih performansi. Može biti do osam puta brži od serijalizacije JSON-a s manjoj veličinom poruka, sa smanjenjem do 80%.

gRPC se koristi u sljedećim slučajevima:

- Sinkrona serverska „mikro servis do mikro servisa“ komunikacija gdje je neposredan odgovor važan za daljnje procesiranje zahtjeva
- Poliglotska okružja koja podržavaju miješanje programskih platformi
- Performanse komunikacije su jako važne – mala latencija i velika propusnost
- Komunikacija od točke do točke u stvarnom vremenu – gRPC može slati poruke u stvarnom vremenu i ima odličnu podršku za dvosmjerno strujanje
- Okružja sa slabijom vezom – binarne gRPC poruke su uvijek manje od ekvivalentnih JSON poruka baziranih na tekstu

2.3.1.9. „Service Mesh“ komunikacijska infrastruktura

Kroz poglavlje komunikacije objašnjene su prepreke kod izrade arhitekture. Rečeno je da programerski timovi trebaju dobro paziti kako servisi serverske strane razgovaraju jedni s drugima. Idealno, što manje komunikacije, to bolje. Istraženi su mnogi pristupi za implementaciju sinkrone HTTP komunikacije i asinkrone komunikacije porukama. U svakom slučaju, programeri moraju sami izraditi komunikacijski kod. Komunikacijski kod je kompleksan i skup u smislu vremena. Krive odluke mogu dovesti do velikih smanjenja performansi. Noviji moderni pristup komunikaciji mikro servisa bazira se na tehnologiji pod nazivom „Service Mesh“. To je podesivi sloj infrastrukture s ugrađenim mogućnostima upravljanja komunikacijom od servisa do servisa. Pomiče odgovornost komunikacijskih problema s mikro servisa na „Service Mesh“ sloj. Napravljena je apstrakcija komunikacije od mikro servisa. Glavna komponenta ovog pristupa je posrednik. U izvornim aplikacijama za oblak, instanca posrednika je obično smještena uz svaki mikro servis. Iako se pokreću u različitim procesima, usko su povezani i dijele zajednički životni ciklus. Ova infrastruktura često se implementira pomoću uzorka dizajna „Sidecar“:



Slika 10. Service Mesh uzorak implementiran pomoću Sidecar uzorka

Vidi se da su poruke presretane od strane posrednika koji je postavljen uz svaki mikro servis. Svaki posrednik može biti konfiguriran s pravilima mrežnog prometa specifičnog za njegov mikro servis. Razumije poruke i može ih preusmjeravati kroz servise i vanjski svijet. Uz upravljanje komunikacijom od servisa do servisa, „Service Mesh“ daje podršku za prepoznavanje servisa i usmjerivača opterećenja. Jednom kad se ispravno konfigurira, ovaj uzorak je jako funkcionalan.

2.3.2. Otpornost

Mikro servisna arhitektura sastoji se od mrežne komunikacije koja pokreće procese. U distribuiranoj arhitekturi, što se dogodi ako Servis B ne odgovara na zahtjev Servisa A? Ili što se dogodi ako Servis C privremeno bude nedostupan i drugi servisi koji šalju zahtjeve budu blokirani zbog toga?

Otpornost je sposobnost sustava da reagira na greške i svejedno ostaje funkcionalan. Smisao nije izbjeći sve greške, nego prihvatiti da će se dogoditi i konstruirati arhitekturu koja može ispravno odgovoriti na greške. Poželjan je povratak punom funkcionalnom stanju što je brže moguće. Naprotiv od klasičnih monolitnih aplikacija, gdje je sve pokrenuto u jednom procesu, izvorne aplikacije za oblak koriste distribuiranu arhitekturu. Svaki mikro servis i

njegove usluge podrške su pokrenuti u svom procesu, što vrijedi za svaki servis u kompletnoj arhitekturi i ti servisi komuniciraju putem mreže. Takvo okružje donosi sljedeće probleme:

- Neočekivana mrežna latencija
- Prolazni kvarovi – kada klijent na kratko ostane bez veze
- Blokada zbog dugotrajnih sinkronih operacija
- Glavni proces koji se srušio i trenutno se ponovno pokreće
- Preopterećen mikro servis koji ne može odgovoriti na zahtjeve
- Orkestralna operacija poput ažuriranja nekog od servisa ili prebacivanje na drugi čvor
- Hardverske greške

Platforme u oblaku mogu detektirati i izbjeći mnoge od ovih problema infrastrukture. Moguće je da se ponovno pokrene, skalira ili čak redistribuira servis na neki drugi čvor.

Prva crta obrane neke aplikacije je otpornost. Da bi se to postiglo, postoje određeni uzorci koji osiguravaju otpornost:

- Uzorak ponavljanja
- Uzorak prekidanja

Uzorak ponavljanja

U distribuiranim okružjima u oblaku, pozivi servisa mogu biti neuspješni zbog kratkotrajnih grešaka, koje se obično isprave nakon kratkog vremena. Implementacijom strategije ponovnog slanja poziva mogu se izbjeći ove greške. Uzorak ponavljanja omogućava servisu da ponovno pošalje neuspješni zahtjev određeni broj puta s eksponencijalnim rastom vremena čekanja na odgovor između poziva. Važno je da se konstantno povećava vrijeme čekanja poziva da se dopusti servisu da se oporavi. Najbolja praksa je da se vrijeme čekanja eksponencijalno povećava kako bi se dopustilo adekvatno vrijeme ispravljanja servisa. Također se predlaže da se uzme slučajan broj zajedno s eksponencijalnim povećavanjem da se ne bi dogodio slučaj u kojem svi klijenti pozivaju mikro servis koji se oporavlja u isto vrijeme. Time se smanjuje mogućnost zagušenja mikro servisa.

Uzorak prekidanja

Dok uzorak ponavljanja može pomoći u spašavanju zahtjeva koji su neuspjeli zbog kratkotrajnih grešaka, postoje situacije kada se greške dogode zbog neočekivanih događaja koji zahtijevaju dosta vremena da bi se riješili. Ove greške se kreću od parcijalnog gubitka veze do kompletnog rušenja servisa. U ovakvim situacijama, nema smisla da aplikacija ponovno pokušava izvršiti zahtjev koji nikad neće uspjeti. Da bi stvari bile još gore, konstantno pozivanje srušenog servisa može dovesti do samouzrokovane blokade aplikacije zbog poplave zahtjeva na srušeni servis koji će eventualno iscrpiti resurse poput memorije, dretvi te veza na bazu podataka i time uzročiti padove ostalih servisa koji se koriste tim resursima. U ovakvim situacijama, bilo bi poželjno da se zahtjevi odmah prekinu i samo pokušaju ponovno ako ima šanse za uspjehom. Uzorak prekidanja može spriječiti klijenta da šalje zahtjeve koji će vrlo vjerojatno biti neuspješni. Poslije određenog broja neuspješnih zahtjeva, blokira se sav promet do tog servisa. Nakon toga, periodično se poziva servis da se može odrediti da li se servis oporavio. Ovaj uzorak je dodatak uzorku ponavljanja.

2.3.3. Distribuirani podaci

Prema dizajnu, svaki mikro servis učahuruje svoje podatke i razotkriva operacije preko javnih sučelja. Kako onda napraviti upite nad podatcima ili implementirati transakcije koje koriste više mikro servisa?

Jedna baza podataka na mnogo načina čini upravljanje podacima jednostavnim. Kreiranje upita nad podacima iz više tablica je trivijalno. Promjene nad podacima se zajedno ažuriraju ili se kod pogreške sve vraća na prvobitno stanje. ACID (eng. *Atomicity, Consistency, Isolation and Durability*) transakcije garantiraju strogu konzistentnost. Kod dizajna izvornih aplikacija za oblak, koristimo drugačiji pristup. Svaki mikro servis učahuruje svoju poslovnu logiku i svoje podatke. Monolitna baza podataka se razbija u distribuirani model podataka koji se sastoji od puno malih baza podataka, svaka u skladu sa svojim mikro servisom. Na kraju dolazimo do dizajna koji sadrži bazu podataka po mikro servisu. Ovaj pristup od baze podataka po mikro servisu ima svoje beneficije:

- Podaci domene su učahureni u mikro servis
- Shema podataka nekog servisa može evoluirati bez direktnog utjecaja na ostale servise
- Svaka baza podataka se može zasebno skalirati
- Rušenje baze podataka u jednom servisu neće direktno utjecat i na druge servise

Segregacija podataka također omogućava da svaki mikro servis implementira svoj tip skladištenja podataka koji je najbolje optimiziran za njegovo radno opterećenje, zahtjeve spremišta i uzorke čitanja/pisanja. Mikro servisi mogu implementirati relacijsku, orijentiranu dokumentima, ključ-vrijednost pa čak i graf bazu podataka. Princip postojanosti poliglotskih trajnih spremišta znači da servisi u oblaku sadrže više različitih tipova spremišta podataka, pa čak i jedan servis više njih. Dok su relacijske baze podataka i dalje relevantne kod kompleksnih struktura, NoSQL baze podataka su dobile značajnu popularnost. One pružaju masivnu skalabilnost i veliku dostupnost. Učahurivanje podataka u različite mikro servise može povećati agilnost, performanse i skalabilnost, ali s druge strane stvara druge probleme, poput upita kroz više servisa, dosljednosti podataka, distribuiranih transakcija itd.

2.3.3.1. Upiti kroz više servisa

Iako su mikro servisi neovisni, često trebaju integraciju sa drugim servisima. Često se događa da jedan servis mora upitati drugi za neke podatke. S obzirom da su podaci učahureni u mikro servise, kod kompleksnijih upita jedan servis neće sadržavati sve potrebne podatke. Da bi se to ostvarilo, moguće je napraviti sinkroni HTTP zahtjev sa servisa na servis. To je već razmotreno i razumljivo je da bi takav uzorak povećao ovisnost između servisa i maknuo njihovu autonomnost. Također se može implementirati, već prije objašnjeni, zahtjev/odgovor uzorak sa redovima poruka. Ali taj uzorak je previše kompliciran i zahtijeva cjevovod zbog korelacije poruka zahtjeva i poruka odgovora. Iako bi razbio ovisnosti servisa, servis pošiljatelj svejedno mora sinkrono čekati na odgovor. Umjesto svega toga, široko prihvaćeni uzorak za razbijanje ovisnosti servisa je uzorak materijaliziranog pogleda. U ovom uzorku kreira se dodatna lokalna tablica u bazi podataka. Ta tablica sadrži denormaliziranu kopiju potrebnih podataka. Kopiranjem podataka u servis kojih ih koristi miče se potreba za skupim pozivima između mikro servisa. Kada su podaci lokalno spremljeni, poboljšava se vrijeme odaziva i pouzdanost. Isto tako, lokalna kopija podataka povećava otpornost jer servis ne ovisi direktno o tim podacima u drugom servisu. Ako drugi servis bude nedostupan, prvi servis ima svoju kopiju podataka. Problem s ovim pristupom su duplicirani podaci u sustavu. Strategijsko dupliciranje podataka u izvornim aplikacijama za oblak je usvojena praksa i ne smatra se anti-uzorkom ili lošom praksom. Treba voditi računa da jedan i samo jedan servis ima skup podataka i autoritet nad njima. Ostali servisi se moraju sinkronizirati sa servisom koji ima autoritet. Sinkronizacija se obično implementira preko asinkronih poruka sa „Publisher/Subscriber“ uzorkom.

2.3.3.2. Distribuirane transakcije

Dok je postavljanje upita nad podacima iz više servisa dosta komplicirano, implementiranje transakcije kroz više mikro servisa je još kompliciranije. Inherentni zadatak da se zadrži dosljednost kroz sve neovisne izvore podataka u različitim mikro servisima ne smije biti podcijenjen. Kada se izvršavaju kompleksne transakcije, svaki mikro servis kojeg poziva transakcija mora uspješno obaviti operacije, ili se sve moraju prekinuti i vratiti na prvobitno stanje. Iako je podrška za transakcije ugrađena u svaki mikro servis pojedinačno, nema nikakve podrške za distribuirane transakcije koje bi prošle kroz više servisa i zadržale podatke dosljednima. Popularan uzorak za dodavanje podrške za distribuirane transakcije je „Saga uzorak“. Implementira se tako da se grupiraju lokalne transakcije programski i sekvencijalno pozivaju jedna po jedna. Ako ikoja od lokalnih transakcija ne uspije, Saga prekida operaciju i poziva skup kompenzirajućih transakcija. Kompenzirajuće transakcije poništavaju promjene napravljene od prethodnih lokalnih transakcija i time vraćaju dosljednost podataka.

2.3.3.3. CQRS

CQRS (eng. *Command and Query Responsibility Segregation*) je arhitekturni uzorak koji može povećati performanse, skalabilnost i sigurnost. Ovaj uzorak razdvaja operacije čitanja podataka od operacija pisanja podataka. U normalnim scenarijima, isti model entiteta i repozitorij podataka se koriste i za operacije čitanja i za operacije pisanja. Za poboljšanje performansi, operacije čitanja mogu postavljati upite nad jako denormaliziranom reprezentacijom podataka da bi se izbjeglo skupo i ponavljajuće spajanje tablica i zaključavanje tablica. Operacija pisanja se izvršava nad jako normaliziranom reprezentacijom podataka koja garantira dosljednost. Treba se implementirati mehanizam sinkronizacije tih reprezentacija podataka. Obično, kada se dogodi operacija pisanja, pošalje se događaj koji replicira operaciju pisanja i konzumira ga baza koja obavlja operaciju čitanja. Svaka operacija pisanja se sprema u spremište pisanja i onda propagira spremištima koja čitaju. Spremiše koje čita se eventualno sinkronizira sa spremištem koje piše, ali mogu se desiti zastoji kod sinkronizacije. Separacija omogućava spremišta pisanja i spremišta čitanja da se skaliraju neovisno jedan o drugome. Spremišta čitanja koriste sheme optimizirane za upite, dok spremišta pisanja koriste sheme optimizirane za ažuriranje podataka. Upiti čitanja se postavljaju nad denormaliziranim podacima, dok se kompleksna poslovna logika može dodijeliti upitima pisanja. Također, može se implementirati jača sigurnost nad operacijama pisanja nego nad operacijama čitanja.

2.3.3.4. Relacijske i NoSQL baze podataka

Relacijske i NoSQL baze podataka su dva najpopularnija tipa koji se koriste u izvornim aplikacijama za oblak. Relacijske baze podataka su relevantne već desetljećima. One su zrele, potvrđene i naširoko popularne. Relacijske baze podataka pružaju spremište za povezane tablice podataka. Ove tablice imaju fiksnu shemu, koriste SQL za upravljanje podacima i garantiraju dosljednost ACID transakcijama. NoSQL baze podataka su visokih performansi i nisu relacijske baze podataka, tj. ne baziraju se na relacijama između podataka. Popularne su zbog lakoće korištenja, skalabilnosti, elastičnosti i visoke dostupnosti. Umjesto da se spajaju tablice normaliziranih podataka, ove baze spremaju nestrukturirane ili polustrukturirane podatke, često kao ključ-vrijednost ili kao JSON (eng. *JavaScript Object Notation*) dokumente. NoSQL baze obično ne pružaju dosljednost ACID transakcijama.

Tablica 2. Karakteristike NoSQL baza podataka

Model	Karakteristike
Baza dokumenata	Podaci i metapodaci su spremljeni u hijerarhiju u JSON dokumentima unutar baze
Ključ-vrijednost baza	Podaci su reprezentirani kao kolekcija ključ-vrijednost parova
Baza širokih stupaca	Povezani podaci su spremljeni kao skup ugniježđenih ključ-vrijednost parova u jednom stupcu
Graf baze	Podaci su spremljeni u strukturu grafa kao čvorovi, rubovi (poveznice između čvorova) i svojstva podataka

CAP teorem

CAP (eng. *Consistency, Availability and Partitioning tolerance*) teorem je jedan od načina razumijevanja razlika između ovih vrsta baza podataka. Teorem nalaže da distribuirani sustavi podataka nude kompromis između dosljednosti, dostupnosti i tolerancije podjeljivanja, te da svaka baza podataka može ponuditi samo dva od tri:

- Dosljednost – Svaki čvor u klasteru odgovara najnovijim podacima, čak i ako sustav mora blokirati zahtjev sve dok se sve kopije ne sinkroniziraju. Ako se pošalje upit na konzistentni sustav i on sadrži elemente koji se ažuriraju, mora se čekati na odgovor dok se sve replike ne sinkroniziraju.

- Dostupnost – Svaki čvor vraća neposredan odgovor, čak i ako podaci nisu dosljedni. Ako se pošalje upit na ovakav sustav i on sadrži elemente koji se ažuriraju, dobit će se najbolji mogući odgovor koji sustav može dati u tom trenutku.
- Tolerancija podjeljivanja – Garantira da sustav nastavlja raditi čak i ako se replika baze podataka sruši ili izgubi vezu sa ostalim replikama.

Relacijske baze podataka obično nude dosljednost i dostupnost, ali ne toleranciju na podjeljivanje. One su obično pokrenute na jednom serveru i skaliraju se vertikalno tako da se doda više resursa računalu na kojem su pokrenute. Mnoge relacijske baze podataka nude ugrađenu značajku repliciranja gdje se mogu napraviti kopije glavne baze podataka na drugim serverima. Operacije pisanja se obavljaju samo na glavnoj bazi i repliciraju se na sve replike. Ukoliko se dogodi greška, operacije se mogu prebaciti na neku od replika. Replike se koriste za distribuciju operacija čitanja. To znači da se operacije pisanja izvršavaju isključivo na glavnoj bazi podataka, operacije čitanja se preusmjeravaju na replike. Podaci se mogu horizontalno podijeliti preko više čvorova tehnikom usitnjavanja. Ali usitnjavanje drastično povećava vrijeme izvršavanja operacija zato što su podaci podijeljeni preko više čvorova, i time se komplicira komunikacija između distribuiranih podataka i njihovo spajanje u neku smislenu cjelinu. Može biti novčano i vremenski skupo održavati takav sustav. Relacijske značajke poput spajanja tablica, transakcija i referentni integritet zahtijevaju puno veće performanse u podijeljenim sustavima. Dosljednost podataka i oporavljanje replika se može konfigurirati na način da radi sinkrono ili asinkrono. Ako replike baze podataka izgube vezu u visoko dosljednim sustavima ili sinkronim relacijskim klasterima, na glavnoj bazi podataka se ne mogu obavljati operacije pisanja.

NoSQL baze podataka obično podržavaju visoku dostupnost i toleranciju podjeljivanja. Skaliraju se horizontalno, često preko više servera. Ovakav pristup pruža jako veliku dostupnost, čak i preko širokih geografskih područja. Dosljednost se obično podešava kroz protokole i mehanizme. Nude više kontrole kod izabira između sinkronog ili asinkronog repliciranja. Ako neka od replika izgubi vezu u visoko dostupnom sustavu, svejedno se mogu izvršiti operacije pisanja. Baza podataka dopušta operacije pisanja i ažuriranja replika čim postanu dostupne. NoSQL baze podataka koje dopuštaju više replika s operacijama pisanja još više podižu dostupnost sustava. Moderne NoSQL baze obično implementiraju svojstvo podjele kao osnovnu značajku. Upravljanje particijama je već ugrađeno u samu bazu.

Visoka dostupnost i masivno skaliranje često su od veće važnosti poslovnom sustavu nego relacijsko povezivanje podataka i referentni integritet. Programeri mogu implementirati uzorke poput Saga uzorka, CQRS-a i asinkronih poruka da bi dobili željenu dosljednost. Danas se mogu koristiti i najnovije NewSQL baze podataka koje proširuju klasične relacijske baze podataka tako što daju podršku za horizontalno skaliranje i skaliranje visokih performansi. Takve baze ujedinjuju sva tri koncepta CAP teorema. Može se reći da uzimaju najbolje od obje vrste baza, dosljednost i integritet relacijskih baza podataka te visoku dostupnost i skaliranje NoSQL baza podataka.

Tablica 3. Odabir između relacijske i NoSQL baze podataka

NoSQL baze podataka	Relacijske baze podataka
Okružja su velikih opterećenja koje moraju imati predvidljivu latenciju	Najviša razina opterećenja je oko par tisuća transakcija po sekundi
Podaci su dinamički i često se mijenjaju	Podaci su visoko strukturirani i zahtijevaju referentni integritet
Veze između podataka mogu biti denormalizirani modeli	Relacije se izražavaju kroz spajanje normaliziranih podataka
Dohvaćanje podataka je jednostavno i ne zahtijeva spajanja tablica	Dohvaćanje podataka je kompleksno
Podaci su geografski široko replicirani	Podaci su centralizirani ili se mogu jednostavno asinkrono replicirati
Aplikacija će raditi na slabijem hardveru, poput javnih oblaka	Aplikacija će raditi na velikim hardverskim sustavima

Baza podataka kao servis

Izvorne aplikacije za oblak daju prednost servisima koji razotkriveni kao „baza podataka kao servis“ (DBaaS, eng. *Database as a Service*). Implementiraju se na način da se napravi apstrakcija nad bazom podataka koja ima ugrađenu sigurnost, skalabilnost i praćenje stanja. Na taj način baza podataka se koristi kao usluga podrške.

2.3.3.5. Predmemoriranje

Predmemoriranje (eng. *caching*) može doprinijeti poboljšanju performansi, skalabilnosti, dostupnosti za pojedinačan mikro servis ili za cijeli sustav. Smanjuje latenciju i probleme rukovanja hrpom istovremenih zahtjeva. Što sustav ima više podataka i korisnika, sve veće i veće prednosti predmemoriranja postaju. Predmemoriranje je najviše efektivno kada korisnik konstantno čita nepromjenjive podatke ili podatke koji se jako rijetko mijenjaju. Iako mikro servisi ne bi trebali imati nikakvo stanje, distribuirana predmemorija može poduprijeti istovremeni pristup podacima, na primjer podacima stanja sesija korisnika, kada je to apsolutno potrebno. Također se može koristiti u slučajevima ponavljanja izračunavanja. Ako neka operacija transformira podatke ili radi kompleksne izračune, dobro je spremiti rezultat u predmemoriju.

Izvorne aplikacije za oblak obično implementiraju distribuiranu predmemorijsku arhitekturu. Predmemorija je isporučena kao servis podrške, odvojena od mikro servisa. Obično se usko povezuje s API pristupnikom. Time dobivamo na velikoj dostupnosti i smanjenoj latenciji jer API pristupnik vraća gotov rezultat iz predmemorije i ne mora pozivati mikro servise. S obzirom da predmemorija nije povezana s mikro servisima, može evoluirati i skalirati se neovisno o njima. Ovaj uzorak je poznat kao uzorak „Predmemorija sa strane“. Kada zahtjev dođe na API pristupnik, prvo se traži odgovor u predmemoriji. Ako se nađe, odgovor se vraća neposredno. Ako odgovor u predmemoriji ne postoji, API pristupnik zove mikro servise da mu vrate rezultat operacije i onda taj rezultat sprema u predmemoriju za buduće zahtjeve, te sam rezultat vraća korisniku. Treba obraćati pažnju da se periodično osvježe podaci u predmemoriji kako bi se osigurala dosljednost i integritet podataka.

2.3.4. Sigurnost

Postavljaju se pitanja sigurnosti, poput: kako će mikro servisi sigurno spremati i upravljati tajnama i lozinkama i osjetljivim konfiguracijskim podacima? Kako će se servisi obraniti i prepoznati hakerske napade?

Hakerski napadi događaju se svakodnevno i jedan veliki dio izrade izvornih aplikacija za oblak je njihovo apsolutno suzbijanje. Veliki problemi se događaju kod korištenja velikog izbora tehnologija i njihovih ovisnosti. Nekada programeri naprave sustav velike sigurnosti, no zbog ovisnosti o drugim tehnologija, te tehnologije imaju neke interne propuste koji rezultiraju

nesigurnosti cijelog sustava. Na primjer, neažuriranje servera, stare verzije programskih okvira i propusti u implementaciji sigurnosti baze podataka mogu rezultirati u nesigurnosti sustava.

