Web aplikacije temeljene na mikroservisima

Šimunović, Karlo

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Karlo Šimunović

WEB APLIKACIJE TEMELJENE NA MIKROSERVISIMA

DIPLOMSKI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Karlo Šimunović

Matični broj: 45273/16-R

Studij: Informacijsko i programsko inženjerstvo

WEB APLIKACIJE TEMELJENE NA MIKROSERVISIMA

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Andročec Darko

Varaždin, rujan 2018.

Karlo Šimunović

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovome radu objasniti će se osnovna teorijska podloga web aplikacija, tehnologija koje se koriste za njihov razvoj i najčešće arhitekture aplikacija. Osim teorijske podloge web aplikacija objašnjena će biti i teorija o mikroservisima i mikroservisnoj arhitekturi, koji se problemi javljaju te kako se rješavaju. Navesti će se i nekoliko uzoraka dizajna koji su karakteristični za web aplikacije i mikroservisnu arhitekturu općenito.

Rad će sadržavati praktični dio kojem je cilj primijeniti teorijska znanja navedena u teorijskom djelu rada te će se pomoću njih kreirati web aplikacija naziva Lunch Plan. Lunch Plan web aplikacija temeljena na mikroservisima predstavljati će ogledni primjer takve web aplikacije koji se vrlo lako može nadograditi i nastaviti razvijati u smjeru ozbiljnog projekta.

Preporuka autora je koristiti mikroservise za web aplikacije ukoliko je skalabilnost, podrška za veliki broj korisnika ili podrška za veliki broj tehnologija koje aplikacija koristi ili s kojima komunicira jedan od osnovnih zahtjeva u specifikaciji web aplikacije koja se kreira. U tom slučaju, prema mišljenju autora, mikroservisna arhitektura idealno je rješenje, a uz prave razvojne alate, alate za automatizaciju procesa razvoja i orkestraciju kontejnera razvoj i održavanje web aplikacije temeljene na mikroservisima ne mora biti znatno teže od one razvijene u jednostavnijem skupu tehnologija.

Ključne riječi: web aplikacija; mikroservisi; arhitektura; uzorci dizajna; kontejneri; golang; docker;

Sadržaj

Uvod

Web aplikacije su polako postale dio našeg svakidašnjeg života. S nekima se susrećemo pomoću našeg osobnog računala dok s nekima upravo kroz uređaje koje koristimo još češće, pametne telefone. Iako aplikacije mobilnih uređaja nazivamo mobilnim aplikacijama često su one samo neka vrsta prozora ili sučelja za interakciju s većom web aplikacijom koja se izvršava na nekom poslužitelju i dostupna je putem preglednika.

Web aplikacije s velikim brojem korisnika imaju nekoliko dodatnih zadaća: moraju podržavati taj veliki broj korisnika, promjenjiva opterećenja velikog raspona koje se može promijeniti u svega nekoliko sekundi (npr. objava prijelomne vijesti), konstantno uvoditi novi sadržaj kako bi se privukli ili zadržali novi korisnici i sl. Sve to, mora se odvijati na financijski isplativ način koji je isto tako održiv za razvojne inženjere i ostale koji rade na aplikaciji. Imati poslužitelj za web aplikaciju koji podržava npr. 1 milijun istovremenih korisnika ako je 95% vremena aktivno samo 100 tisuća korisnika nije financijski isplativo, te se tu kao rješenje nameće skalabilnost.

Želja autora je bila istražiti skalabilnost web aplikacija, njihov razvoj, arhitekturu te koji problemi se javljaju i na što treba pripaziti kod takvih web aplikacija. Kao rješenje skalabilnosti kod web aplikacija najviše se spominje mikroservisna arhitektura te je ona odabrana kao temelj ovoga rada.

U ovome radu objasniti će se osnovna teorijska podloga web aplikacija, tehnologija koje se koriste za njihov razvoj i najčešće arhitekture aplikacija. Osim teorijske podloge web aplikacija objašnjena će biti i teorija o mikroservisima i mikroservisnoj arhitekturi, koji se problemi javljaju te kako se rješavaju. Navesti će se i nekoliko uzoraka dizajna koji su karakteristični za web aplikacije i mikroservisnu arhitekturu općenito.

Osim teorijske podloge rad će sadržavati praktični dio kojem je cilj primijeniti teorijska znanja navedena u teorijskom djelu rada te će se pomoću njih kreirati web aplikacija naziva Lunch Plan. Lunch Plan web aplikacija temeljena na mikroservisima predstavljati će ogledni primjer takve web aplikacije koji se vrlo lako može nadograditi i nastaviti razvijati u smjeru ozbiljnog projekta.

1. Web aplikacije

1.1. Općenito

Web aplikacije računalni su programi kojima je klijent web preglednik (eng. *Web browser*). Najčešće su dostupne putem Interneta ili intraneta. Aplikacije dostupne putem Interneta su javne i svi im mogu pristupiti, ukoliko imaju potrebne pristupne podatke, dok s druge strane, intranet web aplikacije su dostupne računalima unutar zatvorene računalne mreže, najčešće u sklopu nekog poduzeća ili organizacije.

Spektar funkcionalnosti koje može pokriti jedna web aplikacija iznimno je širok. Jedan je osnovni primjer web aplikacija Internet stranica s popisom zadataka na koju se može dodavati zapise i brisati ih, popularno nazvana To-do lista. Primjer web aplikacije s kompleksnom funkcionalnosti bila bi Google Docs aplikacija koja omogućava obradu teksta na razini klasičnih računalnih programa za obradu teksta, npr. Microsoft Word (Nations, 2018).

Glavna prednost web aplikacija je ta da su neovisne o platformi i operativnom sustavu s kojega klijent pristupa. Najvažnije je da klijent ima Internetsku vezu i web preglednik kojega redovito ažurira kako bi bez problema podržavao zadnju verziju web aplikacije. Zbog te neovisnosti, danas se često dizajniraju i implementiraju aplikacije koje su odmah na početku namijenjene za računala, tablete i mobilne uređaje kako bi se pokrio što veći broj korisnika. Krajnji korisnik web aplikacije nema brige oko instaliranja, ažuriranja ili održavanja aplikacije jer se cijeli postupak odvija na poslužitelju gdje se aplikacija izvršava, a ne na računalima korisnika, kao što je to kod klasičnih aplikacija.

Internetska veza omogućava web aplikacijama dostupnost u bilo kojem trenutku i iz bilo kojeg kuta svijeta, ali istovremeno predstavlja i njezinu slabu točku. Ukoliko dođe do prekida ili smetnje u mreži od strane korisnika, on će imati problema s korištenjem aplikacije te će biti nezadovoljan, a ukoliko je smetnja u mreži od strane poslužitelja, tada niti jedan korisnik neće moći koristiti aplikaciju, što bi moglo izazvati veliko nezadovoljstvo korisnika.

Kako se aplikacija izvršava na poslužitelju, u slučaju izlaska nove verzije aplikacije korisniku se najčešće samo nakratko daje pravo na izbor između stare i nove verzije, te je nakon određenog perioda prisiljen koristiti novu verziju.

Dok se korisniku olakšava pristup aplikaciji kada se njoj pristupa putem preglednika to otežava razvojni proces razvojnog tima. Podrška više preglednika (eng. *cross-browser compatibility*) otežava posao razvojnog tima zbog velikog broja preglednika i još većeg broja njihovih verzija. Ponašanje web aplikacije može biti različito između dva različita preglednika,

pa čak i između dva ista preglednika različitih verzija (Prednosti i nedostaci web aplikacija - Magma, 2018).

1.2. Povijest

Izumiteljem nama poznatog weba, smatra se Tim Berners-Lee koji je 1990. godine napravio prvu statičnu web stranicu na vlastitom računalu koje je služilo kao web poslužitelj. Statične web stranice prikazuju dokumente u izvornom obliku (npr. tekst, slika, videozapis itd.). Iako je granica između web stranice i web aplikacije subjektivna, u pravilu se web aplikacijama smatraju sve web stranice koje su definirane interakcijom korisnika, njihovim unosom i procesiranjem podataka.

Ono što je omogućilo razvijanje dinamičnih web stranica, odnosno web aplikacija, razvoj je Javascript-a. Netscape Communications 1995. godine predstavlja Javascript, skriptni programski jezik koji se izvršava u pregledniku korisnika te omogućuje programerima razvoj aplikacija s dinamičnim elementima korisničkog sučelja. Otvaraju se i vrata za procesiranje podataka i interakcije korisnika u samom pregledniku, tako da je potrebno manje poziva prema web poslužitelju, što čini korištenje stranice dinamičnijim. Uz Javascript, osnovni skup tehnologija (eng. *technology stack*) web stranica i aplikacija su HTML i CSS. HTML (eng. Hypertext Markup Language) je standardizirani prezentacijski jezik za kreiranje HTML dokumenata koji predstavljaju statične web stranice. U HTML se dokumentu definira struktura pojedine stranice i na njoj se može nalaziti tekst, slike, videozapisi, poveznice na druge stranice i dr. Aktualna verzija HTML-a je HTML5 iz 2014.-te koji omogućava vizualne, audio i video mogućnosti, čak i bez Javascript-a. CSS (eng. *Cascading Style Sheets*) je jezik koji služi za oblikovanje izgleda web stranica.

Novi korak za web aplikacije dogodio se predstavljanjem Macromedia Flash-a 1996. godine, alata za izradu i prikazivanje vektorskih animacija. Također, može se koristiti za igranje igrica u pregledniku, prikazivanje animacija, zvuka i videozapisa. Omogućio je korisniku da ostvari interakciju s web stranicom putem miša, tipkovnice i mikrofona, a sve to odvijalo se u pregledniku korisnika, bez dodatnih poziva na poslužitelja. Vrlo je brzo prepoznata mogućnost oglašavanja kroz animacije na web stranicama te je popularnost Flash-a s vremenom opala.

Godine 1999. prvi se puta spominje koncept web aplikacije u sklopu programskog jezika Java. 2005. godine Jesse James Garret predstavlja tehniku razvoja asinkronih web aplikacija pod nazivom AJAX, koja je osnovna tehnika svake moderne web aplikacije, a omogućuje dohvaćanje podataka sa servera bez osvježavanja stranice (From History of Web Application Development, 2018).

1.3. Tehnologije za razvoj web aplikacija

Nakon detaljnog analiziranja problema i osmišljavanja glavnih aspekata aplikacija kao što su arhitektura, dizajn, radna okolina i sl. potrebno je odrediti u kojem skupu tehnologija će se aplikacija implementirati. Takav skup tehnologija, koji najčešće obuhvaća operacijski sustav na kojem se izvršava aplikacija, web server, bazu podataka i aplikacijski softver, naziva se skup ili stog (eng. *stack*). Općenito, tehnologije vezane za poslužitelj i procese koje korisnik ne vidi, nazivaju se backend tehnologije.

Svaka web aplikacija, osim što se sastoji od tehnologija za gore navedene dijelove aplikacije, također koristi i tehnologije za prikaz informacija korisniku. Tehnologije kojima se prikazuje i upravlja sadržajem, koji je vidljiv korisniku na ekranu, nazivaju se frontend tehnologije. Iako web aplikacija često ima i funkcionalnosti API servisa koji daje informacije u neformatiranom tekstualnom obliku, ovdje se prvenstveno misli na sučelja koja takve aplikacije pružaju.

1.3.1. HTML

HTML skraćenica dolazi od eng. *Hypertext Markup Language* i predstavlja naziv prezentacijskog jezika za izradu web stranica. Trenutna verzija HTML5 nastala je spajanjem HTML4 i XHTML-a u jedan prezentacijski jezik koji omogućava korištenje sintakse oba svoja prethodnika. HTML5 u odnosu na prethodnu verziju donosi nove elemente koji sam se može reproducirati video i zvuk (audio element), te canvas element koji omogućuje crtanje po stranici pomoću Javascript-a. Osim navedena tri elementa donosi i strukturu u HTML stranice koja nalikuje onoj kod knjiga ili XML dokumenata. Hipertekst se dokumenti stvaraju pomoću HTML jezika koji se sastoji od definiranih oznaka (eng. *tag*) koje imaju svoj početak i kraj. HTML početne oznake prepoznaju se po tome što sadrže ključnu riječ, npr. div, okruženu znakovima veće/manje, npr. . Nakon početne oznake dolazi sadržaj ili druge ugniježđene oznake te naposljetku dolazi završna oznaka koja u sebi ima kosu crtu, npr. .

Iako je HTML jednostavan i lako se uči, njime se mogu kreirati kompleksne strukture dokumenata, poveznice između njih i prikazivati raznorazni sadržaji. HTML nije programerski jezik jer se njime ne mogu izvršavati nikakve naredbe ili zadaće poput zbrajanja brojeva. Za prikaz HTML dokumenata dovoljno ih je otvoriti u web pregledniku (Volarić, 2018).

1.3.2. CSS

CSS (eng. *Cascading Style Sheets*) stilski je jezik kojim se oblikuju i prezentiraju HTML dokumenti. CSS datotekama definiramo izgled i stil HTML dokumenta koji čini izrađenu web stranicu ili web aplikaciju. Primjena definiranih stilova bazira se na CSS pravilima koji se kaskadno primjenjuju ovisno o pravilu, na sve elemente, samo na neke elemente ili na točno određeni element, odnosno HTML oznaku.

Stil se može uređivati direktno u HTML kodu pomoću atributa *style,* no tu se javlja problem ponavljanja stilova. Na primjer, da se želi postići da svi linkovi na drugim dokumentima izgledaju isto, to bi zahtijevalo da svaka oznaka za link ima isti *style* atribut. Time se miješa prezentacijski kod i kod stilova što otežava rad i izmjene na dokumentu. Primjenom CSS stilova u odvojenoj datoteci ili mapi stilova, dobiva se mogućnost bržeg i preglednijeg razvoja vizualnog identiteta web aplikacije.

Trenutna je verzija CSS-a verzija 3, a ona proširuje mogućnosti prethodne verzije CSS 2.1. Od CSS je verzije 3 CSS podijeljen u module te se više ne razvija kao cjelina, nego kao svaki modul zasebno. To omogućava brži razvoj pojedinih modula i nadopunjavanje novim funkcionalnostima, dok s druge strane otežava implementaciju kod preglednika. Naime, opće je poznato da preglednici različitih proizvođača ne prikazuju jednako definirane stilove, čak štoviše, prikaz istih CSS stilova može varirati između dvije uzastopne verzije istoga preglednika. Internet Explorer Microsofta prednjači kao preglednik s najviše problema u prikazu CSS stilova, pogotovo kod svojih starijih verzija. Iako Microsoft ima novu verziju svog preglednika pod nazivom Edge, IE se i dalje koristi u velikom broju, te se često nalazi u praksi kada se razvijaju aplikacije za poduzeća, npr. intranet aplikacije. (Mujadžević, 2008).

1.3.3. Javascript

Javascript je objektno orijentiran skriptni jezik koji se najčešće koristi kako bi se statičnim stranicama dodala dinamična svojstva. Javascript je razvila firma Netscape koja 1995. godine objavljuje njegovu prvu verziju. Javascript se aktivno razvija i danas te je standardiziran od strane organizacije ECMA, koja standardizira i ostale skriptne jezike.

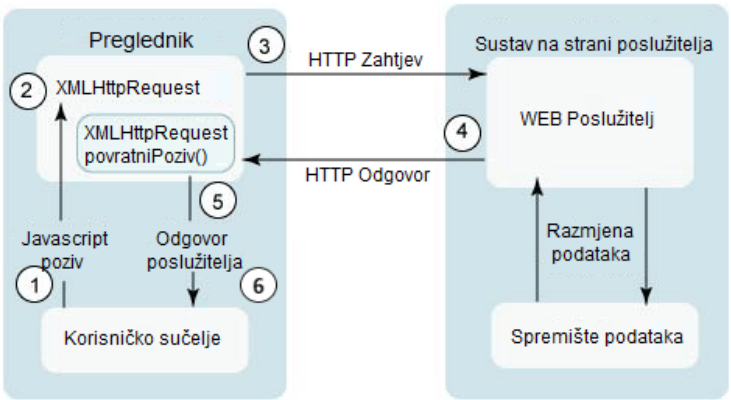
Uključivanjem Javascript-a u HTML kod, dobivamo kontrolu nad sadržajem koji se prikazuje ili kojega korisnik unosi, npr. provjera popunjenog obrasca. Pokretanje Javascript koda najčešće se bazira na događajima koji se kreiraju od strane korisnika aplikacije, preglednika ili čak poslužitelja. Podržan je veliki broj događaja od kojih su neki: click, focus, change, load i dr. Javascript je uključen u većinu modernih web preglednika.

Njegova namjena nije samo korištenje na klijentskoj strani korisnika u Internet pregledniku. Naime, od 2009. godine kada je predstavljen Node.js, Javascript se koristi i na

serverskoj strani web aplikacije, tzv. backend. Od tada se Javascript proširio i na razvoj izvornih (eng. *native*) aplikacija za mobilne uređaje (React Native) te je time postao jedan od najboljih programskih jezika za razvoj aplikacija koje će se pokretati na više platformi. Razvoj u Javascript-u uvelike olakšavaju velik broj dostupnih razvojnih okruženja (eng. *framework*) i dodatnih biblioteka (eng. *library*). Javascript nije bez svojih nedostataka. Glavni nedostatak je sigurnost jer se izvršava na strani klijenta pa je teoretski moguće da napadač (hacker) u svoju HTML stranicu ubaci isječak malicioznog Javascript koda koji se izvrši pri posjetu korisnika (Stančer, 2018).

1.3.4. AJAX

Ajax sam po sebi nije tehnologija, nego tehnika koja koristi druge web tehnologije, ali zbog svoje rasprostranjenosti i prednosti koje donosi, dio je svake moderne web aplikacije. Skraćenica *Ajax* dolazi od eng. *Asynchronous JavaScript and XML*, što naglašava njegovu najveću prednost, asinkronost. Asinkrona će i sinkrona komunikacija biti detaljnije obrađene u kontekstu servisa u mikroservisnoj arhitekturi u nastavku rada. Godine 2005., Jesse James Garret prvi puta spominje naziv *Ajax* u svojem radu „Ajax: A New Approach to Web Application“. Nakon toga i drugi počinju koristit naziv *Ajax* za tehniku dohvaćanja podataka sa servera bez osvježavanja stranice (eng. *refresh*) (Satyasree, 2018).



Slika 1. Postupak izvođenja Ajax tehnike (How AJAX works?, 2018)

Na Slika 1 vidi postupak izvođenja Ajax tehnike kojom se, u ovom slučaju, šalje zahtjev na poslužitelja i dohvaćaju se podaci iz baze. Aktivnosti se odvijaju na sljedeći način:

1. Korisnik svojom interakcijom u korisničkom sučelju ili automatskom brojaču poziva Javascript funkciju kojom se obavlja Ajax poziv.
2. U toj se funkciji može nalaziti i obrada ili skupljanje podataka, npr. upisanog obrasca, te se kreira objekt tipa XMLHttpRequest koji je zadužen za upravljanje HTTP pozivima.

Često je taj objekt u današnjem vremenu zamaskiran u poziv funkcije neke biblioteke, npr. $.ajax funkcija biblioteke jQuery. XMLHttpRequest se objektu dodaju obrađeni podaci iz obrasca, odrede se atributi i zaglavlje zahtjeva, npr. metoda slanja (GET, POST, itd.), vrijeme čekanja na odgovor (eng. *timeout*), vrsta podatka koji se šalje (eng. *encoding*), povratni poziv (eng. *callback*) i sl.

3. Nakon što je zahtjev pripremljen, inicira se njegovo slanje koje preuzima preglednik korisnika i on obavlja sve potrebne zadatke koji se odnose na komuniciranje klijenta, u ovom slučaju web preglednika i poslužitelja kako bi oni međusobno razmijenili podatke.
4. Zahtjev se sada nalazi na strani poslužitelja, gdje se on nekim programskim jezikom poslužitelja, npr. PHP, Java, Phyton, Ruby, Javascript, dalje obrađuje. Rezultat obrade poslužitelja može biti bilo što, od jednostavnog tekstualnog izraza, sadržaj članka, do HTML koda i Javascript objekata. U sklopu obrade poslužitelj nerijetko izvršava i upite na bazi podataka, tako da se i oni mogu nalaziti u rezultatu, tj. odgovoru poslužitelja. Poslužitelj zatim generira odgovor u obliku kojeg funkcija pozivatelj očekuje, npr. JSON, i vraća ga pregledniku.
5. Nakon što je preglednik zaprimio odgovor poslužitelja, vraća ga funkciji koja je i podnijela zahtjev pomoću specificiranog povratnog poziva. Povratnim se pozivom (eng. *callbacks*) određuje koja će se funkcija pozvati s odgovorom poslužitelja kao argumentom, nakon što preglednik zaprimi njegov odgovor. Kako se komunikacija odvija asinkrono, izvršavanje programa koji koristi Ajax nije slijedno, npr. sljedeća linija koda, potrebno je odrediti povratni poziv.
6. Zaprimljen odgovor poslužitelja u obliku argumenta funkcije obrađuje se na korisničkoj strani, ako je potrebno, te se prikazuje korisniku.

