Primjena mikroservisne arhitekture u razvoju softvera

Krešić, Toni

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Toni Krešić

PRIMJENA MIKROSERVISNE ARHITEKTURE U RAZVOJU SOFTVERA

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Toni Krešić

Matični broj: 47228/18-I

Studij: Informacijsko i programsko inženjerstvo

PRIMJENA MIKROSERVISNE ARHITEKTURE U RAZVOJU SOFTVERA

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, srpanj 2021.

Toni Krešić

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj rad se sastoji od teoretskog i praktičnog dijela koji opisuju i demonstriraju mikroservisnu arhitekturu kao jednu od vrsta arhitekture koja se može koristiti za razvoj softvera. Teoretski dio rada počinje s opisivanjem razvoja softvera što uključuje najčešće korištene metodologije, pristupe i okvire u industriji. Zatim se objašnjava mikroservisna ali i ostale često korištene arhitekture koje se mogu koristiti za razvoj softvera od kojih je najvažnija monolitna, arhitektura koja je izravna suprotnost mikroservisnoj arhitekturi. Nakon toga se prelazi na glavni dio rada koji se bavi mikroservisnom arhitekturom i tu se opisuju bitne karakteristike, prednosti i nedostaci mikroservisa s objektivnog stajališta. Nadalje, opisuju se najčešće prakse, tehnologije i alati koji se koriste za razvoj softvera u obliku mikroservisa što uključuje integraciju, skalabilnost, testiranje, sigurnost, isporuku, nadzor i kontejnerizaciju mikroservisa. Osim toga, opisane su i najčešće korištene strategije prilikom tranzicije iz monolitne u mikroservisnu arhitekturu. Praktični dio rada demonstrira neke od odlika mikroservisa koje su opisane u glavnom dijelu teoretskog rada. Cilj rada je pokazati važnost mikroservisa i predstaviti zašto mikroservisna arhitektura danas postaje omiljena za izradu softvera, neovisno o tome radi li se o jednostavnom ili kompleksnom sustavu. Rad je izrađen pod okriljem Laboratorija za Web arhitekture, tehnologije, servise i sučelja.

Ključne riječi: aplikacija; web; razvoj; arhitektura; softver; docker; kontejner; mikroservis

Sadržaj

1. Uvod

Današnja poslovanja više nisu zamisliva bez nekog oblika korištenja tehnologije. Štoviše, što je poslovanje većih razmjera to je podrška poslovanja putem korištenja tehnologije i softvera nužnija. No, ne radi se samo o poslovanju nego i o privatnom životu. Većina ljudi je svakodnevno okružena tehnologijom i pod utjecajem softvera, naročito putem pametnih telefona i računala. Iza sve te tehnologije u industriji softvera postoje razne metodologije i prakse koje se koriste i mijenjaju s vremenom. Prilikom izrade softvera, specifičnije prije početka same izrade, potrebno je odrediti način na koji će se softver izgraditi i kako će funkcionirati. Upravo to donekle predstavlja arhitekturu softvera, a tema ovog rada je jedna od mogućih arhitektura koja se može koristiti za izradu softvera, a to je mikroservisna arhitektura.

Mikroservisna arhitektura softvera se sastoji od manjih servisa koji se zovu mikroservisi, a svaki mikroservis je neovisna aplikacija i ima svoju specifičnu svrhu. Mikroservisi rade u pozadini i kada je određenoj aplikaciji potrebno dohvatiti neke podatke koje korisnik zatražuje, ona komunicira s mikroservisom čija domena obuhvaća tražene podatke. U slučaju da domena određenog mikroservisa ne obuhvaća sve podatke potrebne za izvršavanje vlastite poslovne funkcije, tada ih on zatražuje od mikroservisa čija domena pokriva tražene informacije. Upravo tako funkcionira mikroservisna arhitektura, kao skupina mikroservisa odnosno manjih pozadinskih aplikacija od kojih svaka ima svoju svrhu i one zajedno čine cjelokupni pozadinski ekosustav određenog softvera. Dakle, mikroservisi čine pozadinski sustav, a korisnički dio najčešće predstavlja neka mrežna (eng. *web*), mobilna ili stolna aplikacija koja koristi mikroservise kao izvor podataka i mogućih radnji.

Ovakva arhitektura je potpuna suprotnost od monolitne arhitekture koja se uvelike koristila do prvog desetljeća 21. stoljeća. Motivacija za temu rada je došla upravo iz učestalosti današnjeg korištenja mikroservisa kao arhitekture softvera. Dok je nekada monolitna arhitektura softvera predstavljala prvi izbor prilikom donošenja odluke o vrsti arhitekture koja će činiti okosnicu budućeg softvera i nije se mnogo razmišljalo o ostalim mogućim alternativama, danas to predstavljaju mikroservisi i zbog toga su oni danas toliko važni u programskom inženjerstvu. Poslovanja uvijek nastoje pronaći najbolje načine za što brži razvoj i isporuku kvalitetnog softvera, a mikroservisi im to i omogućavaju zbog jedinstvenih karakteristika specifičnih njima.

2. Metode i tehnike rada

Pri izradi praktičnog dijela rada korištene su sljedeće tehnologije i alati:

- Android Studio 4.2.1 – integrirano razvojno okruženje za razvijanje mobilnih, stolnih i mrežnih aplikacija.
- DataGrip 2020.2.3 – integrirano razvojno okruženje za upravljanje bazama podataka.
- PhpStorm 2020.3.3 – integrirano razvojno okruženje za razvijanje PHP aplikacija.
- IntelliJ IDEA Ultimate 2020.3.3 – integrirano razvojno okruženje za razvijanje Java aplikacija.
- Flutter 2.3.0 – radni okvir za razvoj mobilnih, stolnih i mrežnih aplikacija.
- Dart 2.14.0 – programski jezik korišten u Flutter radnom okviru.
- Lumen 8.0 – PHP mikro-radni okvir za razvoj mikroservisa temeljenih na Laravel radnom okviru.
- PHP 7.4 – programski jezik korišten u Lumen radnom okviru.
- Spring Boot – ekstenzija Spring radnog okvira za razvoj Java aplikacija.
- Java 1.8.0_241 – programski jezik korišten u Spring Boot radnom okviru.
- Kotlin 1.5.10 – programski jezik korišten u Spring Boot radnom okviru koji je u potpunosti interoperabilan s Javom.
- PostgreSQL – objektno-relacijski sustav baza podataka.
- Auth0 – autentikacijska i autorizacijska platforma.
- Docker 19.03.12 – alat za kontejnerizaciju aplikacija.
- https://app.diagrams.net – mrežni alat za izradu dijagrama.

Razvoj cjelokupnog praktičnog dijela rada se odvijao na Ubuntu 18.04.4 operacijskom sustavu.

3. Razvoj softvera

Kako tehnologija s vremenom postaje sve naprednija, mijenjaju se i alati i metode koje se koriste u razvoju softvera. Novi programski jezici i radni okviri (eng. *frameworks*) se sve češće pojavljuju i zamjenjuju dotadašnje aktualne alate i metode za razvoj softvera. U ovom poglavlju predstavit će se neke osnovne definicije uz sam softver i razvoj softvera, zatim će se opisati uobičajeni koraci prilikom razvoja softvera, te će se obrazložiti najčešće korištene metodologije, pristupi i okviri razvoja softvera.

3.1. Osnovne definicije

Softver je danas sveprisutan. Iako većina ljudi pri spomenu riječi softver misli na programe koje se koriste na stolnim računalima, softver osim stolnih aplikacija uključuje i aplikacije koje se koriste na mrežnim stranicama, mobilnim telefonima ali i u ugrađenom softveru poput aplikacija sadržanih na pametnih satova. Opširna definicija softvera bi glasila da je softver skup uputa koje računalu govore što treba učiniti i generalno postoje četiri vrste softvera [1]:

- Sistemski softver (eng. *system software*) – pružanje osnovnih funkcija poput komunikacije s operacijskim sustavom (ili kraće OS), upravljanjem diskom i hardverom.
- Softver za programiranje (eng. *programming software*) – alati poput razvojnih okruženja, uređivača teksta i kompilatora koje koriste programeri.
- Aplikacijski softver (eng. *application software*) ili kraće aplikacije – služe korisnicima za izvršavanje zadataka, razonodu i slično.
- Ugrađeni softver (eng. *embedded software*) – koristi se za upravljanje strojevima i uređajima poput automobila i robota.

Ovisno o vrsti softvera i poslovnim ali i privatnim ciljevima postoje drukčiji pristupi i metode koje se koriste za razvoj softvera. Generalno gledajući razvoj softvera (eng. *software development*) se može definirati kao skup informatičkih aktivnosti posvećenih procesu stvaranja, dizajniranja, isporuke i podrške softvera [1]. Proces razvoja softvera također najčešće uključuje dokumentiranje, testiranje, uklanjanje grešaka (eng. *bugs*) i održavanje samog softvera.

Razvojem softvera primarno se bave programeri (eng. *programmers*), softverski inženjeri (eng. *software engineers*) i programeri softvera (eng. *software developers*) [1]. Te

uloge međusobno se preklapaju, posebice u hrvatskom jeziku, a dinamika između njih uvelike se razlikuje u različitim organizacijama i odjelima za razvoj.

Nadalje, treba imati na umu da razvoj softvera nije ograničen na programere, softverske inženjere i programere softvera. Znanstvenici, proizvođači uređaja i hardvera također pišu programski kôd iako nisu prvenstveno programeri [1].

3.2. Koraci procesa razvoja softvera

Koraci procesa razvoja softvera su varijabilni i ovise o više faktora poput vrste softvera, namjene, odabrane metodologije i slično. Najčešći koraci koji se obavljaju kod razvoja softvera (ne nužno navedenim redoslijedom) su [1]:

- Odabir metodologije za uspostavu radnog okvira u kojem se primjenjuju koraci razvoja softvera.
- Prikupljanje zahtjeva za razumijevanje i dokumentiranje onog što je potrebno krajnjim korisnicima i dionicima tvrtke.
- Odabir (ili pak rjeđe vlastita izgradnja) arhitekture softvera.
- Razvoj dizajna softvera odnosno načina na kojeg će softver rješavati probleme koji su doneseni zahtjevima. Misli se na širu sliku softvera i najčešće se crtaju razni dijagrami kako bi se lakše razumjela generalna ideja.
- Pisanje programskog kôda u odabranom programskom jeziku koji najpraktičnije rješava probleme donesene korisničkim zahtjevima.
- Testiranje s unaprijed planiranim scenarijima. Testovi se mogu uvesti nakon izgradnje logike oko poslovnih zahtjeva, no ponekad se koristi i metoda razvoja softvera korištenjem testova (eng. *test driven development* ili kraće TDD) što znači da se testovi pišu dok se razvija poslovna logika softvera i konstantno se ažuriraju u skladu s dodavanjem novih funkcionalnosti.
- Upravljanje konfiguracijom i nedostacima softvera pri čemu se misli na pravilno verzioniranje softvera u svrhu osiguranja kvalitete i adresiranja trenutačnih nedostataka i/ili grešaka u softveru.
- Isporuka softvera i rješavanje problema nastalih od strane korištenja softvera krajnjih korisnika.
- Migriranje podataka na novi ili ažurirani softver iz postojećeg sustava i izvora podataka (ako su postojali).
- Upravljanje i mjerenje projekta u svrhu održavanja kvalitete i kontinuirane isporuke softvera tijekom njegovog životnog ciklusa.

Ponekad se određeni koraci ne prakticiraju ili su jasni sami po sebi. Primjerice, današnja popularna metodologija agilnog pristupa razvoja softvera često ne uvjetuje pretjerano pisanje dokumentacije ili testova s unaprijed planiranim scenarijima. Opisani koraci procesa razvoja softvera uklapaju se u faze životnog ciklusa softvera (eng. *software development life cycle* ili kraće SDLC), a sam SDLC se sastoji od sljedećih faza [1]:

- Analiza i specifikacija zahtjeva.
- Dizajn i razvoj.
- Testiranje.
- Isporuka.
- Održavanje i podrška.

Dizajn i razvoj su ponekad odvojene faze, a ponekad se stapaju u istu, ovisno o autoru. Prethodno opisani koraci procesa razvoja softvera se mogu grupirati u faze životnog ciklusa softvera, ali važnost životnog ciklusa softvera je ta da je on neprestan proces koji se nakon faze održavanja i podrške ponavlja kako bi se omogućilo kontinuirano poboljšavanje softvera. Primjerice, neisporučeni korisnički zahtjevi ili greške koje se pronađu u fazi održavanja i podrške mogu postati zahtjevi na početku sljedećeg ciklusa.

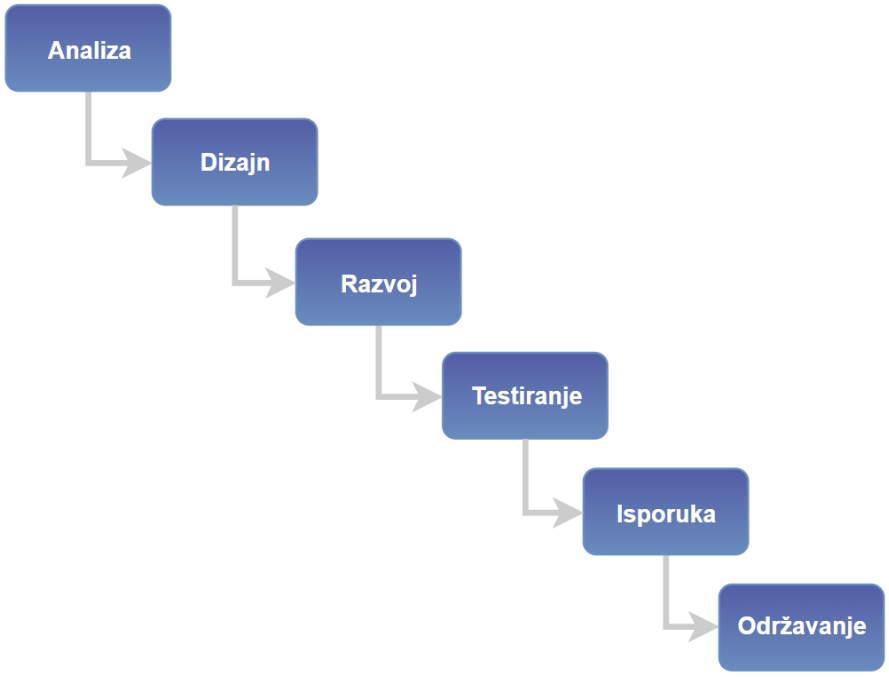
3.3. Metodologije, pristupi i radni okviri

Najčešći prvi korak prije razvijanja softvera je odabir metodologije razvoja softvera ovisno o ciljevima i dostupnim resursima. Odabir metodologije predstavlja radni okvir u kojem se koraci razvoja softvera primjenjuju, a sama metodologija najčešće opisuje cjelokupni radni plan ili hodogram rada za razvoj softvera u okviru životnog ciklusa softvera. Drugim riječima, radi se o skupu načela, metoda i ideja koje određuju način na koji će se razvijati softver. Razne metodologije imaju slične elemente i mnoge su nastale kao odgovor na nedostatke do tada primarno korištene metodologije stoga one načelno samo savjetuju, a ne nalažu što i kako činiti.

3.3.1. Vodopadni pristup

Vodopadni pristup razvoju softvera je najstariji pristup i smatra se tradicionalnom metodologijom razvoja, a čini ga skup kaskadnih linearnih koraka tj. faza od planiranja i prikupljanja zahtjeva do isporuke i održavanja sustava. Kod vodopadnog modela prolazak faza je slijedan i uobičajeno nisu dozvoljene naknadne promjene rezultata faze koja je završena. Ovakva metodologija je prikladna za velike projekte gdje su zahtjevi dobro definirani i općenito se ne očekuju posebne promjene u samim zahtjevima. S obzirom na to da vraćanje na

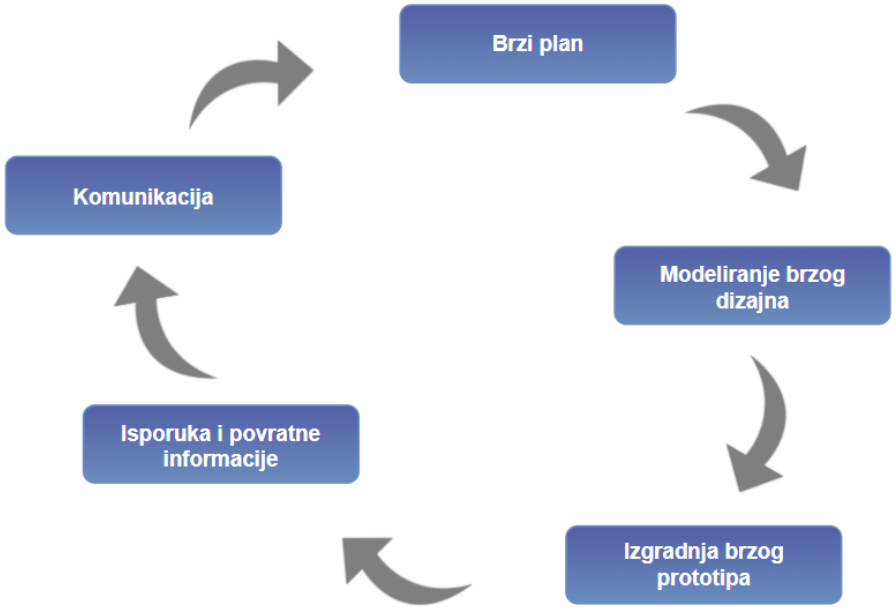
prethodne faze obično nije dopušteno jer je poprilično skupo očekuje se da je svaka faza dobro isplanirana prije nego li se počne provoditi [2, str. 41].



Slika 1. Faze vodopadnog modela (Prema: [2, str. 42])

3.3.2. Prototipiranje

Prototipiranje je metodologija razvoja softvera u kojoj se prije razvijanja stvarnog sustava izrađuje prototipna verzija proizvoda. Izgradnja prototipa omogućava demonstraciju funkcionalnosti budućeg sustava klijentima što povećava njihovo razumijevanje sustava i osigurava veću razinu zadovoljstva i povjerenja prije nego li se krene u realizaciju stvarnog projekta. Nadalje, ovakav pristup razvoju softvera značajno umanjuje rizik od neuspjeha jer se već u ranoj fazi mogu prepoznati potencijalni problemi koji bi se mogli dogoditi tijekom razvoja projekta i samim time se omogućava poduzimanje odgovarajućih koraka za rješavanje istih. Nedostaci ovakvog pristupa su veći troškovi, velik utjecaj klijenta u razvoju sustava što može usporiti općeniti razvojni proces i povećanje očekivanja klijenta koje se inače javlja nakon izrade prototipa jer klijent tada očekuje da se stvarni sustav isporuči brže. Nadalje, izrada prototipa također podrazumijeva preuzimanje rizika nezadovoljstva klijenata nakon što vide početni prototip [2, str. 45].



Slika 2. Paradigma prototipskog razvoja (Prema: [2, str. 46])

3.3.3. Rapidni razvoj aplikacija

Rapidni razvoj aplikacija (eng. *rapid application development* ili kraće RAD) je učinkovita metodologija razvoja softvera koja omogućava mnogo brži razvoj krajnjeg proizvoda izradom prototipa kroz iterativni razvoj. Ova metodologija ima dugu povijest i razne varijante, no ovdje je fokus stavljen na Martinovu metodologiju predloženu 1991. godine koja ima četiri faze:

- Faza planiranja zahtjeva – ova faza kombinira elemente planiranja i analize sustava SDLC-a i u njoj korisnici, menadžeri i informacijsko-komunikacijsko (ili kraće IT) osoblje dogovaraju poslovne ciljeve, opseg projekta i ograničenja i zahtjeve sustava, a faza završava kada se svi slože oko ključnih pitanja.
- Faza korisničkog dizajna – tijekom ove faze korisnici isprobavaju sustav, a razvojni inženjeri osmišljavaju modele i prototipe koji bi najbolje reprezentirali ono što korisnici trebaju. Ovo je kontinuirana faza koja korisnicima omogućava da shvate, mijenjaju i na kraju odobre prototipni model sustava koji zadovoljava njihove potrebe.
- Faza izgradnje – fokusira se na razvoj sustava, a faza također sadrži integraciju komponenti i testiranje. Korisnici i dalje sudjeluju i predlažu promjene ili poboljšanja paralelno s razvojem prototipa sustava.
- Faza prekida – faza u kojoj se implementira i testira sustav i provodi obuka korisnika dok se softver isporučuje u rad.

Ključna stvar ove metodologije je ta da je cijeli proces razvoja softvera komprimiran tj. ubrzan u usporedbi s tradicionalnim pristupima. Programeri rade u brzim iteracijama koje skraćuju vrijeme razvoja i ubrzavaju isporuku konačnog proizvoda. RAD metodologija naglašava upotrebu softvera i korištenje povratnih informacija korisnika umjesto strogog planiranja i prikupljanja zahtjeva kao u vodopadnom modelu. Prednosti RAD metodologije su poboljšana fleksibilnost i prilagodljivost gdje se programeri mogu brzo posvetiti novim zahtjevima tijekom razvojnog procesa. Gledano iz perspektive kupaca, klijenata i korisnika zadovoljstvo je općenito veće zbog visoke suradnje između svih dionika. Finalno, RAD metodologija donosi manje iznenađenja jer u odnosu na vodopadni pristup RAD uključuje integraciju proizvoda rano u procesu razvoja sustava [3].

3.3.4. Spiralni model

Spiralni model razvoja softvera je vjerojatno najpoznatiji po svom naglasku na upravljanju rizika. Ime je dobio po svom dijagramskom prikazu gdje izgleda kao spirala s više petlji. Svaka petlja predstavlja jednu fazu projekta, a broj petlji odnosno faza projekta je varijabilan i to odlučuje voditelj projekta ovisno o projektnim rizicima. Svaka faza spiralnog modela je podijeljena u četiri kvadranta i svaki kvadrant ima svoju funkciju:

1. Planiranje – prva faza koja uključuje procjenu troškova i resursa, definira raspored razvoja i određuje ciljeve.
2. Analiza rizika – druga faza koja identificira potencijalne rizike i određuje se najbolje moguće rješenje za uklanjanje pronađenih rizika.
3. Implementacija – treća faza koja uključuje testiranje, programiranje i isporuku prototipa.
4. Evaluacija – četvrta faza koja uključuje procjenu softvera od strane kupca i planiranje za sljedeću fazu tj. sljedeću iteraciju ciklusa.

Fokus spiralnog modela je na procjeni rizika softvera razdvajanjem cjelokupnog procesa razvoja na manje segmente. Evaluacijom rizika manjih segmenata projekta lakše je primijeniti promjene tijekom razvojnog procesa, a moguće je i razmatranje isplativosti nastavka razvoja projekta. S druge strane, spiralni model nije praktičan za manje projekte jer je skuplji i puno kompleksniji u odnosu na druge modele [2, str. 47].

3.3.5. *Scrum*

Za razliku od prethodno navedenih modela i metodologija, *Scrum* predstavlja radni okvir upravljanja procesom razvoja proizvoda, najčešće softvera (može se primijeniti i u drugim djelatnostima) koji uključuje skup pravila i sljedeće predefiniranih uloga u timu [4, str. 4]:

- Vlasnik proizvoda (eng. *product owner*) – osoba odgovorna za maksimiziranje vrijednosti proizvoda koji je rezultat rada *Scrum* tima, a način na koji vlasnik proizvoda to odrađuje je varijabilan.
- *Scrum master* – osoba odgovorna za ispravno izvođenje *Scrum* praksi kako su navedene u *Scrum* vodiču tako da pomaže članovima tima ali i organizaciji u razumijevanju *Scrum* teorije i prakse.
- Programeri – ljudi u *Scrum* timu koji su predani stvaranju bilo kojeg aspekta korisnog čina u sprintu (eng. *sprint*).

U *Scrum* radnom okviru postoji pet događaja [4, str. 7]:

- Sprint – predstavlja srž *Scruma* gdje se ideje pretvaraju u vrijednost, a označava događaj fiksne duljine od mjesec dana ili manje (najčešće dva tjedna). Novi sprint započinje odmah nakon završetka prethodnog, a fiksne je duljine radi postizanja dosljednosti.
- Planiranje sprinta – pokreće sprint određivanjem posla koji treba obaviti u narednom sprintu, a plan se stvara zajedničkim radom cijelog *Scrum* tima.
- Dnevni *Scrum* – predstavlja 15-minutni sastanak za programere *Scrum* tima, a svrha mu je provjeriti napredak prema sprint cilju.
- Pregled sprinta – događaj kada *Scrum* tim predstavlja rezultate svog rada ključnim dionicima i razgovara se o napretku prema cilju proizvoda, a svrha mu je provjeriti ishod sprinta i odrediti buduće prilagodbe.
- Retrospektiva sprinta – događaj kada *Scrum* tim planira način za povećanje kvalitete i učinkovitosti sprinta provjeravanjem tijeka rada posljednjeg sprinta.

Nadalje, u *Scrumu* postoje tri artefakta koja predstavljaju posao ili vrijednost:

- Zaostaci proizvoda (eng. *product backlog*) – lista važnih zadataka poredanih po prioritetu.
- Zaostaci sprinta (eng. *sprint backlog*) – sastoji se od cilja sprinta, zaostataka proizvoda odabranih za sprint i plana za isporuku inkrementa.
- Inkrement (eng. *increment*) – predstavlja konkretan korak prema cilju proizvoda.

Naziv *Scrum* (kratko od *scrummage*) potiče iz ragbi nogometa gdje *Scrum* predstavlja način restartanja igre. Svoje korijene vuče iz 90-tih [2, str. 78], a temelji se na praksi što omogućava neprestanu nadogradnju radnog okvira u skladu s mijenjanjem tehnologije i tehnika razvijanja softvera. Koristi iterativni i inkrementalni pristup za razvoj proizvoda, a fokus radnog okvira je na transparentnosti, inspekciji i adaptaciji [4, str. 3].

3.3.6. Ekstremno programiranje

Ekstremno programiranje (eng. *extreme programming* ili kraće XP) je disciplina razvoja softvera koja uključuje skup najboljih praksi od kojih su neke dovedene „ekstremnu“ razinu, iz čega i potiče naziv same metodologije. Fokus ekstremnog programiranja je usmjeriti cijeli tim na zajedničke i dostižne ciljeve uvođenjem odgovornosti i transparentnosti u razvoj softvera, a cilj je stvoriti softver visoke kvalitete. Neke od ključnih praksi koje XP nalaže su [5, str. 13]:

- Česte isporuke softvera, ponekad i na dnevnoj razini.
- Korištenje jediničnog testiranja (eng. *unit testing*).
- Programiranje u paru (eng. *pair programming*).
- Pregled programskog kôda (eng. *code review*).
- Refaktoriranje (eng. *refactoring*).
- Česta komunikacija i povratne informacije s kupcem, ali i između programera.

Ono što je bitno kod XP-a je da su navedene prakse kombinirane tako da komplementiraju, ali i kontroliraju jednu drugu. Nadalje, ekstremno programiranje se lakše prilagođava zahtjevima proizvoda i budžetu jer nalaže da se softver gradi u kraćim razvojnim ciklusima što potencijalne promjene čini lakšim za implementirati, a povratne informacije kupca ažurnijim [2, str. 74].

3.3.7. *Lean*

Lean razvoj predstavlja primjenu *lean* praksi i načela iz industrije proizvodnje u domenu softvera. U kontekstu radnog okvira za razvoj softvera *lean* sadrži sedam principa [6, str. 13]:

- Eliminirati otpad (eng. *eliminate waste*) – princip koji tvrdi da je idealna situacija kupcu napraviti i isporučiti točno ono što on želi i što ranije, a ono što koči tu situaciju je otpad. Otpad predstavlja sve što proizvodu ne dodaje vrijednost u smislu kako ju je kupac zamislio, npr. razvijanje funkcionalnosti koje se neće koristiti, mijenjanje razvojnih timova ili ljudi u timu, prebacivanje zadataka s jedne osobu na drugu i slično.
- Pojačati učenje (eng *Amplify learning*) – princip koji tvrdi da je povećanje količine učenja najbolji pristup poboljšanju okruženja za razvoj softvera. Ovdje se navodi da se proces razvoja softvera najbolji uči kroz iskustvo stoga je u početku bolje isprobavati različite metode i tehnike prilikom razvoja softvera, a kasnije primjenjivati naučeno uz varijacije na temu ovisno o vrsti projekta.
- Odlučiti što je kasnije moguće (eng. *decide as late as possible*) – kasno donošenje odluka povećava učinkovitost u neizvjesnim situacijama jer je samim time budućnost bliža i lakše predvidiva. Uz lakše predvidivu budućnost

pogađanja se pretvaraju u činjenice što zahtjeve čini jasnijim i time se smanjuje otpad, skraćuje vrijeme i ušteđuje novac.

