Mikroservisi kao izazov ostalim arhitekturama

Švarc, Fran

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fran Švarc

MIKROSERVISI KAO IZAZOV OSTALIM ARHITEKTURAMA

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fran Švarc

Matični broj: 0016096647

Studij: Baze podataka i baze znanja

MIKROSERVISI KAO IZAZOV OSTALIM ARHITEKTURAMA

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, siječanj 2021.

Fran Švarc

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

---

Sažetak

Unutar ovog diplomskog rada bit će objašnjeno što je to arhitektura računalnog sustava te nekoliko tipova računalnih sustava. Osim toga bit će detaljnije pojašnjeno što je to mikroservisna arhitektura, kako se implementira te prednosti i nedostaci nad ostalim mikroservisnim arhitekturama.

Osim toga rad se sastoji i od praktičnog dijela u kojem je cilj upotrebom prethodno objašnjenog teorijskog dijela implementirati web aplikaciju koja koristi mikroservisnu arhitekturu poštujući sva pravila i najbolje prakse.

Preporuka autora je koristiti mikroservisnu arhitekturu u slučaju kada se implementira veliki sustav s puno različitih tehnologija koji će s vremenom sve više rasti te će ga biti potrebno efikasno skalirati za što veći broj korisnika. U ovom slučaju ovaj tip arhitekture je idealno rješenje. U slučaju jednostavnijih aplikacija preporučuje se korištenje jednostavnijih arhitektura poput monolitne arhitekture koja je također opisana unutar ovog rada.

Ključne riječi: mikroservis, servis, arhitektura, sustav, monolit

Sadržaj

1. Uvod

Arhitektura računalnog sustava vrlo je bitan stavka svake aplikacije te predstavlja osnovnu strukturu svakog računalnog sustava. Bez pravilne arhitekture implementacija sustava bila bi kaotična i bez pravila te održavanje takvog sustava ne bi bilo moguće.

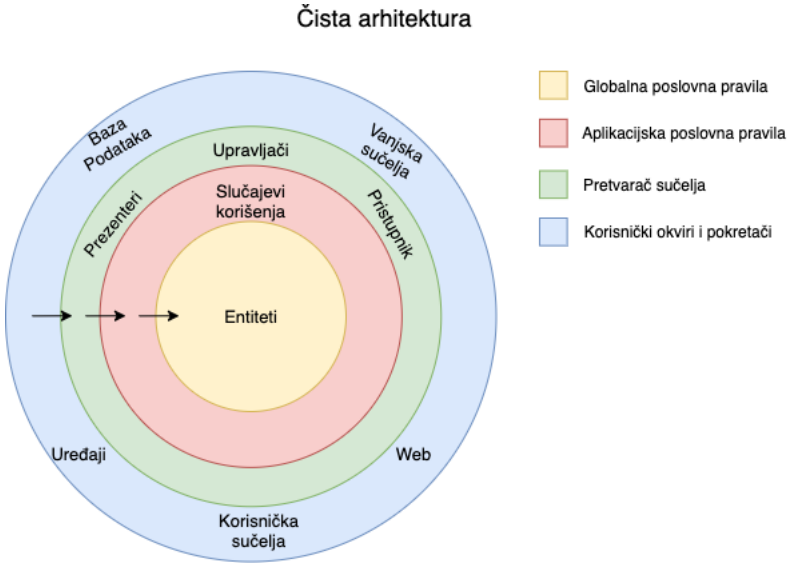
Tema ovog diplomskog rada je prije svega mikroservisna arhitektura. Ovaj tip arhitekture je nov i moderan pristup izradi računalnih sustava. Unutar ovog diplomskog rada uspoređena je mikroservisna arhitektura s drugim starijim pristupima: monolitnoj i servisnoj orijentiranoj arhitekturi. Prikazane su prednosti i nedostatke svih arhitektura kao i načini implementacije pojedinih arhitektura. Nakon toga detaljno je korak po korak opisan način na koji se implementira mikroservisni sustav te najbolje prakse prilikom njegove implementacije.

Osim teorije, drugi dio diplomskog rada sastoji se od praktičnog dijela u kojem je implementirana aplikacija upotrebom mikroservisne arhitekture. Cilj ove aplikacije je prikazati način implementacije mikroservisa te je kao krajnji rezultat dobivena aplikacija koju je vrlo lako moguće nastaviti razvijati i skalirati prema nekom većem sustavu.

2. Arhitekture računalnog sustava

Arhitektura računalnog sustava predstavlja osnovnu strukturu informacijskog sustava. Arhitekturalni uzorak predstavlja osnovne karakteristike i ponašanja implementirane aplikacije. Određeni arhitekturalni uzorci su namijenjeni za vrlo skalabilne aplikacije dok su drugi namijenjeni za agilni razvoj aplikacija. Poznavanje karakteristika određenih uzoraka te njihovih prednosti i nedostatka je vrlo bitno kako bi se na početku razvoja odabrao pravi uzorak koji najbolje odgovara poslovnim potrebama aplikacije. Upravljanje kompleksnosti glavni je posao arhitekture računalnog sustava. Cilj svake arhitekture je podijeliti sustav u određene komponente koje prikazuju određeno ponašanje. [1]

Slika 1 prikazuje jedan primjer arhitekture koju je definirao R. C. Martin i nazvao ju čista arhitektura. Prema njemu ova arhitektura se sastoji od više slojeva u kojima su unutarnji slojevi apstraktniji od vanjskih te su manje podložni promjenama. Kako bi ova arhitektura bila moguća smjer ovisnosti je usmjeren prema unutra. To znači da niti jedan unutarnji sloj ne smije biti svjestan vanjskog sloja kako promjene na vanjskom sloju ne bi utjecale na unutarnji. [2]



Slika 1 Čista arhitektura prema R. C. Martinu [2]

Komponente računalnog sustava dijele sustav u određene dijelove koje imaju uloge i zadaće. Arhitekturalni uzorak definira jasne granice podjele zadaća unutar kojih se definira kako i na koji način se dijele zadaće unutar aplikacija. Bitno je da arhitekturalni uzorak pruža veliku koheziju te nisku povezanost između komponenti. To znači da komponente trebaju

raditi samo jednu stvar kako bi imale veliku koheziju, a moraju biti minimalno ovisne jedna o drugoj. [1]

Arhitekturalni uzorak se također brine i o kvaliteti sustava odnosno treba definirati na koji način se sustav treba implementirati kako bi se zadovoljile njegove karakteristike kao što su dostupnost, povratna kompatibilnost, proširivost, pouzdanost, sigurnost, održivost.

Neki primjeri arhitekturalnih uzoraka su:

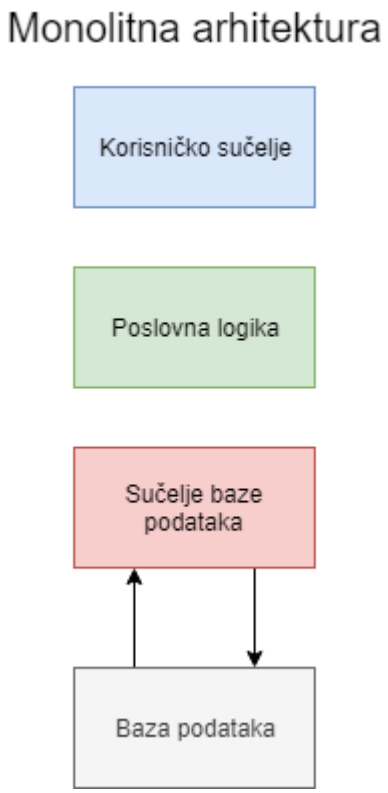
- Slojevita arhitektura – podjela sustava na slojeve unutar kojih slojevi više razine koriste funkcionalnosti slojeva niže razine [1, str 1]
- Arhitektura bazirana na događajima – popularni distribuirani i asinkroni arhitekturalni uzorak namijenjen za skalabilne aplikacije. Sastoji se od vrlo odvojenih komponenata s jednom zadaćom koji asinkrono obrađuju događaj koji im je poslan [1, str 11]
- Mikrojezgrena (en. Microkernel) arhitektura – arhitekturalni uzorak koji se koristi pri izradi nadogradivih produkata. Sustav se sastoji od jezgre sustava i priključnih modula. Jezgra sustava sadrži osnovno ponašanje koje je potrebno produktu za rad, a priključni moduli su odvojene samostalne komponente koje nadograđuju ponašanje produkta. Registracijom priključnih modula na jezgru sustava produkt postaje sve kompleksniji. [1, str 21]

Unutar ovog diplomskog rada pobliže će biti objašnjena tri popularna arhitekturalna uzorka za razvoj web aplikacija: monolitna arhitektura, servisno orijentirana arhitektura te mikroservisna arhitektura.

2.1. Monolitna arhitektura

Monolitna arhitektura je najstariji pristup razvoja sustava koji će biti opisan u ovom radu. Ovo je jedan od najjednostavniji pristup izradi sustava u kojem se sustav tretira kao jedna velika cjelina.

Monolitna arhitektura je najstariji pristup, no i dalje pruža neke prednosti kod razvoja softwarea u usporedbi s drugim modernijim pristupima. Ovaj tip razvoja je najjednostavniji budući da su sve komponente sustava na jednom mjestu i usko povezane. Osim toga u razvoju monolitne aplikacije puno je lakše brinuti o globalnim stvarima koje utječu na cijeli sustav kao što su logiranje, keširanje, praćenje performansi i sl. Također budući da je ovakav način razvoja programa uhodan, većina programera su upoznata s njim te je većina alata optimizirano za razvoj i podršku ovakvog tipa sustava. [3]



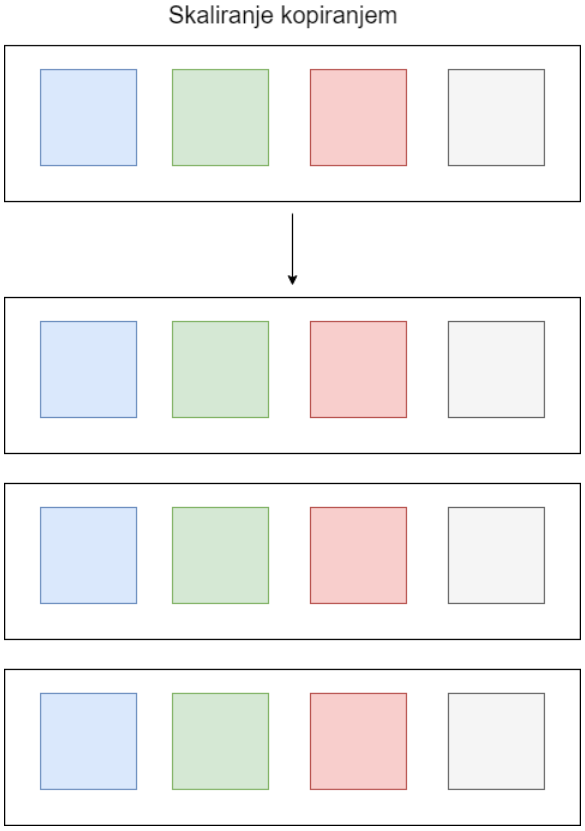
Slika 2 Monolitna arhitektura[4]

Druga prednost monolitne arhitekture je jednostavno postavljanje aplikacije na server budući da se cijela aplikacija može zapakirati u jedan paket i postaviti na jedno mjesto (npr. postavljanjem WAR datoteke). [3]

Osim toga ovakve aplikacije su lagane za testirati. Jednostavnost arhitekture omogućava i jednostavno testiranje cijelog sustava na jednom mjestu, bez potrebe za brigom o ostalim odvojenim dijelovima sustava kao što je to na primjer u mikroservisnoj arhitekturi. [3]

No, iako je ova arhitektura jednostavna i često primjenjivana u praksi, postoje mnogi nedostaci ovakvog pristupa kada se sustav skalira na veće aplikacije. To znači da konstantnim razvojem ovakvog sustava, konstantno dodajemo i kompleksnost što dovodi do teškog razumijevanja sustava te teškog održavanja i nadogradnje. Kod velikih sustava nove implementacije i promjene mogu dovesti do višestrukih promjena na različitim dijelovima aplikacije što dovodi do mogućih pojava grešaka u sustavu te duljim procesom razvoja ovakve aplikacije. [3] [5, str 7]

Skaliranje ovakve aplikacije je još jedna od mana monolitnih sustava. Iako je skaliranje jednostavno i implementira se dodavanjem dodatnih kopija sustava nad kojima se distribuira rad. Ovakav način skaliranja nije optimalan zato što se razni dijelovi sustava različito koriste. Zbog ograničenja arhitekture nije moguće skaliranje samo određenih dijelova sustava odvojeno. [3] [5, str 5-6]



Slika 3 Skaliranje monolitne arhitekture[6]

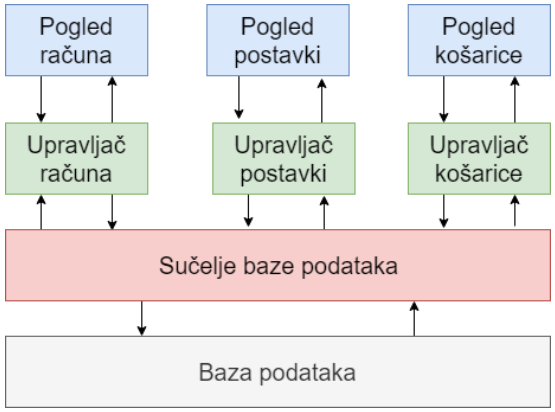
Iako je prije navedeno da je stavljanje ovakve aplikacije na server vrlo jednostavno ono sa sobom nosi i dosta negativnih strana. Postavljanje monolitne aplikacije na server sa sobom uvijek donosi gašenje aplikacije budući da se cijela aplikacija mora zamijeniti. To također znači da bez obzira na veličinu promjene ili nadogradnje cijela aplikacija ponovno mora biti prevedena te postavljena na server s jednakim vremenom nedostupnosti kao i kada se radila neka veća promjena.

Osim gore navedenih tehničkih mana monolitne arhitekture, ovakav pristup donosi i mnoge organizacijske nedostatke u razvoju. Monolitne aplikacije vrlo često zahtijevaju veliki tim za razvoj i održavanje što usporava razvoj, smanjuje agilnost i povećava broj konflikata u razvoju sustava. Isto tako, ovakvi sustavi su vrlo teški za razumjeti što otežava dovođenje novih ljudi u tim te njihovo uvođenje u projekt. Kod velikih monolitnih aplikacija programeru će trebati nekoliko mjeseci pa i godinu dana kako bi se upoznao sa svim dijelovima sustava na kojem bi mogao raditi. Osim dovođenja novih ljudi, odlazak postojećih programera s ovakvih sustava predstavlja veliki problem budući da se gubi veliko znanje na projektu a pronalazak i treniranje zamjene je dugotrajan proces. [3]

Zbog svih gore navedenih nedostataka monolitna arhitektura postaje sve manje popularna u današnje doba te se ovakav tip arhitekture pokušava zamijeniti s novijim granuliranim oblicima. Problem nastaje kod postojećih velikih sustava kod kojih je trošak i rizik takvog projekta prevelik. Zbog toga postoje mnoge tehnike pomoću kojih se aplikacija postepeno razdjeljuje na manje slojeve koji su autonomni i zasebni te se kroz višestruko koraka monolitna aplikacija postepeno pretvara u drugi tip arhitekture.

2.1.1. Primjer monolitnog sustava

Kao primjer jednog monolitnog sustava uzet ćemo jednostavan primjer web trgovine čiji dijagram je prikazan na slici ispod.



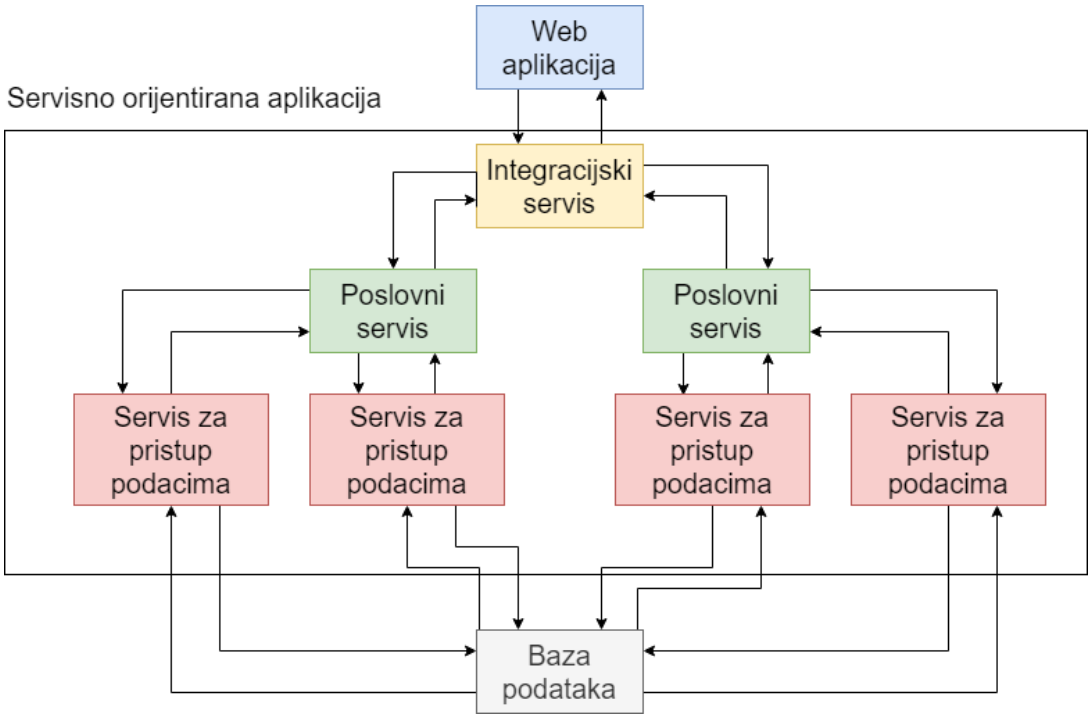
Slika 4 Primjer monolitnog sustava (autorski rad)

Na slici se vidi monolitni sustav koji koristi MVC arhitekturu u kojem su svi slojevi vrlo usko povezani. Svaki pogled ima vlastiti upravljač (en. controller) te pristupa modelu odnosno bazi podataka preko objekata za pristup bazi. Ovaj tip implementacije aplikacije je vrlo popularan te se i danas koristi u praksi no kada bi odjednom umjesto tehnologije na primjer JavaScript server pages htjeli koristit angular.js to bi zahtijevalo gotovo potpunu reimplementaciju sustava na pogledu, ali i u samim upravljačima. Zbog ovakvih ali i mnogih drugih prije navedenih nedostataka nastale su nove arhitekture sustava.

U nastavku će pobliže biti objašnjene arhitekture unutar kojih se pokušavaju riješiti nedostaci monolitne arhitekture a to su servisno orijentirana arhitektura te mikroservisna arhitektura.

2.2. Servisno orijentirana arhitektura

Servisno orijentirana arhitektura (SOA) je nastala kao odgovor nedostacima monolitne arhitekture. SOA je noviji pristup razvoju sustava koji omogućava ponovno korištenje postojećih funkcionalnosti te lakšu implementaciju promjena u sustavu. SOA je prvi korak u granulaciji sustava kako bi se riješili problemi monolitne arhitekture.[7, str 16]



Slika 5 Grafički prikaz servisno orijentirane arhitekture[8]

SOA je tip arhitekture koja je prvenstveno namijenjena izradi poslovnih aplikacija. SOA koristi komponente kao glavnu građevnu jedinicu za implementaciju određene funkcionalnosti. Svaka komponenta bi se trebala ponašati kao crna kutija unutar cijelog sustava kako bi se povećalo ponovno korištenje istih komponenti. Osim toga komponente trebaju biti što labavije povezane. Pojam labavog povezivanja označava način na koje dvije komponente rade jedna s drugom. Npr. jedna komponenta šalje određeni set podataka drugoj komponenti, a druga komponenta obrađuje dobiveni set podataka. Kod povezanosti ove dvije komponente bitna je njihova jednostavnost i autonomija kako bi se na primjer prva komponenta mogla zamijeniti s nekom trećom bez potrebe za velikim promjenama u sustavu. Ovo predstavlja veliku razliku od monolitne arhitekture gdje su jedinice puno uže povezane. Upotrebom jednostavnih

komponenti i njihovim povezivanjem može se kreirati kompleksna logika koja predstavlja određen poslovni servis. [7, str 27]

SOA arhitektura pruža mnoge prednosti nad drugim arhitekturama zbog svojeg granuliranog dizajna i distributivnosti.

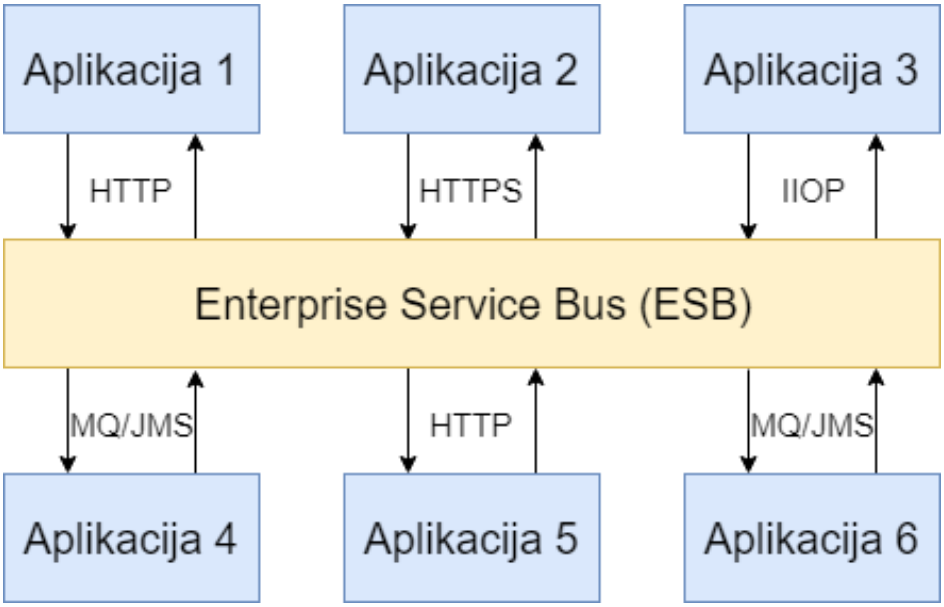
Granulirani dizajn unutar servisno orijentirano arhitekture pruža mnoge prednosti. On omogućava ponovno korištenje servisa za različite svrhe budući da bi servisi prema arhitekturi trebali biti mali labavo povezani dijelovi funkcionalnosti koje je zbog toga vrlo lako ponovno koristiti. Mali servisi također donose prednost kod održavanja sustava. Budući da su funkcionalnosti podijeljene na manje jedinice, njihovo održavanje je puno jednostavnije te promjena jednog servisa ne utječe na funkcionalnost drugog servisa. Zbog toga implementacija određenih promjena u sustav zahtjeva puno manje vremena od na primjer monolitnih sustava. Osim toga manje jedinice su lakše za testiranje te puno pouzdanije budući da su manje funkcionalnosti lakše za implementaciju, testiranje i popravljanje. Također manje jedinice poboljšavaju kvalitetu koda, produktivnost te samim time smanjuju cijenu razvoja ovakvih sustava.[7, str 27-29][9]

Kad se govori o distributivnim prednostima ove arhitekture onda se govori o prednostima kao što su nezavisnost lokacije servisa, poboljšanjima u skaliranju i dostupnosti. Nezavisnost lokacije omogućava servisima da mijenja svoju lokaciju bez da to utječe na rad sustava budući da će korisnik tog servisa uvijek moći pronaći traženi servis. Skalabilnost i dostupnost je također poboljšana u usporedbi s monolitnim sustavima zato što servisi mogu raditi na različitim serverima te skalirati ovisno o potrebama. Također pad jednog servisa ne mora značiti pad cijelog sustava. [7, str 27-29] [9]

Iako ova arhitektura pruža mnoge prednosti nad monolitnom i drugim arhitekturama, također s ovim dizajnom dolaze i neke mane. Prva mana ove arhitekture je početna cijena implementacije ovakvog sustava. SOA arhitektura zahtjeva programere koji su upoznati s ovakvim načinom izrade sustava, osim toga tehnologije i potrebno vrijeme za izradu ovakvog sustava su skuplje od monolitnog sustava budući da su monolitni sustavi vrlo poznata i stara arhitektura.