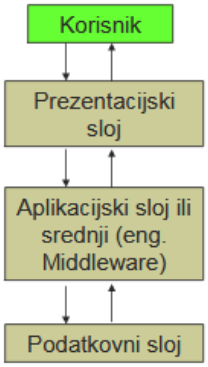
1.4. Arhitektura

Arhitektura se programskoga proizvoda definira kao struktura komponenata, njihova povezanost, principi i upute koje su vodile evoluciji razvoja. Kreiranjem se arhitekture pokušava razdvojiti zahtjeve funkcionalnosti i kvalitete u programske komponente i njihove veze, koristeći neki od razvojnih pristupa. Ovisno o točki gledišta, neke će se komponente više istaknuti, a neke manje. Takvim strukturiranjem sustava i njihovim razdvajanjem u različite perspektive pomaže se u shvaćanju samoga sustava koji se razvija te se tako pridonosi rješavanju arhitekturnih problema (Adamkó, Part II. Architectures for the Web, 2018).

1.4.1. Višeslojna arhitektura

Kod razvoja se web aplikacija često sama aplikacija strukturira u više dijelova ili slojeva. Primjer je takve arhitekture višeslojna ili N-slojna arhitektura, gdje N označava broj slojeva. Kada se priča o višeslojnoj arhitekturi, najčešće se misli na 3-slojnu arhitekturu koja aplikacijske komponente razdvaja na prezentacijski, aplikacijski i podatkovni sloj.

Prezentacijski je sloj u direktnom kontaktu s korisnikom, prikazuje korisniku sučelje i informacije i održava interakciju s korisnikom. Za komunikaciju s korisnikom, prezentacijski sloj koristi preglednik korisnika. Sloj koji se nalazi u sredini naziva se aplikacijski sloj te je zadužen za upravljanje aktivnostima koje pojedina web aplikacija treba izvršiti na temelju interakcije korisnika. Taj je sloj također zadužen za komunikaciju prezentacijskog i podatkovnog sloja te on upravlja tokom aplikacije. Najniži sloj troslojne arhitekture je podatkovni sloj. Na njemu se nalaze podaci i logika pristupa te spremanja podataka. Važno je napomenuti da prezentacijski sloj nema direktan pristup podatkovnom sloju, već svim zahtjevima upravlja aplikacijski sloj (Zekić-Sušac, 2018) (Adamkó, Chapter 4. Layered Architecture for Web Applications, 2018)



Slika 2. Prikaz slojeva troslojne arhitekture (Zekić-Sušac, 2018)

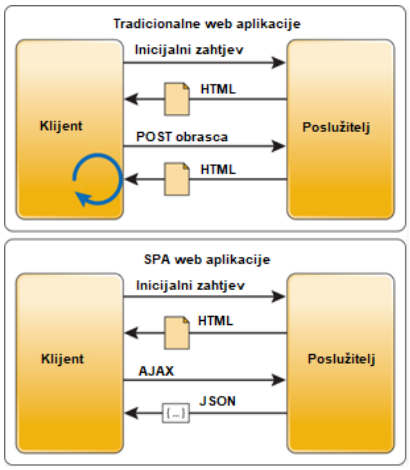
Razdvajanje u slojeve (Slika 2) olakšava dizajn i planiranje aplikacije, razvoj funkcionalnosti programerima te, u konačnici, olakšava i njezino održavanje. Iako je troslojna arhitekture najčešća, broj slojeva je proizvoljan. Primjer proizvoljnih slojeva može biti sloj osnovnih servisa, često zvan platforma, sloj poslovne logike koji se isprepliće s aplikacijskim slojem i dr.

1.4.2. Jednostrane aplikacije

Jednostrane su aplikacije (eng. *single-page applications – SPA*) web aplikacije koje učitaju jednu HTML stranicu i prema interakciji korisnika, dinamično ažuriraju sadržaj stranice. Kako bi postigle svoju dinamiku i fluidnost, SPA koriste AJAX tehniku dohvaćanja podataka i funkcionalnosti HTML5, kako se stranice ne bi neprestano osvježavale. Problem koji se javlja kod SPA je taj da se sva logika prikaza sučelja izvršava u pregledniku korisnika. Otvaranje je web aplikacije zahtjevniji proces nego što je to slučaj kod klasičnih, više straničnih aplikacija, budući da se tada učitava cijela aplikacija koju korisnik može koristiti, a ne samo funkcionalnost

stranice koju otvara. Nakon učitavanja svih potrebnih datoteka i skripta za rad aplikacije, svaka se korisnikova interakcija obrađuje velikim dijelom na korisnikovoj strani, dok se u pozadini komunicira s poslužiteljem kako bi se omogućila neka druga funkcionalnost, kao npr. spremanje dokumenta na kojem radimo.

Pozadinska komunikacija s poslužiteljem, odnosno sva komunikacija s poslužiteljem nakon inicijalnog učitavanja, odvija se pomoću AJAX poziva koji, kao svoj rezultat, vraća najčešće samo podatke koji su u JSON ili XML formatu. Aplikacija putem Javascript-a koda koji se izvršava u pregledniku korisnika, čita JSON podatke i ažurira postojeću stranicu s novim informacijama bez osvježavanja.



Slika 3. Usporedba tradicionalnih i SPA aplikacija (Wasson, Single-Page Applications, 2018)

Slika 3 prikazuje slijed aktivnosti između klijenta i poslužitelja te kako se on razlikuje od tradicionalnih web aplikacija. Kod tradicionalne aplikacije inicijalni zahtjev učita stranicu te se ona prikaže korisniku. U ovom primjeru, korisnik popunjava obrazac te ga dalje šalje na poslužitelj. Prilikom slanja, obrazac se šalje u zahtjevu POST metodom HTTP-a, poslužitelj ga obradi te, kako bi se učitao odgovor poslužitelja na obrazac, stranica se osvježi. Dio s učitavanjem stranice inicijalnim zahtjevom jednak je u oba slučaja, kod tradicionalne i kod SPA aplikacije. No, kako bi se kod SPA izbjeglo osvježavanje, koristi se prethodno opisana Ajax tehnika.

Jedna od očitih prednosti SPA je prethodno navedena fluidnost i odziv same aplikacije, gdje nema više osvježavanja stranice ili njezinog ponovnog prikazivanja, što čini cijelo korisničko iskustvo lošijim. Iako ne tako očita, ali jednako važna prednost, je i ta da se slanjem zahtjeva poslužitelju u obliku JSON formata, ili nekog drugog formata tipa XML, izdvaja prezentacijski sloj web aplikacije. U slučaju SPA aplikacija na taj način izdvaja se HTML kod koji predstavlja prezentacijski sloj od aplikacijska logike, AJAX zahtjeva i JSON odgovora. U idealnim slučajevima, zbog te se prednosti može u potpunosti izmijeniti programski kod

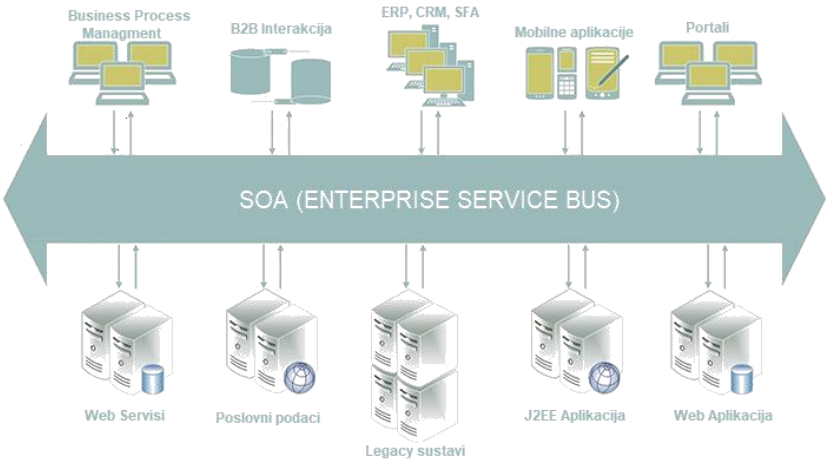
prezentacijskog sloja, bez da se uopće mijenja kod aplikacijskog sloja. Tada se nakon inicijalnog učitavanja poslužitelj ponaša kao servisni sloj koji pruža krajnje točke (eng. *endpoints*). Javascript aplikaciji nije bitna implementacija poslužitelja pa se mogu raditi i izmjene na strani poslužitelja, sve dok se ne promjene krajnje točke. Ova se prednost često iskoristi tako da se napravi klijent za mobilne uređaje koji koristi krajnje točke bez izmjena na poslužitelju (Wasson, ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET, 2018).

1.4.3. Servisno orijentirana arhitektura (SOA)

Servisno orijentirana arhitektura (eng. *service-oriented architecture - SOA*) je arhitektura programskih rješenja koja koristi servise dostupne preko mrežnih tehnologija poput Interneta. Glavni su ciljevi korištenja SOA arhitekture labavo povezivanje programskih komponenti, registar dostupnosti servisa i kontrola pristupa servisima.

Programske se komponente pretvaraju u zasebne servise. Servisi su implementacije prepoznatih i definiranih poslovnih funkcionalnosti koji su kao takvi dostupni na korištenje korisnicima kroz njihove aplikacije ili drugim poslovnim procesima. Servisi također imaju dobro definirana programska sučelja (eng. *interfaces*) za lakše povezivanje s drugim komponentama sustava, bili oni drugi servisi ili druge aplikacije. Tako definirani servisi neovisni su međusobno jedni od drugih te je njihova implementacija neovisna o ostatku sustava, dokle god imaju usklađene na-čine komuniciranja kroz sučelja. SOA omogućuje i isplativost investicije programerskim tvrtkama jer im dozvoljava ponovno korištenje takvih servisa u drugim aplikacijama (Mahmoud, 2018).

Pružanje mehanizma za objavljivanje dostupnih servisa u obliku registra dostupnih servisa također je sastavni dio SOA-e. Servisi se registriraju kod javnih registara servisa koje korisnici pretražuju kako bi našli servis koji odgovara njihovim potrebama. Ako takav servis postoji, registar pruža korisniku programsko sučelje, odnosno funkcionalnosti i zahtjeve upisa/ispisa servisa, te adresu krajnje točke tog servisa. Zbog javnih registara servisa, programeri mogu jednostavno pronaći potrebne servise i uključiti ih u svoju aplikaciju. Primjer takvih javnih registara su standardi WSDL, SOAP i UDDI (Rouse & Nolle, What is service-oriented architecture (SOA)? - Definition from WhatIs.com, 2018).



Slika 4. Prikaz SOA-e i ESB-a (Aprilindo Sentosa, 2018)

Za integraciju, komunikaciju i orkestriranje rada servisa kod SOA-e često se koristi Enterprise Service Bus, skraćeno ESB, programska komponenta koja djeluje kao most između servisa i drugih dijelova sustava (Slika 4). Za komunikaciju sa servisima koristi najčešće redove poruka, a jedna od zadaća joj je i podjela poslova. ESB podrža više kopija iste komponente kako bi se poboljšale performanse aplikacije ili oporavio sustav od ispada. Kako je ESB centralno mjesto aplikacije, praktično je da se u njoj nalaze funkcionalnosti važne za cijeli sustav poput sigurnosnih kontrola, kontrole pristupa, autentifikacije i vođenje dnevnika (Rouse, Churchville, & Nolle, What is enterprise service bus (ESB)? - Definition from WhatIs.com, 2018).

2. Mikroservisi

2.1. Općenito

Mikroservisna se arhitektura sastoji od malih servisa, zvanih mikroservisi (dalje u tekstu: servisi), autonomnih programskih komponenti koje pružaju usluge drugim servisima i s kojima surađuju. Cilj razvoja servisa u mikroservisnoj arhitekturi je da u konačnici servis bude što manji, optimiziran i fokusiran isključivo na svoju domenu.

Servisi su alternativno rješenje monolitnim sustavima koji u sebi sadrže jednu aplikaciju zaduženu za kompletnu interakciju s korisnikom i bazom podataka. S vremenom, kako se radi na aplikaciji, količina programskog koda sve više i više raste. Ciljane promjene postaju teže jer se sve teže snalaziti u sve većem broju linija koda. Iako se pri razvoju koristi uzorke za odvajanje dijelova aplikacije te za strukturiranje koda unutar programske komponente, te komponente postaju sve kompliciranije. Lako se dogodi i da funkcije i metode povezuju više domena odjednom, što otežava implementaciju nove funkcionalnosti ili rješavanja grešaka. Ukoliko se pri razvoju monolitne aplikacije imalo na umu da programski kod čini koheziju, odnosno da su zajednički koncepti čvrsto povezani, tada je prelazak iz postojeće monolitne aplikacije u mikroservisnu arhitekturu nešto jednostavniji.

Kod servisa kohezija postiže se njihovom neovisnošću. Kako bi najlakše odredili doseg jednog servisa, modelira ga se prema granicama poslovne domene. Navedeno također pomaže da se izbjegne prekomjerno proširivanje servisa kako ga ne bismo dodatno zakomplicirali.

Komunikacija se između servisa odvija putem mrežnih poziva, čime se odmah na početku naglašava granica između pojedinih servisa i izbjegava čvrsto povezivanje, odnosno njihova međuovisnost. Kada se navodi kako servisi moraju biti međusobno neovisni, time se želi reći da se isti trebaju moći mijenjati i razmještati (eng. *deploy*), neovisni jedni o drugima i bez promjene njihovih potrošača (eng. *consumer*). Autonomija se servisa čuva kontroliranjem svega što pružaju servisi aplikacije. Ukoliko servis dijeli previše svoje funkcionalnosti, dolazi do mogućega kontra efekta, a taj je da korisnici servisa postanu ovisni o internoj logici toga servisa.

Servisi koriste aplikacijsko programsko sučelje (eng. *application programming interface - API*) za komuniciranje i surađivanje s drugim servisima putem mreže. Svaki servis pruža svoje sučelje drugim servisima pa odabir tehnologije sučelja nije trivijalan zadatak. Krivim se izborom tehnologije sučelja ograničavaju ostali servisi na korištenje iste te tehnologije, čime se isključuje mogućnost korištenja jedne od glavnih prednosti servisa.

Upotrebom se servisa ne mora brinuti o usklađenosti tehnologija unutar samih servisa u aplikaciji. Upravo zbog te neovisnosti između servisa, potpuno je svejedno koje tehnologije koristi koji servis, sve dok ti servisi imaju „zajednički jezik“, API. To omogućava odabir odgovarajuće tehnologije za svaki zadatak, skraćivanje vremena razvoja i poboljšane performanse aplikacije. Kod tradicionalnih monolitnih aplikacija stvari se kompliciraju ako tehnologija koja se koristi u cijeloj aplikaciji nije najbolji alat za neki zadatak ili ta funkcionalnost uopće nije podržana. U tom je slučaju potrebno napraviti svoju implementaciju potrebnog algoritma u korištenoj tehnologiji ili spajati dvije tehnologije pomoću, npr. uzorka Adapter, što otežava daljnji razvoj (Newman, Building Microservices: Designing Fine-Grained Systems, 2015).

2.2. Razlika između SOA-e i mikroservisa

Newman (2015.) navodi kako je glavna razlika između mikroservisne arhitekture i SOA-e u tome što SOA ne daje programerima i ostalim razvojnim inženjerima upute kako da od monolita napraviti servisnu arhitekturu. Ne definiraju se striktne granice i predviđena veličina pojedinog servisa, ne navodi se arhitekt koji može podijeliti monolitnu aplikaciju u servise niti koje bi granulacije trebala biti nova arhitektura. Na sve te odluke dijelom utječu i proizvođači programske potpore svojim izborom tehnologija i implementacijom SOA-e. Kod SOA-e su često izostavljeni praktični primjeri, kako pravilno odvojiti servise jednih od drugih, a upravo iz toga proizlaze problemi povezani sa SOA-om. Mikroservisna se arhitektura razvila iz stvarnih primjera te se može, na neki način, smatrati posebnim SOA pristupom.

Tablica 1. Prikaz razlika između SOA-e i mikroservisne arhitekture

SOA	Mikroservisna arhitektura
Maksimiziranje ponovnog korištenje servisa	Fokusira se razdvajanje
Sustavna promjena zahtjeva izmjenu monolita	Sustavna promjena znači dodavanje novog servisa
DevOps i Continuous Delivery postaju popularni ali ne nužni	Veliki fokus na DevOps i Continuous Delivery
Fokus na ponovnom korištenju poslovne funkcionalnosti	Veća važnost na koncept ograničenog konteksta
Za komunikaciju se koristi Enterprise Service Bus (ESB)	Za komunikaciju se koriste šturi i jednostavni sustavi poruka
Podržava nekoliko protokola poruka	Koristi jednostavne protokole poput HTTP-a, REST-a ili RPC-a
Koristi zajedničku platformu za razmještanje svih servisa	Aplikacijski poslužitelji se ne koriste, normalno je da se koriste platforme u oblaku
Korištenje kontejnera poput Docker-a nije toliko popularno	Kontejneri rade vrlo dobro s servisima
SOA servisi dijele spremište podatka	Svaki servis može imati svoje spremište podataka
Zajedničko upravljanje i standardi	Opušteno upravljanje, s većim naglaskom na suradnju timova i slobodu izbora

Izvor: (Despodovski, 2018)

2.3. Modeliranje servisa

Promatranjem se servisa u mikroservisnoj arhitekturi može reći da su im zajedničke osobine slabo povezivanje i visoka kohezija.

2.3.1. Osobine dobro definiranog servisa

Slabo se povezivanje (eng. *loose coupling*) u mikroservisnoj arhitekturi koristi kako bi imali mogućnost mijenjanja jednog servisa bez utjecaja ili izmjena drugog servisa te je to jedna od najvažnijih prednosti servisa. Slabo povezani servisi ne znaju puno jedni o drugima, čak i kada neposredno surađuju. Ako se pri razvoju ne uzme u obzir da se sva komunikacija odvija kroz mrežu, tada može doći do problema s performansama te je zato bitno pripaziti na „pričljivost“ između servisa. „Pričljivost“ između dva servisa dovodi i do čvršće povezanosti između njih, što je isto suprotno od onoga što se želi postići.

Cilj je visoke kohezije (eng. *high cohesion*) grupiranje logike programa u povezane cjeline. Svaka cjelina sadrži logiku i ponašanje koje je usko vezano za njezin kontekst, dok je sve ostalo, što nije direktno vezano za taj kontekst, u drugoj cjelini. Pronalaženjem granica cjelina osigurava da je ponašanje povezano s jednim kontekstom u jednoj cjelini, a drukčije ponašanje u drugoj cjelini, drugim riječima, granice cjelina kreiraju se prema njihovim kontekstima. Te se cjeline pretvaraju u servise pa se kod zahtjeva za promjenom ponašanja pojedinog servisa ono promijeni samo na jednom mjestu te je servis znatno brže spreman za produkcijsko okruženje (eng. *production environment*). U slučaju da se funkcionalnost proteže kroz više servisa, kao što je to često slučaj kod pogrešno grupiranih cjelina, morat će se za promjenu funkcionalnosti izmijeniti nekoliko servisa i možda će ih se morati pustiti u produkciju u isto vrijeme. Mijenjanjem većeg broja servisa od jednom, drastično se povećava rizik od pogreške te je sami postupak izmjene puno teži, budući da se izvodi na nekoliko servisa (Newman, Building Microservices: Designing Fine-Grained Systems, 2015).

2.3.2. Ograničavanje konteksta

Ograničeni je kontekst koncept koji dobro opisuje ograničenost domena servisa. Za primjer se može uzeti domena nekog programskog proizvoda poput ERP-a, skraćeno od eng. *Enterprise Resource Planning.* Domena su ERP-a poslovne aktivnosti poduzeća od kojih svaka predstavlja upravo jedan ograničeni kontekst, poput nabave, skladišta, financija, proizvodnje, prodaje i dr. Unutar svakog konteksta postoje informacije koje su važne samo tom kontekstu te ih nije potrebno dijeliti izvan njega. No, kako bi program pružio bolju sliku poduzeća, zato postoje informacije koje se razmjenjuju između određenih ograničenih konteksta. Svaki ograničeni kontekst ima svoje eksplicitno sučelje kojim određuje koje

informacije dijeli i na koji način s ostalim ograničenim kontekstima. Kod komunikacije između dva takva konteksta, kontekst koji želi informaciju zahtjeva ili želi izvršiti neku funkcionalnost, komunicira s eksplicitnim sučeljem drugog konteksta (Evans, 2003).