- Dostavi što je brže moguće (eng. *deliver as fast as possible*) – iako se pažljivi i sporiji pristup prije činio važnijim, danas se sve više prakticira agilni pristup razvoja softvera jer brzina igra ključan faktor u donošenju odluka i pobjeđivanju konkurenata.
- Osnažiti tim (eng. *empower the team*) – stara izreka govori da detalji čine razliku, a detalje softvera najbolje znaju osobe koje razvijaju softver. Ovaj princip tvrdi da je uključivanje programera u detalje tehničkih odluka ključno za postizanje izvrsnosti jer će tada pod vodstvom vođe donijeti bolje tehničke i procesne odluke nego kada bi takve odluke donosio netko umjesto njih. S obzirom na brzinu isporuke softvera i kasno donošenje odluka, vođa tima nije u stanju sam orkestrirati sve aktivnosti radnika i osnaživanjem tima programeri imaju mogućnost jedni drugima dati do znanja što treba činiti.
- Ugraditi integritet (eng. *build integrity in*) – ovdje se misli na dvije vrste integriteta – percipirani i konceptualni. Percipiranim integritetom se smatra kada je krajnji korisnik u mogućnosti intuitivno koristiti softver, a konceptualni integritet znači da odvojene komponente softvera zajedno djeluju kao jedna kohezivna cjelina.
- Vidjeti cjelinu (eng. *see the whole*) – kako je integritet u softveru bitan i na razini arhitekture potrebno se i usredotočiti na cjelokupnu izvedbu softvera, a ne samo njegove komponente. Što je softver kompleksniji to će više ljudi raditi na njemu i više će dijelova sustava komunicirati jedan s drugim i zato je bitno da je njihova međusobna interakcija glatka, promjenjiva i nadogradiva.

Lean razvoj softvera stavlja veliki fokus na principe, vrijednosti i dobre prakse temeljene na iskustvu, no treba imati na umu da iako će ovaj pristup razvoja softvera funkcionirati za određene tvrtke to ne znači da će imati isti učinak na sve stoga je navedene principe potrebno prilagoditi organizaciji, timu i projektu.

3.3.8. Kanban

Kaban je metoda za upravljanjem procesom rada koja vuče korijene iz *lean* načela i praksi i temelji se na vizualiziranju hodograma rada i trenutno odrađenog rada [7, str. 123]. To se najčešće postiže korištenjem Kanban ploče koja članovima tima služi kako bi dobili uvid u cjelokupni napredak rada što omogućava kontinuirano poboljšanje i maksimiziranje učinkovitosti. Kanban se u razvoju softvera najčešće koristi u kombinaciji s drugim metodama i radnim okvirima poput *Scruma* ili *Leana* [7, str. 58]. Temelj Kanban metode je Kanban ploča na kojoj se nalaze stupci i ti stupci sadrže zadatke. Svaki stupac predstavlja fazu u kojoj se

trenutno nalaze određeni zadaci, a najčešći primjeri stupaca su „Napraviti“, „U tijeku“ i „Odrađeno“. Kada se završi određena faza zadatka, taj zadatak se pomiče u sljedeći stupac. Ono što je bitno je to da je broj zadataka koji su u tijeku limitiran tako da se u isto vrijeme može raditi samo na ograničenom broju zadataka. Takav princip rada olakšava timu da se usredotoče samo na trenutne zadatke što ubrzava njihovo izvršavanje. Zadaci najčešće sadrže informacije poput statusa, opisa, roka i osobe zadužene za odrađivanje istog [7].

4. Vrste arhitekture softvera

Arhitektura softvera predstavlja nacrt za izradu softvera u smislu da određuje fundamentalnu strukturu sustava i definira način na koji će se izgraditi sustav. Odabir arhitekture najčešće predstavlja jednu od prvih odluka koje se donose prilikom izrade softvera jer promjena arhitekture softvera najčešće zahtijeva ponovnu izradu softvera što predstavlja ogroman trošak novca i vremena. U ovom poglavlju bit će objašnjene najčešće arhitekture softvera, njihove karakteristike, prednosti i nedostaci.

Postoje razna mišljenja struke što arhitektura softvera zapravo predstavlja, a sljedeća definicija objedinjuje ta različita gledišta u jednu rečenicu:

„*Softverska arhitektura sustava je skup struktura potrebnih za rasuđivanje o sustavu, koje sadrže softverske elemente, odnose među njima i svojstva oba*.“ [8, str. 4]

Sada se postavlja pitanje: „Koliko ima različitih vrsta arhitekture softvera?“. Neki psiholozi tvrde da se sve karakteristike ljudske osobnosti svijeta mogu grupirati u samo pet dimenzija. S obzirom na to da postoje različita mišljenja o vrstama arhitekture softvera i što sve čini arhitekturu bilo je potrebno ograničiti se na određen izvor, a u ovom radu referencirat će se na pet vrsta arhitekture softvera prema M. Richardsu [9]:

- Slojevita arhitektura (eng. *layered architecture*).
- Arhitektura vođena događajima (eng. *event-driven architecture* ili kraće EDA).
- Mikrojezgrena arhitektura (eng. *microkernel architecture*).
- Mikroservisna arhitektura (eng. *microservices architecture* ili kraće MSA).
- Prostorno-bazirana arhitektura (eng. *space-based architecture* ili kraće SBA).

4.1. Slojevita arhitektura

Slojevita arhitektura (na eng. još poznata kao i *n-tier architecture*) je najčešća vrsta arhitekture softvera. Ovakav tip arhitekture softvera ne navodi koliko slojeva mora postojati, no najčešće se radi o tri ili četiri sloja, ovisno o složenosti sustava. U slučaju da se radi o četiri sloja oni su najčešće:

- Prezentacijski sloj (eng. *presentation layer*).
- Poslovni sloj (eng. *business layer*) – ponekad se naziva slojem domene (eng. *domain layer*).
- Perzistentni sloj (eng. *persistence layer*) – ponekad se naziva slojem pristupa podacima (eng. *data access layer*).

- Podatkovni sloj (eng. *database layer*).

U manjim sustavima perzistentni i poslovni sloj su ponekad kombinirani u jedan sloj poslovne logike i to se najčešće manifestira kada je perzistentni sloj ugrađen u komponente poslovnog sloja. Svaki sloj slojevite arhitekture ima svoju ulogu i odgovornost u sustavu.

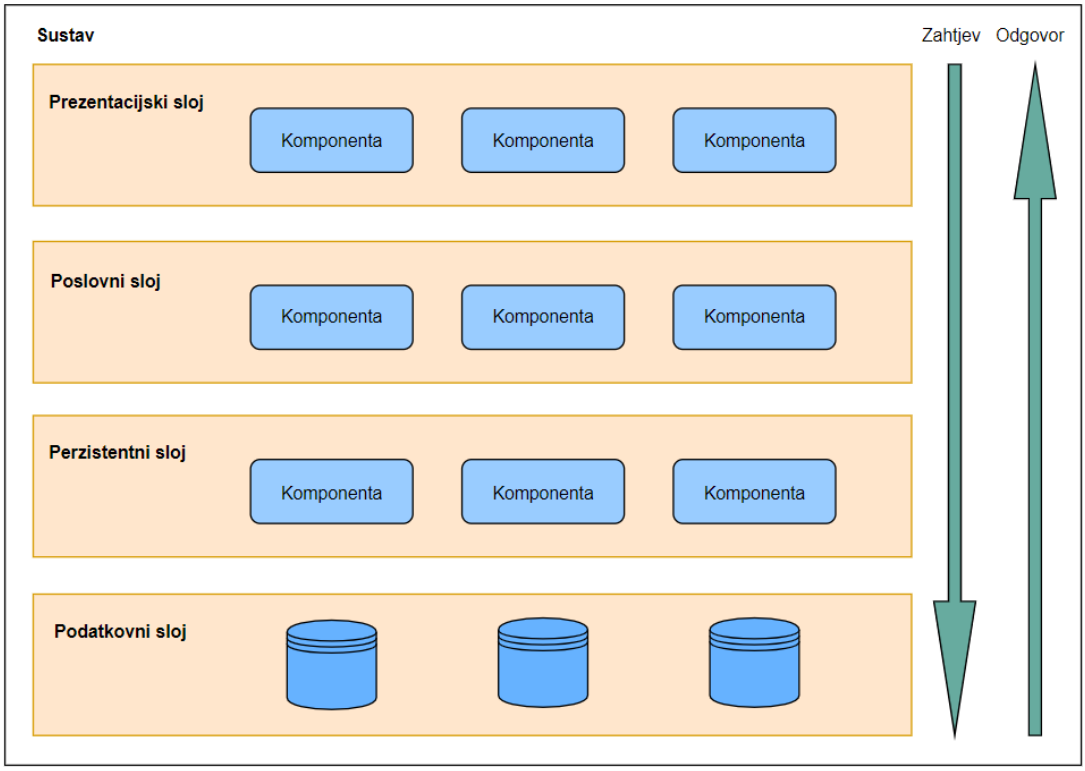
Prezentacijski sloj je zadužen za upravljanje korisničkim sučeljem što znači da prikazuje podatke korisniku i omogućuje interakciju korisnika sa sustavom. Ono što je bitno za prezentacijski sloj je to da se ne mora brinuti o dohvaćanju podataka i o njihovom formatiranju, on brine samo o prikazu podataka.

Sloj poslovne logike ili domene izvršava stvarna poslovna pravila koja određuju kako se podaci mogu stvarati, uređivati i pohranjivati. Stvarna poslovna pravila se odnose na realne poslovne objekte poput korisničkih računa, zaliha, kupovina, zajmova i slično. Primjerice, prilikom kreiranja narudžbe sloj poslovne logike stvara narudžbu u sustavu i izvršava sve ostale stavke koje su potrebne ovisno o elementima i zahtjevima narudžbe poput stvaranja računa, slanja obavijesti korisniku i notifikacije administratoru i slično.

Perzistentni sloj manipulira podacima u podatkovnom sloju što uključuje čitanje, spremanje, ažuriranje i brisanje istih. Naziv dolazi od toga da ovaj sloj čini podatke perzistentnim odnosno stalnim tako da ih zapisuje u spremište podataka (poput baze podataka ili datoteke).

Podatkovni sloj je najčešće baza podataka poput SQL Servera, MySQL-a, PostgreSQL-a, Oracle-a ili SQLite-a koja sadrži sve podatke sustava, no može biti i neko drugo spremište podataka poput datoteke.

Prednosti slojevite arhitekture su lakoća razvoja, apstrakcija slojeva i razdvajanje odgovornosti među njima. Komponente unutar svog sloja izvršavaju samo logiku koja se odnosi na taj sloj, npr. komponenta u sloju poslovne logike nikada neće direktno zatražiti podatke iz podatkovnog sloja već će proslijediti zahtjev perzistentnom sloju koji će izvršiti zahtjev i vratiti odgovor. S druge strane problemi se javljaju kod performansi, skalabilnosti i isporuke softvera jer i najmanja preinaka ili dodavanje nove funkcionalnosti najčešće zahtijeva ponovnu isporuku cijelog sustava što se mora raditi izvan okvira radnog vremena dok se sustav ne koristi [9, str. 8].



Slika 3. Struktura slojevite arhitekture (Prema: [9, str. 2])

4.2. Arhitektura vođena događajima

Arhitektura vođena događajima se bazira na kreiranju, otkrivanju i reakciji na događaje (eng. *events*). Događaj predstavlja bilo koju značajnu promjenu stanja softvera, npr. dodavanje proizvoda u košaricu. Bitno je razumjeti da se kroz sustav prilikom kreiranja događaja propagira poruka tj. obavijest da se događaj dogodio, a ne sam događaj. Ta poruka može nositi sa sobom određene informacije o događaju poput proizvoda kojeg je kupac dodao u košaricu. Ova vrsta arhitekture softvera je popularna za distribuirane, asinkrone i visoko-skalabilne sustave tako da se može koristiti za kompleksne ali i za jednostavne aplikacije [9, str. 11].

Kada se priča o asinkronim događajima i sustavima misli se na događaje koji su neovisni jedan o drugom što znači da jedan događaj nije nužno produkt drugog. Štoviše, proizvođači događaja najčešće ni ne znaju potrošače koji će konzumirati događaj, a ni ishod samog događaja. Uobičajeni tok podataka je taj da se nakon otkrivanja događaja on emitira putem kanala događaja (eng. *event channels*) čija je zadaća propagirati događaj od proizvođača do potrošača. Događaji se emitiraju u obliku zapisa (eng. *records*), a zapisi su obična podatkovna struktura koja se prenosi (npr. proizvod koji se dodao u košaricu). Više zapisa (prenesenih događaja) čine podatkovnu strukturu u koju se mogu dodavati novi zapisi,

a takvi perzistentni zapisi se zovu tokovi podataka (eng. *streams*) i mogu biti spremljeni u datoteku, bazu podataka ili nekom drugom distribuiranom obliku, npr. lancu blokova (eng. *blockchain*). Upravljanje, pristup, spremanje, čitanje događaja iz zapisa i održavanje samih zapisa obavlja posrednik (eng. *broker* ili *mediator*, ovisno o topologiji arhitekture).

Ovakav tip arhitekture se sastoji od visoko odvojenih (eng. *highly decoupled*) jednonamjenskih komponenata koje asinkrono primaju poruke i obrađuju događaje. Iz tog razloga ovakva arhitektura softvera ima visoke performanse, skalabilna je i isporuka je lagana i ograničena samo na komponente koje su se mijenjale. No, s obzirom na to da se radi o odvojenim komponentama koje predstavljaju događaje testiranje cjelokupnog sustava je otežano. Nadalje, razvoj je općenito sporiji zbog veće kompleksnosti sustava koja se javlja uvođenjem posrednika između događaja za razliku od uobičajene zahtjev-odgovor (eng. *request-response*) interakcije [9, str. 18].

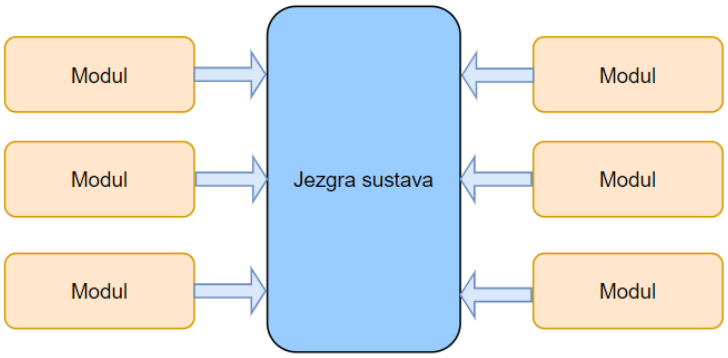
4.3. Mikrojezgrena arhitektura

Mikrojezgrena arhitektura (na eng. još poznata kao i *plug-in architecture*) je vrsta arhitekture koja sadrži jezgru (eng. *core system*) i module koji se mogu priključiti na jezgru (eng. *plug-in modules*). Komponenta jezgre sadrži samo minimalnu funkcionalnost potrebnu za rad sustava, a moduli donose dodatne funkcionalnosti. Naziv ove vrste arhitekture dolazi iz činjenice da mnogi operacijski sustavi implementiraju ovu vrstu arhitekture, a centralni dio operacijskog sustava se zove upravo jezgra (eng. *kernel*) [9, str. 21].

U mikrojezgrenoj arhitekturi jezgra sustava inače nudi samo minimalnu funkcionalnost potrebnu za rad sustava, isto kao i u operacijskim sustavima. Gledano iz poslovne perspektive, jezgra sadrži osnovnu poslovnu logiku (ili sloj) ne uzimajući u obzir posebne slučajeve i pravila ili složenije procese. Dok jezgra sadrži minimalnu funkcionalnost moduli pružaju dodatne mogućnosti ili modificiraju osnovnu funkcionalnost što sustav čini proširivim, ali isto tako odvaja i izolira poslovne funkcije. Moduli u principu funkcioniraju neovisno što znači da ne znaju jedan za drugog i ne komuniciraju međusobno, no moguće je koristiti module za module što se generalno ne preporučuje kako bi se minimalizirali problemi s ovisnošću modula.

Mnogo softvera danas koristi module koji donose dodatnu funkcionalnosti, mijenjaju ili nadjačavaju postojeću. Jedan primjer su mrežni preglednici gdje je moguće instalirati proširenja (eng. *extensions*) koja primjerice ispravljaju gramatičke greške korisnika tijekom unosa teksta. Drugi primjer su integrirana razvojna okruženja koja nude mogućnost instalacije dodatnih modula koji formatiraju unos, upozoravaju korisnika kada radi neku čestu grešku ili ispisuju dodatne informacije.

Slično kao i kod EDA-e, ovakav tip arhitekture ima visoke performanse i česta isporuka softvera ne predstavlja problem. Suprotno od EDA-e, iako se radi o modulima koji na neki način također predstavljaju odvojene komponente ovdje testiranje cjelokupnog softvera nije problem jer se moduli lako mogu dodati u centralni sustav. Nadalje, ovisno o tome kako se moduli izrade skalabilnost je moguća na određenoj razini, no ovaj uzorak arhitekture nije poznat po tome jer je arhitektura bazirana na proizvodu tj. centralnom sustavu i moduli su ovisni o središnjem dijelu [9, str. 25].



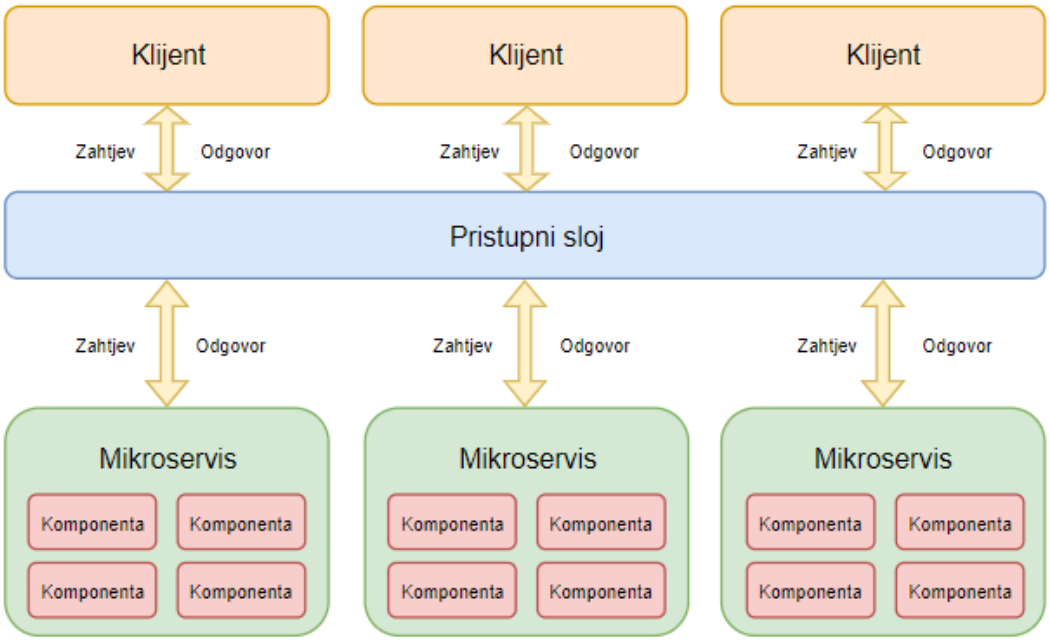
Slika 4. Struktura mikrojezgrene arhitekture (Prema: [9, str. 22])

4.4. Mikroservisna arhitektura

Mikroservisna arhitektura je varijanta servisno-orijentirane arhitekture (eng. *service-oriented architecture* ili kraće SOA) koja funkcionira tako da sustav organizira kao skup labavo povezanih (eng. *loosely coupled*) servisa [9, str. 27]. Labava povezanost je svojstvo kada određene komponente softvera (o ovom kontekstu servisi) znaju jako malo ili ništa o internoj implementaciji ostalih komponenata. Kada se govori o servisu zapravo se misli na neovisan modul koji predstavlja manju aplikaciju koja je zadužena za određenu domenu cjelokupnog sustava. U usporedbi s mikrojezgrenom arhitekturom, ovdje nema centralnog dijela sustava na kojeg se priključuju ostali moduli već su svi mikroservisi neovisni (labava povezanost) i međusobno komuniciraju kada je to potrebno. S obzirom na to da su servisi neovisni i labavo povezani radi se o sustavu distribuirane arhitekture (eng. *distributed architecture*), i mikroservisi, kada trebaju razmijeniti određene informacije, komuniciraju putem neke vrste protokola za daljinski pristup (eng. *remote access protocol*) poput *Hypertext Transfer Protocol-a* (ili kraće HTTP) i arhitekturalnog uzorka *Representational State Transfer-a* (ili kraće REST), *Remote Procedure Calla-a* (ili kraće RPC), *Advanced Message Queuing Protocol-a* (ili kraće AMQP), *Simple Text Oriented Message Protocol* (ili kraće STOMP), *Simple Object Access*

Protocol-a (ili kraće SOAP), *Java Messaging Service-a* (ili kraće JMS), *Remote Method Invocation-a* (ili kraće RMI) etc.

Ne postoji jedinstvena definicija što mikroservisna arhitektura točno predstavlja ali postoji nekoliko uobičajenih koncepata koji se vežu uz mikroservise poput autonomnosti, labave povezanosti, organizacije oko poslovnih funkcija i laganih i brzih komunikacijskih mehanizama. Ovakva priroda distribuirane arhitekture mikroservise čini visoko održivim, labavo povezanim i neovisnim što softver čini skalabilnim i općenito olakšava isporuku pojedinačnih funkcionalnosti. Nadalje, zbog izoliranosti mikroservisa razvoj softvera s ovakvim tipom arhitekture je brži i olakšava koordinaciju u razvojnom timu. S druge strane, s kompleksnijim sustavima se usporavaju performanse mikroservisa jer se mrežni promet povećava zbog učestalije potrebe međusobne komunikacije mikroservisa, što se ne događa u monolitnoj arhitekturi [9, str. 34].



Slika 5. Primjer strukture mikroservisne arhitekture (Prema: [10, str. 260])

4.5. Prostorno-bazirana arhitektura

Prostorno-bazirana arhitektura je osmišljena s ciljem rješavanja problema skalabilnosti i konkurentnosti sustava korištenjem distribuirane zajedničke memorije (eng. *tuple space*) gdje je broj korisnika i istovremenih zahtjeva velik. Način na koji se ovdje postiže skalabilnost je uklanjanjem središnje baze podataka i uvođenjem mreža podataka u memoriji (eng. *in-*

memory data grids). Umjesto spremanja podataka u bazu oni se repliciraju i spremaju u samostalne, aktivne obrađivačke jedinice (eng. *processing units*) koje se mogu dinamički pokretati i gasiti ovisno o trenutnom opterećenju sustava brojem korisnika. Ovo rješava problem skalabilnosti sustava jer se uklanja opterećenje sa središnje baze podataka koje se prebacuje na mreže podataka čiji se broj prilagođava trenutnom broju korisnika sustava što pruža visoku skalabilnost [9, str. 37].

Ovaj uzorak arhitekture je koristan u standardnim mrežnim aplikacijama s velikim brojem korisnika (npr. tržište kriptovaluta) gdje korisnički zahtjev prvo ide na mrežni server, zatim na server aplikacije i konačno na server baze podataka i ta tri servera čine tri uska grla (eng. *bottlenecks*) sustava gdje prilikom velikog broja zahtjeva može doći do usporavanja rada softvera. Problemi uskog grla mrežnog servera i servera aplikacije se mogu riješiti skaliranjem servera, no to nije jeftino i samo će prebaciti teret na server baze podataka što nije poželjno u aplikacijama s velikim brojem istovremenih zahtjeva. Prostorno-bazirana arhitektura rješava upravo taj problem koristeći virtualnog posrednika (eng. *virtual middleware*) koji upravlja zahtjevima i koji se brine za replikaciju podataka na obrađivačkim jedinicama paljenjem i gašenjem istih kada je to potrebno.

Prostorno-bazirana arhitektura ima sposobnost reagiranja na količinu opterećenja u mreži zbog načina na koji obrađuje korisničke zahtjeve koristeći obrađivačke jedinice što samu arhitekturu čini skalabilnom i donosi joj visoke performanse. S druge strane, teže je razvijati sustave bazirane na SBA-i zbog njezine sofisticirane strukture i općenitog nepoznavanja alata i proizvoda od strane programera koji se koriste za razvoj ovakve vrste arhitekture. Nadalje, teško je testirati skalabilnost i performanse ove vrste arhitekture jer je postizanje velikog korisničkog opterećenja skupo i dugotrajno [9, str. 43].

4.6. Ostale vrste arhitektura

S obzirom na to da arhitektura softvera ima široku definiciju postoje i mnogi drugi arhitekturalni uzorci koji se koriste u razvoju softvera, a mogu se koristiti samostalno ili mogu biti dio neke druge, veće arhitekture. Neki od najpoznatijih uzoraka su zasigurno [11]:

- Monolitna arhitektura.
- Servisno-orijentirana arhitektura.
- Klijent-poslužitelj (eng. *client-server*).
- Model-pogled-upravitelj (eng. *Model-View-Controller* ili kraće MVC) i njegove derivacije model-pogled-prezenter (eng. *Model-View-Presenter* ili kraće MVP) i model-pogled-pogled-model (eng. *Model-View-ViewModel* ili kraće MVVM).
- *Pipes and Filters* ili kraće PAF.

- *Presentation-Abstraction-Control* ili kraće PAC.

4.6.1. Monolitna arhitektura

Monolitna arhitektura predstavlja alternativu mikroservisnoj arhitekturi i veći je dio povijesti bila prvi odabir za razvoj softvera. Za razliku od mikroservisa, monolitni sustav je kao što i ime sugerira izgrađen kao jedna cjelina tj. u jednom komadu i može sadržavati više slojeva od kojih su najčešći:

- Autentikacija i autorizacija – identifikacija korisnika i provjera njegovih ovlasti.
- Korisničko sučelje – prezentacija i prikaz podataka.
- Poslovna logika – upravljanje poslovnim pravilima.
- Baza podataka – perzistentno spremište podataka.

Iako monolitna i slojevita arhitektura na prvu izgledaju isto jer mogu sadržavati iste komponente, postoji razlika među njima. Za razliku od monolitne arhitekture gdje je sve povezano (npr. jedna datoteka može sadržavati povezivanje na bazu, reprezentacije entiteta u bazi tj. modele ili čak krajnje točke *Application Programming Interface-a* ili kraće API-ja) kod slojevite arhitekture takve stvari su odvojene i cilj je postići razdvajanje briga (eng. *separation of concerns*) gdje je svakom sloju dozvoljeno koristiti samo sloj ispod njega. Kod monolitne arhitekture komponente poput autentikacije, poslovne logike i pristup bazi podataka su međusobno povezani, ovisni i isprepleteni što rezultira čvrsto povezanim (eng. *tightly coupled*) programskim kôdom. Monolitna arhitektura predstavlja uobičajen pristup razvoju softvera i iako danas gubi utjecaj, nije nužno loša. Njezine prednosti se očituju u jednostavnosti razvoja, testiranja, isporuke i skaliranja. S druge strane, održavanje ovakve arhitekture je zahtjevno, a implementacija promjena i novih funkcionalnosti zahtijeva građenje i isporuku cijele aplikacije što se često mora raditi izvan radnog vremena kako sama isporuka ne bi utjecala na korisnike koji trenutno koriste sustav i kako bi se dalo vremena za ispravak potencijalnih greški koje se mogu pojaviti prilikom isporuke sustava [12].

