Razvoj mikroservisa korištenjem ASP.Net Core

Marijana, Presečki

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marijana Presečki

RAZVOJ MIKROSERVISA KORIŠTENJEM ASP.NET CORE

DIPLOMSKI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marijana Presečki

Matični broj: 0016116544

Studij: Informacijsko i programsko inženjerstvo

RAZVOJ MIKROSERVISA KORIŠTENJEM ASP.NET CORE

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, rujan 2020.

Marijana Presečki

Izjava o izvornosti

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

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

_____

Sažetak

U radu je predstavljena mikroservisna arhitektura. Navedene su i detaljno razrađene njezine glavna karakteristike, prednosti te nedostaci takve arhitekture. Nadalje je opisano korištenje takve arhitekture prilikom razvoja aplikacija te izazovi koji se pritom javljaju: veličina mikroservisa, zavisnosti koda, komunikacija u timu i među timovima i slično. Prikazana je i detaljno opisana razlika između mikroservisne i servisno-orijentirane arhitekture. Također su predstavljeni REST servisi i poruke, odnosno, komunikacija te integracija i kontinuirana isporuka mikroservisa uz korištenje DevOpsa. Kao praktičan dio rada izrađena je aplikacija MyCookbook za čije su potrebe dizajnirani i implementirani mikroservisi pomoću Microsoftove tehnologije ASP.Net Core, korisničko sučelje je izrađeno u Angularu, dok je za pripremu isporuke mikroservisa korišten Docker. Orkestracija kontejnera je napravljena uz pomoć Kubernetes servisa, a aplikacija je isporučena pomoću raznih Azure servisa. Ovaj rad je napravljen u okviru Laboratorija za web arhitekture, tehnologije, servise i sučelja.

Ključne riječi: mikroservisi; mikroservisna arhitektura; REST servisi; integracija; skaliranje; kontejneri;

Sadržaj

1. Uvod

U današnje vrijeme mikroservisi i mikroservisna arhitektura postaju sve više privlačniji prilikom razvoja modernih aplikacija. Tako mikroservisi postaju sve češća fraza korištena u svijetu softvera te se o toj temi može često čuti na raznim konferencijama, susretima, blog objavama i slično. Upravo su popularnost i brojne prednosti ove arhitekture bile glavna motivacija za pisanje ovog rada.

U radu će najprije biti predstavljena mikroservisna arhitektura te će detaljno biti opisane ključne karakteristike, prednosti te nedostaci takve arhitekture. Zatim će biti spomenuti izazovi koji se javljaju kod korištenja mikroservisa te se će oni biti uspoređeni sa servisno-orijentiranom arhitekturom. Nadalje, opisat će se korištenje takve arhitekture prilikom razvoja aplikacija te izazovi koji se pritom javljaju: veličina mikroservisa, zavisnosti koda i slično. U radu će biti predstavljeni REST servisi i poruke te integracija, komunikacija i kontinuirana isporuka mikroservisa. Kao praktičan dio rada izradit će se aplikacija MyCookbook za čije potrebe će se implementirati mikroservisi pomoću Microsoftove tehnologije ASP.Net Core, dok će korisnički dio web aplikacije biti implementiran u Angularu.

2. Metode i tehnike rada

Opseg diplomskog rada zahtijeva teorijski pregled mikroservisne arhitekture i korištenih alata te praktičnu izradu aplikacije čiji se radi temelji na mikroservisima.

Izrada rada je započela pretraživanjem dostupne literature koja je služila kao primarni izvor informacija za pisanje diplomskog rada. Uglavnom se koristi stručna literatura, kao što su knjige i IEEE članci, ali i vodiči (primjerice, Microsoftov vodič za razvoj mikroservisne arhitekture) te ostala literatura.

Za potrebe izrade programskog dijela proučeni su vodiči, primjeri dobre prakse na Internetu te službena dokumentacija. Tehnologije koje su korištene za razvoj su ASP.Net Core za pozadinski dio i Angular za korisničko sučelje, dok su od alata korišteni Visual Studio 2019, Visual Studio Code, Docker, Azure servisi i ostali. Navedeni alati te pomoćne biblioteke su kasnije i opisani.

3. Mikroservisi

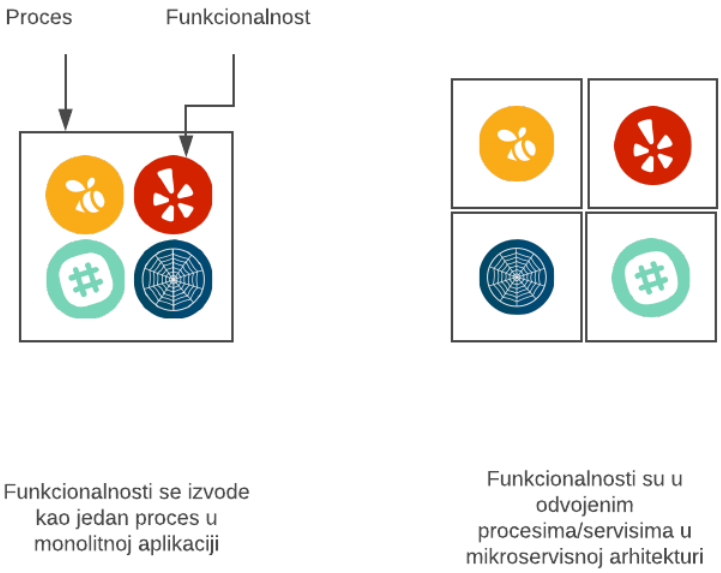
Mikroservisi su arhitekturalni stil za razvoj aplikacije na poslužitelju koja se sastoji od više manjih i neovisnih servisa koji rade zajedno. Međusobno su slabo povezani, a mogu se neovisno instalirati, skalirati, testirati te imaju samo jednu odgovornost. Navedeno označava da mikroservis obavlja samo jedan zadatak, odnosno, implementira određenu poslovnu sposobnost. Svaki mikroservis predstavlja jedan proces, a s drugim mikroservisima može komunicirati koristeći HTTP/HTTPS protokole, WebSocket ili druge mrežne protokole za slanje poruka. Kako je glavna značajka ovog arhitekturalnog uzorka da su servisi međusobno neovisni, svaki može biti pisan u drugom programskom jeziku ili se može temeljiti na drugim tehnologijama za pohranu podataka. Svaki pojedini servis bi trebao imati svoj model podataka i poslovnu logiku. Ono što treba biti zajedničko je platforma na kojoj se izvršavaju, primjerice, virtualne mašine. [1] [2] [3] [4] [5]

Ovaj arhitekturalni stil zapravo predstavlja evoluciju servisno-orijentirane arhitekture (kraće SOA) i novi pristup modularizaciji aplikacija. Razlike između te dvije arhitekture se očituju u izgradnji modula: dok SOA predstavlja integracijsko rješenje više servisa, mikroservisi se koriste za izgradnju pojedinačnih servisa/aplikacija. Detaljnije razlike između arhitektura su opisane u kasnijim poglavljima. [3] [6]

Kako je prethodno navedeno, neovisni servisi trebaju biti **mali i fokusirani** na jedan dio funkcionalnosti. Uz to se veže načelo jedinstvene odgovornosti (eng. *Single Responsibility Principle*) koje govori da se grupira programski kod koji je povezan i koji se mijenja iz istog razloga čime se servisu postavljaju granice te je on fokusiran na poslovnu domenu obuhvaćenu tim granicama. S vremenom to donosi brojne prednosti: servis neće previše „narasti“, a implementacija i ispravljanje pogrešaka će biti olakšani. Pitanje koje se ovdje nameće je koliko zapravo takav servis treba biti malen. U literaturi se navode razne metode kojima se određuje veličina servisa, kao što su broj linija koda ili pak uvjeti domene. Microsoft [7] navodi da prilikom razvoja veličina ne bi trebala biti važna, već bi se pažnja trebala usmjeriti na to da se kreiraju slabo povezani servisi koji omogućuju autonomiju razvoja, skaliranja i instaliranja. Naravno, servisi trebaju biti što manji, imati što veću unutarnju koheziju i što manje direktnih ovisnosti s drugim mikroservisima. Mikroservis treba biti tretiran kao aplikacija za sebe te u mnogim slučajevima ima svoj repozitorij te alate za izgradnju i instalaciju. [1] [2] [4] [5]

Autonomija je vrlo važna karakteristika mikroservisa. Svaki mikroservis je odvojeni entitet koji može imati vlastiti proces ili pak može biti izolirani servis na platformi kao usluzi (eng. *Platform as a Service*, PaaS). Kako je svaki servis odvojen i neovisan, sva

komunikacija između njih se odvija kroz mrežne pozive što omogućuje česte i brze promjene na produkciji. Svaki servis se stoga može neovisno mijenjati, što utječe na to da se korisnicima mogu brže dostaviti nove funkcionalnosti i poboljšanja. [1] [4] Autonomija predstavlja veliku prednost u odnosu na monolitne aplikacije koji su vrlo opsežne i čije se čvrsto povezane komponente izvode u istom procesu.



Slika 1 Procesi u mikroservisnim i monolitnim aplikacijama [autorski rad]

Iako su mikroservisi povezani tako da tvore jednu kompleksnu aplikaciju, mogu biti pisani u različitim programskim jezicima. **Jezična neutralnost** omogućuje da se koriste one tehnologije koje su najprikladnije za zadatak kojeg servis obavlja. Servis obično otkiva sučelje za programiranje aplikacija (eng. Application Programming Interface, API) koje je jezično neutralno te se koriste REST pozivi za međusobnu komunikaciju. [4]

Svaki mikroservis je zadužen za samo jedan zadatak iz domene i treba ga dobro obavljati. Kako bi se postigao taj **ograničeni kontekst** mikroservisa, potrebno je napraviti dobar model servisa i API-a koji uglavnom predstavljaju najteži zadatak prilikom korištenja ovog arhitekturnog stila. [4]

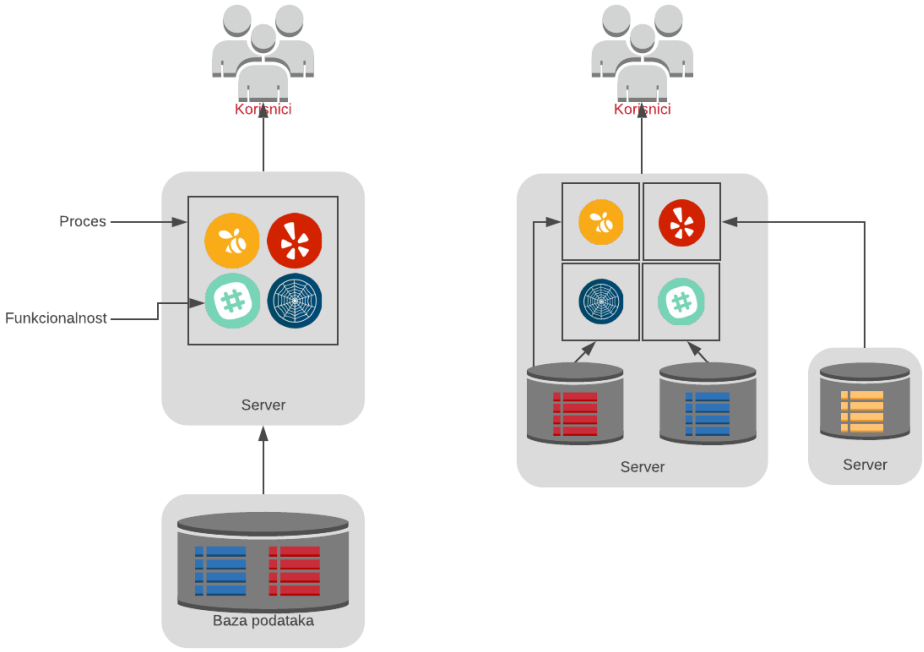
3.1. Prednosti mikroservisa

Za popularnost mikroservisa zadužene su brojne prednosti koje ovaj stil donosi. Neke od njih su već spomenute te one predstavljaju ujedno i najvažnije prednosti: brže promjene, skalabilnost te velika autonomija. Mikroservisi imaju neke prednosti servisno-orijentirane

arhitekture i distribuiranih sustava, ali oni te prednosti bolje iskorištavaju. One koje će biti opisane su: modularizacija, heterogenost tehnologija, skaliranje, otpornost na greške, zamjenjivost, održiv razvoj, kraće vrijeme za isporuku, kontinuirana isporuka, organizacijsko usklađivanje te ponovna iskoristivost funkcionalnosti.

Mikroservisi strogo poštuju **koncept modularizacije**. Komuniciranjem samo preko sučelja - REST poziva ili poruka sprječavaju da se kreiraju i koriste neželjene ovisnosti te time olakšavaju daljnji razvoj i održavanje. Svrha takve arhitekture je da se razvijaju zasebni moduli kako bi se lakše prebrodila kompleksnost zadatka. Zasebni moduli olakšavaju posao programerima jer moraju znati samo o modulu na kojem rade što pak pozitivno utječe na produktivnost tima. [5]

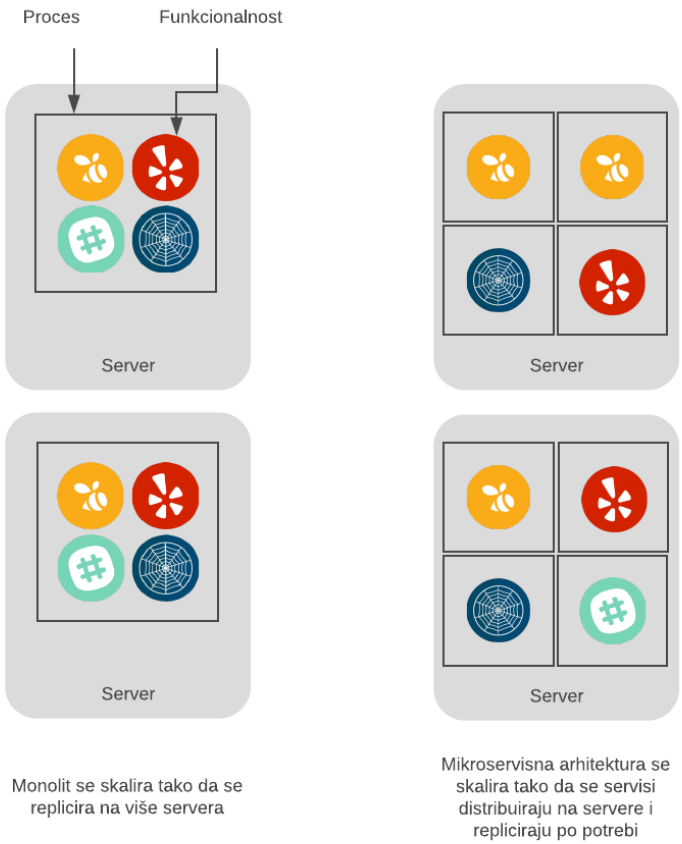
Kao što je prethodno navedeno, mikroservisi mogu biti pisani u različitim programskim jezicima. **Heterogenost tehnologija** omogućuje da se koriste prikladni alati za svaki servis, a ne oni koji će zadovoljiti većinu potreba, kao što je to slučaj kod monolitnih aplikacija. Nadalje, svaki servis je neovisan pa, ako je potrebno, mogu se vrlo lako zamijeniti tehnologije koje neke servis koristi (primjerice, za poboljšanje performansi). Isto tako se lakše mogu primijeniti poboljšanja ili nove tehnologije, jer se ne mora mijenjati cijela aplikacija, već samo manji mikroservis bez utjecaja na druge. To može pozitivno utjecati na produktivnost programera jer ih potiče na inovativnost i sprječava zastarijevanje aplikacije. Iako se za razvoj mogu koristiti razni alati, poduzeća se najčešće odlučuju za one alate koji su im poznati, dovoljno pouzdani i imaju prihvatljive performanse. [1] [5] Na slici 2 je prikazano kako svaki mikroservis dopušta upravljanje vlastitom bazom podataka – prilikom razvoja baze može se koristiti ista tehnologija ili pak sasvim drugačiji sustav, dok monolitne aplikacije preferiraju jednu bazu podataka za perzistenciju podataka.



Slika 2 Baze podataka kod mikroservisnih i monolitnih aplikacija [autorski rad]

Jedna od velikih prednosti ove arhitekture je **otpornost** na greške. Kada se u nekoj komponenti dogodi greška, potrebno je spriječiti njezino širenje i izolirati problem kako bi aplikacija nastavila s radom. Kako su mikroservisi odvojeni procesi i imaju svoja domenska ograničenja, prilikom greške neće doći do pada sustava ako se pravilno rukuje s greškom i ako se njezin utjecaj ograniči na funkcionalnost gdje se ta greška dogodila. To nije slučaj kod monolita gdje sve prestane s radom ako se u servisu dogodi greška. Iako mikroservisi pružaju veću otpornost na greške, javljaju se i novi problemi na koje treba obratiti pozornost: mreža i dostupnost mreže, pad platforme i slično. Navedeno treba uzeti u obzir prilikom razvoja te implementirati pravilno rukovanje takvim greškama. [1]

Skaliranje mikroservisa pojednostavljuje infrastrukturu i operacije povezane s istim. Naime, svaki mikroservis može biti skaliran neovisno od ostalih pa se ne mora skalirati cijeli sustav (kao kod monolitnih aplikacija) već samo one funkcionalnosti koje se češće koriste. To isto tako omogućuje da se ostali, rijetko korišteni dijelovi sustava izvršavaju na slabijem hardveru i u manjem broju. [1] [5] Razlika između skaliranja monolitnih i mikroservisnih aplikacija je prikazana na slici 3.



Slika 3 Skaliranje mikroservisnih i monolitnih aplikacija [autorski rad]

Sljedeće prednosti mikroservisa su **jednostavna implementacija i zamjenjivost**. Mikroservisi se mogu lakše izmijeniti nego moduli u monolitnim aplikacijama jer se kod promjena ne implementira/instalira cijela aplikacija, već samo dio - mikroservis čiji se programski kod mijenjao. Taj mikroservis se može implementirati neovisno od ostatka sustava što omogućuje i bržu implementaciju. Prilikom toga se često mogu pojaviti greške, što kod velikih monolitnih aplikacija predstavlja visoki rizik i može imati veliki utjecaj na njihov rad. S druge pak strane, takav problem je lakše izolirati kod mikroservisa gdje se problem može brže riješiti i staviti u produkciju. To je ujedno i jedan od razloga zašto su se neke od velikih organizacija odlučili upravo za ovu arhitekturu. [1]

Zamjenjivost i jaka modularizacija omogućuju **održiv razvoj** programskog proizvoda. Takva arhitektura omogućuje neovisnost aplikacije o tehnologiji, što omogućuje da svaki modul/ mikroservis može biti neovisno zamijenjen kad tehnologija zastari. [5]

Prednost koja se veže na prethodne dvije je **kraće vrijeme potrebno za izaći na tržište** (eng. *time-to-market*) s nekom funkcionalnošću. Mikroservisi i njihove funkcionalnosti mogu razvijati i implementirati odvojeno te tako i dolaziti zasebno na produkciju. Slično je s promjenama jer se one također mogu razviti i dovesti brže na produkciju. Svaki tim je obično

zadužen za jedan mikroservis te se prilikom razvoja ne mora koordinirati s ostalim timovima, već se mikroservisi mogu paralelno razvijati što skraćuje vrijeme razvoja te omogućuje brže dostavljanje promjena krajnjim korisnicima. [5]

Još jedna korist od mikroservisa je što zahtijeva manje timove koji rade na projektu manjeg opisa, a takvi timovi su obično produktivniji od onih većih i od onih koji rade na velikom projektu koji zahtijeva koordiniranje. Time arhitektura projekta omogućuje **organizacijsko usklađivanje** i formiranje manjih distribuiranih timova kako bi se postigla odgovarajuća produktivnost i veličina tima. S ekonomskog gledišta, to znači da se manji i usredotočeni timovi bolje usklađuju s poslovnim prihodima i rashodima jer dobivaju bržu povratnu informaciju od korisnika. Timovi mogu brže napraviti preinake i agilniji su. [1] [4]

Ono što su mikroservisi naslijedili od servisno-orijentirane arhitekture i distribuiranih sustava je **ponovna iskoristivost funkcionalnosti**. Naime, funkcionalnost mikroservisa se može koristiti na različite načine i u različite svrhe. U današnje vrijeme korisnici mogu koristiti različite uređaje za pregled online sadržaja: web, desktop računala, nativne mobilne aplikacije, mobilni web, tableti, nosivi uređaji i tako dalje. Prilikom razvoja proizvoda, važno je odabrati arhitekturu koja će moći podržati takve sposobnosti i opsluživati razne aplikacije. [1]

Mikroservisna arhitektura omogućuje prebrođivanje problema kod daljnjeg razvoja postojećih (eng. *legacy*) aplikacija. To su aplikacije koje koriste zastarjelu tehnologiju i programski kod kojeg je teško razumjeti. Usprkos tome, takav sustav se može unaprijediti korištenjem mikroservisa ukoliko je potrebno obraditi dodatne zahtjeve, dok je za ostalo i dalje zadužen postojeći sustav. Time mikroservis nije vezan za tehnologije koje se koriste u zastarjelom sustavu te nije potrebno zamijeniti cijeli sustav. [1]

Kontinuirana isporuka je disciplina kod razvoja softvera koja u današnje vrijeme sve više dobiva na značaju. Ukratko, ova disciplina omogućuje isporuku proizvoda na produkcijsku okolinu u bilo koje vrijeme, a uključuje automatizirane testove i automatiziranu isporuku. Mikroservisni pristup je vrlo povoljan za kontinuiranu isporuku upravo zbog toga što su mikroservisi maleni i neovisni što čini kanal isporuke (eng. *continuous delivery pipeline*) jednostavnim. Pomoću mikroservisa se lakše osigurava sigurna isporuka bez pogrešaka, za razliku od velikih monolita. [5] [8]

3.2. Kad ne koristiti mikroservise?

Prije mikroservisa, mnogo poduzeća je razvijalo monolite – velike aplikacije čije se komponente izvode u istom procesu te komuniciraju kroz pozive metoda. Često su problemi kod takvih aplikacija bili vezani uz skaliranje, održavanje i upravljanje takvim sustavom,

osobito kad se on nalazi u oblaku. Zbog toga su se postupno počele razvijati slabo povezane kolekcije mikroservisa koje su rješavale spomenute probleme. U današnje vrijeme je takva arhitektura uzela maha i postala trend te se mikroservisni pristup javlja na sve više projekata. Usprkos tome, mikroservisi nisu univerzalno rješenje te valja uzeti u obzir i mane te arhitekture. Takva analiza pomaže ograničiti trošak prilikom implementiranja mikroservisne arhitekture. [4]

Prvi slučaj u kojem se ne preporuča korištenje mikroservisa je početak razvoja nove aplikacije. Naime, u početku su aplikacije vrlo male i na njima radi tek nekolicina programera koji vrlo dobro poznaju kod. Kako je aplikacija tek u začetku, nema korisnika ili ih ima vrlo malo pa nema potrebe za mikroservisima koji se koriste za rješavanje problema skaliranja. Međutim, kad aplikacija naraste i približi se statusu monolita, vrlo važno je prepoznati nove potrebe i reagirati na vrijeme, što bi značilo dekomponirati monolit i uvesti mikroservisnu arhitekturu. [4]

Pojam mikroservisa je usko vezan uz kontinuiranu isporuku i DevOps - skup alata i praksi koji omogućuju brzu isporuku funkcionalnosti (detaljnije o DevOpsu u poglavlju 5.9). Dakle, prilikom implementiranja mikroservisne arhitekture važno je omogućiti automatiziranu isporuku i nadziranje te se u suprotnom ne preporuča koristiti ovaj pristup. Nadalje, mikroservisi koriste različite baze podataka, posrednike za poruke i ostale servise, a koji svi trebaju biti održavani i grupirani. Zbog toga može biti otežano upravljanje infrastrukturom te se umjesto infrastrukture kao usluge (eng. *Infrastructure as a Service*, IaaS) preporuča korištenje platforme kao usluge. [4]

Sljedeće što treba imati na umu je broj mikroservisa. Njih ne smije biti previše jer svaki od njih troši resurse, a kumulativno korištenje resursa može imati negativan učinak ako se prekorači broj mikroservisa koje DevOps procesi, alati i operacije mogu podnijeti. Ukoliko se kreira mnogo premalih mikroservisa, može biti otežana njihova integracija. Zbog toga je preporučljivo napraviti nešto veće servise i razdvojiti ih tek kada se jave dijelovi s konfliktima u skaliranju, podacima ili životnom ciklusu. [4]

Prilikom razvoja mikroservisa, treba voditi računa i o potencijalnim kašnjenjima. Naime, do problema kašnjenja može dovesti prevelika zrnatost mikroservisa ili korištenje ovisnosti. Nadalje, kad se sustav dekomponira u mikroservise, povećava se broj poziva na mreži prema usluzi koja je zadužena za zahtjev. Ti pozivi mogu biti od servisa prema servisu ili od servisa prema perzistentnoj komponenti - bazi podataka. Sve to može usporiti rad sustava pa je potrebno izvoditi testove performansi kako bi se otkrila kašnjenja i uska grla. [4]

3.3. Izazovi arhitekture

Mikroservisna arhitektura se, kao i ostale, susreće s brojnim izazovima. Glavni izazovi koji prate tu arhitekturu su skrivene veze, refaktoriranje, arhitektura domene, izvođenje mikroservisa te izazovi distribuiranih sustava.