Prisjećajući se primjera ERP-a i promatranjem komunikacije između ograničenih konteksta financija i skladišta, može se uočiti više različitih modela od kojih su neki skriveni, a neki dijeljeni pomoću eksplicitno sučelja. Skladište ima informacije o tome koliko viličara ima na korištenje, koliko je polica i redova, na kojoj se polici nalazi pojedini proizvod, količini proizvoda i sl. S druge strane, odjel financija posjeduje informacije o knjiženjima, financijske planove, razne izvještaje i sl. Kako bi financije mogle sastaviti račun dobiti i gubitka, potrebne su informacije, tj. pristup modelu, zaduženom za količinu proizvoda koji se nalazi u kontekstu, odnosno, odjelu skladišta. Pojavom takvog slučaja, model „količina proizvoda“ postaje dijeljeni model između dva ograničena konteksta koji se dijeli kroz eksplicitno sučelje. Odjel financija ne zanima na kojoj se polici i u kojem se redu nalazi koji proizvod, tako da je količina informacije koja se dijeli svedena na minimum.

2.4. Integracija servisa

Uzevši u obzir tehnologije koje se koriste u mikroservisnoj arhitekturi, tehnologije integracije imaju najveću važnost. Ispravnim odabirom tehnologije integracije ona se neće morati mijenjati za vrijeme razvoja aplikacije, čime se izbjegava teška izmjena svih servisa zbog takve promjene. Isto tako, odgovarajuća tehnologija dopušta izmjenu servisa i podataka koje šalje servis, bez ili uz minimalne izmjene korisnika tog servisa. Korištenjem tehnologija integracije koje nisu specifične za jednu platformu (neke su podržane od velikog broja platformi i programskih jezika), ostavljene su otvorene opcije za odabir drugih tehnologija implementacije servisa.

Otvorene opcije kod izbora tehnologija nisu bitne samo za integraciju između servisa, nego i za komunikaciju s korisnikom. Cilj je korisniku ostaviti na odabir tehnologiju kojom će konzumirati servis jer se samim time smanjuje povezanost između servisa i korisnika. S druge strane, moguće je rješenje isporučiti korisniku dijeljenu biblioteku (eng. *shared library*) koja će u sebi sadržavati logiku integracije korisnika i servisa, no ta će biblioteka čvršće povezivati servise i korisnika, što je suprotno od onoga što se želi postići ovom arhitekturom.

Za korisnika je bolje da ne poznaje internu implementaciju servisa, već da je samo „slijepo“ koristi. Time se održava slabo povezivanje između servisa i korisnika. U slučaju da korisnik ovisi o internoj implementaciji, izmjenom implementacije servisa može doći do greške ili kvara kod korisnika. Strah od takve greške stoji na putu novim funkcionalnostima i

izmjenama servisa pa je bolje izabrati tehnologiju koja ne otkriva internu implementaciju servisa (Newman, Building Microservices: Designing Fine-Grained Systems, 2015).

2.4.1. Baza podataka

Jedna je od najčešćih metoda integracije nekoliko servisa bazom podataka. Ta je metoda dobila na svojoj popularnosti zbog sličnosti s integracijom kod monolita. Kako bi servisi razmjenjivali informacije, između sebe koriste bazu podataka kao posrednika, a ona također upravlja zahtjevima za izmjenu tih informacija. Lako se može vidjeti da je to poželjan način upravljanja podacima i integracijom programskih komponenti kod monolitnih aplikacija.

Problem je ovog pristupa što omogućuje međusobno čitanje i pisanje podataka među kontekstima, a to su u monolitu najčešće moduli ili programske komponente, čak i kada su oni kontekstualno nepovezani. Kod monolitnih se aplikacija to ne smatra problemom jer smo „naučili“ konstanto propagirati promjene jedne komponente kroz ostale i održavati shemu baze podataka. No, time se gubi sami smisao mikroservisne arhitekture. Ako svi servisi ovise o jednoj bazi podataka i njezinoj shemi, u konačnici se dobiva aplikacija s karakteristikama monolitne aplikacije, bez slabe povezanosti i visoke kohezije.

Najčešći odabir baze podataka kod jednostavnih web aplikacija jesu SQL relacijske baze podataka, iako se to s razvojem može promijeniti. U početku se razvoja mogu predvidjeti zahtjevi koji ne odgovaraju relacijskim bazama podataka za pojedine servise. Tada je bolji izbor NoSQL nerelacijska baza podataka. Odabirom tehnologije baze podataka ograničavaju se i svi servisi na navedenu tehnologiju. To znači da, iako postoje servisi kojima više odgovora NoSQL tip baze podataka, ipak će se morati koristiti SQL bazom podataka ili će biti potrebno napraviti skupu i opsežnu izmjenu cijele aplikacije, kako bi se iskoristile prednosti drugih tehnologija.

Korištenjem integracije, bazom se podataka često može kod korisnika potkrasti interna implementacija upravljanja podacima koja bi se trebala nalaziti isključivo na strani aplikacije, a kada je cilj slabo povezivanje. Kako se ne bi stvorile greške ili kvarovi kod korisnika, ne mogu se raditi izmjene na servisima, već se izmjene moraju pomno planirati, što dovodi i do straha od izmjena, odnosno straha od potencijalne greške (Newman, Building Microservices: Designing Fine-Grained Systems, 2015).

2.4.2. Sinkrona i asinkrona komunikacija

Kada se govori o sinkronoj komunikaciji kao načinu integraciji servisa u cjelini, misli se na *zahtjev/odgovor* (eng. *request/response*) način komunikacije. Kod načina *zahtjev/odgovor*, uspostavi se poziv prema poslužitelju od kojeg se očekuje odgovor. Veza je između klijenta i

poslužitelja u komunikaciji aktivna za vrijeme čekanja odgovora te blokira određene resurse sudionika. Sinkrona komunikacija olakšava razvoj web aplikacije jer, kada se jednom uputi poziv, dobije se na njega odgovor pa se na temelju toga odlučuje je li radnja uspješno izvršena ili nije. S druge strane, dugi blokirajući pozivi mogu usporiti aplikaciju, ne samo jednog klijenta, nego i svih ostali koji koriste isti servis. Ako se uzmu u obzir i mobilni uređaji sa svojom nestabilnom mrežom, dobiva se web aplikacija koja lako može odbiti korisnika od sebe zbog dugotrajnih čekanja i sporih dohvaćanja podataka.

Za upravljanje procesa koji se protežu kroz nekoliko poslovnih konteksta i samim time nekoliko servisa koristi se tehnika upravljanja procesa zvana orkestracija. Orkestracijom se uvodi dodatni servis koji služi kao centralno upravljačko mjesto zaduženo za pozivanje servisa određenim redoslijedom uz ispunjene uvjete. Vidljivo je da će mjesto s takvom odgovornosti vrlo brzo sadržati veliku količinu implementacijske logike kako efikasno upravljati servisima te će taj servis postati jedinstvena točka prekida (eng. *single point of failure*).

Drugi način izvedbe sinkronizirane komunikacije između servisa, uz koreografiju, je registracija povratnog poziva (eng. *callback*) kod servisa. Povratnim pozivom traži se od servisa da obavijesti svoga pozivatelja po završetku obavljanja zadatka, odnosno klijenta. Na taj način, klijent ne ostaje u konstantnoj vezi sa servisom te samim time nema blokirajućeg efekta kojeg donosi takva veza. Preporučuje se kombiniranje tih dviju tehnika, gdje se klasična tehnika *zahtjeva/odgovora* koristi u većini slučaja, dok se tehnika s povratnim pozivom koristi kod zadataka dužega trajanja kako bi se dobio najbolji rezultat obiju tehnika.

Asinkronom se komunikacijom mogu povezati servisi tako da se izbjegne blokirajuća veza između klijenta i servisa. Poziv kod asinkrone komunikacije izgleda nešto drukčije nego kod sinkrone. Kod asinkrone komunikacije klijent šalje zahtjev servisu koji sadrži naziv događaja koji se dogodio zajedno s pripadajućem setom podataka, a koji dodatno opisuju taj događaj. Takav način komunikacije naziva se komunikacija bazirana na događajima (eng. *event-based*).

Prednost takve komunikacije je što se odvija vrlo brzo te je vrijeme inicijalnoga odgovora klijentu jednako za sve zadatke. Razlog tome je činjenica da servis ne zadržava klijenta dok obavlja zadani zadatak, nego se klijentu odmah šalje potvrda poruka o zaprimljenom događaju. U tom se slučaju otežava rad razvojnim programerima, budući da aplikacija postaje nepovezana. Klijentski dio šalje događaje poput upita o stanju zaliha, gdje u jednoj od mogućih implementacija servis odgovara URL adresom. Klijent tada metodom ispitivanja (eng. *polling*) ispituje dobivenu URL adresu dok na njoj ne nađe traženi rezultat ili grešku servisa u izvršavanju zadatka.

Tehnika se upravljanja procesa između nekoliko servisa (kada koristi asinkronu komunikaciju) naziva koreografijom. U tom slučaju nema centralnog mjesta za orkestraciju, nego se servisi pretplaćuju (eng. *subscribe*) na pojedine događaje te ih osluškuju. Kada klijent objavi događaj, servis dobije obavijest te odrađuje svoj dio zadatka. Broj pretplaćenih servisa može biti proizvoljan, što uvelike pridonosi slaboj povezanosti cijelog sustava.

Problem koji povlači integracija servisa, temeljena na asinkronoj komunikaciji i događajima, je taj da je sami razvoj i nalaženje grešaka puno teže. No, zbog toga se dobije na nepovezanosti između servisa, njihovoj jednostavnoj promjeni i većoj fleksibilnosti. Jedan je od načina olakšavanja rada kod takve integracije dobro definirani sustav nadgledanja (eng. *monitoring*) i izvještavanja (Newman, Building Microservices: Designing Fine-Grained Systems, 2015).

2.4.3. REST

REST (eng. *Representational State Transfer – REST*) je arhitekturni stil baziran na nizu principa koji opisuju kako su mrežni resursi definirani i adresirani, s naglaskom na laku razmjenu informacija i skalabilnost. Resursi su izvori specifičnih informacija koji se spominju individualno s univerzalnim identifikatorom resursa poput URL-a, npr. svaki resurs ima svoj URL. Pod REST najčešće spadaju jednostavna sučelja za komunikaciju podataka specifične domene preko HTTP protokola, bez dodatnih slojeva poruka ili praćenja sjednice. Smatra se da je najveći primjer REST aplikacije sam Web, gdje se koristi HTTP protokol za transport običnih tekstualnih dokumenata koji su adresirani pomoću URL-a. REST-om se mogu prenositi razni tipovi dokumenata. Najčešći su HTML, JSON, XML, XHTML, RSS i dr. (Kay, 2018).

Pri implementaciji, REST sadrži nekoliko ograničenja koji osiguravaju da se svaka implementacija REST-a ponaša slično, odnosno, razvojni inženjeri koji koriste REST znaju što očekivati od njega. Ograničenja kojima se opisuje REST su:

- Resurs mora biti pohranjen samo na poslužitelju, a ne klijentu, no klijent ga može dohvatiti i promijeniti pomoću poslanih zahtjeva na poslužitelj
- Poslužitelj ne smije spremati status svojih klijenta jer bi to onemogućilo skalabilne pogodnosti REST-a. Isto tako, kada se govori o velikom broju klijenata i većim brojem poslužitelja onda bih to predstavljalo dodatne probleme kod spremanja tih statusa i praćenja sesija u bazi podataka.
- Svaki zahtjev klijenta mora sadržavati sve informacije koje su potrebne za izvršavanje tog zahtjeva, odnosno, informacije se ne koriste ponovno između zahtjeva
- Informacije o sjednici klijent sprema na svojoj, klijentskoj strani

- Moguće je postojanje više reprezentacija istoga resursa u različitim formatima, npr. JSON, XML, i dr.
- Kada poslužitelj vraća odgovor klijentu, taj odgovor mora sadržavati informaciju je li u odgovoru resurs koji se može spremiti u priručnu memoriju (eng. *cache*)
- Opcionalno: poslužitelj može klijentu slati isječke koda u odgovorima, npr. Javascript-a, kojima nadopunjava funkcionalnosti klijenta (Versrynge, 2018)

Razlog zašto se upravo HTTP protokol koristi za transport „krije“ se u jednostavnosti korištenja tog protokola. HTTP je jedan od osnovnih protokola Web-a te u svojoj specifikaciji koristi glagole (poput POST, GET, PUT, DELETE i dr.) i statusne kodove (200 OK, 404 Not Found i dr.) koji su dobro definirani i time olakšavaju implementaciju REST-a. Prednost korištenja HTTP-a kod REST-a je i ta što s HTTP-om dolazi veliki broj podržanih tehnologija i alata.

Tablica 2. HTTP REST metode

Metoda	**Doseg**	**Sigurna**	**Idempotentna**	**Opis metode**
GET	kolekcija	da	da	Dohvaća sve resurse u kolekciji
GET	resurs	ne	ne	Dohvaća jedan resurs
HEAD	kolekcija	da	da	Dohvaća zaglavlja resursa u kolekciji
HEAD	resurs	da	da	Dohvaća zaglavlje jednog resursa
POST	kolekcija	ne	da	Kreira novi resurs u kolekciji
PUT	resurs	ne	da	Ažurira resurs
PATCH	resurs	ne	da	Ažurira resurs
DELETE	resurs	ne	da	Briše resurs
OPTIONS	sve	da	da	Vraća dostupne metode i opcije

Izvor: (Jansen, 2018)

Tablica 2 prikazuje HTTP metode koje se koriste kod REST-a i način na koji se koriste. Stupac „Sigurna“ označava je li metoda sigurna za resurs, odnosno, hoće li njezinim izvršavanjem resurs ostati nepromijenjen. Stupac „Idempotentna“ označava metode koje se mogu pozivati proizvoljan broj puta i to tako da rezultat bude isti svakoga puta. Važno je napomenuti da svaka implementacija REST-a preko HTTP-a ne mora sadržavati implementaciju svih navedenih metoda, što je i slučaj u praksi kada se implementiraju samo potrebne metode.

Kod odabira tipa dokumenata koji će služiti za reprezentaciju resursa najčešći su izbori JSON i XML. U usporedbi je s XML-om JSON lakši, jednostavniji te zahtjeva manje dodatnih informacija o samom resursu. Usporedbom veličine JSON i XML dokumenta identičnog resursa s identičnim informacijama, JSON pobjeđuje s manjom veličinom, dok XML bolje kontrolira i definira tipove podatka svojeg resursa svojim dodatnim opisnim informacijama podataka koje predstavlja. XML-om se mogu definirati i XML sheme koje predstavljaju strukturu podatka pa će tako klijent tako znati koje će strukture svaki put dobiti podatak, dok kod JSON-a to nije tako strogo definirano. Osim ta dva tipa dokumenata, HTTP protokolom može se slati binarni zapis ili HTML-om, koji onda predstavlja i korisničko sučelje i podatke.

2.4.4. RPC

RPC (eng. *Remote Procedure Call - RPC*) je protokol koji omogućava pozive udaljenih procedura programa koji se ne nalaze u istom radnom okruženju kao i pozivatelj metode. U praksi, RPC maskira pozive udaljenih metoda te se one koriste kao da su lokalne. Udaljeni program može biti na drugom računalu ili dostupan preko mreže, bez da pozivatelj ima informaciju o tome gdje se taj program nalazi i bez odgovornosti o prijenosu podataka kroz mrežu.

RPC protokol koristi jednostavnu klijent-poslužitelj arhitekturu, gdje je pozivatelj udaljene procedure klijent, a udaljeni program koji pruža uslugu procedure poslužitelj. RPC pozivi vrlo dobro maskiraju udaljene pozive jer su pozivi sinkroni, odnosno, izvršavanje poziva klijent je u stanju blokade dok se ne dobije odgovor od poslužitelja, isto kao i kod lokalnih poziva. Kako bi se izbjeglo blokiranje klijenta, moguće je pokrenuti istovremeno nekoliko RPC poziva kroz procese ili dretve (Rouse, What is Remote Procedure Call (RPC)? - Definition from WhatIs.com, 2018).

Postoji nekoliko različitih RPC tehnologija, a svima im je zajednička osnovna karakteristika ta da čine udaljeni poziv jednak lokalnom pozivu. Korištenjem tehnologija koje se oslanjaju na definiranje sučelja, poput SOAP, Thrift i protocol buffers, moguće je lakše generiranje panjeva (eng. *stubs*) i klijenata za druge, različite tehnologije. Korištenjem, npr. Java RMI-a, klijent se i poslužitelj ograničavaju na istu tehnologiju te se tako gubi slabo povezivanje.

Od navedenih tehnologija, Thrift, protocol buffers i Java RMI koriste binarni zapis za razmjenu informacija, dok SOAP koristi XML putem HTTP protokola. Kod tehnologija se koje koriste binarni zapis mogu koristi različiti tipovi mrežnih protokola, poput TCP-a i UDP-a, od kojih svaki ima svoje prednosti, ali i mane.

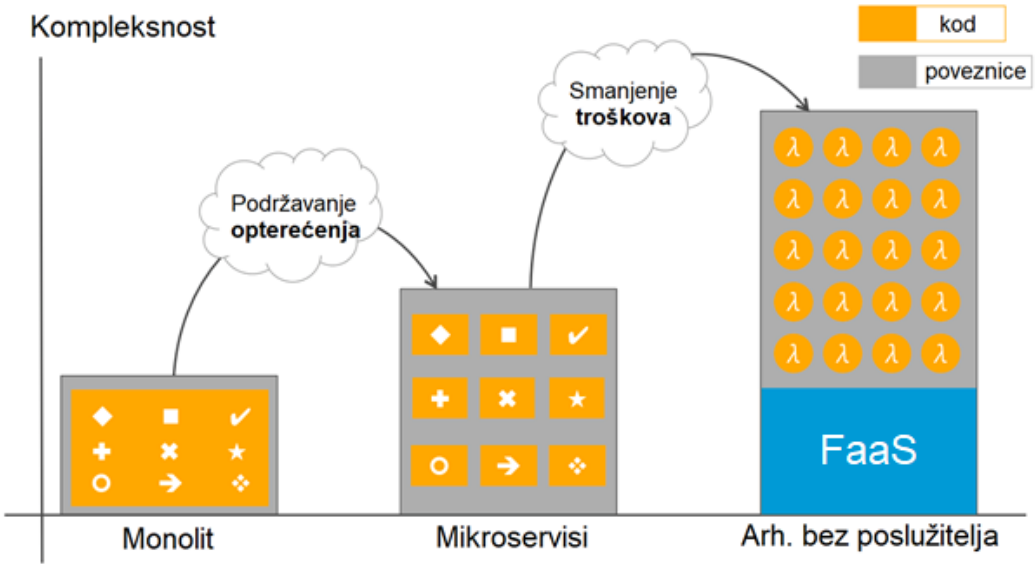
RPC donosi prednosti, naročito u binarnom obliku. Neke od njih su: male količine dodatnih informacija u pozivima, velike brzine i jednostavnost razvoja. S druge strane, glavna značajka RPC protokola, maskiranje udaljenih poziva, može biti dvosjekli mač. Lako je upasti u zamku i napraviti puno udaljenih poziva procedura jer su jednostavne poput lokalnih, no ovise o cijelom setu čimbenika vezanih uz računalnu mrežu. Serijalizacija i de-serijalizacija podataka, kako bi bili spremni za slanje mrežom, troši resurse procesora, pogotovo kada se koristi SOAP i XML. Drugi je problem i slanje mrežom. Maskiranjem mrežne komponente poziva programer lako previdi da se tijekom slanja poziva može dogoditi prekid veze.

Prilikom razvoja važno je uzeti u obzir da računalne mreže, uz sve današnje tehnologije, nisu potpuno stabilne i može se dogoditi prekid veze u bilo kojem trenutku, čak i ako poslužitelj i klijent normalno funkcioniranju. Osim što se problemi u mreži mogu trenutno očitovati i to prekidom veze, mogu se očitovati i postepenim zagušenjem te malformacijom mrežnih paketa. Servisi moraju biti spremni na sve moguće zastoje u mrežnoj komunikacije kako bi se pravovremeno oporavili od takvih zastoja i o njima obavijestili sustav nadgledanja (Green, 2015).

