Od monolita do mikroservisa - Sveobuhvatni pristup refaktorisanju u softverskom inženjerstvu

Milica Bakić, Bojana Dimić Surla

***Sadržaj* — Rad prikazuje proces migracije sa monolitnih sistema na mikroservisnu arhitekturu u kontekstu softverskog inženjerstva. Fokus je na teorijskoj analizi arhitektura i praktičnih aspekata migracije. Predstavljene su strategije i preporuke koje se koriste tokom procesa, kao i ključni faktori koji utiču na ishod migracije. U radu je istaknuta važnost procene i planiranja prelaska, kao i situacije u kojima se ne preporučuje vršiti transformaciju ovog tipa.**

***Ključne reči* — softverske arhitekture, mikroservisi, monolit, skalabilnost, održavanje, testiranje, implementacija, prakse migracije**

I. Uvod

SOFTVERSKI sistemi zastupljeni su u različitim sferama kako poslovnog tako i svakodnevnog života. Zbog svoje dinamične prirode podložni su promenama kroz vreme zbog mnogobrojnih faktora. Zahtevi klijenata, optimizacija resursa, nove integracije i tehnološki napreci, sigurnost, usklađivanje sa zakonodavnim normama jedne zemlje, samo su neki od razloga potrebe da se softver promeni. Uzeći u obzir tu osobinu, jedan od bitnih kvaliteta softvera jeste njegova arhitektura. Arhitektura kao takva značajno će diktirati koliko su promene teško izvodljive ili lakše za usvojiti, koliko iziskuju resursa i novca.

Arhitektura se može posmatrati kao rešenje koje definiše strukturu, komponente, komunikaciju i karakteristike softverskog i/ili hardverskog sistema [1]. Ona pruža okvir za razvoj i implementaciju sistema uključujući i

M. Bakić. Autor, Računarski fakultet, Beograd, Srbija (email: mbakic@raf.rs).
B. Dimić Surla. Autor, Računarski fakultet, Beograd, Srbija; (email: bdimicsurla@raf.rs)

odluke o samom dizajnu. Značaj arhitekture je u tome što postavlja temelj za budući razvoj softvera i često je početna faza. Performanse, pouzdanost, skalabilnost, održavanje, bezbednost, komunikacija, podela posla i organizacija tima, usvajanje novih tehnologija predstavljaju osnovne posledične aspekte usled odabira arhitekture.

Monolitna ahitektura predstavlja tradicionalni pristup u dizajnu softverskih sistema gde se čitava aplikacija razvija kao jedinstvena i celovita jedinica. Drugim rečima, svi njeni delovi, komponente i funkcionalnosti, integrisani su u istom programskom kodu [2].

Mikroservisi se mogu posmatrati kao dekompozicija sistema s obzirom da je softver podeljen na više manjih delova koji se nazivaju servisi. Svaka komponenta je zadužena za jedan deo poslovnog domena sistema i komunicira sa drugim komponentama preko definisanih mehanizama. Na taj način realizovana je poslovna funkcija softvera [3].

Pojava mikroservisne arhitekture ima dobar odziv poslednjih 10 do 15 godina. Njena popularnost u mnogim kompanijama i rešenjima značajno je porasla u tom periodu, ali i dalje je dosta aktuelna i popularna. Donosi dosta prednosti poput skalabilnosti, lakšeg raspoređivanja (eng. *deployment*) i generalno bolje iskorišćenosti resursa, nezavisnost servisa, fleksibilnosti i mogućnosti agilnog razvoja softvera koji predstavlja jedan od popularnijih pristupa danas [4].

Usled pojave mikroservisa, mnogi tradicionalni sistemi izvršili su refaktorisanje i migraciju upravo na ovu arhitekturu. Jedna od najzastupljenijih arhitektura kada je reč o tradicionalnim sistemima jeste arhitektura monolita. Međutim, premeštenje na mikroservise sa sobom nosi niz izazova, a često i nije najbolje rešenje. Ovaj tekst za cilj ima da predstavi i ponudi potencijalna rešenja prelaska sa monolita na mikroservise sa teorijskog aspekta, ali i sa strane prakse. Pored detaljnog teorijskog pregleda, kao i prustupa za migraciju ovakvog tipa biće priložen i praktični deo rada koji će dokazati ove teorijske analize.

II. MOTIVACIJA ZA UVOĐENJE MIKROSERVISNE ARHITEKTURE

Monolitna arhitektura zbog svoje prirode često se smatra najjednostavnijom za implementiranje. Nije potrebno voditi računa o orkestraciji više komponenti s obzirom da se sve što je neophodno nalazi na jednom mestu [5]. Monolit se može proširiti komponentom određenog zaduženja ili koja podržava drugu tehnogoliju. Kod mikroservisa od samog početka, cilj je da se sistem struktuira tako da se sastoji od mikroservisa pri čemu svaki mikroservis ima svoju bazu i interfejs.

Mikroservisi nasuprot pomenutoj arhitekturi imaju dosta prednosti, ali i

dodatnog posla, kao što su komunikacija između mikroservisa, dekompozicija baze i ostalo. Preporuka je ne praviti ih ukoliko to nije neophodno.

U radu [6] zaključeno je da je za kompanije koje previše narastu i čije su baze koda prevelike mikroservisi predstavljaju dobro rešenje za probleme kompleksnosti. Rad [7] tvrdi da su najveći motivi za refaktorisanje monolitnog sistema održivost i skalabilnost. Dodaje još da iako je očekivano vreme povraćaja uloženih sredstava i investicija u migraciju dug period, smanjeni naporni na održavanju to nadoknađuju. Sa druge strane, rad [8] naglašava da manja preduzeća možda neće imati iste koristi kao velike globalne kompanije prilikom prelaska na mikroservisa. Rad ističe nekoliko ključnih tačaka:

- očekivanja i realnost: koristi koje su dobile velike kompanije možda neće doživeti i manje,
- ograničena istraživanja: s obzirom da ne postoji veliki broj istraživanja za prelazak na mikroservisa kod sistema koji nemaju više hiljada korisnika, pa su samim tim izvori ograničeni, odluke se mogu doneti brzopleto,
- razmatranje skaliranja: za sisteme koji se mogu skalirati vertikalno koristi migriranja na mikroservise možda neće biti očigledni, a nekada je i dovoljno, pa prelazak na „skuplju" arhitekturu nije najbolje rešenje.

