Izrada sustava za upravljanje skladištem na bazi mikroservisne arhitekture

Tocko, Filip

Master's thesis / Diplomski rad

2023

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

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:534236

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Tocko

IZRADA SUSTAVA ZA UPRAVLJANJE SKLADIŠTEM NA BAZI MIKROSERVISNE ARHITEKTURE

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Tocko

Matični broj: 0016137478

Studij: Informacijsko i programsko inženjerstvo

IZRADA SUSTAVA ZA UPRAVLJANJE SKLADIŠTEM NA BAZI MIKROSERVISNE ARHITEKTURE

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, rujan 2023.

Filip Tocko

Izjava o izvornosti

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

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

Sažetak

Pregled, opis i usporedba temeljnih web arhitektura s naglaskom na mikroservisnu arhitekturu. Opis interoperabilnosti, koja je neizostavni i neophodni dio mikroservisne arhitekture u kojoj prevladava komunikacija između aplikacija. Detaljan opis mikroservisne arhitekture u vidu principa, tehnologija, prednosti i nedostataka te uzoraka dizajna. Ovo sve predstavlja teorijski dio rada, dok u praktičnom dijelu rada slijedi opis i prikaz izrađenog sustava za upravljanje skladištem baziranog na mikroservisnoj arhitekturi. Spomenuti opis i prikaz izrađenog sustava je pružan kroz opise: načina korištenja sustava, njegovih elemenata, arhitekture, korištenih tehnologija i načina funkcioniranja te realizacije istoga. Isti predstavlja praktičnu primjenu elemenata i koncepata mikroservisne arhitekture, što je zapravo svrha ovog rada.

Ključne riječi: web arhitekture, mikroservisi, mikroservisna arhitektura, uzorci dizajna, tehnologije mikroservisne arhitekture, upravljanje skladištem.

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

Sadržaj

1. Uvod

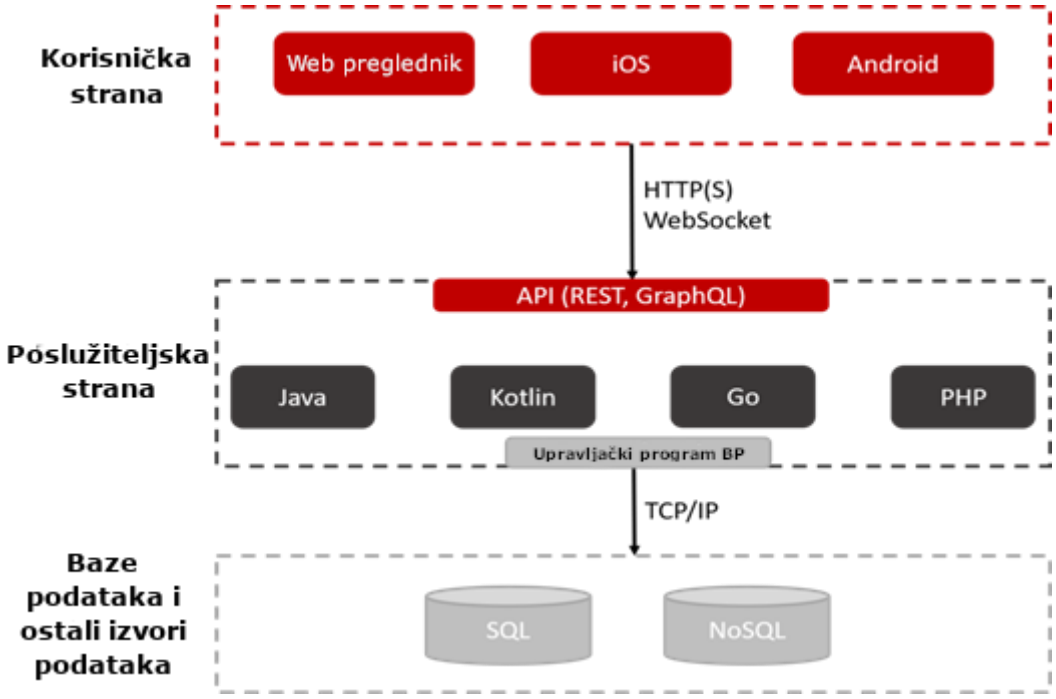
Mikroservisna arhitektura je danas sve više zastupljena kao arhitekturna osnova kod izgradnje web sustava. Ranije su bile u većoj mjeri korištene ostale web arhitekture poput: monolitne i servisno orijentirane arhitekture, koje je s vremenom mikroservisna arhitektura počela sve više zamjenjivati, zbog svojih pogodnosti. U ovom radu će se opisati i međusobno usporediti temeljne web arhitekture kako bi se pružio pregled i razvoj istih kroz povijest, da bi u nastavku bio fokus obrade na suvremenoj, mikroservisnoj arhitekturi. Kod svih web sustava baziranih na mikroservisnoj arhitekturi prevladava komunikacija, budući da je prisutna distribuiranost i labava povezanost između komponenti istih. Kod navedene komunikacije, neizostavnu i ključnu ulogu ima interoperabilnost. *Mikroservisi* su osnovne komponente od kojih je izgrađena mikroservisna arhitektura. Za izgradnju mikroservisa koriste se određene tehnologije namijenjene toj svrsi, a pritom kao smjernice „najbolje prakse“ služe osnovni principi i svojstva istih. Osim brojnih prednosti mikroservisa, postoje i određeni nedostaci korištenja istih. Kod izgradnje mikroservisne arhitekture, koriste se posebni uzorci dizajna, specifični za istu, koji služe kao arhitekturne smjernice.

U praktičnom dijelu će se izraditi sustav za upravljanje skladištem na principima mikroservisne arhitekture. To je jedan interni dio poslovnog sustava zamišljenog trgovačkog poduzeća, koji se koristi kao korisnička web aplikacija SPA (eng. *Single-Page Application*) arhitekture, koja komunicira s pozadinskim dijelom sustava temeljenog na mikroservisnoj arhitekturi. Kod obrade praktičnog dijela rada, opisati će se korištene tehnologije za izradu spomenutog sustava: Java Spring Boot, Node.js, RabbitMQ, MySQL, MongoDB i React.js. Spomenuti sustav može poslužiti kao generička podrška poslovanju bilo kojem trgovačkom poduzeću, koje se bavi nabavom i prodajom artikala, jer su u poslovnoj domeni prisutni generički poslovni procesi, karakteristični za tu djelatnost. Glavna ideja rada je praktično izraditi i prikazati korištenje mikroservisne arhitekture na primjeru sustava za upravljanje skladištem, za što je potrebno poznavanje obrađenih teorijskih koncepata vezanih uz istu.

2. Web arhitekture

Web arhitekture za web sustave su izgrađene od različitih komponenti u međusobnoj interakciji. Komponente pripadaju različitim slojevima, gdje svaki sloj nosi svoj dio odgovornosti. Govori se o tri glavna sloja (eng. *Three Principal Layers*): *prezentacijski*, *aplikacijski* (sloj domene) i *podatkovni* (sloj izvora podataka) [1]. Prezentacijski sloj pruža vidljivo korisničko sučelje i interakciju s korisnicima (npr. sučelje u web pregledniku). Aplikacijski sloj povezuje prezentacijski i podatkovni sloj. On obavlja poslovnu logiku dohvaćanjem i obradom podataka iz podatkovnog sloja pružajući navedene potrebne podatke prezentacijskom sloju. Time se izbjegava direktna komunikacija podatkovnog i prezentacijskog sloja. Podatkovni sloj je zadužen za upravljanje podacima poslovne domene, što se može iščitati iz ranijeg opisa. Korisnik kroz interakciju s prezentacijskim slojem izvršava određenu akciju na istome, koju obrađuje aplikacijski sloj pribavljanjem i obradom potrebnih podataka iz podatkovnog sloja i konačno prezentacijski sloj prikazuje navedene podatke. Ovo je bio sažeti opis funkcioniranja komunikacije između slojeva.

Stavljanjem opisanih slojeva u suvremeni kontekst modernih web arhitektura, prezentacijski sloj odgovara *korisničkoj strani* (eng. *front-end*), aplikacijski sloj *poslužiteljskoj strani* (eng. *back-end*) i podatkovni sloj *bazama podataka i ostalim izvorima podataka* [2]. Za izgradnju svakog sloja se koriste različite web tehnologije. Ovo je *troslojna arhitektura* na kojoj se temelje gotovo sve današnje web arhitekture. Na slici 1. je ilustriran primjer jedne moderne troslojne web arhitekture. Korisničku stranu iste čine: web preglednik i mobilne aplikacije (Android i iOS), koje pružaju vidljivo sučelje korisniku i preko HTTPS ili WebSocket protokola komuniciraju s poslužiteljskom stranom tj. aplikacijskim programskim sučeljem (eng. *Application Programming Interface (API)*) web servisa, zaduženih za obavljanje poslovne logike kroz obradu i pristupanje podacima iz perzistentne strane tj. baza podataka, koristeći određene upravljačke programa (eng. *driver*) i protokole specifične za pojedinu bazu podataka. Vidljivo je da svaki sloj sadrži komponente i komunikacija između slojeva se bazira na komunikaciji između komponenata slojeva. Najčešće web arhitekture koje imaju zastupljen oblik slojevitosti su: *monolitna*, *servisno orijentirana* i *mikroservisna arhitektura*.



Slika 1. Primjer moderne troslojne web arhitekture [3].

2.1. Monolitna arhitektura

Monolitna arhitektura (eng. monolithic architecture) je tradicionalna web arhitektura u kojoj su *sve komponente integrirane u jedan izvorni kod*. Više je konvencionalna i pripada starijim stilovima razvoja softvera te se može promatrati kao integrirani arhitektonski dizajn, za razliku od npr. modularnog dizajna [5]. Sve komponente sustava su dio jednog izvornog koda koji se izvršava u nekom web kontejneru (poslužitelju) [4]. Sukladno tome prema literaturi [4], prednosti ove arhitekture su:

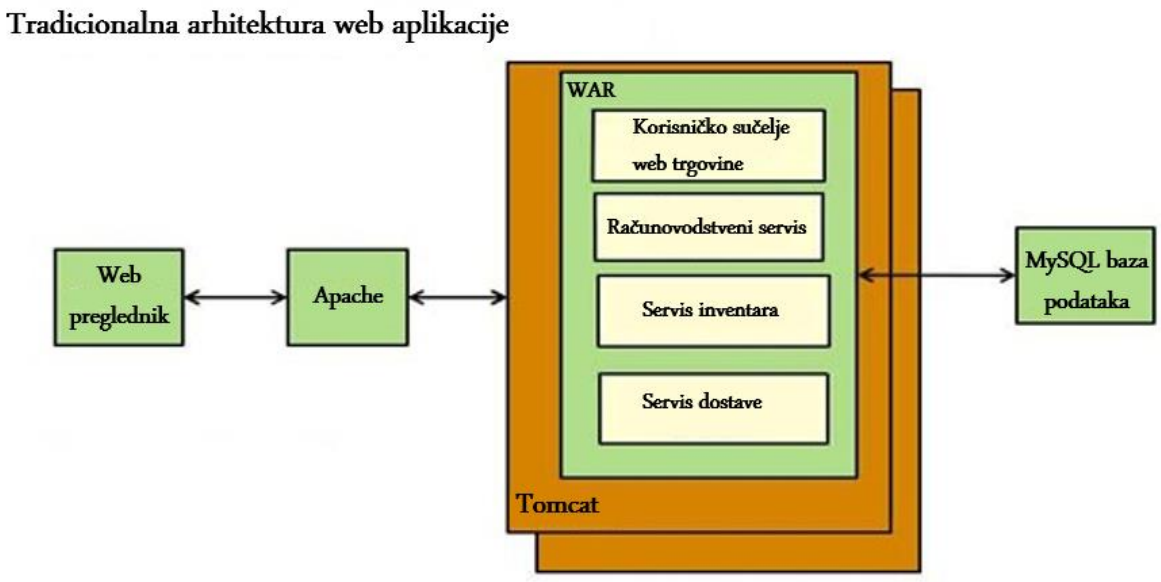
- jednostavnost razvoja – cilj razvojnih alata i okruženja je potpora razvoju monolitnih sustava
- jednostavnost implementacije – potrebno je samo jednu datoteku izvornog koda implementirati (eng. *deploy*) na određeno mjesto izvršavanja (eng. *runtime*)
- jednostavnost skaliranja – aplikacija (sustav) se može skalirati jednostavnim izvršavanjem više kopija iste preko uravnotežitelja opterećenja (eng. *load balancer*).

Još neke od važnijih prednosti iste su: brže performanse jer se podaci čitaju izravno s diska kroz datotečni sustav i jednostavnija i manje složena interakcija između komponenata [5]. Ova arhitektura je pogodna kod manjih sustava, koji nisu previše složeni i kod njih dolaze do izražaja navedene prednosti. Povećavanjem veličine i složenosti sustava zastupljeni su nedostaci ovakve arhitekture pa je za tu svrhu bolje koristiti pogodnije web arhitekture.

Prema literaturi [4] nedostaci su:

- velika monolitna baza koda teža za razumijevanje i mijenjanje, što usporava i otežava razvoj
- preopterećenje razvojnog okruženja, zbog velike baze koda
- preopterećenje web kontejnera, jer aplikacija zauzima više prostora pa je potrebno duže vrijeme pokretanja
- otežana kontinuirana implementacija (eng. *continuos deployment*), jer se za ažuriranje jedne komponente mora ponovno postaviti cijela aplikacija
- mogućnost skaliranja samo u jednoj dimenziji, pokretanjem više kopija (instanci) aplikacije uz nemogućnost skaliranja s povećanjem količine podataka
- povezani i koordinirajući razvojni timovi, bez mogućnosti neovisnog rada svakog tima
- dugoročna ovisnost o tehnologiji i razvoj novih tehnologija može značiti ponovno pisanje koda cijele aplikacije

Navedene nedostatke ove arhitekture ispravljaju servisno orijentirana i mikroservisna arhitektura, koje su razvijene s ciljem ispravljanja tih nedostataka. Slika 2. prikazuje primjer monolitne arhitekture, gdje komponente sustava: korisničko sučelje (pruža vidljiv prikaz korisniku preko web preglednika) i web servisi (obavljaju poslovnu logiku kroz komunikaciju s bazom podataka i interakciju s komponentom korisničkog sučelja) su integrirane u jednu Java WAR datoteku, koja se nalazi na Tomcat web kontejneru (poslužitelju).



Slika 2. Primjer monolitne arhitekture [4].

2.2. Servisno orijentirana arhitektura

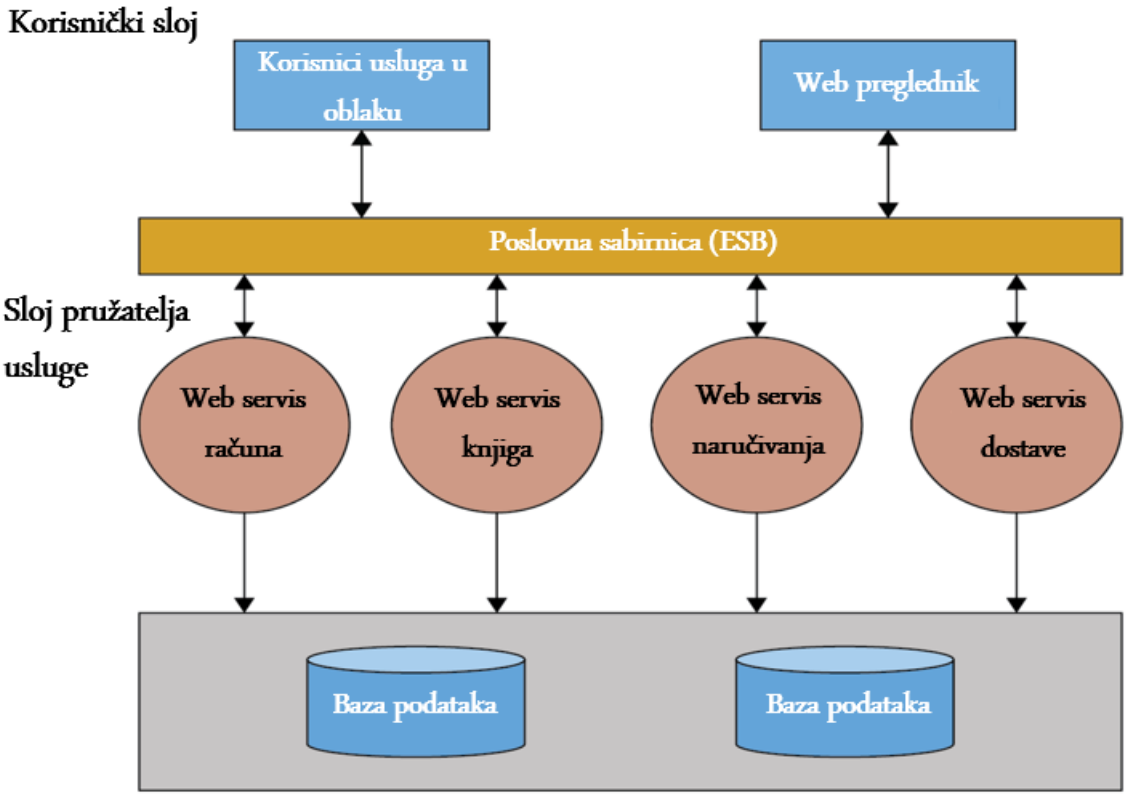
Servisno orijentirana arhitektura (eng. service oriented architecture (SOA)) je web arhitektura temeljena na interakciji između labavo povezanih i autonomnih komponenti, *web servisa* [5]. Svaki web servis obavlja svoj dio posla (najčešće poslovnu funkciju) i izlaže procese i ponašanje kroz ugovore, koji se sastoje od poruka, a pristup do istoga je moguć preko krajnjih točaka (eng. *endpoints*) [5]. Web servisi su tipično implementirani oko tehnološki neovisnih tehnologija: SOAP (Simple Object Access Protocol), WSDL (Web services description language) i XML (Extensible Markup Language). SOAP služi za komunikaciju između servisa i izgradnju tzv. SOAP web servisa. WSDL je jezik opisa web servisa, koji opisuje lokaciju, formate ulaznih i izlaznih poruka i operacije servisa te pruža način na koji se može pristupiti servisu. Poruke kojima komuniciraju web servisi su u XML formatu, koji je pogodan za razmjenu poslovnih podataka neovisno o tehnologiji. Osnovne tri komponente karakteristične za ovu arhitekturu su: pružatelj servisa (eng. *service provider*), korisnik servisa (eng. *service consumer*) i registar servisa (eng. *service registry*) [5]. Pružatelj servisa registrira svoj web servis kod registra servisa (npr. UDDI), koji sadrži WSDL dokumente za sve registrirane web servise. Kada korisnik servisa želi koristiti tj. pristupiti određenom web servisu, kontaktira registar servisa, kako bi dobio željeni WSDL dokument servisa i pomoću njega se znao povezati sa samim servisom.

Prema literaturi [6], osnovni koncepti u servisno orijentiranoj arhitekturi, koji predstavljaju glavnu ideju iste su:

- *labava povezanost* – smanjenje ovisnosti između komponenata sustava, što omogućuje lakšu ponovnu iskoristivost i nadogradnju
- *visoka interoperabilnost* – omogućuje koordiniranu komunikaciju i povezivanje web servisa, koji su distribuirani i mogu biti razvijeni i implementirani u različitim tehnologijama i platformama
- *orkestracija* – centraliziran pristup povezivanja web servisa, gdje su web servisi povezani oko središnjeg web servisa, koji ima ulogu orkestratora tj. koordinira interakciju između web servisa
- *koreografija* – decentraliziran pristup povezivanja web servisa, gdje web servisi međusobno komuniciraju porukama bez središnjeg mjesta, poštivanjem pravila i dogovora interakcije

Iako web servisi mogu biti razvijeni i implementirani u različitim tehnologijama i platformama, gdje svaki od njih obavlja određenu poslovnu funkciju, potrebno je povezati iste u smislenu cjelinu pomoću tzv. poslovne sabirnice (eng*. Enterprise Service Bus (ESB)*). *ESB*

kao što je navedeno, povezuje web servise u jednu cjelinu i sva komunikacija između istih se odvija preko te sabirnice [7]. Dakle, SOA sustav je izgrađen od više web servisa, koji su međusobno povezani i integrirani oko poslovne sabirnice. Problem koji se može javiti kod takve organizacije i koordinacije servisa oko „kanala komunikacije“ tj. ESB-a je jedinstvena točka kvara za sve servise, zbog čega se danas kod novijih web arhitektura koristi API za tu svrhu [7]. Slika 8. prikazuje primjer servisno orijentirane arhitekture, koja se obično sastoji od dva sloja: korisnički i sloj pružatelja usluga [8]. Kroz korisnički sloj korisnici dolaze u dodir sa SOA sustavom [8]. ESB povezuje korisnički sloj i sloj pružatelja usluge, pretvaranjem formata poruka, preusmjeravanjem i kontroliranjem razmjene poruka između servisa [8]. Svi servisi u sloju pružatelja usluga međusobno komuniciraju porukama preko ESB-a, pružaju usluge za korisnike i koriste *zajednički izvor podataka* [8].



Slika 3. Primjer servisno orijentirane arhitekture [8].

2.3. Mikroservisna arhitektura

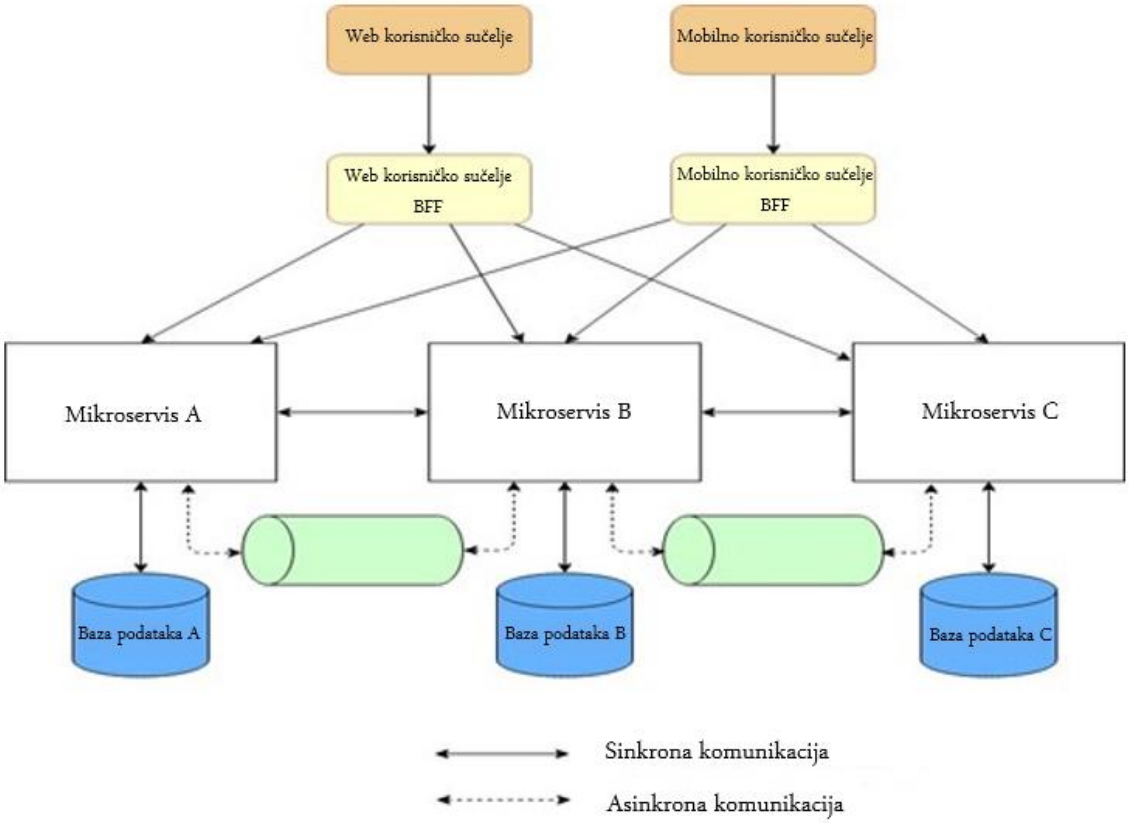
Mikroservisna arhitektura (eng. microservice architecture) je arhitekturalni stil izgradnje sustava od manjih aplikacija, gdje je svaka od njih zaseban vlastiti proces i prisutna je međusobna komunikacija između istih pomoću *API-ja* [10]. Takve manje aplikacije se nazivaju *mikroservisi*. Drugim riječima, mikroservisi su male, modularne i autonomne jedinice, koje

međusobno surađuju kroz sinkronu (HTTP, REST) ili asinkronu (JMS, AMQP) komunikaciju [10]. Kod mikroservisa prevladava visoki stupanj distribuiranosti i labave povezanosti. Sukladno opisu mikroservisa, glavni principi mikroservisne arhitekture, prema literaturi [10] su:

- princip samo jedne odgovornosti (eng. *Single Responsibility Principle*) – kao kod SOLID principa, svaki mikroservis treba imati samo jednu odgovornost
- mikroservisi su autonomni – samostalne aplikacije neovisno razmještene, koje sadrže vlastite ovisnosti poput: biblioteka, izvršnih okruženja (web poslužitelji i kontejneri)
- izlaganje usluga preko API-ja uz sakrivanje i učahurivanje unutarnje logike, arhitektura i tehnologija

Ovakva arhitektura je pogodna za velike sustave, zbog prisutne modularnosti i labave povezanosti, čime je omogućeno zasebno skaliranje modularnih komponenata (mikroservisa) i lakše upravljanje istima [9]. Kod takvih sustava je također fokus na zamjenjivosti (eng. *replaceability*) i promjenjivosti (eng. *changeability*) komponenata, mikroservisa [9]. Često vrlo korisna karakteristika mikroservisne arhitekture je dekompozicija velikih i kompleksnih monolitnih aplikacija na manje mikroservise, koji su lakši za razvoj, upravljanje i održavanje [10]. Time dolazi do podjele u razvoju sustava na manje, agilne razvojne timove, gdje je svaki razvojni tim zadužen za jedan dio funkcionalnosti sustava (mikroservisa). Međusobnom interakcijom (komunikacijom) izgrađenih mikroservisa se postiže cjelokupna funkcionalnost sustava. U mikroservisnoj arhitekturi isto tako prevladava pristup dizajna vođenog domenom (eng. *Domain-driven design (DDD)*), u kojem se poslovanje promatra kao domena, koja se sastoji od više pod domena i svaki mikroservis je zadužen za jedan dio poslovne pod domene (sadrži podatke i obavlja poslovnu logiku jedne poslovne pod domene) [5].

Slika 4. prikazuje pogodan primjer mikroservisne arhitekture. Svaki mikroservis obavlja svoj dio odgovornosti i ima vlastitu bazu podataka. Vidljiva je sinkrona i asinkrona komunikacija između mikroservisa. Klijent sa svojim web ili mobilnim sučeljem preko web ili mobilne komponente posrednika, pristupa tj. dolazi u interakciju s pojedinim mikroservisom. Određena komponenta posrednika pruža API do svojih mikroservisa, koji su personalizirani korisničkom sučelju klijenta (tzv. *„Backends for Frontends (BFF)* pristup) [11]. Ovime se može zaključiti, da u mikroservisnoj arhitekturi se sve bazira na komunikaciji i zbog toga je neophodna uloga interoperabilnosti.



Slika 4. Primjer mikroservisne arhitekture [11].

2.4. Usporedba web arhitektura

Svaka web arhitektura je pogodna ovisno o kontekstu korištenja. Danas dolazi do izražaja i sve više zastupljena je mikroservisna arhitektura, pošto se radi o modernijoj arhitekturi, koja je ispravila važnije nedostatke ranijih web arhitektura. Tablica 1. prikazuje usporedbu opisanih web arhitektura s obzirom na elemente: *veličina sustava*, *način razvoja*, *implementacija*, *komunikacija*, *izvori podataka*, *skalabilnost*, *pouzdanost* i *sigurnost*. Kroz usporedbu će se uvidjeti najvažnije karakteristike, prednosti, nedostaci, dodirne točke, ali i razlike pojedinih web arhitektura.

Monolitna arhitektura je pogodna za manje sustave niske složenosti, dok servisno orijentirana i mikroservisna arhitektura su pogodne za veće sustave. Kod servisno orijentiranih arhitektura je više naglasak na velikim sustavima poslovnih domena, dok kod mikroservisne arhitekture prevladava domenska neovisnost i prisutni su sustavi širokih razmjera. Razvoj web arhitektura kroz povijest je u sinergiji s povijesnim razvojem načina razvoja softvera. Pa tako u monolitnoj arhitekturi prevladava vodopadni način razvoja, jer se cijela aplikacija odnosno sustav razvija u cijelosti kao jedinka. Kod servisno orijentirane arhitekture sustav postaje

modularan (podjela odgovornosti na web servise) i samim time agilni način razvoja dolazi do izražaja. DevOps način razvoja je prisutan kod mikroservisne arhitekture, zbog visoke razine distribuiranosti iste pa je razmještaj, skaliranje i praćenje takvih multi-servisnih sustava dosta kompleksnije i sam DevOps pomaže u smanjenju napora i vremena [10]. Monolitni sustavi se uglavnom razvijaju u jednoj tehnologiji i smješteni su u jednoj izvršnoj datoteci na određeno mjesto izvršavanja [4]. Stoga je implementacije kod monolitne arhitekture najmanje složena. Srednje složena implementacija je kod mikroservisne arhitekture. Svaki mikroservis se može razviti u zasebnoj tehnologiji i mora se postaviti u zaseban web kontejner te se mikroservisi moraju međusobno uskladiti kroz sustave komunikacija. Najsloženija implementacija je kod servisno orijentirane arhitekture. Svaki tehnološki neovisan web servis se može nalaziti na zasebnom web poslužitelju (kontejneru), ali je potrebno iste integrirati na posebnoj platformi oko ESB-a. Unutarnja komunikacija između komponenti bez mnogo vanjske, mrežne komunikacije je zastupljena kod monolitne arhitekture, jer su sve komponente sustava dio jedne cjeline. Sinkrona komunikacija preko ESB-a je prisutna kod servisno orijentirane arhitekture, dok kod mikroservisne arhitekture je moguća sinkrona i asinkrona komunikacija pomoću API-ja [7]. Postoji jedna velika baza podataka za cijelu poslovnu domenu kod monolitne arhitekture. Za razliku od monolitne arhitekture, kod servisno orijentirane arhitekture za jednu poslovnu domenu može biti jedna baza podataka ili podjela na više manjih baza podataka, koju/e dijele servisi, svaki obavljajući svoj dio poslovne logike. Kod mikroservisne arhitekture je najčešće umjesto jedne velike baze podataka za cijelu domenu, ista podijeljena na više manjih baza podataka, kao što to može biti slučaj kod servisno orijentirane arhitekture, ali razlika je u tome da svaki servis sadrži vlastitu bazu podataka takvog oblika, bez međusobnog dijeljenja izvora podataka. Monolitna arhitektura ima nisku skalabilnost, jer ima mogućnost skaliranja samo u jednoj dimenziji, pokretanjem više instanci iste aplikacije iza uravnotežitelja opterećenja [4]. S druge strane kod servisno orijentirane i mikroservisne arhitekture, moguće je zasebno skaliranje svakog servisa bez potrebe za skaliranjem cijelog sustava, što nudi visoku razinu skalabilnosti. Najmanje pouzdana je monolitna arhitektura, jer ispadom web poslužitelja na kojem se nalazi sustav, cijeli sustav biva onemogućen. Servisno orijentirana arhitektura je srednje pouzdana iz razloga, da ispadanjem nekog servisa, cijeli sustav ne biva onemogućen, ali usko grlo predstavlja ESB oko koje su integrirani servisi, čijim ispadanjem dolazi do prestanka rada svih servisa tj. sustava. Mikroservisna arhitektura se smatra najviše pouzdanom, jer ne postoji centralna točka integracije poput ESB-a, nego prevladava decentralizacija servisa i ispadanje pojedinog servisa ne rezultira prestankom rada cijelog sustava. Monolitna arhitektura ima visoku razinu sigurnosti, zbog spomenute male razine mrežne komunikacije. Najmanju sigurnost ima mikroservisna arhitektura, jer se cijela arhitektura temelji na mrežnoj komunikaciji između servisa i zato postoji velika vjerojatnost sigurnosnih rizika. Servisno orijentirana arhitektura također je temeljena na mnogo mrežne

komunikacije, ali ona ide preko ESB-a, čija platforma nudi određene sigurnosne mehanizme, međutim i dalje postojanje sigurnosnih rizika nije isključeno.

Tablica 1. Komparativna tablična analiza web arhitektura [autorski rad].

Web arhitektura	Veličina sustava	Način razvoja	Implementacija	Komunikacija	Izvori podataka	Skalabilnost	Pouzdanost	Sigurnost
Monolitna	Manje aplikacije niske složenosti	Vodopadni [12]	Najmanje složena	Unutarnja interakcija između komponenata koje su dio jedne cjeline	Jedna baza podataka za cijeli sustav (domenu)	Niska	Niska	Visoka
Servisno orijentirana	Velike i kompleksne poslovne aplikacije	Agilni [12]	Vrlo složena	Sinkrona komunikacija između komponenata preko ESB-a [7]	Jedna ili više baza podataka domene, zajednička/e svim servisima	Visoka	Srednja	Srednja
Mikroservisna	Veliki sustavi širokih razmjera s potrebom za visokom skalabilnošću	DevOps [12]	Srednje složena	Sinkrona i asinkrona komunikacija između komponenata pomoću API-ja [7]	U pravilu jedna baza podataka po servisu (svaka baza podataka za jedan dio domene)	Visoka	Visoka	Niska

3. Uloga i važnost interoperabilnosti

Pošto je fokus rada na mikroservisnoj arhitekturi, koja je distribuirana i *temeljena na komunikaciji*, izrazito je važno opisati ulogu *interoperabilnosti*. Interoperabilnost je „sposobnost uspješne suradnje dviju ili više softverskih komponenti, unatoč razlikama u jeziku, sučelju ili izvršnoj platformi“ [13]. Zahvaljujući interoperabilnosti, softverske komponente mogu komunicirati neovisno o tehnologiji i platformi u kojima su implementirane. Podrazumijeva se da su pritom sve strane u međusobnoj interakciji, poštuju pravila i norme za razmjenu podataka. Format razmjene podataka mora biti standardiziran [14].