2.5. Arhitektura bez servera

Arhitektura bez servera (eng. *serverless architecture*) nastaje kao slijed razvoja arhitektura iz mikroservisne arhitekture. Prema nazivu se da zaključiti da se serveri za pogon takve aplikacije uopće ne koriste, no u stvarnosti se oni koriste, samo ne na uobičajen način. Kod monolitnih aplikacija, mikroservisne i drugih arhitektura, postoji server na kojem se pokreću cijele aplikacije odjednom, neki njihovi moduli ili skup servisa, dok kod arhitekture bez servera ne postoji server u istom smislu. Arhitektura bez servera najlakše se može opisati već poznatim servisima mikroservisne arhitekture pretvorenih u funkcije, koje se onda izvršavaju na sustavima za izvršavanje funkcija.

Arhitekturom se bez servera postiže veća granulacija u arhitekturi, što donosi svoje prednosti i mane. Skalabilnost je jedna od najvećih prednosti koja se postiže tom arhitekturom, no ta skalabilnost nije beskonačna, već je ograničena maksimalnim brojem zahtjeva jedne funkcije i njezinim trajanjem. Glavna mana takve arhitekture je dodana kompleksnost održavanja i razvijanja takve aplikacije. Ako se kreira preveliki broj funkcija, dobit će se aplikacija kojom se teško upravlja. S druge strane, ako je prisutan premali broj funkcija, postoji opasnost od kreiranja malih monolitnih aplikacija.



Slika 5. Odnos kompleksnosti i arhitektura (Zimine, 2018)

U usporedbi s ostalim arhitekturama, arhitektura bez servera povećava mogućnosti nošenja s opterećenjem i smanjuje količinu programskog koda. Posljedica toga je povećana kompleksnost uspostavljanja i održavanja takvih aplikacija i to zbog velikoga broja pokretnih dijelova, kao što je to prikazano na Slika 5.

Funkcija kao usluga (eng. *Function as a Service - Faas*) predstavlja pružanje klijentima platformu u oblaku, na kojoj mogu izvršavati svoje funkcije. Arhitektura bez servera direktno ovisi o pružateljima FaaS i to na način da se mogu koristit samo tehnologije koje odabrani pružatelj podržava. Osim toga, kako je cijela aplikacija u rukama vlasnika platforme pitanje su stabilnost i dostupnost pružatelja usluge i aplikacije, jer platforma nije u fizičkom dosegu i na nju se nema utjecaja.

Kod arhitekture bez servera, integracijsko testiranje postaje problematično i teško. Objekti su takvog testiranja kod ove arhitekture puno manji nego kod ostalih, tako da je oslanjanje na integracijsko testiranje veće. Uz to, problem postaje automatiziranje procesa razvoja jer su zbog nerasprostranjenosti arhitekture alati za automatsko raspoređivanje, pokretanje i testiranje relativno nerazvijeni te podrška za automatiziranje nije na razini one kao kod mikroservisa (Maruti, 2018).

2.6. Tehnologije za razvoj mikroservisa

Izbor tehnologija za razvoj mikroservisa je, slično kao i u slučaju tehnologija za razvoj web aplikacija, vrlo velik. Većina se današnjih programskih jezika (ne samo oni koji su specifični za web) može koristiti za razvoj servisa. U nastavku će biti opisan skup tehnologija, koji je prema autorovoj procjeni vrlo dobar upravo za potrebe mikroservisa, te će isti biti korišten i kod izrade praktičnog djela ovog rada.

2.6.1. Go programski jezik

Go, poznat i pod nazivom Golang, programski je jezik prvi puta predstavljen 2009. godine od strane Google-ovih inženjera Roberta Griesemer-a, Roba Pike-a i Ken-a Thompsona. Nastao je kao eksperiment kojem je cilj bio kreirati programski jezik koji će riješiti probleme i kritike razvojnih inženjera na druge programske jezike koje koriste (npr. C++), ali s ciljem da zadrži ostale pozitivne karakteristike tih jezika. Neke od glavnih karakteristika Go programskog jezika su:

- Statički tipiziran i skalabilan na velike sustave
- Produktivan i čitljiv bez previše koda predloška potrebnog za svaki program
- Bez potrebe za integriranim razvojnim okruženjima (eng. *IDE*), ali s njihovom dobrom podrškom
- Podržavanje mrežnog programiranje i izvršavanje na višejezgrenim sustavima (Frequently Asked Questions (FAQ) - The Go Programming Language, 2018)

Go programski jezik izabran je zbog nekoliko razloga za implementaciju servisa praktičnog djela ovoga rada. Prvi i najvažniji razlog su performanse koje postižu programi napisani u Go-u. Go programi imaju vrlo mali utisak u memoriji i korištenju procesorskog vremena te se isto tako vrlo lako optimiziraju za konkurentni rad procesa. Kompilirani su Go programi samostalni te ne zahtijevaju dodatne ovisnosti o drugim programima pri pokretanju, pod uvjetom da su pravilno kompilirani (za platformu na kojoj će se izvršavati). Ta karakteristika omogućuje da su u konačnici Go izvršne datoteke male u veličini i da se zbog svoje neovisnosti lako isporučuju na različite platforme. Jedan od razloga nastanka Go-a bio je olakšavanje razvoja razvojnim inženjerima, što je prema autorovom mišljenju uspješno ostvareno. Pisanje je programa u Go-u inženjerima s ikakvom programskom pozadinom vrlo lako i pruža dobru ravnotežu između jednostavnosti korištenja i funkcionalnosti.

2.6.2. MongoDB

MongoDB je, prema definiciji, baza podataka koja se bazira na otvorenom kodu, podacima orijentiranim dokumentima i nestrukturiranim upitnim jezicima. Baze podataka s nestrukturiranim upitnim jezikom nazivaju se NoSQL baze podataka, a MongoDB je jedna od najpoznatijih takvih baza podataka (What is MongoDB, 2018). NoSQL su baze podataka cijelo jedno područje samo za sebe, isto kao i relacijske baze podataka, no neke od važnih razlika između NoSQL i SQL baza podataka su sljedeće:

- SQL baze se podataka baziraju na relacijama između entiteta, dok se NoSQL baze podataka baziraju na dokumentima, ključ-vrijednost parovima, grafovima ili zapisima širokih stupaca (engl. *wide column stores)*
- SQL baze imaju predefinirane sheme prema kojima se upisuju podaci, dok NoSQL nema definirane sheme, nego se ona dinamički izmjenjuje prema unesenom dokumentu
- SQL se baze podataka daju vertikalno skalirati što znači da je u slučaju preopterećenja potrebno nadograditi poslužitelj novim, jačim hardverom. S druge strane, NoSQL se baze daju horizontalno skalirati, što omogućuju jeftinije i jednostavnije skaliranje. Kod NoSQL je dovoljno uspostaviti novu instancu iste te baze, zajedno s raspoređivačem opterećenja koji će onda dolazeće zahtjeve dijeliti između dvije instance i time poboljšati performanse baze.
- SQL su baze bolji izbor ako želimo pisati kompleksne upite na bazu te dohvaćati limitirani set informacija te su moćniji alat za pisanje upita. NoSQL se fokusira na dohvaćanje kolekcija dokumenata koja odgovara zadanim uvjetima. NoSQL se upiti razlikuju od jednog do drugog proizvođača baza podataka, dok su SQL upiti pretežito slični između različitih proizvođača.
- Primjeri su SQL baza podataka: MySql, Oracle, Sqlite, Postgres i MS-SQL. Primjeri NoSQL baza podataka: MongoDB, Redis, Cassandra, CouchDB i RavenDB (Issac, 2018).

Kao što je navedeno, u razlikama između SQL i NoSQL baza podataka, NoSQL ima veliku prednost u skaliranju naspram relacijskih baza podataka. Upravo je to razlog zašto je jedan od najčešćih odabira kod razvijanja skalabilnih servisa. Osim skalabilnosti, način rada NoSQL baze pridonosi i pravilnom odvajanju konteksta između pojedinih servisa te se pravilnim razvojem postiže dobra unutarnja kohezija i slaba vanjska povezanost servisa.

Kada govorimo o odnosu baze podataka i servisa, možemo ih rasporediti na nekoliko načina. Način u kojem je baza podataka jedna te se svi servisi spajaju na nju, opisan je prethodno u poglavlju „Integracija servisa“. Sljedeći je način da više servisa dijeli jednu bazu

podataka, ali tada ti servisi ne komuniciraju međusobno, nego razmjenjuju podatke putem baze.

Ako svaki servis ima svoju bazu podataka, to znači puno veći mrežni promet, budući da se sve informacije moraju posebno zahtijevati i slati mrežom. No, s druge strane, takvo odvajanje daje puno veću autonomiju kod skaliranje servisa i njegove baze.

2.6.3. Docker

Docker je alat koji olakšava uporabu Linux kontejnera. Linux su kontejneri oblik virtualizacije koja daje mogućnost da jedno fizičko računalo možemo „raspodijeliti“ u više virtualnih računala, od kojih svako virtualno računalo ima svoje resurse fizičkog računala. Srodni načini virtualizacije su hipervizori i paravirtualizacija. Prednost kontejnera je u tome što mogu sadržavati minimalni podskup potrebnih datoteka i programa za izvršavanje aplikacije te su zbog toga iznimno prenosivi i neovisno o operativnom sustavu, ukoliko je taj sustav podržan od strane Docker-a. Za razvojnoga inženjera takve karakteristike znače da se ne mora pridavati pažnja razlikama između produkcijskog okruženja i razvojnog jer će ono, ako se koriste kontejneri u oba slučaja, biti isto, što uvelike olakšava posao razvojnim inženjerima (What is Docker?, 2018).

Docker se bazira na slikama, prema kojima se kasnije kreiraju kontejneri. Od pojedine je slike moguće kreirati proizvoljan broj kontejnera. Često se pokreće nekoliko kontejnera odjednom, svaki sa svojim parametrima za pokretanje, te je tada pogodnije koristiti konfiguracijsku datoteku za pokretanja. U Lunch Plan aplikaciji Docker se kontejneri pokreću pomoću *docker-compose.yml* datoteke, premda je moguće više načina konfiguracijskih datoteka (Vrčić, 2018).

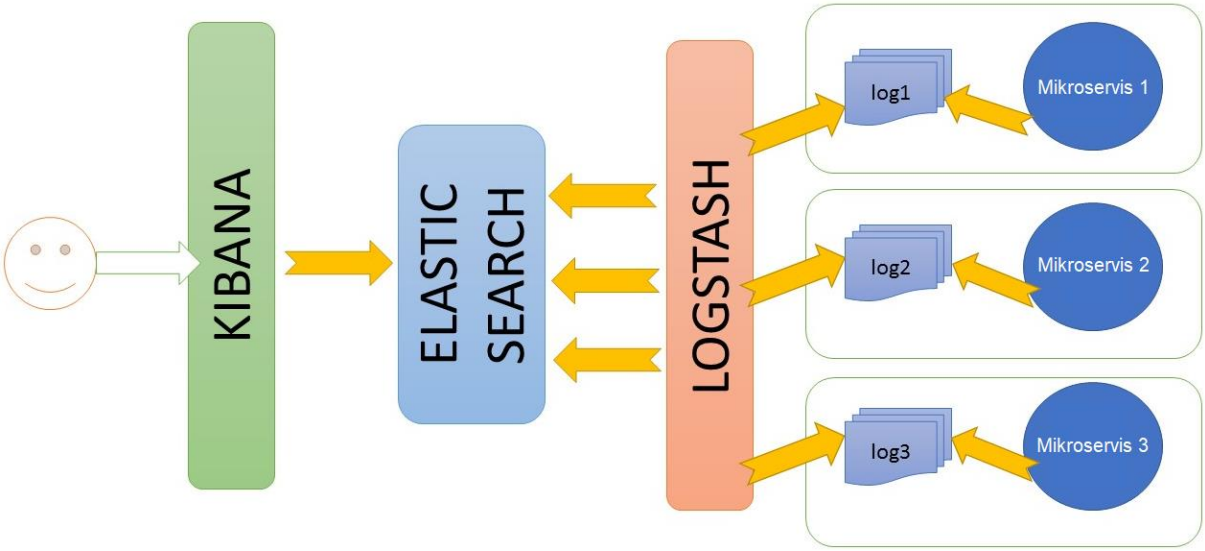
2.6.4. ELK stack

ELK stack je engleski naziv za skup tehnologija otvorenog koda, a koje se koriste u sinergiji kako bi se postigli najbolji rezultati u polju praćenja rada razvijenih aplikacija. ELK je akronim od naziva sljedećih alata: Elasticsearch, Logstash i Kibana. Sva tri alata dolaze od istog proizvođača softvera, otvorenog koda Elastic, a napravljeni su da funkcioniraju neovisno jedno o drugome. Iako, njihova je iskoristivost još veća kada se koriste u kombinaciji.

Elasticsearch alat služi kao baza podataka log zapisa. Visoke je skalabilnosti, otvorenog koda te omogućava pretraživanje punog teksta i analize podataka. Može se koristiti za klasično pretraživanje punog teksta, pohranu analitike, sustav upozoravanja, provjera pravopisa i sl. (Brasetvik, 2018).

Logstash u ELK stack-u služi za sakupljanje i normalizaciju podataka iz različitih izvora u realnom vremenu. Inicijalno je namijenjen za skupljanje log zapisa, no s trenutnim mogućnostima može transformirati bilo koji događaj te ga filtrirati ili obraditi s raznim algoritmima. Klasični je scenarij korištenja Logstash-a prikupljanje log zapisa od strane aplikacije, pretvaranje u JSON format te arhiviranje i pripremanje za daljnje analize pomoću drugih alata (Logstash Introduction, 2018).

Kibana je posljednji alat ELK stack-a te je njegova primarna zadaća vizualizacija i prikaz informacija korisniku koje je Logstash skupio i pospremio u Elasticsearch (Slika 6). Kibana je specijalizirana za velike setove podataka i za podatke koji se prikupljaju u realnom vremenu. Kibana se također koristi za istraživanje podataka i to kako bi se došlo do traženih informacija (What is Kibana?, 2018).



Slika 6. Interakcija korisnika s alatima ELK stack-a (Chakraborty, 2018)

2.6.5. Logspout

Logspout je alat specijaliziran za Docker kontejnere, u kojima se i on sam izvršava. Logspout se spaja na sve pokrenute kontejnere i usmjerava log zapise svih kontejnera prema postavkama. Funkcionalnost Logspout-a je poprilično jednostavna i on nije namijenjen za upravljanje log zapisima ili njihovo čitanje, već samo za njihovo usmjeravanje. Trenutno su podržani standardni izlaz – stdout i standardni izlaz za poruke o greškama – stderr (gliderlabs/logspout, 2018).

3. Uzorci dizajna

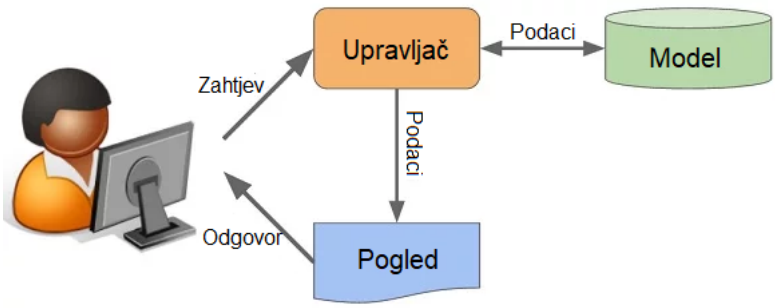
Uzorak dizajna u programskom je inženjerstvu univerzalno rješenje koje se može ponoviti za probleme koji se učestalo javljaju. Takvi uzorci nisu gotova rješenja koja se mogu samo replicirati, već se oni moraju programirati. Uzorak dizajna služi kao opis ili predložak rješenja problema koji je ponovo upotrebljiv u različitim scenarijima.

Razlog za korištenje uzoraka dizajna je taj da su uzorci testirani i predstavljaju provjerene razvojne paradigme te se time ubrzava proces razvoja. Prednost korištenja uzoraka je ta što pomažu izbjegavanju često skrivenih i suptilnih problema koji se potkradaju programerima kada razvijaju vlastita rješenja. Uzorci dizajna također poboljšavaju čitljivost programskog koda programerima i razvojnim inženjerima koji su upoznati s tim uzorcima.

Kako bi prepoznati uzorci bili iskoristivi u što više situacija, generaliziraju se i ne vežu za specifične probleme. Programeri koriste uzorke kako bi komunicirali kroz dobro poznate i dobro shvaćene nazive za interakcije unutar programske komponente. Kod refaktoriranja se programskog koda (eng. *code refactoring*) koriste uzorci dizajna kako bi završni rezultat bio razumljiviji i jednostavniji za buduće izmjene ili održavanje (Design Patterns, 2018).

3.1. MVC

Arhitektonski uzorci imaju identične osobine, kao i uzorci dizajna, i u nekima se čak i preklapaju, no važna razlika je ta da programeri pomažu posložiti strukturu programskog koda i aplikacije koje izrađuju, ali oni ne rješavaju poznate implementacijske probleme. Jedan od takvih uzoraka je Model-View-Controller, skraćeno MVC, koji razdvaja aplikaciju u tri dijela. Dijelovi su aplikacije prema MVC-u: model podataka, pogled (eng. *view*) i upravljač (eng. *controller*) (Buschmann, Meunier, Rohner, Peter, & Stal, 1996). Slika 7 pokazuje tijek zahtjeva korisnika kroz dijelove aplikacije.



Slika 7. Proces zahtjeva i odgovora u MVC-u (Hollingworth, 2018)

Model podataka je najniža razina aplikacije u kojoj se nalaze osnovni podaci i funkcionalnosti te je njegova glavna odgovornost održavanje podataka. Model je potpuno

neovisan o specifičnim prikazima podataka ili unosima korisnika. Komponente pogleda prikazuju korisniku informacije i podatke. Pogled odlučuje o samom prikazu podataka, pokazuju li se svi podaci ili samo neki i na koji način. Svaki pogled ima svojeg pripadajućeg upravljača, a svaki model može imati nekoliko pogleda. Upravljač je odgovoran za svu komunikaciju između pogleda i modela. Upravljač je komponenta koja zaprima unos korisnika u obliku događaja, npr. pritisak linka na stranici te upravljanje tim zahtjevom (Basic MVC Architecture, 2018).

3.2. Adapter

Adapter uzorak u programiranju omogućuje povezivanje dvije nekompatibilne klase ili sučelja. Uzorak se najčešće implementira klasom koja služi kao omotač oko jedne od postojećih klasa i daje drukčiji ili izmijenjeni pristup toj klasi. Adapter može biti odgovoran za funkcionalnost koju adaptirana klasa ne podržava te je na taj način proširuje. Kod mikroservisne se arhitekture uzorak adaptera koristi s istim ciljem, samo što predmet adaptiranja nisu klase i sučelja, nego drugi servisi.

Uzorak adaptera kod servisa daje odgovor na problem kako iskoristiti postojeće korisne servise pri razvoju novih ili pri transformiranju aplikacije u mikroservisnu arhitekturu. Razlog za korištenje adaptera može biti želja za ponovnim korištenjem postojećeg servisa ili programa i to kako bi se izbjeglo njihovo ponovno razvijanje. Ponekad postoji mogućnost izmjene tog postojećeg servisa, no posljedice mogućega rizika da se kreiraju problemi za klijente su prevelike. Isto tako je moguće da postojeći servis nema dovoljno dobro razvijen API za komunikaciju s drugim servisima i time odudara od ostalih novokreiranih servisa u mikroservisnoj arhitekturi. Sve navedeno su opravdani razlozi koji nas upućuju na korištenje adaptera umjesto izmjena postojećeg servisa.

U puno su slučajeva takvi postojeći servisi bazirani na SOAP tehnologiji i pružaju funkcijski orijentirana sučelja, za razliku od REST-a gdje je naglasak na sučeljima entiteta. Pretvaranje takvog funkcijski orijentiranog sučelja u sučelje entiteta često se svodi na povezivanje pristupa glagolima CRUD (od eng. *create, read, update, delete*) operacija s REST metodama GET, POST, PUT i DELETE, gdje URL predstavlja entitet.

Servisi koji imaju ulogu adaptera privremenoga su vijeka i traju dok se servis kojeg adaptiraju ne zamjeni potpunim rješenjem u obliku ponovnog razvijenog servisa u mikroservisnoj arhitekturi. Adapter servis može implementirati i mehanizam keširanja kako bi se smanjilo opterećenje postojećeg, često zastarjelog, servisa (Brown & Woolf, 2018).

3.3. API Gateway

API Gateway se uzorkom kreira prolaz (eng. *gateway*) za sve pozive prema aplikaciji. API Gateway stoji ispred servisa u mikroservisnoj arhitekturi te je on ulazna točka za sve pozive koje kreira klijent ili aplikacija.