III. NAJBOLJE PRAKSE U MIGRACIJI MONOLITA NA MIKROSERVISA

Migracija monolita na mikroservise je izazovan proces i zahteva dosta planiranja i ispitivanja. Kako su mikroservisi postali sve popularniji, mnoge kompanije svoju arhitekturu izmeštaju na mikroservise što i nije uvek najbolja odluka. Iako se monolitni sistemi smatraju tradicionalnim, oni nisu loši. Odluka o prelasku na mikroservise može biti preuranjena i samo zakomplikovati sistem koji funkcioniše bez problema. Prvi korak jeste razumeti mikroservise kao arhitekturu, jer nose više izazova nego što je to možda očigledno na prvi pogled. Ukoliko je aplikacija jednostavna i manjeg obima, bez jasnih poslovnih domena, migracija ovog tipa može biti loša u praksi. Implementacija mikroservisa treba biti vođena stvarnim razlozima, a ne trenutnim trendovima u industriji [9].

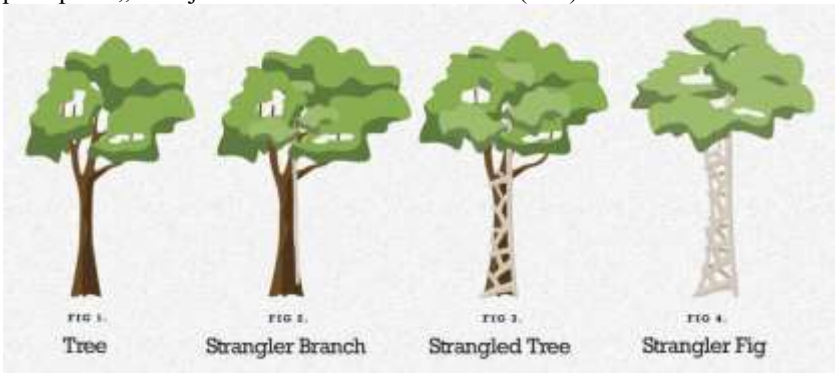
Koraci koji se izdvajaju u procesu migracije uključuju, prethodno spomenutu, detaljnu analizu monolita, razumevanje arhitekture, komponenti, podataka i svih resursa od kojih zavisi sistem. Zatim definisanje jasnih poslovnih domena u okviru sistema može pomoći u lakšoj identifikaciji mikroservisa. Uspostavljanje komunikacije, izbor tehnologija, testiranje i upravljanje podacima spadaju u osnovne korake koje treba razmotriti [10].

Kod koji poštuje clean code i SOLID principe lakše će se razdvojiti na

mikroservise od koda kod kojeg nije vođeno računa prilikom pisanja [11]. Ovi principi promovišu dobre prakse i pomažu u pisanju održivog i fleksibilnog koda. Jasno i dobro strukturiran kod lakše je održavati, samim tim postaje fleksibilan za buduće izmene. Poštovanjem SOLID principa vodi se računa o odgovornosti komponenti što omogućava lakšu migraciju na mikroservise.

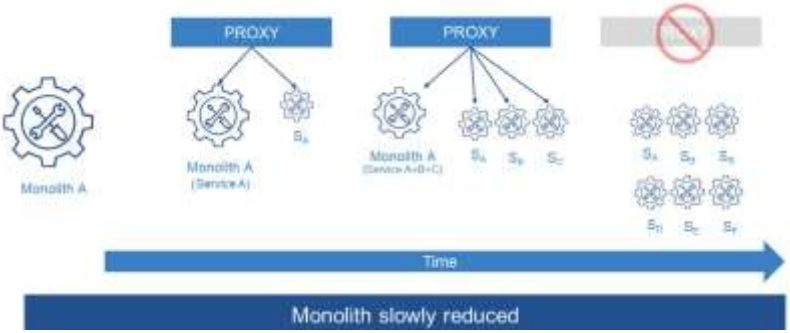
Iako migracija monolitnog sistema na mikroservise može varirati u zavisnosti od specifičnosti samog sistema, postoje dobro poznati šabloni i prakse koji se često koriste i olakšavaju proces dekompozicije.

Šablonsko gušenje (eng. *Strangler Fig Pattern)* je tehnika migracije koja savetuje postepenu zamenu postojeće monolitne aplikacije novim mikroservisima kako bi se izbegle grube i velike promene u prepravljanju čitavog sistema [12]. Iz tog razloga u nazivu stoji reč gušenje, jer se monolit postepeno „obavija" mikroservisnim šablonima (Sl 1).



Sl. 1 Ilustracija šablonskog gušenja [1]

Na početku je neophodno identifikovati funkcionalnosti koje će se preneti u nove mikroservise. Zatim se razvijaju novi mikroservisi koji će obavljati nove funkcionalnosti identifikovane u prethodnom koraku. Novi mikroservisi bivaju postepeno uvedeni u sistem i rade paralelno sa monolitom. Postojeći monolit se idalje koristi za postojeće funkcionalnosti. Uglavnom se priključivanje i sinhronizacija sa novim mikroservisima radi u fazama. Kako mikroservisi u sve većoj meri preuzimaju odgovornosti od monolita, monolit gubi smisao. Migracija je uspešno obavljena kada se sve funkcionalnosti prenesu na mikroservise. Na kraju, monolit se eliminiše. Na Sl. 2 prikazana je ilustracija procesa migracije korišćenjem šablonskog gušenja.



Sl. 2 Proces migracije monolita na mikroservise korišćenjem šablonskog gušenja [13]

Šablon omogućava postepenu migraciju čime se smanjuje rizik i dobija na vremenu da se nove tehnologije utemelje u novom sistemu. Postojeće funkcionalnosti ostaju netaknute dok se novi mikroservisi uvode, pa je strah od prekida sistema otklonjen. Uvođenjem novih mikroservisa, delovi sistema se mogu nezavisno testirati, pa se greške mogu otkloniti na vreme. Ipak, postoje i poteškoće do kojih može doći primenom ovog šablona. Koordinacija između monolita i novih mikroservisa može biti kompleksna, posebno ukoliko postoje zavisnosti u različitim delovima sistema [13]. Razvoj i održavanje mikroservisa pored monolita može biti značajno skuplje od održavanje jednog tipa sistema.