Svaki od izazova je ukratko opisan u nastavku [5]:

- Skrivene veze – arhitektura sustava u kojem se koriste mikroservisi je takva da postoje veze između mikroservisa, no nije vidljivo koji mikroservisi se međusobno pozivaju što svakako predstavlja jedan od većih izazova ove arhitekture.
- Refaktoriranje – kod ove arhitekture teže je provoditi refaktoriranje ako je potrebno seliti funkcionalnost između mikroservisa zbog jake modularizacije. Isto tako je otežano mijenjati tu modularizaciju koja se temelji na mikroservisima, no ovi problemi se mogu premostiti nekim pristupima koji se neće ovdje spominjati.
- Važnost arhitekture domene – modularizacija u mikroservise se odvija na temelju arhitekture domene što pak određuje podjelu na timove. Timovi ne bi trebali ovisiti jedan o drugome, kao ni mikroservisi, te stoga jedino dobro definirana arhitektura domena omogućuje neovisan razvoj mikroservisa. Već je spomenuto da je teško promijeniti modularizaciju pa se greške kasnije teško ispravljaju te je potrebno dobro odrediti granice servisa i domena.
- Izvođenje mikroservisa – sustav koji je načinjen od mikroservisa ima mnogo komponenata koje trebaju biti isporučene i upravljane zajedno te se trebaju izvoditi zajedno. Kako bi se smanjila kompleksnost operacija i broj infrastruktura koje sustav koristi, potrebno je operacije automatizirati i tako osigurati da izvođenje mikroservisa ne bude opterećujuće.
- Kompleksnost distribuiranih sustava – sustav s mikroservisnom arhitekturom je distribuiran sustav pa je tako naslijedio izazove i mane takvih sustava. Naime, pozivi među mikroservisima se mogu ne izvršiti zbog problema na mreži i sporiji su od poziva koji se događaju unutar istog procesa.

3.4. Usporedba s ostalim arhitekturama

Mikroservisna arhitektura predstavlja novi pristup izgradnji distribuiranih sustava te je proizašla iz servisno-orijentirane arhitekture. Više je preskriptivna od SOA arhitekture te neke funkcionalnosti koje su tipične u SOA arhitekturi predstavljaju anti-uzorke u mikroservisnoj arhitekturi. [7] Kako bi se ove dvije arhitekture mogle bolje usporediti, **SOA arhitektura** će biti pobliže opisana u nastavku.

Servisno-orijentirana arhitektura može se opisati sljedećim definicijama [4]:

- Skup servisa i poslovnih funkcionalnosti koji se pruža kupcima, partnerima ili drugim dijelovima organizacije.
- Arhitekturalni stil kojeg karakteriziraju pružatelj servisa, korisnik servisa, opis servisa koji se pružaju te moguće posrednike.
- Skup principa, uzoraka i kriterija u arhitekturi aplikacije koji određuju karakteristike kao što su modularizacija, enkapsulacija, ponovna iskoristivost, slabo povezivanje, i tako dalje.
- Model koji se koristi u programiranju, a koji uključuje standarde, alate i tehnologije te podržava web, REST i ostale vrste servisa.
- Posrednički sloj (eng. *middleware*) koji je optimiran za sastavljanje, orkestraciju, nadzor i upravljanje servisima.

Iz navedenih definicija može se zaključiti da SOA nastoji riješiti veći set problema i servise staviti na raspolaganje svima koji ih žele koristiti. S druge strane, mikroservisi su limitirani na manje opsežan i fokusirani cilj koji je dio većeg distribuiranog sustava. Kako je distribuirani sustav često kreiran dekompozicijom monolita, mikroservisi rade zajedno kao jedna aplikacija te ne opslužuju više sustava odjednom. Nadalje, mikroservisi ne zahtijevaju opise jer su dobro poznati korisnicima, ne otkrivaju se prilikom vremena izvođenja i ne zahtijevaju posrednika. Neki mikroservisni sustavi nude otkrivanje servisa kako bi bili više fleksibilni, no taj arhitekturalni uzorak nije obavezan. Navedeno je omogućeno pomoću konfiguracijskih datoteka koje omogućuju da su mikroservisi svjesni jedni drugih. Prilikom vremena razvoja, timova mogu otkriti dostupne servise pomoću kataloga, registra ili alata za upravljanje API-jem. [4]

Kako je mikroservisna arhitektura specijalizacija SOA-e, granice funkcionalnih područja koriste se za definiciju modela domene koji se dodjeljuju timovima. Timovi pak mogu odabrati svoj programski jezik i okvir te detalje isporuke. Za razliku od SOA-e, isporuka, orkestracija, nadzor i upravljanje servisima predstavljaju glavni zahtjev i problem kod mikroservisne arhitekture. Nadalje, SOA arhitektura je više fokusirana na protokole i formate kao što su SOAP, WSDL i WS-obitelji protokola koji se smatraju kompliciranjima i težima, dok se mikroservisi oslanjaju na REST, HTTP i ostale formate koji se smatraju lakšima i nativnima za web razvoj. [3] [4]

Još jedna razlika između tih dviju arhitektura tiče se poslovne i tehničke motivacije. Naime, kod mikroservisa, povrat ulaganja (eng. *Return on investmen*t, ROI) je vođen bržim realizacijama promjena i funkcionalnosti, a ne cjelokupnom transformacijom poslovanja.

Tome je tako jer je manje vjerojatno da poduzeća investiraju u velike i duge transformacijske inicijative. [4]

Može se zaključiti da su fokus i opseg problema dvije glavne razlike između arhitektura. Servisno-orijentirana arhitektura nastoji riješiti veći opseg problema i nije fokusirana na samo jedan cilj. S druge strane, mikroservisi namjerno žele postići manje i fokusirati se na manji problem. Potreba za takvim novim pristupom se javila upravo zbog toga što su servisno-orijentirani sustavi vremenom postajali sve kompleksniji. Zahtijevali su sve kompleksnije alate i apstrakciju, jer su rješavali prevelik opseg problema bez upotrebe najboljih praksi. Zatim je takav velik monolit inkrementalno evoluirao u distribuirani sustav mikroservisa kojeg se lakše kontrolira i isporučuje kroz korištenje praksi kontinuirane integracije. [4]

Što se tiče ponovne iskoristivosti, ona se ne događa na servisnoj razini kod mikroservisa jer su oni dizajnirani tako da ih pokreće jedna glavna aplikacija. Te mikroservise onda mogu ponovno koristiti druge aplikacije. Dakle, mikroservisi su dizajnirani kako bi se sustav mogao skalirati na webu, kako bi se omogućio kontinuirani razvoja te timovi bili što više efikasniji. Nadalje, mikroservisi pružaju eksperimentalnu poslovnu transformaciju i manje troškove, dok SOA zahtijeva ozbiljnije i unaprijed definirane financijske i arhitekturalne planove. [4]

Zaključak navedenog je da se oba dvije arhitekture servisne jer oba dvije pružaju servise koji komuniciraju preko mreže. Razlika se očituje u tome da je kod SOA arhitekture naglasak na otkrivanju servisa i ponovnoj iskoristivosti, dok mikroservisi teže sustavu kojim se lako upravlja i koji se postupno razvija.

4. Karakteristike arhitekture

U ovom poglavlju će biti opisane ključne karakteristike i principi mikroservisne arhitekture koji objašnjavaju njezinu popularnost. Karakteristike mikroservisa se velikim dijelom poklapaju sa prednostima te arhitekture. Prilikom dizajna i razvoja važno je pažnju obratiti na mnoga pitanja, a u tome pomaže jezik uzoraka arhitekture. Jezik uzorka predstavlja set uzoraka koji se koriste prilikom implementacije mikroservisne arhitekture te ima 2 svrhe:

- Pomaže donijeti odluku o tome da li je mikroservisna arhitektura prikladna za aplikaciju
- Omogućuje ispravno korištenje i implementaciju mikroservisa

Kroz opise karakteristika će biti spomenuti uobičajeni uzorci koji se koriste kod razvoja aplikacija sa mikroservisnom arhitekturom.

Orijentirana na poslovanje

Mnoge mikroservisne aplikacije nastale su tako da se monolit 'rastavi' na manje dijelove ili servise. Međutim, tada se javlja rizik da će dekompozicija sustava biti odrađena prema postojećim granicama unutar organizacije pa može doći do toga da postoji puno više dijelova kojima je potrebno upravljati ili da nije dobro odrađena njihova integracija. Loše odrađena dekompozicija čini sustav ranjivim, a postoji mogućnosti da se razvije sustav čije servise nije moguće ponovno koristiti. Ako su servisi dizajnirani prema organizacijskim ili tehnološkim granicama, to može dovesti do povećanih troškova razvoja i održavanja. [4]

Iznimno je važno dizajnirati mikroservise imajući na umu poslovne ciljeve. To može značiti i da se, umjesto korištenja mikroservisa za usmjeravanje komunikacije među timovima, koristi više timova koji se okupljaju i zajedno rade na dizajnu servisa. [4]

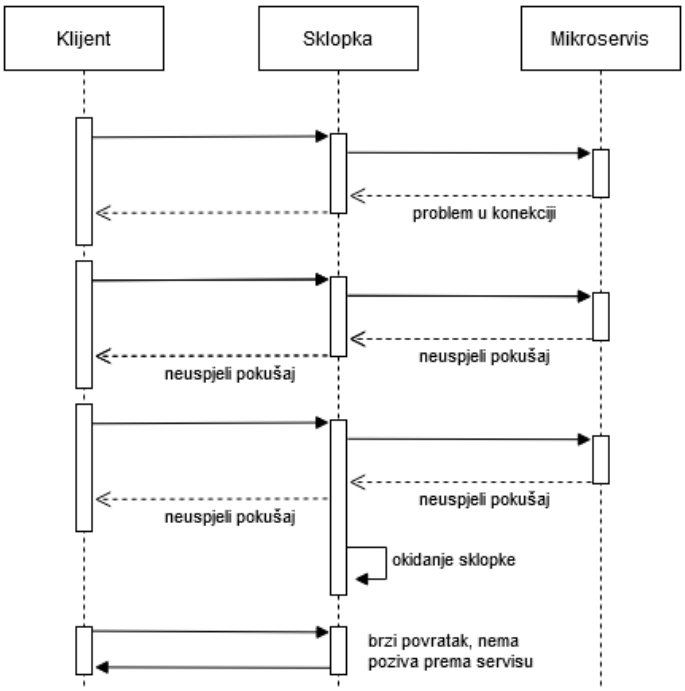
S druge strane, ako se aplikacija gradi koristeći mikroservisnu arhitekturu, tada se timovi okupljaju oko granica servisa, odnosno, domene zadatka. Drugim riječima, servisi se organiziraju oko poslovnih sposobnosti. Ovakva organizacija timova bolje ukazuje na veličinu i opseg projekta te utječe na razne dimenzije razvoja softvera jer timovi okupljaju članove koji zajedno pokrivaju sve aspekte arhitekture. Koristi se opsežan stog tehnologija za poslovno područje, a to uključuje korisničko sučelje, trajnu pohranu, poslovnu logiku te bilo kakvu suradnju s ostalim dijelovima. [4]

Dizajn za neuspjeh

Dizajn za neuspjeh ukazuje na to da se prihvaća činjenica da su greške neizbježne, ali cilj je održati sustav funkcionalnim što je duže moguće. To pozitivno utječe na dizajn jer se razvijaju pouzdaniji sustavi. Takvi stabilni sustavi koriste neke od uzoraka dizajn upravo za tu namjenu, a ukratko će biti opisana sljedeća dva: uzorak sklopke/prekidača (eng. *circuit breaker*) te uzorak pregrada (eng. *bulkheads*). [4]

Uzorak sklopke osigurava da greška u nekom servisu ne utječe na cijeli sustav. Radi kao i prekidač u električnom sustavu. Pozivi prema mikroservisu su „umotani“ u objekt koji se ponaša kao sklopka. Kada servis padne, taj objekt omogućuje pozive prema servisu sve dok se ne dosegne određeni broj neuspjelih pokušaja. Tada se prekidač za taj servis okida i svi daljnji pozivi neće rezultirati pozivima prema servisu koji je pao. Ovaj uzorak dizajna osigurava stabilnosti sustava te štedi resurse jer se ne izvršavaju nepotrebni pozivi koji će ionako biti neuspjeli. Kad se prekidač okine i strujni krug je otvoren, može se pokrenuti povratna logika koja ima malo ili nimalo obrade i vraća vrijednost. Kako se ona pokreće kao rezultat neuspjeha, mora imati malu vjerojatnost za neuspjeh. [4]

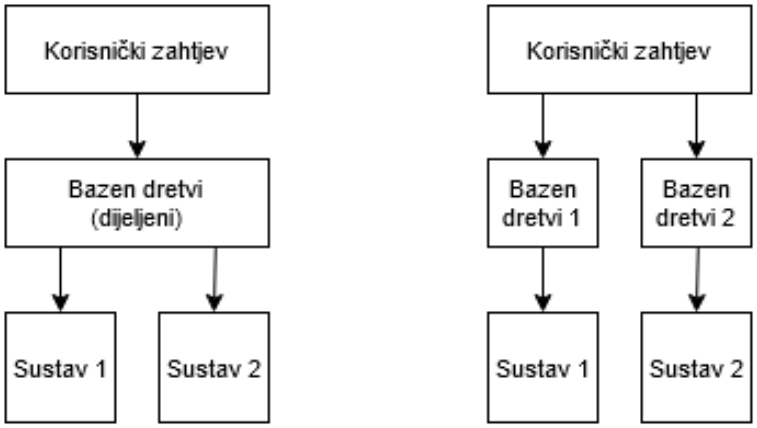
Na slici 4 je prikazan dijagram slijeda za prethodno opisan uzorak. Klijent šalje poziv objektu koji predstavlja sklopku, koji zatim prosljeđuje pozive servisu. Nakon problema u vezi prema servisu, izvrše se još dva poziva koja su neuspješna te se okine sklopka te objekt više ne prosljeđuje pozive prema servisu koji je nedostupan.



Slika 4 Dijagram slijeda za uzorak sklopke (prema [4], str 22)

Uzorak pregrada se koristi kao pristup odjeljivanja/izoliranja grešaka na male dijelove sustava. U mikroservisnoj arhitekturi takve pregrade prirodno čine sami mikroservisi i njihove granice. Podjela funkcionalnosti na zasebne mikroservise omogućuje izoliranje greške i njezinih posljedica na samo jedan mikroservis. [4]

Osim izoliranja svakog mikroservisa, ovaj uzorak se može primijeniti i unutar servisa. Primjer za to je korištenje više bazena dretvi kako bi se omogućilo izoliranje greške na samo jedan bazen ukoliko do greške dođe [4]. Navedeno je prikazano slikom 5 gdje se u prvom sustavu koristi dijeljeni bazen te moguće greške ne bi bile izolirane. S druge strane, u drugom sustavu se koriste dva bazena te, primjerice, zastoji u prvom bazenu neće utjecati na drugi bazen i ostatak sustava.



Slika 5 Uzorak pregrada na primjeru dretvi (prema [4])

Decentralizirano upravljanje podacima

Upravljanje podacima kod monolita je mnogo lakše jer su sve komponente dio tog monolita. Distribuirana mikroservisna arhitektura mora se nositi s podacima koji su rašireni kroz više servisa što čini pristup podacima uvelike kompleksnijim. Monoliti uobičajeno imaju jednu relacijsku bazu podataka nad kojom se vrše ACID transakcije koje osiguravaju atomarnost (nedjeljivost, eng. *atomicity*), konzistentnost (eng. *consistency*), izolaciju (eng. *isolation*) te izdržljivost (eng. *durability*). S druge strane, u mikroservisnoj arhitekturi se preferiraju BASE svojstva transakcija [4]:

- U osnovi dostupno (eng. *basically available*) – podaci su dostupni čak i kad postoje višestruki neuspjesi zbog visokog stupnja replikacije podataka.
- Meko stanje (eng. *soft state*) – označava da je dosljednost podataka odgovornost programera, odnosno, dozvoljene su privremene nekonzistentnosti i mijenjanje podataka koji su trenutno u upotrebi.

- Konzistentnost u konačnici (eng. *eventual consistency*) – svojstvo koje zahtijeva da u nekom trenutku u budućnosti podaci dođu u dosljedno stanje.

Nadalje, izbjegavaju se distribuirane transakcije kad god je to moguće. Najčešće svaki mikroservisi upravlja vlastitom bazom podataka što pak omogućuje poliglotsku perzistenciju podataka. Drugim riječima omogućuje korištenje različitih baza podataka i spremišta. Postoje slučajevi i kad više mikroservisa treba koristiti istu bazu podataka (primjerice, kad je potrebno osigurati ACID svojstva). No, tada se to treba implementirati s oprezom i valjanim razlogom jer dijeljenje baze krši principe arhitekture: prelaze se granice servisa te se promjene u bazi trebaju usklađivati između njih. Jedna od glavnih značajki mikroservisne arhitekture je slaba povezivost pa mikroservisi trebaju dijeliti što manje resursa. [4]

Mogućnost otkrivanja

Ova karakteristika ukazuje na to da se mikroservisi grade na način da ukoliko dođe do kreiranja ili uništavanje osnovne infrastrukture, oni mogu biti rekonfigurirani s lokacijom drugih servisa na koje se spajaju. To posebno dolazi do izražaja kod računalstva u oblaku i korištenja kontejnera jer se takvi servisi trebaju dinamički rekonfigurirati. Na taj način svi u mreži mogu otkriti novokreiranu instancu servisa i komunicirati s njom. [4]

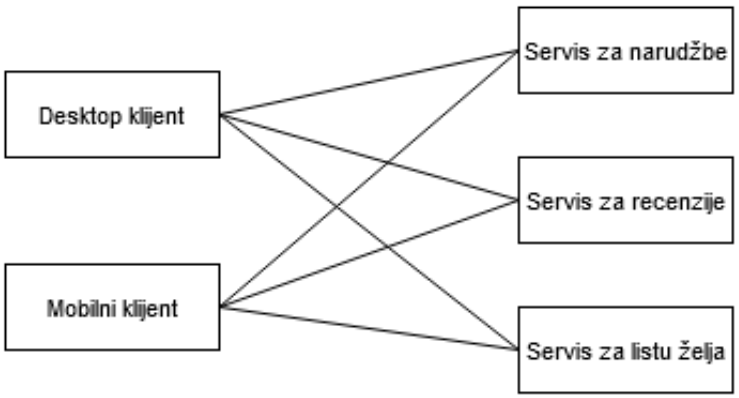
Važan pojam koji se tu javlja je registar servisa. Većina modela za otkrivanje servisa se oslanjaju upravo na registar na kojeg se registriraju svi servisi. Servisi nakon registracije obavljaju pozive prema registru kako bi komunicirali sa servisima o kojima ovise. Drugi modeli uključuju oglašavanje servisa i slično. Otkrivajući servisi obično sadrže mnogo metapodataka koje registrirajući servisi mogu pružati. To su primjerice neke ovisnosti, specifični zahtjevi i drugo. Nadalje, otkrivajući servisi mogu imati i mogućnost balansiranja opterećenja. Kad su previše opterećeni, imaju mogućnost preusmjeravanja zahtjeva na druge instance servisa. [4]

Dizajn komunikacije među servisima

Mikroservisi mogu biti isporučeni na razne platforme na kojima će se izvršavati: poslužitelji, virtualne mašine, kontejneri. Svaki od njih je neovisan o drugima i upravo zbog toga treba implementirati neki oblik komunikacije među njima jer će u većini slučajeva trebati usluge od drugi servisa. Osim komunikacije među servisima, potrebno je misliti i na komunikaciju s klijentima.

Jedan od načina na koji klijenti (primjerice, web UI ili mobilna aplikacija) dobivaju podatke je preko RESTful API poziva. U tom slučaju svaki klijent šalje pozive prema servisu koji mu zatreba. Komunikacija putem direktnih API poziva je prikazana slikom 6 i prikazuje

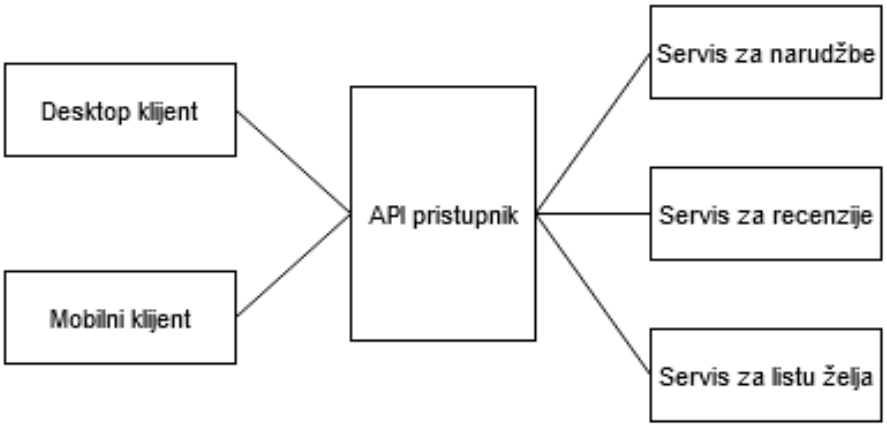
kako klijenti moraju znati za svaki servis te mu slati pozive. To ujedno predstavlja i najlakši način, ali ima i neke nedostatke. Naime, klijenti mogu zatrpavati servise pozivima kako bi dobili potrebne podatke. Isto tako treba uzeti u obzir različita skaliranja servisa kao i rukovanje greškama koje se mogu dogoditi. [4]



Slika 6 Direktni API pozivi prema servisima (prema [4])

Kako bi se takvi problem izbjegli, bolje je koristiti pristupnu točku ili takozvani API gateway. API pristupnik se nalazi između klijenta i mikroservisa te pruža API-je koji su prilagođeni klijentima. Dakle, služi kao fasada za skrivanje složenosti tehnologije pri čemu pruža dijelovima/komponentama aplikacije jedinstven pogled vanjskih resursa. Sadrži pojednostavljeno sučelje koje olakšava korištenje i testiranje servisa – slično kao kod facade uzorka dizajna. Nadalje, može pružati različite razine granularnosti različitim klijentima. Primjerice, mobilnim klijentima su namijenjena krupnozrnata API sučelja, dok one sitnozrnate koriste desktop klijenti koji koriste mrežu visokih performansi. Na taj način je omogućeno spajanje više poziva u jedan koji je optimiziran za klijenta. Kao posljedica, klijent osjeti kašnjenje mreže samo jednom ili manji broj puta te iskorištava serversku snagu i nisku latenciju veze.[4]

Komunikacija preko API pristupnika je prikazana na slici ispod. Svi pozivi se odvijaju preko tog pristupnika koji može različitim klijentima pružati različite usluge te šalje pozive odgovarajućim mikroservisima.



Slika 7 Korištenje pristupnika (prema [4])

Što se tiče međuprocesne komunikacije, ona je ostvariva na sljedeća dva načina:

- sinkrone HTTP poruke kao što su REST, SOAP i websocketi – karakterizira ih komunikacijski kanal između poslužitelja i klijenta koji se temelji na zahtjevima i odgovorima, odnosno, vođen je događajima
- asinkrone poruke – ovakva komunikacija uključuje korištenje posrednika za poruke

Sinkrono slanje poruka jednostavan je i često korišten mehanizam, međutim, ne podržava ostale uzorke kao što su model pretplate i redovi poruka. Model objave i pretplate (eng. *publish/subscribe*) olakšava dodavanje mikroservisa u aplikaciju jer je više fleksibilan i omogućuje da se više pretplatnika pretplati na istu temu, odnosno, na isti protok poruka. Kod sinkrone komunikacije i klijent i poslužitelj moraju istovremeno biti dostupni, a klijent mora znati adresu i port poslužitelja. [4]

Asinkrono slanje poruke odvaja servise jedne od drugih u vremenu. Kod takvog komuniciranja, dretva pošiljatelj može nastaviti s poslom dok sustav za poruke obavlja daljnje aktivnosti potrebne za slanje poruke na destinaciju. Asinkrono slanje omogućuje i korištenje posrednika/brokera za poruke pri čemu se razlikuju proizvođači i potrošači poruka. Broker omogućuje spremanje poruka u spremnik (eng. *buffer*) sve dok ih potrošač ne konzumira i obradi. [4]

Vrste komunikacije se međusobno ne isključuju te nije potrebno odabrati samo jedan tip poruka. Većina aplikacija koristi kombinaciju nekih od pristupa slanju poruka ovisno o potrebama.

Rad s kompleksnošću

Kako se sustav s mikroservisnom arhitekturom sastoji od puno dijelova te koristi dodatne resurse (na primjer, više baza podataka), on je puno kompleksniji. Dok je ponašanje svakog pojedinog dijela jednostavno i lako ga je optimizirati, ponašanje cijelog sustava je kompleksno i teže razumljivo.