Postoje sljedeće četiri razine interoperabilnosti, prema literaturi [15]:

- *pravna* – usklađivanje i suradnja između organizacija (sustava) različitih pravnih okvira, politika i strategija
- *organizacijska* – usklađivanje procesa organizacija (sustava), za uspješnu razmjenu informacija između procesa
- *semantička* – uključuje i sintaktičku interoperabilnost (opis kompatibilnih formata podataka), ista je preduvjet samoj semantičkoj interoperabilnosti, koja se odnosi na značenje formatiranih podataka, tj. struktura podataka mora biti unaprijed dogovorena i ujednačena
- *tehnička* – pokriva aplikacije i infrastrukture, uključuje: specifikacije sučelja, usluge međusobnog povezivanja, integracija podataka servisa, prezentacija podataka i razmjena, sigurni komunikacijski protokoli

Pravna i organizacijska interoperabilnost moraju najprije biti zadovoljene, jer one govore o usklađenosti razmjene podataka između sustava. Zatim treba biti zadovoljena tehnička interoperabilnost, koja definira tehničke aspekte komunikacije, da bi se na kraju zadovoljila semantička interoperabilnost i time bilo omogućeno uspješno interoperabilno djelovanje sustava.

Kao što je navedeno, u mikroservisnoj arhitekturi prevladava komunikacija između aplikacija (komponenata), stoga takva „univerzalna pravila“ komunikacije su krucijalna. Potrebno je omogućiti aplikacijama uspješnu razmjenu podataka, bez obzira na kojoj se platformi nalaze i u kojoj su platformi implementirane. *Web servisi* omogućuju interoperabilnost pružanjem *vanjskog sučelja*, na kojeg se mogu povezati drugi sustavi bez obzira na platformu. Kod implementacije web servisa, zanemarivanje interoperabilnosti je nezamislivo, jer ista definira formate poruka, protokole i strukturu podataka poruka. Dakle, sučelja web servisa

omogućuju uspješnu komunikaciju između različitih sustava uz razmjenu univerzalnih formata poruka. Ovo je od velike važnosti za mikroservisnu arhitekturu, jer je ista bazirana na web servisima u međusobnoj interakciji. Kod današnjih web servisa (mikroservisa), opisano sučelje web servisa predstavlja *API*. REST (Representational state transfer) je popularna metoda za izgradnju web servisa, koji pružaju i komuniciraju preko navedenog API-a [16]. Ona omogućava interoperabilnost između klijenta i poslužitelja, korištenjem HTTP-a (Hypertext Transfer Protocol) i uobičajenog formata podataka, danas uglavnom JSON-a (JavaScript Object Notation) [16]. API je visoko standardiziran, pogodan za implementaciju u raznim tehnologijama sa sučeljem i formatom te strukturom ulaznih i izlaznih podataka, uvijek u univerzalnom i dogovorenom obliku. Na taj način REST servisi implementirani u različitim tehnologijama, mogu uspješno međusobno komunicirati preko API-ja, koji su u opisanom univerzalnom i dogovorenom obliku. Iz ovog se jasno očituje uloga i važnost interoperabilnosti, u kontekstu mikroservisne arhitekture, ali i drugih web arhitektura temeljenih na komunikaciji između aplikacija.

4. Mikroservisi

Mikroservisi su osnovna komponenta mikroservisne arhitekture. Temelj iste upravo čine mikroservisi u međusobnoj interakciji. Mikroservis je „mala aplikacija, zasebno implementirana, zasebno skalirana i zasebno testirana te sadržava samo jednu odgovornost“ [17]. Uloga mikroservisa može biti: obavljanje dijela poslovne logike, posluživanje određenih resursa, određeni funkcionalni ili nefunkcionalni zadatak, čitanje podataka iz reda poruka i daljnja obrada istih itd. [17]. Vrlo čest slučaj je i podjela poslovne domene na manje domene, s ulogom svakog mikroservisa u podržavanju jedne od istih. Svaki mikroservis je zaseban proces, koji međusobno sinkrono (npr. HTTP) ili asinkrono (npr. AMQP) komunicira s ostalim mikroservisima. Time se svaki mikroservis u mikroservisnoj arhitekturi može zasebno nadograditi, izmijeniti, zamijeniti, skalirati i implementirati u različitoj tehnologiji, što predstavlja napredak u odnosu na monolitnu arhitekturu, gdje postoji samo jedna velika, čvrsto povezana aplikacija. U nastavku će biti detaljnije opisani mikroservisi kroz principe mikroservisa, tehnologije za izgradnju mikroservisa i prednosti te nedostatke samih mikroservisa.

4.1. Principi mikroservisa

Principi mikroservisa opisuju kako bi mikroservisi kao male autonomne jedinice trebale izgledati kroz smjernice odnosno „najbolje prakse“ načina izgradnje i upravljanje istima. Oni su ujedno zorni opis glavnih karakteristika jednog mikroservisa. Pridržavanjem tih principa osigurava se najbolji stupanj uređenosti mikroservisa. Broj principa koji će biti upotrijebljen kod razvoja mikroservisa ovisi o samoj potrebi i kontekstu, iako je preporučljivo pokušati koristiti sve principe, kako bi se postigla izgradnja što uređenijih mikroservisa. Prema literaturi [18], principi mikroservisa su:

- *Model oko poslovnih koncepata* (eng. *Model Around Business Concepts*) – sučelja mikroservisa organizirana oko poslovnih koncepata su stabilnija od onih organiziranih oko tehničkih koncepata, jer su sposobnija odražavati promjene u poslovanju
- *Usvajanje kulture automatizacije* (eng. *Adopt a Culture of Automation*) – automatizirano testiranje svakog mikroservisa, uniformni poziv naredbenog retka za jednako razmještanje svih mikroservisa u izvršne okoline za dobivanje brze povratne informacije o produkcijskoj kvaliteti, kao dio usvajanja kontinuirane isporuke
- *Sakrivanje internih implementacijskih detalja* (eng. *Hide Internal Implementation Details*) – svaki mikroservis obavlja poslovnu logiku i ostale komunikacije te je u interakciji s bazom podataka, a sve navedene detalje sakriva jasno definiranim sučeljem REST API, kroz koje pruža potrebne podatke korisniku

- *Decentraliziranje svih stvari* (eng. *Decentralize All the Things*) – osiguranje samostalne komunikacije između mikroservisa, bez postojanja središnjeg mjesta, koji upravlja istom, za zadržavanje logike i podataka unutar granica mikroservisa, pomažući da stvari ostanu kohezivne
- *Neovisno razmještanje* (eng. *Independently Deployable*) – budući da je mikroservis samostalan proces, zastupljeno je samostalno razmještanje mikroservisa u izvršna okruženja, omogućujući lakšu promjenjivost, brzi razvoj novih značajki i povećanje autonomije timova zaduženih za pojedini mikroservis
- *Izolacija kvara* (eng. *Isolate Failure*) – iako je arhitektura mikroservisa otpornija od monolitnog sustava, treba uzeti u obzir i planirati tretiranje neuspjeha pojedine komponente (korištenje i postavljanje vremena čekanja, implementacija posebnih uzoraka dizajna poput „circuit breakers“), jer u protivnome može doći do kaskadnog kvara u sustavu
- *Visoka promatranost* (eng. *Highly Observable*) – praćenje sustava kao cjeline bez posebnog praćenja svakog mikroservisa, jer daje širu sliku ponašanja sustava. Agregiranje zapisnika i statistika, kako bi se mogao lakše locirati izvor, gdje se dogodio problem, a pritom je preporuka korištenje korelacijskih identifikatora za olakšavanje praćenja

4.2. Tehnologije za mikroservise

Opisanim principima mikroservisa se ukazuje da se svaki mikroservis najčešće implementira zasebno, od strane pojedinog tima zaduženog za upravljanje i razvoj istoga. Pritom se koriste univerzalne web tehnologije i standardi. Spomenute tehnologije za mikroservise su: *REST*, *tehnologije sinkrone i asinkrone komunikacije*, *baze podataka* i *kontejneri*.

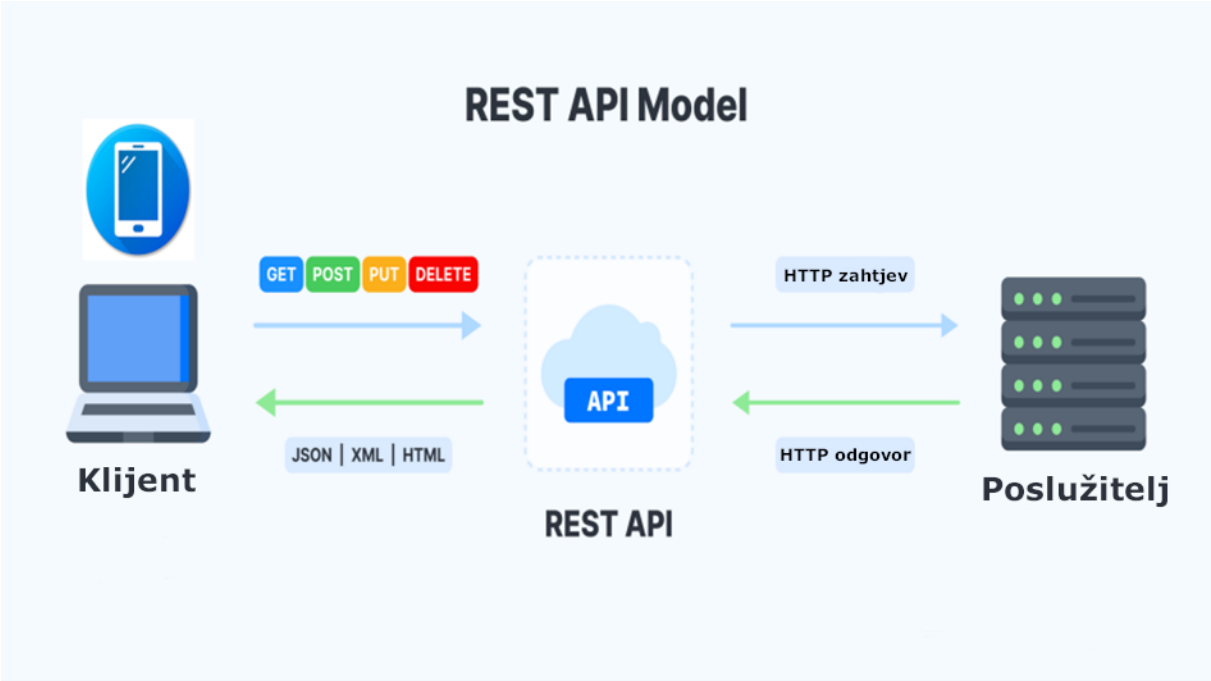
4.2.1. REST

REST (Representational state transfer) je arhitekturalni stil za izgradnju web servisa, koji koristi programerima već popularni niz standarda i metoda [20]. Ono što je temeljno za poznavanje REST-a je pojam resursa. *Resurs* je osnovna jedinica, koja predstavlja koncept i sadrži podatke (informacije) te se njegova reprezentacija vraća korisniku, nakon pozivanja URI-a (Uniform Resource Identifier) [20]. Primjer resursa može biti obična HTML stranica, JSON ili XML podatak itd. Komponente REST-a provode akcije na resursima koristeći njihove reprezentacije, tako što se vraća uvijek reprezentacija najaktualnijeg stanja resursa [20].

Osnovne komponente REST-a za izgradnju web servisa su: *format podataka* (uglavnom JSON i XML), *HTTP* (metode GET, HEAD, POST, PUT i DELETE ukazuju na akciju, koja će se izvršiti nad resursom), *URI* (identifikator resursa koji locira web servis), *MIME* (Multipurpose Internet Mail Extensions) tip (tip podatka u kojem se vraćaju podaci resursa, može biti: JSON (application/json), XML (text/xml), HTML (text/html) itd.) [20]. REST web servis nudi pristup do resursa putem API-ja, dostupnog preko URI-a. URI se poziva odgovarajućom metodom HTTP zahtjeva, koja provodi određenu akciju nad resursom, koji predstavlja podatke iz baze podataka servisa:

- *GET* – vraća resurs
- *POST* – kreira novi resurs
- *PUT* – ažurira postojeći resurs ili kreira novi, ukoliko isti ne postoji
- *DELETE* – briše resurs

Obično se HTTP metodom ukazuje na jednu od CRUD (Create, Read, Update And Delete) akcija, koja će se provesti u bazi podataka. Dakle, sve ide preko HTTP-a, u čijem odgovoru se vraćaju: statusni kod, podaci resursa i ostali podaci. Svi podaci resursa se prenose u tijelu HTTP zahtjeva ili odgovora u danas uobičajenom i vrlo često korištenom JSON formatu. Mikroservisi koriste REST za pružanje REST API-ja, putem kojeg su dostupni resursi i provode se akcije nad istima. Kao što REST služi za izgradnju web servisa, jednako služi i za izgradnju mikroservisa, koji sadrže još neke dodatne specifičnosti. Slika 5. prikazuje primjer REST API modela, koji ilustrira već sve ranije opisane stvari. Klijent zahtijeva resurs postavljanjem HTTP zahtjeva odgovarajućom metodom, spajajući se na API mikroservisa (pozivanjem URI-a), nakon čega mikroservis, koji se nalazi na poslužitelju, obrađuje akciju nad resursem i zatim vraća resurs u obliku JSON ili XML podatka, kao dio HTTP odgovora. Korištenje REST API-ja podrazumijeva interoperabilnost, jer sadrži univerzalne standarde i metode te struktura podataka, koja se razmjenjuje je uvijek u dogovorenom obliku. Stoga REST je pogodan za korištenje u bilo kojem programskom jeziku usmjerenom na web tehnologije.



Slika 5. Primjer REST API modela [21].

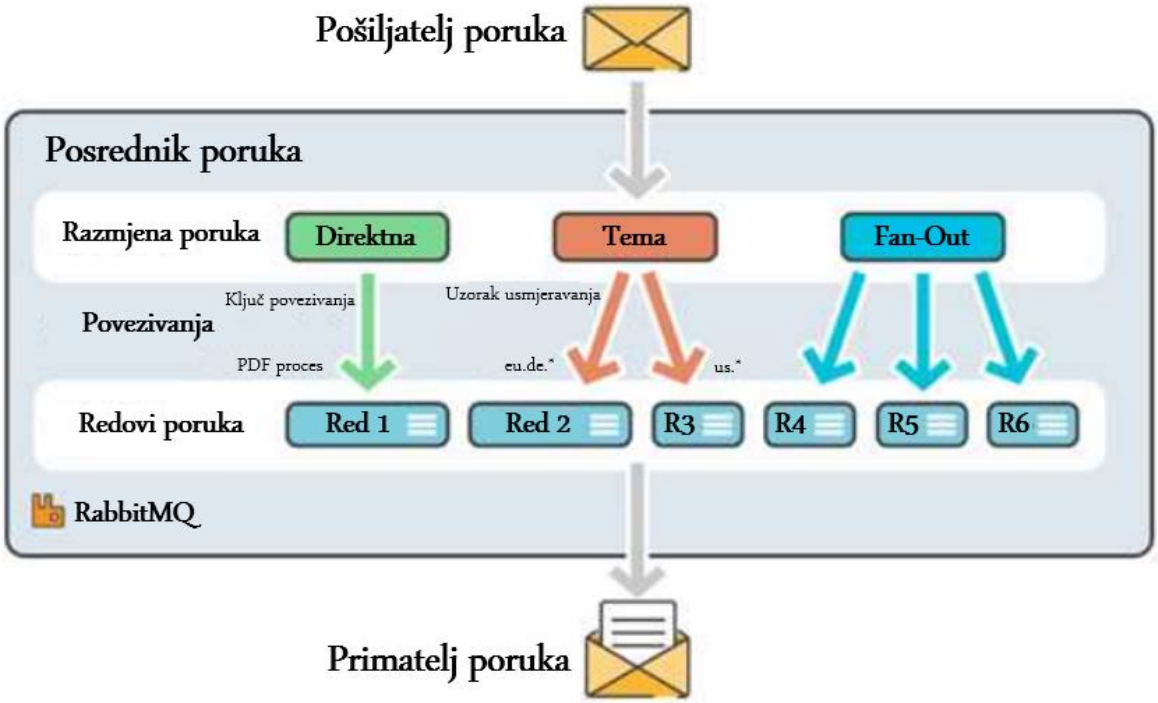
4.2.2. Tehnologije sinkrone i asinkrone komunikacije

Mikroservisi izgrađeni REST stilom, komuniciraju s drugim mikroservisima koristeći tehnologije sinkrone i asinkrone komunikacije. Sinkroni oblik komunikacije je jednak komunikaciji između mikroservisa i klijenta putem REST API-ja. REST preko HTTP-a je tehnologija, koja služi za sinkronu komunikaciju između mikroservisa [18]. Ona je u potpunosti jednaka opisanoj komunikaciji klijenta i mikroservisa, kod poglavlja „REST“. Takve tehnologije sinkrone komunikacije su oblika *zahtjev/odgovor* (eng. *request/response*), jer dolazi do blokirajućeg čekanja pošiljatelja, sve do ne pristizanja odgovora primatelja i obično imaju veću latenciju, zbog zaglavlja HTTP poruka [18]. Iako se mogu koristiti za komunikaciju između mikroservisa, često su izrazito nepraktične za korištenje, jer sudjelujući mikroservisi moraju biti aktivni za vrijeme komunikacije, postoji blokirajuće čekanje na odgovor i prisutna je čvrsta povezanost, gdje pogreškom u komunikaciji, može doći do kvara kod oba sudjelujuća mikroservisa.

S druge strane asinkroni oblik komunikacije je pogodniji za komunikaciju između mikroservisa, pošto pruža pozadinsku i ne blokirajuću komunikaciju, smanjujući potrebno vrijeme čekanja korisnika. Tehnologije asinkrone komunikacije su *temeljene na događajima* (eng. *event-based*) [18]. *Redovi poruka* su primjer tehnologije temeljene na događajima, koja omogućuje asinkronu komunikaciju između mikroservisa. Uglavnom funkcioniraju prema principu *„objavi- pretplati“ (*eng. *publish-subscribe),* kod kojeg objava rezultira događajem,

kojeg obrađuju pretplaćeni na isti. Primatelj poruka se pretplaćuje na temu, tako što osluškuje pripadni red poruka, a pošiljatelj šalje nove poruke u red poruka teme. Mikroservis može imati ulogu primatelja i pošiljatelja poruke ovisno o kontekstu. Redovi poruka pružaju veće performanse u odnosu na sinkronu komunikaciju HTTP-om, jer njihove poruke nemaju toliku količinu podataka, kao što imaju HTTP poruke sa svojim zaglavljima. Isto tako pružaju veću pouzdanost, zato što primatelj poruke ne mora biti aktivan kada pošiljatelj šalje poruku, već poruka ostaju u redu poruka, sve dok je primatelj ne pročita. Postoji sigurna garancija prijema poruka i nema blokirajućeg čekanja do pristizanja odgovora primatelja. Takve poruke moraju imati dogovorenu strukturu podataka odnosno mora biti osigurana interoperabilnost, kako bi mikroservisi jednako interpretirali podatke. Najpoznatiji primjeri tehnologije redova poruka su: JMS (Jakarta Message Service), RabbitMQ, Apache Kafka i ActiveMQ. Navedeni primjeri tehnologija su zapravo *posrednici poruka* (eng. *Message Brokers*), zato što upravljaju pretplatama, omogućujući pretplatnicima, da budu obaviješteni o dolasku događaja (dolasku nove poruke u red čekanja) [18]. To omogućuje labavu povezanost između mikroservisa, jer se oba mikroservisa spajaju na posrednik poruka, unutar kojeg se nalaze resursi poput teme i redova poruka tema i pojedini mikroservis može imati ulogu pošiljatelja ili primatelja poruke. Drugim riječima, posrednici poruka omogućuju asinkronu komunikaciju između mikroservisa i čine takve sustave razdvojenijima i skalabilnijima [18]. Mikroservisi implementirani i razmješteni u različitim tehnologijama, mogu uspješno komunicirati preko tih posrednika poruka.

Slika 6. prikazuje primjer posrednika poruka, kojeg predstavlja RabbitMQ tehnologija. Posrednik poruka sadržava razmjene poruka (teme) i redove poruka. Pošiljatelj poruka se spaja kod posrednika poruka (eng. *Broker*) i šalje novu poruku unutar određene teme (eng. *topic*), koristeći uzorak usmjeravanja (eng. *routing pattern*), kako bi se poruka pohranila u željeni red poruka teme, pošto tema može imati više redova poruka [22]. Primatelj poruke se također spaja kod posrednika poruka i osluškuje poruke u željenom redu poruka. Ovim primjerom se slikovito potkrjepljuje i dodatno razjašnjava sve ranije opisano u vezi tehnologija asinkrone komunikacije.



Slika 6. Primjer posrednika poruka RabbitMQ [22].

Uz brojne pogodnosti tehnologija asinkrone komunikacije, postoje i određeni problemi vezani uz kompleksnost. Oni su: veliki broj tema i redova poruka za održavanje i pitanje da li je poslana poruka u redu poruka pristigla do željenog primatelja, jer može biti više primatelja, koji osluškuju isti red poruka, a poruka se najčešće briše iz reda poruka, nakon što je pročita bilo koji primatelj i često kodovi mogu biti složeniji u odnosu na tehnologije sinkrone komunikacije [18].

4.2.3. Baze podataka

Baze podataka su perzistentni izvor podataka mikroservisa. Podaci su neophodni za rad mikroservisa, jer se rad mikroservisa bazira na izvršavanju određenih obrada nad podacima. Ovisno o prirodi i strukturi podataka mikroservisa, koriste se najčešće relacijske i nerelacijske baze podataka, ali ovisno o specifičnim potrebama, mogu biti korištene i neke druge vrste baza podataka. Kada se govori o bazama podataka u kontekstu mikroservisa, postoje dva relevantna uzorka dizajna: *baza podataka po servisu* (eng. *Database per Service*) i *zajednička baza podataka* (eng. *Shared Database*) [23]. Uzorak dizajna „baza podataka po servisu“ osigurava uvjete mikroservisa, da oni budu labavo povezani, skalabilni i neovisni te omogućava tehnološku heterogenost baza podataka mikroservisa, tako da svaki mikroservis sadržava vlastitu vrstu baze podataka, koja najbolje odgovara njegovoj potrebi [23]. Kod uzorka dizajna „zajednička baza podataka“, postoji jedna baza podataka, koju dijele svi

mikroservisi. To je pogodno za korištenje kada je podatkovni sloj fiksan, bez planiranih promjena istoga i transakcije su presudne u radu aplikacije [23]. Ova vrsta uzorka dizajna odstupa od klasične mikroservisne arhitekture i mikroservisa te je više vezana uz servisno orijentiranu arhitekturu. Navedeni uzorci dizajna će biti detaljnije razrađeni kod poglavlja „Uzorci dizajna mikroservisne arhitekture“, a ovdje su iznesene ključne činjenice i zaključci.

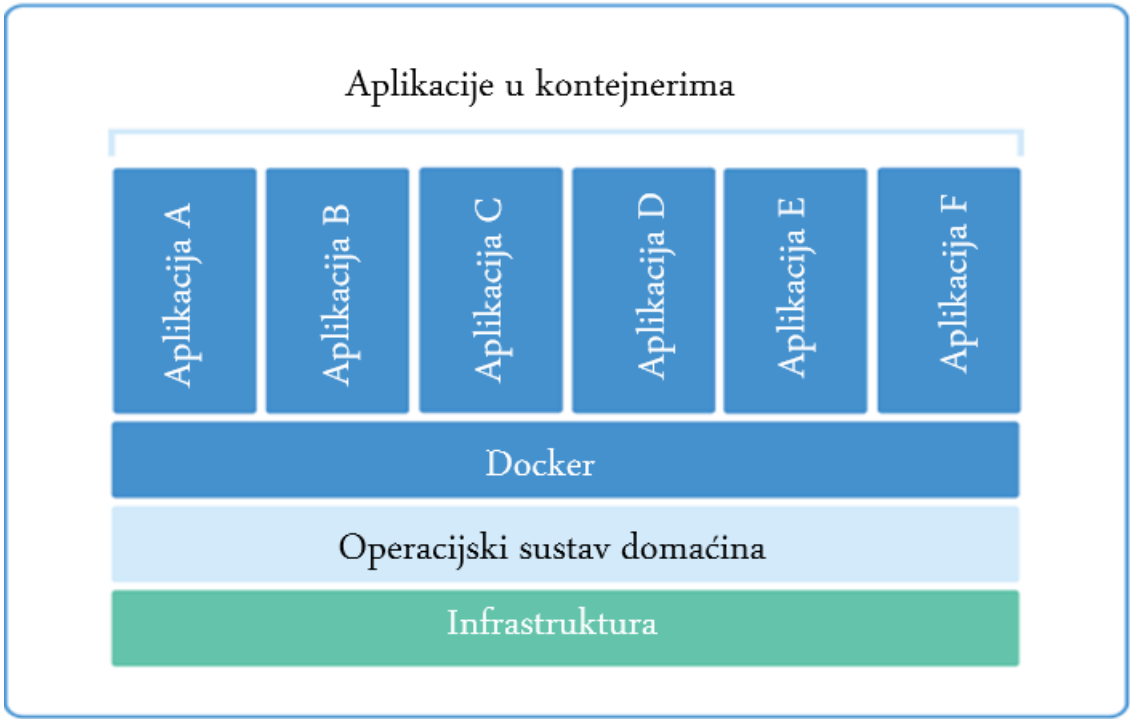
Kod odabira baze podataka mikroservisa, prvo je potrebno odabrati jedan od uzoraka dizajna: jedna baza podataka po servisu ili zajednička baza podataka [23]. Nakon toga se bira *najpogodnija tehnologija baze podataka* [23]. Prema literaturi [23], faktori koji mogu biti od utjecaja kod izbora najpogodnije tehnologije baze podataka su:

- Brzina čitanja – brzina dohvaćanja upita ili broj operacija u sekundi, važna kod elektroničkih trgovina, bankarskih sustava i sustava odnosa s korisnicima
- Brzina pisanja – traže se slični zahtjevi kao i kod brzine čitanja, samo je ovdje u pitanju pisanje u bazu podataka, važna kod mikroservisa koji spremaju mnogo podataka
- Latencija – kašnjenje između korisničke akcije i odgovora poslužitelja, važna kod komponenti, gdje je presudno visoko korisničko iskustvo poput: aplikacije za prijenos uživo i igranje u stvarnom vremenu
- Efikasnost resursa – manja potrošnja resursa, rezultira bržim izvršenjima, smanjenim opterećenjem domaćina (eng. *Host*) i smanjenju mogućih troškova ovisno o platformi
- Utjecaj na razvoj, razmještaj i testiranje mikroservisa – pitanje pozitivnih ili negativnih utjecaja baze podataka kod razvoja, razmještaja i testiranja mikroservisa

Stoga odabir baze podataka mikroservisa je vrlo važna arhitekturalna odluka, koju je potrebno pomno razmotriti, jer itekako utječe na rad mikroservisnih sustava.

4.2.4. Kontejneri

Kod mikroservisne arhitekture, svaki mikroservis je razmješten u vlastito izvršno okruženje omogućujući zasebno upravljanje, neovisnost i izolaciju pojedinog mikroservisa. Stoga nije pogodno na istom web poslužitelju (kontejneru) imati razmješteno više mikroservisa. Zato jer se tada odstupa od osnovnih ideja autonomnosti i neovisnosti mikroservisa. *Docker kontejner* je tehnologija otvorenog koda i najbolja inovacija u tehnologiji kontejnera, koja sadrži sve potrebne sposobnosti za pružanje izvršnog okruženja aplikacije [24]. Sve potrebne ovisnosti i virtualni strojevi za rad aplikacije se nalaze u takvom kontejneru zajedno s istom. Stoga aplikacije u docker kontejnerima mogu raditi na bilo kojim platformama, bez ikakvih promjena [24]. Na platformama domaćina nije potrebno imati instalirane sve ovisnosti potrebne, za smještaj aplikacija u virtualna okruženja, već ovisno o specifičnim potrebama pojedine aplikacije je sve to sadržano unutar docker kontejnera. Na platformi domaćina može biti smještena bilo koja količina docker kontejnera, ovisnih o linux jezgri (eng. *linux kernel*) i sama platforma domaćina pruža potrebno okruženje linux jezgre (eng. *linux kernel facility*) istima [24]. Definicija docker kontejnera koja najbolje karakterizira i potkrepljuje opisane činjenice u vezi istoga glasi: „lagana virtualizacijska tehnologija razine operacijskog sustava, koja omogućava rad aplikacije i njezinih ovisnosti kroz izolaciju resursa“ [25]. Slika 7. prikazuje osnovnu arhitekturu Docker-a. Na operacijskom sustavu domaćina se nalazi instaliran Docker, unutar kojeg se može kreirati više instanci kontejnera i na svakoj od njih se nalazi pojedina aplikacija.



Slika 7. Prikaz arhitekture Docker-a [26].

„Male težine“ (eng. lightweight) i *velike brzine* takvih kontejnera su značajna prednost u odnosu na okruženja virtualnih strojeva, kao i *znatno manje brzine pokretanja i rada* [24]. Docker inženjeri mogu raditi vrlo učinkovito i jednostavno, jer aplikacija i njezin kod se izgrađuju unutar kontejnera, što se vrlo lagano premješta na bilo koju razvojnu, testnu ili produkcijsku okolinu [24]. Kao što je navedeno, sve ovisnosti se po osnovi izgrađuju i učahurene su unutar kontejnera [24].

Mikroservisi se najčešće smještaju u vlastite docker kontejnere, osiguravajući princip *jedne instance servisa po kontejneru*, kako bi se postigla neovisnost, brža izgradnja i pokretanje, manja potrošnja resursa i zasebno upravljanje mikroservisa [25]. Princip, jedna instanca servisa po kontejneru je ujedno uzorak dizajna mikroservisne arhitekture. Prema literaturi [25], proces razmještaja mikroservisa u docker kontejner se sastoji od sljedeće tri aktivnosti:

- *Izgradnja slike kontejnera* – nakon izgradnje mikroservisa, obično se kreira docker datoteka (eng. *dockerfile*), pomoću koje se izgrađuje slika kontejnera izgrađenog mikroservisa odnosno slika kontejnera se instancira kao kontejner, unutar kojeg će „živjeti“ mikroservis
- *Upravljanje slikama kontejnera* – sustav mikroservisne arhitekture se sastoji od više slika kontejnera mikroservisa, kojima treba upravljati na uniforman način. Postoje specifični alati za uvid i upravljanje slikama kontejnera, poput Dockerhub-a. Dockerhub pruža privatne repozitorije za pronalaženje i dijeljenje slika kontejnera
- *Koreografija kontejnera* – kako svaki mikroservis se nalazi u svojem docker kontejneru, poželjno je pokretanje sustava izvršiti automatskim pokretanjem svih kontejnera mikroservisa, a ne zasebnim ručnim pokretanjem svakog kontejnera i za tu svrhu služe tehnologije poput: Kubernetes. Kubernetes omogućuje razmještaj mikroservisnih sustava u jednom kliku (eng. *one-click deployment*) i upravljanje te skaliranje mikroservisa u kontejnerima

Kontejneri kao tehnologija je idealna platforma za razmještaj mikroservisa, zato jer omogućava lakši razmještaj sustava, koji se sastoje od više mikroservisa [25]. Kod razvoja, održavanja i operacija takvih sustava je to platforma „najbolje prakse“ DevOps-a [25].

4.3. Prednosti i nedostaci mikroservisa

Mikroservisi pružaju brojne prednosti od kojih većina pripada općenitim prednostima distribuiranih web arhitektura, međutim oni dovode same prednosti na višu razinu. Prednosti mikroservisa su ujedno i glavni razlozi korištenja istih. Najvažnije prednosti mikroservisa, prema literaturi [18] su:

- *Tehnološka heterogenost* – mogućnost korištenja različitih programskih jezika za izgradnju pojedinog mikroservisa i baza podataka za izvor podataka mikroservisa. Time je moguće odabrati najpogodniju vrstu baze podataka za prirodu podataka mikroservisa i isprobavanje pojedinih mogućnosti različitih programskih jezika. Ovisnost o pojedinim tehnologijama se također sprječava, jer se nudi široki spektar tehnoloških mogućnosti i omogućeno je lakše prihvaćanje novih tehnologija
- *Elastičnost* – kvar jedne komponente ne znači prestanak rada cijelog sustava, jer je moguća izolacija problema kvara komponente, što ponekad može rezultirati i degradacijom pojedinih funkcionalnosti sustava, ukoliko se radi o povezanoj komunikaciji između komponenti. U većini slučajeva želi se postići da ispad jedne komponente ne utječe na rad drugih komponenti i rad samog sustava
- *Skaliranje* – moguće zasebno skaliranje (osiguravanje više instanci mikroservisa), onih mikroservisa gdje postoji nužna potreba za time, što rezultira učinkovitijim upravljanjem troškova (resursa)
- *Jednostavnost razmještaja* – mogućnost promjene programskog koda i kontinuirane nadogradnje mikroservisa, neovisno i bez utjecanja na druge mikroservise, jer se svaki mikroservis zasebno razmješta u svoju izvršnu okolinu (kontejner) nakon svake promjene istoga, što omogućuje veću razvojnu fleksibilnost i produktivnost.
- *Organizacijsko usklađivanje* – osiguranje najboljeg omjera veličine timova i produktivnosti. Manji agilni timovi zaduženi za rad nad manjim bazama kodova (pojedinom mikroservisom) su obično produktivniji pa se u organizaciji obično svaki tim usklađuje oko jednog mikroservisa
- *Sastavljivost* – mogućnost ponovne upotrebe funkcionalnosti (mikroservisa) na različite načine i za različite svrhe
- *Optimiziranje za zamjenjivost* – lakša zamjena mikroservisa drugim mikroservisom iste namjene, zbog male baze koda i to ne rezultira zamjenom cijelog sustava, nego jedne komponente istoga, koja se na isti način interoperabilno povezuje s ostalim komponentama

Iz opisanih prednosti se može zaključiti, da su mikroservisi otvoreni za buduće nadogradnje (promjene), prisutna je visoka razina ponovne upotrebe, zamjenjivosti i skaliranja istih kao i mogućnost poboljšanja njihovog rada i performansi te dodavanje novih mikroservisa u postojeći sustav baziran na mikroservisnoj arhitekturi. Prisutni su isto tako znatno kraći razvojni ciklusi za pojedini mikroservis u agilnom razvoju, čime se dobiva brža povratna informacija o kvaliteti mikroservisa, a to rezultira većom konačnom kvalitetom istoga. Svakako najveći benefit mikroservisa je mogućnost mijenjanja ili nadogradnje pojedinog mikroservisa, bez mijenjanja ili nadogradnje cijelog sustava (aplikacije) [19]. Danas nije rijedak slučaj, razbijanje monolitnih sustava na manje mikroservise, iskorištavanjem prednosti i mogućnosti istih. Zapravo se radi o prelasku s monolitne na mikroservisnu arhitekturu, koja sadrži sve opisane prednosti mikroservisa.