Strangler Fig šablon pruža postepenu migraciju sistema bez prekida u radu svodeći rizik od problema na minimum. Ono što je važno u ovom pristupu jeste pažljivo upravljanje i usaglašavanje monolita i mikroservisa. Preporuka je koristiti ga kod složenih monolitnih sistema gde nije moguće izvršiti prelazak u jednom koraku. Dobra je praksa i u situacijama kada je potrebno očuvati postojeće funkcionalnosti, uvesti nove funkcionalnosti u paraleli sa migracijom, kao i kada je potrebno uvesti nove tehnologije u timove

Dekompozicija po poslovnoj sposobnosti je pristup kod kojeg se razlaganje na mikroservise izvodi na osnovu funkcionalnosti. Drugim rečima, delovi monolita se razdvajaju prema poslovnim aspektima. Prvo potrebno je analizirati i identifikovati ključne sposobnosti [14]. Za svaku prepoznatu grupu funkcionalnosti razvija se mikroservis pri čemu je svaki fokusiran na funkcionalnosti koje su u opsegu jedne sposobnosti. Ovim se postiže da svaki mikroservis ima jasno definisanu odgovornost. Mikroservisi se povezuju različitim stilovima komunikacije čime se omogućava integrisani sistem. Ovakva refaktorizacija se daleko lakše postiže ukoliko monolit poštuje clean code i SOLID principe.

Odgovornosti unutar mikroservisa su jasno definisane i podeljene što olakšava održavanje i proširivanje sistema. Ovaj pristup dekompozicije omogućava brzu prilagodljivost sistema na promene, jer se svaki mikroservis

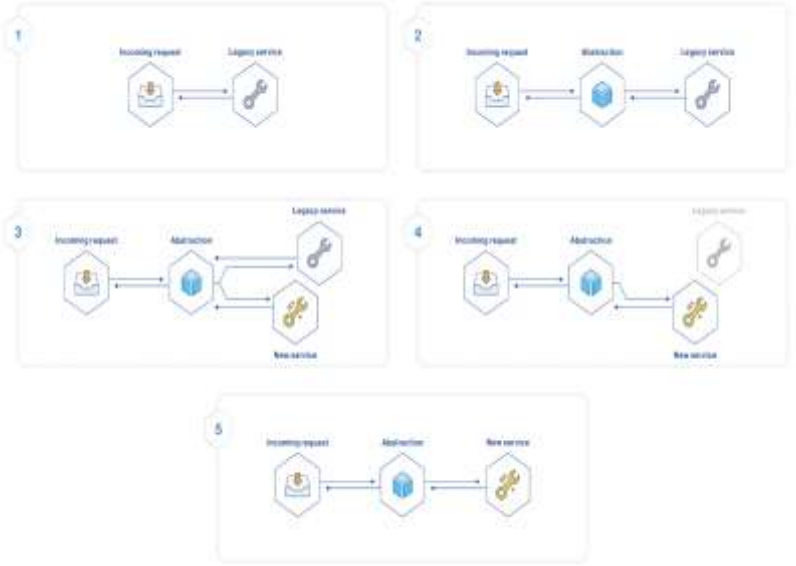
i iterativno i nezavisno razvija. Takođe, skaliranje je nezavisno, pa je upravljanje resursima efikasnije. Izazovi nastupaju u komunikaciji s obzirom da veliki broj mikroservisa utiče na kompleksnost. Potrebna je dodatna administracija i upravljanje i praćenje verzija.

Dekompozicija na ovakav način omogućava agilnost u razvoju i prilagodljivost sistema na promene. Najbolje ga je koristiti kada je potrebno prilagoditi arhitekturu poslovnim potrebama. Na primer, promene u poslovnim zahtevima su česte, potrebno je podržati više različitih tehnologija, omogućiti nezavisnost timova i paralelno im davati zadatke. Naravno, uvek je neophodno pažljivo razmisliti i primetiti trenutne potrebe kompanije.

Sličan šablon dekompozicije jeste dekompozicija na osnovu transakcija, gde se umesto poslovnih sposobnosti izdvajaju transakcije koje obrazuju mikroservise. I jedna i druga tehnika imaju za cilj da olakšaju upravljanje i održavanje sistema uzimajući u obzir potrebe kompanije. Izbor tehnika treba da se bazira na strategiji i fokusu poslovanja. Često se ova dva obrasca mogu kombinovati ili prilagoditi sistemu koji je potrebno migrirati.

Grananje pomoću apstrakcije (eng. *Branch by abstraction*), predstavlja obrazac gde je cilj omogućiti postepenu migraciju sa monolitne na mikroservisnu arhitekturu oslanjajući se na apstrakcije. Najpre se uočavaju potrebne promene u sistemu koje mogu biti vezane za arhitekturu, tehnologije ili funkcionalnosti. Umesto direktnog menjanja koda, uvodi se apstrakcija (interfejs) koji predstavlja buduće izmene. Interfejs služi kao apstrakcija iznad postojećeg koda, a implementacija se dodaje postepeno tokom vremena. Sledeći korak naziva se kreiranje ograničavajuće grane (eng. *branch*). Ovaj korak podrazumeva novu granu u nekom od sistema za kontroliranje verzija, najčešće je to git. Nova grana uključuje interfejs i sve potrebne promene, ali ne i implementacije novih funkcionalnosti, otuda naziv ograničavajuća grana. Nakon toga, različiti timovi mogu dodavati implementaciju uvedenih promena. To se postiže postepenim dodavanjem na glavnu granu sistema dok se idalje održava podrška za postojeći sistem. Kada su svi timovi završili sa implementacijom promena, apstrakcija se može ukloniti (Sl 3). Finalno, novi kod postaje deo glavne grane sistema.

Glavne prednosti u pristupu grananja pomoću apstrakcije jesu postepena migracija, smanjenje rizika od grešaka u samoj migraciji, kontinuirano isporučivanje novih funkcionalnosti dok se sistem transformiše i lakše upravljanje izmenama [15]. Međutim, uvođenjem dodatnih apstrakcija, povećava se složenost koda dok se ne ukloni, sistem postaje kompleksan za održavanje. Ovakav pristup u migraciji zahteva izuzetno pažljivo i detaljno planiranje, kao i odličnu saradnju između timova.