SOA arhitektura ima sporije performanse zbog svog distribuiranog pristupa zato što komunikacija između odvojenih servisa dodaje određeno vrijeme za obradu zahtjeva korisnika. [9]

No, glavna mana ovakvog sustava je u konačnici potreba za kompleksnim upravljanjem raznim servisima. Kako ova arhitektura raste, tako i upravljanje sa servisima i njihovim komuniciranjem postaje sve kompleksnije. Zbog toga se najčešće unutar ove arhitekture uvodi sloj za upravljanjem porukama unutar sustava – ESB. ESB (enterprise service bus) je sloj čiji je glavni zadatak upravljanje s porukama između komponenta. Način na koji ESB radi je sljedeći. Komponenta X se spaja na ESB i šalje poruku koristeći specificirani format zajedno s adresama komponenti koje tu poruku moraju dobiti. ESB zatim koristi te adrese te šalje poruku na sve komponente koje tu poruku moraju dobiti. Ovaj sloj s rastom sustava vrlo brzo postaje iznimno kompleksan. ESB je moguće implementirati samostalno, no u pravilu se koriste gotova rješenja nekih od velikih tvrtki kao što su Oracle ili IBM. Kompleksnost i upravljanje ESB-om je jedna od glavnih razloga zbog kojeg su se ljudi okrenuli još granuliranijem tipu sustava – mikroservisima. [7, str 46-48] [9]

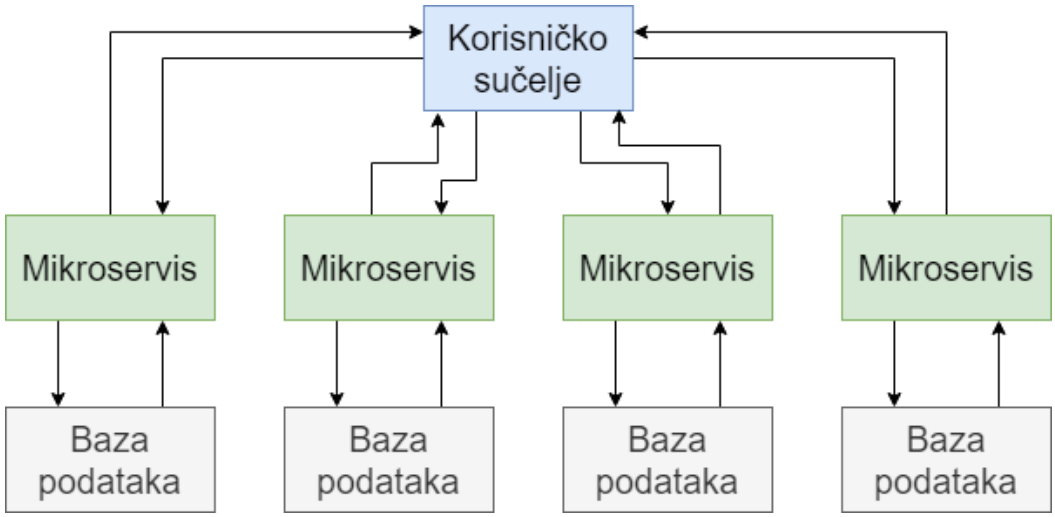


Slika 6 Grafički prikaz ESB-a[10]

2.3. Mikroservisna arhitektura

Mikroservisna arhitektura je noviji tip arhitekture koji je postao popularan u posljednjih nekoliko godina. Mikroservisna arhitektura je kao što joj i samo ime govori, arhitektura bazirana na mikroservisima. Mikroservisi su mali autonomni servisi koji rade zajedno kako bi izvršili određenu funkcionalnost. Unutar mikroservisne arhitekture, kohezija koda postiže se praćenjem principa Robert C. Martina „Single Responsibility Principle“ prema kojem se grupiraju funkcije koje se moraju mijenjati iz istog razloga, a razdvajaju se one koje se mijenjaju iz različitih. To znači da mikroservisi moraju biti vrlo male cjeline čije promjene ne zahtijevaju puno vremena.[5, str 2]

Kada kažemo da su mikroservisi autonomni to znači da je mikroservis odvojeni entitet koji se može izolirano postaviti na odvojenu platformu te će ona kao takva moći raditi s ostatkom sustava. U načelu, unutar ovakve arhitekture izbjegava se pakiranje više mikroservisa zajedno. Svi mikroservisi ove arhitekture komuniciraju preko mreže i API poziva koje svaki mikroservis definira za sebe. Ovakav način implementacije kreira potpuno odvojene servise koji se mogu mijenjati neovisno jedan o drugome. Zlatno pravilo ove implementacije je pitanje „Može li se napraviti promjena na servisu te postaviti na server bez da se išta drugo mijenja“. Ako je odgovor da onda je mikroservisna arhitektura pravilno implementirana, dok u slučaju odgovora ne, može se vidjeti manjkavosti u implementaciji koje utječu na glavne prednosti ovakve arhitekture. [5, str 3]

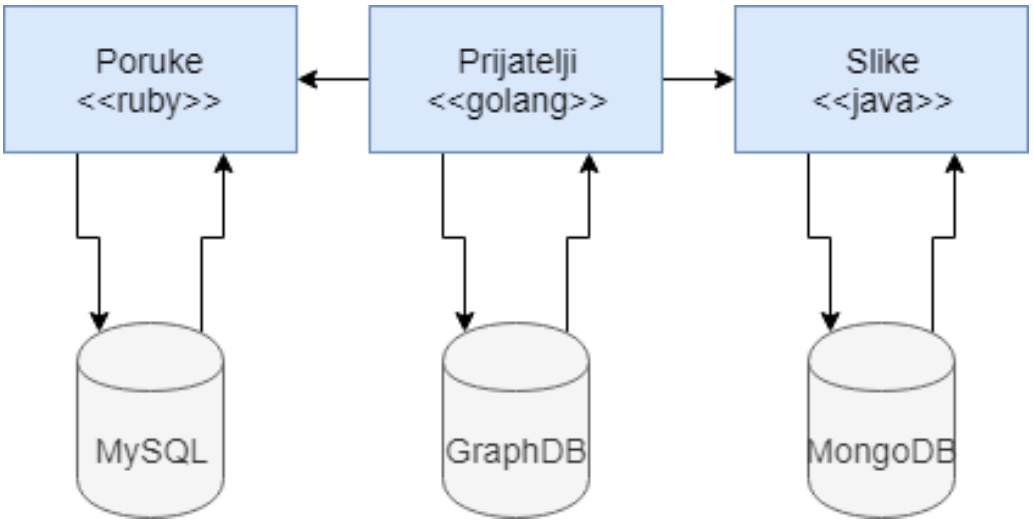


Slika 7 Grafički prikaz mikroservisne arhitekture [11]

2.3.1. Ključne prednosti mikroservisne arhitekture

Glavne prednosti mikroservisna arhitektura dijeli s prednostima ostalih distribuiranih sustava. No, mikroservisna arhitektura te prednosti ima na višoj razini budući da se i sam koncept distribuiranosti u ovoj arhitekturi implementira na višoj razini. [5, str 4]

Prva prednost ove arhitekture je heterogenost tehnologija. Odvojenost i izoliranost mikroservisa u mirkoservisnoj arhitekturi omogućava upotrebu različitih tehnologija za implementaciju servisa bez utjecanja na cijeli sustav. Ovo omogućuje upotrebu najboljih tehnologija za zadatak koji je potrebno odraditi. Na primjer, jedan servis može biti implementiran u jeziku Java te ga se može povezivati sa SQL bazom podataka, dok drugi servis može biti implementiran u JavaScriptu i node.js-u te koristiti noSQL bazu podataka. Ovakva fleksibilnost omogućava vrlo brze promjene i praćenje novih tehnologije. Taj tip implementacija nije moguća na monolitnim aplikacijama zato što bi jedna promjena utjecala na cjelokupni sustav te bi zahtijevala mnoge promjene u implementaciji. Iako je ovakva fleksibilnost dobra, neke kompanije ipak imaju restrikcije nad tehnologijama budući da različite tehnologije zahtijevaju različite alate te različite eksperte koji mogu raditi na takvim sustavima. Tako na primjer Netflix u svojoj implementaciji većinom koristi Javu i JVM budući da imaju najbolje razumijevanje takvih sustava te alate za rad s tim sustavima. [5, str 4-5]

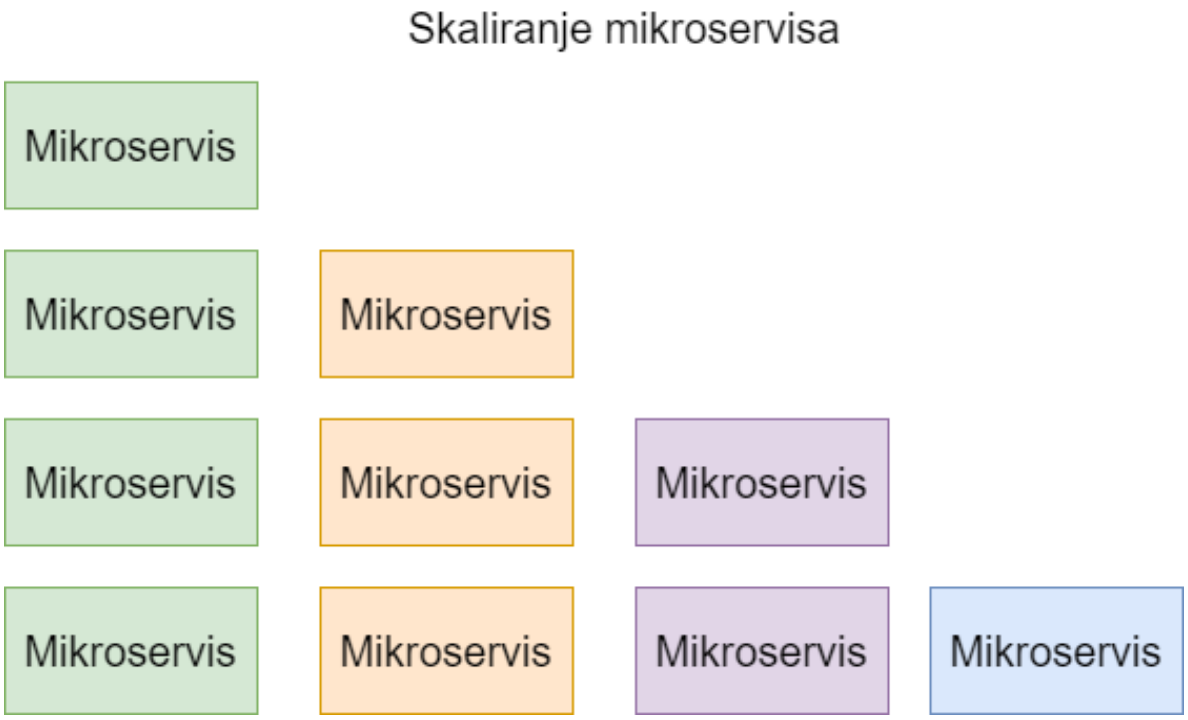


Slika 8 Heterogenost mikroservisne arhitekture [5, str 4]

Otpornost je druga prednost mikroservisne arhitekture. Za razliku od monolitne arhitekture, mikroservisna arhitektura bi trebala moći podnijeti pad određenog mikroservisa bez da ta greška u potpunosti sruši sustav. Na primjer ako na stranici postoji servis za pretraživanje

te servis za gledanje videa, u slučaju pada servisa za pretraživanje, stranica bi i dalje bila dostupna te bi omogućavala korisniku pregled videa dok bi samo pretraga sadržaja bila onemogućena. [5, str 5]

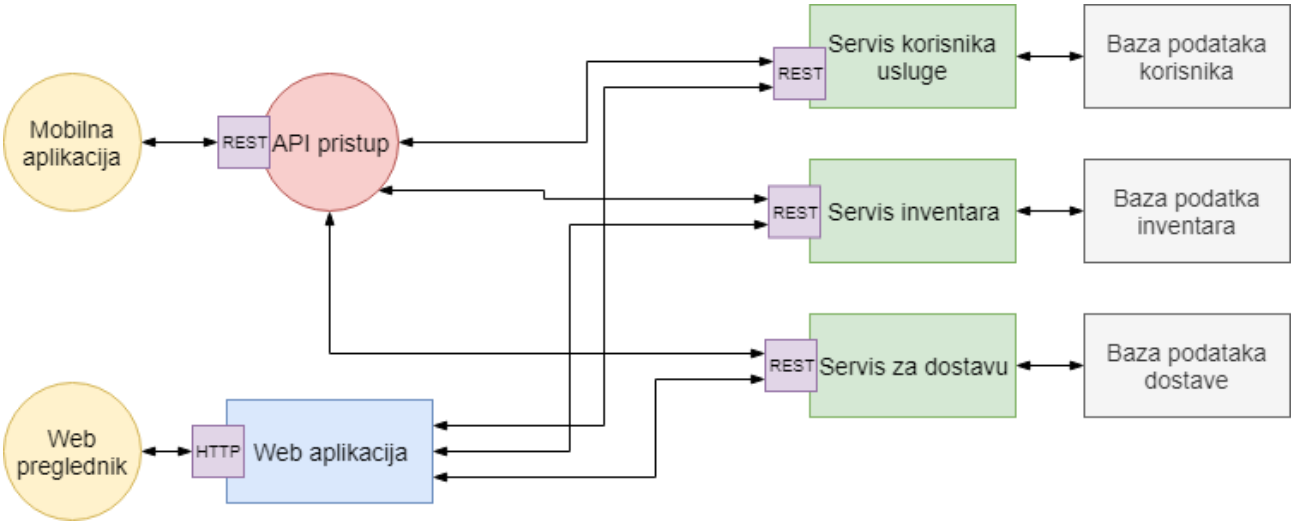
Skaliranje aplikacije je još jedna prednost ove arhitekture. Dok stare monolitne aplikacije zahtijevaju skaliranje cijele aplikacije bez obzira na to koji se dio najviše koristi, u mikroservisnoj arhitekturi može se ciljano skalirati dio aplikacije koji to zahtjeva. Tako na primjer na stranici koja pruža mogućnost gledanja videa, može biti skaliran samo sustav za distribuciju videa kako bi povećali performanse, dok na primjer sustav za pretragu videa može biti ostavljen na istoj razini budući da taj dio sustava nije pod tolikim opterećenjem. [5, str 5-6]



Slika 9 Primjer skaliranja mikroservisa[6]

Lakoća postavljanje aplikacija na server također se očituje u nezavisnosti servisa. Dok u starim monolitnim aplikacijama promjena jednog dijela koda zahtjeva prevođenje čitavog koda aplikacije, u mikroservisnoj arhitekturi prevođenje se radi samo na servisu zahvaćenom promjenom. Postavljanje velike aplikacije na server može biti rizičan te vremenski dug posao koji zahtjeva gašenje sustava na određeno vrijeme. U mikroservisnoj arhitekturi vrijeme postavljanja aplikacije je kratko budući da aplikaciju može biti postavljena na servere servis po servis bez potrebe za dugotrajnim gašenjem sustava. [5, str 6-7]

Kompozitnost sustava je ključna stavka arhitekture budući da pruža mnogo prilika za ponovnim korištenjem funkcionalnosti. Mikroservisima omogućuju aplikacijama da upotrebom API-a konzumiraju sadržaj na razne načine. To znači da se sustav može napraviti takvim da sadržaj mikroservisa mogu konzumirati aplikacije za računala, mobitele, ali i same web aplikacije.[5, str 7]



Slika 10 Grafički prikaz kompozitnosti mikroservisne arhitekture[12]

Osim gore tehničkih prednosti mikroservisne arhitekture, ona također donosi i prednosti u organizacijskom smislu. Mikroservisna arhitektura omogućava podjelu održavanja i implementiranja mikroservisa na manje i agilne timove što smanjuje vrijeme potrebno za organizaciju timove ta smanjuje konflikte u implementaciji. Također omogućuje timovima da budu autonomni jedan od drugih te da sami odlučuju o tehnologijama koje će koristiti. Osim toga mikroservisna arhitektura je optimizirana za mijenjanje. Mikroservisi prema pravili nisu veliki i direktno ne utječu na ostale dijelove sustava zbog čega se smanjuju situacije sa starim velikim dijelom sustava koji nitko ne želi dirati budući da utječe na mnoge dijelova aplikacije i čije refaktoriranje ili reimplementacija proizvodi veliki rizik. [5, str 7]

2.3.2. Nedostaci mikroservisne arhitekture

Iako mikroservisna arhitektura pruža mnoge prednosti nad ostalim arhitekturama, ona sa sobom donosi i neke nedostatke i teškoće kod razvoja sustava. Ova arhitektura predstavlja relativno nov način razvoj sustava što znači da svi alati za razvoj još ne podržavaju ovaj način razvoja te da većina programera još nije upoznata s implementacijom ovakvih sustava.

Mikroservisni sustavi su arhitektonski puno složeniji od klasičnih monolitnih sustava što usporava njihovu implementaciju. Distribuiranost sustava može dovesti do češćih sigurnosnih propusta zbog čega je implementacija sigurnosti zahtjevnija. Također, distribuiranost dovodi do lošijih performansi zbog latencije u komunikaciji putem mreža i poruka.

Kompleksnost mikroservisne arhitekture također otežava praćenje performansi cjelokupnog sustava, logiranje, keširanje te njegovo testiranje. [5, str 11]

2.4. Usporedba arhitektura sustava

U prethodnim poglavljima prikazane su tri tipa arhitektura sustava: monolitna arhitektura, servisno orijentirana arhitekture i mikroservisna arhitektura. Unutar ovog poglavlja bit će uspoređene ove arhitekture u jednostavnosti implementiranja, skaliranju, stabilnosti, performansama.

Implementacija monolitne arhitekture je najjednostavnija u usporedbi sa servisno orijentiranom arhitekturom i mikroservisnom arhitekturom. Ona je najjednostavnija zbog svoje povijesti, odnosno zbog toga što postoji velika podrška alata za razvoj ovakvih sustava, velikih broj referenci prema kojima se sustav može implementirati te edukacijom programera koji su upoznati sa starijim tipovima arhitektura. Servisno orijentirana arhitektura i mikroservisna arhitektura su kompliciranije za implementaciju zbog toga što su relativno novi tip implementacije sustava. Mikroservisna arhitektura i servisno orijentirana arhitektura donose slične izazove u implementaciji, no zbog svoje popularnosti mikroservisna arhitektura je jednostavnija za implementaciju zbog velikog broja novih alata koji pripomažu u implementaciji te broja referenci implementacija (Netflix, Twitter, Amazone) prema kojima se može implementirati novi sustav.

Kada se govori o skaliranju ove tri arhitekture vrlo lako je zaključiti da se distribuirana arhitektura kao što je mikroservisna puno bolje skalira od monolitne arhitekture te nešto bolje od servisno orijentirane arhitekture. Mikroservisna arhitektura omogućava skaliranje samo onih servisa koji su opterećeni u sustavu, dok monolitna arhitektura zahtjeva skaliranje cjelokupnog sustava. Mikroservisna arhitektura je također skalabilnija od servisno orijentirane arhitekture budući da ne zahtijeva sloj za upravljanjem porukama koji može postati iznimno kompliciran. Osim skaliranja u performansnom smislu, servisno orijentirana arhitektura se najbolje skalira i u smislu kompleksnosti. Ovaj tip arhitekture dijeli sustav na manje odvojene cjeline koje je lakše održavati i nadograđivati. Povećanjem kompleksnosti u mikroservisnoj arhitekturi puno je lakše podijeliti servise koji postanu preveliki na više dijelova. Također ova neovisnost

omogućava upotrebu najbolje tehnologije za određen servis što može smanjiti kompleksnost implementacije budući da nova implementacija ne ovisi o tehnologijama odabranim na početku implementacije sustava. Monolitna arhitektura povećanjem kompleksnosti postaje sve veća i veća te težom za održavanje s usko povezanim dijelovima sustava gdje jedna promjena u jednom dijelu sustava može uzrokovati promjene na cijelom sustavu. Servisno orijentirana arhitektura se bolje skalira s kompleksnosti od monolitne arhitekture budući da su dijelovi implementacije odvojene u zasebne komponente koje komuniciraju putem mreže, no unatoč tome sustav za razmjenu poruka postaje vrlo težak za održavanje. Upravo je kompleksni ESB razlog zbog kojeg je iz ovakvog tipa arhitekture nastala mikroservisna arhitektura koji ovaj problem nema.

Performanse sustava su također vrlo bitna stavka sustava. U ovom pogledu monolitna arhitektura je brže od servisno orijentirane arhitekture i mikroservisne arhitekture. Razlog brzine monolitne arhitekture je upravo u tome što nije distribuirana. To znači da se u monolitnoj arhitekturi ne gubi vrijeme na komunikaciju servisa putem mreže već se sve nalazi na jednom mjestu i može direktno komunicirati. Zbog toga se i danas monolitni tip arhitekture koristi u sustavima u kojima je svaka ušteđena milisekunda bitna. Razlika u performansama između servisno orijentirane arhitekture i mikroservisne arhitekture gotovo da i nema budući da obje arhitekture ovise o mreži za komunikaciju između servisa.

Pouzdanost mikroservisnih sustava također je jedna od prednosti ovog tipa arhitekture nad monolitnom arhitekturom. Budući da je sustav distribuiran pad jednog servisa ne mora značiti pad cijelog sustava. Mikroservisna arhitektura izbjegava jednu točku greške koja uzrokuje pad sustava upotrebom labave povezanosti te distributivnosti baza podataka i servisa. Za razliku od nje monolitna arhitektura podložnija je padu cjelokupnog sustava budući da pad servera uzrokuje pad cjelokupnog sustava. U monolitnoj arhitekturi pad servera izbjegava se višestrukim kopijama aplikacije na više servera no to i dalje ne sprječava pad sustava u slučaju greške u aplikaciji do koje može vrlo lako doći zbog uske povezanosti monolitnih sustava. Servisno orijentirana arhitektura je zbog svoje odvojenosti također pouzdanije od monolitne arhitekture no u usporedbi sa mikroservisima ipak je manje pouzdana budući da komunikacija ovisi o kompleksnom ESB-u.

Mikroservisna arhitektura predstavlja novi korak u implementaciji velikih sustava. Mikroservisna arhitektura pruža prednosti kao što su skalabilnost, pouzdanost, labava povezanost sustava i kohezija sustava. Ovaj tip arhitekture je kompleksniji za implementaciju, praćenje rada i testiranje no u konačnici prednosti koja ova arhitektura nudi nad monolitnom

arhitekturom i servisnom orijentiranom arhitekturom nadjačavaju mane kada su u pitanju veliki sustavi čija će kompleksnost s vremenom rasti te je to upravo jedan od razloga zbog kojeg velike tvrtke poput Netflixa i Twittera prelaze na ovaj tip sustava. U tablici ispod može se vidjeti sažeti prikaz prednosti i nedostataka pojedinih arhitektura.

Tablica 1 Prikaz prednosti i nedostataka pojedinih arhitektura

	Monolitna arhitektura	Servisno orijentirana arhitektura	Mikroservisna arhitektura
Kompleksnost implementacije	Najjednostavnija	Najkompleksnija	Srednje kompleksna
Skaliranje	Najlošije skaliranje	Srednje skaliranje	Najbolje skaliranje
Performanse	Najbolje performanse	Srednje performanse (ovisne o mreži)	Srednje performanse (ovisne o mreži)
Pouzdanost	Najmanje pouzdana	Srednje pouzdana	Najpouzdanija

3. Implementacija mikroservisne arhitekture

Implementacija mikroservisnog sustava je vrlo složen proces koji zahtjeva dobru pripremu. Osim poznavanja mikroservisne arhitekture te najboljih praksa za implementaciju također je potrebno poznavati domenu za koju se sustav radi.

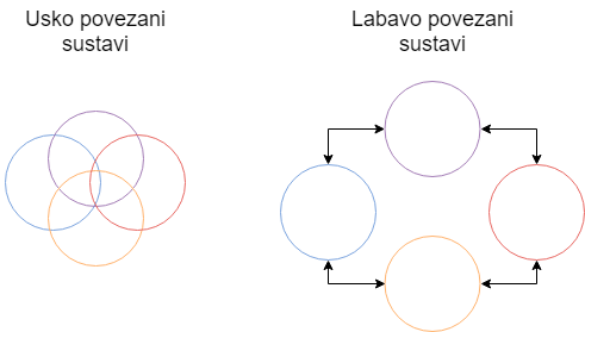
Unutar ovog poglavlja bit će objašnjena tri koraka kod implementacije mikroservisnog sustava. U prvom djelu bit će objašnjeno na koji način se modelira mikroservisni sustava, te na koje stvari je potrebno paziti. U drugom poglavlju prikazat će se razni načini integracije mikroservisa te najbolje prakse kod te integracije. U trećem i zadnjem poglavlju bit će opisani razni načini testiranja sustava općenito te određene stvari na koje je potrebno pripaziti kod testiranje mikroservisnog sustava.

3.1. Modeliranje sustava

Unutar ovog poglavlja bit će pobliže objašnjeno kako modelirati sustav baziran na mikroservisima. Objasnit će se kako odrediti granice između mikroservisa i kako modelirati sustav kako bi se dobila najveća korist od ovog tipa arhitekture.

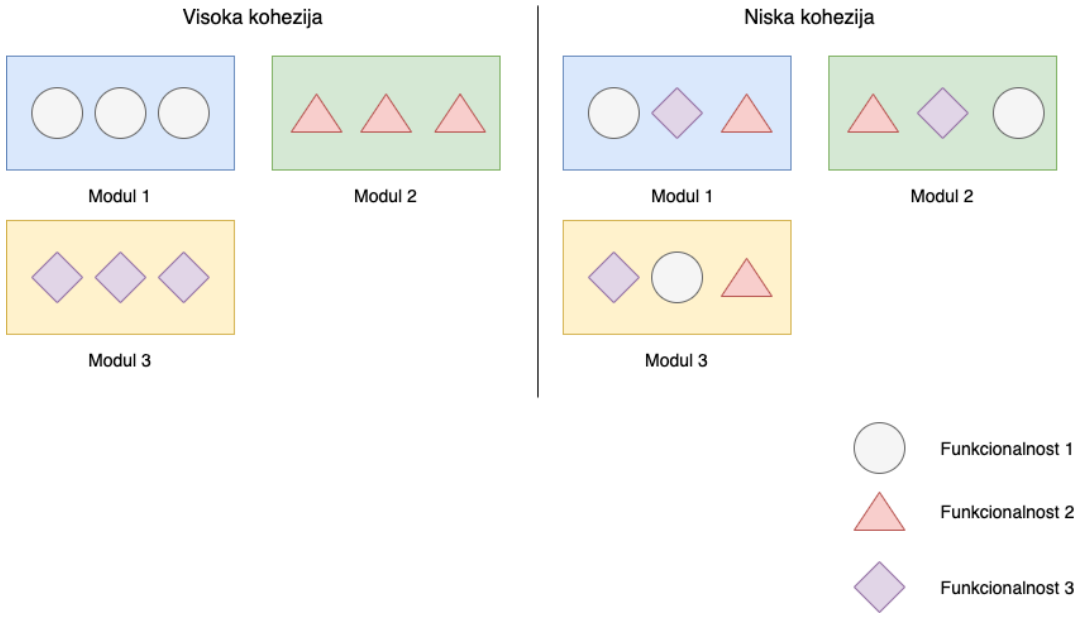
3.1.1. Definiranje granica

Dvije ključne karakteristike koje treba uzeti u obzir kod definiranja granica između mikroservisa su labava povezanost i visoka kohezija. Labava povezanost je svojstvo koje ako servis posjeduje označava da se određeni servis ne mora mijenjati ako se mijenjao drugi servis koji radi s njim. Labavo povezan servis zna samo onoliko koliko treba o drugom servisu i ništa više, također labavo povezan servis komunicira s drugim servisom u što manje navrata budući da česta komunikacija između servisa dovodi do uske povezanosti.