4.6.2. Servisno-orijentirana arhitektura

Servisno-orijentirana arhitektura je vrsta arhitekture koja se temelji na servisima kao primarnim komponentama arhitekture za implementaciju i obavljanje poslovnih ciljeva. S obzirom na to da je i SOA jedna vrsta distribuirane arhitekture, servisi kao mehanizam komunikacije koriste određene protokole poput SOAP-a, HTTP-a, AMQP-a ili JMS-a. Svaki servis u SOA-i ima specifičnu funkciju i izvršava određen poslovni cilj, npr. provjera stanja računa. Servisi se koriste putem njihovih sučelja koji su labavo povezani tako da programer ne mora ni znati njihovu unutarnju implementaciju, on mora samo znati koje usluge sučelja

servisa nude. U SOA-i postoje dvije glavne uloge: davatelji usluga (eng. *service providers*) i potrošači usluga (eng. *service consumers*). Davatelji usluga izlažu krajnje točke i opisuju dostupne radnje na svakoj krajnjoj točki, a potrošači usluga zatražuju usluge od davatelja i obrađuju odgovore. SOA nudi određene prednosti u odnosu na monolitnu i slojevitu arhitekturu poput bolje skalabilnosti, labave povezanosti, veće modularnosti i bolje kontrole nad razvojem, testiranjem, implementacijom i isporukom sustava. S druge strane SOA je puno kompleksnija i skuplja za razvijati i održavati u kontekstu ljudstva, razvoja i tehnologije [13].

4.6.2.1. Usporedba mikroservisne i servisno-orijentirane arhitekture

Kako je mikroservisna arhitektura varijanta SOA-e postoje određena preklapanja između ove dvije arhitekture. Obe arhitekture su distribuirane i bazirane su na servisima koji komuniciraju pomoću nekog protokola za daljinski pristup poput HTTP-a, SOAP-a, AMQP-a etc. i kao takve dijele određene karakteristike poput dobre podrške za skalabilnost, bolje odvajanje (eng. *decoupling*) odgovornosti i bolju kontrolu nad razvojem, testiranjem i isporukom softvera. No, iako dijele mnoge odlike postoje neke ključne razlike između SOA-e i MSA-e [13]:

- Servisi kod MSA-e su većinom privatni i njihove usluge su dostupne samo drugim internim servisima u organizaciji za ispunjavanje poslovnih ciljeva, dok SOA ima širu kategorizaciju mogućih vrsta servisa poput apstraktnih poslovnih servisa koji ne implementiraju ništa nego definiraju usluge drugih servisa na razini poduzeća (eng. *enterprise*) putem *Web Services Definition Language-a* ili kraće WSDL-a.
- Vlasnik mikroservisa (onaj tko razvija i održava servis) u MSA-i je najčešće razvojni tim dok u SOA-i koja podržava više različitih vrsta servisa postoje vlasnici za poslovne servise, za servise na razini poduzeća, za servise na razini aplikacije etc.
- Veličina koja je vjerojatno i najveća razlika. Granulacija servisa kod MSA-e je puno manja kao što i samo ime mikroservisa govori dok se kod SOA-e servisi mogu kretati od malih, specijaliziranih aplikacijskih servisa do velikih servisa koji mogu izvršavati usluge na razini poduzeća što rezultira razlikama u opsegu i broju komponenti.
- Dok u MSA-i servisi najčešće koriste REST kao primarni mehanizam komunikacije, SOA nema ograničenja i može koristiti desetak različitih mehanizama komunikacije od kojih se najčešće koristi SOAP.

5. Mikroservisi

Zadnjih desetak godina mikroservisna arhitektura softvera dominira IT tržištem. Uspon mikroservisa je u nekom smislu prilagodba na nove zahtjeve tržišta i odgovor na određene nedostatke monolitne i servisno-orijentirane arhitekture u ispunjavanju tih zahtjeva. U ovom poglavlju prikazat će se izazovi, karakteristike, principi, metodologije i tehnologije mikroservisne arhitekture kako bi se dobio uvid u jednu od najbitnijih softverskih arhitektura dvadeset i prvog stoljeća.

5.1. Izazovi i nedostaci

Najveći izazov u MSA-i je kako podijeliti sustav na mikroservise. Idealno, svaki servis bi trebao imati mali broj odgovornosti. To se postiže korištenjem principa jedne odgovornosti (eng. *single responsibility principle* ili kraće SRP) koji se inače primjenjuje na klase tako da se grupiraju stvari koje se mijenjaju iz istog razloga, a odvoje one koje se mijenjaju iz različitih razloga. Dvije najčešće strategije dekompozicije mikroservisa su [10]:

- Dekompozicija po poslovnim funkcijama – poslovne funkcije govore što organizacija radi i generalno se smatra nečim čime organizacija stvara vrijednost. Primjerice, za mrežnu trgovinu mikroservisi bi mogli biti *Upravljanje zalihama, Otprema Narudžbi* i slično koji bi odgovarali poslovnim funkcijama u organizaciji.
- Dekompozicija poddomenama dizajna vođenom domenom (eng. *domain-driven design* ili kraće DDD) – DDD govori da je poslovanje tvrtke domena i da se ta domena sastoji od dvije važne stvari: više poddomena od kojih svaka odgovara različitom odijelu poslovanja i ograničavanju konteksta. Primjerice, za neku mrežnu trgovinu mikroservisi koji bi odgovarali poddomenama domene organizacije bi mogli biti *Upravljanje narudžbama* i *Dostava*.

Ove dvije strategije izgledaju slično i razlike su u nijansama od autora do autora, no razlika između njih se može gledati na sljedeći način. Dekompozicija po poslovnim funkcijama će po Conwayevom zakonu (eng. *Conway's Law*) preslikati način na koje je poslovanje organizacije ustrojeno u arhitekturu softvera [14]. To znači da će organizacijska struktura tvrtke biti prenesena u mikroservisnu arhitekturu što sa sobom može prenijeti i same neučinkovitosti organizacije u arhitekturu softvera. S druge strane, DDD pruža skup metodologija i alata koji bi trebali pomoći u razumijevanju domene poslovanja kako bi se ta domena mogla preslikati u dizajn softvera. Drugim riječima, dekompozicija po domenama nastoji dizajnirati sustav prema

analizi procesa i protoka informacija u poslovanju jer ne mora značiti da će poslovna struktura u nekoj organizaciji biti idealno posložena. To znači da DDD može razotkriti neučinkovitosti u postojećoj organizacijskoj strukturi poslovanja što može pomoći u evoluiranju razvojnog tima i organizacije kako bi se dobila željena struktura. Idealna situacija bi bila kada bi tehnološka arhitektura bila izomorfna s poslovnom arhitekturom. Ovaj princip razvijanja dizajna softvera analizirajući domenu poslovanja u svrhu dobivanja idealne tehnološke i organizacijske arhitekture se zove inverzni Conwayev manevar (eng. *Inverse Conway Maneuver*) [10, str. 30].

Nakon dekompozicije servisa slijedi izazov imenovanja samih servisa. Za imenovanje mikroservisa je prvo potrebno identificirati operacije koje će servis izvršavati. Nakon toga može se osmisliti naziv koji će nagovještavati moguće klase i funkcije servisa bazirane na operacijama koje će vršiti. Generalno postoje dva načina imenovanja mikroservisa:

- Baziran na glagolima ili po slučaju upotrebe (eng. *use-case*) i definiranje servisa koji su zaduženi za određene akcije, npr. *Otprema Narudžbi*.
- Baziran na imenicama ili resursima i definiranje servisa koji su zaduženi za sve operacije nad određenim entitetima tj. resursima, npr. *Financije*.

Sustav se može dekomponirati i po glagolima i po imenicama tako da se ne mora nužno odabrati jedan način koji će cijeli ekosustav pratiti.

Iako postoje razni koraci i strategije dekompozicije arhitekture na mikroservise, to nije znanost nego se više smatra umjetnošću i slobodom arhitekta softvera jer uvelike ovisi o situaciji i kontekstu. Osim navedenih izazova postoje razna pitanja koja se javljaju u mikroservisnoj arhitekturi od kojih su neka:

- Kako održati konzistentnost podataka kada se oni nalaze u više različitih baza podataka?
- Kako implementirati upite koji moraju izvući podatke iz više različitih mikroservisa?
- Kako održati labavu povezanost da verzioniranje jednog servisa ne utječe na ostale?
- Kako osigurati skup vještina i znanja tima za dizajniranje visoko distribuiranog sustava?

Uz navedene izazove mikroservisna arhitektura ima i niz nedostataka:

- Programeri se moraju nositi s većom kompleksnošću koja dolazi s distribuiranim sustavima.
- Potrebno je implementirati mehanizam komunikacije između servisa i nositi se s mogućim neuspjesima komunikacije na korektan način.

- Implementacija zahtjeva koji moraju proći kroz više mikroservisa je kompliciranija. U slučaju da različiti timovi rade na tim mikroservisima potrebna je dobra koordinacija među njima.
- Testiranje interakcija između servisa je teže zbog distribuirane prirode sustava.
- Povećana je potrošnja memorije zbog većeg broja aplikacija.
- Može doći do zagušenosti i kašnjenja mreže zbog međuservisne komunikacije.
- Dosljednost i integritet podataka donosi svoj skup izazova jer najčešće ne postoji centralizirana baza podataka.

Kako bi se sustav nosio s izazovima i nedostacima koje donosi distribuirana arhitektura osmišljene su razne metode i tehnike koje su objašnjene u sljedećim potpoglavljima.

5.2. Karakteristike

Jedna od najbitnijih karakteristika mikroservisa je njihova autonomnost što znači da je svaki servis zasebna baza programskog kôda i to omogućuje sljedeće stvari:

- Neovisna isporučivost – u usporedbi s monolitnom arhitekturom ovdje se ne mora izgraditi i isporučiti cijela aplikacija kada se dodaje nova ili ažurira postojeća značajka već samo servis koji je odgovoran za nju.
- Povećana otpornost na kvar – ako se dogodi greška u sustavu ona je izolirana na samo jedan servis i time se sprječavaju kaskadni kvarovi sustava što ga čini otpornijim.
- Tehnološka heterogenost ili poliglotsko programiranje – različiti servisi ne moraju dijeliti iste programske jezike i tehnologije.

Osim autonomnosti, težnja mikroservisa je da budu manji, finije granulirani i labavo povezani jer se time poboljšava održivost cjelokupnog sustava. Nadalje, u slučaju skaliranja projekta omogućuje se i bolja podrška za proširivanje razvojnih napora jer je lakše organizirati poslovne funkcije i razvojne timove oko mikroservisne arhitekture nego što je oko monolitne. Što se tiče razgovora između mikroservisa interna implementacija svakog servisa treba ostati skrivena, a za komunikaciju se potiče korištenje laganih i brzih komunikacijskih mehanizama koji troše malo resursa poput HTTP-a kako bi se smanjila latencija između servisa.

Već je rečeno da mikroservisi trebaju biti organizirani oko poslovnih funkcija i ta funkcija treba biti implementirana u ograničenom kontekstu. Ograničeni kontekst predstavlja prirodnu podjelu raznih odjela unutar tvrtke i pruža granicu do koje domena određenog mikroservisa treba ići. Iz ovoga se može zaključiti da mikroservisna arhitektura donekle replicira organizacijsku strukturu tvrtke što tvrdi i Conwayev zakon. Primjerice, u nekom sustavu

mrežne trgovine baziranom na mikroservisnoj arhitekturi, isto kao i u tvrtki koja vodi tu trgovinu, mogu postojati dva odjela odnosno mikroservisa: *Računovodstvo* i *Skladište*. Mikroservis *Računovodstvo* će biti zadužen za financije, a mikroservis *Skladište* za upravljanje artiklima. Kako tvrtka raste i kako se komunikacija između odjela mijenja tako će se mijenjati i komunikacija između mikroservisa. Ovo svojstvo sa sobom donosi i neke od najvećih izazova mikroservisa, a to su veličina i podjela.

Mikroservisi su generalno manji od servisa u servisno-orijentiranoj arhitekturi i njihov naziv insinuira da bi trebali biti jako maleni, no koliko su zapravo oni mali? Jedna od čestih mjera koja se koristi za mjerenje veličine određene komponente softvera je u linijama programskog kôda. Takav način mjerenja je često problematičan i nepraktičan zbog više razloga [15]:

- Određeni programski jezici su izražajniji od drugih ili potiču različite načine pisanja programskog kôda.
- Alati za brojanje linija kôda često uključuju ovisnosti mikroservisa koje mogu sadržavati puno linija programskog kôda.
- Neki servisi su jednostavno kompleksniji od drugih.

Najčešće je najbolja mjera osobni osjećaj programera za kompleksnost servisa jer su oni prvi koji uoče kada je određena komponenta softvera postala prevelika i teška za održavati. Jon Eaves je lijepo opisao veličinu mikroservisa tako što je okarakterizirao mikroservis kao nešto što se može iznova napraviti u vremenskom periodu od dva tjedna [15, str. 2]. Kako mikroservisi postaju manji tako ga je razvojnom timu lakše održavati, no postoji granica. Premali mikroservisi čine cjelokupnu arhitekturu sustava kompleksnom, čvrsto povezanom, sporijom i težom za održavati stoga treba naći tanku liniju koja će distribuiranu arhitekturu softvera činiti maksimalno profitabilnom za organizaciju ali i za razvojni tim.

5.2.1. Autonomnost

Mikroservisi su samostalni i neovisni što znači da se zasebno isporučuju i ne ovise o drugim servisima. Svaki mikroservis ispunjava specifičnu poslovnu funkciju, a zajedno čine fleksibilan i funkcionalan sustav. S obzirom na to da su mikroservisi neovisni potrebno je težiti cilju da se oni mogu mijenjati i isporučivati bez utjecaja na ostale mikroservise koji konzumiraju njihove usluge. Drugim riječima, promjena jednog mikroservisa ne bi trebala povlačiti za sobom promjene u mikroservisima koji konzumiraju usluge promijenjenog mikroservisa. Što se tiče usluga mikroservisa, njihova poslovna logika bi trebala biti skrivena tj. drugi mikroservisi bi trebali biti svjesni samo što jedan mikroservis nudi, a ne i kako to obavlja. To se najčešće radi korištenjem sučelja, slično kao i u servisno-orijentiranoj arhitekturi.

Autonomnost i labava povezanost mikroservisa također omogućava razvojnim timovima da budu samostalni. Organizacijska struktura razvojnog dijela tvrtke može biti organizirana kao skup manjih timova gdje je svaki tim zadužen za razvoj jedne ili više usluga tj. servisa neovisno od drugih timova. Drugi timovi jedino trebaju biti svjesni dostupnih usluga pojedinog mikroservisa i kako pozivati njegove usluge što se može riješiti prikladnom dokumentacijom. Time se stvara jasnija raspodjela posla i razvoj cjelokupnog sustava je lakši i brži. Upravo zbog toga maksimiziranje autonomnosti i samostalnosti mikroservisa donosi tehničke, organizacijske i razne druge koristi.

Iako je maksimalna autonomnost mikroservisa važna, treba imati na umu da mikroservisi skoro nikada neće biti potpuno izolirani i samoodrživi. Mikroservisi zajedno čine cjelovit sustav ali samostalno često trebaju tražiti usluge i podatke jedni od drugih za ispunjavanje vlastitih poslovnih ciljeva. Primjerice, za neki sustav mrežne trgovine bazirane na mikroservisima, između ostalih, mogu postojati dva mikroservisa od kojih prvi u sebi sadrži i procesuira narudžbe, a drugi koji sadrži sve dostupne proizvode i nudi informacije o njima. Prilikom kreiranja narudžbe neophodno je da mikroservis koji procesuira narudžbe zatraži informacije poput dostupnosti proizvoda na skladištu i cijene o naručenim proizvodima iz mikroservisa koji sadrži proizvode jer će se prilikom kreiranja narudžbe poslati samo identifikatori proizvoda i njihova količina kako bi se smanjila mogućnost manipulacije podataka s korisničke strane.

Ovdje se odmah može postaviti pitanje zašto uopće postoje dva mikroservisa od kojih jedan obrađuje narudžbe, a drugi sadrži proizvode. U nekoj jednostavnoj mrežnoj trgovini koja nema puno funkcija ta dva mikroservisa se mogu spojiti u jedan i to bi zasigurno olakšalo posao jednom programeru koji razvija takav sustav. No, u većem sustavu koji bi sadržavao funkcije poput upravljanja skladištem, prodajom, zalihama, uslugama, financijama, analitikom etc. koji bi očito razvijala veća tvrtka mikroservisi bi razvojnom timu donijeli razne prednosti od kojih su neke [15, str. 3]:

- Lakši i brži razvoj zbog efektivnije organizacijske raspodjele posla.
- Sprječavanje gomilanja programskog kôda i eventualno stvaranje monolita.
- Održivost, pouzdanost i olakšana isporučivost servisa zbog njihove autonomnosti i labave povezanosti.
- Povećanje skalabilnosti.

5.2.2. Tehnološka heterogenost

S obzirom na to da je u nekom sustavu baziranom na mikroservisnoj arhitekturi svaki mikroservis zasebna komponenta (aplikacija), oni mogu koristiti različite tehnologije za

postizanje vlastitih ciljeva i to se svojstvo zove tehnološka heterogenost. Neovisnost mikroservisa im donosi mogućnost da budu pisani u različitim programskim jezicima i radnim okvirima i jedino što je bitno je da dijele mehanizam komunikacije, npr. protokol HTTP. To funkcionira zato što mehanizmi komunikacije transformiraju podatke koje primaju i šalju u neobrađene podatke (eng. *raw data*), a onda mikroservisi koji dobivaju neobrađene podatke pretvaraju dobivene podatke u željeni tip i strukturu.

Tehnološka heterogenost omogućuje razvojnim timovima da mikroservise razvijaju pomoću alata i tehnologija koje će najbolje riješiti specifičan problem. To omogućava programerima da se ne moraju ograničavati i birati jednu tehnologiju za ispunjavanje svih zahtjeva u sustavu nego da biraju ono što će najbolje riješiti probleme određenog segmenta softvera. Uz mogućnost odabira tehnologije s kojom će se razvijati određeni mikroservis prirodno je i zaključiti da se za svaki servis može i odabrati vrsta spremišta podataka, a spremište podataka će biti odabrano na temelju onog što najbolje odgovara odabranoj tehnologiji za taj mikroservis. Tehnologija i spremište podataka za svaki mikroservis se biraju tako da se prouči koja tehnologija i vrsta spremišta podataka najbolje rješavaju korisničke zahtjeve i poslovnu logiku vezanu uz taj mikroservis.

Primjerice, za neki sustav mrežne trgovine bazirane na mikroservisima, između ostalih, mogu postojati dva mikroservisa od kojih prvi u sebi sadrži i procesuira narudžbe, a drugi sadrži sve korisnike i informacije o njima i koji sprema korisničke podatke u relacijsku bazu podataka poput PostgreSQL-a. U ovom slučaju narudžbe moraju imati referencu na korisnika i mora se znati koji je korisnik kreirao koju narudžbu. Takva situacija gdje su bitne veze među podacima pogodna je za graf-baze podataka poput Neo4J-a jer su u njima odnosi među podacima eksplicitno definirani, za razliku od relacijskih baza podataka gdje se to implicitno radi pomoću stranih ključeva. Iz tog razloga mikroservis koji sprema narudžbe mogao bi biti izrađen uz pomoć tehnologija koje podržavaju graf-baze podataka poput Go programskog jezika što će ubrzati izvršavanje kompleksnijih upita (eng. *queries*) na bazu podataka u situacijama kada se traži koji je korisnik naručio narudžbu koja ispunjava određene uvjete.

Drugi primjer različitih spremišta podataka može biti korištenje NoSQL baze podataka poput MongoDB-a koja sprema podatke u obliku dokumenata. Dokument je uređeni skup ključeva i pridruženih vrijednosti koji su slični jednom retku u relacijskoj bazi podataka, a spremaju se u kolekcije što bi odgovaralo tablicama u relacijskim sustavima. Razlika između redaka u relacijskim sustavima i dokumenata u NoSQL bazama podataka je u tome što dokumenti koji se spremaju u određenu kolekciju ne moraju imati istu strukturu tj. ne moraju svi sadržavati iste podatke tako da se u NoSQL bazama podataka ne mora definirati struktura kolekcije kao što se mora tablica u relacijskim sustavima. Takav način spremanja podataka je pogodan za servise koji ne znaju kakav će format podataka biti što im daje veću fleksibilnost i

dinamičnost. Primjerice, za neki sustav društvene mreže bazirane na mikroservisima, između ostalih, mogu postojati dva mikroservisa od kojih prvi sadrži korisnike u nekom relacijskom sustavu, a drugi akcije koje korisnici mogu raditi na društvenoj mreži poput kreiranja komentara, pregleda poruka, reagiranja na novosti i slično. S obzirom na to da će svaka akcija sadržavati drukčije podatke, a i da će akcija biti puno (NoSQL dobro podržava skalabilnost) pametno bi bilo da mikroservis koji sprema akcije to odrađuje u NoSQL bazi podataka zbog veće brzine i skalabilnosti.

No, treba imati na umu da tehnološka heterogenost također može donijeti određene rizike u organizaciju. Varijabilnost i sloboda tima da bira tehnologije koje najviše odgovaraju zahtjevima može usporiti razvoj sustava što organizaciju košta vremena i novca. Nadalje, korištenje tehnologija i alata koji se do sada nisu koristili u razvojnom timu može donijeti razne probleme u kasnijim fazama razvoja s kojima članovi tima do sada nisu bili upoznati. Finalno, krivulja učenja novih tehnologija timu ili članovima tima bit će strma jer ljudima općenito treba vremena da se prilagode na nove stvari [15, str. 4].

5.2.3. Integracija i komunikacija

Klijentske aplikacije, a najčešće su to mobilne, mrežne ili stolne aplikacije, dobivaju podatke s pozadinske strane sustava. U monolitnoj arhitekturi slanje zahtjeva i upita klijenata u svrhu dobivanja podataka s pozadinske strane je jednostavno jer u pozadini postoji jedan sustav s jednom bazom podataka. U distribuiranoj arhitekturi stvari se zakompliciraju jer pozadinska strana više nije jedan sustav već skupina mikroservisa i najčešći obrazac arhitekture koji se koristi je taj da svaki mikroservis ima svoju bazu podataka što se zove uzorak baze podataka po servisu (eng. *database per service pattern*). Tada se javlja pitanje: „Što učiniti kada su podaci koje klijent zahtijeva raštrkani između više mikroservisa?“ Upravo zbog tog integracija mikroservisa predstavlja jedan od najbitnijih aspekata u dizajnu softvera mikroservisne arhitekture što proizlazi iz zahtjeva da mikroservisi budu autonomni i labavo povezani, a da s druge strane zajedno čine cjelovit i funkcionalan sustav.

5.2.3.1. Pretraživanje podataka

Kada se priča o implementaciji zahtjeva s operacijama upita na bazu podataka u mikroservisnoj arhitekturi postoje dva različita uzorka [10, str. 221]:

- Uzorak API kompozicije – najjednostavniji pristup koji funkcionira tako da postoji API kompozitor koji zatražuje podatke od više različitih mikroservisa i on je odgovoran za slanje zahtjeva na mikroservise i kombiniranje rezultata.

- Uzorak razdvajanja odgovornosti za naredbe (eng. *command query responsibility segregation* ili kraće CQRS) – bolji ali kompleksniji uzorak koji održava jednu ili više baza podataka čija je jedina svrha podržavati upite.

Kod uzorka API kompozicije imamo kompozitora koji je zadužen za pozivanje mikroservisa koji sadrže potrebne podatke. Kompozitor je najčešće sam klijent, npr. mobilna aplikacija koja mora prikazati određene podatke korisniku i kao što uzorak definira mobilna aplikacija je odgovorna za pozivanje i kombiniranje rezultata više različitih mikroservisa. No, moguće je da kompozitor ne bude klijent već svoj servis koji se može implementirati na dva načina:

- Korištenjem API pristupnika (eng. *gateway*) koji će uz ostale zadaće imati ulogu API kompozitora.
- Implementacijom API kompozitora kao samostalnog servisa.

API pristupnik je servis koji je ulazna točka u pozadinski ekosustav mikroservisa iz vanjskoga svijeta i može obavljati sljedeće funkcije [10, str. 271]:

- Usmjeravanje zahtjeva na odgovarajuće mikroservise.
- Implementiranje uzorka API kompozicije za kombiniranje rezultata iz više različitih mikroservisa.
- Implementiranje rubnih funkcija (eng. *edge functions*).
- Prijevod protokola tj. prilagođavanje protokola između klijenata i mikroservisa.

Kao što je već rečeno API pristupnik može implementirati uzorak API kompozicije za kombiniranje rezultata iz više različitih mikroservisa, no API kompozitor može biti i vlastiti servis, van API pristupnika. API pristupnik bi trebao imati ulogu API kompozitora kada je operacija upita dio vanjskog API-ja aplikacije, a API kompozitor bi trebao biti svoj servis kada se operacija upita interno koristi od strane više mikroservisa.

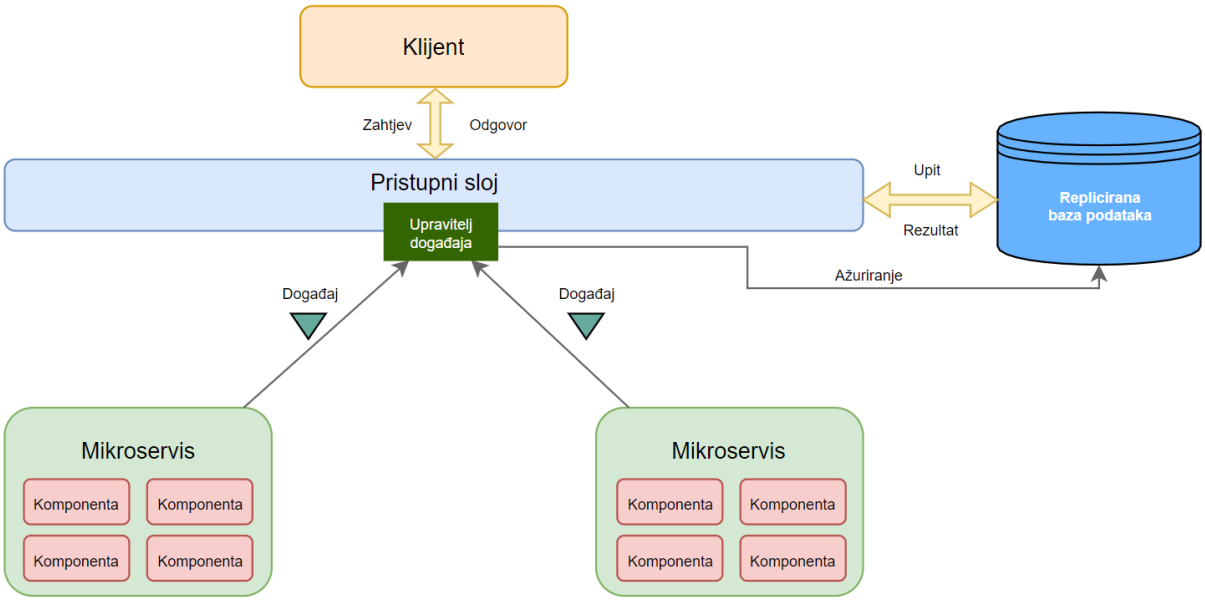
Razlika između servisnog API kompozitora i pristupa kada je klijent ujedno i API kompozitor je u količini zahtjeva koja se šalje preko mreže – kada klijent nije API kompozitor onda se šalje jedan API poziv više preko mreže. Naravno, i s tim jednim API pozivom od strane klijenta servisni API kompozitor će kada primi klijentov jedini zahtjev i dalje morati poslati više zahtjeva na različite mikroservise, no velika je razlika šalje li više zahtjeva klijent ili servis. Internet ima puno manju propusnost i veću latenciju nego što ima lokalna mreža, a s obzirom na to da će API kompozitor i mikroservisi vjerojatno biti na istoj mreži ovakav pristup će biti puno brži što poboljšava korisničko iskustvo i performanse sustava.