Uz brojne prednosti mikroservisa, postoje i pojedini nedostaci, koji ukazuju na neke slabosti kod razvoja i upravljanja istima. Oni su: sama kompleksnost distribuiranih sustava, veća vjerojatnost pogreške kod komunikacije između mikroservisa, teže održavanje velikog broja mikroservisa, razvojni inženjeri moraju sami riješiti problem latencije mreže i uravnoteženja opterećenja i otežano testiranje, zbog distribuiranog okruženja [19]. Kompleksnost komunikacija i potreba za većim brojem resursa kod istih nije pogodno za manje sustave, već se očituje kao pogodnost tek kod većih i kompleksnijih sustava. Naime mikroservisi u pozadini sinkrono i asinkrono komuniciraju s ostalim mikroservisima, što zasigurno utječe na brzinu i performanse sustava. Zato jer mikroservisi možda trebaju nedostajuće podatke drugih mikroservisa, kao i obavještenja o stanju podataka pa ovdje može biti uključeno mnogo sinkronih i asinkronih kanala komunikacije. Osim problema kompleksnosti i performansi kod komunikacija, može doći i do pogrešaka, koje se mogu negativno reflektirati na rad mikroservisa. Zasigurno je prisutan i problem sigurnosnih rizika, zbog mnoštva komunikacije. Dakle, velika pažnja se treba posvetiti optimalnom dizajnu sustava komunikacija, ali se ne smije pritom nipošto zanemariti pitanje sigurnosti. To uključuje potrebu za većim znanjem i vještinama ljudi. Ovo su bili izazovi s kojim se trebaju nositi razvojni inženjeri mikroservisa, a olakšavajuća okolnost su brojne pogodnosti tj. prednosti, koje se dobivaju korištenjem mikroservisa.

5. Uzorci dizajna mikroservisne arhitekture

Mikroservisi su osnovne komponente mikroservisne arhitekture i prilikom izgradnje iste potrebno je poštivati sljedeća pravila: *neovisnost*, *labava povezanost* i *organiziranje oko poslovnog cilja* mikroservisa [27]. U mikroservisnoj arhitekturi dolazi i do nekih izazova i problema s kojima se treba suočiti i koje je potrebno implementirati osim same izgradnje mikroservisa, kako bi se osigurala navedena pravila [27]. *Uzorci dizajna mikroservisne arhitekture* služe kao opis izazova i problema i pružanje rješenja istih [27]. Kombiniranjem i korištenjem određenih uzoraka dizajna mikroservisne arhitekture, postiže se izgradnja idealnog rješenja sustava baziranog na mikroservisnoj arhitekturi. Isti su podijeljeni u pet kategorija, prema literaturi [27]:

- *Uzorci dekompozicije* (eng. *Decomposition Design Patterns*)
- *Uzorci integracije* (eng. *Integration Design Patterns*)
- *Uzorci baze podataka* (eng. *Database Design Patterns*)
- *Uzorci uočljivosti* (eng. *Observability Design Patterns*)
- *Uzorci međusobne brige* (eng. *Cross Cutting Concern Design Patterns*)

5.1. Uzorci dekompozicije

Uzorci dekompozicije pružaju rješenje za način optimalnog dizajniranja labavo povezanih mikroservisa i razbijanja velike monolitne aplikacije na manje labavo povezane mikroservise [27]. Postoje sljedeća tri uzorka dekompozicije, prema literaturi [27]:

- *Dekompozicija prema poslovnoj sposobnosti* (eng. *Decompose By Business Capability*)
- *Dekompozicija prema pod domeni* (eng. *Decompose By Subdomain*)
- *„Strangler“ dekompozicija* (eng. *Decompose By Strangler*)

Dekompozicija prema poslovnoj sposobnosti obuhvaća organiziranje mikroservisa prema poslovnim sposobnostima, na način da svaki mikroservis je zadužen za obavljanje jedne poslovne sposobnosti, koja se referira kao jedan poslovni objekt domene [27]. Poslovna sposobnost je bilo koja poslovna aktivnost usmjerena na stvaranje vrijednosti tj. odnosi se na sve aktivnosti nad jednim poslovnim objektom domene (npr. za poslovni objekt domene „narudžbe“ pripadna poslovna sposobnost je „upravljanje narudžbama“ i kreirani mikroservis bi bio odgovoran za isto) [27]. Međutim, za organiziranje po poslovnim sposobnostima, potrebno je imati već definirane poslovne objekte domene, što se ujedno može smatrati izazovom i nedostatkom, jer zahtijeva izrazito dobro poznavanje poslovanja [27]. U svakom

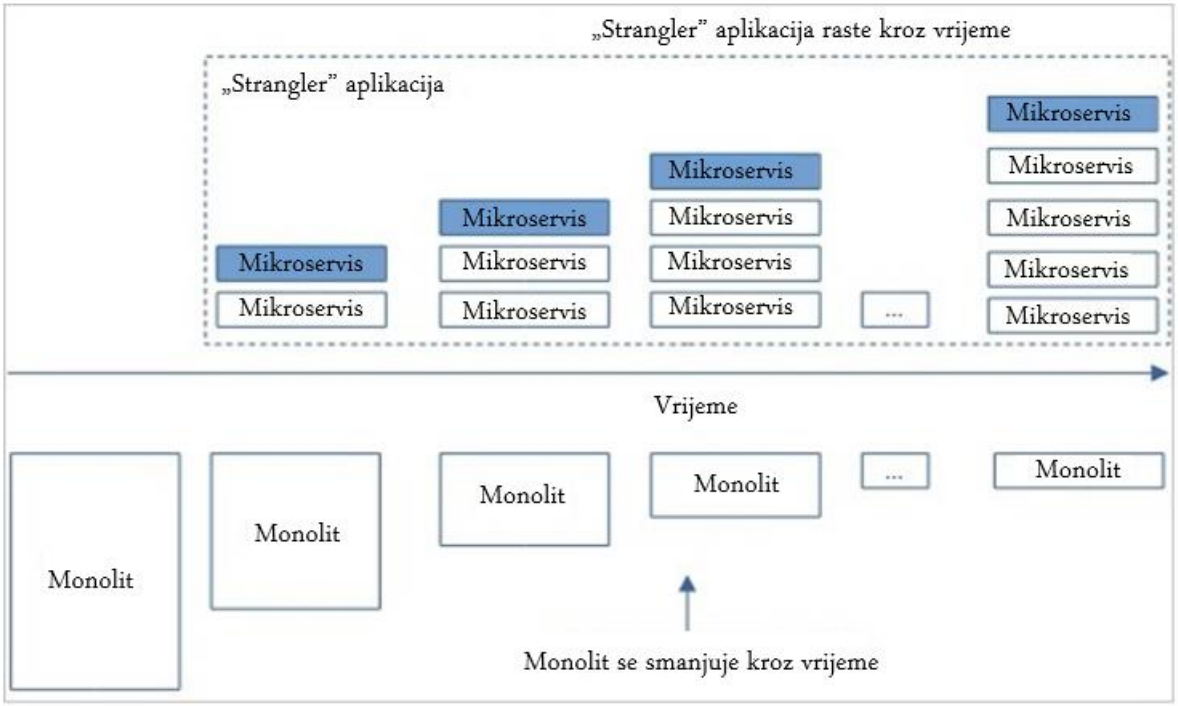
slučaju, prednosti takve dekompozicije mikroservisa su: stabilna arhitektura (budući da su poslovne sposobnosti relativno stabilne, arhitektura je također stabilna), među funkcionalni timovi (svaki tim radi odvojeno na pojedinom mikroservisu i timovi su organizirani oko funkcionalnih zahtjeva umjesto tehničkih zahtjeva) i labava povezanost mikroservisa (mikroservisi su labavo povezani i kohezivni) [27].

Dekompozicija prema pod domeni je usko povezana s *dizajnom vođenim domenom*. Poslovanje se promatra kao jedna poslovna domena, koja se sastoji od više poslovnih pod domena. Prisutna je organizacija mikroservisa na način, da je svaki mikroservis odgovoran za jednu poslovnu pod domenu. Prije izvršenja ove dekompozicije, potrebno je definirati poslovnu domenu i pripadne poslovne pod domene, što zahtijeva poslovne kompetencije i predstavlja izazov i nedostatak [27]. Prednosti takve dekompozicije su jednake kao i za dekompoziciju prema poslovnoj sposobnosti, postiže se labava povezanost mikroservisa, stabilnost arhitekture i postojanje među funkcionalnih timova [27].

„Strangler" dekompozicija je pogodna kod prelaska s monolitne arhitekture na mikroservisnu arhitekturu („Strangler" sustav). S vremenom se veliki monolitni sustav razbija na manje mikroservise i kombinirano (istovremeno) se koristi monolitni sustav odnosno naslijeđeni sustav (eng. *legacy system*) s trenutno razvijenim mikroservisima. Dolazi do postepene zamjene naslijeđenog sustava mikroservisima tj. funkcionalnosti sustava se postepeno zamjenjuju mikroservisima, bez nagle zamjene svih funkcionalnosti sustava mikroservisima (tzv. „big bang" pristup) [18]. Za određene funkcionalnosti se pozivaju razvijeni mikroservisi, dok se za preostale, mikroservisima još nepokrivene funkcionalnosti poziva naslijeđeni sustav [18]. Sljedeća tri koraka zorno opisuju „Strangler" dekompoziciju, prema literaturi [27]:

- Transformacija – neovisno razvijanje mikroservisa za implementaciju određene funkcionalnosti monolitnog sustava
- Koegzistencija – *paralelni rad monolitnog sustava i mikroservisa*, korisnik može pristupiti funkcionalnostima obje komponente
- Eliminacija – uklanjanje funkcionalnosti iz monolitnog sustava, tek nakon što je funkcionalnost iz istoga pokrivena mikroservisom i spremna za produkciju

Kroz vrijeme *monolitni sustav u funkcionalnom smislu se sve više smanjuje*, uklanjaju mu se funkcionalnosti, koje su pokrivene dovršenim i testiranim mikroservisima i time *„Strangler" aplikacija (sustav) s mikroservisima sve više raste* [27]. U konačnici će „Strangler" sustav u potpunosti zamijeniti monolitni sustav, a slika 8. jasno ilustrira navedeno.



Slika 8. Prikaz „Strangler" dekompozicije [27].

Osim postepene zamjene funkcionalnosti, mikroservisima se mogu dodati nove funkcionalnosti, kako bi se proširilo ponašanje sustava [27]. Prednosti ove dekompozicije su: neovisni timovi (timovi mogu paralelno raditi i na monolitima i na mikroservisima, čime se stvara robustan mehanizam isporuke) i mogućnost korištenja razvoja vođenog testiranjem (eng. *Test Driven Development (TDD)),* jer se usluge razvijaju u komadima pa se TDD može koristiti za poslovnu logiku i osiguranje kvalitete koda [27].

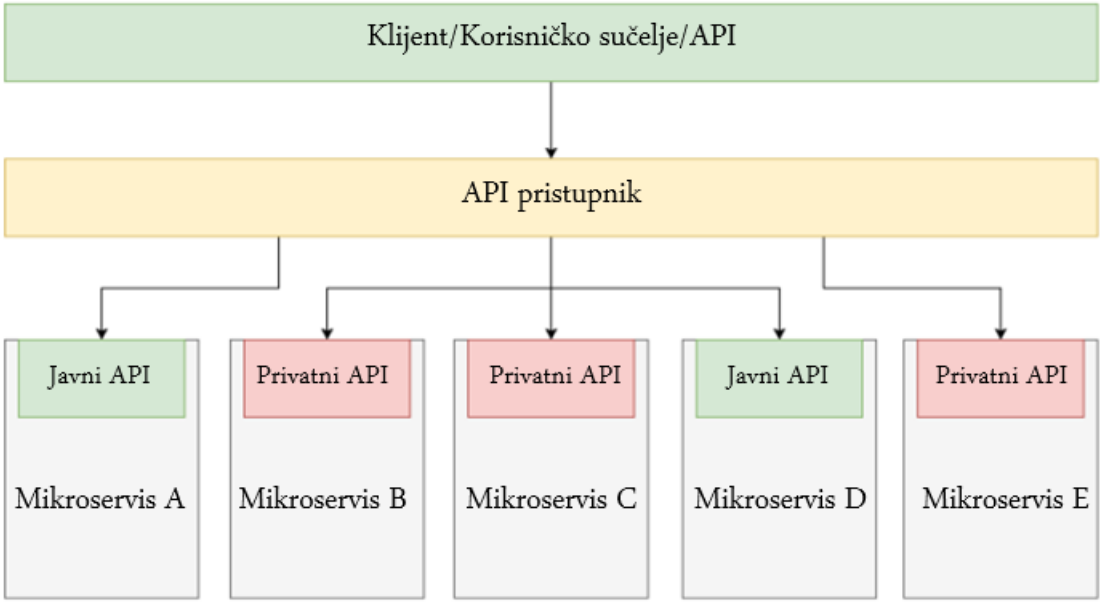
5.2. Uzorci integracije

Uzorci integracije pružaju rješenje za problem omogućavanja klijentima nesmetan pristup do pojedinog mikroservisa, bez vođenja brige oko komunikacijskih protokola i ostalih mogućih složenosti [27]. U mikroservisnoj arhitekturi mikroservisi komuniciraju na različite načine, od korištenja REST-a do asinkronih kanala komunikacije poput AMQP [27]. Stoga je potrebno isključiti od klijenta poznavanje svih navedenih komunikacijskih složenosti i pružiti mu uniformno sučelje za pristup mikroservisima. Prema literaturi [27], postoji šest uzoraka integracije:

- *API pristupnik* (eng. *API Gateway*)
- *Agregator* (eng. *Aggregator*)
- *Proxy*
- *Kompozicija korisničkog sučelja na strani klijenta* (eng. *Client Side UI Composition*)

- *Lanac odgovornosti* (eng. *Chain Of Responsibilties*)
- *Branch*

API pristupnik je jedinstvena točka pristupa svih vrsta klijenta do mikroservisa u sustavu [27]. On djeluje kao centralna točka, koja pravilno usmjerava sve korisničke zahtjeve do potrebnog mikroservisa i time isključuje direktnu komunikaciju klijenta i pojedinog mikroservisa te samu potrebu klijenta za poznavanjem svih mikroservisa u sustavu. Može sadržavati određene sigurnosne mehanizme poput: autentikacije, dozvola i ostalih sigurnosnih validacija, čime pruža jedinstven i siguran centraliziran pristup do svih mikroservisa u sustavu [28]. Klijent se spaja na API pristupnik i zahtijeva određene podatke, pri čemu API pristupnik najčešće provodi sigurnosne validacije klijenta i u slučaju uspješno provedene iste, usmjerava zahtjev korisnika do određenog mikroservisa. Može se reći da se API pristupnik ponaša kao „proxy“ sa sigurnosnim mehanizmima za mikroservise u sustavu. Slika 9. jasno prikazuje poziciju API pristupnika u odnosu na mikroservise, svi zahtjevi klijenta se upućuju prema API pristupniku i dalje usmjeravaju prema određenom API-ju mikroservisa (prije usmjeravanja se najčešće provodi autentikacija klijenta za siguran pristup do mikroservisa, kako bi se spriječilo neautorizirano usmjeravanje tj. neautoriziran pristup do mikroservisa) [28].

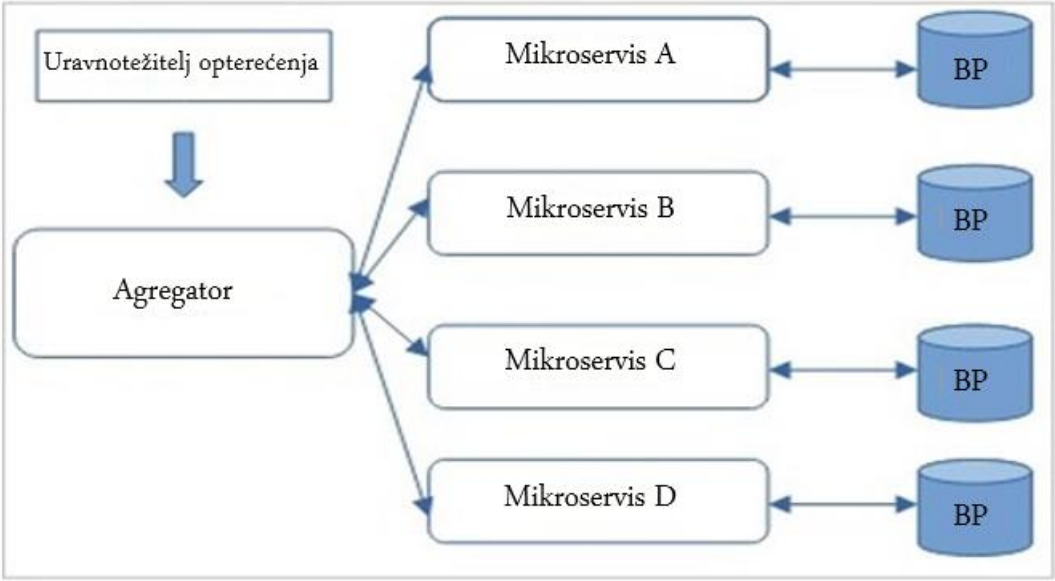


Slika 9. API pristupnik [28].

U mikroservisnoj arhitekturi API pristupnik ima značajnu ulogu i glavne pogodnosti iste su: izolacija klijenta (izolacija klijenta od poznavanja lokacije i načina pozivanja mikroservisa), višestruki pozivi servisa (API pristupnik može rukovati s višestrukim zahtjevima mikroservisa i dati jedinstveni povratni rezultat i time smanjiti povratna putovanja i povećati performanse) i

standardizirano sučelje (pružanje standardiziranog i uniformnog sučelja klijentima za dobivanje odgovora od mikroservisa) [27].

Agregator je jedinstvena pristupna točka za informacije, odgovorna za orkestriranje podataka za klijenta [28]. Koristi se kada određeni zahtjev klijenta uključuje višestruke pozive mikroservisa, jer je konačna informacija za klijenta, sastavljena od niza informacija, koje se dobivaju u odgovoru pozvanih mikroservisa. Time za jedan korisnički zahtjev je potrebno više zahtjeva mikroservisa, a implementacijom agregatora se taj broj smanjuje na jedan, pošto isti obavlja poslovnu logiku agregiranjem i obradom rezultata svih potrebnih mikroservisa [28]. Agregator za korisnički zahtjev, poziva jedan po jedan mikroservis i dobivene informacije agregira i obrađuje u jedinstvenu informaciju za klijenta [27]. Djeluje kao uravnotežitelj opterećenja, povećavanjem performansi isključujući komunikaciju klijenta s više mikroservisa za jedan korisnički zahtjev i pojednostavljuje poslovnu logiku obrade zahtjeva, pozivanjem potrebnih mikroservisa i agregiranjem te obradom dobivenih rezultata u jedan izlazni rezultat za klijenta [27]. Navedeno jasno ilustrira slika 10.

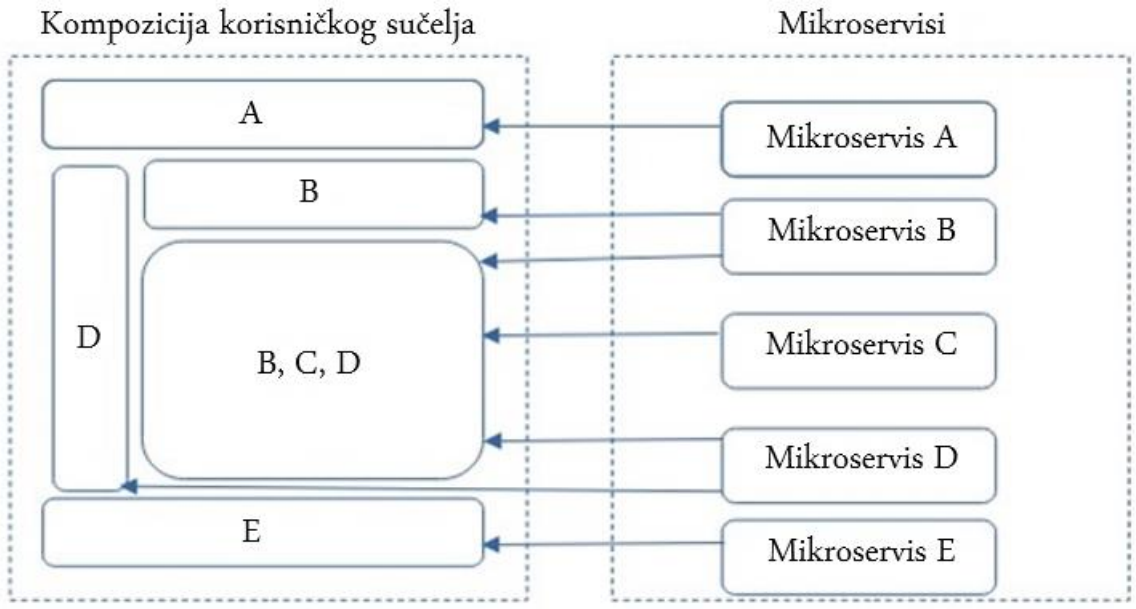


Slika 10. Agregator [27].

Proxy je varijacija uzorka dizajna agregator i koristi se kada se želi pružiti uniformno sučelje za mikroservise, bez potrebe za agregacijom vrijednosti [28]. Za razliku od agregatora, proxy ne provodi poslovnu logiku tj. nema utjecaja na poslovni sloj, već predstavlja tehničku odluku dizajna, pružajući jedinstvenu točku pristupa do mikroservisa [28]. Zapravo je ponašanje proxya vrlo blisku API pristupniku, pošto izvodi preusmjeravanje zahtjeva korisnika do potrebnog mikroservisa. Dakle, proxy provodi izolaciju klijenta od poznavanja lokacije i načina pozivanja mikroservisa, jednako kao i API pristupnik, ali API pristupnik pritom može još

dodatno sadržavati određene sigurnosne mehanizme i ostale mogućnosti. Postoje dva modela proxya: Dumb i Smart proxy [28]. Dumb proxy je model bez ikakve inteligencije, gdje je jedini cilj pružanje jedinstvene točke pristupa do mikroservisa i odgovara dosad opisanom ponašanju proxya [28]. Smart proxy model osim preusmjeravanja zahtjeva, provodi još ovisno o specifičnoj potrebi korisnika, dodatnu modifikaciju podataka prije vraćanja odgovora korisniku [28].

Kompozicija korisničkog sučelja na strani klijenta se odnosi na sastavljanje komponenata korisničkog sučelja, koje mogu prikazivati podatke različitih mikroservisa [27]. Korisničko sučelje se sastoji od više komponenata, gdje svaki razvojni tim razvija komponentu korisničkog sučelja, koja implementira ili odgovara određenom mikroservisu ili više mikroservisa [27]. Na slici 11. je prikazan ogledan primjer ovog uzorka dizajna. Prikazano je korisničko sučelje sastavljeno od komponenata (kompozicija korisničkog sučelja), koje odgovaraju različitim mikroservisima. Komponente korisničkog sučelja: A, B, D i E odgovaraju pripadnim mikroservisima: A, B, D i E, dok komponenta B, C, D za razliku od ostalih komponenata odgovara većem broju mikroservisa, mikroservisima: B, C i D.

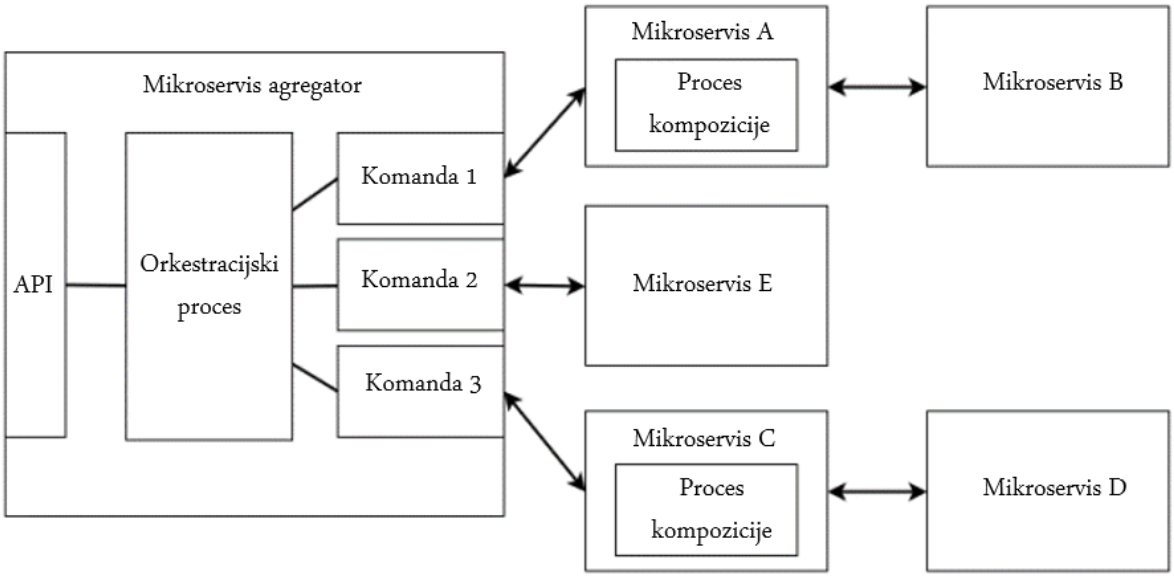


Slika 11. Primjer kompozicije korisničkog sučelja na strani klijenta [27].

Prednosti opisanog uzorka dizajna su: neovisno timovi za razvoj korisničkog sučelja (svaki tim je odgovoran za izgradnju jedne komponente korisničkog sučelja, s čijim razvojem počinje od trenutka dovršenosti potrebnog mikroservisa, bez čekanja na dostupnost ostalih mikroservisa), upravljiv i lakši razvoj korisničkog sučelja (korisničko sučelje razvijeno u komponentama je održivije i efikasnije) [27].

Lanac odgovornosti ne koristi posrednika za indirektan pristup klijenta do mikroservisa, već omogućuje direktan pristup klijenta do mikroservisa [27]. Mikroservisi su ulančani i klijent šalje zahtjev prvom mikroservisu u lancu te ukoliko mikroservis ima nedostajuće podatke za kompletiranje zahtjeva, isti šalje zahtjev za potrebnim podacima idućem mikroservisu u lancu, sve dok mikroservis, koji je dobio zahtjev klijenta konačno ne vrati, duž lanca sastavljen odgovor [28]. Glavni nedostatak ovog pristupa je blokiranje klijenta, sve do završetka procesa ulančavanja, stoga je preporučljivo implementirati što kraći lanac mikroservisa [27].

Branch je kombinacija između uzoraka dizajna: agregator i lanac odgovornosti [28]. Svi zahtjevi klijenata se šalju do API-ja mikroservisa agregatora, koji ima orkestracijsku ulogu [28]. Njegov orkestracijski proces utvrđuje za zahtjev klijenta, koju od definiranih komandi je potrebno aktivirati [28]. Aktiviranje komande predstavlja komunikaciju iste s određenim mikroservisom, koji ovisno o svojoj vrsti, može odmah vratiti odgovor ili provesti proces kompozicije odgovora, daljnjom sinkronom komunikacijom s mikroservisima duž lanca, koji imaju nedostajuće podatke, kako bi se sastavio konačan odgovor [28]. Cijeli opisani postupak funkcioniranja je ilustriran na slici 12.



Slika 12. Prikaz funkcioniranja uzorka dizajna Branch [28].

Velika prednost ovog uzorka dizajna je učahurivanje pristupa do mikroservisa, kao kod API pristupnika uz istovremenu mogućnost kompozicije odgovora i orkestracije [28]. Neki nedostaci istoga su: poteškoće pri otklanjanju pogrešaka, moguće točke kašnjenja i poteškoće u razumijevanju vlasništva podataka [28].

5.3. Uzorci baze podataka

Uzorci baze podataka pružaju rješenje za problem definiranja optimalne strukture (arhitekture) baze podataka i načina upravljanja podacima za mikroservise [27]. Mikroservisi svoj rad baziraju na obradi podataka i komuniciraju s drugim mikroservisima na principu razmjene podataka. Stoga određivanje optimalne strukture baze podataka i podataka je krucijalno. Prema literaturi [27], postoji šest uzoraka baze podataka:

- *Baza podataka po servisu* (eng. *Database per Service*)
- *Zajednička baza podataka po servisu* (eng. *Shared Database per Service*)
- *Command Query Responsibility Segregator* (*CQRS*)
- *Saga*
- *Asinkrona razmjena poruka* (eng. *Asynchronous Messaging*)
- *Event Sourcing*

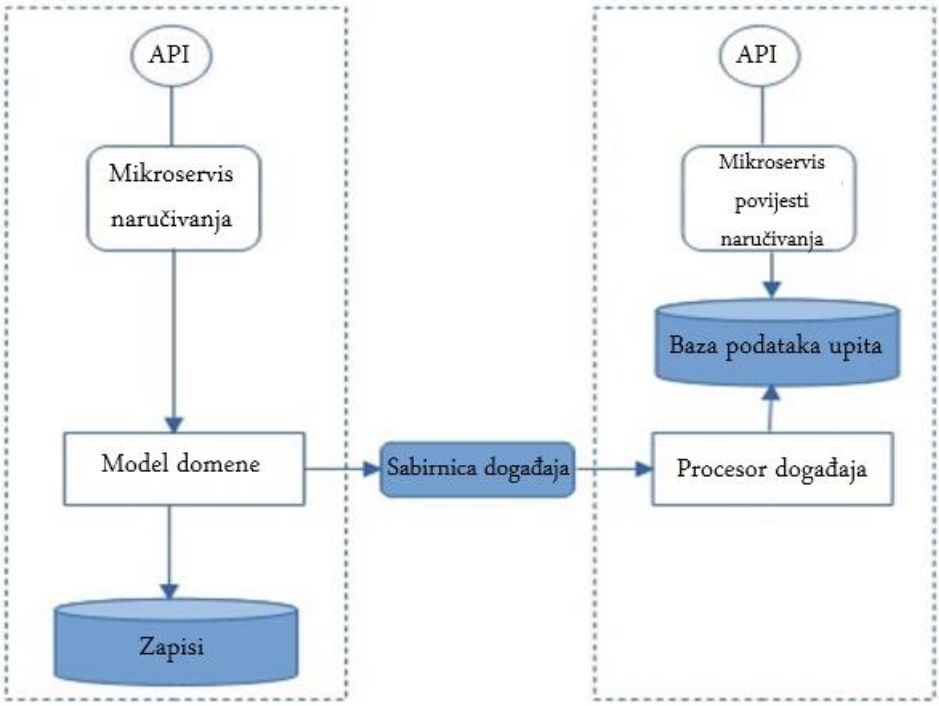
Baza podataka po servisu se odnosi na princip strukturiranja, kod kojeg svaki mikroservis ima vlastitu (privatnu) bazu podataka za obavljanje transakcija [27]. Svaki mikroservis upravlja samo vlastitim podacima, koji su mu potrebni za obavljanje rada, a u slučaju potrebe za nedostajućim podacima, iste dohvaća isključivo kroz komunikaciju s drugim mikroservisima, koji su „vlasnici“ tih podataka. Takva segregacija baze podataka na jednu bazu podataka po mikroservisu, omogućuje veću skalabilnost u podatkovnom sloju i labavu povezanost između mikroservisa [28]. Obično podjela baze podataka domene na više manjih baza podataka pod domena, za koje je odgovoran svaki mikroservis, proizlazi iz uzorka dizajna, dekompozicija prema pod domeni. Ranije spomenuta labava povezanost između mikroservisa (mogućnost neovisne nadogradnje i izmjene mikroservisa) i tehnološka heterogenost baza podataka (mogućnost korištenja tehnologije baze podataka, koja najbolje odgovara odgovornosti mikroservisa) su svakako najveće pogodnosti ovog uzorka dizajna. Međutim, mogući problemi koji se mogu pojaviti su: problem održanja konzistentnosti podataka i otežane provedbe transakcija koje uključuju nedostajuće podatke, koji nisu povezani vanjskim ključevima, jer pripadaju drugim mikroservisima. To rezultira većom potrebom za mrežnom komunikacijom između mikroservisa i težim upravljanjem podataka mikroservisa, što predstavlja nedostatke ovog uzorka dizajna.