Slika 11 Simbolički prikaz usko i labavo povezanog sustava

Druga karakteristika je visoka kohezija. Ova karakteristika sustava podrazumijeva da određena funkcionalnost treba biti implementirana na jednom mjestu, dok druge funkcionalnosti trebaju biti odvojene i implementirane na drugom mjestu. Ovakav način implementacije omogućava da promjena u određenoj funkcionalnosti mijenja sustav samo na jednom mjestu. Simbolički prikaz kohezije vidljiv je na slici 12.



Slika 12 Simbolički prikaz kohezije

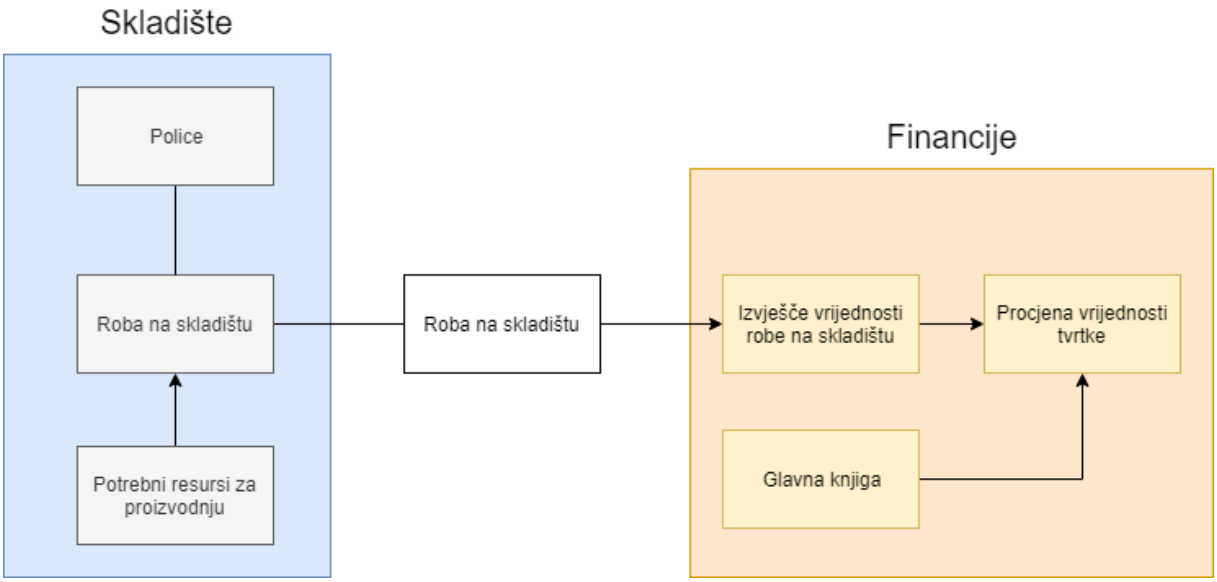
U kontekstu mikroservisa ovo svojstvo omogućava mikroservisima da u slučaju određene promjene, samo jedan servis mora biti ponovno preveden i postavljen na server bez da to utječe na druge servise. [5, str 30]

U kratko kod definiranja granica između mikroservisa cilj je pronaći granicu koja definira sustav tako da je jedna funkcionalnost definirana na jednom mjestu, a da su te funkcionalnosti povezane na najlabaviji mogući način.

3.1.2. Modeliranje konteksta

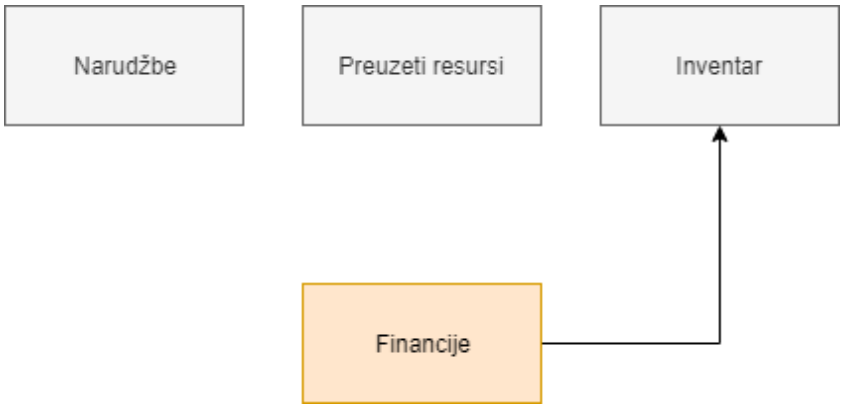
Kod modeliranja mikroservisnog sustava modeliraju se domene sustava iz pravog života. Bitan koncept za mikroservise koji Eric Evans uvodi u knjizi Domain Driven Design[13] je koncept ograničenog konteksta. Ideja ovog koncepta je da se svaka domena sastoji od jednog ili više ograničena koncepta koji se nalaze jedan pored drugog. Svaki kontekst sadrži informacije s kojima komunicira unutar ograničenog konteksta te informacije koje su

eksternalizirane te ih mogu koristiti drugi konteksti. Svaki kontekst ima svoju sučelje u kojem definira koje podatke dijeli s ostalim ograničenim kontekstima. Prema Samu Newmanu jedna od definicija ograničenog konteksta bi mogla biti: „Specifična odgovornost potaknuta eksplicitnim granicama.“ [5]



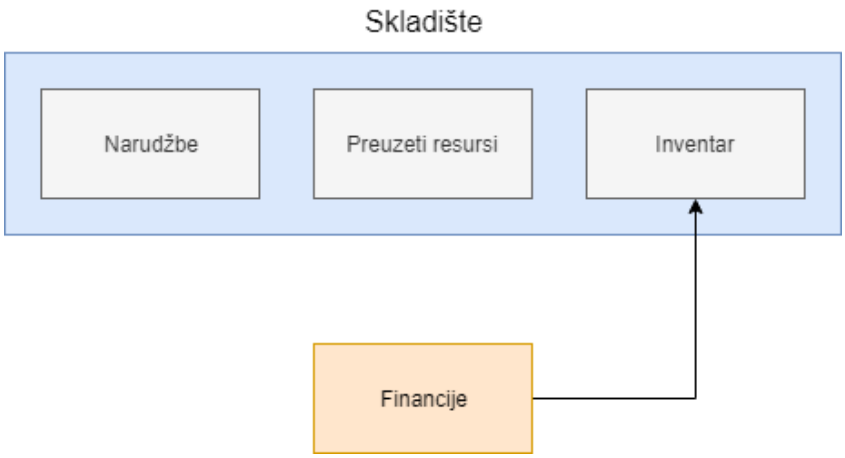
Slika 13 Prikaz dva ograničena konteksta te dodirne točke između njih[5]

Kod modeliranja ovih konteksta bitno je prepoznati domene iz pravog života te definirati dodirne točke između domena kako bi se mogli definirati konteksti te granice između njih (slika 13). Na primjer određena tvrtka X se među ostalim sastoji od skladišta i financija. Iako model skladišta za robu mora pratiti mnoge podatke, na primjer u kojem se skladištu nalazi, kada je roba napravljena, koliko robe ima i slično. Ti svi podaci nisu bitni za financije već tu domenu zanima trenutno stanje kako bi mogla voditi izvještaj o na primjer trenutnoj vrijednosti robe na skladištu. Unutar ovog primjera vidi se da su određene informacije interne za skladište dok se pak druge trebaju eksternalizirati drugim domenama kako bi mogle raditi svoj posao. Također iako se većina informacija nalazi na modelu robe, ne trebaju se dijeliti svi podaci s vanjskim servisima već se taj model podataka može pretvoriti u drugi ograničenije tip sa samo potrebnim eksternim informacijama. S pravilnim definiranjem i ne dijeljenjem internih informacija izbjegavamo usku povezanost sustava koja bi postala velika boljka ovakvog sustava. Primjer ograničenog konteksta s visokom granulacijom te labavom povezanosti prikazan je na slici 13.



Slika 14 Prikaz ograničenog konteksta s većom granulacijom[5]

Na početku modeliranja ovakvih sustava prvo je najbolje napraviti krupniju granulacija ograničenih konteksta koji unutar sebe mogu sadržavati dodatne ograničene kontekste. Na primjer financije mogu sadržavati ograničene kontekste koje se bave prodajom i nabavom. Ne postoji pravilo prema kojem se manji konteksti trebaju dodatno razlagati. Ovakve odluke ponajviše ovise o organizacijskoj strukturi tvrtke za koju se sustav modelira. Ugniježđeni pristup prikazan na slici 15 će dovesti do većih mikroservisa koji smanjuje prednosti te arhitekture no donosi i prednosti kao što su lakše testiranje većih dijelova sustava.



Slika 15 Prikaz ugniježđenih ograničenih konteksta unutar konteksta skladište[5]

Najveća zamka kod definiranja ograničenih konteksta je prerana dekompozicija. Problem prerane dekompozicije nastaje kada se domena sustava podijele na male cjeline bez potpunog poznavanje domene. Česta pojava koja se događa je da se tijekom izgradnje sustava, sustavi logično podijele na ograničene kontekste prepoznajući dodirne točke između koncepata. No vrlo često ono što izgleda logično ispadne krivo budući da posebnosti unutar sustava mogu značajno promijeniti na koji način sustav funkcionira. Zbog toga se na primjer sustav koji je

početno podijeljen na pet ograničena konteksta mora spojiti u četiri što dovodi do dodatnih troškova i utroška vremena. [5, str 31]

Unutar ovog poglavlja opisano je na koje temeljne koncepte treba obratiti pažnju prilikom modeliranja mikroservisne arhitekture za određenu domenu iz pravog života. U idućem poglavlju bit će opisana integracija sustava te način na koji se koncepti iz ovog poglavlja implementiraju u mikroservisnom sustavu.

3.2. Integracija sustava

Nakon što je napravljen model sustava koji je potrebno implementirati slijedi njegova implementacija te integracija svih servisa kako bi sustav radio zadaću za koju je namijenjen. Integracija mikroservisa je najbitniji dio implementacije mikroservisnog sustava. Ako je integracija loša izgubit će se većina prednosti koju ova arhitektura pruža. Unutar ovog poglavlja bit će objašnjeni neki glavni pristupi kod implementacije i integracije sustava baziranog na mikroservisnoj arhitekturi.

3.2.1. Glavni principi integracije sustava

Prema Newmanu [5, str 39-40] kada se govori o integraciji mikroservisnih sustava postoji nekoliko glavnih principa o kojima je bitno razmišljati prilikom integracije.

Prvi princip je govori da je potrebno izbjegavati promjene koje mogu pokvariti sustav. Kod implementacije mikroservisnog sustava postoje trenutci kada se trebaju promijeniti podaci koje konzumiraju drugi servisi. Zbog toga potrebno je odabrati tehnologiju koja će nam omogućiti da promjene na servisima koje konzumiraju sadržaj moraju biti što rjeđe moguće. To na primjer znači da ako dodamo novo polje u podatak koji šaljemo, to neće imati utjecaj na servis koji taj podatak konzumira.

Drugi princip je govori da API mora biti što neovisniji o tehnologiji. Kao što je već u prethodnim poglavljima rečeno, jedna od glavnih prednosti mikroservisne arhitekture je ta da za svaki mikroservis može biti odabrana najbolja tehnologija moguća bez utjecanja na druge mikroservise. Budući da se u današnje vrijeme tehnologije mijenjaju svakodnevno, potrebno je napraviti API sustav za komunikaciju takav da je neovisan o tehnologijama kako bi što više opcija bilo otvoreno u budućnosti. To znači da se trebaju izbjegavati integracijske tehnologije koje diktiraju koje tehnologije se moraju koristiti u implementaciji mikroservisa.

Treći princip kod integracije govori da servisi trebaju biti što jednostavniji za korisnike tog servisa. Ovaj princip je jednostavan te govori da servis treba biti takav da ga njegov korisnik

može lagano integrirati te koristiti u svojoj aplikaciji. U idealnoj situaciji to znači da korisnik ima potpunu slobodu o tehnologijama za integraciju s mikroservisom koji pruža određeno funkcionalnost.

Četvrti i zadnji princip koji treba slijediti govori da detalji unutarnje implementacije trebaju biti skriveni. Ovaj princip govori da svi detalji implementacije servisa trebaju biti sakriveni od korisnika tog servisa. Svu unutarnju implementaciju treba biti sakrivena kako bi se smanjila povezanost između servisa te samim time smanjile situacije gdje promjene u unutarnjoj implementaciji utječu na korisnika servisa.

3.2.2. Dijeljena baze podataka

Integracija s dijeljenom bazom podataka je jedan od najpopularnijih najjednostavnijih načina integracija sustava. Zbog svoje jednostavnosti ovaj tip integracije je i najbrži za implementirati. Princip ove integracije je vrlo jednostavan. Ako određen servis treba neki podataka onda može pristupiti bazi podataka i pronaći određen podatak. Ako servis treba zapisati neki podatak onda također može pristupiti bazi podataka i zapisati taj podatak. Primjer jednog takvog sustava je prikazan na slici 16.

Slika 16 Prikaz integracije s dijeljenom bazom podataka[5]

Na slici 16 prikazan je sustav u kojem različiti servisi direktno pristupaju istoj bazi podataka kada im je potrebna neki podatak. Na primjer registracijski servis kreira korisnika u bazi podataka, skladišni servis editira podatke o narudžbi korisnika u bazi dok dio za podršku čita podatke o korisniku iz baze. Ovo je vrlo uobičajen način integracije no dovodi do mnogih teškoća u implementaciji.

Prvi problem koji se javlja u ovoj implementaciji je izrazita povezanost svih servisa. To dovodi do vrlo uske povezanosti interne implementacije budući da svi servisi rade s jednakom shemom baze podataka. Prilikom nadogradnja ovakvih servisa promjene koje utječu na

korisnike servisa su vrlo česte. Ako je na primjer potrebno prilagoditi shemu baze podataka da bolje odgovara skladištu, potrebno je paziti da promjene ne utječu na druge sustave ili je potrebno također vršiti promjene na drugim sustavima.

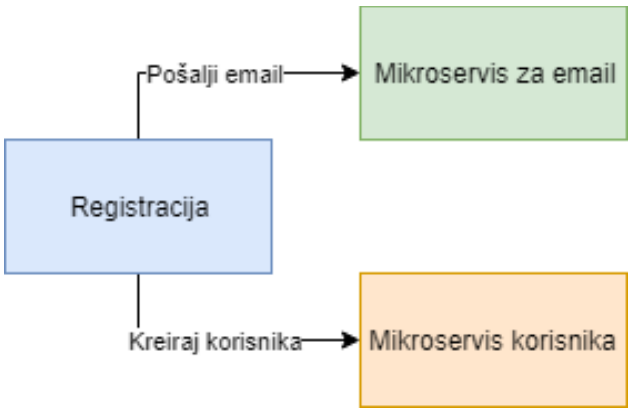
Drugi problem kod ovakve implementacije je vrlo uska povezanost sustava s tehnologijom. U ovakvom sustavu promjena tehnologije baze podataka na, na primjer, noSQL tehnologiju za određeni servis nije moguća. Iako je ta tehnologija bolja za dani servis, promjenu nije moguće napraviti izolirano već će ona utjecati na sve servise koji koriste bazu podataka. Ovo je još jedna mana uske povezanosti servisa.

Treći problem ovakvog sustava je izrazito loša kohezija. Unutar ovakvog sustava svi servisi obavljaju određene operacije s korisnikom u bazi podataka. To znači da se logika za korisnika nalazi na više mjesta odjednom. Zbog toga određena promjena na bazi podataka će zahtijevati promjenu u logici na svim servisima koji tu bazu podataka koriste.

Glavna prednost mikroservisne arhitekture je velika kohezija te labava povezanost što nije slučaj u ovakvom tipu integracije. Zbog toga će u nastavku biti prikazani načini integracije u kojima servisi surađuju te sakrivaju svoju internu implementaciju. [5, str 40-41]

3.2.3. Komunikacija Zahtjev / Odgovor

Jedan od načina implementacija komunikacija servisa je komunikacija bazirana na zahtjevu i odgovoru. U ovoj komunikaciji klijent započinje komunikaciju putem zahtjeva te čeka odgovor. Ovaj tip komunikacije može biti implementiran sinkrono i asinkrono. Slika 17 prikazuje dijagram komunikacije zahtjev i odgovor. Na slici je prikazano kako registracija šalje zahtjev direktno mikroservisima kako bi oni odradili određenu akciju.

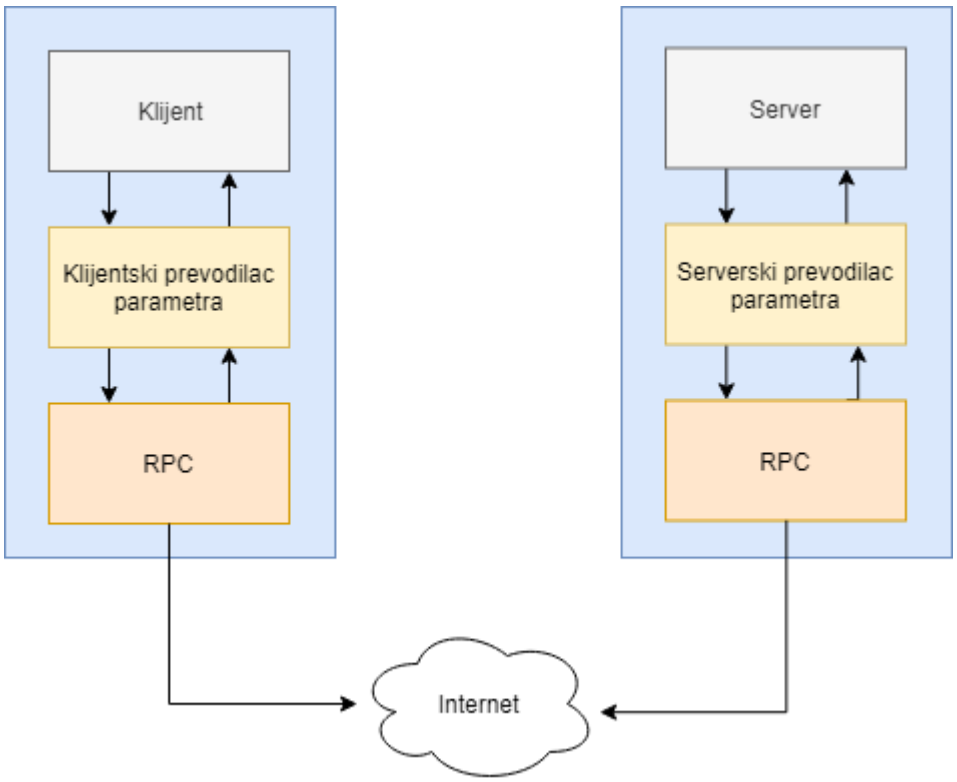


Slika 17 Komunikacija zahtjev / odgovor [5]

U daljnjem nastavu bit će prikazana dva tipa implementacije komunikacije baziranoj na zahtjevu i odgovoru: RPC i REST.

3.2.3.1. RPC

RPC je kratica od Remote Procedure Call. RPC je tehnika u kojoj određeni zahtjev poziva lokalno, a samo njegovo izvršavanje se odvija na udaljenom serveru. Dijagram rada RPC tehnike je prikazan na slici 18.



Slika 18 Prikaz RPC dijagrama [5]

Postoje mnoge tehnologije koje se koriste u implementaciji RPC-a kao što su SOAP, Thrift i protocol bufferi. Velika većina ovih tehnologija omogućava generiranje koda koji ubrzava implementaciju te vrlo brzu izradu komunikacije između raznih servisa.

Negativna strana nekih od tih tehnologija je uska povezanost s drugim tehnologijama. Na primjer Java RMI je usko povezan s Java platformom te samim time ograničava slobodu kod implementacije sustava. Cilj ovakvih sustava je sakrivanje kompleksnosti udaljenih poziva, no problem kod toga je taj što određene tehnologije ponekad previše maskiraju udaljene pozive što može dovesti do njihovog prevelikog korištenja. Udaljeni pozivi su vrlo različiti od lokalnih poziva budući da se komunikacija odvija putem mreže što je sporije od lokalnih poziva te je zbog toga potrebno obratiti pažnju na udaljene pozive unutar RPC-a. [5, str 46]

Iako RPC ima neke nedostatke, ovaj tip načina integracije servisa je korak naprijed u koheziji te labavoj povezanosti servisa u usporedbi s dijeljenom bazom podataka.

3.2.3.2. REST

REpresentation State Transfer (REST) je tip popularnog arhitekturnog stila inspiriranog webom. Najvažniji koncept unutar REST implementacije je resurs. Resurs je tip podatka (npr. korisnik) koji servisi koriste. Reprezentacija korisnika koju server pruža na zahtjev može biti potpuno različita od reprezentacije korisnika u bazi podataka. Format u kojem server vraća informacije može biti potpuno različit od formata u kojemu je taj podatak spremljen, a podaci koji se šalju klijentu također mogu biti djelomični ili modificirani. Prednost ovog načina implementacije je u tome što je ovisnost o drugim tehnologijama vrlo mala, a unutarnja logika servera je sakrivena od klijenta. Posljedica toga je labava povezanost što je velika prednost mikroservisne arhitekture.

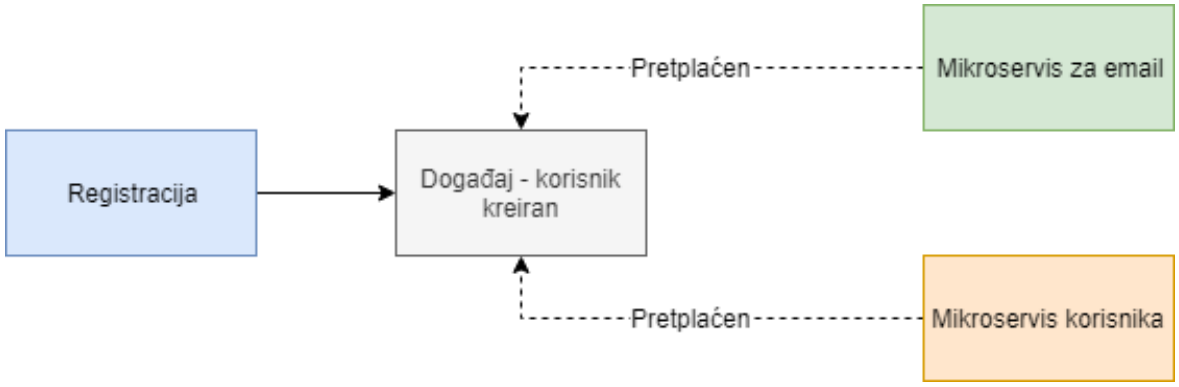
REST ne definira protokol kojim se koristi no u ovom radu opisan je REST putem HTTP-a. REST putem HTTP-a implementira se koristeći tipove HTTP zahtjeva (GET, POST i PUT). U REST-u kreiramo servis koji je na primjer odgovoran za rad s korisnicima. Ovisno o tipu zahtjeva odrađuje se druga operacija. GET vraća informacije o korisniku, POST kreira korisnika dok PUT zahtjev editira korisnika.

S druge strane REST implementacija zahtjeva više vremena od RPC implementacije budući da se kod takve implementacije ne generira samostalno. Također korištenje REST servisa zahtjeva više posla u usporedbi s nekim RPC implementacijama. REST preko HTTP-a nije najpogodniji za komunikaciju između servera gdje je potrebna mala latencija te male poruke budući da HTTP nije najbolje rješenje za takve situacije. Uz ove manje nedostatke REST je logičan prvi odabir za komunikaciju između servisa. [5, str 50]

3.2.4. Komunikacija bazirana na događaju

Komunikacija bazirana na događaju je obrnuta od komunikacije baziranoj na zahtjevu i odgovoru. U ovoj komunikaciji klijent govori servisima što se dogodilo te očekuje da servisi znaju što je potrebno napraviti umjesto da klijent zahtjeva da se određena akcija izvrši kao što je to u komunikaciji baziranoj na zahtjevu i odgovoru. Ovakvi sustavi su po svojoj prirodi asinkroni te je logika izrazito distribuirana po sustavu. Također ovakvi sustavi su izrazito

odvojeni budući da sam klijent niti ne zna koje će se akcije sve izvršiti što znači da je vrlo lako moguće dodati novu akciju koja će se izvršiti bez ikakvih promjena na klijentu.