Potrebna je infrastruktura koja će osigurati dobar nadzor nad sustavom i operacijama. Već je spomenuto da treba postojati visoka razina automatizacije isporuke kako bi se mikroservisi premjestili iz razvoja u produkciju. Prilikom tog procesa važno je uzeti u obzir sljedeće [4]:

- Testiranje – potrebni su mnogobrojni jedinični i testovi integracije, test opterećenja, simuliranje grešaka u komponentama, ponašanje sustava u cjelini i tako dalje. Pri tom je dobro koristiti softverski okvir (eng. *software framework*) koji podržava navedeno te osigurava točnost i pouzdanost sustava. Takvi okviri se često mogu koristiti i za nadzor sustava.
- Nadzor – nadzor igra važnu ulogu kod mikroservisa te je dobro dizajnirati servise tako da pružaju informacije o provjeri komponenti pomoću principa pretplate na poruke. Na taj način servis zadužen za nadzor skuplja informacije koje se mogu iskoristiti za razne provjere: ima li neki servis dovoljan kapacitet za obradu zahtjeva koji pristižu, zadovoljava li onu razinu performansi koja je određena očekivanjima i slično.
- DevOps – ovakva praksa kod izvođenja operacija osigurava da se mikroservisi stalno izvode i da funkcioniranju. Za tako nešto je potrebno DevOps vještine ugraditi u cijeli tim te se, primjerice, testiranje odvija automatizirano.
- Poliglotsko programiranje – svaki mikroservis može biti napisan u različitom programskom jeziku što predstavlja dodatnu razinu kompleksnosti za održavanje takvog sustava.

Evolucijski dizajn

Ovakav pristup dizajnu osigurava da se nove funkcionalnosti i mogućnosti mogu dodavati izmjenjujući jedan mikroservis. Samo promijenjen mikroservis se isporučuje što znatno ubrzava taj proces, a promjena utječe samo na klijente tog servisa. Ukoliko sučelje servisa nije mijenjano, klijenti se mogu dalje neometano izvoditi.

Upravo ta lakoća ažuriranja mikroservisa omogućuju evoluciju, a te promjene mogu biti brze, česte i propagiraju se s minimalnim rizikom. Nadalje, česta je situacija da se čitav

mikroservis iznova prepiše umjesto da se održava stariji kod ili nadograđuje jer je on neovisan, malen i obično je za njega zadužen manji tim. Uglavnom vrijedi da što je više mikroservisa u dekompoziciji monolita, to su oni i manji te predstavljaju manji rizik prilikom isporuke promjena. No to povlači sljedeće: ako su mikroservisi sitno dekomponirani, neki zahtjev može zahtijevati promjene na više mikroservisa. Posljedično, više mikroservisa treba biti izgrađeno i isporučeno te je rizik veći. [4]

5. Dizajn i implementacija mikroservisa

Prilikom implementacije važno je pažnju obratiti na brojna pitanja te će ovdje biti predstavljena ona najznačajnija. Poglavlje se bavi dizajnom i implementacijom mikroservisa te evaluacijom učinka tog pristupa na organizaciju.

5.1. Opseg i identifikacija mikroservisa

Za određivanje opsega i identificiranje mikroservisa, korisno je provesti proces dizajnerskog promišljanja (eng. design thinking). Proces se fokusira na predviđanje cjelokupnog korisničkog iskustva, a ne na funkcionalnost. Potrebno je odrediti što korisnici misle, rade i osjećaju prilikom korištenje proizvoda, a to pomaže raspoređivanju rada na izgradive i upotrebljive funkcionalne jedinice. Drugim riječima, omogućuju dekompoziciju i identifikaciju mikroservisa. Dizajnersko promišljanje se sastoji od nekoliko koncepata koji će biti ukratko opisani [4]:

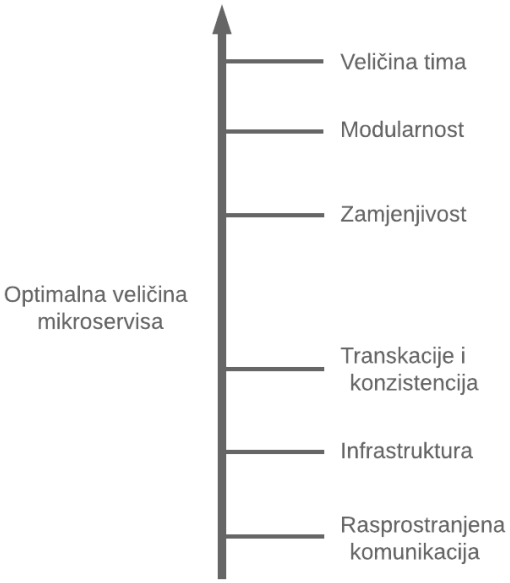
- Hills – ovaj koncept određuje poslovni cilj koji treba biti postignut u vremenu. Pritom se definira tko, što i kako će se zahtjev postići. Ova definicija bi trebala biti mjerljiva i usmjerena na korisnika te se obično određuje nekoliko Hills definicija po projektu.
- Hills Playback – sadrži sažetak onog što se želi postići, a određuju se vremenski period i poslovi. Nakon toga se odvijaju reprodukcije (playbacks) na tjednoj bazi kod kojih timovi pokušavaju izvršiti Hills definicije.
- Scenarij – predstavlja tijek rada kroz korisničko iskustvo. Prema scenariju se postavlja kontekst u Hills Playback reprodukciji.
- Korisnička priča – predstavlja zahtjev koji se može biti napisan u kodu u kraćem vremenskom periodu (1-2 dana).
- Epovi – grupiraju korisničke priče i mogu biti ponovno korišteni kroz scenarij.
- Ciljni korisnici – angažirani su na projektu i reprezentiraju ciljanu publiku. Sudjeluju u reprodukcijama ili ih vode.
- Identificiranje mogućnosti – odnosi se na identificiranje mogućnosti za ponovnu iskoristivost servisa za neki drugi dizajn.

5.2. Veličina mikroservisa

Prilikom dizajniranja mikroservisa, veliku ulogu igra njihova veličina. Na nju utječu razni faktori [5]:

- Gornju granicu postavlja veličina tima – Za mikroservis ne bi trebao biti zadužen veliki tim niti više manjih jer bi timovi trebali raditi neovisno, te isto tako i isporučivati servise. Takva neovisnost se jedino postiže ako tim radi odvojen mikroservis koji nije prevelik. Zbog toga, jedan tim može raditi i na više mikroservisa, dok obratno nije poželjno.
- Modularizacija također ograničava veličinu – Mikroservis bi trebao biti takav da programer razumije cijeli servis i dalje da razvija. Ova granica se nalazi ispod veličine tima jer ono što jedan razvojni programer razumije, tim će moći razvijati dalje.
- Veličinu mikroservisa smanjuje i zamjenjivost – uvjet za zamjenjivost mikroservisa je da programer razumije taj mikroservis i njegove aspekte pa se ova granica nalazi ispod one za modularizaciju.
- Infrastruktura postavlja donju granicu veličine – ako je preteško isporučiti infrastrukturu za svaki pojedini mikroservis, onda broj mikroservisa mora biti manji, što pak utječe na njihovu veličinu, to jest, sami mikroservisi će biti veći.
- Donju granicu postavlja i rasprostranjena komunikacija – Ona se povećava s brojem mikroservisa, stoga mikroservisi ne bi trebali biti mali i brojni jer će to uzrokovati mnogo međusobnih poziva koji su nepotrebni (eng. *Communication Overhead*).
- S donje strane, veličinu ograničavaju i konzistencija podataka te transakcije – Konzistencija i transakcije mogu biti osigurane jedino unutar jednog mikroservisa što znači da mikroservisi ne smiju biti toliko mali da se konzistencija podataka i transakcije osiguravaju kroz više mikroservisa.

Slika 8 na skali prikazuje faktore koji utječu na veličinu mikroservisa te njihove granice. Može se primijetiti da faktori koji utječu na veličinu mikroservisa ocrtavaju i samu ideju mikroservisa. Ta ideja i glavne prednosti mikroservisa pred drugim arhitekturama su neovisan rad više timova i neovisna isporuka pri čemu je bitna zamjenjivost mikroservisa. Iz tih željenih značajki može se odrediti optimalna veličina mikroservisa.



Slika 8 Faktori koji utječu na veličinu mikroservisa (prema [5])

Osim navedenih, postoje i drugi razlozi za odabir mikroservisne arhitekture, kao na primjer skaliranje. Ono također utječe na veličinu mikroservisa jer veličina mora biti takva da osigurava da je mikroservis cjelina koja se neovisno skalira. Dakle, na veličinu ne utječu samo navedeni faktori već i mnogi drugi. Tu je također i tehnologija koja se koristi, a o kojoj ovise komunikacija i napor da se osigura potrebna infrastruktura za mikroservise. Iz svega navedenog se može zaključiti da ne postoji idealna veličina mikroservisa te da će stvarna veličina ovisiti o raznima faktorima, tehnologiji i slučajevima korištenja svakog pojedinog mikroservisa. [5]

5.3. Conwayov zakon i arhitektura mikroservisa

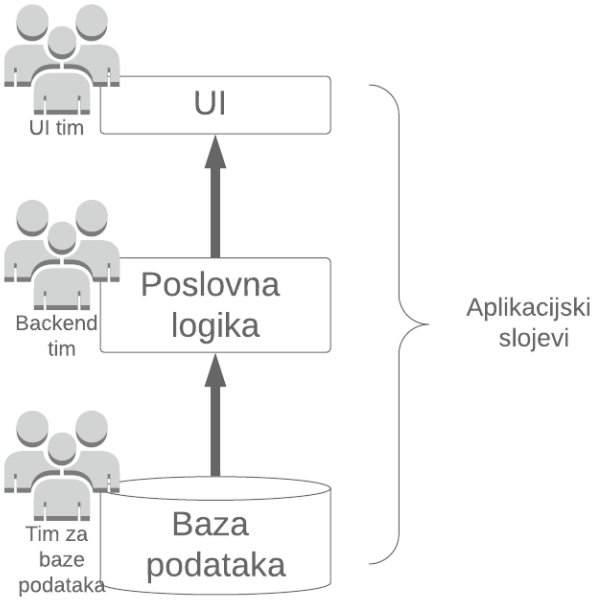
Pojam koji se često spominje u literaturi prilikom dizajniranja arhitekture softvera je Conwayov zakon (eng. *Conway's Law*). On naznačuje da će svaka organizacija koja dizajnira sustav proizvesti takav dizajn da njegova struktura odgovara strukturi komunikacije u organizaciji. Važno je naglasiti da se ovaj zakon odnosi na bilo koji dizajn, a ne samo na dizajn softvera. Pritom, komunikacijske strukture ne moraju biti identične organizacijskoj shemi jer često postoje neformalne komunikacijske strukture ili na njih može utjecati geografska lokacija timova i slično. [5]

Ovaj zakon proizlazi iz činjenice da svaka organizacijska jedinica dizajnira određeni dio arhitekture. Ako dva dijela arhitekture imaju sučelje, potrebna je koordinacija u vezi tog sučelja što za posljedicu ima to da je potrebna i komunikacijska veza između organizacijskih jedinica koje dizajniraju te dijelove. Nadalje, Conwayov zakon utječe i na modularizaciju

dizajna. Ako je svaka organizacijska jedinica zadužena za određeni dio arhitekture, tada je moguće osigurati da se članovi tima ne moraju stalno međusobno koordinati. Programeri koji rade na istom modulu se često koordiniraju, dok se s drugim članovima (koji rade na drugim modulima) moraju sinkronizirati tek kada se razvija sučelje. [5]

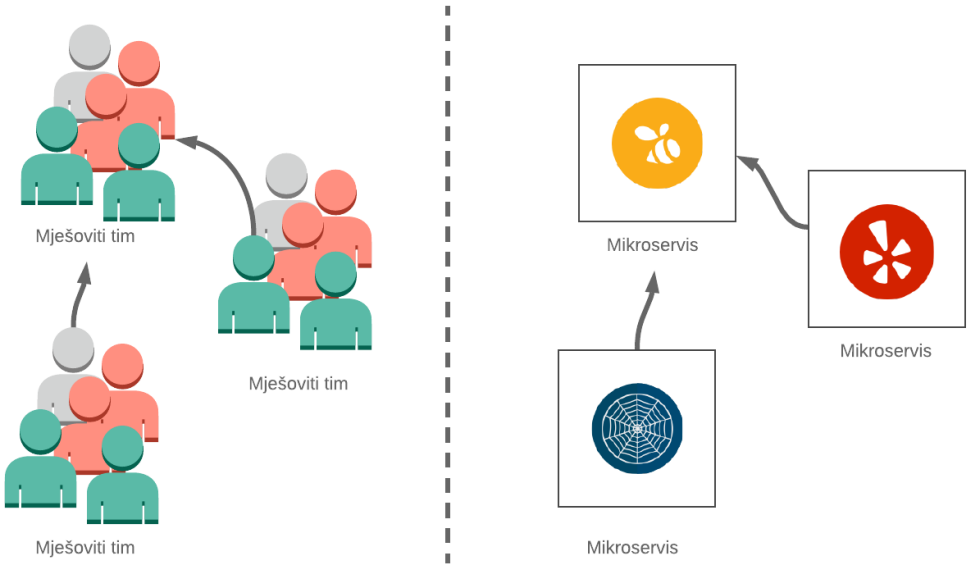
Spomenuto je da na komunikacijsku strukturu organizacije utječe i geografska razmještenost timova i članova tima. Jasno je da je lakše surađivati s timom ako se on nalazi na istoj lokaciji, za razliku od tima koji je u drugom gradu ili državi. Posljedično, arhitekturalni dijelovi koji imaju brojne komunikacijske veze su bolje implementirani ako su timovi bliže jedni drugima jer lakše međusobno komuniciraju. Dakle, zakon nije određen samo organizacijskom shemom već i stvarnim komunikacijskim vezama. Prilikom dizajniranja arhitektura treba biti oprezan ako je organizacija velika ili ako postoje mnogobrojne komunikacijske veze jer to može loše utjecati na samu arhitekturu što predstavlja veliki rizik za projekt i njegov razvoj. [5]

Conwayov zakon predstavlja ograničenje sa perspektive razvoja softvera. Naime, ako se aplikacija modularizira na temelju tehnoloških aspekata, javljaju se 3 vrste tima. Jedna vrsta su programeri koji razvijaju korisničko sučelje (eng. *User Interface*, kraće UI), drugoj vrsti pripadaju programeri koji su zaduženi za pozadinski dio (eng. *backend*), a treća vrsta okuplja stručnjak za rad s bazama podataka. Navedeno omogućava da svaka tehnologija ima stručnjake u svom području i lako se kreira takva organizacija. Na taj način se timovi međusobno lako podupiru te je olakšana tehnička razmjena. Ovakva distribucija timova kreira 3 tehnička sloja: korisničko sučelje, poslovnu logiku i bazu podataka što znači da odgovara organizaciji, to jest, organizacijskoj shemi. Ovakva slojevitost aplikacije i odgovarajuća joj organizacijska shema timova je prikazana na slici 9. Međutim, takva distribucija ima velik nedostatak da razvoj tipične funkcionalnosti zahtijeva promjene na sva 3 tehnička sloja: korisničko sučelje treba prikazati nove funkcionalnosti korisnicima, poslovna logika treba biti implementirana te je potrebno kreirati odgovarajuće strukture u bazi. Navedeno rezultira velikim brojem ovisnosti te komunikacijom i koordinacijom koje direktno ne pridonose korisnom radu. Tako osoba koja želi implementirati novu funkcionalnost mora komunicirati sa sva 3 tima, a timovi moraju koordinirati rad i kreirati nova sučelja. Pritom poslovna logika ne može raditi ako nema podataka u bazi ili pak korisničko sučelje ne može raditi ako nije implementirana poslovna logika. To uzrokuje i vremenska kašnjenja ako timovi rade u takozvanim sprintevima. Potrebno je provesti njih 3, ako svaki tim radi u jednom sprintu, da bi se implementirala jedna funkcionalnost. Jasno je da takva kašnjenja nemaju smisla ako je glavni cilj implementirati i isporučiti nove funkcionalnosti što brže. [5] [9]



Slika 9 Conwayov zakon i 3-slojna arhitektura [autorski rad]

S druge strane, ako se aplikacija gradi koristeći mikroservisnu arhitekturu, tada se timovi okupljaju oko granica servisa, odnosno, domene zadatka. Drugim riječima, servisi se organiziraju oko poslovnih sposobnosti. Ovakva organizacija timova bolje ukazuje na veličinu i opseg projekta te utječe na razne dimenzije razvoja softvera jer timovi okupljaju članove koji zajedno pokrivaju sve aspekte arhitekture. Koristi se opsežan stog tehnologija za poslovno područje, a to uključuje korisničko sučelje, trajnu pohranu, poslovnu logiku te bilo kakvu suradnju s ostalim dijelovima. Takvi timovi su multifunkcionalni i uključuju brojne vještine potrebne za razvoj softvera: korisničko iskustvo (eng. *User eXperience*, kraće UX), baze podataka i upravljanje projektom. Što su granice servisa eksplicitnije određene, to su jasnije strukture timova, i obrnuto. Na slici 10 je prikazana vezana između Conwayovog zakona i mikroservisne arhitekture, odnosno, ukoliko se servisi organiziraju oko poslovnih mogućnosti, tada se dobivaju mješoviti timovi. [8] [9]



Slika 10 Conwayov zakon i mikroservisi [autorski rad]

5.4. Odabir implementacijskog stoga

Mikroservisi nekog sustava se izvršavaju kao zasebni procesi te se za komunikaciju koriste tehnologije koje podržavaju uobičajene komunikacijske protokole, kao što su HTTP REST ili MQTT (MQ Telemetry Transport). Prilikom tog izbora, potrebno je uzeti u obzir i tip komunikacije – asinkrono ili sinkrono. Neke tehnologije (primjerice Java) rade sa sinkronom komunikacijom koja blokira mrežne zahtjeve pa se oni moraju izvršavati u posebnim dretvama kako bi se postigla konkurentnost. [4]

Važno je razmotriti i zahtjeve za memorijom kao i procesorskom snagom. Nikad se ne pokreće samo jedan mikroservis, već više njih te se svaki od njih dalje skalira prema potrebi. Zbog toga je bitno ocijeniti performanse cijelog sustava jer izvršava mnoge procese. [4]

Iako je arhitekturalno moguće da svaki mikroservis bude razvijan u različitoj tehnologiji, to u većini vremena nije slučaj niti je poželjno. Tome je tako jer broj različitih tehnologija ograničavaju vještine programera, poželjna ponovna iskoristivost koda i ostalo. Bez obzira na to, korištenje mikroservisne arhitekture uvijek dopušta isprobavanje novih tehnologija s minimalnim rizikom. [4]

5.5. REST API i poruke

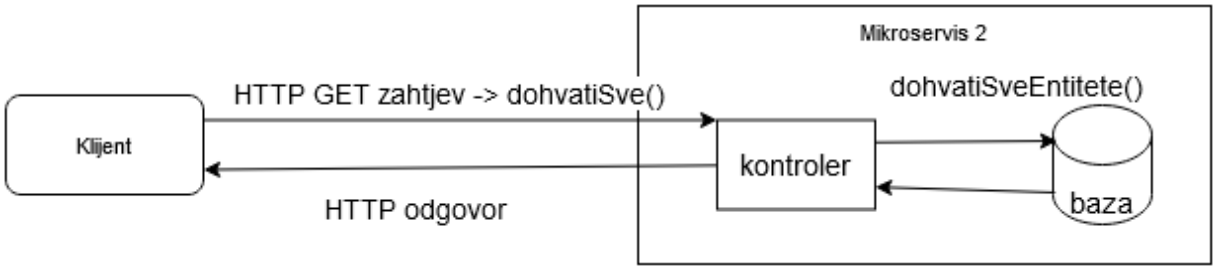
Ovo poglavlje obrađuje REST API i razmjenu poruka kod aplikacija koje koriste mikroservisni stil.

RESTful servis je onaj servis koji se bazira na REST-u, a **REST** predstavlja arhitekturalni pristup za umrežene aplikacije koje razvijaju lagane, skalabilne servis koje je lako održavati. RESTful servisi uglavnom koriste HTTP protokol za komunikaciju iako REST ne ovisi o tom protokolu (ni o bilo kojem drugom). Jedan od razloga zašto je to tako je taj što podržava velik broj alata i tehnologija kao što su spremanje u predmemoriju, autentikacija, certifikati i tako dalje. Glavna svrha REST-a je omogućiti komunikaciju među resursima kroz razmjenu informacija. Danas je prevladavajući model dizajna web servisa stoga ne čudi da skoro svaki popularni programski jezik sadržava programski okvir za razvoj RESTful web servisa. [1] [4]

REST API koristi HTTP metode kako bi pozivao resurse te uspostavlja mapiranje između CREATE, READ, UPDATE, DELETE operacija za manipulaciju nad podacima i odgovarajućih GET, POST, PUT i DELETE metoda. Njegovo sučelje je usredotočeno na resurse i njihove uloge te ga ne zanima unutarnja implementacija. Zahtjevi između klijenta i poslužitelja kod takvih servisa mogu putovati kroz slojeve i posrednike pa interakcija treba biti bez stanja. Posrednici mogu biti proxy poslužitelji ili pak pristupnici koji imaju mogućnost spremanja podataka u predmemoriju (eng. *cache*). Zbog interakcije koja ne održava stanje, zahtjevi trebaju biti potpuni, odnosno, sadržavati sve informacije potrebne za interpretaciju tog zahtjeva. Posljedica su veće poruke koje se izmjenjuju, ali takva komunikacija osigurava veću pouzdanost, skalabilnost i vidljivost pa su negativne posljedice zanemarive. Formati koji se koriste za razmjenu su najčešće JSON i XML. JSON je nešto popularniji i jednostavniji format, pa je i njegovo korištenje lakše. [1] [4]

REST se temelji na zahtjevima i odgovorima, dakle, dizajniran je prema zahtjev/odgovor modelu. Ako resurs ili odgovor nije dostupan, potrebno je ponovno obaviti poziv pri čemu treba voditi računa o učestalosti poziva te razmisliti o korištenju već spomenutog uzorka prekidača. [4]

Primjer REST poziva prikazan je na slici ispod. Klijent pošalji poziv prema servisu koristeći HTTP GET metodu. U kontroleru je definirano mapiranje koje označava da HTTP GET metoda na navedenoj putanji pokreće metodu dohvatiSve(). Zatim se mogu obavljati razni zadaci i dohvatiti podaci iz baze. Servis zatim vraća odgovor u JSON formatu. Ako je sve bilo u redu, vraća se HTTP statusni kod 200 OK te, u ovom slučaju, svi entiteti iz baze podataka.

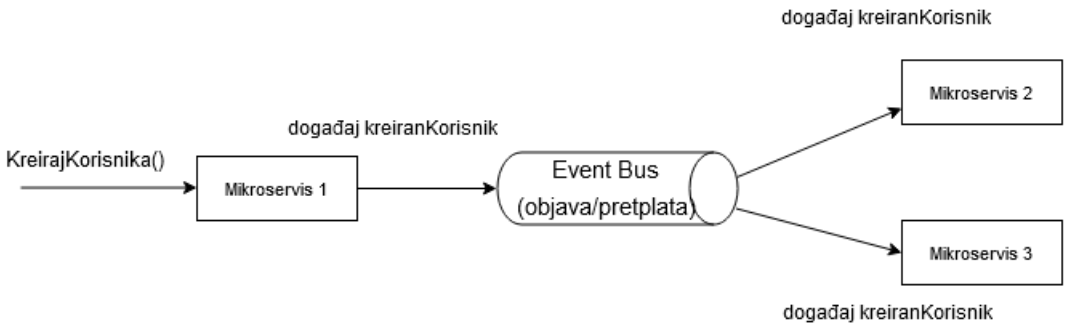


Slika 11 REST poziv [autorski rad]

Iako su mikroservisi međusobno slabo povezani, često treba postojati neki oblik komunikacije među njima. Jedan od načina su **poruke** i asinkrono povezivanje kroz slanje istih. Komponente mogu komunicirati koristeći redove poruka ili teme. [4]

Teme, ili drugim riječima, uzorak objava/pretplata, se koriste na način da se poruka objavi svim zainteresiranim stranama, to jest, pretplatnicima na temu. Na taj način se smanjuje mrežni promet između aplikacija i ne koriste se česti REST pozivi. Kao posrednik se javlja broker poruka koji je zadužen za dostavljanje poruka pretplatnicima. Ovakav pristup omogućuje aplikaciji da ne bude ovisna o dostupnosti primatelja poruka. [4]

Na sljedećoj slici je prikazana komunikacija koja koristi objavu i pretplatu za prenošenje obavijesti. Dakle, temelji se na događajima. Kad se na mikroservisu 1 izvede akcija KreirajKorisnika(), dogodi se kreiraniKorisnik događaj i poruka se šalji preko Event Bus-a zainteresiranik stranima, konkretno, mikroservisima 2 i 3. Oni mogu na temelju te obavijesti pokrenuti neku akciju, na primjer, zapisati korisnika u svoju bazu podataka.