Zajednička baza podataka po servisu se odnosi na princip strukturiranja, gdje mikroservisi međusobno dijele podatke tj. baza podataka je jedinstvena za sve mikroservise. Svaki mikroservis može slobodno pristupiti podacima, koji su dostupni i drugim mikroservisima [27]. Dijeljena baza podataka mikroservisa je poput monolitne baze podataka i nije rijedak slučaj, da mikroservisi koriste istu kao privremeno rješenje (kasnije se ista segregira po

principu uzorka, baza podataka po servisu), kod prijelaza iz monolitne na mikroservisnu arhitekturu. Stoga se može smatrat privremenim uzorkom za naslijeđene aplikacije, koje su u fazi tranzicije [28]. Obično se radi i o anti-uzorku, koji se ne bi trebao primjenjivati kod mikroservisne arhitekture, ali postoje slučajevi korištenja, kada su prisutne dvojbe oko strukture podataka, komunikacijski sloj između mikroservisa nije dobro definiran i kod opisanih naslijeđenih (monolitnih) aplikacija u fazi tranzicije [28]. Glavna prednost ovog uzorka dizajna proizlazi iz čestog slučaja korištenja istoga: ubrzanje i olakšavanje procesa segregacije baze podataka i evaluacije konzistentnosti podataka, kod tranzicije s monolitne na mikroservisnu arhitekturu [28]. Veliki nedostatak je da ovisnost svih mikroservisa o bazi podataka predstavlja jedinstvenu točku kvara za mikroservise [28]. Ovime je mikroservisna arhitektura manje pouzdana i ne sadrži svojstvo labave povezanosti mikroservisa, jer neočekivani događaj u podatkovnom sloju jednog mikroservisa, može rezultirati kvarom za sve mikroservise.

CQRS govori o podjeli odgovornosti čitanja i pisanja podataka [28]. Definira se baza podataka za čitanje podataka pod odgovornosti određenog mikroservisa, koja sadrži sinkronizirane podatke tako što se isti pretplaćuje na događaje, okinute od strane drugih servisa, koji su „vlasnici“ podataka [27]. Drugi mikroservisi izvršavaju komande pisanja u primarnoj bazi podataka domene i nakon toga okidaju događaje, kako bi mikroservis pretplaćen na te događaje mogao imati aktualno stanje podataka u bazi podataka namijenjenoj za čitanje [27]. Dakle, baza podataka za čitanje sadrži sinkronizirane podatke primarne baze podataka domene u kojoj se izvršavaju akcije pisanja. Može se reći da baza podataka za čitanje sadrži uvijek ažurnu kopiju podataka iz primarne baze podataka. Mikroservis koji upravlja opisanom bazom podataka za čitanje je zadužen da opskrbljuje korisničke poglede uvijek ažurnim podacima. Razlog korištenja takvog pristupa prisutnosti dviju baza podataka: baze podataka za čitanje i primarne baze podataka domene za pisanje je velika opterećenost samo jedne baze podataka zahtjevima za akcijama i čitanja i pisanja [28]. Nad tom jednom bazom podataka zaduženom za određenu domenu (pod domenu) mikroservisa se obavljaju sve CRUD akcije, što utječe na performanse baze podataka i može dovesti do zastoja, prekida i sporosti iste [28]. Zato se javlja potreba za rasterećenjem i ubrzanjem performansi baze podataka, opisanim pristupom ovog uzorka dizajna. Situacija koja također ukazuje na potrebu za primjenom ovog uzorka je kada prikazani podaci korisnicima su zastarjeli (nisu ažurni), jer su možda već promijenjeni od strane drugih korisnika [28]. Ovdje su dostupni uvijek ažurni podaci korisnicima iz baze podataka za čitanje, koje mikroservis zadužen za isto sinkronizira kroz asinkronu komunikaciju s drugim mikroservisima, koji izvršavaju akcije pisanja u primarnoj bazi podataka domene. Slika 13. prikazuje primjer strukture opisanog uzorka dizajna. Za model domene postoji primarna baza podataka na kojoj mikroservisi izvršavaju akcije pisanja

i šalju događaje do mikroservisa zaduženog za upravljanje bazom podataka za čitanje, da ažurira stanje podataka u istoj [27].

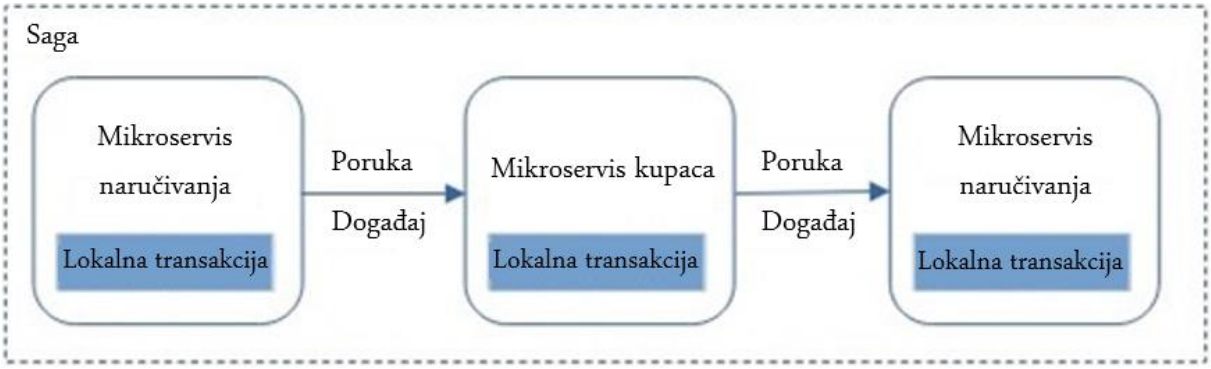


Slika 13. Primjer strukture CQRS-a [27].

Glavne prednosti ovog uzorka dizajna su: akcije čitanja i pisanja se ne natječu za isti resurs i smanjenje opterećenja baze podataka [28]. Glavni nedostatak istoga je sama kompleksnost izgradnje takvog pristupa, stoga je važno da se isti primjenjuje samo onda, kada postoji nužna potreba za time [28].

Saga se koristi kao rješenje problema, kada u međusobnoj suradnji mikroservisa postoji potreba za distribuiranim transakcijama i pritom je izazovno održavanje konzistentnosti podataka, ako se transakcija proteže kroz više mikroservisa [29]. Ona je skup lokalnih sekvencijalnih transakcija, od kojih svaka od njih ažurira bazu podataka mikroservisa i okida događaj ili asinkrono šalje poruku za aktiviranjem iduće transakcije u sagi [27]. Navedene transakcije su međusobno povezane i zajedno čine rješenje zadatka domene. Ukoliko bilo koja od tih transakcija doživi neuspjeh, saga će aktivirati slijed transakcija za poništavanje svega dosad učinjenog od strane istih [27]. Saga transakcije ne poštuju u potpunosti ACID (atomicity, consistency, isolation, durability) princip transakcije, već poštuju ACD (atomicity, consistency, durability) princip transakcije, u kojem nedostaje izolacija, što znači da je dopušteno čitanje i pisanje iz nedovršene transakcije i to može rezultirati određenim anomalijama, zbog čega se javlja potreba za poduzimanjem protumjera [29]. Kod implementacije sage se obično mogu

koristiti dva pristupa: koreografija ili orkestracija [29]. U slučaju pristupa koreografije, mikroservisi sami obrađuju događaje u domeni tijekom lokalnih transakcija i zatim dovršavaju ili poništavaju transakciju, dok kod pristupa orkestracije objekt orkestratora rukuje događajima tijekom lokalnih transakcija i nakon toga govori, koja od idućih lokalnih transakcija treba biti aktivirana [27]. Slika 14. prikazuje funkcioniranje ovog uzorka dizajna. Koristi se pristup koreografije, budući da svaki mikroservis obavlja svoju lokalnu transakciju iz koje šalje poruku ili okida događaj do idućeg mikroservisa, za aktiviranje njegove lokalne transakcije.

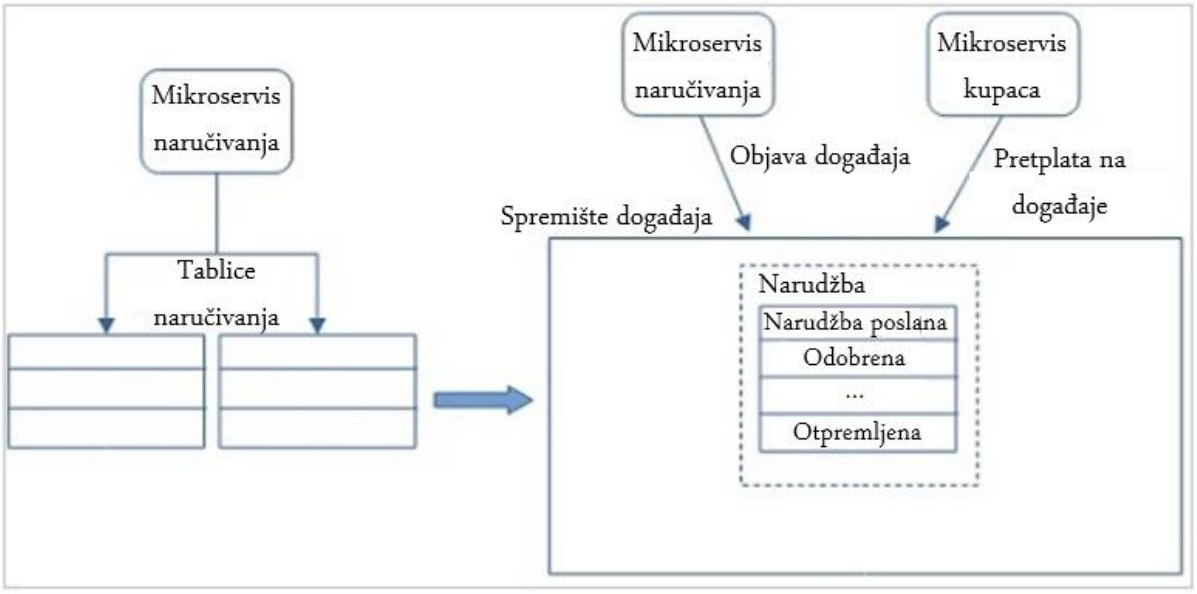


Slika 14. Primjer uzorka dizajna Saga [27].

U sagi je također osigurana prevencija pogrešnih izvršenja akcija u bazi podataka, pošto se sve akcije lokalnih transakcija spremaju na memorijskoj razini, obično u pred memoriji (eng. *cache*) i tek u slučaju uspješnog izvršenja svih lokalnih transakcija se podaci iz pred memorije sinkroniziraju u bazi podataka [29]. Takvim premještanjem transakcija iz razine baze podataka na memorijsku razinu, ublažava se nedostatak izolacije iz ACID principa [29].

Asinkrona razmjena poruka se odnosi na strukturiranje labavo povezane i ne blokirajuće komunikacije između mikroservisa. Obično je temeljena na događajima i mikroservis može okidati ili osluškivati događaje. Okidanje događaja podrazumijeva slanje nove poruke u red poruka teme i svi mikroservisi, koji osluškuju događaj tj. pretplaćeni su na temu konzumiraju poruku i izvode daljnje akcije. Primjeri tehnologija za implementaciju ovog uzorka dizajna su: RabbitMQ i Apache Kafka [27]. Detaljan opis i razrada istoga je ranije pružan kod poglavlja „Tehnologije sinkrone i asinkrone komunikacije“ za mikroservise.

Event Sourcing je povezano s labavo povezanom, asinkronom komunikacijom mikroservisa preko događaja. Svaki mikroservis zadržava (eng. *persist*) događaje u spremištu događaja (eng. *event store*) za određene poduzete akcije [27]. Isto tako svaki mikroservis se može pretplatiti na te događaje i sukladno tome ažurira svoj status transakcije tj. svoje stanje podataka [27]. Slika 15. prikazuje izgled ovog uzorka dizajna. Postoje dva mikroservisa, od kojih jedan objavljuje događaje za određeni objekt domene u spremište događaja, a drugi je pretplaćen na iste. Pogodnost uzorka dizajna se ogleda kao implementacija arhitekture vođene događajem (eng. *Event Driven Architecture*) [27].



Slika 15. Event Sourcing [27].

5.4. Uzorci uočljivosti

Uzorci uočljivosti pružaju rješenje za problem praćenja statusa mikroservisa u mikroservisnoj arhitekturi u pogledu njihovih performansi, „zdravlja“ i zapisnika sustava [27]. Prema literaturi [27], postoji četiri uzoraka uočljivosti:

- *Agregacija dnevnika* (eng. *Log Aggregation*)
- *Metrike performansi* (eng. *Performance Metrics*)
- *Distribuirano praćenje* (eng. *Distributed Tracking*)
- *Provjera zdravlja* (eng. *Health Check*)

Agregacija dnevnika pomaže u pohrani zapisa oko korisničkog zahtjeva, koji obuhvaća pozive više mikroservise [27]. Prvi obuhvaćeni mikroservis generira korelacijski identifikator (eng. *Corelation ID*) prema kojem se bilježe i prate zapisi, za upućeni korisnički zahtjev (poziv)

i prosljeđuje dalje isti drugim mikroservisima, koji također sudjeluju u obradi zahtjeva, kako bi i oni mogli pohranjivat zapise prema istom [27]. Može se koristiti centralizirani mikroservis za zapise (eng. *centralized logging service*), koji agregira zapise svih mikroservisa i korisnik je u mogućnosti pretraživati i analizirati zapise na jednom mjestu [27].

Metrike performansi služe kao rješenje za praćenje performansi aplikacija i provjere uskih grla kao i mogućnost praćenja više mikroservisa uz minimalne režijske troškove izvođenja (eng. *Runtime Overhead*) [27]. Rješenje je u vidu implementacije centralnog servisa za metrike (eng. *central metrics service*), koji bi trebao agregirati metrike mikroservisa i pružati izvještavanje i upozoravanje [27]. Navedeni servis može prikupljati podatke metrika performansi mikroservisa na sljedeća dva načina: „push“ (mikroservisi šalju svoje metrike do centralnog servisa za metrike) i „pull“ (centralni servis za metrike zahtijeva metrike od mikroservisa) [27]. Najpoznatiji primjer tehnologije koja se koristi za opisanu svrhu je Prometheus [27].

Distribuirano praćenje služi kao rješenje za praćenje razbacanih zapisa kroz više mikroservisa, obično kada mikroservisi koriste baze podataka, redove poruka i Event Sourcing [27]. Za implementaciju ovog uzorka dizajna koriste se sljedeće operacije: kreiranje korelacijskog identifikatora (identifikator koji se dodjeljuje zahtjevu i dijeli između svih mikroservisa, koji sudjeluju u tom zahtjevu), vođenje zapisa prema korelacijskom identifikatoru (bilo koja poruka zapisa, generirana od strane sudjelujućeg mikroservisa bi trebala imati uz sebe korelacijski identifikator) i bilježenje pojedinosti u zapisima (bilježenje vremena početka i završetka te druge relevantne pojedinosti u zapisima, kada mikroservis obrađuje zahtjev) [27].

Provjera zdravlja pruža mehanizam za prepoznavanje nedostupnih mikroservisa i sprječavanje upućivanja zahtjeva do istih te detektiranje može li nedostupna instanca mikroservisa uopće obraditi zahtjev [27]. Mehanizam se implementira dodavajući HTTP putanju „/health“ svakom mikroservisu, koja vraća status „zdravlja“ istoga [27]. Operacije koje može provesti navedena putanja odnosno krajnja točka mikroservisa su: provjera statusa veze (status veze mikroservisa s bazom podataka ili status veze s infrastrukturnim uslugama, koje koristi isti), provjera statusa domaćina (status domaćina poput diskovnog prostora, upotrebe memorije i procesora itd.), provjera aplikacijski specifične logike (poslovna logika koja određuje dostupnost mikroservisa) [27]. Sada servis zadužen za praćenje zdravlja, može periodično pozivati krajnju točku mikroservisa za provjeru zdravlja istoga, kako se ne bi dogodilo, da upućeni zahtjev stigne do nedostupnog mikroservisa [27]. Zato servis zadužen za praćenje zdravlja mora imati centralnu ulogu poput API pristupnika, unutar kojeg je pogodno implementirati opisanu logiku praćenja zdravlja.

5.5. Uzorci međusobne brige

Uzorci međusobne brige pomažu u rješavanju određenih problema karakterističnih za mikroservise. Prema literaturi [27], postoji četiri uzoraka međusobne brige:

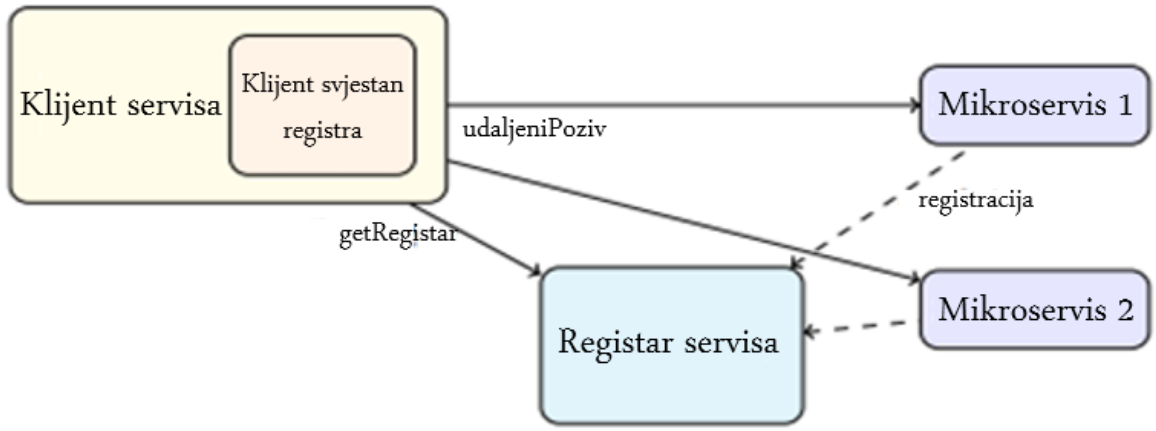
- *Eksterna konfiguracija* (eng. *External Configuration*)
- *Otkrivanje servisa (*eng. *Service Discovery)*
- *Prekidač (*eng*. Circuit Breaker)*
- *Razmještaj plavozelenog okruženja (*eng*. Blue Green Deployment)*

Eksterna konfiguracija eksternalizira sve konfiguracije mikroservisa na jedno mjesto pod odgovornosti konfiguracijskog servisa [27]. Konfiguracijski servis je zadužen za upravljanje svim konfiguracijama mikroservisa i mikroservisi prilikom svojeg pokretanja kontaktiraju isti, za dobivanje najsvježijih konfiguracijskih podataka, koji se spremaju u varijable okoline mikroservisa [27]. Nakon toga su aktualni konfiguracijski podaci dostupni mikroservisima preko varijabli okolina. Naime, mikroservisi mogu komunicirati s mnogo vanjskih infrastrukturnih servisa i servisa trećih strana, koji mogu varirati (npr. okruženje baze podataka može biti razvojno, testno, produkcijsko itd.) i za čiji pristup su potrebni određeni varijabilni konfiguracijski podaci te isto tako može varirati okruženje u kojem se mikroservisi trenutno nalaze [27]. Zato je potrebno spremiti konfiguracije na jedno mjesto dostupno svim mikroservisima, koji zatim mogu biti razmješteni u različita okruženja (razvojno, testno, produkcijsko) bez potrebe za ikakvom modifikacijom koda istih [27].

Otkrivanje servisa se koristi kada postoje mikroservisi spremljeni u kontejnerska ili virtualna okruženja, čiji se broj instanci i lokacija dinamički mijenja [27]. Zato ovaj uzorak dizajna pruža mehanizam, omogućivanjem klijentima mikroservisa upućivanje zahtjeva dinamički promjenjivim instancama istih [27]. Registar servisa (eng. *service registry*) je servis koji upravlja informacijama o lokacijama mikroservisa i pruža otkrivanje servisa (eng. service discovery) za klijente, kako bi isti dobili informacije o trenutnoj lokaciji određenog mikroservisa [30]. Mikroservisi sami kontaktiraju i objavljuju svoju lokaciju u registar servisa [30]. Prilikom otkrivanja servisa, registar servisa može pružati lokaciju trenutno dostupne instance istoga, djelujući kao uravnotežitelj opterećenja. Time se uklanja potreba mikroservisa za poznavanjem lokacija svih mikroservisa s kojima komunicira. Prema literaturi [30], postoje dvije strategije implementacije ovog uzorka dizajna:

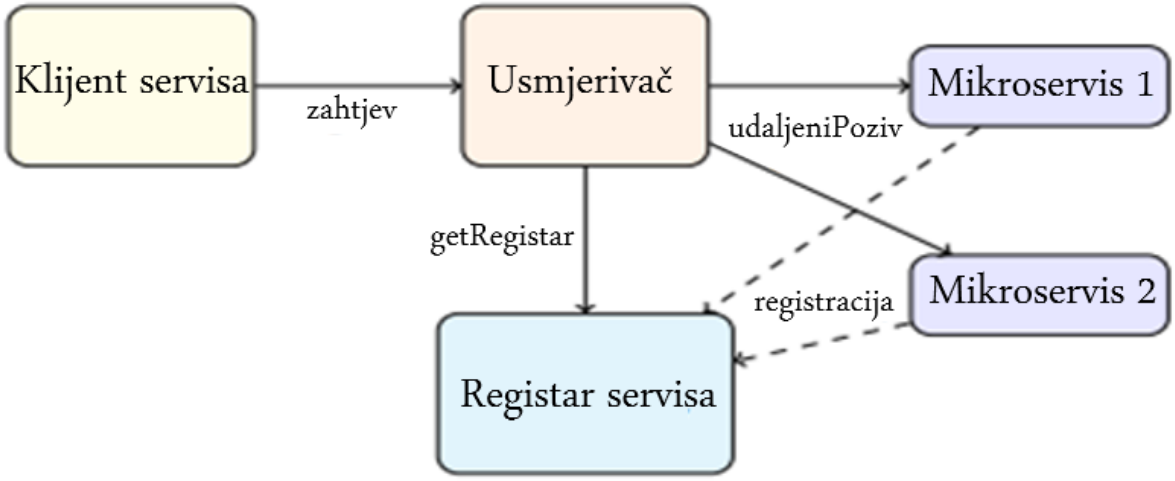
- Uzorak otkrivanja na strani klijenta (eng. *client-side discovery pattern*)
- Uzorak otkrivanja na strani poslužitelja (eng. *server-side discovery pattern*)

Kod uzorka otkrivanja na strani klijenta, klijent je svjestan postojanja registra servisa, kojeg direktno kontaktira za dobivanje željenih lokacija mikroservisa [30]. Nakon dobivanja lokacija mikroservisa od registra servisa, klijent direktno kontaktira iste upućivanjem zahtjeva [30]. Može se reći da klijent otkriva lokacije mikroservisa, a sve opisano je zorno prikazano na slici 16.



Slika 16. Uzorak otkrivanja na strani klijenta [30].

S druge strane kod uzorka otkrivanja na strani poslužitelja, otkrivanje lokacija servisa je prebačeno s klijenta na poseban usmjerivač (eng. *router*), koji direktno kontaktira registar servisa za dobivanje željenih lokacija mikroservisa od strane klijenta i zatim prosljeđuje zahtjeve klijenta do istih [30]. To rezultira smanjenjem kompleksnosti koda na strani klijenta.



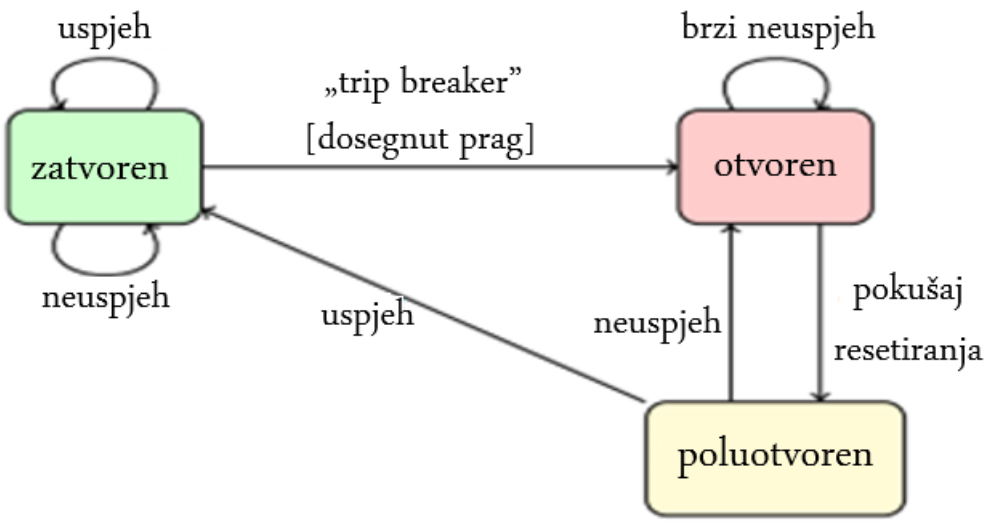
Slika 17. Uzorak otkrivanja na strani poslužitelja [30].

Potreba za ovim uzorkom dizajna se javlja kod aplikacija širokih razmjera [30]. Isti nije standardiziran, već postoje prilagođene implementacije poput: Eureka i AWS Elastic Load Balancing [30].

*Prekidač (*eng*. Circuit Breaker)* rješava problem kaskadnog kvara mikroservisa, kada nedostupnost jednog mikroservisa može utjecati na nedostupnost drugih mikroservisa, koji ovise o istome [30]. Treba omogućiti mehanizam, koji neće prosljeđivati zahtjeve, ako je samo jedan mikroservis nedostupan, čime će se spriječiti kaskadan kvar mikroservisa [27]. Glavni moto ovog uzorka dizajna je: „kada servis ne reagira, pozivatelji istoga bi trebali prestati čekati na njega i početi se baviti činjenicom, da je on možda nedostupan“ [30]. Princip rada prekidača je prosljeđivanje poziva prema ciljanom mikroservisu i praćenje stope kvarova istih [30]. Kada mikroservis postane nedostupan, prekidač će isključiti prosljeđivanje poziva prema istome i odmah će javiti grešku [30]. Prema literaturi [30], ponašanje prekidača je poput konačnog stroja stanja sa sljedećim stanjima i pripadnim ponašanjem u istima:

- Zatvoren – omogućeno je prosljeđivanje zahtjeva prema ciljanom mikroservisu i ukoliko kod obrade zahtjeva dođe do iznimaka ili isteka vremena, povećava se određeni brojač kvarova i isteka vremena prekidača. Ako vrijednost brojača premaši određeni prag, prekidač ulazi u otvoreno stanje
- Otvoren – zahtjevi se ne prosljeđuju prema ciljanom mikroservisu i umjesto toga se odmah vraća poruka neuspjeha. Prekidač može iz ovog stanja prijeći u poluotvoreno stanje, periodičkim ispitivanjem ponovne dostupnosti mikroservisa ili nakon određenog vremena
- Poluotvoren – dopušten je ograničen broj zahtjeva prema mikroservisu. Ukoliko mikroservis šalje uspješne odgovore, prekidač se vraća u zatvoreno stanje i resetiraju se njegovi brojači kvarova i isteka vremena. Ako bilo koji od zahtjeva doživi neuspjeh, prekidač se vraća u otvoreno stanje

Slika 18. prikazuje dijagram stanja prekidača, koji prati prethodno opisano ponašanje istoga kao konačan stroj stanja.



Slika 18. Dijagram stanja prekidača [30].

Postoje tri načina implementacije prekidača: prekidač na strani klijenta (eng. Client-side Circuit Breaker), prekidač na strani servisa (eng. Service-side Circuit Breaker) i Proxy prekidač [30]. Kod prekidača na strani klijenta, svaki klijent ima svoj prekidač za presretanje poziva klijenta prema servisu [30]. Prednost istoga je kada se prekidač nalazi u otvorenom stanju, ciljani servis neće dobiti ni jedan zahtjev od klijenta i nije potrebno, da servis implementira sličan zaštitni mehanizam [30]. Nedostaci istoga su: implementacija koda kod klijenta i poznavanje dostupnosti servisa lokalno od strane klijenta [30]. Kod prekidača na strani servisa, prekidač se implementira unutar svakog servisa, koji provjera sve zahtjeve prema servisu te odlučuje oko procesiranja istih [30]. Izazovi kod ove implementacije su: potreba za promjenom ponašanja servisa (npr. promjene u izvornom kodu) i veće trošenje resursa istoga [30]. Najpogodniji način implementacije je Proxy prekidač, koji predstavlja kombinaciju prethodno opisanih načina [30]. U ovom načinu, prekidač je smješten točno između klijenta i servisa i upravlja zahtjevima između istih [30]. Postoji prekidač za svakog klijenta i za svaki servis u sustavu [30]. Glavne prednosti ovog načina su: komunikacija klijenta sa servisima je indirektna preko proxya (nema potrebe za implementacijom logike prekidača kod klijenta i servisa, već je ista implementirana na jednom mjestu, kod proxya) i klijent i servis su podjednako zaštićeni jedan od drugoga, klijent od nedostupnog servisa i servis od klijenta, koji šalje mnogo zahtjeva [30].

Razmještaj plavozelenog okruženja (eng. Blue Green Deployment) pomaže u smanjenju vremena zastoja ili čak sprječavanju zastoja prilikom migracije s monolitnog na mikroservisni sustav [27]. Korisnički promet se preusmjerava pomoću uravnotežitelja opterećenja tj. API pristupnika sa stare monolitne aplikacije na novi mikroservis, koji pokriva određenu funkcionalnost iste i spreman je za produkciju [27]. Razlikuju se dva okruženja: plavo

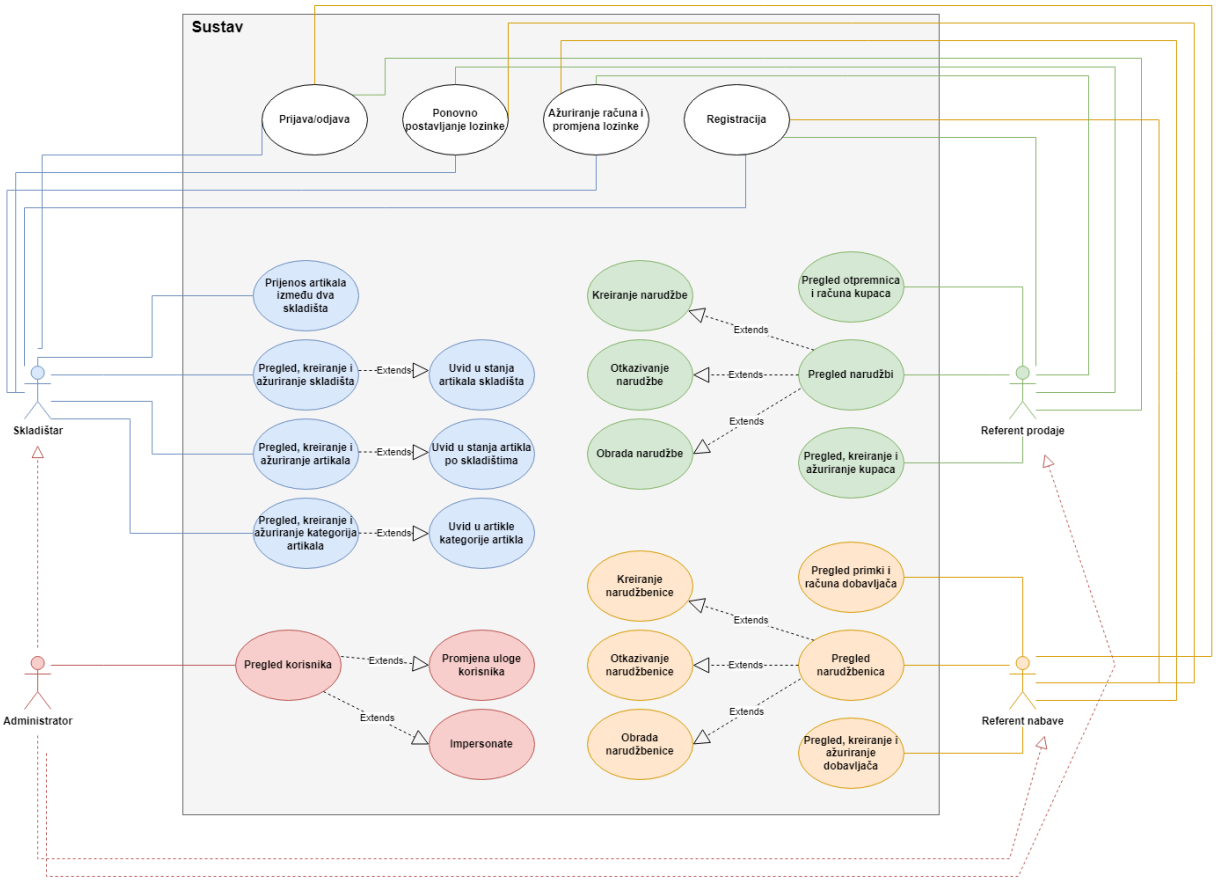
(eng. *Blue Environment*) i zeleno okruženje (eng. *Green Environment*) [27]. U plavom okruženju se nalazi stara monolitna aplikacija u produkciji, a u zelenom okruženju su novo razmješteni mikroservisi, koji repliciraju određeni dio stare monolitne aplikacije [27]. Kroz određeno vrijeme se funkcionalnosti monolitne aplikacije u plavom okruženju smanjuju, jer se prebacuju iste na novo razvijene mikroservise u zelenom okruženju [27]. Ovaj uzorak dizajna je povezan s ranije opisanim uzorkom dekompozicije „Strangler dekompozicija“.

6. Sustav za upravljanje skladištem

Sustav za upravljanje skladištem je izrađen na *principima mikroservisne arhitekture*. Sastoji se od korisničke (korisnička web aplikacija SPA arhitekture) i poslužiteljske strane (pozadinski dio sustava temeljen na mikroservisnoj arhitekturi). *Aplikacijska domena sustava za upravljanje skladištem* se odnosi na *zamišljeno trgovačko poduzeće, generičkih poslovnih procesa karakterističnih za isto*. Ista je podijeljena na više pod domena, gdje je *za svaku pod domenu zadužen jedan mikroservis poslužiteljske strane*, što se može iščitati i iz samih naziva mikroservisa. Za izradu sustava je korištena kombinacija više modernih tehnologija pogodnih za tu svrhu. U nastavku će se prvo ilustrirati i opisati način korištenja sustava, što će poslužiti kao pogodan uvod u izrađeni sustav i njegovu aplikacijsku domenu. Zatim će slijediti opis korištenih tehnologija za izradu sustava. Nakon toga će biti pružan detaljan opis sustava i njegovih komponenti uz prikaz i opis arhitekture istoga. Na kraju će se prikazati način funkcioniranja sustava i način njegove realizacije uz korištenje: isječaka kodova i slika ekrana korisničke web aplikacije.