Slika 19 Komunikacija bazirana na događaju [5]

Da bi ovakav sustav radio potrebno je implementirani način na koji mikroservisi mogu kreirati i emitirati događaje te slušati na određene događaje. Najčešći tradicionalni način implementacije su posrednici porukama (message brokers). Mikroservisi kreiraju događaje koristeći API a posrednik je odgovoran za javljanje mikroservisima da se događaj na koji je pretplaćen dogodio. Drugi noviji način je korištenjem HTTP zahtjeva i ATOM specifikacije koja specificira na koji način se mogu objavljivati i slušati događaji.

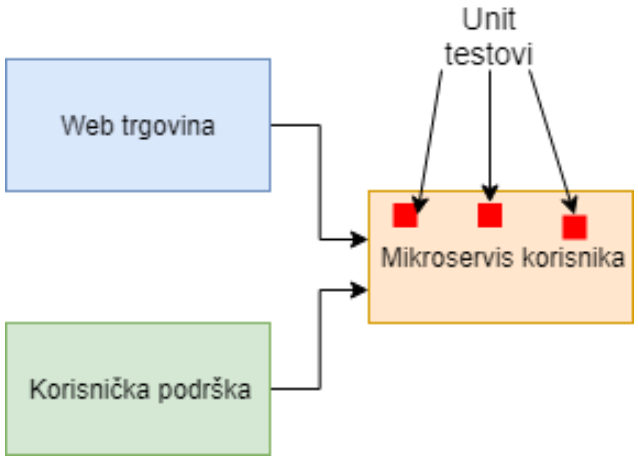
Ovaj tip implementacije pruža mnoge prednosti kao što su skalabilnost, distributivnost, kohezija i labava povezanost, no ovaj tip sustav je vrlo kompleksan za implementaciju i zahtjeva veliku ekspertizu kod implementacije zbog čega je preporučen oprez kod implementacije ovakvog tipa sustava. [5, str 56-57]

3.3. Testiranje sustava

Testiranje je vrlo bitna stavka u razvoju koja osigurava stabilnost i pravilan rad informatičkog sustava tijekom njegove implementacije ili nadogradnje. Informatički sustav se može testirati ručno i automatski, a u današnje doba cilj je što više automatizirati sustav. Testove koji testiraju sustav razlikujemo po obujmu sustava koji testira te se dijela na: jedinične testove (en. unit test), testove servisa te testove od kraja do kraja(en. end to end test). Unutar ovog poglavlja bit će objašnjena primjena svih gore navedenih testova te će biti prikazane najbolje prakse za testiranje mikroservisnih sustava.

3.3.1. Jedinični testovi

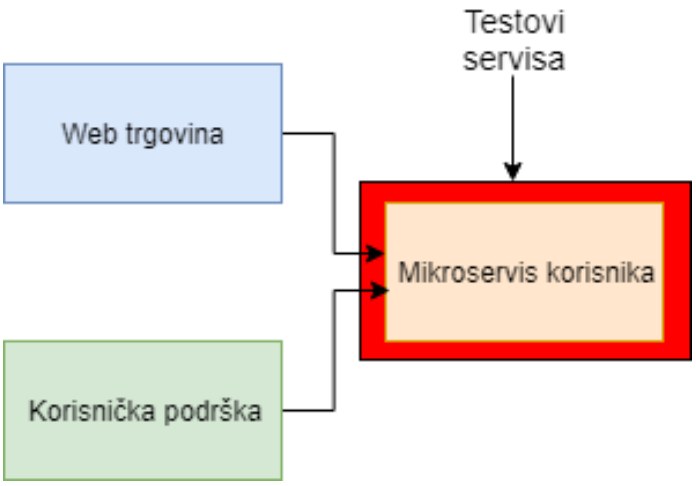
Jedinični testovi su testovi koji testiraju najmanji opseg sustava. Ovi testovi testiraju jednu metodu ili funkciju unutar sustava izoliranu od bilo kakvih vanjskih utjecaja. U ove testove spadaju i svi testovi napisani kod implementacije pomoću tehnike Test driven development. Jedinični testovi su vrlo brzi te testiraju programsku logiku metode ili funkcije bez pokretanja vanjskih servisa, čitanja datoteka ili komunikacije pomoću mreža. U sustavu je vrlo dobro imati što više ovakvih testova budući da se vrlo brzo izvode te su prva linija zaštite u pronalasku logičkih grešaka. Ovi testovi su testovi za programere te olakšavaju implementaciju i refaktoriranje budući da ,ako su dobro napisani, vrlo brzo pronalaze grešku u kodu. Na slici 20 prikazan je sustav te opseg koji jedinični test testira. [5, str 134]



Slika 20 Prikaz opsega testiranja jediničnih testova [5, 2015]

3.3.2. Testovi servisa

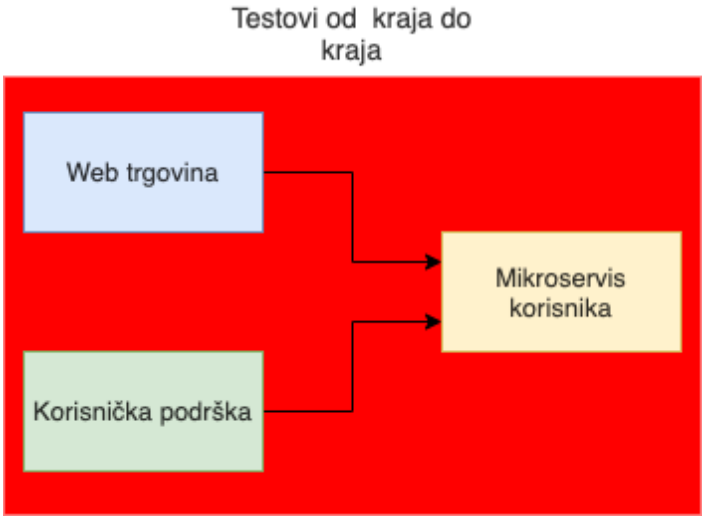
Testovi servisa su obujmom veći od jediničnih testova. Ovi testovi također izoliraju servis od vanjskih utjecaja te testiraju funkcionalnost servisa u izolaciju. Izolacija od vanjskih servisa se postiže lažiranjem servisa (en. service mock). Lažiranjem mijenjamo poziv pravog servisa s pozivom lažnog servisa koji uvijek vraća očekivani rezultat. Ovi testovi također pomažu u pronalasku grešaka u kodu no zbog većeg opsega testa pronalazak uzroka greške zahtjeva veći trud nego što to zahtjeva jedinični test. Na slici 21 prikazan je opseg testa servisa. [5, str 134]



Slika 21 Prikaz opsega testiranja servisnih testova [5]

3.3.3. Test od kraja to kraja

Testovi od kraja do kraja su najopširniji tip testova. Ovi testovi testiraju cjelokupnu aplikaciju najčešće kroz korisničko sučelje oponašajući stvarnog korisnika. Uspješno izvršavanje ovakvih testova potvrđuje dobru integraciju i implementaciju sustava. Na slici 22 prikazan je opseg testova od kraja do kraja.



Slika 22 Prikaz opsega testiranja testa od kraja do kraja [5]

Iako testovi od kraja do kraja testiraju cijeli sustav, ovaj tip testova sa sobom nosi mnoge negativne karakteristike. Budući da ovi testovi testiraju cijeli sustav, pronalazak greške u kodu kada je test neuspješan zahtjeva puno vremena. Ovi testovi nisu najpogodniji za mikroservisnu arhitekturu zbog distribuirane arhitekture sustava. Bilo koja momentalna greška u mreži može dovesti do neuspješnog testa što dovodi do nepotrebnog traženja grešaka u kodu. U pravilu što više mikroservisa i dijelova sustava test od kraja do kraja testira, to je on nestabilniji i dovodi

do situacija gdje test bude neuspješan, dok je kod ponovnog pokretanja test uspješan. Ovakvi testovi smanjuju povjerenje u testove te ih treba maknuti iz izvršavanja dok se njegova nestabilnost ne ispravi. [5, str 135]

3.3.4. Testiranje mikroservisnog sustava

Testiranje mikroservisnih sustava dodaje mnoge kompleksnosti u usporedbi s monolitnim sustavima. Budući da mikroservisi mogu biti distribuirani između mnogo timova potrebno je podijeliti posao pisanja testova. Logično rješenje je to da testove piše tim koji je odgovoran za mikroservis no u ovom slučaju i dalje ne postoji odgovornost oko testova od kraja do kraja budući da ti testovi testiraju sve mikroservise. Organizacijske podijele, ali i nestabilnosti koje su spomenute u prethodnom primjeru su razlog zbog kojih bi u mikroservisnom sustavu trebalo postojati što manje testova od kraja do kraja koji testiraju određen korisnički proces koji testira što veći dio sustava.

Umjesto ekstenzivnih testova od kraja do kraja u mikroservisne testove uvode se testovi bazirani na korisniku mikroservisa. Cilj ovakvog tipa testa je testirati promjene na servisu koje utječe na korisnike tog servisa te dovodi do greški koje mogu pokvariti sustav o kojima je pisano u poglavlju 3.2.1. Ovim testovima identificiraju se greške prije nego verzija koda dođe u produkciju te pokvari komunikaciju između mikroservisa. Jedan od rješenja za ovakav tip testova je Pact [14]. Pact je testni alat otvorenog koda (en. open source) za testove orijentirane korisniku koji olakšava kreiranje i pisanje ovakvih testova, a baziran je na Ruby jeziku.

3.4. Najbolje prakse

Unutar ovog poglavlja bit će prikazani i objašnjeni različiti pristupi kod razvoja sustava baziranog na mikroservisnoj arhitekturi. Teme koje će biti obrađen su: praćenje rada sustava, sigurnost sustava te skaliranje sustava. Za svaku temu bit će prikazane razne tehnike implementacije te prednosti i nedostaci odabrane tehnike. Osim toga navesti će se neki alati koji se mogu koristiti za implementaciju određene tehnike u mikroservisnom sustavu.

3.4.1. Razvoj mikroservisnog sustava

Razvoj i postavljanje na server monolitnih aplikacija uobičajen je standardan proces koji se koristi već dugo vremena. Jedan repozitorij na kojem se pokreću svi testovi te jedan server na kojem je aplikacija uvelike su jednostavniji od distribuiranog mikroservisnog sustava.

Kod razvoja svih sustava preporučeno je koristiti kontinuirani razvoj i integraciju. Kontinuirana integracija je način razvoja sustava u kojoj se kod prije integracije sa sustavom prevodi i pokreče na odvojenom serveru koji pokreće jedinične, servisne i integracijske testove te vraća rezultat izvedenih testova. Ovakav sustav osigurava da se u konačnu granu razvoja integrira samo kod koji je točan. Kontinuirani razvoj kod mikroservisne arhitekture je zahtjevniji za postaviti od monolitne arhitekture te postoji nekoliko načina na koji je to moguće odraditi.

Prvi i najjednostavniji način oponaša postavke monolitne arhitekture. U ovom načinu kontinuiranog razvoja i integracije svi servisi se nalaze na jednom repozitoriju koda i kod svake integracije se pokreće jedan integracijski build. Ovo je najjednostavnija postavka kontinuirane integracije te najjednostavnija za programere koji moraju raditi na više mikroservisa odjednom, no ovaj način ima mnoge negativne strane. Prva negativna strana je vrijeme potrebno za prevođenje i testiranje koda. Budući da se sve nalazi u jednom repozitoriju nakon svake i najmanje promjene na jednom servisu ponovno se testiraju svi servisi sustava. Osim toga ako dođe do pogreške u integraciji, razvoj svih servisa staje do ispravka greške. [5, str 105]

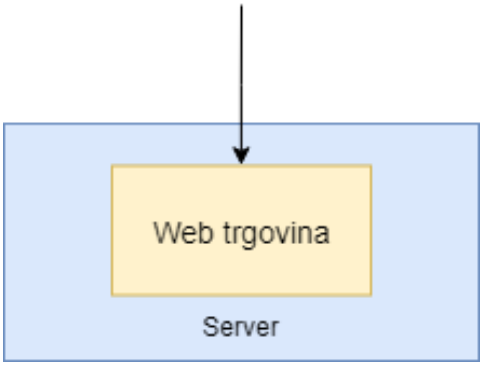
Drugi način koristi također jedan repozitorij no više različitih integracijskih buildova koji se pokreću ovisno o lokaciji promijenjenog koda. Prednost ovog načina je ta što se testovi vrte samo za servis koji je promijenjen, a jedan repozitorij i dalje olakšava rad programerima. Negativna strana ovakvog načina rada je ta što kod ovakvog načina rada se promjene na više servisa mogu raditi odjednom što može dovesti do uskog vezanja različitih servisa.

Treći i najbolji način za razvoj mikroservisa je korištenje odvojenih repozitorija i odvojenih integracija za svaki mikroservis sustava. Na ovaj način u sam razvoj je već uvedena distribucija koda što smanjuje trajanje builda i testova te je organizacija posla puno jasnija. Jedina negativna strana ovog pristupa je nešto kompleksniji razvoj na više repozitorija te veće vrijeme potrebno za postavljanje integracije za sve mikroservise. [5, str 106-107]

3.4.2. Praćenje rada sustava

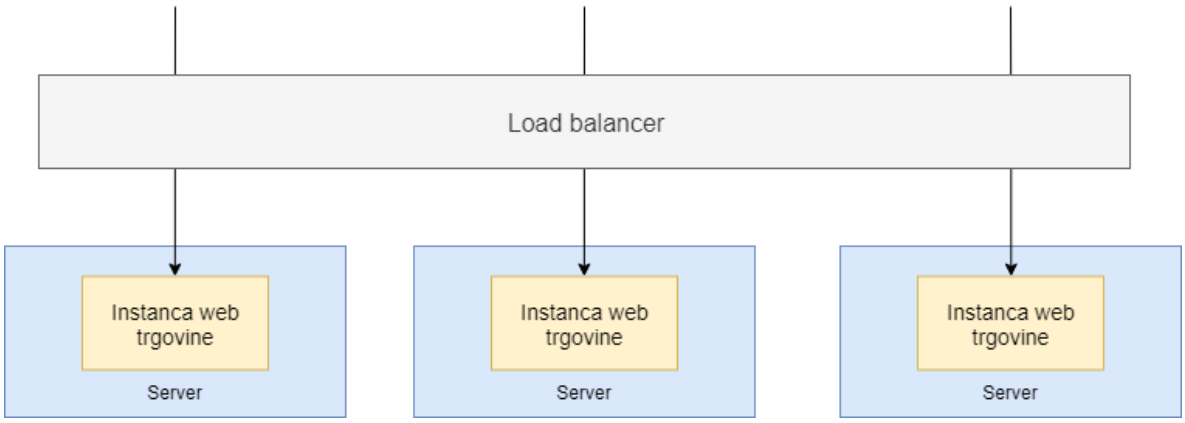
Kao što je navedeno i prethodnim poglavljima, jedan od izazova mikroservisne arhitekture je praćenje rada cjelokupnog distribuiranog sustava. Dok u monolitnim sustavima u slučaju grešaka ili pada aplikacije imamo samo jednu točku u kojoj je potrebno tražiti uzrok nestabilnosti u mikroservisnim sustavima ovaj problem je kompleksniji zbog čega je pravilno postavljeno praćenje sustava iznimno bitno kako bi se ovaj problem olakšao.

Ovisno o složenosti arhitekture praćenje rada sustava može biti jednostavno ili vrlo složeno. U najjednostavnijem slučaju u malim sustavima jedan servis se nalazi na jednom serveru. U ovoj arhitekturi potrebno je pratiti podatke samo na jednom serveru. Podaci koji bi se u pravilu trebali pratiti su stanje CPU-a i memorije servera, logovi grešaka i minimalno vrijeme odgovora servisa. Primjer ovakvog jednostavnog servisa vidimo na slici 23. [5, str 156]



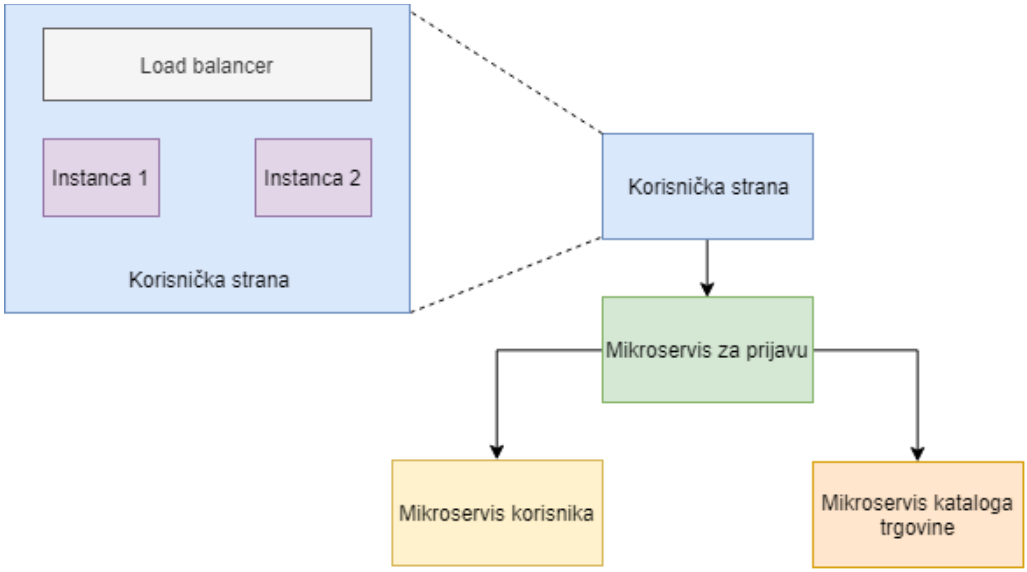
Slika 23 Prikaz sustava - 1 servis 1 server [5, 2015]

U slučaju većih sustava neki servisi mogu raditi na nekoliko različitih servera. Kod ovakvih situacija praćenje sustava postaje kompleksnije. Potrebno je pratiti jednake podatke stanja servera kao i u prethodnoj situaciji, no u ovom slučaju potrebno je agregirati podatke svih servera s mogućnosti razlikovanja podataka. Osim toga pretraga logova zahtjeva bolje rješenje od ručnog pretraživanja datoteka dnevnika rada server po server. U ovom slučaju alat poput ssh-multiplexera omogućava pokretanja istih naredbi na više servera istovremeno što uvelike ubrzava posao. Kod praćenja vremena odgovora može se upotrijebiti praćenje uravnoteživača opterećenja (en. load balancer). Dijagram servisa na više različitih servera može se vidjeti na slici 24. [5, str 156-157]



Slika 24 Prikaz sustava - 1 servis na više servera [5]

U većim i složenijim sustavima postojat će više servisa koji se nalaze na više različitih servera te zajedno surađuju kako bi pružili određenu funkcionalnost. U ovakvim sustavima potrebno je napraviti centralnu agregaciju podataka sa svih servera na kojima servisi rade. Agregaciju dnevnika rada na centralnom mjestu moguće je napraviti s alatom kao što je to logstash koji može obrađivati i skupljati različite datoteke dnevnika rada te ih slati na centralno mjesto za daljnju analizu. Na centralnom mjestu za agregiranje dnevnika rada može se koristiti program Kibana[15] koji omogućava pretraživanje ili generiranje grafova na bazi podataka iz datoteka dnevnika rada (en. log). [5, 2015, str 157-158]



Slika 25 Prikaz sustava - više servisa na više servera [5]

Kod praćenje rada servera potrebno je agregirati podatke CPU-a i memorije te ih prikazati na jednoj centralnoj lokaciji. Kod prikupljanja podataka potrebno je agregiranje napraviti tako da se svaki podaci mogu povezati s instancom servisa i servera s kojeg su podaci došli. Sustav koji omogućava lako prikupljanje i integraciju za praćenje rada sustava je Graphite [16]. Graphite je sustav koji pruža jednostavan API koji omogućava slanje podataka servera u realnom vremenu. Kada su podaci agregirani Graphite omogućava pregled podataka koristeći razne grafove koji olakšavaju analizu potrošnje serverskih resursa. Osim pronalaska greške ovakvo praćenje daje informacije o servisima koja će biti potrebno skalirati u budućnosti.

Za razliku od praćenja servera koji sam generira sve podatke koju agregiramo (stanje CPU, memorije...) kod implementiranih servisa potrebno je tu metriku izložiti vanjskim alatima kako bi ju mogli koristiti kod analize rada sustava. Neke minimalne stvari koje je preporučeno

pratiti su vrijeme potrebno servisu za odgovor na zahtjev te broj grešaka koji se dogodi kod obrade zahtjeva. Osim ovih minimalnih metrika, preporučuje se praćenje i specifičnih stvari kojim se servis bavi. Na primjer kod web trgovine preporučuje se praćenje obavljenih kupnji, broj dodanih artikala u košaricu, tip artikala i slično. Svi ovi podaci daju dodatne informacije ne samo u stanju sustava već i o načinu na koji korisnik koristi sustav. Ove informacije se zatim mogu koristiti u budućnosti kod skaliranja sustava, ali i kod poboljšanja sustava kako bi se maksimizirao profit, ali i olakšala upotreba aplikacije. U načelu, rijetke su situacije kada praćenje određene statistike rada sustava može biti negativno za sam sustav. Za implementaciju praćenja stanja sustava, moguće je koristiti razne biblioteka kao što je na primjer Metrics [17] za JVM sustave.[5, str 160]

3.4.3. Sigurnost sustava u mikroservisnoj arhitekturi

Vrijednost informacija koje kompanije posjeduju o svojim korisnicima sve je veća. Zbog toga, sigurnost informacija u sustavima postaje jedna od najbitnijih stavka informacijskog sustava. Unutar ovog poglavlja ukratko će biti objašnjeni sigurnosti aspekti kojih je potrebno biti svjestan kod dizajniranja mikroservisnog sustava. Ti aspekti su sigurnost podataka u razmjeni između dvije točke, sigurnost podataka u samom sustavu, sigurnost operacijskog sustava na kojemu se aplikacija nalazi te sigurnost mreža putem kojih komunikacija komunicira. Osim toga objasnit će se ljudski aspekt kao što je autentificiranje osobe te autorizacija njegovih prava.

3.4.3.1. Autentifikacija i Autorizacija

Autentifikacija i autorizacija čine jezgru koncepata zaštite kada su u pitanju osobe koje imaju interakciju s našim sustavom.

Autentifikacija označava proces u kojem potvrđujemo da je osoba ona za kojom se predstavlja, odnosno u autentifikacijskom procesu jedinstveno prepoznajemo osobu. Za mnoge aplikacije autentifikacijski proces sastoji se od upisivanja korisničkog imena i lozinke, dok danas na mobitelima postoje i kompleksniji načini autentifikacije na bazi biometrijskih podataka.

Autorizacija je mehanizam koji definira koja prava određen korisnik ima, odnosno ovaj mehanizam daje pravo pristupa korisniku na sadržaj koji smije pristupiti dok mu brani pravo za sadržaj kojem ne smije pristupiti. Na primjer administrator u sustavu smije pristupiti listi korisnika sustava te ga suspendirati dok običan korisnik nema pravo pristupa niti tom korisničkom sučelju niti toj funkcionalnosti. [5, str 169]

U distribuiranim sustavima autentifikacija i autorizacija je složeniji proces nego u monolitnim sustavima budući da zahtjeva implementaciju autentifikacije i autorizacije tako da potvrda identiteta na jednom mikroservisu bude distribuirana na sve mikroservise te da jedan identitet vrijedi za sve distribuirane sustave.

Uobičajena praksa kod implementacije autentifikacije i autorizacije u mikroservisnim sustavima je korištenje nekog od tipa Single Sign-On (SSO) solucije. Dvije najpopularnije tehnologije za SSO su SAML i OpenID [18]. Princip na kojem SSO radi je sljedeći. Kada korisnik pokuša pristupiti određenom resursu, on je preusmjeren servisu za autentificiranje. Ako se korisnik uspješno autentificira korištenjem korisničkog imena i lozinke taj servis daje do znanja prvom servisu da se korisnik autentificirao te prvi servis odlučuje ima li taj korisnik prava pristupa ili ne. Identifikacija korisnika može biti interna ili korištenjem vanjskih resursa. Na primjer stranice poput Facebooka ili Googlea pružaju mogućnost integracije s njihovim sustavima kako bi se korisnik mogao identificirati. Preporučuje se korištenje vanjskih sustava identifikacije kada se radi o javnim servisima budući da su ti sustavi implementirani korištenjem zadnjih standarda te bi u načelu implementacija takvih velikih kompanija trebala biti bolja od vlastite implementacije. Ako se radi o nekoj internoj aplikaciji gdje je potrebna veća kontrola nad podacima korisnika preporučuje se vlastita implementacija davatelja identiteta (en. Identity provider). [5, str 170-171]

Kako bi se izbjegla implementacija preusmjeravanja i identifikacije u svakom od mikroservisa, preporučuje se implementacija određenog prolaza (en. gateway) kroz koji zahtjevi moraju proći kako bi došli do servisa. Ovaj način implementacije omogućava centraliziranu autentifikaciju koja svakom korisniku pridaje određenu ulogu. Temeljem te uloge ostali mikroservisi mogu kasnije točno odlučiti za koje akcije je korisnik autoriziran a za koje nema prava.

