Migriranje poslužiteljskog dijela aplikacije s monolitne na arhitekturu mikrousluga

Kupanovac, Filip

SVEUČILIŠTE JOSIPA JURJA STOSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

MIGRIRANJE POSLUŽITELJSKOG DIJELA APLIKACIJE S MONOLITNE NA ARHITEKTURU MIKROUSLUGA

Diplomski rad

Filip Kupanovac

Osijek, 2023.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 12.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Filip Kupanovac
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1216R, 07.10.2021.
OIB studenta:	12846069760
Mentor:	prof. dr. sc. Goran Martinović
Sumentor:	,
Sumentor iz tvrtke:	Nemanja Plavšić, mag.ing.comp.
Predsjednik Povjerenstva:	izv. prof. dr. sc. Alfonzo Baumgartner
Član Povjerenstva 1:	prof. dr. sc. Goran Martinović
Član Povjerenstva 2:	doc. dr. sc. Tomislav Galba
Naslov diplomskog rada:	Migriranje poslužiteljskog dijela aplikacije s monolitne na arhitekturu mikrousluga
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskom dijelu diplomskog rada treba analizirati probleme i izazove upravljanja performansama nogometnog kluba, korištenja monolitne multi-tenancy arhitekture i domenski pokretanog dizajna u razvoju programskog rješenja. Na temelju analize postojećih sličnih rješenja, te mogućnosti monolitne i arhitekture mikrousluga, potrebno je definirati funkcionalne i nefunkcionalne zahtjeve na poslužiteljsku stranu rješenja, predložiti model i arhitekturu monolitnog i rješenja temeljenog na arhitekturi mikrousluga, kao i postupak migriranja. Poslužiteljska strana programskog rješenja, na temelju podataka prikupljenih unosom od strane igrača i s odgovarajućih senzora treba obaviti prikladnu analizu, te stvarati preporuke, upozorenja i informacije koje uzimaju u
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	12.09.2023.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 28.09.2023.

Ime i prezime studenta:	Filip Kupanovac
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1216R, 07.10.2021.
Turnitin podudaranje [%]:	4

Ovom izjavom izjavljujem da je rad pod nazivom**: Migriranje poslužiteljskog dijela aplikacije s monolitne na arhitekturu mikrousluga**

izrađen pod vodstvom mentora prof. dr. sc. Goran Martinović

i sumentora ,

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Razvojem programskog inženjerstva kao grane znanosti i industrije mijenjaju se pristupi stvaranja programske podrške. Skupom određenih načela prema ostvarenju željenih karakteristika ostvaruje se arhitektura programske podrške. Razvojem novih arhitektura uočava se bolje odgovaranje drugog arhitekturnog stila od onog primijenjenog. Tada je moguće migriranje programske podrške kako bi se bolje istakle željene karakteristike programske podrške. Posljednjih godina arhitektura mikrousluga postaje sve popularnije rješenje u razvoju programske podrške. Razloga za to je mnogo, a najvažniji su mogućnost podjele odgovornosti na zasebne jedinice – usluge, bolje upravljanje performansama i povećanje skalabilnosti pojedinih dijelova aplikacije ovisno o potrebama različitih cjelina koje čine aplikaciju. Ovime se postiže veća fleksibilnost tijekom održavanja poslužiteljske aplikacije u svrhu postizanja boljih performansi za veće zadovoljstvo klijenata koji koriste aplikaciju kao i veće efikasnosti i isplativosti aplikacije organizaciji koja ju održava. Radi ovih prednosti mnoge organizacije žele svoje postojeće monolitne aplikacije koje su u upotrebi migrirati na arhitekturu mikrousluga kako bi ostvarili prednosti koje ovakva arhitektura omogućava.

Ovaj rad ima za svrhu prikazati migriranje postojeće aplikacije izrađene u monolitnoj arhitekturi u arhitekturu mikrousluga. Istraživanjem i analizom postojećih arhitektura uspoređene su karakteristike različitih arhitektura. Prikazani su problemi monolitne arhitekture koji mogu biti umanjeni korištenjem raspodijeljene arhitekture i prednosti ostvarive korištenjem arhitekture mikrousluga. Uspoređeni su i objašnjeni različiti obrasci provođenja migracije postojeće programske podrške iz monolitne u arhitekturu mikrousluga. Na praktičnom primjeru aplikacije za menadžment sportskog kluba detaljno je prikazano migriranje uzimajući u obzir potrebne radnje i analizu postojećeg rješenja prije prelaska na novu arhitekturu, mogućnosti i nove pogodnosti koje se postižu migriranjem.

U drugom poglavlju rada teorijski je objašnjen pojam arhitekture i različitih vrsta arhitektura programske podrške. Prikazani su izazovi migracije programske podrške i usporedba različitih arhitektura s prikazanim prednostima arhitekture mikrousluga te obrasci migriranja programske podrške. Treće poglavlje opisuje zahtjeve na aplikaciju koje ispunjava monolitna aplikacija, način izvedbe funkcionalnosti i plan migriranja aplikacije na arhitekturu mikrousluga. U četvrtom poglavlju prikazan je praktični primjer migriranja programske podrške na arhitekturu mikrousluga korištenjem obrazaca migracije prema koracima migriranja, dok je zatim u petom poglavlju provedena analiza migriranja danog sustava uz provjeru ispunjavanja traženih prednosti koje trebaju biti postignute migriranjem.

2. IZAZOVI MIGRIRANJA PROGRAMSKE PODRŠKE S MONOLITNE NA ARHITEKTURU MIKROUSLUGA I STANJE U PODRUČJU

U ovom poglavlju bit će teorijski opisano značenje arhitekture programske podrške i detaljnije objašnjene neke značajne arhitekture uz opis migriranja programske podrške.

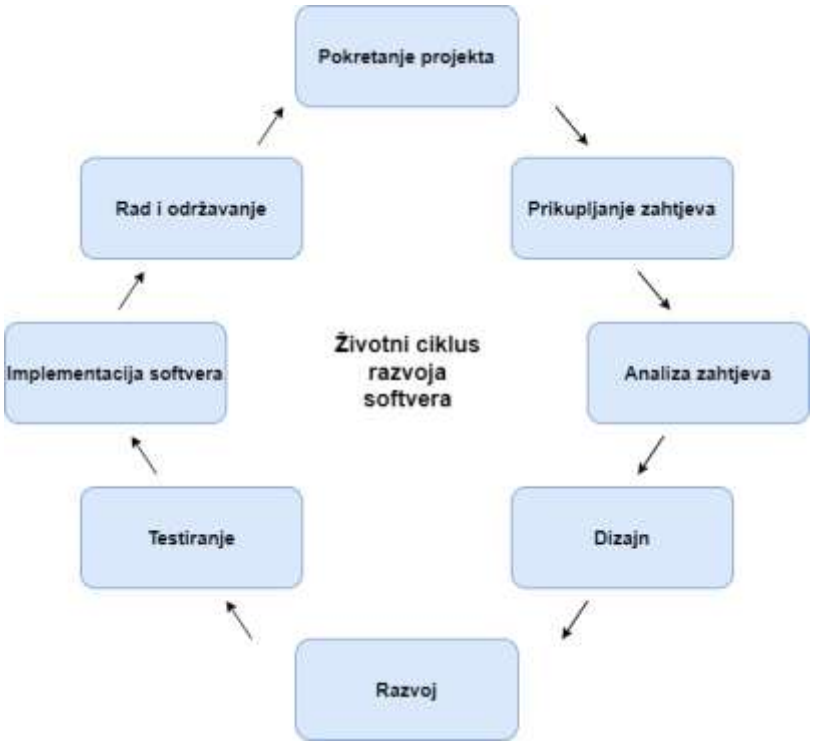
2.1. NASTANAK ARHITEKTURE PROGRAMSKE PODRŠKE

2.1.1. Definicija arhitekture programske podrške

Arhitektura programske podrške (*eng. software arhitecture*) je sintagma koja se u računarstvu koristi često bez potpunog razumijevanja što ona predstavlja, niti ima jedinstvenu definiciju nego je opisana konceptualno. Stoga je potrebno objasniti što ona predstavlja unutar programskog inženjerstva i razvoja programske podrške. Arhitektura ne označava radi li program ili ne, štoviše, ima jako malu korelaciju s time. Od stvaranja prve programske podrške, postojala su bolja i lošija programska rješenja u vidu korištene arhitekture. Lošija rješenja koja su korištena su ispunjavala svoju svrhu, odrađivala potrebne zadatke i ispunjavala svoju svrhu. No, možemo ih smatrati lošijima prema drugim aspektima. Osim da ispunjavaju zahtjeve dane na aplikaciju, poželjno je da programska podrška bude efikasna i jeftina za izmjene, zahtijeva manju računalnu snagu odnosno manje sklopovske komponente, da bude pouzdana i dostupna korisnicima uz poboljšanje korisničkog iskustva sa što manje kvarova i ispadanja sustava radi održavanja i popravaka programske podrške.

Programska rješenja koja ispunjavaju navedene zahtjeve smatraju se boljim rješenjima. Takva programska podrška zahtijeva više analiziranja i promišljanja prije samog ostvarenja pisanjem programskog koda, no dugoročno donosi bolje rezultate i više uspjeha. Znanja sadržana kroz rad na takvim rješenjima su strukturirana i dokumentirana kao dobre programerske prakse i sadržana u vidu arhitektura programske podrške– načela za bolju i efikasniju izradu programske podrške.

Arhitektura programske podrške je naziv dan kao metafora na arhitekturu u građevinarstvu. Ona predstavlja nacrte i okvirne planove rješenja, način slaganja i rasporeda komponenti programske podrške u sustavu, njihovo povezivanje, način komunikacije. Odabir arhitekture uvelike određuje kako će aplikacija biti strukturirana. Svrha korištenja arhitekture programske podrške je bolje upravljanje procesom izrade podrške i praćenje kroz glavne faze životnog ciklusa programske podrške (*eng. SDLC – Software Development Life Cycle*), opisane prema standardu [1] Međunarodne organizacije za standardizaciju (ISO)



Slika 2.1: Životni ciklus razvoja programske podrške

Unutar životnog ciklusa razvoja programske podrške sadržane su faze na koje treba obratiti pažnju prilikom odabira arhitekture. To su sljedeće faze:

- faza razvoja (*development*)
- faza implementacije (*deployment*)
- faza rada (*operation*)
- faza održavanja (*maintenance*)

U fazi razvoja, odabir arhitekture ovisi o raspoređenosti kadrova koji razvijaju podršku. Ako sustav nije lako razviti, vrlo je vjerojatno da neće imati dug vijek te će biti obilježen brojnim problemima. Potrebno je predvidjeti broj timova i programera koji rade na projektu te u skladu s time odrediti zaduženja, kako bi bilo dovoljno komponenti da timovi ne smetaju jedni drugima, ali ne previše da ne budu preopterećeni opsegom posla. Također, kako bi bio efikasan, sustav programske podrške mora biti izvediv (*eng. deployable*). Što je veća cijena izvedivosti, manje će biti koristan – cilj arhitekture treba biti razvoj podrške koja je lako izvediva u što manje koraka. U praksi se ova faza rijetko promatra prilikom analize zahtjeva i pokretanja projekta, stoga se može dogoditi da se primjera radi lakšeg razvoja našteti izvedivosti, odnosno stvori sustav s prevelikim brojem mikrousluga čije je povezivanje teško i vrijeme inicijacije sustava iznimno veliko, što dovodi do manje izvedivosti. Potrebno je uskladiti razvoj s izvedivosti i tako pronaći balans između ove dvije faze.

Utjecaj arhitekture je od prethodno navedene četiri faze najmanji na fazu rada. Obzirom da se ova faza odnosi ponajviše na računalnu infrastrukturu, odnosno sklopovlje, probleme radi sporog rada, nedovoljno memorije ili nedovoljne snage računalnog sustava možemo riješiti dodavanjem dodatne infrastrukture povećavajući tako performanse poslužitelja na kojem se izvodi programska podrška. Iako je ovo najjeftinija faza što se tiče nadomještanja i ispravljanja nedostataka, ne treba se zanemariti. Dobrom arhitekturom može se predvidjeti potrebe za snagom i kapacitetima sustava koji će izvoditi programsku podršku te tako spriječiti dodatne troškove.

Zadnja faza na koju treba obratiti pažnju prilikom odabira arhitekture je faza održavanja. To je najskuplja faza u životnom ciklusu podrške. Unutar SDLC-a je usko povezana s fazom rada, no odnosi se na održavanje programske podrške – rješavanje novootkrivenih problema i lošeg rada podrške te dodavanja novih značajki radi promjene zahtjeva ili poslovne logike. Kako bi smo izmijenili programsku podršku koja je već puštena u rad potrebno je pažljivo razmotriti gdje se i kako može napraviti izmjena, imajući na umu da ne smijemo narušiti ostala ponašanja podrške, niti prekinuti njezin rad prouzrokovanjem pogreške ili kvara. Pažljivo odabrana arhitektura značajno ublažava mogućnosti pojave ovakvih problema, jer se može postići fleksibilnije programsko rješenje. Podjelom programske podrške na manje komponente i njihovom izolacijom korištenjem sučelja moguće je razdvojiti ovisnosti, napraviti ih slabo povezanima uz zadržavanje kohezije čime se olakšava buduća izmjena postojeće poslovne logike, kao i implementacija novih značajki za proširenje aplikacije. Uz dobro definiranu arhitekturu prilagođenu potrebama programske podrške možemo uvelike smanjiti troškove održavanja.

Ovisno o potrebama ovih faza kroji se arhitekturno rješenje pri čemu je potrebno imati u vidu kako će se podrška ponašati u svakoj od faza životnog ciklusa te u skladu s time mijenjati plan razvoja podrške. Iz ovoga je vidljivo da iako nema ključnu ulogu za ispravan rad, arhitektura ima vrlo važnu ulogu u podržavanju ispravnog i efikasnog rada sustava te omogućava dugoročniji životni ciklus programske podrške uz manje budućih troškova [2].

Od prvih dana računarstva traženi su načini za pojednostavljenje programskih rješenja, efikasniju programsku podršku koja bi zahtijevala manje utrošenih resursa i koja se mogla izvoditi na sklopovskoj okolini manjih kapaciteta. Isprva je razlog tomu bio visoka cijena računalne opreme i samih računala.

2.1.2. Povijesni preduvjeti razvoja arhitekture programske podrške

Kod prvih računala koja su koristila jezike niske razine (*eng. low-level language*), programeri su unosili naredbe i podatke izravno u memoriju računala. Optimizacija rada na takvim sustavima podrazumijevala je automatizaciju kroz ustaljen raspored memorije i ažuriranje

referenci na naredbe. Ovakva referenciranja su dovela do razvoja ljudima čitljivijih asemblerskih jezika, gdje je su simboli predstavljali skup često korištenih operacija.

Daljnjim razvojem računarstva stvoreni su viši programski jezici – jezici koji su ljudima lako čitljivi, a prilikom izvođenja na računalu se kompajliraju i prevode u strojni kod koji računala koriste. Takvi programi i dalje nisu bili veliki, ograničeni su računalnom snagom tadašnjih računala koja nisu imala mnogo memorije niti procesorske snage za izvršavanje velikog broja instrukcija. Kako bi razvijali veće programe, potrebno je bilo koristiti odgovarajuće podatkovne strukture i razviti efikasnije algoritme.

Povećanjem snage računala, njihove memorije i većom dostupnošću, rasla je i kompleksnost sustava programske podrške. Aplikacije su rasle od nekoliko stotina instrukcija do tisuća instrukcija unutar jedne aplikacije. Kako su aplikacije stvarane s puno više različitih mogućnosti, nastala je potreba za razdvajanjem odgovornosti. Takve aplikacije su prerasle probleme podatkovnih struktura i algoritama te su se pojavili novi problemi – strukturalni problemi. Strukturalni problemi podrazumijevaju organizaciju i kontrolu nad cjelokupnom strukturom. Uvođenjem višestrukih komponenti i modula radi razdvajanja odgovornosti, potrebno je osigurati njihovu komunikaciju, sinkronizaciju, pristup podacima, dizajn, kompoziciju i fizičku raspodjelu na računalnoj infrastrukturi kako bi omogućili skalabilnost aplikacije i osigurali dobre performanse [3]. Navedeni problemi pokušavaju se riješiti primjenom arhitekture programske podrške.

2.1.3. Početci arhitekturnih obrazaca

Arhitekturni stilovi, ili obrasci, opisuju odnose između komponenti koje čine neku aplikaciju. Unutar obrasca opisana je struktura i raspored komponenti, njihova međusobna komunikacija, komunikacija prema van s korisnicima ili drugim uslugama, međuovisnosti i podjela odgovornosti među komponentama sustava. Kroz povijest se u praksi ponavljaju određeni obrasci koji su se pokazali dobrima u pogledu strukturiranja komponenti koje čine programsku podršku, organizaciji koda, izvodljivosti i drugim aspektima na koje treba obratiti pažnju prilikom stvaranja podrške.

Prije nego su stvoreni sustavi kakvi su danas najčešće korišteni, osnovni arhitekturni obrasci su bili drugačije koncipirani i bili su vezani uz vrste računala na kojima se izvode i ograničena računalnom arhitekturom. Tako su nastali poznati arhitekturni obrasci:

- Jedinstvena arhitektura (*eng. Unitary architecture*)
- Klijent-poslužitelj
- Anti-obrazac Big Ball of Mud

Jedinstvena arhitektura označava sustave koji su stvoreni da cjelokupnu funkcionalnost izvode na jednom računalu. Jedna je to od prvih korištenih arhitektura, dok su računala služila za obavljanje pojedinačnih operacija bez komunikacije s mrežom.

Arhitektura klijent-poslužitelj nastala je kao nastavak razvoja računala i korištenja mrežne komunikacije između više računala. Sa klijentskih računala slani su zahtjevi i na njima prikazivani podatci zaprimani od poslužitelja, a na poslužiteljima su obavljane kompleksnije operacije, služio za spremanje podataka i održavanje baze podataka ili ograničenje komunikacije. Poslužiteljem su definirani protokoli komunikacije kojima se klijent mora prilagoditi i koristiti ih kako bi dobio podatke od poslužitelja.

Anti-obrazac *Big Ball of Mud* [4] je primjer loše arhitekture programske podrške. Valja ga spomenuti iz razloga što unatoč tome što označava prakse koje pri razvoju treba izbjegavati, biva zastupljen u postojećim rješenjima. On označava aplikacije bez jasne unutarnje strukture, s lošom kohezijom komponenti, ukoliko su one uopće zastupljene. Takvi sustavi obično nastaju održavanjem aplikacija koje počnu s malo značajki i bez puno funkcionalnosti, no održavanjem i dodavanjem novih značajki se isprepliću pozivi unutar aplikacije, kada sustavi nemaju jasnu podjelu odgovornosti ili su usko vezani dijelovi sustava. Dodavanje novih značajki tada unosi velike probleme jer je teško pratiti sve dijelove programskog koda na koje nove promjene utječu. Ovakav pristup također otežava pronalaženje greški i otklanjanje jednom kada nastanu, jer promjena jednog dijela koda može povući za sobom potrebu za promjenom mnogobrojnih drugih komponenti i procedura.

2.2. ARHITEKTURE POSLUŽITELJSKIH APLIKACIJA

Razvojem programske podrške, arhitektura klijent-poslužitelj je postajala sve popularnija i poslužiteljska strana aplikacija je postajala složenija stvarajući nove izazove pri dizajniranju aplikacija poslužiteljske strane. Danas su sustavi tolikih dimenzija da govorimo o samostalnim poslužiteljskim aplikacijama sa zasebnom strukturom i vlastitom arhitekturom poslužiteljskih aplikacija. Prema van one pružaju programsko sučelje klijentima na korištenje, koji mogu biti frontend, desktop, mobilne ili neki drugi oblik aplikacija koje omogućuju korisničko sučelje za komunikaciju s poslužiteljskim aplikacijama.

Među poslužiteljskim aplikacijama postoji nekoliko vrlo popularnih arhitektura koje su često zastupljene u praksi radi svojih prednosti koje omogućuju. Obzirom da se svaki projekt razlikuje, a tako i zahtjevi na aplikaciju, potrebno je odabrati ispravan tip arhitekture koji zadovoljava zahtjeve aplikacije. U ovom slučaju se ponajviše misli na nefunkcionalne zahtjeve na aplikaciju koji podrazumijevaju u prvom redu pouzdanost, skalabilnost, sigurnost, upotrebljivost,

održavljivost, dostupnost, testabilnost i mnoge druge. Ne postoji univerzalni tip arhitekture koji je najbolji u svim ovim aspektima, ili primjenjiv u svakoj situaciji. Iz tog razloga nastalo je mnogo različitih arhitektura s ciljem boljeg razvoja programske podrške ovisno o traženim zahtjevima.

2.2.1. Podjela poslužiteljskih aplikacija prema implementaciji

Jedna od najbitnijih karakteristika poslužiteljskih aplikacija je na kakvom su sustavu pokretane. One mogu biti implementirane i pokretane na jednom računalu ili na više računala odnosno logičkih čvorova u mreži. Prema tome možemo ih podijeliti na monolitne i raspodijeljene aplikacije. Kako se takve aplikacije bitno razlikuju međusobno, postoje različite arhitekture ovisno o tome na kakvom se sustavu izvode, a u nastavku su navedene neke popularne i u praksi često korištene arhitekture za izradu poslužiteljskih aplikacija, podijeljene prema sustavu na kojemu su pokretane:

- Monolitne
  - Slojevita
  - Mikrojezgrena
  - Cjevovodna arhitektura
- Raspodijeljene
  - Servisno-orijentirana arhitektura
  - Arhitektura pokretana događajima
  - Prostorno-orijentirana arhitektura
  - Arhitektura mikrousluga

Za oba tipa poslužiteljskih aplikacija karakteristične su određene prednosti i nedostaci te je potrebno dobro razumjeti ih da bi se mogao odabrati najbolji pristup i struktura aplikacije. Prema tome zatim možemo suziti izbor arhitekture koja bi najbolje odgovarala zahtjevima. Raspodijeljeni sustavi su moćniji u pogledu performansi, skalabilnosti i dostupnosti u odnosu na monolitne sustave. Unatoč ovim prednostima, imaju i određene nedostatke opisane u nastavku, te ih ne treba nužno koristiti kao bolje rješenje od monolitnih bez razmatranja svih prednosti i nedostataka.

2.2.2. Nedostaci raspodijeljenih sustava u odnosu na monolitne

Od kada su stvoreni raspodijeljeni sustavi, mnoge nove aplikacije su rađene na ovakvim sustavima. Iako donose određene prednosti u radu, često se zanemaruju ili previde određene mane raspodijeljenih sustava, što može negativno utjecati na sam rad aplikacije te tako učiniti da se ponište dobre strane ovakvog razvoja. Neki od najčešćih previda podrazumijevaju zanemarivanje mrežne komunikacije između računala koja pokreću različite dijelove raspodijeljene aplikacije, a uz to javljaju se i određeni rizici i izazovi koji dolaze s ovakvim načinom razvoja programske podrške s kojima se ne susrećemo u monolitnim sustavima i arhitekturama.