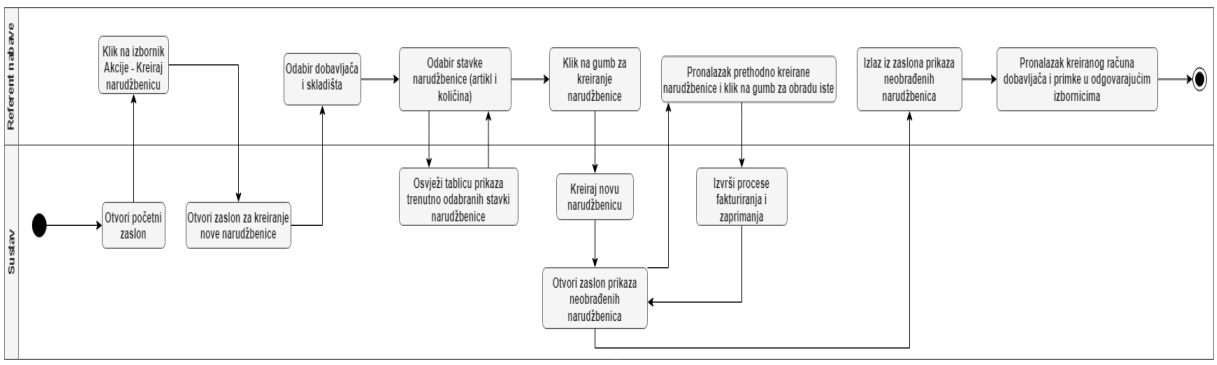
6.1. Način korištenja sustava

Prikaz načina korištenja sustava od strane korisnika s pojedinom ulogom je ilustriran na *dijagramu slučajeva korištenja*, prikazanim na idućoj slici 19. Svakom korisniku su dostupne funkcionalnosti karakteristične za njegovu poslovnu ulogu (funkcionalnosti su istaknute različitim bojama, zbog lakšeg praćenja pripadnosti pojedinoj ulozi korisnika). Postoje i generalne funkcionalnosti (označene bijelom bojom) dostupne svakom korisniku sustava. Iz funkcionalnosti i uloge pojedinog korisnika se mogu uvidjeti poslovni procesi aplikacijske domene, koje izrađeni sustav podržava. Ovo ujedno predstavlja *funkcionalne zahtjeve sustava*. *Sustav prati, upravlja i bilježi promjene artikala na skladištu, sukladno izvršenjima procesa nabave i prodaje, uobičajenima za trgovačka poduzeća*, što će detaljnije biti opisano kroz dijagrame aktivnosti.

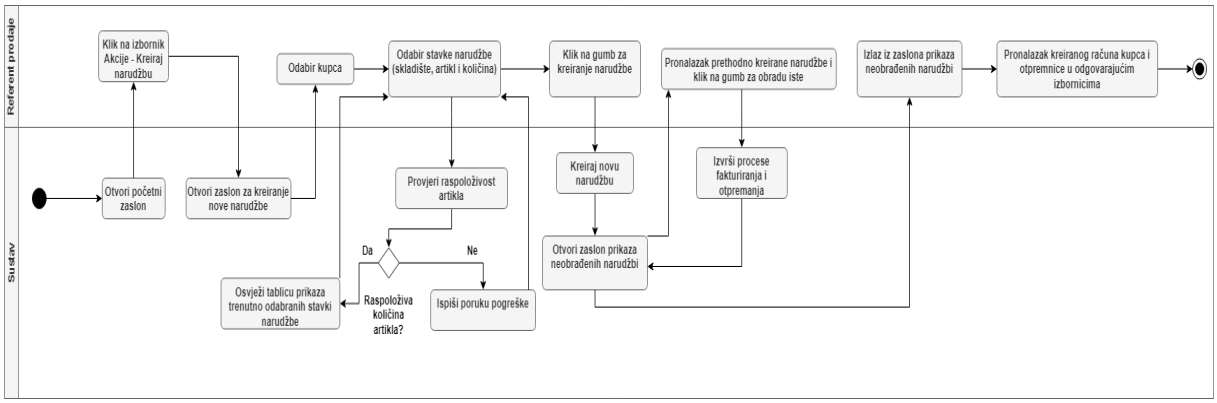


Slika 19. Dijagram slučajeva korištenja [autorski rad].

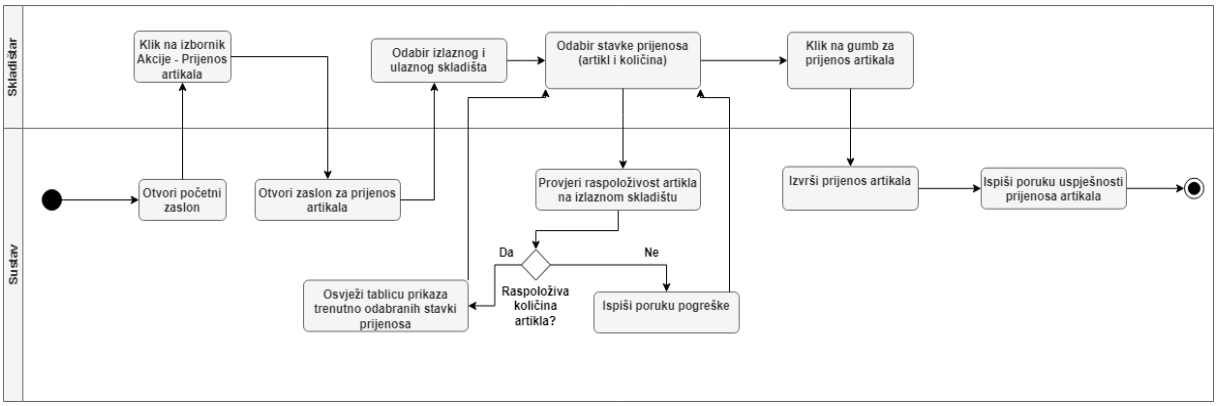
Prilikom korištenja sustava, *najčešće aktivnosti korisnika sustava pojedine uloge* su: *kreiranje i obrada narudžbenice* (izvodi referent nabave), *kreiranje i obrada narudžbe* (izvodi referent prodaje) i *prijenos artikala između dva skladišta* (izvodi skladištar). U nastavku je prikazan dijagram aktivnosti za svaku navedenu aktivnost. *Sve aktivnosti rezultiraju promjenom tj. novim stanjem artikala na skladištu, što je jedan od glavnih zadataka upravljanja skladištem sustava. U upravljanje skladištem pripadaju i ostale mogućnosti korisnika s ulogom skladištara.*



Slika 20. Dijagram aktivnosti kreiranja i obrade narudžbenice [autorski rad].



Slika 21. Dijagram aktivnosti kreiranja i obrade narudžbe [autorski rad].



Slika 22. Dijagram aktivnosti prijenosa artikala između dva skladišta [autorski rad].

6.2. Opis korištenih tehnologija

Za izradu mikroservisa i ostalih komponenti mikroservisne arhitekture (API Gateway i servis konfiguracije), korištene su sljedeće tehnologije: Node.js i Java Spring Boot. Asinkrono komuniciranje mikroservisa je implementirano pomoću RabbitMQ tehnologije. Korištene tehnologije za baze podataka su: MongoDb i MySQL. Korisnička web aplikacija je izrađena u React.js tehnologiji. Za svaku od spomenutih tehnologija će se pružati sažeti opis značajki pojedine tehnologije, koje su korištene kod izrade sustava.

6.2.1. Node.js

Node.js je poslužiteljsko JavaScript okruženje temeljeno na Google-ovoj implementaciji procesorskog vremena, tzv. „V8 stroj (eng. *V8 engine*)“ [31]. Node.js i V8 su razvijeni u programskim jezicima: C i C++ s fokusom na performanse i malu memorijsku potrošnju [31]. U odnosu na ostala moderna razvojna okruženja, Node proces se ne oslanja na višedretvenost, već je baziran asinkronom ulazno-izlaznom modelu događaja (eng. *asynchronous I/O eventing model)* [31]. Stoga se Node poslužiteljski proces može zamisliti kao jednodretveni pozadinski proces (eng. *daemon*), koji ugrađuje JavaScript mehanizam (eng. *JavaScript engine*) za podršku prilagodbe [31]. Time za izgradnju Node.js aplikacija se koristi sintaksa programskog jezika JavaScript-a i neke korisne značajke istoga poput povratnih poziva događaja (eng. *event callbacks*) [31]. Razvojni inženjeri koji imaju iskustva u radu s JavaScript-om mogu se vrlo lagano prebaciti na programiranje na strani poslužitelja u Node.js-u, pošto se koristi, kao što je prethodno navedeno, ista sintaksa za pisanje programskog koda. Glavna pogodnost Node.js-a je brzi razvoj skalabilnih mrežnih aplikacija poput web servisa, jer se istovremeno može upravljati s više veza, gdje se nakon svake veze pokreće povratni poziv (eng. *callback*), a Node proces „spava“ kada nema posla [32]. Budući da Node.js nije temeljen na modelu višedretvenosti i pripadne zastupljene konkurentnosti, u istome nema blokiranja pa je skalabilne sustave vrlo razumno razvijati u istome [32]. Node.js aplikacije se najčešće koriste uz MongoDB bazu podataka, jer u Node.js-u postoji pogodna podrška u obliku JavaScript objekata za istu. Za korištenje JWT-a (eng. *JSON Web Token*) u svrhu autorizacije, postoji također iznimno odlična podrška. Zbog navedenih razloga, Node.js je korišten za razvoj sljedećih komponenti sustava: API Gateway i mikroservis upravljanje korisnicima.

6.2.2. Java Spring Boot

Java Spring Boot je svjetski najpoznatiji Java razvojni okvir, fokusiran na brzini, jednostavnosti i produktivnosti, jer omogućava lakše, brže i sigurnije programiranje u Javi [33]. Prema literaturi [33], glavnih šest prednosti, koje ujedno i ukratko opisuju ključne mogućnosti ovog Java razvojnog okvira su:

- Sveprisutnost – programeri diljem svijeta vjeruju bibliotekama Spring Boot-a i isto tako ima doprinose od svih poznatih velikih imena u tehnologiji poput: Alibaba, Amazon, Google, Microsoft itd.
- Fleksibilnost – veliki skup proširenja i biblioteka trećih strana omogućava izradu gotovo svake zamislive aplikacije. U svojoj jezgri značajke: inverzija kontrole (eng. *inversion of control (IoC))* i inverzija ovisnosti (eng. *dependency injection (DI)*) pružaju temelj za široki raspon značajki i funkcionalnosti
- Produktivnost – za što jednostavniji razvoj mikroservisa, Spring Boot kombinira potrebe poput: konteksta aplikacije (eng. *application context*) i automatski konfiguriranog, ugrađenog web poslužitelja (ugrađeni Tomcat kontejner). Spring Boot također sadrži niz biblioteka i ostalih potrebnih stvari (Spring Cloud) za siguran razmještaj cijelih sustava baziranih na mikroservisnoj arhitekturi u oblak
- Brzina – prisutnost brzog pokretanja, gašenja i optimiziranog izvršavanja aplikacija prema osnovi. Sve je to moguće zahvaljujući ugrađenim web poslužiteljima, automatskoj konfiguraciji i ostalim zastupljenim paradigmama, kao što su: reaktivni (ne blokirajući) model programiranja i podrška za „LiveReload“. Prisutno je također vrlo brzo pokretanje novog projekta u svrhu izrade aplikacije u nekoliko sekundi, koristeći „Spring Initializr“ na „start.spring.io“, koji kreira novi Java Spring Boot projekt sa svim potrebnim ovisnostima
- Sigurnost – postoji dokazano iskustvo u brzom i odgovornom rješavanju sigurnosnih problema. Razvojni inženjeri zaduženi za održavanje i razvoj ovog okvira surađuju sa sigurnosnim profesionalcima, za potrebe zakrpe i testiranja svih prijavljenih ranjivosti i također postoji redovito praćenje svih ovisnosti trećih strana. Zahvaljujući tome, razvijene Spring Boot aplikacije su sigurnije
- Podrška – iznimno velika Spring zajednica globalno rasprostranjena pruža široki raspon resursa i podrške kod razvoja aplikacija

Uz sve navedene pogodnosti ovog Java razvojnog okvira, može se lako zaključiti da je isti idealan za brz i siguran razvoj mikroservisa. Isti je korišten za izradu ključnih mikroservisa poslovne domene: mikroservis nabave, mikroservis upravljanje artiklima i mikroservis prodaje te servisa konfiguracije za eksternalizaciju konfiguracije tih mikroservisa (postoji odlična podrška okvira za isto).

6.2.3. RabbitMQ

RabbitMQ je najrasprostranjeniji posrednik poruka otvorenog koda [34]. Vrlo je jednostavan i lagan za razmještaj „on premise“ i u oblaku [34]. Pogodan je za izvršavanje na mnogim operacijskim sustavima i oblačnim okruženjima te pruža široki raspon razvojnih alata za najpopularnije programske jezike [34]. Prema literaturi [34], najvažnije značajke istoga su:

- Asinkrona razmjena poruka (razni protokoli za razmjenu poruka, vrste razmjene poruka itd.)
- Iskustvo za razvojne inženjere - razmještaj u nove tehnologije poput: Docker i Kubernetes. Razvoj mehanizma razmjene poruka u mnogim programskim jezicima, kao što su: Java, .Net, JavaScript, Python itd.
- Distribuiran razmještaj – razmještaj u obliku klastera za visoku dostupnost i propusnost
- „Enterprise & Cloud Ready“ – lagan i jednostavan za razmještaj u javnim i privatnim oblacima. Sadrži ugrađenu autentikaciju i autorizaciju te ima podršku za TLS i LDAP
- Alati i dodaci – niz alata i dodataka za kontinuiranu integraciju, operativne metrike i integraciju s drugim poslovnim sustavima
- Upravljanje i nadzor – sadrži HTTP-API, alat naredbenog retka i korisničko sučelje za upravljanje i nadzor

Ova tehnologija je pogodna za asinkronu komunikaciju između mikroservisa, jer je neovisna o platformi i sadrži podršku za brojne tehnologije. U razvijenom sustavu, korištena je za asinkrono komuniciranje mikroservisa. Razmještena je u Docker-u, a sama asinkrona razmjena poruka pomoću AMQP protokola je razvijena koristeći RabbitMQ biblioteke u programskim jezicima, u kojima su ujedno razvijeni i mikroservisi: Node.js i Java Spring Boot, što ukazuje na njezinu, spomenutu neovisnost o platformi.

6.2.4. MongoDB

MongoDB je NoSQL baza podataka bazirana na dokumentima s visokom razinom skalabilnosti i fleksibilnosti [35]. Podaci su pohranjeni u obliku fleksibilnih JSON dokumenata, što znači da se atributi mogu razlikovati od dokumenta do dokumenta i struktura podataka se može mijenjati tijekom vremena [35]. Dakle, jedna kolekcija JSON dokumenata (tablica u SQL kontekstu) može sadržavati JSON dokumente s različitim atributima. Takav model dokumenata se vrlo lagano preslikava u aplikacijske objekte, čime je olakšan rad s podacima u aplikacijama i time je razvojnim inženjerima omogućeno lakše učenje i korištenje samog MongoDB-a [35]. „Ad hoc“ upiti, indeksiranje i agregacija u stvarnom vremenu pružaju moćne načine za pristup i analizu podataka [35]. Za izvornu provjeru valjanosti JSON dokumenata i istraživanje shema istih koristi se alat MongoDBCompass [35]. Navedeni alat pruža vizualno korisničko sučelje za upravljanje MongoDB bazom podataka. MongoDB je besplatan za korištenje i pruža upravljačke programe za više od deset programskih jezika [35]. Mikroservis upravljanje korisnicima koristi ovu vrstu baze podataka, pošto je ista vrlo pogodna za korištenje u paru s Node.js-om u kojem je isti razvijen.

6.2.5. MySQL

MySQL je svjetski najpoznatija relacijska baza podataka otvorenog koda [36]. Jedna je od najpopularnijih baza podataka za razvojne inženjere, zbog njezinih visokih performansi, pouzdanosti i jednostavnosti korištenja [36]. Sadrži upravljačke programe za brojne popularne programske jezike: PHP, Java, C#/.Net, Python itd. [36]. Postoje mnogi vizualni alati za upravljanje i dizajn MySQL bazom podataka poput: MySQL Workbench, HeidiSQL, phpMyAdmin, dbForge Studio for MySQL i brojni drugi. Prema literaturi [36], ključne pogodnosti MySQL-a su:

- Jednostavnost korištenja – instalacija MySQL-a u svega nekoliko minuta i lako upravljanje bazom podataka
- Pouzdanost – jedna od najzrelijih i najčešće korištenih baza podataka, testirana u širokom rasponu scenarija više od 25 godina, uključujući mnoge od najvećih svjetskih organizacija
- Skalabilnost – omogućuje skaliranje za ispunjenje zahtjeva najviše pristupačnih aplikacija, zahvaljujući svojoj izvornoj replikacijskoj arhitekturi
- Performanse – dobro pozicionirana u odnosu na ostale baze podataka u kontekstu performansi, prema mnogim „benchmark-ima“
- Visoka dostupnost – postoji kompletan skup izvornih, potpuno integriranih tehnologija replikacije za visoku dostupnost i oporavak od katastrofe

- Sigurnost – u svojoj plaćenoj verziji nudi napredne sigurnosne značajke: autentikacija/autorizacija, transparentno šifriranje podataka, nadzor, maskiranje podataka i vatroštit baze podataka
- Fleksibilnost – podrška za SQL i NoSQL

Iako sadrži podršku za SQL i NoSQL, uglavnom se koristi za izgradnju relacijskih baza podataka u kojima su svi podaci strukturirani i povezani relacijama te je za iste namijenjen SQL. Mikroservis nabave, mikroservis upravljanje artiklima i mikroservis prodaje koriste MySQL bazu podataka i svaki od njih sadrži svoju bazu podataka te vrste. Razlog korištenja je tome, da poslovni podaci mikroservisa su dobro strukturirani i slijede fiksne sheme.

6.2.6. React.js

React.js je JavaScript biblioteka za izgradnju korisničkih sučelja [38]. Omogućeno je dizajniranje jednostavnih prikaza (eng. *views*) za svako stanje aplikacije i React će efikasno ažurirati samo potrebne komponente, kada se promijeni stanje podataka [38]. React je baziran na komponentama, gdje svaka od njih ima svoje stanje i prisutna je kompozicija istih za izgradnju kompleksnijih korisničkih sučelja [38]. Dakle, korisničko sučelje je sastavljeno od više komponenata. Svaka komponenta implementira metodu „render()“, koja uzima ulazne podatke i prikazuje ih zajedno s HTML-om, za što se koristi XML bazirana sintaksa, JSX (eng. *JavaScript XML*) [38]. Budući da svaka komponenta ima svoje stanje, za svaku promjenu podataka, ista će u HTML prikazu ažurirati te podatke, pozivajući ponovo svoju metodu „render()“ [38]. React komponente zapravo primaju podatke i vraćaju ono što bi se trebalo prikazati na korisničkom zaslonu, upravljaju korisničkih interakcijama i ažuriraju prikaze, da odgovaraju novim podacima [38]. Za korištenje React-a potrebno je znanje HTML-a i JavaScript-a, pošto React komponente učahuruju isto preko JSX-a. Stoga arhitektura React-a je bazirana na opisanim komponentama i njihovom životnom ciklusu [38]. U React-u je isto tako zastupljena SPA arhitektura, u kojoj komponente „renderiraju“ sadržaj u tijelu (eng. *body*) jedne zajedničke HTML stranice. React se koristi za programiranje na strani klijenta, a istovremeno se može koristiti i za razvoj mobilnih aplikacija u svojoj React Native verziji [38]. Ova tehnologija je korištena za izradu korisničke strane sustave tj. korisničke aplikacije, koja predstavlja vizualno korištenje sustava.

6.3. Opis sustava

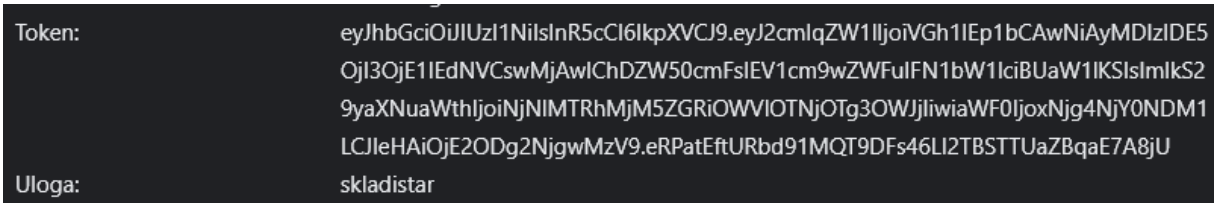
Sustav je *tipa interni B2B* (eng. *Business-to-business*), što se tiče korištenja istoga i kao što je već ranije spomenuto, sastoji se od sastoji od *korisničke* (korisnička web aplikacija) i *poslužiteljske strane* (pozadinski dio sustava temeljen na mikroservisnoj arhitekturi). Komponenta korisničke strane je *korisnička web aplikacija SPA arhitekture*, koja predstavlja *način korištenja sustava*. Komponente poslužiteljske strane su:

- *API Gateway*
- *Mikroservis upravljanje korisnicima*
- *Mikroservis upravljanje artiklima*
- *Mikroservis nabave*
- *Mikroservis prodaje*
- *Servis konfiguracije*

U nastavku će biti opisana svaka zasebno komponenta poslužiteljske strane, a komponenta korisničke strane će biti opisana nešto kasnije unutar prikaza funkcioniranja sustava i načina njegove realizacije. Funkcioniranje sustava kao cjeline na generalnoj razini će se opisati kod prikaza i opisa arhitekture sustava.

6.3.1. API Gateway

API Gateway predstavlja API pristupnik odnosno jedinstvenu točku pristupa korisničke web aplikacije prema svim mikroservisima sustava. Svi zahtjevi iz korisničke web aplikacije se šalju na API Gateway, koji provodi *autorizaciju do mikroservisa* na temelju sljedećih podataka u zaglavlju HTTP zahtjeva: *JWT i uloga korisnika*, a isti su vidljivi na slici 23.



Slika 23. Prikaz podataka za autorizaciju mikroservisa [autorski rad].

Kod žetona (eng. *token*) se provjerava je li još aktivan tj. da li nije istekao, pošto isti ima ograničeno vrijeme trajanja (vrijeme trajanja istoga postavljeno u konfiguraciji *mikroservisa upravljanje korisnicima*, koji generira isti nakon prijave korisnika). Osim žetona uloga korisnika također sudjeluje prilikom autorizacije mikroservisa pa je tako omogućen pristup prijavljenim

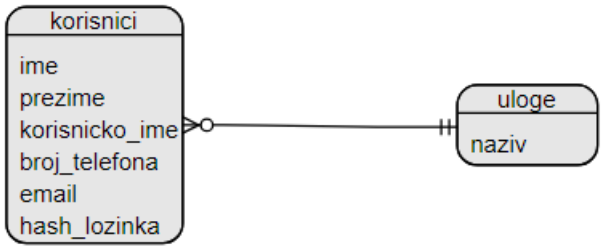
korisnicima (imaju aktivan žeton) s određenom ulogom do određenih mikroservisa. *Pravila autorizacije API Gatewaya* su sljedeća:

- Za određene zahtjeve prema mikroservisu upravljanje korisnicima: aktivan žeton i dopuštene uloge korisnika: administrator
- Za većinu zahtjeva prema mikroservisu upravljanje artiklima: aktivan žeton i dopuštene uloge korisnika: administrator i skladištar
- Za većinu zahtjeva prema mikroservisu nabave: aktivan žeton i dopuštene uloge korisnika: administrator i referent nabave
- Za većinu zahtjeva prema mikroservisu prodaje: aktivan žeton i dopuštene uloge korisnika: administrator i referent prodaje

Kao što se očituje iz pravila autorizacije, korisnik s ulogom administratora ima pristup do svih mikroservisa. U slučaju uspješno provedene autorizacije API Gateway preusmjerava zahtjev iz korisničke web aplikacije prema potrebnom mikroservisu. Međutim prije same provedbe autorizacije, provodi se sljedeće: *pozivanje putanje mikroservisa „/health“* (svaki mikroservis ima navedenu putanju), *kako bi se saznalo „zdravlje“ mikroservisa* (ugašenost istoga ili nedostupnost zbog nedostupnosti infrastrukturnih usluga poput: baze podataka, posrednika poruka). Naime nema smisla provoditi autorizaciju za mikroservis, ukoliko je isti nedostupan ili ugašen. Stoga API Gateway prvo provodi provjeru „zdravlja“ mikroservisa i u slučaju dostupnosti mikroservisa, slijedi provedba opisane autorizacije.

6.3.2. Mikroservis upravljanje korisnicima

Mikroservis upravljanje korisnicima je zadužen za upravljanje podacima korisnika i njihovih uloga. U aplikaciji postoje korisnici sa sljedećim *poslovnih ulogama*: *administrator*, *skladištar*, *referent nabave* i *referent prodaje*. Svaki korisnik ovisno o ulozi ima pristup određenim funkcionalnostima sustava. Spomenuti podaci korisnika i njihovih uloga se spremaju u MongoDB bazi podataka u obliku JSON dokumenata (NoSQL), koju koristi ovaj mikroservis. Na slici 24. je prikazan model podataka baze podataka mikroservisa. Kao što se vidi jedan korisnik može imati samo jednu poslovnu ulogu, dok jednu poslovnu ulogu može imati više korisnika. Za korisnika se spremaju određeni podaci, a s obzirom da ovdje ne postoji primarni ključ, kao nepromjenjivi identifikator korisnika se koristi njegov email. Dakle, podaci pojedinog korisnika se vraćaju s obzirom na njegov email.



Slika 24. Model podataka mikroservisa upravljanje korisnicima [autorski rad].

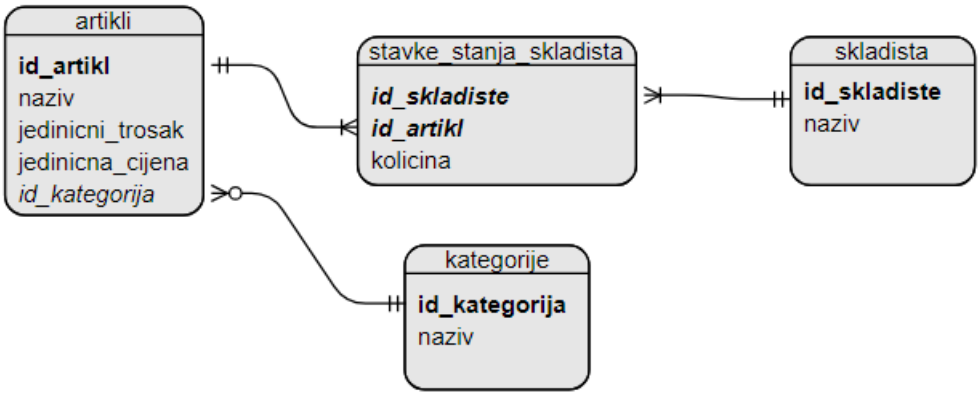
Ovaj mikroservis je zadužen za sljedeće funkcionalnosti sustava:

- *Prijava* (nakon prijave u slučaju uspješne autentikacije korisnika, mikroservis generira JWT i šalje isti zajedno s ulogom korisnika)
- *Ponovno postavljanje lozinke* (ukoliko je korisnik zaboravio lozinku, može prije prijave postaviti novu lozinku preko dobivene hiperveze za tu svrhu u generiranoj email poruci)
- *Registracija* (registracija korisnika uz definiranje osnovnih podataka i odabir poslovne uloge, bez mogućnosti registriranja kao administrator)
- *Odjava*
- *Dohvaćanje potrebnih podataka korisnika i njihovih uloga*
- *Ažuriranje računa korisnika* (dopuštena promjena osnovnih podataka korisnika, osim emaila i poslovne uloge)
- *Promjena lozinke*
- *„Impersonate“* (prijava kao bilo koji drugi korisnik aplikacije, bez potrebe za autentikacijom, što je dostupno samo korisnicima s ulogom administratora)

Mikroservis je razvijen u Node.js tehnologiji i koristi kao što je opisano svoju MongoDB bazu podataka. Isti *asinkrono komunicira s mikroservisom nabave i mikroservisom prodaje*, slanjem odgovarajućih podataka korisnika istima (oni sadrže *lokalne kopije podataka korisnika*). Vlastito razvijen, lokalni web poslužitelj Node.js-a je mjesto razmještaja ovog mikroservisa.

6.3.3. Mikroservis upravljanje artiklima

Mikroservis upravljanje artiklima služi za evidentiranje i upravljanje artiklima i njihovim stanjima na skladištima. Svaki artikl pripada određenoj kategoriji artikla i ima svoja stanja na skladištima. Podaci o artiklima, kategorijama artikla, skladištima i stanjima artikala na skladištu se spremaju u obliku relacijske baze podataka MySQL. Na slici 25. je prikazan ERA model mikroservisa upravljanje artiklima. Svaki artikl pripada točno jednoj kategoriji, dok jedna kategorija može imati više artikala. Artikl se može nalaziti na stanjima više skladišta, dok jedno skladište može bilježiti stanje više artikala. Budući da se radi o izmišljenom trgovačkom poduzeću, za svaki artikl se spremaju podaci o jediničnom trošku i jediničnoj cijeni, potrebnima kod nabave artikala od dobavljača i prodaje artikala kupcima, kao i kod ostalih trgovačkih procesa i podataka. Ovaj mikroservis pokriva osnovnu ulogu sustava za upravljanje skladištem spomenutog, izmišljenog trgovačkog poduzeća.



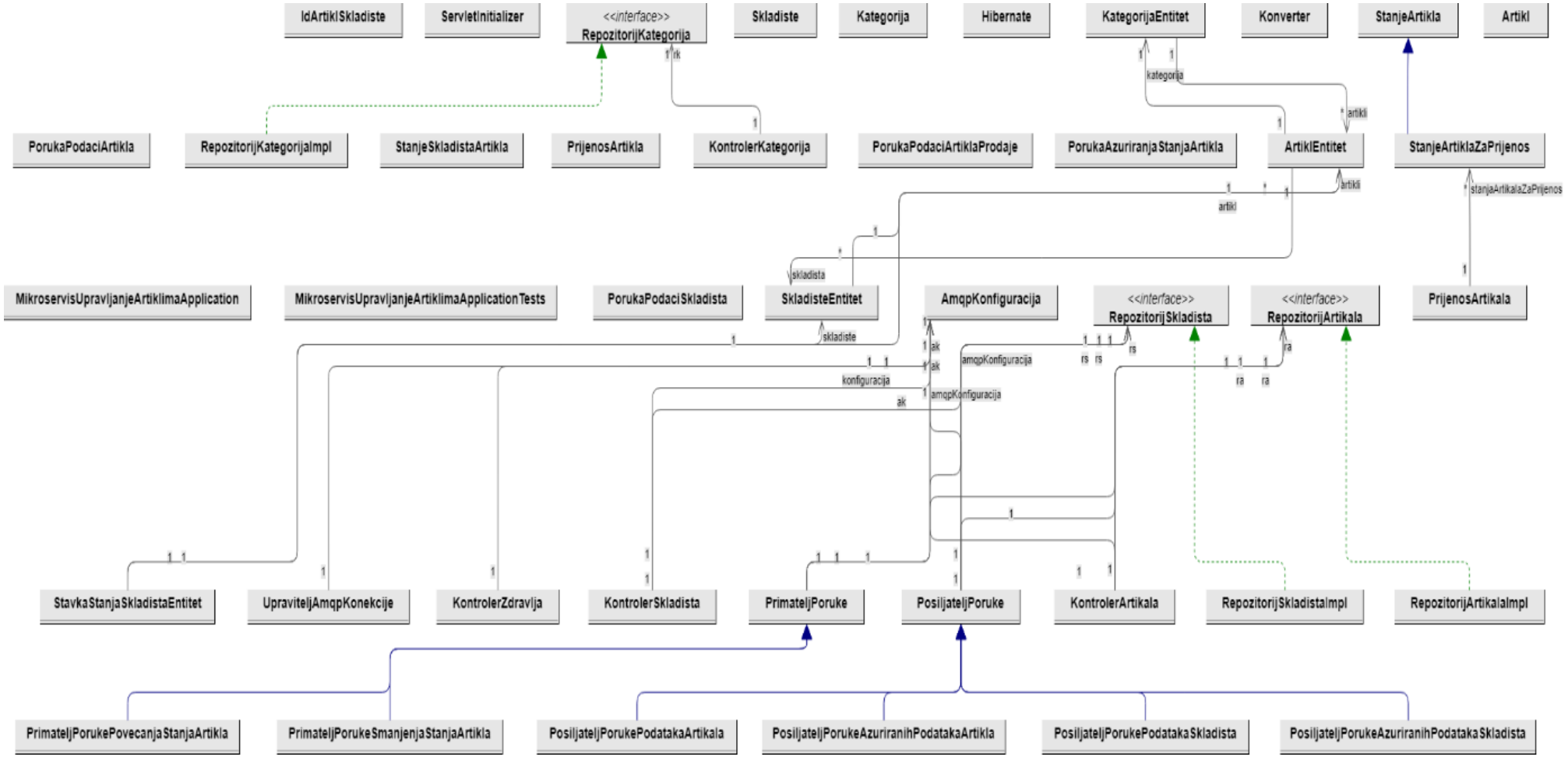
Slika 25. ERA model mikroservisa upravljanje artiklima [autorski rad].