Uzorak API kompozitora je intuitivan i jednostavan za kombiniranje operacija upita, no ima određene nedostatke:

- Veći broj zahtjeva u odnosu na monolitnu arhitekturu gdje je samo jedan.
- Povećan rizik nedostupnosti podataka jer većim brojem mikroservisa raste šansa za nedostupnost jednog.
- Nedostatak dosljednosti podataka zbog korištenja više baza podataka.

Uz uzorak API kompozitora za implementaciju zahtjeva s operacijama upita na bazu podataka u mikroservisnoj arhitekturi postoji i CQRS uzorak. CQRS uzorak funkcionira tako da se definira baza podataka koja sadržava podatke iz više različitih mikroservisa i iz koje je moguće vršiti samo akciju pretraživanja (eng. *read-only database*). Takva baza podataka sadrži replicirane podatke iz različitih mikroservisa i ažurira ih pretplatom (eng. *subscribe*) na događaje koje je objavio (eng. *publish*) mikroservis koji sadrži izvorne podatke.

Replicirana baza podataka mora samo podržavati upite koji su potrebni klijentima, ne mora kopirati sve podatke iz svih baza podataka. CQRS uzorak omogućuje učinkovitu implementaciju jednostavnih i kompleksnih upita u mikroservisnoj arhitekturi što pojednostavljuje naredbe i upite i poboljšava razdvajanje briga. S druge strane, nedostaci ovakvog pristupa su ti da arhitektura sustava postaje kompleksnija i postoji određena latencija dok se ažuriraju podaci u repliciranoj bazi podataka. Implementacija upita koji dohvaćaju podatke iz različitih mikroservisa je općenito izazovna kada se radi o uzorku baze podataka po servisu stoga je preporuka koristiti uzorak API kompozicije zbog njegove jednostavnosti, a u slučaju kompleksnijih upita i složenijih zahtjeva, može se koristiti CQRS uzorak [10, str. 229].

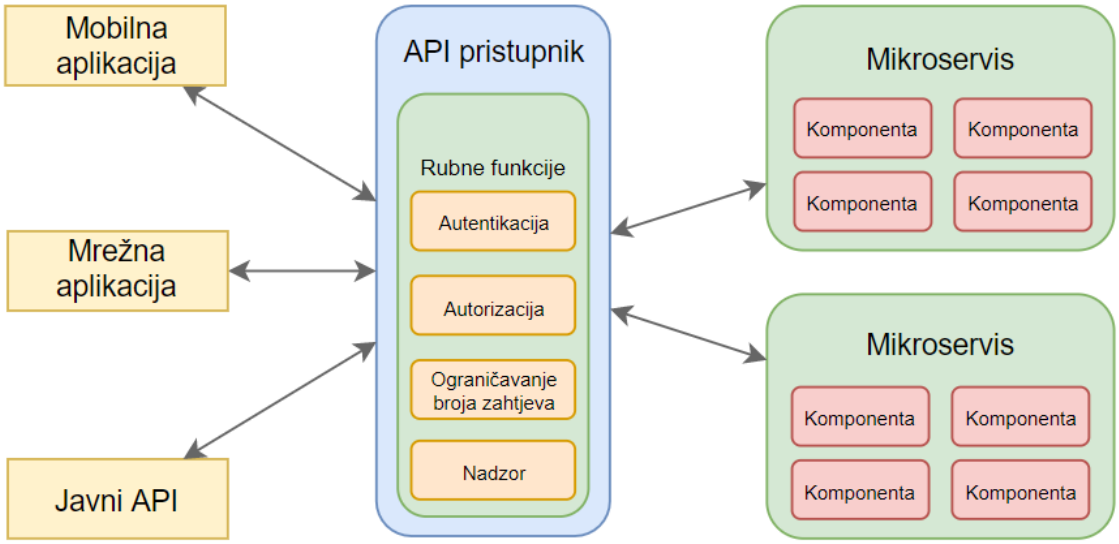


Slika 6. Arhitektura CQRS uzorka (Prema: [10, str. 234])

5.2.3.2. Pristup API-jima za konzumaciju

Spomenuto je da se kao ulazna točka u pozadinski ekosustav mikroservisa može koristiti API pristupnik. Osim što API pristupnik preusmjerava zahtjeve na odgovarajuće mikroservise i što može imati ulogu API kompozitora, on može prevoditi protokole između klijenata i mikroservisa i može sadržavati rubne funkcije. Primjerice, API pristupnik može klijentima nuditi RESTful API-je dok s mikroservisima može komunicirati putem AMQP-a. Rubne funkcije su određene metode koje API pristupnik može implementirati u ulaznoj točki pozadinskog sustava, a primjeri najčešćih rubnih funkcija su [10, str. 262]:

- Autentikacija.
- Autorizacija.
- Ograničavanje broja zahtjeva.
- Predmemoriranje (eng. *caching*).
- Prikupljanje mjernih podataka.
- Nadzor servisa i zapisivanje zahtjeva (eng. *request logging*).



Slika 7. Funkcija API pristupnika (Prema: [10, str. 260])

S prethodne slike se može vidjeti da API pristupnik najčešće nudi različite API-je za različite klijente. Primjerice, mrežna stranica će prikazivati puno više informacija o narudžbi nego što će prikazivati mobilna aplikacija. Ovakav pristup odgovara manjim sustavima, no u slučaju kompleksnije mikroservisne arhitekture koju razvija više timova i koja podržava više klijenata (npr. mobilna, mrežna i stolna aplikacija i javni API-ji) tada je zamagljena odgovornost koji tim mora održavati i razvijati API pristupnik. Rješenje tog problema je varijanta API

pristupnika koja se naziva *Backends for frontends* uzorak ili kraće BFF. BFF uzorak definira da mora postojati API pristupnik za svakog klijenta i tako se rješava problem odgovornosti API pristupnika jer je svaki tim odgovoran za svoj API pristupnik. Osim što razjašnjava odgovornost BFF također povećava izolaciju, pouzdanost, uočljivost, skalabilnost i vrijeme pokretanja API pristupnika [10, str. 265]. Gledajući prethodnu sliku to bi značilo da će postojati tri API pristupnika, jedan za mobilnu aplikaciju, jedan za mrežnu stranicu i jedan za javne API-je.

5.2.3.3. Definiranje komunikacije

Jedna od važnijih odluka prilikom razvoja softvera je odabir sinkrone ili asinkrone vrste komunikacije između klijenata i pozadinskog sustava. Pod sinkronom komunikacijom se misli na zahtjev/odgovor vrstu komunikacije gdje se kod poziva udaljenog poslužitelja čeka na njegov odgovor i samim time se može odrediti uspješnost izvršavanja operacije, dok kod asinkrone komunikacije pozivatelj ne čeka odgovor tj. operaciju da se završi već nastavlja dalje s radom. Asinkrona vrsta komunikacije se koristi u sustavima baziranim na događajima i korisna je kod zahtjeva koji se moraju dugo čekati jer tada nije praktično blokirati rad korisnika dok on čeka na odgovor. Za sinkronu komunikaciju se najčešće koristi REST baziran na HTTP-u ili RPC dok se kod asinkrone komunikacije koristi AMQP ili STOMP. Uz odabir vrste komunikacije servisi i klijenti moraju također odabrati formate poruka koje će koristiti. To su najčešće *JavaScript Object Notation* (ili kraće JSON) ili *Extensible Markup Language* (ili kraće XML), no moguće je koristiti i binarne formate poruka koji se rjeđe koriste ali mogu biti učinkovitiji [16].

5.2.4. Skalabilnost

Autonomnost i labava povezanost mikroservisa omogućavaju još jedan bitan aspekt mikroservisne arhitekture – skalabilnost odnosno svojstvo softvera da dobro podržava skaliranje sustava. U monolitnoj arhitekturi kada se dodaje nova ili ispravlja već postojeća funkcionalnost cijeli je sustav potrebno ponovno izgraditi i isporučiti. Kako sustav raste trajanje isporuke softvera postaje sve dulje i rizičnije i najčešće se mora raditi izvan radnog vremena kako softver ne bi postao nedostupan dok se koristi. Nadalje, ako dođe do neočekivane greške prilikom isporuke softvera potrebno je vratiti (eng. *rollback*) promjene ili popraviti greške, ako je to uopće moguće prije nego radno vrijeme počne. Monolitna arhitektura generalno nije loša i struka ponekad programerima preporučuje da svoju karijeru započnu s njom jer je jednostavna, izravna i omogućava brz razvoj softvera. No, kako sustav raste i dodaju se nove funkcionalnosti kompleksnost softvera raste, a s porastom broja korisnika raste i opterećenje na sam sustav što općenito smanjuje performanse softvera.

Kako bi se riješili problemi koji se stvaraju povećavanjem opterećenosti sustava i porastom broja korisnika monolitnog sustava on se skalira, i to prvo horizontalno, a zatim vertikalno. Horizontalno skaliranje predstavlja pokretanje više kopija softvera najčešće na više poslužitelja iza uravnoteživača opterećenja (eng. *load balancer*) i svaka kopija softvera preuzima dio mrežnog prometa. Uravnoteživač opterećenja služi za ravnomjerno distribuiranje zahtjeva svim kopijama aplikacije i ovo je jednostavan i klasičan način skaliranja softvera baziranog na monolitnoj arhitekturi. Ovaj pristup donosi sa sobom problem ignoriranja povećanog razvoja softvera i same složenosti sustava. Nadalje, potrebno je i više memorije koja mora biti osigurana zbog toga što svaka kopija softvera pristupa svim podacima sustava. Nakon što horizontalno skaliranje više nije dovoljno uvodi se vertikalno skaliranje. Vertikalno skaliranje je isporučivanje aplikacije na veće, snažnije i brže poslužitelje koji su u stanju podnijeti više opterećenja jer jednostavno imaju kvalitetnije hardverske komponente [17, str. 148].

Postoji i trodimenzionalni model skaliranja softvera na bazi kocke (eng. *scale cube*) koji tvrdi da postoje X, Y i Z os skaliranja softvera. Skaliranje osi X je jednako kao i klasično horizontalno skaliranje gdje se softver replicira iza uravnoteživača opterećenja. Skaliranje osi Y je razdvajanje softvera na više manjih servisa gdje je svaki servis odgovoran za jednu ili više usko povezanih funkcija što je zapravo i cilj mikroservisa. Ovakav tip skaliranja također rješava problem povećanog razvoja softvera i same složenosti sustava što uklanja nedostatke horizontalnog skaliranja. Iz ovog se može vidjeti da su mikroservisi sami po sebi prirodno skalabilni sudeći po modelu skaliranja softvera na bazi kocke jer bolje balansiraju opterećenje cjelokupnog softvera nego što to radi monolitna arhitektura [18, str. 6].

Učinkovitost je važna u svakom softveru i generalno nije teško evaluirati vrijeme izvršavanja određenog zahtjeva u sustavu baziranom na monolitnoj arhitekturi. S druge strane, procjena učinkovitosti izvršavanja zahtjeva u sustavu distribuirane arhitekture poput mikroservisa je zahtjevna jer se poslovne funkcije najčešće izvršavaju kroz više mikroservisa. Ovisno o kompleksnosti sustava jedan korisnički zahtjev može uzrokovati komunikaciju desetaka ili čak stotina mikroservisa. U takvom slučaju kompleksnog ekosustava gdje postoji mnogo mikroservisa učinkovitost jednog mikroservisa u odnosu na cijeli sustav dramatično opada ako svi mikroservisi nisu podvrgnuti istim standardima koji maksimiziraju performanse i skalabilnost, a to su [19, str. 88]:

- Razumijevanje kvantitativne i kvalitativne ljestvice rasta kako bi se mikroservis pripremio za očekivani rast prometa.
- Učinkovito korištenje hardverskih resursa.
- Biti svjestan uskih grla sustava i zahtjeva potrebnih za ispunjavanje poslovnih ciljeva.

- Prikladno planiranje kapaciteta.
- Korištenje skalabilne ovisnosti u mikroservisima.
- Skalabilno i učinkovito upravljanje mrežnim prometom.
- Upravljanje i procesuiranje zadataka na učinkovit način.
- Spremanje podataka na skalabilan način.

Metrika kvalitativne ljestvice rasta se odnosi na to gdje je određen mikroservis integriran u cijeli ekosustav mikroservisa. Drugim riječima, kvalitativna ljestvica rasta se koristi za povezivanje skalabilnosti mikroservisa s poslovnom metrikom na visokoj razini kako bi se odredilo kada je potrebno skalirati određeni mikroservis. Primjerice, u dostavnoj aplikaciji određeni pozadinski mikroservis koji je zadužen za procesuiranje narudžbi može se skalirati s brojem narudžbi koje ljudi kreiraju. Ovakve metrike kvalitativne ljestvice rasta tehnički nisu povezane s određenim mikroservisom nego s cjelokupnim dostavnim sustavom. To znači da će na određenoj razini organizacija imati predodžbu o tome kako će se metrike mijenjati s vremenom i koliko će se narudžbi stvarati. Kada se te informacije prenesu razvojnom timu oni će imati određenu sliku koliko će mrežnog prometa mikroservis zadužen za procesuiranje narudžbi imati i ovisno o procjenama mogu se donijeti određeni koraci za prilagodbu većem prometu. Kvantitativna ljestvica rasta se odnosi na dobro definirano i mjerljivo razumijevanje količine prometa koju mikroservis može podnijeti. Ovakve ljestvice se najčešće mjere sa zahtjevima u sekundi (eng. *requests per second* ili kraće RPS) i upitima u sekundi (eng. *queries per second* ili kraće QPS) koje mikroservis može podnijeti.

Sustav se često razvija tako da podržava ono što organizacija trenutno treba i često se zaboravi na skalabilnost sustava jer je ona nešto što budućnost nosi i potrebno ju je posebno planirati. S većim brojem korisnika raste i broj zahtjeva i sustav postaje sve opterećeniji i ako sustav trenutno dobro podnosi 1.000 konkurentnih (istovremeno aktivnih) korisnika, pitanje je kako će sustav reagirati kada taj broj naraste na 10.000 ili 100.000 korisnika. Ovisno o vrsti sustava i poslovnim zahtjevima arhitekti softvera moraju biti svjesni različitih parametara opterećenja (eng. *load parameters*) koji će utjecati na skalabilnost njihovog sustava od kojih su neki [20]:

- Broj zahtjeva po sekundi upućenih mrežnom serveru.
- Omjer čitanja i spremanja u bazu podataka.
- Broj konkurentnih korisnika.

Nakon što se definiraju parametri koji će biti vezani uz skalabilnost sustava mogu se istražiti kakvi će biti performansi sustava kada se opterećenje poveća. Cilj testiranja skalabilnosti sustava je vidjeti kako će sustav reagirati na opterećenje u budućnosti bez promjene resursa (npr. bolji procesor, više memorije) i koliko je potrebno povećati resurse kako

bi performansi ostali nepromijenjeni. Širenje na tržištu je u interesu svake organizacije i ovisno o parametrima opterećenja razvojni arhitekti moraju misliti šire i biti svjesni operacija koje implementiraju i što one za sobom povlače jer ono što je dovoljno sad ne znači da će biti i u budućnosti.

5.2.5. Testiranje

Testiranje softvera kao što i sama riječ govori predstavlja vrstu provjere kvalitete sustava ili njegovih pojedinih komponenti kako bi se pružio objektivan stav o trenutnom stanju sustava. Testiranje najčešće podrazumijeva izvršavanje određene funkcije sustava s namjerom pronalaska nepredvidivih kvarova i općenite provjere kvalitete, a najčešće se provodi na različitim nivoima sustava.

Iako postoje određene razlike slični testovi se provode i na monolitnoj i na mikroservisnoj arhitekturi, no kako je testiranje softvera znanost za sebe i kako postoje mnoge tehnike i prakse u testiranju softvera fokus će ovdje biti stavljen na najčešćim vrstama testiranja ovisno o nivou softvera [19, str. 152]:

- Statička analiza kôda s fokusom na *lint* testiranju.
- Jedinično testiranje.
- Integracijsko testiranje (eng. *integration testing*).
- Testiranje od kraja do kraja (eng. *end to end testing*).

Lint testiranje pregledava programski kôd i služi za uočavanje sintaksnih i stilskih pogrešaka vezanih uz određen programski jezik. Ovakav tip testiranja se još češće koristi za pridržavanje preporučenih specifikacija koje su na neki način vodiči za trenutno najprihvaćeniji standard i stil programiranja u određenom programskom jeziku.

Jedinično testiranje je zasigurno jedna od najčešće korištenih metoda testiranja softvera i predstavlja metodu ispitivanja u kojoj se testiraju pojedinačne jedinice (dijelovi softvera) u kontekstu programskog kôda. Te jedinice mogu biti funkcije, klase ali ponekad i cijele poslovne funkcije stoga je doseg jediničnog testiranja diskutabilan među autorima. Cilj ovakvog testiranja je osigurati da pojedine komponente softvera ispravno funkcioniraju i da ne sadrže greške. Jedinični testovi imaju najmanji doseg, no najbrži su i dobro izoliraju komponente softvera stoga je općenito je preporuka da se ovakvi testovi često pišu jer se s njima najčešće pronalaze greške.

Dok jedinično testiranje provjerava samo pojedinačne komponente softvera integracijsko testiranje provjerava ispravnost funkcioniranja jedne cjelokupne usluge (servisa) stoga se na integracijsko testiranje može gledati kao skup svih jediničnih testova pojedinačnog servisa u praksi. Iz ovog je očito da integracijski testovi imaju puno veći doseg stoga je

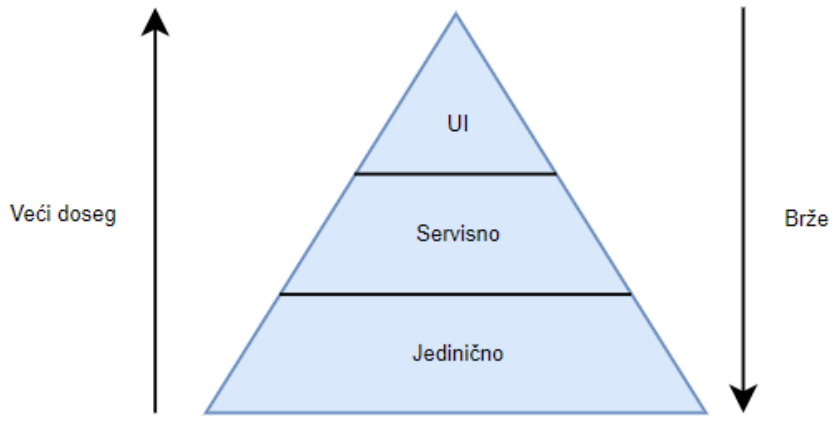
pronalazak greške u slučaju neprolaska testova teži nego kod jediničnih testova. S druge strane, zbog povećanog dosega integracijskog testiranja uspješnost kod ovakvog testiranja donosi više pouzdanja i vjere u ispravnost sustava jer se testira cijeli servis.

Testiranje od kraja do kraja (ponekad se naziva i sistemsko testiranje) predstavlja testove koji se izvode na cijelom sustavu. Ovakav tip testiranja se ponekad naziva testiranje korisničkog sučelja (eng. *user interface* ili kraće UI), no neki autori tvrde da postoji razlika između ta dva nazivlja [15, str. 133]. Ipak, najčešće se radi o simulaciji korisničke aktivnosti putem sučelja softvera. Ova vrsta testiranja ima najveći doseg stoga je najmukotrpnija i najsporija za provesti ali zbog najvećeg dosega donosi i najviše pouzdanja i vjere u softver jer se testira cjelokupni sustav.

5.2.5.1. Doseg testova

Tijekom razvoja softvera baziranog na mikroservisnoj arhitekturi testove piše i provodi razvojni tim. Što se tiče izvođenja testova ono bi trebalo biti automatizirano i trebalo bi se koristiti kao dio razvojnog ciklusa softvera što znači da bi se trebali izvoditi svaki put kada se mijenja određena funkcionalnost ili dodaje nova. U slučaju neprolaska jediničnih ili integracijskih testova promjene bi trebale biti pregledane i popravljene od strane razvojnog tima. Kada promjene programskog kôda uspješno prođu testove tada se nova verzija softvera može poslati na isporuku.

Količina, doseg i važnost navedenih testova ne uključujući *lint* testiranje se najbolje mogu izraziti piramidom testova (eng. *pyramid test*).



Slika 8. Piramida testova (Prema: [21, str. 312])

Broj pojedinačnih testova bi trebao biti manji gledajući od dna prema vrhu piramide zato što testovi prema vrhu piramide postaju krhki (jedna promjena u korisničkom sučelju može

mnoge testove napraviti neuspješnim), skupi i dugotrajni za pisanje. Drugim riječima, trebalo bi najviše pisati jedinične, zatim integracijske pa testove od kraja do kraja. Često zna doći do invertiranja piramide što predstavlja anti-obrazac (eng. *anti-pattern*) u programskom inženjerstvu. Anti-obrazac je nešto što nije preporuka raditi pa se primjerice zna dogoditi da postoji malo jediničnih, a puno integracijskih testova ili testova od kraja do kraja.

5.2.5.2. Implementacija testova

Jedinični testovi su poprilično jednostavni za napraviti i većina radnih okvira i programskih jezika ima dobru podršku za takvu vrstu testova. Integracijski testovi su u mikroservisnoj arhitekturi kompliciraniji nego u monolitnoj zbog prirode same arhitekture. Mikroservisi će najčešće komunicirati s drugim mikroservisima u svrhu ispunjenja neke od vlastitih poslovnih funkcija stoga je potrebno pronaći način kako izbaciti te druge mikroservise kako testirani mikroservis ne bi ovisio o njima. Način na koji se to radi je kreiranje komponente koja će imati pripremljene odgovore i sve pokrivene slučajeve u svrhu simuliranja komunikacije kao što bi i pravi sudionik imao. U kreiranoj komponenti koja simulira poziv na drugi mikroservis se najčešće ručno pripremi slučaj. Primjerice, ako imamo mikroservis koji npr. ovisi o mikroservisu koji upravlja stanjem računa korisnika u mrežnoj trgovini, umjesto poziva na mikroservis koji upravlja stanjem računa kreirat će se komponenta koja će simulirati odgovor u svrhu integracijskog testiranja. Na primjer, u komponentu koja simulira mikroservis za stanje računa može se upisati da kada se zatraži stanje računa za korisnika čiji je identifikator 1, da se tada vrati odgovor 123,00 HRK. Nije bitno koliki je iznos i koji je korisnik, u integracijskom testu je bitno samo da se ne poziva mikroservis već da se kreira komponenta koja će simulirati odgovor na poseban slučaj što je u ovom slučaju korisnik s identifikatorom 1. Sada je u toj komponenti moguće pripremiti više slučajeva i mogu se vraćati razni odgovori, npr. da korisnik nije pronađen ili da je korisnik u minusu i slično. Tako će se pokriti svi slučajevi i riješit će se ovisnost mikroservisa o drugom servisu. Ovakva implementacija integracijskih testova predstavlja uklanjanje ovisnosti (eng. *stubbing*) odnosno kreiranje komponente koja oponaša servis (eng. *stub service*). Drugi pristup predstavlja korak dublje i umjesto oponašanja zapravo će se i poslati zahtjev samo ne na pravi mikroservis već na lažnog suradnika koji se također mora kreirati. Cilj ovog pristupa je provjeriti hoće li se zahtjev zapravo poslati i hoće li biti uspješan [15, str. 137].

Implementacija testova od kraja do kraja je kompliciranija jer se nastoji testirati cijeli ekosustav mikroservisa. Drugim riječima, potrebno je isporučiti sve mikroservise i onda pokrenuti testove koji će simulirati akcije korisnika u globalnom dosegu. Tu se javlja problem jer implementacija testova od kraja do kraja u svakom mikroservisu implicira dupliciranje programskog kôda jer će u većini slučajeva mikroservisi dijeliti zadatke pojedine poslovne

funkcije. Nadalje, ako se testovi od kraja do kraja implementiraju u svakom servisu potrebno je odrediti koju verziju ostalih mikroservisa koristiti prilikom testova – one koje su u produkciji ili one koje su u redu čekanja za isporuku. Ovakvi problemi se rješavaju implementacijom testova od kraja do kraja na jednom mjestu u jednoj fazi ispitivanja koja se pokrene svaki put kada se isporuči nova verzija bilo kojeg mikroservisa. Tako se testovi neće duplicirati i pokretat će se prilikom isporuke nove produkcijske verzije bilo kojeg servisa. Iako testovi od kraja do kraja mogu biti korisni, oni su kompleksni, krhki, teški za održavati i skupi stoga ih je preporuka koristiti samo u posebnim slučajevima.

5.2.6. Kontejnerizacija

Autonomnost mikroservisa donosi razne prednosti za cjelokupni sustav, no sa sobom također donosi i veću složenost sustava zbog većeg broja servisa koji se moraju održavati. Dok se u monolitnoj arhitekturi softver najčešće pakira u jednu izvršnu datoteku, u distribuiranoj arhitekturi stvari se zakompliciraju jer se sustav raščlanjuje na varijabilan broj mikroservisa koje je potrebno isporučiti. Nadalje, sa skaliranjem cjelokupnog sustava vjerojatno će biti potrebno isporučivati više instanci jednog servisa kako bi se sustav efikasnije rasteretio. Kako bi se riješili problemi isporučivanja i skaliranja distribuiranog sustava javili su se kontejneri (eng. *containers*) čiji je nagli skok u popularnosti 2010-ih godina omogućio učvršćavanje pozicije mikroservisne arhitekture kao dominantne na IT tržištu.

Kontejneri pružaju virtualno okruženje koje pakira sve aplikacijske ovisnosti, procese, metapodatke, strukturu direktorija itd. kako bi se omogućilo pokretanje aplikacije neovisno o infrastrukturi na kojoj se pokreće. Zapakirana aplikacija se onda može pokretati na isti način i na svim vrstama poslužitelja (npr. bilo koje osobno računalo) i njihovim okruženjima. Najpoznatija i najkorištenija tehnologija za kontejnerizaciju aplikacija je Docker koji koristi klijent-server arhitekturu za pružanje svih usluga, a neke od njegovih najbitnijih komponenata su [22]:

- Docker poslužitelj – nalazi se na glavnom sustavu (poslužitelju, npr. osobno računalo) i upravlja svim kontejnerima pokrenutim na glavnom sustavu.
- Docker kontejner – virtualni sustav koji sadrži sve ovisnosti, datoteke, procese i slično potrebno za pokretanje servisa. U suštini predstavlja instancu Docker slike koja se može pokretati, zaustavljati, distribuirati ili brisati.
- Docker klijent – korisničko sučelje (ili naredbeni redak) koje se koristi za komunikaciju s Docker poslužiteljem.
- Docker slike – datoteke predloška Docker kontejnera koje se mogu samo čitati. Ove datoteke sadrže skup instrukcija za kreiranje kontejnera na Docker platformi. Slike se mogu verzionirati, distribuirati i također se može vidjeti razlika

između slika, npr. u slučaju da se doda nova funkcionalnost u aplikaciji (servisu) može se vidjeti točna razlika nove i prethodne slike.

- Docker registar – spremište za dijeljenje i spremanje Docker slika, a prema zadanim postavkama to je Docker Hub, javni registar kojeg može koristiti bilo tko.
- *Dockerfile* – tekstualna datoteka koja omogućuje specificiranje naredbi za kreiranje Docker slika poput uputa za instaliranje ovisnosti aplikacije, postavljanje vrijednosti varijabla okoline, radne direktorije i slično.

Kontejneri i slike kontejnera su postali popularni zato što predstavljaju temelj za izgradnju pouzdanih i učinkovitih distribuiranih sustava jer je moguće ostvariti uzorak ponovno iskoristivih komponenti koje omogućavaju učinkovitiju i pouzdaniju izgradnju distribuiranih sustava zato što:

- Omogućavaju učinkovitije korištenje sistemskih resursa jer kontejneri nemaju vlastiti OS za razliku od virtualnih strojeva (eng. *virtual machines* ili kraće VM).
- Omogućavaju brže cikluse isporuke softvera zbog lagane isporučivosti, verzioniranja i portabilnosti samih servisa u formi Docker kontejnera.