Slika 12 Objava/pretplata uzorak komunikacije (prema [7])

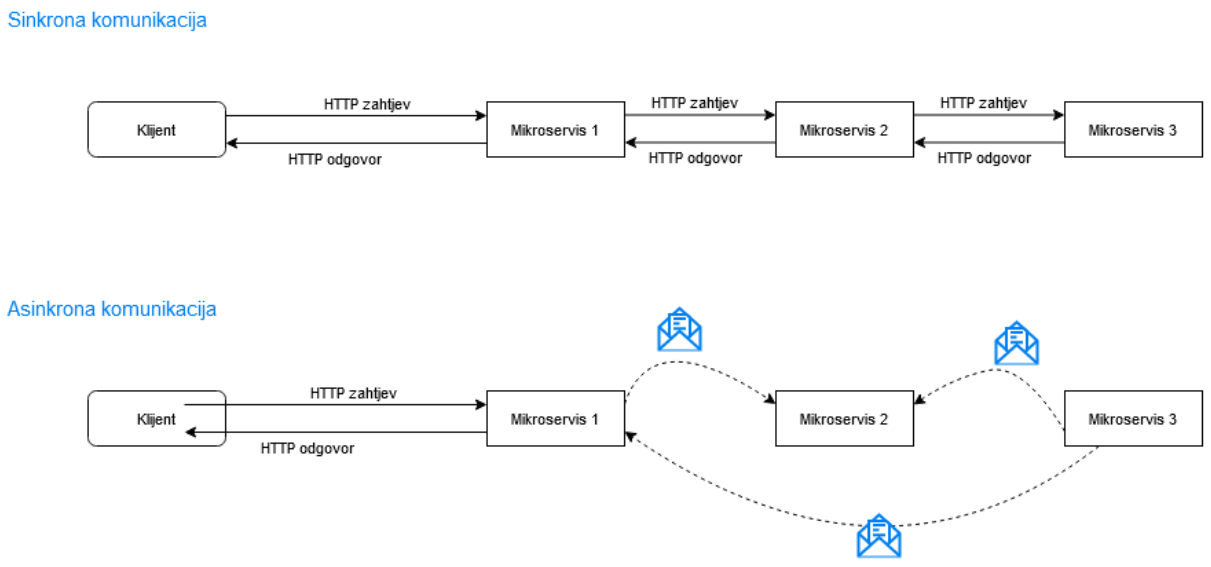
Redovi poruka funkcioniraju na način da proizvođač stavlja poruke u red, dok ih potrošač čita iz reda. Korištenje redova predstavlja veliku prednost prilikom rasta aplikacije i broja funkcionalnosti. Svaka nova funkcionalnost se razvija tako da komunicira porukama s postojećim komponentama, što pak ne zahtijeva promjene u postojećim dijelovima aplikacije.

Ako već neki oblik komunikacije postoji u sustavu, novi mikroservis bi trebao usvojiti taj postojeći način komunikacije. [4]

Slanje poruka omogućuje responzivnost i fleksibilnost – procesi se obavljaju asinkrono, a aplikacija može nastaviti s izvođenjem čak i ako drugi sustav sporo odgovara ili nije dostupan. Slučajevi kada je poželjno koristiti poruke su sljedeći [4]:

- Obavijesti o događajima – ovaj scenarij je vođen događajima, a podrazumijeva slanje poruke i poduzimanje akcija kada dođe do događaja od interesa (primjerice, podaci su učitani, poslužitelj je vratio odgovor i tako dalje).
- Rasterećivanje poslova – slučaj se bazira na rasterećenju posla koji se distribuira na procese koji se izvršavaju asinkrono.

Na slici ispod je prikazano odvijanje različitih vrsta komunikacije. Ako se koriste HTTP pozivi za komunikaciju među servisima, komunikacija je sinkrona. Ovakav poziv od klijenta prema mikroservisu 1 zapravo čini čitav ciklus zahtjeva i odgovora. Kad se koristi ovakav pristup, mogu se javiti kašnjenja u dohvaćanju podataka te su mikroservisi više ovisni jedni o drugom. Drugi način je asinkrona komunikacija korištenjem poruka koja je vođena događajima. Na primjer, dogodi se neka promjena u mikroservisu te on šalje obavijest svim mikroservisima. Takvo slanje poruka ne blokira izvršavanje dretve tog mikroservisa.



Slika 13 Sinkrona i asinkrona komunikacija (prema [7])

Korištenje kombinacije poruke i REST API može imati više prednosti nego kad se koristi samo jedan način komunikacije. Prilikom razvoja ne mora se birati samo jedan način,

već se mogu iskoristiti oba – ponekad je to i optimalno rješenje ako aplikacija ima takve potrebe te je to čini dinamičnijom. [4]

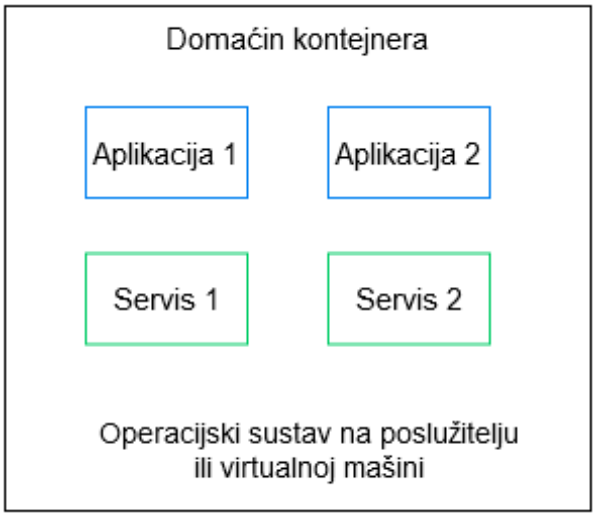
5.6. Registar servisa

Svaki mikroservis imaju svoju URL adresu koja se koristi za određivanje njegove lokacije. Njihova imena trebaju biti adresabilna i unikatna kako bi bili neovisni o infrastrukturi na kojoj se nalaze. Dakle, mora postojati registar servisa koji ih otkriva, a to je zapravo baza podataka koja sadrži mrežne lokacije servisa. Od takvih registara se očekuje da su dostupni većinu vremena i da sadrže ažurirane podatke. Servisi se mogu registrirati i time postati dostupni ostalim servisima ili komunicirati s onima na koje se oslanjaju. [4] [7]

5.7. Kontejneri

Pristup koji se koristi u razvoju softvera kako bi se aplikacije ili servisi pakirali zajedno sa svojim ovisnostima i konfiguracijom u kontejnersku sliku (eng. *container image*) naziva se kontejnerizacija (eng. *containerization*). Takve aplikacije se testiraju kao cjelina i mogu se isporučivati kao instance kontejnerske slike na operacijskom sustavu (kraće OS) koji se koristi. Navedeno omogućuje programerima da isporučuju kontejnere na različite okoline s ništa ili gotovo ništa promjena. Još jedna prednost kontejnera je što predstavljaju jedinicu isporuke, a ta jedinica može sadržavati različite ovisnosti i programski kod. Ukoliko se koristi dijeljeni operacijski sustav, kontejneri omogućuju odvajanje aplikacija jedne od drugih. Oni se izvršavaju pomoću kontejnerskog domaćina koji pak može biti pokrenut na poslužitelju ili virtualnoj mašini. Na slici 14 prikazan je domaćin na kojem se pokreću nekoliko aplikacija i servisa. Svaki kontejner pokreće jednu cijelu aplikaciju ili servis. Jedan od najpopularnijih domaćina je svakako Docker koji se može izvršavati i na Windows i na Linux platformi. [7]

Kontejnerizacija omogućuje i efikasno skaliranje. Ono se jednostavno i brzo provodi putem kreiranja novih kontejnera. Drugim riječima, instancira se nova slika koja postaje novi proces. U svrhu veće pouzdanosti sustava, dobro je više instanci iste slike izvršavati na različitim serverima ili virtualnim mašinama što ujedno i izolira servise u slučaju greške. Dakle, kontejneri omogućuju brojne prednosti kao što su agilnost, prenosivost, izolaciju i skalabilnost servisa. [7]



Slika 14 Kontejneri se izvršavaju na domaćinu (prema [7])

Iako kontejneri omogućuju mikroservisnu arhitekturu i izvrsno se uklapaju u takvom sustavima, oni nisu obavezni za samu arhitekturu te se koncepti arhitekture mogu primijeniti i bez njih. Bez obzira na to, kontejneri se često koriste prilikom razvoja mikroservisnih sustava. [7]

5.8. Testiranje

Iako korištenje mikroservisa donosi mnogo prednosti, ova arhitektura također dolazi s određenim izazovima koji vrlo lako mogu prerasti u problem ako im se ne pristupi ispravno. Jedan od tih izazova je i testiranje.

Uobičajene metode testiranja su:

- Jedinični test
- Integracijski test
- Test komponente
- Test ugovora
- Test sustava
- Test performansi

Jedinični test u mikroservisnoj arhitekturi je isti kao i u ostalim sustavima, a odnosi se na pokretanje najmanje jedinice koda koja se može testirati kako bi se odredilo da li se ponaša očekivano. Ovakvo testiranje otkriva najviše grešaka u sustavu te se najčešće koristi jer daje brze povratne informacije. [4]

Integracijsko testiranje određuje da li bitni dijelovi/moduli surađuju kao što se i očekivalo. U mikroservisnoj arhitekturi se ovakvo testiranje koristi kako bi se provjerile interakcije među slojevima koda i vanjskih komponenata koje se koriste. Vanjske komponente mogu uključivati druge mikroservise, skladišta podataka i drugo. Test koji provjerava interakciju sa spremištem podataka se naziva integracijski test perzistencije i osigurava da shema koji koristi kod odgovara shemi koji se koristi u spremištu. Postaje puno kompleksniji ukoliko se koristi objektno-relacijsko mapiranje (eng. *object-relational mapping*, kraće ORM). [4]

Test komponente ograničava opseg programa na dio koji se testira tako da manipulira sustavom kroz sučelja i koristi tehnike za izoliranje koda od ostalih komponenata. Svrha takvog testa je eliminacija kompleksnog ponašanja ostalih komponenata koje bi moglo utjecati na rezultat testa. Takav test je brz i fokusira se samo na ponašanje komponente koja se testira. U mikroservisnom sustavu, komponente su sami servisi te postoje dva načina takvog testiranja [4]:

- Test koji se izvršava u istom procesu kao servis – kreira se mikroservis koji koristi samo unutarnju memoriju. To znači da se servis, simulatori i spremišta podataka nalaze u samoj memoriji što eliminira potrebu za mrežnim prometom, test se izvršava brže i manja je kompleksnost izgradnje servisa.
- Test kao odvojeni proces – vrši se testiranje nad potpuno implementiranim komponentama koje su isporučene kao odvojeni procesi. Odvijaju se pravi pozivi preko mreže te se koristi pravo spremište podataka.

Ugovor, koji se definira između servisa i konzumirajućih servisa u vidu sučelja, se ispituje pomoću testa ugovora. Sastoji se od očekivanih zahtjeva za ulazne i izlazne podatke, zahtjeva za performanse, konkurentnost i tako dalje. Ugovor je, u mikroservisnoj arhitekturi, ispunjen kroz skup API-ja koje izlažu mikroservisi, konzumirajući servisi su zaduženi za provjeru aspekata ugovora. [4]

Test sustava provjerava da li cjelokupan sustav odgovara poslovnim zahtjevima i postiže ciljeve. Granice takvog testiranja su mnogo veće nego kod drugih testova jer se ispituje ponašanje cijelog integriranog sustava pa su takvi testovi kompleksni za razvoj. Smjernice se razvoj ove vrste testa su da on bude što manji i da se testiranje usmjeri na druge vrste testiranje, zatim treba izbjegavati ovisnost o postojećim podacima, oponašati produkcijsku okolinu te razvijati testove na način da su vođeni klijentima. [4]

Testovi performansi se u mikroservisnoj arhitekturi izvršavaju kod poziva među servisima te kod izvođenja jednog izoliranog servisa kako bi ih se moglo pratiti individualno i održavati zadovoljavajuće performanse. Uobičajeno je da se takav test radi pomoću zahtjeva

koji se postupno povećavaju i opterećuju sustav ili servis te da okolina bude što sličnija produkcijskoj kako bi rezultati bili točniji. Smjernice za razvoj ove vrste testa su sljedeće: potrebno je koristiti što realnije podatke, koristiti opterećenje koje se očekuje na produkciji, omogućiti uvjete koji su što sličniji onima na produkcijskoj okolini, izvršavati testove redovito i rano te, ukoliko je moguće, pratiti i nadzirati performanse produkcije. [4]

5.9. Mikroservisi i DevOps

Korištenje DevOps praksi može biti ključan faktor uspjeha prilikom razvoja mikroservisnog sustava. Naime, prilikom razvoja se teži agilnosti, a mikroservisi omogućuju upravo to: brzo odgovaranje na poslovne mogućnosti. Često isporuka novih mogućnosti/funkcionalnosti pak zahtijeva korištenje DevOps metoda. [4]

DevOps predstavlja skup alata, koncepata, praksi i timskih struktura koji omogućuju organizacijama da brzo isporučuju nove funkcionalnosti klijentima. Pritom se želi što lakše isporučivati i nadzirati servise. Osim što takve organizacije brzo odgovaraju na nove zahtjeve, brže rješavaju i probleme u produkciji. DevOps uključuje sljedeće prakse [4]:

- Agilnost
- Kontinuirana integracija
- Automatizacija isporuke
- Funkcijski jedinični testovi
- Integracijski testovi
- Nadzor servisa i infrastrukture

Kako se mikroservisni sustav sastoji od više manjih aplikacija koje se često isporučuju, ono što omogućuje upravljanje tim servisima, ali i timovima (čiji broj raste s brojem mikroservisa) je DevOps. Ukoliko se DevOps prakse ne koriste, proces isporuke, alati i automatizacija su otežani jer nisu zajednički za sve, već svaki tim kreira svoju infrastrukturu i servise. To predstavlja i dodatan trošak prilikom razvoja te se javljaju različite razine kvalitete, sigurnosti i dostupnosti mikroservisa po timovima. [4]

Timovi se obično okupljaju oko poslovnih funkcionalnosti i zaduženi su za sve dijelove razvojnog procesa: definiranje zahtjeva, razvoj, testiranje, isporuka i nadzor. Kako bi se efikasnije upravljalo timovima, alatima i procesom isporuke, korisno je oformiti i DevOps tim koji će podržavati timove zadužene za razvoj mikroservisa. [4]

6. Praktičan primjer

Kao praktičan primjer izrađena je web aplikacija MyCookbook koja omogućuje pregledavanje recepata, unos novih i izmjenu postojećih, spremanje recepata u kuharicu, kao i planiranje obroka. Osim toga, nudi i dodatne funkcionalnosti kao što su prilagodba serviranja, to jest, broja osoba za koje je recept napisan te preračunavanje mjera.

Aplikacija se temelji na četiri ASP.Net Core WebAPI projekta koji ujedno predstavljaju 4 mikroservisa te na jednom Angular projektu koji čini korisnički sučelje. Svi projekti su kontejnizirani, to jest, izvršavaju se u kontejneru na zajedničkom domaćinu.

U sljedećim poglavljima će biti opisani korišteni alati, dizajn sustava te će biti navedene glavne funkcionalnosti kao i njihov detaljniji opis.

6.1. Alati

Za razvoj poslovne logike korišten je alat Visual Studio 2019. Alat predstavlja integrirano razvojno okruženje koje omogućuje razvoj i analizu koda, ispravljanje grešaka, testiranje i isporuku aplikacija te surađivanje članova tima. Korištena je Community verzija 16.3. Kao tehnologija za izradu web aplikacija korišten je ASP.Net Core razvojni okvir otvorenog koda koji pak koristi standardizirane tehnologije: HTML i Javascript. Upravo pomoću tih tehnologija, implementirani su API projekti koji se temelje na RESTful HTTP servisima. [10]

Alat Visual Studio Code u verziji 1.48.0 je korišten za razvoj korisničkog sučelja web aplikacije. Njegove glavne značajke su da se pokreće svugdje, otvorenog je koda te je besplatan. Kao razvojni okvir odabran je Angular jer omogućuje brzi razvoj web aplikacija pomoću komponenti, ponovno korištenje koda te mnogobrojne predloške i alate koji olakšavaju razvoj. Angular je korišten u verziji 10. [11] [12]

SQL Server Management Studio je integrirano okruženje za upravljanje raznim SQL infrastrukturama. Za potrebe praktičnog dijela, korišten je SQL Server kao SQL infrastruktura, a sam alat je korišten u verziji 18.5.1. On omogućuje pristup, upravljanje, administriranje i razvoj komponenti SQL Servera. [13]

Entity Framework Core (kraće EF Core) je lagana verzija popularne Entity Framework tenhologije za pristup podacima. Otvorenog je koda i dostupna je na različitim platformama. EF Core može poslužiti kao objektno-relacijski preslikač što omogućuje rad s bazom podataka koji eliminira potrebu za većinom koda za pristup podacima jer se koriste .NET objekti. Značajna karakteristika je da pruža podršku za mnoge baze podataka, kao što su:

SQL Server, Sqlite, PostgreSQL, MySql i drugi. Pristup podacima temelji se na modelu koji se sastoji od entitetskih klasa i kontekstnog objekta koji predstavlja sesiju s bazom podataka koja omogućuje spremanje podataka i izvršavanje upita nad njima. EF Core omogućuje generiranje modela iz postojeće baze, programiranje modela prema bazi ili korištenje migracija za stvaranje baze podataka iz modela. Pomoću migracija baza podataka može evoluirati ovisno o promjenama u modelu te je za razvoj korišten upravo ovaj pristup. [14]

Alat koji je korišten za kontejnerizaciju je Docker. Od mogućnosti je korištena i funkcionalnost Docker Hub koja omogućuje stavljanje Docker slike na repozitorij. Nadalje, za orkestraciju servisa je korišten Kubernetes kao Azure servis koji može na temelju slike iz Docker Hub repozitorija kreirati kontejner. Kubernetes je platforma za orkestraciju kontejnera. Njegova zadaća biti odgovoran za pokretanje, distribuciju, skaliranje i ispravljanje aplikacija koje se sastoje od kolekcije spremnika. Osim Azure Kubernetes servisa, korišteni su Azure servisi za bazu podataka: SQL server i SQL database, te web aplikaciju: App Service. [15]

Osim navedenih alata, korištene su i neke pomoćne biblioteke koje će biti ukratko opisane u nastavku.

Kako se API projekti temelje na kontrolerima koji primaju i vraćaju podatke u JSON formatu, za lakše testiranje metoda korišten je alat Swagger. Alat je poslužio kao zamjena za vrlo popularan alat Postman, konfigurira se u samoj aplikaciji, a olakšava dokumentiranje i testiranje aplikacije.

Automapper biblioteka olakšava mapiranje iz entiteta u model i obrnuto, a kako su takva mapiranja potrebna u svakom projektu, napravljena su mapiranja za entitete i modele koji se koriste u projektima. Nadalje, Automapper ubrzava razvoj i ukoliko se za prikaz pojedinih entiteta koriste različiti poslovni modeli.

Angular Material UI bliblioteka komponenata pruža brojne gotove komponente koje se temelje na Material Design specifikaciji za Angular. Osim toga, sadrži i predefinirane teme koje se mogu primijeniti. Neke od korištenih komponenata u aplikaciji su kartica, alatna traka, dijalog i tako dalje.

6.2. Pregled funkcionalnosti

MyCookbook je web aplikacija za spremanje recepata i planiranja obroka. Temelji se na mikroservisima, a svaki od njih predstavlja jednu funkcionalnost:

- **Prijava i korisnički profil**

Korisnik se prijavljuje pomoću svojeg korisničkog imena i lozinke te može upravljati svojim profilom. Prijava je namijenjena korisnicima koji imaju napravljen račun i kojeg mogu sami kreirati. Korisnik se prijavljuje u aplikaciju pomoću forme za prijavu sa svojim korisničkim podacima. Aplikacija provjerava ispravnost unesenih podataka te se koristi autentikacija koja se temelji na JSON Web tokenima (kraće JWT).

- **Recepti**

U bazu podataka se unose novi recepti koji sadrže naziv recepta, popis potrebnih sastojaka, način pripreme, težinu pripreme, potrebno vrijeme, broj serviranja te je li recept vidljiv ostalim korisnicima. Korisnik može pregledavati vlastite recepte te ih uređivati.

- **Kuharica**

Prilikom kreiranja ili uređivanja recepta, recept se može označiti kao javan kako bi bio dostupan ostalim korisnicima. Ostali korisnici mogu spremiti tuđe recepte u svoju kuharicu ili ih maknuti iz kuharice.

- **Planiranje obroka**

Ova funkcionalnost omogućuje izradu plana obroka za sljedeća dva tjedna. Dostupne kategorije su doručak, ručak, večera te međuobrok. Prilikom planiranja, korisnik odabire recept, kategoriju obroka te datum.

- **Prilagodba recepata**

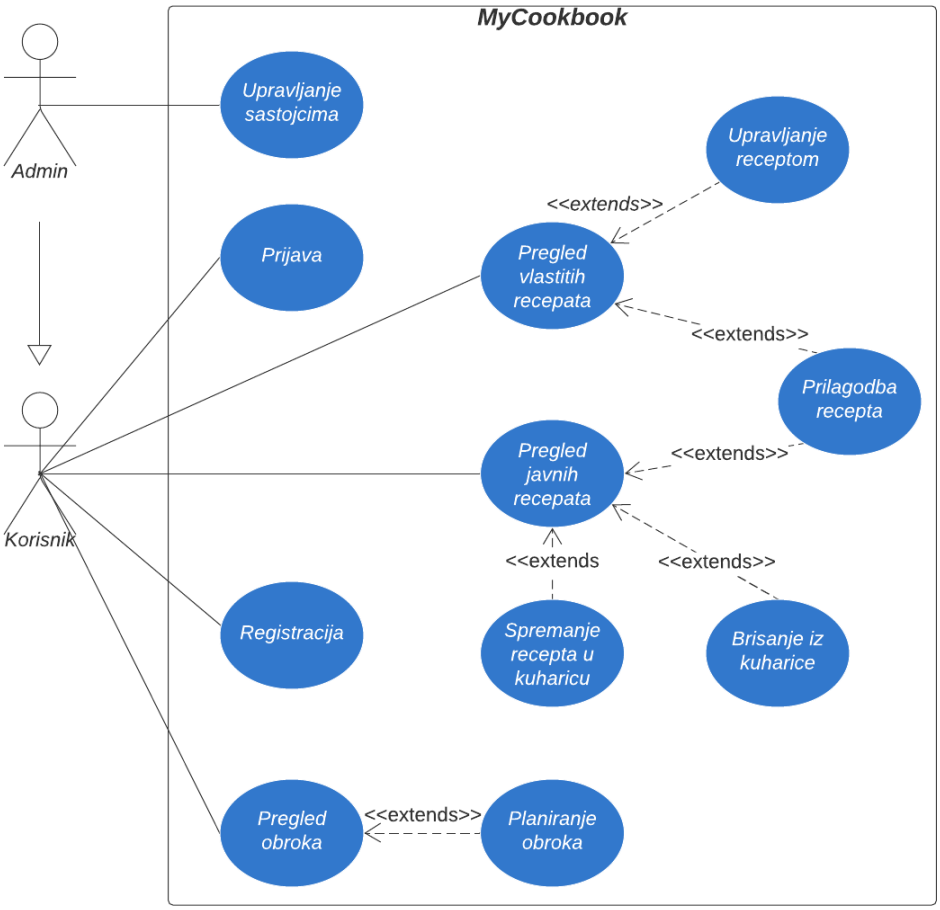
Prilagodba omogućuje pretvaranje mjernih jedinica te preračunavanje jedinica za temperaturu prilikom pečenja.

6.3. Opis dizajna sustava

U ovom poglavlju sadržani su: opis dizajna sustava, razni dijagrami te detaljniji opisi pojedinih funkcionalnosti.

Na slici ispod prikazan je dijagram slučajeva korištenja koji pobliže opisuje korištenje aplikacije. Na use case dijagramu postoje 2 sudionika, a to su korisnik i administrator. Administrator može upravljati sastojcima što uključuje unos, uređivanje i brisanje sastojaka. Ostale mogućnosti nasljeđuje od korisnika. Korisnik se može prijaviti unosom korisničkih podataka ili registrirati ukoliko nema otvoren korisnički račun. Aplikacija omogućuje i pregled vlastitih recepata, kao i upravljanje njima. Navedeno uključuje kreiranje, čitanje, ažuriranje i brisanje recepta, odnosno, CRUD (eng. *create-read-update-delete*) operacija nad njima.

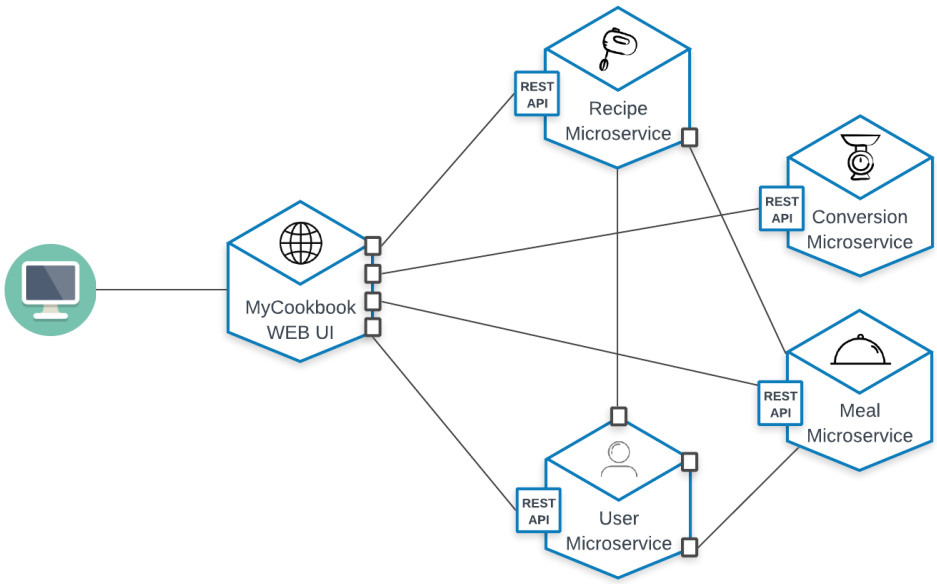
Prilikom pregleda recepta, može isti prilagoditi u smislu broja potrebnih serviranja ili mjernih jedinica. Nadalje, može pregledavati tuđe recepte koji su označeni kao javni. Takve recepte može spremiti u svoju kuharicu. Omogućeno je i uklanjanje recepata iz kuharice. Još jedan slučaj korištenja je i pregled spremljenih obroka. Korisnik može planirati obroke 2 tjedna unaprijed. Također može i upravljati obrocima što uključuje dodavanje, ažuriranje i brisanje obroka.