Sl. 3 Proces grananja pomoću apstrakcije [15]

III. PRAKTIČNI DEO RADA

U praktičnom delu rada implementiran je sistem za rezervaciju. Booking System ima za cilj da olakša rezervisanje smeštaja širom sveta. Sam sistem se može grubo podeliti na dve celine s obzirom da je potrebno podržati postavljanje smeštaja, kao i pretraživanje i rezervaciju istih. Iz svega navedenog, na jednom mestu nalaze se svi dostupni oglasi za određenu destinaciju, kao i sve potrebne informacije o samom smeštaju čime se postiže značajna ušteda vremena. Aplikacija dizajnirana kao takva može se posmatrati posrednikom između korisnika koji nudi i koji traži smeštaj.

A. Specifikacija sistema

Sistem je namenjen korisnicima koji planiraju svoje putovanje. Pored osnovnog tipa korisnika, postoje i provider i admin. Provider ima drugačiju ulogu u sistemu u odnosu na korisnika. On nudi aranžmane korisnicima, korisnici bukiraju smeštaj. Kada korisnik rezerviše smeštaj, može komunicirati sa provajderom smeštaja koji je rezervisao. Admin se može posmatrati kao administrator sistema koji ima najširi skup operacija i permisija u sistemu. Njegova uloga je da upravlja sistemom.

Pre putovanja, korisnici rezervišu smeštaje u kojima će odsedati. Sama rezervacija se sastoji od pretrage različitih tipova smeštaja na destinaciji za koju su putnici zainteresovani. Različiti tipovi smeštaja obuhvataju: hotele, hostele, motele, apartmane, rizortove, kolibe, seoske kuće, kampove i sl. Korisnik na početku unosi neophodne podatke kako bi mu se prikazala ponuda dostupnih aranžama. Pod neophodnim podacima misli se na:

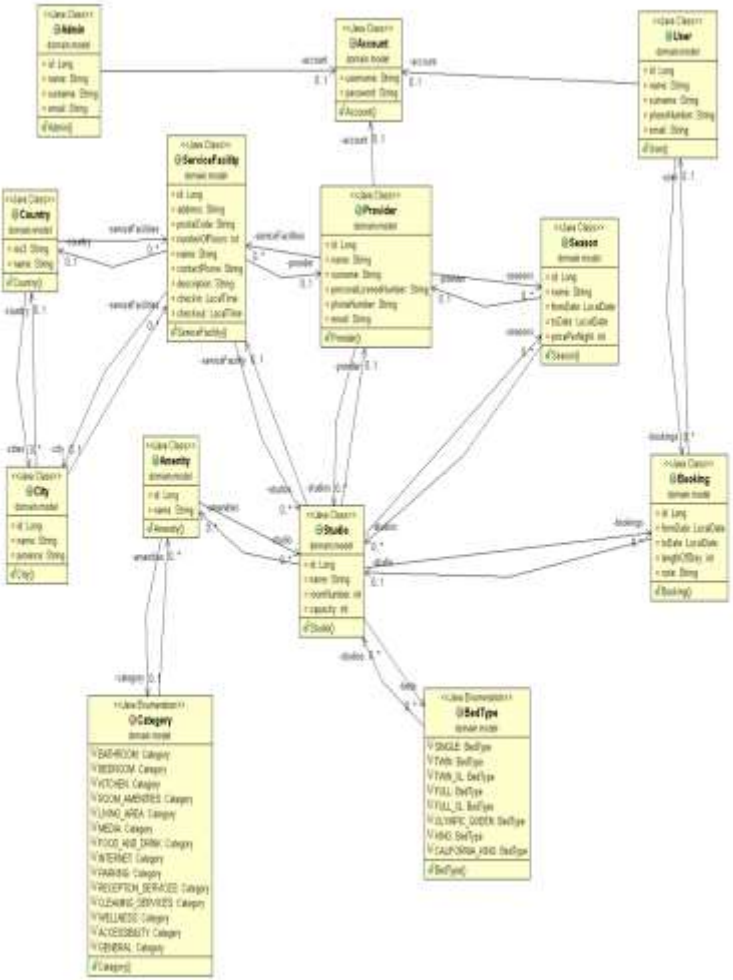
- destinaciju tj. mesto za koje je potrebno izlistati ponude,
- početni datum i krajnji datum tj. period za koji je potrebno izdati smeštaj,
- broj osoba za koji je potreban smeštaj.

Nakon toga, korisniku se prikazuju rezultati na osnovu unetih podataka. Rezultati se mogu sortirati i filtrirati po različitim kriterijumima. Nakon bukiranja smeštaja, korisnik ima uvid u rezervisane smeštaje u narednom periodu, kao i u svoja prethodna putovanja. Provajder u okviru booking sistema ima ulogu u postavljanju oglasa, uređivanjem smeštaja, kao i pregledom rezervacija za svoje smeštaje od strane korisnika. Svi tipovi korisnika mogu da uređuju svoj profil.

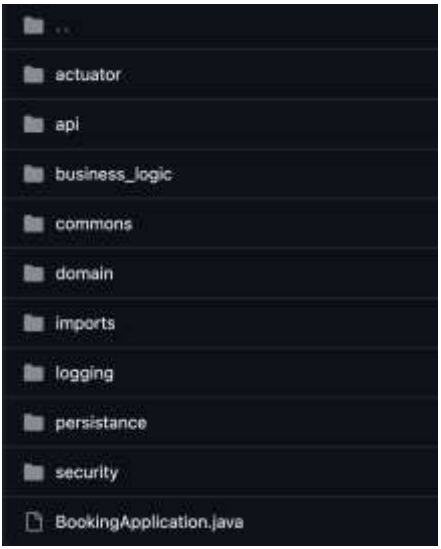
B. Implementacija aplikacije kao monolitnog sistema

Nakon objedinjene specifikacije kreiran je klasni dijagram (Sl. 4). Klasni dijagram odgovara i modelu podataka Booking sistema. Booking sistem implementiran je kao Spring Boot java aplikacija.