Iz navedenog se može vidjeti da svojstvo kontejnerizacije tj. onog što Docker nudi i mikroservisne arhitekture idu ruku pod ruku – lagano, portabilno, izolirano i autonomno. No, Docker ne rješava sigurnosne probleme nego upravo naprotiv – donosi sa sobom vlastiti skup sigurnosnih problema kojima je potrebno upravljati. Nadalje, za korištenje Dockera treba svladati novi skup vještina i tehnologija i njegova krivulja učenja je često strma jer ne predstavlja klasičan alat kojeg programeri koriste [22].

5.2.7. Sigurnost

Poslovni pritisak na tvrtke za bržu isporuku boljeg softvera i plasiranja istog na tržište neprestano raste što ponekad zna biti uzrok zanemarivanju sigurnosti sustava. Kako bi se softver maksimalno zaštitio tvrtka mora poduzeti razne sigurnosne korake neovisno o tome radi li se o monolitnoj ili mikroservisnoj arhitekturi, a neki od najčešćih koraka su:

- Fizička zaštita hardvera.
- Enkripcija podataka u prijenosu i mirovanju.
- Implementacija autentifikacije i autorizacije.
- Prakticiranje politike zakrpe ranjivosti softvera.

Osnovni koncepti sigurnosti softvera tijekom korištenja su autentikacija i autorizacija. Autentikacija je postupak provjere identiteta korisnika ili procesa dok je autorizacija provjera

ovlasti korisnika ili procesa za akciju koju želi izvršiti. Kako bi se autentikacija i autorizacija sigurno provodile postoje neke generalne značajke koje se moraju poštivati poput:

- Mrežna komunikacija treba biti enkriptirana, npr. koristeći protokole *Hypertext Transfer Protocol Secure* (ili kraće HTTPS) ili *Transport Layer Security* (ili kraće TLS).
- Svi zahtjevi pristupa moraju biti autenticirani – to znači da iako se osoba uspješno autenticirala svaki sljedeći zahtjev bi trebao biti ponovno autenticiran.
- Lozinke, certifikate i bilo kakve tajne ne smiju biti upisane (eng. *hardcoded*) u programski kôd.

Što se tiče autentikacije korisnika i autorizacije akcija, one se najčešće obavljaju u sljedećim koracima:

- Korisnik unosi vjerodajnice, npr. korisničko ime i lozinku i šalje zahtjev za prijavu.
- Zahtjev za prijavu se obrađuje tako što se vjerodajnice verificiraju, stvara se sjednica (eng. *session*) i u nju se spremaju informacije o autenticiranom korisniku.
- Na korisničku stranu se vraća žeton (eng. *token*) koji se sprema u spremište korisničkog dijela aplikacije i šalje se svakim sljedećim zahtjevom.
- Svaki sljedeća akcija od strane korisnika se presreće u pozadinskom dijelu sustava prije nego što se izvrši i provjerava se ispravnost i datum isteka poslanog žetona u zahtjevu.
- Ako je provjera žetona uspješna obavlja se autorizacijska provjera smije li korisnik obaviti traženu akciju.
- Ovisno o autorizacijskoj provjeri, akcija se izvršava ili odbija.

Za autorizaciju softveri najčešće koriste provjeru baziranu na ulogama i ovlastima. Takva sigurnost temeljena na ulogama svakom korisniku u sustavu dodjeljuje jednu ili više uloga, a svaka uloga ima popis dozvoljenih akcija i na temelju korisnikove uloge provjerava se smije li korisnik obaviti traženu akciju. Primjerice, u sustavu mrežne trgovine mogu postojati uloge „Administrator“ i „Gost“ i za ulogu „Gost“ neće biti dozvoljeno da registrira nove korisnike u sustav.

Zbog svoje distribuirane prirode mikroservisna arhitektura ima više specifičnih ranjivosti što je čini podložnim za maliciozne napade. U monolitnoj arhitekturi komunikacija između komponenti softvera je zatvorena jer se radi o jednoj aplikaciji dok mikroservisi trebaju međusobno komunicirati kako bi razmjenjivali podatke. To donosi neke specifične probleme, npr. u mikroservisnoj arhitekturi se mora implementirati mehanizam za prijenos identiteta

korisnika ili procesa iz jednog mikroservisa u drugi. Nadalje, zbog svoje poliglotske prirode razvojni timovi mikroservisa trebaju imati više sigurnosne stručnosti kako bi riješili izazove koje donosi MSA. Osim toga, mikroservisi se često kontejneriziraju a rješenja bazirana na kontejnerizaciji donose svoj skup prijetnji poput ranjivosti slika, nesigurnih konekcija s registrima ili neograničenog administratorskog pristupa koje tradicionalni sigurnosni alati ne rješavaju.

5.2.7.1. Ključne značajke

Autentikacija i autorizacija su osnovni koncepti sigurnosti softvera i koriste se u skoro svakom većem sustavu gdje je potrebno ograničavanje pristupa korisnicima ili procesima. U mikroservisnoj arhitekturi autentikacija i autorizacija se često obavljaju u API pristupniku koji najčešće prima kontekst klijenta (eng. *client context*) koji sadrži osnovne informacije o klijentskoj aplikaciji koja šalje zahtjev i kontekst korisnika (eng. *user context*) koji sadrži osnovne informacije o krajnjem korisniku. Nakon autentikacije i autorizacije korisnika u API pristupniku ili vanjskom servisu (API pristupnik ne mora samostalno obavljati autentikaciju i autorizaciju već može preusmjeriti zahtjev na drugi servis) on prosljeđuje zahtjev s kontekstom klijenta i kontekstom korisnika određenom mikroservisu što osigurava pozadinski sustav na rubu i drži mikroservise izoliranim.

Mikroservisi mogu komunicirati i međusobno kada je to potrebno što znači da ne moraju implementirati uzorak API kompozitora. U tom slučaju mikroservis kada treba proslijediti zahtjev drugom mikroservisu mora prenijeti vlastiti identitet i mora propagirati kontekst korisnika (ponekad i kontekst klijenta). U slučaju da se za komunikaciju servisa koristi HTTP propagiranje korisnika se često prakticira koristeći sljedeća dva pristupa [23]:

- Slanje konteksta korisnika putem HTTPS zaglavlja – korisnik se može lako izvući u mikroservisu koji prima zahtjev, no tada taj mikroservis mora vjerovati da je poslani zahtjev došao iz pouzdanog izvora i da kontekst korisnika nije promijenjen u transportu.
- Korištenje JSON mrežnog žetona (eng. *JSON Web Token* ili kraće JWT) – kreira se žeton koji sadrži zaglavlje, korisnika u poruci (eng. *payload*) i potpis kojeg mikroservis koji prima zahtjev može provjeriti i potvrditi da kontekst korisnika nije promijenjen.

Korištenje JSON mrežnog žetona nema razlike od HTTPS zaglavlja kada mikroservis koji šalje zahtjev drugom mikroservisu sam izdaje žeton jer ga on može i mijenjati. Sigurniji način je koristiti vanjski tj. udaljeni servis za izdavanje sigurnosnih žetona kojem svi mikroservisi vjeruju i tada promijenjeni žeton može biti uočen provjerom potpisa. Nedostaci JWT-a su ti da jednom kada je žeton izdan on je neopoziv i trajat će do njegovog datuma

isteka. Jedan od najpoznatijih i najčešće korištenih protokola koji rješava taj problem je OAuth 2.0 koji je postao *de facto* sigurnosni standard za autorizaciju različitih servisa (ne nužno u mikroservisnoj arhitekturi). OAuth 2.0 pruža skup pravila i procedura za autentikaciju i autorizaciju korisnika i procesa i definira posebne žetone (eng. *refresh token*) koji se mogu opozvati i služe za osvježavanje pristupnih žetona (eng. *access token*).

Osim autentikacije i autorizacije krajnjeg korisnika u pozadinskom sustavu treba misliti i o autentikaciji i autorizaciji međuservisne komunikacije. Prva opcija za osiguravanje komunikacije između servisa je vjerovanje mreži (eng. *trust-the-network*). Ovakav pristup ne provodi nikakve sigurnosne korake u međuservisnoj komunikaciji već se oslanja na sigurnost na mrežnoj razini koja mora jamčiti da nitko ne može presresti komunikaciju između mikroservisa. Drugim riječima, svaki servis vjeruje drugom servisu i podacima koji dolaze unutar njihovog dijeljenog perimetra. Ovakva vrsta zaštite je dovoljna za neke tvrtke koje ne zahtijevaju visoku sigurnost podataka korisnika, no ako napadač prodre u mrežu neće postojati skoro nikakva zaštita za klasični napad čovjeka u sredini (eng. *man-in-the-middle attack*). Druga opcija za osiguravanje komunikacije između servisa je koristiti međusobni TLS (eng. *mutual* TLS ili kraće mTLS). Ovdje se provjera zahtjeva obavlja na transportnom sloju i svaki servis koristi par javnih i privatnih ključeva za provjeru autentičnosti poslanog zahtjeva. Ovakav tip provjere je sigurniji od vjerovanja mreži, no sa sobom donosi razne operativne izazove poput upravljanja certifikatima i omogućavanja opoziva, rotacije i nadzora ključeva. Treća opcija je koristiti uzorak pristupnog žetona poput već spomenutog JWT-a, samo ovaj put za autentikaciju servisa, a ne korisnika ili procesa. Ovakva vrsta provjere se obavlja na aplikacijskom sloju za razliku od mTLS-a koji se obavlja na transportnom sloju i mikroservisi prihvaćaju zahtjev od drugih servisa samo ako je certifikat koji je korišten za potpis JWT-a poznat i vjerodostojan. Kod ovakve komunikacije svaki servis koristi certifikat za potpis JWT-a [23].

5.2.7.2. Preporuke

Postoje razni sigurnosni standardi koji bi se trebali prakticirati od kojih su bitniji [15]:

- Određeni podaci (npr. lozinke) bi trebali biti enkriptirani tako da napadač ne može ništa učiniti s njima čak i ako uspješno dođe do njih.
- Uvijek bi trebalo koristiti provjerene i često korištene industrijske algoritme za enkripciju, a nikada izmišljati vlastite.
- Sigurnosne kopije važnih podataka treba redovito ažurirati.
- Sigurnosne stijene (eng. *firewalls*) su razumne mjere predostrožnosti i nekada se isplati imati više od jedne.

- Zapisivanje (eng. *logging*) ne predstavlja sigurnosnu prevenciju ali može pomoći u otkrivanju i oporavku nakon napada.
- Ažuriranje operacijskog sustava, vlastitog softvera, biblioteka i ovisnosti bi se trebalo redovito izvoditi kako napadači ne bi iskorištavali starije sigurnosne ranjivosti.

5.2.8. Isporuka

Dok je isporuka monolitnog sustava poprilično jednostavan postupak, mikroservisna arhitektura predstavlja malo složeniji proces s obzirom na to da se radi o više autonomnih, međusobno povezanih aplikacija koje mogu biti napisane u različitim programskim jezicima, radnim okvirima i njihovih verzijama. Nadalje, kako bi se omogućio veći broj zahtjeva po sekundi, protok i dostupnost informacija što poboljšava skaliranje sustava svaki mikroservis bi trebao biti isporučen u varijabilan broj instanci na temelju opterećenja određenog servisa kako bi se dolazeći zahtjevi mogli ravnomjerno raspodijeliti. Ovisno o veličini sustava to bi značilo da se u produkciji istovremeno može vrtjeti stotine ili tisuće instanci mikroservisa stoga treba omogućiti visoko automatiziranu infrastrukturu za isporuku jer ručna konfiguracija poslužitelja i pojedinih servisa u takvoj situaciji nije praktična. Finalno, ovisno o vrsti isporuke distribuirane arhitekture treba osigurati pouzdan nadzor mikroservisa kako bi se greške u produkcijskom okruženju što prije uočile i otklonile. Postoji više različitih uzoraka za isporuku mikroservisa, a neki od najčešće korištenih su [18]:

- Više instanci servisa po poslužitelju (eng. *multiple service instances per host*).
- Pojedinačna instanca servisa po poslužitelju (eng. *single service instance per host*) – ovdje postoje dvije specijalizacije uzorka:
    - Instanca servisa po VM-u (eng. *service instance per VM*).
    - Instanca servisa po kontejneru (eng. *service instance per container*).
- Isporuka bez poslužitelja (eng. *serverless deployment*).

5.2.8.1. Više instanci servisa po poslužitelju

Više instanci servisa po poslužitelju u neku ruku predstavlja tradicionalni način isporuke softvera i podrazumijeva izvođenje više instanci različitih servisa na fizičkom poslužitelju ili VM-u. U ovom uzorku svaka se instanca servisa izvodi na poznatom mrežnom priključku i na jednom ili više poslužitelja. Velika prednost ovakvog načina isporuke je ta što je korištenje resursa relativno učinkovito zato što više instanci različitih servisa dijele poslužitelj i njegov operacijski sustav. Kod ovog uzorka isporuka je također relativno brza jer je potrebno samo kopirati servis na poslužitelj i pokrenuti ga. No, ovaj uzorak ima više nedostataka [18]:

- Teško je ograničiti resurse koje troši svaka instanca servisa i instanca servisa koja ne radi ispravno može konzumirati sve resurse poslužitelja.
- U slučaju da je više instanci servisa isporučeno u istom procesu ne postoji izolacija instanci i teško je nadzirati potrošnju resursa svake instance servisa.
- Postoji rizik od različitih verzija ovisnosti.
- Nadzor, upravljanje i isporuka servisa je kompleksnija.

5.2.8.2. Pojedinačna instanca servisa po poslužitelju

Za razliku od prethodno opisanog uzorka gdje se više instanci različitih servisa vrti na jednom poslužitelju u ovom se uzorku svaka instanca mikroservisa nalazi na vlastitom poslužitelju. Ovakav pristup donosi rješenje za nedostatke prethodnog pristupa jer čini servise izoliranim, ne postoji rizik od različitih verzija ovisnosti servisa i potrošnja resursa svakog mikroservisa je ograničena na dostupne resurse jednog poslužitelja. Nadalje, nadzor, upravljanje i isporuka pojedine instance svakog mikroservisa je jednostavnija. S obzirom na to da svaka instanca svakog mikroservisa treba imati svoj poslužitelj ovakav pristup uzrokuje manje učinkovito korištenje resursa u usporedbi s prethodnim pristupom jer je broj poslužitelja puno veći. Kao što je već rečeno, postoje dvije specijalizacije ovog uzorka.

Instanca servisa po VM-u podrazumijeva pakiranje svakog servisa kao slike virtualnog stroja i isporuku svake instance servisa kao zasebnog VM-a. Velika prednost ovakvog pristupa je ta što svaka instanca servisa radi u potpunoj izolaciji što znači da ima fiksnu količinu procesora i memorije i ne može uzimati resurse drugih servisa. Druga prednost je ta što postoji mnoštvo pružatelja zrele i bogate infrastrukture u oblaku poput AWS-a koji pruža uslugu pakiranja servisa u slike VM-ova i vrlo dobro skalira i balansira opterećenje servisa. Nedostatak ovakvog pristupa je taj što je zbog svoje veličine izrada VM slike spora i dugotrajna što također znači da se VM obično i sporo pokreće.

Instanca servisa po kontejneru podrazumijeva pakiranje svakog servisa kao slike kontejnera koristeći alat za kontejnerizaciju (npr. Docker) i isporuku svake instance servisa kao kontejnera. Prednosti ovakvog pristupa su lakoća skaliranja sustava zato što se broj instanci kontejnera lako može povećavati i smanjivati ovisno o opterećenju sustava. Isto kao i kod VM-ova, servisi su izolirani i tehnologija implementacije je enkapsulirana pa se svi mikroservisi pokreću i zaustavljaju na isti način. Nadalje, količina memorije i procesora je ograničena kao i kod VM-ova tako da mikroservisi ne mogu preuzimati sve resurse jedni od drugih. Razlika između kontejnera i VM-ova je ta što se za razliku od VM-ova kontejneri ekstremno brzo grade i pokreću. VM ima vlastiti operacijski sustav za razliku od kontejnera koji koristi operacijski sustav poslužitelja što znači da se prilikom pokretanja kontejnera pokreće samo aplikacija, dok se kod VM-ova pokreće cijeli operacijski sustav. No, s obzirom na to da

kontejneri koriste operacijski sustav poslužitelja na kojem se nalaze imaju više sigurnosnih rizika nego VM-ovi [18].

5.2.8.3. Isporuka bez poslužitelja

Isporuka bez poslužitelja se odvija tako da razvojni tim koristi usluge pružatelja infrastrukture za isporuku koja skriva bilo koji koncept fizičkog ili VM poslužitelja i kontejnera. Pružatelj usluga (najčešće u oblaku) uzima programski kôd mikroservisa i pokreće ga, a svaki zahtjev se naplaćuje na temelju utrošenog vremena i resursa, u ovom kontekstu memorije. Najčešće se to odvija tako da se programski kôd zapakira (npr. u .zip datoteku) i prenese (eng. *upload*) u infrastrukturu pružatelja usluge s opisom željenih karakteristika izvedbe. Uz karakteristike izvedbe najčešće se moraju pružiti i metapodaci koji između ostalog navode naziv funkcije koja se poziva za obradu zahtjeva. Za izolaciju mikroservisa se najčešće koriste kontejneri ili VM-ovi što znači da pružatelj usluga interno koristi neki od drugih uzoraka isporuke poput uzorka pojedinačne instance servisa po poslužitelju, no ti detalji su skriveni od razvojnog tima i nitko u organizaciji nije odgovoran za upravljanje servisima. Primjeri najvećih i najpoznatijih pružatelja usluga isporuke bez poslužitelja su AWS Lambda, Google Cloud Functions i Azure Functions. Iako su sva tri pružatelja usluge slična u karakteristikama i cijenama koje nude, AWS Lambda je zasigurno najpopularnija i najzrelija platforma koja se može koristiti za primjenu uzorka isporuke bez poslužitelja.

Jedna od prednosti isporuke bez poslužitelja je ta da organizacija nije odgovorna za upravljanje IT infrastrukturom što znači da se može usredotočiti na programski kôd i razvoj softvera. Nadalje, model plaćanja temeljen na broju zahtjeva i utrošenim resursima znači da će organizacija plaćati samo za posao kojeg mikroservisi stvarno obavljaju. Finalno, ovakav tip isporuke softvera je skalabilan jer pružatelji usluge automatski skaliraju mikroservise kako bi podnijeli veće opterećenje. Iako je ideja da nitko u organizaciji ne mora brinuti o bilo kojem aspektu poslužitelja jako privlačna postoje nedostaci ovakvog pristupa. Ovisno o pružatelju usluga servisi se najčešće mogu pisati u samo nekoliko programskih jezika (npr. najpopularniji pružatelj usluga AWS Lambda podržava samo Python, Javu i Node.js). Nadalje, servisi moraju biti napravljeni tako da ne zadržavaju nikakvo stanje podataka poput sjednica i baza podataka (eng. *stateless*) jer je moguće da pružatelj usluga koristi drugu instancu istog servisa za svaki zahtjev. Sljedeći nedostatak je taj da se servisi moraju pokretati i obavljati zahtjeve brzo jer može doći do isteka predviđenog vremena za obavljanje zahtjeva i ukidanje samog servisa. Zadnji nedostatak je taj da infrastruktura pružatelja usluge isporuke često ne podržava sve vrste servisa, npr. AWS Lambda ne podržava servise koji konzumiraju poruke trećeg posrednika (npr. RabbitMQ) i ovakvi servisi općenito nisu namijenjeni isporuci bez poslužitelja [18].

5.2.9. Nadzor

Nadzor je kritični dio bilo kojeg IT sustava zato što u svakom sustavu eventualno dođe do kvara. No, sustav se ne nadzire samo zbog uočavanja kvarova, nadzor također može dati vrijedne podatke o ponašanju cjelokupnog sustava i ti podaci se mogu koristiti za poboljšanje performansi sustava, ali i za uočavanje karakteristika softvera koje bi mogle dovesti do kvara. Distribuirana arhitektura sa sobom donosi razne prednosti, no također donosi povećanu kompleksnost kada se priča o nadzoru mikroservisa u produkcijskom okruženju. Za razliku od monolitne arhitekture u mikroservisnoj arhitekturi zahtjev može proći kroz više instanci različitih ili istih mikroservisa i potrebno je na neki način povezati podatke iz različitih servisa kako bi se dobila jasna slika stanja sustava u nekoj točki u povijesti.

Kod nadzora se najčešće mjeri broj obrađenih zahtjeva kroz određeno vrijeme, vrijeme potrebno za obradu zahtjeva i sistemske vrijednosti poput upotrebe procesora i iskorištenosti memorije. Osim toga, programeri bi trebali zapisivati (eng. *log*) ključne i važne informacije tijekom izvršavanja zahtjeva kako bi se bolje razumjelo stanje pojedinog mikroservisa u povijesti. Nadalje, implementacija djelotvornog i učinkovitog upozoravanja na određene ključne metrike sustava omogućava rješavanje potencijalnih problema prije nego što dođe do prekida rada. Razvojni timovi također mogu koristiti sučelja (eng. *dashboards*) koja mogu odražavati vrijednosti ključnih metrika, trenutačna stanja i statuse mikroservisa. U većim organizacijama najčešće postoji jasno definirana procedura za upravljanje incidentima koja definira postupak uklanjanja kvarova.

5.2.9.1. Zapisivanje

U distribuiranoj arhitekturi često se događa da je više mikroservisa uključeno u pružanje određene funkcionalnosti korisniku. Da stvar bude kompleksnija, kako bi se rasteretio sustav moguće je i da postoji više instanci jednog mikroservisa iza uravnoteživača opterećenja. U slučaju kvara pregledavanje dnevnika zapisa aplikacije, baza podataka i mreže iz toliko različitih izvora je nepraktično stoga se podaci koji se prikupljaju iz više mikroservisa i poslužitelja moraju centralizirati na jednom mjestu.

Zapisivanje stanja sustava u dnevnike je jedna od najbitnijih stavki praćenja sustava i implementira se u programskom kôdu svakog mikroservisa. Kada dođe do kvara u sustavu često je jedini način otkrivanja uzroka kvara pregledavanje dnevnika i otkrivanje stanja mikroservisa kako bi se utvrdilo zašto se zahtjev nije izvršio. Stanja se najčešće zapisuju u obične datoteke, a datoteke moraju pružiti dovoljno informacija kako bi programeri mogli zaključiti što je pošlo po zlu. Određivanje onog što se mora zapisivati ovisi o mikroservisu, a podaci koji bi se trebali zapisivati su oni koji su ključni za opis trenutnog stanja sustava tijekom izvršavanja određenog važnijeg zahtjeva. Podaci na razini infrastrukture i poslužitelja ne bi

trebao bilježiti sam mikroservis, već usluge i alati koji pokreću sam mikroservis. Korisnički identifikatori, imena, lozinke ili bilo koji privatni podaci koji bi mogli predstavljati sigurnosni rizik za sustav i za korisnika se nikada ne bi trebali zapisivati [19].

5.3. Tranzicija iz monolitne u mikroservisnu arhitekturu

Mikroservisna arhitektura ne predstavlja nekakav cilj prema kojemu svaki sustav treba težiti niti ona rješava sve izazove i probleme svakog sustava. Primjerice, autonomnost mikroservisa poboljšava autonomnost razvojnih timova zbog prirode distribuirane arhitekture gdje svaki tim može razvijati skup vlastitih mikroservisa, no autonomnost timova se može postići i u monolitnoj arhitekturi dodjeljivanjem odgovornosti određenih segmenata softvera ključnim ljudima koji najbolje razumiju te dijelove softvera. Nadalje, ako se nešto koristi od strane većeg broja ljudi i organizacija to ne znači da će to isto odgovarati za svakoga. Prije nego što se donese odluka za usvajanje mikroservisne arhitekture postoje neka ključna pitanja koja organizacija mora razmotriti, npr. što žele postići i jesu li razmotrili alternative mikroservisnoj arhitekturi. Finalno, potrebno je definirati ishod koji se želi postići tranzicijom na mikroservisnu arhitekturu jer će on definirati vrstu migracije i način dekompozicije sustava.

5.3.1. Uzorci dizajna

Za tranziciju na mikroservisnu arhitekturu nijedna tehnika nije univerzalno točna stoga je potrebno razumjeti prednosti i nedostatke svake i odabrati onu koja će najbolje udovoljiti organizacijskim potrebama, a neki od najpoznatijih uzoraka za tranziciju su [24]:

- *Strangler Fig Application.*
- *Parallel Run.*
- *Branch by Abstraction.*
- *Decorating Collaborator.*

5.3.1.1. *Strangler Fig Application*

Strangler Fig Application uzorak dobiva ime po biljkama čije sjeme klija u pukotinama povrh drugog drveća i njihove sadnice rastu svoje korijenje prema dolje i obavijaju drvo domaćina dok istovremeno rastu prema gore kako bi dosegle sunčevu svjetlost što ponekad može dovesti do toga da biljke „zadave“ domaćina i da on umre nakon čega biljka tada ostane sa šupljom središnjom jezgrom. Preneseno u kontekstu softvera koristeći *Strangler Fig Application* novi sustav bi u početku trebao biti ovisan i podržan od strane postojećeg sustava što znači da bi oba sustava koegzistirala neko vrijeme dok novi sustav u potpunosti ne zamijeni stari. U ovom uzorku dizajna migracija na novi sustav je inkrementalna i postupna što smanjuje

rizik i omogućava zaustavljanje migracije. U slučaju da je razvojnom timu programski kôd originalnog sustava dostupan i da su odlučili koristiti mikroservisnu arhitekturu s istim tehnologijama najčešće se kreće od kopiranja programskog kôda ili ponovnog implementiranja samih funkcionalnosti što često uključuje i refaktoriranje programskog kôda. Implementacija *Strangler Fig Application* uzorka dizajna se provodi u tri koraka [24]:

1. Identificiranje dijelova postojećeg sustava koji se žele migrirati.
2. Implementacija odabranog dijela u novom mikroservisu.
3. Preusmjeravanje poziva s monolita na novi mikroservis.

Ovakav način tranzicije omogućava prebacivanje funkcionalnosti na mikroservisnu arhitekturu bez mijenjanja postojećeg sustava, no potrebno je pripaziti na sinkronizaciju podataka jer je moguće da će monolit i dalje trebati podatke kojima sada upravlja mikroservis.

5.3.1.2. *Parallel Run*

Kod *Parallel Runa* se pozivaju oba sustava što omogućuje usporedbu rezultata kako bi se organizacija osigurala da su oni jednaki. Ovaj uzorak u neku ruku predstavlja testiranje novog sustava u produkcijskom okruženju jer novom sustavu omogućava provedbu svih mogućih scenarija. Unatoč pozivanju obje implementacije u paralelnom radu se obično stari sustav smatra izvorom istine sve dok organizacija ne potvrdi da se novom sustavu može vjerovati. U slučaju različitih podataka u paralelnom radu sustava potrebno je provjeriti koji su točni jer je moguće da novi mikroservis radi korektno dok stari sustav sadrži greške koje nisu bile poznate i samim time daje različite ishode. Osim što ovaj uzorak dizajna omogućava usporedbu rezultata oba sustava on također može dati odgovore djeluje li novi mikroservis u okviru prihvatljivih kvalitativnih i kvantitativnih parametara. To znači da bi uz usporedbu rezultata novi mikroservis trebalo podvrgnuti provjerama nefunkcionalnih aspekata softvera poput mjerenja vremena izvršavanja zahtjeva, ispunjavanja prihvatljive stope kvara i slično. Implementacija paralelnog rada nije trivijalna stvar i obično se koristi za one slučajeve u kojima se funkcionalnost koja se mijenja smatra visokim rizikom [24].