Slika 15 Dijagram slučajeva korištenja sustava [autorski rad]

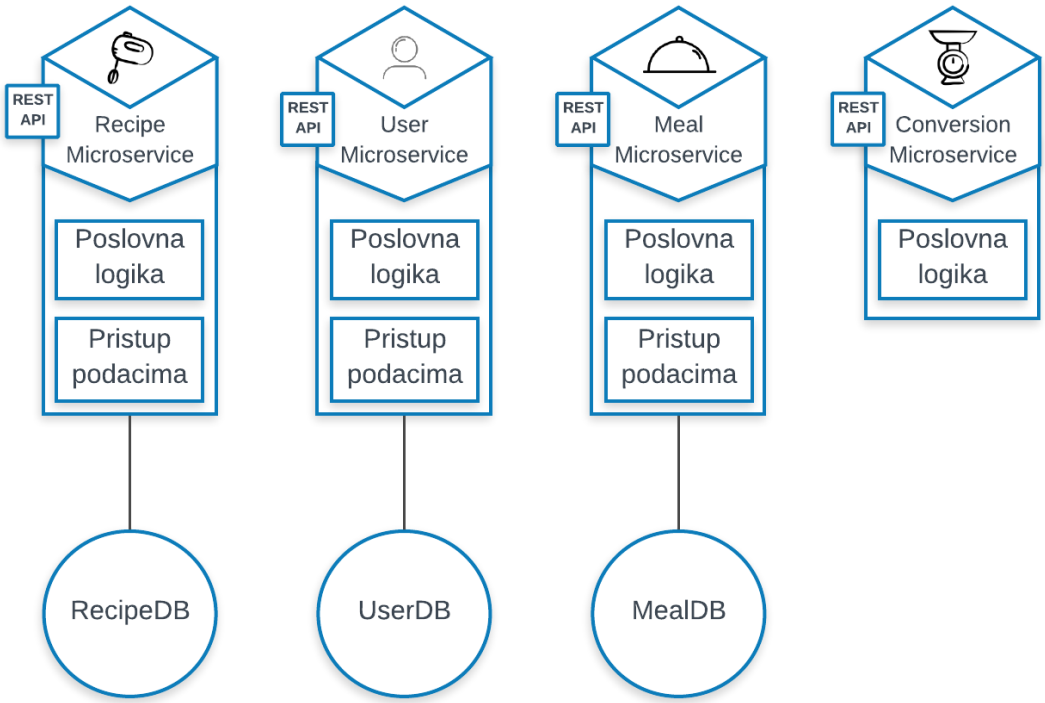
Već je spomenuto da se aplikacija sastoji od nekoliko mikroservisa i web aplikacije. Na slici 16 je dijagram komponenata koji prikazuje različite dijelove sustava te njihove veze. Objašnjava strukturu sustava, odnose među komponentama te naglašava ponašanje servisa jer se koriste sučelja. Koristeći pretraživač može se pristupiti web aplikaciji koja je jedina takva u sustavu. UI servisi te komponente pozivaju ostale servise kako bi dobili i prikazali potrebne podatke. Svako funkcionalno područje aplikacije je implementirano od strane vlastitog mikroservisa. Svaki pozadinski mikroservis izlaže REST API i većina mikroservisa konzumira API-je koje pružaju ostali mikroservisi. Primjerice, RecipeMicroservice koristi

UserMicroservice kako bi provjerio podatke za autentikaciju ili da li postoji korisnik za kojeg se unosi novi recept. Za pozivanje operacija servisa upotrebljava se asinkrona komunikacija.



Slika 16 Dijagram komponenata [autorski rad]

Mikroservisna arhitektura značajno utječe na odnos aplikacije i baze podataka. Umjesto dijeljenja jedne sheme baza podataka s drugim servisima, svaki servis ima svoju shemu baza podataka. Ovakav pristup često rezultira umnožavanjem nekih podataka, no ako se koriste mikroservisi, shema baze podataka po pojedinom servisu nužna je jer omogućuje lagano povezivanje komponenata. Dijagram na slici 17 prikazuje arhitekturu servisa i baze podataka za primjer aplikacije. Vidljivo je da svaki mikroservis ima svoju bazu podataka, osim mikroservisa ConversionMicroservice kojem ona nije potrebna. Spomenuto je da svaki servis može upotrebljavati vrstu baze podataka koja najbolje odgovara njegovim potrebama, no zbog jednostavnosti modela i poznavanja tehnologije svaki mikroservis koristi SQL Server koji ima zadovoljavajuće performanse. Nadalje, svaki mikroservis koristi troslojnu arhitekturu te odjeljuje sloj za pristup podacima, sloj poslovne logike koji implementira servise te izlaže REST API sučelje pomoću kontrolera.



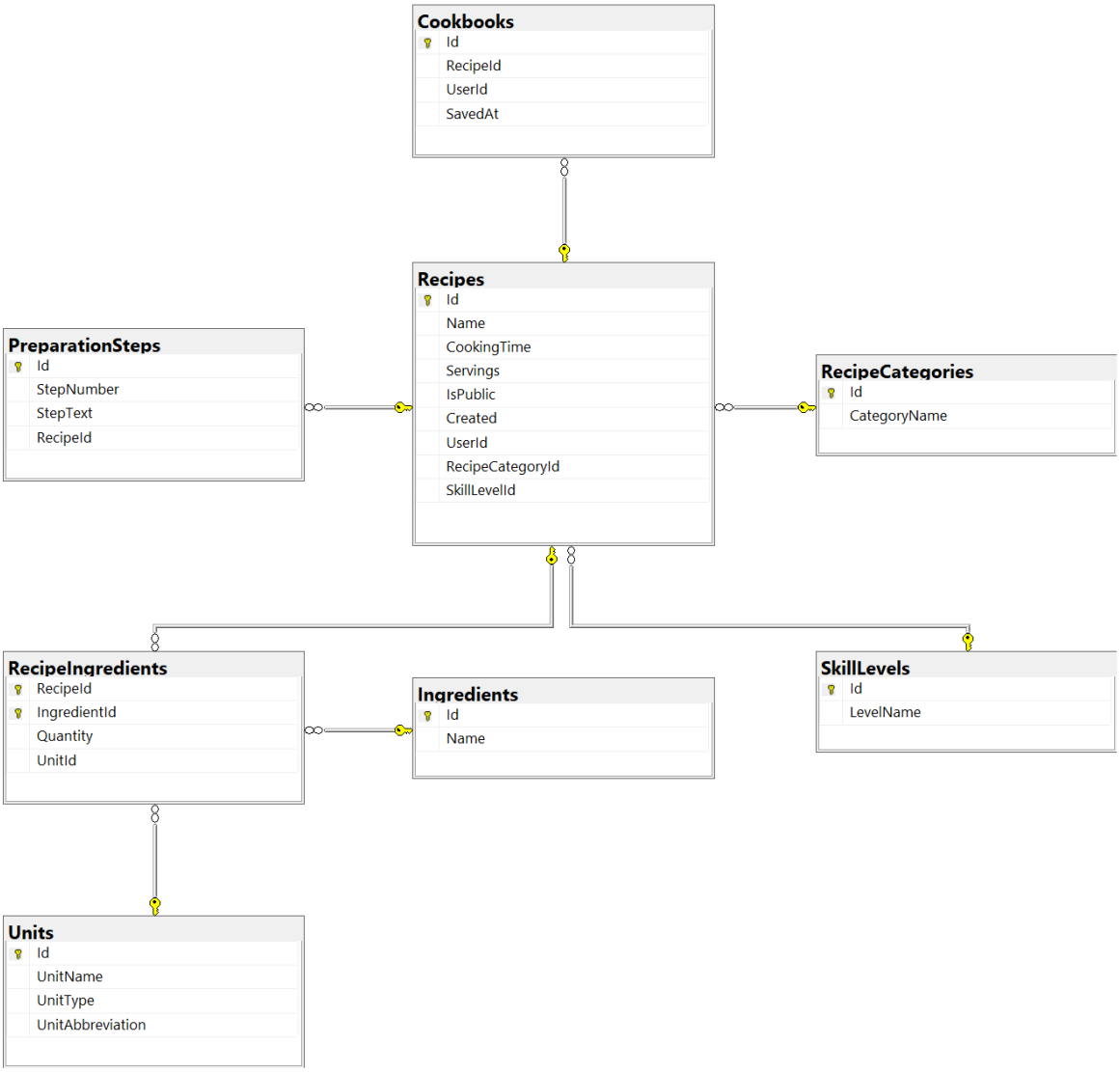
Slika 17 Arhitektura mikroservisa [autorski rad]

ERA modeli

Tri mikroservisa imaju svoju SQL Server bazu podataka (slika 17): RecipeDB, UserDB i MealDB. Model RecipeDB baze podataka sastoji se od 8 tablica te sadrži samo one tablice potrebne za rad s receptima i kuharicom. ERA model ove baze je prikazan slikom 18. Osnovne tablice su kategorije recepata (RecipeCategories), mjerne jedinice (Units), težine pripreme (SkillLevels) te sastojci (Ingredients). Složenije tablice i njihove veze su sljedeće:

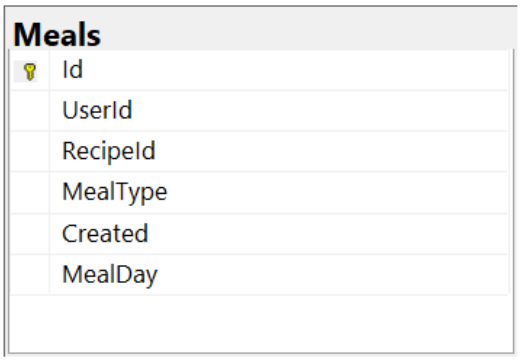
- recepti (Recipes)
  - jedan recept ima vezu na jednu i samo jednu kategoriju recepta
  - jedan recept ima jednu i samo jednu težinu pripreme
- koraci pripreme (PreparationSteps)
  - korak pripreme ima vezu na jedan i samo jedan recept
- sastojci recepta (RecipeIngredients) – ova tablica je proizašla iz veze više-na-više između tablica Recipes i Ingredients jer recept može imati više sastojaka, a sastojak može biti u više recepata
  - jedan sastojak recepta se odnosi na samo jedan recept
  - jedan sastojak recepta se odnosi na samo jedan sastojak
  - jedan sastojak recepta ima vezu na jednu i samo jednu jedinicu mjere
- kuharice (Cookbooks)

- jedan zapis iz ove tablice se odnosi na samo jedan recept
- jedan zapis iz ove tablice se odnosi na samo jednog korisnika (tablica korisnika je u drugoj bazi podataka pa ne postoji pravi vanjski ključ)



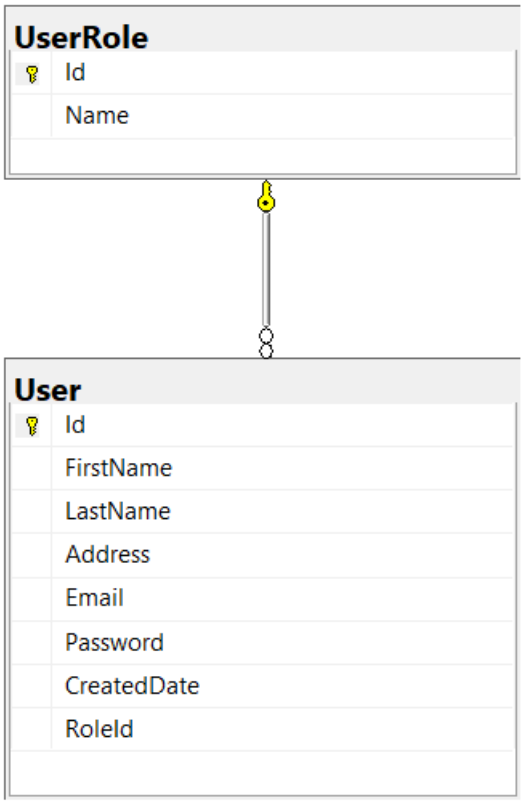
Slika 18 ERA dijagram RecipeDB baze podataka [autorski rad]

Baza podataka MealDB sadrži podatke o planiranim obrocima. Sastoji se od samo jedne tablice: Meals u kojoj se spremaju podaci kao što su ID korisnika, ID recepta, tip obroka (može biti doručak, ručak, večera ili međuobrok), datum obroka te datum kreiranja zapisa (prikazano ERA dijagramom na slici 19). Kao i kod prethodne baze, ova tablica ne sadrži vanjske ključeve jer se podaci o korisnicima i receptima nalaze u drugim bazama podataka.



Slika 19 ERA dijagram MealDB baze podataka [autorski rad]

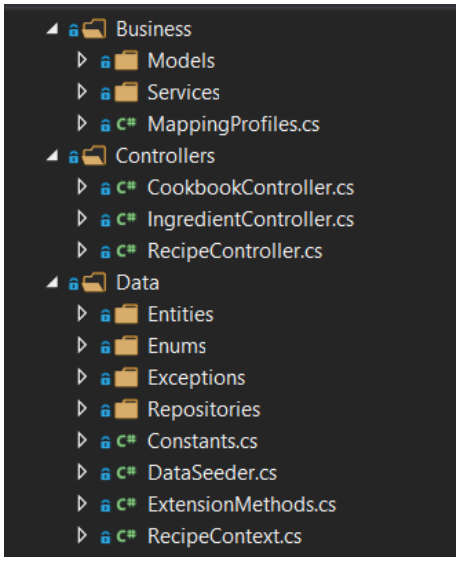
Još jedna baza podataka koja se koristi naziva se UserDB i sadrži podatke o korisnicima i njihovim ulogama koje mogu biti dvije: administrator i korisnik. ERA dijagram je prikazan na slici 20 pri čemu korisnik može imati samo jednu ulogu, dok jedna uloga može biti dodijeljena većem broju korisnika.



Slika 20 ERA dijagram UserDB baze podataka [autorski rad]

Dijagrami klasa

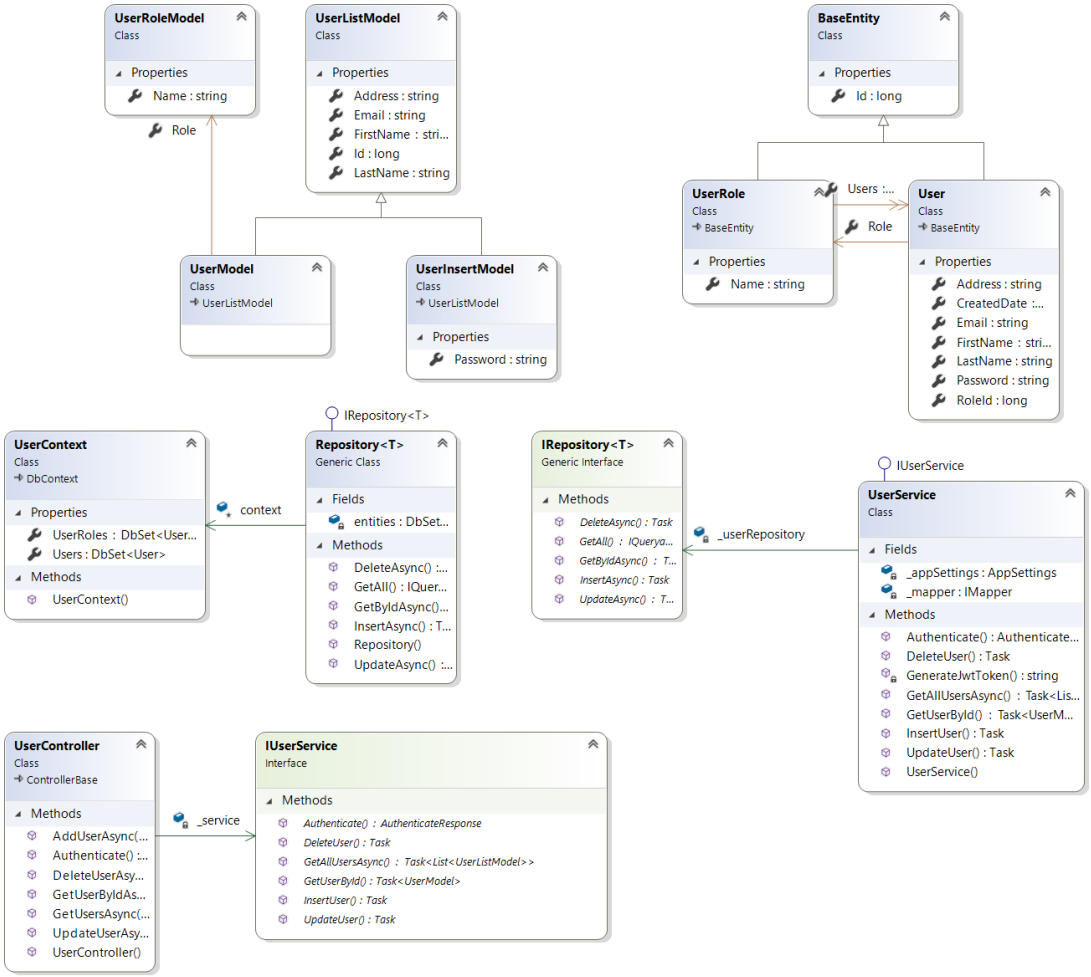
Svaki od mikroservisa ima troslojnu arhitekturu te je podijeljen na podatkovni, poslovni i sloj prikaza. Iznimka je mikroservis ConversionMicroservice koji se ne služi bazom podataka pa nema ni podatkovni sloj. U podatkovnom sloju se nalaze klase i sučelja koja su potrebna za rad s Entity Framework Core-om i bazom. To su, primjerice, entitetne klase, klasa konteksta, repozitorij koji služi za manipulaciju podacima u bazi. Poslovni sloj čine servisi, modeli i klasa za mapiranje iz entitetnih klasa u modele i obrnuto. Mikroservisi zapravo nemaju klasičan sloj prikaza, već se taj sloj odnosi na kontrolere i REST API sučelje koje se izlaže korisnicima. Pomoćna biblioteka Swagger implementira web stranicu koja olakšava rad sa API-jem te služi kao korisničko sučelje. Struktura mikroservisa RecipeMicroservice je prikazana na slici 21, a prikazuje podatkovni (Data), poslovni (Business) i sloj prikaza (Controllers) sa pripadajućim klasama i mapama. Ista struktura prati i ostale mikroservise.



Slika 21 Troslojna arhitektura mikroservisa [autorski rad]

Na dijagramu klasa UserMicroservice mikroservisa (slika 22),zbog bolje preglednosti, prikazane su najvažnije klase koje se koriste u servisu te njihove veze. Svaka od entitetnih klasa nasljeđuje klasu BaseEntity što osigurava da one imaju svojstvo Id. Svaka od tih klasa ima i odgovarajući model, odnosno, klasu koja predstavlja objekt iz domene. Klasa UserContext zadužena je za definiranje baze podataka i tablica. Za implementaciju repozitorija je korišten generički uzorak istog, a klasa koja implementira operacije sučelja IRepository za rad s bazom je Repository. Sučelje koje definira poslovnu logiku za rad s korisnicima je IUserService, a implementacija je sadržana u klasi UserService. Ovaj servis koristi repozitorij kako bi dohvatio entitete iz baze te ih pomoću mapiranja koje je definirano u klasu MappingProfiles mapira u modele. Kontroler naziva UserController, poziva operacije

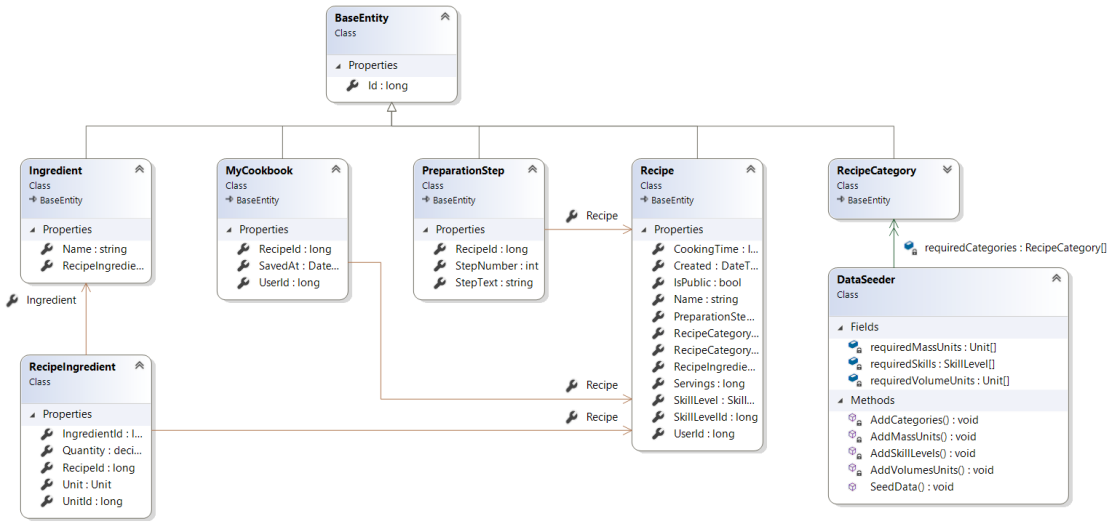
servisa. Prilikom poziva servisa i repozitorija koristi se uzorak dizajna poznat kao injektiranje ovisnosti (eng. *dependency injection*).



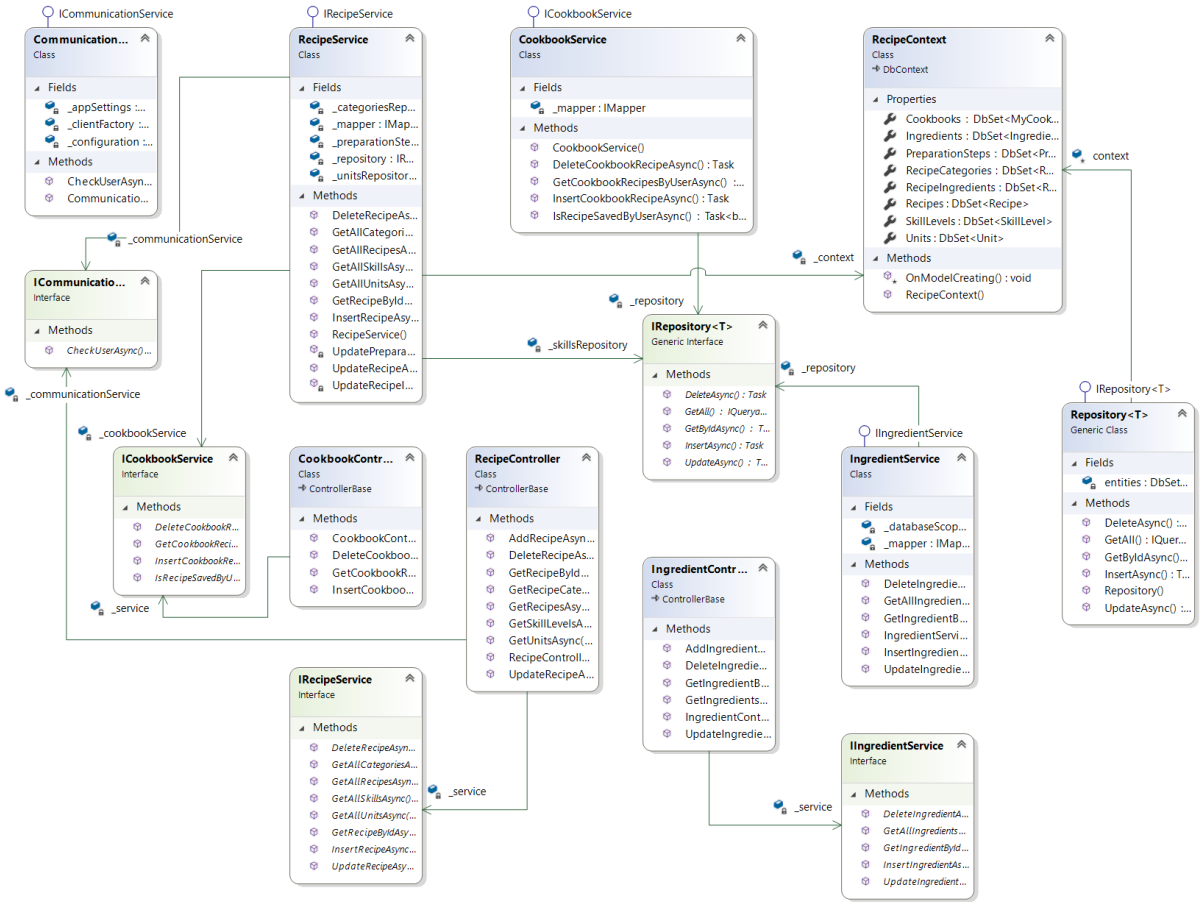
Slika 22 Dijagram klasa mikroservisa za korisnike [autorski rad]

Mikroservis RecipeMicroservice je nešto složeniji, sadrži veći set podataka te ima više klasa. Dijagram koji sadrži samo entitetne klase i njihove veze je prikazan slikom 23. Na sličan način su povezani i odgovarajući modeli. Pristup radu s bazom i mapiranju iz entiteta u modele je isti, a na dijagramu je prikazana i klasa DataSeeder koja sadrži statične metode za postavljanje početnih podataka u bazi. Time se osigurava da baza podataka uvijek sadrži osnovne podatke za rad, primjerice, mjernice jedinice, kategorije recepata, vještine i drugo. U ovom servisu se također koristi injektiranje ovisnosti pa nema direktne veze prema konkretnim klasama koje implementiraju servise ili repozitorije. Također se koristi i uzorak generičkog repozitorija pa svi servisi koriste isti repozitorij, ali s različitim tipom entitetne klase. Važno je naglasiti i klasu CommunicationService. Servis CommunicationService je zadužen za komunikaciju s UserMicroservice mikroservisom te dohvaća podatke o

korisnicima ili o uspješnoj prijavi. Na slici 24 je prikazan dijagram s najvažnijim klasa za rad mikroservisa te njihove veze, a koji uključuje repozitorij, servise, njihova sučelja te kontrolere.