U scenariju bez takvog prolaza za pozive klijent, kako bi izvršio radnju koja, npr. slijedno povezuje 3 servisa prvi, drugi i treći tim redoslijedom, mora znati komunicirati s njima. Kako bi klijent znao komunicirati sa servisima, mora znati njihovu adresu, npr. prviServis.app.com, drugiServis.app.com i sl. Odmah se može vidjeti kako će te adrese servisa biti potrebno negdje pohraniti u programski kod, konfiguracijsku datoteku ili neki drugi način pospremanja konfiguracije. Kako su servisi na različitim adresama, klijent mora napraviti tri različita mrežna poziva umjesto samo jednoga, ako se uvede API Gateway.

Uz broj mrežnih poziva koji raste s povećanjem broja servisa, problem je i odgovarajuće povezivanje klijenta sa servisima. Kada su servisi sitnog zrna, odnosno pokrivaju samo mali opseg funkcionalnosti, teško je upariti potrebe klijenta s odgovarajućim servisom. Ukoliko implementirani servisi koriste različite tehnologije, poput RPC-a ili MQTT protokola, tada dolazi do toga da klijent mora imati implementirani način komunikacije i za njih, što podiže razinu kompleksnosti koda samog klijenta.

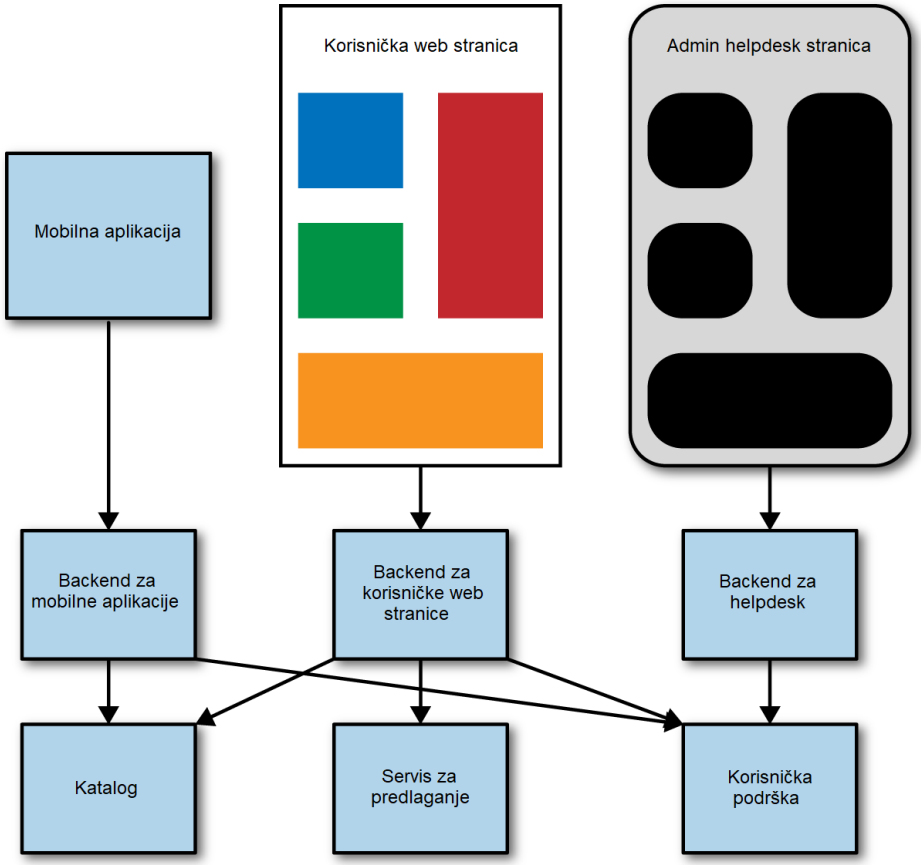
Problem s više mrežnih poziva za izvršavanje jedne radnje zaobilazi se uvođenjem API Gateway-a. API Gateway implementira se kao poslužitelj koji prima sve zahtjeve korisnika te na temelju unutarnje logike upravlja daljnjim pozivima prema drugim servisima. Poslužitelj tako inkapsulira unutarnje arhitekturu sustava i pruža API klijentu. Moguće je da poslužitelj bude odgovoran i za funkcionalnosti, poput autentifikacije, nadgledanja, raspoređivanja opterećenja, keširanje, upravljanje statičkim odgovorima te izmjena i upravljanje zahtjevima (Richardson, Building Microservices Using an API Gateway | NGINX, 2018).

3.4. Backends for Frontends

Uzorak Backends for Frontends, skraćeno BFF, vrlo je sličan API Gateway uzorku te se može smatrati njegovom izvedenicom. Za razliku od API Gateway uzorka, kod BFF-a svaka grupa klijenata imaj svoj poslužitelj. Pod grupom se klijenata smatraju klijenti koji koriste identične ili vrlo slične funkcionalnosti aplikacije na sličan način. Za primjer se takve grupe mogu uzeti korisnici Android i iOS mobilnih aplikacija. Iako ti sustavi podržavaju veliki broj različitih uređaja poput mobitela, tableta i sl., aplikacije im izgledaju poprilično slično unutar jednog operativnog sustava. To omogućuje razvojnom timu fokus na jednu grupu klijenata te za tu grupu kreiraju posebni poslužitelj.

Takav posebni poslužitelj ulazna je točka za komunikaciju mobilne aplikacije s ostatkom sustava, a kako je specijaliziran za jednu grupu korisnika, može se njoj prilagoditi posebnim funkcionalnostima ili optimizacijama performansi. Prednost BFF-a nije samo to što se odvaja programski kod koji nije odgovoran za istu grupu korisnika, nego i to što se prema posebnim poslužiteljima mogu formirati i timovi.

Tim koji je zadužen za razvoj Android mobilne aplikacije preuzima i razvoj poslužitelja koji pripada grupi klijenata Android aplikacije. Na taj način Android tim postaje u potpunosti odgovoran za svoj poslužitelj/servis i komunikacijski kanal između Android mobilne aplikacije i web aplikacije (Newman, Backends For Frontends, 2018).



Slika 8. Grupe korisnika i njihovi pripadajući servisi (Newman, Building Microservices: Designing Fine-Grained Systems, 2015)

Slika 8 pokazuje BFF uzorak na primjeru web aplikacije koja u sebi sadrži katalog proizvoda, a ima odvojene grupe korisnika: mobilna aplikacija, korisnička web stranica i admin helpdesk stranica. Svaka će grupa korisnika imati zasebni servis, izložen za komunikaciju s tom grupom u kojem se nalazi funkcionalnost i implementacija specifična za tu grupu korisnika.

3.5. Bulkhead

Bulkhead je uzorak dobio svoj naziv prema praksi da se pregrađuje transportni prostor u trupu broda. Ako je nastala šteta na trupu broda, izgubit će se tereti na sekciji te pregrade pa će se samo oštećena sekcija napuniti vodom te brod neće potonuti. Takve pregrade se primjenjuju i u programskom inženjerstvu u obliku Bulkhead uzorka kako bi se odvojili resursi. Poznato je da su resursi limitirani te se njihovim odvajanjem na navedeni način osigurava njihova neiscrpnost.

Za primjer se može uzeti bazen konekcija (eng. *connection pool*) baze podataka kod kojega klijenti vrše dvije različite operacije, čitanje i pisanje. Razlike između konekcije za čitanje i pisanje nema. Bez Bulkhead uzorka, postojao bi jedan veliki bazen konekcija u kojem bi se sve one nalazile te prema potrebi dodjeljivale klijentima. Ukoliko se dogodi da veliki broj klijenata koristi aplikaciju, njihove potrebe za konekcijama za pisanje premašuju broj konekcija te će aplikacija vrlo vjerojatno postati nedostupna ili se čak srušiti. U tom trenu, svi će klijenti biti pogođeni ispadom.

Bulkhead predlaže sljedeću alternativu: odvojiti uzorak konekcije za čitanje i pisanje u zasebne konekcijske bazene. Na taj način, kod opisanog scenarija, iscrpile bi se samo konekcije klijenata za pisanje u bazi podataka, dok bi dio aplikacije koji se bazira samo na dohvaćanju podataka funkcionirao normalno. Odvajanje ne mora nužno biti prema tipu operacije. Bazeni se mogu kreirati i prema servisima ili prema skupini povezanih servisa (Márton, 2018).

3.6. Strangler

Uzorak se Strangler ponaša prema svom nazivu, aplikaciju koja je zastarjela on „davi“, dok se sustav nije spreman u potpunosti prebaciti na novu verziju aplikacije. Strangler u mikroservisnoj arhitekturi pomaže u prelasku s monolitne aplikacije na mikroservisnu arhitekturu, ali se može koristiti i između verzija servisa.

Kako je kompletna zamjena cijelog kompleksnog sustava drugim sustavom veoma rizična i zahtjevna, preporučuje se odraditi zamjenu kroz nekoliko faza. Često se kod takvih zamjena mora sačuvati stara verzija aplikacije i to kako bi se pokrile funkcionalnosti koje možda nisu implementirane u novoj verziji, ili, kako bi se korisnicima dao prijelazni period za ažuriranje svoga klijenta. Održavanje dvije verzije kompletne aplikacije odjednom donosi dodatne komplikacije i potrebno je kod svake migracije servisa ažurirati klijente o njihovim lokacijama.

Rješenje se predstavlja u obliku uzorka Strangler koji dopušta inkrementalno izmjenjivanje i otpuštanje servisa stare verzije aplikacije i servisa. Strangler se implementira pomoću posredničkog poslužitelja (eng. *proxy*), najčešće u tehnologiji koja se već koristi na razini web poslužitelja (Apache. Ngnix i sl.).

Kako implementacijom Strangler-a aplikacija ima posrednika kroz kojega prolaze i dolaze svi mrežni pozivi, više nije potrebno ažurirati klijente o izmjenama lokacija servisa. Posrednik, u ovom slučaju servis koji implementira uzorak Strangler, omogućava dodjeljivanje statične adrese servisa te definiranjem pravila kojima se određuje gdje se točno nalazi servis kojeg klijent pokušava pronaći. Na taj način postoji samo jedno mjesto izravno odgovorno za ažuriranje lokacija servisa te je olakšano migriranje između servisa.

S vremenom, kako se sve više starijih verzija servisa prespoji na nove verzije, posrednički poslužitelj „davi“ staru verziju aplikacije. U konačnici, posrednički poslužitelj koristi samo servise nove verzije te je tako migracija sa stare verzije aplikacije u potpunosti završena. Po završetku se migracije posrednički poslužitelj može ugasiti ili ukloniti (Narumoto & Wasson, 2018).

4. Praktični dio rada

4.1. Opis

Praktični dio ovog rada ima za cilj pokazati primjenu opisane teorije u praksi, odnosno, povezati objašnjenje koncepata razvoja koji se javljaju kod web aplikacije, te ih ukomponirati u mikroservisnu arhitekturu u primjeni.

Jedan od najboljih načina za shvaćanje novih koncepata je pomoću primjera. Za primjer kojega će obrađivati praktični dio rada osmišljena je vrlo jednostavna web aplikacija naziva „Lunch Plan“. Aplikacija služi planiranju ručka ili nekog drugog obroka na tjednoj ili mjesečnoj bazi. Iako postoje slične aplikacije dostupne na Internetu ili kao mobilne aplikacije, nije pronađena ni jedna koja sadrži traženi set mogućnosti.

Inspiracija za aplikaciju dolazi od vrlo jednostavnog problema s kojim se susreće svaka osoba zadužena za kuhanje u kućanstvu pa i autorova majka. Odlučivanje o tome što će se kuhati za ručak ili večeru narednih dana često iziskuje previše energije. Svakodnevnim se kuhanjem vrlo lako dođe do nedostatka inspiracije pri izboru jela za sljedeći obrok. Najčešće problem nije financijskoga aspekta (ako uzmemo u obzir jela koja se inače kuhaju u domaćinstvu), već je više psihološkoga aspekta. Kako je svako od uobičajenih jela u domaćinstvu ponavljano nekoliko desetaka ili stotina puta kroz godine, dobiva se osjećaj da je svako od tih jela nedavno bilo skuhano.

Jedan od načina na koji se može takva nedoumica olakšati jest prepustiti tu odluku drugoj osobi, algoritmu ili stroju, u ovom slučaju, web aplikaciji. Upravo je to cilj Lunch Plan-a – omogućiti osobi zaduženoj za kuhanje vođenje popisa mogućih jela te kalendarski prikaz kada je koje jelo kuhano. Osim kalendarskog prikaza Lunch Plan-a, na kojem korisnik može sam dodavati i brisati jela za pojedine dane, aplikacija nudi i opciju generiranje mogućega jela. Generiranje jela, odnosno, ideje za kuhanje, olakšava korisniku psihološki aspekt samoga odabira koje će se jelo kuhati. Proces generiranja obroka može biti potpuno nasumičan, no može biti i ovisan o informacijama, kao što su: kada je to jelo zadnji puta bilo skuhano, koja je cijena namirnica za to jelo, koje oznake (eng. *tag*) sadrži i sl.

Praktični dio rada, tj. web aplikacija, sadržavat će implementaciju Lunch Plan-a na temelju mikroservisne arhitekture. Aplikacija će biti usitnjena na nekoliko servisa koji su ograđeni svojom domenom. Isto tako, cilj nije napraviti što više servisa jer se time donose problemi u njihovoj integraciji i održavanju. Svi će se servisi pokretati u programskim kontejnerima kako bi se mogla postići maksimalna neovisnost implementacije o platformi te sama skalabilnost aplikacije, koja je jedna od ključnih karakteristika mikroservisne arhitekture.

4.2. Izvedba praktičnog djela

4.2.1. Autentifikacija

Lunch plan aplikacija namijenjena je da je koristi veliki broj različitih korisnika, što je u konačnici utjecalo i na arhitekturne odluke. Kako bi se takva funkcionalnost mogla podržati, bilo je potrebno omogućiti registraciju, prijavu i odjavu korisnika koji će je koristiti. Kako je sigurnost sveprisutna tema u programskom inženjerstvu, cilj je autora odabrati i implementirati najbolje prakse za autentifikaciju korisnika s obzirom na različite oblike korištenja aplikacije preko API-a i kroz web sučelje.

Gledajući iz perspektive API-a, preporučeni način autentifikacije korisnika za koji možemo reći da je siguran jesu pristupni tokeni. Osim pristupnih tokena, među preporučenim su se načinima autentifikacije našli i API ključevi te JWT tokeni.

Autentifikacija pristupnim tokenima, zvanim OAuth, otvoreni je standard za autentifikaciju i autorizaciju korisnika koji može koristiti bilo tko. OAuth nudi klijentskim aplikacijama da se u njih prijavi korisnik svojim korisničkim računom neke treće aplikacije, poput Facebook-a, Google Mail-a, Microsoft-a ili sl. OAuth ima više mogućih načina za implementacija, a za potrebe Lunch Plana izvedena je jednostavnija verzija koja ne uključuje aplikacije treće strane.

Kako bi korisnik došao do autentifikacijskog tokena, on prvo šalje svoje korisničke ime i lozinku POST metodom na adresu /users/login. Ukoliko su korisnički podaci ispravni, odnosno, korisnik već postoji u bazi, vraća mu generirani token, koji je ujedno i spremljen u bazu podataka.

S tokenom kojeg korisnik dobije kao odgovor, on može dalje pristupati drukčije zaštićenim dijelovima API-a aplikacije. Važno je napomenuti da je token potrebno priložiti svakom HTTP zahtjevu upućenom prema API-u aplikacije u „Authorization“ zaglavlju u formatu:

Authorization: Bearer token_korisnika

Nekoliko je mogućnosti i za autentifikaciju i pohranjivanje tokena kod korištenja web sučelja. Klasičnim se pristupom koriste kolačići (eng. *cookies*) koji se šalju zajedno sa svakim HTTP zahtjevom aplikaciji. Kolačići su i odabrani način pohranjivanja informacija o sesiji u Lunch Plan aplikaciji. Jedna od mogućih alternativa kolačićima za pohranu su lokalno spremište podataka preglednika i korištenje URL upita (eng. *query string*).

4.2.2. Sesija

Generirani autentifikacijski token korisnika sprema se u MongoDB bazu podataka. Trajanje se sesije obnavlja kod svakog poziva između korisnika i aplikacije u kojem je sadržan token. Korištena baza podataka MongoDB ima u sebi ugrađenu mogućnost postavljanja vremena u kojem u njoj ističe određena informacija te se ona automatski briše (eng. *expiry timeout*). Ta je mogućnost korištena kod implementacije autentifikacijskog tokena te je postavljeno da token ističe u vremenu od 1h. Nakon isteka tokena on postaje nevažeći te se korisnik treba ponovo prijaviti.

4.2.3. Logging

Zapisivanje (eng. *logging*) u aplikacijama pomaže otkriti što se dogodilo, kada i kako, ukoliko dođe do greške u izvođenju. Ne mora se isključivo zapisivati onda kada se dogodi greška, nego se mogu koristiti i razne kontrolne točke ili aktivnosti kako bi imali zapisani trag događaja. Log se zapisi najčešće pregledavaju ukoliko neka od funkcionalnosti ne radi kako je zamišljeno te se pomoću njih pokušava otkriti u čemu bi mogao biti problem. Isto tako mogu služiti i za brzu kontrolu pravilnoga rada aplikacije ili sustava pregledavanjem log zapisa.

Da bi se log zapisi mogli koristiti za pregledavanja statusa izvršavanja aplikacije ili traženja uzroka greške, važno ih je pravilno koristiti. Potrebno ih je, kao prvo, generirati unutar koda aplikacije, a isto je tako vrlo važno da se zapisi generiraju na odgovarajućim mjestima unutar samog koda, kako bi se iz njih moglo iščitati što više informacija.

Problem koji se javlja kod kontejnera je to što svaki servis u svojem kontejneru generira log zapise. To čini nepraktičnim pregledavanje svih zapisa jer se mora prolaziti kroz sve log zapise ručno. Rješenje tog problema nalazi se u opisanom ELK stacku i aplikaciji logsput, gdje se svi log zapisi filtriraju Logstash-om, usmjeravaju na jednu centralnu bazu podataka Elastic Search te pregledavaju alatom Kibana.

4.2.4. Integracija servisa

Za komunikaciju između servisa korišten je već spomenuti RPC. RPC je odabran zbog svoje brzine, male količine dodatnih podataka i dobre podržanosti u Golang programskom jeziku. Kako su svi servisi pisani u Golangu, nije bilo problema s kompatibilnosti RPC-a kod razvoja. U maloj je mjeri korišten i REST koji je služio razmjenu konfiguracijske datoteke između servisa. Točnije, servis *configservice* imao je ulogu servera s konfiguracijskom datotekom koja je bila dostupna svim ostalim servisima putem REST-a i HTTP-a.

4.2.5. Baza podataka

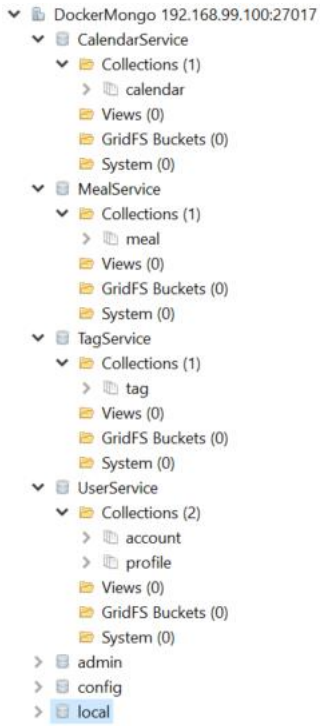
Za bazu podataka odabrana je MongoDB koja pripada nerelacijskim, tzv. NoSQL bazama podataka. U slučaju Lunch Plan aplikacije, zahtjev je kod baze podataka bio da se može skalirati, što vrlo dobro podržava i sam MongoDB.

Odnos se baze podataka i servisa kod tradicionalnih SQL baza podataka može posložiti na nekoliko načina:

- Privatne tablice po servis – svaki servis ima svoje privatne tablice u koje samo on zapisuje podatke
- Shema po servisu – svaki servis ima svoju shemu unutar baze podataka kojoj pristupa samo on
- Baza podataka po servisu – svaki servis ima svoj poslužitelj baze podataka (Richardson, Database per service, 2018)

Kod MongoDB baze podataka nema sheme pa je odabran način gdje svaki servis ima svoju bazu podataka unutar jednog fizičkog poslužitelja. Točnije, sam MongoDB poslužitelj pokrenut je u jednoj replici, ali u sebi sadrži bazu podataka svakoga pojedinoga servisa.