Izvorne aplikacije za oblak ujedno mogu biti lakše i teže za osigurati nego tradicionalne aplikacije. S jedne strane, mora se osigurati više manjih aplikacija i utrošiti više energije pri izgradnji sigurnosne infrastrukture. S druge strane, mali servisi, svaki sa svojim izvorom podataka, smanjuju prostor za napad. Ako napadač uspije probiti u jedan servis, vjerojatno je dosta teže da iz njega probije i u druge nego u monolitnoj aplikaciji. Isto tako, ako se dogodi curenje podataka iz neke baze, prostor napada je manji jer svaki servis ima svoju bazu podataka koja sadrži samo jedan dio cijelog sustava.

2.3.4.1. Modeliranje prijetnji

Važno je ne zaboraviti da se i kod izvornih aplikacija za oblak treba držati osnovnih principa zaštite. Sigurnost i razmišljanje o sigurnosti trebaju biti prisutni kod svakog koraka izgradnje aplikacije. Uvijek treba postavljati pitanja poput:

- Što će se desiti ako se izgube podaci?
- Kako smanjiti štetu ako se ubace neželjeni podaci?
- Tko ima pristup podacima?
- Postoje li politike revizije oko procesa izgradnje i isporuke?

Sva ova pitanja dio su procesa modeliranja prijetnji. Proces pokušava odgovoriti na pitanje „koje su sve prijetnje moguće ovom sustavu?“, koliko su te prijetnje ozbiljne i kolika je potencijalna šteta ako se dogode. Jednom kad se lista prijetnji sastavi, treba razmotriti koje su prijetnje dostojne pažnje.

2.3.4.2. Princip najmanje ovlasti

Princip najmanje ovlasti je jedan od temeljnih principa računalne sigurnosti. Princip nalaže da svaki korisnik ili proces trebaju imati najmanje moguće ovlasti da izvrše zadatak. Jedan od dobrih primjera je pravo korisnika pri vezi na bazu podataka. U mnogim slučajevima se koristi isti korisnički račun za izradu baze podataka i kasnije vezu na nju u produkciji. Ali postoje ekstremni slučajevi u kojima, na primjer, u produkciji, korisnik ne smije imati mogućnost ažuriranja podataka. Ili primjer gdje postoje više vrsta korisničkih računa svaki sa svojim privilegijama. Te privilegije se trebaju prenesti i kod ovlasti pri izvršavanju upita u bazi

podataka. Ovaj način zahtijeva više različitih korisničkih računa u bazi podataka s pripadnim ovlastima.

2.3.4.3. Ispitivanje penetracije

Kako aplikacija postaje više kompleksna, tako vektor napada raste. Princip modeliranja prijetnji ima problem, zato što se izvršava od istih ljudi koji izgrađuju sustav. Postoji šansa da programeri ne vide cijeli vektor napada i jednostavno izostave potencijalne prijetnje. Tu dolazi tehnika ispitivanja penetracije. To je tehnika koja nalaže da se dovedu vanjski akteri koji pokušavaju probiti u sustav. Ti napadači mogu biti neka druga tvrtka koja se bavi time ili neki drugi programeri iskusni u sigurnosti. Oni traže rupe u sustavu koje treba ažurirati.

2.3.4.4. Praćenje stanja

Ako se slučajno dogodi da napadač uđe u sustav, o tome treba biti upozorenje. Često se dogodi da se napadi otkriju kod pregleda dnevnika sustava. Napadi mogu ostaviti znakove prije nego što se uspiju izvršiti do kraja. Na primjer, ako napadač pokušava pogoditi nečiju lozinku, napravit će puno zahtjeva autentifikacijskom servisu. Praćenjem dnevnika mogu se uočiti „čudne“ radnje koje se događaju u sustavu. To praćenje dnevnika se također može pretvoriti i u alarmni sustav koji podiže uzbunu kada detektira određeno čudno ponašanje. Može obavijestiti korisnika da primjećuje čudno ponašanje izazvano od strane njegovog računa ili čak može obaviti preventivne radnje da automatski suzbije napad.

2.3.4.5. Osigurati izgradnju izvršne verzije

Jedno mjesto gdje se sigurnost dosta često izostavlja je izgradnja izvršne verzije. Izgradnja treba sama sadržavati provjeru sigurnosti tako da skenira nesiguran kod ili slučajno ostavljene akreditacije.

2.3.4.6. Razvoj sigurnog koda

Prvobitno treba odabrati programske jezike i programske okvire koji su sami po sebi sigurni i imaju dobro implementiranu sigurnost. Također treba pratiti najnovije smjernice za sigurnost računalnih sustava. Samostalno izgraditi alate koji detektiraju potencijalne nesigurnosti u kodu, na primjer, budući da većina aplikacija koristi vanjske ovisnosti, alat za detekciju ažuriranja tih ovisnosti je više nego potreban.

2.3.4.7. Sigurnost tajni

Lozinke i certifikati su vrlo često vektor napada hakera. Jako je važno da su lozinke snažne, da se koriste spori algoritmi sažimanja i da se natjera korisnike na izradu snažnih lozinki.

2.3.5. Mane mikro servisa

Mikro servisi nisu uvijek magično rješenje kako se to čini iz prve. Rastavljanjem monolitne aplikacije na više različitih samostalnih servisa uvelike povećava kompleksnost sustava. Mikro servisi dolaze s velikim troškovima i kompromisima ako se razmatraju doslovno: kupuje se izbornost tako što se plaća razvoj po narudžbi i kontinuirano održavanje. Neki od glavnih nedostataka mikro servisa su:

1. Povećani troškovi održavanja
2. Organizacijska složenost
3. Složena koordinacija
4. Rizik od kaskada grešaka
5. Problemi performansi i pouzdanosti

2.3.5.1. Povećani troškovi održavanja

Agilnost i fleksibilnost mikro servisa uvode povećanje operativne složenosti i time povećavaju troškove održavanja. Svaki servis treba održavati posebno i to platiti. Ponekad je potrebno platiti vanjske tvrtke za održavanje. Najuobičajeniji primjer je programerska tvrtka koja plaća korištenje oblaka neke druge tvrtke.

2.3.5.2. Organizacijska složenost

Mikro servisi dopuštaju programerima da rade samostalno i u servisima koriste alate i programske okvire koje oni žele. Time mogu puno brže implementirati kod i isporučiti ga. Ali to nije toliko povoljno za poslodavce, autonomni timovi stvaraju troškove. Potrebo je izraditi ispravnu separaciju odgovornosti i detaljnu dokumentaciju svakog servisa, te same unutarnje komunikacije između servisa, i kako će se servisi dugoročno održavati te tko će ih održavati. Ovo dovodi do povećanja tehničkih troškova, dodatnih slojeva menadžmenta i servisa čije funkcionalnosti razumiju samo oni koji su ih i implementirali.

2.3.5.3. Složena koordinacija

Treba razumjeti da je potrebno voditi računa o više servisa i koordinaciji tih servisa. Iako su servisi sami za sebe samostalni, moraju raditi zajedno da bi ispunili korisničke zahtjeve. Nagomilavanje ovisnosti često dovodi do kompleksnih i lako lomljivih isporuka. Velike kompanije imaju i preko dvije tisuće mikro servisa koje održava preko petsto programera. U takvim sustavima jako je teško pronalaziti greške. Također je sve teže i teže implementirati nove značajke jer one povlače mnogo promjena za sobom.

2.3.5.4. Rizik od kaskada grešaka

Zbog podijeljenosti servisa, svaki sa svojom implementacijom, arhitekturom i performansama, signifikantno se podiže rizik od kaskada grešaka. Otkrivanje ovakvih grešaka zahtjeva sposobnost praćenja i otkrivanja grešaka od ranih faza implementacije. Način da se izbjegne niz grešaka je kao prvo ispravna implementacija sučelja unutar sustava, standardizirani protokoli komunikacije i sustavi za oporavak od grešaka.

2.3.5.5. Problemi performansi i pouzdanosti

Unutarnja komunikacija između servisa može vrlo brzo postati vrlo kompleksna. Različite tehnologije unutar servisa dovode do različitih načina komunikacije između servisa koji uvelike mogu utjecati na performanse sustava. Slabe performanse samo jednog servisa, utjecat će na sve pozive koji prolaze kroz taj servis i tako usporiti rad cijelog sustava.

2.4. Kontejneri

Kontejneri su jedan od osnova izvornih aplikacija za oblak. Oni pakiraju softver u standardizirane jedinice za izgradnju, otpremu i isporuku na server. Pakiraju cijeli kod, zajedno sa svim ovisnostima tako da aplikacija radi jednako i pouzdano na svim računalnim okružjima. To znači da jedan kontejner sadrži sve potrebno da bi se aplikacija ispravno pokrenula i ispravno radila: kod, izvršno okružje, sistemske alate, sistemske knjižnice i postavke. Dostupni su i za Linux i Windows operacijske sustave i osiguravaju da aplikacija uvijek radi jednako, neovisno o infrastrukturi. Oni izoliraju softver od okružja i osiguravaju da softver radi uniformno, neovisno o razlikama između izgradnje i isporuke. Kreiranje kontejnera od mikro servisa je dosta jednostavno. Kod, izvršno okružje i ovisnosti su upakirane u binarnu datoteku koja se naziva slika kontejnera. Slike se spremaju u registar koji se ponaša kao repozitorij. Kada se

želi pokrenuti aplikacija, jednostavno se slika pokrene u kontejneru. Instanca aplikacije se pokreće u nekom stroju kontejnera, poput Docker-a. Može se pokrenuti više instanca kontejnera, ograničeno jedino specifikacijama sustava na kojem se pokreću. Svaki kontejner ima svoje izvršno okružje i ovisnosti. To omogućava pokretanje iste aplikacije s različitim ovisnostima. Odličan primjer je testiranje aplikacije s različitim verzijama ovisnosti. Svaki kontejner dijeli dio računala na kojem je pokrenut u smislu operacijskog sustava, radne memorije i procesora, ali su ti kontejneri svejedno izolirani jedni od drugih. [6]

Kontejneri pružaju portabilnost i garantiraju dosljednost kroz različita okružja. Tako što učahuruju sve u jedan paket, izoliraju se mikro servisi i njihove ovisnosti od temeljne infrastrukture. Kontejneri se mogu isporučiti u bilo koje okružje koje koristi Docker stroj. Okružja koja koriste kontejnere izbjegavaju proces konfiguracije u smislu instaliranja potrebnih programskih okvira, softverskih knjižnica i strojeva koji pokreću sustav. Tako što dijele operacijski sustav i resurse domaćina, kontejneri ostavljaju puno manji trag od virtualnih računala. [7, str. 13]

Dok alati poput Docker-a kreiraju slike i pokreću kontejnere, potreban je i alat za upravljanjem. Upravljanje kontejnerima se vrši od strane specijalnih softvera koji se nazivaju orkestratori kontejnera. Orkestratori pružaju načela zamjenjivosti i dosljednosti. Orkestratori mogu stvarati nove instance bilo kada se neka od instanca sruši i time ispuniti načelo zamjenjivosti. Samim time što se instance ruše i stvaraju, kreiraju dosljedan sustav i time se ispunjava načelo dosljednosti. Jedan on najpoznatijih orkestratora kontejnera je Kubernetes. [8, str. 149]

2.5. Servisi podrške

Izvorne aplikacija za oblak ovise o mnogim pomoćnim resursima poput spremišta podataka, posrednika poruka, dnevnika i servisa identiteta. Ti servisi se nazivaju servisi podrške. Servisi podrške se mogu sami izrađivati, ali to bi utrošilo previše novca i vremena, zahtijevalo puno više stručnjaka iz različitih oblasti, te napravilo probleme licenciranja, nabavljanja i upravljanja tim servisima. Umjesto da sami posjedujemo te servise, mogu se jednostavno platiti i konzumirati. Najbolja praksa je tretiranje servisa podrške kao priložene resurse, koji su dinamički povezani s mikro servisom s konfiguracijskim informacijama spremljenim u eksternoj konfiguraciji. S ovim uzorkom, servis podrške se može priložiti i

maknuti bez promjene koda. Servisi podrške pružaju sučelja kako bi se moglo komunicirati s njima. Komunikacija direktno preko tih sučelja će usko vezati aplikaciju uz taj servis podrške. Široko je prihvaćena praksa da se izolira ta komunikacija. Uvođenjem posredničkog sloja ili sučelja, i izlaganjem generičkih funkcija aplikaciji koje objedinjuju komunikaciju, razbija se uska ovisnost. Time se dobiva mogućnost zamjene jednog servisa podrške za drugi.

2.6. Automatizacija

Kako je već rečeno, izvorne aplikacije za oblak koriste mikro servise, kontejnere i moderni dizajn informatičkih sustava. Postavljaju se pitanja automatizacije svega toga. Kako osigurati okružja u oblaku u kojem ovi sustavi rade? Kako konstantno isporučivati nove značajke i ažuriranja aplikacije?

Široko prihvaćena praksa je infrastruktura kao kod (IaC, eng. *Infrastructure as Code*). Ova praksa pruža automatiziranje procesa isporuke aplikacije. Uglavnom se primjenjuju prakse softverskog inženjerstva poput testiranja i verzioniranja. Infrastruktura i isporuka postaju automatizirane, dosljedne i lako ponavljajuće. Poanta je da se izgradnja infrastrukture zabilježi u skriptu ili neku opisnu datoteku. Imena resursa, lokacije, kapaciteti i tajne/lozinke su parametrizirani i dinamični. Skripta se verzionira i postavlja u kontrolu izvora kao dio projekta. Skripta se poziva da se osigura dosljedna i ponovljiva infrastruktura kroz sistemska okružja poput okružja za testiranje ili produkciju. „Ispod haube“, infrastruktura kao kod je idemponentna, što znači da će svako pokretanje skripte rezultirati istim rezultatom. Ako su potrebne promjene, skripta se ažurira i ponovno pokrene. Ažuriraju se samo potrebni resursi. Kubernetes orkestrator koristi YAML (eng. *YAML Ain't Markup Language*) datoteke koje su opisne konfiguracijske datoteke. One opisuju kako će se neki Kubernetes resurs isporučiti u mrežu.

Sljedeći proces je automatizacija isporuke. Moderni CI/CD (eng. *Continuous Integration and Continuous Delivery*) sustavi uvelike pomažu kod automatizacije procesa isporuke novih ili ažuriranih verzija. Pružaju posebne korake izgradnje i isporuke koji osiguravaju dosljedan i kvalitetan kod koji je odmah dostupan korisnicima. Mikro servisi su manji od monolitnih aplikacija i mogu se isporučivati samostalno, tako da su manje rizični od isporuke velikih aplikacija [9]. Proces se sastoji od sljedećih koraka:

1- Programer izgradi značajku u razvojnom okruženju, iterira kroz programiranje, pokretanje i ispravljanje pogrešaka
2- Kada je gotov, objavljuje kod na nekom od sustava za verzioniranje koda, na primjer GitHub
3- Objava okida korak izgradnje koje pretvara kod u binarni artefakt, tj. izvršnu verziju. Ovaj proces je implementiran s cjevovodom kontinuirane integracije. Automatski se izgrađuje, testira i pakira aplikacija.
4- Faza isporuke preuzima binarni artefakt, aplicira eksternu konfiguraciju aplikacije i okružja te proizvodi nepromjenjivo izdanje. To izdanje se isporučuje specifičnom okružju. Ovaj proces je implementiran s cjevovodom konstantne isporuke. Svako izdanje mora biti jedinstveno prepoznatljivo.
5- Na kraju se izdana značajka pokreće u ciljanom izvršnom okružju. Izdanja su nepromjenjiva što znači da bilo koja promjena uzrokuje kreaciju novog izdanja.

Koristeći ove prakse, mnoge tvrtke su se radikalno promijenile u smislu kako isporučuju softver. Mnogo ih se prebacilo s kvartalne isporuke na isporuku po potrebi. Cilj je pronaći greške rano u ciklusu razvoja kada ih je lako popraviti. Što je veći interval između iteracije, to su greške „skuplje“ za popraviti. S dosljednošću u integracijskom procesu, timovi mogu potvrđivati promjene češće, što dovodi do bolje suradnje i kvalitetnijeg softvera.

Dobar primjer je kreiranje novih verzija na Azure DevOps sustavu. Obično se spoji grana na kojoj je rađena nova značajka ili grana u kojoj su napravljeni ispravci, u glavnu granu u sustavu za verzioniranje, taj sustav je obično Git sustav za verzioniranje koda koji je i ujedno sustav za verzioniranje unutar Azure DevOps-a. Nakon toga se glavna grana označuje s nekom verzijom, na primjer verzija „1.0.0“. Označivanjem verzije okida se cjevovod u Azure DevOps sustavu koji odrađuje pojedine korake koji su mu određeni. Obično se obavljaju procesi provjere stila pisanja koda (eng. *linting*) i nekih internih pravila pisanja koda te se pokreću automatizirani testovi. Ako su ti procesi uspješni, uobičajeni sljedeći korak je izgradnja Docker slike servisa i njegovo isporučivanje u registar s označenom verzijom.

2.7. Izvorna Java Cloud rješenja

Izvorna Java Cloud rješenja su tehnologije koje nam pomažu u pripremi i izradi izvršne datoteke aplikacije koju je moguće pokrenuti u oblaku. Ovakve tehnologije pomažu kod kompiliranja, izrade izvršne datoteke aplikacije i pripreme kontejnera za isporuku u oblak. Jedno od ovakvih rješenja specifičnih za Java-u je GraalVM.

GraalVM je Java razvojna oprema koja ima mogućnost kompiliranja Java aplikacija unaprijed u samostalne binarne datoteke, nasuprot starom JIT kompiliranju (eng. *Just-In-Time compailing*) koje se događalo neposredno prije pokretanja aplikacije i time uvelike usporilo proces pokretanja. JIT kompilacija obično kompilirala aplikaciju u bajt kod, koji ne ovisi o platformi izvršavanja, i potom se pokreće na JVT-u (eng. *Java Virtual Machine*). Kada se aplikacija pokrene, JVM koristi JIT kompilaciju da kompilira bajt kod u strojni kod koji je specifičan za hardver na kojem se aplikacija pokreće. To znači da se kompilacija pokreće neposredno prije pokretanja aplikacije i uvelike usporava proces pokretanja aplikacije. GraalVM koristi AOT kompilaciju (eng. *Ahead-Of-Time*) koje kompilira kod u strojni kod prije samog pokretanja aplikacije. Ovakav proces je moguć, jer se samim korištenjem izgradnje GraalVM izvorne slike, maknula ovisnost o platformi, tj. GraalVM će, ovisno o platformi na kojoj je alat pokrenut, izgraditi binarnu verziju baš za tu platformu. To znači, ako programer programira na Windows operacijskom sustavu i lokalno kreira izvornu sliku, ta slika će biti prilagođena za Windows operacijski sustav i neće raditi u Linux kontejneru. Obično se procesi izgradnje izvršne datoteke aplikacije i izgradnja slike kontejnera odvojeni procesi, no ovdje se vidi da ako kontejner i razvojno okruženje ne koriste istu platformu, to neće biti moguće i proces izgradnje izvršne datoteke i slike kontejnera se moraju ujediniti. AOT kompilacija kreira puno manje izvršne binarne datoteke, aplikacija se pokreće do sto puta brže, postižu se gornje granice performansi bez inicijalnog zagrijavanja, tj. aplikacija se pokreće u milisekundama i gotovo odmah postiže stop posto svojih mogućnosti, i te binarne datoteke koriste manje radne memorije i procesora nego aplikacije koje se pokreću na JVM. Ovi problemi mogu se riješiti unakrsnom kompilacijom ili CI/CD cjevovodima. Unakrsna kompilacija zahtijeva dodatnu konfiguraciju Windowsa u ovom slučaju, može se instalirati Linux podsustav za Windows unutar kojeg će se obavljati kompilacija. Dosta čest pristup je CI/CD cjevovod koji je pokrenut na serveru s Linux operacijskim sustavom ili kontejnerima baziranim na Linuxu, time se osigurava da će AOT kompilacija napraviti strojni kod prilagođen Linux operacijskom sustavu. [10]

Napravljeno je testiranje performansi aplikacije koja je pokrenuta u Kubernetes sustavu i izgrađena pomoću GraalVM u usporedbi s istom aplikacijom koja je izgrađena pomoću OpenJDK 13 Java platforme. U obzir su se uzimali konkurentnost i paralelno programiranje, kako bi se u testovima što realnije koristila moderna, konkurentna, objektno orijentirana aplikacija. Napravljen je REST API koji sadrži obične CRUD (eng. *Create, Read, Update and Delete*) operacije u različitim programskim okvirima poput: Spring Boot-a, Quarkus-a i Micronaut-a. API koristi MySQL bazu podataka preko Hibernate programskom okvira koji služi za konekciju na bazu podataka i izvršavanje upita i transakcija putem koda na bazi podataka. Baza podataka ima jednu tablicu s proizvodima i sadrži oko dvjesto pedeset redova. API se sastoji od tri krajnje točke [11]:

- GET „/products“ – dohvaćanje svih proizvoda
- GET „/products/{id}“ – dohvaćanje jednog proizvoda prema identifikatoru
- POST „/products“ – spremanje novog proizvoda u bazu podataka

Svi testovi napravljeni su na identičnom sustavu. Rezultati su pokazali masivno poboljšanje vremena odgovora iz REST API-a koji je kompiliran pomoću GraalVM-a. U testovima su korišteni sljedeći REST API-i i njihovo prosječno vrijeme odgovora iznosi: Sprint Boot + OpenJDK 13 (41.41 milisekunda), Micronaut + OpenJDK 13 (28.4 milisekunde), Quarkus + OpenJDK 13 (34.1 milisekunda) i Quarkus + GraalVM (7.56 milisekundi). Iz rezultata se vidi da je aplikacija napravljena pomoću Quarkus-a i GraalVM-a 81.74% brža od iste implementacije napravljene u Spring Boot-u. Iz rezultata se vidi da je aplikacija kompilirana pomoću GraalVM-a signifikantno brža i ima signifikantno manje vrijeme odaziva. [11]

3. Praktični rad

U ovom poglavlju je objašnjen način na koji je napravljen praktični dio ovog rada te kako su primijenjene tehnologije i tehnike izvornih aplikacija i arhitektura za oblak. Također su napomenuti neki problemi i vrlo korisna rješenja prilikom izrade rada. Također su objašnjeni svi servisi pojedinačno, sa svim svojim funkcionalnostima, ER dijagramom baze podataka servisa i Docker i Kubernetes kodom za isporuku.

3.1. Opis

Napravljena je web aplikacija za upoznavanje ljudi pomoću arhitekture za oblak pod nazivom „Finder“. Glavna funkcija aplikacije je upoznavanje ljudi pa prema tome aplikacija podržava funkcionalnosti poput pregledavanja ostalih korisnika prema određenim kriterijima koje korisnik sam može modificirati te čavrljanje dvaju korisnika. Da bi korisnici mogli započeti čavrljanje, prvobitno se moraju jedan drugom sviđati, to jest oba korisnika moraju potvrditi sviđanje međusobno. Da bi korisnici mogli potvrditi sviđanje, mogu vidjeti ostale korisnike koji su sortirani prema udaljenosti od najbližeg prema najdaljem, također se uzimaju u obzir i preferencije godina koje korisnik sam može odabrati. Korisnik prije nego odluči može vidjeti dodatne informacije o drugom korisniku te se kasnije vratiti i odlučiti da li mu se drugi korisnik sviđa ili ne. Korisnik može urediti svoje informacije poput korisničkog imena, imena (ime koje se prikazuje drugim korisnicima), preferencije godina, datuma rođenja, slika te opisa i zanimanja.