Osim identifikacije korisnika u sustavu u mikroservisima treba voditi računa i o identifikacije servisa prema drugim servisima kako bi izbjegli krivo korištenje servisa te curenje povjerljivih informacija iz sustava. Postoji nekoliko strategija autentifikacije i autorizacije servisa koji će biti objašnjeni: dozvoljavanje svih zahtjeva unutar granice (en. perimeter), HTTPS osnovna autentifikacija, SAML/OpenID, klijentski certifikati, HMAC preko HTTPa, API ključevi.

Dozvoljavanje svih zahtjeva unutar granice bez ikakve autentifikacije je najjednostavniji način implementacije komunikacije između servisa budući da u komunikaciji servisa ne postoji autentifikacija i autorizacija. Ideja ove opcije je ta da se zaštiti ulaz u mrežu unutar koje mikroservisi komuniciraju te se zatim dozvoli sva komunikacija servisa unutar te

mreže. Očita negativna strana je ta da u slučaju da zaštita mreže podbaci, cijeli sustav postaje ranjiv te otvoren za korištenje napadačima.

HTTP(S) osnovna autentifikacija je druga opcija koja je vrlo popularna a sam protokol je dobro podržan od strane raznih tehnologija. Ovaj način autentifikacije radi tako da korisnik šalje korisničko ime i lozinku u zaglavlju zahtjeva koji drugi servis čita i autentificira korisnika. Kod ove autentifikacije potrebno je koristiti HTTPS kako bi slanje lozinke i korisničkog imena bilo sigurno i enkriptirano od vanjskih korisnika. Negativna strana ove autentifikacije je upravljanje sa SSL certifikatima kako bi se mogao koristiti HTTPS te keširanje budući da SSL zahtjevi ne mogu biti keširani od reverse proxya već je to potrebno raditi na serveru ili klijentu. Također u ovom pristupu je poznato samo da su lozinka i korisničko ime dobri no ne zna se od kuda je zahtjev došao.

SAML/OpenID opcija je SSO opcija o kojoj je već pisano kod autentifikacije korisnika. Ova opcija je pogodna za korištenje kada sustav već koristi jednu od ove dvije tehnologije za identifikaciju korisnika. Ova solucija zahtjeva izradu korisnika za servise te njihovo upravljanje. Negativna strana ove solucije je upravljanje korisničkim imenom i lozinkom jednako kao i u HTTPS autentifikaciji te kompliciranost implementacije, a naročito korištenjem SAML tehnologije. [5, str 174]

Klijentski certifikati koriste TLS gdje klijent koristi X.509 certifikate za autentificiranje sa serverom. Server provjerava valjanost certifikate te ako je certifikat validan odrađuje akciju koju klijent zahtjeva. Glavne negativne strane ove implementacije je upravljanje certifikatima svih klijenta te kreiranje novih i povlačenje starih certifikata.

HMAC (hash-based messaging code) je tip autentifikacije koji je ekstenzivno korišten od strane Amazone Web Servicea. Ovaj tip autentifikacije koristi hash algoritam koji koristi tijelo poruke i tajni ključ za kreiranje hasha korištenog u autentifikacije. Server koristi isti tajni ključ na serverskoj strani te ako je hash jednak dozvoljava zahtjev od klijenta. Ovaj način autentifikacije sprječava napad čovjeka u sredini (en. man in the middle) budući da se hash mijenja kod bilo kakve promjene tijela poruke. Loša strana ovog načina je to što ova implementacija nije standardna te postoje mnogi načini implementacije. Također upravljanje šiframa može biti problem budući da se tajni ključ ili mora na sigurni način razmijeniti ili i klijent i server moraju imati ključeve na svojoj strani što dovodi do problema promjene šifra istovremeno na obje lokacije. [5, str 176]

API ključ je vrlo popularan način autentifikacije u implementaciji raznih javno dostupnih API-a. Najpopularniji način implementacije je korištenjem para javnog i privatnog

ključa. Glavna prednost ove implementacije je ta što je vrlo jednostavna za programiranje u usporedbi s na primjer SAML implementacijom.

Osim što je potrebno osigurati da autentikacija i autorizacija pravilno rade u sustavu, potrebno je i osigurati da su kritični podaci tajni kada se šalju putem mreže, ali da su i tajni kada su spremljeni u sustavu ako dođe do uspješnog napada na sustav. Kako bi podaci ostali tajni potrebno ih je enkriptirati koristeći neki od popularnih i sigurnih enkripcijskih algoritama kao što su AES-256. Za šifre preporučuje se hashiranje korištenjem soli gdje prilikom spremanja hashiranje šifre u šifru dodaju i nasumično generirani znakovi (sol) kako bi se izbjegao napad s rječnikom već izračunatih hash vrijednosti. Tajni podaci nikad ne smiju biti spremani u dekriptiranom obliku te se dekripcija podataka radi samo na zahtjev. Također ključ korišten kod enkripcije treba biti spremljen u posebnom trezoru kojem servisi mogu pristupiti.

Osim gore navedenih zaštita podataka i sustava potrebne su i neke dodatne zaštite kako bi sustav imao što veću dubinu zaštite te samim time bio otporniji na razne napade. Vatrozid, pravilno logiranje i praćenje sustava, sigurni operacijski sustav, sigurna podjela mreže te mnoge druge tehnike pridonose cjelokupnoj zaštiti sustava.

3.4.4. Skaliranje mikroservisnog sustava

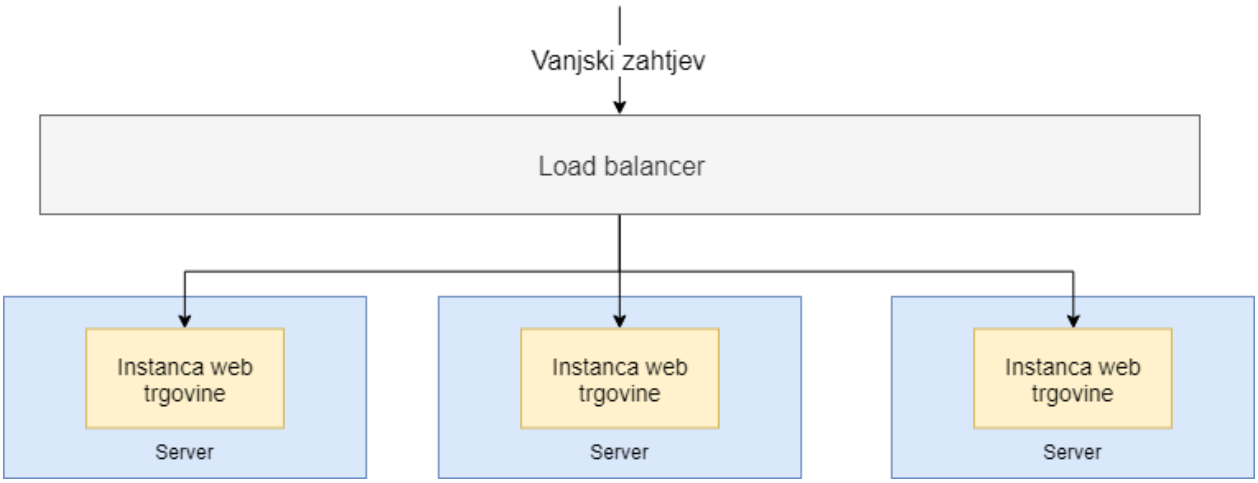
Skaliranje mikroservisa jedno je od bitnih prednosti mikroservisne arhitekture. Kao što je već objašnjeno u prethodnim poglavljima, mikroservisna arhitektura omogućava skaliranje svakog servisa posebno izolirano od cijelog sustava.

Kada se govori o skaliranju sustava, skaliranje se najčešće radi iz dva razloga: smanjivanje rizika za pad sustava i povećavanje performansi i ubrzavanje rada sustava. Kod skaliranja za performanse postoji nekoliko stvari koje je moguće napraviti kako bi se sustav ubrzao.

Prva i najjednostavnija tehnika je povećanje performansi servera na kojem je servis pokrenut. Povećanje brzine procesora i memorije može biti brzo rješenje za ubrzavanje rada sustava, no ovo rješenje nije idealno i ne pomaže u svim situacijama. Nije moguće stalno nadograđivati sustav jačim, a osim toga jači sustav ne mora značiti bolje performanse ako sama implementacije ne iskorištava taj sustav maksimalno. Drugi način na koje je moguće povećati performanse sustava je podjela servisa na više različitih servera kako bi se obujam posla podijelio na više instance. Osim podjela na server moguće je podijeliti i sam mikroservis u dva manja servisa ovisno o obujma posla koji obavljaju. Tom podjelom odvajaju se manje zahtjevni

servis koji se nalaze u mikroservisu od onih zahtjevnijih te se skaliraju samo mikroservisi koji su performansama zahtjevniji.

Kada se sustav skalira kako bi se smanjio rizik postoji također nekoliko najboljih praksi koje je bitno pratiti. Preporučuje se podjela sustava tako da se minimizira rizik. To znači da je najbolje podijeliti servise tako da se jedan servis nalazi na jednom serveru. Osim toga preporučuje se korištenje više poslužitelja te više podatkovnih centara na nekoliko lokacija. Ovim načinom podijele izbjegavamo jednu točku kvara (engl. single point of failure) koja bi pokvarila cijeli sustav. Podjela mikroservisa na više servera najčešće se radi upotrebom uravnoteživača opterećenja. Uravnoteživač opterećenja se koristi tako da sluša zahtjev ispred mikroservisa te šalje zahtjeve na odvojene instance mikroservisa tako da su sve instance jednako zaposlene. Također, uravnoteživači opterećenja pomažu kod nadogradnje sustava budući da je moguće nadograditi sustav bez prestanka rada. Upotrebom uravnoteživača opterećenja moguće je preusmjeriti promet na sustave koji se ne nadograđuju. Nakon uspješne nadogradnje promet se preusmjerava na nadograđeni server dok se ostali serveri nadograđuju bez da korisnik vidi ikakvu pauzu u radu sustava. Dijagram rada uravnoteživača opterećenja moguće je vidjeti na slici 26. [5, str 217]



Slika 26 Skaliranje sustava na više servera upotrebom uravnoteživača opterećenja[5]

Korištenjem tehnika navedenim iznad minimizira se rizik gdje pad jednog servera ili jednog servisa dovodi do pada cijelog sustava. No s tehnikama iznad nisu odgovorile na pitanja vezana uz bazu podataka. Ako servisi zapisuju informacije u jednu bazu podataka i dalje postoji rizik da pad jedne točke u sustavu (baza podataka) cijeli sustav postane nedostupan. Osim same nedostupnosti servisa u slučaju pada baze podataka bitno je da su podaci otporni, odnosno da u slučaju greške u bazi svi podaci nisu izgubljeni. Trajni gubitak podataka može dovesti do

velikih troškova te trajnog pada sustava. U nastavku će biti objašnjene neke tehnike koje rješavaju ovaj problem. [5, str 218]

Bazu podataka moguće je skalirati za pisanje i čitanje. Kod skaliranje za čitanje koristimo tehniku repliciranja podataka na više instanca baze podataka. Ova tehnika pridonosi otpornosti podataka budući da se podaci nalaze na više lokacija, ali također može ubrzati čitanje podataka iz baze. Ova tehnika radi tako da se jedna instance koristi za zapisivanje novih podataka dok se ostale instance koriste za čitanje. Nakon zapisivanje novih podataka u određenom vremenskom intervalu novi podaci se repliciraju na instance namijenjene za čitanje. Nedostatak ovog procesa je taj da se repliciranje ne događa odmah te se može dogoditi situacija da u određenom periodu podaci koji se čitaju nisu ažurni. Prikaz ove tehnike je vidljiv na slici 27.

Slika 27 Skaliranje baze podataka za čitanje[5]

Kod optimizacije za zapisivanje u bazu podataka koristi se tehnika sharding. U ovoj tehnici postoji više instanci baze podataka te se korištenjem hash funkcije određuje gdje će se koji podatak zapisati. Ova tehnika je vrlo komplicira budući da upiti nad bazom podataka postaju složeni zato što se upit mora izvršiti na svim instancama kako bi se dobio valjan rezultat. Osim toga dodavanje nove instance za pisanje zahtjeva rebalansiranje podataka između svih instanci što može dovesti do dugog trajanja nedostupnosti sustava. [5, str 223-224]

4. Analiza praktičnog dijela diplomskog rada

Za temu praktičnog rada odabrana je implementacija web aplikacije za agregiranje, kreiranje i obradu sportskih rezultata. Unutar ovog poglavlja bit će prikazani glavni dijelovi koda kako bi se pobliže objasnila unutarnja logika cjelokupne aplikacije. Budući da je aplikacija složena, unutar ovog poglavlja neće biti prikazan sav kod već samo određeni primjeri koji prikazuju glavne ideje iza implementacije određenih dijelova mikroservisa.

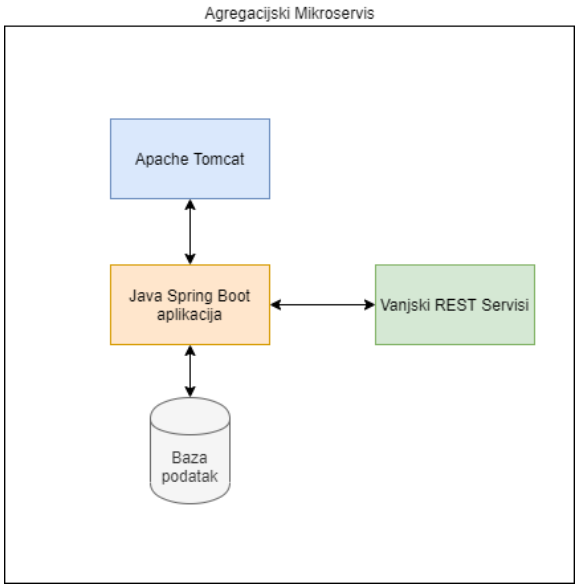
4.1. Arhitektura implementiranog sustava

Implementirani sustav koristi mirkoservisnu arhitekturu koja se sastoji od 5 mikroservisa koji zajedničkim radom omogućuju rad aplikacije. Prvi mikroservis se bavi agregiranjem sportskih podataka na internetu, drugi mikroservis zadužen je za kreiranje, čitanje, uređivanje i brisanje vlastitih sportskih podataka. Treći mirkoservis je registracijski i konfiguracijski mikroservis koji se koristi za konfiguriranje te otkrivanje svih mikroservisa aplikacije. Četvrti mikroservis je oAuth2 mikroservis koji služi za autentifikaciju i autorizaciju korisnika u aplikacije. I zadnji peti mikroservis je pristupni mikroservis, vrata mikroservisa koji sadrži korisničko sučelje te objedinjuje sve mikroservise u jednu cjelinu.

Slika 28 Arhitektura implementiranog sustava

4.1.1. Agregacijski mikroservis

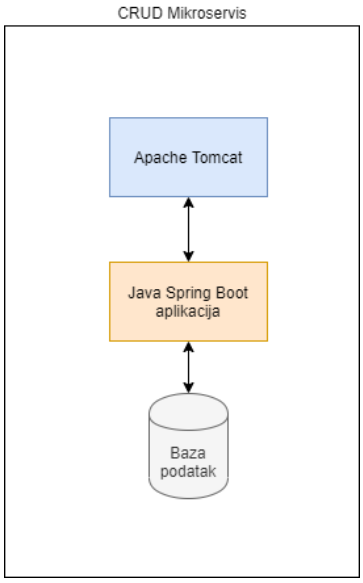
Agregacijski mikroservis, kao što mu i ime govori, je mikroservis koji je zadužen za agregaciju podataka na internetu. Ovaj mikroservis spaja se na vanjske REST servise te pohranjuje relevantne podatke za njih u vlastitu bazu kako bi se ti podaci kasnije mogli koristiti u vlastitoj aplikaciji. Mikroservis se sastoji od MySQL baze podatak te Java implementacije baziranom na Spring Boot okviru [19]. Server na kojem se aplikacija izvodi je Apache Tomcat.



Slika 29 Arhitektura agregacijskog mikroservisa

4.1.2.CRUD Mikroservis

CRUD (create, read, update delete) mikroservis je mikroservis zadužen za kreiranje, čitanje, ažuriranje i brisanje podataka na stranici. Ovaj servis služi za obradu podataka koje je korisnik kreirao te ti podaci nisu nastali automatizmom agregiranja. Ovaj mikroservis ima vrlo sličnu arhitekturu kao i agregacijski mikroservis no CRUD mikroservis ne ovisi o vanjskim servisima za kreiranje podataka. On također koristi MySQL bazu podataka, implementiran je u Java Spring Boot programskom okviru te za server koristi također Apache Tomcat.



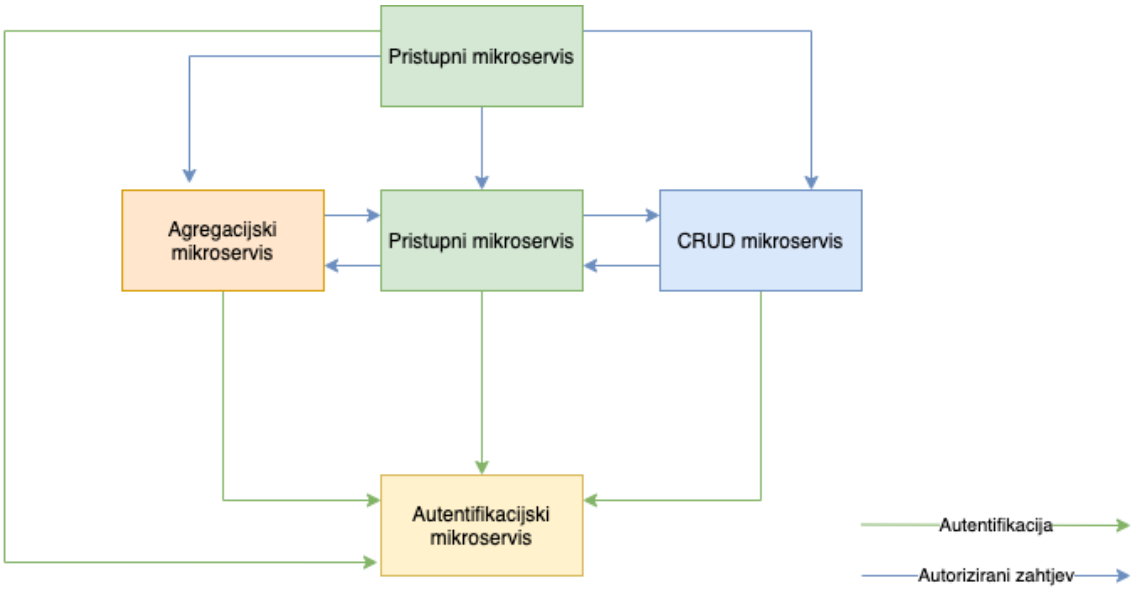
Slika 30 Arhitektura CRUD mikservisa

4.1.3. Autentikacijski mikroservis

Treći mikroservis u aplikaciji implementira autentikaciju i autorizaciju cjelokupne aplikacije što znači da je ovaj mikroservis zadužen za sigurnost aplikacije. U implementaciji ovog mikrosevira koristi se JHipster UAA (user accounting and authorizing)[20] implementacija bazirana na OAuth2 protokolu. Kako bi ovaj mikroservis zadovoljavao Oauth2 protokol mora poštivati šest pravila:

- Centralna autentifikacija
  - Autentifikacija se odvija samo u ovom tipu mikroservisa kako bi autentifikacija bila konzistentna kroz cijelu mikroservisnu arhitekturu
- Bez stanje (Statelessness)
  - U komunikacijskom protokolu nije potrebno zapisivati informaciju u sesiju o svakom komunikacijskom partneru budući da je informacija razumljiva u izolaciji
- Razlikovanje pristupa korisnika i računala
  - Server razlikuje autentifikaciju između korisnika i računala
- Fino granulirana kontrola pristupa
  - Moguće je detaljno definirati prava pristupa za svaki mikroservis
- Sigurnost od napada
  - Servis mora biti što otporniji od vanjskih napada
- Mogućnost skaliranja
  - Skaliranje aplikacije treba biti moguće te što jednostavnije

UAA server je kombinacija podatkovnog i autentifikacijskog servera te se smatra vlasnikom svih podataka u mikroservisima zato što je zadužen za autentifikaciju te autorizaciju svih podataka u mikroservisnoj arhitekturi. To znači da svaki zahtjev klijenta za nekim podatkom prvo prolazi autentifikaciju i autorizaciju kroz UAA. UAA unutar sebe ima definirane sve klijente koji pristupaju pristupnim točkama. Na slici ispod vidi se kako svaki zahtjev klijenta prvo mora proći autorizaciju UAA servera.



Slika 31 Prikaz rada autentifikacijskog mikroservisa

4.1.4. Registracijski i konfiguracijski mikroservis

Registracijski i konfiguracijski mikroservis je kao što mu i ime govori namijenjen za centralnu konfiguraciju svih mikroservisa sustava te registraciju odnosno prijavu svih mikroservisa. Osim toga ovo je i administratorski server namijenjen za praćenje statistika rada sustava i svih mikroservisa.

Za konfiguraciju mikroservisa koristi se Spring cloud config[22] tehnologija koja je namijenjena za vanjsko konfiguriranje distribuiranih sustava. Ova tehnologija omogućava da konfiguriramo sve servise sustava na jednom serveru koristeći .yml i .properties datoteke. Osim toga moguće je konfigurirati svaki tip mikroservisa ili svaki mikroservis pojedinačno ovisno profilu u kojem mikroservis radi.

Za registraciju i otkrivanje mikroservisa ovaj mikroservis koristi Spring Eureka [23] tehnologiju.

Netflix Eureka je REST servis namijenjen pronalaženju svih servisa sustava. Omogućava lako postavljanje novih mikroservisa u sustav bez potrebe za posebnom konfiguracijom svih drugih servisa. Ukratko ova tehnologija omogućava komunikaciju mikroservisa bez potrebe hard kodiranja IP i adrese portova drugih mikroservisa s kojima trenutni mikroservis treba komunicirati.

4.1.5. Pristupni mikroservis

Peti i zadnji mikroservis ove aplikacije je pristupni mikroservis. Ovaj mikroservis je ulaz u aplikaciju te se sastoji od korisničkog sučelja baziranom na React.js [24] biblioteci. React.js je biblioteka napravljena od strane Facebooka a primarna svrha joj je prikaz podataka u browseru. Osim ove tehnologije mikroservis koristi i React router [25] i React Redux [26]. React router je tehnologija koja se koristi za navigiranje na web stranici bez osvježavanja stranica što dovodi do gubljenja stanja aplikacije na korisničkoj strani dok je React Redux tehnologija namijenjena za praćenje stanje aplikacije kod klijenta. Budući da ovo korisničko sučelje koristi sve gore navedene tehnologije, sama aplikacija je zbog toga i SPA (single page application) što znači da se aplikacija učitava samo jednom prilikom pristupa te se nakon toga svi dodatni sadržaji dinamički mijenjaju upotrebom AJAX servisa bez potrebe za osvježavanjem cjelokupne stranice.

4.2. Analiza implementacije sustava

Unutar ovog poglavlja bit će analizirana implementacija svih mikroservisa ovog sustava. Svi serverski dijelovi sustava baziraju se na Java tehnologiji, konkretno programskom okviru Spring Boot. Za izradu cjelokupne aplikacije korišten je JHipster [21]. JHipster je besplatni generator koji ubrzava razvoj modernih web aplikacija. JHipstera omogućava brzo generiranje takozvanog predložnog (boilerplate) koda kako bi se što manje vremena koristilo na konfiguraciju servera, a više vremena koristilo uz konkretnu implementaciju zadaća sustava.

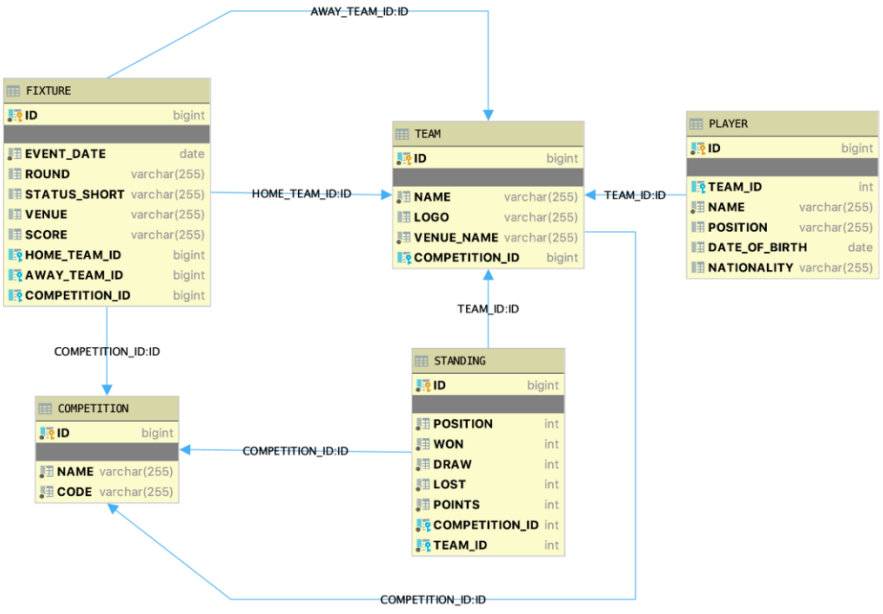
4.2.1. Agregacijski mikroservis

Agregacijski mikroservis sastoji se od tri glavna dijela:

- Model i baze podataka
- Servisi za agregiranje
- API servisi

4.2.1.1. Model i baza podataka

Kako bi agregacija podatak bila što jednostavnija, kreiran je model podataka koji najviše sliči modelu podataka REST servisa s kojih ovaj mikroservis preuzima podatke. Model se sastoji od pet tablica za spremanje podataka: igrači (Player), klubovi (Team), natjecanje (Competition), termin utakmice (Fixture) i stanje natjecanja (Standing). Dijagram baze podataka agregacijskog mikroservisa prikazan je na slici ispod.