Bilježenje događaja označava zapisivanje određenih ključnih događaja u aplikaciji koji se bilježe s ciljem praćenja sustavnih poziva i operacija. Služi za praćenje i nadzor rada sustava te može ukazati na izvor pogreške i neispravnog rada aplikacije. Kod monolitnih aplikacija postoji jedan zapisnik sustava (*eng. system log*) te je bilježenje i praćenje rada sustava kod takvih sustava jednostavno. Pri raspodijeljenom računarstvu pojavljuje se izazov raspodijeljenog bilježenja, iz razloga što arhitekture raspodijeljenih aplikacija podrazumijevaju više različitih zapisnika, rasprostranjenih na različitim čvorovima i s mogućnošću različitog formata, što čini otkrivanje uzroka neispravnog rada određene funkcionalnosti teškim za otkrivanje te može zahtijevati puno vremena za identifikaciju i popravak određenog nedostatka [5].

Raspodijeljene transakcije su sljedeći izazov raspodijeljenog računarstva. Pod pojmom transakcija u programskom inženjerstvu podrazumijeva se skup operacija koje se izvršavaju kao cjelina, te ako jedna naredba iz skupa se ne uspije izvršiti, sve se poništavaju. Obično se odnose na operacije u radu s bazom podataka, gdje jednim pozivom na programsko sučelje aplikacije se može mijenjati nekoliko različitih zapisa sadržanim u jednoj ili više tablica baze podataka. Svojstva koja transakcije moraju zadovoljiti opisane su u ACID (*eng. Atomicity, Consistency, Isolation, Durability*) načelima, koja podrazumijevaju atomičnost, dosljednost, izolaciju i trajnost. Transakcijama je jednostavno upravljati u monolitnim aplikacijama gdje se koristi jedna baza podataka i lako je pridržavati se ACID načela. Nasuprot tome, u raspodijeljenim sustavima susrećemo se s raspodijeljenim transakcijama. Poziv funkcije u jednoj usluzi može zahtijevati obradu i izmjenu podataka u drugim uslugama. Za osiguranje dosljednosti, u raspodijeljenim arhitekturama koristi se princip eventualne dosljednosti (*eng. eventual consistency*). Promjene podataka među različitim čvorovima se ne propagiraju istog trenutka, nego tijekom određenog vremenskog perioda se događa potpuna sinkronizacija među čvorovima [6]. Na ovaj način se žrtvuje dosljednost i integritet podataka radi ostvarenja boljih performansi, skalabilnosti i dostupnosti same aplikacije.

2.2.3. Osam zabluda raspodijeljenog računarstva

Previdi korištenja raspodijeljenih arhitektura u smislu zanemarivanja mrežne komunikacije u računalima raspodijeljenih sustava su poznati kao zablude raspodijeljenog računarstva [7], a one uključuju sljedeće stavove, čija detaljnija objašnjenja slijede u nastavku:

- Mreža je uvijek pouzdana
- Vrijeme odziva je nula
- Propusnost mreže je neograničena
- Mreža je sigurna

- Topologija mreže je nepromjenjiva
- Samo je jedan administrator mreže
- Trošak prijenosa podataka mrežom je nula
- Mreža je homogena

Mreža je pouzdana – prva zabluda uključuje vjerovanje da je mreža potpuno pouzdana i radi u bilo kojem trenutku. Unatoč visokoj razini pouzdanosti, mreža nije uvijek ispravna, događaju se trenuci kada mreža ne radi i stoga treba to uzeti u obzir te pripremiti strategiju kako upravljati aplikacijom u tom slučaju. Iako je to iznimno mali udio vremena u ukupnom radu, treba biti svjestan ovog nedostatka, posebno ako aplikacija komunicira mrežno s vanjskim partnerima koji nisu pod našom nadležnošću i kontrolom.

Vrijeme odziva je nula – sljedeća zabluda odnosi se na slanje poziva udaljenim objektima putem mreže (*eng. remote access protocol*), odnosno drugim uslugama na mreži jednim od protokola poput REST (*eng. REpresentional State Transfer*), razmjene poruka ili RPC (*eng. Remote Procedure Call*). Kod lokalnih poziva između komponenata monolita, vrijeme odziva je iznimno malo obzirom da pozivi ne prolaze mrežom, ne trebaju biti validirani u tim međukoracima, poput provjere sigurnosti poziva na udaljenoj usluzi koju je inicirala druga usluga. U raspodijeljenim sustavima moramo uračunati i trajanje komunikacije između usluga, koje usporava odziv [8], poznavati prosječno trajanje komunikacije, ali i kritične slučajeve radi upravljanja pozivima i istekom vremena za zahtjev u tim slučajevima.

Sljedeća zabluda je „Propusnost mreže je neograničena“. U monolitnom sustavu ovo nije problem stoga što se zahtjevi obrađuju unutar jednog računala, ne koristi se mreža za Svaki poziv između usluga raspodijeljenog sustava odvija se putem mreže i korišten je dio propusnog pojasa. Ukoliko aplikacija ima puno korisnika, za očekivati je veliki broj (gotovo) istovremenih poziva. U slučaju kada nema dovoljno propusnosti mreže, duža su vremena odziva što za sobom povlači probleme i s prethodno dvije opisane zablude. Zato je potrebno imati na umu i propusnost mreže pri dizajniranju programskog rješenja.

Sljedeća zabluda je da je mreža sigurna. Unatoč mnogim razinama sigurnosti koje se primjenjuju, i dalje ne postoji potpuno sigurna zaštita od napada. Monolitni sustavi koji obrađuju zahtjeve unutar više različitih dijelova sustava imaju samo jedan zahtjev za pristupom i potrebna je samo jedna sigurnosna provjera. Kod raspodijeljenih sustava svaka usluga ima svoje pristupne točke, odnosno sučelje na koje se vanjski korisnici spajaju, ali i usluge međusobno koriste te točke. Stoga pri svakom pristupu zahtjeva na svaku od usluga je potrebno provjeriti sigurnost. Ovime se povećava ukupni opseg pristupnih točaka na uslugama koje moramo zaštititi, te je moguće da

imamo duže vrijeme odziva radi višestrukih sigurnosnih provjera kada jedan poziv koristi višestruke usluge.

Peta zabluda je kako se topologija mreže nikada ne mijenja. Ovo se odnosi na strukturu mreže i uređaja koji sudjeluju u mreži kao što su usmjerivači, vatrozid i općenito korišteni aparati u mreži. Promjena neke od tih stavki utječe na brzinu komunikacije, što može dovesti do povećanog vremena odziva i vezano s time, učiniti da odziv stigne nakon što smo odlučili prekinuti komunikaciju i javiti da je vrijeme zahtjeva isteklo. Zato je potrebno poznavati topologiju mreže i biti upoznat s promjenama koje se događaju.

Rad mreže nadzire i njime upravlja mrežni administrator. Kod raspodijeljenih sustava možemo imati uređaje rasprostranjene tako da nemaju istog mrežnog administratora. Otuda dolazi sljedeća zabluda – Samo je jedan mrežni administrator. Arhitekti moraju surađivati s više mrežnih administratora te je to potencijalno problem prilikom promjene topologija mreže ili neispravnog rada aplikacije radi predugog vremena odziva. Ova zabluda ukazuje na kompleksnost i potrebnu količinu koordinacije s administratorima kako bi sustav radio ispravno.

Iduća zabluda je da je trošak prijenosa podataka je nula. Iako sličan i često pomiješan sa zabludom da je vrijeme odziva nula, odnosi se na druge aspekte komunikacije. Potrebno je osigurati dovoljnu infrastrukturu kako bi se sva mrežna komunikacija među uslugama nesmetano odvijala. U monolitnom sustavu to je jednostavnije utoliko što nema komunikacije između različitih usluga. No prilikom migriranja na raspodijeljene sustave, javlja se potreba za većom i jačom infrastrukturom. Potrebno je osigurati dodatno sklopovlje, poslužitelje, usmjerivače, vatrozide i bolju zaštitu. Sve ovo ima svoju određenu cijenu te je potrebno uračunati trošak izgradnje potrebite infrastrukture za podržavanje raspodijeljenog sustava.

Zadnja i najrjeđe vjerovana od osam zabluda raspodijeljenog računarstva je: Mreža je homogena. U mreži računala malo je vjerojatno da će sva računala biti jednaka, s istim operacijskim sustavom i da će mreža takva ostati. Radi toga trebamo koristiti široko prihvaćene standardne tehnologije i protokole komunikacije koje većina sustava podržava. Tako neće biti potrebno posebno prilagođavati umrežena računala na proizvoljni protokol i moći će međusobno komunicirati bez suvišnih prilagodbi.

2.3. MONOLITNA ARHITEKTURA

Monolitna arhitektura je tradicionalna arhitektura u razvoju programske podrške. Podrazumijeva razvoj aplikacije kao jedne cjeline, neovisne o drugim aplikacijama. Sva baza koda se nalazi u jednoj aplikaciji, povezujući sve funkcionalnosti unutar cjeline, od poslovne logike aplikacije do programskog sučelja aplikacije koje omogućuje pristup aplikaciji. Također se

uobičajeno koristi jedna baza podataka s višestrukim tablicama unutar iste baze [9]. Monolitna arhitektura je jedna od najstarijih arhitektura programske podrške. Moguće ju je ostvariti na više različitih načina, stoga postoje mnogobrojni različiti tipovi monolitne arhitekture, no u neke od najčešće korištenih spadaju sljedeće:

- Slojevita arhitektura
- Cjevovodna arhitektura
- Mikrojezgrena arhitektura

Kako ovakav tip arhitekture omogućava jednostavnu izradu aplikacija, posebice manjih i jednostavnih bez puno funkcionalnih zahtjeva, monolitna arhitektura je dobar izbor za razvoj programske podrške u određenim slučajevima kada možemo zanemariti neke prisutne nedostatke.

2.3.1. Prednosti i nedostaci monolitne arhitekture

Ovisno o različitim čimbenicima, monolitna arhitektura ima određene prednosti u odnosu na raspodijeljene arhitekture. Prvenstveno, prednosti se očituju u razvoju programske podrške. Monolitna arhitektura omogućava brz razvoj radi jednostavnosti izrade podrške na jedinstvenoj bazi koda. Sve komponente sustava su povezane ovisnostima ili kao biblioteke koje druge komponente koriste. Rezultat toga je velika razina međuovisnosti i nemogućnost zasebnog prevođenja. Kroz rast zahtjeva i proširenje aplikacije, povećava se i baza koda aplikacije. Porast baze proporcionalno otežava i razumijevanje koda te praćenje novih promjena. No, bez obzira na to koliki je rast programske podrške monolita, zadržava se pokretanje monolita s jednom izvršnom datotekom. Sva logika i sve funkcionalnosti su pokretane kroz jedan zajednički proces. Ovo uvelike olakšava implementaciju aplikacije [10].

S obzirom da se radi o jednoj cjelini koja se pokreće zajedno, odnosno jednom procesu, testiranje monolitnih aplikacija je relativno jednostavno. Također, nadziranje i traženje uzroka problema u zapisima aplikacije su značajno olakšani činjenicom da postoji jedinstvena baza koda umjesto da je kod podijeljen na nekoliko procesa. Unatoč dobrim stranama monolitne arhitekture, jako su ranjive. Jedna pogreška može imati utjecaja na prekid rada cijele aplikacije. Razlog tomu je povezanost cjelokupnog koda i što se sve oslanja na isti proces koji pokreće aplikaciju.

Razvojni tim za cijelu monolitnu aplikaciju mora koristiti isti programski jezik, nije moguće pisati pojedinačne komponente u različitim jezicima. To predstavlja značajan problem ukoliko radi poslovne odluke odlučimo promijeniti programski jezik ili okvir. Promjena tada zahvaća sveukupnu programsku podršku, što zahtjeva veliki financijski trošak i utrošak vremena. Također radi promjene jednog dijela podrške i ponovnog pokretanja nove inačice, mora se zaustaviti rad sveukupne podrške. Radi ovakve povezanosti cjelokupne podrške, ne smiju se odvijati niti česte izmjene i poboljšanja u aplikaciji.

Prema navedenim karakteristikama, prednosti možemo sažeto predstaviti kao sljedeće:

- Jedinstvena baza koda
- Jednostavno zapisivanje, testiranje i otklanjanje pogrešaka
- Jedan proces, jednostavna implementacija

S druge strane, glavni nedostaci bili bi:

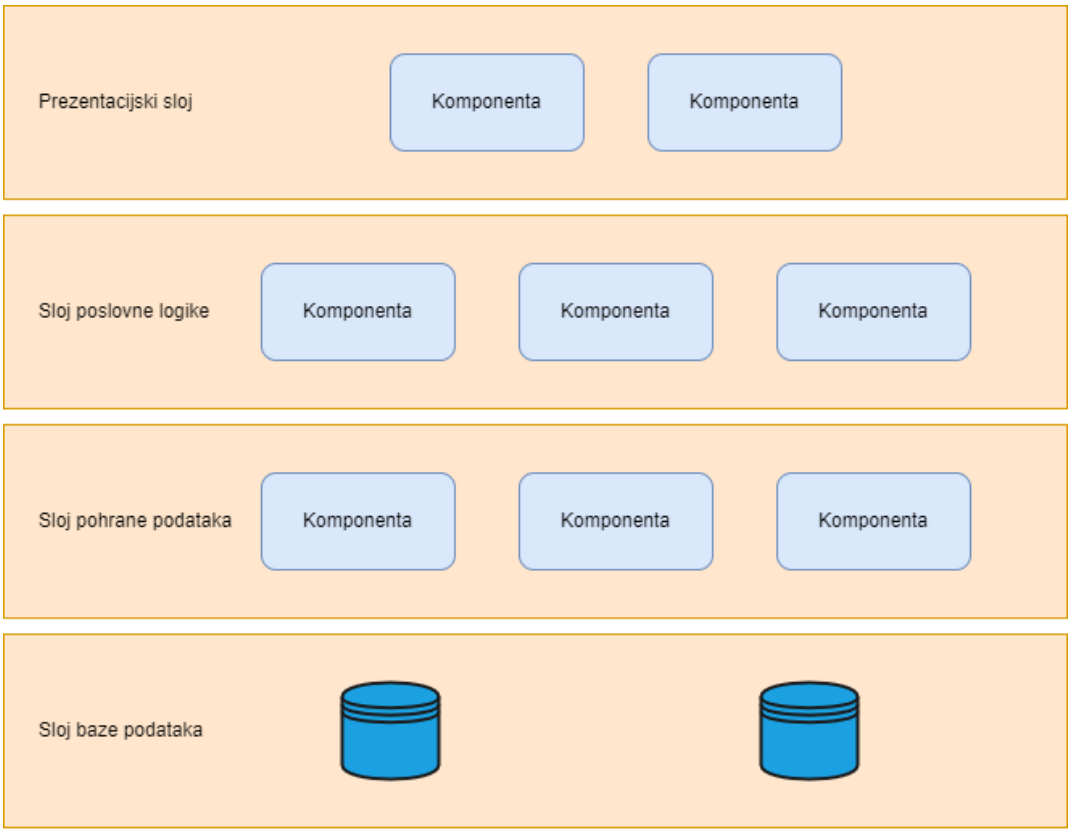
- Otežano razumijevanje porastom aplikacije
- Skalabilnost
- Pogreške prekidaju rad cjelokupne podrške
- Nemogućnost čestih izmjena i poboljšanja

2.3.2. Slojevita arhitektura

Slojevita ili višeslojna arhitektura je najpoznatiji stil arhitekture programske podrške. Popularizirana je radi jednostavnosti i niske cijene. Također predstavlja prirodan način razvoja podrške prema Conwayevom zakonu [11] – arhitektura sustava odražava komunikaciju i organizaciju unutar tima ili organizacije za koju se sustav razvija. Važno je poznavati unutarnju komunikaciju i poslovnu logiku te prema tome dizajnirati sustav kako bi bio što više nalik stvarnom funkcioniranju organizacije.

Komponente unutar sustava slojevite arhitekture možemo prikazati kao slojeve od kojih svaki ima svoju ulogu prateći tako načelo SRP *(Single Responsibility Principle)* razvoja programske podrške. Uobičajeno se koriste četiri standardna sloja kako je prikazano na slici 2.2 – prezentacijski sloj, sloj poslovne logike, sloj pohrane podataka i sloj baze podataka, no ne mora nužno biti tako. Moguće su razne varijante razdvajanja komponenti kao na primjer odvajanje baze podataka u zasebnu cjelinu, ili spajanja slojeva poslovne logike i pohrane podataka u jedan, ostavljajući tako tri sloja ili dodavajući dodatne slojeve u aplikaciju ovisno o potrebama.

Svaki sloj ima svoju odgovornost i služi kao razina apstrakcije oko izvršavanja zahtjeva aplikacije. Sloj baze podataka ne zna, niti treba imati poveznice s time kako će podaci biti prikazani krajnjem korisniku putem prezentacijskog sloja. Također, prezentacijski sloj nema veze sa slojem poslovne logike – on obavlja operacije i obradu podataka koje zadobije od sloja ispod sebe te ih ponovno vraća prezentacijskom sloju u prilagođenom formatu. Takav način podjele odgovornosti jasno odvaja granice slojeva i omogućava timovima da surađuju tako da se potreban posao pri razvoju podjeli prema slojevima. Ovakva arhitektura razdvaja komponente prema ulozi, nasuprot domenskog razdvajanja. Tako su domene programske podrške raštrkane po komponentama koje predstavljaju slojeve – klase jedne domene možemo pronaći i u prezentacijskom sloju, sloju poslovne logike, sloju baze podataka i dodatnim specifičnim slojevima ukoliko ih aplikacija sadrži.



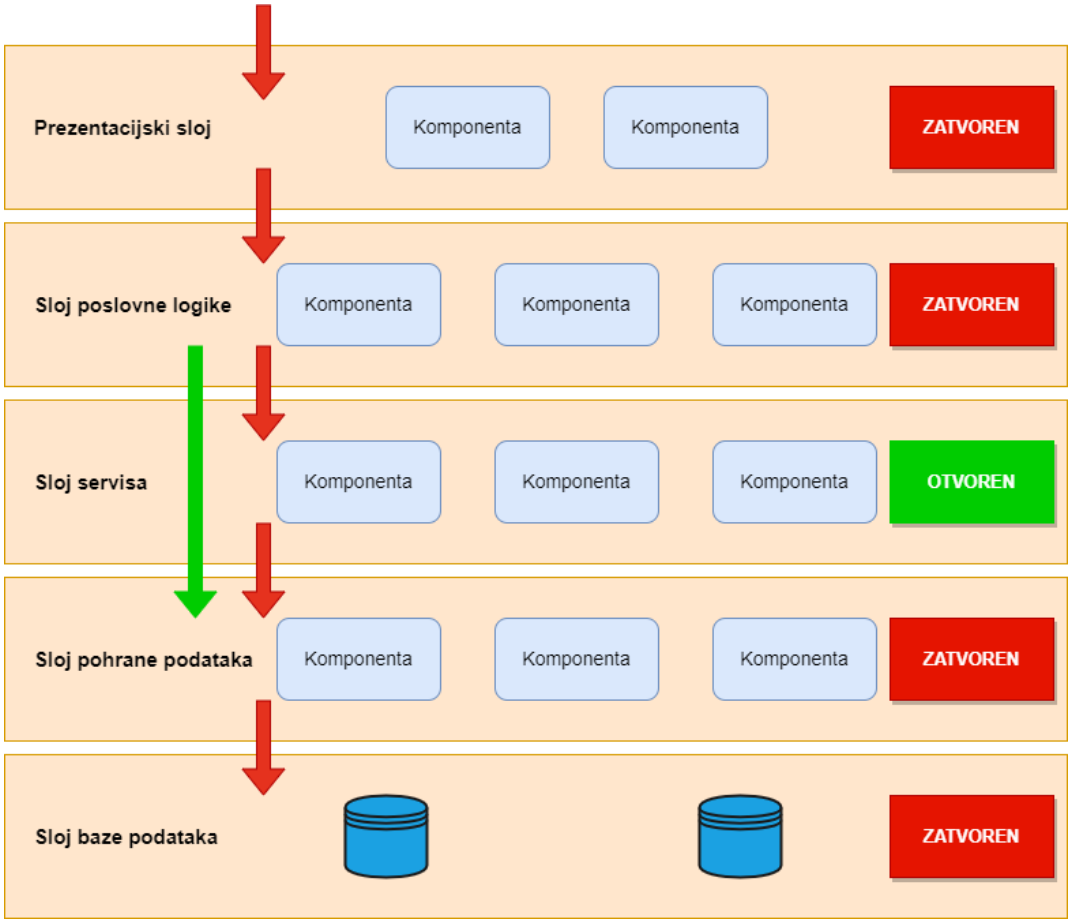
Slika 2.2: Topologija slojevite arhitekture

Prema konceptu izolacijskih slojeva *(eng. Slayers of isolation),* slojevi mogu biti otvoreni ili zatvoreni. Zatvoreni označavaju da pozivi metoda ne mogu preskakati slojeve tijekom obrade – zahtjev mora ići kroz sve slojeve na putu do traženog, za primjer pri dodavanju novih podataka u bazu, mora proći prezentacijski sloj, sloj poslovne logike, pohrane podataka i tek onda dolazi do sloja baze podataka. Slojevi trebaju biti zatvoreni kako bi se izbjegla čvrsta povezanost (*tight coupling*) slojeva, jer svi slojevi tako prate uspostavljene norme komuniciranja među slojevima i promjene u jednom sloju ne utječu na druge slojeve.

U suprotnom, promjena u jednom sloju može značajno utjecati i na slojeve koji preskaču taj sloj tijekom obrade zahtjeva. Ipak postoje slučajevi kada je poželjno ostaviti slojeve otvorenima. To se očituje kod dodavanja novih slojeva. Za primjer, usluge koje služe kao dijeljene komponente u aplikaciji, poput pomoćnih klasa manipulacije i pretvaranja podataka, klasa za zapisivanje i sl. se dodaju unutar aplikacije ispod sloja poslovne logike – poslovna logika delegira i preusmjerava zahtjeve na taj novi sloj, dok se sada između sloja pohrane i poslovne logike nalazi novi sloj, koji ne mora sudjelovati u obradi zahtjeva koji prolaze do slojeva ispod njega. U tom slučaju potrebno je omogućiti preskakanje određenih slojeva, čime se postižu otvoreni slojevi u aplikaciji, kako je prikazano slikom 2.3 gdje je zelenom strelicom označeno preskakanje otvorenog sloja koji omogućuje da ga se preskoči u obradi zahtjeva obzirom da ne sudjeluje nego bi samo

zahtjev prošao dalje bez obrade. Tada je dobro koristiti otvorene slojeve radi ubrzanja obrade zahtjeva.