3.2. Arhitektura

Arhitektura je napravljena prema standardima izvornih aplikacija i arhitektura za oblak. Da bi se ova arhitektura ostvarila korištene su tehnologije poput Docker-a, Kubernetes-a i RabbitMQ-a. Korišteni su HTTP i RPC protokoli. Arhitektura je podijeljena na pet manjih neovisnih servisa, jedan API pristupnik koji je ujedno i ulazna točka te aplikacijski servis koji servira web aplikaciju. Svaki od pet manjih servisa je neovisan servis koji može raditi sam za sebe. Svaki servis ima svoju bazu podataka dok API pristupnik i aplikacijski servis nemaju jer im nije potrebna. Da bi se ostvarila interna komunikacija u sustavu korišten je RabbitMQ posrednik poruka. Svaki servis otvara svoj red u posredniku poruka i čeka na nadolazeće

poruke koje potom obrađuje i vraća podatke u uzvratni red. Uzvratni red je red u posredniku poruka koji je privremen i koristi se samo kako bi klijent koji radi poziv na određeni servis mogao čekati na odgovor servisa. Time je ostvaren takozvani RPC protokol. Ulazna točka arhitekture je API pristupnik, što znači da servisi rade u pozadini sustava i vanjski klijenti ne mogu doći do njih direktno. Sva vanjska komunikacija obavlja se putem HTTP protokola od vanjskog klijenta do API pristupnika i potom API pristupnik zove određene servise koji su mu potrebni da formira odgovor na korisnički zahtjev. Tim pristupom se ostvarila i dodatna sigurnost sustava jer se autentifikacija korisnika obavlja na jednom mjestu i servisi nemaju nikakve odgovornosti u vezi sigurnosti. Sada se postavlja pitanje kako je to moguće ako postoji autentifikacijski servis? Autentifikacijski servis obavlja registraciju korisnika i upravlja autentifikacijskim stanjem korisnika u bazi, no sama verifikacija i provjera korisničkih žetona se obavlja odmah na API pristupniku kako bi se izbjegao poziv autentifikacijskog servisa kod svakog zaštićenog poziva na API pristupnik. API pristupnik i autentifikacijski servis zajedno dijele lozinku za potpisivanje pristupnog žetona kako bi autentifikacijski servis mogao prijaviti korisnika i kreirati potpisane žetone, a s istom lozinkom API pristupnik može potvrditi da su žetoni valjani.

Da bi servisi mogli neovisno raditi, svaki servis ima svoju konfiguracijsku datoteku, svoju bazu podataka i kreirana je Docker slika svakog servisa kako bi se kasnije servis mogao pokrenuti i u svom zasebnom kontejneru. Svaki servis također ima poseban repozitorij na Git-u kako bi se neovisno mogao verzionirati kod. Da bi se osigurala dosljednost podataka, verzioniranje same baze i olakšalo ažuriranje strukture baze podataka, napravljena je automatska migracija stanja baze podataka. Migracije su ostvarene kroz SQL skripte. Kod pokretanja servisa, servis se spaja na bazu podataka i osigurava da postoji potrebna shema, potom kreira tablicu koja verzionira migracije i pokreće SQL skriptu koja kreira sve potrebne objekte u bazi. SQL skripta može kreirati tablice, relacije, indekse, poglede itd. i inicijalno napuniti bazu podataka nekim inicijalnim, osnovnim podacima za rad servisa. Tablica verzioniranja migracija prati koje su migracije izvršene nad bazom podataka i njih preskače. Da bi se izvršila nova migracija stanja baze podataka, dovoljno je napraviti novu skriptu sa sljedećom sekvencijalnom vrijednošću. Osnovni način rada automatike migracije je da servis učita sve migracijske datoteke, za svaku datoteku provjerava da li je već izvršena, tj. čita iz tablice verzioniranja migracija, ako je migracija već izvršena onda ju preskače, ako nije izvršena onda ju izvršava te zapisuje u tablicu verzioniranja migracija. Proces se ponavlja prilikom svakog pokretanja servisa. Time se lako mogu raditi izmjene nad stanjem objekata u bazi podataka (tablica, podataka, procedura, indeksa, pogleda, relacija) ako server baze

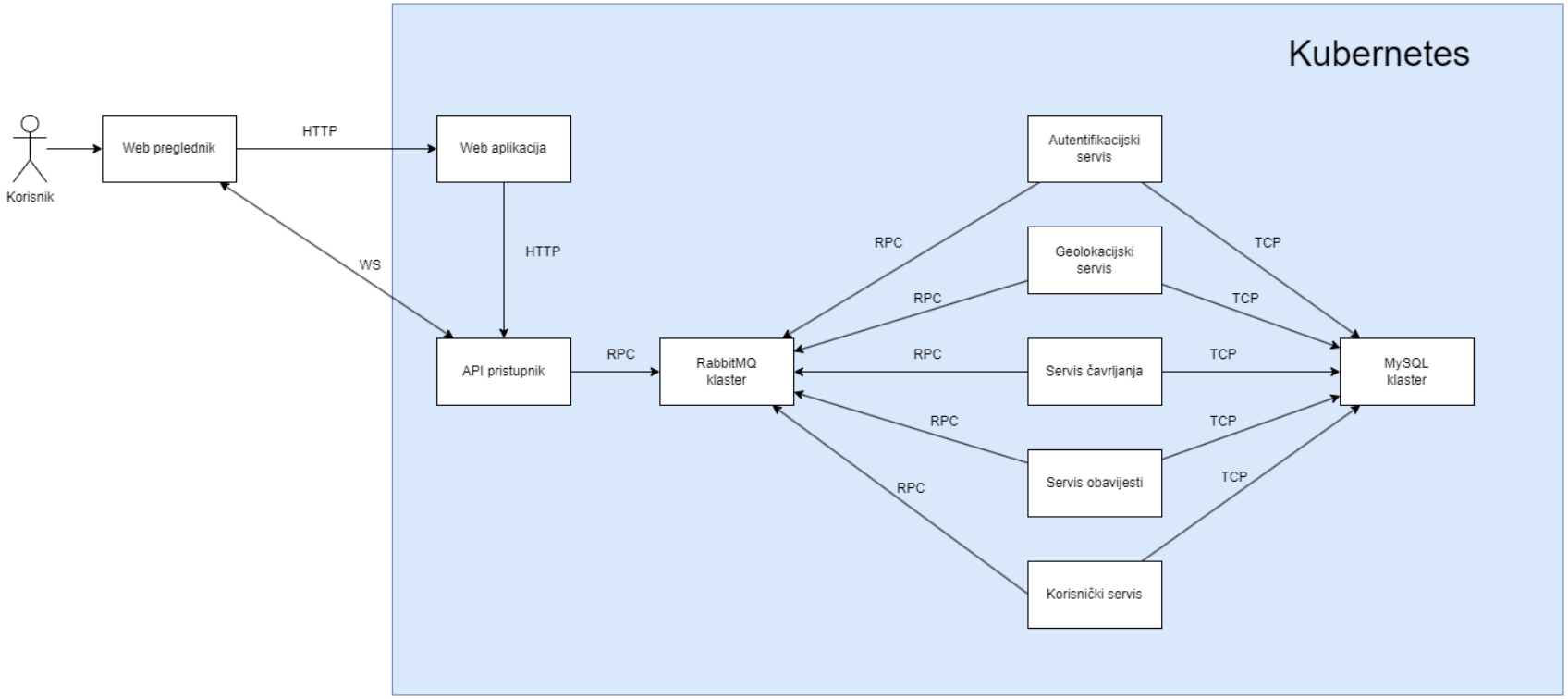
podataka radi negdje u oblaku i pogotovo ako nemamo direktan pristup do njega, poput otvaranja SQL ljuske server-a.

Kroz cijeli sustav koristi se UUID (eng. *Universally Unique Identifier*) kao identifikacijski tip podataka kako bi se osigurala jedinstvenost svih entiteta u sustavu.

Svaki servis ima svoju konfiguracijsku datoteku koju učitava prilikom pokretanja. Možda se to čini nepotrebno, jer se vrijednosti poput korisničkog imena i lozinke za bazu podataka mogu direktno navesti u kodu, ali ovaj pristup uvelike olakšava promjenu tih vrijednosti kada je aplikacija pokrenuta u oblaku. Kada bi imali takve podatke navedene direktno u kodu, ako se slučajno, na primjer, promjeni lozinka baze podataka, programer bi morao promijeniti kod, kompilirati aplikaciju, napraviti izvršnu verziju, kreirati Docker sliku, zaustaviti kontejner u oblaku i ponovno isporučiti servis u oblak. Ukoliko su vrijednosti sadržane u konfiguracijskoj datoteci, dovoljno je samo promijeniti vrijednosti u datoteci i ponovno pokrenuti kontejner u oblaku.

Docker slike omogućavaju samostalnost servisa prilikom rada i uvelike povećavaju otpornost i robusnost sustava. Svaki servis ima svoju Docker sliku i vrti se u svom zasebnom kontejneru. U radu se koriste slike operacijskog sustava Alpine Linux s predinstaliranim Java 22 razvojnim okruženjem. Alpine Linux su nešto manje slike operacijskog sustava Linux, tj. manje od Ubuntu i Debian verzija u smislu koliko memorije zauzimaju, i prilagođene su baš za rad s mikro servisima.

Da bi se cijeli ovaj sustav uskladio, korišten je orkestrator kontejnera Kubernetes. Svi servisi, API pristupnik i web aplikacija isporučeni su u Kubernetes sustav. Korišten je Rancher Desktop koji je besplatna lokalna verzija Kubernetes sustava s dodatnim grafičkim sučeljem radi lakšeg upravljanja sustavom. Servisi su isporučeni kao obične Java aplikacije koje se spajaju na bazu podataka i posrednik poruka i nisu izloženi niti prema van, niti u internoj Kubernetes mreži. To znači da se nitko ne može spojiti direktno na servis, već se sva komunikacija odvija preko posrednika poruka. API pristupnik i web aplikacija su isporučeni kao web Java servisi i oni su dostupni izvana i unutar Kubernetes mreže. Da bi se API pristupnik i web aplikacija izložili, korišten je usmjerivač opterećenja koji u Kubernetes sustav može dobiti vanjsku IP adresu i biti dostupan izvan Kubernetes mreže.



Slika 11. Arhitektura sustava

3.3. Autentifikacijski servis

Odgovornosti autentifikacijskog servisa su registracija korisnika, prijava korisnika, odjava korisnika, obnova žetona te dostavljanje lozinki za potpisivanje žetona i ostalih informacija koje API pristupnik zahtijeva. Kako bi se ostvarila maksimalna sigurnost koriste se JWT (eng. *JSON Web Token*) žetoni. Svaki korisnik se mora registrirati u sustav i time unosi svoje autentifikacijske podatke u ovaj servis, poput korisničkog imena i lozinke. Proces prijave se odvija na način da korisnik šalje svoje korisničko ime i lozinku, servis provjerava valjanost podataka i kreira pristupni žeton, žeton osvježavanja i slučajan UUID koji predstavlja obitelj žetona tog korisnika. Servis svaki žeton potpisuje sa svojom lozinkom. Pristupni žeton traje pet minuta jer je on manje siguran i ispravno ga je češće obnavljati dok žeton osvježavanja traje sedam dana. Pristupni žeton služi za osnovnu autentifikaciju korisnika, dok žeton osvježavanja služi kako bi korisnik mogao osvježiti, tj. dobiti nove žetone. Kada korisnikov pristupni žeton istekne, API pristupnik će reći da žeton nije valjan. Tada korisnik ima opciju slanja žetona osvježavanja i dobiva nove žetone ukoliko mu je žeton osvježavanja još uvijek validan, tj. nije istekao i nije krivotvoren. Kako bi se osigurala maksimalna sigurnost, radi se već objašnjena rotacija žetona te se stari žeton osvježavanja sprema u crnu listu. Crna lista je dodatna pomoć budući da se potpisani žetoni ne mogu mijenjati i time je žeton koji je potpisan na sedam dana zapravo i valjan svih sedam dana. Ovom tehnikom stavljamo valjani žeton na crnu listu i tako se može znati da žeton više nije valjan. Servis koristi i tehniku obitelji žetona. Time se osigurava od krađe žetona i automatskoj odjavi svih uređaja koji koriste korisnički račun koji je napadnut. Problem nastaje kada napadač ukrade žeton osvježavanja i odmah obavi radnju osvježavanja žetona te dobije nove valjane žetone. Kada korisnik pokuša napraviti sljedeće osvježavanje žetona, servis će prepoznati već iskorišteni žeton osvježavanja koji je na crnoj listi i razumjeti da je nešto pošlo po krivu. Servis tada zaključava korisnički račun na način da briše korisnikovu obitelj i ne prihvaća nikakve načine autentifikacije osim prijave putem korisničkog imena i lozinke, koji se podrazumijevaju da su samo u posjedu pravog korisnika. Kada se korisnik ponovno prijavi u sustav, generira se nova obitelj žetona i svi budući žetoni biti će potpisani s tom obitelji. Time se osigurava da svi preostali žetoni, ujedno i ukradeni budu neispravni jer su potpisani krivom obitelji žetona. Odjava iz sustava se obavlja na način da servis žeton osvježavanja stavlja na crnu listu. Za sažimanja lozinke koristi se algoritam Argon2 i njegova implementacija u Java-i. Algoritam ovisno o postavkama daje izlazni tekst koji sadrži osnovne postavke te sažetak lozinke koji je uvijek iste dužine, npr. „$argon2id$v=19$m=32000,t=22,p=1$vaHXFr75D/U7X8z/Imj1UQ$tEbFdF10KMXuLeE+P0ZYKMa9YPCNqBgDSX6PC3eHKU0“. Prvi podatak označava vrstu Argon2 algoritma (u ovom slučaju „argon2id“), „v“ označava Argon2 verziju, „m“ označava koliko će algoritam trošiti radne

memorije u Kibibajtima, „t“ označava broj iteracija te „p“ sam sažetak lozinke koji je kreiran. Objašnjenje Argon2 algoritma, sažetka i izlaznog teksta je dosta bitno u daljnjem tekstu koji objašnjava strukturu baze podataka.

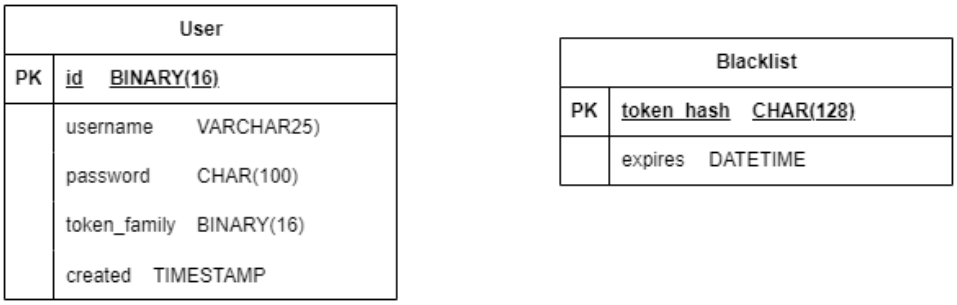
Struktura baze podataka ovog servisa sadrži dvije tablice: „User“ koja predstavlja korisnika i „Blacklist“ koja predstavlja crnu listu žetona. Tablica „User“ sastoji se od sljedećih stupaca:

- „id“ – jedinstveni identifikator korisnika, tipa „BINARY (16)“. To je binarni tip podatka veličine 16 u koji se sprema UUID unutar MySQL baze podataka u binarnom obliku.
- „username“ - korisničko ime, tipa „VARCHAR (25)“
- „password“ – lozinka korisnika, tipa „CHAR (100)“ koje je očekivana dužina Argon2 izlaznog teksta nakon sažimanja lozinke
- „token_family“ – familija žetona, tipa „BINARY (16) u koji se sprema UUID i time osigurava da je za svakog korisnika ovaj podatak jedinstven
- „created“ – vrijeme registracije korisnika, tipa „TIMESTAMP“

Tablica „Blacklist“ sastoji se od sljedećih stupaca:

- „token_hash“ – sažetak žetona sažet pomoću SHA512 algoritma radi uštede prostora, tipa „CHAR (128)“ jer navedeni algoritam sažimanja uvijek daje izlaznu vrijednost od 128 znakova
- „expires“ – vrijeme kada žeton bude istekao, tipa „DATETIME“. Stupac se koristi radi uštede prostora jer kada žeton istekne, sam servis će to prepoznati prilikom validacije žetona i odmah će vratit odgovor bez da radi poziv na bazu podataka. To znači da se žeton može slobodno obrisati iz tablice radi uštede prostora.

Na sljedećoj slici je prikazan ER dijagram baze podataka autentifikacijskog servisa:



Slika 12. ER dijagram baze podataka autentifikacijskog servisa

Kod za izradu Docker slike je nešto drugačiji za ovaj servis nego za ostale. Problem je što ovaj servis koristi Argon2 algoritam za sažimanje lozinki. Da bi algoritam radio unutar kontejnera, potrebno ga je samostalno instalirati. Izrada ove slike podijeljena je u dvije faze: 1. Instalacija Argon2 algoritma i 2. Instalacija servisa. Kod se sastoji od sljedećih koraka:

- Linija 1 – dohvaćanje Alpine Linux slike s instaliranim Java 22 razvojnim okruženjem kao osnova kontejnera (baza, bazna slika), započinje faza 1
- Linija 3 – instaliranje svih potrebnih alata za dohvaćanje i instalaciju Argon2 algoritma za sažimanje
- Linija 11 – dohvaćanje Argon2 repozitorija s Git sustava
- Linija 13 – postavljanje trenutnog direktorija na prethodno dohvaćeni repozitorij
- Linija 14 – kompiliranje Argon2 izvornog koda i instalacija
- Linija 17 – ponovno dohvaćanje Alpine Linux slike, započinje faza 2
- Linija 19 – kopiranje instalirane knjižnice za Argon2 algoritam u novo dohvaćenu Alpine Linux sliku
- Linija 21 – promjena direktorija unutar bazne slike
- Linija 23 – kopiranje izvršne datoteke servisa s lokalnog računala u baznu sliku
- Linija 24 – kopiranje produkcijske konfiguracijske datoteke s lokalnog računala u baznu sliku
- Linija 25 – kopiranje migracijskih SQL datoteka s lokalnog računala u baznu sliku

- Linija 27 – osnovna naredba za pokretanje servisa

Sljedeći kod prikazuje izradu Docker slike autentifikacijskog servisa:

```
1 FROM alpine/java:22 AS builder
2
3 RUN apk add --no-cache \
4 git \
5 build-base \
6 autoconf \
7 automake \
8 libtool \
9 make
10
11 RUN git clone https://github.com/P-H-C/phc-winner-argon2.git
12
13 RUN cd phc-winner-argon2 && \
14 make && \
15 make PREFIX=/usr install
16
17 FROM alpine/java:22
18
19   COPY   --from=builder   /usr/lib/x86_64-linux-gnu/libargon2.so
/usr/local/lib/
20
21 WORKDIR /app
22
23 COPY target/auth-service-1.0.0.jar /app/app.jar
24 COPY app.production.config /app/app.config
25 COPY db/migrations /app/db/migrations
26
27 ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

Ovaj servis isporučen je bez servisa za otkrivanje. Isporučen je kao tip „Deployment“ s jednom replikom, oznakom „auth-service“ i dodijeljeno mu je 512 MB radne memorije te „500m“ CPU-a što znači da mu je dodijeljeno 50% jedne jezgre procesora.

Sljedeće je prikazan kod za isporuku autentifikacijskog servisa u Kubernetes:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: auth-service
5    namespace: default
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
```

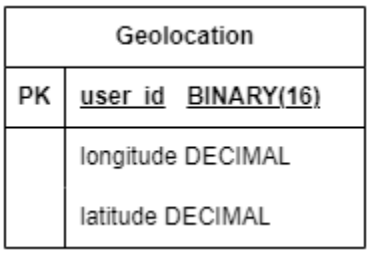
```
10        app: auth-service
11    template:
12      metadata:
13        labels:
14          app: auth-service
15      spec:
16        containers:
17        - name: auth-service
18          image: localhost:5000/finder/auth-service:latest
19          resources:
20            requests:
21              memory: "512Mi"
22              cpu: "500m"
23            limits:
24              memory: "1Gi"
25              cpu: "1"
```

3.4. Geolokacijski servis

Odgovornosti geolokacijskog servisa su vraćanje popisa svih najbližih korisnika od trenutačnog korisnika te računanje udaljenosti između dva korisnika. Servis sadrži geolokacijske podatke o svakom korisniku, točnije geografsku širinu i dužinu. Na zahtjev, izračunava udaljenost Harvesinusnom formulom.

Struktura baze podataka ovog servisa sadrži jednu tablicu „Geolocation“ koja predstavlja geolokacijsku poziciju korisnika. Tablica „Geolocation“ sastoji se od sljedećih stupaca:

- „user_id“ – jedinstveni identifikator korisnika, tipa „BINARY (16)“
- „longitude“ – geografska dužina, tipa „DECIMAL“
- „latitude“ – geografska širina, tipa „DECIMAL“



Slika 13. ER dijagram baze podataka geolokacijskog servisa

Kod za izradu Docker slike geolokacijskog servisa se sastoji od sljedećih koraka:

- Linija 1 - dohvaćanje Alpine Linux slike s instaliranim Java 22 razvojnim okruženjem kao osnova kontejnera
- Linija 4 – Smoke test za provjeru java okruženja
- Linija 6 - promjena direktorija unutar bazne slike
- Linija 8 – kopiranje izvršne datoteke servisa s lokalnog računala u baznu sliku
- Linija 9 – kopiranje produkcijske konfiguracijske datoteke s lokalnog računala u baznu sliku
- Linija 10 – kopiranje migracijskih SQL datoteka s lokalnog računala u baznu sliku
- Linija 11 – osnovna naredba za pokretanje servisa

```
1    FROM alpine/java:22 AS builder
2
3    # Smoke test za provjeru jave
4    RUN java -version
5
6    WORKDIR /app
7
8    COPY target/geolocation-service-1.0.0.jar /app/app.jar
9    COPY app.production.config /app/app.config
10   COPY db/migrations /app/db/migrations
11   ENTRYPOINT ["java","-jar","/app/app.jar"]
```

Ovaj servis isporučen je bez servisa za otkrivanje. Isporučen je kao tip „Deployment“ s dvije replike, oznakom „geolocation-service“ i dodijeljeno mu je 256 MB radne memorije te „500m“ CPU-a. Servisu trenutno nije dodijeljeno puno resursa jer se očekuje manji broj korisnika na početku. S povećanjem broja korisnika, izračun udaljenosti između korisnika zahtijeva više resursa, pa će se servis u budućnosti postepeno skalirati. Skaliranje se obavlja na način da mu se dodijeli više jezgri procesora i više memorije te se mogu napraviti i dodatne replike. Sljedeće je prikazan kod za isporuku geolokacijskog servisa u Kubernetes:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: geolocation-service
5    namespace: default
6  spec:
7    replicas: 2
8    selector:
9      matchLabels:
10       app: geolocation-service
```