Kao što je prethodno istaknuto, arhitektura je u potpunosti monolitna sa osloncem na slojeviti pristup. Slojevi koji se mogu izdvojiti jesu api, biznis logika, sloj perzistencije i sloj podataka (Sl. 5). Api sloj sadrži komponente koje interaguju sa klijentom. Preciznije, sadrži kontolere u kojima je smeštena logika za obradu zahteva i vraćanje odgovora klijentu. Kontroleri komuniciraju sa komponentima biznis logike, servisima. U samim servisima smeštena je logika koja zadovoljava funkcionalne zahteve sistema, otuda i naziv sloj biznis logike. Treba napomenuti da je za implementaciju servisa korišćena apsktracija, pa komponente sloja biznis logike nisu vezane ni za jedan frejmvork. Servisi ne komuniciraju direktno sa repozitorijumom već su uvedeni medijatori između njih, menadžeri. Zadatak komponenti sloja perzistencije osim da budu posrednici između servisa i repozitorijuma jeste da, ukoliko je to potrebno, rade premapiranja modela između slojeva ukoliko postoje odstupanja. Na kraju, repozitorijumi imaju pristup podacima i rade manipulaciju nad podacima.



Sl. 4 Klasni dijagram sistema



Sl. 5 Pregled paketa aplikacija

Monolitnim pristupom, postignuto je da svaki sloj ima opseg nadležnosti. Tokom implementacije značajan fokus stavljen je na slabu povezanost, pa promene u okviru jednog sloja nemaju uticaja na druge slojeve. Lošije performanse ogledaju se u tome što je potrebno premapiranje podataka između slojeva. Takođe, često se logika ponavlja upravo zbog premapiranja podataka između slojeva. Kompleksnost aplikacije utiče na dodavanje međuslojeva, pa se u budućnosti usled dodavanja novih funkcionalnosti može očekivati povećanje broja slojeva.

C. Dekompozicija monolitnog sistema i prevođenje na mikroservise

Kao platforma za rezervaciju i oglašavanje smeštaja, prekidi rada aplikacije mogu izazvati ozbiljne probleme. Ovo takođe ima negativan uticaj na reputaciju same aplikacije. Trebalo bi omogućiti izolaciju i lakše upravljanje prekidima kako ovo ne bi imalo uticaja na druge funkcionalnosti, kao i na korisnikov negativan doživljaj tokom korišćenja bukinga. Tokom praznika ili popularnih sezonskih intervala, aplikacija može biti preopterećena zahtevima korisnika. U ovoj situaciji, potrebno je skalirati relevantne delove aplikacije koji su pod većim opterećenjem. Cilj buking sistema kao što je ovaj jeste da pronađe put na svetsko tržište, pa se može očekivati da ima podršku za različite lokalne zakone, valute i jezike.

Kao odgovor na sve pomenute izazove nameću se mikroservisi.

Prebacivanjem na mikroservise, sistem može postati visoko funkcionalan i optimizovan, jer bi se na taj način poboljšale performanse i pružila fleksibilnost. U nastavku poglavlja opisan je postupak dekompozicije opisanog monolitnog sistema i prelazak na mikroservisnu arhitekturu.

Najpre je izvršena dekompozicija koda, a zatim baze podataka. Kao šablon izabrana je dekompozicija po poslovnoj sposobnosti koja nalaže da je prvo potrebno uočiti sve funkcionalnosti koje spadaju u određen opseg poslovanja. U trenutnoj implementaciji mogu se identifikovati sledeće poslovne sposobnosti: upravljanje korisnicima, upravljanje smeštajima i rezevacija smeštaja. Svaka sposobnost se sada izdvaja kao poseban mikroservis koji odgovara specifičnom domenu poslovanja.

Izdvojeni mikroservisi imaju svoj opseg zaduženja. Mikroservis za upravljanje korisnicima, nazvan IAM skraćeno od eng. *Identity and Access Management,* bavi se registracijom i prijavom korisnika, rolama i dozvolama korisnika. Mikroservis za upravljanje smeštajem ima zaduženja oko dodavanja, brisanja i ažuriranja smeštaja, pogodnosti u okviru smeštaja, sezone i svih ostalih relevantnih entiteta. Nadležnost mikroservisa za rezervacije je rezervisanje smeštaja od strane korisnika u određenom terminu.

Međutim, kako je monolitni sistem već postojao i bio funkcionalan, ovakav pristup pri dekompoziciji značio bi rasturanje postojećeg sistema odjednom. Ovakva migracija ne bi mogla da se izvrši brzo i mogla bi se okarakterisati kao „sve ili ništa". Takođe, mikroservis za upravljanje smeštajima i rezervaciju nije tako lako izdvojiti u jednom pokušaju s obzirom da postoje zavisnosti između entiteta. Zato je daljim ispitivanjem doneta odluka da se za migraciju ipak upotrebi drugi šablon, tačnije, eng. *Strangler Fig Pattern.* Na ovaj način, osigurano je da neće biti naglih i grubih promena koje se mogu završiti loše po sistem. Promenom šablona rizici za neuspeh smanjeni su na minimum i data je fleksibilnost u procesu migracije s obzirom da se postepenim rasturanjem monolita tempo procesa može prilagođavati trenutnoj
situaciji. Kako su poslovne sposobnosti već identifikovane, nije potrebno to ponovo učiniti.