5.3.1.3. *Branch by Abstraction*

Uzorak *Branch by Abstraction* se koristi kada se nastoji u mikroservis izvući modul dublje iz monolita o kojemu drugi moduli u monolitu ovise. To znači da bi trebalo raditi promjene u programskom kôdu monolita gdje istovremeno mogu raditi drugi programeri što može kreirati smetnje i pomutnje u radu. *Branch by Abstraction* omogućava da se to inkrementalno odradi u postojećoj bazi kôda bez smetnje drugim programerima tako da se stvara nova implementacija postojeće funkcionalnosti i obje implementacije koegzistiraju u istoj verziji kôda bez smetnji. To se odrađuje u sljedećih pet koraka:

1. Kreiranje apstrakcije za funkcionalnost koja se planira zamijeniti.
2. Promjena klijenata koji koriste postojeću funkcionalnost na novu apstrakciju.
3. Kreiranje nove implementacije apstrakcije s prerađenom funkcionalnošću, a ta nova implementacija je mikroservis.
4. Prebacivanje apstrakcije da koristi novu implementaciju – tek sada se događa promjena ponašanja u sustavu.
5. Čišćenje apstrakcije i uklanjanje stare implementacije ako je sve u redu s novom implementacijom.

Ako je moguće preporuka je koristiti uzorak *Strangler Fig Application* prije *Branch by Abstractiona* jer je jednostavniji, no postoje situacije kada postoji previše ovisnosti o nekom modulu u monolitu. Uzorak *Branch by Abstraction* je koristan u takvoj situaciji u kojoj će trebati vremena za promjenu postojeće baze kôda ili gdje je moguće ometanje kolega u razvojnom timu. Nadalje, *Branch by Abstraction* pretpostavlja da se može mijenjati kôd postojećeg sustava. U slučaju da to nije moguće, mogu se koristiti drugi uzorci dizajna od kojih je jedan *Decorating Collaborator* [24].

5.3.1.4. *Decorating Collaborator*

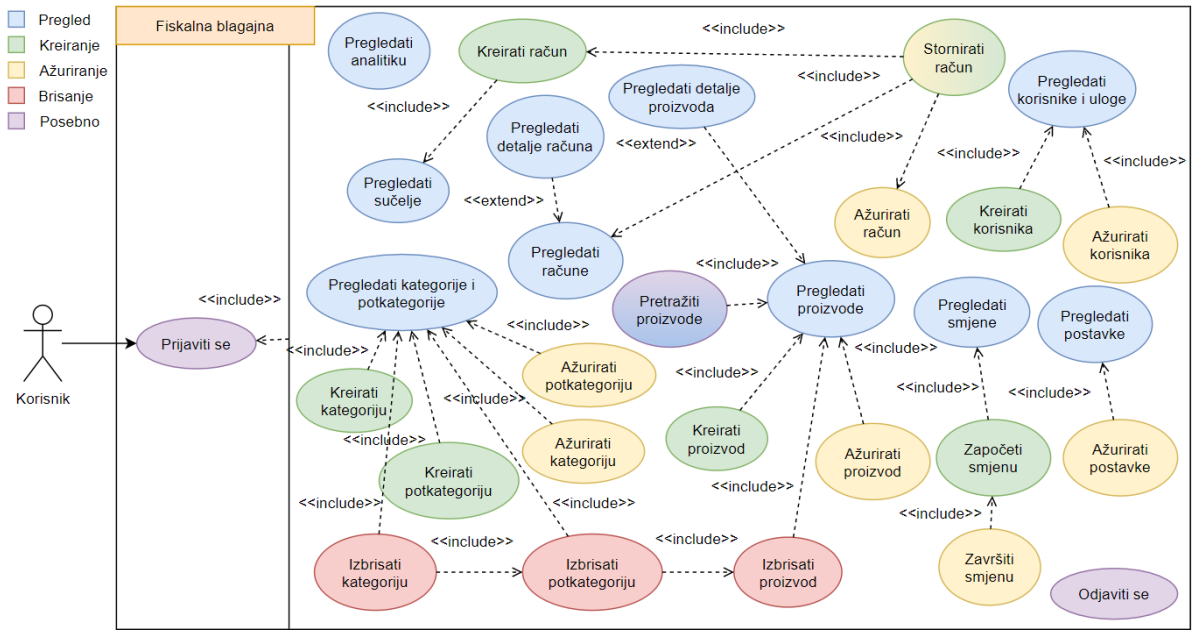
Uzorak *Decorating Collaborator* je prilagodba klasičnog *Decorator* uzorka dizajna koji omogućava da se nečemu pridruži nova funkcionalnost, a da temeljna stvar toga nije svjesna tj. da ne zna za to. Drugim riječima, *Decorator* omogućava dinamičko pridruživanje dodatne odgovornosti pojedinačnom objektu bez utjecaja na ponašanje drugih objekata iz iste klase što omogućava proširenje funkcionalnosti bazne klase [25]. *Decorating Collaborator* funkcionira na sličan način i koristi se kada se želi pokrenuti neko ponašanje na temelju nečeg što se događa unutar monolita, ali razvojni tim nije u mogućnosti mijenjati sam monolit. *Decorating Collaborator* funkcionira tako da se zahtjev prema monolitu odvija normalno, a odgovor od strane monolita se presreće i na temelju rezultata odgovora poziva se vanjski mikroservis. Nakon što zastupnik (eng. *proxy*) dobije odgovor od strane mikroservisa on taj odgovor prosljeđuje natrag klijentu. Iz tog razloga zastupnik ponekad može postati kompleksniji nego što je bila prvobitna zamisao stoga ga treba implementirati što jednostavnije kako ne bi postao zaseban mikroservis. Nadalje, moguće je da zahtjev i odgovor koji se šalju između klijenta i monolita neće imati sve informacije koje su potrebne mikroservisu i tada bi mikroservis trebao slati zahtjev monolitu kako bi dobio sve potrebne podatke. To će vezati novi mikroservis na monolitnu bazu podataka što će narušiti labavu povezanost mikroservisa stoga se ne preporučuje koristiti *Decorating Collaboratora* ako sve informacije nisu već prisutne u zahtjevu ili odgovoru [24].

6. Praktični dio

Praktični dio rada će demonstrirati neke od značajki i karakteristika mikroservisa opisanih u teoretskom dijelu rada. Koristeći metode i tehnike navedene u drugom poglavlju rada izrađena je stolna aplikacija *Fiskalna blagajna* koja se oslanja na mikroservisnu arhitekturu u pozadini. Postoje ukupno četiri mikroservisa od kojih svaki pokriva određen segment funkcionalnosti klijentske aplikacije, a komunikacija između aplikacije i mikroservisa se odvija putem REST API-ja. U nastavku poglavlja detaljnije će se predočiti cjelokupna implementacija korisničkog i pozadinskog dijela sustava.

6.1. Opis

Klijentska aplikacija *Fiskalna blagajna* napisana je u programskom jeziku Dart koristeći Flutter radni okvir i koristi resurse koje joj nude četiri mikroservisa: *Shop*, *Finance*, *Corporate* i *Accounts*. *Shop* i *Finance* mikroservisi su napisani u PHP-u koristeći Lumen radni okvir, dok su *Corporate* i *Accounts* mikroservisi napisani u Kotlinu koristeći Spring Boot radni okvir što demonstrira klasično svojstvo tehnološke heterogenosti mikroservisne arhitekture. Osim navedena četiri mikroservisa koristi se i platforma Auth0 koja nudi usluge autentikacije i autorizacije korisnika. *Fiskalna blagajna* je namijenjena bilo kojem poslovnom objektu koji treba sustav upravljanja poslovanja i izdavanja računa. Sama klijentska aplikacija nije fokus ovog rada, no vrlo je bitna jer demonstrira sve ono o čemu rad govori s pozadinske strane.

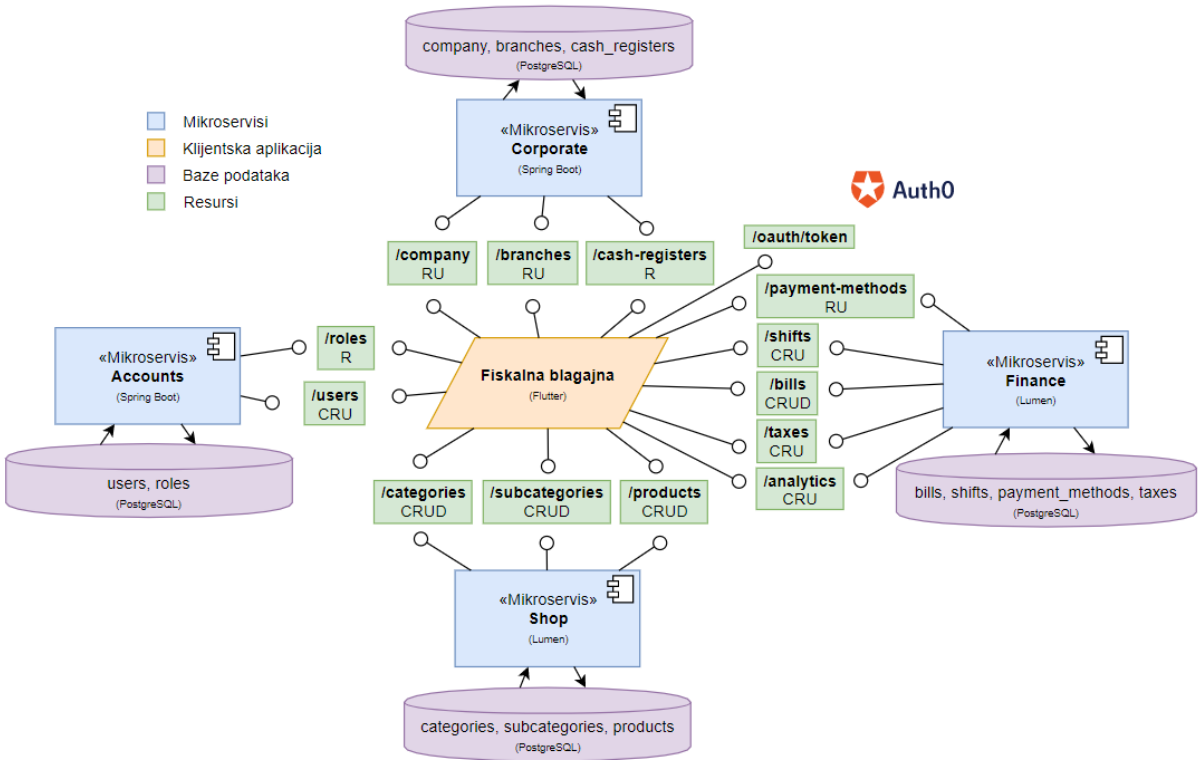


Slika 9. Dijagram slučajeva korištenja *Fiskalne blagajne*

Prethodna slika prikazuje slučajeve korištenja klijentske aplikacije iz perspektive korisnika. Slučajevi su grupirani po vrsti akcije i postoji pet standardnih vrsta akcije nad resursima: *Pregled*, *Kreiranje*, *Ažuriranje*, *Brisanje* (ove četiri akcije se često označavaju kraticom CRUD što na engleskom označava *Create, Read, Update, Delete*) i *Posebno*. Sve akcije su izolirane i zatvorene autentikacijskom provjerom stoga se korisnik prvo mora prijaviti kako bi mogao pristupiti resursima i svim dostupnim funkcionalnostima.

6.2. Arhitektura

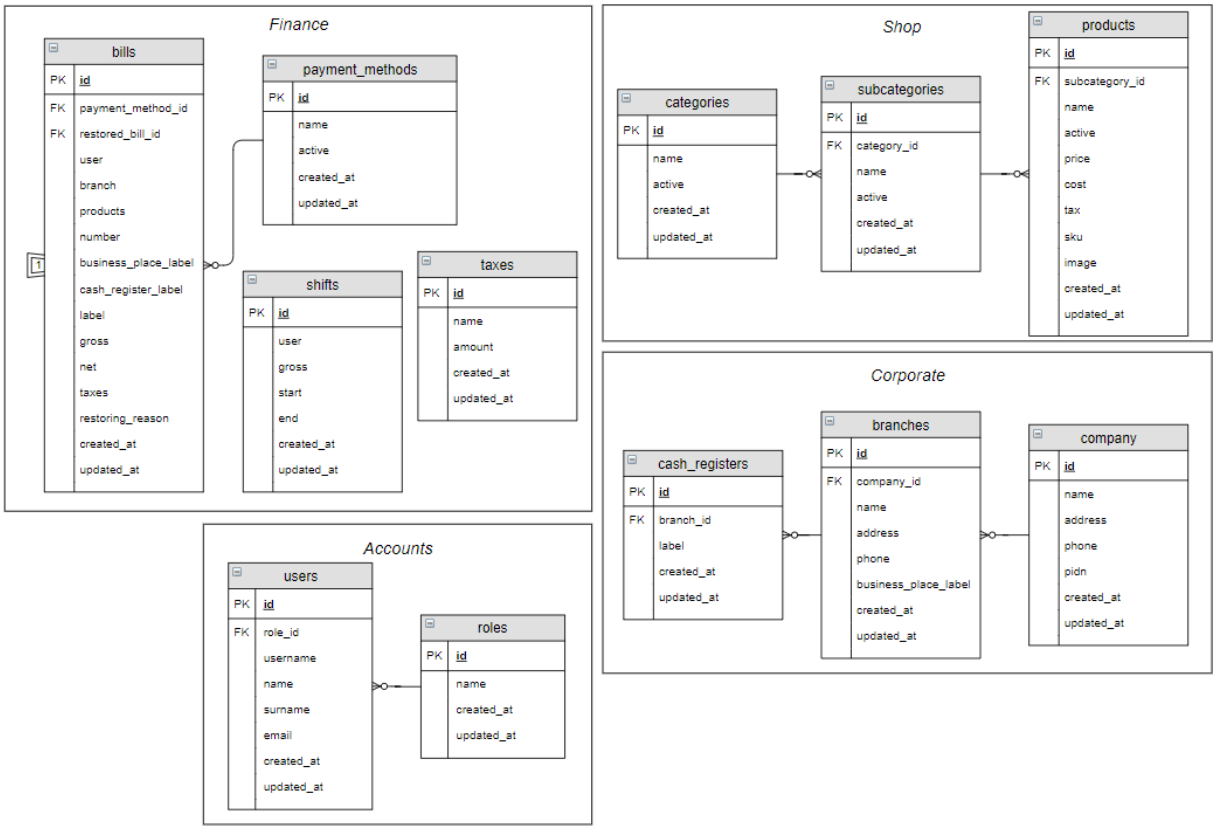
Kao što je već rečeno cjelokupni sustav se sastoji od autentikacijske platforme Auth0, četiri mikroservisa i klijentske stolne aplikacije. Dekompozicija mikroservisa se vršila po poslovnim funkcijama zamišljene organizacije i cilj je bio implementirati mali broj funkcionalnosti po mikroservisu kako bi se održao princip jedne odgovornosti, ali u isto vrijeme ne granulirati servise na premale komponente kako se cijeli sustav ne bi dodatno zakomplicirao. Sljedeća slika prikazuje sve dostupne resurse koje klijentska aplikacija može zvati.



Slika 10. Arhitektura cjelokupnog sustava

6.2.1. Baze podataka

Svi mikroservisi koriste PostgreSQL objektno-relacijski sustav baza podataka, a svaki mikroservis implementira uzorak baze podataka po servisu tj. ima svoju bazu podataka što znači da postoje ukupno četiri baze podataka. Sljedeća slika prikazuje shemu svih baza podataka.



Slika 11. Shema svih baza podataka

U Lumen radnom okviru veza s bazom podataka se može ostvariti ispunjavanjem zadanih varijabli koje se nalaze u *.env* datoteci okoline kao što prikazuje sljedeći isječak.

```
DB_CONNECTION=pgsql
DB_HOST=127.0.0.1
DB_PORT=5432
DB_DATABASE=shop
DB_USERNAME=postgres
DB_PASSWORD=postgres
```

Nakon ispunjavanja varijabli okoline potrebno je kreirati migracijske datoteke u kojima se definiraju tablice i atributi baze podataka. *Shop* mikroservis ima tri tablice stoga ima i tri

migracije koje stvaraju `categories`, `subcategories` i `products` tablice. Svaka migracija ima `up` metodu koja služi za stvaranje tablice i njezinih atributa i `down` metodu koja služi za poništavanje onoga što `up` metoda radi kako bi se promjene mogle poništiti u slučaju greške tijekom pokretanja migracija servisa. Lumen nudi `php artisan make:migration` naredbu za kreiranje migracijskih datoteka, a sljedeći isječak programskog kôda prikazuje sadržaj *2021_04_28_204647_create_products_table.php* migracijske datoteke koja stvara `products` tablicu.

```
class CreateProductsTable extends Migration
{
    public function up() {
        Schema::create('products', function (Blueprint $table)  {
            $table->id();
            $table->foreignId('subcategory_id')->constrained()-
>onDelete('cascade');
            $table->boolean('active')->default(true);
            $table->json('tax');
            $table->string('name')->unique();
            $table->integer('price');
            $table->integer('cost')->nullable();
            $table->string('sku')->unique();
            $table->string('image')->nullable();
            $table->timestamps();
        });
    }

    public function down()  {
        Schema::dropIfExists('products');
    }
}
```

Nakon kreiranja migracija moguće ih je pokrenuti `php artisan migrate` naredbom koja će pokušati stvoriti tablice definirane migracijskim datotekama u bazi podataka definiranoj u *.env* datoteci okoline. U *Finance* mikroservisu stvari funkcioniraju na isti princip dok se u *Corporate* i *Accounts* mikroservisima kreiranje tablica odvija na malo drugačiji način. Povezivanje na bazu se odvija na sličan način gdje se u *application.properties* datoteci postavljaju varijable okoline za povezivanje na bazu podataka.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/corporate
spring.datasource.username=postgres
```

```
spring.datasource.password=postgres
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
```

Umjesto kreiranja migracijskih datoteka i definiranja tablica i atributa u Spring Boot radnom okviru entiteti se mapiraju u tablice i atribute koristeći anotacije nad definiranim klasama. Sljedeći isječak programskog kôda prikazuje `Company` klasu u *Corporate* mikroservisu.

```
@Entity
@Table(name = "company")
data class Company(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long,
    var name: String,
    var address: String,
    var pidn: String,
    var phone: String,
    @CreationTimestamp
    @Column(name = "created_at", nullable = false, updatable = false)
    val createdAt: LocalDateTime = LocalDateTime.now(),
    @UpdateTimestamp
    @Column(name = "updated_at", nullable = false)
    var updatedAt: LocalDateTime = LocalDateTime.now()
)
```

Anotacija `@Entity` definira da se klasa `Company` može preslikati u istoimenu tablicu u bazi podataka i Spring Boot će prilikom pokretanja aplikacije kreirati tablicu `company` s atributima definiranim u samoj klasi. Ostale klase u servisu su definirane na isti način i tako se stvaraju sve tablice u samom mikroservisu.

6.2.1.1. Relacije između entiteta sadržanih u različitim bazama podataka

Kod mikroservisne arhitekture gdje se koristi uzorak baze podataka po servisu će eventualno doći do situacije u kojoj treba postojati veza između entiteta koji su pohranjeni u različitim bazama podataka. U aplikaciji *Fiskalna blagajna* potrebno je osigurati da svaki proizvod ima pridružen jedan porez, a na prethodnoj slici se može vidjeti da se proizvodi i

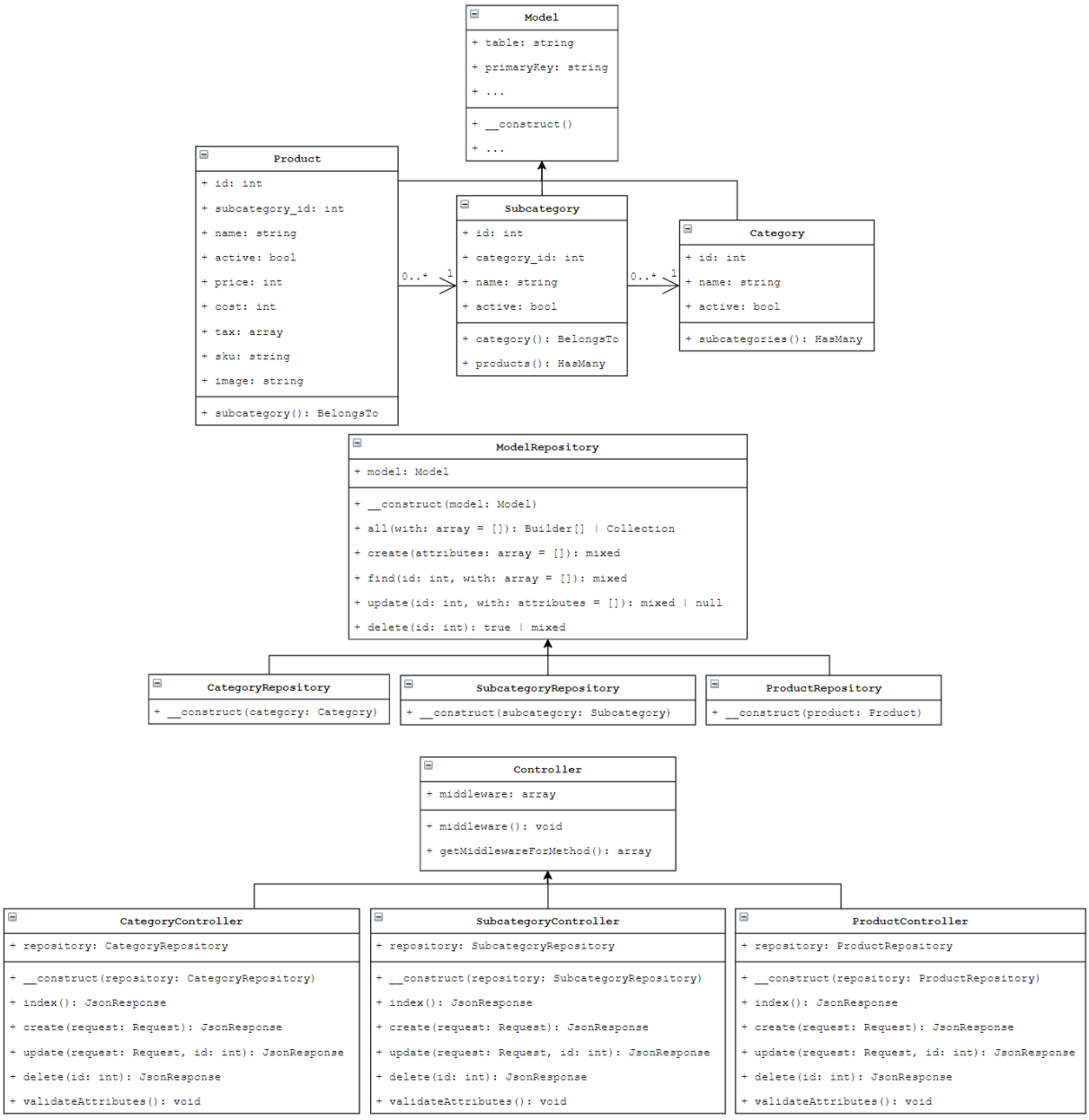
porezi nalaze različitim mikroservisima tj. u *Shop*, odnosno u *Finance* servisu. Taj problem se rješava tako da se u entitet proizvoda spremi cijela reprezentacija entiteta poreza pa se na prethodnoj shemi može vidjeti da tablica `products` sadrži atribut `tax`. S klijentske strane *Fiskalna blagajna* prilikom kreiranja i uređivanja proizvoda dohvaća sve poreze s *Finance* mikroservisa, a kada je proizvod spreman za pohranu uz njegove atribute se šalje i cijela reprezentacija entiteta odabranog poreza kojeg *Shop* mikroservis sprema u `tax` atribut. Većina radnih okvira pa i Lumen ima ugrađene metode u objektno-relacijskom mehanizmu (eng. *object-relational mapper* ili kraće ORM) za pretraživanje atributa koji su spremljeni kao JSON objekti, što je u ovom slučaju atribut `tax` stoga nije teško vršiti upite u kojima bi se npr. trebali grupirati proizvodi po njihovom porezu.

6.2.2. Specifičnosti mikroservisa

Shop i *Finance* mikroservisi implementiraju *Repository* uzorak dizajna, a kada se zahtjev autorizira `Authenticate` posrednik (eng. *middleware*) ga preusmjeri na upravitelja koji izvršava zahtjev koristeći jednu od metoda dostupnih u repozitoriju traženog entiteta. S obzirom na to da *Shop* i *Finance* mikroservisi ne sadrže *Service* sloj poslovna logika je podijeljena između upravitelja i repozitorija kako bi se stvari održale što jednostavnijim. Za razliku od *Shop* i *Finance* mikroservisa, *Corporate* i *Accounts* mikroservisi implementiraju *Service-Repository* uzorak dizajna što znači da postoji još jedan sloj apstrakcije koji izvršava poslovnu logiku i nalazi se između upravitelja i repozitorija. Ovo omogućuje finiju strukturu programskog kôda, no povećava broj klasa i zahtjev mora proći kroz još jedan sloj stoga se preporučuje koristiti samo u slučajevima gdje je poslovna logika kompleksnija. Ovo dodatno naglašava prednosti tehnološke heterogenosti jer osim što se mikroservisi mogu pisati u različitim tehnologijama i alatima, oni također mogu implementirati različite uzorke dizajna.

Repository i *Service-Repository* uzorci dizajna se mogu klasificirati kao slojevita arhitektura u mikroservisnoj arhitekturi, a sloj pogleda za razliku od monolitne arhitekture je potpuno odvojen i njega implementiraju klijentske aplikacije što je u ovom slučaju *Fiskalna blagajna*. Sljedeća slika prikazuje sve važnije klase u *Shop* mikroservisu i njihovu međusobnu povezanost. `Product`, `Subcategory` i `Category` klase nasljeđuju klasu `Model` koju pruža Lumen i ona omogućava reprezentaciju entiteta i njihovih relacija u proizvoljnoj bazi podataka što je u ovom slučaju PostgreSQL. Apstraktna klasa repozitorija `ModelRepository` sadrži instancu klase `Model` i pet metoda: `all`, `create`, `find`, `update` i `delete`. Svaki entitet ima svoju klasu repozitorija pa tako postoje `CategoryRepository`, `SubcategoryRepository` i `ProductRepository` koji osim pet baznih metoda mogu sadržavati njima specifične metode. Upravitelji koriste klase repozitorija u vlastitom konstruktoru kako bi mogli pretraživati, kreirati, uređivati i brisati entitete pa npr. `CategoryController` koristi

`CategoryRepository` koji nasljeđuje metode iz bazne klase `ModelRepository` za CRUD upravljanje entitetima.



Slika 12. Dijagram klasa *Shop* mikroservisa

S obzirom na to da je *Finance* mikroservis izrađen koristeći iste tehnologije, struktura klasa *Finance* servisa je slična samo što sadrži entitete kojima pokriva svoju domenu: `Bill`, `Shift`, `PaymentMethod` i `Tax`. S druge strane, *Corporate* i *Finance* mikroservisi kao što je već rečeno imaju i *Service* sloj pa je struktura klasa i tok zahtjeva malo drukčiji. Umjesto da je poslovna logika podijeljena između upravitelja i repozitorija, ona je smještena u *Service* sloju

što omogućava upraviteljima i repozitorijima da imaju samo jednu zadaću: primanje i vraćanje odgovora odnosno izvršavanje akcija nad spremištem podataka. Sljedeća slika prikazuje sve važnije klase u *Corporate* mikroservisu, a struktura *Accounts* mikroservisa prati isti uzorak samo što sadrži entitete `User` i `Role`.

Slika 13. Dijagram klasa *Corporate* mikroservisa

Za razliku od *Shop* i *Finance* mikroservisa gdje se repozitoriji instanciraju u upravitelju, ovdje se servisni sloj instancira u upravitelju i on zatim koristim funkcije pripadajućeg repozitorija za upravljanje entitetima nad spremištem podataka.

6.2.2.1. Tok zahtjeva

Kako bi se lakše pojmila prethodno opisana arhitektura pojedinih mikroservisa prikazat će se tok određenih zahtjeva koje izvršava klijentska aplikacija. Prilikom stvaranja nove smjene klijentska aplikacija šalje `POST` zahtjev na `/api/shifts` resurs *Finance* mikroservisa. Sljedeći isječak kôda prikazuje definiran resurs na *Finance* mikroservisu.