Servisi ne mogu međusobno komunicirati putem baze podataka, npr. čitati ili pisati podatke u drugu bazu podataka. Time su servisi potpuno odvojeni i slabo povezani te je zapravo to cilj mikroservisa i njihove integracije. Za izvršavanje i testiranje, koje je provedeno s Lunch Plan aplikacijom, u konačnici nije uočena potreba za skaliranjem baze podataka na više od jedne pokrenute replike ili skaliranje putem razlamanja na krhotine (eng. *sharding*). To znači da, ukoliko se pokrene 10 replika servisa *userservice*, neće biti potrebno imati deset replika njegove baze podataka, već je dovoljna samo jedna.



Slika 9. Pregled baza podataka i kolekcija na poslužitelju baze

Slika 9 ekran je zaslona iz alata Studio 3T koji se za vrijeme razvoja koristio za spajanje i rad s MongoDB bazom. Na slici se vidi raspored baza podataka (CalendarService, MealService) na poslužitelju baze podataka te kolekcije svake baze koja se koristi u Lunch Plan aplikaciji.

4.3. Aplikacija Lunch Plan

4.3.1. Priprema

Pokretanje se aplikacije Lunch Plan izvodi pomoću alata Docker Toolbox, koji je dio programskog paketa Docker. Prije samog pokretanja Lunch Plan aplikacije potrebno je pripremiti pomoćne alate, što znači pokrenuti alat Docker Toolbox i pomoću njega kreirati skup Docker strojeva koje su povezane u svoju mrežu. Za lakše upravljanje skupom strojeva koristit će se Docker u Swarm načinu rada. Na Slika 10 vidi izlaz Docker-a, prema kojem se može zaključiti da je kreiran virtualni stroj, pokrenut je Swarm način rada i kreirana je virtualna mreža.

```
Karlo@DESKTOP-8FJ2K7N MINGW64 /d/GolangProjects/lunchplan (master)
$ docker-machine create --driver virtualbox --virtualbox-memory 4096 --virtualbox-disk-size 20000 swarm-manager-0
Running pre-create checks...
Creating machine...
(swarm-manager-0) Copying C:\Users\Karlo\.docker\machine\cache\boot2docker.iso to C:\Users\Karlo\.docker\machine\machine
s\swarm-manager-0\boot2docker.iso...
(swarm-manager-0) Creating VirtualBox VM...
(swarm-manager-0) Creating SSH key...
(swarm-manager-0) Starting the VM...
(swarm-manager-0) Check network to re-create if needed...
(swarm-manager-0) Windows might ask for the permission to configure a dhcp server. Sometimes, such confirmation window i
s minimized in the taskbar.
(swarm-manager-0) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: C:\Program Files\Doc
ker Toolbox\docker-machine.exe env swarm-manager-0

Karlo@DESKTOP-8FJ2K7N MINGW64 /d/GolangProjects/lunchplan (master)
$ eval "$(docker-machine env swarm-manager-0)"

Karlo@DESKTOP-8FJ2K7N MINGW64 /d/GolangProjects/lunchplan (master)
$ docker swarm init --advertise-addr 192.168.99.101
Swarm initialized: current node (sbyolkqu7ft9n1ybn7jkyczdu) is now a manager.

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

    docker swarm join --token SWMTKN-1-3iz6ku1kw01ffzsoj514x9kyqkfn4h805scseeljamhkd2i6bx-356g0yeney2c8ryotec9ehqu9 192.
168.99.101:2377

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

Karlo@DESKTOP-8FJ2K7N MINGW64 /d/GolangProjects/lunchplan (master)
$ docker network create --driver overlay my_network
xwxgmkqf8dje480repb0g3db5
```

Slika 10. Pokretanje Docker-a u Swarm načinu rada

4.3.2. Kompiliranje

Sljedeći je korak kompilirati izrađene Docker slike u trenutnom čvoru (eng. *node*), zvanom „swarm-manager-0“, koji je jedini čvor u napravljenom skupu te je ujedno i menadžer skupa. Izrađene je slike kontejnera potrebno kompilirati jer se to kod pokretanja ne odvija automatski, nego se najčešće preuzimaju već gotove i pripremljene slike s Interneta, kao golang, mongodb i dr. koje se koriste u Lunch Plan-u.

Servisi Lunch Plan aplikacije pokreću se zasebno, svaki pomoću svoje slike pa ih je potrebno tako i kreirati. No, prije nego kreiramo slike servisa, kreirat će se slika univerzalna za sve servise koja će služiti kao okružje u kojem se kompiliraju servisi.

Slika kontejnera za kompiliranje drugih servisa nazvana je *lunchplan/gocompileimage* i sadrži instalirane biblioteke koje su potrebne za kompiliranje Go izvršnog programa. Slika 11 prikazuje primjer „Dockerfile“ datoteke za *lunchplan/gocompileimage* sliku prema kojoj je Docker izrađuje.

```
FROM golang:latest AS builder
ARG foldername
LABEL maintainer = ksimunovic
WORKDIR $foldername
COPY . .
RUN go get -u github.com/gorilla/handlers
RUN go get -u github.com/gorilla/mux
RUN go get -u golang.org/x/crypto/bcrypt
RUN go get -u gopkg.in/mgo.v2
RUN go get -u gopkg.in/mgo.v2/bson
```

Slika 11. Dockerfile za sliku *lunchplan/gocompileimage*

Nakon što je *lunchplan/gocompileimage* kreirana, na redu je kreiranje slika pojedinih servisa. Slika servisa se kreira na sljedeći način:

1. Učita se *lunchplan/gocompileimage* s nazivom „builder“
2. Postavi se trenutni radni direktorij prema proslijeđenom argumentu pokretanja
3. Kopiraju se datoteke za kompiliranje koje se nalaze gdje i pokretani Dockerfile
4. Izvrši se kompiliranje
5. Učita se nova slike naziva *alpine* koja sadrži minimalno okružje za pokretanje izvršnih programa
6. U *alpine* se sliku iz *builder*-a prebacuju sve datoteke te u konačnici izvršava servi naredbom CMD [“./app“]

Ovakav način kreiranja slika naziva se višefazni (eng. *multi-stage*), a moguć je od Docker verzije 17.05, dok je trenutna korištena 18.03. Tim se načinom iskorištava *lunchplan/gocompileimage* slika sa svim svojim učitanim datoteka, a konačna je slika servisa minimalne veličine jer se regulirajuća slika sastoji od izvršnog programa i minimalnog okružja za pokretanje kojeg pruža alpine slika. Na Slika 12 može vidjeti izlaz komandne linije kod kreiranja lunchplan/configservice slike.

```
Karlo@DESKTOP-8FJ2K7N MINGW64 /d/GolangProjects/lunchplan (master)
$ docker build --no-cache -t lunchplan/configservice --build-arg foldername=/usr/src/app configservice/
Sending build context to Docker daemon  20.19MB
Step 1/12 : FROM lunchplan/gocompileimage AS builder
 ---> ab084dd006af
Step 2/12 : ARG foldername
 ---> Running in cecd8d8daa38
Removing intermediate container cecd8d8daa38
 ---> e16695cbc71f
Step 3/12 : LABEL maintainer = ksimunovic
 ---> Running in bb059236b486
Removing intermediate container bb059236b486
 ---> 9a816b3176b0
Step 4/12 : WORKDIR $foldername
 ---> Running in ffbcb93b0217
Removing intermediate container ffbcb93b0217
 ---> 2786346583cc
Step 5/12 : COPY . .
 ---> 828f1c87b03c
Step 6/12 : RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
 ---> Running in 3d8f5d058fa5
Removing intermediate container 3d8f5d058fa5
 ---> 942dc20ee263
Step 7/12 : FROM alpine:latest
 ---> 11cd0b38bc3c
Step 8/12 : ARG foldername
 ---> Running in 006f3979c7d0
Removing intermediate container 006f3979c7d0
 ---> 13ec6fd8ccfa
Step 9/12 : WORKDIR /root/
 ---> Running in 29d10e9da8d8
Removing intermediate container 29d10e9da8d8
 ---> 961e31036ce8
Step 10/12 : COPY --from=builder $foldername/app .
 ---> b4c59f738223
Step 11/12 : EXPOSE 50000
 ---> Running in ce61aa5bb942
Removing intermediate container ce61aa5bb942
 ---> 784e445758fd
Step 12/12 : CMD ["./app"]
 ---> Running in ea6ef78ae244
Removing intermediate container ea6ef78ae244
 ---> b9080cda236c
Successfully built b9080cda236c
Successfully tagged lunchplan/configservice:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All fi
les and directories added to build context will have '-rwxr-xr-x' permissions. It is recommended to doub
le check and reset permissions for sensitive files and directories.
```

Slika 12. Kreiranje slike lunchplan/configservice

Na isti se način kreira i ostalih 7 servisa Lunch Plan aplikacija, zajedno sa slikom baze podataka nazvane *lunchplan/mongo-seed*. Slika baze podataka u sebi nema nikakve izmjene u odnosu na preuzetu sliku, no ima učitane pomoćne datoteke koje sadrže inicijalne podatke za lakše testiranje aplikacije.

4.3.3. Pokretanje

Nakon što je Swarm mode pokrenut i nakon što su kreirane slike prema kojima će se kreirati kontejneri, sljedeći je korak isporuka (eng. *deploy*) svih pripremljenih servisa u skup. Kako je Docker u Swarm načinu, koristi se naredba „docker stack deploy“ kojoj su ulazni argumenti naziv datoteke u kojoj se nalaze upute za isporuku servisa, najčešće naziva docker-compose.yml, i naziv skupa.

```
Karlo@DESKTOP-8FJ2K7N MINGW64 /d/GolangProjects/lunchplan (master)

$ docker stack deploy -c docker-compose.yml lunchplan
Creating network lunchplan_default
Creating service lunchplan_mongodb
Creating service lunchplan_portainer
Creating service lunchplan_logspout
Creating service lunchplan_elasticsearch
Creating service lunchplan_kibana
Creating service lunchplan_calendarservice
Creating service lunchplan_viz
Creating service lunchplan_apigateway
Creating service lunchplan_configservice
Creating service lunchplan_mealservice
Creating service lunchplan_htmlservice
Creating service lunchplan_userservice
Creating service lunchplan_tagservice
Creating service lunchplan_logstash
Creating service lunchplan_apiservice
```

Slika 13. Isporuka svih servisa u skup lunchplan

Na Slika 13 je vidljivo da su svi servisi uspješno kreirani bez grešaka. Osim izrađenih 8 servisa u Golang-u i baze podataka MongoDb koriste se već spomenuti servisi kao što su Elasticsearch, Logsput, Kibana, Logstash, te dva servisa, portainer koji služi za vizualizaciju servisa i kontejnera po čvorovima i njihovo upravljanje i viz koji služi samo za vizualizaciju kontejnera.

```
Karlo@DESKTOP-8FJ2K7N MINGW64 /d/GolangProjects/lunchplan (master)
$ docker service ls
ID                  NAME                      MODE                REPLICAS            IMAGE                                PORTS
yo6ry7qc831a        lunchplan_apigateway      replicated          1/1                 lunchplan/apigateway:latest          *:80->80/tcp, *:4430->4430/tcp
xzcig1ay5x7k        lunchplan_apiservice      replicated          1/1                 lunchplan/apiservice:latest          *:50001->50001/tcp
u2nutsq0xt63        lunchplan_calendarservice replicated          1/1                 lunchplan/calendarservice:latest     *:50006->50006/tcp
yyoh1z0paxbe        lunchplan_configservice   replicated          1/1                 lunchplan/configservice:latest       *:50000->50000/tcp
ywgyas5jimo8        lunchplan_elasticsearch   replicated          1/1                 elasticsearch:latest
hzf7i9s0sghc        lunchplan_htmlservice     replicated          1/1                 lunchplan/htmlservice:latest         *:50002->50002/tcp
vyhgj3to5zgk        lunchplan_kibana          replicated          1/1                 kibana:latest                        *:5601->5601/tcp
wmxki7isrczb        lunchplan_logspout        replicated          1/1                 gliderlabs/logspout:v3
eac9gmos5yn2        lunchplan_logstash        replicated          1/1                 logstash:5.6.8                       *:5000->5000/tcp
w5wl6kptquc6        lunchplan_mealservice     replicated          1/1                 lunchplan/mealservice:latest         *:50004->50004/tcp
i3k95v3f5q5i        lunchplan_mongodb         replicated          1/1                 lunchplan/mongo-seed:latest          *:27017->27017/tcp
f4u5m6emscwz        lunchplan_portainer       replicated          1/1                 portainer/portainer:latest           *:9000->9000/tcp
x1h159w4tkhx        lunchplan_tagservice      replicated          1/1                 lunchplan/tagservice:latest          *:50005->50005/tcp
u6z2pa46ig8f        lunchplan_userservice     replicated          1/1                 lunchplan/userservice:latest         *:50003->50003/tcp
ijrfoghxrr4r        lunchplan_viz             replicated          1/1                 dockersamples/visualizer:latest      *:9090->8080/tcp
```

Slika 14. Status pokrenutih servisa

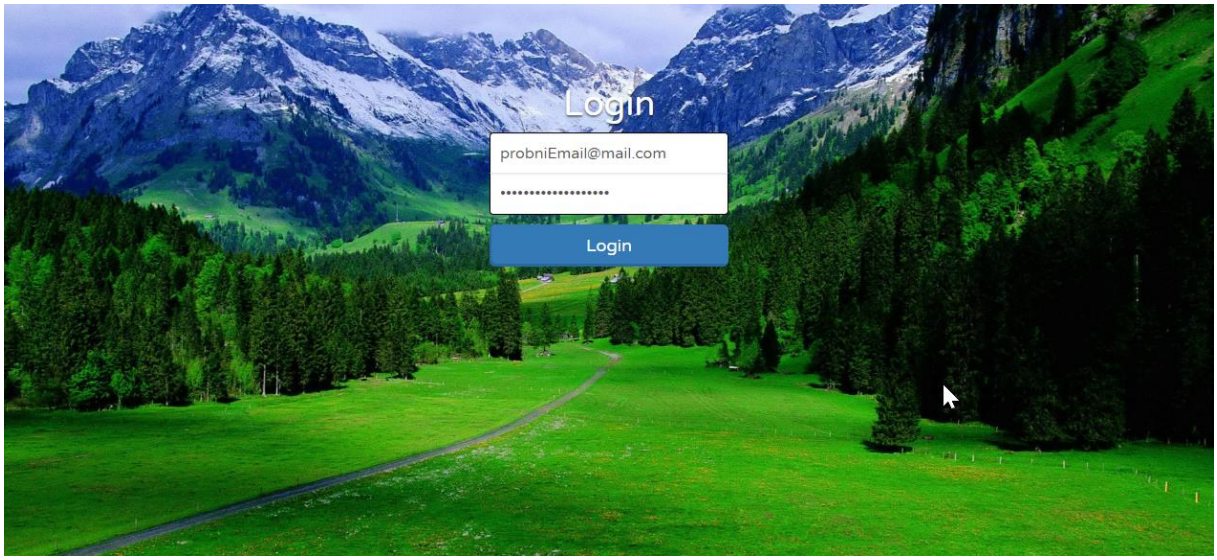
Slika 14 pokazuje ispis naredbe „docker service ls“ koji pokazuje koji su servisi svi pokrenuti. Svaki pokrenuti servis dobije svoj univerzalni identifikator ID, ime koje se sastoji od naziva skupa, u ovom slučaju lunchplan, i naziva servisa, informacije u koliko replika je pokrenut, slike prema kojoj je servis nastao te kako su povezana unutarnja i vanjska vrata (eng. *ports*) mreže skupa.

Svi su servisi u ovome trenutku pokrenuti samo s jednom replikom, što znači da se trenutno ne koriste nikakve pogodnosti skaliranja koje Docker podržava.

Lunch Plan je aplikacija sada pokrenuta i u potpunosti spremna za korištenje. Najlakši je način za njeno testiranje koristeći web preglednik. Za potrebe razvoja aplikacije i slikanja zaslona korišten je web preglednik Google Chrome verzije 68.

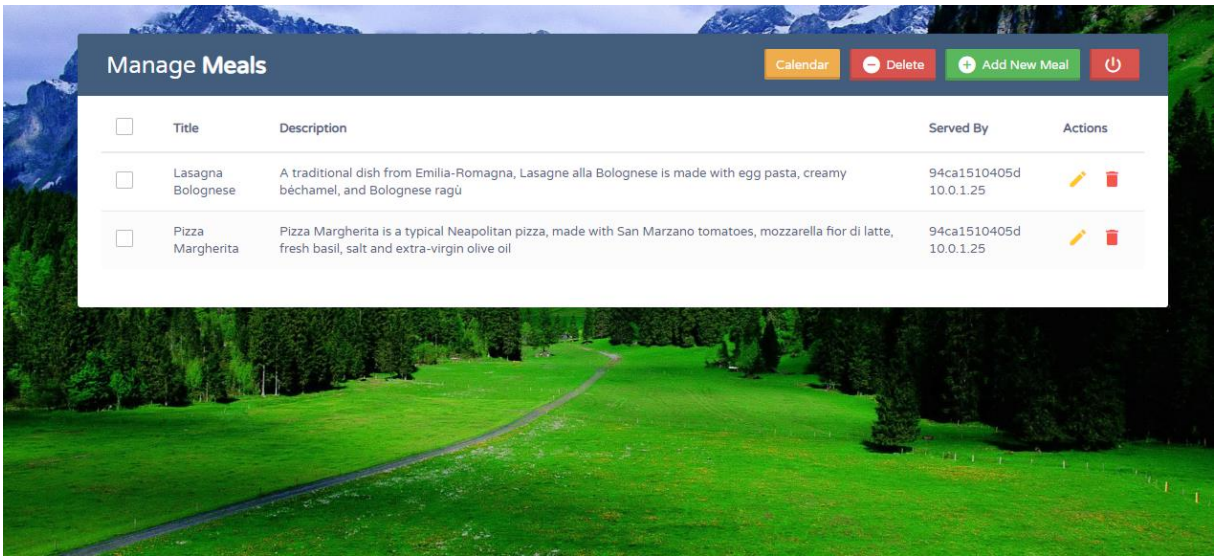
4.3.4. Korištenje

Cilj je Lunch Plan aplikacije da bude korištena od više različitih korisnika te je prva stvar koja se prikazuje posjetom adrese docker/ upravo stranica za prijavu (Slika 15) korisnika jer bez prijave, korisniku nisu dostupni ostali dijelovi aplikacije.



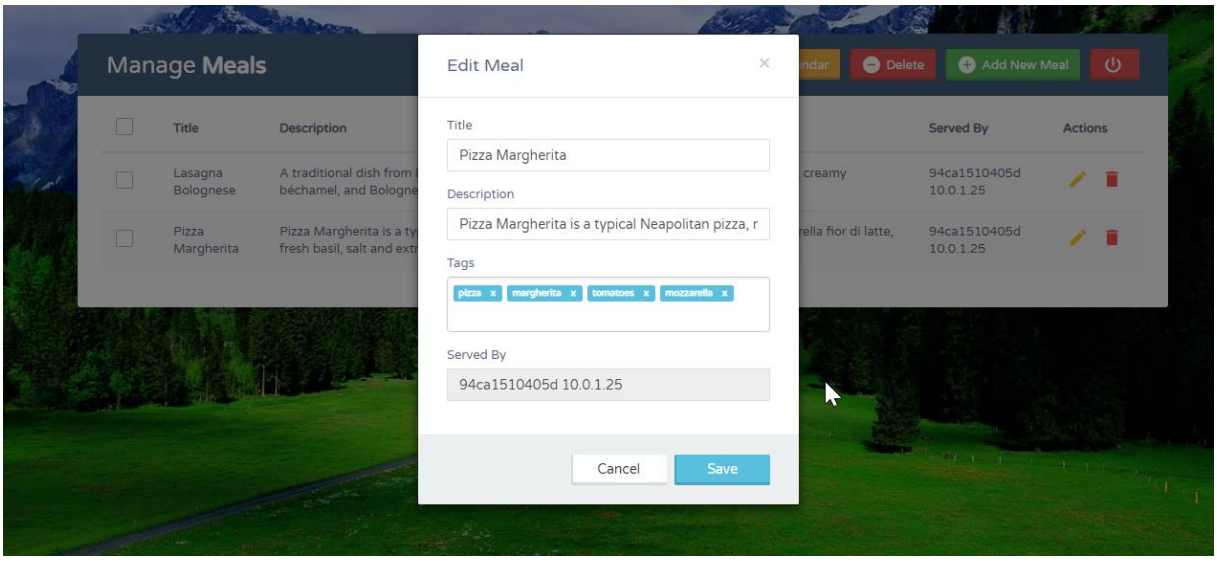
Slika 15. Stranica za prijavu u Lunch Plan

Uspješnom se prijavom korisnik dalje preusmjerava na stranicu u kojem ima pregled nad svim svojim definiranim obrocima, što se vidi na Slika 16. Isto tako, na trenutnoj se stranici može dodavati, uređivati i brisati obroke povezane s njegovim računom. Osim osnovnih informacija o obroku, također se može vidjeti i identitet servisa s kojega je dohvaćen objekt obroka. Identitet servisa nije identičan onome kojega Docker dodjeli, već se određuje u programskom kodu pomoću naziva poslužitelja i njegove lokalne IP adrese. Kako su svi obroci za stranicu pregleda obroka dohvaćeni u jednom pozivu i kako je trenutno samo jedna replika *mealservice* servisa, postoji samo jedan identitet servisa.