Ovaj arhitekturni stil pogodan je pri razvoju manjih, jednostavnijih aplikacija, ili u slučaju kada se još ne zna specifično koji će arhitekturni stil biti korišten, a razvoj mora krenuti radi opsega i količine zahtjeva u aplikaciji. Kako je vrlo poznat programerima i jednostavan, predstavlja dobar izbor i s financijske strane. Ipak, porastom aplikacije značajno se smanjuje održavljivost, skalabilnost, agilnost i testabilnost aplikacije te bi radi toga za velike aplikacije bolje pristajali višemodularni arhitekturni stilovi.



Slika 2.3: Otvoreni i zatvoreni slojevi

2.3.3. Cjevovodna arhitektura

Poznata i kao arhitektura cijevi i filtara, ova arhitektura je jedna od ključnih arhitektura programske podrške. Nastala je iz potrebe za razdvajanjem funkcionalnosti aplikacija u zasebne dijelove te je razvijena tako da izvršava niz diskretnih koraka prema unaprijed definiranom redoslijedu, poput filtra koji svaki obavlja svoju zadaću, a pipe služe kao poveznica između filtara. Topologija cjevovodne arhitekture prikazana je na slici 2.4.

U ovom arhitekturnom obrascu, cijevi služe komunikaciji između različitih filtara. Cijevi su obično jednosmjerne i spajaju filtre od točke do točke - služe kao poveznica između dva filtra,

ne odašilje se na višestruke filtre i ne obrađuje podatke ni na koji način – kako ih zaprimi tako proslijedi dalje. Time se osigurava konzistentnost i određenost u obradi zahtjeva. Podržavaju bilo koje formate podataka, format ovisi o implementaciji, ali je poželjno da budu manje količine radi osiguravanja dobrih performansi.

Slika 2.4: Topologija cjevovodne arhitekture

Filtar je samostalna komponenta, neovisna o ostalim filtrima i uobičajeno bez stanja (*eng. stateless*). Svaki treba izvršavati isključivo jednu zadaću, dok za složenije zadatke se ulančava više filtara. Postoje četiri različita tipa filtra [12]:

- Izvor
- Transformator
- Tester
- Potrošač

Izvor je početna točka procesa, zaprima zahtjev i prosljeđuje idućem filtru. Transformatori su filtri koji obavljaju rad. Zaprimaju ulaz, izvrše operaciju nad zaprimljenim podacima te šalju dalje sljedećem filtru u nizu. Tester filtri testiraju zaprimljeni ulaz i ovisno o prolazu testa daju izlaz. Zadnji filtar u nizu je takozvani potrošač. On zaprima ulaz i daje konačan izlaz koji može biti zapis u bazu ili povrat podataka za prikaz korisniku.

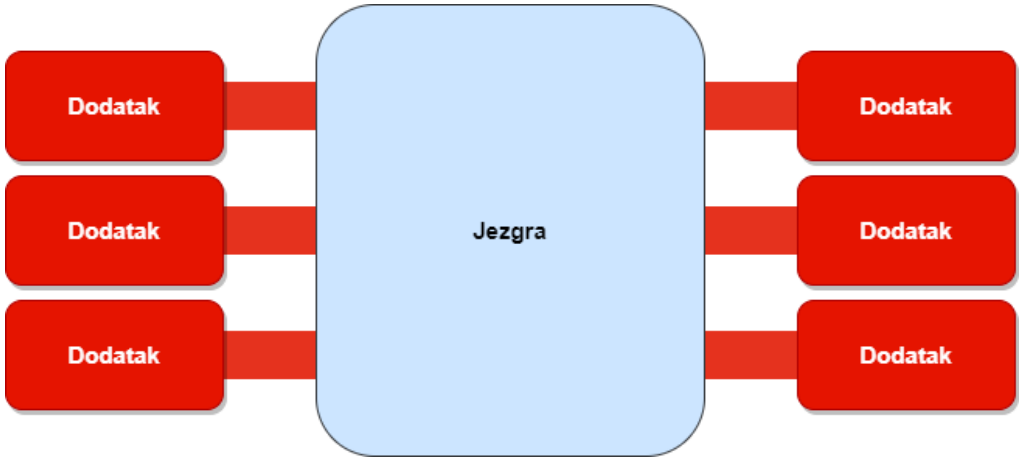
Slabom povezanosti filtara i jednostavnošću izvedbe pipa postiže se prednost ovog arhitekturnog stila, mogućnost ponovnog iskorištenja postojećih komponenti (filtara). Izmjena logike filtra je moguća bez utjecaja na druge filtre jer su međusobno neovisni, može ih se tretirati poput crnih kutija koje zaprimaju ulaz i nakon obrade daju određeni izlaz. Ovaj arhitekturni stil ipak nije povoljan za rad s velikim, opsežnim zadacima koji zahtijevaju puno koraka, iz razloga što se većim brojem filtriranja značajno smanjuju performanse i učinak ovog arhitekturnog stila i gubi se vrijeme na prijenos podataka putem pipa.

Primjenu pronalazi u alatima za obradu teksta, kompajlerima, operacijskim sustavima temeljenima na Unix-u kao što su za primjer Linux i MacOS.

2.3.4. Mikrojezgrena arhitektura

Mikrojezgrena ili arhitektura dodataka je široko rasprostranjena i prihvaćena arhitektura za dizajn programske podrške koja omogućava i koristi opcionalne dodatke. Sastoji se od jedne jezgre

koja sadrži osnovne funkcionalnosti za rad programske podrške i dodataka koji se mogu dodavati radi poboljšanja ili izmjene postojećih funkcionalnosti [13]. Topologija sustava prikazana je ispod na slici 2.5.



Slika 2.5: Topologija mikrojezgrene arhitekture

Kako je vidljivo sa slike, ova arhitektura se sastoji od dva tipa komponenti – jezgre i dodataka. Jezgra je jedinstvena središnja jedinica aplikacije koja pruža sve funkcionalnosti, ali u najosnovnijem obliku potrebnom za funkcioniranje, odnosno ispravan rad aplikacije. Za proširenje i poboljšanje osnovnih funkcionalnosti služe dodatci – moduli. Svaki modul bi trebao imati jednu odgovornost i odnositi se samo na jedan skup funkcionalnosti (domenu). Oni mogu pružati i dodatne funkcionalnosti proširujući tako mogućnosti aplikacije. Moduli dodani jezgri bi trebali biti neovisni jedni o drugima. Postoje slučajevi kada su moduli ovisni o nekom drugom modulu, no generalno pravilo je da su neovisni. Jezgra mora znati koji moduli su joj pridruženi te kako s njima raditi. Ispravna implementacija ovakve arhitekture omogućava lakše održavanje aplikacije i njeno testiranje

Ovisno o veličini aplikacije, postoje različiti načini implementacije mikrojezgrene arhitekture, pri čemu možemo imati jezgru s dizajnom poput slojevite arhitekture ili modularnog monolita. Također je moguće odvojiti prezentacijski sloj od jezgre, pa i njemu omogućiti dodatke, ostvarujući tako i mikrojezgreni dizajn prezentacijskog sloja.

Komunikacija između jezgre i dodanih modula je obično od točke do točke, korištenjem cijevi (*eng. pipe*), za komunikaciju. Razlikujemo dva tipa dodataka – one koji se prevode kad i cijeli sustav i one koji se prevode za vrijeme izvršavanja [13]. Moduli koji se prevode tokom izvršavanja mogu biti izmijenjeni tijekom rada aplikacije bez da se nužno naruši rad aplikacije. Također, moduli mogu biti dodani kao dijeljene biblioteke funkcija (JAR, DLL, Gem, …). Moduli ne moraju uvijek imati izravnu komunikaciju s jezgrom metodom od točke do točke, već ju je moguće izvesti tako da koriste mrežne protokole poput REST-a ili komunikacije porukama,

implementirajući tako module kao zasebne jedinice, ili čak mikrousluge. Iako to može pomoći povećanju skalabilnosti i djeluje poput raspodijeljene arhitekture, i ovakve izvedbe mikrojezgrene arhitekture spadaju pod monolitnu arhitekturu iz razloga što i dalje postoji jedna središnja jedinica upravljanja – jezgra, bez koje svi ti samostalni moduli ne bi imali svrhu.

Namijenjena je aplikacijama temeljenima na proizvodu (*eng. product-based*). U to spadaju razni alati za razvoj i implementaciju programske podrške poput razvojnog okruženja Eclipse, PMD, Jira, Jenkins. Također i preglednici koji omogućuju dodatke su primjer mikrojezgrenog dizajna – Chrome, Firefox. Ova arhitektura predstavlja dobar izbor i za programsku podršku koja treba ovisno o različitim faktorima primijeniti drugačiju poslovnu logiku, na primjer osiguranje koje mora slijediti zakone ovisno o tome u kojoj se državi koristi – za svaku državu se može napraviti poseban dodatak odnosno set pravila i drugačija implementacija standardnih poslovnih funkcionalnosti.

2.4. ARHITEKTURA MIKROUSLUGA

Arhitektura mikrousluga je iznimno popularan arhitekturni stil koji uzima sve više maha u dizajniranju programske podrške. Naziv je osmislio Martin Fowler zajedno s Jamesom Lewisom u blogu kojeg je nazvao upravo *mikrousluge* [14]. Radi raznih sličnosti, ali i unaprjeđenja, predstavlja nadogradnju na servisno-orijentiranu arhitekturu (SOA). Pripada raspodijeljenim arhitekturnim stilovima, iz razloga što u radu aplikacije s arhitekturom mikrousluga sudjeluje više pokrenutih procesa.

Ovaj pristup se oslanja na domenski pokretani dizajn (*eng. Domain Driven Development*). Aplikacija se sastoji od višestrukih usluga koje predstavljaju različite komponente, odnosno domene sustava, a jedan proces pokreće jednu uslugu. Svaka usluga pruža jedan dio poslovne logike cjelokupne aplikacije i predstavlja nezavisnu jedinicu. Kako su usluge nezavisne, lako se ostvaruju slabo povezane (*eng. loose coupled*) usluge što je jedna od glavnih prednosti arhitekture mikrousluga.

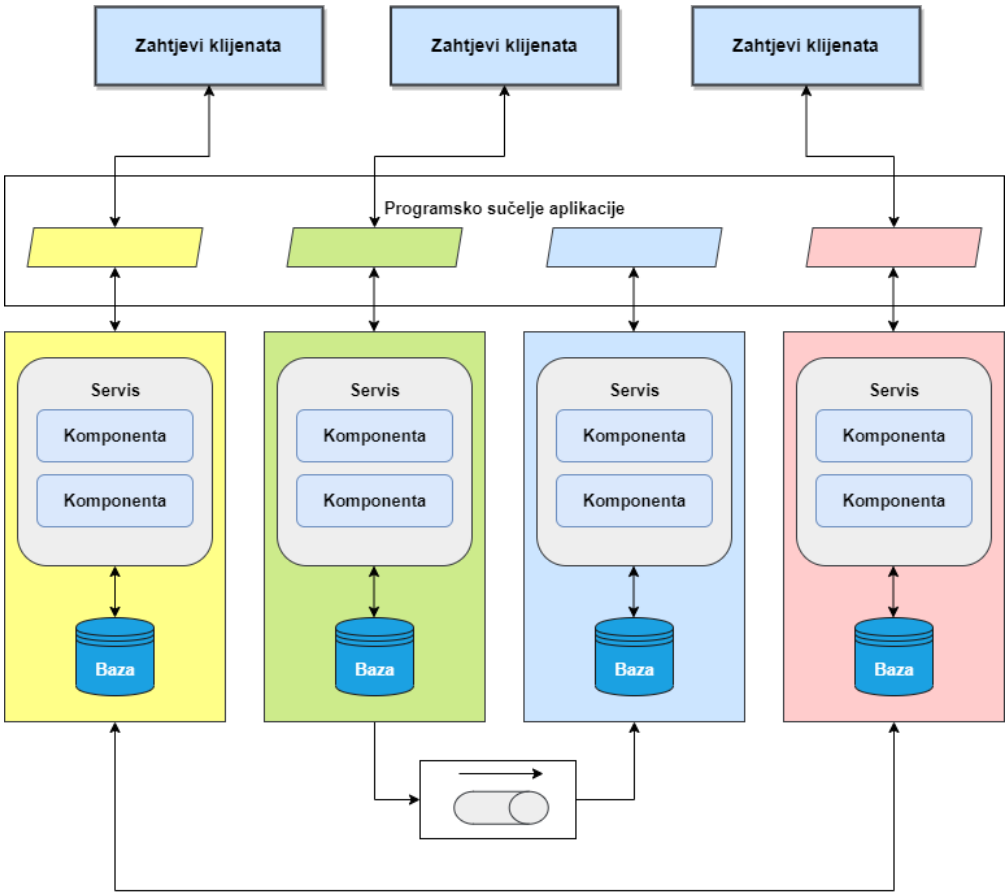
2.4.1. Struktura usluga u arhitekturi mikrousluga

Kako je već navedeno, mikrousluge su samostalne komponente. Svaka usluga ima svoju poslovnu domenu čije funkcionalnosti pokriva, čime se postiže veća granularnost u odnosu na SOA gdje može biti više domena sadržanih u jednoj usluzi. U slučaju da neka operacija zahtjeva interakciju s višestrukim domenama, usluge komuniciraju međusobno i tako je moguće povezati funkcionalnosti ili podatke iz različitih domena, što je prikazano slikom 2.6 u nastavku.

Radi ovakvog načina interakcije među uslugama omogućeno je razvijanje i implementacija usluga neovisno jednih o drugima. Komunikacija se odvija putem određenih protokola kao što su

REST i komuniciranje porukama putem različitih brokera (primjer RabbitMQ). Slijeđenje ovih protokola za komunikaciju donosi mogućnost neovisnosti tehnologije jedne usluge o tehnologiji druge usluge, potrebno je samo prilagoditi sučelje da slijedi protokole komunikacije. To otvara mogućnost da aplikacija bude napisana u više različitih programskih jezika – svaka usluga može biti napisana u jeziku koji najbolje odgovara potrebama usluge [14].

Tablice u bazi podataka su vezane za usluge i stoga trebaju također kao usluge biti neovisne jedne o drugima. Kada ne bi tako bilo, promjenom logike u jednoj usluzi i posljedično radi toga promjenom tablice u bazi mogao bi se narušiti princip neovisnosti između usluga. Stoga svaka usluga održava vlastitu bazu podataka. Kako jedna usluga uz sebe ima jednu bazu podataka, moguće je za svaku uslugu prilagoditi bazu podataka koja najbolje odgovara potrebama usluge. Tako jedna usluga može imati relacijsku bazu podataka, dok druga radi svojih zahtjeva može imati nerelacijsku(NoSQL) bazu podataka.



Slika 2.6: Topologija arhitekture mikrousluga

2.4.2. Komunikacija u arhitekturi mikrousluga

U arhitekturi mikrousluga komunikacija je bitan aspekt kako bi usluge mogle međudjelovati i biti razdvojene, a zatim kada trebaju podatke od druge usluge vrše komunikaciju. Komunikacija mikrousluga putem mreže je značajno sporija od poziva metoda između različitih

usluga/komponenata unutar monolitne arhitekture, ali je potrebna kako bi se one razdvojile, što predstavlja jedan od temelja ove arhitekture. Pronalazak najučinkovitije granularnosti i odvajanja odgovornosti na različite usluge predstavlja izazov za arhitekte i programere. Potrebno je uspostaviti dobru izolaciju podataka usluga i postići učinkovitu komunikaciju, ali svakako učiniti da ne bude suviše nepotrebne komunikacije i prevelike granularnosti među uslugama.

Komunikacija se dijeli na sinkronu i asinkronu. Sinkrona komunikacija zaustavlja izvršavanje koda dok se ne dobije odgovor od strane druge usluge s kojom se komunicira. Kako bi usluge mogle međusobno komunicirati, potrebno je ispuniti određene preduvjete. Obzirom da nema centraliziranog integracijskog čvora radi izbjegavanja povezanosti usluga, trebaju znati kako se pozivati međusobno. U tu svrhu se koriste na razini aplikacije dogovoreni protokoli, poput REST-a i poziva udaljenih procedura (*eng. Remote Procedure Calls – RPC*). Moraju podržavati heterogenost, jer je moguće da usluga s kojim komuniciraju nije napisana u istom jeziku. Stoga je potrebno koristiti protokole koji podržavaju korištene jezike na obje strane komunikacijskog kanala [15]. Korištenjem ovih smjernica omogućeno je međudjelovanje i suradnja usluga kroz komunikaciju.

Asinkrona komunikacija provodi se emitiranjem događaja i slanjem poruka, poput komunikacije u arhitekturi pokretanoj događajima. Asinkrona komunikacija omogućava komunikaciju na način koji ne blokira pozivatelja jer se ne čeka na odgovor, nakon poziva iz usluge koja inicira komunikaciju nastavlja se izvršavanje pozvane operacije. Ovo omogućuje rasterećenje sustava i manju spregu među uslugama. Ukoliko neka usluga nije dostupna kod ovakvog tipa komunikacije, poruka se može pohraniti u red za poruke i ponovno pokušati isporučiti nakon određenog vremena kako bi se operacija koju je inicirala pozivajuća usluga izvršila, značajno povećavajući pouzdanost i otpornost na pogreške u aplikaciji mikrousluga [16].

Kod odabira načina komunikacije usluga potrebno je imati na umu potrebe usluga. Sinkrona se koristi kada je za nastavak rada nužno dobiti odgovor druge usluge, odnosno trebamo trenutni odgovor. Asinkrona komunikacija je pak pogodna u situacijama kada nije potreban rezultat iz druge usluge nego se može emitirati događaj koji pokreće operaciju u drugoj usluzi bez čekanja na odgovor. Takva komunikacija je brža, ne blokira se proces pozivatelj koji inicira komunikaciju i uz korištenje redova poruka stvara veću otpornost na pogreške jer je moguće ponoviti slanje poruke ukoliko nije bilo uspješno. U mnogim slučajevima, usluge koriste oba od opisanih načina komunikacije ovisno o potrebi pojedinih funkcionalnosti.

2.4.3. Prednosti i nedostaci mikrousluga

Fleksibilnost pri odabiru tehnologija je značajna prednost mikrousluga. Također je jednostavna i izmjena tehnologije određene usluge ukoliko postoji potreba za izmjenom i nakon

što je već implementiran jer niti jedna promjena u usluzi ne zahvaća druge usluge. Izmjene koda i puštanje u rad novih inačica se mogu frekventnije izvoditi, jer za novu inačicu potrebno je ponovno pokretanje samo te usluge koji se mijenja, ne treba cijelu aplikaciju ponovno pokretati. Radi ovakve neovisnosti moguće je i korištenje višestrukih timova za razvoj programske podrške – posao se može među timovima podijeliti na usluge i tako značajno ubrzati razvoj. No, kada se treba implementirati operacija koja se prostire na više usluga, potrebna je koordinacija među timovima kako bi bilo uspješno izvedeno [17].

Nastanak pogreške u mikrouslugama ne narušava rad cijele aplikacije. Pogreška je izolirana na razini svoje usluge i radi neovisnosti među uslugama, ne znači nužno pad cijele aplikacije, druge usluge mogu obrađivati svoje zahtjeve dok je neka usluga u kvaru, za razliku od monolitnih sustava gdje pogreška zaustavlja rad cijele aplikacije.

Uvođenjem granularnosti, odnosno smanjenjem veličine usluga na manje domene, omogućeno je i lakše razumijevanje programerima koji se priključuju timu u radu na usluzi. I testiranje usluge pojedinačno je jednostavnije radi manjeg opsega funkcionalnosti i ovisnosti, no testiranje i otklanjanje pogrešaka cjelokupne aplikacije postaje značajno teže u arhitekturi mikrousluga radi integracijskog testiranja koje povezuje višestruke usluge tijekom testiranja.

Kada u monolitnoj arhitekturi nastane potreba za povećanjem resursa kao što su procesorska snaga ili memorija, potrebno je povećati resurse za cjelokupnu aplikaciju. Kod arhitekture mikrousluga to nije slučaj. Pojedine usluge mogu imati više prometa, biti češće korištene nego druge. U tom slučaju moguće je povećati resurse samo onih usluga koje to zahtijevaju, ciljano povećavajući resurse samo tamo gdje je to potrebno. Ovime je postignuta iznimno velika skalabilnost gdje svaku uslugu možemo skalirati prema potrebi bez obzira na druge usluge [18]. Nedostatak ovakvog pristupa je što nam je u početku potrebno više računalnih resursa radi toga što svaka usluga zahtjeva vlastitu memoriju, pokreće se u vlastitom kontejneru što zahtijeva zasebnu alokaciju resursa za pojedinu uslugu.

Upravljanje mikrouslugama je složen postupak. Potreban je koordiniran nadzor i upravljanje uslugama kako bi se otkrili nedostaci ili eventualne pogreške. Praćenje zapisa aplikacije (*logging*) je također otežano iz razloga što moramo imati zapise svake usluge koji su rasprostranjeni po memoriji, ne nalaze se na istom mjestu. Osiguranje svake ulazne točke u uslugu predstavlja puno veću površinu koju je potrebno osigurati. Obzirom da ima više manjih usluga, postoji i više pristupnih točaka. U nekim slučajevima i kada jedna usluga pozove drugu uslugu, potrebno je provjeriti sigurnost tog poziva. Tada se isti zahtjev može višestruko provjeravati na različitim uslugama. Ovo može značajno usporiti rad mikrousluga zbog trajanja sigurnosnih provjera, ali je nužno za napraviti kako bi se spriječili neovlašteni pristupi.