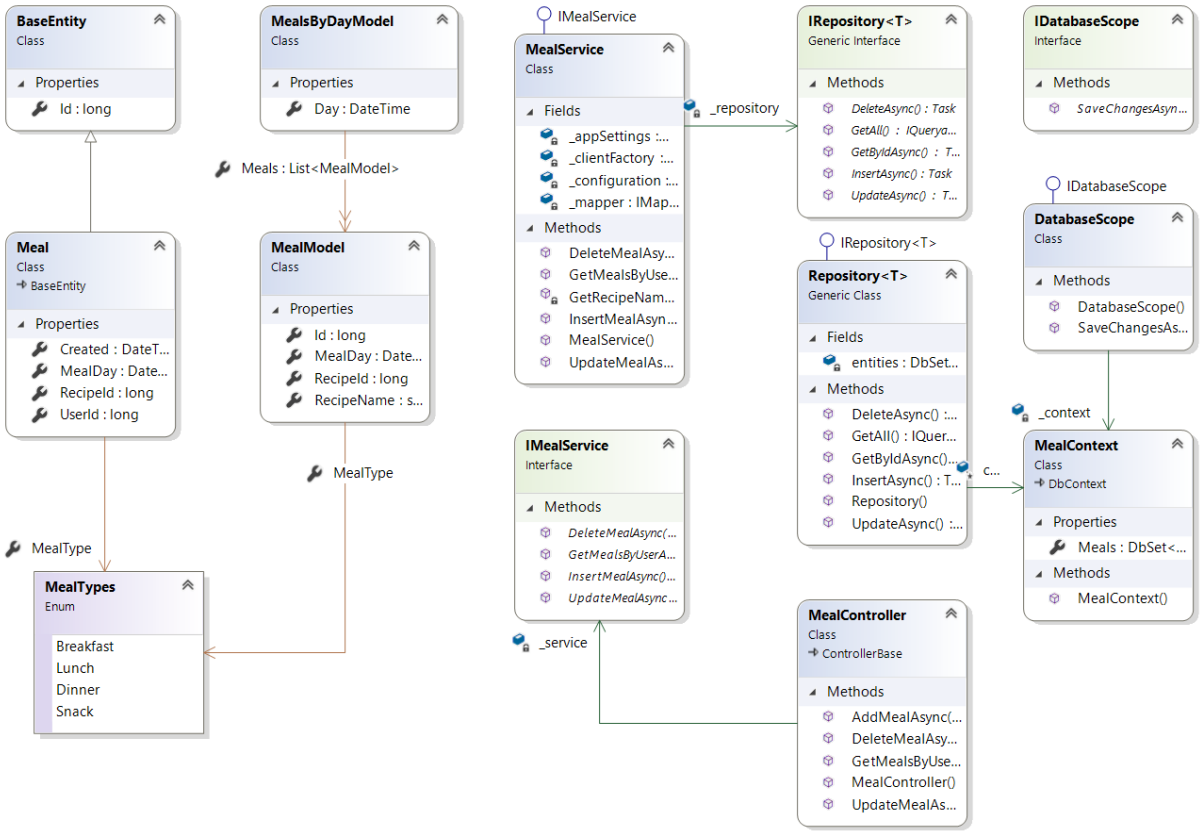


Slika 23 Dijagram entitetnih klasa mikroservisa za recepte [autorski rad]



Slika 24 Dijagram klasa mikroservisa za recepte

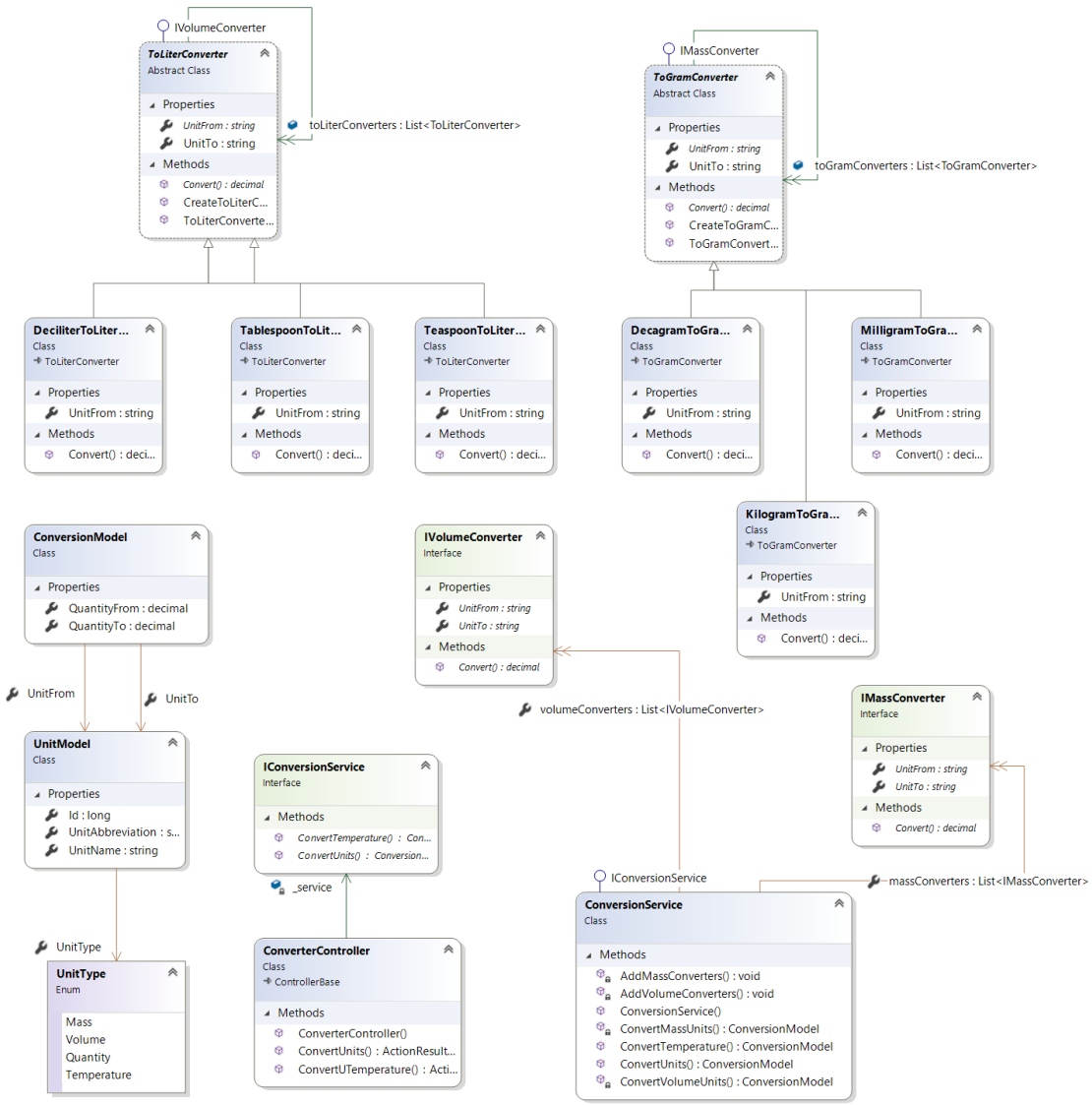
Dijagram klasa mikroservisa MealMicroservice sadrži slične veze prema servisima i repozitorijima koje su do sad objašnjene te sadrži klase koje su zadužene za rad s obrocima. Korišten je i enum kako bi se lakše odredio tip obroka. Također su korišteni koncepti kao što su injektiranje ovisnosti i generički repozitorij. U ovom mikroservisu, komunikacija s ostalima nije izdvojena u poseban servis već se nalazi u klasi MealService. Dijagram klasa za ovaj mikroservis je prikazan na slici 25.



Slika 25 Dijagram klasa mikroservisa za obroke [autorski rad]

Posljednji mikroservis ConversionMicroservice ne koristi bazu podataka pa nema entitetne klase, modele, repozitorij i slično. Umjesto toga, on dobiva podatke o konverziji jedinica koja se želi provesti te na temelju dostupnih konvertere preračunava vrijednosti neke jedinice u drugu. Zbog jednostavnosti i preglednosti dijagrama (slika 26), dio konvertera je izostavljen, te je vidljivo par konvertera za volumen i masu. Svaki konverter je implementiran tako da implementira neko sučelje (IMassConverter ili IVolumeConverter). Zatim postoji apstraktna klasa za jedinicu u koju se pretvara, primjerice, ToGramConverter koja implementira dio sučelja. Tu apstraktnu klasu zatim nasljeđuje klasa koja definira iz koje mjerne jedinice se radi pretvorba, primjerice, KilogramToGramConverter, koja daje konkretnu implementaciju metoda Convert() za pretvaranje kilograma u grame. Apstraktna klasa sadrži

statičku listu u koju se spremaju sve konkretne implementacije za pretvaranje u neku jedinicu kako bi se u servisi jednostavno dohvatio potreban konverter.



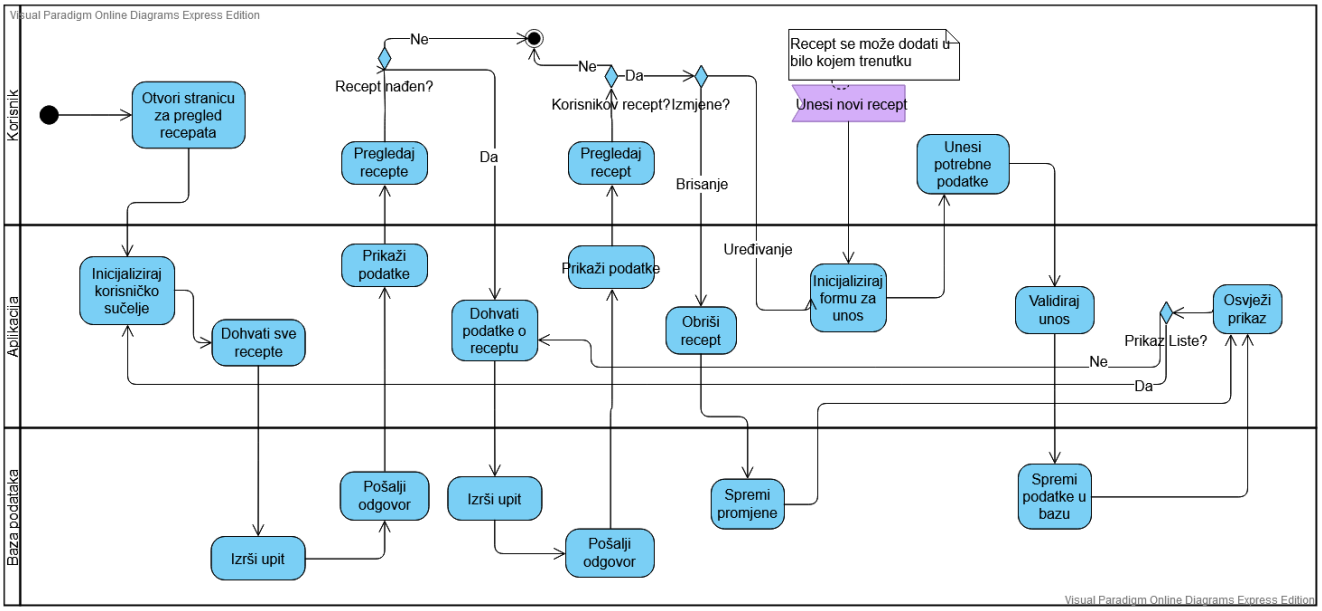
Slika 26 Dijagram klasa mikroservisa za pretvorbu jedinica [autorski rad]

Funkcionalnosti kroz dijagrame aktivnosti

Glavne funkcionalnosti aplikacije su rad s receptima, kuharicom i obrocima pa će u nastavku njihovo korištenje, kao i rad sustava, biti objašnjeni pomoću dijagrama aktivnosti.

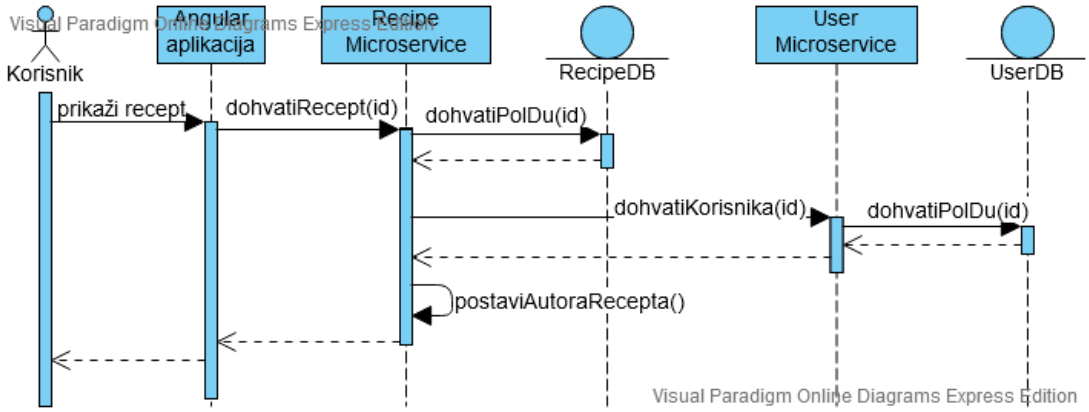
Nakon prijave, korisnik se otvara stranica za pregledavanje postojećih recepata koji su označeni kao javni. Najprije se prikaže korisničko sučelje koje se zatim inicijalizira podacima dohvaćenim iz baze podataka. Korisnik ih zatim može pregledavati te kliknuti na neki određeni recept. Ako je korisnik unio taj recept, može ga mijenjati ili obrisati iz baze. U bilo kojem trenutku može kliknuti na gumb za dodavanje novog recepta, ispuniti potrebne

podatke te se nakon uspješne validacije recept sprema u bazu. Sustav zatim osvježava prikaz kako bi se prikazao dodan ili izmijenjen recept. Ove aktivnosti su prikazane dijagramom na slici 27.



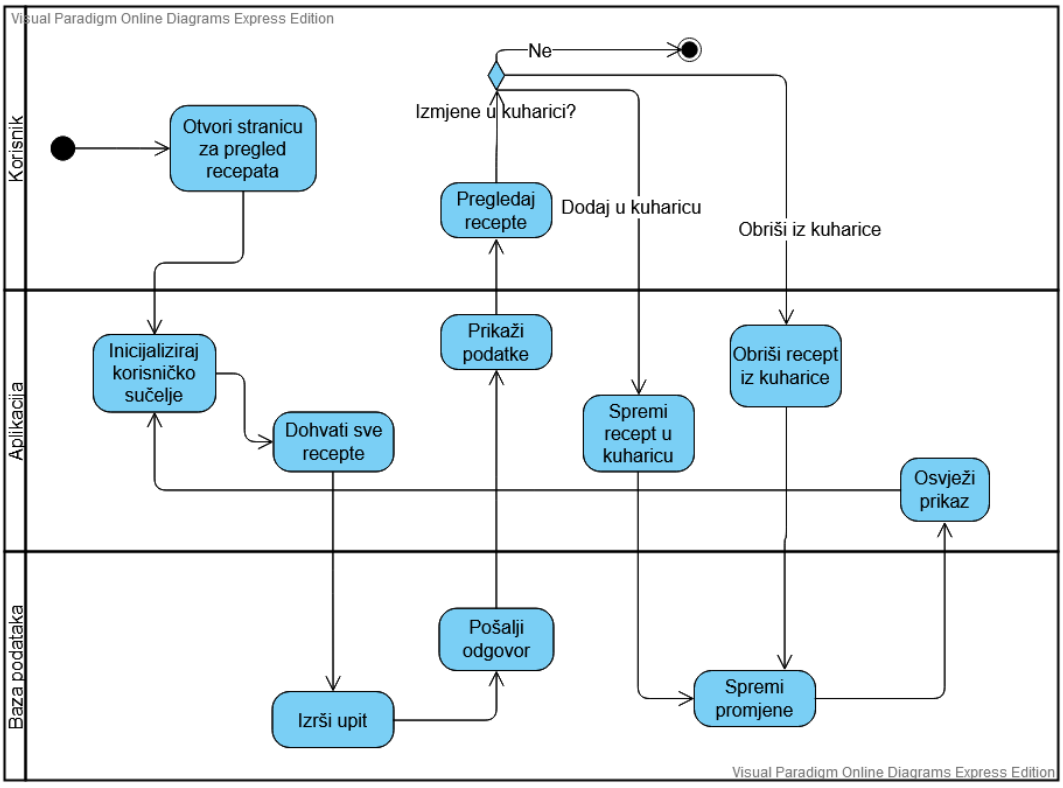
Slika 27 Dijagram aktivnosti za rad s receptima [autorski rad]

Kako je na prethodnom dijagramu prikazan sažet rad cijelog sustava s receptima iz perspektive korisnika, kroz dijagram slijeda je prikazana i komunikacija samih komponenata sustava. Kada dođe zahtjev za prikazom detalja jednog recepta, Angular aplikacija šalje zahtjev na mikroservis za recepte koji zatim iz svoje baze dohvaća podatke o recepte. Međutim, uz recept se prikazuje i korisničko ime autora recepta, pa je potrebno poslati zahtjev na mikroservis koji je zadužen za rad s korisnicima, a koji pribavlja podatke o autoru iz svoje baze podataka. Nakon što je mikroservis za recepte primio potrebne vrijednosti, šalje odgovor natrag pošiljatelju zahtjeva, odnosno, prema Angular aplikaciji koja prikazuje recept korisniku. Dijagram slijeda je prikazan na sljedećoj slici.



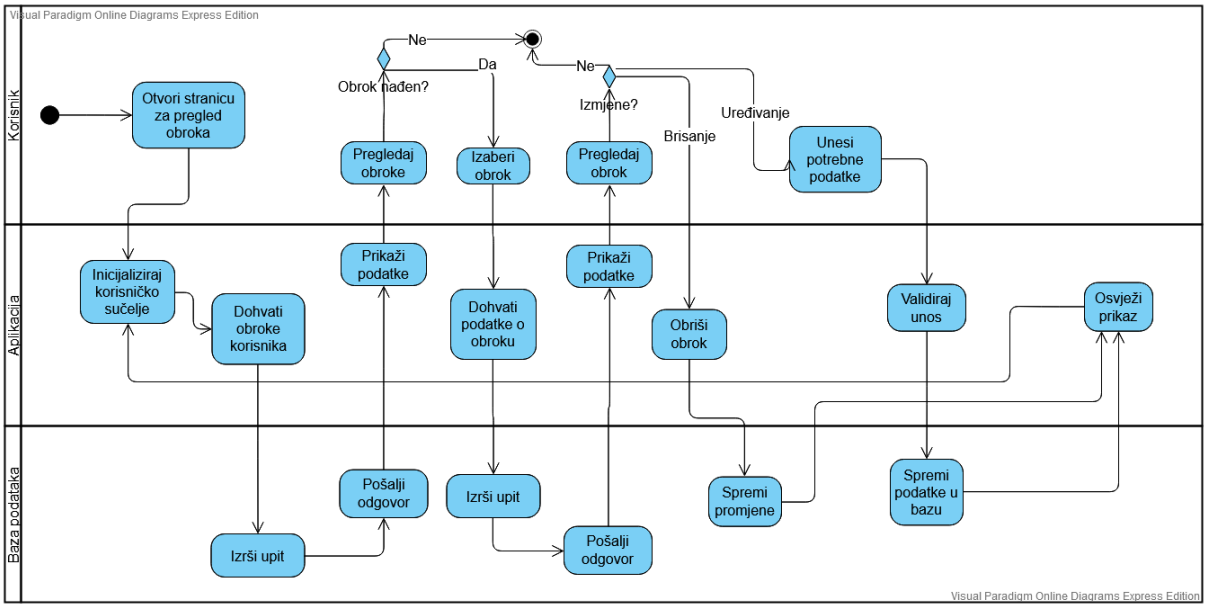
Slika 28 Dijagram slijeda za dohvat jednog recepta [autorski rad]

Rad s kuharicom se svodi na to da korisnik može odabrati da li će spremiti neki recept u svoju kuharicu ili ne. Kuharica predstavlja spremanje tuđih recepata među omiljene, odnosno, u one kojima se želi lakše pristupiti kasnije. Kada korisnik pregledava sve dostupne recepte, može neki recept označiti, odnosno, spremiti u kuharicu ako želi. Pritom se promjene spremaju na bazi podataka i osvježava se prikaz. Slično je i sa uklanjanjem recepta iz kuharice. Ako se on već nalazi u kuharici, korisnik može kliknuti na ikonicu koja briše recept iz kuharice te on više neće biti dostupan za pregled u kuharici. Navedeno je prikazano slikom 29.



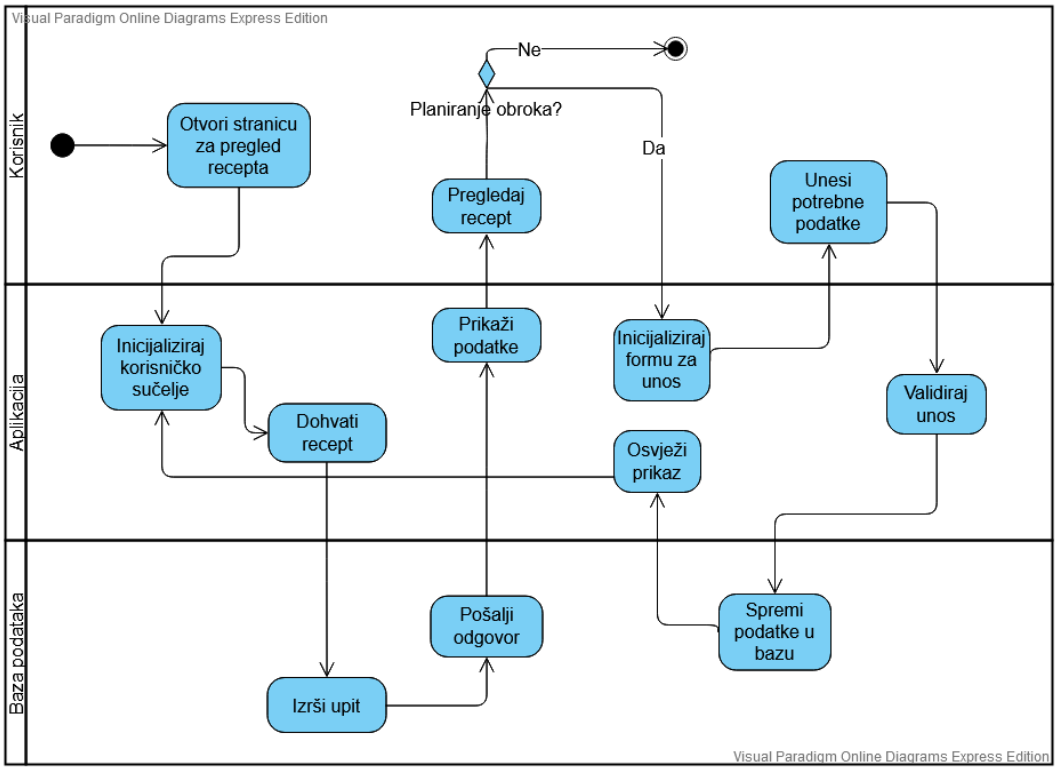
Slika 29 Dijagram aktivnosti za rad s kuharicom [autorski rad]

Pregledavanje obroka se odvija na sličan način kao i pregledavanje recepata. Korisnik može vidjet unesene obroke u naredna 2 tjedna te može pregledati svaki pojedinačni. Kod pregleda pojedinačnog obroka, istog može obrisati ili izmijeniti. Nakon izmjene podatka, oni se validiraju te se promjene spremaju na odgovarajućoj bazi podataka. Dijagram aktivnosti za pregledavanje, brisanje i izmjenu obroka je prikazan na slici 30. Sam unos planiranog obroka je prikazan na slici 31.



Slika 30 Dijagram aktivnosti za pregled obroka [autorski rad]

Unos novog obroka se odvija preko stranice za pregled pojedinačnog recepta pa su aktivnosti prikazane dijagramom na slici 31. Ako korisnik odluči neki recept koristiti za planiranje obroka, to može učiniti kod pregleda tog recepta pri čemu mora unijeti potrebne podatke, kao što su datum i tip obroka, te ih spremiti. Obrok se zatim dodaje u bazu podataka te ga je moguće pronaći na stranici sa obrocima.