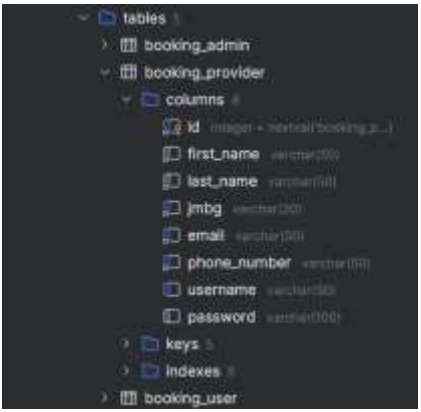
Prvi korak sada jeste izdvojiti najmanje zavistan servis u poseban mikroservis. Potrebno je uočiti celinu koda koja ima jasno definisane granice u pogledu funkcionalnosti i malo ili nimalo direktne povezanosti sa ostalim funkcionalnostima sistema. U identifikaciji najmanje zavisnog servisa treba sagledati još i koja to celina koda uključuje rad sa nezavisnim ili manje zavisnim podacima kako bi manipulacija nad podacima bila nezavisna od ostalih servisa. Kako funkcionalnosti poput prijave i registracije korisnika, ažuriranje šifre i profila nemaju posledice na ostale poslovne sposobnosti koje su vezane za aktivnosti korisnika kada je reč o smeštaju i rezervaciji, celina koda za upravljanje korisnicima može se izdvojiti kao prvi mikroservis. Kako za funkcionisanje ostatka sistema učestvuje ID korisnika, tako izdvojeni servis manipulisao bi podacima koji ne bi imali uticaja na funkcionalnosti ostatka

koda. Izdvajanje servisa se može izvesti relativno lako i sa manjim prepravkama.

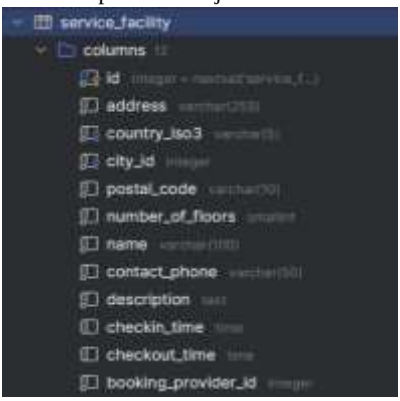
Sl. 6. Pregled paketa novog mikroservisa za autentikaciju i autorizaciju (IAM)

Kako su uočene funkcionalnosti koje će se prve prevesti u mikroservis, kreira se novi servis i kod se prebacuje iz monolita u novokreirani servis. U monolitu više ne postoje delovi koda koji uključuju IAM funkcije. Novi mikroservis poštuje iste prakse kao implementirani monolit, pa se paketi organizuju po slojevima (Sl. 6) U tom trenutku postoje dva servisa, jedan za upravljanje korisnicima i drugi koji sadrži „sve ostalo" od monolita tj. uključuje rukovanje svim ostalim entitetima sa slike 4.1. Ono što je potrebno još izvršiti u ovom koraku jeste dekompozicija baze podataka. Iz postojeće baze podataka treba izbaciti tabele koje su sada opseg IAM mikroservisa. Entiteti BookingUser, BookingProvider i BookingAdmin vezani su za IAM s obzirom da je srž poslovne sposobnosti ovog servisa upravljanje korisnicima u sistemu. Navedene entitete potrebno je izdvojiti u nove tabele u novoj bazi podataka koju koristi samo prvoizdvojevi mikroservis.

Kako su ID-ijevi različitih vrsta korisnika vezani za ostale entitete u sistemu, veza stranog ključa se raskida. Referenciranje po stranom ključu više nije moguće podržati s obzirom da sada tabele ne postoje u bazi podataka koju koristi monolit. Mesta stranih ključeva u pogledu entiteta korisnika, provajdera i admina postaju obične kolone koje služe kao referenca na ID nabrojanih entiteta (Sl. 7 i Sl. 8). Pomenuti korak predstavlja možda i najteži deo posla, jer utiče na složenost održavanja sistema [16].



Sl. 7 Tabele baze podataka koju koristi IAM mikroservis



Sl. 8 Kolona booking_provider_id više nije strani ključ u okviru tabele baze podataka koju koristi monolit (analogno i za kolone ostalih tabela)

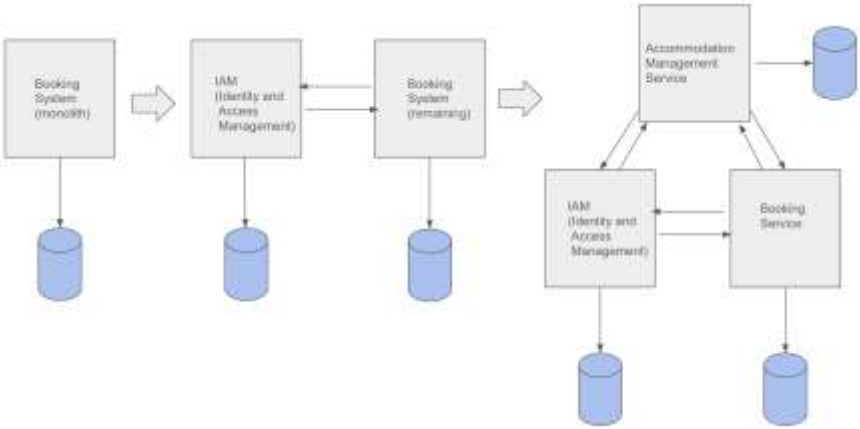
Kada je tako podeljen sistem implementiran, testiran i funkcionalan, potrebno je izvršiti finalnu podelu. Budući da se prvo razmatrala dekompozicija po poslovnoj sposobnosti prepoznata su tri opsega sposobnosti sistema,pa je sledeći korak ujedno i poslednji korak u refaktorisanju monolita.

Odvaja se još jedan servis, servis za rezervaciju smeštaja. Upravljanje smeštajem ostaje kao zasebni servis koji u ovom trenutku ima najveći opseg funkcionalnosti i koji će zapravo biti ostatak monolita po izdvajanju servisa za rezervaciju smeštaja. Koraci se ponavljaju kao i u prošloj iteraciji, pa je pored koda potrebno podeliti i bazu podataka. Posmatrajući model podataka na Sl. 4, može se uočiti da entitet koji predstavlja okosnicu novog mikroservisa za rezervaciju smeštaja jeste entitet Booking. U monolitnom sistemu, Booking je imao dva strana ključa pri čemu je veza stranog ključa za korisnika raskinuta

u prošloj iteraciji kada se izdvajao mikroservis za upravljanje korisnicima (Sl. 9). Kako je nadležnost mikroservisa za upravljanje smeštajem manipulacija nad fasilitijem i studiom, raskida se veze stranog ključa u okviru Booking-a za koji se pamti id smeštaja. Studio je sada tabela drugog mikroservisa, pa sada kao i u prethodnom koraku, studio id postaje obična kolona koja služi kao referenca na rezervisani smeštaj. Na ovaj način sistem je u potpunosti podeljen na mikroservise pri čemu je izdvojeno tri mikroservisa; servis za upravljanje korisnicima (IAM), servis za upravljanje smeštajem i servis za rezervaciju smeštaja pri čemu svaki servis koristi svoju bazu podataka (Sl. 10).