Slika 32 Model agregacijskog mikroservisa

U dijagramu je prikazano pet tablica mikroservisa te relacije između njih. Tablica termina utakmice (Fixture) ima dvije relacije prema klubovima za domaćina i gosta termina te relaciju prema natjecanju kojem taj termin pripada. Osim toga vidljivo je da tablica stanja natjecanja (Standing) ima relacije prema timu i natjecanju kojem pripada. Tablica timova (Team) također ima relaciju na natjecanje kojem pripada. Zadnja relacija se nalazi između tablice igrača (Player) te tablice klubova iz koje se vidi kojem klubu igrač pripada.

Za rad s bazom podataka u implantaciji je korišten Spring Repository sučelje koje omogućava brzo i jednostavno slanje upita nad bazom podataka. Na slici ispod prikazan je primjer implementacije repozitorijskog sučelja za natjecanja.

```
package com.github.svarcf.football.repository;
import com.github.svarcf.football.domain.Competition;
import org.springframework.data.jpa.repository.*;
import org.springframework.stereotype.Repository;


/**
 * Spring Data  repository for the Competition entity.
 */
@SuppressWarnings("unused")
@Repository
public interface CompetitionRepository extends JpaRepository<Competition, Long> {

}
```

Slika 33 Implementacija sučelja repozitorija

Na slici iznad prikazana je implementacija sučelja koje proširuje postojeće Spring sučelje JpaRepository te upotrebom Java Generics-a definira kojem tipu objekta je ovaj repository namijenjen. Ovo proširenje zajedno s anotacijom Repository automatski implementira sve osnovne funkcije nad tablicom Competition u bazi podataka. Neke od automatski implementiranih funkcionalnosti su: spremanje podatka, dohvaćanje svih podataka unutar tablice, pronalazak podatka upotrebom primarnog ključa, brisanje podataka, ažuriranje postojećih podataka.

Druga tehnologija koja je korištena za lak rad s modelom baze podataka je Hibernate [27]. Hibernate je alat koji automatski mapira relacijsku bazu podataka u objektnu orijentiranu domenu. To znači da ova tehnologija preslikava podatke iz relacijske tablice u klasu entiteta koja u kodu predstavlja tablicu unutar baze podataka.

```java
@Entity
@Table(name = "competition")
public class Competition implements
Serializable {

    private static final long
serialVersionUID = 1L;

    @Id
    private Long id;

    @NotNull
    @Column(name = "name", nullable =
false)
    private String name;

    @NotNull
    @Column(name = "code", nullable =
false)
    private String code;

    @OneToMany(mappedBy = "competition")
    private Set<Fixture> fixtures = new
HashSet<>();

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public Competition name(String name)
{
        this.name = name;
        return this;
    }
```

Slika 34 Klasa entiteta

Na slici 34 prikazan je primjer jedne klase koja se koristi kod preslikavanja podataka iz tablice Competition u objekt. Ova klasa je običan POJO (Plain Old Java Object) objekt koji nije ograničen nikakvim posebnim restrikcijama no koristi anotacije pomoću kojih Hibernate preslikava podatke u objekt. Prva anotacija je Entity koja govori da je ova klasa prikaz modela baze podataka. Zatim anotacija Table koja govori kojoj tablici pripadaju podaci koji se preslikavaju u ovu klasu

4.2.1.2. Servisi za agregiranje

Servisi za agregiranje su kao što im i samo ime govorio dio implementacije koje je zaduženo za spajanje na vanjske REST servise te prikupljanje podataka u vlastitu bazu podataka. Agregiranje podataka odvija se periodično jednom dnevno u zakazano vrijeme. Periodično izvođenje agregiranja je implementirano upotrebom funkcionalnosti Spring Scheduler unutar Spring razvojnog okvira. Ova funkcionalnost omogućava korisniku da definira metodu koja će biti periodično izvršena u sustavu.

```
@Component
public class AggregationScheduler  {

    private Aggregator fixtureAggregator;
    private Aggregator playerAggregator;
    private Aggregator teamAggregator;
    private Aggregator standingAggregator;

    public AggregationScheduler(Aggregator fixtureAggregator, Aggregator
playerAggregator, Aggregator teamAggregator, Aggregator standingAggregator) {
        this.fixtureAggregator = fixtureAggregator;
        this.playerAggregator = playerAggregator;
        this.teamAggregator = teamAggregator;
        this.standingAggregator = standingAggregator;
    }

    @Scheduled(cron = "0 0 1 * * ?")
    public void scheduledTask(){
        teamAggregator.aggregate();
        playerAggregator.aggregate();
        fixtureAggregator.aggregate();
        standingAggregator.aggregate();
    }

}
```

Slika 35 Primjer Spring scheduler implementacije

Na slici 35 prikazan je primjer glavne klase koja periodično poziva sve agregacijske servise. Anotacija Scheduled je bitan dio koji definira kada se metoda treba izvršiti. U ovom primjeru izvršavanje je definirano u cron vremenu te je postavljeno na svakodnevno izvršavanje u jedan ujutro.

Agregacijski mikroservis sastoji se od četiri mikroservisa za agregaciju koji su vrlo slično implementirani. Na slici ispod prikazan je servis za agregiranje igrača koji je logički najkompleksniji

```
@Component
public class PlayerAggregator extends FoiAbstractRestRequest implements Aggregator
{


    private TeamRepository teamRepository;
    private PlayerRepository playerRepository;
    private ConversionService mvcConversionService;
    private ApplicationProperties applicationProperties;


    public PlayerAggregator(TeamRepository teamRepository, PlayerRepository
playerRepository, ConversionService mvcConversionService, ApplicationProperties
applicationProperties) {
        this.teamRepository = teamRepository;
        this.playerRepository = playerRepository;
        this.mvcConversionService = mvcConversionService;
        this.applicationProperties = applicationProperties;
    }

    @Override
    public void aggregate() {
        for (Team team : teamRepository.findAll()) {
            SoccerAPIData soccerAPIData =
this.getRequest(String.format(applicationProperties.getEndpoints().getPlayer(),
team.getId()), applicationProperties.getToken()).getBody();
            Arrays.stream(soccerAPIData.getSquad()).forEach(playerData -> {
                playerData.setTeam(team.getId());
                playerRepository.save(mvcConversionService.convert(playerData,
Player.class));
            });
            try {
                TimeUnit.SECONDS.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Slika 36 Servis za agregiranje igrača

Iz implementacije se vidi da ovaj agregator koristi četiri vanjske komponente kako bi izvršavao svoju zadaću. TeamRepository za rad s tablicom ekipa, PlayerRepository za rad s tablicom igrača, ConversionService za konvertiranje podataka iz transfernih objekata u model te ApplicationProperties za učitavanje centralne konfiguracije mikroservisa. Agregirajuća funkcija unutar for petlje prolazi kroz sve agregirane timove u bazi podataka te šalje zahtjev za igračima te ekipa na vanjski REST servis. Krajnja točka REST servisa te token za spajanje na vanjski web servis učitavaju se s centralnog mikroservisa koristeći objekt applicationProperties. Nakon što su podaci vraćeni iz vanjskog mikroservisa, svi igrači se u petlji konvertiraju u model koristeći konverzijski servis te se konvertirani model sprema u bazu podataka. Ova metoda šalje zahtjev za novim setom podataka igrača svakih 10 sekundi kako se ograničenja vanjskog servisa za brojem zahtjeva ne bi prekoračilo. Na slici 36 ne postoji implementacija REST

zahtjeva zato što je ta implementacija ponovno korištena u svim servisima te je zbog toga implementacija stavljena u klasu FoiAbstractRestRequest koju svi agregacijski mikroservisi implementiraju.

```
public abstract class FoiAbstractRestRequest {

    protected ResponseEntity<SoccerAPIData> getRequest(String url, String token) {
        RestTemplate restTemplate = new RestTemplate();
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));
        headers.set("X-Auth-Token", token);
        HttpEntity request = new HttpEntity(headers);

        ResponseEntity<SoccerAPIData> response = restTemplate.exchange(
            url,
            HttpMethod.GET,
            request,
            SoccerAPIData.class
        );

        return response;
    }
}
```

Slika 37 Implementacija apstraktne klase REST servisa

Slika 37 prikazuje implementaciju FoiAbstractRestRequest. Može se vidjeti da metoda getRequest koristi standardnu implementaciju poziva REST zahtjeva upotrebom RestTemplate-a te kao povratnu informaciju vraća odgovor servera kojeg je metoda pozvala.

Kao što je već rečeno, svi podaci se dohvaćaju upotrebom transfernih objekata (DTO) koji odgovaraju strukturi JSON odgovora REST završne točke. Upotrebom Apache Jacksona JSON odgovor REST servisa se automatski mapira u DTO objekt a jedan primjer takvog objekta nalazi se na slici 38.

```java
package com.github.svarcf.football.service.dto.external;

public class PlayerData {
    private long id;
    private String name;
    private String position;
    private String dateOfBirth;
    private String nationality;
    private long team;

    public PlayerData() {
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }

    public String getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(String dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }

    public String getNationality() {
        return nationality;
    }

    public void setNationality(String nationality) {
        this.nationality = nationality;
    }

    public long getTeam() {
        return team;
    }

    public void setTeam(long team) {
        this.team = team;
    }
}
```

Slika 38 Primjer transfernog objekta

PlayerData je POJO (plain old Java Object) te ne implementira nikakvu logiku već se samo koristi kao reprezentacija JSON objekta.

Kako bi se DTO objekti mogli spremiti u bazu podataka potrebno ih je pretvoriti u objekte koji reprezentiraju model baze podataka. Za implementaciju ove funkcionalnosti koristi se Sprint type conversion implementacija. Ova implementacija sastoji se od komponenata za konvertiranje koje putem Java Generica definiraju izvorni i krajnji objekt konvertiranja. Sve komponente konvertiranja se zatim automatski registriraju u Spring konverzijski servis koji se koristi u implementaciji agregacije. Spring programski okvir analizom tipa podataka koji se konvertira automatski pronalazi konverzijsku komponentu te ju koristi kako bi pretvorio DTO u model. Primjer jedne konverzijske komponente prikazan je na slici 39.

```
@Component
public class PlayerDataToPlayerConverter implements Converter<PlayerData, Player> {

    @Autowired
    private TeamRepository teamRepository;

    @Override
    public Player convert(PlayerData playerData) {
        Player playerModel = new Player();

        if(playerData.getDateOfBirth() != null){
            DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("yyyy-MM-
dd'T'HH:mm:ssXXX");
            playerModel.setDateOfBirth(LocalDate.parse(playerData.getDateOfBirth(),
dateFormat));
        }

        playerModel.setId(playerData.getId());
        playerModel.setName(playerData.getName());
        playerModel.setPosition(playerData.getPosition());
        playerModel.setNationality(playerData.getNationality());

        Optional<Team> team = teamRepository.findById(playerData.getTeam());
        team.ifPresent(value -> playerModel.setTeam(value));
        return playerModel;
    }
}
```

Slika 39 Primjer pretvarača igrača

Na slici 38 prikazan je pretvarač Igrača. Ova komponenta kao argument prima izvorni DTO objekt te koristeći datumska formatiranja te ekipni repozitorij pravilno postavlja podatke u novi model kako bi se on upotrebom Hibernate-a mogao spremiti u bazu podataka.

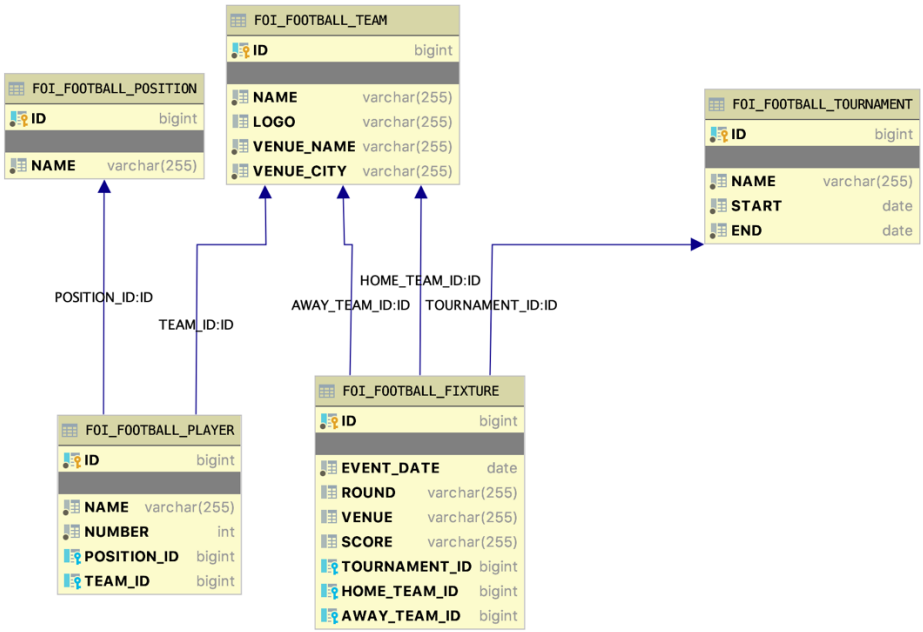
4.2.2. CRUD Mikroservis

CRUD mikroservis sastoji se također od 3 glavna dijela:

1. Model i baza podataka
2. Poslovna logika
3. API servisi

4.2.2.1. Model i baza podataka

Model baze podataka CRUD mikroservisa sastoji se od 5 tablica. Diagram modela prikazan je na slici 40.



Slika 40 Prikaz modela CRUD mikroservisa

Prva tablica je tablica Foi_Football_Position te služi za pohranu svih pozicija na kojima pojedini igrač može igrati. Druga tablica je tablica Foi_Football_Team te sadrži sve klubove unutar sustava. Tablica Foi_Football_Player je tablica svih igrača unutar sustava te sadrži vanjske ključeve na tablicu pozicija i klubova kako bi svaki igrač imao pripadajuću poziciju i klub za koji igra. Tablica Foi_Football_Tournament sadrži sve lige koje mikroservis sadrži. Zadnja tablica je tablica Foi_Football_Fixture koja sadrži podatke o odigranim utakmicama.

Ova tablica sadrži dva vanjska ključa na tablicu klubova te vanjski ključ na tablicu liga kojoj utakmica pripada. Ova tablica se koristi za generiranje rang tablice klubova za određenu ligu.

4.2.2.2. Poslovna logika

Poslovna logike aplikacije ovog servisa vrlo je jednostavna. Izvan standardnih servisa za obradu CRUD zahtjeva implementirana je logika za generiranje tablice prema odabranom turniru. Budući da je tablica turnira podatak izveden iz odigranih utakmica, ovi podaci se ne spremaju u bazu već se izračunavaju po zahtjevu korisnika.

Na slici 41 prikazana je implementacije logike za izračun tablice turnira.

```
public List<FoiFootballTableDTO> findByTournamentId(Long tournamentId) {
    log.debug("Request to get FoiFootballTable : {}", tournamentId);
    Map<FoiFootballTeam, FoiFootballTableDTO> mappedValues = new HashMap<>();
    List<FoiFootballFixture> fixtures =
foiFootballFixtureRepository.findAllByTournament_Id(tournamentId);
    for (FoiFootballFixture fixture : fixtures) {
        FoiFootballTableDTO homeTeamTable =
mappedValues.computeIfAbsent(fixture.getHomeTeam(), team ->
createEmptyTable(fixture.getHomeTeam().getName()));
        FoiFootballTableDTO awayTeamTable =
mappedValues.computeIfAbsent(fixture.getAwayTeam(), team ->
createEmptyTable(fixture.getAwayTeam().getName()));
        updateTournamentTable(homeTeamTable, awayTeamTable,
fixture.getScore().split(":"));
    }

    return
mappedValues.values().stream().sorted(Comparator.comparingInt(FoiFootballTableDTO::
getPoints).thenComparingInt(FoiFootballTableDTO::getWins).reversed()).collect(Colle
ctors.toList());
}
```

Slika 41 Implementacija generiranja tablice turnira

Ova implementacija poziva servis repozitorija odigranih utakmica te od njega dohvaća sve odigrane utakmice za određenu ligu. Nakon što je repozitorij vratio listu rezultata slijedi for petlja unutar koje se kreira mapa agregiranih podataka. Kod agregiranja se koristi struktura podataka mapa kako bi se lako ažurirao bodovni rezultat svakog tima bez potrebe za ponovim prolaskom kroz listu podataka. Ključ ove mape je tim dok je vrijednost transferni objekt tablice. Unutar for petlje kreiraju se prazna vrijednost transfernog objekta upotrebom metode createEmptyTable ako ta vrijednost ne postoji pod tim ključem. Na kraju se poziva funkcija updateTournamentTable koja ažurira obje vrijednosti transfernog objekta. Budući da klijentu nije potrebna struktura mape već samo lista poretka tablice turnira, na kraju metode mapa koja je korištena za lakšu obradu podataka se konvertira u listu podataka poredanu po broju bodova te broju pobjeda počevši od one s najvećim brojem bodova i pobjeda.

Pomoćne metode koje se spominju u implementaciji generiranja tablice rezultata su createEmptyTable i updateTournamentTable. CreateEmptyTable je metoda koju kreira početno stanje transfernog objekta i prikazana je na slici 42.

```
  private FoiFootballTableDTO createEmptyTable(String team) {
      FoiFootballTableDTO table = new FoiFootballTableDTO();
      table.setId(RandomUtils.nextLong());
      table.setWins(0);
      table.setDraws(0);
      table.setLoses(0);
      table.setPoints(0);
      table.setTeam(team);
      return table;
  }
```

Slika 42 Pomoćna metoda za kreiranje prazne tablice

Metoda updateTournamentTable je nešto složenija funkcija koja parsira rezultat utakmice te dodaje bodove timovima ovisno o rezultatu. Prikaz implementacije nalazi se na slici 43.

```
private void updateTournamentTable(FoiFootballTableDTO homeTeamTable,
FoiFootballTableDTO awayTeamTable, String[] splitScore) {
    if (!(splitScore[0].equals("-") || splitScore[1].equals("-"))) {
        int homeScore = Integer.parseInt(splitScore[0]);
        int awayScore = Integer.parseInt(splitScore[1]);

        if (homeScore > awayScore) {
            updateWinner(homeTeamTable);
            updateLoser(awayTeamTable);
        } else if (homeScore < awayScore) {
            updateLoser(homeTeamTable);
            updateWinner(awayTeamTable);
        } else {
            updateDraw(homeTeamTable);
            updateDraw(awayTeamTable);
        }
    }
}
```

Slika 43 Pomoćna metoda za ažuriranja polja tablice

Ova metoda sastoji se od 3 argumenata. Domaćina i gosta utakmice, te polja rezultata gdje je string na poziciji nula broj golova domaćina a string na poziciji jedan broj golova gosta. Na početku metoda provjerava je li broj golova iti jedne momčadi jednak znaku – te ako je ovaj rezultat se ignorira zato što utakmica nije odigrana. Ako je rezultat različit od toga, rezultat se pretvara u broj te uspoređuje u ifologiji. U prvom grananju pobjednik je domaćin, u drugom gost a u trećem rezultatu utakmica je bila neriješena. Ažuriranje podataka tablice poziva se upotrebom pomoćnih metoda: updateWinner, updateLoser i updateDraw koji ažuriraju broj

bodova te broj pobijeđenih /izgubljenih/ izjednačenih utakmica. Implementacija ovih metoda prikazana je na slici ispod.

```
private void updateWinner(FoiFootballTableDTO table) {
    table.setWins(table.getWins() + 1);
    table.setPoints(table.getPoints() + 3);
}

private void updateLoser(FoiFootballTableDTO table) {
    table.setLoses(table.getLoses() + 1);
}

private void updateDraw(FoiFootballTableDTO table) {
    table.setDraws(table.getDraws() + 1);
    table.setPoints(table.getPoints() + 1);
}
```

Slika 44 Pomoćne metode za ažuriranje tablice ovisno o rezultatu

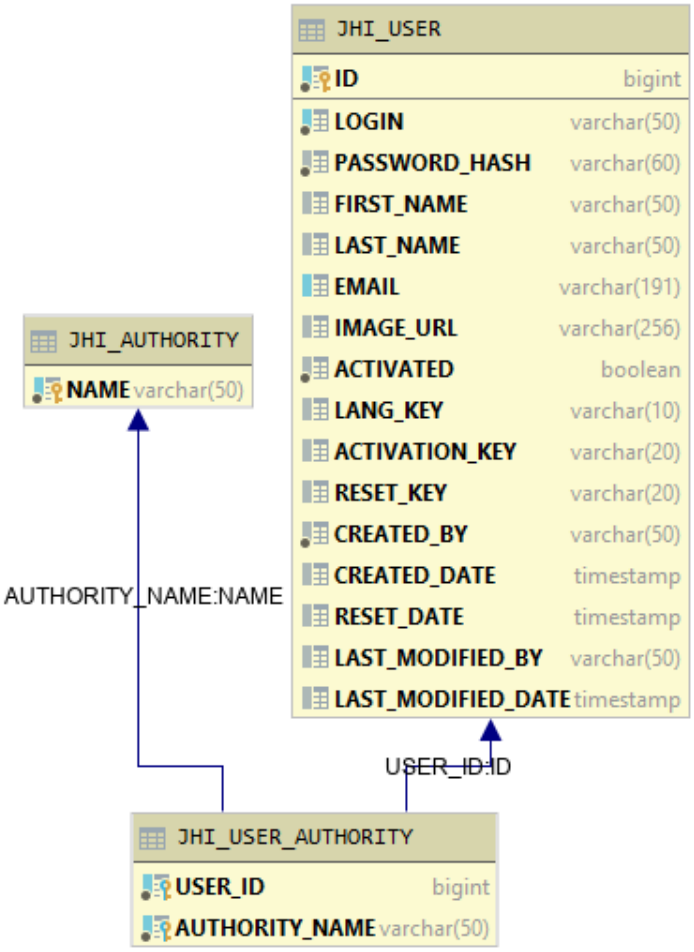
4.2.3. Autentikacijski mikroservis

Autentikacijski mikroservis je napravljen od tri glavna dijela:

- Model i baza podataka
- Unutarnja autentikacijska logika
- API REST servisi

4.2.3.1. Model i baza podataka

Model i baza podataka autentikacijskog mikroservisa je vrlo jednostavna a sastoji se od dvije tablice unutar baze podataka: tablici korisnika i tablici tipa korisnika baze podataka. Struktura baze podataka može se vidjeti na slici ispod.



Slika 45 Prikaz modela mikroservisa za autentikaciju

Tablica korisnika sadrži sljedeće informacije o korisniku:

- id korisnika
- korisničko ime (login) – korisničko ime je jedinstveno
- lozinka (password_hash)
- ime
- prezime
- email – email je jedinstven unutar baze
- slika
- aktivacija – istina ako je korisnik prihvatio aktivacijski email
- jezik – korisnikov odabrani jezik
- aktivacijski ključ – provjerava se kod aktivacije, ključ se mora podudarati kako bi se korisnikov račun aktivirao
- ključ za resetiranje lozinke – provjerava se prilikom resetiranje lozinke.
- kreiran od
- datum kreiranja
- datum resetiranja
- zadnje modificiran od
- zadnji datum modificiranja

Tablica tipa korisnika je vrlo jednostavna te se sastoji samo od imena tipa što je ujedno i primarni ključ tablice. Ove dvije tablice su povezane M:N vezom kako bi se svakom korisniku mogla pridružiti određen tip.

4.2.3.2. Unutarnja autentikacijska logika

Unutarnja logika mikroservisa za autentikaciju sastoji se od dva glavna servisa od kojih je jedan namijenjen radom s elektroničkom poštom, a drugi radom s korisnicima.

Prva bitna metoda unutar servisa za rad s korisnicima bavi se kreiranjem novog korisnika. Ova metoda je vrlo jednostavna te prihvaća transferni objekt te podatke iz njega zapisuje u model. Ova implementacija vidljiva je na slici 46.

```
public User registerUser(UserDTO userDTO, String password) {

userRepository.findOneByLogin(userDTO.getLogin().toLowerCase()).ifPresent(existingUser ->
{
        boolean removed = removeNonActivatedUser(existingUser);
        if (!removed) {
            throw new UsernameAlreadyUsedException();
        }
    });
    userRepository.findOneByEmailIgnoreCase(userDTO.getEmail()).ifPresent(existingUser ->
{
        boolean removed = removeNonActivatedUser(existingUser);
        if (!removed) {
            throw new EmailAlreadyUsedException();
        }
    });
    User newUser = new User();
    String encryptedPassword = passwordEncoder.encode(password);
    newUser.setLogin(userDTO.getLogin().toLowerCase());
    newUser.setPassword(encryptedPassword);
    newUser.setFirstName(userDTO.getFirstName());
    newUser.setLastName(userDTO.getLastName());
    newUser.setEmail(userDTO.getEmail().toLowerCase());
    newUser.setImageUrl(userDTO.getImageUrl());
    newUser.setLangKey(userDTO.getLangKey());
    newUser.setActivated(false);
    newUser.setActivationKey(RandomUtil.generateActivationKey());
    Set<Authority> authorities = new HashSet<>();
    authorityRepository.findById(AuthoritiesConstants.USER).ifPresent(authorities::add);
    newUser.setAuthorities(authorities);
    userRepository.save(newUser);
    this.clearUserCaches(newUser);
    log.debug("Created Information for User: {}", newUser);
    return newUser;
}
```

Slika 46 Implementacija metode za kreiranje korisnika

Budući da je svaki korisnik nakon registracije neaktivan potrebna je metoda za aktivaciju korisnika. Ova metoda se poziva prilikom korisnikova klika na aktivacijski link. Na slici 47 prikazana je implementacija aktivacije.