Slika 31 Dijagram aktivnosti za unos obroka [autorski rad]

6.4. Implementacija

U ovom dijelu će biti opisani i prikazani najvažniji dijelovi implementacije programskog rješenja na primjeru jednog mikroservisa: RecipeMicroservice. Rad mikroservisa će biti prikazan kroz nekoliko različitih slojeva: baza podataka, repozitorij, servisi i kontroleri. Nakon toga će biti navedeni još neki bitni dijelovi razvoja, kao što je autorizacija, a zatim će biti opisana i Angular aplikacija te korištenje kontejnera.

Baze podataka

Za razvoj svih baza podataka korišten je pristup naziva „Code-First“. To je pristup kod kojeg se prvo razvija kod iz kojeg se kasnije kreira baza podataka. Svi entiteti u bazi moraju imati identifikator, pa je u programskom kodu definirana klasa `BaseEntity` koju nasljeđuju uglavnom sve entitetne klase. Ona sadrži svojstvo `Id` koje je anotirano sa `Key`, što označava da to svojstvo predstavlja primarni ključ u bazi podataka:

```
public class BaseEntity
{
      [Key]
      public long Id { get; set; }
}
```

Primjer entiteta je klasa `Recipe` koja je djelomično prikazana u nastavku. Ona nasljeđuje iz gornje spomenute klase te pomoću anotacija `Required` i `StringLength` definira ograničenja nad atributima u bazi kao što su `NOT NULL` i maksimalna veličina polja. Moguće je postaviti i zadane vrijednosti, kao što je to kod svojstva `CreatedAt`. Vanjski ključevi se definiraju pomoću konvencija prilikom davanja imena svojstvima. Primjerice, vanjski ključ za kategoriju recepta se dodaje tako da se svojstvo zove nazivom klase koju se referencira s dodatkom riječi 'Id', kao što je `RecipeCategoryId`. Nakon toga slijedi referenca na konkretan objekt koja je istog naziva kao i klasa.

```
public class Recipe : BaseEntity
{
	[Required]
	[StringLength(255)]
	public string Name { get; set; }

	[Required]
	public long CookingTime { get; set; }

	public DateTime Created { get; set; } = DateTime.Now;

	[Required]
	public long RecipeCategoryId { get; set; }

	public RecipeCategory RecipeCategory { get; set; }

	…
}
```

Da bi se kreirala baza podataka iz definiranih entiteta, potrebno je kreirati klasu koja nasljeđuje iz klase `DbContext`. Toj klasi se zatim dodaju setovi koji definiraju tablice u bazi:

```
public DbSet<Recipe> Recipes { get; set; }
```

U toj klasi konteksta se definiraju i primarni ključevi klasa koje ne nasljeđuju `BaseEntity`. One imaju složeni primarni ključ koji se ne može definirati anotacijama, već moraju biti definirane pomoću `ModelBuilder`-a. Tako je primjerice nastala međutablica proizašla iz veze više-na-više između tablice recepata i sastojaka. Njezin primarni ključ se sastoji od ID-a recepta i ID-a sastojka. Takav primarni ključ je ujedno i vanjski:

```
modelBuilder.Entity<RecipeIngredient>().HasKey(ri => new {
	ri.RecipeId, ri.IngredientId
});
```

Entity Framework Core kreira i ažurira bazu pomoću migracija. Najprije je potrebno kreirati migraciju pomoću naredbe `add-migration <ime-migracije>`. EF Core zatim iz anotacija i klase konteksta kreira migraciju i SQL upite koje će izvršiti. Takve migracije su parcijalne klase, što znači da dio koda generira sustav, ali i da korisnik može dodati vlastite upite koji će se izvršiti kod primjene migracije. Kako bi se promjene iz migracije propagirale

na bazu podataka potrebno je izvršiti naredbu `update-database`. Ovakav pristup je vrlo jednostavan i omogućuje evoluiranje modela: promjene se mogu napisati u kodu, zatim se kreira nova migracija s kojom se ažurira baza podataka.

Generički repozitorij

Nakon kreiranja baze podataka, potrebno je implementirati logiku čitanja i manipuliranja podacima. Za navedeno je korišten uzorak dizajna generičkog repozitorija. On se koristi kako bi se izbjeglo ponavljanje koda jer bi, u suprotnom, svaki entitet imao svoj repozitorij. Prilikom razvoja sustava, broj entiteta se povećava, a implementacija metoda za dohvat, spremanje ili brisanje iz baze je obično slična i ponavljana. Upravo zbog toga je dobro koristiti generički pristup kako bi se izbjeglo nepotrebno ponavljanje koda i kako bi se jednom napisani upiti mogli iskorištavati za sve entitete i metode.

Klasa `Repository` implementira sučelje `IRepository` koje definira potrebne metode za rad s podacima. Pritom se koristi generički tip podataka i uvjet da taj tip nasljeđuje iz klase `BaseEntity`. Navedeno osigurava da se repozitorij koristi samo nad entitetima koji se nalaze u bazi podataka. Koristi se injektiranje ovisnosti za prosljeđivanje instance klase konteksta koja je potrebna za rad s entitetima iz baze. U programskom kodu u nastavku su navedene 2 metode za dohvat podataka: `GetAll()` i `GetByIdAsync()`. Osim što prva vraća sve entitete nekog tipa, a druga vraća samo jedan, razlika je u tipu podataka koji se vraća. Naime, `GetAll()` vraća `IQueryable` kolekciju koja omogućava daljnju modifikaciju upita u servisima te ne vraća materijalizirane podatke. S druge strane, `GetByIdAsync()` vraća instancu entiteta već pročitanu iz baze. Za bilo kakvu manipulaciju s podacima u bazi se koriste asinkrone metode kako bi postigla responzivnost.

```
public class Repository<T> : IRepository<T> where T : BaseEntity
{
      protected readonly RecipeContext context;

      private DbSet<T> entities;

      public Repository(RecipeContext context) {
          this.context = context;
          entities = context.Set<T>();
      }

      public IQueryable<T> GetAll() {
          return entities.AsNoTracking();
      }

      public async Task<T> GetByIdAsync(long id) {
          return await entities.SingleOrDefaultAsync(s => s.Id ==
id);
      }

      …
}
```

Servisi

Prilikom korištenja repozitorija u servisima, potrebno je definirati entitet, odnosno, tip podataka s kojima će raditi:

```
private readonly IRepository<Recipe> _recipeRepository;

private readonly IRepository<Unit> _unitsRepository;
```

Dakle, servisi pozivaju operacije repozitorija za dohvat podataka i mapiraju ih modele. Prilikom spremanja u bazu, mapiraju se modeli u entitete. Za navedeno, koristi se AutoMapper biblioteka i injektiranje instance `Mapper`-a u servis. Kako bi `Mapper` znao kako obaviti preslikavanje, potrebno mu je proslijediti profil s mapiranjima. Primjer definiranja preslikavanja između recepta koji je entitet u bazi (`Recipe`) i recepta koji se prikazuje korisniku u listi (`RecipeListModel`) je sljedeći:

```
CreateMap<Recipe, RecipeListModel>()
  .ForMember(d => d.UserName, opt => opt.MapFrom(src => src.UserId))
  .ForMember(d => d.SkillLevel, opt => opt.MapFrom(src =>
      src.SkillLevel.LevelName))
  .ForMember(d => d.CategoryName, opt => opt.MapFrom(src =>
      src.RecipeCategory.CategoryName));
```

Zadano ponašanje mapiranja je da se svojstva s istim imenom preslikavaju jedan u drugi pa nije potrebno navoditi svako svojstvo. Dodatna mapiranja svojstva definiraju se pomoću metode `ForMember` kao kod gornjeg primjera gdje `RecipeListModel` sadrži svojstvo `SkillLevel` koje je tipa `string`, dok je kod klase `Recipe` istoimenom svojstvo objekt pa treba se radi preslikavanje na temelju svojstva `LevelName` tog objekta.

Primjer metode servisa `RecipeService` koja objedinjuje do sad navedene koncepte (poziv operacija repozitorija i mapiranje) prikazan je sljedećim programskim kodom:

```
public async Task<RecipeModel> GetRecipeByIdAsync(long id,
      string accessToken)
{
      var recipe = await _repository.GetAll()
            .AsNoTracking()
            .Where(r => r.Id == id)
            .Include(r => r.PreparationSteps)
            .Include(r => r.RecipeCategory)
            .Include(r => r.SkillLevel)
            .Include(r => r.RecipeIngredients)
                  .ThenInclude(ri => ri.Ingredient)
            .FirstOrDefaultAsync();

      var recipeModel = _mapper.Map<RecipeModel>(recipe);
      var user = await _communicationService.CheckUserAsync(
            recipe.UserId, accessToken);
      recipeModel.UserName = user.FirstName;

      return recipeModel;
}
```

Servis najprije pozove `GetAll` metodu repozitorija pomoću koje može oblikovati upit koji će se tek izvršiti nad bazom podataka. Pritom se koriste metode `Include` i `ThenInclude` kako bi se učitali povezani objekti. Navedene metode omogućuju takozvano eager-loading učitavanje podataka, odnosno, podaci se dohvaćaju odmah prilikom dohvata glavnog entiteta. Zatim se vrši mapiranje pri čemu je potrebno proslijediti klasu u koju se želi preslikati: ovdje je prikazano preslikavanje entiteta `Recipe` u model `RecipeModel`. Spomenuti model ima dodatno svojstvo `UserName`, koje sam entitet nema, a da bi se postavila njegova vrijednost potrebno je pozvati mikroservis koji sadrži bazu korisnika. Na kraju metoda servisa vraća model recepta.

Kontroleri

Servise i njihove metode koriste kontroleri. Prosljeđuju im se također preko konstruktora i injektiranja ovisnosti. Kontroleri su zaduženi za rukovanje web API zahtjevima te nasljeđuju klasu `ControllerBase`. Mogu biti anotirani s nekoliko atributa, a osnovni su `Route` (određuje URL uzorak, odnosno, putanju na kojoj se nalazi kontroler), `ApiController` (naznačuje da se tip koristi za posluživanje HTTP API zahtjeva), `Authorize` (naznačuje da kontroler zahtijeva autorizaciju), `Produces` (određuje tip podatka kojeg kontroler vraća) i `Consumes` (određuje tip podatka koji kontroler prihvaća).

Kontroler `RecipeController` je anotiran s atributima `Route`, pri čemu se nalazi na adresi 'api/Recipe', zatim `ApiController` i `Authorize`. Kao i svi kontroler, nasljeđuje klasu `ControllerBase`:

```
[Route("api/[controller]")]
[ApiController]
[Authorize]
public class RecipeController : ControllerBase
{
   …
}
```

Primjer mapiranja iz HTTP zahtjeva u metodu kontrolera prikazan je u nastavku. Metoda `GetRecipeById` je anotirana s atributima `HttpGet` i `Route` što znači da kad dođe HTTP GET zahtjev na tu putanju, kontroler će izvršiti upravo tu metodu. Najprije se iz zahtjeva dohvaća token pristupa koji se koristi kod poziva mikroservisa za korisnike. Nakon toga slijedi poziv servisa i vraćanje podataka korisniku.

```
[HttpGet]
[Route("{recipeId:long}")]
public async Task<RecipeModel> GetRecipeByIdAsync(long recipeId)
{
   var accessToken = Request.Headers[HeaderNames.Authorization]
                        .ToString();
   var recipe = await _service
            .GetRecipeByIdAsync(recipeId, accessToken);
```

```
    return recipe;
}
```

Registriranje servisa za korištenje se odvija u `Startup` klasi. Pritom se određuje sučelje servisa, implementirajuća klasa te životni vijek servisa. Tako je moguće registrirati singleton, prolazan servis (eng. *transient*) ili servis koji traje za vrijeme korisničkog zahtjeva/konekcije (eng. *scoped*). Repozitoriji (ili drugim riječima, servisi koji rade s bazom podataka) se uobičajeno definiraju kao *scoped* servisi, dok su poslovni servisi definirani kao *transient*. U `Startup` klasu se dodaju i razne konfiguracije, primjerice za Automapper i Swagger, zatim se definira korištenje autorizacije, CORS politika i drugo.

Komunikacija

Komunikacija s ostalima mikroservisima je implementirana pomoću HTTP zahtjeva. Kako svaki mikroservis zahtijeva autorizaciju, potrebno je prosljeđivanje pristupnog tokena korisnika te dodavanje autorizacijskog zaglavlja u HTTP zahtjev. Poziv drugog mikroservis je jednostavan: potrebno je kreirati poruku zahtjeva s tipom HTTP zahtjeva i uri adresom te HTTP klijenta i poslati poruku. Pritom se koristi asinkrono slanje. Nakon što stigne odgovor, podatke je potrebno deserijalizirati u odgovarajuću klasu:

```
var uri = _appSettings.UserAPI + "/" + userId;
var request = new HttpRequestMessage(HttpMethod.Get, uri);
request.Headers.Add("Authorization", accessToken);

var client = _clientFactory.CreateClient();
var response = await client.SendAsync(request);
```

Ovaj pristup je izabran jer se temelji na principu zahtjev-odgovor. Iako takvi pozivi mogu biti spori, ne koristi se ulančavanje poziva i pozivi su asinkroni, pa je responzivnost sustava zadovoljavajuća. Već je spomenuto da su asinkrone poruke s posrednikom bolji pristup za komunikaciju među mikroservisima, no u ovom slučaju zahtjev nije potrebno slati zainteresiranim stranama, već se očekuje i odgovor.

Autorizacija

Svi mikroservisi koji koriste bazu podataka zahtijevaju i autorizaciju prilikom korištenja njihovih resursa. Autorizacija je implementirana pomoću JSON web token, koristi se Bearer shema, a tokeni su validni 7 dana. Angular aplikacija je razvijana tako da svakom HTTP zahtjevu prije slanja doda autorizacijsko zaglavlje sa pristupnim tokenom pomoću 'presretača' zahtjeva. Token se dobiva prilikom uspješne prijave te se sprema u lokalno spremište (eng. local storage). Osim toga, Angular nudi i sučelja koja određuju može li korisnik pristupiti nekoj putanji. U implementaciji tih sučelja se ispituje je li korisnik prijavljen i je li mu dopušten pristup do nekog resursa.

Početno popunjavanje baze podacima

Kako je isporuka aplikacije ili kod proširenje tima i instaliranja lokalne baze podatka bitno imati početne podatke za rad, implementirano je i popunjavanje baze početnim podacima. To se može izvesti na dva načina: preko migracija i preko prilagođene inicijalizacije baze u programu. Kako su migracije nešto kompliciranije za izvođenje i nisu toliko fleksibilne, koristio se drugi način. Prilikom pokretanja aplikacije se poziva statička metoda koja je zadužena provjeru postojećih podataka i unos onih obveznih. Programski kod u nastavku prikazuje inicijalizaciju potrebnih kategorija recepata:

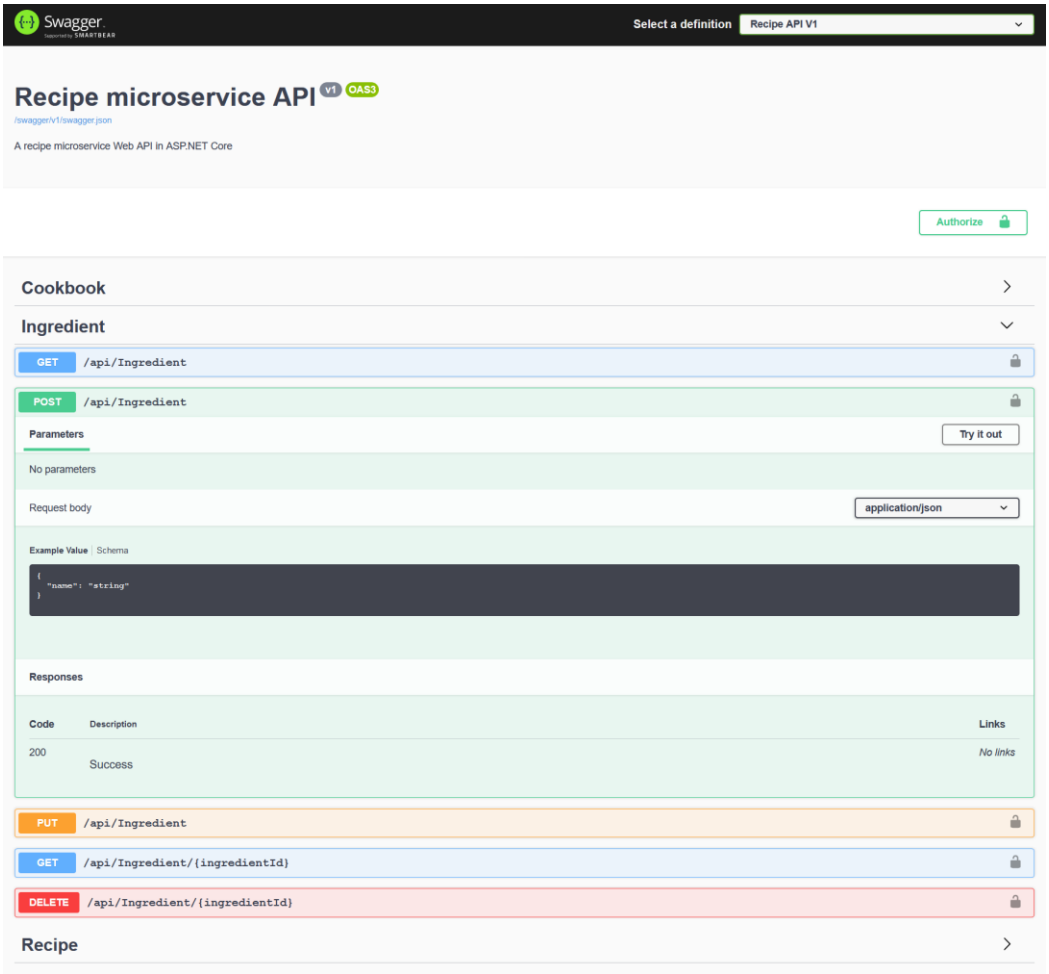
```
      private static RecipeCategory[] requiredCategories = new
RecipeCategory[]
{
new RecipeCategory { CategoryName = Constants.CATEGORY_WARM_APPETIZER },
new RecipeCategory { CategoryName = Constants.CATEGORY_COLD_APPETIZER },
new RecipeCategory { CategoryName = Constants.CATEGORY_MAIN_DISH }
}
```

Swagger

Navedeno je da je Swagger alat koji olakšava dokumentiranje i testiranje API-ja. Njegovo korištenje se definira u `Startup` klasi pri čemu se može odrediti verzija, naslov te opis API-ja. Definiranje Swagger dokumenta u kodu izgleda ovako:

```
      services.AddSwaggerGen(c =>
      {
            c.SwaggerDoc("v1", new OpenApiInfo
            {
                  Version = "v1",
                  Title = "Recipe microservice API",
                  Description = "A recipe microservice Web API in ASP.NET
Core",
            });
      }
```

Izgled Swagger stranice za mikroservis RecipeMicroservice prikazan je na slici 32. Vidljivo je da API nudi tri pristupne točke, to jest, tri kontrolera: Cookbook, Ingredient i Recipe. Za svaki pojedini kontroler, dostupan je i pregled HTTP metoda koje on podržava, a uz to je naveden i tip podatka koje metoda očekuje, te mogući odgovori. Stranica nudi testiranje API-a, a već je pripremljen i JSON format poruke koja se šalje u zahtjevu. Osim toga, Swagger podržava i različite tipove autorizacije.

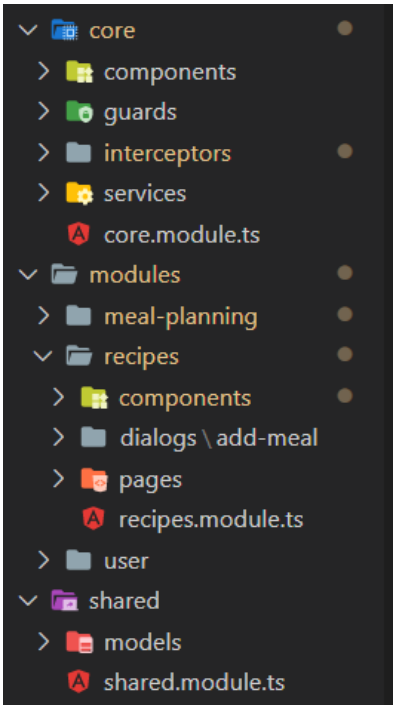


Slika 32 Swagger dokument za API mikroservisa

Korisničko sučelje

Korisničko sučelje je izrađeno koristeći Angular i biblioteku Angular Material UI. Takva aplikacija se temelji na modulima i komponentama koje grade stranicu. One pak pozivaju operacije servisa koji komuniciraju s mikroservisima. Osim toga, moguće je koristiti i razne 'presretače' zahtjeva (eng. *interceptors*) i 'čuvare' (eng. *guards*) kako bi se omogućila autorizacija, presretanje i rukovanje pogreškama ili pak spriječili neautorizirani pristupi prema pojedinim rutama. Angular projekt ima slojevitu strukturu (pri tom se misli na raspored mapa, modula, servisa…) kako bi se pravilno odvojili pojedini dijelovi i poštivali koncepti koje Angular donosi. To je rađeno prema najboljim praksama u Angularu pa tako postoje Core i Shared moduli, dok se u mapi Modules nalaze moduli vezani uz pojedine funkcionalnosti. Core modul sadrži klase koje se instanciraju samo jedinom, primjerice, servisi i interceptori. U Shared modulu se nalaze dijelovi koje mogu koristiti svi ostali moduli i komponente (primjerice, modeli podataka). Funkcijski moduli sadrže razne komponente te oni definiraju navigaciju za vlastite komponente. Struktura projekta je prikazana na slici 33. Vidljivo je da Core modul sadrži još komponente (konkretno, komponenta za prijavu), servise, 'presretače'

i 'čuvare'. Funkcijski moduli su *meal-planning*, *recipes* i *user*, a svaki sadrži komponente koje se, ovisno o ulozi, dijele na komponente koje su dio neke stranice (ili više njih), stranice i dijaloge.



Slika 33 Angular struktura projekta

Komunikacija Angular aplikacije sa mikroservisima i njihovim API sučeljem je implementirana u servisima. Oni se definiraju pomoću dekoratora `@Injectable` nakon čega ih se može injektirati kroz konstruktor ostalim komponentama. Definiranje servisa je prikazano sljedećim programskim kodom:

```
@Injectable({
  providedIn: 'root'
})
export class RecipeService { … }
```

Poziv prema mikroservisu se odvija tako da se koristi klasa `HttpClient` i njezine metode `get`, `post`, `delete` i tako dalje. Potrebno joj je proslijediti URL adresu na koju se šalje zahtjev te podatke u tijelu poruke u slučaju `post` ili `put` metode. Slanje GET prikazano je u nastavku.

```
getRecipe(id: number): Observable<RecipeModel> {
   const url = `${this.baseUrl}/Recipe/${id}`;
   return this.http.get<RecipeModel>(url).pipe(
     tap(_ =>
       console.log('fetched recipe')),
       catchError(this.handleError<RecipeModel>('getRecipe'))
   );
 }
```

Za Angular je karakteristično korištenje raznih operatora, a na gornjem primjeru su to `pipe` i `tap`. `Pipe` kombinira više operatore koji će se izvršiti kada dođe odgovor, dok `tap` služi za izvođenje dodatnih akcija, kao što je ispisivanje informacija u log. U prikazanom primjeru, ispisat će se informacija o dohvaćenim receptima ako odgovor dođe, a ako dođe do greške, onda će se pozvati funkcija za rukovanje greškom. Nadalje, funkcija `getRecipe` vraća podatak koji je tipa `Observable`. Tip `Observable` se uobičajeno koristi za rukovanje događajima, asinkrono programiranje i rukovanje višestrukim vrijednostima te je dio Observer uzorka dizajna. Dakle, komponenta koja poziva gornje navedenu metodu servisa mora se pretplatiti na nju, a servis će komponentu obavijestiti jednom kad dođu podaci sa mikroservisa. To omogućuje inicijalizaciju korisničkog sučelja bez čekanja da se svi podaci učitaju te čini aplikaciju više responzivnom. Sljedeći isječak koda iz komponente `RecipeDetailComponent` prikazuje pozivanje spomenute metode servisa i pretplaćivanje:

```
getRecipe(){
    const id = +this.route.snapshot.paramMap.get('id');
    if (id){
      this.recipeService.getRecipe(id).subscribe(data => {
        this.recipe = data;
      });
    }
}
```

Komponenta najprije dohvaća ID recepta iz URL putanje kako bi znala za koji recept treba dohvatiti podatke. Zatim poziva servis i prosljeđuje mu dohvaćeni ID recepta. Daljnje izvođenje programa se nastavlja, a jednom kada mikroservis pošalje odgovor, to jest, potrebne podatke izvodi se anonimna funkcija unutar `subscribe` metode. Dobiveni podaci se zapisaju u varijablu `recipe`. Ta varijabla je javna pa joj može pristupiti HTML datoteka te komponente. Koristi se jednostavna sintaksa s dvjema vitičastim zagradama za dohvaćanje objekta i njegovih polja:

```
<mat-card class="recipe-card">
    <mat-card-header>
      <mat-card-title> {{ recipe.name }} </mat-card-title>
      <mat-card-subtitle> Created by {{ recipe.userName }} </mat-
card-subtitle>
      …
</mat-card >
```

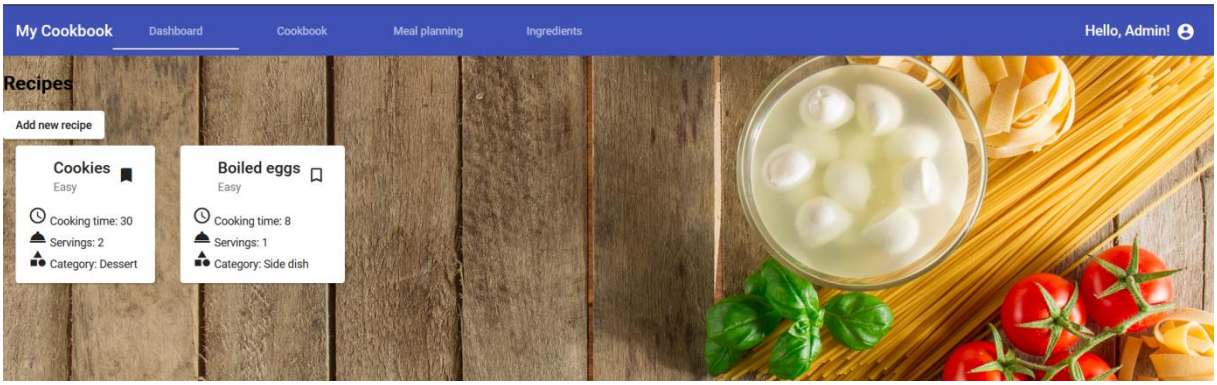
U gornjem isječku koda je također vidljivo korištenje komponente `MatCard` koja je dio Angular Material UI biblioteke.