Implementacija cijelog sustava složenija je nego kod monolitne arhitekture. Monolit ima samo jedan proces koji se pokreće, stoga je potrebna konfiguracija, dodjela resursa i postavljanje okruženja samo za taj jedan proces. Kod arhitekture mikrousluga, svaka usluga ima zasebne te stavke – svakom je potrebno dodijeliti njegove resurse, konfigurirati i postaviti okruženje u kojemu će biti pokretan. Iako je pokretanje pojedinačne usluge jednostavnije jer pojedinačna usluga ima manje zahtjeve za resursima od monolitnog rješenja, kod podizanja cijele aplikacije moramo uskladiti sve usluge da rade zajedno, konfigurirati mrežu kojom usluge komuniciraju i pripremiti infrastrukturu za sve usluge. U tome uvelike pomaže DevOps metodologija, odnosno alati za kontinuiranu integraciju i implementaciju (CI/CD), automatizaciju procesa, koheziju procesa. Ispravna implementacija usluga da rade zajedno kao cjelina je iznimno bitna stavka arhitekture mikrousluga, stoga je osmišljeno nekoliko različitih pristupa implementaciji poput orkestracije, kontejnerizacije, bezposlužiteljskog pristupa (*eng. serverless*), kontinuirane integracije i implementacije i mnogih drugih, čime se bavi DevOps metodologija [19]. Sažeto, prednosti i nedostatke mikrousluga mogu biti prikazane:

- Prednosti
    - Brz razvoj, moguća lakša podjela na timove
    - Fleksibilnost u odabiru tehnologija
    - Neovisnost među uslugama, manje jedinice lakše za razumijevanje
    - Izolacija grešaka
    - Mogućnost čestih nadogradnji bez narušavanja rada sustava
    - Veća pouzdanost sustava
    - Visoka razina skalabilnosti
- Nedostatci
    - Zahtjeva koordinaciju timova pri pozivima koji prolaze kroz više usluga
    - Otežano praćenje nastalih pogrešaka, nadzor i monitoring
    - Teže integracijsko testiranje sustava
    - Komunikacija između usluga usporava rad
    - raspodijeljene transakcije i složenost upravljanja i sinkronizacije podataka u bazama
    - Inicijalno zahtjeva veće resurse
    - Kompleksna implementacija
    - Veća izloženost sustava prema van, potreban veći naglasak na sigurnost

2.5. MIGRIRANJE PROGRAMSKE PODRŠKE

Do sada su objašnjene karakteristike i razlike između monolitnih arhitektura i arhitekture mikrousluga. Kada imamo monolitnu arhitekturu, a primijetimo kako bolje odgovara arhitektura mikrousluga potrebno je izvršiti migriranje. Ovaj pothvat nije trivijalan i stoga sadrži korake koji trebaju biti pravilno izvedeni da bi migriranje bilo uspješno te maksimalno iskorištene dobre strane mikrousluga koje je cilj postići migriranjem.

2.5.1. Izazovi migriranja programske podrške

Migriranje može biti uzrokovano različitim razlozima tijekom životnog ciklusa programske podrške. Za glavni cilj ima unaprjeđenje određenih karakteristika koje su prednost u arhitekturi mikrousluga u odnosu na monolitnu. Pri tome se susrećemo s brojnim izazovima koji postaju prisutni u aplikaciji tek kada započne migriranje – promjena arhitekture uzrokuje promjene u načinu rada sustava, ali i same organizacije koja provodi migriranje.

Jedan od mogućih razloga migriranja je rast količine koda programske podrške iznad razine koju se može efikasno održavati kao dio monolitnog rješenja. Tada mikrousluge mogu biti razmatrane kao dobro rješenje. Kod takvih monolitnih rješenja koja zbog svoje veličine zahtijevaju migriranje nailazimo na mnoge međuovisnosti komponenata (modula) unutar aplikacije. One su statički povezane i imaju određene ovisnosti jedni o drugima. Za uspješno migriranje potrebno je detaljno analizirati module i dokumentirati njihove ovisnosti. Kada razdvajamo module nije dobra praksa sve odmah razdvojiti. Puno bolje rješenje je uz pomoć rezultata analize ovisnosti pronaći module koji sadrže najmanje ovisnosti i odvajati module slijedno prema tome. Ovim načinom uvelike se pojednostavljuje proces migriranja obzirom da se iz monolitne arhitekture izdvajaju pojedinačni moduli i inkrementalno se monolit razdvaja na mikrousluge. Ovim pristupom je značajno jednostavnije provjeriti ispravnost migriranja i postupno povećavati broj izdvojenih modula dok ne razdvojimo sve module u mikrousluge.

Dobra praksa prilikom migriranja je zadržati postojeći kod sve dok nismo sigurni kako je uspješno odvojen u zasebnu uslugu. Cilj je kopiranje funkcionalnosti danog modula u novu uslugu, zadržavajući funkcionalnosti koje je imao, no ukoliko iz bilo kojeg razloga nastane pogreška, moguće je lako vratiti stanje aplikacije kakvo je bilo prije neispravnog pokušaja migriranja modula. Ovakvim pristupom ostavljamo višestruke opcije za nastavak migriranja, a modul i usluga stvorena na osnovu tog modula tijekom testiranja mogu čak raditi paralelno. Tek nakon što nova usluga prođe sva testiranja i migriranje uspješno izvedeno, moguće je sigurno brisanje modula iz monolitnog rješenja.

Komunikacija u mikrouslugama je bitno drugačija nego u monolitu. Moduli monolita su statički povezani i mogu koristiti određene funkcionalnosti izravno referenciranjem na drugi modul. Kod mikrousluga mora biti uspostavljena komunikacija na ranije opisane načine, pomoću mrežnih protokola ili slanjem poruka. Izazov ovoga je ispravno povezivanje svih točaka komunikacije – svaka točka na usluzi preko koje ona komunicira s ostalim uslugama mora biti ispravno povezana. Kod aplikacija s mnoštvom modula to znači mnogo točaka komunikacije, uspostavljanje svih komunikacijskih kanala je zahtjevan posao za koji treba biti osiguran ispravan kod i uspostavljena odgovarajuća infrastruktura ne zanemarujući zablude raspodijeljenog računarstva opisane ranije u odjeljku 2.2.3.

Sljedeći bitan aspekt koji treba imati u vidu tijekom migriranja je upravljanje transakcijama baze podataka. Kod baza podataka postoje dva načina implementacije – jedna baza po usluzi i dijeljena baza podataka. Dijeljena baza nije dobro rješenje iz razloga što je tada baza točka povezivanja usluga i onemogućava se konfiguriranje baze i skalabilnost ovisno o potrebama pojedine usluge. Stoga je kod mikrousluga preferirana jedna baza po usluzi. Tako usluge koji žele pristupiti podacima iz baze druge usluge moraju komunicirati i proći sigurnosne provjere usluge kojem pristupaju. Kao rezultat toga dobivamo manju suspregnutost usluga i bolju sigurnost i integritet podataka. Negativna strana ove implementacije su transakcije u razdvojenim bazama. Strani ključevi mogu referencirati samo one atribute koji se nalaze u istoj bazi podataka. Kako to nije uvijek slučaj kod baza podataka u mikrouslugama, nemamo mogućnost transakcija koje osiguravaju konzistentnost podataka i ako jedan zapis u tablicu ne uspije, sve promjene koje je izazvala neuspjela operacija se poništavaju.

Pristupi za osiguranje ispravnosti u raspodijeljenim bazama kod mikrousluga su eventualna konzistentnost, kompenziranje transakcija i raspodijeljene transakcije. Eventualna konzistentnost znači da neuspjele operacije se spremaju u spremnik, poput reda, i pokušaju izvesti ponovno nakon određenog vremena. Kod tog pristupa neko vrijeme sustav neće biti usklađen, ali će u jednom periodu operacija biti izvršena u potpunosti i sustav će se uskladiti. Ovaj način je prigodan kada nemamo čestih promjena istih podataka u bazi, a donosi veliku skalabilnost jer usluge koje koriste ovakav pristup nemaju velikog doticaja jedna s drugom niti provjeravaju uspješnost promjene u bazi druge usluge. Druga opcija je kompenziranje transakcija. Ovaj pristup se temelji na provjerama uspješnosti promjena u drugim uslugama. Usluga u kojoj se inicira promjena komunicira s drugim uslugama koje trebaju mijenjati podatke u svojim bazama. Druge usluge odgovaraju s informacijom o uspješnosti operacije te ako je jedna operacija u skupu bila neuspješna, okida se mehanizam koji pokreće brisanje svih uspješnih promjena iniciranih tim pozivom prema aplikaciji. Iako je brže i pouzdanije od eventualne konzistentnosti kod baza koje

često imaju promjene podataka, otežava praćenje promjena. Alternativno rješenje kompenziranju su raspodijeljene transakcije. Raspodijeljene transakcije koriste transakcijskog menadžera da upravlja transakcijama u višestrukim bazama. Koristi se dvofazna potvrda za osiguranje ispravnosti promjena podataka. Ovaj proces umanjuje mogućnost skalabilnosti usluga, čime se narušava jedna od izraženijih karakteristika mikrousluga.

Kako svaki pristup ima svojih pozitivnih i negativnih strana, niti jedan nije univerzalno dobro rješenje. Preporuča se korištenje eventualne konzistentnosti gdje god je to moguće, iz razloga što ne ograničava skalabilnost usluga. Kod mnogih situacija gdje eventualna konzistentnost nije prihvatljiva, preporučeno je i razmisliti je li dobro razdvajanje tih funkcionalnosti u različite usluge s različitim bazama [20].

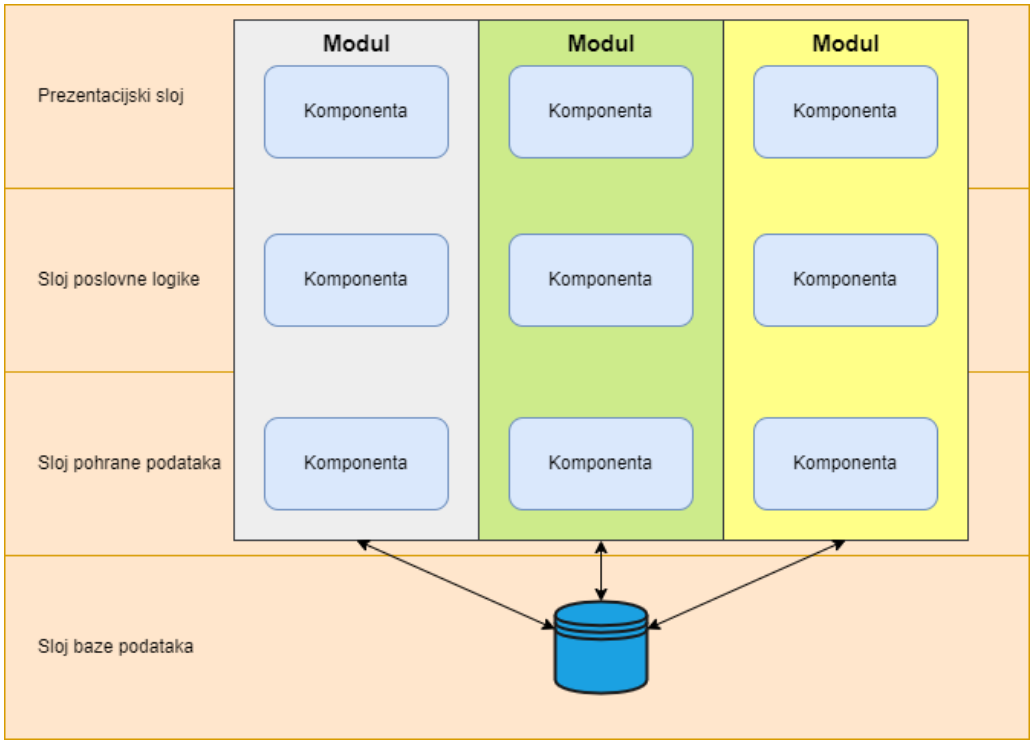
2.5.2. Odvajanje domena i uspostavljanje granica

Kako je navedeno u prethodnom odjeljku, za razdvajanje monolita na mikrousluge, potrebno je identificirati granice modula. Kod uobičajenog razvoja sa slojevitom arhitekturom, različiti moduli nisu grupirani prema logičkim domenama. Aplikacija je podijeljena na slojeve te su tako i grupirani dijelovi modula prema slojevima – klase se grupiraju u grupe poput prezentacijskog sloja, sloja poslovne logike, pohrane podataka. Naglašeno je razdvajanje odgovornosti prema slojevima u aplikaciji. Ovakav raspored klasa u aplikaciji nije pogodan za migriranje u mikrousluge, štoviše čak i otežava migriranje na mikrousluge. Klase koje su dio iste domene, nisu grupirane zajedno, stoga je teže izdvojiti zasebne logičke domene i jasno postaviti granice između domena. Također njihova preraspodjela zahtijeva i dodatnu promjenu ovisnosti u aplikaciji, gdje je na nekim mjestima potrebno dodati ovisnosti prema drugim klasama i bibliotekama, dok se negdje mogu ukloniti.

Pristupačniji model za mogućnost migriranja je modularni monolit (slika 2.7) i upotreba domenski pokretanog dizajna (*DDD - eng. Domain Driven Design*). Domenski pokretani dizajn stavlja naglasak na strukturiranje aplikacije prema domenama koje čine poslovnu logiku aplikacije. Za ovakav pristup potrebno je dobro vladati materijom i poznavati kako u stvarnosti funkcionira sustav koji želimo implementirati kroz programsku podršku. Karakteristična je suradnja sa stručnjacima iz područja poslovne logike aplikacije, kako bi se što vjerodostojnije stvorio sustav nalik na stvarni poslovni model. Praćenjem ovakvog dizajna možemo jasno povući granice konteksta (*eng. Boundary context*), koji omogućava da razdvojimo različite domene unutar sustava kako bi se mogli iz njih stvoriti mikrousluge [21]. Granice konteksta su mjesta gdje u sustavu prestaje odgovornost jedne domene, odnosno usluge, i one uokviruju domene u zasebne jedinice djelovanja unutar sustava. Unutar svakog konteksta sadržani su svi elementi, u vidu klasa, potrebni za implementaciju poslovne domene u sustav programske podrške, dok se na granicama

definira sučelje za komunikaciju s ostalim domenama. Ovaj koncept pomaže u organizaciji kompleksnih domena i razdvajanju odgovornosti čime se omogućava modularnost i skalabilnost te bolje upravljanje sustavom – karakteristike koje pronalazimo i kod mikrousluga [22].

Modularni monolit je vrsta monolitne aplikacije u kojoj su dijelovi grupirani prema logičkim domenama. Takva struktura ima jasno odvojene granice između različitih odgovornosti unutar aplikacije i mogu se lakše vidjeti granice modula koji čine aplikaciju. Unatoč grupiranosti modula i razdvajanjem odgovornosti poput izvedbe u arhitekturi mikrousluga, i dalje je monolitna aplikacija radi toga što je pokretana jednim procesom. Ipak, uz ovakvu implementaciju monolitnog rješenja, imamo dobre preduvjete za jednostavnije migriranje na mikrousluge. Moduli su razdijeljeni prema odgovornostima i lakše je izdvajati ih iz strukture monolitne aplikacije za postupno odvajanje kakvo je opisano ranije.

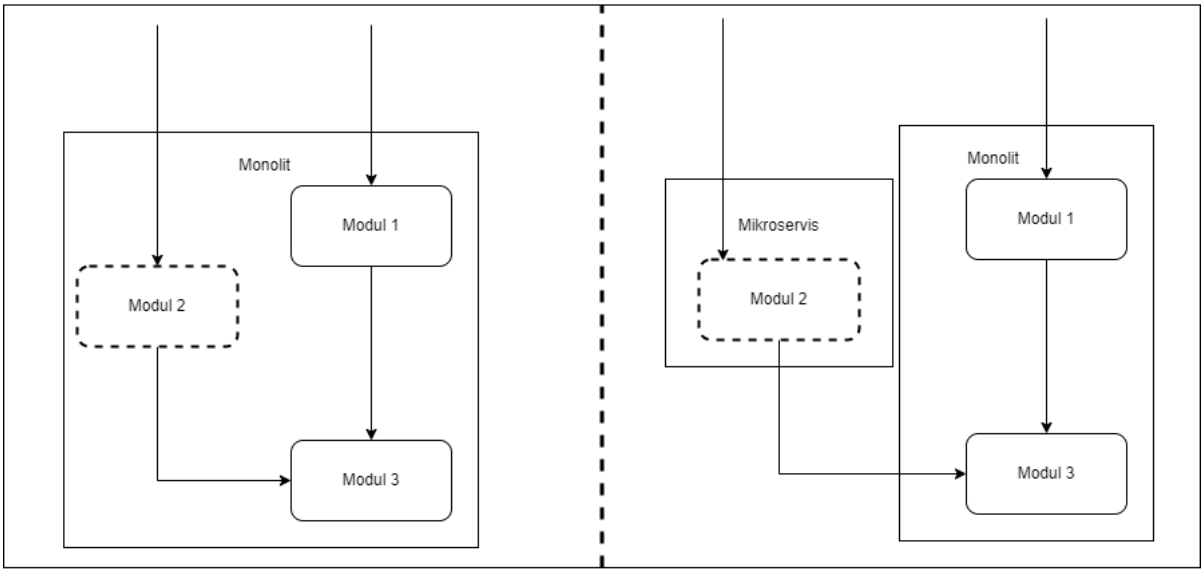


Slika 2.7: Topologija modularnog monolita

2.5.3. Obrasci migriranja monolitne u arhitekturu mikrousluga

Kako je migriranje iz monolitne arhitekture u arhitekturu mikrousluga iznimno popularna i česta praksa kod postojećih aplikacija, osmišljeno je nekoliko različitih obrazaca kako to može biti izvršeno. Pogodno ih je koristiti ovisno o strukturi aplikacije, ne postoji univerzalno dobro rješenje niti pristup koji odgovara svakoj aplikaciji. Zbog toga je potrebno razumjeti i obrasce i strukturu aplikacije u monolitnoj izvedbi. U nastavku slijedi pregled najčešćih obrazaca migriranja monolita u mikrousluge.

Prvi obrazac je obrazac gušenja monolita (*eng. strangler fig/strangulation*) prikazan slikom 2.8. Nazvan metaforički prema vrsti biljaka koje rastu oko drveta i svojim daljnjim razvojem omotavaju u potpunosti dok domaćin ne umre ostavljajući na životu samostalnu nametničku biljku. Na sličnom principu je zasnovan i ovaj obrazac, a sastoji se od tri glavna koraka. Prvi korak je pronalazak dijela koji možemo izdvojiti u zasebnu mikrouslugu i definiranje granica odgovornosti za tu uslugu. Nakon toga slijedi implementacija odabrane usluge (omotavanje), u kojoj se kreira usluga jednakih mogućnosti kao što ih sadrži u monolitu. U ovoj fazi još uvijek korisnik pristupa sučelju monolita, no poziv se preusmjerava na novu uslugu – tehnički još nije odvojen od domaćina (monolita). Tijekom ove faze moguće je testiranje i usporedba rezultata modula u monolitu i nove usluge, čime se podržava inkrementalno migriranje, postupnim prebacivanjem malih dijelova aplikacije u drugačiju arhitekturu koje je lako vratiti u slučaju pogreške [23]. Kada je provjereno i testirano da novi zadovoljava sve zahtjeve poput vlastitog dijela unutar monolita, možemo sigurno izvršiti i zadnju fazu – preusmjeravanje poziva izravno na sučelje nove mikrousluge, čime se funkcionalnost jednog modula iz monolita u potpunosti prebacuje na mikrouslugu, gušeći tako replicirani dio iz monolitne aplikacije. Po završetku se dio funkcionalnosti koji je repliciran može ukloniti iz monolita.



Slika 2.8: Prikaz obrasca *gušenja*

Kako bi pravilno koristili ovaj obrazac, potrebno je znati koje dijelove, i kojim redoslijedom odvajati iz monolitne aplikacije. Obzirom da se temelji na izdvajanju pristupnih točaka, odnosno sučelja aplikacije, najpogodnije je odvajati module koji su izravno povezani sa sučeljem aplikacije, kao na primjer modul 2 (sl. 2.8). Na taj način možemo jednostavno izdvojiti uslugu i njezine pozive prema monolitu, za razliku od modula 3 (sl. 2.8), koji ima višestruke pozive iz drugih usluga, te ih ne možemo preusmjeravati kao pozive koji dolaze izvan sustava. Takvo

izdvajanje iz sustava zahtijevalo bi izmjene i usluga koje pozivaju uslugu koje se nalazi dublje u sustavu čime se otežava migriranje jer je potrebno i u drugim uslugama prilagoditi komunikaciju prema toj koja je izdvojena.

Obrazac koji podržava migriranje dubljih dijelova aplikacije je obrazac zvan grananje putem apstrakcije (*eng. Branch by abstraction*). Proces migriranja ovim obrascem sastoji se od pet koraka:

- Kreiranje apstrakcijskog sloja za željenu funkcionalnost
- Preusmjeravanje klijenata na apstrakciju umjesto funkcionalnosti unutar monolita
- Implementacija funkcionalnosti na apstrakcijskom sloju
- Preusmjeravanje funkcionalnosti da koristi novu implementaciju
- Zamjena stare funkcionalnosti novom

Prvi korak je dakle stvaranje apstraktne klase ili sučelja koje definira interakcije s drugim dijelovima aplikacije. Zatim drugi dijelovi koji su koristili funkcionalnost (modul) koju želimo migrirati se mijenjaju da koriste apstrakcijski sloj i metode koje on nudi umjesto postojećeg rješenja unutar monolita. Zatim se kreira nova implementacija na vanjskoj usluzi dok je stara još uvijek aktivna i u uporabi. Kada je nova usluga dovršena, mijenjamo sustav tako da koristi tu novu uslugu pozivanjem iz prethodne. Jednom kada se testiranjem i korištenjem utvrdi kako nova usluga sadrži sve jednake funkcionalnosti kao i modul kojeg mijenja, možemo ukloniti funkcionalnost iz monolita čime se završava postupak migriranja tog dijela koda obrascem grananja putem apstrakcije.

Obrazac paralelnog pokretanja je još jedan obrazac migriranja sustava, a temelji se na usporedbi rezultata dvaju sustava. Prvi sustav je stari sustav kojeg zamjenjujemo (primjer modul u monolitu), a drugi je novi (mikrousluga proizašla iz modula monolita). Kada je novi sustav spreman za rad, paralelno pokrećemo i održavamo oba sustava. Svi zahtjevi koji dolaze na sustav se obrađuju u oba te se njihovi rezultati spremaju za usporedbu. Stari sustav koristimo kao izvor istine i njegove rezultate smatramo ispravnima, a novi kontroliramo usporedbom s izvorom istine. Kada se pokaže da novi sustav radi jednako kao stari, možemo sigurno prijeći na novi sustav i ukloniti stari. Prilikom migriranja iz monolita u mikrousluge izvodi se slično obrascu gušenja, stvarajući novu uslugu jednakih funkcionalnosti kao unutar modula monolita, ali neko vrijeme ova dva sustava rade paralelno dok se novi ne verificira kao ispravan.

Korištenjem ovih obrazaca moguće je efikasno migrirati postojeću programsku podršku s monolitnom arhitekturom na arhitekturu mikrousluga, što će biti prikazano na praktičnom primjeru u nastavku rada.

3. ZAHTJEVI NA APLIKACIJU, ARHITEKTURA I POSTUPAK MIGRIRANJA

Unutar ovog poglavlja bit će navedeni i opisani zahtjevi na aplikaciju, funkcionalni i nefunkcionalni, opisana arhitektura aplikacije kao monolitnog rješenja te plan migriranja na arhitekturu mikrousluga putem teoretski opisanih obrazaca u prethodnom poglavlju.