Funkcionalnosti ovog mikroservisa su:

- *Kreiranje, ažuriranje i čitanje podataka o artiklima, kategorijama i skladištima*
- *Vođenje evidencije stanja artikala po pojedinim skladištima*
- *Ažuriranje stanja artikala po pojedinim skladištima*
- *Prijenos artikala između skladišta* (omogućava prijenos više artikala između dva odabrana skladišta)

Mikroservis ažurira stanje artikala na pojedinim skladištima kroz *asinkronu komunikaciju s mikroservisom nabave i mikroservisom prodaje* (nakon obrade narudžbenice ili narudžbe mikroservis nabave ili mikroservis prodaje, šalje podatke o povećanju ili smanjenju stanja artikala na pojedinim skladištima ovom mikroservisu). *Mikroservisi nabave i prodaje isto tako sadrže lokalne kopije podataka artikala i skladišta*, koje održavaju uvijek ažurnim također

asinkronom komunikacijom s ovim mikroservisom. Ovaj mikroservis je razvijen u Java Spring Boot tehnologiji i koristi kao što je opisano svoju MySQL bazu podataka (za optimiziran pristup do iste koristi tehnologiju *Hibernate*). *Konfiguracijske podatke* vezane uz opisanu asinkronu komunikaciju, dobiva od *servisa konfiguracije*, koji je razvijen isto u Java Spring Boot tehnologiji i biti će opisan kasnije. Vlastiti Tomcat web kontejner je mjesto razmještaja ovog mikroservisa. U nastavku je prikazan i dijagram klasa opisanog mikroservisa. Preko naziva klasa se može iščitati uloga pojedine klase.



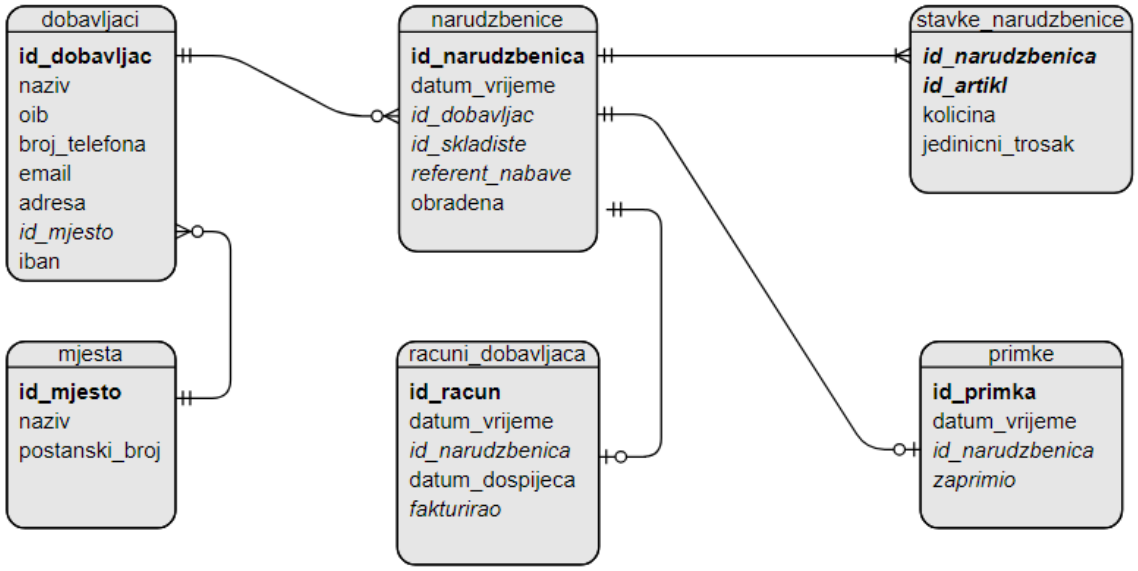
Slika 26. Dijagram klasa mikroservisa upravljanje artiklima [autorski rad].

6.3.4. Mikroservis nabave

Mikroservis nabave služi za *upravljanje svim trgovačkim procesima vezanima uz nabavu*. Isti upravlja podacima o dobavljačima, narudžbenicama i pripadnim primkama i računima dobavljača. Navedeni podaci se spremaju u obliku relacijske baze podataka MySQL. Slika 27. prikazuje ERA model mikroservisa nabave. Kao što je navedeno, spremaju se potrebni podaci o dobavljačima uz relaciju prema mjestima (poštanski broj mjesta je jedinstveni). Narudžbenica ima sljedeće podatke:

- id_narudzbenica (jedinstveni identifikator narudžbenice)
- datum_vrijeme (vrijeme kada je narudžbenica generirana)
- id_dobavljac (relacija na tablicu dobavljača, svaka narudžbenica mora sadržavati dobavljača od kojeg se naručuju artikli)
- id_skladiste (relacija na skladište na koje će naručeni artikli biti zaprimljeni, a *podaci skladišta* se dobivaju kroz *asinkronu komunikaciju s mikroservisom upravljanje artiklima*)
- referent_nabave (relacija na korisnika koji je izvršio kreiranje narudžbenice, a *podaci korisnika* se dobivaju kroz *asinkronu komunikaciju s mikroservisom upravljanje korisnicima*)
- obradena (binarni podatak koji ukazuje je li narudžbenica obrađena (ima izvršene procese fakturiranja i zaprimanja))

Podaci narudžbenice su i stavke narudžbenice u kojoj su pohranjeni podaci o naručenim artiklima i njihovim količinama uz bilježenje podatka o jediničnom trošku svakog artikla. *Podaci vezani uz artikl: id_artikl i jedinicni_trosak* se dohvaćaju kroz *asinkronu komunikaciju s mikroservisom upravljanje artiklima* (jedinični trošak artikla se može mijenjati kroz vrijeme pa je važno u trenutku naručivanja zabilježiti isti, za svaku stavku narudžbenice). Podaci primki i računa dobavljača se generiraju nakon procesa obrade narudžbenice (nakon fakturiranja za račun dobavljača i nakon zaprimanja za primku). Za primku se sprema vrijeme kada je generirana, relacija na narudžbenicu i relacija na korisnika, koji je izvršio generiranje primke (podaci korisnika se također dobivaju kroz *asinkronu komunikaciju s mikroservisom upravljanje korisnicima*). Za račun dobavljača se spremaju isti podaci kao kod primke, uz dodatno evidentiranje datuma dospijeća do kada je račun potrebno platiti dobavljaču (u konfiguracijskim podacima mikroservisa, postoji podatak o broju dana dospijeća, za bilo koji generirani račun).

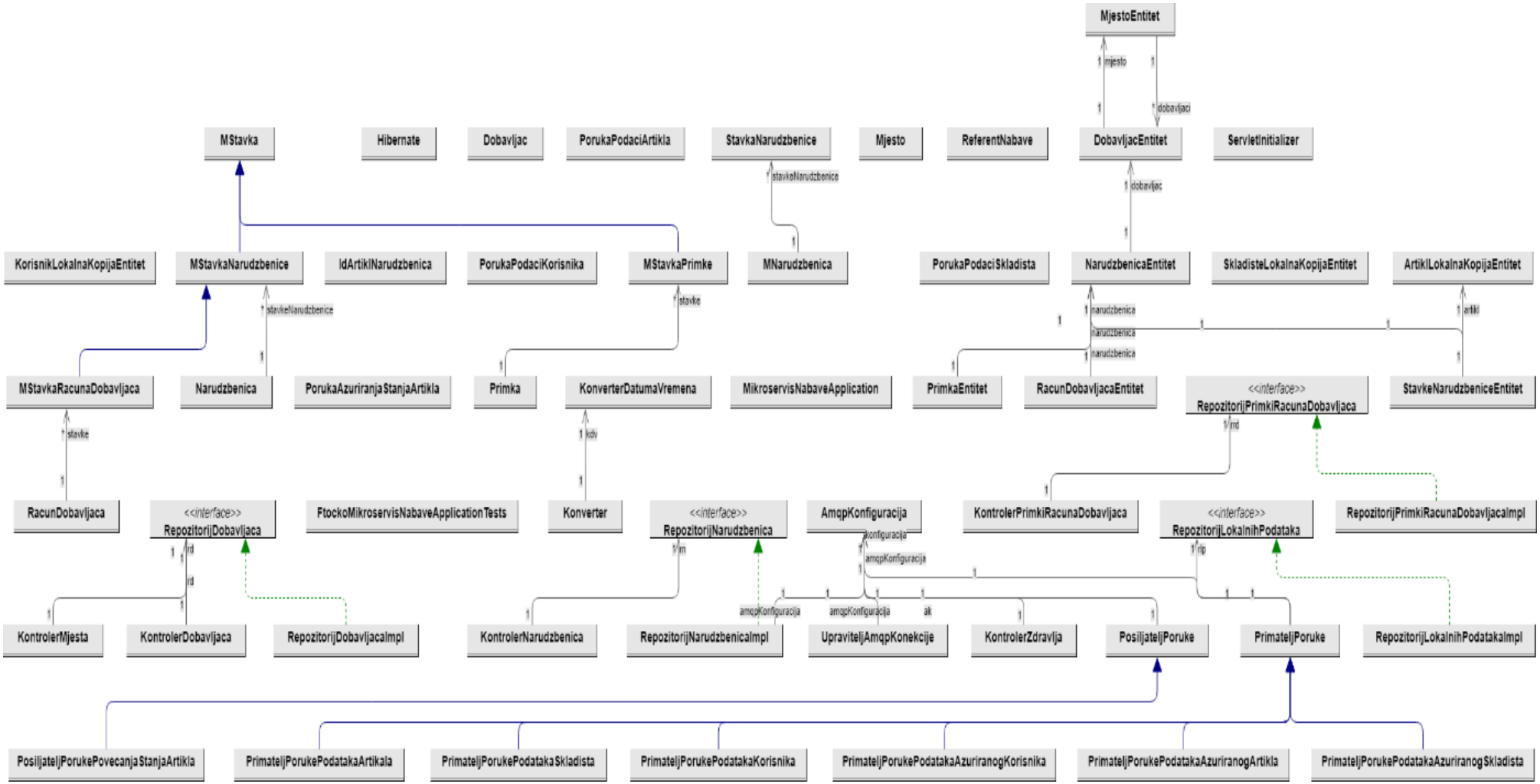


Slika 27. ERA model mikroservisa nabave [autorski rad].

Funkcionalnosti ovog mikroservisa su:

- *Kreiranje, ažuriranje i čitanje podataka o dobavljačima*
- *Kreiranje novih narudžbenica*
- *Obrada (izvršenje procesa fakturiranja i zaprimanja) i otkazivanje narudžbenica*
- *Upravljanje podacima o primkama i računima dobavljača*

Nakon *kreiranja nove narudžbenice*, ista se može *obraditi* ili *otkazati*. Mikroservis kod *obrade narudžbenice*: *kreira novi račun dobavljača* (odgovara procesu fakturiranja) i *primku* (odgovara procesu zaprimanja), ažurira status podatka narudžbenice, *„obradena“ na 1* (narudžbenica je obrađena) i *asinkrono šalje podatke o povećanju stanja artikala na skladištu mikroservisu upravljanje artiklima*, kako bi isti mogao evidentirati povećanje zaprimljenih artikala na skladištu. U slučaju *otkazivanja narudžbenice,* ista se potpuno briše iz baze podataka mikroservisa. Ovaj mikroservis je razvijen u Java Spring Boot tehnologiji i koristi kao što je opisano svoju MySQL bazu podataka (za optimiziran pristup do iste koristi tehnologiju *Hibernate*). *Konfiguracijske podatke* vezane uz opisanu asinkronu komunikaciju, dobiva od *servisa konfiguracije*. Vlastiti Tomcat web kontejner je mjesto razmještaja ovog mikroservisa. U nastavku je prikazan i dijagram klasa opisanog mikroservisa. Preko naziva klasa se može iščitati uloga pojedine klase.



Slika 28. Dijagram klasa mikroservisa nabave [autorski rad].

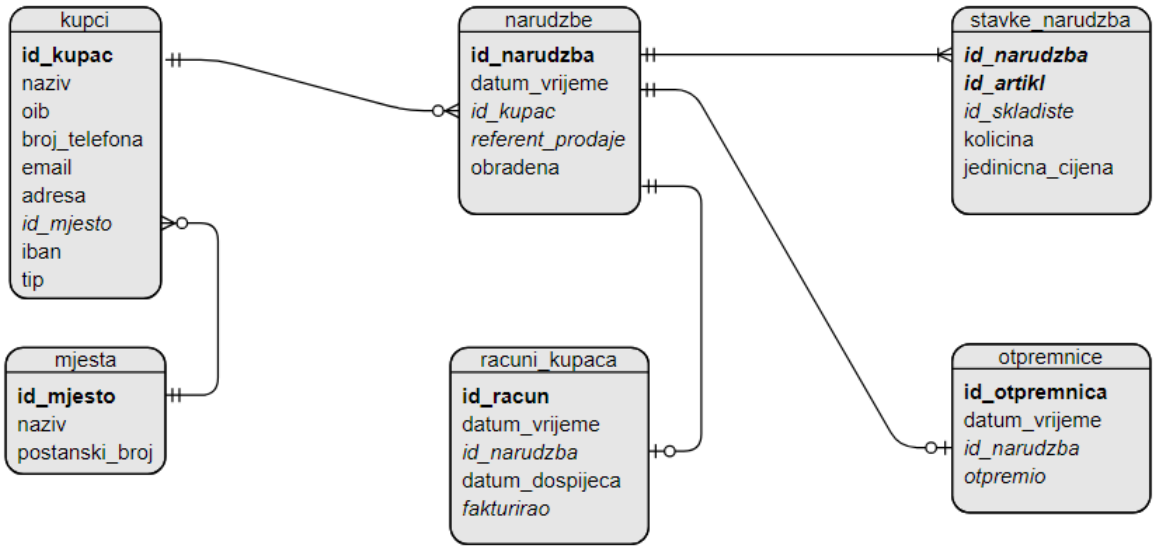
6.3.5. Mikroservis prodaje

Mikroservis prodaje služi za *upravljanje svim trgovačkim procesima vezanima uz prodaju*. Isti upravlja podacima o kupcima, narudžbama i pripadnim otpremnicama i računima kupaca. Navedeni podaci se spremaju u obliku relacijske baze podataka MySQL. Slika 29. prikazuje ERA model mikroservisa prodaje. Kao što je navedeno, spremaju se potrebni podaci o kupcima uz relaciju prema mjestima. Postoje dva tipa kupaca (binarni podatak „tip“ u tablici kupaca): poslovni (tip ima vrijednost 1) i privatni (tip ima vrijednost 0). *Poslovni kupci imaju određeni popust kod narudžbi*, koji *generira i uračunava mikroservis nakon kreiranja narudžbe* (zadani iznos popusta (u obliku postotka) za narudžbu je definiran u konfiguracijskim podacima mikroservisa). Narudžba ima sljedeće podatke:

- id_narudzba (jedinstveni identifikator narudžbe)
- datum_vrijeme (vrijeme kada je narudžba generirana)
- id_kupac (relacija na tablicu kupaca, svaka narudžba mora sadržavati kupca, koji izvršava kupnju artikala)
- referent_prodaje (relacija na korisnika koji je izvršio kreiranje narudžbe, a *podaci korisnika* se dobivaju kroz *asinkronu komunikaciju s mikroservisom upravljanje korisnicima*)
- obradena (binarni podatak koji ukazuje je li narudžba obrađena (ima izvršene procese fakturiranja i otpremanja))

Podaci narudžbe su i stavke narudžbe u kojoj su pohranjeni podaci o kupljenim artiklima i njihovim količinama uz bilježenje podatka o jediničnoj cijeni svakog artikla. *Podaci vezani uz artikl: id_artikl i jedinicna_cijena* se dohvaćaju kroz *asinkronu komunikaciju s mikroservisom upravljanje artiklima* (jedinična cijena artikla se može mijenjati kroz vrijeme pa je važno u trenutku narudžbe odnosno kupnje zabilježiti istu, za svaku stavku narudžbe). Isto tako *za svaku stavku narudžbe se bilježi skladište* (*podaci skladišta* se dobivaju kroz *asinkronu komunikaciju s mikroservisom upravljanje artiklima*) *s kojeg se uzima pripadna količina artikla*. Podaci otpremnica i računa kupaca se generiraju nakon procesa obrade narudžbe (nakon fakturiranja za račun kupca i nakon otpremanja za otpremnicu). Za otpremnicu se sprema vrijeme kada je generirana, relacija na narudžbu i relacija na korisnika, koji je izvršio generiranje otpremnice (podaci korisnika se također dobivaju kroz *asinkronu komunikaciju s mikroservisom upravljanje korisnicima*). Za račun kupca se spremaju isti podaci kao kod otpremnice, uz dodatno evidentiranje datuma dospijeća do kada kupac mora platiti račun (u

konfiguracijskim podacima mikroservisa, postoji podatak o broju dana dospijeća, za bilo koji generirani račun).

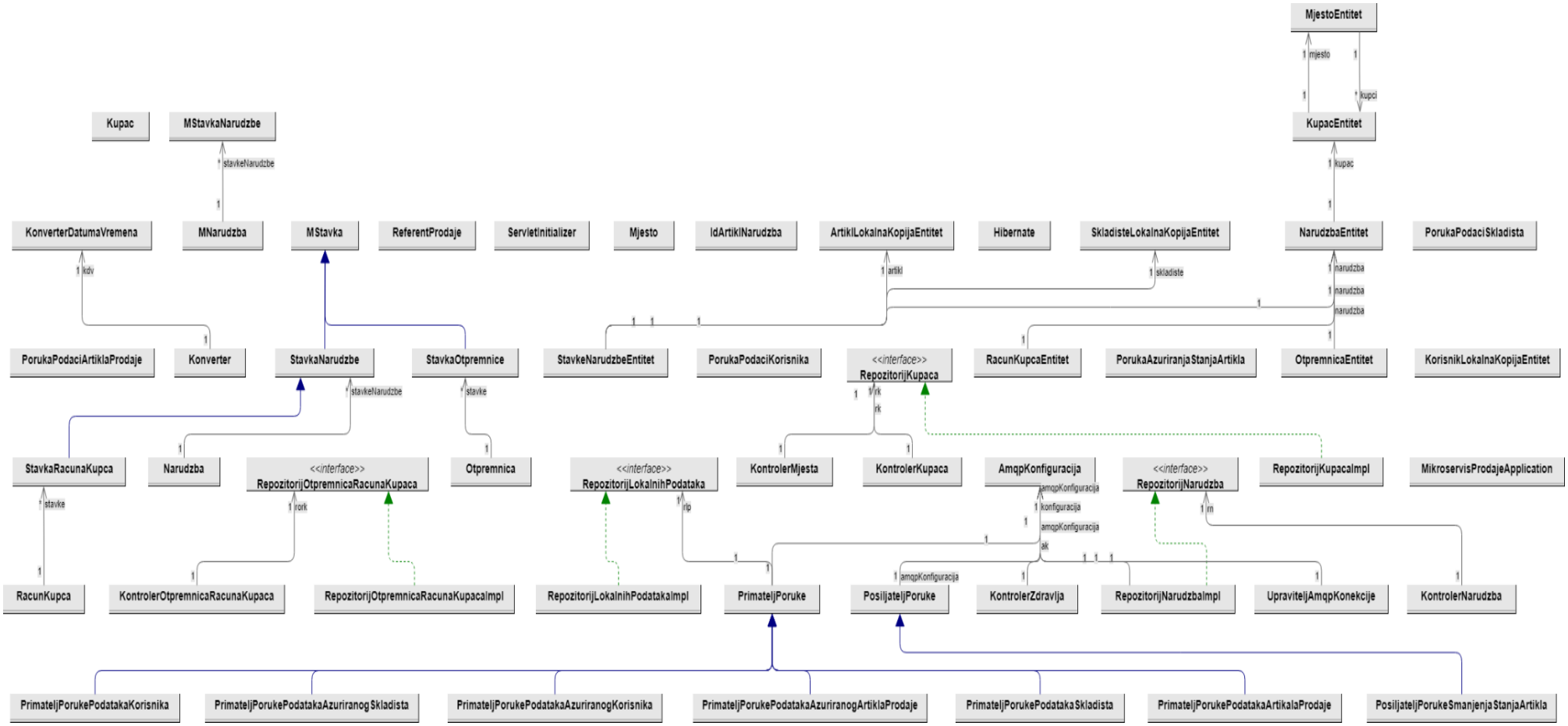


Slika 29. ERA model mikroservisa prodaje [autorski rad].

Funkcionalnosti ovog mikroservisa su:

- *Kreiranje, ažuriranje i čitanje podataka o kupcima*
- *Kreiranje novih narudžbi*
- *Obrada (izvršenje procesa fakturiranja i otpremanja) i otkazivanje narudžbi*
- *Upravljanje podacima o otpremnicama i računima kupaca*

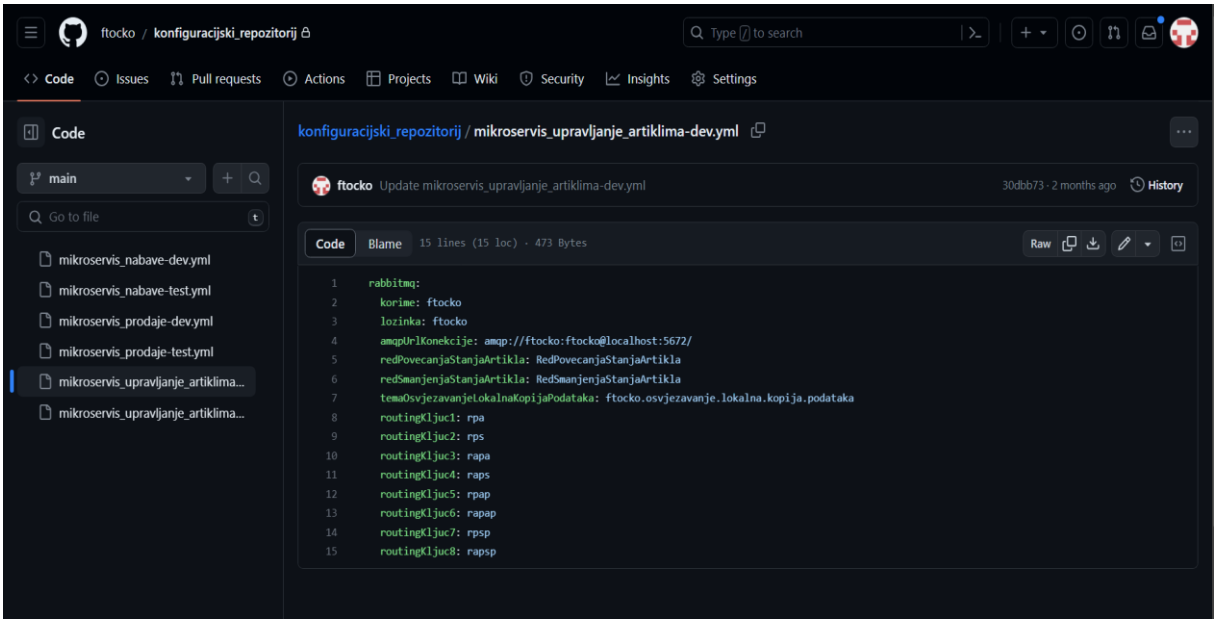
Nakon *kreiranja nove narudžbe*, ista se može *obraditi* ili *otkazati*. Mikroservis kod *obrade narudžbe*: *kreira novi račun kupca* (odgovara procesu fakturiranja) i *otpremnicu* (odgovara procesu otpremanja), ažurira status podatka narudžbe, *„obradena“ na 1* (narudžba je obrađena) i *asinkrono šalje podatke o smanjenju stanja artikala na skladištima mikroservisu upravljanje artiklima*, kako bi isti mogao evidentirati smanjenje otpremljenih artikala na potrebnim skladištima. U slučaju *otkazivanja narudžbe,* ista se potpuno briše iz baze podataka mikroservisa. Ovaj mikroservis je razvijen u Java Spring Boot tehnologiji i koristi kao što je opisano svoju MySQL bazu podataka (za optimiziran pristup do iste koristi tehnologiju *Hibernate*). *Konfiguracijske podatke* vezane uz opisanu asinkronu komunikaciju, dobiva od *servisa konfiguracije*. Vlastiti Tomcat web kontejner je mjesto razmještaja ovog mikroservisa. U nastavku je prikazan i dijagram klasa opisanog mikroservisa. Preko naziva klasa se može iščitati uloga pojedine klase.



Slika 30. Dijagram klasa mikroservisa prodaje [autorski rad].

6.3.6. Servis konfiguracije

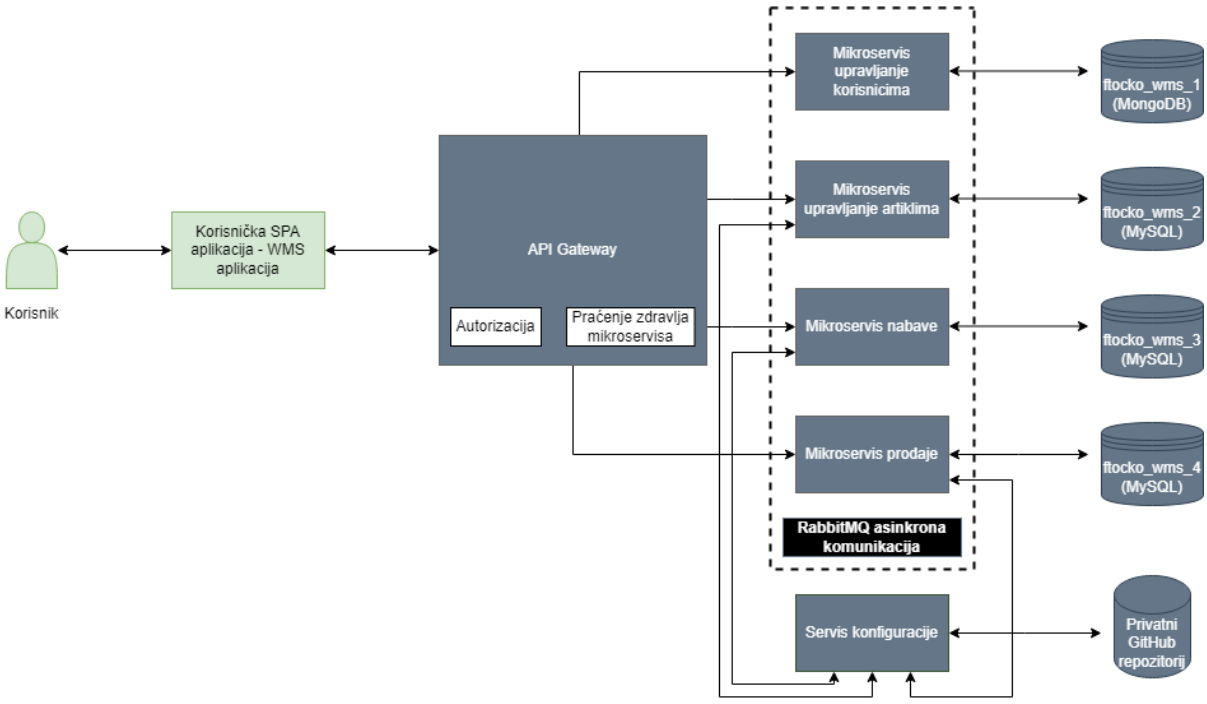
Servis konfiguracije služi za *eksternalizaciju konfiguracijskih podataka mikroservisa upravljanje artiklima, mikroservisa nabave i mikroservisa prodaje*. Konfiguracijski podaci su vezani uz *asinkronu komunikaciju mikroservisa* (podaci za vezu, nazivi tema, uzorci (ključevi) usmjeravanja, nazivi redova poruka itd.). Isti su spremljeni za svaki mikroservis u *privatnom GitHub repozitoriju*. Ovaj servis dohvaća iste iz tog mjesta i prenosi do svakog spomenutog mikroservisa. Na slici 31. je prikazan primjer konfiguracijskih podataka za mikroservis upravljanje artiklima, koji su spremljeni u privatnom GitHub repozitoriju za konfiguraciju. Kao što je opisano, servis konfiguracije se spaja na taj repozitorij, dohvaća konfiguracijske podatke pojedinog mikroservisa i šalje ih prema mikroservisu, koji je s njim trenutno povezan. Servis je razvijen u Java Spring Boot tehnologiji, kao i mikroservisi kojima pruža eksterne konfiguracijske podatke, što predstavlja razlog korištenja istoga.



Slika 31. Prikaz konfiguracijskih podataka mikroservisa u privatnom GitHub repozitoriju [autorski rad].

6.4. Arhitektura sustava

Na generalnoj razini, funkcioniranje sustava kao cjeline je ilustrirano na idućoj slici 32. *Zelenom bojom* su označene *komponente korisničke strane sustava*, a *sivom bojom komponente poslužiteljske strane sustava*. Kao što je vidljivo, korisnik i korisnička web aplikacija čine korisničku stranu sustava. Korisnik s određenom poslovnom ulogom koristi korisničku web aplikaciju. Sve akcije korisnika iz korisničke web aplikacije se šalju u obliku HTTP zahtjeva na API Gateway, koji provodi prvo provjeru zdravlja, zatim autorizaciju potrebnog mikroservisa i ukoliko su navedene radnje uspješno provedene, zahtjev se preusmjerava do tog mikroservisa. Svaki mikroservis je zadužen za svoju poslovnu pod domenu i ima pripadnu bazu podataka iste. Mikroservisi međusobno asinkrono komuniciraju pomoću AMQP protokola preko posrednika poruka, RabbitMQ. Mikroservis upravljanje artiklima, mikroservis nabave i mikroservis prodaje su povezani sa servisom konfiguracije, koji iz privatnog GitHub repozitorija istima dohvaća njihove eksterne konfiguracijske podatke, vezane uz asinkronu komunikaciju. Stoga poslužiteljsku stranu sustava čine: API Gateway, mikroservisi i njihove pripadne baze podataka, posrednik poruka za asinkronu komunikaciju mikroservisa, RabbitMQ i servis konfiguracije.



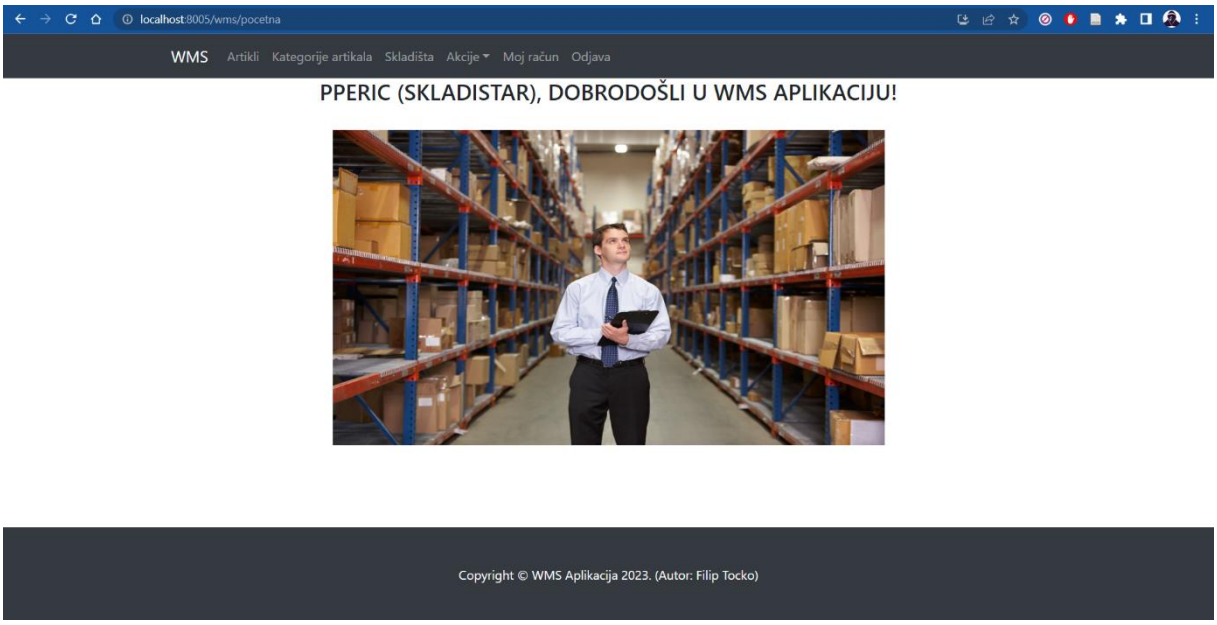
Slika 32. Prikaz arhitekture sustava [autorski rad].

6.5. Prikaz funkcioniranja sustava i načina njegove realizacije

Korisnička web aplikacija predstavlja *način korištenja sustava* i ujedno *korisničku stranu sustava*. Ona indirektno komunicira s poslužiteljskom stranom sustava odnosno mikroservisima sustava preko API Gatewaya. Sva poslovna logika se provodi na poslužiteljskoj strani, a korisnička strana tj. korisnička web aplikacija prati istu. Postoje sljedeće uloge korisnika u istoj: *administrator*, *skladištar*, *referent nabave* i *referent prodaje*. Svaka uloga korisnika ima pristup do određenih funkcionalnosti i predstavlja registriranog korisnika (aplikaciju mogu koristiti samo registrirani korisnici). Ono što je zajedničko svim korisnicima bez obzira na ulogu su sljedeće funkcionalnosti: *registracija*, *prijava*, *ponovno postavljanje lozinke*, *ažuriranje računa* i *promjena lozinke* te *odjava*. *Funkcionalnosti korisnika s ulogom skladištara* su:

- *Pregled, kreiranje i ažuriranje skladišta uz mogućnost uvida u stanja artikala pojedinog skladišta*
- *Pregled, kreiranje i ažuriranje artikala uz mogućnost uvida u stanja po skladištima za svaki artikl*
- *Pregled, kreiranje i ažuriranje kategorija artikala uz mogućnost uvida artikala pojedine kategorije artikla*
- *Prijenos artikala između dva skladišta*

Navedene funkcionalnosti skladištara odgovaraju *mikroservisu upravljanje artiklima* (za autoriziran pristup istome, aplikacija kod svakog HTTP zahtjeva prema API Gatewayu, mora u zaglavlju zahtjeva poslati sljedeće podatke: *aktivni žeton i podatak uloge korisnika: administrator* ili *skladištar*). Slika 33. prikazuje izgled početnog zaslona korisnika s ulogom skladištara, nakon procesa prijave.