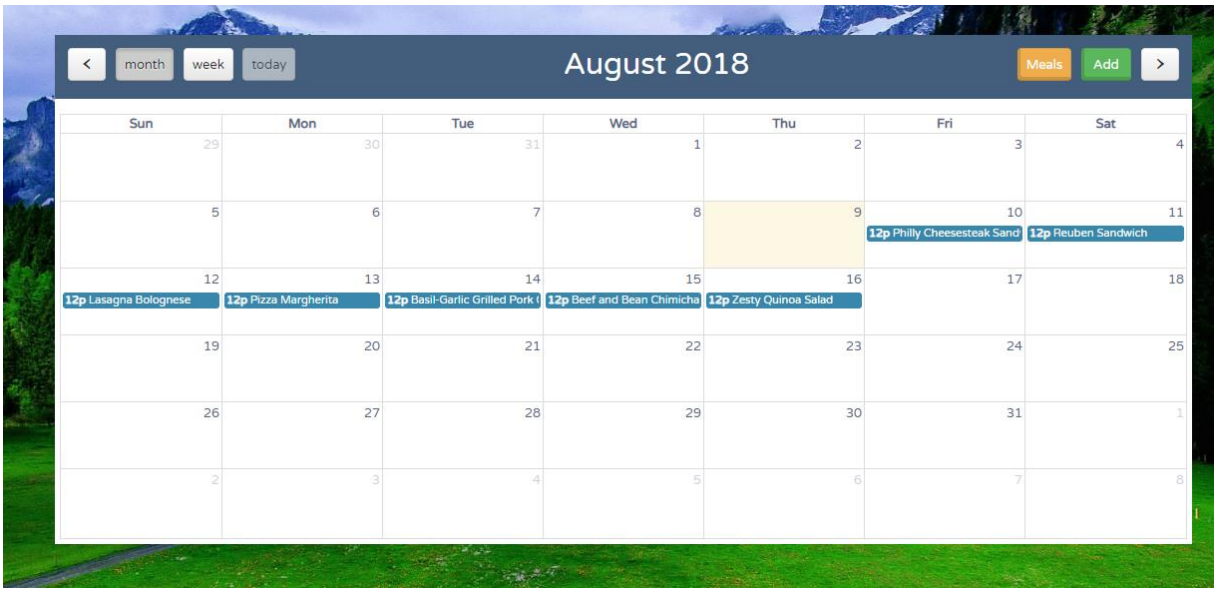
Slika 16. Stranica sa svim obrocima korisnika

Na Slika 17 prikazan prozor kojim se uređuju podaci vezani za obrok. Na njemu se isto tako može vidjeti naziv servisa s kojega je dohvaćen. Koliko god puta otvorili taj prozor, naziv servisa će ostati isti jer postoji samo jedna njegova replika.



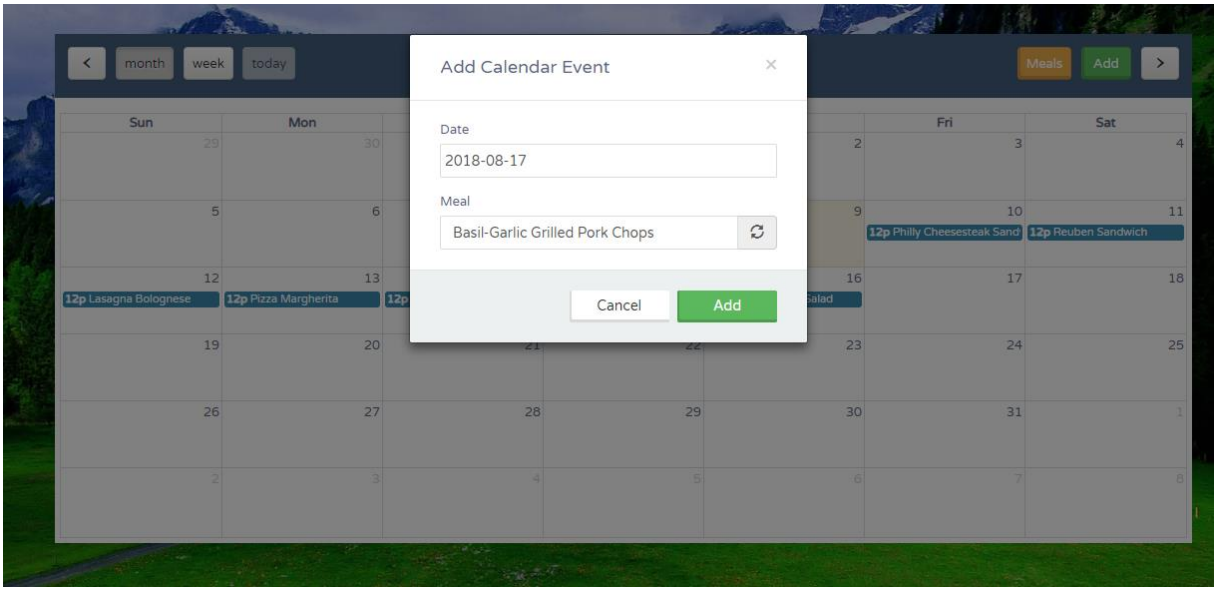
Slika 17. Prozor za uređivanje obroka

Nakon što je korisnik unio jela koja želi imati na svojem popisu, on može krenuti s planiranjem obroka. Na stranici za obroke postoji poveznica s nazivom *Calendar* koja vodi korisnika na kalendarski prikaz njegovoga plana.



Slika 18. Kalendarski prikaz korisnikovog plana

Posjetom stranice naziva *Calendar*, korisnik dolazi do kalendarskog prikaza planiranih jela. U gornjem lijevom kutu korisnik može birati između različitih prikaza kalendara, ovisno želi li vidjeti plan na mjesečnoj, tjednoj ili dnevnoj bazi. Na Slika 18 vidi kalendarski prikaz korisnikovoga ekrana gdje je unesen raspored obroka za interval od tjedan dana od 9. do 16. kolovoza 2018. godine.



Slika 19. Prozor za dodavanje obroka u kalendar

Klikom na bilo koji dan na kalendaru, korisniku se otvara prozor za dodavanje novog događaja, odnosno obroka. Upravo se na ovom prozoru (Slika 19) dolazi do idejne funkcionalnosti Lunch Plan aplikacije, a to je mogućnost nasumičnog generiranja obroka za plan.

4.3.5. Skaliranje

Skaliranjem se utječe na resurse ili instance aplikacije s ciljem povećavanja njezinih performansi, boljem podnošenju opterećenja, povećanje stabilnosti i sl. Skalirati aplikaciju možemo vertikalno i horizontalno. Vertikalno je skaliranje tipično za klasične monolitne web aplikacije i SQL baze podataka, gdje performanse i ostale karakteristike povećavaju dodavanjem jačih komponenti u postojeći poslužitelj, npr. bolji procesor, više RAM memorije, itd. Nakon vertikalnog skaliranja dobije se jači poslužitelj koji zbog svojih boljih komponenti poboljšava navedene karakteristike aplikacije.

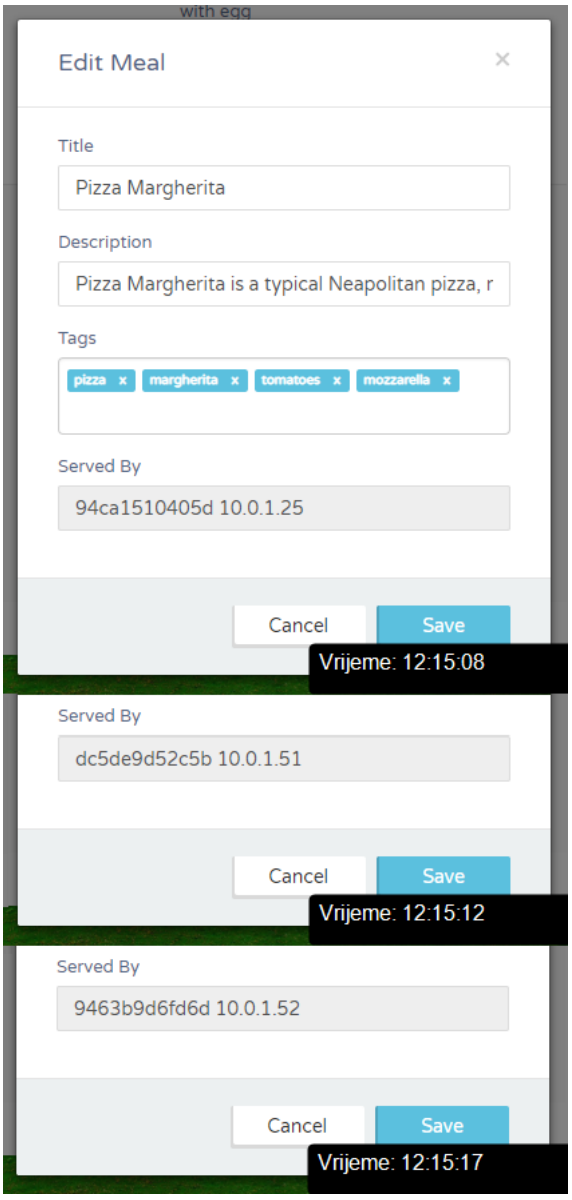
Kod horizontalnog je skaliranja priča nešto drugačija. Horizontalno skaliranje tipično je za mikroservisnu arhitekturu i NoSQL baze podataka općenito. Taj način skaliranja, umjesto nadogradnje ili novog jačeg poslužitelja aplikacije, poboljšava njezine karakteristike tako da se uvede još jedan ili više dodatnih poslužitelja. Horizontalno skaliranje zahtjeva i dodatni poslužitelj ili servis koji će upravljati opterećenjem i raspoređivati ga na postojeći i dodane poslužitelje. Prednost horizontalnog skaliranja je da kod, npr. ispada jedne replike servisa, postoje druge koje mogu preuzeti zahtjeve pa se potpuni ispadi aplikacije rjeđe događaju.

Kako bi se pokazala jednostavnost skaliranja servisa u Dockeru i utjecaj na karakteristike aplikacije *mealservice*, servis će Lunch Plan-a biti skalirani horizontalno, na 3 replike, kroz *docker-compose.yml* datoteku.

```
Karlo@DESKTOP-8FJ2K7N MINGW64 /d/GolangProjects/lunchplan (master)
$ docker service ls
ID                  NAME                        MODE                REPLICAS            IMAGE                                PORTS
yo6ry7qc831a        lunchplan_apigateway        replicated          1/1                 lunchplan/apigateway:latest          *:80->80/tcp, *:4430->4430/tcp
xzcig1ay5x7k        lunchplan_apiservice        replicated          1/1                 lunchplan/apiservice:latest          *:50001->50001/tcp
u2nutsq0xt63        lunchplan_calendarservice   replicated          1/1                 lunchplan/calendarservice:latest     *:50006->50006/tcp
yyoh1z0paxbe        lunchplan_configservice     replicated          1/1                 lunchplan/configservice:latest       *:50000->50000/tcp
ywgyas5jimo8        lunchplan_elasticsearch     replicated          1/1                 elasticsearch:latest
hzf7i9s0sghc        lunchplan_htmlservice       replicated          1/1                 lunchplan/htmlservice:latest         *:50002->50002/tcp
vyhgj3to5zgk        lunchplan_kibana            replicated          1/1                 kibana:latest                        *:5601->5601/tcp
wmxki7isrczb        lunchplan_logspout          replicated          1/1                 gliderlabs/logspout:v3
eac9gmos5yn2        lunchplan_logstash          replicated          1/1                 logstash:5.6.8                       *:5000->5000/tcp
w5wl6kptquc6        lunchplan_mealservice       replicated          3/3                 lunchplan/mealservice:latest         *:50004->50004/tcp
i3k95v3f5q5i        lunchplan_mongodb           replicated          1/1                 lunchplan/mongo-seed:latest          *:27017->27017/tcp
f4u5m6emscwz        lunchplan_portainer         replicated          1/1                 portainer/portainer:latest           *:9000->9000/tcp
x1h159w4tkhx        lunchplan_tagservice        replicated          1/1                 lunchplan/tagservice:latest          *:50005->50005/tcp
u6z2pa46ig8f        lunchplan_userservice       replicated          1/1                 lunchplan/userservice:latest         *:50003->50003/tcp
ijrfoghxrr4r        lunchplan_viz               replicated          1/1                 dockersamples/visualizer:latest      *:9090->8080/tcp
```

Slika 20. Mealservice skaliran na tri replike

Nakon što je *mealservice* uspješno skaliran na tri replike, može se kroz sučelje aplikacije provjeriti jesu li sve tri replike dostupne. Ako se vratimo na stranicu sa svim obrocima (Slika 16) i uzastopno otvorimo pa zatvorimo prozor za uređivanje obroka, vidjet će se da su replike u uporabi.



Slika 21. Prozor za uređivanje obroka s tri različite replike

Slika 21 pokazuje prozor koji u sebi sadrži informaciju koji je servis odgovoran za dostavljene podatke, polje *Served By*. Vidljivo je da su sva tri identifikatora servisa različita, iz čega se može zaključiti da su podaci došli s tri različita servisa i da replike ispravno rade. Podaci sva tri servisa su identični, ukoliko između prozora nije bilo promjena podataka od nekog drugog korisnika, jer sve tri replike koriste istu bazu podataka.

4.4. Performanse nakon skaliranja

Kako bi se pokazao utjecaj skaliranja na performanse servisa, izabran je servis koji služi za registraciju, prijavu i validaciju korisnika, naziva *userservice*. Metoda za prijavu na adresi /api/login jedna je od kompleksnijih te zahtjeva veću količinu resursa za spajanje na

bazu podataka, provjeru podataka prijave, kreiranje i spremanje nove korisničke sesije i vraćanje iste korisniku.

Za simuliranje opterećenja korišten je alat Gatling koji je pokretan sljedećom naredbom:

mvn gatling:execute -Dusers=100 -Dduration=60 -DbaseUrl=https://docker:4430/api/login

Argumenti korišteni pri pokretanju su sljedeći:

- users: označava broj istovremenih korisnika koje će test simulirati
- duration: vrijeme kroz koje će test simulirati zahtjeve u sekundama
- baseUrl: označava URL na koji će se slati zahtjevi

Zahtjevi se šalju POST metodom zajedno s login podacima jednog korisnika. Simulira se 100 korisnika koji se istovremeno pokušavaju prijaviti preko API-ja aplikacije, a koji se generiraju linearno kroz vremensko razdoblje od 60 sekundi. Podatak koji će se gledati kao referenta vrijednost performansi kod ovih simulacija bit će prosječno vrijeme odgovora servisa.

Prvi je test pokrenut s jednom replikom servisa *userservice*. Cijela simulacija traje u prosjeku 120 sekundi, dok alat zaprimi odgovore svih poslanih zahtjeva u prvih 60 sekundi.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1207	1207	0	0%	10.035	72	3792	6373	8693	10068	11190	4180	2689
ApiLogin	1207	1207	0	0%	10.035	72	3792	6373	8693	10068	11190	4180	2689

Slika 22. Rezultati simulacije 1 opterećenja servisa *userservice*

Rezultati simulacije 1 pokazuju da je prema *userservice* poslano ukupno 1207 zahtjeva od kojih su svi bili uspješni (stupac „OK“), prosječnom brzinom od 10,035 zahtjeva po sekundi (stupac „Req/s“). Prosječno vrijeme odgovora na 100 istovremenih korisnika u rasponu od 60 sekundi je 4180 milisekundi (stupac „Mean“) uz standardnu devijaciju (stupac „Std Dev“) od 2689 milisekundi. To znači da servis *userservice* odgovara u prosjeku svakih 4,2 s kada se izvršava jedna njegova replika.

Drugi test s istim servisom izveden je kada je servis bio pokrenut u pet replika, a vrijeme je trajanja faze zahtjeva 60 sekundi te je maksimalan broj korisnika 100.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1271	1271	0	0%	10.612	71	3914	5714	8179	9285	10858	3895	2457
ApiLogin	1271	1271	0	0%	10.612	71	3914	5714	8179	9285	10858	3895	2457

Slika 23. Rezultati simulacije 2 opterećenja servisa *usersevice*

Rezultati simulacije 2 pokazuju da je poslano ukupno 1271 zahtjeva prosječnom brzinom od 10,612 zahtjeva u sekundi. Prosječno je vrijeme odgovora servisa 3895 milisekundi, uz standardnu devijaciju od 2457 milisekundi. Ako se usporede prosječna vremena servisa s 1 i s 5 replika, vidljivo je da je izvedba s 5 replika u prosjeku brža za ~280 milisekundi.

Iako poboljšanje nije veliko, ono postoji, a razlog sličnosti rezultata je činjenica da su oba testa pokrenuta na prijenosnom računalu autora, točnije, na istom virtualnom stroju s Docker Swarm čvorom i istom hardveru. U praksi, uz kreiranje replika, dodao bi se i novi čvor Swarm skupu na koji bi Docker svojim Swarm mehanizmom podijelio opterećenje servisa te bi se bolje iskoristili dostupni resursi.

Povećanje broja replika na 10 navedenoga servisa, blago se povećava prosječno vrijeme odgovora na 4100 milisekundi, što bi značilo da se dosegnula točka u kojoj povećavanje broja replika unutar trenutnog Swarm skupa od jednog čvora negativno utječe na performanse servisa. U tom je slučaju potrebno Swarm skup proširiti novim čvorovima radnicima (eng. *worker*).

Kako bi se vidio utjecaj broja istovremenih korisnika koji se pokušavaju prijaviti u aplikaciju putem API-a, pokrenuta je još jedna simulacija, no ovoga puta s 10 istovremenih korisnika (za razliku od prethodnih 100). I u ovoj simulaciji korišteno je 5 replika servisa dok su preostali parametri pokretanja nepromijenjeni.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	480	480	0	0%	4.173	69	219	353	570	749	1572	259	179
ApiLogin	480	480	0	0%	4.173	69	219	353	570	749	1572	259	179

Slika 24. Rezultati simulacije 3 opterećenja servisa *usersevice*

Rezultati simulacije 3 pokazuju da je poslano ukupno 480 uspješnih zahtjeva prosječnom brzinom od 4,173 zahtjeva u sekundi. Prosječno vrijeme odgovora značajno je manje nego kod 100 istovremenih korisnika i iznosi 259 milisekundi po zahtjevu uz standardnu devijaciju od 179 milisekundi.

Odgovor od 259 milisekundi puno je prihvatljiviji glede korisničkog iskustva i performansi aplikacije općenito. Vidljivo je također, da s istim brojem replika, broj korisnika koji je 10 puta veći u istom vremenskom periodu, daje povećanje prosječnog odgovora od čak petnaest puta.

4.5. Opis servisa

Lunch plan aplikacija realizirana je pomoću osam servisa ukupno, a svi su oni pisani u programskome jeziku Golang. Servisi imaju razne uloge te neki od njih implementiraju spomenute uzorke dizajna.

ConfigService je jednostavan servis koji u svom programskom kodu ima zapisan objekt s konfiguracijskim parametrima. Jedina uloga servisa je da pri posjetu njegove putanje, pomoću HTTP-a, vrati JSON reprezentaciju objekta s konfiguracijom. Pri paljenju, svi servisi prvo dohvaćaju taj konfiguracijski objekt kako bi se mogli povezati na bazu podataka, otvorili svoja mrežna vrata i povezali se s drugim servisima.