3.1. FUNKCIONALNI ZAHTJEVI NA APLIKACIJU

Aplikacija na kojoj će biti izvršeno migriranje je aplikacija za menadžment sportskog kluba. Razvojem sporta i popratne infrastrukture razvijaju se mnoga rješenja izvan sportskih terena koja neizravno pomažu sportskim klubovima u ostvarenju boljih rezultata u natjecanjima u kojima se natječu. Stoga ova aplikacija služi za objedinjenje informacija i praćenje raznih aspekata rada sportskog kluba na jednom mjestu s ciljem lakšeg planiranja i ostvarenja boljih strategija za napredak kluba.

Unatoč tome što je osnovni razlog postojanja sportskih klubova i društava sportsko natjecanje s drugim momčadima, razvojem i sve većim ulaganjima u sport profesionalni klubovi su prerasli okvire djelovanja od isključivog natjecanja na sportskim borilištima i terenima. U modernom sportu klubovi imaju veliku infrastrukturu iza same momčadi koja je zadužena za uspješno funkcioniranje kluba u raznim područjima kako bi se sportašima omogućili bolji uvjeti, a samim time i klubu bolji rezultati [24]. Djelovanje kluba tako uključuje i brigu o igračima, praćenje njihovog zdravstvenog stanja te čak i zapošljavanje medicinskog osoblja isključivo za potrebe kluba. Razni sportski analitičari mogu biti zaposleni u klubu za razvijanje strategija i pronalaženje taktika koje bi osigurale bolje rezultate klubu. Za uspješno poslovanje kluba potrebni su ekonomski stručnjaci čija je uloga rukovanje financijama kluba i izvršavanje obveza koje dospijevaju klubu te vođenje financijskog dijela poslovanja kluba. Porastom odgovornosti i utjecaja klubova, veća se pažnja pridodaje i ispunjavanju zakonskih obveza te pravilnika i statuta sportskih saveza. Jedan od aspekata tog dijela poslovanja je i postizanje dogovora s igračima i popratnim osobljem koje se ostvaruje ugovorima između njih i kluba. Radi toga je potrebno imati i evidenciju o ugovorima kako rukovodeći članovi kluba bi imali uvid u detalje ugovora i mogli pravovremeno reagirati s produljenjem ili otkazivanjem ugovora sa strankama. Na kraju, trener i stručno osoblje trebaju mogućnost praćenja spremnosti i forme igrača, njihovog zdravstvenog stanja i dostupnosti radi planiranja strategija utakmice te koje igrače mogu koristiti na utakmici.

Prema gore navedenim potrebama sportskog kluba proizlaze sljedeći funkcionalni zahtjevi koje aplikacija za menadžment sportskog kluba mora ispuniti:

- Upravljanje entitetom igrača; kreiranje, dohvaćanje i uklanjanje podataka

- Upravljanje entitetom člana osoblja; kreiranje, dohvaćanje i uklanjanje podataka
- Praćenje zdravstvenog stanja igrača
- Evidencija ozljeda igrača; dijagnoza, praćenje i oporavak
- Evidencija liječničkih pregleda i tretmana igrača
- Dohvaćanje informacija o oporavku igraču od ozljede, praćenje tretiranja ozljede kroz skup liječničkih pregleda
- Dodavanje ugovora igrača i članova osoblja
- Praćenje stanja ugovora
- Evidencija financijskih transakcija u klubu
- Stvaranje izvještaja financijskih tokova u klubu
- Dohvaćanje dokumenata ugovora
- Dohvaćanje medicinskih dijagnoza ozljeda igrača
- Dohvaćanje potvrda izvršenih transakcija

3.2. NEFUNKCIONALNI ZAHTJEVI NA APLIKACIJU

Nefunkcionalni zahtjevi na aplikaciju odnose se na to kako aplikacija radi. To je skup zahtjeva koji služe osiguravanju kvalitete programskog rješenja aplikacije i odnose se na opisivanje kako aplikacija treba raditi. Oni opisuju kako se aplikacija treba ponašati u određenim kritičnim slučajevima i ukazuju na kvalitetu sustava, osiguravajući da sustav bude dobar s gledišta mogućnosti korištenja aplikacije [25]. Neki od zahtjeva koji spadaju u tu kategoriju su:

- Skalabilnost
- Sigurnost
- Pouzdanost
- Upotrebljivost
- Sposobnost održavanja
- Usklađenost sa zakonskim propisima

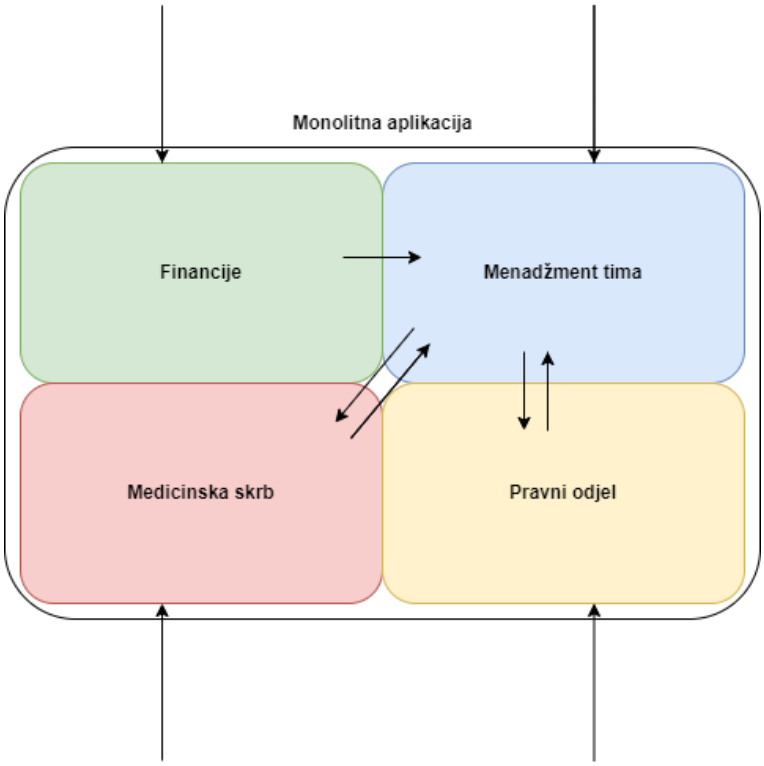
Nepridržavanjem ispunjenja nefunkcionalnih zahtjeva moguće su mnoge štetne posljedice poput nezadovoljstva korisnika i klijenata, nekonzistentna programska podrška, rast troškova održavanja aplikacije radi učestalijih popravaka. Često su ovi zahtjevi međusobno ovisni, što znači da promjenama u pogledu jednog nefunkcionalnog zahtjeva se postiže promjena i kod drugog. Primjer toga bi bilo dodavanje dodatne razine sigurnosti koje zahtjeva nove provjere s duljim trajanjem što izravno narušava performanse i brzine odziva u aplikaciji. Stoga je potrebno dobro razmotriti potrebne nefunkcionalne zahtjeve i razinu implementacije svakog od njih [26].

Unutar ove aplikacije omogućena je skalabilnost. U slučaju velikog broja zahtjeva koji bi narušili ponašanje i performanse aplikacije, moguće je dodati dodatnu računalnu infrastrukturu i tako postići skaliranje sustava. Ovo je posebice izraženo u arhitekturi mikrousluga gdje je moguće skaliranje svake usluge zasebno što omogućava efektivnije skaliranje gdje se mogu postići željeni rezultati uz manje troškove nadopunjavanja infrastrukture. Obzirom da sustav ne rukuje velikim količinama podataka po zahtjevu, očekuju se dobre performanse i mala vremena odziva. Pri normalnim uvjetima rada očekivano vrijeme odziva je ispod jedne sekunde, uzimajući u obzir postojanu dobru mrežnu povezanost. Na ovakvu razinu performansi sustava svakako utječe i sigurnost u sustavu. Obzirom da ova aplikacija još uvijek nije dostupna klijentima na korištenje, trenutno nije implementirana sigurnost čime je moguće brže izvršenje zahtjeva. U slučaju da je potrebno dodati sigurnosne mehanizme, postoji mogućnost implementiranja sigurnosti koju nudi *Spring Boot*. Putem te sigurnosti omogućeno je jednostavno dodavanje zaštite podataka i ograničavanja pristupa podacima različitim korisnicima putem autentikacije i korisničkih računa, kao i postavljanje različitih razina korisničkih računa s ovlastima za različite korisnike. Ukoliko se tijekom rada aplikacije radi lošeg zahtjeva dogodi pogreška u radu aplikacije, aplikacija neće stati s radom – podiže se iznimka, a korisniku se vraća obavijest o pogrešci dok aplikacija nastavlja s radom. Ovime se očituje pouzdanost aplikacije koja ne prestaje raditi radi određenih pogrešaka radi nepravilnog korištenja. Razlozi zbog kojih bi mogla prestati ispravno raditi su nedostatak mrežne veze ili kvar na računalnoj infrastrukturi na kojoj se aplikacija pokreće. Omogućuje veliku kompatibilnost s drugim aplikacijama korisničkih strana, što je postignuto korištenjem komunikacije putem HTTP (*eng. Hypertext Transfer Protocol*) protokola koji je najrašireniji protokol mrežne komunikacije, te uz korištenje JSON (*eng. JavaScript Object Notation*) objekata za prijenos podataka putem komunikacije s ranije navedenim protokolom.

Aplikacija je jednostavna za korištenje, pružajući funkcionalnosti za rad s podacima na jednostavan način kroz pristupne točke programskog sučelja. Nužno je samo koristiti unaprijed definirane zahtjeve i parametre u zahtjevima potrebne za izvršenje tražene operacije.

3.3. ARHITEKTURA APLIKACIJE KAO MONOLITNOG RJEŠENJA

Arhitektura aplikacije kao monolita izvršena je slojevitom arhitekturom korištenjem 4 sloja – prezentacijski sloj, sloj poslovne logike, sloj pohrane podataka i sloj baze podataka, uz korištenje dizajna modularnog monolita, gdje je aplikacija podijeljena na module koji jasno stvaraju granice odgovornosti između različitih skupova funkcionalnosti koje aplikacija pruža, kako je prikazano na slici 3.1.

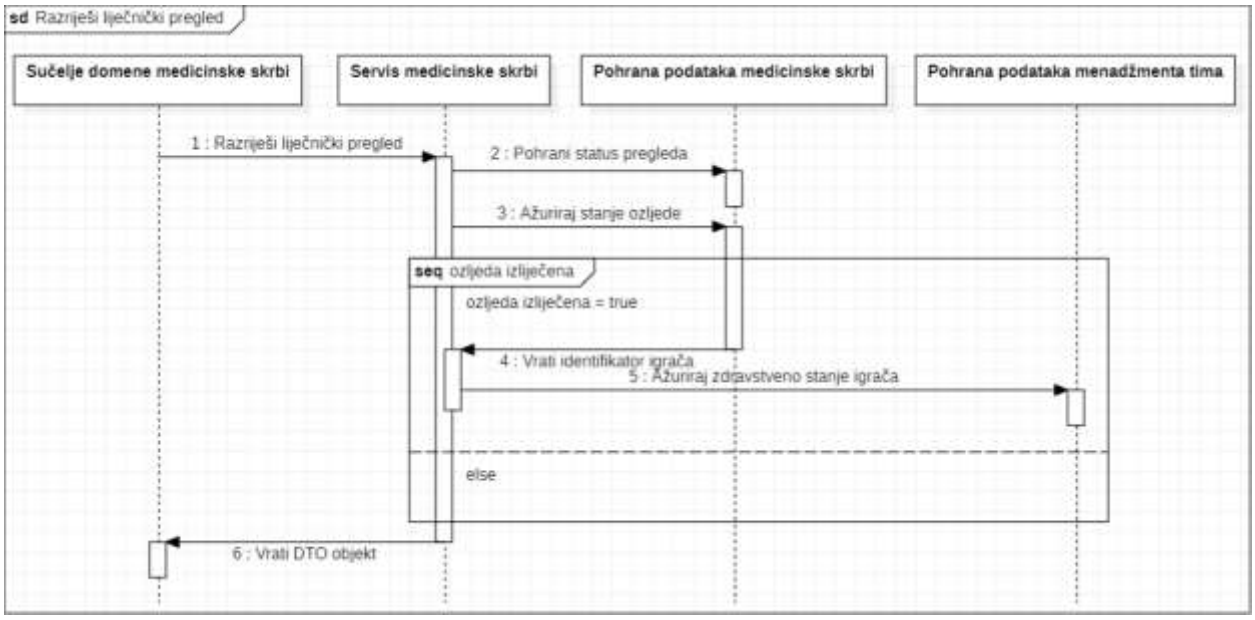


Slika 3.1: Prikaz modula aplikacije s pristupnim točkama sučelja

Kao što je vidljivo na slici, moduli zastupljeni unutar aplikacije su:

- Menadžment tima – praćenje igrača i trenerskog osoblja
- Medicinska skrb – evidencija i praćenje ozljeda te liječničkih pregleda igrača
- Financije – praćenje financijskih tokova
- Pravni odjel – održavanje ugovora i praćenje statusa

U aplikaciji se koristi jedna relacijska baza podataka, PostgreSQL, s višestrukim tablicama. Moduli komuniciraju s drugim modulima na različitim slojevima kada zaprime poziv koji uključuje višestruke domene, za primjer prikazan na slici 3.2, prilikom liječničkog pregleda ozljeda se označava izliječenom, što treba promijeniti i stanje igrača iz ozlijeđenog u ponovno spremnog za igru, odnosno aktivnog. Tada sloj poslovne logike unutar modula za medicinsku skrb pristupa i sloju pohrane podataka iz domene medicinske skrbi kako bi dalje pohranio u bazi podataka informacije o pregledu i ažuriranje stanja ozljede, ali također pristupa i sloju pohrane podataka iz domene menadžmenta tima kako bi se promijenilo stanje igrača u bazi podataka da se prikaže kako je ponovno dostupan, odnosno spreman za nastupiti. Naposljetku, vraća se DTO (*eng. Data Transfer Object*) – objekt za prijenos podataka koji služi za prijenos podataka između slojeva aplikacije.



Slika 3.2: Sekvencijalni dijagram za slučaj razrješenja liječničkog pregleda u aplikaciji

Prilikom korištenja ovakvih metoda koje uzrokuju promjene podataka u više tablica iste baze, sinkronizacija podataka se osigurava transakcijama baze podataka, što označava da ukoliko jedan od upisa u bazu nije ispravan, neće se izvršiti niti jedna promjena u tablicama koje su modificirane iz metode koja je inicirala te promjene. Taj mehanizam koji je moguć u monolitnim aplikacijama nam osigurava dosljednost i ispravnost podataka na vrlo jednostavan način.

Radi korištenja jedne baze podataka, omogućeno je i referenciranje entiteta putem primarnih i stranih ključeva čime se osigurava dodatna razina dosljednosti između zapisa u tablicama baze. Na ovaj način nije moguće da jedna tablica sadrži zapis u kojem stranim ključem referencira zapis iz druge tablice koji ne sadrži taj ključ. Korištenjem programskog okvira *Spring boot*, to je omogućeno korištenjem anotacija u programskom kodu koje označavaju veze između različitih tablica baza podataka poput jedan-na-jedan, više-na-jedan i drugih. Primjer takvog referenciranja dan je kodom na slikama 3.3 i 3.4 u nastavku.

Struktura datotečnog sustava aplikacije prati načela razvoja programske podrške slojevite arhitekture, odvajajući tako u različite mape klase koje se odnose na različite slojeve aplikacije. Dodatno, radi domenski pokretanog dizajna unutar tih mapa koje označavaju slojeve, klase su podijeljene prema domeni, kao što je prikazano na slici 3.5 gdje je vidljivo kako postoje zasebne mape za sloj pohrane podataka (*persistence*) i prezentacijski sloj (*rest*).

```sql
create table appointment
(
    id                  serial primary key not null,
    uuid                char(36)           not null,
    injury_id           bigint,
    player_id           bigint             not null,
    status              varchar(7)         not null,
    appointment_datetime timestamp         not null,
    appointment_type    varchar(15)        not null,

    constraint appointment__injury__fk foreign key (injury_id) references injury (id),
    constraint appointment__player__fk foreign key (player_id) references player (id)
);
```

Slika 3.3: SQL skripta za stvaranje tablice liječničkih pregleda

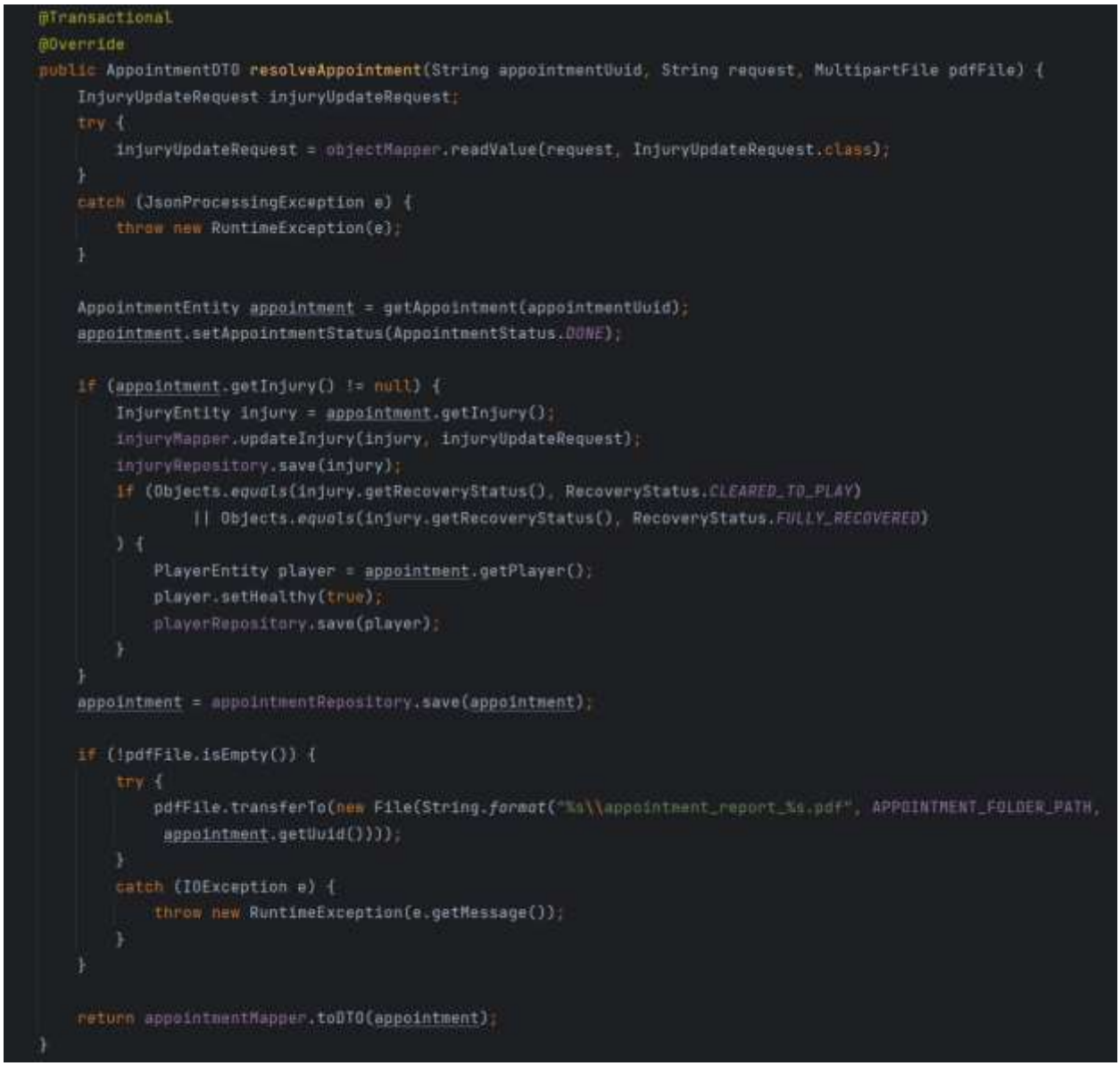
```java
24 usages  KupanovacFilip
@Entity
@Table(name = "appointment")
public class AppointmentEntity {
    3 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;
    3 usages
    @NotNull
    @UUID(version = 4)
    @Column(name = "uuid")
    private String uuid;
    3 usages
    @Nullable
    @ManyToOne
    @JoinColumn(name = "injury_id")
    private InjuryEntity injury;
    3 usages
    @ManyToOne
    @JoinColumn(name = "player_id")
    private PlayerEntity player;
```

Slika 3.4: Programski kod za opis entiteta po uzoru na tablicu baze podataka s referenciranjem drugih entiteta i korištenjem više-na-jedan anotacija

Slika 3.5: Prikaz datotečnog sustava monolitne arhitekture aplikacije

Prilikom poziva na sučelje aplikacije koji rade s više domena i pristupaju te mijenjaju zapise u više tablica, potrebno je pobrinuti se za dosljednost podataka. Ukoliko jedan dio poziva ne uspije te se ne izvrši promjena podataka u bazi koji su trebali biti izmijenjeni, cijelu operaciju je potrebno poništiti i vratiti podatke u bazi na stanje prije poziva aplikacije. Opisani mehanizam su transakcije, koje su teoretski objašnjene u prethodnom poglavlju. U programskom okviru Spring Boot za izvršenje transakcija i upravljanje istima zadužen je transakcijski menadžer kojeg Spring nudi spremnog bez potrebe za konfiguriranjem, ali je također moguće i prilagoditi vlastitim potrebama za složenije slučajeve.

Primjer korištenja transakcije unutar aplikacije je slučaj korištenja prikazan ranije slikom 3.2 prilikom razrješenja liječničkog pregleda. Tada se pristupa aplikaciji prvenstveno u domeni medicinske skrbi - mijenja se stanje liječničkog pregleda čime se označava izvršenim. Kao rezultat liječničkog pregleda može proizići i promjena stanja ozljede (npr. kontrolnim pregledom se potvrdi kako je ozljeda izliječena te se treba promijeniti stanje ozljede na izliječeno). Tada igračeva ozljeda postaje neaktivna, prestaje ograničavati ga da igra i potrebno je promijeniti stanje spremnosti igrača. Ovaj zadnji korak se tiče druge domene, domene tima gdje je potrebno koristiti sloj pohrane podataka i sloj baze podataka. Sloju pohrane podataka iz timske domene pristupa se putem sloja poslovne logike domene medicinske skrbi. Zatim se nadalje pristupa sloju baze podataka i mijenja zapis u bazi podataka za danog igrača. Iako se promjene izvršavaju u različitim bazama, sve je obuhvaćeno jednim procesom, u jednoj aplikaciji. Radi toga je omogućeno korištenje transakcijskog menadžera unutar Spring Boota koji osigurava dosljednost podataka u aplikaciji. Programski primjer opisanog slučaja dan je kôdom na slici 3.6.