```
$router->group(['prefix' => 'api', 'middleware' => 'auth'], function
() use ($router) {
```

```
    ...
    $router->group(['prefix' => 'shifts'], function () use ($router)
{
        ...
        $router->post('/', 'ShiftController@create');
        ...
    });
    ...
});
```

Iz prikazanog isječka se može vidjeti da se zahtjev zatim preusmjerava na upravitelj `ShiftController` i njegovu metodu `create`. Sljedeći isječak programskog kôda prikazuje navedeni upravitelj, njegove najvažnije dijelove i metodu `create`.

```
class ShiftController extends Controller {
    private ShiftRepository $shiftRepository;

    public function __construct(ShiftRepository $shiftRepository) {
        $this->shiftRepository = $shiftRepository;
    }
    ...
    public function create(Request $request): JsonResponse {
        $this->validateAttributes($request);
        $shift = $this->shiftRepository->latest();
        if ($shift && !$shift->end) {
            return response()->json(['end' => ['Zadnja smjena nije
završila']], Response::HTTP_UNPROCESSABLE_ENTITY);
        }
        $data = [
            'user' => $request->input('user'),
            'start' => $request->input('start')
        ];
        $shift = $this->shiftRepository->create($data);
        return response()->json($shift, Response::HTTP_OK);
    }
    ...
    private function validateAttributes(Request $request, int $id =
-1) {
        $rules = [
            'user' => 'required|array',
```

```
            'user.id' => 'required|integer',
            'user.username' => 'required|string|max:255',
        ];
        if ($id != -1) {
            $rules['end'] = 'required|date_format:Y-m-d H:i:s';
        } else {
            $rules['start'] = 'required|date_format:Y-m-d H:i:s';
        }
        $this->validate($request, $rules);
    }
}
```

Upravitelj `ShiftController` ima konstruktor u kojemu se pomoću tehnike zvanom umetanje ovisnosti (eng. *dependency injection*) postavlja vrijednost privatne varijable `$shiftRepository` na repozitorij `ShiftRepository`. Nakon toga zahtjev ulazi u `create` metodu i prvo se poziva metoda `validateAttributes` koja provjerava poslane vrijednosti s klijentske aplikacije i osigurava da su svi podaci sadržani i da su ispravnog tipa. Nakon validacije parametara pomoću varijable `$shiftRepository` se dohvaća zadnja smjena. Ako smjena postoji i ako nije završena ne dopušta se kreiranje nove smjene i vraća se prikladan odgovor. U slučaju da je s podacima sve u redu i da nema aktivne smjene dopušta se kreiranje nove smjene koristeći `ShiftRepository` i ona se zatim vraća klijentskoj aplikaciji. Sličan tok zahtjeva se odvija i nad ostalim klasama u *Shop* i *Finance* mikroservisima.

U Spring Boot radnom okviru su stvari opet malo drukčije i ne postoji datoteka u kojoj se definiraju svi resursi već su oni definirani pomoću anotacija iznad klasa. Sljedeći isječak programskog kôda *Corporate* mikroservisa prikazuje `CashRegisterController` upravitelj.

```
@RestController
@RequestMapping("/api/cash-registers")
class CashRegisterController(val service: CashRegisterService) {
    @GetMapping("/{id}")
    fun read(@PathVariable id: String): Optional<CashRegister> =
        service.read(id)
}
```

Anotacijama `@RestController` i `@RequestMapping` se omogućava definiranje resursa `/api/cash-registers/{id}` koji podržava `GET` zahtjev što je osigurano `@GetMapping` anotacijom. Za dohvaćanje blagajne se zatim koristi servisni sloj `CashRegisterService` koji se instancira u konstruktoru samog upravitelja. Servisni sloj u

vlastitom konstruktoru koristi sloj repozitorija `CashRegisterRepository` kao što prikazuje sljedeći isječak kôda.

```
@Service
class CashRegisterService(val db: CashRegisterRepository) {
    fun read(id: String): Optional<CashRegister> =
        db.findById(id)
}
```

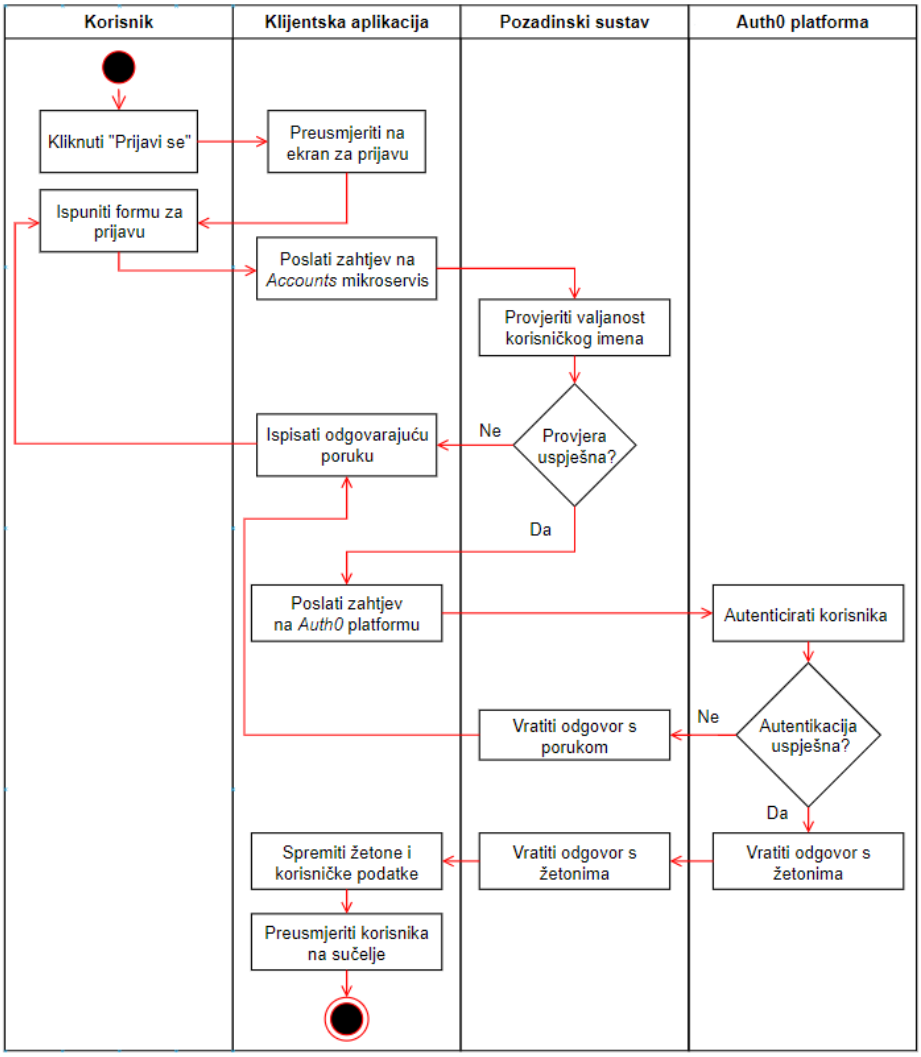
Anotacija `@Service` predstavlja specijalizaciju anotacije `@Component` i koristi se za označavanje klase kao pružatelja usluga i tu se najčešće odvija poslovna logika sustava. `CashRegisterRepository` repozitorij koristi sučelje `JpaRepository` koje pruža standardne metode za upravljanje Java objektima u relacijskim bazama podataka i u ovom slučaju se koristi metoda `findById` za pretraživanje blagajne po njezinom identifikatoru. Sljedeći isječak kôda prikazuje sučelje `CashRegisterRepository` koje nasljeđuje metode sučelja `JpaRepository`, a sličan tok zahtjeva se odvija i nad ostalim klasama u *Corporate* i *Accounts* mikroservisima.

```
interface CashRegisterRepository : JpaRepository<CashRegister,
String>
```

Prethodna dva primjera prikazuju *Repository* i *Service-Repository* uzorke dizajna i kako zahtjevi prolaze kroz slojeve različitih servisa. Dok se u *Shop* i *Finance* mikroservisima poslovna logika djelomično odvija u upravitelju, a djelomično u repozitoriju, u *Corporate* i *Accounts* mikroservisima poslovna logika je kompletno odvojena i izvršava se u vlastitom sloju. U primjeru s blagajnom nema nikakve kompleksne poslovne logike osim dohvaćanja blagajne, no u slučaju kompleksnijeg sustava u servisnom sloju bi se moglo umetnuti više različitih repozitorija i zatim kombinirati njihove podatke kako bi se formirali kompleksniji objekti i izvršavala složenija pravila. Bitno je napomenuti da prije nego što bilo koji zahtjev pristupi resursima bilo kojeg mikroservisa izvršava se njihova autentikacija i autorizacija kako bi se osiguralo da resursima mogu pristupiti samo ovlaštene osobe što bi u ovom kontekstu predstavljalo zaposlenike.

6.3. Implementacija sigurnosti

Autentikacija i autorizacija korisnika se odvija uz pomoć Auth0 platforme koja prilikom uspješne prijave izdaje JWT koji se pohranjuje u spremište klijentske aplikacije i koji se zatim šalje u zaglavlju svakog sljedećeg zahtjeva upućenog na neki od mikroservisa.



Slika 14. Dijagram slijeda prijave

U ovom primjeru se ne koristi API pristupnik koji obavlja autentikaciju i autorizaciju korisnika tako da se zahtjevi ne presreću u jednoj točki prije nego dođu do mikroservisa već svaki mikroservis obavlja to samostalno koristeći skup JSON mrežnih ključeva (eng. JSON *Web Key Set* ili kraće JWKS). JWKS je skup ključeva koji sadrži javne ključeve koji se koriste za provjeru bilo kojeg JWT-a izdanog od strane autorizacijskog poslužitelja koji je u ovom slučaju Auth0. Auth0 koristi privatni ključ prilikom izdavanja JWT-a, a mikroservisi, kada trebaju autenticirati JWT poslan od strane klijentske aplikacije dohvaćaju javno izložen JWKS i s njim provjeravaju i potvrđuju postojanost i ispravnost poslanog JWT-a. U svim mikroservisima svi resursi su obavijeni posrednikom koji obavlja autentikaciju zahtjeva i zatim autorizira ili odbija zahtjev ovisno o ispravnosti žetona. Sljedeći isječak programskog kôda prikazuje sve dostupne resurse koje *Shop* mikroservis nudi *Fiskalnoj blagajni*.

```
$router->group(['prefix' => 'api', 'middleware' => 'auth'], function
() use ($router) {
```

```
  $router->get('/dashboard, 'DashboardController@index');
  $router->group(['prefix' => 'categories'], function () use
($router) {
    $router->get('/', 'CategoryController@index');
    $router->post('/', 'CategoryController@create');
    $router->put('/{id}', 'CategoryController@update');
    $router->delete('/{id}', 'CategoryController@delete');
  });
  $router->group(['prefix' => 'subcategories'], function () use
($router) {
    $router->get('/', 'SubcategoryController@index');
    $router->post('/', 'SubcategoryController@create');
    $router->put('/{id}', 'SubcategoryController@update');
    $router->delete('/{id}', 'SubcategoryController@delete');
  });
  $router->group(['prefix' => 'products'], function () use ($router)
{
    $router->get('/', 'ProductController@index');
    $router->post('/', 'ProductController@create');
    $router->post('/{id}', 'ProductController@update');
    $router->delete('/{id}', 'ProductController@delete');
  });
});
```

Iz isječka se može vidjeti da su svi resursi obavijeni posrednikom `auth` koji je mapiran na posredničku klasu `Authenticate` u inicijalizacijskoj datoteci *app.php* Lumen radnog okvira:

```
$app->routeMiddleware([
  'auth' => App\Http\Middleware\Authenticate::class,
]);
```

Dakle, prije nego zahtjev dođe do traženog resursa mora proći autentikacijsku provjeru u `Authenticate` posredniku koji je odrađuje u `handle` metodi na sljedeći način:

```
public function handle(Request $request, Closure $next)
{
  $token = $request->bearerToken();
  if (!$token) {
    return response()->json('Žeton nije prisutan', 401);
  }
  $decoded = $this->validateToken($token);
```

```
  if ($decoded == null) {
    return response()->json('Pogrešan žeton', 401);
}
  return $next($request);
}
public function validateToken($token)
{
  try {
    $jwksUri = env('AUTH0_DOMAIN') . '.well-known/jwks.json';
    $jwksFetcher = new JWKFetcher(null, [ 'base_uri' => $jwksUri]);
    $signatureVerifier = new AsymmetricVerifier($jwksFetcher);
    $tokenVerifier = new TokenVerifier(env('AUTH0_DOMAIN'),
env('AUTH0_AUD'), $signatureVerifier);
    return $tokenVerifier->verify($token);
  } catch (InvalidTokenException $e) {
    return false;
  }
}
```

Prvo se provjerava da li je žeton prisutan, zatim se dohvaćaju javni ključevi s Auth0 platforme i na kraju se provjerava ispravnost potpisanog žetona. U slučaju da su sve provjere točne zahtjev se prosljeđuje na prvobitno ciljani resurs. *Finance* mikroservis autenticira zahtjeve na isti način, no on također sadrži i autorizaciju za resurs analitike kojoj u klijentskoj aplikaciji ne mogu pristupiti zaposlenici nego samo administratori. Prvo se u resursima definira da je za pristup analitici potrebna posebna ovlast *read:analytics* koja je definirana u Auth0 platformi:

```
$router->get('/analytics', ['middleware' => 'auth:read:analytics',
'uses' => 'AnalyticController@index']);
```

Zatim se u `handle` metodi `Authenticate` posredničke klase nakon autentikacije korisnika provjerava sadrži li žeton traženu ovlast:

```
if ($scopeRequired && !$this->tokenHasPermissions($decodedToken,
$scopeRequired)) {
  return response()->json('Nedovoljne ovlasti', 403);
}
```

Autentikacija zahtjeva *Corporate* i *Accounts* mikroservisa funkcionira na sličan princip specifičan Kotlinu i Spring Bootu gdje se u klasi `WebSecurity` i njezinoj metodi `configure` zabranjuje pristup svim resursima osim ako zahtjev nije autoriziran.

```
@Configuration
```

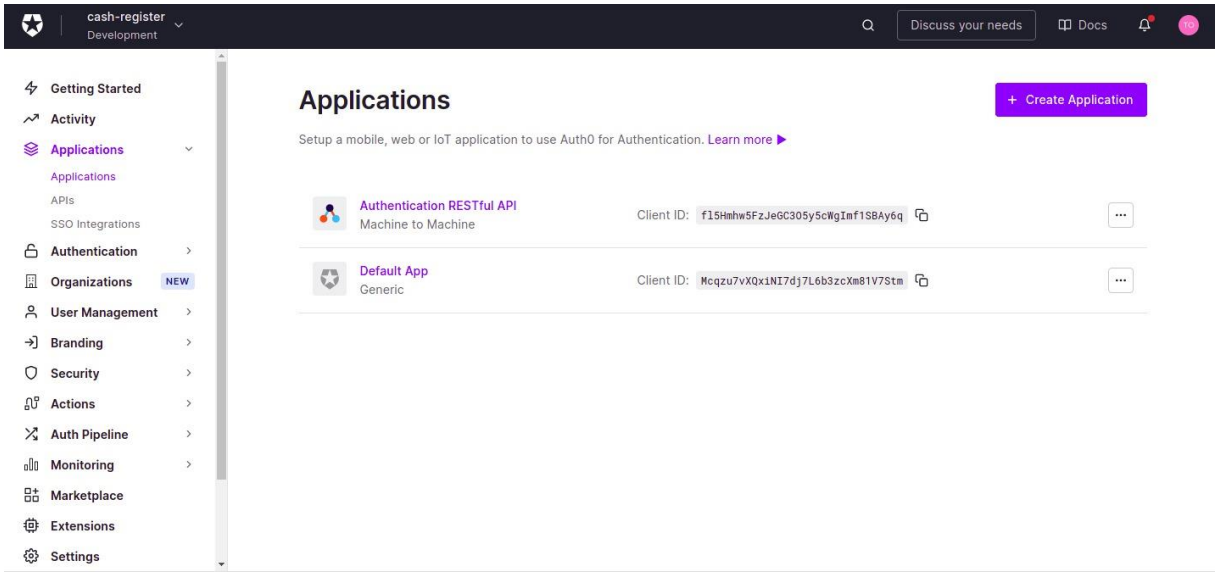
```
@EnableWebSecurity
class WebSecurity : WebSecurityConfigurerAdapter() {
  @Value("\${auth0.audience}")
  private val audience: String? = null

  @Value("\${auth0.issuer}")
  private val issuer: String? = null

  @Throws(Exception::class)
  override fun configure(http: HttpSecurity) {
    http.cors()
      .and()
      .authorizeRequests()
      .antMatchers("/")
      .permitAll()
      .anyRequest()
      .authenticated()

    JwtWebSecurityConfigurer.forRS256(audience, issuer!!)
      .configure(http)
  }
}
```

Ovakav način autentikacije zahtjeva pomoću JWKS-a umjesto korištenja API pristupnika ne uzrokuje moguću pojedinačnu točku kvara ili usko grlo, no zahtijeva implementaciju autentikacije i autorizacije u svakom servisu.

6.3.1. Auth0

Auth0 je prilagodljiva platforma za autentikaciju i autorizaciju korisnika i procesa putem mobilnih, mrežnih i pozadinskih aplikacija. Osim opisanog JWKS načina autentikacije zahtjeva, Auth0 pruža i razne druge kanale poput usluge kratkih poruka (eng. *short message service* ili kraće SMS), e-maila, društvenih mreža i slično. Uz razne kanale Auth0 također nudi široki raspon postavki, biblioteka s velikim brojem tehnologija, detaljnu dokumentaciju s jasnim primjerima programskog kôda i još mnogo toga što standardni softveri sa sustavom prijave inače trebaju. Nakon registracije na Auth0 stvara se administratorsko sučelje u kojemu je moguće kreirati aplikaciju u kojoj se mogu stvarati API-ji koji će služiti kao autentikacijska i autorizacijska platforma. Za novokreiranu aplikaciju i API-je Auth0 stvara i JWKS koji mikroservisima omogućava autentikaciju i autorizaciju korisnika. Aplikacija također sadrži

korisnike kojima se mogu dodjeljivati uloge, a uz uloge se mogu vezati ovlasti. Iako Auth0 platforma sadrži korisnike ona služi samo za njihovu autentikaciju i autorizaciju, a novokreirani korisnici se prvo spremaju u bazu podataka *Accounts* mikroservisa. Nakon toga administrator klijentske aplikacije novokreirane korisnike treba dodati u Auth0 platformu nakon čega se oni mogu prijaviti u aplikaciju. U kontekstu *Fiskalne blagajne* na Auth0 postoje uloge „Administrator„ i „Zaposlenik“ i kreirana je jedna ovlast *read:analytics* koja je dodijeljena ulozi „Administrator“. Administratori i zaposlenici generalno mogu odrađivati iste akcije, no pristupu analitici u *Fiskalnoj blagajni* mogu pristupiti samo administratori što osigurava *Finance* mikroservis. Klijentska aplikacija prilikom prijave sprema JWT koji sadrži sve ovlasti korisnika, a u *Finance* mikroservisu se definira da je za pristupanje analitici potrebna *read:analytics* ovlast. Nadalje, kontekst korisnika koji je sadržan u JWT-u se u slučaju međuservisne komunikacije prosljeđuje u zaglavlju zahtjeva.



Slika 15. Izgled Auth0 administratorskog sučelja

6.4. Jedinično testiranje

Stvaranje računa u *Fiskalnoj blagajni* predstavlja jednu od najvažnijih funkcionalnosti cjelokupnog softvera i zbog toga je u *Finance* mikroservisu napravljen jedinični test `CreateBillTest` koji provjerava `make` metodu u klasi `BillRepository`. U testu se korisnik, poslovnica i proizvodi kreiraju s bibliotekom `Faker` koja generira lažne podatke, a zatim se s generiranim podacima pokušava kreirati novi račun.

```
class CreateBillTest extends TestCase {
    public function setUp(): void {
        parent::setUp();
        $this->faker = Factory::create();
        ...
    }

    public function testCreateBill() {
        $data = [
            'user' => $this->user,
            'branch' => $this->branch,
            'cashRegisterLabel' => $this->faker->numberBetween(1,5),
            'payment_method_id' => $this->faker->numberBetween(1,2),
            'products' => $products
        ];

        $bill = $billRepository->make($data);
        $this->assertInstanceOf(Bill::class, $bill);
    }
}
```

Cilj ovog testa je osigurati se da ova komponenta softvera ispravno funkcionira i da ne sadrži nikakve greške. Nadalje, pokretanjem ovog testa nakon ažuriranja strukture računa ili proizvoda se može provjeriti da nije došlo do promjena koje mogu uzrokovati neispravnost softvera (eng. *breaking changes*).

Osim stvaranja računa još jedna bitna funkcionalnost u *Fiskalnoj blagajni* je kreiranje proizvoda i zbog toga je kreiran test `CreateProductTest` u *Shop* mikroservisu.

```
class CreateProductTest extends TestCase {
    private \Faker\Generator $faker;

    public function setUp(): void {
        parent::setUp();
        $this->faker = Factory::create();
    }

    public function testCreateBill() {
        $this->withoutMiddleware();
        $data = [
```

```
            'name' => $this->faker->word,
            'sku' => $this->faker->word,
            'subcategory_id' => $this->faker->numberBetween(1,10),
            'active' => $this->faker->boolean,
            'price' => $this->faker->numberBetween(100,5000),
            'cost' => $this->faker->numberBetween(100,5000),
            'image' => $this->faker->imageUrl(),
            'tax' => [
                'id' => $this->faker->numberBetween(1,2),
                'name' => $this->faker->word,
                'amount' => $this->faker->numberBetween(1,100)
            ]
        ];
        $this->post('/api/products', $data);
        $this->assertResponseStatus(Response::HTTP_OK);
    }
}
```

Prethodni isječak kôda prikazuje da test zapravo i zove resurs `/api/products` kako bi se testiralo hoće li se zahtjev ispravno odraditi u `ProductController` upravitelju. Prije toga se isključe posrednici, u ovom slučaju `Authenticate` posrednik kako zahtjev ne bi prolazio kroz autentikaciju jer bi tada test dobio odgovor da zahtjev nije autoriziran. Testovi se u Lumen radnom okviru pokreću naredbom `vendor/bin/phpunit` i nakon pokretanja naredbe svi se testovi uspješno izvršavaju.

```
PHPUnit 9.5.4 by Sebastian Bergmann and contributors.
1 / 1 (100%)
Time: 00:00.092, Memory: 18.00 MB
OK (1 test, 1 assertion)
```

6.5. Isporuka mikroservisa

Mikroservisi su isporučeni koristeći Docker alat za kontejnerizaciju aplikacija. Kao što je već rečeno kontejneri Dockera pružaju virtualno okruženje koje pakira sve aplikacijske ovisnosti i metapodatke stoga na poslužitelju nije potrebno ništa postavljati osim Dockera i *docker-compose.yml* datoteke za isporuku servisa. Svaki servis u svom korijenskom direktoriju sadrži *Dockerfile* datoteku koja omogućuje specificiranje naredbi za instalaciju ovisnosti aplikacije i postavljanje vrijednosti varijabli okoline za bazu podataka i slično. Koristeći *Dockerfile* datoteku kreirane su Docker slike za sve mikroservise koje su zatim postavljene na

Docker Hub registar za dijeljenje i spremanje Docker slika. Na poslužitelju je zatim napravljena *docker-compose.yml* datoteka koja nudi način za konfiguriranje svih servisa i njihovih ovisnosti na jednom mjestu preuzimajući njihove slike iz Docker Hub registra. Finalno, koristeći Docker Compose alat za naredbeni redak (eng. *command-line tool* ili kraće CLI) stvara se jedan ili više kontejnera za svaki servis naredbom `docker-compose up -d` gdje oznaka `-d` pokreće kontejnere u pozadini (na engleskom se to zove *daemon* način rada).

```
version: '3'

services:
  shop:
    image: tkresic/shop
    depends_on:
     - db
    restart: always
    ports:
     - "8000:80"
    environment:
     - SWOOLE_HTTP_HOST=0.0.0.0
     - SWOOLE_HTTP_PORT=80
     - DATABASE_URL=postgres://postgres:postgres@db:5432/shop
     - APP_ENV=local
     - APP_DEBUG=true

  finance:
    image: tkresic/finance
    depends_on:
     - db
    restart: always
    ports:
     - "8001:80"
    environment:
     - SWOOLE_HTTP_HOST=0.0.0.0
     - SWOOLE_HTTP_PORT=80
     - DATABASE_URL=postgres://postgres:postgres@db:5432/finance
     - APP_ENV=local
     - APP_DEBUG=true
```

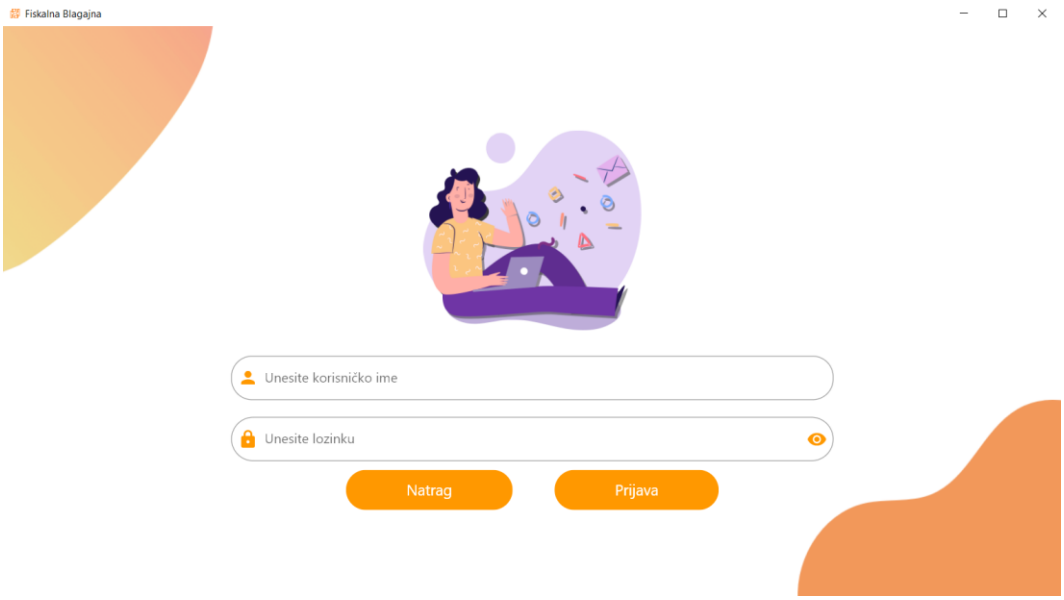
```
  corporate:
    image: tkresic/corporate
    ports:
     - "8080:8080"
    depends_on:
     - db
    environment:
     - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/corporate
     - SPRING_DATASOURCE_USERNAME=corporate
     - SPRING_DATASOURCE_PASSWORD=corporate
     - SPRING_JPA_HIBERNATE_DDL_AUTO=update

  accounts:
    image: tkresic/accounts
    ports:
     - "8081:8081"
    depends_on:
     - db
    environment:
     - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/accounts
     - SPRING_DATASOURCE_USERNAME=accounts
     - SPRING_DATASOURCE_PASSWORD=accounts
     - SPRING_JPA_HIBERNATE_DDL_AUTO=update

  db:
    image: postgres:12.2
    restart: always
    environment:
     POSTGRES_PASSWORD: postgres
     POSTGRES_DB: db
    volumes:
     - data:/var/lib/postgresql/data

volumes:
  data:
```

Nakon pokretanja `docker-compose up -d` naredbe servisi se pokreću, baze podataka se kreiraju i pune testnim podacima i mikroservisi se izlažu na adresi poslužitelja gdje je postavljena *docker-compose.yml* datoteka i definiranim mrežnim vratima. Docker

također omogućava skaliranje servisa i to vrlo jednostavno pa je npr. servise *Finance* i *Shop* moguće skalirati na četiri replike pomoću naredbe `docker-compose up -scale finance=4 shop=4 -d`. Replike koriste istu bazu podataka tako da nije bitno s koje replike *Fiskalna blagajna* dobiva podatke. Nakon pokretanja kontejnera s Dockerom klijentska je aplikacija u mogućnosti pristupiti svim resursima mikroservisa.

6.6. Klijentska aplikacija

Razvoj klijentske aplikacije se odvijao na Linux OS-u, a demonstracija funkcionalnosti će biti prikazana na Windows OS-u. Nakon pokretanja aplikacije i navigiranja s početnog ekrana prikazuje se ekran za prijavu.



Slika 16. Ekran za prijavu

Nakon uspješne prijave koja se odvija kao što prikazuje Slika 14 klijentska aplikacija dobiva žeton koji sadrži kontekst korisnika i njegove ovlasti. Žeton se zatim sprema u lokalno spremište aplikacije i šalje se sa svakim sljedećim zahtjevom kojeg mikroservisi zatim autenticiraju i autoriziraju. S klijentske strane se to odrađuje tako da postoji presretačka klasa koja prilikom slanja svakog zahtjeva izvlači žeton iz spremišta i sprema ga u zaglavlje samog zahtjeva. Za lokalno spremište podataka se koristi ovisnost `shared_preferences`, a za presretačku klasu se koristi ovisnost `http_interceptor`.