Sl. 9 Tabela Booking u prvoj iteraciji dekompozicije baze podataka ima jedan strani ključ (studio_id), dok je booking_user_id obična kolona koja čuva refenencu na korisnika

Kao što je prikazano, prvi izbor tehnike dekompozicije možda neće uvek biti i najlakši za izvesti. Potrebno je dobro proučiti šta svaka od tehnika dekompozicije zahteva i kako se ti zahtevi uklapaju u kontekstu monolita koji već postoji. Izbor odgovarajućeg šablona direktno utiče na performanse, održavanje i skalabilnost. Iz tog razloga, u redu je iterirati po mogućim šablonima i izabrati onaj koji je najbolji za primeniti. U slučaju monolita za rezervisanje smeštaja, doneta je odluka da to bude šablon gušenja.



Sl. 10 Postupak dekompozicije šablonskim gušenjem monolitnog bukinga na mikroservise

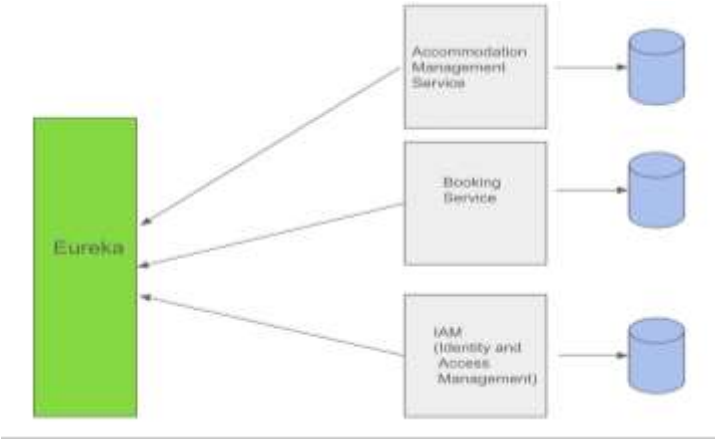
D. Implementacija komunikacije između mikroservisa

Bitan aspekt svakog sistema je komunikacija. U slučaju monolita sve funkcionalnosti obezbeđene su pozivima operacija unutar jednog sistema. Kako je monolit prebačen u mikroservise, ovo više nije moguće. Ideja sada jeste da mikroservisi međusobno komuniciraju i to je moguće učiniti sinhronim i asinhronim putem.

Sinhroni način komunikacije uključuje direktne API pozive između dva mikroservisa pri čemu se mikroservis koji šalje poziv blokira sve dok ne dobije odgovor od mikroservisa kojeg potražuje. Ovakav stil komunikacije izabran je u situacijama kada je neophodan odgovor drugog mikroservisa i nije moguće obezbediti očekivano funkcionisanje sistema dok se ne dobije odgovor. Jedan od primera direktnih API poziva između mikroservisa je situacija kada korisnik potražuje dostupne studije za rezervaciju u određenom intervalu. Kako je studio nadležnost mikroservisa za upravljanje smeštajem, klijent ga kontaktira. Uz dohvatanje svih studija potrebno ih je izfiltrirati u odnosu na dostupnost, jer rezultati koje klijent dobija treba da sadrže samo studije koje je moguće rezervisati u traženo vremenskom opsegu. Kako je ovo blokirajuća situacija te je neophodno filtriranje primeniti pre nego što se odgovor vrati klijentu, potrebno je da servis za upravljanje smeštajem iskomunicira direktno sa servisom za rezervacije. Servis za upravljanje smeštajem šalje poziv servisu za rezervacije tražeći sve rezervacije u prosleđenom intervalu koje kasnije primenjuje kao filter. Komunikacija se vrši direktnim API pozivom koristeći

eng. *Feign Client*.

Komunikacija između mikroservisa asinhronim putem realizovana je korišćenjem brokera poruka (eng. *Message Broker*). Kao što je to prethodno objašnjeno, asinhrona komunikacija koristi se u situacijama kada nije potrebno čekati odgovor već servis može nesmetano da nastavi sa radom. Primer asinhrone komunikacije u buking sistemu je situacija kada korisnik briše svoj nalog. Kako je prilikom dekompozicije baze podataka raskinuta veza stranog ključa, nakon brisanja korisnika iz sistema potrebno je obavestiti ostale servise koji je to korisnik koji je obrisan kako bi servisi mogli da ažuriraju podatke u bazi. Drugim rečima, potrebno je da servisi obrišu redove iz tabela koje čuvaju referencu na korisnika koji briše svoj nalog. Kako ovo nije blokirajuća situacija, jer nije potrebno sačekati da ostali servisi uklone podatke vezane za obrisanog korisnika kako bi se korisnik zaista obrisao iz sistema, dovoljno je samo poslati poruku u red. Servisi koji su pretplaćeni na red čiji je eng. *topic* „brisanje-korisnika", u trenutku kada poruka stigne, obradiće događaj i preduzeti odgovarajuću akciju na osnovu ID korisnika koji je stigao. Akcija u ovom slučaju jeste eliminisanje svih podataka povezanih sa korisnikom koji nije više deo sistema.



Sl. 11 Booking sistem i Eureka

Pored implementiranih mikroservisa, dodata je i implementacija Eureke (Sl. 11). Eureka je implementacija registra servisa koji omogućava mikroservisima da se registruju i otkriju druge mikroservise. Sastoji se od dva dela, Eureke servera i Eureke klijenta. Eureka server je komponenta koja se koristi za registraciju i otkrivanje mikroservisa. Server sadrži adrese mikroservise, kao i njegove metapodatke. Eureka klijent je biblioteka koja se integriše u mikroservise kako bi se registrovali i kako bi mogli da se otkriju od strane

drugih mikroservisa. Svaki mikroservis koji želi da bude registrovan od strane Eureke servera, mora uključiti klijenta. Eureka klijent periodično šalje informacije o svom stanju serveru.