```
11    template:
12      metadata:
13        labels:
14          app: geolocation-service
15      spec:
16        containers:
17        - name: geolocation-service
18          image: localhost:5000/finder/geolocation-service:latest
19          resources:
20            requests:
21              memory: "256Mi"
22              cpu: "500m"
23            limits:
24              memory: "256Mi"
25              cpu: "500m"
```

3.5. Korisnički servis

Odgovornosti korisničkog servisa su održavanje stanja korisničkih informacija i veza između korisnika. Servis pohranjuje korisničke informacije poput imena, datuma rođenja, interesa, zanimanja itd. Servis je također odgovoran za održavanje stanja veza između korisnika u smislu da sadrži informacije koji korisnik se kome sviđa. „Podudaranje“ se događa kada se oba korisnika međusobno sviđaju jedan drugome. Tada se omogućuje opcija čavrljanja. Servis ima funkciju filtriranja korisnika prema određenim kriterijima.

Struktura baze podataka ovog servisa sadrži pet tablica: „User“ koja predstavlja korisnika, „Interest“ koja predstavlja predefinirane interese, „Media“ koja predstavlja slike korisnika, „Match_status“ koja predstavlja podatke o sviđanjima i tablicu spajanja „User_interest“ koja predstavlja interese pojedinog korisnika. Tablica „User“ sastoji se od sljedećih stupaca:

- „id“ – jedinstveni identifikator korisnika, tipa „BINARY (16)“
- „name“ – ime korisnika, tipa „VARCHAR(50)“
- „sex“ – spol korisnika, tipa „CHAR(1)“
- „profile_picture“ – adresa slike profila korisnika, tipa „VARCHAR (500)“
- „created“ – vrijeme registracije, tipa „TIMESTAMP“
- „birth_date“ – datum rođenja, tipa „DATE“
- „age_interest_min“ – najmanji broj godina za koje je korisnik zainteresiran kod pretrage, tipa „INT“
- „age_interest_max“ – najveći broj godina za koje je korisnik zainteresiran kod pretrage, tipa „INT“

- „description“ – opis ili zanimljivosti o korisniku, tipa „VARCHAR (500)“
- „occupation“ – zanimanje korisnika, tipa „VARCHAR (50)“

Tablica „Interest“ je predefinirana tablica (ima unesene podatke, korisnici ne mogu mijenjati podatke u tablici) i sastoji se od sljedećih stupaca:

- „id“ – identifikator, tipa „INT“
- „name“ – naziv, tipa „VARCHAR (25)“

Tablica „User_interest“ je spojna tablica koja predstavlja vezu više prema više između korisnika i interesa. Tablica se sastoji od sljedećih stupaca:

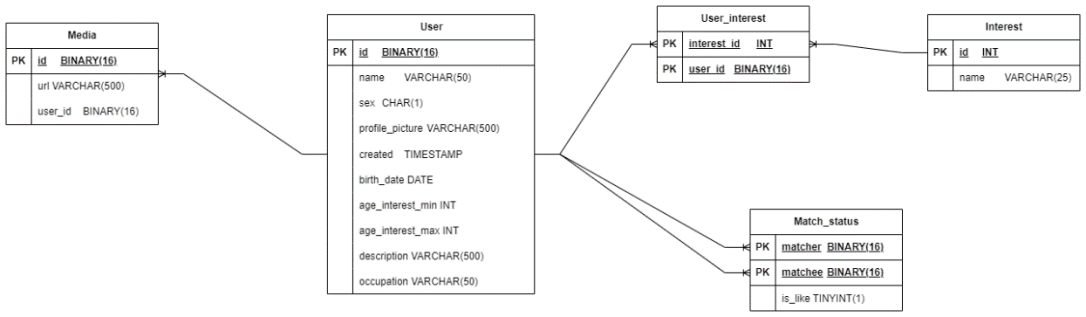
- „interest_id“ – identifikator interesa, tipa „INT“
- „user_id“ – identifikator korisnika, tipa „BINARY (16)“

Tablica „Media“ sastoji se od sljedećih stupaca:

- „id“ – identifikator, tipa „BINARY (16)“
- „url“ – web putanja do slike, tipa „VARCHAR (500)“
- „user_id“ – identifikator korisnika, tipa „BINARY (16)“

Tablica „Match_status“ sastoji se od sljedećih stupaca:

- „matcher“ – identifikator korisnika koji šalje zahtjev, tipa „BINARY (16)“
- „matchee“ – identifikator korisnika za kojeg se šalje zahtjev, tipa „BINARY (16)“
- „is_like“ – logička vrijednost koji govori da li se radi o sviđanju ili ne sviđanju, tipa „TINYINT (1)“



Slika 14. ER dijagram baze podataka korisničkog servisa

Kod za izradu Docker slike korisničkog servisa se sastoji od sljedećih koraka:

- Linija 1 - dohvaćanje Alpine Linux slike s instaliranim Java 22 razvojnim okruženjem kao osnova kontejnera
- Linija 4 – Smoke test za provjeru java okruženja
- Linija 6 - promjena direktorija unutar bazne slike
- Linija 8 – kopiranje izvršne datoteke servisa s lokalnog računala u baznu sliku
- Linija 9 – kopiranje produkcijske konfiguracijske datoteke s lokalnog računala u baznu sliku
- Linija 10 – kopiranje migracijskih SQL datoteka s lokalnog računala u baznu sliku
- Linija 11 – osnovna naredba za pokretanje servisa

```
1  FROM alpine/java:22 AS builder
2
3  # Smoke test za provjeru java verzije
4  RUN java -version
5
6  WORKDIR /app
7
8  COPY target/user-service-1.0.0.jar /app/app.jar
9  COPY app.production.config /app/app.config
10 COPY db/migrations /app/db/migrations
11 ENTRYPOINT ["java","-jar","/app/app.jar"]
```

Ovaj servis isporučen je bez servisa za otkrivanje. Isporučen je kao tip „Deployment“ s tri replike, oznakom „user-service“ i dodijeljeno mu je 256 MB radne memorije te „500m“ CPU-a. Sljedeće je prikazan kod za isporuku korisničkog servisa:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: user-service
5    namespace: default
6  spec:
7    replicas: 3
8    selector:
9      matchLabels:
10       app: user-service
11   template:
12     metadata:
13       labels:
14         app: user-service
15     spec:
16       containers:
17       - name: user-service
18         image: localhost:5000/finder/user-service:latest
19         resources:
20           requests:
21             memory: "256Mi"
22             cpu: "500m"
23           limits:
24             memory: "256Mi"
25             cpu: "500m"
```

3.6. Servis čavrljanja

Odgovornosti servisa čavrljanja su sadržavanje poruka između korisnika. Servis također ima kompleksnu logiku dohvaćanja zadnjih korisnikovih sesija kronološki prema datumima poruka. Servis ima funkcionalnost vraćanja svih korisnikovih poruka u sesiji kronološki od zadnje poruke prema prvoj, koja je napravljena pomoću rekurzivne CTE tehnike (eng. *Recursive CTE*) u kojoj svaka poruka sadrži i identifikator „roditelja“, tj. poruke koja je kronološki bila prije trenutne poruke i pomoću rekurzivne funkcije lako je dohvatiti kronološki potrebne poruke. Ova funkcionalnost također podržava straničenje. Rekurzivni način uvelike ubrzava proces kronološkog dohvaćanja poruka jer se izbjegava „skupo“ sortiranje poruka po datumu.

Struktura baze podataka ovog servisa sadrži tri tablice: „Message“ koja predstavlja poruke, „Session“ koja predstavlja sesije i „User_session“ koja je spojna tablica između korisnika i sesija. Tablica „Message“ sastoji se od sljedećih stupaca:

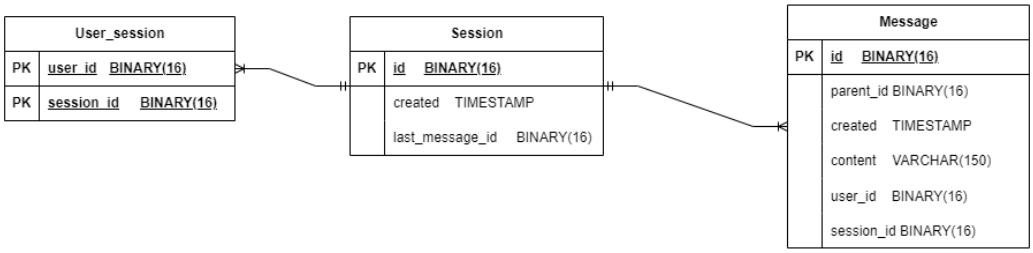
- „id“ – jedinstveni identifikator poruke, tipa „BINARY (16)“
- „parent_id“ – identifikator kronološki prethodne poruke, tipa „BINARY (16)“
- „created“ – vrijeme slanja poruke, tipa „TIMESTAMP“
- „content“ – sadržaj poruke, tipa „VARCHAR (150)“
- „user_id“ – identifikator korisnika pošiljatelja, tipa „BINARY (16)“
- „session_id“ – identifikator sesije, tipa „BINARY (16)“

Tablica „Session“ sastoji se od sljedećih stupaca:

- „id“ – identifikator sesije, tipa „BINARY (16)“
- „created“ – vrijeme kreiranja sesije, tipa „TIMESTAMP“
- „last_message_id“ – identifikator kronološki zadnje poruke, tipa „BINARY (16)“

Tablica „User_session“ se sastoji od sljedećih stupaca:

- „user_id“ – identifikator korisnika, tipa „BINARY (16)“
- „session_id“ – identifikator sesije, tipa „BINARY (16)“



Slika 15. ER dijagram baze podataka servisa čavrljanja

Kod za izradu Docker slike servisa čavrljanja se sastoji od sljedećih koraka:

- Linija 1 - dohvaćanje Alpine Linux slike sa instaliranim Java 22 razvojnim okruženjem kao osnova kontejnera
- Linija 4 – Smoke test za provjeru java okruženja

- Linija 6 - promjena direktorija unutar bazne slike
- Linija 8 – kopiranje izvršne datoteke servisa s lokalnog računala u baznu sliku
- Linija 9 – kopiranje produkcijske konfiguracijske datoteke s lokalnog računala u baznu sliku
- Linija 10 – kopiranje migracijskih SQL datoteka s lokalnog računala u baznu sliku
- Linija 11 – osnovna naredba za pokretanje servisa

```
1  FROM alpine/java:22 AS builder
2
3  # Smoke test za provjeru java verzije
4  RUN java -version
5
6  WORKDIR /app
7
8  COPY target/chat-service-1.0.0.jar /app/app.jar
9  COPY app.production.config /app/app.config
10 COPY db/migrations /app/db/migrations
11 ENTRYPOINT ["java","-jar","/app/app.jar"]
```

Ovaj servis isporučen je bez servisa za otkrivanje. Isporučen je kao tip „Deployment“ s dvije replike, oznakom „chat-service“ i dodijeljeno mu je 256 MB radne memorije te „500m“ CPU-a. Sljedeće je prikazan kod za isporuku servisa čavrljanja:

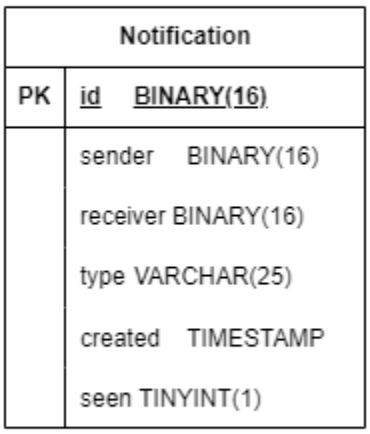
```
1  apiVersion: apps/v1
 2  kind: Deployment
 3  metadata:
 4    name: chat-service
 5    namespace: default
 6  spec:
 7    replicas: 2
 8    selector:
 9      matchLabels:
10        app: chat-service
11    template:
12      metadata:
13        labels:
14          app: chat-service
15      spec:
16        containers:
17        - name: chat-service
18          image: localhost:5000/finder/chat-service:latest
19          resources:
20            requests:
21              memory: "256Mi"
22              cpu: "500m"
23            limits:
24              memory: "256Mi"
25              cpu: "500m"
```

3.7. Servis obavijesti

Servis obavijesti ima funkcionalnosti pohranjivanja i dohvaćanja korisnikovih obavijesti prema određenim parametrima. Postoje dvije vrste obavijesti: 1. nova poruka i 2. novo sviđanje. Obavijest se kreira kada jedan korisnik pošalje poruku, a drugi korisnik nije na mreži. Ukoliko isti korisnik pošalje više poruke, obavijest će se ažurirati i neće se kreirati nove obavijesti za svaku novu poruku. Novo sviđanje kreira obavijest za korisnika koji je prethodno potvrdio da mu se sviđa trenutni korisnik. Korisnik može vidjeti svoje obavijesti i označiti ih kao pročitane.

Struktura baze podataka ovog servisa sadrži jednu tablicu „Notification“ koja predstavlja obavijesti. Tablica „Notification“ sastoji se od sljedećih stupaca:

- „id“ – jedinstveni identifikator obavijesti, tipa „BINARY (16)“
- „sender“ – korisnik koji je okinuo obavijest, tipa „BINARY (16)“
- „receiver“ – korisnik koji prima obavijest, tipa „BINARY (16)“
- „type“ – vrsta obavijesti, tipa „VARCHAR 25“
- „created“ – vrijeme kreiranja obavijesti, tipa „TIMESTAMP“
- „seen“ – status koji govori da li je obavijest viđena, tipa „TINYINT(1)“



Slika 16. ER dijagram baze podataka servisa obavijesti

Kod za izradu Docker slike servisa obavijesti se sastoji od sljedećih koraka:

- Linija 1 - dohvaćanje Alpine Linux slike s instaliranim Java 22 razvojnim okruženjem kao osnova kontejnera
- Linija 4 – Smoke test za provjeru java okruženja
- Linija 6 - promjena direktorija unutar bazne slike
- Linija 8 – kopiranje izvršne datoteke servisa s lokalnog računala u baznu sliku
- Linija 9 – kopiranje produkcijske konfiguracijske datoteke s lokalnog računala u baznu sliku
- Linija 10 – kopiranje migracijskih SQL datoteka s lokalnog računala u baznu sliku
- Linija 11 – osnovna naredba za pokretanje servisa

```
1  FROM alpine/java:22 AS builder
2
3  # Smoke test to check Java version
4  RUN java -version
5
6  WORKDIR /app
7
8  COPY target/notification-service-1.0.0.jar /app/app.jar
9  COPY app.production.config /app/app.config
10 COPY db/migrations /app/db/migrations
11 ENTRYPOINT ["java","-jar","/app/app.jar"]
```

Ovaj servis isporučen je bez servisa za otkrivanje. Isporučen je kao tip „Deployment“ s jednom replikom, oznakom „notification-service“ i dodijeljeno mu je 256 MB radne memorije te „500m“ CPU-a. Sljedeće je prikazan kod za isporuku servisa obavijesti u Kubernetes:

```
 1  apiVersion: apps/v1
 2  kind: Deployment
 3  metadata:
 4    name: notification-service
 5    namespace: default
 6  spec:
 7    replicas: 1
 8    selector:
 9      matchLabels:
10        app: notification-service
11    template:
12      metadata:
13        labels:
14          app: notification-service
15      spec:
16        containers:
17        - name: notification-service
```

```
18          image: localhost:5000/finder/notification-service:latest
19          resources:
20            requests:
21              memory: "256Mi"
22              cpu: "500m"
23            limits:
24              memory: "256Mi"
25              cpu: "500m"
```

3.8. API pristupnik

API pristupnik je ulazna točka prema servisima. Sav promet mora proći kroz njega. Također je odgovoran za autentifikaciju korisnika koji pozivaju zaštićene rute. Napravljen je pomoću REST API arhitekture i koristi HTTP protokol te je baziran na JAX-RS specifikaciji. Dijeli zajedničku tajnu za potpisivanje žetona s autentifikacijskim servisom. Time se validacija žetona prebacuje na API pristupnik i smanjuje se jedan poziv manje na servise, točnije na autentifiakcijski servis, koji bi morao biti pozvan na svakoj zaštićenoj ruti radi provjere validnosti žetona. API se sastoji od 5 kontrolera: autentifikacijski kontroler, kontroler čavrljanja, kontroler zdravlja, kontroler obavijesti i korisnički kontroler. Može se primijetiti kako u sustavu postoji geolokacijski servis, no nema geolokacijskog kontrolera. To je zato što je za potrebe ovog sustava geolokacijski kontroler nepotreban jer kontroleri agregiraju više servisa, pa ujedno i geolokacijski servis. Sljedeće će biti objašnjeni kontroleri, tj. njihovi nazivi, verzije te bazične i putanje krajnjih točaka i HTTP metoda svake putanje krajnje točke.

Krajnje točke autentifikacijskog kontrolera su:

- Bazična putanja „/v1/auth“ – verzija i naziv kontrolera u putanji
- POST „/signup“ – registracija korisnika
- POST „/login“ – prijava korisnika
- POST „/logout“ – odjava korisnika
- POST „/refresh“ – osvježivanje žetona
- PATCH „/“ – ažuriranje korisničkih informacija
- GET „/checkUsername“ – provjera da li korisničko ime već postoji u bazi podataka

Krajnje točke kontrolera čavrljanja su:

- Bazična putanja „/v1/chat“ – verzija i naziv kontrolera u putanji
- WS „/“ – WebSocket krajnja točka za komunikaciju u stvarnom vremenu

Krajnje točke kontrolera zdravlja su:

- Bazična putanja „/v1/health“ – verzija i naziv kontrolera u putanji
- GET „/“ – jednostavna putanja za provjeru zdravlja servisa, vraća HTTP status 200 OK, koristi se za poziv samo da se vidi da li je API „živ“, može se iskoristiti u Kuberenetes-u gdje će Kubernetes sustav automatski periodično zvati API i provjeravati zdravlje servisa.

Krajnje točke kontrolera obavijesti su:

- Bazična putanja „/v1/notifications“ - verzija i naziv kontrolera u putanji
- GET „/“ – dohvaća korisnikove obavijesti
- POST „/lock“ – postavlja određenu korisnikovu obavijest na status „viđena“

Krajnje točke korisničkog kontrolera su:

- Bazična putanja "/v1/users" – verzija i naziv kontrolera u putanji, podržava straničenje
- GET „/index“ – dohvaća korisnike za početnu stranicu aplikacije
- POST „/match“ – upisuje status korisnikovog sviđanja drugog korisnika
- GET „/“ – dohvaća osnovne informacije o nekom korisniku
- GET „/self“ – dohvaća osnovne informacije o trenutnom korisniku
- GET „/interests“ – dohvaća predefinirane vrijednosti interesa
- GET „/chats“ – dohvaća korisnikove sesije kronološki sortirane prema zadnjoj poruci u sesiji
- GET „/messages“ – dohvaća poruke pojedine sesije, poruke su kronološki sortirane od zadnje prema prvoj, podržava straničenje
- PATCH „/“ – ažuriranje osnovnih korisničkih informacija
- PATCH „/interests“ – ažuriranje korisnikovih interesa
- PATCH „/media“ – ažuriranje korisnikovih slika

Kod za izradu Docker slike je vrlo jednostavan jer se koristi „payara/micro“ bazna slika koje je upravo prilagođena za pokretanje Jakarta EE aplikacija i dolazi s instalacijom svih potrebnih ovisnosti tako da nema nikakvog ručnog instaliranja ovisnosti. Sastoji se od sljedećih koraka:

- Linija 1 – dohvaćanje „payara/micro“ docker slike kao baze kontejnera
- Linija 3 – promjena direktorija unutar kontejnera
- Linija 5 - kopiranje izvršne datoteke aplikacije s lokalnog računala u baznu sliku

- Linija 7 – otvaranje porta 8080 na kojem je izložen Payara Micro poslužitelj
- Linija 9 – komanda za pokretanje poslužitelja koja također navodi isporuku aplikacije prilikom pokretanja

```
1  FROM payara/micro
2
3  WORKDIR /opt/payara
4
5  COPY target/finder-api.war /opt/payara/deployments/
6
7  EXPOSE 8080
8
9    ENTRYPOINT  ["java",  "-jar",  "payara-micro.jar",  "--deploy",
"/opt/payara/deployments/finder-api.war", "--noCluster"]
```

API pristupnik se isporučuje u Kubernetes kroz dva procesa: 1. Isporuka aplikacije i 2. Isporuka servisa za otkrivanje. Isporuka aplikacije radi se s tipom „Deployment“ dok se isporuka servisa za otkrivanje radi tipom „Service“. Da bi te dvije isporuke bile sinkronizirane, potrebno je paziti na oznake pojedine isporuke. Kada isporuku aplikacije označimo nekom oznakom, servis za otkrivanje će tražiti aplikaciju po toj oznaci. API pristupnik je isporučen sa tipom „Deployment“, napravljena je jedna replika, ima oznaku „finder-api-gateway“, koristi 512 MB radne memorije i „500m“ CPU-a, te je izložen port 8080, tj. isti port na kojem radi aplikacijski server unutar kontejnera. Servis za otkrivanje je isporučen s tipom „Service“ i specifičnije tipom „Load Balancer“. Vidi se da se oznaka isporuke API pristupnika i oznaka u servisu za otkrivanje podudaraju. Servis za otkrivanje prosljeđuje promet na port 8080 u kontejner API pristupnika, tj. isti port koji je izložen u kontejneru API pristupnika. Servis za otkrivanje koristi port 8080 na kojem prosljeđuje promet izvana u Kubernetes sustav.

Sljedeće je prikazan kod za isporuku API pristupnika u Kubernetes:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: finder-api-gateway-deployment
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: finder-api-gateway
```

```
10   template:
11     metadata:
12       labels:
13         app: finder-api-gateway
14     spec:
15       containers:
16       - name: finder-api-gateway-container
17         image: localhost:5000/finder/api-gateway:latest
18         ports:
19         - containerPort: 8080
20         resources:
21           requests:
22             memory: "512Mi"
23             cpu: "500m"
24           limits:
25             memory: "512Mi"
26             cpu: "500m"
```

Sljedeće je prikazan kod za isporuku servisa za otkrivanje API pristupnika u Kubernetes:

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: finder-api-gateway-service
5  spec:
6    selector:
7      app: finder-api-gateway
8    ports:
9      - protocol: TCP
10       port: 8080
11       targetPort: 8080
12    type: LoadBalancer
```

3.9. RPC protokol

RPC protokol ostvaren je preko RabbitMQ posrednika poruka. Posrednik je glavni entitet kroz kojeg prolazi sva interna komunikacija između API pristupnika i servisa. RPC protokol se bazira na klijentu koji šalje određeni zahtjev ili komandu nekom obliku poslužitelja i čeka na odgovor, tako da se RabbitMQ posrednik poruka savršeno može iskoristiti kao posrednik u RPC protokolu. RPC protokol je implementiran na način da se kao prvo „dogovori“ osnovni način kako će izgledati zahtjevi i odgovori, tj. najbolja praksa je imati osnovni objekt koji je jednak za sve zahtjeve i objekt koji je jednak za sve odgovore. Time se potiče standardizacija kod serijalizacije i deserijalizacije objekata i uvelike olakšava komunikaciju, što je velika

prednost i samim programerima koji znaju kakav zahtjev trebaju slati i kakav odgovor mogu očekivati. Sam proces jednog RPC zahtjeva sastoji se od sljedećih koraka:

- Servis se spaja na RabbitMQ klaster i otvara red u kojem očekuje zahtjeve
- Servis konstantno sluša na nadolazeće zahtjeve u redu
- Klijent se spaja na RabbitMQ klaster i kreira privremeni red u kojem čeka na odgovor
- Klijent šalje određeni zahtjev u određeni red i unutar zahtjeva nalazi se i jedinstveno ime privremenog reda
- Servis čita zahtjev iz reda, odrađuje ga i potom vraća odgovor u privremeni red
- Klijent zaprima odgovor iz privremenog reda, zatvara privremeni red i prekida vezu s RabbitMQ klasterom te nastavlja s daljnjim radom

Svaki zahtjev se sastoji od 2 atributa: „action“ i „payload“. Atribut „action“ označava akciju koju želi da servis odradi, a atribut „payload“ označava dodatne podatke koji su potrebni za izvršavanje akcije. Atribut "action" je obični tekst koji označava ime akcije dok je atribut „payload“ obično kompleksniji objekt koji sadrži više informacija pa moguće i liste.

Svaki odgovor se sastoji od 3 atributa: „error“, „errorDetails“ i „dana“. Atribut „error“ značava ključ ili kratki naziv odnosno skraćenicu greške koja se dogodila, ukoliko je atribut prazan, znači da se nije desila pogreška kod procesiranja zahtjeva. Atribut „errorDetails“ je dodatno objašnjenje greške, ukoliko se dogodila. Atribut „data“ sadrži potrebne podatke, ovisno o zahtjevu. Neki zahtjevi ne trebaju povratne podatke pa je atribut prazan. Atribut je također prazan ako se dogodila pogreška. Atributi „error“ i „data“ ne mogu u isto vrijeme sadržavati informacije, ili je bila pogreška kod procesiranja ili nije bila pa odgovor sadrži nekakve povratne informacije.