```
public Optional<User> activateRegistration(String key) {
    log.debug("Activating user for activation key {}", key);
    return userRepository.findOneByActivationKey(key)
        .map(user -> {
            // activate given user for the registration key.
            user.setActivated(true);
            user.setActivationKey(null);
            this.clearUserCaches(user);
            log.debug("Activated user: {}", user);
            return user;
        });
}
```

Slika 47 Implementacija aktiviranja korisnika

Aktivacija je implementirana na sljedeći način. Metoda prima argument aktivacijskog ključa te pronalazi korisnika u bazi s jednakim aktivacijskim ključem. Ako korisnik postoji, sustav ga aktivira postavljajući zastavu aktivacije na istinu te se aktivacijski ključ briše kako se isti proces ne bi mogao ponavljati.

Ovaj servis također sadrži metode za resetiranje šifre. Proces za resetiranje je implementiran koristeći dvije metode. Prva metoda započinje proces resetiranja tako da kreira ključ za resetiranje te zapisuje datum zahtjeva resetiranja. Implementacija te metode prikazana je na slici 48.

```
public Optional<User> requestPasswordReset(String mail) {
    return userRepository.findOneByEmailIgnoreCase(mail)
        .filter(User::getActivated)
        .map(user -> {
            user.setResetKey(RandomUtil.generateResetKey());
            user.setResetDate(Instant.now());
            this.clearUserCaches(user);
            return user;
        });
}
```

Slika 48 Implementacija zahtjeva za resetiranje lozinke

Druga metoda u procesu pronalazi korisnika s ključem resetiranja te postavlja korisniku novi lozinku ako korisnik s tim ključem resetiranja postoji i ako ključ nije stariji od jednog dana. Implementacijski kod je prikazan na slici 49.

```
public Optional<User> completePasswordReset(String newPassword, String key) {
    log.debug("Reset user password for reset key {}", key);
    return userRepository.findOneByResetKey(key)
        .filter(user ->
user.getResetDate().isAfter(Instant.now().minusSeconds(86400)))
        .map(user -> {
            user.setPassword(passwordEncoder.encode(newPassword));
            user.setResetKey(null);
            user.setResetDate(null);
            this.clearUserCaches(user);
            return user;
        });
}
```

Slika 49 Implementacija resetiranja lozinke

Osim ovih implementacija servis sadrži i druge metode za kreiranje, brisanje i ažuriranje korisnika no zbog jednostavnosti implementacije neće biti posebno opisane.

Drugi servis unutarnje logike sadrži standardnu logiku za slanje elektroničke pošte a funkcionalnosti ovog servisa koristi se prilikom slanja aktivacijskog maila te maila za resetiranje lozinke.

Implementacija za slanje elektroničke pošte sastoji se od dijela za slanje te predloška koji se koristi za strukturiranje. Metoda za slanje elektroničke pošte prihvaća tri argumenta: prvi argument je korisnik i koristi se za kontekst pošte, drugi argument je ime predloška dok je treći argument lokalizirani ključ naslova emaila. Implementacija ja prikazana na slici 50.

```
@Async
public void sendEmailFromTemplate(User user, String templateName, String titleKey)
{
    Locale locale = Locale.forLanguageTag(user.getLangKey());
    Context context = new Context(locale);
    context.setVariable(USER, user);
    context.setVariable(BASE_URL, jHipsterProperties.getMail().getBaseUrl());
    String content = templateEngine.process(templateName, context);
    String subject = messageSource.getMessage(titleKey, null, locale);
    sendEmail(user.getEmail(), subject, content, false, true);
}
```

Slika 50 Implementacija slanja elektroničke pošte

Predlošci za email su vrlo jednostavni te koriste tehnologiju Thymeleaf [28] za njihovu implementaciju. Thymeleaf je jedan od popularnijih serverskih tehnologija za predloške te omogućava kreiranje HTML predloška čiji se sadržaj mijenja ovisno o vrijednosti varijable koje su mu proslijeđene. Primjer predloška elektroničke pošte za aktivaciju prikazan je na slici 51.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" th:lang="${#locale.language}">
    <head>
        <title th:text="#{email.activation.title}">JHipster activation</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <link rel="shortcut icon" th:href="@{|${baseUrl}/favicon.ico|}" />
    </head>
    <body>
        <p th:text="#{email.activation.greeting(${user.login})}">
            Dear
        </p>
        <p th:text="#{email.activation.text1}">
            Your JHipster account has been created, please click on the URL below
to activate it:
        </p>
        <p>
            <a
th:with="url=(@{|${baseUrl}/account/activate?key=${user.activationKey}|})"
th:href="${url}"
            th:text="${url}">Activation link</a>
        </p>
        <p>
            <span th:text="#{email.activation.text2}">Regards, </span>
            <br/>
            <em th:text="#{email.signature}">JHipster.</em>
        </p>
    </body>
</html>
```

Slika 51 Predložak za aktivaciju

4.2.3.3. API REST servisi

API dio implementacije služi za povezivanje drugih mikroservisa s autentikacijskim mikroservisom. API dio dijelimo na dio za autentikaciju te dio za kreiranje korisnika. Klasa UserResource služi za kreiranje korisnika dok se klasa AccountResource koristi za proces autentikacije te ažuriranja informacija trenutnog korisnika.

Klasa UserResource je standardan CRUD API REST servis te se sastoji od metoda za kreiranje korisnika, ažuriranje i brisanje korisnika te metoda za dohvat jednog ili više korisnika iz baze podataka. Metoda za ažuriranje korisnika metoda unutar klase može se vidjeti na slici 52.

```
@PutMapping("/users")
@PreAuthorize("hasRole(\"" + AuthoritiesConstants.ADMIN + "\")")
public ResponseEntity<UserDTO> updateUser(@Valid @RequestBody UserDTO userDTO) {
    log.debug("REST request to update User : {}", userDTO);
    Optional<User> existingUser =
userRepository.findOneByEmailIgnoreCase(userDTO.getEmail());
    if (existingUser.isPresent() &&
(!existingUser.get().getId().equals(userDTO.getId()))) {
        throw new EmailAlreadyUsedException();
    }
    existingUser =
userRepository.findOneByLogin(userDTO.getLogin().toLowerCase());
    if (existingUser.isPresent() &&
(!existingUser.get().getId().equals(userDTO.getId()))) {
        throw new LoginAlreadyUsedException();
    }
    Optional<UserDTO> updatedUser = userService.updateUser(userDTO);

    return ResponseUtil.wrapOrNotFound(updatedUser,
        HeaderUtil.createAlert(applicationName, "userManagement.updated",
userDTO.getLogin()));
}
```

Slika 52 Metoda za ažuriranje korisnika

Specifičnost ove implementacije je ta što ovoj metodi mogu pristupiti samo autentificirani korisnici koji su administratori u sustavu. Autorizacija unutar spring-a je vrlo jednostavna te je potrebno koristiti anotaciju PreAuthorize kako bi se ograničio pristup određenoj metodi. Sama implementacija je jednostavna te provjerava postoji li u bazi već korisnik s istom email adresom i korisničkim imenom, ako ne postoji korisnikove informacije se ažuriraju unutar baze podataka.

Klasa AccountResource također ima CRUD operacije no unutar ove klase operacije se odvijaju na trenutnom korisniku. Tako na primjer operacija account ažurira informacije

trenutnog korisnika za razliku od operacije na slici 46 gdje administrator ažurira informacije bilo kojeg korisnika unutar sustava.

```
@PostMapping("/account")
public void saveAccount(@Valid @RequestBody UserDTO userDTO) {
    String userLogin = SecurityUtils.getCurrentUserLogin().orElseThrow(() -> new
AccountResourceException("Current user login not found"));
    Optional<User> existingUser =
userRepository.findOneByEmailIgnoreCase(userDTO.getEmail());
    if (existingUser.isPresent() &&
(!existingUser.get().getLogin().equalsIgnoreCase(userLogin))) {
        throw new EmailAlreadyUsedException();
    }
    Optional<User> user = userRepository.findOneByLogin(userLogin);
    if (!user.isPresent()) {
        throw new AccountResourceException("User could not be found");
    }
    userService.updateUser(userDTO.getFirstName(), userDTO.getLastName(),
userDTO.getEmail(),
        userDTO.getLangKey(), userDTO.getImageUrl());
}
```

Slika 53 Ažuriranje informacije trenutnog korisnika

Na slici 53. uočljiva je razlika od implementacije metode za slike 46. Ova metoda nema anotaciju preauthorise budući da korisnik ažurira sam svoje podatke. Ostala implementacija je vrlo slična no razlika je još jedino vidljiva u prvom koraku gdje se dohvaća trenutni korisnik upotrebom klase SecurityUtils.

Provjera autentikacije korisnika izvodi se unutar metode isAuthenticated. Implementacija ove metode prikazana je na slici 54.

```
@GetMapping("/authenticate")
public String isAuthenticated(HttpServletRequest request) {
    log.debug("REST request to check if the current user is authenticated");
    return request.getRemoteUser();
}
```

Slika 54 Prikaz metode za autentikaciju

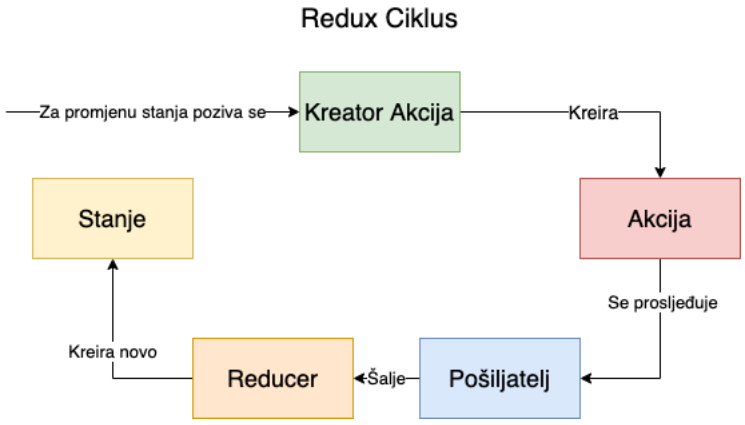
Implementacija je jednostavna te poziva getRemoteUser na zahtjevu korisnika. Ova vrijednost će biti null ako korisnik nije autentificiran.

4.2.4. Pristupni mikroservis

Pristupni mikroservis je servis na koji se spaja korisnik i koji sadrži cjelokupni korisnički dio implementacije aplikacije.

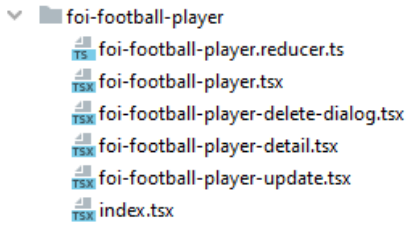
4.2.4.1. Implementacija korisničke strane

Kao što je već spomenuto u poglavlju 4.1.5 , korisnička strana aplikacije implementirana je pomoću tehnologije React. Cjelokupna implementacija implementirana je koristeći jednak uzorak implementacije za React Redux aplikacije. Sva implementacija ažuriranja sustava prati sljedeći uzorak dizajna. Svaka komponenta definira metode za obradu određenog događaja (klik, potvrda, itd..). Ako određeni događaj zahtjeva ažuriranje stanja aplikacije potrebno je poslati akciju za ažuriranjem u sustav pozivom funkcije za kreiranje akcija (action creator). Poslana akcija se obrađuje u specificiranom mjestu za obradu te akcije unutar reducera te se nakon obrade ažurira stanje cjelokupnog sustava. Prikaz dijagrama redux ciklusa prikazan je na slici ispod.



Slika 55 Redux ciklus

Primjeri implementacije korisničke strane bit će prikazana na komponenti igrača. Struktura mape komponente igrača prikazana je na slici 56.



Slika 56 Struktura React komponente

Datoteka index.tsx sadrži putanje kojima pripada ova komponenta. Unutar ove datoteke sadržane su sve URL putanje te komponente koje se trebaju učitati kada je određena ruta unutar URL-a.

```
const Routes = ({ match }) => (
  <>
    <Switch>
      <ErrorBoundaryRoute exact path={`${match.url}/new`}
component={FoiFootballPlayerUpdate} />
      <ErrorBoundaryRoute exact path={`${match.url}/:id/edit`}
component={FoiFootballPlayerUpdate} />
      <ErrorBoundaryRoute exact path={`${match.url}/:id`}
component={FoiFootballPlayerDetail} />
      <ErrorBoundaryRoute path={match.url}
component={FoiFootballPlayer} />
    </Switch>
    <ErrorBoundaryRoute path={`${match.url}/:id/delete`}
component={FoiFootballPlayerDeleteDialog} />
  </>
);

export default Routes;
```

Slika 57 Implementacija putanja za komponentu igrač

Slika 57 prikazuje komponentu switch react routera unutar koje su definirane sve putanje za komponentu igrača. Unutar implementacije se vidi da na primjer komponenta FoiFootballPlayer update pripada putanjama /new i /:id/edit s varijabilnim prefiksom koji je definiran unutar rutera više razine.

Jedan od primjera ekrana je ekran za ažuriranje. Ovaj ekran sadrži logiku za prikaz informacije te logiku za kreiranje novog igrača ako je na ekran pristupljeno s putanje /new odnosno logike za ažuriranje igrače ako je ekranu pristupljeno s putanje /:id/edit.

```
componentDidMount() {
  if (this.state.isNew) {
    this.props.reset();
  } else {
    this.props.getEntity(this.props.match.params.id);
  }

  this.props.getFoiFootballPositions();
  this.props.getFoiFootballTeams();
}
```

Slika 58 ComponentDidMount - ažuriranja igrača

Slika 58 prikazuje dio logike za ažuriranje odnosno kreiranje igrača. ComponentDidMount je nativna metoda životnog ciklusa react komponente koja se poziva automatski nakon prvog

prikaza komponente. Unutar metode vidljivo je grananje i drugačija implementacija kod novog igrača te ažuriranja postojećeg igrača. Metoda koja se poziva kod spremanja obrasca nakon kreiranja ili ažuriranja je metoda saveEntity prikazana na slici 59.

```
saveEntity = (event, errors, values) => {
  if (errors.length === 0) {
    const { foiFootballPlayerEntity } = this.props;
    const entity = {
      ...foiFootballPlayerEntity,
      ...values
    };

    if (this.state.isNew) {
      this.props.createEntity(entity);
    } else {
      this.props.updateEntity(entity);
    }
  }
};
```

Slika 59 Prikaz implementacije spremanja obrasca

Ova metoda poziva createEntity metodu odnosno updateEntity metodu ako u obrazcu ne postoje greške te tim metodama prosljeđuje novo kreirani odnosno ažurirani entitet. CreateEntity i updateEntity su funkcije za kreiranje akcija te kreiraju akciju koja se obrađuje u reduceru.
Primjer funkcije za kreiranje akcije prikazan je na slici 60.

```
export const updateEntity: ICrudPutAction<IFoiFootballPlayer> = entity =>
async dispatch => {
  const result = await dispatch({
    type: ACTION_TYPES.UPDATE_FOIFOOTBALLPLAYER,
    payload: axios.put(apiUrl, cleanEntity(entity))
  });
  dispatch(getEntities());
  return result;
};
```

Slika 60 Prikaz kreiranja akcije ažuriranja

Ova funkcija obrađuje asinkroni zahtjev za ažuriranjem korisnika. Nakon što je ova funkcija pozvana ona kreira te šalje akciju za ažuriranje korisnika a kao sadržaj akcije prosljeđuje se rezultat PUT zahtjeva na serveru. Na kraju nakon ažuriranja prosljeđuje se još jedna akcija upotrebom naredbe dispatch za dohvat svih entiteta igrača.
Ove akcije obrađuje reducer koji je implementiran kao switch koji obrađuje svaku akciju na svoj specifičan način.

```
case REQUEST(ACTION_TYPES.CREATE_FOIFOOTBALLPLAYER):
case REQUEST(ACTION_TYPES.UPDATE_FOIFOOTBALLPLAYER):
case REQUEST(ACTION_TYPES.DELETE_FOIFOOTBALLPLAYER):
  return {
    ...state,
    errorMessage: null,
    updateSuccess: false,
    updating: true
  };
```

Slika 61 Implementacija reducera

Slika 61 prikazuje dio implementacije reducera za sve zahtjeve. Ova akcija kopira prethodno postojeće stanje te postavlja stanje sustava u stanje ažuriranje (updating:true) zato što je zahtjev za brisanjem, ažuriranjem ili kreiranjem tek počeo. Slika 62 prikazuje logiku reducera kada je zahtjev za kreiranje ili ažuriranje uspješno završen.

```
case SUCCESS(ACTION_TYPES.CREATE_FOIFOOTBALLPLAYER):
case SUCCESS(ACTION_TYPES.UPDATE_FOIFOOTBALLPLAYER):
  return {
    ...state,
    updating: false,
    updateSuccess: true,
    entity: action.payload.data
  };
```

Slika 62 Implementacija obrade uspješne akcije

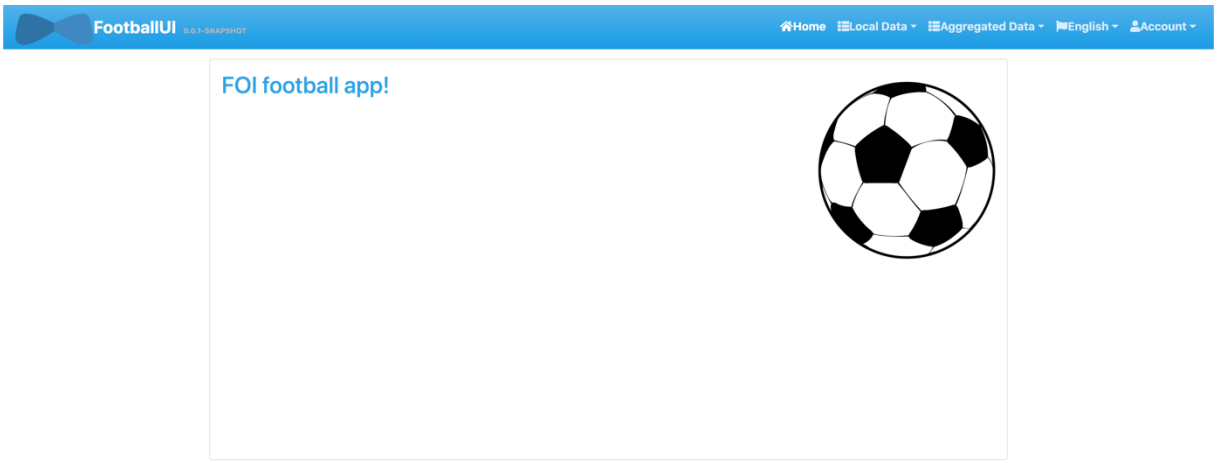
Vidljivo je da u ovom slučaju sustav više nije u stanju ažuriranja te je entitet ažuriran s novom vrijednošću.

4.3. Opis rada implementiranog sustava

Unutar ovog poglavlja ukratko će biti prikazan implementirani sustav te način na koji se on koristi. Sustav se može koristiti iz perspektive običnog neregistriranog korisnika te iz perspektive administratora. Ovisno o roli koju korisnik koristi sustav se ponaša drugačije.

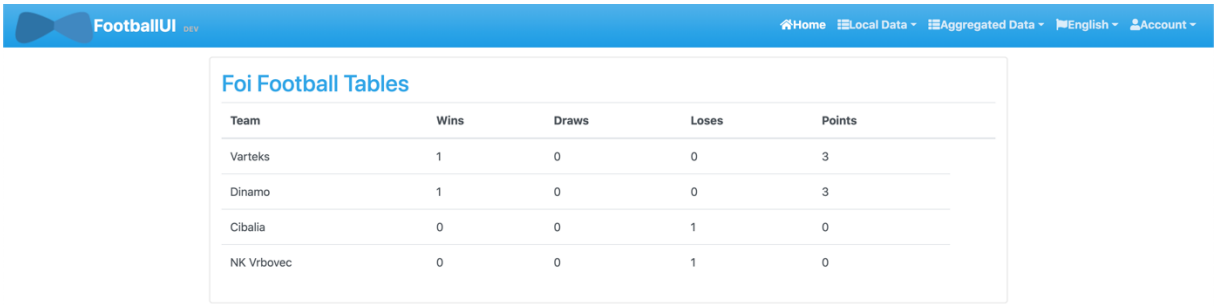
4.3.1. Neregistrirani korisnik

Kada neregistrirani korisnik pristupi aplikaciji prikazat će se početni ekran s porukom dobrodošlice te će na vrhu ekrana biti vidljiva navigacija za kretanje aplikacijom. Početni ekran prikazan je na slici 63.



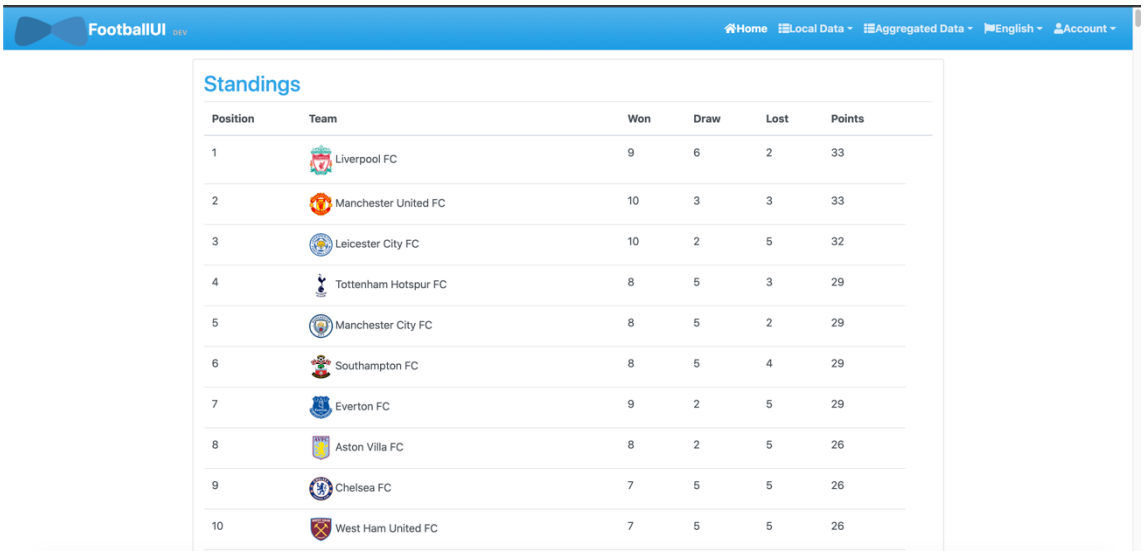
Slika 63 Početna stranica aplikacije

Kao što je na slici 64 vidljivo navigacija neregistriranog korisnika ima sljedeće opcije: početna stranica, lokalni podaci, agregirani podaci, promjena jezika, profil. Pod opcijom lokalni podaci korisnik može vidjeti sve lokalno ažurirane podatke različitih liga. Ti podaci uključuju sve odigrane utakmice u svim turnirima te rezultatske tablice grupirane prema turnirima.



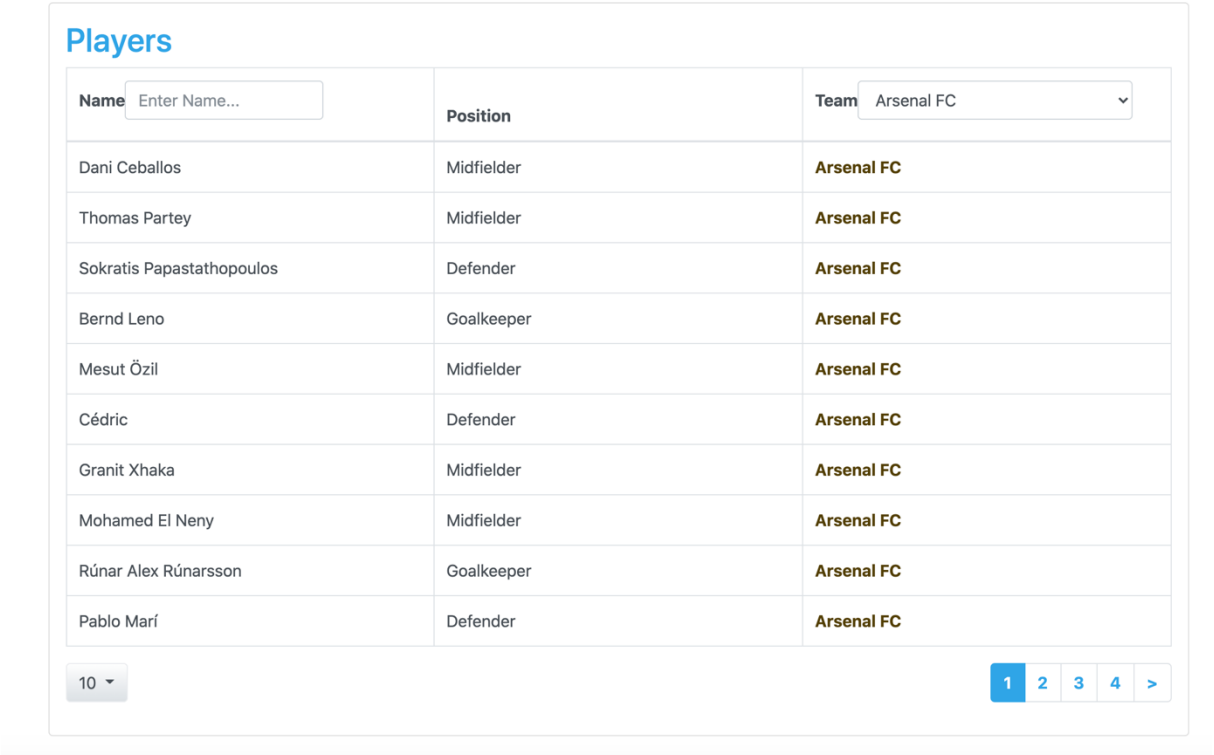
Slika 64 Prikaz tablice lokalno ažuriranog turnira

Osim toga korisnik može pregledavati sve automatski agregirane podatke mikroservisa za agregiranje. Ti podaci uključuju sve igrače lige, utakmice, klubove, natjecanja i slično. Agregirani podaci koje mikroservis agregira u ovom slučaju su podaci engleske premier lige.