Slika 3.6: Korištenje transakcija putem transakcijskog menadžera u Spring Bootu

3.4. ARHITEKTURA APLIKACIJE ZASNOVANA NA MIKROUSLUGAMA

Prije početka migriranja aplikacije, potrebno je definirati arhitekturu na koju se migrira aplikacija uz sve specifičnosti koje ta arhitektura donosi. Kako je riječ o raspodijeljenoj arhitekturi, potrebno je definirati ciljani način preraspodjele usluga, održavanje konzistentnosti podataka i način komunikacije.

3.4.1. Komunikacija među mikrouslugama

Iako međusobno neovisni, moduli trebaju komunicirati. Kako bi smo mogli primijeniti granice konteksta opisane ranije, modul treba zaprimiti zahtjev od drugog za izvršavanje određene operacije. Komunikaciju između modula je potrebno implementirati određenim protokolom.

Prilikom odabira protokola komunikacije treba imati na umu koja obilježja komunikacije su potrebna u ovom slučaju i na koji način se mogu minimizirati troškovi implementacije komunikacije uz što veću efikasnost i pouzdanost.

Jedan od mogućih načina je slanje REST API poziva putem HTTP protokola. To je jedan od najpopularnijih protokola komunikacije među uslugama i vrlo jednostavan za implementaciju u aplikaciji. No postoji niz razloga protiv ovog načina komunikacije među mikrouslugama. Prvi je dijeljenje podataka između usluga. Usluge mogu rukovati s velikim količinama podataka, ovisno o namjeni. Kada bi jedna usluga trebala podatke iz druge, to bi rezultiralo velikim brojem REST API poziva, uz korištenje velike količine podataka koju treba prenositi mrežom i tako trošiti mrežnu infrastrukturu koja može ometati i značajno usporiti odzive usluga prema korisnicima i uzrokovati slabije performanse usluga. Drugi problem koji treba uzeti u obzir je vrsta komunikacije. HTTP je sinkroni protokol, koji blokira izvršavanje niti dok čeka odziv na zahtjev koji je upućen. Kako ne bi prouzrokovao veliko čekanje, potreban je brz odziv. Za ispunjenje toga, moramo imati vrlo pouzdanu vezu i uslugu kojoj pristupamo. Ukoliko je veza preopterećena, odziv će biti sporiji što pruža iznimno lošije performanse. Lošija situacija je kada usluga nije aktivna. Tada se poziv šalje prema usluzi, no pošiljatelj ne može dobiti odziv nego se javlja iznimka i tada nije u mogućnosti izvršiti zahtijevanu funkcionalnost [27]. Unutar ovih slučajeva vidimo neke od zabluda raspodijeljenog računarstva opisanog ranije te ovaj način treba izbjegavati.

Drugi i pogodniji način komunikacije je putem poruka. Korištenjem brokera poruka osiguravamo veću neovisnost među uslugama koje trebaju komunicirati. Usluga šalje poruku brokeru koji zatim prema parametrima i atributima poruke dalje šalje u određene redove. Usluga koja treba zaprimiti poruku osluškuje redove, odnosno preplaćena je na njih i pojavom poruke ju preuzima i obrađuje njen sadržaj. Prednost ovog načina je što je slanje poruka asinkroni proces, što znači da usluga koja inicira komunikaciju ne čeka odziv nego nastavlja dalje s izvršavanjem funkcionalnosti, dok se broker brine dalje o isporuci poruke. Ukoliko nema aktivnih usluga (radi kvara, održavanja, podizanja nove inačice nakon promjene) koje osluškuju red, poruke se čuvaju u redu sve dok se ne pojavi usluga koja je preplaćena na taj red. Ovime se osigurava isporuka poruke onda kada ju usluga može prihvatiti i ne moraju obje usluge biti aktivna istovremeno za komunikaciju. Tako se osigurava neovisnost među uslugama prilikom komunikacije.

Dodatno za smanjenje potrebne komunikacije među uslugama koristimo komunikaciju porukama za pohranu podataka koje usluge trebaju dijeliti u obje usluge koje komuniciraju uz stvaranje granica konteksta, a za valjanost dijeljenih podataka se oslanjamo na eventualnu konzistentnost podataka koji će biti opisani u nastavku.

3.4.2. Granice konteksta

Usluge će biti raspodijeljene prema domenama kako je prikazano slikom 3.1, dakle na 4 usluge. Svakoj usluzi je nužno osigurati da djeluje samostalno, neovisno o tome je li uključena druga usluga – što je jedan od glavnih ciljeva primjene arhitektura mikrousluga. Kako bi to bilo omogućeno, uslugama moramo omogućiti spremanje podataka iz drugih usluga unutar domene te usluge. U tu svrhu dodaju se zasebni entiteti i tablice u bazi podataka.

Klase koje predstavljaju reprezentaciju modela iz drugih domena prilagođavaju se potrebama domene u kojoj će biti smještene, stvarajući ih tako da sadrže samo nužne atribute za novu domenu u kojoj se stvara reprezentacijski model. Primjerice, unutar domene medicinske skrbi predviđeni korisnici su klupski liječnici i ostalo medicinsko osoblje čija je odgovornost briga za zdravlje igrača. Od svih podataka koje sadrži entitet igrača u domeni menadžmenta tima, liječničkom osoblju su potrebni samo jedinstveni identifikator za evidenciju ozljeda i praćenje pregleda, ime i prezime igrača, dok ostale atribute možemo zanemariti u domeni medicinske skrbi. Stoga reprezentacija igrača unutar medicinske skrbi ima samo ta tri navedena atributa, smanjujući količinu podataka na one nužne za rad s ostatkom domene i potrebnu količinu podataka koja se šalje porukama putem brokera za efikasniju komunikaciju među uslugama. Ovim načinom se stvaraju ranije opisane granice konteksta – granice poslovne logike aplikacije gdje se unutar domene uokviruju sve potrebne klase i reprezentacije modela za funkcioniranje jedne domene, odnosno skupa funkcionalnosti potrebnih jednoj specifičnoj grupi korisnika.

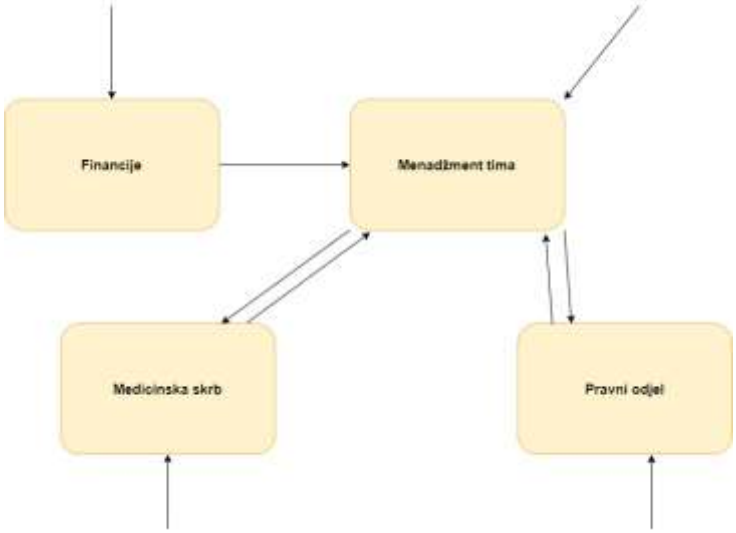
3.4.3. Konzistentnost podataka

Korištenjem monolitne arhitekture možemo jednostavno upravljati bazom podataka. Kod relacijskih baza podataka tako koristimo primarne i strane ključeve za ostvarenje veza među entitetima i osiguravanje potrebnih međuovisnosti podataka. Također na raspolaganju imamo vrlo jednostavnu implementaciju transakcija čime osiguravamo poništavanje izmjene zapisa u bazi podataka ukoliko drugi povezani zapis nije uspješan. Kod mikrousluga je drugačiji slučaj, obzirom da se koristi više baza podataka, obično jedna po usluzi. Unatoč tome, potrebno je osigurati konzistentnost podataka među bazama za podatke koji se dijele. Ti podaci se šalju porukama kako je navedeno prilikom opisa komunikacije među uslugama. Zaprimljeni podaci se tada spremaju i izmjenjuju zapise u bazi ovisno o funkcionalnosti definiranoj nakon zaprimanja poruke. Ukoliko usluga koja zaprima poruku nije aktivna kada pošiljatelj šalje poruku, promjena se neće izvršiti odmah, no poruka čeka u redu brokera poruka na primatelja da postane aktivan i spreman zaprimiti poruku. Tako će primatelj obraditi poruku kada bude u mogućnosti i u svojoj bazi podataka replicirati potrebne podatke koje zaprima od druge usluge ostvarujući tako eventualnu konzistentnost u bazama podataka. To znači da se u rukovanju podacima oslanjamo na garanciju

da će u jednom trenutku u budućnosti nakon promjene u jednoj bazi podataka, nastati jednaka promjena u drugoj te da će s vremenom postati sinkronizirane [28].

3.5. POSTUPAK MIGRIRANJA POSLUŽITELJSKOG DIJELA APLIKACIJE

Kako bi migriranje bilo provedeno, potrebno je prvo osmisliti plan. U nastavku će biti prikazan teoretski plan migriranja sustava iz monolitne arhitekture u arhitekturu mikrousluga. Za uspješno i što efikasnije migriranje nužno je poznavati strukturu programske podrške i postojeću arhitekturu. Potrebno je također poznavati načine komunikacije unutar aplikacije – koji dijelovi komuniciraju s drugim dijelovima aplikacije, kakva je vrsta komunikacije. Osim načina komunikacije, za planirano migriranje iz monolitne arhitekture u arhitekturu mikrousluga, važno je znati i proteže li se neka funkcionalnost kroz višestruke domene – nakon migriranja takva funkcionalnost koristi više usluga, za tu funkcionalnost potrebno je implementirati komunikaciju među uslugama. U nastavku, na slici 3.7 prikazana je struktura domena (modula) monolitne aplikacije, zajedno sa smjerovima komunikacije između različitih domena.



Slika 3.7: Prikaz smjerova komunikacije između domena

Sa slike je vidljivo kako ne komunicira svaka domena međusobno – komunikacija se odvija samo na nekim relacijama između domena. Kako je ranije navedeno u poglavlju 2, za migriranje je najpraktičnije pronaći modul koji sadrži najmanje interakcije s ostalim modulima te prvo njega izdvojiti u zasebnu uslugu. Nakon izdvajanja jedne usluge, potrebno je pronaći sljedeći s najmanje interakcija te njega izdvojiti i taj postupak ponavljati sve dok svi moduli monolitne aplikacije nisu odvojeni u zasebne mikrousluge. Ovakav pristup umanjuje mogućnost grešaka obzirom da se postupno mijenja struktura aplikacije inkrementalnim promjenama, a manji inkrementi omogućuju lakše otkrivanje pogrešaka uzrokovanih promjenama te povratak na staro stanje.

Obzirom da svaki od modula ima pristupnu točku, odnosno programsko sučelje za komunikaciju s drugim uslugama i aplikacijama (prikazano strelicom koja samo vrhom pokazuje na blok modula), niti jedan ne smatramo dubljim što se tiče strukturnog pogleda na aplikaciju. Stoga djeluje najpogodnije koristiti obrazac *gušenja* za migriranje svih usluga. No, također vidimo kako neki moduli međusobno sadrže dvosmjernu komunikaciju – za određene funkcionalnosti poziv dolazi izvana na jedan modul koji zatim poziva funkcionalnosti drugog modula prilikom obrade zahtjeva i izvršenja funkcionalnosti, a u određenim slučajevima komunikacija teče u suprotnom smjeru. Tada dijelove modula možemo smatrati dubljima, iz razloga što ih pozivi iz drugih modula iniciraju, umjesto izravnog pristupa iz neke korisničke aplikacije. Oni nisu pogodni za migriranje putem obrasca *gušenja*. U tu svrhu koristi se obrazac *grananje putem apstrakcije*, te će se za migriranje usluga koristiti oba obrasca, ovisno o vrsti funkcionalnosti i komunikaciji s ostalim modulima.

Planirani redoslijed migriranja u ovom slučaju je prvo izdvojiti modul financija u zasebnu uslugu. Ta usluga je najpogodnija obzirom da ona sadrži samo funkcijske pozive prema drugim uslugama što upućuje na malo ovisnosti s drugim dijelovima aplikacije. Preostali moduli sadrže dvosmjernu komunikaciju koja čini postupak migriranja kompleksnijim i zahtjevnijim od migriranja financijskog modula. Obzirom da moduli medicinske skrbi i pravnog odjela sadrže komunikaciju samo s jednim modulom, jednostavniji su za migriranje od modula menadžmenta tima koji ima dvosmjernu komunikaciju s dva modula, te će radi toga taj modul biti posljednji izdvojen. Preostalo je odlučiti o redoslijedu između modula medicinske skrbi i pravnog odjela – obzirom da sadrže podjednake ovisnosti i komunikaciju, ne postoji nužno jednostavnije rješenje i ovdje je odluka proizvoljna te je odluka prvo migrirati medicinsku skrb, a zatim pravni odjel. Naposljetku, plan migriranja modula iz monolitne aplikacije je prema sljedećem rasporedu:

1) Financije
2) Medicinska skrb
3) Pravni odjel
4) Menadžment tima

4. PROGRAMSKO RJEŠENJE APLIKACIJE

Ovo poglavlje sadrži opise implementiranja poslužiteljske aplikacije za menadžment sportskog kluba, prikazuje kako je izvedena programska podrška uz ukazivanje na ključne dijelove i riješene probleme prilikom stvaranja programske podrške.

4.1. PROGRAMSKI JEZICI, TEHNOLOGIJE I RAZVOJNA OKOLINA

U nastavku će biti opisani alati, programski okviri i programski jezici korišteni za razvoj navedene aplikacije za menadžment sportskog kluba.

4.1.1. Programski jezik Java

Java je programski jezik razvijen u tvrtci Sun Microsystems 1995. godine, prvotno pod imenom Oak, no zbog autorskih prava promijenjeno je u ime koje se zadržalo sve do danas. Najbitnije karakteristike jezika su da je Java objektno-orijentirani jezik opće namjene i otvorenog koda, što ga čini dostupnim svim korisnicima koji ga žele koristiti za razvoj vlastitih aplikacija. Razvijen je s filozofijom „*napiši jednom, izvodi bilo gdje*“ (u izvorniku *write once, run anywhere*), što označava mogućnost da se program pisan u Javi može izvršavati na svim operacijskim sustavima za koje postoji virtualni stroj JVM (*eng. Java Virtual Machine*). Radi toga, programeri ne moraju za svaki operacijski sustav pisati drugačiji kod, nego mogu samo prenijeti kod na druge uređaje i jednako ga izvršavati – Java kod je neovisan o sklopovlju na kojem se izvršava, a JVM ga prevodi u strojni kod prilagođen operacijskom sustavu na kojemu se izvodi. U vrijeme kada je Java nastala to je pružalo značajnu prednost u odnosu na druge postojeće jezike te je upravo ta prenosivost, između ostalih prednosti, zaslužna za veliku popularnost programskog jezika Java [29]. Sintaksa jezika je bliska s jezicima C i C++, a kao odgovor Microsofta na Javu nastao je još jedan sličan jezik, C#. Java također nudi i automatsko rukovanje smećem (*eng. Garbage collection*), koje samostalno uklanja reference na dijelove memorije koji se više ne koriste, oslobađajući tako memoriju. U nekim drugim jezicima korisnici to moraju sami činiti, tako da ova značajka Jave olakšava razvoj programske podrške gdje programere oslobađa ručnog rukovanja memorijom. Radi neovisnosti o sustavu, primjene Jave su mnogobrojne te uključuju Web aplikacije, aplikacije u ugradbenim sustavima, poslovne aplikacije, aplikacije za velike količine podataka (*big data*), aplikacije na oblaku računala i mnoge druge. Također, mobilne aplikacije za pametne telefone s Android OS su u početku bile pisane u Javi, do razvoja programskog jezika Kotlin, no i on također za prevođenje na strojni kod koristi JVM.

4.1.2. Programski okvir Spring boot

Spring Boot je programski okvir otvorenog koda koji pomaže razvoj aplikacija na Javi, i drugim programskim jezicima koji se prevode na JVM-u. Glavne karakteristike Spring Boota su

ubrizgavanje ovisnosti (*Dependency Injection*) i inverzija kontrole (eng IoC - *Inversion of Control*). Pomoću toga, omogućen je razvoj aplikacija koje koriste slabu povezanost (*loosely coupled*), što omogućava veću fleksibilnost, održavanje i skalabilnost aplikacija. Nastao je kao nadogradnja na programski okvir Spring, s pružanjem dodatnih mogućnosti za razvoj Web aplikacija [30]. Okvir pruža korisnicima već gotove biblioteke s konfiguriranim klasama potrebnim za pokretanje Web aplikacija, ubrzavajući i pojednostavljujući tako proces razvoja programske podrške gdje programeri trebaju brinuti o logici aplikacije, dok Spring Boot pruža automatsku konfiguraciju kako bi aplikacija mogla raditi samostalno. Unutar Spring Boot-a postoje razne gotove biblioteke i klase koje sadrže podršku za:

- Pristup i rad s bazom podataka – putem JPA (*Java Persistence API*) i Spring ORM (*Object Relational Mapping*)
- Sigurnost – sigurnosni mehanizmi zaštite podatka, autorizacije i autentikacije
- Testiranje – sadrži klase i anotacije koje olakšavaju jedinično i integracijsko testiranje
- Oblak računala – sadrži biblioteku s klasama koje pružaju podršku razvoju aplikacija na oblaku računala
- Razvoj Web aplikacija – sadrži podršku za razvoj aplikacija putem Springove MVC arhitekture (Model-View-Controller) i razvoj Web aplikacija zasnovan na HTTP protokolu

Iako je prvotno razvijen za korištenje s Javom, Spring Boot je moguće koristiti i s drugim jezicima koji se prevode putem JVM, kao što su Kotlin i Scala.

4.1.3. Razvojno okruženje IntelliJ IDEA

IntelliJ IDEA je razvojno okruženje pisano u Javi namijenjeno razvoju programske podrške u jezicima koji se prevode na JVM-u – Java, Kotlin, Groovy, Scala. Razvija ga i održava kompanija JetBrains, a prva inačica ovog razvojnog okruženja objavljena je 2001. godine. Razvojno okruženje moguće je koristiti na svim vodećim računalnim operacijskim sustavima: Windows, Linux i MacOS, a karakterizira ga iznimno velika podrška pri programiranju korištenjem mnogih alata i dodataka [31]. Svojom prvom inačicom ovo okruženje je postalo jedno od prvih razvojnih okruženja koje je pružalo podršku za navigaciju u kodu i refaktoriranje koda u Javi. Spomenutom podrškom za navigaciju moguće je izravno otvaranje klasa kroz odabir iz koda neke druge klase. Također, prepoznaje pogreške u pisanju koda i nudi mogućnost ispravka nudeći već gotove mogućnosti ispravka neispravnog koda.

Osim podrške u kodiranju, IntelliJ IDEA sadrži niz alata koji pomažu u razvoju aplikacija. Moguće je pokretati aplikacije s raznim konfiguracijama (profilima) koje se mogu koristiti za

uključivanje određenih postavki i varijabli pri pokretanju aplikacije. Omogućava i otklanjanje grešaka (*debugging*) kroz zaseban alat. Sadrži podršku i za mnoge popularne programske okvire za testiranje – *JUnit, TestNG, Arquillian, Selenium*. Pomoću toga moguće je provesti testiranje i pokretanje automatskih testova kroz razvojno okruženje, uz dodatnu analizu i prikaz statistike i metrika pokrenutih testova, kao i prikaz udjela pokrivenosti koda testiranjem. Korištenjem ugrađenog terminala moguć je rad s jezgrom operacijskog sustava kao putem naredbenog retka (*command-line*) bez potrebe za korištenjem drugih aplikacija, a podržava jednake naredbe kao i operacijski sustav na kojemu se pokreće. Baze podataka se također mogu konfigurirati iz razvojnog okruženja uz korištenje alata za rad s bazama podataka. Suradnja u timu i verzioniranje omogućeno je kroz IntelliJ IDEA razvojno okruženje korištenjem alata za podršku sustavima za kontrolu inačica kao što su Git, Mercurial, Subversion i Perforce. Razvojno okruženje također pruža i podršku za razne programske okvire poput Spring Boot-a, Jakarta EE, JPA, Hibernate, Ktor i mnogih drugih, a omogućava i suradnju u stvarnom vremenu putem značajke *Code with me* gdje istovremeno više korisnika može putem mreže zajedno raditi na istim datotekama. Mnoge druge značajke koje razvojno okruženje nudi mogu se pronaći na Web stranicama okruženja [31].

4.2. PROGRAMSKO RJEŠENJE POSLUŽITELJSKE APLIKACIJE

Poslužiteljska aplikacija implementirana je izvorno s monolitnom arhitekturom. U nastavku slijedi pregled dizajna izvorne implementacije postojećeg rješenja.

4.2.1. Opis implementacije monolitne arhitekture

Monolitna aplikacija ima strukturu slojevitog monolita – dakle sadrži različite slojeve unutar kojih su definirane i izvedene funkcionalnosti koje se izvršavaju prema uobičajenim postupcima implementacije slojevitog monolita. Prezentacijski sloj pruža pristupne točke prema poslužiteljskoj aplikaciji za vanjske korisnike, odnosno razne klijentske aplikacije u vidu mobilnih aplikacija, Web frontend aplikacija i slično. Tamo su definirane postavke REST API poziva kojima se ispravno pristupa prezentacijskom sloju, poput potrebnih zaglavlja zahtjeva, tijela (sadržaja) zahtjeva, specifičnih varijabli koje treba sadržavati hiperveza kojom se poziva funkcionalnost na prezentacijskom sloju. Po zaprimanju odgovora od nižih slojeva vraća odgovor pošiljatelju zahtjeva.

Nadalje, prezentacijski sloj po zaprimanju ispravnih zahtjeva poziva sloj poslovne logike. Tamo se obrađuje zahtjev i izvršava tražena funkcionalnost. Sloj poslovne logike koristi komponente iz sljedećeg sloja u topologiji aplikacije – sloja pohrane podataka kako bi dohvatio, izvršio određene operacije nad podacima i vratio rezultat prezentacijskom sloju. Sloj pohrane podataka pristupa bazi podataka i prikazuje reprezentaciju podataka iz baze kroz entitete. Putem

objekata iz ovog sloja može se pristupiti bazi podataka i izvršiti naredbe za manipulaciju podacima unutar baze podataka. Aplikacija je izvedena sa zatvorenim slojevima što znači da svi pozivi moraju ići slijedno kroz svaki sloj do onog potrebnog za izvršenje pozvane funkcionalnosti bez preskakanja bilo kojeg sloja.