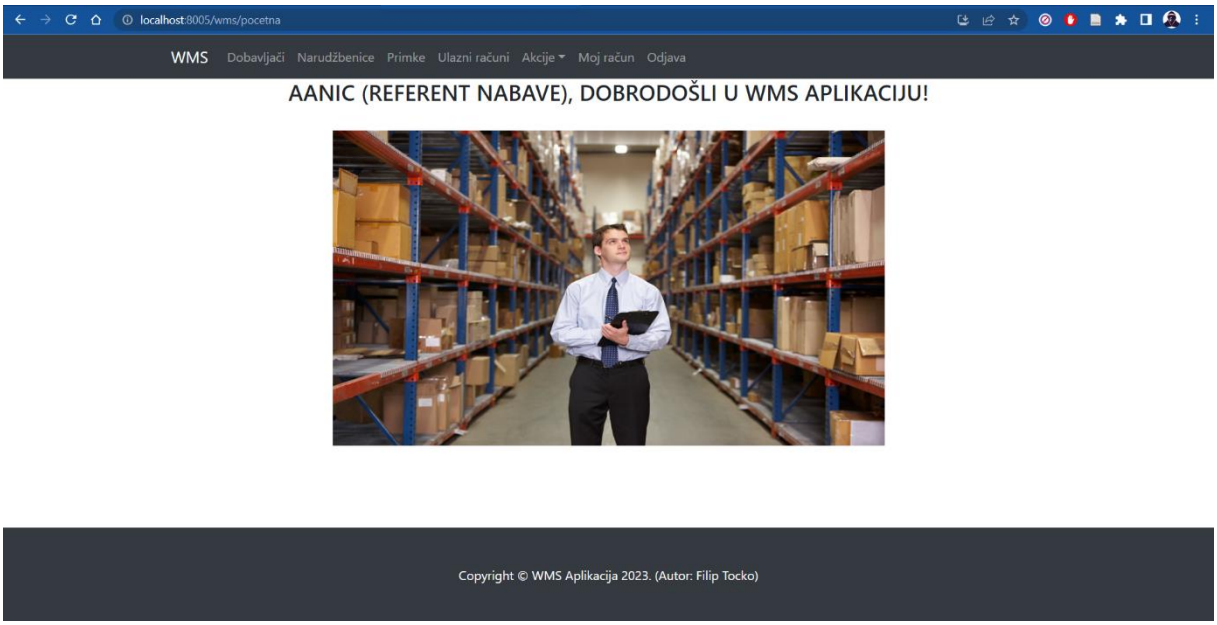


Slika 33. Prikaz početnog zaslona aplikacije skladištara [autorski rad].

Funkcionalnosti korisnika s ulogom referenta nabave su:

- *Pregled, kreiranje i ažuriranje dobavljača*
- *Pregled neobrađenih narudžbenica uz mogućnost kreiranje nove narudžbenice, obrade i otkazivanja pojedine narudžbenice*
- *Pregled primki i računa dobavljača*

Navedene funkcionalnosti referenta nabave odgovaraju *mikroservisu nabave* (za autoriziran pristup istome, aplikacija kod svakog HTTP zahtjeva prema API Gatewayu, mora u zaglavlju zahtjeva poslati sljedeće podatke: *aktivni žeton i podatak uloge korisnika: administrator* ili *referent nabave*). Slika 34. prikazuje izgled početnog zaslona korisnika s ulogom referenta nabave, nakon procesa prijave.

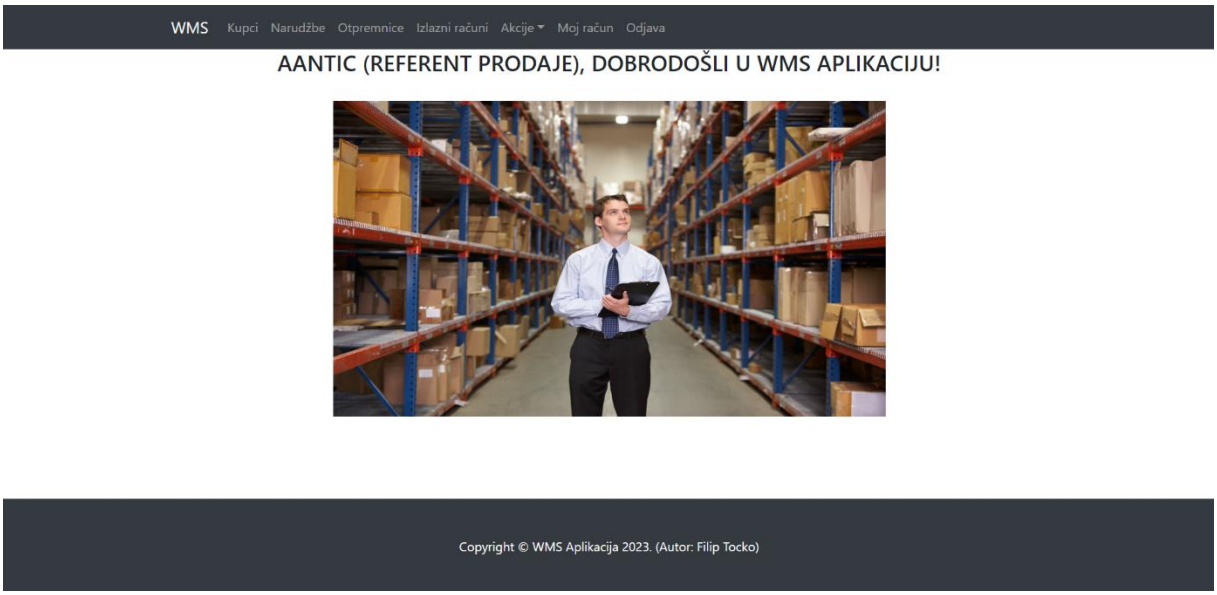


Slika 34. Prikaz početnog zaslona aplikacije referenta nabave [autorski rad].

Funkcionalnosti korisnika s ulogom referenta prodaje su:

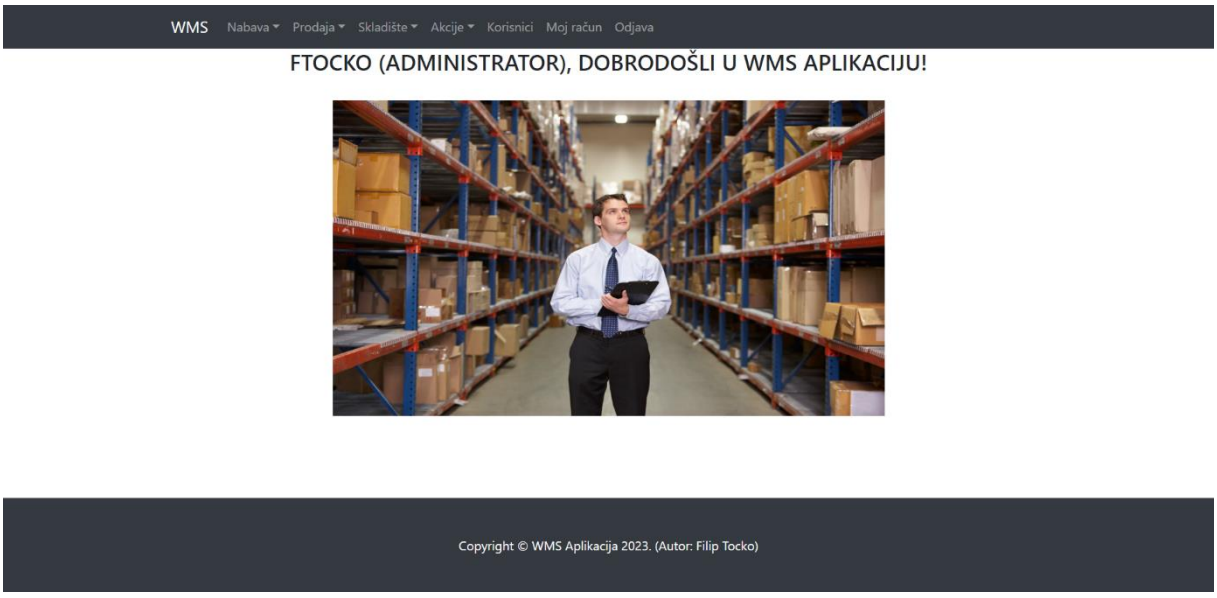
- *Pregled, kreiranje i ažuriranje kupaca*
- *Pregled neobrađenih narudžbi uz mogućnost kreiranje nove narudžbe, obrade i otkazivanja pojedine narudžbe*
- *Pregled otpremnica i računa kupaca*

Navedene funkcionalnosti referenta prodaje odgovaraju *mikroservisu prodaje* (za autoriziran pristup istome, aplikacija kod svakog HTTP zahtjeva prema API Gatewayu, mora u zaglavlju zahtjeva poslati sljedeće podatke: *aktivni žeton i podatak uloge korisnika: administrator* ili *referent prodaje*). Slika 35. prikazuje izgled početnog zaslona korisnika s ulogom referenta prodaje, nakon procesa prijave.



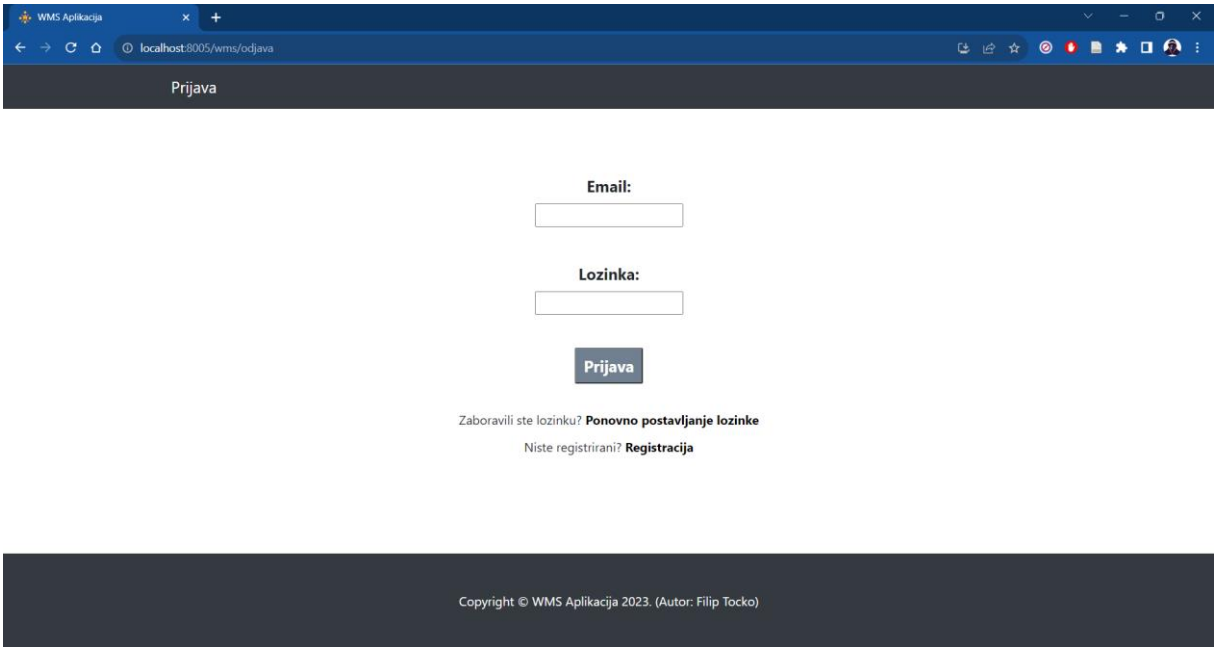
Slika 35. Prikaz početnog zaslona aplikacije referenta prodaje [autorski rad].

Korisnik s ulogom administratora ima pristup do svih funkcionalnosti uloga korisnika uz *mogućnost pregleda korisnika* (tamo ima *mogućnost promjene uloge i „impersonatea“ korisnika*). Slika 36. prikazuje izgled početnog zaslona korisnika s ulogom administratora, nakon procesa prijave. Iz iste se jasno vidi, da su mogućnosti pojedine uloge korisnika za administratora redom podijeljene u globalne izbornike: nabava (referent nabave), prodaja (referent prodaje) i skladište (skladištar).



Slika 36. Prikaz početnog zaslona aplikacije administratora [autorski rad].

Prije početka korištenja sustava, korisnik se mora prijaviti u sustav, unosom svojih identifikacijskih (autentikacijskih) podataka kod zaslona prijave. Korisniku je omogućeno i ponovno postavljanje lozinke, ukoliko je istu zaboravio te registracija, ako još nema kreiran svoj poslovni račun. Nakon prijave u sustav *korisniku je vidljiv izbornik s obzirom na njegovu poslovnu ulogu, autorizacijom na korisničkoj i poslužiteljskoj strani*, kao što je prethodno opisano.



Slika 37. Prikaz zaslona prijave [autorski rad].

Mikroservis upravljanje korisnicima nakon uspješne prijave korisnika, generira JWT i vraća preko API Gatewaya korisničkoj web aplikaciji, sljedeće podatke: generirani žeton, email i uloga korisnika. Navedene podatke ista sprema u svoje lokalno spremište (eng. *local storage*). JavaScript programski kod iz Node.js-a metode kontrolera mikroservisa upravljanje korisnicima je prikazan u nastavku.

```
router.post("/prijava", (req, res) => {

    var email = String(req.body.email)
    var lozinka = String(req.body.lozinka)

    Korisnik.findOne({ email: email }, function (err, korisnik) {
        if (korisnik) {
            bcrypt.compare(lozinka, korisnik.hash_lozinka, function (err,
provjera) {
                if (provjera) {
                    let tokenPodaci = {
                        vrijeme: Date(),
```

```
                        idKorisnika: korisnik._id,
                    }

                    const token = jwt.sign(tokenPodaci, jwtTajniKljuc, {
                        expiresIn: jwtTrajanje
                    })

                    Uloga.findOne({ korisnici: email }, function (err,
uloga) {
                        if (uloga) {

                            let resJson = { korisnik: korisnik.email,
uloga: uloga.naziv, token: token }

                            res.status(200).send(resJson).end()
                        }
                    })
                }
                else {
                    res.status(401).send("Neuspješna autentikacija").end()
                }
            });
        }

        else {
            res.status(401).send("Neuspješna autentikacija!").end()
        }
    })
})
```

Korisnička web aplikacija obavlja autorizaciju na korisničkoj strani za svaku zahtijevanu putanju (eng. *route*), kako bi se osiguralo, da su korisniku dostupni zasloni i izbornici, predviđeni za njegovu poslovnu ulogu. U nastavku je prikazan JavaScript programski kod React.js komponente, za autorizaciju korisnika s ulogom skladištara (ako je isti prijavljen u sustav). Provjeravaju se postojanje i status žetona te korisnička uloga iz lokalnog spremišta aplikacije i na temelju toga se odlučuje hoće li se korisniku generirati zaslon zahtijevane putanje ili će ga se preusmjeriti na početni zaslon.

```
function useAuth() {

    if (localStorage.getItem("token") === null) {
        return false
    }
```

```
    if(localStorage.getItem("uloga") !== "skladistar" &&
localStorage.getItem("uloga") !== "administrator"){
        return false
    }

    const { exp } = jwtDecode(localStorage.getItem("token"))
    const expVrijeme = exp * 1000

    if (Date.now() >= expVrijeme) {

        localStorage.removeItem("korisnik")
        localStorage.removeItem("uloga")
        localStorage.removeItem("token")
        localStorage.clear();
        return false
    }

    return true
}

const ZasticeneSkladistarRute = () => {
    return useAuth()? <Outlet/>:<Navigate to="/wms"/>
}

export default ZasticeneSkladistarRute
```

Za svaku poslovnu ulogu prijavljenog korisnika, postoji prikazana i opisana React.js komponenta. *Učitavanjem zaslona zahtijevanje putanje korisniku i bilo kojom korisničkom akcijom nad istim, šalje se HTTP zahtjev prema API Gatewayu, koji provodi autorizaciju na poslužiteljskoj strani tj. pruža zaštićen pristup do mikroservisa*. Prikazana JavaScript klasa API Gatewaya u Node.js-u, validira korisničku ulogu skladištara, provjeravanjem podataka (žetona i uloge korisnika) u dobivenom HTTP zaglavlju od korisničke web aplikacije.

```
class TokenSkladistarValidator extends TokenValidator{

    validiraj(req){

        var validirano = false;

        try{
            let token = String(req.header("token"));
            validirano = jwt.verify(token, jwtTajniKljuc);
        }
        catch(error){
            validirano = false;
        }
```

```
        if(String(req.header("uloga")) !== "skladistar" &&
String(req.header("uloga")) !== "administrator"){
            validirano = false;
        }
        return validirano;
    }
}

module.exports = TokenSkladistarValidator;
```

U slučaju uspješne validacije, *API Gateway* dopušta pristup do potrebnog mikroservisa i preusmjerava HTTP zahtjev prema istome. Kao što je vidljivo na idućem programskom kodu, za HTTP GET zahtjev (upućen iz korisničke web aplikacije od strane korisnika s ulogom skladištara), za dohvaćanje stanja na svim skladištima traženog ID artikla, API Gateway prvo provjerava dostupnost mikroservisa i u slučaju uspješne dostupnosti istoga izvršava provjeru autorizacije (koristi se prethodno prikazana i opisana JavaScript klasa), nakon čega opet u slučaju uspješnosti iste, preusmjerava zahtjev do mikroservisa upravljanje artiklima.

```
router.get("/artikli/:id/stanjaArtikla", async (req, res) => {

    if (await servisValidator.provjeriDostupnostServisa(adresaMUA)) {

        if (tokenSkladistarValidator.validiraj(req)) {

            axios.get(adresaMUA + "/artikli/" + req.params.id +
"/stanjaArtikla", { headers })
                .then(response => {
                    response.status === 200 ?
res.status(200).send(response.data).end() : res.status(400).send("Bad
request!").end()
                }).catch(error => {
                    res.status(error.response.status).send(error.response.d
ata).end()
                })
        }
        else {
            res.status(401).send("Neuspješna autorizacija!").end()
        }
    }
    else {
        res.status(403).send("Servis odbija zahtjev!").end()
    }
})
```

API Gateway izvodi sličnu logiku, sadržanu u prethodnom programskom kodu (uz razliku korištenja drugačije klase za validaciju, ovisno o ulozi korisnika), *za svaki dobiveni HTTP zahtjev od korisničke web aplikacije akcijom korisnika*.

Provjera dostupnosti mikroservisa uvijek prethodi provjeri autorizacije istoga, jer nema smisla prosljeđivati zahtjeve korisničke web aplikacije neaktivnom ili „nespremnom“ mikroservisu. JavaScript klasa API Gatewaya, „ServisValidator“ izvršava provjeru dostupnosti mikroservisa istoimenom metodom.

```
class ServisValidator {

        async provjeriDostupnostServisa(adresaServisa) {

                var headers = config.get("headers")
                var response

                try {
                        response = await axios.get(adresaServisa +
"/health", { headers }, {timeout: 2000})
                }

                catch (e) {
                        return false
                }

                if (response.data.healthStatus === 0) {
                        return false
                }

                return true
        }
}

module.exports = ServisValidator;
```

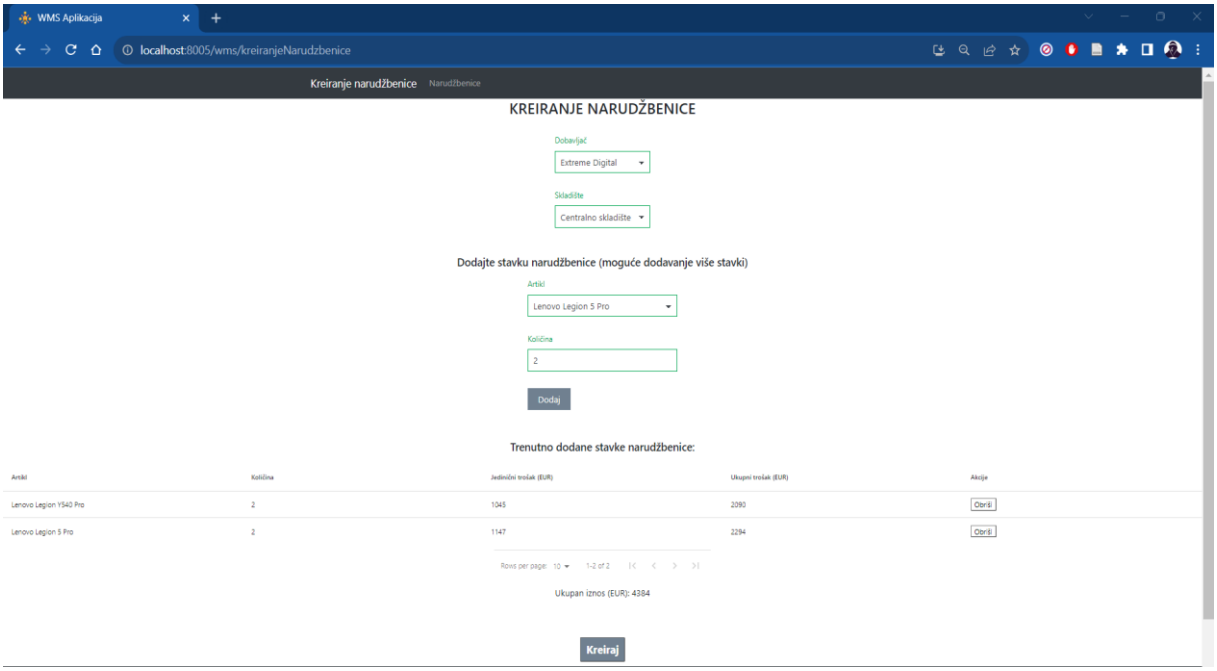
U metodi se prvo provjerava ugašenost mikroservisa (ukoliko mikroservis ne šalje odgovor u predviđeno vrijeme) i samo u slučaju da je mikroservis aktivan, šalje se HTTP zahtjev provjere zdravlja istome. Vrijednost dobivenog podatka „healthStatus“ u odgovoru mikroservisa mora imati vrijednost 1 (time se smatra da je mikroservis povezan sa svim svojim potrebnim infrastrukturnim uslugama: baza podataka i posrednik poruka). Svaki mikroservis ima kontroler zdravlja, čija metoda „getHealthStatus“ dostupna na putanji *„/health“* utvrđuje njegov status zdravlja (navedene povezanosti mikroservisa sa svojim infrastrukturnim

uslugama) i šalje za pristigle zahtjeve API Gatewaya u odgovoru, navedenu vrijednost statusa zdravlja „healthStatus“.

```
@RestController
@RequestMapping("/health")
public class KontrolerZdravlja {
    @Autowired
    private AmqpKonfiguracija ak;

    @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Object> getHealthStatus() {

        try (Connection konekcija = new
UpraviteljAmqpKonekcije(ak).dohvatiKonekciju(); Session sesija =
Hibernate.getInstanca().getSf().openSession()) {

            konekcija.getPort();

            TypedQuery<Object[]> upit = sesija.createQuery("SELECT
a.idArtikl from ArtiklEntitet a", Object[].class);
            upit.getResultList();

        } catch (Exception e) {
            return new ResponseEntity<>(new HealthStatus(0),
HttpStatus.OK);
        }

        return new ResponseEntity<>(new HealthStatus(1), HttpStatus.OK);

    }

    private static class HealthStatus {
        private int healthStatus;

        public HealthStatus(int status){
            this.healthStatus = status;
        }

        public void setHealthStatus(int healthStatus){
            this.healthStatus = healthStatus;
        }

        public int getHealthStatus(){
            return this.healthStatus;
        }
    }
}
```

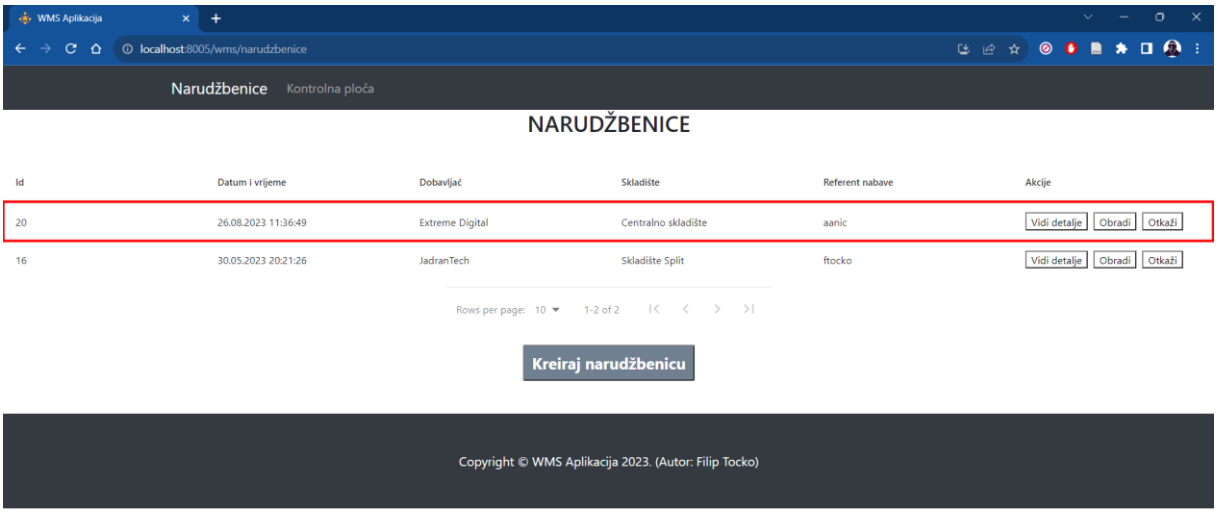
Opisana *provjera dostupnosti mikroservisa*, odgovara *uzorku dizajna mikroservisne arhitekture, „Provjera zdravlja“*. Sada će se opisati i prikazati nekoliko najčešćih slučajeva korištenja ovog sustava, unutar čega će biti opisan daljnji način funkcioniranja i realizacije sustava.

Najčešći slučajevi korištenja korisnika s ulogom referenta nabave su kreiranje i obrada (zaprimanje i fakturiranje) narudžbenice. Na idućoj slici je vidljiv zaslon korisničke web aplikacije, koji prikazuje proces kreiranja narudžbenice. Korisnik odabire dobavljača od kojeg će naručiti artikle i skladište na koje će biti zaprimljeni artikli, nakon izvršenja procesa zaprimanja nad narudžbenicom. Odabiru se i stavke narudžbenice (artikli i pripadne količine) odnosno artikli koji će biti naručeni od odabranog dobavljača na odabrano skladište.



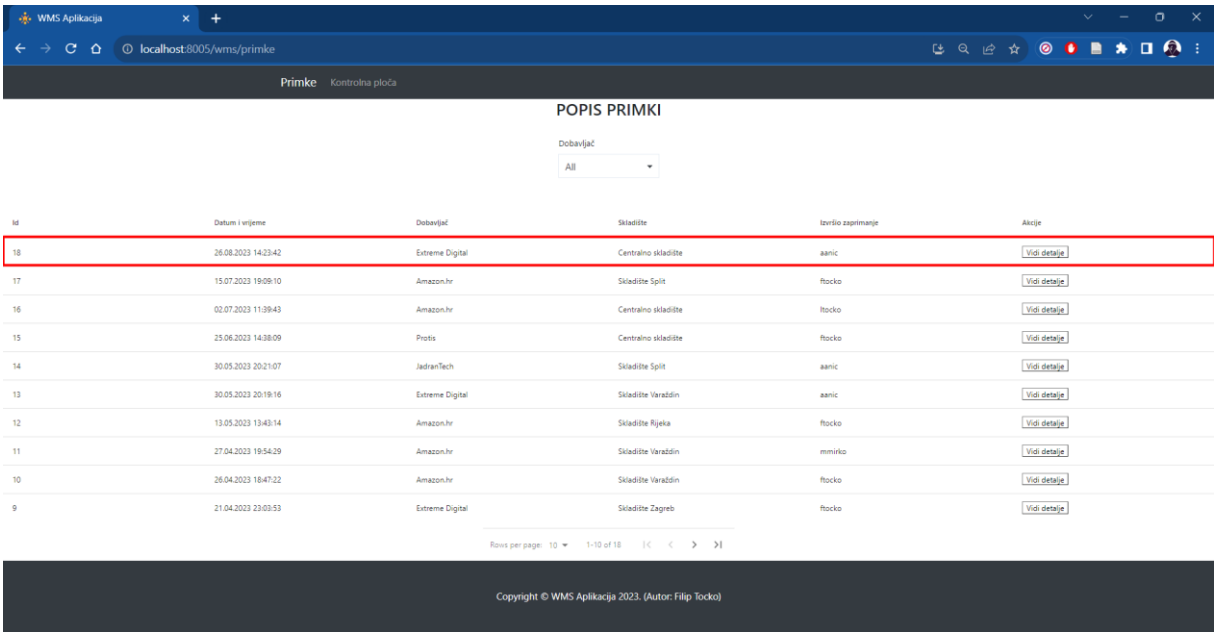
Slika 38. Prikaz zaslona kreiranja nove narudžbenice [autorski rad].

Nakon što su odabrani svi podaci nove narudžbenice, ista se može kreirati jednostavnim klikom na gumb „Kreiraj“. Nakon kreiranja iste, korisnik je preusmjeren na zaslon popisa trenutno neobrađenih narudžbenica, sortiranih prema vremenu kreiranja i novo kreirana narudžbenica je vidljiva na tom popisu.



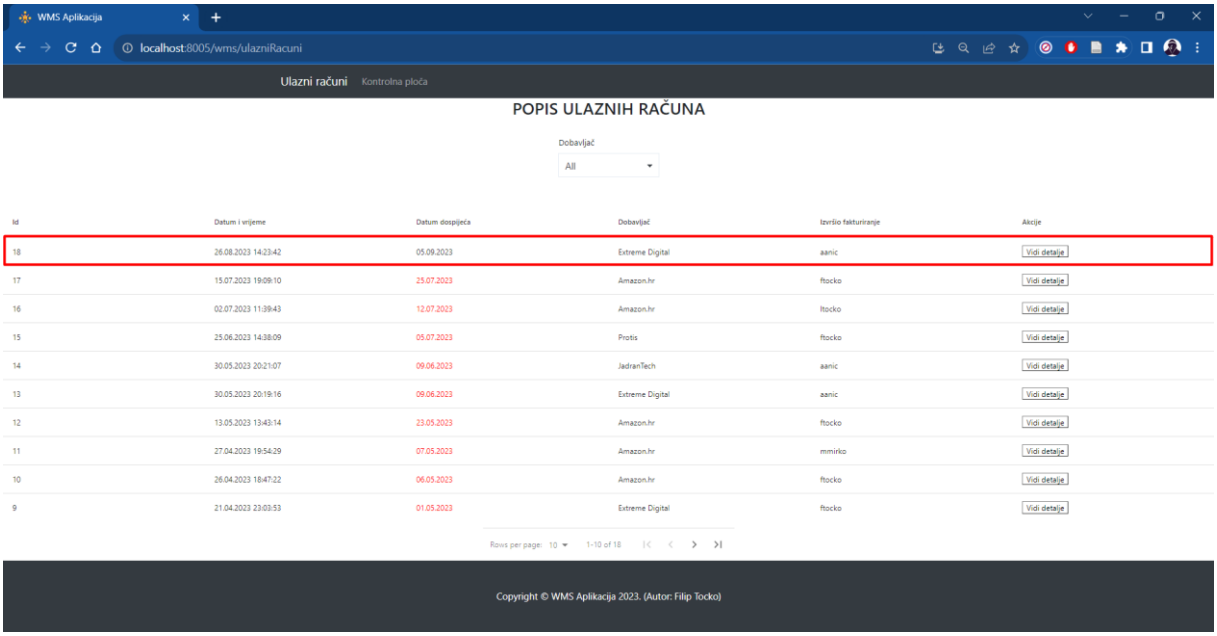
Slika 39. Prikaz zaslona popisa neobrađenih narudžbenica [autorski rad].

Klikom na gumb „Obradi“ označene novo kreirane narudžbenice, sustav izvodi proces obrade iste. Zapravo, mikroservis nabave izvodi proces obrade narudžbenice: kreiranjem primke (proces zaprimanja narudžbenice) i kreiranjem računa dobavljača (proces fakturiranja narudžbenice) te šalje asinkronu poruku mikroservisu upravljanje artiklima, kako bi isti mogao ažurirati stanje (povećati stanje) na skladištu zaprimljenih artikala. Kreirana primka je vidljiva na početku popisa primki.



Slika 40. Prikaz zaslona popisa primki [autorski rad].

Kreiran račun dobavljača je vidljiv na početku popisa ulaznih računa. Podatak datum dospijeća istoga nije crveno označen, jer mu datum dospijeća, još uvijek ne premašuje trenutni datum.



Slika 41. Prikaz zaslona popisa ulaznih računa [autorski rad].