Slika 65 Tablica agregiranih podataka Premier lige

Za ekrane gdje tablice sadrže veći broj podataka implementirano je filtriranje te straničenje kako bi pregled podataka bio lakši.



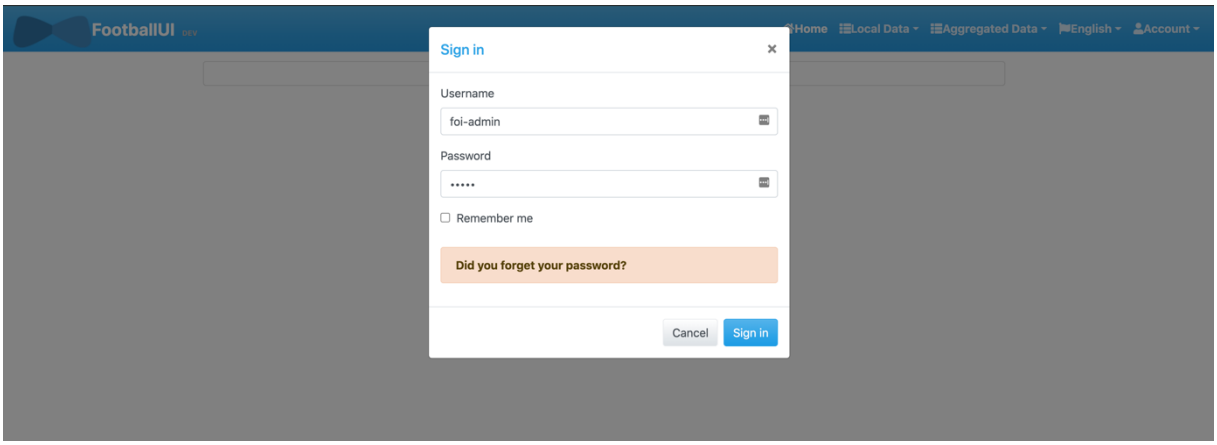
Slika 66 Filtriranje i straničenje

Osim ovih funkcionalnosti, neregistrirani korisnik još ima opcije promjene jezika unutar aplikacije te prijave u sustav pod opcijom profil unutar navigacije.

4.3.2. Administrator

Administrator kao korisnik ima puno više mogućnosti od neregistriranog korisnika. Osim pregleda podataka administrator može dodavati nove podatke i izmjenjivati postojeće. Osim toga administrator ima pregled svih entiteta u aplikacije te prikaz stanja sustava u administracijskom dijelu navigacije.

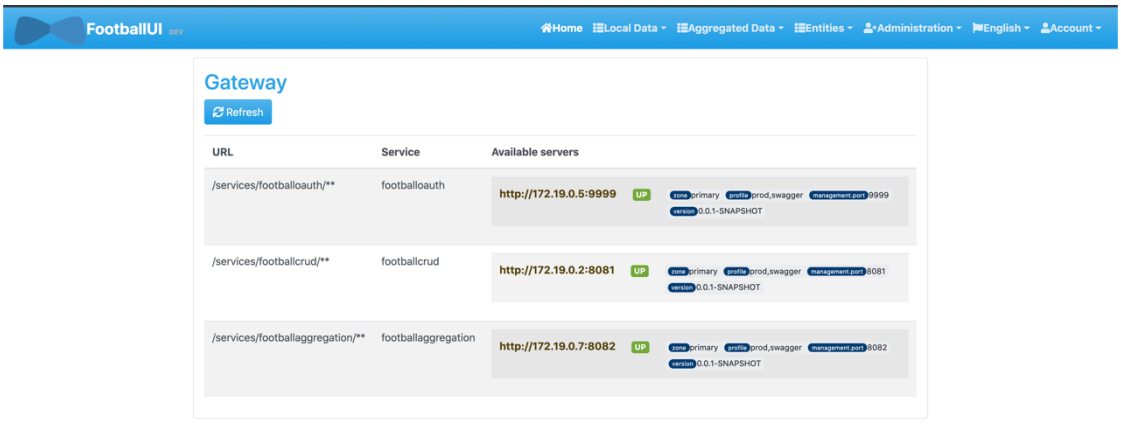
Kako bi se pokazao administracijski dio aplikacije te njegove opcije korisnik se prvo treba autentificirati u aplikaciju korištenjem ekrana za prijavu.



Slika 67 Ekran za prijavu

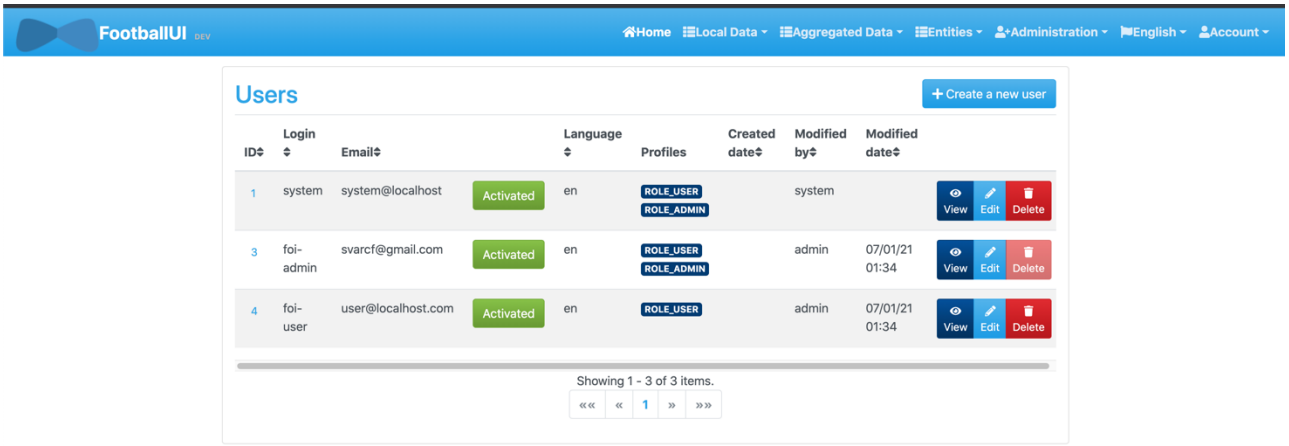
Administracijski dio aplikacije administratoru nudi uvid u cjelokupni sustav aplikacije. U sljedećem dijelu bit će navedene najbitniji administratorski ekrani unutar aplikacije.

Prva opcija u administracijskom dijelu prikazuje sve servise koji se koriste unutar sustava. Ovaj ekran detaljno prikazuje informacije o svim servisima kao što su: URL servisa, ime servisa te IP adresa i port. Ovaj ekran je vrlo koristan kada je potrebno provjeriti jesu li svi servisi u funkciji.



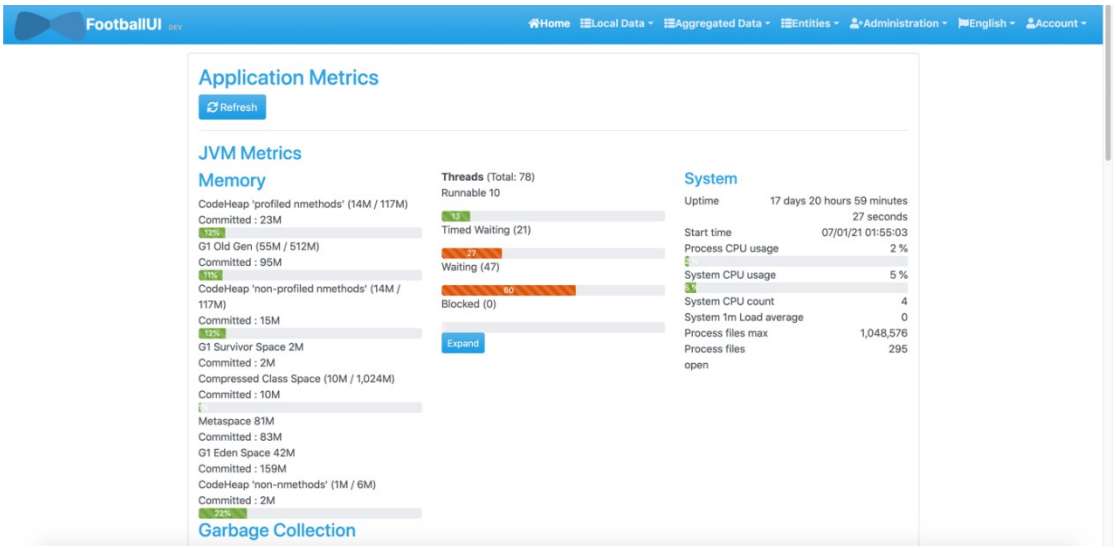
Slika 68 Ekran za prikaz servisa u aplikaciji

Drugi administracijski ekran zadužen je za upravljanje registriranim korisnicima aplikacije. U ovom ekranu administratori mogu kreirati nove korisnike, pobrisati postojeće, deaktivirati ih te im mijenjati postavke.



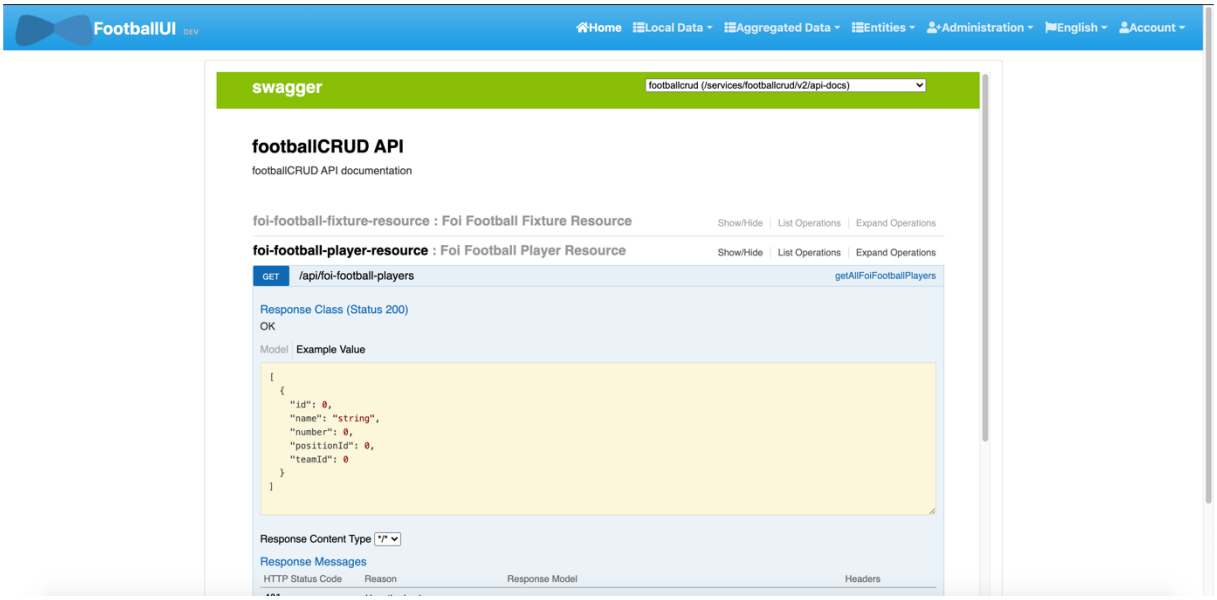
Slika 69 Ekran za upravljanje korisnicima

Treći ekran prikazuje metriku stanja sustava. Na ovom ekranu moguće je vidjeti statistiku rada sustava kao što su korištenje memorije i CPU-a, broj dretvi u sustavu, prosječno vrijeme odgovora svake krajnje točke sustava te mnoge druge informacije koje olakšavaju praćenje i analiziranje rada sustava.



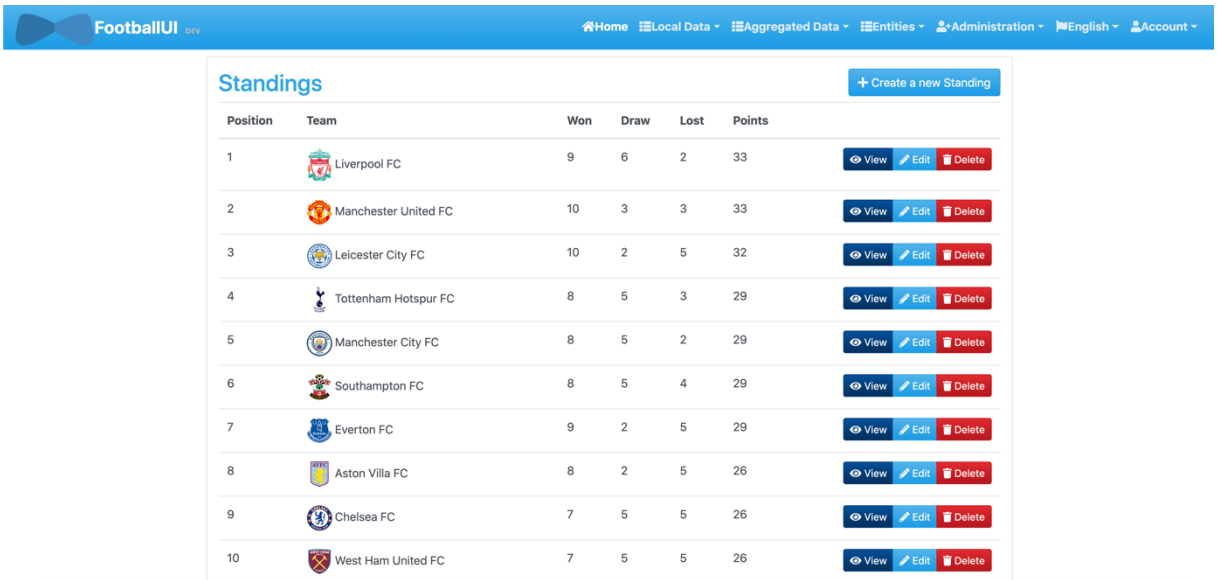
Slika 70 Metrika sustava

Četvrti administratorski ekran je vrlo koristan za druge programere budući da sadrži sve krajnje točke svih servisa te strukturu JSON-a objekta koju određena krajnja točka šalje. Ovaj ekran je implementiran upotrebom Swagger jezika za opis REST sučelja [29].



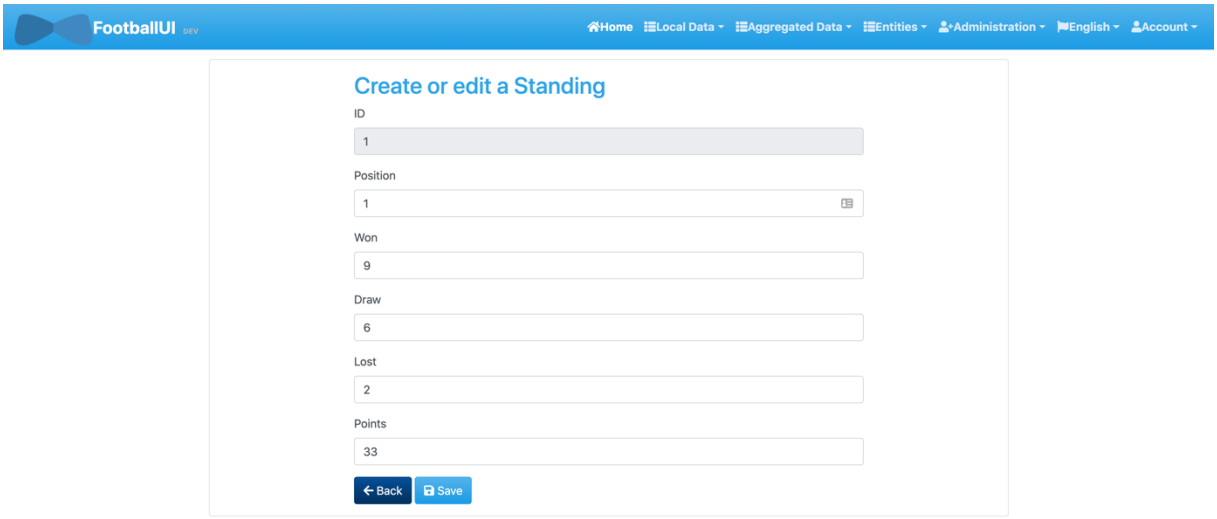
Slika 71 Prikaz REST sučelja aplikacije upotrebom Swagger jezika

Osim analize stanja sustava te upravlja korisnicima administrator također ažurira i te dodaje nove podatke u sustav. Ažuriranje podataka omogućeno je kroz gumbe koji se prikazuju samo kada sustav prepozna da je trenutni korisnik u aplikaciji administrator.



Slika 72 Prikaz gumbi za ažuriranje podataka

Na slici 72 s desne strane moguće je vidjeti gumbe za izmjenu podataka. Administrator ima tri opcije a to su detaljni pregled podatka, ažuriranje te brisanje.
Klikom na gumb ažuriranje otvara se ekran za ažuriranje podataka. Isti ekran otvorit će se i za kreiranje podataka no u slučaju kreiranja podaci neće biti unaprijed popunjeni te će sustav slati podatke upotrebom POST zahtjeva umjesto PUT zahtjeva korištenog za ažuriranje.



Slika 73 Ekran za ažuriranje podataka

5. Zaključak

Mikroservisna arhitektura noviji je i moderan pristup razvoju računalnih sustava. Ova arhitektura je sve popularnija danas zbog velike nepovezanosti sustava, lakog skaliranja te mogućnosti korištenja različitih tehnologija. Sve ove pogodnosti posebno su bitne danas kada je većina aplikacija na internetu dostupno milijunima ljudi.

No iako ovaj tip arhitekture nudi mnoge prednosti, ona nije rješenje za sve slučajeve te je potrebno dobro razmisliti prilikom implementiranja sustava koristeći ovu arhitekturu. Razlog tomu je veća kompleksnost implementacije koja zahtjeva stručnjake upoznate s ovim novim načinom rada. Osim toga veća kompleksnost zahtjeva i duže vrijeme implementacije. Zbog toga se ova arhitektura preporučuje kod implementacije velikih sustava koji će se dugo nadograđivati te će biti korišteni od velikog broja ljudi.

Izradom praktičnog djela diplomskog rada viđene su prednosti i nedostaci ove arhitekture. Jedan od većih problema koji je nastao upotrebom mikroservisne arhitekture je cijena servera te velika početna potreba za resursima bez obzira na veličinu same aplikacije. Kako bi se aplikacija iz praktičnog djela mogla izvoditi zahtijevala je ili veliki broj odvojenih servera na koji se mikroservisi postavljaju ili jedan vrlo jak server na koji se svi mikroservisi mogu postaviti. Jedan od većih razlog je početna cijena resursa svih odvojenih baza podataka za svaki mikroservis. No, s druge strane velika prednost korištenja ovakve arhitekture je bila odvojenost sustava kod postavljanja aplikacija na server zato što bi se razvoj i postavljanje jednog mikroservisa odvijao bez rizika na ostale mikroservise. Upotreba monolitne arhitekture u ovom slučaju bi bila jeftinija te zahtijevala manje resursa, no postavljanje na server i razvoj aplikacije bi nosio određen rizik na rad ostalih dijelova sustava.

Arhitektura računalnog sustava vrlo je bitna stavka kod svakog razvoja novog sustava. Poznavanje različitih tipova implementacije sustava, njihovih prednosti i nedostataka će uvelike pomoći pri odabiru prave arhitekture za zadatak što će doprinijeti uspješnosti implementacije sustava.

Popis literature

1. M. Richards (2015) *Software Architecture Patterns* (1. Izd) USA: O'Reilly Media
2. R. C. Martin, The Clean Architecture (2012.) https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
3. J. Akhtar (2018) *Microservices Introduction (Monolithic vs. Microservice Architecture)* Preuzeto 12.07.2019. sa https://dzone.com/articles/microservices-1-introduction-monolithic-vs-microse
4. R. Gnatyk, Monolitna arhitektura [Slika] (2018.) Preuzeto 19.07.2019. sa https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/
5. S. Newman (2015). *Building Microservices* (1. Izd.) USA: O'Reilly Media
6. M. Fowler, Monoliths and Microservices [Slika] (2014.) Preuzeto 19.07.2019. sa https://martinfowler.com/articles/microservices.html
7. J. Hurwitz, R. Bloor, C. Baroudi, M. Kaufman (2007.) *Service Oriented Architecture For Dummies* (1. Izd) USA: Wiley Publishing, Inc
8. Pegl Service Oriented Application [Slika] (bez dat.) Preuzeto 19.07.2019. sa https://www.ibm.com/support/knowledgecenter/en/SSMQ79_9.5.1/com.ibm.egl.pg.doc/topics/pegl_serv_overview.html
9. S. Takale (2018). *Advantages and Disadvantages of Service-oriented Architecture* Preuzeto 12.07.2019. s https://techspirited.com/advantages-disadvantages-of-service-oriented-architecture-soa
10. Diagram of Enterprise Service Bus [Slika] (bez dat.) Preuzeto 19.07.2019. sa https://www.it.ucla.edu/news/what-esb
11. Description of monolithic vs microservice [Slika] (bez dat.) Preuzeto 19.07.2019. sa https://docs.oracle.com/en/solutions/learn-architect-microservice/index.html
12. C. Richardson, Kompozitnost mikroservisne arhitekture [Slika] (2019.) Preuzeto 19.07.2019. sa https://microservices.io/patterns/microservices.html
13. E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software USA: Addison-Wesley, 2003.
14. B. Skurrie, M. Boudreau, T. Jones *Pact* (Verzija 2.0) (2021) [Testni alat] Pristupano: https://docs.pact.io

15. Elastic *Kibana* (Verzija 7.11.0) (2021). [Program za vizualizaciju podataka] Pristupano: https://www.elastic.co/kibana
16. Orbitz Worldwide, Inc. *Graphite* (Verzija 1.2.0) (2021). [Alat za prikupljanje i praćenje podataka o radu sustava] Pristupano: https://graphiteapp.org
17. *Metrics* (Verzija 4.1.2)(2017). [Alat za praćenje rada sustava] Pristupano: https://metrics.dropwizard.io/4.1.2/index.html
18. OpenID Foundation *OpenId* (Verzija 2.0) (2007.) [Decentralizirani protokol za autentifikaciju] Pristupano: https://openid.net
19. Pivotal Software, Inc *Spring Boot* (Verzija 2.4.2) (2021) [Aplikacijski okvir] Pristupano: https://spring.io/projects/spring-boot
20. Julian Dubois i sur. *JHipster UAA* (Verzija 6.10.5) (2021) [Servis za autentifikaciju i autorizaciju mikroservisa] Pristupano: https://www.jhipster.tech/using-uaa/
21. Julian Dubois i sur. *JHipster* (Verzija 6.8.0) (2020) [Generator web aplikacija] Pristupano: https://www.jhipster.tech/
22. Pivotal Software, Inc *Spring Cloud Config* (Verzija 3.0.2) (2021) [Tehnologija za eksternu konfiguraciju distribuiranih sustava] Pristupano: https://spring.io/projects/spring-cloud-config
23. Pivotal Software, Inc *Spring Cloud Netflix* (Verzija 3.0.1) (2021) [Integracija Netflixovih programa otvorenog koda sa Spring razvojnim okvirom] Pristupano: https://spring.io/projects/spring-cloud-netflix
24. Facebook i sur. *React* (Verzija 17.0.1) (2020) [JavaScript bilioteka za izgradnju korisničkog sučelja] Pristupano: https://reactjs.org/
25. React Training i sur. *React Router* (Verzija 5.2.0) (2020) [Kolekcija React komponenti za navigaciju] Pristupano: https://reactrouter.com/
26. Dan Abramov i sur. *React Redux* (Verzija 7.2.2) (2020) [React implementacija skladišta stanja] Pristupano: https://react-redux.js.org/
27. Red Hat, Inc. *Hibernate* (Verzija 5.4.27) (2021) [Objektno relacijski alat za mapiranje] Pristupano: https://hibernate.org/
28. Daniel Fernandez *Thymeleaf* (Verzija 3.0.11) (2018.) [Java tehnologija za izradu predložaka] Pristupano: https://www.thymeleaf.org/
29. SmartBear Software *Swagger* (Verzija 2.1.6) (2020.) [Jezik opisa sučelja REST API-a] Pristupano: https://swagger.io/

Popis slika

Popis tablica