4.2.2. Primjena domenski pokretanog dizajna u razvoju aplikacije

Pri dizajnu aplikacije korišten je domenski pokretani dizajn. Klase koje se tiču jedne domene poslovanja aplikacije grupirane su u zasebnu biblioteku, razdvojene od ostalih domena. Ovim načinom postignuto je odvajanje različitih odgovornosti aplikacije kako bi postigli veću razinu neovisnosti među klasama te je moguće detaljnije uvoditi ovisnosti određenih biblioteka prema potrebama različitih dijelova aplikacije. Tako je prezentacijski sloj dodatno podijeljen na module prezentacijskog sloja svake zasebne domene aplikacije – menadžmenta tima, medicinske skrbi, pravnog odjela i financija. Na isti način podijeljeni su i ostali slojevi koji čine slojeviti monolit. Za upravljanje ovisnostima i grupiranje različitih modula u strukturi stvorenih biblioteka korišten je alat za upravljanje projektima Maven. Zbog karakteristika programskog okvira *Spring Boot* poput ubrizgavanja ovisnosti dodatno su razdvojene komponente na apstrakciju funkcionalnosti (sučelja) i implementacije tih apstrakcija. Ovakav način strukturiranja klasa unutar projekta omogućuje slabu povezanost komponenata i specifičnije uključivanje ovisnosti kako ne bi uključivali velike biblioteke s mnogobrojnim funkcijama od kojih komponenta koristi samo mali dio sadržanih mogućnosti.

4.3. OSVRT NA PROGRAMSKO RJEŠENJE POSLUŽITELJSKE APLIKACIJE

Poslužitelj pruža korisnicima aplikacije rukovanje s podacima koje djelatnici sportskog kluba koriste prilikom upravljanja klubom. Ovisno o ulozi, različiti zaposlenici trebaju različite odgovornosti i trebaju izvršavati različite funkcionalnosti. U nastavku bit će prikazana implementacija zahtijevanih funkcionalnosti na aplikaciju i prikazan rad iste.

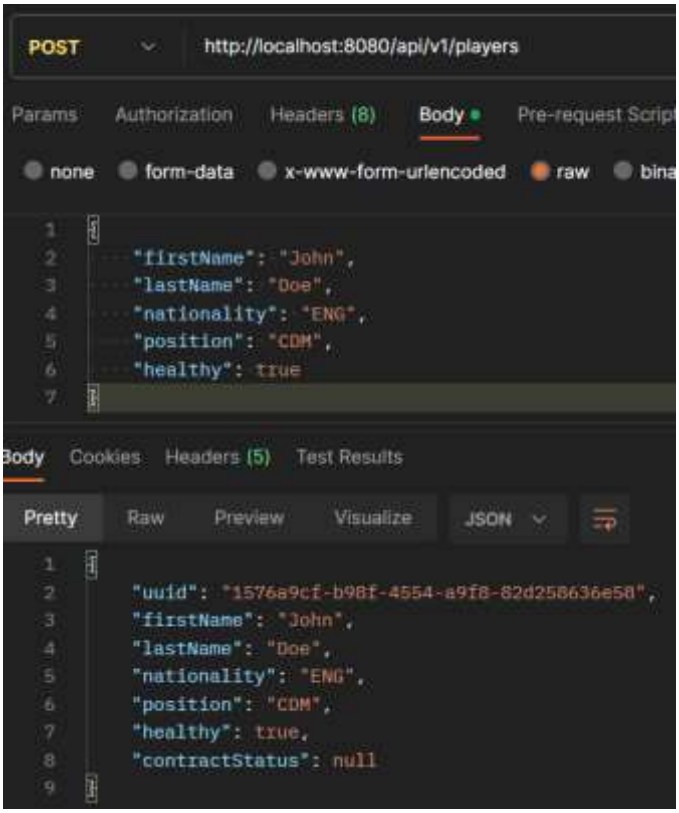
4.3.1. Domena menadžmenta tima

Domena menadžmenta tima služi za evidenciju igrača i osoblja u klubu. Ovdje su izložene funkcionalnosti za stvaranje, dohvaćanje i uklanjanje igrača i osoblja. Prilikom pozivanja metoda za rad s igračima poput prikazane slikom 4.1, aplikacija vraća reprezentaciju entiteta nad kojim je izvršena operacija te je odziv metode s reprezentacijom entiteta prikazan na slici 4.2.

```java
@Override
public PlayerDTO create(CreatePlayerRequest createPlayerRequest) {
    PlayerEntity playerEntity = playerMapper.toEntity(createPlayerRequest)
            .setUuid(idGenerator.generateId().toString());
    playerEntity = playerRepository.save(playerEntity);
    return playerMapper.toDTO(playerEntity);
}
```

Slika 4.1: Metoda za stvaranje igrača

Sa slike 4.2 vidljivo je kako reprezentacija igrača sadrži i atribut koji prikazuje status ugovora igrača. Navedeni atribut dodaje se prilikom stvaranja odgovora kako bi klijentska aplikacija mogla pružiti dodatne informacije o igraču, koje se nalaze izvan entiteta ali su vezane uz njega, prema krajnjim korisnicima. Ovim načinom izvedene su i druge metode iz navedene domene, za obje vrste entiteta, igrače i osoblje.



Slika 4.2: Tijelo zahtjeva i odziv prilikom stvaranja novog igrača

4.3.2. Domena financija

Domena financija omogućava računovodstvu kluba upravljanje transakcijama, gdje su izložene metode stvaranja transakcije i dohvaćanja provedenih transakcija te dohvaćanje izvještaja o transakciji koji se spremaju u podatkovni sustav u PDF formatu, funkcionirajući poput registra računa. Na slici 4.3 prikazan je način stvaranja transakcije unutar aplikacije.

```java
@Transactional
@Override
public TransactionDTO createTransaction(CreateTransactionRequest request) {
    TransactionEntity transactionEntity = transactionMapper.toEntity(request)
            .setUuid(idGenerator.generateId().toString());

    transactionEntity = transactionRepository.save(transactionEntity);
    try {
        createTransactionReport(transactionEntity);
    }
    catch (IOException e) {
        throw new RuntimeException(e);
    }
    return transactionMapper.toDTO(transactionEntity);
}
```

Slika 4.3: Prikaz stvaranja zapisa o transakciji kluba

Entitet transakcije također sadrži i atribut koji definira kome je isplaćena transakcija, odnosno od koga je zaprimljena ukoliko se radi o primitku. Time je omogućeno filtriranje zapisa o transakcijama prema tri različite strane, igračima, osoblju i vanjskim strankama u svrhu stvaranja izvještaja i praćenja tokova financija prema različitim aspektima. Također je osigurano da ukoliko zahtjev za transakciju definira novu transakciju prema igraču ili osoblju, entitet mora biti zapisan u bazi podataka, radi čega sloj poslovne logike iz financija pristupa domeni menadžmenta tima kako bi provjerio postojanje člana tima prije stvaranja zapisa.

Kako je vidljivo sa slike 4.3, metoda za stvaranje transakcije je označena anotacijom *@Transactional*. Ona definira da u slučaju javljanja pogreške u bilo kojem dijelu metode svi prethodno stvoreni zapisi za vrijeme izvršavanja metode budu poništeni. To je potrebno u slučaju da se ne uspije stvoriti PDF datoteka kao dokaz transakcije, koja se stvara metodom *createTransactionReport*, prikazana slikom 4.4.

```java
private void createTransactionReport(TransactionEntity transactionEntity) throws IOException {
    PDDocument transactionDocument = new PDDocument();
    PDPage page = new PDPage();
    transactionDocument.addPage(page);

    PDPageContentStream contentStream = new PDPageContentStream(transactionDocument, page);
    contentStream.setFont(PDType1Font.TIMES_ROMAN, fontSize: 12);
    contentStream.beginText();
    contentStream.showText(transactionEntity.toString());
    contentStream.endText();
    contentStream.close();

    transactionDocument.save(String.format("%s\\transaction_%s.pdf", TX_FOLDER_PATH, transactionEntity
     .getUuid()));
}
```

Slika 4.4: prikaz stvaranja PDF dokumenta sa zapisom transakcije

Spremljenim datotekama moguće je pristupiti pozivanjem zasebne metode koja korisnicima vraća datoteku PDF formata, što služi za upotrebu u klijentskim aplikacijama kako bi korisnici mogli dohvatiti zapis u svrhu daljnje distribucije i za evidenciju svih financijskih tokova kluba.

4.3.3. Domena medicinske skrbi

Domena medicinske skrbi zadužena je za evidenciju i praćenje oporavaka igrača od ozljeda, kao i praćenje njihovih zakazanih liječničkih pregleda. Ozljede igrača klubovima stvaraju velike probleme u natjecanjima obzirom da tada igrači nisu sposobni nastupati ili dati svoj maksimum dok ozljeda nije zaliječena. Stoga klubovi angažiraju vlastitu liječničku službu ili surađuju sa zdravstvenim ustanovama za što bolji tretman i rehabilitaciju. Ova domena služi upravo kako bi medicinsko osoblje unosilo svu evidenciju o ozljedama i tretiranju istih za praćenje zdravstvenog stanja igrača i spremanje svih provedenih postupaka i pratećih dokumenata.

Entiteti koji služe za evidenciju medicinske skrbi su ozljeda i pregled. Prilikom nastanka ozljede, za unos u aplikaciju, potrebno je poslati zahtjev s podacima o ozljedi, referencu na igrača čija je ozljeda, kao i datoteku koja predstavlja nalaz ozljede. Podaci o ozljedi se spremaju u bazu podataka ukoliko je poslan ispravan zahtjev s nalazom. Po uspješnom stvaranju entiteta ozljede, nalaz je spremljen u datotečnom sustavu kako bi se mogao dohvatiti radi daljnjeg tretiranja, a automatski se mijenja stanje atributa zdravlja igrača na vrijednost *0* odnosno *false* ukazujući da je ozlijeđen, što je prikazano slikom 4.5. Obzirom da metoda prikazana na slici pristupa i sloju pohrane podataka domene menadžmenta tima, odnosno mijenja stanje u bazi, također je označena kao transakcijska. Uz to, ukoliko iz nekog razloga nije moguće spremiti liječnički nalaz o ozljedi, također će biti poništene sve promjene osiguravajući tako konzistentnost među podacima.

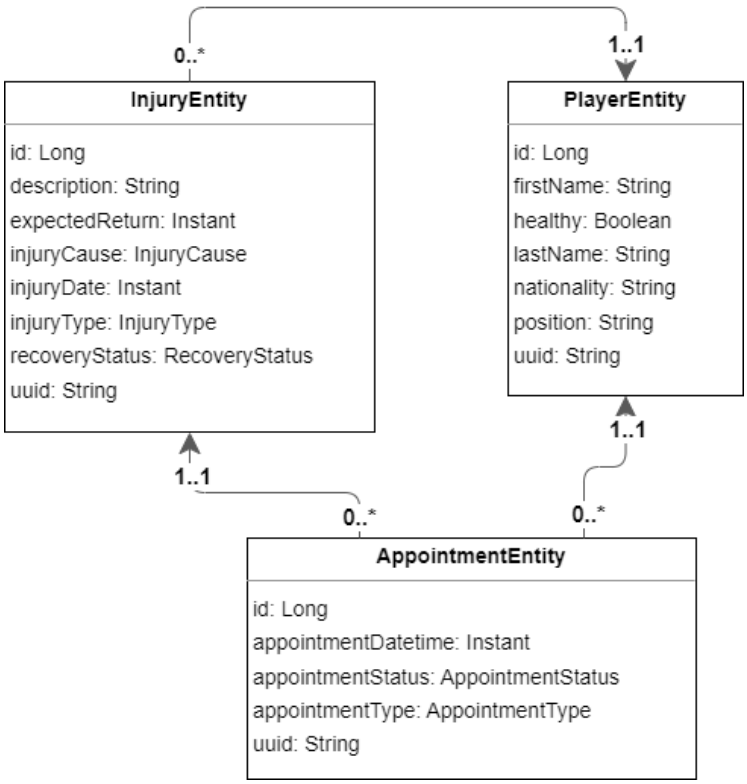
Stanje oporavka od ozljede predstavljeno je enumeracijom koja poprima četiri stanja – oporavak u tijeku, rehabilitacija, dopuštena aktivnost i potpuno izliječen. Kako bi se promijenilo stanje ozljede potrebno je kroz evidenciju liječničkih pregleda izmijeniti stanje bolesti po dovršenom pregledu. Liječnički pregled se stvara kao objekt koji sadrži vrijeme pregleda, identifikator igrača kojem je zakazan pregled, te identifikator ozljede. Moguće je također stvoriti i kontrolni odnosno preventivni pregled koji ne uključuje ozljedu, na isti način samo bez identifikatora ozljede, koji služi kako bi se spriječile povrede igrača i unaprijed otkrili rizici. Kako bi smo pregled iz stanja čekanja prebacili u izvršen, potrebno je unijeti podatke koji predstavljaju nalaz liječničkog pregleda. U zahtjevu se šalje liječnički nalaz te ukoliko je ozljeda prisutna i stanje ozljede utvrđeno pregledom te datum očekivanog završetka ozljede. Ukoliko se pregledom utvrdi kako je ozljeda završena, mijenja se stanje ozljede prema zahtjevu i automatski igraču provjerava

jesu li sve ozljede izliječene te ukoliko više nema aktivnih ozljeda vraća se stanje zdravlja na pozitivnu vrijednost, što je prikazano ranije slikom 3.6 prilikom opisa transakcija.

```java
@Transactional
@Override
public InjuryDTO create(String playerUuid, String injuryRequest, MultipartFile pdfFile) {
    CreateInjuryRequest createInjuryRequest;
    try {
        createInjuryRequest = objectMapper.readValue(injuryRequest, CreateInjuryRequest.class);
    }
    catch (Exception e) {
        throw new RuntimeException(e);
    }

    PlayerEntity player = getPlayer(playerUuid).setHealthy(false);
    InjuryEntity injury = injuryMapper.toEntity(createInjuryRequest);
    injury.setPlayer(getPlayer(playerUuid)).setUuid(idGenerator.generateId().toString());
    playerRepository.save(player);

    injury = injuryRepository.save(injury);
    if (!pdfFile.isEmpty()) {
        try {
            pdfFile.transferTo(
                    new File(String.format("%s\\injury_%s.pdf", INJ_FOLDER_PATH, injury.getUuid()))
            );
        }
        catch (IOException e) {
            throw new RuntimeException(e.getMessage());
        }
    }

    return injuryMapper.toDTO(injury);
}
```

Slika 4.5: Metoda za spremanje ozljede u aplikaciju

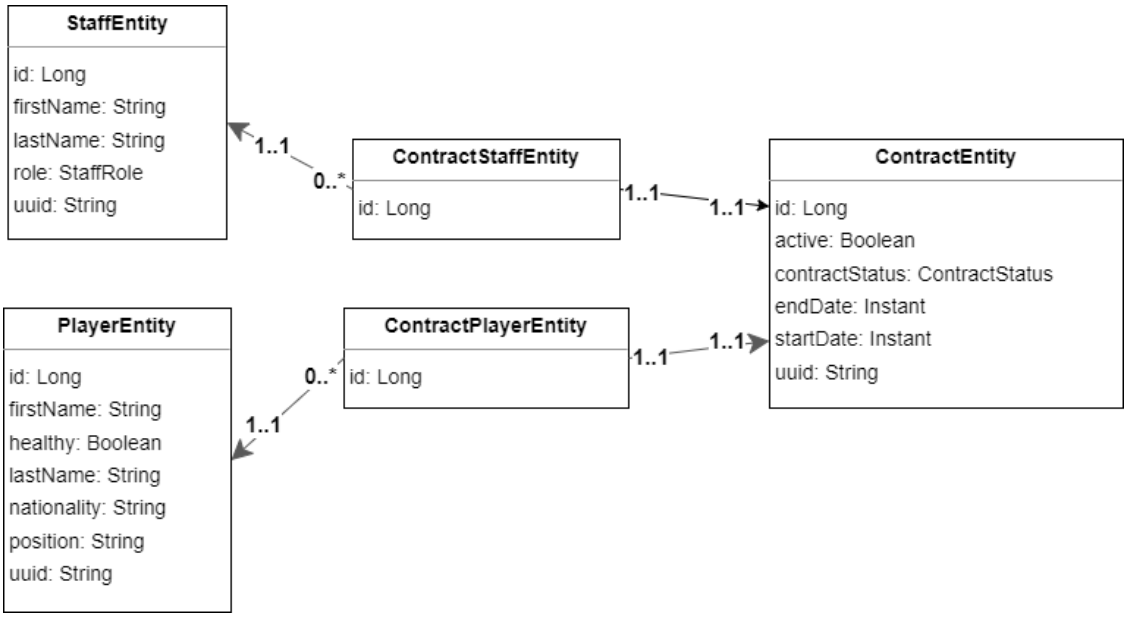
Iz ovog primjera vidimo kako oba entiteta koja se tiču medicinske skrbi komuniciraju i povezana su s menadžmentom tima kako bi osigurali konzistentnost među podacima za praćenje zdravstvenog stanja igrača te postoji povezanost entiteta u bazi podataka prikazana ER (entitet-relacija) dijagramom na slici 4.6.



Slika 4.6: ER dijagram entiteta igrača i medicinske skrbi

4.3.4. Domena pravnog odjela

Pravni odjel ima zadaću voditi brigu o ugovorima igrača, evidentirati i pohranjivati dokumente u digitalnom zapisu. U tu svrhu stvoren je entitet ugovora, koji sadrži informacije o granicama trajanja ugovora, trenutnom stanju ugovora i člana momčadi kojem pripada. Kako bi ugovor mogao biti stvoren, moramo pronaći u bazi podataka igrača ili člana osoblja kojeg ugovor referencira, što znači da domena pravnog odjela mora komunicirati s domenom menadžmenta tima. Obzirom da su članovi momčadi u tablici ugovora u bazi podataka definirani identifikatorima koji poprimaju vrijednosti cijelih brojeva prema različitim sekvencama za tablicu igrača i osoblja, moguće je nastajanje dvoznačnosti prilikom povezivanja ugovora sa članom kome ugovor pripada. Kako bi smo mogli jednoznačno povezati entitet ugovora s pripadajućim članom osoblja ili igračem, ovisno kome pripada, stvaramo dodatno dva asocijativna entiteta koji predstavljaju poveznicu između ugovora i pripadajućeg člana momčadi – entitet *ugovor-igrač* i *ugovor-osoblje*. Asocijativni entiteti u ovom slučaju sadrže samo reference na člana momčadi i ugovor kako bi omogućili ispravno povezivanje ugovora s članovima u slučaju jednakih identifikatora dvaju različitih tipova članova momčadi što je prikazano ER dijagramom na slici 4.7.



Slika 4.7: Povezivanje ugovora s entitetima momčadi putem asocijacija

Za rukovanje ugovorima sada ima tri entiteta. Putem entiteta ugovora možemo dobiti informacije o jednom ugovoru, no on ne sadrži podatke o članu čiji je ugovor. Prilikom stvaranja novog ugovora imamo dvije zasebne pristupne točke, ovisno o tome radi li se o ugovoru igrača ili osoblja. Tada se provjerava postoji li taj član osoblja te ukoliko postoji nastavlja se stvaranje ugovora. Nakon stvaranja ugovora i spremanja u bazu podataka stvara se i novi asocijativni element koji referencira novostvoreni ugovor povezujući ga s već postojećim članom momčadi. Priložena datoteka koja predstavlja dokument ugovora se sprema u podatkovni sustav za evidenciju svih ugovora kluba s članovima. Obzirom da je metoda (prikazana slikom 4.8) transakcijska, i ovdje se u slučaju bilo koje pogreške poništavaju sve promjene zapisa u bazi podataka.

```java
@Transactional
@Override
public ContractPlayerDTO createPlayerContract(String playerUuid, String request, MultipartFile pdfFile) {
    PlayerEntity player = findPlayer(playerUuid);
    CreateContractRequest createContractRequest = null;
    try {
        createContractRequest = objectMapper.readValue(request, CreateContractRequest.class);
    }
    catch (JsonProcessingException e) { throw new RuntimeException(e); }
    ContractEntity contract = createContract(createContractRequest);

    try {
        contractRepository.save(getActivePlayerContract(playerUuid).setActive(false));
    }
    catch (Exception ignored) { }
    if (!pdfFile.isEmpty()) {
        saveContractDocument(pdfFile, contract);
    } else {
        throw new RuntimeException("Contract document not provided!");
    }

    ContractPlayerEntity contractPlayerEntity = new ContractPlayerEntity()
            .setPlayer(player).setContract(findContract(contract.getUuid()));
    return contractMapper.toDTO(contractPlayerRepository.save(contractPlayerEntity));
}
```

Slika 4.8: prikaz metode za stvaranja novog ugovora igrača.

4.4. PROVEDBA MIGRIRANJA POSLUŽITELJSKOG DIJELA APLIKACIJE S MONOLITNE NA ARHITEKTURU MIKROUSLUGA

Unutar ovoga dijela bit će opisana izvedba migriranja programske podrške za aplikaciju za menadžment sportskog kluba iz monolitne arhitekture u arhitekturu mikrousluga prema ranije opisanom planu i redoslijedu migriranja pojedinih modula.

4.4.1. Maven i struktura biblioteka aplikacije

Monolitna izvedba aplikacije ima tako strukturirane biblioteke da su razdvojene odgovornosti, primarno s ciljem mogućnosti ponovnog iskorištenja biblioteka za višestruke aplikacije. Kao pomoć u takvom organiziranju i povezivanju biblioteka kao ovisnosti i dodataka u projektu služi nam alat za automatizaciju izgradnje i upravljanje projektima *Maven*.

Maven omogućuje povezivanje modula putem ovisnosti, kontrolu inačica uključenih biblioteka, definiranje faza i zadataka pojedine faze projekta (kompilacija, testiranje, distribucija), uključivanje dodataka (*eng. plug-in*) i konfiguriranje profila. Sve ove karakteristike projekta definiraju se u datoteci *pom.xml* (*POM – Project Object Model*), koja služi za konfiguraciju projekta i pojedinih biblioteka. Definicija aplikacije je razdvojena na višestruke module, od čega je (u slučaju monolitne aplikacije) prisutan jedan aplikacijski modul koji pruža ulaznu točku aplikacije – klasu *Main* – koja pokreće aplikaciju i u objektnom modelu projekta tog modula

definirane su ovisnosti koje predstavljaju druge biblioteke koje aplikacija koristi. Ostali moduli funkcioniraju kao biblioteke, nudeći razne klase i komponente. Prikaz korištenja biblioteka unutar aplikacije kao ovisnosti dan je primjerom na slici 4.9.