Sljedeće je prikazan primjer klijentskog koda za slanje RPC zahtjeva:

```
1 public RpcResponse<AuthCreateUserResponse>
create(AuthCreateUserRequest request, Connection connection) throws
Exception {
2        Channel channel = connection.createChannel();
3        final String corrId = UUID.randomUUID().toString();
4        String replyQueueName = channel.queueDeclare().getQueue();
5        AMQP.BasicProperties props = new AMQP.BasicProperties
6            .Builder()
7            .correlationId(corrId)
8            .replyTo(replyQueueName)
9            .build();
10       try {
```

```
11            RpcRequest rpcRequest = new RpcRequest("create_user",
request);
12            String json =
objectMapper.writeValueAsString(rpcRequest);
13            channel.basicPublish("", queue, props,
json.getBytes("UTF-8"));
14            final CompletableFuture<String> response = new
CompletableFuture<>();
15            String ctag = channel.basicConsume(replyQueueName,
true, (consumerTag, delivery) -> {
16                if
(delivery.getProperties().getCorrelationId().equals(corrId)) {
17                    response.complete(new
String(delivery.getBody(), "UTF-8"));
18                }
19            }, consumerTag -> {});
20            String result = response.get();
21            channel.basicCancel(ctag);
22            TypeReference<RpcResponse<AuthCreateUserResponse>>
typeReference = new
TypeReference<RpcResponse<AuthCreateUserResponse>>(){};
23            RpcResponse<AuthCreateUserResponse> rpcResponse =
objectMapper.readValue(result, typeReference);
24            return rpcResponse;
25        } catch(Exception exception) {
26            throw exception;
27        }
28    }
```

U kodu iznad je primjer slanja zahtjeva za registraciju korisnika u autentifikacijski servis. Funkcija se nalazi unutar „AuthService“ klase koja implementira sve zahtjeve prema autentifikajskom servisu. Funkcija vraća tip „RpcResponse“ koje je već objašnjen i sastoji se od greške, detalja greške i podataka. Budući da svaki RPC poziv ima neki svoj specifičan odgovor, „RpcResponse“ prima generički tip, i taj tip će poprimiti atribut „data“ unutar objekta „RpcResponse“ klase. Funkcija prima tip „AuthCreateUserRequest“ koji predstavlja oblik zahtjeva, to jest dodatne podatke koji se šalju uz akciju. Funkcija također prima objekt veze prema posredniku poruka (RabbitMQ) zbog toga što kontroleri kada obrađuju korisnički zahtjev, ponekad trebaju zvati više servisa kako bi formirali odgovor, na ovaj način se ne stvara nova veza za svaki RPC zahtjev, nego se stvara jedna veza za cijeli HTTP zahtjev prema API pristupniku, i tada API pristupnik može iskoristiti istu vezu za zahtjeve prema više servisa. U liniji 2 se kreira kanal prema RabbitMQ. U kontekstu RabbitMQ-u, veza je skupa operacija i obično se koristi jedna konekcija iz aplikacije prema posredniku poruka. Tu dolazi koncept kanala, koji u biti predstavljaju virtualne veze unutar ove inicijalne TCP (eng. *Transmission Control Protocol*) veze. U liniji 3 se kreira UUID koji će služiti kao korelacijski identifikator zahtjeva i odgovora. U liniji 4 otvara se novi red, tj. red odgovora. Taj red je specifičan i ima neke specifične postavke kojih se treba pridržavati:

- Imenovanje od strane servera – red dobiva jedinstveno ime unutar RabbitMQ-a i time se izbjegavaju kolizije imena
- Ekskluzivnost – ovakav red mogu konzumirati samo klijenti s istom vezom koja ga je i kreirala. Time se izbjegava uplitanje neke treće opasne strane.
- Funkcionalnost samostalnog brisanja – briše se kada se zadnji klijent odjavi, time se osigurava da je red privremen i briše se kada se završi RPC zahtjev
- Netrajnost – red se također briše ukoliko se posrednik poruka ugasi ili ponovno pokrene, time se također osigurava privremenost reda odgovora

Kada se ne deklariraju dodatne postavke, kreirat će se red s gore navedenim postavkama. U liniji 5 se postavljaju osnovne postavke zahtjeva, poput korelacijskog identifikatora i naziva reda u kojem će API pristupnik čekati odgovor. U linijama 11 i 12 se kreiraju podatni podaci zahtjev i oni se serijaliziraju u tekst. Nakon toga se šalje zahtjev u red u kojem autentifikacijski servis sluša za zahtjeve u liniji 13. U liniji 15 kreira se slušač koji sluša u odgovoru reda na odgovor servisa, također očekuje odgovor s jednakim korelacijskim identifikator koji je poslan i u zahtjevu. Time se osigurava sigurnost komunikacije u smislu da napadač ne bi mogao u tom kratkom periodu ubaciti neki svoj odgovor u privremeni red odgovora i time poremetiti rad sustava. U liniji 20 API pristupnik čeka da se pojavi odgovor. U liniji 21 se API pristupnik odjavljuje iz reda odgovora jer je odgovor stigao i time automatski pokreće brisanje reda odgovora zbog gore navedenih postavka. U liniji nakon toga se odgovor serijalizira u „RpcResponse“ klasu s odgovarajućim generičkim tipom i vraća se odgovor kao odgovarajući Java objekt.

Da bi RPC zahtjev bio ispunjen, treba postojati i nekakva implementacija na serverskoj strani. Sljedeće je prikazan primjer koda sa servisa koji obrađuje RPC zahtjev:

```
1    public void connect(String host, String port, String username,
String password) throws Exception {
2
3        ConnectionFactory factory = new ConnectionFactory();
4        factory.setHost(host);
5        factory.setPort(Integer.parseInt(port));
6        if(username != null && password != null) {
7            factory.setUsername(username);
8            factory.setPassword(password);
9        }
10       connection = factory.newConnection();
11       Channel channel = connection.createChannel();
12       channel.queueDeclare("auth_service", false, true, false,
null);
13       channel.queuePurge("auth_service");
```

```
14        channel.basicQos(1);
15
16        DeliverCallback deliverCallback = (consumerTag, delivery) -
> {
17            AMQP.BasicProperties replyProps = new
AMQP.BasicProperties
18                    .Builder()
19
.correlationId(delivery.getProperties().getCorrelationId())
20                    .build();
21            String message = new String(delivery.getBody(), "UTF-
8");
22            MyLogger.log(message);
23            ObjectMapper objectMapper = new ObjectMapper();
24            RPCRequest request = null;
25            String json;
26            try {
27                request = objectMapper.readValue(message,
RPCRequest.class);
28                json = handle(request);
29            } catch (Exception exception) {
30                MyLogger.log(exception.getMessage());
31                RPCResponse response =
ErrorHelper.create(ErrorHelper.JSON_PARSING_ERROR,
exception.getMessage());
32                json = objectMapper.writeValueAsString(response);
33            }
34            MyLogger.log(json);
35            try {
36                String replyTo =
delivery.getProperties().getReplyTo();
37                if(replyTo != null && !replyTo.isEmpty()){
38                    channel.basicPublish("",
delivery.getProperties().getReplyTo(), replyProps,
json.getBytes("UTF-8"));
39                }
40            } catch (Exception exception) {
41                MyLogger.log(exception.getMessage());
42            }
43
44        };
45        channel.basicConsume("auth_service", true, deliverCallback,
(consumerTag -> {}));
46    }
```

U kodu iznad prikazana je funkcija „connect“ koja se koristi u autentifikacijskom servisu za stvaranje veze prema RabbitMQ-u. Od linije 3 do linije 11 postavljaju se osnovne postavke veze, poput korisničkog imena i lozinke, te se nakon toga stvara veza prema posredniku poruka i kanal. U liniji 12 se deklarira red sa sljedećim postavkama:

- Specifično ime – red je nazvan „auth_service“, time je red lako prepoznatljiv i razumije se da ga konzumira autentifikacijski servis

- Netrajnost – red nije trajan i neće preživjeti kada se RabbitMQ ugasi ili ponovno pokrene. Time se osigurava da je red „živ“, jedino kad je i servis iza njega „živ“. Ako red ne postoji, može se pretpostaviti da je servis ugašen.
- Ekskluzivnost – red je ekskluzivan i može ga konzumirati samo ista veza koja ga je i kreirala. Time se osigurava da samo specifični servis može konzumirati svoj red.
- Funkcionalnost samostalnog brisanja – red nema ovu postavku uključenu. Red se ne smije brisati kada se zadnji klijent odjavi. Red mora biti „živ“ sve dok je i servis „živ“ i konstantno slušati nove RPC zahtjeve

U liniji 13 brišu se sve poruke, za svaki slučaj, iako u ovakvom sustavu red bi prilikom deklaraciju uvijek trebao biti novi i ne sadržavati poruke. U liniji 14 se postavlja gornja granica nepriznatih poruka koji osigurava koliko poruka će servis primiti od posrednika poruka prije nego što ih prizna. U ovom slučaju to je jedna poruka, što znači da kada RabbitMQ proslijedi zahtjev servisu, onda čeka da mu servis prizna poruku, dok ne proslijedi sljedeću. Time se osigurava da servis obrađuje jednu poruku i postiže se konkurentnost sustava. Ukoliko se želi dobiti na većoj propusnosti, može se kreirati konkurentan servis koji može obrađivati više poruka istodobno i ostvariti sinkronizaciju i dosljednost podataka preko baze podataka. Onda se ova gornja granica može povećati. U liniji 16 kreiran je slušač koji sluša na nadolazeće RPC zahtjeve i obrađuje ih. Slušač čita i deserijalizira RPC zahtjev, šalje ga u funkciju „handle“ koja ga dalje obrađuje. Obrada vraća serijalizirani rezultat u obliku teksta koji slušač tada šalje u red odgovora koji je pročitao iz zahtjeva s korelacijskim identifikatorom kojeg je također pročitao iz zahtjeva. Obrada se obično sastoji od obrade podataka ili spremanja, dohvaćanja ili ažuriranja podataka u bazi podataka ili na nekom udaljenom servisu. U liniji 45 na kanal se postavlja slušač da sluša u određenom redu.

U ovom sustavu API pristupnik je klijent koji zaprima HTTP zahtjeve i prilikom obrade poziva više servisa da kreira odgovor. Da bi se osigurale najbolje prakse kod komunikacije s posrednikom poruka, napravljena je sljedeća klasa koja osigurava jednu vezu:

```
1   @ApplicationScoped
2   @Named("rpcClient")
3   public class RpcClient {
4
5       private Connection connection;
6
7       public Connection getConnection() {
8           return connection;
```

```
9       }
10
11      public void setConnection(Connection connection) {
12          this.connection = connection;
13      }
14
15      @PostConstruct
16      public void init() {
17          ConnectionFactory factory = new ConnectionFactory();
18          factory.setHost(Settings.RabbitMQHost);
19          factory.setPort(Settings.RabbitMQPort);
20          factory.setUsername(Settings.RabbitMQUser);
21          factory.setPassword(Settings.RabbitMQPassword);
22          try {
23              connection = factory.newConnection();
24          } catch (IOException e) {
25              e.printStackTrace();
26          } catch (TimeoutException e) {
27              e.printStackTrace();
28          }
29      }
30  }
```

Klasa ima anotacije „@ApplicationScoped“ i „@Named“. Anotacija „@Named“ osigurava da objekt klase bude zrno (eng. *bean*) unutar konteksta aplikacije. Ova anotacija osigurava proces injekcije ovisnosti (eng. *dependency injection*). Anotacija „@ApplicationScoped“ označava životni ciklus zrna unutar konteksta aplikacija, tj. u ovom slučaju je zrno živo kroz cijeli životni ciklus aplikacije što znači da će se kreirati jedna instanca klase koja se može koristiti kroz cijelu aplikaciju, nešto tipa „Singleton“ uzorak dizajna. Time se osigurava da će se kroz cijelu aplikaciju koristiti jedna veza na posrednik poruka. Vidi se da postoji funkcija „init“ koja ima anotaciju „@PostConstruct“. Anotacija „@PostConstruct“ osigurava da će se funkcija pokrenuti čim se objekt klase konstruira. Kasnije se klasa može koristiti u kontrolerima, na primjer:

```
1   @Inject
2   private RpcClient rpcClient;
```

U gornjem kodu se vidi da se na deklaraciju stavlja anotacija „@Inject“. Anotacija „@Inject“ označava da će se Java pobrinuti da injektira objekt klase, tj. da obavi injekciju ovisnosti.

3.10. Web aplikacija

Klijentska web aplikacija koja konzumira API pristupnik. Web aplikacija je napravljena u Jakarta EE 10 programskom okviru i korišten je Web profil. Problem aplikacije je što je Jakarta EE 10 bazirana na izgradnji HTML na serverskoj strani. Obično se u mikro servisnim arhitekturama koristi programski okvir koji nudi izgradnju HTML na klijentskoj strani. Prvi problem je bio što se mora koristiti vanjska knjižnica za HTTP klijenta jer klasična Java knjižnica ne može pozivati vanjske servise. Sljedeće je bio problem održavati autentifikaciju i kolačiće. Kolačići se prepisuju s odgovora iz autentifikacijskog servisa i prosljeđuju nazad klijentu kako bi ih klijent pohranio u svoj web preglednik. Također se na svakom zahtjevu klijenta, kolačići moraju čitati iz zahtjeva, postaviti u zaglavlje HTTP klijenta koji onda može ispravno poslati zahtjev na API pristupnik. I treći problem je WebSocket veza koja je odrađena u čistom JavaScript-u jer bi u suprotnom slučaju, da je rađena direktno putem Jakarta programskog okvira, bilo dosta komplicirano. U tom slučaju trebale bi dvije WebSocket veze koje su sinkronizirane, jedna od klijenta do aplikacijskog servera i druga od aplikacijskog servera do API pristupnika. Da bi se izbjegli ovi problemi, napravljena je direktna komunikacija od klijenta do API pristupnika napisana u JavaScript-u.

Za izradu Docker slike koristi se bazna „payara/micro“ slika i kod se sastoji od sljedećih koraka:

- Linija 1 – dohvaćanje „payara/micro“ docker slike kao baze kontejnera
- Linija 3 – postavljanje radnog direktorija unutar kontejnera
- Linija 5 - kopiranje izvršne datoteke aplikacije s lokalnog računala u baznu sliku
- Linija 7 – otvaranje porta 8080 na kojem je izložen Payara Micro poslužitelj
- Linija 9 – komanda za pokretanje poslužitelja koja također navodi isporuku aplikacije prilikom pokretanja

```
1  FROM payara/micro
2
3  WORKDIR /opt/payara
4
5  COPY target/finder-web.war /opt/payara/deployments/
6
7  EXPOSE 8080
8
9  ENTRYPOINT ["java", "-jar", "payara-micro.jar", "--deploy",
"/opt/payara/deployments/finder-web.war", "--noCluster"]
```

Sljedeće je prikazan kod za isporuku Web aplikacije u Kubernetes:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: finder-web-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: finder-web
  template:
    metadata:
      labels:
        app: finder-web
    spec:
      containers:
      - name: finder-web-container
        image: localhost:5000/finder/web:latest
        ports:
        - containerPort: 8080
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "512Mi"
            cpu: "500m"
```

Sljedeće je prikazan kod za isporuku servisa za otkrivanje Web aplikacije u Kubernetes:

```
apiVersion: v1
kind: Service
metadata:
  name: finder-web-service
spec:
  selector:
    app: finder-web
  ports:
  - protocol: TCP
    port: 8081
    targetPort: 8080
  type: LoadBalancer
```

Aplikacija se sastoji od 6 stranica: registracija, prijava, indeks, čavrljanje, moj profil i profil korisnika.

3.10.1. Registracija

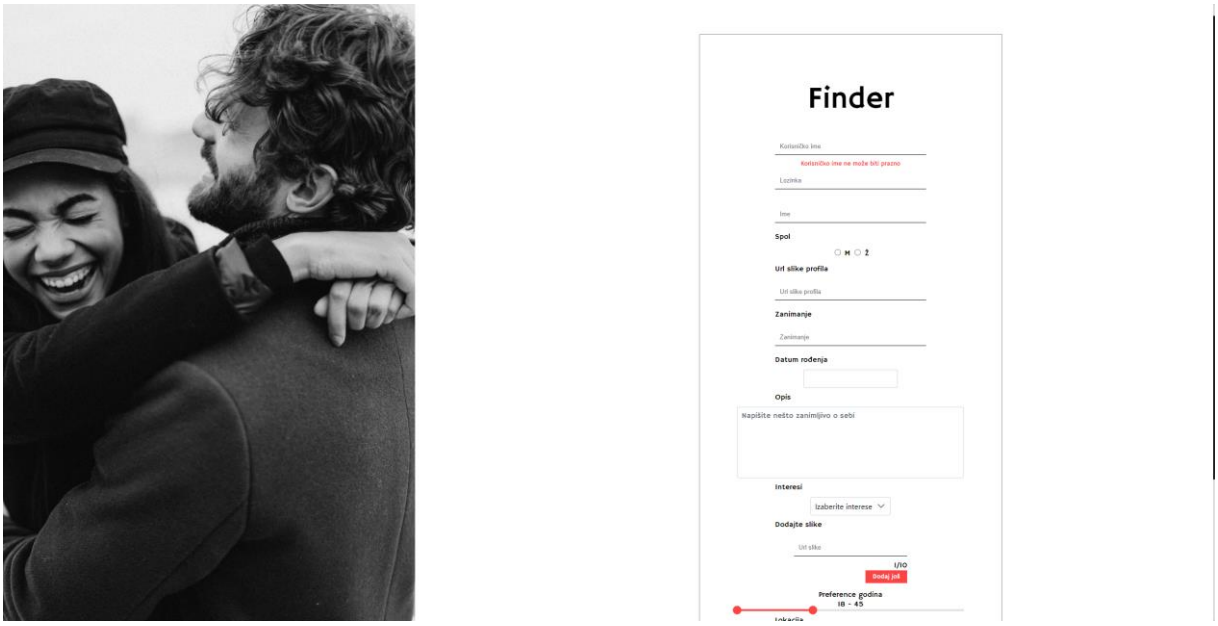
Stranica registracije sastoji se od forme za unos svih potrebnih podataka. Sva polja na formi su obavezna. Korisnik mora unesti korisničko ime, lozinku, nakon toga odabrati spol, unesti adresu slike profila, zanimanje kojim se bavi, datum rođenja, opis ili nešto o sebi, izabrati interese, potom može dodati do 10 dodatnih slika, izabrati preferencije godina, te unesti svoju lokaciju. Nakon toga korisnik može pritisnuti gumb „Registriraj se“ i ukoliko je registracija uspješna, aplikacija će ga usmjeriti na stranicu za prijavu.

Sljedeće je prikazan kod registracije s API pristupnika:

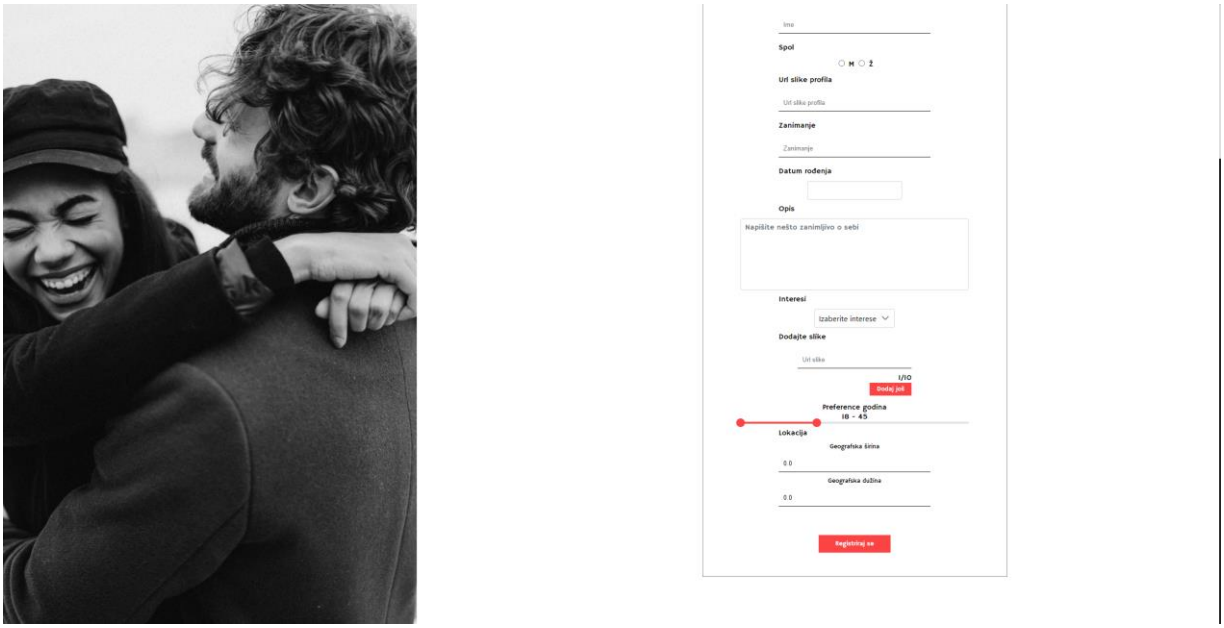
```
1  @POST()
2  @Path("/signup")
3  @Produces({ MediaType.APPLICATION_JSON })
4  @Consumes({ MediaType.APPLICATION_JSON })
5  public Response signup(SignUpRequest request) {
6      Connection connection = rpcClient.getConnection();
7      try {
8          AuthCreateUserRequest authRequest = new
AuthCreateUserRequest(request.getUsername(), request.getPassword());
9          RpcResponse<AuthCreateUserResponse> authRpcResponse =
authService.create(authRequest, connection);
10          if (authRpcResponse.getError() != null) {
11              ErrorGeneral error =
ErrorHelper.handle(authRpcResponse.getError(),
authRpcResponse.getErrorDetails());
12              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), false).build();
13          }
14          AuthCreateUserResponse authResult =
(AuthCreateUserResponse) authRpcResponse.getData();
15
16          UserCreateUserRequest userRequest = new
UserCreateUserRequest(
17              authResult.getId(),
18              request.getName(),
19              request.getSex(),
20              request.getProfilePictureUrl(),
21              request.getBirthDate(),
22              request.getAgeInterestMin(),
23              request.getAgeInterestMax(),
24              request.getDescription(),
```

```
25              request.getOccupation(),
26              request.getInterests(),
27              request.getPictureUrls()
28          );
29          RpcResponse<UserCreateUserResponse> userRpcResponse =
userService.createUser(userRequest, connection);
30          if (userRpcResponse.getError() != null) {
31              ErrorGeneral error =
ErrorHelper.handle(userRpcResponse.getError(),
userRpcResponse.getErrorDetails());
32              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), false).build();
33          }
34          UserCreateUserResponse userResponse =
(UserCreateUserResponse) userRpcResponse.getData();
35
36          GeoCreateUserRequest geoRequest = new
GeoCreateUserRequest(authResult.getId(), request.getCoordinates());
37          RpcResponse<GeoCreateUserResponse> geoRpcResponse =
geoService.create(geoRequest, connection);
38          if (geoRpcResponse.getError() != null) {
39              ErrorGeneral error =
ErrorHelper.handle(geoRpcResponse.getError(),
geoRpcResponse.getErrorDetails());
40              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), false).build();
41          }
42          GeoCreateUserResponse geoResponse =
(GeoCreateUserResponse) geoRpcResponse.getData();
43          SignUpResponse authSignUpResponse = new
SignUpResponse(authResult, userResponse, geoResponse);
44          return ResponseBuilder.getResponseGeneral(Status.OK,
authSignUpResponse, null, false).build();
45      } catch (Exception e) {
46          ErrorGeneral error = new ErrorGeneral();
47          error.setStatus(Status.INTERNAL_SERVER_ERROR);
48          error.setError(e.getMessage());
49          return
ResponseBuilder.getResponseGeneral(Status.INTERNAL_SERVER_ERROR,
null, error.getError(), false).build();
50      }
51  }
```

API pristupnik agregira tri servisa kod registracije: autentifikacijski, korisnički i geolokacijski. API pristupnik agregira odgovore sa sva tri servisa i vraća smisleni odgovor klijentu. API pristupnik uvijek prvo zove autentifikacijski servis u kojem se korisnik mora prvi upisati, kako bi se kreirao UUID prilikom upisa. S tim UUID-om API pristupnik zove ostale servise. Prema tom identifikatoru možemo identificirati korisnika.



Slika 17. Izgled stranice "Registracija" gornji dio



Slika 18. Izgled stranice "Registracija" donji dio

3.10.2. Prijava

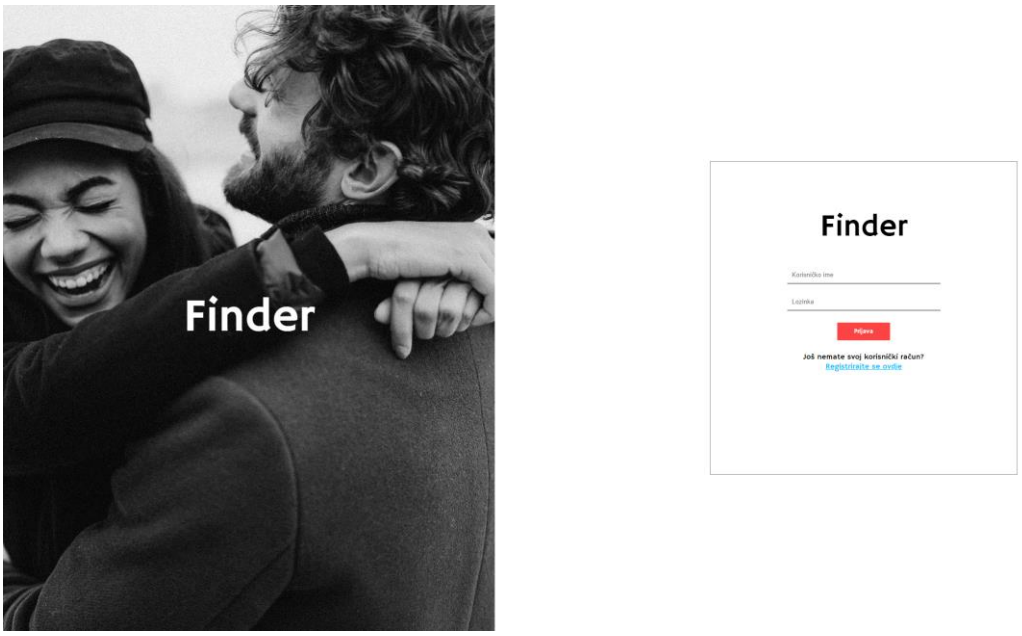
Stranica prijave je jednostavna stranica koja sadrži formu za prijavu. Korisnik mora unesti korisničko ime i lozinku. Ukoliko je prijava uspješna, aplikacija će ga usmjeriti na indeks stranicu, a ukoliko nije, ispisat će se pogreška.

Sljedeće je prikazan kod s API pristupnika:

```
1  @POST()
2  @Path("/login")
3  @Produces({ MediaType.APPLICATION_JSON })
4  @Consumes({ MediaType.APPLICATION_JSON })
5  public Response login(LoginRequest request) {
6      Connection connection = rpcClient.getConnection();
7      try {
8          AuthLoginRequest authRequest = new
AuthLoginRequest(request.getUsername(), request.getPassword());
9          RpcResponse<AuthLoginResponse> authRpcResponse =
authService.login(authRequest, connection);
10         if (authRpcResponse.getError() != null) {
11             ErrorGeneral error =
ErrorHelper.handle(ErrorHelper.UNAUTHORIZED,
authRpcResponse.getErrorDetails());
12             return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), false).build();
13         }
14         AuthLoginResponse authResponse = (AuthLoginResponse)
authRpcResponse.getData();
15         Cookie accessCookie = new
Cookie.Builder("access_token").value(authResponse.getAccessToken()).
build();
16         Cookie refreshCookie = new
Cookie.Builder("refresh_token").value(authResponse.getRefreshToken()
).build();
17         NewCookie newAccessCookie = new
NewCookie.Builder(accessCookie).httpOnly(true).path("/").build();
18         NewCookie newRefreshCookie = new
NewCookie.Builder(refreshCookie).httpOnly(true).build();
19         return ResponseBuilder.getResponseGeneral(Status.OK,
authResponse, null, true)
20             .cookie(newAccessCookie, newRefreshCookie).build();
21     } catch (Exception exception) {
22         ErrorGeneral error = new ErrorGeneral();
23         error.setStatus(Status.INTERNAL_SERVER_ERROR);
24         error.setError(exception.getMessage());
25         return
ResponseBuilder.getResponseGeneral(Status.INTERNAL_SERVER_ERROR,
null, error.getError(), false).build();
26     }
```

```
27 }
```

U ovoj krajnjoj točki API pristupnika zove se samo autentifikaicjski servis. Servis vraća pristupnik žeton i žeton osvježavanja koje API pristupnik stavlja u kolačiće radi poboljšane sigurnosti. Kolačići su tipa „httpOnly“ što označava da se njima ne može pristupiti na klijentskoj strani putem JavaScript-a. Žetoni se kasnije mogu upotrijebiti za autentifikaciju na API pristupnik. Pristupni žeton se koristi kao osnovni žeton autentifikacije, dok se žeton osvježavanja šalje na posebnoj krajnjoj točki na kojoj se žetoni mogu osvježiti. Pristupni žeton obično traje puno kraće, na primjer pet, deset ili petnaest minuta. Žeton osvježavanja obično traje dosta duže, na primjer sedam dana ili par mjeseci. Ukoliko žeton osvježavanja traje dugo, korisnik se nema potrebu ponovno prijavljivati u aplikaciju ako ju ne koristi nekoliko dana, na primjer. Na prvom zahtjevu s klijentske strane se žeton osvježavanja jednostavno može zamijeniti automatski za nove žetone i korisnik neće ni primijetiti što se dogodilo u pozadini i imat će osjećaj da se automatski prijavio u aplikaciju.



Slika 19. Izgled stranice "Prijava"

Slika 20. Izgled stranice "Prijava" s greškom

3.10.3. Indeks

Indeks je početna stranica aplikacije na kojoj korisnik može pregledavati druge korisnike. Tu se odvija proces sviđanja i nesviđanja korisnika. Aplikacija korisniku nudi izbor drugih korisnika baziranih na udaljenosti, spolu i godinama. Kada se drugi korisnik prikaže u aplikaciji, trenutni korisnik ima opcije sviđanja, nesviđanja i detaljnijeg pregledavanja profila. Stranica ima opciju beskonačnog prelistavanja (eng. *infinity scroll*) i konstantno će se prikazivati drugi korisnici sve dok korisnik ne odabere sviđanje ili nesviđanje za sve korisnike. Ukoliko se dogodi da korisnik dođe do kraja, prikazat će mu se stranica da nema više korisnika i da pokuša kasnije. U međuvremenu se novi korisnici mogu registrirati i ako zadovoljavaju uvijete, oni će se tada prikazati trenutnom korisniku. Korisnik također može promijeniti preferencije godina i time će se ovisno o stanju nakon promjene prikazati korisnici koji zadovoljavaju nove uvjete. Ukoliko se dogodi podudaranje, tj. da je korisnik kojeg je trenutni korisnik označio sa sviđa mi se, već trenutnog korisnika označio sa sviđa mi se, pojaviti animacija da se dogodilo podudaranje.

Sljedeće je prikazan kod s API pristupnika:

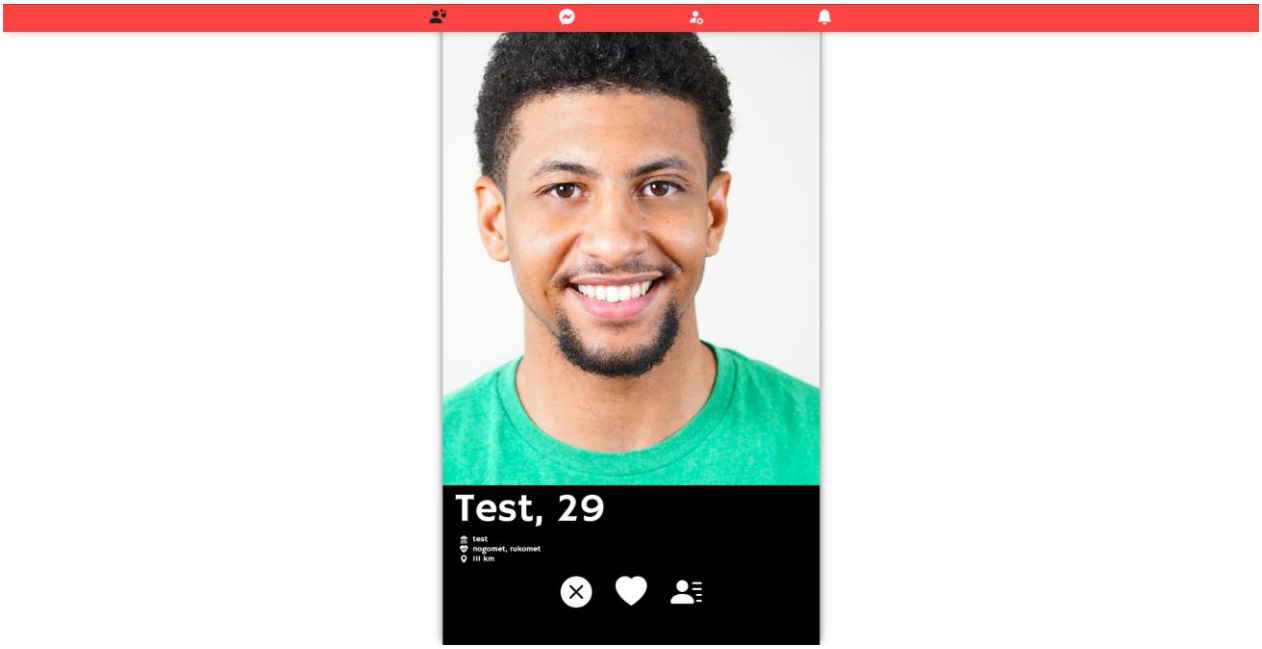
```
1  @GET
2  @Path("index")
3  @Produces({ MediaType.APPLICATION_JSON })
4  @Consumes({ MediaType.APPLICATION_JSON })
5   public Response getIndex(@QueryParam("skipFirst") int skipFirst,
@CookieParam("access_token") Cookie cookie) {
6       Connection connection = rpcClient.getConnection();
7       if (cookie == null) {
8                                               ErrorGeneral    error    =
ErrorHelper.handle(ErrorHelper.UNAUTHORIZED, "Access token does not
exist");
9                                                                   return
ResponseBuilder.getResponseGeneral(Status.UNAUTHORIZED,         null,
error.getError(), false).build();
10      }
11      String accessToken = cookie.getValue();
12
13      DecodedJWT jwt = null;
14      try {
15          jwt = JwtHelper.verifyAccess(accessToken);
16      } catch (Exception exception) {
17                                              ErrorGeneral    error    =
ErrorHelper.handle(ErrorHelper.UNAUTHORIZED, "Invalid access token");
18                                                                  return
ResponseBuilder.getResponseGeneral(Status.UNAUTHORIZED,         null,
error.getError(), false).build();
19      }
20      try {
21          UUID userId = UUID.fromString(jwt.getSubject());
22          int page = 1;
23          GetIndexResponse response = new GetIndexResponse();
24          response.hasMore = false;
25          boolean firstLoop = true;
26          while (response.getUsers().size() < 10) {
27              GeoGetRequest geoRequest = new GeoGetRequest(userId,
page);
28                      RpcResponse<GeoGetResponse> geoRpcResponse =
geoService.get(geoRequest, connection);
29              if (geoRpcResponse.getError() != null) {
30                                                                  if
(geoRpcResponse.getError().equals(ErrorHelper.NOT_FOUND_ERROR)     &&
response.getUsers().size() > 0) {
31                      break;
32                  }
33                                              ErrorGeneral   error   =
ErrorHelper.handle(geoRpcResponse.getError(),
geoRpcResponse.getErrorDetails());
34                                                                  return
ResponseBuilder.getResponseGeneral(error.geStatus(),        response,
error.getError(), false).build();
35              }
```

```
36                    GeoGetResponse geoResponse = (GeoGetResponse)
geoRpcResponse.getData();
37
38              List<UUID> uuids = new ArrayList<>();
39
40              for (NearestUser user : geoResponse.getUsers()) {
41                  uuids.add(user.getId());
42              }
43
44                        UserFilterUsersRequest userRequest = new
UserFilterUsersRequest(userId, uuids);
45               RpcResponse<UserFilterUsersResponse> userRpcResponse
= userService.filter(userRequest, connection);
46              if (userRpcResponse.getError() != null) {
47                                              ErrorGeneral error =
ErrorHelper.handle(userRpcResponse.getError(),
userRpcResponse.getErrorDetails());
48                                                                  return
ResponseBuilder.getResponseGeneral(error.geStatus(),            null,
error.getError(), false).build();
49              }
50
51                           UserFilterUsersResponse filteredUsers =
(UserFilterUsersResponse) userRpcResponse.getData();
52
53              if (filteredUsers.getFilteredUsers().size() == 0) {
54                  continue;
55              }
56
57                                       for (FilteredUser user :
filteredUsers.getFilteredUsers()) {
58                  response.getUsers().add(user);
59              for (NearestUser nearestUser : geoResponse.getUsers())
{
60                      if (user.getId().equals(nearestUser.getId()))
{
61                          user.setDistance(nearestUser.getKm());
62                      }
63                  }
64              }
65
66                  if (firstLoop && skipFirst > 0 && skipFirst <=
response.getUsers().size()) {
67                  for (int i = 0; i < skipFirst; i++) {
68                      response.getUsers().remove(0);
69                  }
70                  firstLoop = false;
71              }
72
73              page++;
74          }
75
76          if (response.getUsers().size() > 10) {
77                                                 response.setUsers(new
ArrayList<>(response.getUsers().subList(0, 9)));
78              response.hasMore = true;
```

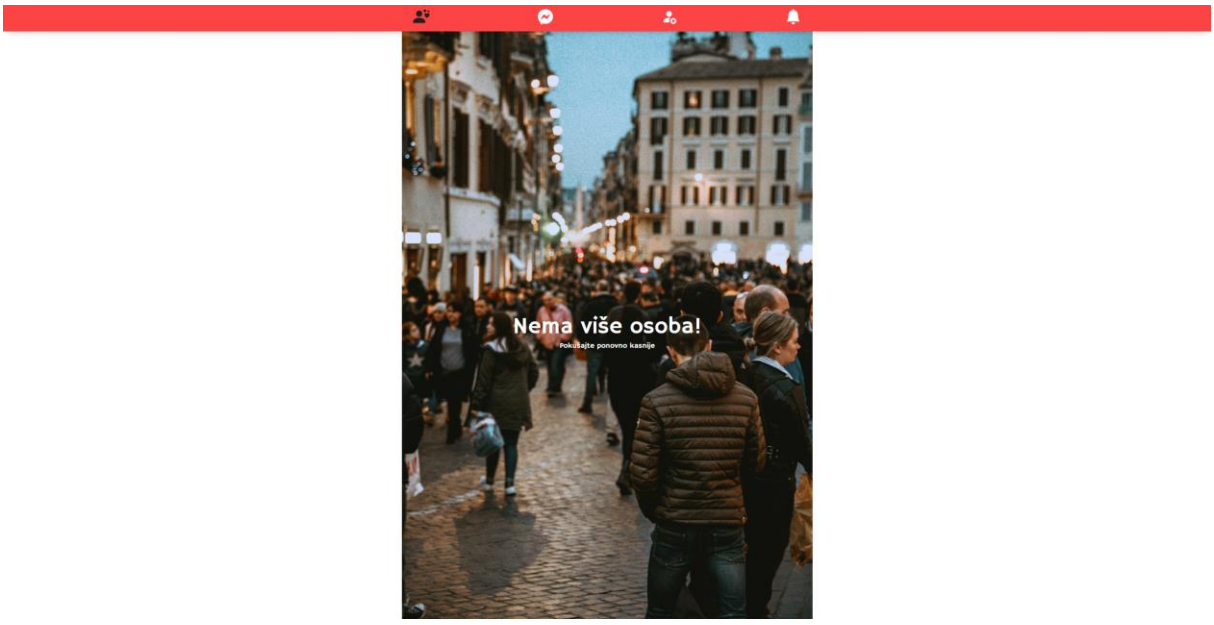
```
79          }
80
81              return ResponseBuilder.getResponseGeneral(Status.OK,
response, null, true).build();
82      } catch (Exception exception) {
83          ErrorGeneral error = new ErrorGeneral();
84          error.setStatus(Status.INTERNAL_SERVER_ERROR);
85          error.setError(exception.getMessage());
86                                                              return
ResponseBuilder.getResponseGeneral(Status.INTERNAL_SERVER_ERROR,
null, error.getError(), false).build();
87      }
88  }
```

Ova krajnja točka API pristupnika je zaštićena, zato kao parametar prima kolačić u kojem očekuje pristupni žeton. Prvi korak je provjera žetona i deserijalizacija korisnikovog UUID-a kako bi pristupnik znao o kojem se korisniku radi. Ovdje se agregiraju dva servisa: geolokacijski i korisnički. Prvo se poziva geolokacijski servis jer nam je udaljenost između dva korisnika prvi filter koji se mora primijeniti, tj. trenutnog korisnika koji šalje zahtjev i ostalih korisnika. API pristupnik na raspolaganje dobiva pedeset ili manje kandidata, tj. ostalih korisnika sortiranih od najbližeg prema najudaljenijem. API pristupnik dalje šalje zahtjev za filtriranjem korisnika na korisnički servis. Korisnički servis filtrira korisnike prema kriterijima godina, spola i da li su se korisnici već podudarili. API pristupnik tada gleda koliko je korisnički servis kandidata maknuo. Ukoliko je korisnički servis maknuo sve kandidate, API pristupnik ponavlja proces i zahtjeva sljedećih pedeset najbližih korisnika od geolokacijskog servisa. Ukoliko je API pristupnik zaprimio neki broj kandidata od korisničkog servisa, primjenjuje se drugi parametar koji prima krajnja točka, a to je parametar „skipFirst“. Parametar „skipFirst“ označava koliko prvih korisnika u prvoj iteraciji petlje API pristupnik treba izbrisati iz odgovora. Ovaj pristup je odličan kada se na klijentu želi ostvariti tehnika beskonačnog prelistavanja. Na klijentskoj strani se obično na pojedini element u listi zakvači slušač koji će pozvati API pristupnik da dohvati još korisnika. Slušač se obično zakači na predzadnji ili pred predzadnji element u listi. Ukoliko se slušač zakači na pred predzadnji element u listi, to bi značilo da u sljedećem dohvaćanju korisnika sa servisa, prva tri korisnika moramo preskočiti jer će oni biti isti kao zadnja tri u već postojećoj listi korisnika na klijentskoj strani. Nakon toga API pristupnik gleda da li ispunjava uvjet da vrati deset korisnika po zahtjevu. Ukoliko postoji više od deset korisnika, API pristupnik briše ostale korisnike iz odgovora i u odgovoru označava da postoji još osoba. Prema toj oznaci, na klijentskoj strani se može zakvačiti novi slušač na, na primjer, predzadnji element u listi kako bi se proces nastavio ili ukoliko u odgovoru piše da nema više korisnika, klijent može maknuti ili ne zakvačiti novi slušač jer mu je poznato da korisnika više nema. Ukoliko API pristupnik nema ispunjen uvjet od deset korisnika po odgovoru, onda se nastavlja sljedeća iteracija petlje i poziva se geolokacijski servis da dohvati sljedećih mogućih

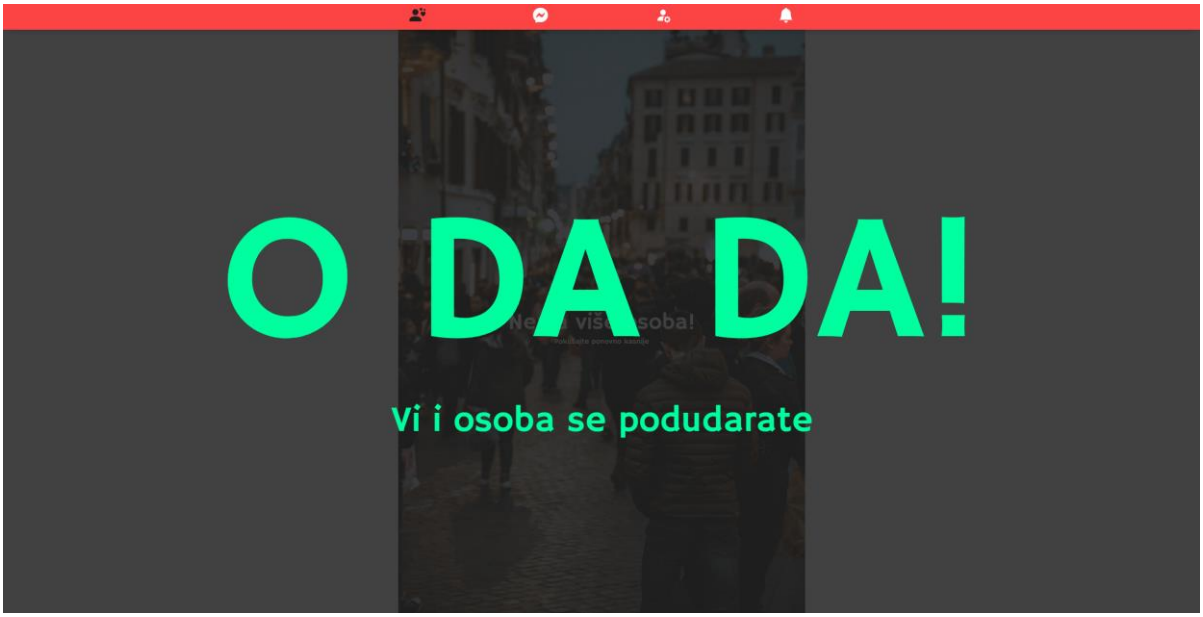
pedeset kandidata. Proces se ponavlja sve dok geolokacijski servis označi da nema više korisnika i time se zaustavlja petlja te API pristupnik vraća sve korisnike za koje je uspio obraditi zahtjev. Ovaj proces bi se mogao poboljšati tako da se napravi tehnika materijaliziranog pogleda. To bi značilo da bi geolokacijski servis imao sve podatke potrebne za izračun, poput godina korisnika, statusa podudaranja i slično. Time bi geolokacijski servis sam mogao vratiti gotovu listu kandidata, a korisnički servis bi samo vratio dodatne informacije potrebne za suvisli odgovor API pristupnika. Mana ovog pristupa bi bila dupliciranje podataka u servisima.



Slika 21. Izgled stranice "Indeks"



Slika 22. Izgled stranice "Indeks" kad nema više osoba



Slika 23. Izgled stranice "Indeks" kad se dogodi podudaranje

3.10.4. Čavrljanje

Stranica čavrljanja se sastoji od izbornika sa sesijama s lijeve strane i sekcije za čavrljanje s desne. Izbornik sa sesijama je sortiran kronološki ovisno o zadnjim porukama u sesijama. Kada dođe nova poruka, izbornik se ažurira. S desne strane je sekcija za čavrljanje u kojoj korisnik može čavrljati s drugim korisnikom. Poruke su kronološki poredane od zadnje

prema prvoj i sekcija podržava beskonačno prelistavanje sve dok se ne dođe do prve poruke u sesiji. Čim se dogodi podudaranje, oba korisnika će na stranici za čavrljanje vidjeti novu sesiju. Stranica koristi WebSocket vezu tako da se izmjena poruka događa u stvarnom vremenu.

Sljedeće je prikazan kod za WebSocket vezu s klijentske strane:

```
1  function connectWs(accessToken) {
2      var websocketURL = "ws://pc.mshome.net:8080/finder-
api/api/v1/chat?access_token=" + accessToken;
3      websocket = new WebSocket(websocketURL);
4
5      websocket.onmessage = function(event) {
6          onNewMessage([{name: 'message', value:
event.data.toString()}])
7      };
8  }
9
10  function sendMessage(message) {
11      if (websocket && websocket.readyState === WebSocket.OPEN) {
12          websocket.send(message);
13      } else {
14          console.error("WebSocket is not open.");
15      }
16  }
```

Ovo je kod napisan u JavaScript-u koji se povezuje na WebSocket krajnju točku na API pristupniku. Funkcija „connectWS“ služi za otvaranje WebSocket veze s API pristupnikom. Funkcija prima jedan argument „accessToken“ koji označava pristupni token koji API pristupnik zahtijeva kako bi mogao identificirati korisnika. Funkcija „onmessage“ je slušač koji sluša na nadolazeće poruke. Funkcija prosljeđuje poruku u zrno na web aplikaciju. Funkcija „sendMessage“ šalje poruku preko WebSocket veze i nju poziva zrno iz web aplikacije.

Sljedeće je prikazan kod funkcije „onNewMessage“ koja se poziva iz JavaScript-a u web pregledniku na web aplikaciju:

```
1  public void onNewMessage() {
2      FacesContext context = FacesContext.getCurrentInstance();
3      Map<String, String> params =
context.getExternalContext().getRequestParameterMap();
4      String message = params.get("message") != null ?
params.get("message") : null;
```