Isječak metode obrade narudžbenice kontrolera narudžbenice mikroservisa nabave prikazuje slijedeći Java programski kod. Ista obrađuje narudžbenicu pozivom metode „obradiNarudzbenicu“ objekta „rn“ klase repozitorija nabave.

```
@PostMapping(value="/{id}/obrada", consumes =
MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Object> postObradiNarudzbenicu(@PathVariable("id")
int id, @RequestBody ReferentNabave referentNabave){

    try{
        if(rn.obradiNarudzbenicu(id, referentNabave.getObradio()) == 1){
            return new ResponseEntity<Object>("Uspješno zaprimljena i
fakturirana narudžbenica!", HttpStatus.OK);
        }

    }

    catch(Exception e){
        return new ResponseEntity<Object>("Neuspješno zaprimanje i
fakturiranje narudžbenice!",HttpStatus.BAD_REQUEST);
    }

    return new ResponseEntity<Object>("Neuspješno zaprimanje i fakturiranje
narudžbenice!",HttpStatus.BAD_REQUEST);
}
```

Prikazana metoda obrade narudžbenice, poziva unutar sebe metodu „posaljiPoruku“ objekta klase „PosiljateljPorukePovecanjaStanjaArtikla“, kako bi poslala poruku u potrebni red poruka (koristi potrebnu temu i ključ usmjeravanja) posrednika poruka RabbitMQ, kojeg osluškuje mikroservis upravljanje artiklima.

```
public class PosiljateljPorukePovecanjaStanjaArtikla extends
PosiljateljPoruke {

    public PosiljateljPorukePovecanjaStanjaArtikla(Connection konekcija,
AmqpKonfiguracija amqpKonfiguracija){
        super(konekcija, amqpKonfiguracija);
    }
    @Override
    public void posaljiPoruku(String poruka) throws Exception {

        Channel kanal = null;

        try {
            kanal = konekcija.createChannel();

            kanal.basicPublish(amqpKonfiguracija.tema,
amqpKonfiguracija.routingKljuc, null, poruka.getBytes());

        } catch (Exception e) {
            System.out.println(e.getMessage());
            if(kanal != null) kanal.close();
        }
        if(kanal != null) kanal.close();

    }
}
```

Mikroservis upravljanje artiklima osluškuje red poruka povećanja stanja artikla, pomoću metode „osluskujPoruke“ klase „PrimateljPorukePovecanjaStanjaArtikla“. Pristiglu poruku stanja artikala pretvara iz JSON formata u odgovarajući Java objekt i nakon toga poziva metodu repozitorija skladišta „azurirajStanjeArtiklaNaSkladistu“, kako bi se u bazi podataka ovog mikroservisa ažuriralo povećanje stanja artikala na skladištu.

```
public class PrimateljPorukePovecanjaStanjaArtikla extends PrimateljPoruke{

    public PrimateljPorukePovecanjaStanjaArtikla(Connection konekcija,
AmqpKonfiguracija amqpKonfiguracija, RepozitorijSkladista rs){
        super(konekcija, amqpKonfiguracija, rs);
    }
    @Override
    public void osluskujPoruke() throws Exception {

        Channel kanal = konekcija.createChannel();

        kanal.basicConsume(amqpKonfiguracija.redPovecanjaStanjaArtikla,
true, ((consumerTag, message) -> {

            String pristiglaPoruka = new String(message.getBody());
```

```
            PorukaAzuriranjaStanjaArtikla poruka = new
Gson().fromJson(pristiglaPoruka, PorukaAzuriranjaStanjaArtikla.class);

            rs.azurirajStanjeArtiklaNaSkladistu(poruka, "povećanje stanja
artikla na skladištu");

        }), consumerTag -> {
            System.out.println(consumerTag);
        });
    }
}
```

Objekt iznad prikazane klase, izvršava dretva u pozadini mikroservisa, čitavo vrijeme rada istoga. Ista se pokreće prilikom pokretanja mikroservisa. Stoga mikroservis ne mora biti aktivan, u trenutku kada mu drugi mikroservis pošalje poruku u red poruka, već pokretanjem istoga se u pozadini pokreće njegova dretva, koja se spaja i osluškuje red poruka posrednika poruka i čita još nepročitane poruke iz istoga te zatim provodi potrebne radnje. Poruka ostaje spremljena u redu poruka, sve dok ne biva pročitana od strane mikroservisa.

```
PrimateljPoruke pppsa = new
PrimateljPorukePovecanjaStanjaArtikla(uak.dohvatiKonekciju(),
        ak, new RepozitorijSkladistaImpl());

Thread pppsaDretva = new Thread(() -> {
    try {
        pppsa.osluskujPoruke();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
});
```

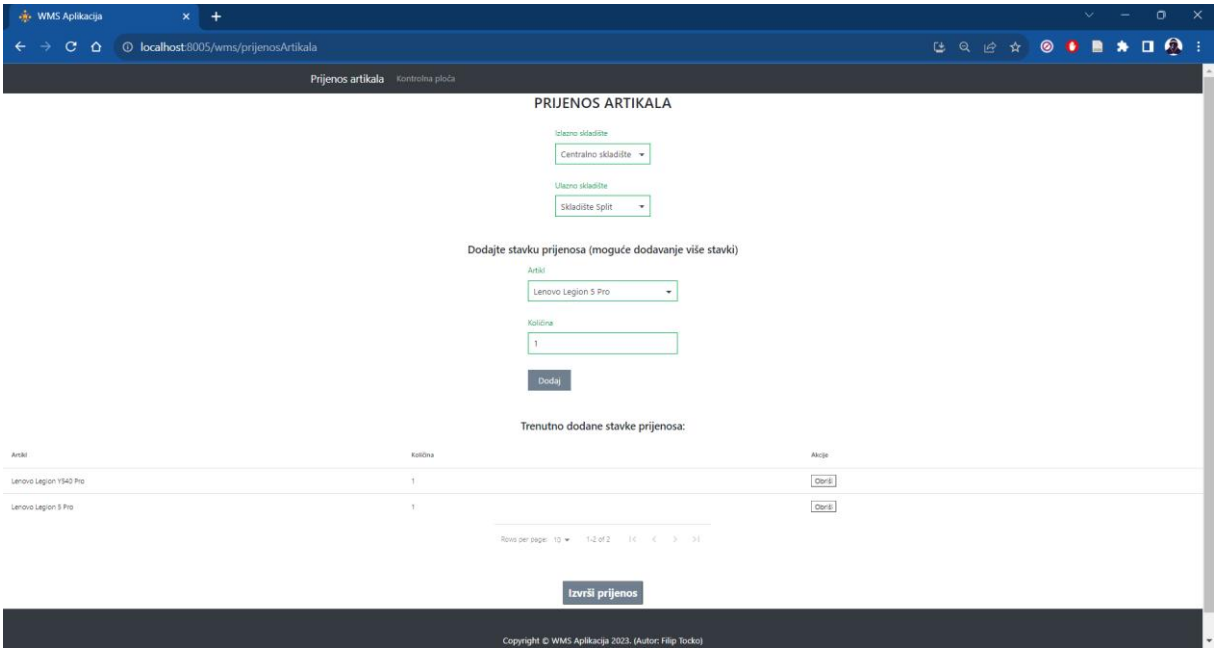
Opisani način *asinkrone komunikacije mikroservisa* na primjeru opisanih slučajeva korištenja aplikacije, vrijedi za sve mikroservise ovog sustava.

Za korisnika s ulogom skladištara, najčešći slučaj korištenja je prijenos artikala između dva skladišta. Korisnik prvo bira izlazno skladište (skladište iz kojeg se prenose odgovarajuće količine artikala) i ulazno skladište (skladište na koje će biti prenesene navedene količine artikala). Zatim se biraju artikli i njihove količine za prijenos. U trenutku odabira artikala i njegove količine, provjerava se dostupnost istoga na odabranom izlaznom skladištu, slanjem zahtjeva korisničke web aplikacije prema mikroservisu upravljanje artiklima, preko API Gatewaya. Prikazani isječak metode kontrolera mikroservisa upravljanje artiklima, provjerava za pristigli zahtjev dostupnost odabrane količine artikla na odabranom skladištu.

```
@GetMapping(value = "/provjeraStanjaArtikla", produces =
MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Object>
getProvjeraStanjaArtikla(@RequestParam("idIzlaznogSkladista") int
idIzlaznogSkladista, @RequestParam("idArtikla") int idArtikla,
```

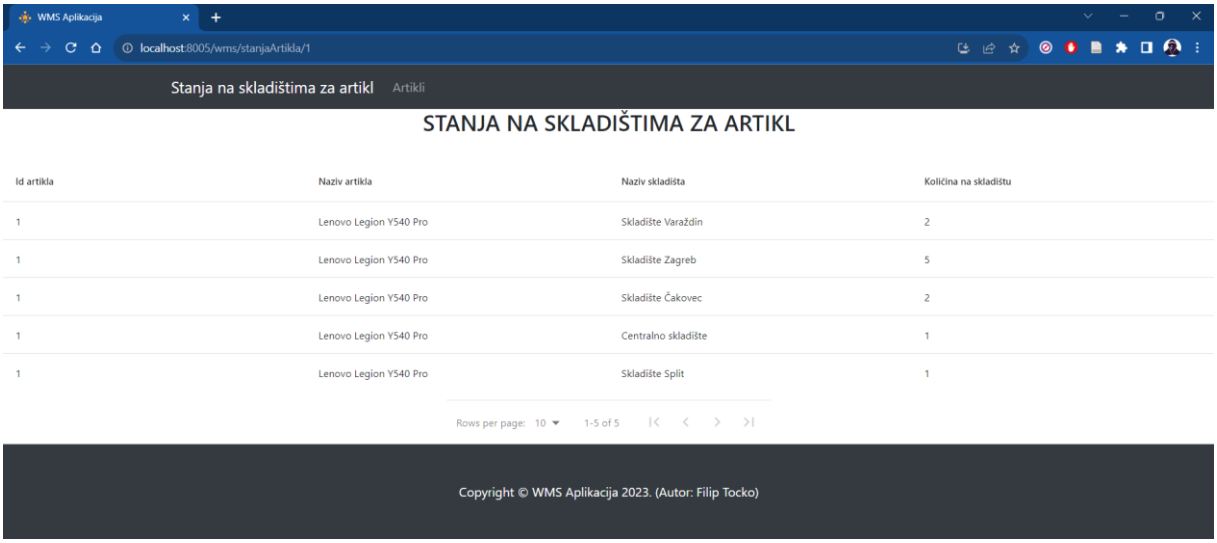
```
@RequestParam("kolicina") int kolicina){
        try{
            List<StanjeSkladistaArtikla> stanjaArtikla =
rs.provjeriStanjeNaSkladistuZaArtikl(new PrijenosArtikla(idArtikla,
kolicina, idIzlaznogSkladista));

            if(stanjaArtikla.size() > 0){
                return new ResponseEntity<>(stanjaArtikla, HttpStatus.OK);
            }
        }

        catch(Exception e){
            return new ResponseEntity<>("Neuspješno zahtijevanje provjere
stanja artikla!", HttpStatus.BAD_REQUEST);
        }

        return new ResponseEntity<>("Ne postoji raspoloživo stanje artikla
na izlaznom skladištu!", HttpStatus.NOT_ACCEPTABLE);
    }
}
```

Ako se dobiva uspješan odgovor od mikroservisa (status odgovora „200 OK“), tada se odabrana količina artikla dodaje u tablicu trenutno odabranih artikala za prijenos, u protivnome se ispisuje poruka pogreška korisniku. Konačno izvršenje prijenosa artikala se provodi korisničkim klikom na gumb „Izvrši prijenos“. Opisani prijenos artikala između dva skladišta je vidljiv na slici 42.



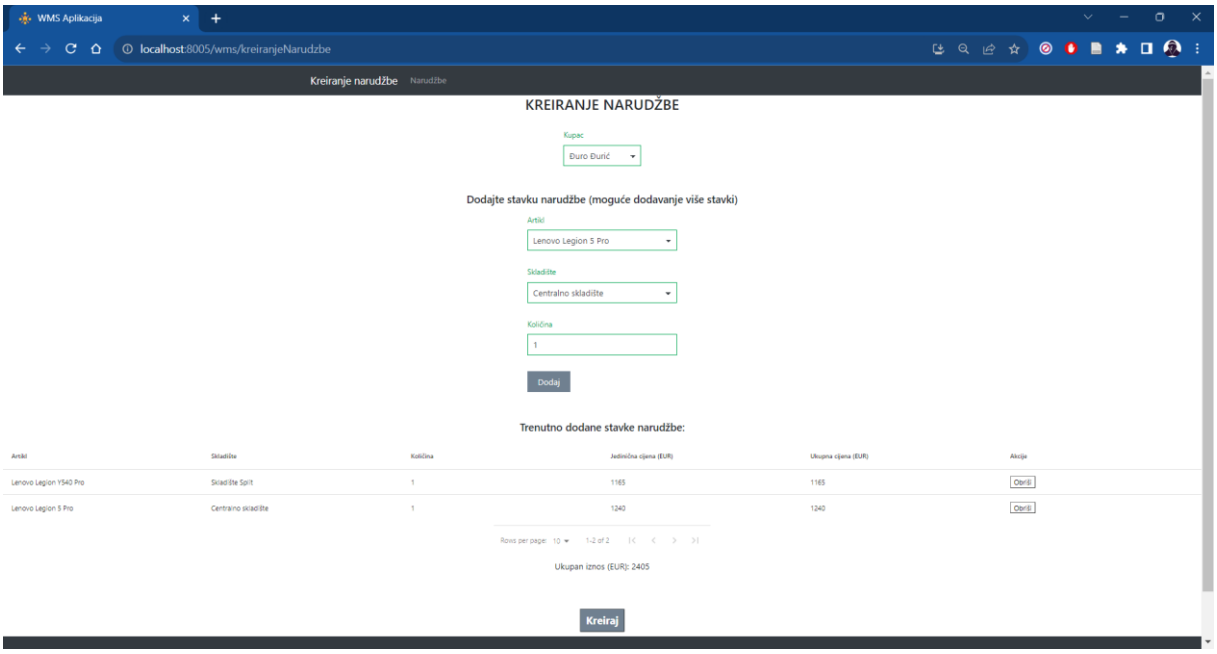
Slika 42. Prikaz zaslona prijenosa artikala između dva skladišta [autorski rad].

Za provjeru ispravnosti prijenosa artikala, korisnik može koristiti slijedeći zaslon stanja na skladištima za željeni artikl. Na slici 43. je prikazan za artikl iz prethodnog prijenosa artikala između dva skladišta.



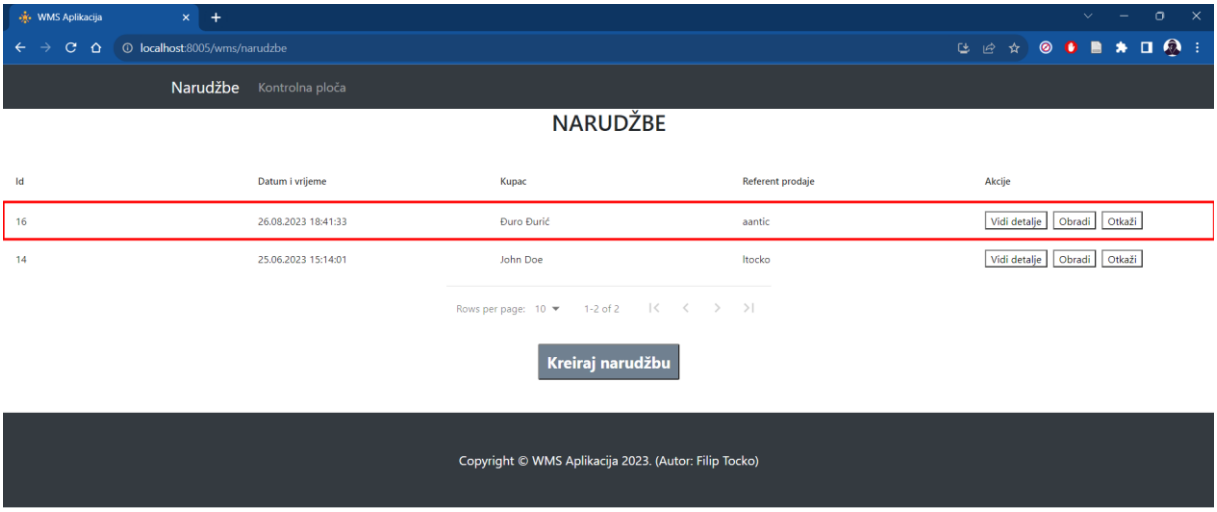
Slika 43. Prikaz zaslona stanja na skladištima za artikl [autorski rad].

Za referenta prodaje, najčešći slučajevi korištenja su kreiranje i obrada (otpremanje i fakturiranje) narudžbe. Na idućoj slici je vidljiv zaslon korisničke web aplikacije, koji prikazuje proces kreiranja narudžbe. Korisnik prvo odabire kupca koji kupuje artikle i zatim odabire stavke narudžbe (artikli, pripadne količine i skladište s kojeg se uzimaju isti). Prilikom odabira stavke narudžbe, provjerava se dostupnost odabrane količine artikla na odabranom skladištu, jednakim načinom, kao i kod prijenosa artikala između dva skladišta. Naime, kupac ne može kupiti artikl, kojeg nema na stanju u skladištu. Taj dio provjere korisnička web aplikacija obavlja već opisanom indirektnom komunikacijom s mikroservisom upravljanje artiklima, preko API Gatewaya.



Slika 44. Prikaz zaslona kreiranja narudžbe [autorski rad].

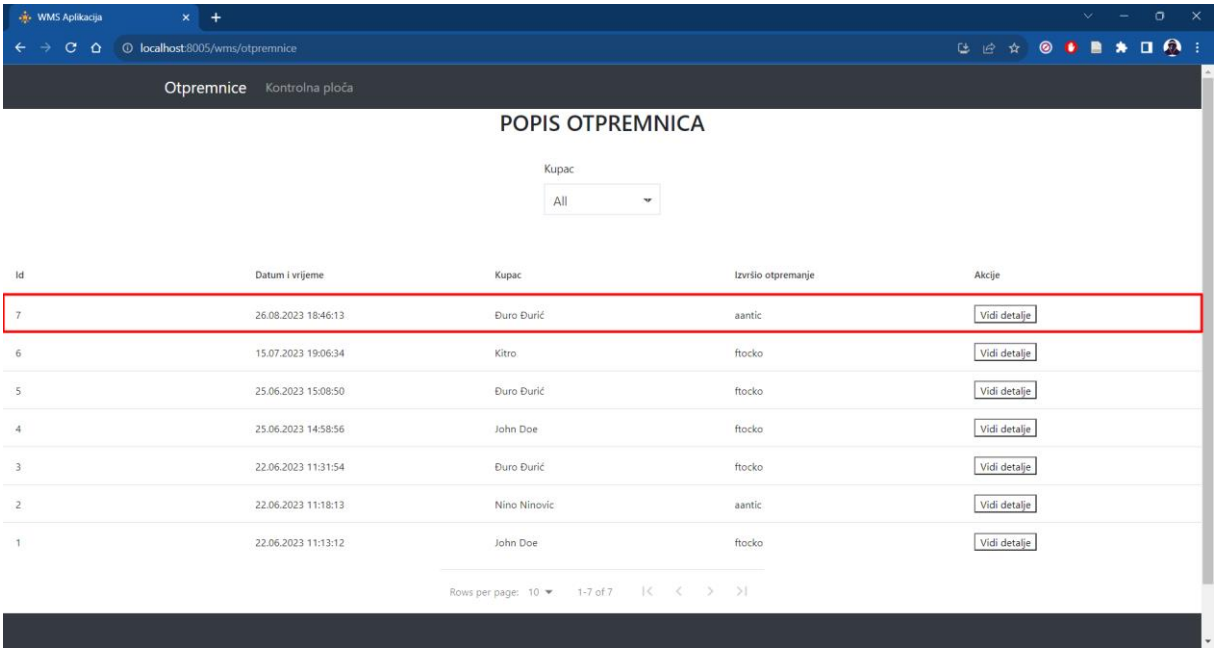
Nakon što su uspješno odabrani svi podaci nove narudžbe, ista se može kreirati jednostavnim klikom na gumb „Kreiraj“. Nakon kreiranja iste, korisnik je preusmjeren na zaslon popisa trenutno neobrađenih narudžbi, sortiranih prema vremenu kreiranja i novo kreirana narudžba je vidljiva na tom popisu.



Slika 45. Prikaz zaslona popisa neobrađenih narudžbi [autorski rad].

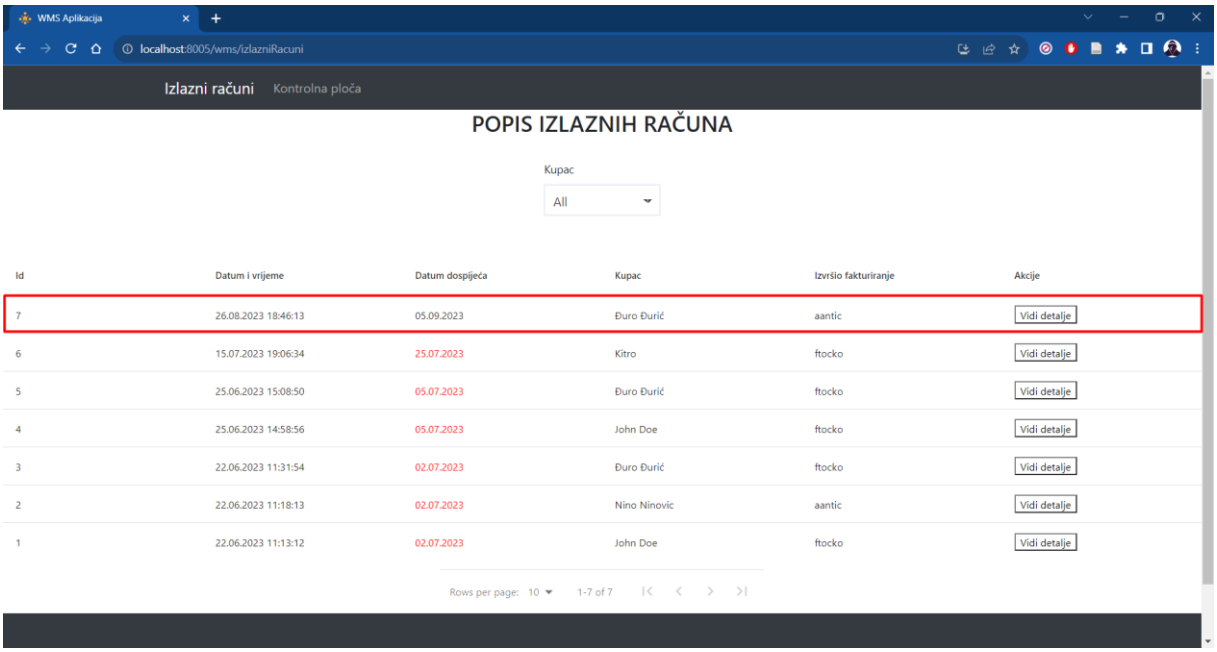
Klikom na gumb „Obradi“ označene novo kreirane narudžbe, sustav izvodi proces obrade iste. Zapravo, mikroservis prodaje izvodi proces obrade narudžbe: kreiranjem otpremnice (proces otpremanja narudžbe) i kreiranjem računa kupca (proces fakturiranja narudžbe) te šalje asinkronu poruku mikroservisu upravljanje artiklima, kako bi isti mogao

ažurirati stanje (smanjiti stanje) na skladištima otpremljenih artikala. Kreirana otpremnica je vidljiva na početku popisa otpremnica.



Slika 46. Prikaz zaslona popisa otpremnica [autorski rad].

Kreiran račun kupca je vidljiv na početku popisa izlaznih računa. Podatak datum dospijeća istoga nije crveno označen, jer mu datum dospijeća, još uvijek ne premašuje trenutni datum.



Slika 47. Prikaz zaslona popisa izlaznih računa [autorski rad].

Asinkrona komunikacija između mikroservisa prodaje i mikroservisa upravljanje artiklima za smanjenje stanja artikala na skladištima je istovjetna ranije opisanoj, kod obrade narudžbenice mikroservisa nabave. Mikroservisi nabave i prodaje dobivaju određene podatke korisnika od mikroservisa upravljanje korisnicima i određene podatke skladišta i artikala od mikroservisa upravljanje artiklima, također opisanom asinkronom komunikacijom (slanje asinkronih poruka podataka se pokreće nakon kreiranja ili ažuriranja tih podataka i definiranim periodom sinkronizacije). Oni dobivene podatke spremaju kao svoje lokalne kopije podataka. Pokretanje slanja asinkronih poruka, nakon pristizanja zahtjeva je već ranije detaljno opisano. Što se tiče definiranog perioda sinkronizacije, sinkronizacijska dretva mikroservisa upravljanje artiklima u pozadini čitavog rada istoga, svakih određenih broj minuta izvršava proces sinkronizacije (slanje asinkronih poruka pomoću pošiljatelja poruka) određenih podataka artikala i skladišta, kako bi mikroservisi nabave i prodaje, imali što svježije lokalne kopije izvornih podataka.

```
Thread sinkronizacijskaDretva = new Thread(() -> {
    try {
        while(true){
            posiljateljPorukePodatakaArtikala.posaljiPoruku();
            posiljateljPorukePodatakaSkladista.posaljiPoruku();
            sleep(ak.sinkronizacijaMin*60*1000);
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
});
```

Na sličan način i mikroservis upravljanje korisnicima provodi sinkronizaciju određenih podataka korisnika.

```
setInterval(function(){proizvodac.posaljiPoruku()},
sinkronizacijaMin*60*1000)
```

7. Zaključak

U teorijskom dijelu rada je detaljno opisana mikroservisna arhitektura i uspoređena u odnosu na ostale web arhitekture. Glavne pogodnosti i trenutna aktualnost mikroservisne arhitekture, rezultiraju sve većom prisutnošću iste kao bazne arhitekture sustava mnogih poduzeća. Teorijska obrada mikroservisne arhitekture je bila potrebna za razumijevanje i praktičnu primjenu iste izradom sustava za upravljanje skladištem. U praktičnom dijelu, sustav za upravljanje skladištem pokriva tj. sadrži glavne koncepte mikroservisne arhitekture. Svaki mikroservis je zadužen za svoju pod domenu i ima pripadnu bazu podataka te asinkrono komunicira s ostalim mikroservisima. API Gateway pruža autoriziran pristup do mikroservisa i prati zdravlje istih. Korisnička web aplikacija sve zahtjeve prema mikroservisima šalje preko API Gatewaya. Ovako izrađen sustav je pogodan za laganu daljnju nadogradnju, održavanje, proširenje i integraciju s ostalim sustavima. Zahvaljujući labavoj povezanosti mikroservisa, određeni dijelovi korisničke web aplikacije se mogu koristiti uz aktivnost samo nekih mikroservisa. Time ispad ili neaktivnost jednog mikroservisa ne rezultira ispadom cijelog sustava. Budući da se isto tako sva komunikacija u sustavu odvija preko API-ja i posrednika poruka, isti sadrži mješavinu modernih tehnologija i postoji mogućnost lagane promjenjivosti pojedine od njih, ukoliko se čitavo vrijeme poštuje i održava interoperabilnost. Sama prisutnost generičkih poslovnih procesa zamišljenog trgovačkog poduzeća u izrađenom sustavu, omogućuje korištenje tog sustava od strane bilo kojeg stvarnog trgovačkog poduzeća, ukoliko isto sadrži kompatibilne poslovne procese ili želi prilagoditi svoje poslovne procese. Iz svega navedenog se jasno može zaključiti važnost znanja o izgradnji i funkcioniranju sustava baziranih na mikroservisnoj arhitekturi, jer se danas sve više poduzeća orijentira prema korištenju istih.

Popis literature

[1] Fowler, Martin. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2003.

[2] „Web Application Architecture: The Latest Guide 2023“, *Peerbits*. https://www.peerbits.com/blog/web-application-architecture.html (pristupljeno 21. veljača 2023.).

[3] „Anatomy of a Modern 3-Tier Architecture“, *in-factory GmbH*, 26. travanj 2022. https://www.in-factory.com/en/anatomy-of-a-modern-3-tier-architecture-en/ (pristupljeno 21. veljača 2023.).

[4] „Microservices Pattern: Monolithic Architecture pattern“, *microservices.io*. http://microservices.io/patterns/monolithic.html (pristupljeno 21. veljača 2023.).

[5] C. Strîmbei *et al*, "Software Architectures - Present and Visions," *Informatica Economica,* vol. 19, *(4),* pp. 13-27, 2015. Available: https://www.proquest.com/scholarly-journals/software-architectures-present-visions/docview/1758012314/se-2. DOI: https://doi.org/10.12948/issn14531305/19.4.2015.02.

[6] S. Biswal *et al*, "Enterprise Application Integration Based Based On Service Oriented Architecture & Enterprise Service Bus: A Review," *International Journal of Advanced Networking and Applications, Suppl.Special Issue,* vol. 10, *(5),* pp. 116-119, 2019. Available: https://www.proquest.com/scholarly-journals/enterprise-application-integration-based-on/docview/2762020963/se-2.

[7] „SOA vs Microservices: What’s the Difference? | CrowdStrike“, *crowdstrike.com*. https://www.crowdstrike.com/cybersecurity-101/cloud-security/soa-vs-microservices/ (pristupljeno 22. veljača 2023.).

[8] „Service-Oriented Architecture - TTOW0130 - Service-Oriented Applications“. https://ttow0130.pages.labranet.jamk.fi/01.-SOA-%26-Microservices/01.soa/ (pristupljeno 22. veljača 2023.).

[9] I. Nadareishvili, R. Mitra, M. McLarty, i M. Amundsen, Microservice Architecture: Aligning Principles, Practices, and Culture. O’Reilly Media, Inc., 2016.

[10] G. Blinowski, A. Ojdowska, i A. Przybyłek, „Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation“, *IEEE Access*, sv. 10, str. 20357–20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[11] M. Kamaruzzaman, „Microservice Architecture and its 10 Most Important Design Patterns“, *Medium*, 31. prosinac 2021. https://towardsdatascience.com/microservice-architecture-and-its-10-most-important-design-patterns-824952d7fa41 (pristupljeno 22. veljača 2023.).

[12] A. K. Hamid, „Elastic Engineering“, Araf Karsh Hamid. https://arafkarsh.com/cloud-native-apps (pristupljeno 23. veljača 2023.).

[13] Wegner, Peter. "Interoperability." *ACM Computing Surveys (CSUR)* 28.1 (1996): 285-287.
[14] N. Puspitasari, E. Budiman, Y. N. Sulaiman, i M. B. Firdaus, „Microservice API Implementation For E-Government Service Interoperability“, *Journal of Physics: Conference Series*, sv. 1807, izd. 1, tra. 2021, doi: 10.1088/1742-6596/1807/1/012005.

[15] „The new European interoperability framework as a facilitator of digital transformation for citizen empowerment | Elsevier Enhanced Reader“. https://reader.elsevier.com/reader/sd/pii/S153204641930084X?token=47124136F46B763771AC9C967FE2DBC483046E886EED7476244229C3645A2736CDB00431B13C71015554E7CD03F5208A&originRegion=eu-west-1&originCreation=20230224174240 (pristupljeno 24. veljača 2023.).
[16] A. Tarkowska et al, "Eleven quick tips to build a usable REST API for life sciences," PLoS Computational Biology, vol. 14, (12), 2018. Available: https://www-proquest-com.ezproxy.nsk.hr/scholarly-journals/eleven-quick-tips-build-usable-rest-api-life/docview/2250637864/se-2. DOI: https://doi-org.ezproxy.nsk.hr/10.1371/journal.pcbi.1006542.

[17] J. Thönes, „Microservices“, IEEE Software, sv. 32, izd. 1, str. 116–116, sij. 2015, doi: 10.1109/MS.2015.11.

[18] S. Newman, Building Microservices: Designing Fine-Grained Systems. O’Reilly Media, Inc., 2015.

[19] „Advantages and Disadvantages of Microservices - javatpoint“. https://www.javatpoint.com/advantges-and-disadvantages-of-microservices (pristupljeno 06. ožujak 2023.).

[20] R. Richards, Representational State Transfer (REST), In: Pro PHP XML and Web Services. 2006. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4302-0139-7_17

[21] A. Agnihotri, „Restful API In .NET Core Using EF Core And Postgres“. https://www.c-sharpcorner.com/article/restful-api-in-net-core-using-ef-core-and-postgres/ (pristupljeno 07. ožujak 2023.).

[22] Hegde, G. Ranjith, and G. S. Nagaraja. "Low latency message brokers." Int. Res. J. Eng. Technol. 7 (2020): 5.

[23] A. Stec, „Database Design in a Microservices Architecture | Baeldung on Computer Science“, 14. svibanj 2021. https://www.baeldung.com/cs/microservices-db-design (pristupljeno 08. ožujak 2023.).

[24] V. Sharma, H. K. Saxena, i A. K. Singh, „Docker for Multi-containers Web Application“, u 2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA), ožu. 2020, str. 589–592. doi: 10.1109/ICIMIA48430.2020.9074925.

[25] J. Cao, „Design on Deployment of Microservices on Container-based Cloud Platform“, J. Phys.: Conf. Ser., sv. 1624, izd. 6, str. 062008, lis. 2020, doi: 10.1088/1742-6596/1624/6/062008.

[26] „What is a Container? | Docker“, 11. studeni 2021. https://www.docker.com/resources/what-container/ (pristupljeno 10. ožujak 2023.).

[27] „Microservices Design Patterns Tutorial“. https://www.tutorialspoint.com/microservices_design_patterns/index.htm (pristupljeno 10. ožujak 2023.).

[28] V. F. Pacheco, Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices. Birmingham, UNITED KINGDOM: Packt Publishing, Limited, 2018. Pristupljeno: 10. ožujak 2023. [Na internetu]. Dostupno na: http://ebookcentral.proquest.com/lib/uzuccs/detail.action?docID=5259455

[29] E. Daraghmi, Z. Cheng-Pu and S. Yuan, "Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture," Applied Sciences, vol. 12, (12), pp. 6242, 2022. Available: https://www.proquest.com/scholarly-journals/enhancing-saga-pattern-distributed-transactions/docview/2679673542/se-2. DOI: https://doi.org/10.3390/app12126242.

[30] F. Montesi i J. Weber, „Circuit Breakers, Discovery, and API Gateways in Microservices“. arXiv, 21. rujan 2016. Pristupljeno: 22. ožujak 2023. [Na internetu]. Dostupno na: http://arxiv.org/abs/1609.05830

[31] S. Tilkov i S. Vinoski, „Node.js: Using JavaScript to Build High-Performance Network Programs“, IEEE Internet Computing, sv. 14, izd. 6, str. 80–83, stu. 2010, doi: 10.1109/MIC.2010.145.

[32] „Node.js“, Node.js. https://nodejs.org/en (pristupljeno 29. svibanj 2023.).

[33] „Home“, Home. https://spring.io/ (pristupljeno 29. svibanj 2023.).

[34] „Messaging that just works — RabbitMQ“. https://www.rabbitmq.com/ (pristupljeno 29. svibanj 2023.).

[35] „MongoDB: The Developer Data Platform“, MongoDB. https://www.mongodb.com (pristupljeno 29. svibanj 2023.).

[36] „What is MySQL?“ https://www.oracle.com/mysql/what-is-mysql/ (pristupljeno 29. svibanj 2023.).

[37] „MySQL | Most Popular Open Source Relational Database | AWS“, Amazon Web Services, Inc. https://aws.amazon.com/rds/mysql/what-is-mysql/ (pristupljeno 29. svibanj 2023.).

[38] „React“. https://react.dev/ (pristupljeno 29. svibanj 2023.).

Popis slika

Popis tablica