```
class ApiInterceptor implements InterceptorContract {
```

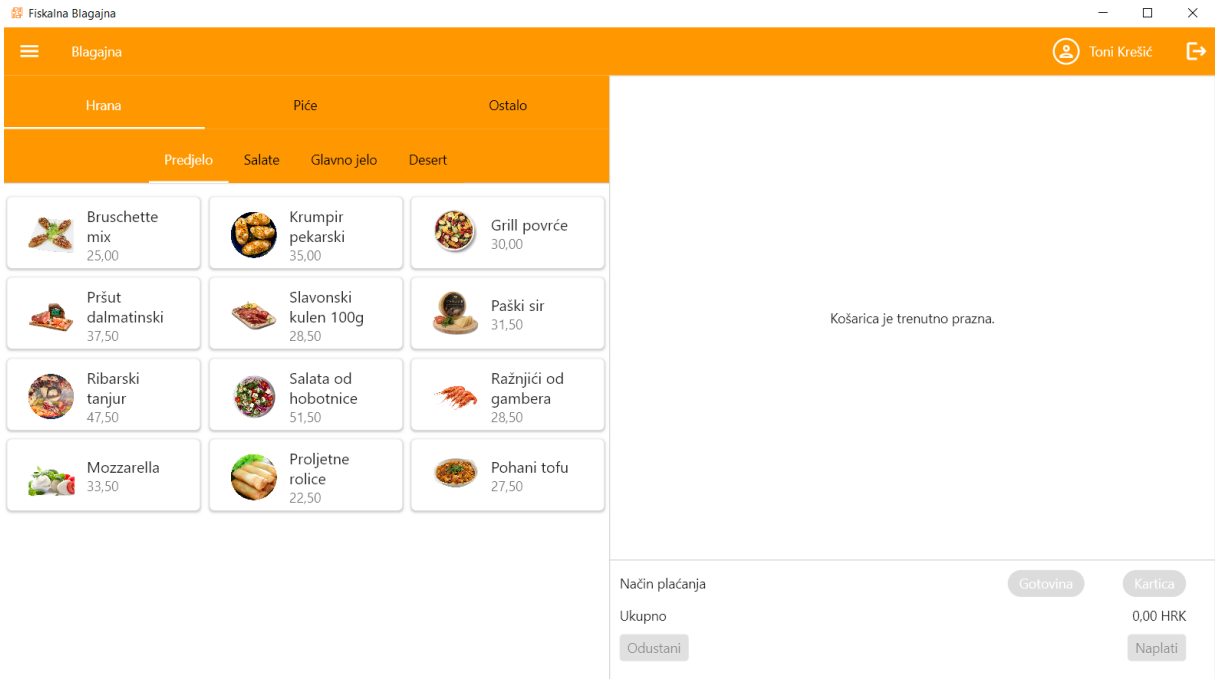
```
  @override
  Future<RequestData> interceptRequest({required RequestData data})
async {

    SharedPreferences prefs = await SharedPreferences.getInstance();
    String? token = prefs.getString("accessToken");

    data.headers["Content-Type"] = "application/json";
    data.headers["Authorization"] = "Bearer " + token!;

    return data;
  }

  @override
  Future<ResponseData> interceptResponse({required ResponseData
data}) async => data;
}
```

Prethodni isječak kôda prikazuje presretačku klasu, a sljedeći isječak kôda prikazuje kako korisničko sučelje koristi presretačku klasu za umetanje žetona u zaglavlje zahtjeva kako bi se dohvatili proizvodi i načini plaćanja.

```
http.Client client = InterceptedClient.build(interceptors: [
  ApiInterceptor(),
]);


Future<Map<dynamic, dynamic>> fetchData() async {
  var products = await
client.get(Uri.parse("${dotenv.env['SHOP_API_URI']}/api/dashboard")
  var paymentMethods = await
client.get(Uri.parse("${dotenv.env['FINANCE_API_URI']}/api/payment-
methods?active=1"));
  return {
    "products" : Product.parseGroupedData(products.body),
    "paymentMethods" :
PaymentMethod.parsePaymentMethods(paymentMethods.body)
  };
}
```

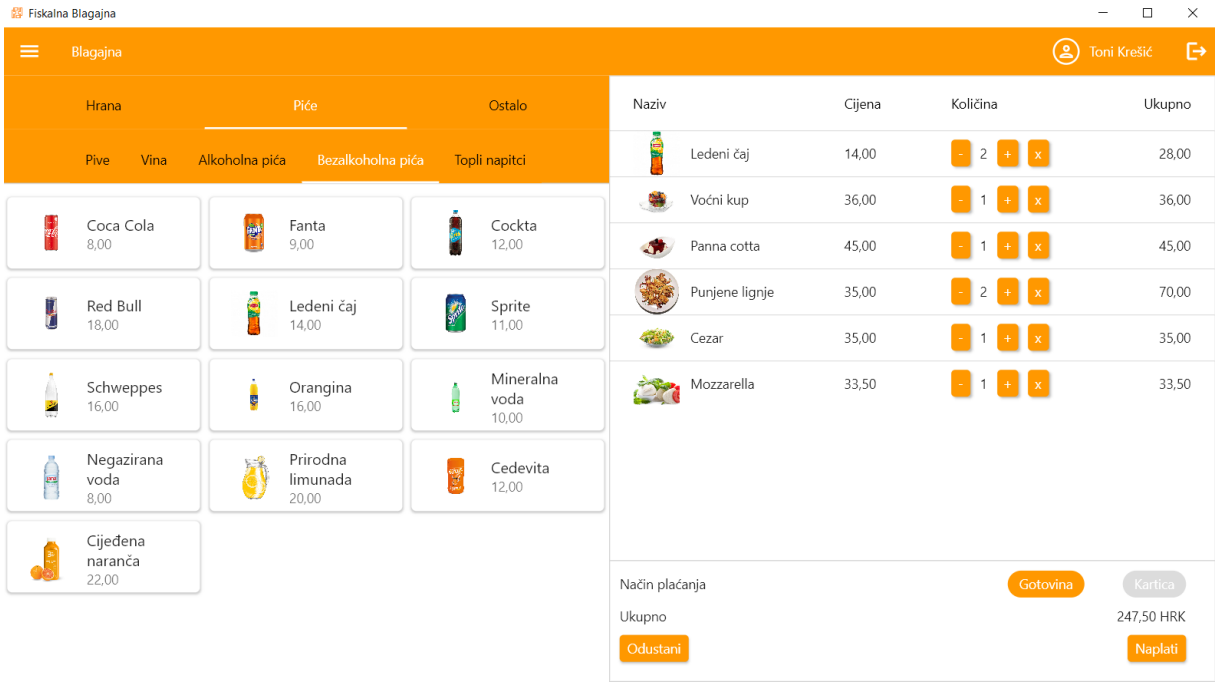
Nakon uspješne prijave korisnik je preusmjeren na korisničko sučelje gdje su prikazani aktivni načini plaćanja i svi proizvodi grupirani po potkategorijama i kategorijama. S obzirom na to da se načini plaćanja nalaze u *Finance* mikroservisu, a proizvodi u *Shop* mikroservisu,

klijentska aplikacija preuzima ulogu API kompozitora i šalje dva zahtjeva kako bi dohvatila sve potrebne resurse za obavljanje rada.



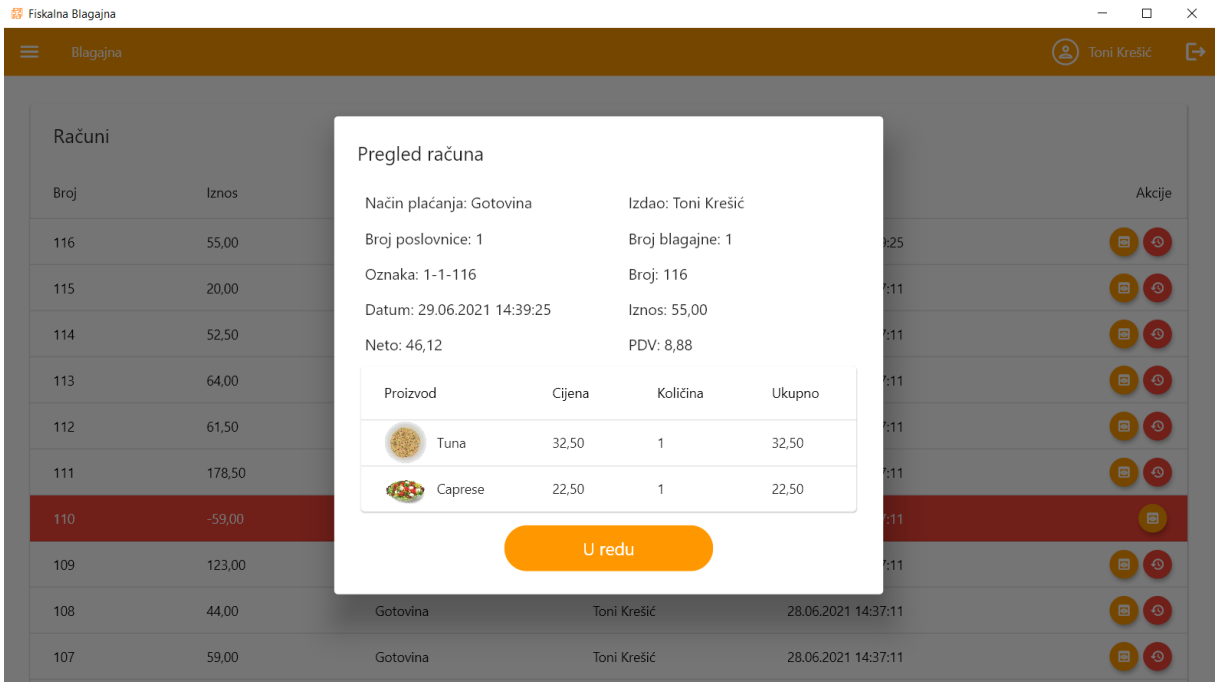
Slika 17. Prikaz korisničkog sučelja

Klikom na kartice proizvoda ažurira se košarica s desne strane i nakon odabira načina plaćanja moguće je kreirati novi račun.



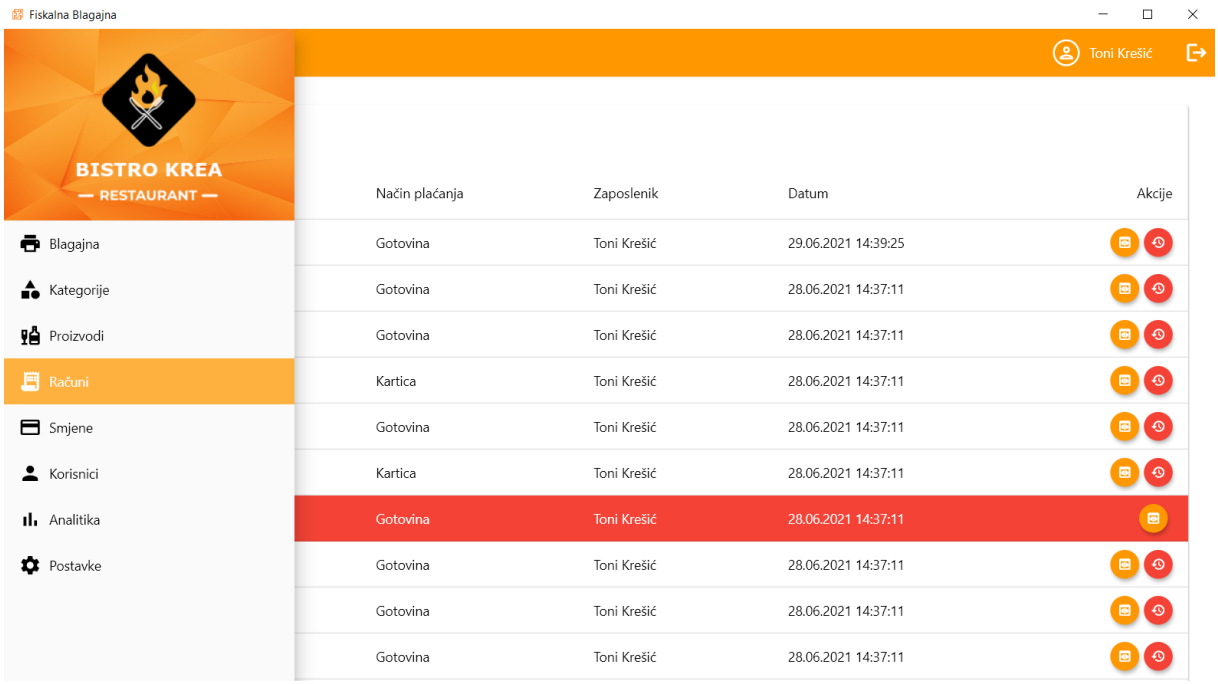
Slika 18. Kreiranje novog računa

Nakon što se račun kreirao moguće je vidjeti sve njegove detalje navigiranjem do računa preko izbornika.



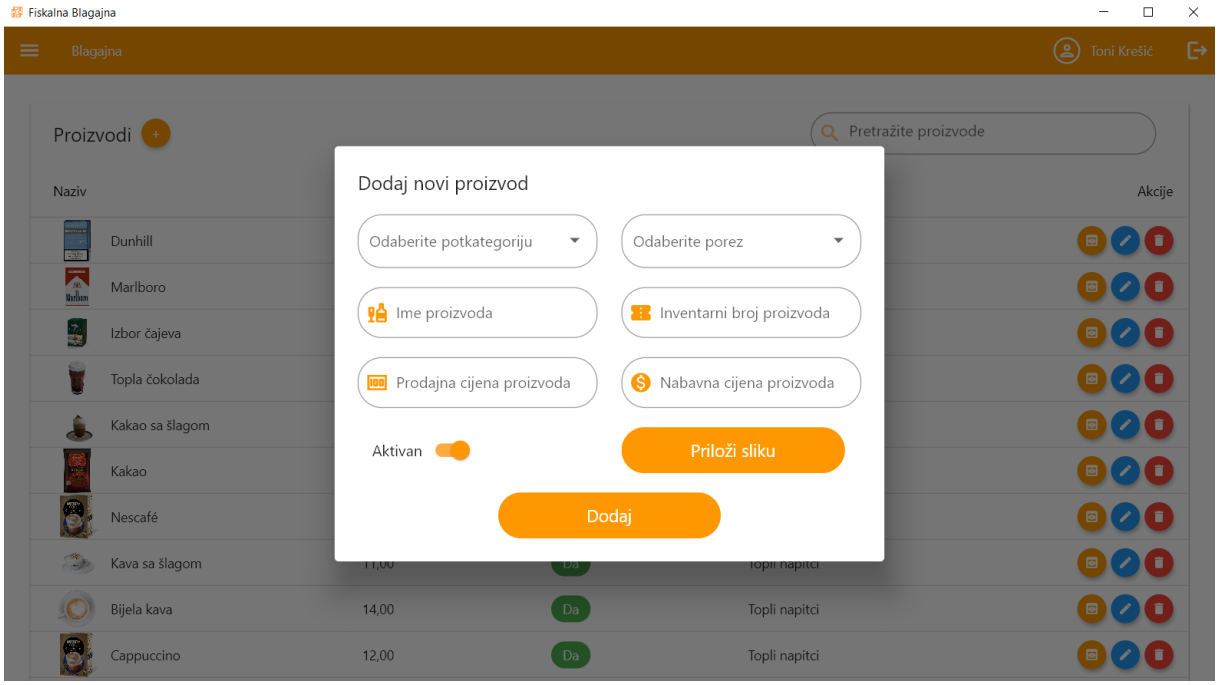
Slika 19. Pregled detalja računa

Svaki račun ima oznaku koja pokazuje u kojoj je poslovnici i na kojoj je blagajni kreiran račun za ovu tvrtku. S obzirom na to da jedna poslovnica može imati više blagajni svaka blagajna ima jedinstveni identifikator čija je vrijednost postavljena u varijabli okoline klijentske aplikacije. Jedinstveni identifikator blagajne se prilikom kreiranja računa šalje zajedno s proizvodima na *Finance* mikroservis koji zatim šalje zahtjev na *Corporate* mikroservis kako bi preuzeo informacije o poslovnici i blagajni jer je svaki račun potrebno povezati s poslovnicom i blagajnom na kojoj je izdan. Računi se mogu i stornirati čime se stvara poveznica između originalnog i storno računa. Storno račun sadrži jednake proizvode i iznos u suprotnoj vrijednosti, a nakon storniranja računa za originalni i storno račun je dopuštena samo akcija pregleda.

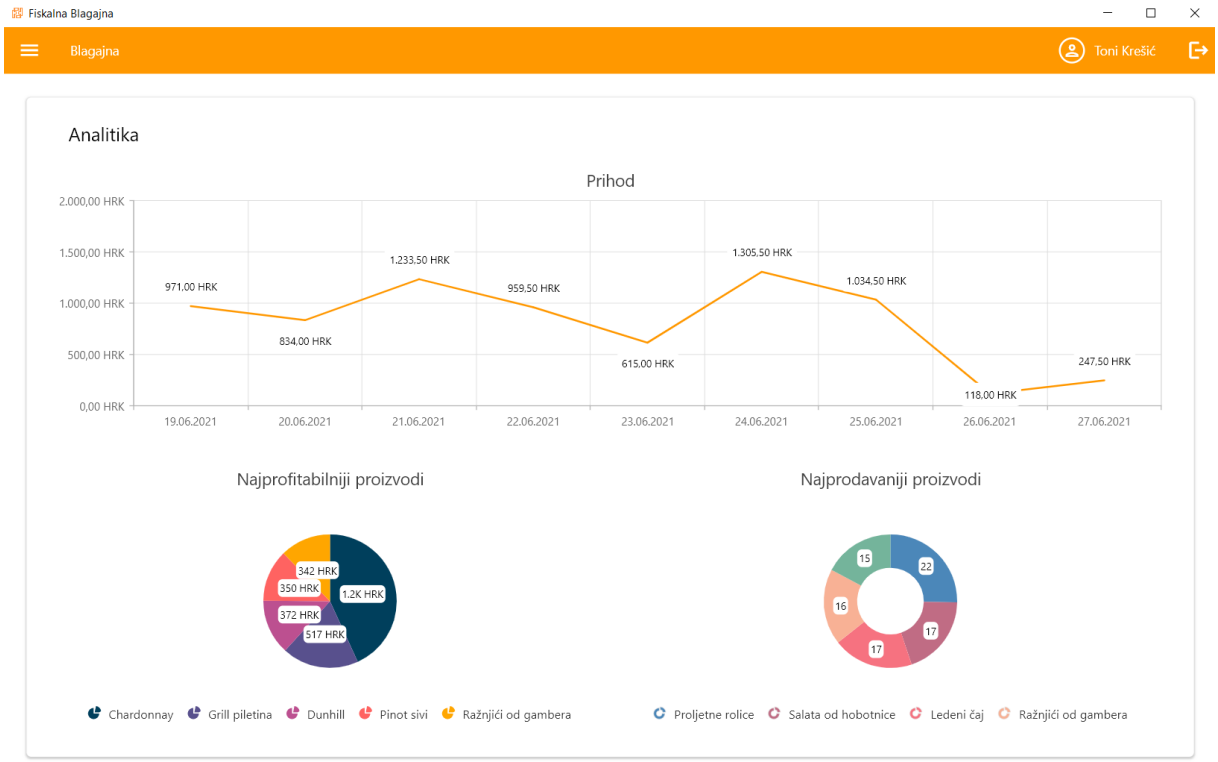


Slika 20. Pregled izbornika

Osim kreiranja računa *Fiskalna blagajna* nudi mogućnost upravljanja kategorijama, potkategorijama, proizvodima, smjenama, korisnicima i postavkama.



Slika 21. Pregled proizvoda i dodavanje novog proizvoda



Slika 22. Pregled analitike

Analitika je dostupna samo administratorima i u slučaju da zaposlenik pokušava pristupiti analitici *Finance* mikroservis će vratiti odgovor da korisnik nema dovoljne ovlasti. Za

analitiku ne postoji poseban servis koji bi izvlačio podatke s drugih servisa već *Finance* mikroservis to sam odrađuje na sljedeći način:

```
public function index()
{
  $dates = $this->billRepository->all([], ['created_at', 'ASC'])
    ->groupBy(function($date) {
      return Carbon::parse($date->created_at)->format('d.m.Y');
    });

  $income = [];
  $quantities = [];
  $sales = [];

  foreach ($dates as $date => $bills) {
    foreach ($bills as $bill) {
      if (array_key_exists($date, $income)) {
        $income[$date] += round(($bill->gross / 100), 2);
      } else {
        $income[$date] = round($bill->gross / 100, 2);
      }
      foreach ($bill->products as $product)
        if (array_key_exists($product['name'], $quantities)) {
          $quantities[$product['name']] += $product['quantity'];
          $sales[$product['name']] += round(($product['quantity'] *
$product['price'] / 100), 2);
        } else {
          $quantities[$product['name']] = $product['quantity'];
          $sales[$product['name']] = round($product['quantity'] *
$product['price'] / 100, 2);
        }
      }
    }
  }

  arsort($quantities);
  $quantities = array_slice($quantities, 0, 5, true);

  arsort($sales);
```

```
  $sales = array_slice($sales, 0, 5, true);

  return response()->json([
    'income' => $income,
    'quantities' => $quantities,
    'sales' => $sales
  ], Response::HTTP_OK);
}
```

Ovakvim pristupom gdje je svaki mikroservis zadužen za pružanje analitičkih podataka svoje domene se smanjuje ukupan broj zahtjeva za jedan.

7. Zaključak

U ovom radu opisana je teorijska pozadina mikroservisne arhitekture i prikazana je praktična primjena mikroservisa u kontekstu stolne aplikacije. Uz mikroservisnu arhitekturu opisane su najčešće metodologije koje se koriste za razvoj softvera i navedene su i neke od ostalih najčešćih vrsta arhitektura softvera. Za same mikroservise opisane su najbitnije značajke i prednosti i nedostaci tih značajki. Nadalje, za svaku karakteristiku mikroservisa navedene su najčešće prakse koje se provode ali tehnologije i alati koji se koriste. Finalno, s obzirom na to da se monolitna arhitektura, glavna alternativa mikroservisima, pretežito koristila do trenutka pisanja ovog rada opisani su i najčešći uzorci dizajna koji se koriste prilikom tranzicije softvera iz monolitne u mikroservisnu arhitekturu.

Distribuirana arhitektura uzima sve veći zamah i to s razlogom. Pogodnosti takvog oblika arhitekture softvera mogu organizaciji donijeti puno koristi, no s druge strane potrebna je veća koordinacija razvojnog tima i više znanja programskih inženjera što organizaciju košta vremena i novca. Razlog tome je taj što je mikroservisna arhitektura općenito kompleksnija i neke njezine najbitnije karakteristike poput autonomnosti i izoliranosti sa sobom donose i cjelokupnu složeniju arhitekturu softvera, otežavaju protok podataka kroz softver i čine isporuku softvera zahtjevnijom. Međutim, kako je i opisano u radu s ovom dodatnom kompleksnošću se može nositi i postoje razne metode i tehnike koje rješavaju probleme koje sa sobom nosi distribuirana arhitektura.

Industrija softvera je bazirana na tehnologiji, a kako se tehnologija rapidno mijenja industrija softvera se mijenja s njom i prilagođava aktualnom trendu. Mikroservisna arhitektura je trenutno aktualna u razvoju softvera, ali je i vrlo specifična jer se bazira na poslovnoj domeni, a kako svaka organizacija ima drukčiju svrhu i ustroj svakoj je organizaciji potrebno napraviti vlastitu procjenu spremnosti i isplativosti korištenja ovakve vrste arhitekture softvera. Ono što je već rečeno i bitno je napomenuti je to da prije nego što se donese odluka za korištenje mikroservisne arhitekture organizacija mora odgovoriti na pitanja zašto bi koristila mikroservisnu arhitekturu i postoji li alternativa koja bi donijela bolje rezultate.

Popis literature

[1] „What is software development?“, tra. 12, 2019. https://www.ibm.com/topics/software-development (pristupljeno tra. 27, 2021).

[2] Roger S. Pressman i Bruce Maxim, *Software Engineering: A Practicioner's Approach*, 8th Edition. McGraw-Hill Education, 2014.

[3] James Martin, *Rapid Application Development*. Macmillan Coll Div, 1991.

[4] Ken Schwaber i Jeff Sutherland, *The Scrum Guide*. https://scrumguides.org, 2020.

[5] Kent Beck i Cynthia Andres, *Extreme Programming Explained: Embrace Change*, 2nd Edition. Addison-Wesley, 2004.

[6] Mary Poppendleck i Tom Poppendleck, *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, 2003.

[7] Eric Brechner, *Agile Project Management with Kanban (Developer Best Practices)*, 1st Edition. Microsoft Press, 2015.

[8] Len Bass, Paul Clements, i Rick Kazman, *Software Architecture In Practice*, 3rd Edition. Addison-Wesley Professional, 2012.

[9] Mark Richards, *Software Architecture Patterns*. O'Reilly Media, Inc., 2015.

[10] Chris Richardson, *Microservices Patterns: With examples in Java*, 1st Edition. Manning Publications, 2018.

[11] Pethuru Raj, Anupama Raman, i Harihara Subramanian, *Architectural Patterns: Uncover essential patterns in the most indispensable realm of enterprise architecture*. Packt Publishing, 2017.

[12] Joseph Ingeno, *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*, 1st Edition. Packt Publishing, 2018.

[13] Mark Richards, *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Inc., 2016.

[14] Melvin Conway, „How do committees invent?“, *Datamation*, tra. 1968.

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

[16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, i Mike Amundsen, *Microservice Architecture*. O'Reilly Media, Inc., 2016.

[17] Eberhard Wolff, *Microservices: Flexible Software Architecture*, 1st Edition. Addison-Wesley Professional, 2016.

[18] Chris Richardson i Floyd Smith, *Microservices: From Design to Deployment*. NGINX inc., 2016.

[19] Susan J. Fowler, *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization*, 1st Edition. O'Reilly Media, Inc., 2016.

[20] Martin Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, Inc., 2017.

[21] Mike Cohn, *Succeeding with Agile: Software Development Using Scrum*, 1st Edition. Addison-Wesley Professional, 2009.

[22] Parminder Kocher, *Microservices and Containers*, 1st Edition. Addison-Wesley Professional, 2018.

[23] Prabath Siriwardena i Nuwan Dias, *Microservices Security in Action*, 1st Edition. Manning Publications, 2020.

[24] Sam Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*, 1st Edition. O'Reilly Media, Inc., 2019.

[25] Erich Gamma, Richard Helm, Ralph Johnson, i John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st Edition. Addison-Wesley Professional, 1994.

Popis slika