```
5       if (message != null) {
6           ObjectMapper objectMapper = new ObjectMapper();
7           objectMapper.registerModule(new JavaTimeModule());
8
objectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
;
9           try {
10              WsChatMessageResponse response =
objectMapper.readValue(message, WsChatMessageResponse.class);
11              Message newMessage = new Message(response.getId(),
response.getCreated(), response.getContent(), response.getUserId());
12              getMessagesResponse().getMessages().add(0,
newMessage);
13              Chat chat = getChatsResponse().getChats().stream()
14                  .filter(item ->
item.getSessionId().equals(activeSession))
15                  .findFirst().orElse(null);
16              getChatsResponse().getChats().remove(chat);
17              chat.setMessage(newMessage.getContent());
18              chat.setMessageTimeStamp(newMessage.getCreated());
19              getChatsResponse().getChats().add(0, chat);
20          } catch (Exception e) {
21              e.printStackTrace();
22          }
23      }
24  }
```

Kada dođe nova poruka u WebSocket vezi koja je uspostavljena između web preglednika i API pristupnika, poziva se ova funkcija u zrnu u web aplikaciji i prosljeđuje se nova poruka. Web aplikacija zaprima poruku, deserijalizira ju, te ju dodaje kao najnoviju poruku u listi poruka. Također se sortira i lista sesija. Sesija kojoj ova poruka pripada se stavlja na prvo mjesto u listi. Ovo je cijeli proces zaprimanja poruke iz WebSocket veze i ažuriranja stanja web aplikacije i korisničkog sučelja. Korisnik će ovu poruku vidjeti kao skroz donju u listi poruka, a sesiju će vidjeti kao skroz gornju u listi sesija.

Slanje poruke odvija se na obrnut način od zaprimanja. Slanje poruke se prvo odvija u zrnu u web aplikaciji, a potom se poziva JavaScript funkcija „sendMessage“ u web pregledniku. Sljedeće je prikazan kod za slanje poruke iz zrna iz web aplikacija:

```
1  public void sendMessage() {
2       if (message.equals("") || message == null) {
3           return;
4       }
5       ObjectMapper objectMapper = new ObjectMapper();
6       WsChatMessageRequest request = new
WsChatMessageRequest(activeSession, message);
7       String wsMessage = "";
```

```
8       try {
9           wsMessage = objectMapper.writeValueAsString(request);
10       } catch (JsonProcessingException e) {
11           e.printStackTrace();
12       }
13       PrimeFaces.current().executeScript("sendMessage('" +
wsMessage + "');");
14       setMessage("");
15   }
```

Web aplikacija zaprima poruku iz preglednika, kreira novi zahtjev za poruku i šalje ga JavaScript funkciji u web pregledniku koja će tada poruku proslijediti u WebSocket vezu.

API pristupnik ima WebSocket krajnju točku na koju se korisnici spajaju. Krajnja točka je zaštićena i zahtijeva pristupni žeton. Krajnja točka implementira dvije osnovne funkcije, jednu za obradu kada korisnik uspostavi vezu i jednu kada pristigne nova poruka. Krajnja točka agregira dva servisa: servis čavrljanja i servis obavijesti.

Sljedeće je prikazan kod za obradu kada korisnik uspostavi vezu:

```
1   @OnOpen
2   public void onOpen(Session session, EndpointConfig config) {
3       URI requestUri = session.getRequestURI();
4       String query = requestUri.getQuery();
5       Map<String, String> queryParams = getQueryMap(query);
6       String accessToken = queryParams.get("access_token");
7       if (accessToken == null || accessToken.isEmpty()) {
8           try {
9               session.close();
10              return;
11          } catch (Exception e) {
12              e.printStackTrace();
13          }
14      }
15
16      DecodedJWT jwt = null;
17      try {
18          jwt = JwtHelper.verifyAccess(accessToken);
19      } catch (Exception exception) {
20          try {
21              session.close();
22              return;
23          } catch (Exception e) {
24              e.printStackTrace();
25          }
26      }
```

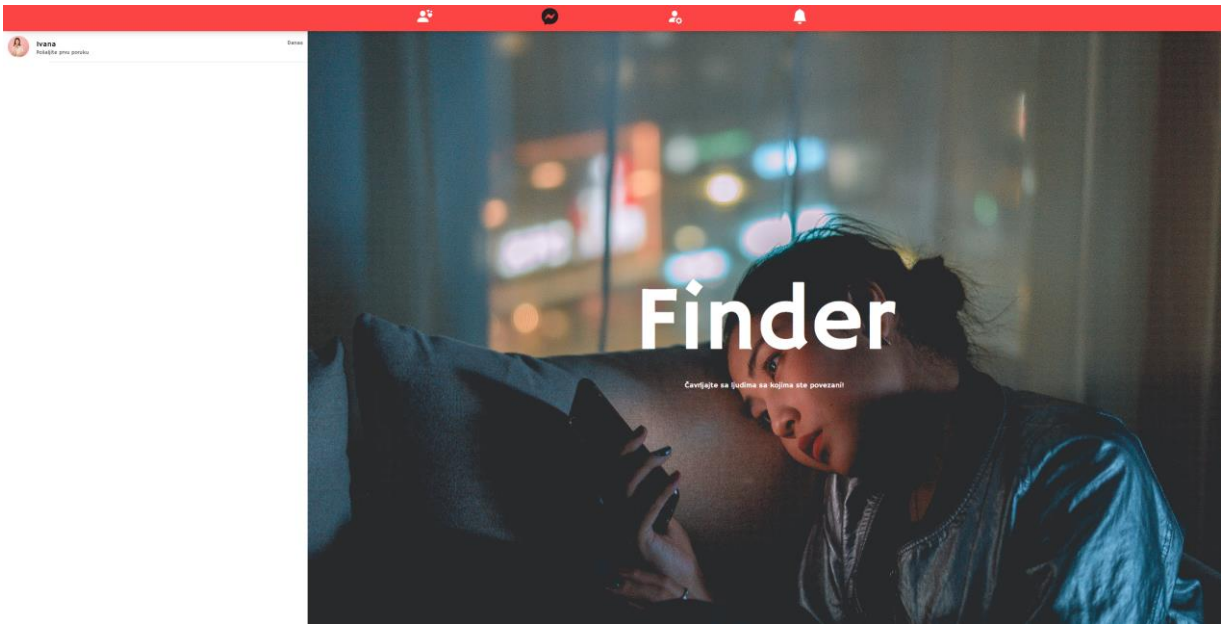
```
27      String userId = jwt.getSubject();
28      session.getUserProperties().put("userId", userId);
29  }
```

Funkcija ima anotaciju „@OnOpened“ što označava da će se pokrenuti kada korisnik uspostavi WebSocket vezu s API pristupnikom. Funkcija tada dohvaća pristupni žeton i provjerava njegovu validnost te izvlači korisnikov UUID koji će se koristiti u daljnjoj komunikaciji. Kada se uspostavi veza, Jakarta u pozadini stvara sesiju i ta sesija označava jednu WebSocket vezu. Funkcija zbog toga stavlja svojstvo koje predstavlja korisnikov UUID na sesiju. To svojstvo se koristi kada dođe nova poruka na API pristupnik, s obzirom da Jakarta upravlja sesijama, iz sesije se izvlači baš to svojstvo i identificira se korisnik. Sljedeće je prikazan kod kada pristigne nove poruka:

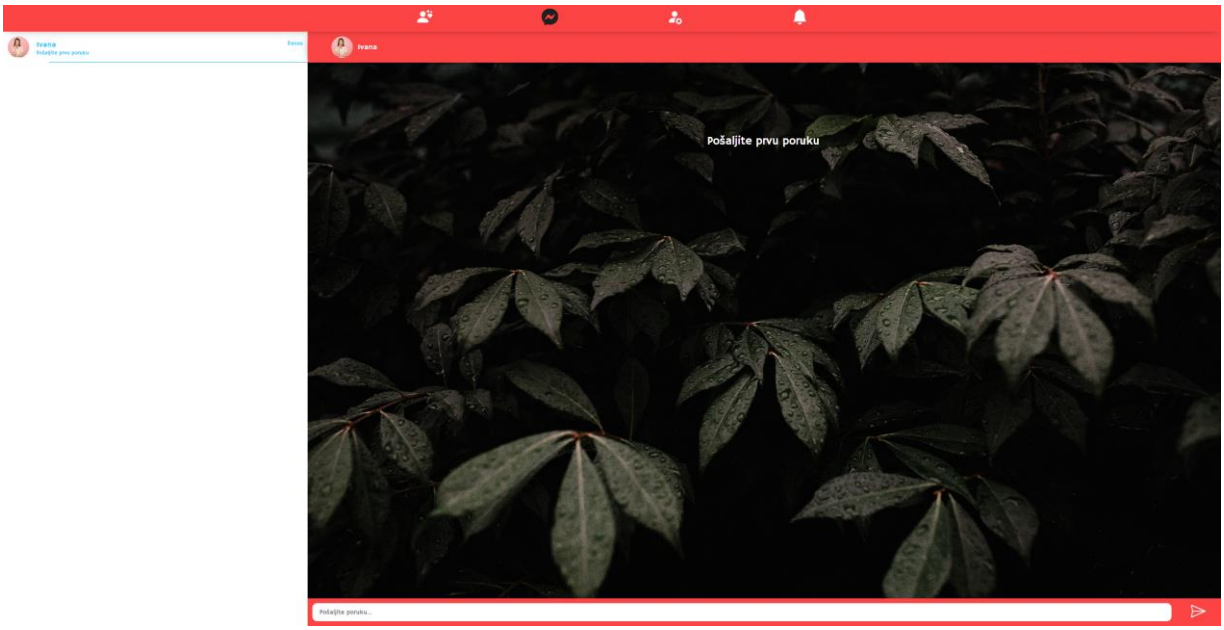
```
1  @OnMessage
2  public void onMessage(String message, Session session) {
3      Connection connection = rpcClient.getConnection();
4      ObjectMapper objectMapper = new ObjectMapper();
5      objectMapper.registerModule(new JavaTimeModule());
6
objectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS)
;
7      String userIdString = (String)
session.getUserProperties().get("userId");
8      UUID userId = UUID.fromString(userIdString);
9      System.out.println("Received: " + message + " from: " +
userId);
10      try {
11          WsChatMessageRequest wsChatMessage =
objectMapper.readValue(message, WsChatMessageRequest.class);
12          ChatCreateMessageRequest chatRpcRequest = new
ChatCreateMessageRequest(wsChatMessage.getSessionId(), userId,
wsChatMessage.getMessage());
13          RpcResponse<ChatCreateMessageResponse> chatRpcResponse =
chatService.createMessage(chatRpcRequest, connection);
14          if (chatRpcResponse.getError() != null) {
15
session.getBasicRemote().sendText(objectMapper.writeValueAsString(ch
atRpcResponse));
16          } else {
17              ChatCreateMessageResponse response =
(ChatCreateMessageResponse) chatRpcResponse.getData();
18              boolean delivered = false;
19              for (Session s : session.getOpenSessions()) {
20                  if (s.isOpen()) {
21                      for (var item : response.getUsers()) {
22                          if
(s.getUserProperties().get("userId").equals(item.toString())) {
```

```
23
s.getBasicRemote().sendText(objectMapper.writeValueAsString(response
));
24                                  if
(!item.equals(response.getUserId())) {
25                                      delivered = true;
26                                  }
27                              }
28                          }
29                      }
30                  }
31                  if (!delivered) {
32                      var sender = userId;
33                      var receiver =
response.getUsers().stream().filter(item ->
!item.equals(userId)).findFirst().orElse(null);
34                      NotifCreateNotificationRequest notifRpcRequest =
new NotifCreateNotificationRequest(sender, receiver, "message");
35                      notifService.create(notifRpcRequest,
connection);
36                  }
37              }
38          } catch (Exception exception) {
39              exception.printStackTrace();
40          }
41      }
```

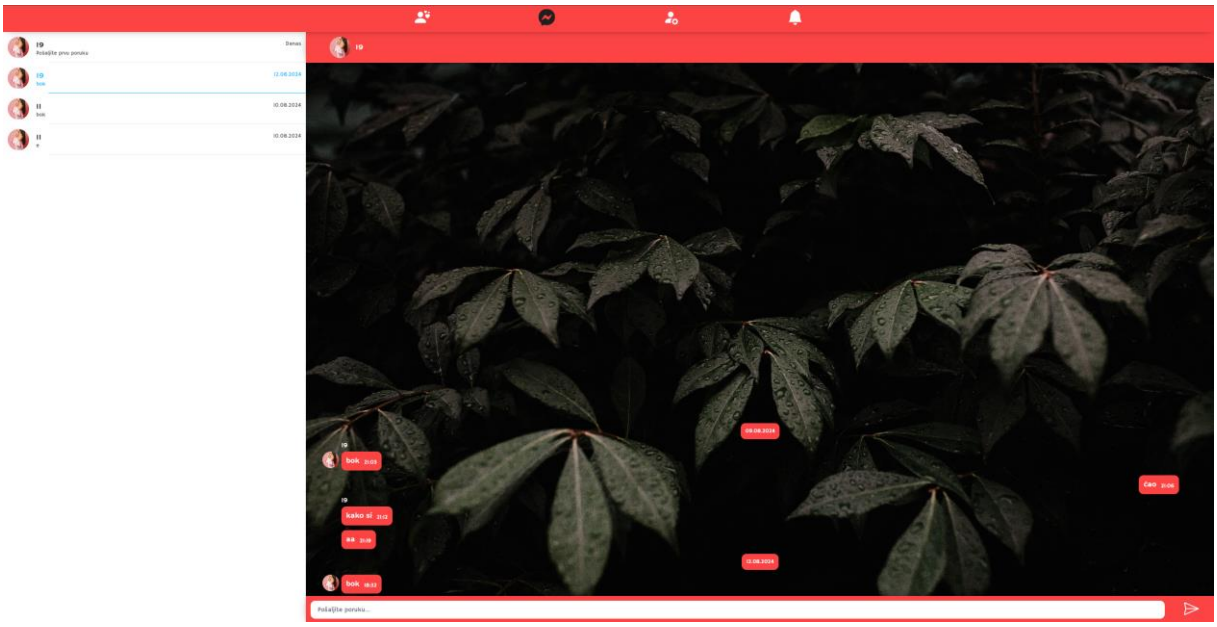
Funkcija je označena s anotacijom „@OnMessage“ što znači da će se pokrenuti kada pristigne nova poruka na API pristupnik. Funkcija deserijalizira zaprimljenu poruku i dohvaća UUID korisnika koji je postavljen kao svojstvo sesije u funkciji koja obrađuje uspostavu nove veze. Ti podaci se šalju servisu čavrljanja koji upisuje novu poruku u svoju bazu. Kada servis vrati odgovor da je upisao poruku, funkcija pokušava dostaviti poruku oba korisnika, i pošiljatelju i primatelju. Ovim principom je primatelj siguran da je sve prošlo u redu, tj. da je poruka uspješno poslana, a primatelj dobiva poruku u stvarnom vremenu. Ukoliko primatelj nije aktivan, tj. nema uspostavljenu WebSocket vezu s API pristupnikom, tada API pristupnik ne može poslati primatelju poruku u stvarnom vremenu, već umjesto toga kreira novu obavijest u servisu obavijesti. Obavijest se kreira samo za primatelja.



Slika 24. Izgled stranice "Čavrljanje"



Slika 25. Izgled stranice "Čavrljanje" kada se započinje razgovor



Slika 26. Izgled stranice "Čavrljanje" s razgovorom

3.10.5. Moj profil

Na ovoj stranici korisnik može pregledavati i uređivati svoje podatke. Korisnik ima opcije promjene kroničkog imena ako je dostupno, imena, datuma rođenja, preferencije godina, opisa, web adrese slike profila, interesa i slika. Korisnik se također može odjaviti iz aplikacije na ovoj stranici.

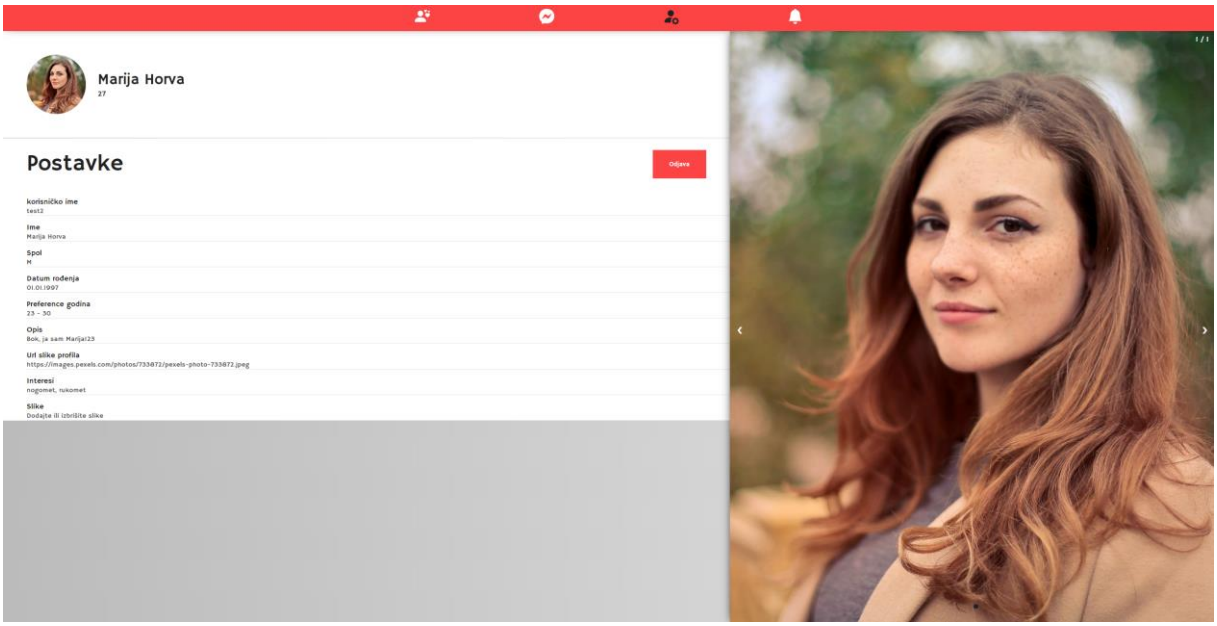
Sljedeće je prikazan kod s API pristupnika za dohvaćanje informacija o profilu trenutnog korisnika:

```
1  @GET()
2  @Path("self")
3  @Produces({ MediaType.APPLICATION_JSON })
4  @Consumes({ MediaType.APPLICATION_JSON })
5  public Response getSelf(@CookieParam("access_token") Cookie
cookie) {
 6      Connection connection = rpcClient.getConnection();
7      if (cookie == null) {
8          ErrorGeneral error =
ErrorHelper.handle(ErrorHelper.UNAUTHORIZED, "Access token does not
exists");
9          return
ResponseBuilder.getResponseGeneral(Status.UNAUTHORIZED, null,
error.getError(), false).build();
10      }
11      String accessToken = cookie.getValue();
```

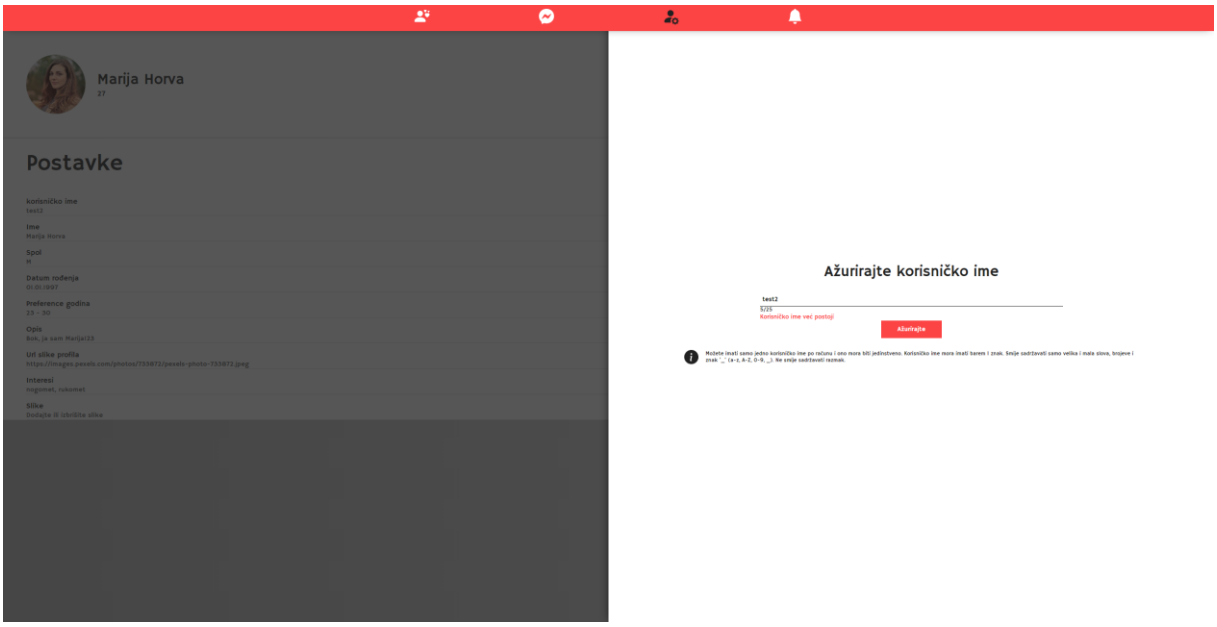
```
12
13      DecodedJWT jwt = null;
14      try {
15          jwt = JwtHelper.verifyAccess(accessToken);
16      } catch (Exception exception) {
17          ErrorGeneral error =
ErrorHelper.handle(ErrorHelper.UNAUTHORIZED, "Invalid access
token");
18          return
ResponseBuilder.getResponseGeneral(Status.UNAUTHORIZED, null,
error.getError(), false).build();
19      }
20
21      try {
22          UUID userId = UUID.fromString(jwt.getSubject());
23
24          AuthGetUsernameRequest authRequest = new
AuthGetUsernameRequest(userId);
25          RpcResponse<AuthGetUsernameResponse> authRpcResponse =
authService.getUsername(authRequest, connection);
26          if (authRpcResponse.getError() != null) {
27              ErrorGeneral error =
ErrorHelper.handle(authRpcResponse.getError(),
authRpcResponse.getErrorDetails());
28              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), true).build();
29          }
30          AuthGetUsernameResponse authResponse =
(AuthGetUsernameResponse) authRpcResponse.getData();
31
32          UserGetSelfRequest userRequest = new
UserGetSelfRequest(userId);
33          RpcResponse<UserGetSelfResponse> userRpcResponse =
userService.getSelf(userRequest, connection);
34          if (userRpcResponse.getError() != null) {
35              ErrorGeneral error =
ErrorHelper.handle(userRpcResponse.getError(),
userRpcResponse.getErrorDetails());
36              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), true).build();
37          }
38          UserGetSelfResponse userResponse = (UserGetSelfResponse)
userRpcResponse.getData();
39
40          GetSelfResponse response = new GetSelfResponse(
41              userId,
42              authResponse.getUsername(),
43              userResponse.getName(),
44              userResponse.getAge(),
45              userResponse.getBirthDate(),
46              userResponse.getSex(),
47              userResponse.getCreated(),
48              userResponse.getProfilePictureUrl(),
49              userResponse.getOccupation(),
```