Vezivanje podataka (eng. *binding*) se u Angular radi na više načina. Jedan od njih je *ngModel koji služi za dvostrano povezivanje podataka, a koje se najčešće koristi kod uređivanja modela podataka. Dakle, ako korisnik promijeni vrijednost nekog podatka na formi, promjena će biti vidljiva na modelu. Vrijedi i obratno, ako se podatak promijeni kod

izvršavanja aplikacija, promjena će se prikazati na ekranu korisnika. Drugi primjer je vezivanje akcija na neki događaj pri čemu se koriste oble zagrade. U isječku koda u nastavku, a koji je dio forme za uređivanje recepta, prikazano je uređivanje broja serviranja te je određeno izvođenje metode `adjustServings` kad se dogodi promjena na `input` elementu.

```
<input matInput type="number"
      placeholder="Ex. 12"
      min="1" max="50"
     [(ngModel)]="adjustedServings"
     (change)="adjustServings()">
```

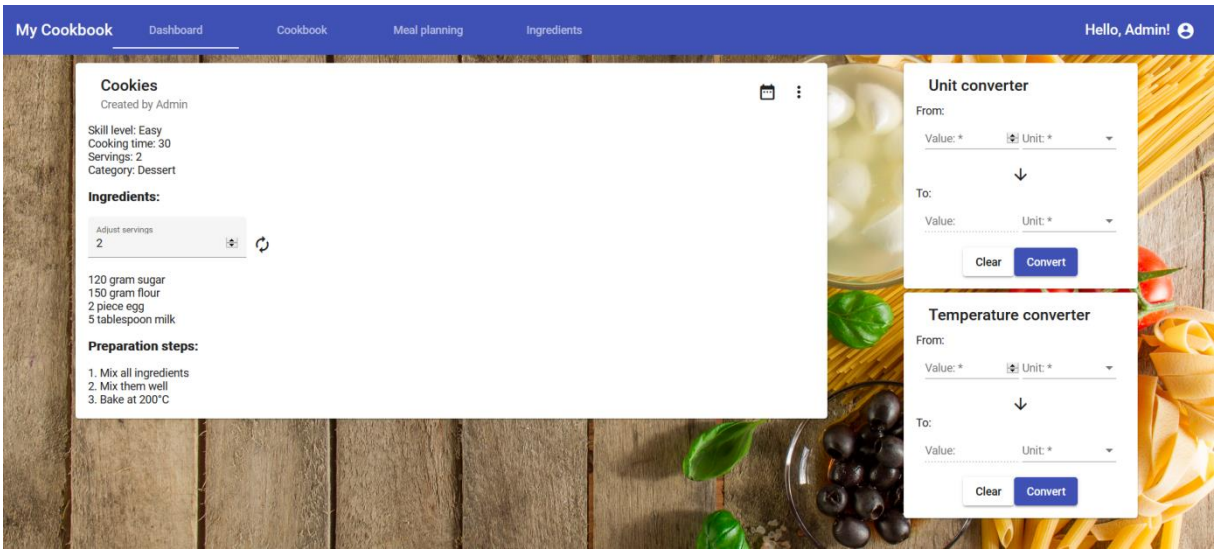
U nastavku će biti opisan rad s web aplikacijom uz prikaze najvažnijih ekrana. Početna stranica prijavljenog korisnika predstavlja dostupne recepte koje korisnik može pregledavati. Dostupna je i alatna traka pomoću koje može doći pregleda svih recepata, kuharice, planiranih obroka te svojeg profila i recepata ili se može odjaviti. Nadalje, omogućuje dodavanje novog recepta pri čemu se otvara forma za unos recepta. Svaki recept na ovoj stranici prikazuje osnovne informacije kao što su vrijeme pripreme, kategorija recepta, zahtjevnost te za koji broj osoba je recept namijenjen. Osim toga, na kartici recept se nalazi oznaka koji naznačava nalazi li se recept već u kuharici (ispunjena oznaka) ili ne (prazna oznaka). Stranica kuharice prikazuje recepte kao i početna na identičan način, ali dohvaća samo recepte koji se nalaze u kuharici. Ove dvije stranice za svoj rad koriste API mikroservisa za recepte. Prikaz početne web stranice nalazi se na slici 34.



Slika 34 Početna stranica s prikazom recepata

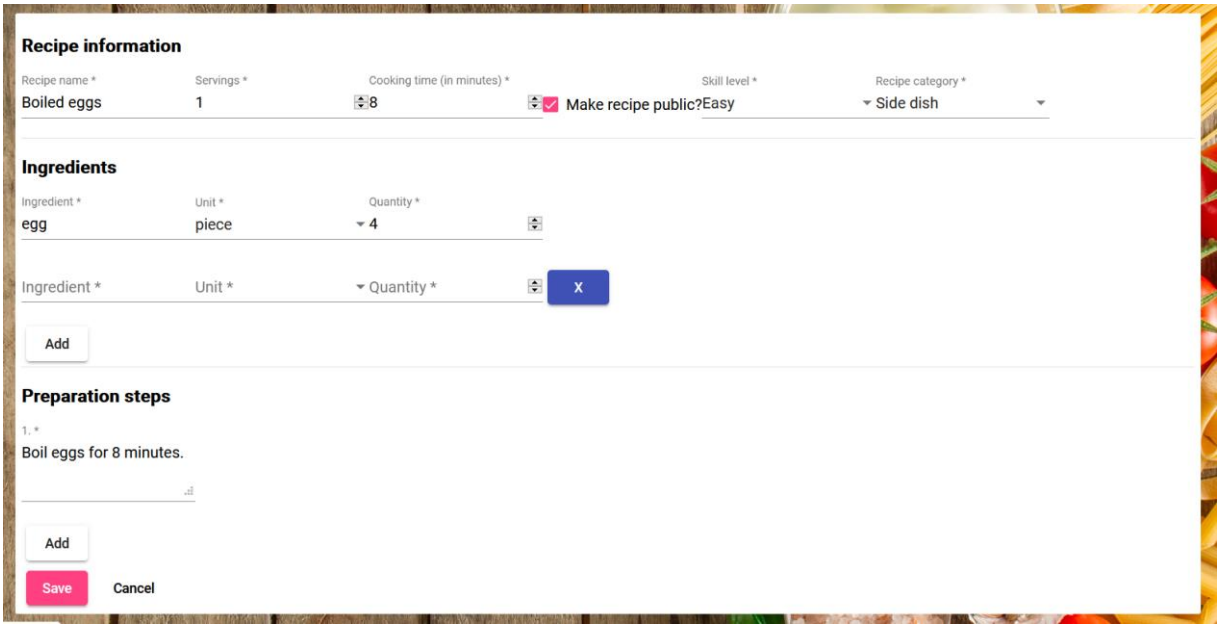
Pregled jednog recept (slika 35) omogućen je kroz odabir recepta na početnoj stranici ili na stranici kuharice. On sadrži sve informacije o receptu, a na toj stranici se nalazi i komponenta za preračunavanje jedinica. Dakle, za prikaz recepta se poziva mikroservis za recepte, dok se za preračunavanje jedinica koristi mikroservis za preračunavanje. Pregled pojedinog recepta omogućuje i prilagodbu broja serviranja. Inicijalno je zamišljeno da ovaj posao obavlja mikroservis, ali se došlo do zaključka da bi mikroservis za preračunavanje trebao obavljati samo zadaću koja se tiče preračunavanja i da je prilagodbu serviranja lakše

obaviti u samoj Angular aplikaciji. Prilagodba omogućuje povećanje ili smanjenje broja osoba za serviranja pri čemu se količina sastojaka prilagođuje broju serviranja. Također je omogućeno i uređivanje ili brisanje recepta klikom na izbornik u kartici recepta. Nadalje, pomoću ikonice kalendara, otvara se dijalog za planiranje obroka. Kod preračunavanja jedinica, dostupna su dva konvertera, jedan za masu i volumen, a drugi za temperaturu. Logički su odvojeni jer se prvi odnosi na količinu sastojaka, a drugi na temperaturu pečenja koja se kao entitet ne sprema u bazi podatak, već se spominje u tekstu pripreme.



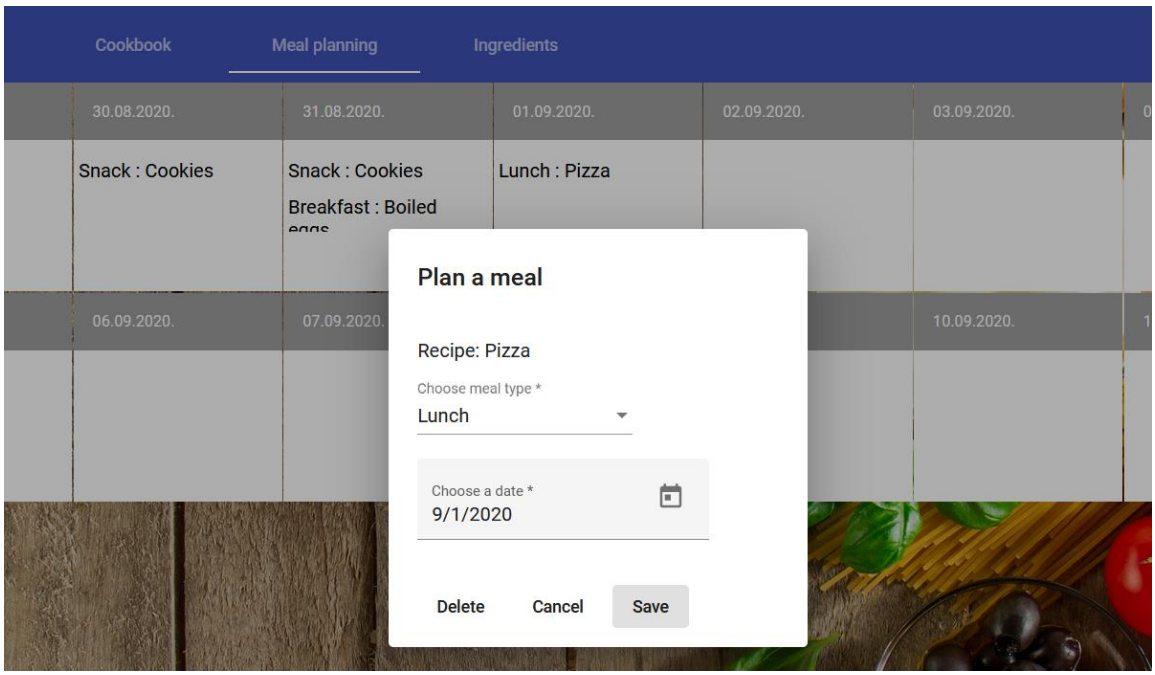
Slika 35 Pregled recepta s preračunavanjem jedinica

Za unos i uređivanje recepta koristi se ista komponenta, ali se u prvom slučaju inicijalizira prazna formu, dok se u drugom forma inicijalizira s podacima recepta. Na slici 36 je prikazan unos recepta koji uključuje unos osnovnih podatka o receptu te unos sastojaka i koraka pripreme koji se mogu brisati i dodavati po potrebi.



Slika 36 Forma za unos i uređivanje recepta

Na stranici za pregled obroka, prikazuju se obroci planirani unutar dva tjedna. Prvo je naveden tip obroka, a zatim naziv recepta. Klikom na obrok otvara se dijalog koji omogućuje uređivanje obroka – mijenjanje tipa ili datum obroka ili pak brisanje. Izgled ove funkcionalnosti prikazan je slikom 37.



Slika 37 Pregled i uređivanje obroka

Osim glavnih funkcionalnosti, aplikacija omogućuje prijavu i registraciju čiji izgled nije prikazan, a nudi i pregled vlastitih recepata te uređivanje vlastitog profila. Administrator, uz

sve to, može i pregledavati sve sastojke u tablici te može unositi nove. Običnim korisnicima to nije dopušteno.

6.5. Kontejneri i Azure servisi

Kako se u praksi mikroservisna arhitektura veže uz kontejnere, implementirani mikroservisi su također kontejnerizirani. Visual Studio nudi izvrsnu podršku za Docker pa su uz pomoć tog razvojnog alata generirane datoteke koje se nazivaju Dockerfile, a koje se koriste za izgradnju slike (eng. *Docker image*). Generirana Docker datoteka koja definira isporuku mikroservisa RecipeMicroservice izgleda ovako:

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.0-buster-slim AS base
WORKDIR /app
EXPOSE 80

ENV ConnectionStrings:DefaultConnection="Server=(localdb)\\MSSQLLocalDB;
Database=RecipeDb;Trusted_Connection=True;MultipleActiveResultSets=true"

FROM mcr.microsoft.com/dotnet/core/sdk:3.0-buster AS build
WORKDIR /src
COPY ["RecipeMicroserviceAPI/RecipeMicroserviceAPI.csproj",
"RecipeMicroserviceAPI/"]
RUN dotnet restore "RecipeMicroserviceAPI/RecipeMicroserviceAPI.csproj"
COPY . .
WORKDIR "/src/RecipeMicroserviceAPI"
RUN dotnet build "RecipeMicroserviceAPI.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "RecipeMicroserviceAPI.csproj" -c Release -o
/app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "RecipeMicroserviceAPI.dll"]
```

Datoteka iz primjera nije sadržavala varijable okoline, pa je dodana varijabla koja definira tekst za spajanje na bazu podataka pomoću ključne riječi `ENV`. Pomoću Docker datoteke se zatim izgradi slika. Navedeno se može izvršiti pomoću naredbi u komandnom retku ili također pomoću Visual Studio alata prilikom objave/isporuke aplikacije. Kad je slika izgrađena, ona se može pokrenuti u Docker kontejneru ili pak postaviti u Docker Hub kako bi bila javno dostupna na Internetu. Ovakva slika predstavlja produkcijsku okolinu, međutim, kod pokretanja servisa iz alata Visual Studio, alat izgradi novu sliku koja je označena kao razvojna te se servis izvršava u kontejneru. Prilikom objave (eng. *publish*) projekta može se isporučiti projekt kao slika u registar kontejnera, što je u ovom slučaju Docker Hub. Visual Studio zapravo sam odrađuje izgradnju, označavanje i objavu slike na repozitorij što omogućava iznimno jednostavnu izradu Docker slika. S ovako izgrađenom sliku znatno je

olakšano pokretanje servisa jer ono ne zahtijeva od korisnika da instalira potrebne pakete ili izvrši migracije nad bazom jer su ovi koraci zapravo već određeni pomoću Docker datoteke.

Nakon što je slika dostupna na repozitoriju, mogu je pronaći i koristiti alati koji izvršavaju slike u kontejnerima. Jedan od njih je i Kubernetes koji je kao usluga dostupan na Azure portalu. Najprije je potrebno kreirati klaster, na koji se potom isporučuju servisi. Za isporuku je potrebna .yaml datoteka koja opisuje isporuku servisa. Primjer datoteke koja se postavlja na klaster u svrhu isporuke mikroservisa za recept izgleda ovako:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: recipemicroservice
spec:
  replicas: 1
  selector:
    matchLabels:
      app: recipemicroservice
  template:
    metadata:
      labels:
        app: recipemicroservice
    spec:
      containers:
      - name: recipemicroservice
        image: mpresecki/recipemicroserviceapi:latest
        ports:
        - containerPort: 80
        env:
        - name: ASPNETCORE_URLS
          value: http://*:80
        - name: AppSettings__UserAPI
          value: http://usermicroservice/api/User
---
apiVersion: v1
kind: Service
metadata:
  name: recipemicroservice
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: recipemicroservice
```

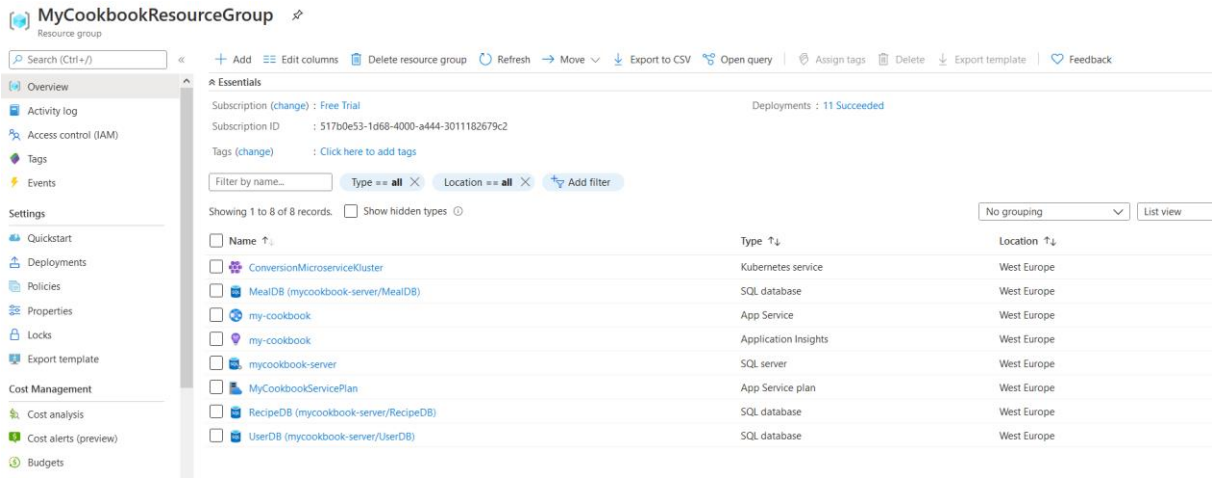
Datoteka sadržava ime novog servisa te lokaciju slike na DockerHubu. Osim toga, dodane su joj i varijable okoline za pristup mikroservisu UserMicroservise pri čemu je važno napomenuti da ukoliko se komunicira sa servisom koji se izvršava u istom klasteru, ne navodi se njegova IP adresa, već njegovo ime: `http://usermicroservice/api/User`. Ako servis promijeni adresu, klaster će ga sam pronaći prema imenu te preusmjeriti na pravilnu adresu. Isporuka se radi preko naredbenog retka pomoću naredbe:

```
kubectl apply -f deploy-mealmicroservice.yaml
```

Nakon toga je servis isporučen i dostupan za upotrebu. Moguće ga je lako skalirati pomoću naredbe:

```
kubectl scale --replicas=3 deployment/recipemicroservice
```

Nakon što su svi servisi isporučeni, bilo je potrebno pripremiti Angular aplikaciju za isporuku. Ona se ne izvršava u kontejneru već je samo trebalo izvršiti izgradnju aplikaciju za produkcijsku okolinu, na Azure portalu napraviti resurs App Service te je prenijeti datoteke /dist mape na virtualni server za web aplikaciju. Popis svih potrebnih Azure resursa za isporuku aplikacije MyCookbook je prikazan slikom 38.



Slika 38 Potrebni Azure resursi za sustav s mikroservisnom arhitekturom

Iako vršenje navedenih koraka zvuči jednostavno, isporuka i usklađivanje svih resursa je daleko kompliciranije i zahtijeva vremena, naročito ako se osoba susreće prvi puta s time. Važno je pravilno postaviti sve varijable okoline, izvršiti izgradnju za odgovarajuću produkciju, misliti na HTTP i HTTPS protokole te na politiku međusobne podjele resursa (eng. *Cross-Origin Resource Sharing*, kraće CORS).

7. Zaključak

Mikroservisna arhitektura postaje sve popularnija te se sve više koristi u praksi, i to najčešće kada se monolitna aplikacija rastavlja na manje dijelove. Ovakva arhitektura donosi brojne prednosti: mikroservisi su lakši za održavanje, međusobno su neovisni pa više timova može razvijati više servisa te ih neovisno i isporučivati. Takvi servisi se mogu i skalirati pomoću repliciranja instanci na poslužitelju što smanjuje opterećenje nekog servisa ukoliko je prenatrpan zahtjevima. Osim toga, svaki servis može koristiti alate koji su najprikladniji za zadaću koju treba obaviti. U usporedbi sa servisno-orijentiranom arhitekturom, iz koje mikroservisna arhitektura potječe, mikroservisi nude veću neovisnost i modularizaciju te su bolje prilagođeni za kontinuiranu isporuku na različite okoline.

Prilikom razvoja aplikacija sa mikroservisnom arhitekturom važno je odrediti veličinu i broj mikroservisa. Nadalje, potrebno je implementirati način komunikacije sa ostalim servisima, a na sve navedeno utječe i organizacijska struktura te komunikacija među timovima. Uobičajeno je da svaki servis koristi vlastitu bazu podataka, pri čemu je dopušteno čak i dupliciranje podataka, te da je zadužen za samo jednu ili dio funkcionalnosti.

Kap praktični dio ovog rada razvijena je web aplikacija čiji rad se temelji na mikroservisima. Aplikacija implementira najvažnije koncepte mikroservisa: međusobna neovisnost te mogućnost neovisne isporuke i skaliranja. Neki koncepti su prirodno ugrađeni, primjerice, uzorak pregrada koji se koristi kao dio dizajna sustava u slučaju greške. Većina ostalih koncepata se odabire na temelju potreba razvojnog tima: oblik komunikacije s klijentima i među servisima. Aplikacija za komunikaciju s klijentima koristi direktne HTTP pozive prema svakom servisu, bez upotrebe API gateway-a, dok se za komunikaciju među servisima koristi također asinkrona komunikacija putem HTTP poziva. Svaki servis koristi vlastitu bazu podataka te implementira svoje metode za manipulaciju podacima. Svaki mikroservis je moguće izvršavati u kontejneru te isporučiti njegovu sliku na Docker Hub, Kubernetes servis ili drugo. Važno je napomenuti da se u praksi sustav temeljen na mikroservisima zapravo većinom razvija iz već postojećeg sustava te da postoje brojni timovi koji su zaduženi za razne dijelove i funkcionalnosti sustava. Upravo zbog toga je bilo nešto teže odrediti broj mikroservisa i raspodijeliti njihove zadaće jer nije bilo organizacijske strukture te komunikacije među timovima.

Popis literature

[1] Sam Newman, *Building Microservices: Designing Fine-Grained Systems*.: O'Reilly Media Inc., 2015.

[2] Johannes Thönes, "Microservices," *IEEE Software*, vol. 32, pp. 113-116, 2015.

[3] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Software*, vol. 35, pp. 24-35, 2018.

[4] Shahir Daya et al., *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*.: IBM Redbooks, 2016.

[5] Eberhard Wolff, *Microservices: Flexible Software Architecture*.: Addison-Wesley Professional, 2016.

[6] Roland Barcia, Kyle Brown, and Richard Osowski. (2018) Microservices point of view guide: Understanding microservices. [Online]. https://www.ibm.com/downloads/cas/ORNMYNRW

[7] Cesar de la Torre, Bill Wagner, and Mike Rousos. (2020).NET Microservices: Architecture for Containerized.NET Applications. [Online]. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/

[8] James Lewis and Martin Fowler. (2014) Microservices. [Online]. https://www.martinfowler.com/articles/microservices.html

[9] Neal Ford, Rebecca Parsons, and Patrick Kua, *Building evolutionary architectures : support constant change*. Beijing: O'Reilly, 2017.

[10] Microsoft. (2020) Visual Studio 2019. [Online]. https://visualstudio.microsoft.com/vs/

[11] Microsoft. (2020) Visual Studio Code. [Online]. https://code.visualstudio.com/?wt.mc_id=DX_841432

[12] Google. (2020) Angular: Features & Benefits. [Online]. https://angular.io/features

[13] Microsoft. (2019) What is SQL Server Management Studio (SSMS)? [Online]. https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15

[14] Microsoft. (2016) Entity Framework Core. [Online]. https://docs.microsoft.com/en-us/ef/core/

[15] Microsoft. (2020).NET Tutorial - Deploy a microservice to Azure. [Online]. https://dotnet.microsoft.com/learn/aspnet/deploy-microservice-tutorial/intro

Popis slika