IV. ZAKLJUČAK

U radu je predstavljen teorijski pogled na dekompoziciju monolitnog sistema na mikroservisnu arhitekturu. Pružen je detaljan uvid u pomenute arhitekture, principe i ograničenja implementacije, kao i najbolje prakse u migraciji. Pored teorijskog osvrta, pružen je i praktični deo rada koji je za cilj imao da ispita pomenute preporuke u refaktorisanju u softverskom inženjerstvu. Teorijski deo rada može služiti kao priručnik za implementaciju mikroservisa. U samom radu, više puta je istaknuto da ne treba slepo pratiti inženjerske trendove, već da je potrebno detaljno proučiti kompatibilnost sistema sa izabranom arhitekturom.

Prilikom prelaksa na mikroservisnu arhitekturu, nije promenjen izbor u tehnologiji pa su svi mikroservisi kao i monolit implementirani u javi korišćenjem Spring Boot-a. Sama java pogodna je za sisteme ovakvog tipa, jer daje mogućnost korišćenja mnogobrojnih biblioteka koje se dobro uklapaju sa pomenutom arhitekturom. Načinjenom migracijom, ukoliko u budućnosti bude postojala potreba za prelaskom na neku drugu tehnologiju, to neće predstavljati problem, jer će biti potrebno prebaciti samo jedan mikroservis a ne čitav sistem. Prelaskom na mikroservise, sistem je postao fleksibilan za buduće izmene i usvajanje novih tehnologija. Još neke prednosti koje sistem sada ima, a nije ih imao kao monolit, jesu skalabilnost po potrebi, na osnovu opterećenja moguće je skalirati opterećene mikroservise, zatim bolja otpornost, jer greške u okviru jednog mikroservisa ne utiču na druge.

Nedostaci praktičnog dela rada, napuštanjem monolita povećana je kompleksnost razvoja i održavanja. Uvođenjem mikroservisne ahitekture potrebno je usvojiti i mnoge druge alate kako bi sistem bio funkcionalan.

U pogledu poboljšanja i budućih proširenja, potrebno je implementirati servis koji će se baviti plaćanjem rezervisanih smeštaja. Pomenuti mikroservis proširio bi funkcionalnosti postojećeg sistema i na taj način Booking sistem imao bi sve što mu je potrebno da samostalno funkcioniše. S obzirom da će se razmatrati povećanje broja mikroservisa, potrebno je razmisliti i o uvođenju API Gateway-a s obzirom da se mikroservisi mogu preopteretiti pozivima, a i cilj je imati jednu centralizovanu tačku komunikacije. Cilj sledeće iteracije jeste da se sistem sastoji od četiri mikroservisa, tačnije mikroservisa za upravljanje korisnicima, upravljanje smeštajem, rezervaciju smeštaja i plaćanje, kao i da se dobre performanse održe i sa usvajanjem navedenih proširenja.

LITERATURA

[1] Sam Newman (2014). “Building Microservices - Designing Fine-Grained Systems”, O’REILLY, ISBN 978-1492034025

[2] Sam Newman (2019). “Monolith to Microservices - Evolutionary Patterns to Transform Your Monolith”, O’REILLY, ISBN 978-1492047841

[3] Chris Richardson (2018). “Microservices Patterns: With Examples in Java”, Manning, ISBN 978-1617294549

[4] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, Michael Amundsen (2016). “Microservice Architecture: Aligning Principles, Practices and Culture”, O’REILLY, ISBN 978-1491956250

[5] Morgan Bruce, Paulo A. Pereira (2018), “Microservices in Action”, Manning, ISBN 978-1617294457

[6] Kalske, Miika, Niko Mäkitalo, and Tommi Mikkonen (2017). "Challenges when moving from monolith to microservice architecture." *Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17*. Springer International Publishing, 2018.

[7] D. Taibi, V. Lenarduzzi and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation," in IEEE Cloud Computing, vol. 4, no. 5, pp. 22-32, September/October 2017, doi: 10.1109/MCC.2017.4250931.

[8] V. Velepucha and P. Flores, "Monoliths to microservices - Migration Problems and Challenges: A SMS," 2021 Second International Conference on Information Systems and Software Technologies (ICI2ST), Quito, Ecuador, 2021, pp. 135-142, doi: 10.1109/ICI2ST51859.2021.00027.

[9] Nela Ford, Rebecca Parsons, Patrick Kua (2017), “Building Evolutionary Architectures: Support Constant Change”, Manning, O’REILLY, ISBN 978-1491986363

[10] CoderCo (2023). “Monolith vs. Microservices: A Guide To Choosing the Right Architecture for Your Application”. [Website article]. Retrieved from https://blog.coderco.io/p/monoliths-vs-microservices-a-guide

[11] Robert C. Martin (2008). “Clean code”, Pearson Education, ISBN 978-0132350884

[12] Tzachi Strugo (2021). “Authentication & Authorization in Microservices architecture”. [Website article]. Retrieved from https://dev.to/behalf/authentication-authorization-in-microservices-architecture-part-i-2cn0

[13] Marty Abbott (2019). “Strangler Pattern Dos and Donts”. [Website article]. Retrieved https://akfpartners.com/growth-blog/strangler-pattern-dos-and-donts

[14] Neeraj Kushwaha “Microservices Decomposition Design Patterns”. [Website article]. Retrieved https://www.learncsdesign.com/microservices-decomposition-design-patterns/

[15] Talia Nassi (2020). “Branch by Abstraction Process”. [Website article]. Retrieved https://www.split.io/blog/branch-by-abstraction/

[16] Edson Yanaga (2017), “Migrating to Microservice Databases: From Relational Monolith to Distributed Data”, O’REILLY, ISBN 978-1492077124

ABSTRACT

The paper presents the migration process from monolithic systems to microservices in the context of software engineering. The focus is on the theoretical analysis of architectures and practical aspects of migration. Strategies and recommendations used during the process are presented, along with key factors influencing the outcome of migration. The importance of assessment and planning for the transition is emphasized, as well as situations in which such transformation may not be advisable.

FROM MONOLITH TO MICROSERVICES - A COMPREHENSIVE APPROACH TO REFACTORING IN SOFTWARE ENGINEERING

Milica Bakić, Bojana Dimić Surla