```
50              userResponse.getDescription(),
51              userResponse.getAgeInterestMin(),
52              userResponse.getAgeInterestMax(),
53              userResponse.getInterests(),
54              userResponse.getMedias()
55          );
56
57          return ResponseBuilder.getResponseGeneral(Status.OK,
response, null, true).build();
58      } catch (Exception exception) {
59          ErrorGeneral error = new ErrorGeneral();
60          error.setStatus(Status.INTERNAL_SERVER_ERROR);
61          error.setError(exception.getMessage());
62          return
ResponseBuilder.getResponseGeneral(Status.INTERNAL_SERVER_ERROR,
null, error.getError(), true).build();
63      }
64  }
```

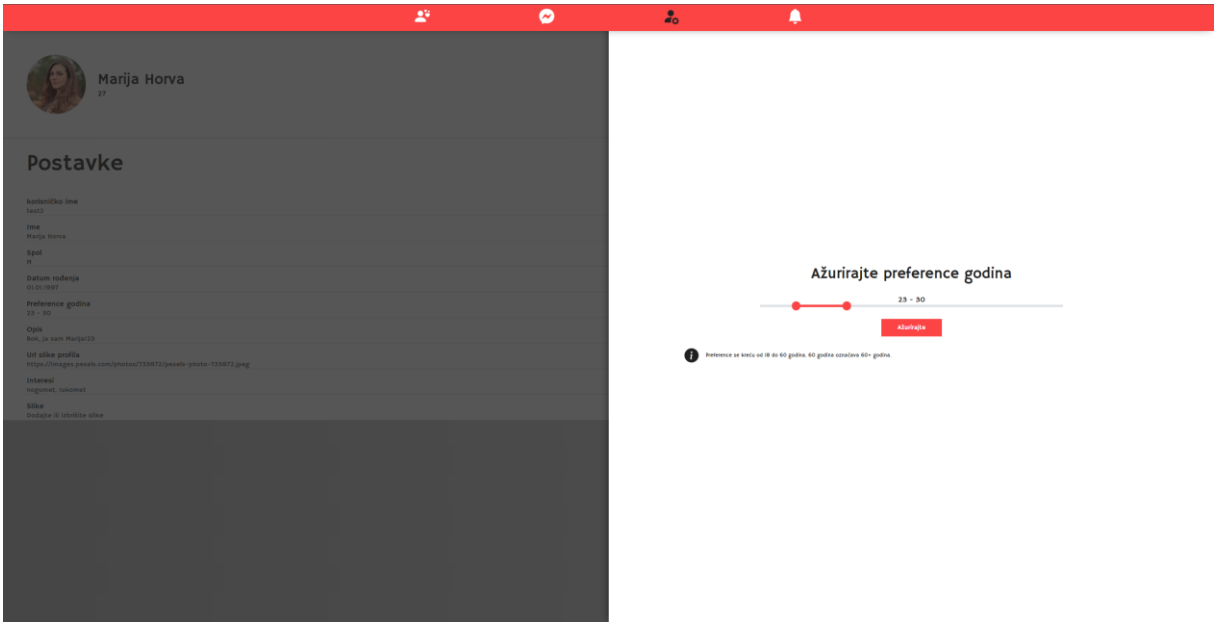
Ova krajnja točka agregira dva servisa: autentifikacijski i korisnički. Krajnja točka je zaštićena i zahtijeva pristupni žeton u kolačićima. S pristupnim žetonom se identificira korisnik. API pristupnik iz autentifikacijskoj servisa zahtijeva samo korisničko ime, kojeg je moguće ažurirati na ovoj stranici. Ostale informacije se zahtijevaju iz korisničkog servisa, i za ažuriranja tih informacija se koristi korisnički servis.



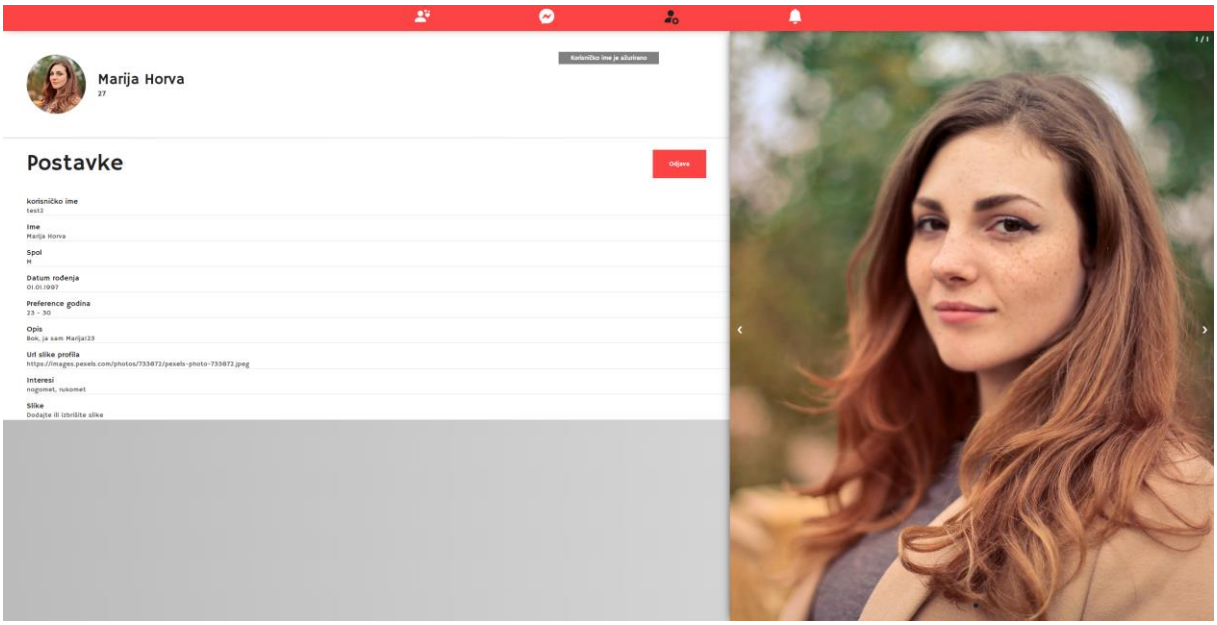
Slika 27. Izgled stranice "Moj profil"



Slika 28. Izgled stranice "Moj profil" s otvorenom formom za ažuriranje korisničkog imena



Slika 29. Izgled stranice "Moj profil" s otvorenom formom za ažuriranje preferencija godina



Slika 30. Izgled stranice "Moj profil" nakon uspješnog ažuriranja s obavijesti na vrhu

3.10.6. Profil korisnika

Trenutni korisnik može pregledati profil drugog korisnika da vidi dodatne informacije i slike. U adresu ove stranice mora se nadodati parametar s ključem „id“ i vrijednosti identifikatora korisnika. Taj identifikator se prosljeđuje API pristupniku kako bi on znao čiji profil trenutni korisnik želi pogledati.

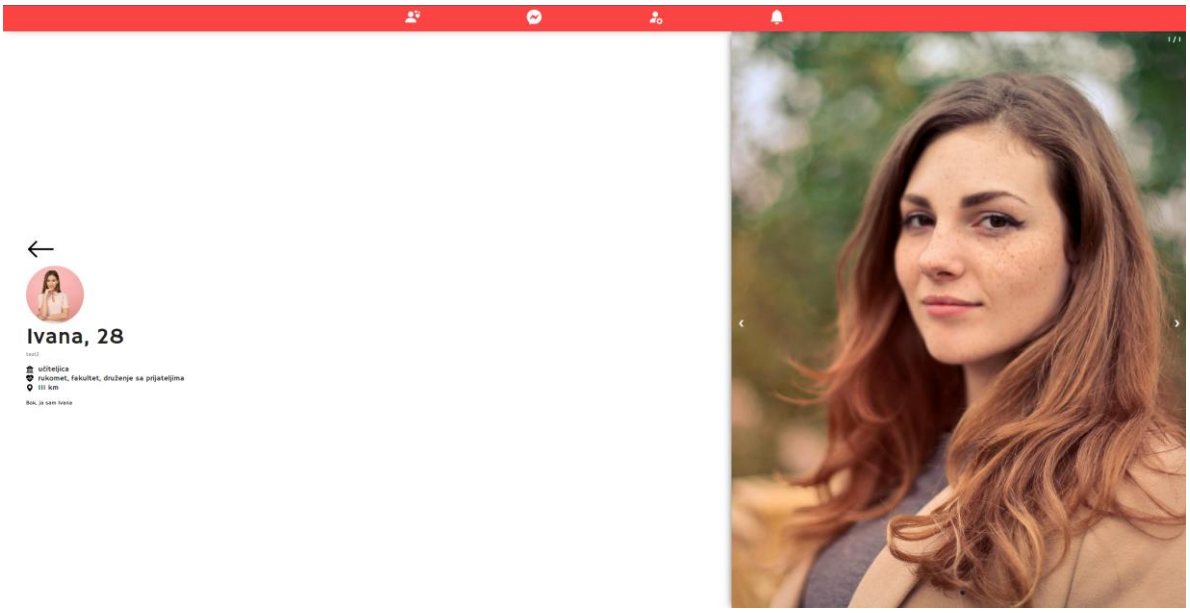
Sljedeće je prikazan kod s API pristupnika za dohvaćanje informacija korisnika:

```
1  @GET()
2  @Produces({ MediaType.APPLICATION_JSON })
3  @Consumes({ MediaType.APPLICATION_JSON })
4  public Response getUser(@QueryParam("id") UUID id,
@CookieParam("access_token") Cookie cookie) {
5       Connection connection = rpcClient.getConnection();
6       if (cookie == null) {
7           ErrorGeneral error =
ErrorHelper.handle(ErrorHelper.UNAUTHORIZED, "Access token does not
exists");
8           return
ResponseBuilder.getResponseGeneral(Status.UNAUTHORIZED, null,
error.getError(), false).build();
9       }
10       String accessToken = cookie.getValue();
11
12       DecodedJWT jwt = null;
```

```
13      try {
14          jwt = JwtHelper.verifyAccess(accessToken);
15      } catch (Exception exception) {
16          ErrorGeneral error =
ErrorHelper.handle(ErrorHelper.UNAUTHORIZED, "Invalid access
token");
17          return
ResponseBuilder.getResponseGeneral(Status.UNAUTHORIZED, null,
error.getError(), false).build();
18      }
19
20      if (id == null) {
21          ErrorGeneral error =
ErrorHelper.handle(ErrorHelper.BAD_REQUEST, "Id can't be null");
22          return
ResponseBuilder.getResponseGeneral(Status.BAD_REQUEST, null,
error.getError(), true).build();
23      }
24
25      try {
26          UUID userId = UUID.fromString(jwt.getSubject());
27
28          AuthGetUsernameRequest authRequest = new
AuthGetUsernameRequest(id);
29          RpcResponse<AuthGetUsernameResponse> authRpcResponse =
authService.getUsername(authRequest, connection);
30          if (authRpcResponse.getError() != null) {
31              ErrorGeneral error =
ErrorHelper.handle(authRpcResponse.getError(),
authRpcResponse.getErrorDetails());
32              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), true).build();
33          }
34          AuthGetUsernameResponse authResponse =
(AuthGetUsernameResponse) authRpcResponse.getData();
35
36          GeoGetDistanceRequest geoRequest = new
GeoGetDistanceRequest(userId, id);
37          RpcResponse<GeoGetDistanceResponse> geoRpcResponse =
geoService.getDistance(geoRequest, connection);
38          if (geoRpcResponse.getError() != null) {
39              ErrorGeneral error =
ErrorHelper.handle(geoRpcResponse.getError(),
geoRpcResponse.getErrorDetails());
40              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), true).build();
41          }
42          GeoGetDistanceResponse geoResponse =
(GeoGetDistanceResponse) geoRpcResponse.getData();
43
44          UserGetUserRequest userRequest = new
UserGetUserRequest(userId, id);
45
```

```
46          RpcResponse<UserGetUserResponse> userRpcResponse =
userService.getUser(userRequest, connection);
47          if (userRpcResponse.getError() != null) {
48              ErrorGeneral error =
ErrorHelper.handle(userRpcResponse.getError(),
userRpcResponse.getErrorDetails());
49              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), true).build();
50          }
51          UserGetUserResponse userResponse = (UserGetUserResponse)
userRpcResponse.getData();
52
53          GetUserResponse response = new GetUserResponse(
54              id,
55              authResponse.getUsername(),
56              userResponse.getName(),
57              userResponse.getAge(),
58              userResponse.getBirthDate(),
59              geoResponse.getDistance(),
60              userResponse.getSex(),
61              userResponse.getProfilePictureUrl(),
62              userResponse.getOccupation(),
63              userResponse.getDescription(),
64              userResponse.getInterests(),
65              userResponse.getMedias()
66          );
67
68          return ResponseBuilder.getResponseGeneral(Status.OK,
response, null, true).build();
69      } catch (Exception exception) {
70          ErrorGeneral error = new ErrorGeneral();
71          error.setStatus(Status.INTERNAL_SERVER_ERROR);
72          error.setError(exception.getMessage());
73          return
ResponseBuilder.getResponseGeneral(Status.INTERNAL_SERVER_ERROR,
null, error.getError(), true).build();
74      }
75  }
```

Ova krajnja točka API pristupnika dohvaća informacije o korisniku na zahtjev trenutnog korisnika. Krajnja točka je zaštićena i agregira tri servisa: autentifikacijski servis, geolokacijski servis i korisnički servis. API pristupnik prvo poziva autentifikacijski servis kako bi dohvatio korisničko ime korisnika, potom zove geolokacijski servis da dobije udaljenost trenutnog od zahtjevnog korisnika i potom korisnički servis da dohvati ostale informacije o zahtijevanom korisniku.



Slika 31. Izgled stranice "Profil korisnika"

3.10.7. Obavijesti

Korisnik može pregledavati obavijesti i označiti ih kao pročitane klikom na određenu obavijest. Obavijesti se nalaze u zaglavlju stranice.

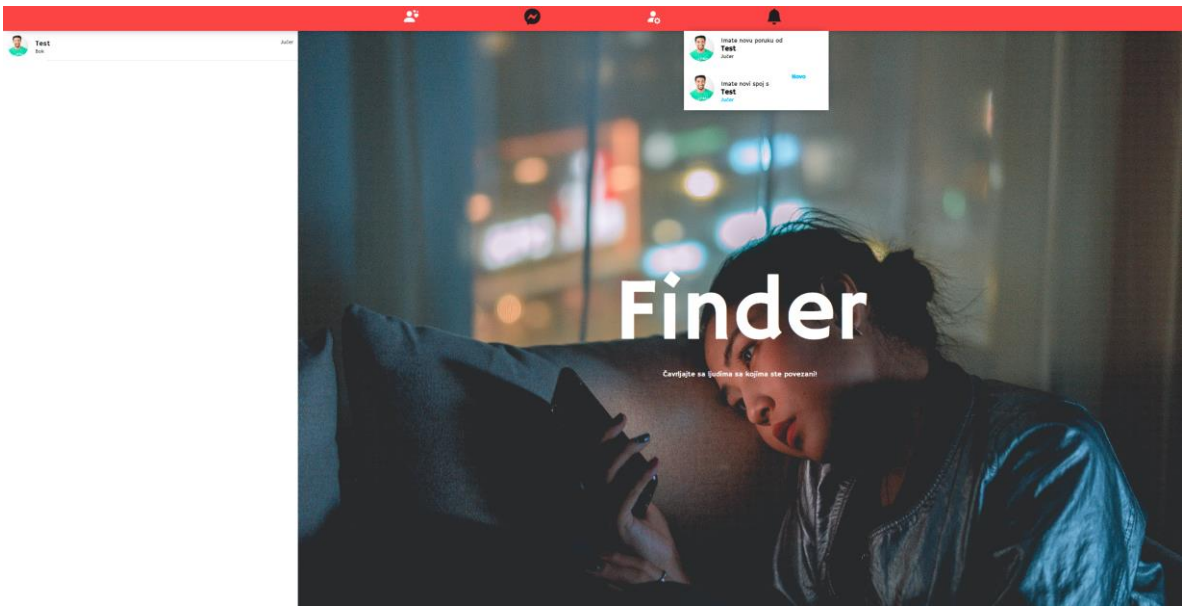
Sljedeće je prikazan kod sa API pristupnika:

```
1  @GET()
2  @Produces({ MediaType.APPLICATION_JSON })
3  @Consumes({ MediaType.APPLICATION_JSON })
4  public Response get(@CookieParam("access_token") Cookie cookie) {
5      Connection connection = rpcClient.getConnection();
6      if (cookie == null) {
7          ErrorGeneral error =
ErrorHelper.handle(ErrorHelper.UNAUTHORIZED, "Access token does not
exists");
8          return
ResponseBuilder.getResponseGeneral(Status.UNAUTHORIZED, null,
error.getError(), false).build();
9      }
10      String accessToken = cookie.getValue();
11
12      DecodedJWT jwt = null;
13      try {
14          jwt = JwtHelper.verifyAccess(accessToken);
15      } catch (Exception exception) {
```

```
16          ErrorGeneral error =
ErrorHelper.handle(ErrorHelper.UNAUTHORIZED, "Invalid access
token");
17          return
ResponseBuilder.getResponseGeneral(Status.UNAUTHORIZED, null,
error.getError(), false).build();
18      }
19
20      try {
21          UUID userId = UUID.fromString(jwt.getSubject());
22
23          NotifGetNotificationsRequest notifRequest = new
NotifGetNotificationsRequest(userId);
24          RpcResponse<NotifGetNotificationsResponse>
notifRpcResponse = notifService.get(notifRequest, connection);
25          if (notifRpcResponse.getError() != null) {
26              ErrorGeneral error =
ErrorHelper.handle(notifRpcResponse.getError(),
notifRpcResponse.getErrorDetails());
27              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), true).build();
28          }
29          NotifGetNotificationsResponse notifResponse =
(NotifGetNotificationsResponse) notifRpcResponse.getData();
30
31          List<UUID> users = new ArrayList<>();
32
33          for (Notification notification :
notifResponse.getNotifications()) {
34              users.add(notification.getSender());
35          }
36
37          UserGetNotificationsInfoRequest userRequest = new
UserGetNotificationsInfoRequest(users);
38          RpcResponse<UserGetNotificationsInfoResponse>
userRpcResponse = userService.getNotificationsInfo(userRequest,
connection);
39          if (userRpcResponse.getError() != null) {
40              ErrorGeneral error =
ErrorHelper.handle(userRpcResponse.getError(),
userRpcResponse.getErrorDetails());
41              return
ResponseBuilder.getResponseGeneral(error.geStatus(), null,
error.getError(), true).build();
42          }
43          UserGetNotificationsInfoResponse userResponse =
(UserGetNotificationsInfoResponse) userRpcResponse.getData();
44
45          List<NotificationResponse> notificationResponses = new
ArrayList<>();
46          for (Notification notification :
notifResponse.getNotifications()) {
47              NotificationUser user =
userResponse.getUsers().stream().filter(item ->
```

```
item.getId().equals(notification.getSender())).findFirst().orElse(nu
ll);
48                  notificationResponses.add(new
NotificationResponse(notification.getId(), notification.getSender(),
user.getName(), user.getProfilePictureUrl(), notification.getType(),
notification.getCreated(), notification.isSeen()));
49              }
50
51              GetNotificationsResponse response = new
GetNotificationsResponse(notificationResponses);
52
53              return ResponseBuilder.getResponseGeneral(Status.OK,
response, null, true).build();
54          } catch (Exception exception) {
55              ErrorGeneral error = new ErrorGeneral();
56              error.setStatus(Status.INTERNAL_SERVER_ERROR);
57              error.setError(exception.getMessage());
58              return
ResponseBuilder.getResponseGeneral(Status.INTERNAL_SERVER_ERROR,
null, error.getError(), true).build();
59          }
60      }
```

Ova krajnja točka API pristupnika je zaštićena i agregira dva servisa: servis obavijesti i korisnički servis. API pristupnik prvo dohvaća pet najnovijih obavijesti trenutnog korisnika i potom šalje listu korisnika koji su vezani uz pojedinu obavijesti. Korisnički servis vraća dodatne informacije o korisnicima, za svaku notifikaciju posebno. API pristupnik na kraju objedinjuje oba odgovora sa servisa i vraća listu obavijesti koja sadrži i dodatne informacije o korisniku koji je vezan uz tu obavijest.



Slika 32. Izgled okvira obavijesti u zaglavlju stranice

4. Zaključak

Izvorne aplikacije i arhitekture za oblak danas su standard za izradu velikih aplikacija koje imaju probleme skalabilnosti, velik broj funkcionalnosti, očekivan velik broj korisnika pogotovo s različitih kontinenata i za koje su potrebni timovi programera. Najveća prednost ovakvih aplikacija je skalabilnost zbog modularne arhitekture koju je lako skalirati horizontalno i time se puno lakše prilagođavaju masivnom broju korisnika nego monolitne aplikacije. Također, timovi su podijeljeni u neke logičke cjeline u kojima svaki tim ima svoju odgovornost, što dovodi do podjele posla i negomilanja odgovornosti na pojedince. Samim skaliranjem ovakvih aplikacija ujedno se dobiva i na brzini sustava, tj. brzini odgovora koje će korisnik imati tijekom rada aplikacija. Budući da je aplikacija podijeljena u smislene module, od kojih svaki ima svoju odgovornost, kao na primjer autentifikacijski mikro servis, takav servis je iskoristiv i za druge projekte jer se modularnom arhitekturom izbjegla ovisnost između pojedinih funkcionalnosti cjelokupne aplikacije. Korištenjem Docker stroja nestaje ovisnost o operacijskom sustavu koji je trenutno instaliran na serveru i svim potrebnim ovisnostima za rad aplikacije, jer se unutar Docker slike nalazi sve potrebno za ispravno pokretanje i rad aplikacije unutar kontejnera. Kubernetes je podigao arhitekturu za oblak na višu razinu zbog toga što je napravio apstrakciju nad upravljanjem kontejnerima i internom mrežom. Također je uvelike olakšao dodavanje novih servera u mrežu i time vertikalno skaliranje. Nedostatak je velika kompleksnost sustava koju u praksi moraju odraditi timovi programera, podijeljeni svaki na različite dijelove, neki na samo programiranje mikro servisa, a neki na izgradnju arhitekture. Time raste i cijena izrade i održavanja ovakvih aplikacija. Kompleksan sustav koji izrađuje više timova istovremeno zahtijeva detaljno planiranje arhitekture te vrlo dobru sinkronizaciju i komunikaciju između timova. Za izradu potrebno je puno znanja, vještine i prije svega iskustva. Inženjeri koje grade arhitekture za oblak nazivaju se DevOps inženjeri.

Izvorne aplikacija i arhitekture za oblak su „de facto“ standard u velikim tvrtkama koje se bave velikom količinom proizvoda, poput Google-a, Facebook-a, Amazon-a, Netflix-a i ostalih giganta, dok su manje tvrtke još zadovoljne sa svojim jeftinijim i jednostavnijim monolitnim rješenjima.

Popis literature

[1] L. De Lauretis, „From Monolithic Architecture to Microservices Architecture“, 2019. [Na internetu]. Dostupno: https://ieeexplore.ieee.org/abstract/document/8990350 [pristupano 21.8.2024.]

[2] CNFC, „Cloud Native Computing Foundation (“CNCF”) Charter“, 2021. [Na internetu].

Dostupno: https://github.com/cncf/foundation/blob/main/charter.md [pristupano 15.5.2023.]

[3] Microsoft, „What is Cloud Native?“, 2022. [Na internetu]. Dostupno: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/definition [pristupano 15.5.2023.]

[4] K. A. Adhiguna, F. M. Rusli, H. Irawan, „Building an ID Card Repository with Progressive Web Application to Mitigate Fraud based on the Twelve-Factor App methodology“, 2021. [Na internetu]. Dostupno: https://ieeexplore.ieee.org/abstract/document/9527413 [pristupano 24.8.2024.]

[5] X. Wang., H. Zhao, J. Zhu, „GRPC: A Communication Cooperation Mechanism in Distributed Systems“, 1993. [Na internetu]. Dostupno: https://dl.acm.org/doi/abs/10.1145/155870.155881 [pristupano 15.5.2023.]

[6] Docker, „What is container?“, 2023. [Na internetu]. Dostupno: https://www.docker.com/resources/what-container/ [pristupano 15.5.2023.]

[7] S. Sharma, R. RV, D. Gonzalez, Microservices: Build scalabile software, Birmingham, UK: Packt Publishing Ldt., 2016.

[8] V. Murugesan, Microservices Deployment Cookbook, Birmingham, UK: Packt Publishing Ldt., 2017.

[9] E. Wolff, Microservices: Flexible Software Architecture, USA: Addison-Wesley, 2016.

[10] GraalVM, „Introduction“, 2024. [Na internetu]. Dostupno: https://www.graalvm.org/latest/docs/introduction/ [pristupano 24.8.2024.]

[11] M. Šipek, D. Muharemagić, B. Mihaljević, A. Radovan, „Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus“, 2020. [Na internetu]. Dostupno: https://ieeexplore.ieee.org/abstract/document/9245290 [pristupano: 24.8.2024.]

Popis slika

Popis tablica

Prilozi (1, 2, …)