ApiGateway implementira, kako mu i samo ime govori, API Gateway uzorak. Prema URL-u kojim korisnik ili aplikacija (Javascript u pozadini) pristupa servisu (odnosno ima li on u sebi prefiks „/api/“ ili nema), prosljeđuje zahtjev prema *apiservice* ili *htmlservice* servisima. Osim provjere URL-a, ovaj servis vrši i validaciju korisnika pomoću zaglavlja njegovog HTTP zahtjeva s *userservice*-om. *ApiGateway* osluškuje nadolazeće zahtjeve na mrežnima vratima 4430 jer je želja autora bila implementirati SSL kod pristupa HTML stranica, što je i postignuto, no, zbog problema s operativnim sustavom macOS, postavljena su vrata 4430. Osim *configservice*-a, *apigateway* je servis jedini kojemu se može pristupiti direktno, npr. putem preglednika. Tablica 3 pokazuje da su ostali servisi dostupni na drugim vratima, no ta su vrata lokalne mreže u kojoj servisi komuniciraju putem RPC-a, a ne HTTP-a.

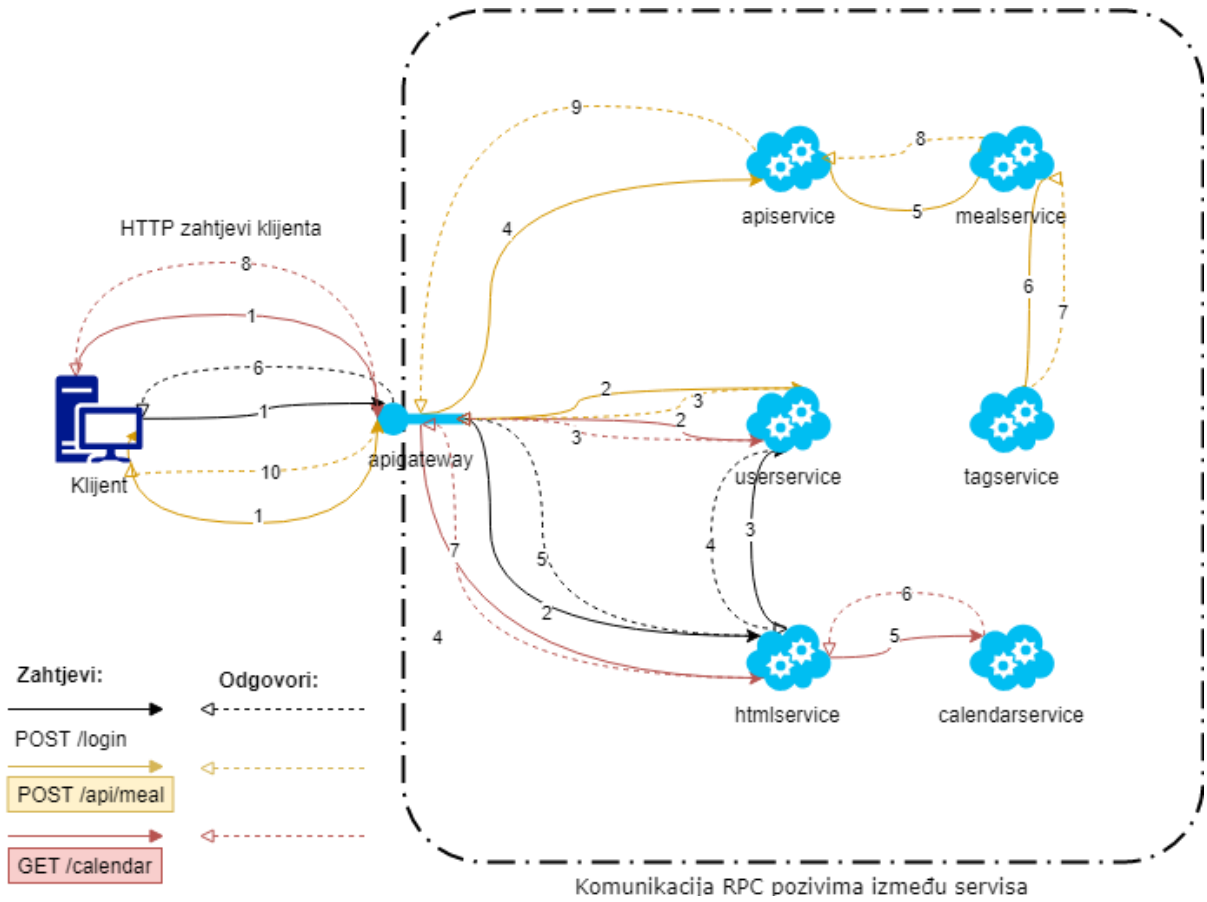
ApiService preuzima sve zahtjeve koji u svojem URL-u imaju prefiks „/api/“ te svojim unutarnjim usmjeravanjem (eng. *routing*) dalje prosljeđuje zahtjeve odgovarajućim servisima. Odgovor servisa prosljeđuje dalje izvoru zahtjeva.

HtmlService servis služi za prikaz i upravljanje korisničkog sučelja kada se aplikaciji pristupa putem preglednika. On u sebi ima implementiran MVC uzorak, gdje su jasno odvojeni pogled, upravljač i model podataka. Kako na modelu podataka nije bilo potrebe za dodatnom implementacijom, objekt se samo preuzima od drugog servisa, pretvara u odgovarajući tip objekta (npr. Meal ili Tag) i koristi dalje. *HtmlService* i *ApiService* su primjer uzorka Backends For Frontends jer potpuno odvajaju dvije različite grupe klijenata, preglednika i aplikacije.

MealService, *TagService* i *CalendarService* preostala su tri servisa koji su međusobno veoma slična. U sebi sadrže poslovnu logiku aplikacije i omogućuju jednostavne CRUD kontrole nad objektima u bazi

Tablica 3. Servisi, njihove metode, putanje i odgovori

Naziv	**Vrata**	**Putanje servisa**		
		HTTP Metoda	**Adresa**	**Odgovor**
configservice	50000	GET	/	Vraća JSON objekt koji sadrži konfiguracijske parametre cijele aplikacije
apigateway	4430	GET/POST	/	Provjerava je li korisnik prijavljen i koliko URL zahtjeva počinje s "/api/" prosljeđuje zahtjev *apiservice*-u, u suprotnome prosljeđuje zahtjev *htmlservice*-u
apiservice	50001	GET/POST	/api/{nazivMetode}	Vraća rezultat izvršavanja tražene metode API-a
htmlservice	50002	GET/POST	/{nazivStranice}	Vraća HTML traženu stranicu
userservice	50003	POST	/api/register	Registrira korisnika u aplikaciju i vraća mu objekt njegovog profila ako je registracija uspješna
		POST	/api/login	Prijavljuje korisnika u aplikaciju i vraća mu ID sesije ako je prijava uspješna
		GET	/api/account	Vraća podatke o prijavljenom korisniku pomoću ID-a sesije iz zahtjeva
mealservice	50004	POST	/api/meal	Unosi obrok zajedno s pripadajućim oznakama I vraća objekt unesenog obroka
		GET	/api/meal/all	Vraća kolekciju obroka prijavljenog korisnika
		GET	/api/meal/{id}	Vraća obrok korisnika prema ID-u
		POST	/api/meal/{id}	Ažurira obrok pomoću poslanog objekta
		DELETE	/api/meal/{id}	Briše obrok korisnika prema ID-u
tagservice	50005	POST	/api/tag	Unosi oznaku i vraća objekt unesene oznake
		GET	/api/tag/all	Vraća kolekciju oznaka prijavljenog korisnika
		GET	/api/tag/{id}	Vraća oznaku korisnika prema ID-u
		POST	/api/tag/{id}	Ažurira oznaku pomoću poslanog objekta
		DELETE	/api/tag/{id}	Briše oznaku korisnika prema ID-u
calendarservice	50006	POST	/api/tag	Unosi događaj kalendara i vraća objekt unesene oznake
		GET	/api/tag/all	Vraća događaja prijavljenog korisnika
		GET	/api/tag/{id}	Vraća događaj korisnika prema ID-u
		POST	/api/tag/{id}	Ažurira događaj pomoću poslanog objekta
		DELETE	/api/tag/{id}	Briše događaj korisnika prema ID-u



Slika 25. Komunikacija između servisa Lunch Plan aplikacije

Radi lakše predodžbe odnosa i načina obrade zahtjeva, na Slika 25 prikazana tri različita zahtjeva te njihovi koraci obrade. Točkastom linijom označeni su odgovori na zahtjeve. Jedan od najkompleksnijih zahtjeva za izvršavanja koji prolazi kroz najviše servisa označen je žutom bojom i predstavlja POST zahtjev za kreiranje obroka na putanji „/api/meal“.

U prvom koraku navedeni zahtjev dolazi do servisa *apigateway* gdje se zbog autentifikacijog tokena u zaglavlju provjerava sesija korisnika uz pomoć *userservice*-a. Ako je validacija korisnika uspješna, zahtjev se prosljeđuje na *mealservice*. *MealService*, zbog poslovne logike kod kreiranja obroka, provjerava postoje li već navedeni tagovi u bazi podataka ili su novo kreirani te prema tome daje posao *tagservice*-u. Po završetku svih aktivnosti odgovor na zahtjev vraća se istim putem kojim je došao, s time da ne prolazi kroz *userservice*.

4.6. Programski kod

Kako programskog koda ima puno, mišljenje je autora da ga nije praktično stavljati u sklopu ovoga rada, ali smatra da bi trebao biti dostupan javno na uvid, tako da se ustupa Git repozitorij Lunch Plan aplikacije koji se nalazi na sljedećoj adresi:

https://github.com/ksimunovic/lunchplan

5. Zaključak

Mikroservisna arhitektura popularan je odabir za implementaciju web aplikacija zbog svih pogodnosti koje se dobivaju spajanjem ta dva koncepta. Web nam aplikacije omogućuju dostupnost u bilo kojem trenutku, u bilo kojem dijelu svijeta, dok god imamo vezu s Internetom. Njihove mogućnosti rastu svakim danom te je već sada rijetkost pronaći softver koji nema neku vrstu svoje online verzije, odnosno softver u obliku web aplikacije. Mikroservisna arhitektura, sa svojom skalabilnošću i otvorenošću prema skupu raznih tehnologija koje se mogu ujediniti, daje odgovarajuću fleksibilnost koja je potrebna za prilagođavanje, kako rastućem broju korisnika Interneta i web aplikacija, tako i broju tehnologija i alata dostupnih za razvoj istih.

Iako implementiranje web aplikacija kroz mikroservise donosi puno pogodnosti, ono imaju svoju cijenu. Razvojni proces takvih web aplikacije znatno je kompliciraniji i zahtjeva puno više vremena, znanja i koordinacije rada na projektu. Ako se usporedi vrijeme potrebno za razvoj aplikacije s istim funkcionalnostima, procjena je autora da je ono 4-5 puta veće kada se koristi skup tehnologija i način opisan u ovome radu, zajedno s procesom učenja, nego da se aplikacija radila u programskom jeziku PHP. To samo naglašava problematiku rada s velikim brojem tehnologija i alata u isto vrijeme i isto tako važnost alata za automatizaciju procesa razvoja.

Preporuka je autora koristiti mikroservise za web aplikacije ukoliko je skalabilnost, podrška za veliki broj korisnika ili podrška za veliki broj tehnologija koje aplikacija koristi ili s kojima komunicira, jedan od osnovnih zahtjeva u specifikaciji web aplikacije koja se kreira. U tom slučaju, prema mišljenju autora, mikroservisna je arhitektura idealno rješenje. Uz prave razvojne alate, alate za automatizaciju procesa razvoja i orkestraciju kontejnera, razvoj i održavanje web aplikacije temeljene na mikroservisima ne mora biti znatno teže od razvoja aplikacije u jednostavnijem skupu tehnologija s npr. PHP-om.

Popis Literature

Adamkó, D. A. (5. Ožujak 2018). *Chapter 4. Layered Architecture for Web Applications.* Dohvaćeno iz https://gyires.inf.unideb.hu/GyBITT/08/ch04.html

Adamkó, D. A. (5. Ožujak 2018). *Part II. Architectures for the Web.* Dohvaćeno iz https://gyires.inf.unideb.hu/GyBITT/08/pt02.html

Aprilindo Sentosa, D. (13. Kolovoz 2018). *Soa Architecture Best Of Dasa Aprilindo Sentosa - Architecture Design.* Dohvaćeno iz http://architecture-nice.com/soa-architecture/soa-architecture-best-of-dasa-aprilindo-sentosa/

Basic MVC Architecture. (10. Ožujak 2018). Dohvaćeno iz https://www.tutorialspoint.com/struts_2/basic_mvc_architecture.htm

Brasetvik, A. (13. Srpanj 2018). *Uses of Elasticsearch, and Things to Learn.* Dohvaćeno iz ElasticAlex Brasetvik: https://www.elastic.co/blog/found-uses-of-elasticsearch

Brown, K., & Woolf, B. (22. Ožujak 2018). *Implementation Patterns for Microservices Architectures.* Dohvaćeno iz https://hillside.net/plop/2016/papers/two/8.pdf

Buschmann, F., Meunier, R., Rohner, H., P. S., & Stal, M. (1996). *Pattern - Oriented Software Architecture A System of Patterns.* Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Dohvaćeno iz http://www.laputan.org/pub/papers/POSA-MVC.pdf

Chakraborty, S. (13. Kolovoz 2018). *ELK Stack Tutorial with Example.* Dohvaćeno iz HowToDoInJava: https://howtodoinjava.com/microservices/elk-stack-tutorial-example/

Design Patterns. (9. Ožujak 2018). Dohvaćeno iz https://sourcemaking.com/design_patterns

Despodovski, R. (13. Ožujak 2018). *Microservices vs. SOA – Is There Any Difference at All? - DZone Integration.* Dohvaćeno iz https://dzone.com/articles/microservices-vs-soa-is-there-any-difference-at-al

Evans, E. (2003). *Domain-Driven Design.* Addison-Wesley Professional.

Frequently Asked Questions (FAQ) - The Go Programming Language. (16. Lipanj 2018). Dohvaćeno iz https://golang.org/doc/faq#history

From History of Web Application Development. (2. Ožujak 2018). Dohvaćeno iz https://www.devsaran.com/blog/history-web-application-development

gliderlabs/logspout. (14. Srpanj 2018). Dohvaćeno iz GitHub: https://github.com/gliderlabs/logspout

Green, S. (2015). *How To Build Microservices: Top 10 Hacks To Modeling, Integrating & Deploying Microservices.* Scott Green.

Hollingworth, D. (13. Kolovoz 2018). *The model-view-controller pattern.* Dohvaćeno iz Dave Hollingworth - IT Trainer: https://daveh.io/blog/the-model-view-controller-pattern/

How AJAX works? (13. Kolovoz 2018). Dohvaćeno iz Virtual Training Center: http://vtc.topologypro.institute/lessons/how-ajax-works/

Issac, L. P. (13. Srpnja 2018). *SQL vs NoSQL Database Differences Explained with few Example DB.* Dohvaćeno iz https://www.thegeekstuff.com/2014/01/sql-vs-nosql-db/?utm_source=tuicool

Jansen, G. (13. Kolovoz 2018). *Methods.* Dohvaćeno iz http://restful-api-design.readthedocs.io/en/latest/methods.html

Kay, R. (18. Ožujak 2018). *Representational State Transfer (REST).* Dohvaćeno iz https://www.computerworld.com/article/2552929/networking/representational-state-transfer--rest-.html

Likness, J. (10. Ožujak 2018). *Model-View-ViewModel (MVVM) Explained - CodeProject.* Dohvaćeno iz https://www.codeproject.com/Articles/100175/Model-View-ViewModel-MVVM-Explained

Logstash Introduction. (14. Srpanj 2018). Dohvaćeno iz Logstash Reference [6.3] | Elastic: https://www.elastic.co/guide/en/logstash/current/introduction.html

Lussier, D. (10. Ožujak 2018). *MVVM Compared To MVC and MVP.* Dohvaćeno iz http://geekswithblogs.net/dlussier/archive/2009/11/21/136454.aspx

Mahmoud, Q. H. (7. Ožujak 2018). *Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI).* Dohvaćeno iz http://www.oracle.com/technetwork/articles/javase/soa-142870.html

Márton, P. (23. Ožujak 2018). *Designing a Microservices Architecture for Failure | @RisingStack.* Dohvaćeno iz https://blog.risingstack.com/designing-microservices-architecture-for-failure/

Maruti, T. (25. Ožujak 2018). *What is Serverless Architecture? What are its criticisms and drawbacks?* Dohvaćeno iz https://medium.com/@MarutiTech/what-is-serverless-architecture-what-are-its-criticisms-and-drawbacks-928659f9899a

Mujadžević, E. (05. Srpanj 2008). *Uvod u CSS.* Dohvaćeno iz Carnet: https://www.srce.unizg.hr/files/srce/docs/edu/osnovni-tecajevi/c220_polaznik.pdf

Narumoto, M., & Wasson, M. (24. Ožujak 2018). *Strangler pattern.* Dohvaćeno iz https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler

Nations, D. (28. Veljača 2018). *What Exactly Is a Web Application?* Dohvaćeno iz https://www.lifewire.com/what-is-a-web-application-3486637

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media.

Newman, S. (22. Ožujak 2018). *Backends For Frontends.* Dohvaćeno iz https://samnewman.io/patterns/architectural/bff/

Prednosti i nedostaci web aplikacija - Magma. (28. Veljača 2018). Dohvaćeno iz http://sr.magma.rs/blog/web-aplikacije/prednosti-i-nedostaci-web-aplikacija

Richardson, C. (22. Ožujak 2018). *Building Microservices Using an API Gateway | NGINX.* Dohvaćeno iz https://www.nginx.com/blog/building-microservices-using-an-api-gateway/

Richardson, C. (15. Kolovoz 2018). *Database per service.* Dohvaćeno iz http://microservices.io/patterns/data/database-per-service

Rouse, M. (19. Ožujak 2018). *What is Remote Procedure Call (RPC)? - Definition from WhatIs.com.* Dohvaćeno iz http://searchmicroservices.techtarget.com/definition/Remote-Procedure-Call-RPC

Rouse, M., & Nolle, T. (7. Ožujak 2018). *What is service-oriented architecture (SOA)? - Definition from WhatIs.com.* Dohvaćeno iz http://searchmicroservices.techtarget.com/definition/service-oriented-architecture-SOA

Rouse, M., Churchville, F., & Nolle, T. (7. Ožujak 2018). *What is enterprise service bus (ESB)? - Definition from WhatIs.com.* Dohvaćeno iz http://searchmicroservices.techtarget.com/definition/enterprise-service-bus-ESB

Satyasree, D. (5. Kolovoz 2018). *Ajax.* Dohvaćeno iz https://sftp.hs-furtwangen.de/~heindl/ebte-09ss/Ajax-Term-Paper.pdf

Stančer, D. (5. Kolovoz 2018). *Uvod u Javascript.* Dohvaćeno iz https://www.srce.unizg.hr/files/srce/docs/edu/osnovni-tecajevi/c501_polaznik.pdf

Versrynge, J. (13. Kolovoz 2018). *Introduction To REST Concepts.* Dohvaćeno iz https://www.javacodegeeks.com/2012/10/introduction-to-rest-concepts.html

Volarić, T. (5. Kolovoz 2018). *HTML5.* Dohvaćeno iz https://tvolaric.com/preuzimanja/HTML_5.pdf

Vrčić, D. (13. Srpanj 2018). *Uvod u Docker.* Dohvaćeno iz https://www.srce.unizg.hr/files/srce/docs/edu/edu4it/edu4it_uvod_u_docker_20161025.pdf

Wasson, M. (13. Kolovoz 2018). *Single-Page Applications.* Dohvaćeno iz Build Modern, Responsive Web Apps with ASP.NET: https://msdn.microsoft.com/en-us/magazine/dn463786.aspx

Wasson, M. (5. Ožujak 2018). *ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET.* Dohvaćeno iz https://msdn.microsoft.com/en-us/magazine/dn463786.aspx

What is Docker? (13. Srpanj 2018). Dohvaćeno iz Opensource.com: https://opensource.com/resources/what-docker

What is Kibana? (14. Srpanj 2018). Dohvaćeno iz Definition from WhatIs.com: https://searchitoperations.techtarget.com/definition/Kibana

What is MongoDB. (13. Srpanj 2018). Dohvaćeno iz Introduction to MongoDB Architecture & Features: https://intellipaat.com/blog/what-is-mongodb/

Zekić-Sušac, M. (5. Ožujak 2018). *Web aplikacije.* Dohvaćeno iz http://www.mathos.unios.hr/wp/wp2009-10/P14_Web_aplikacije.pdf

Zimine, D. (29. Srpanj 2018). *Serverless is cheaper, not simpler.* Dohvaćeno iz freeCodeCamp: https://medium.freecodecamp.org/serverless-is-cheaper-not-simpler-a10c4fc30e49

Popis slika

Popis tablica