Prilikom migriranja aplikacije biblioteke će se moći ponovno koristiti i u mikrousluzi. Mikrousluga se izrađuje na jednak način kao i monolitna aplikacija – *Maven* modul s definiranom ulaznom točkom aplikacije. Za ponovno korištenje biblioteke i u novoj usluzi potrebno je samo uključiti potrebne ovisnosti u objektni model projekta kako bi imali dostupne željene module za korištenje u usluzi.

```
<parent>
    <groupId>hr.master_thesis.kupanovac</groupId>
    <artifactId>master-monolith</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
</parent>

<artifactId>monolith-application</artifactId>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- modules -->
    <dependency>
        <groupId>hr.master_thesis.kupanovac</groupId>
        <artifactId>monolith-common</artifactId>
    </dependency>
    <dependency>
        <groupId>hr.master_thesis.kupanovac</groupId>
        <artifactId>monolith-configuration</artifactId>
    </dependency>
    <dependency>
        <groupId>hr.master_thesis.kupanovac</groupId>
        <artifactId>monolith-contract-api-impl</artifactId>
    </dependency>
    <dependency>
        <groupId>hr.master_thesis.kupanovac</groupId>
        <artifactId>monolith-finance-api-impl</artifactId>
    </dependency>
    <dependency>
        <groupId>hr.master_thesis.kupanovac</groupId>
        <artifactId>monolith-medical-api-impl</artifactId>
    </dependency>
    <dependency>
        <groupId>hr.master_thesis.kupanovac</groupId>
        <artifactId>monolith-team-api-impl</artifactId>
    </dependency>
```

Slika 4.9: Prikaz dijela uključenih ovisnosti u aplikaciju

4.4.2. Migriranje modula bez primanja poziva drugih modula aplikacije

Prethodno definiranim planom migriranja određeno je kako će prvi migrirani modul biti modul financija. Odabran je kao prvi iz razloga što ne zaprima pozive iz drugih modula monolitne aplikacije, nego samo putem sučelja za komunikaciju s drugim aplikacijama, a komunikaciju s drugim modulima inicira isključivo taj modul iniciranjem zahtjeva prema drugim modulima – u ovom slučaju prema modulu menadžmenta tima (vidi sliku 3.7).

Migriranje modula započinjemo stvaranjem novog *Maven* aplikacijskog modula – nazvanog *microservice-finance* za migriranje domene financija u zasebnu mikrouslugu. Mikrousluga je u POM strukturi kreirana kao dijete projektnog modula, koji obuhvaća sve module kreirane za potrebe ove aplikacije. Zatim uključujemo željene ovisnosti koje su potrebne unutar usluge financija. Unutar modula usluge postavljamo željenu konfiguraciju i konfiguracijske postavke prema potrebama zahtjeva na uslugu – poput konfiguriranja rada aplikacije, odabiranja priključka na kojem će aplikacija biti izložena, konfiguracije različitih profila, postavljanja vlastitih postavki i varijabli. Kako bi usluga bila neovisna, potrebno je i da ima vlastitu bazu koju bi mogli dizajnirati prema potrebama te usluge neovisno o ostalim uslugama. PostgreSQL nudi mogućnost na jednom poslužitelju održavanje više baza podataka, te samo kreiramo unutar poslužitelja novu bazu s drugačijim imenom i definiramo postavke veze u datoteci *application.yaml*, kako je prikazano slikom 4.10.

```yaml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/finance
    driver-class-name: org.postgresql.Driver
    username: root
    password: root
```

Slika 4.10: Konfiguriranje veze mikrousluga s novom bazom

Novonastala usluga sadrži isključivo jednu domenu koja treba funkcionirati kao samostalna cjelina koja se može pokrenuti i koristiti neovisno o aktivnosti drugih usluga. Kako bi omogućili samostalan rad, potrebno je uokviriti sve odgovornosti unutar domene kako bi usluzi bile dostupne sve klase koje su potrebne za izvršavanje funkcija koje izvršava.

Unutar nove usluge sada nastaje problem s funkcionalnostima koje su komunicirale s drugim domenama monolitne aplikacije. Implementacija poslovne logike monolitnog rješenja uključivala je i pozive prema drugim domenama kao primjerice provjeru postojanosti igrača ili člana osoblja u bazi podataka prilikom stvaranja zapisa o izvršenoj transakciji isplate plaće zaposleniku kluba. Kako se ta provjera izvršava izvan područja odgovornosti domene financija,

usluga ju ne može izvršiti te je potrebno definirati način na koji će usluga moći pristupati tim podacima.

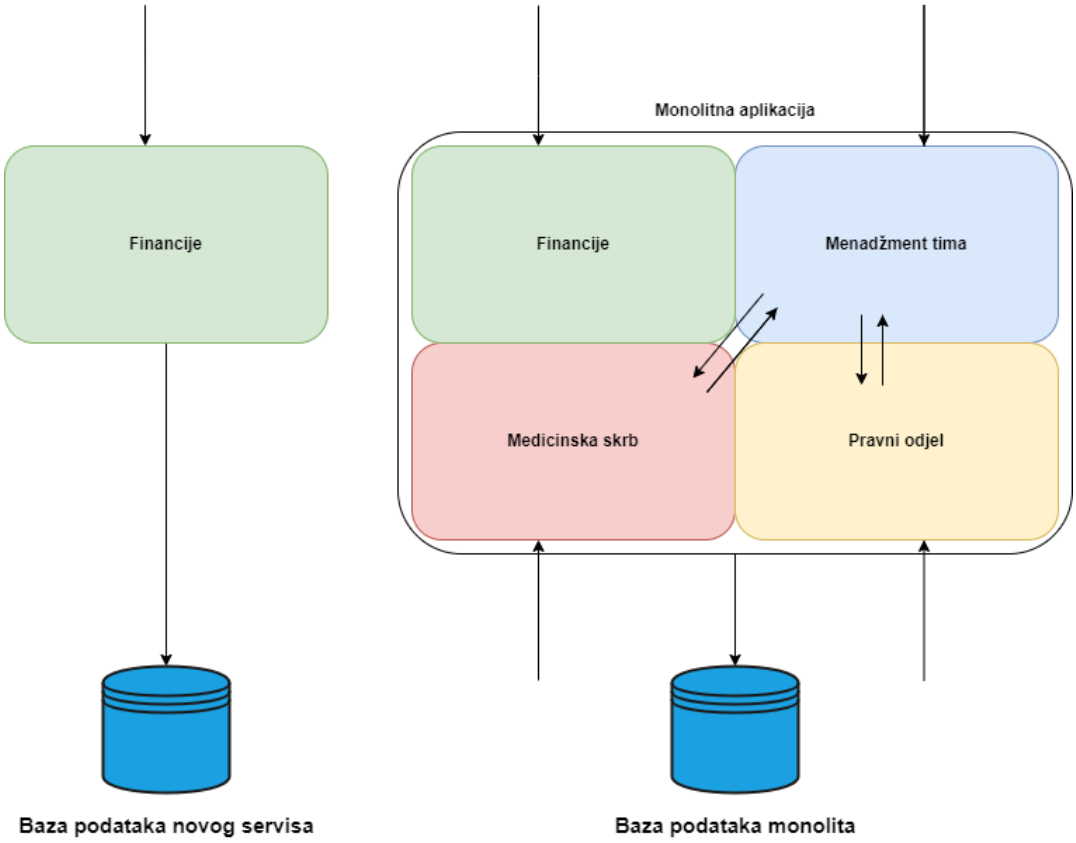
Za pristup podacima iz druge domene jedna mogućnost je korištenje HTTP klijenta koji bi slao zahtjev prema modulu kojeg treba za provjeru postojanosti igrača. Ovim načinom omogućeno je dohvaćanje potrebnih podataka radi provjere i daljnje obrade bez dodavanja novih klasa i mijenjanja funkcionalnosti u usluzi. No, potrebno je prisjetiti se ranije navedenih zabluda raspodijeljenog računarstva kako bi uvidjeli kako ovo ne predstavlja dobro rješenje – primarno činjenicu da propusnost mreže nije neograničena i održavanje infrastrukture zahtjeva veći trošak što bolje karakteristike trebamo [7]. Ovakvim pozivima narušavamo i performanse usluge obzirom da usluga kojoj pristupamo preko HTTP klijenta mora obrađivati zahtjeve mikrousluga u isto vrijeme dok obrađuje i zahtjeve korisnika koji su upućeni izravno toj usluzi. Kako bi sve to mogla obraditi efikasno, usluga mora imati veću računalnu snagu nego samo za obradu izravnih zahtjeva koji dolaze od strane korisničkih aplikacija. Također, ukoliko usluga kojoj pristupamo nije aktivna, sruši se iz nekog razloga, mikrousluga koja ga poziva također neće moći dobiti odgovor niti zatim izvršiti funkcionalnost koju je korisnik zatražio. Iz toga primjera vidimo kako nije postignuta neovisnost među uslugama, što je jedna od glavnih karakteristika koju želimo postići arhitekturom mikrousluga.

Kako bi smo ostvarili neovisnost među uslugama potrebno je primijeniti drugačije rješenje. Potrebno je u uslugu dodati klase koje predstavljaju podatke koje tražimo od druge usluge i spremiti ih u bazu te mikrousluge. Mikrousluga financija ne treba sve atribute zaposlenih (igrača i osoblja) koje nudi domena menadžmenta tima. Stoga se entitet zaposlenika prilagođava potrebama ove usluge i kreira samo s atributima potrebnima mikrousluzi. Ovim postupkom smanjuje se potrebna količina podataka za slanje između usluga i stvaraju granice konteksta na način da usluga unutar sebe sadrži sve podatke koji su mu potrebni za rad bez da mora zahtijevati informacije od drugih usluga. Za slanje podataka između usluga koristi se drugačija komunikacija od HTTP protokola – komunikacija porukama putem brokera za poruke kao što je RabbitMQ.

Koristeći RabbitMQ broker možemo postići komunikaciju između usluga koja ne zahtijeva da oba kraja komunikacije budu aktivna u trenutku kada se šalje poruka. Broker funkcionira na principu da pošiljatelj pošalje poruku usmjerivaču (*eng. exchange*), usmjerivač je povezan vezama (*eng. bindings*) na redove poruka (*eng. queue*). Ovisno o pravilima veza, usmjerivač šalje poruku u redove putem veza gdje ključ unutar poruke odgovara ključu veze. Na prijemnoj strani komponente osluškuju redove i po pojavi poruke zaprimaju te obrađuju poruku. Ukoliko nema aktivnih slušatelja, poruka stoji u redu sve dok se ne pojavi aktivni slušatelj. Također redovi traže potvrdu uspješne obrade poruke i ako se pojavi iznimka ili pogreška tijekom obrade poruke od

strane slušatelja, poruku je moguće ponovno poslati u red ili preusmjeriti u drugi red za obradu neuspješnih ili pogrešnih poruka putem mehanizama koje omogućava RabbitMQ.

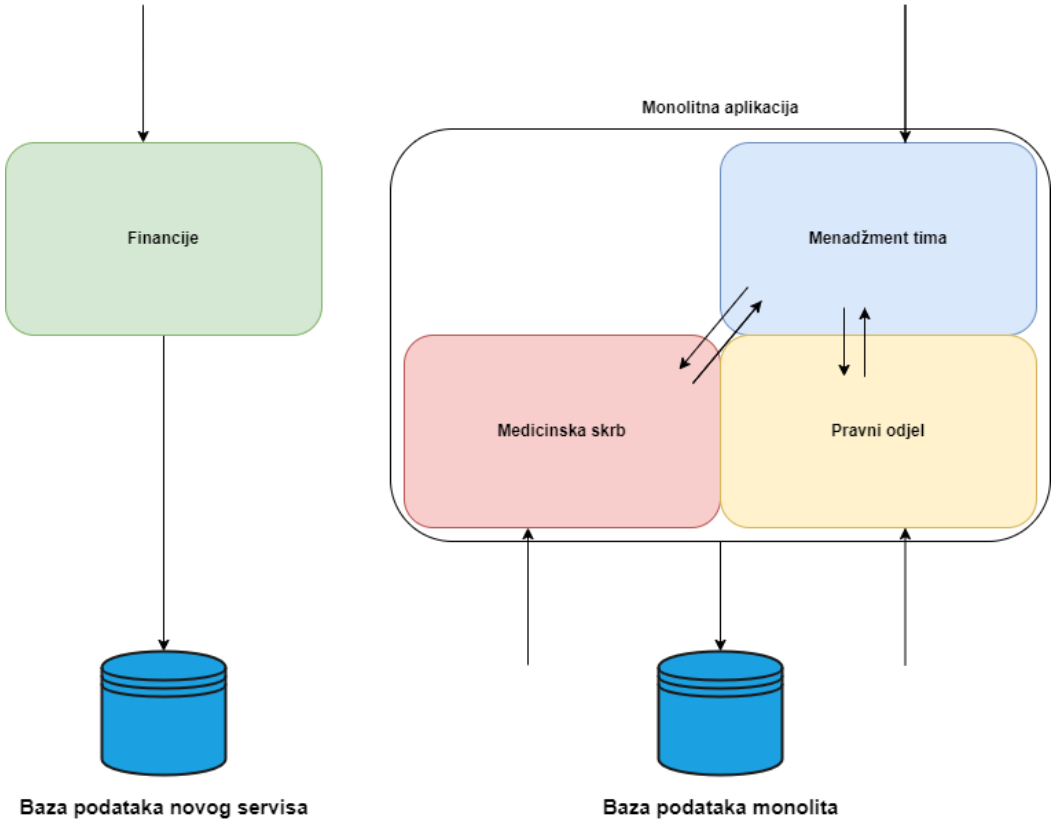
Kako bi smo mogli paralelno pokrenuti obje implementacije modula – onu unutar monolita i u zasebnoj usluzi, potrebno je da imamo dvije implementacije jednakih funkcionalnosti – jednu za modul u monolitu koja je već prisutna i novu koja bi unutar mikrousluge izvršavala jednake funkcionalnosti. Spring Boot za tu namjenu nudi profile kao jednu od opcija konfiguracije aplikacije i zatim prema odgovarajućem profilu učitava one komponente koje su označene profilom koji koristimo kod pokretanja aplikacije. Tako možemo napraviti profile *monolith* i *microservice*, i po jednu implementaciju za svaki profil te istovremeno pokrenuti različite implementacije sloja poslovne logike, imajući tako aktivne i monolit s modulom financija i mikrousluga financija što je prikazano slikom 4.11. Ovo je potrebno kako bi se moglo utvrditi valjanost nove mikrousluge i da na jednaki način obavlja zahtijevane funkcionalnosti, što možemo provjeriti ručnim i automatskim testiranjem.



Slika 4.11: Prikaz paralelnog rada modula financija u usluzi i monolitu

Nakon što se testiranjem utvrdi jednako djelovanje mikrousluga kao i izvedbe istih funkcionalnosti u monolitu, moguće je *ugušiti* rješenje unutar monolita, odnosno ukloniti ga i ostaviti samo implementaciju mikrousluga. Ostavljanjem samo jedne implementacije sloja

poslovne logike, profili nisu više potrebni te je moguće ukloniti anotaciju profila s preostale implementacije koja je koristila profile kako bi se uklonila nepotrebna konfiguracija. Uklanjanjem tih preostalih dijelova koji postaju nepotrebni nakon migriranja dobivamo sustav kao što je prikazan slikom 4.12.



Slika 4.12: Topologija sustava nakon migriranja modula financija u mikrouslugu

4.4.3. Migriranje modula koji primaju pozive drugih modula aplikacije

Prethodnim odjeljkom opisan je postupak migriranja modula koji je inicirao komunikaciju s drugim modulima u aplikaciji, ali nijedan nije njega pozivao. Kako više nema modula u monolitu s takvim svojstvima, migriranje nastavljamo sa sljedećom razinom kompleksnosti komunikacije. Migrirati će se moduli koji iniciraju komunikaciju u nekim funkcionalnostima, dok nude i određene funkcionalnosti koje drugi moduli iniciraju pozivajući se na njih. U toj kategoriji su svi preostali moduli – medicinska skrb, pravni odjel i menadžment tima, koji će biti migrirani tim redoslijedom, kao što je opisano u planu migriranja unutar prethodnog poglavlja ovog rada (vidi 3.5).

Prilikom migriranja ovih modula, radi toga što nisu samo na inicirajućoj strani komunikacije među modulima nego i prijemnoj u nekim slučajevima, nije dovoljno koristiti samo prethodno opisani obrazac *gušenja*. Potrebno je upotrijebiti i obrazac *grananje putem apstrakcije*.

Unutar ovoga odjeljka će stoga biti detaljnije prikazan postupak migriranja modula koji zaprimaju pozive iz drugih modula aplikacije, s naglaskom na korake obrasca *grananja putem apstrakcije*.

Kako bi smo primijenili obrazac *grananja putem apstrakcije* potrebno je slijediti ranije opisane korake primjene ovog obrasca. Prvi korak je kreiranje apstrakcijskog sloja. Kako Spring Boot već potiče ubrizgavanje ovisnosti pri razvoju aplikacija putem ovog programskog okvira, već su prisutne apstrakcije za gotovo sve elemente koji čine aplikaciju. Sučeljima je definiran ugovor koji sve implementacije sučelja moraju ispuniti, odnosno imati implementaciju svih potrebitih funkcija za ispunjenje zahtjeva na komponentu. Obzirom da klijenti već pristupaju apstrakcijskom sloju i u monolitnoj izvedbi aplikacije, korak preusmjeravanja klijenata na apstrakciju umjesto na izravnu implementaciju funkcionalnosti je učinjen. Nastavno na to, kreiramo novu implementaciju za uslugu koju želimo migrirati uz implementaciju sučelja koje predstavlja apstrakciju funkcionalnosti.

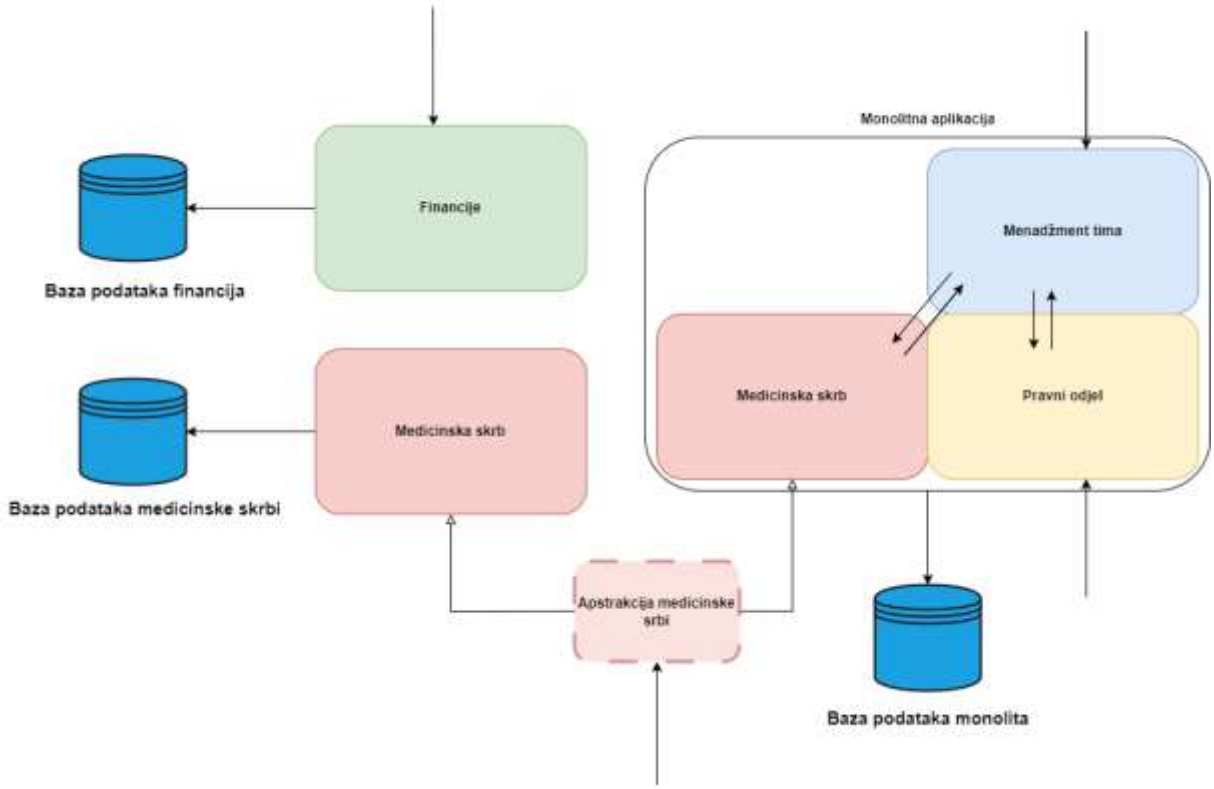
Kada smo pripremili novu implementaciju apstrakcije koja bi trebala u mikrousluzi izvršavati jednake funkcionalnosti kao prije migriranja, potrebno je testiranjem utvrditi je li zaista na jednak način ispunjava dane zahtjeve. Kako bi smo to mogli provjeriti, u usluzi primjenjujemo novu implementaciju dok u monolitu ostavljamo staru te ih pokrećemo da rade paralelno. Ručnim i automatskim testovima utvrđujemo valjanost nove implementacije. Kada testiranjem dokažemo kako je nova implementacija u potpunosti ispunila dane zahtjeve, možemo sigurno ukloniti staru funkcionalnost ostavljajući aktivnu samo novu aplikaciju. Iz monolita uklanjamo i biblioteke koje sadrže funkcionalnosti modula kojeg smo migrirali čime se dovršava proces migriranja modula putem obrasca *grananja*. Moguće je tada ukloniti i apstrakciju u vidu sučelja, no radi načela *Spring Boot*-a i ubrizgavanja ovisnosti ostavljamo sučelje apstrakcije.

Primjenu ovog obrasca prikazati ćemo prvo na modulu medicinske skrbi. Ovaj modul unutar monolitne aplikacije sadrži dvosmjernu komunikaciju s modulom menadžmenta tima, stoga je za implementaciju rješenja arhitekture mikrousluga potrebno modificirati obje strane komunikacije. Modul ispunjava funkcionalnosti za praćenje i evidenciju ozljeda igrača, kao i vođenje te dokumentaciju liječničkih pregleda. Kako bi mogli povezati ozljede i preglede s igračem na kojeg se odnose, potrebne su informacije o igraču te radi toga stvaramo novi entitet unutar domene medicinske skrbi za unutrašnju reprezentaciju igrača koji sadrži njegov jedinstveni identifikator, ime i prezime – jedine informacije potrebne za praćenje zdravstvenog stanja. Osim toga, menadžment tima treba informacije o aktivnim ozljedama i nadolazećim pregledima kako bi mogli pratiti zdravstveno stanje igrača i njihovu spremnost za treniranje i nastup. Stoga također u modulu menadžmenta tima stvaramo nove entitete kako je prikazano objektnim modelom iz alata za prikaz modela baze podataka kojeg nudi IntelliJ IDEA (slika 4.13).

Slika 4.13: Objektni model novih entiteta u bazi podataka za menadžment tima

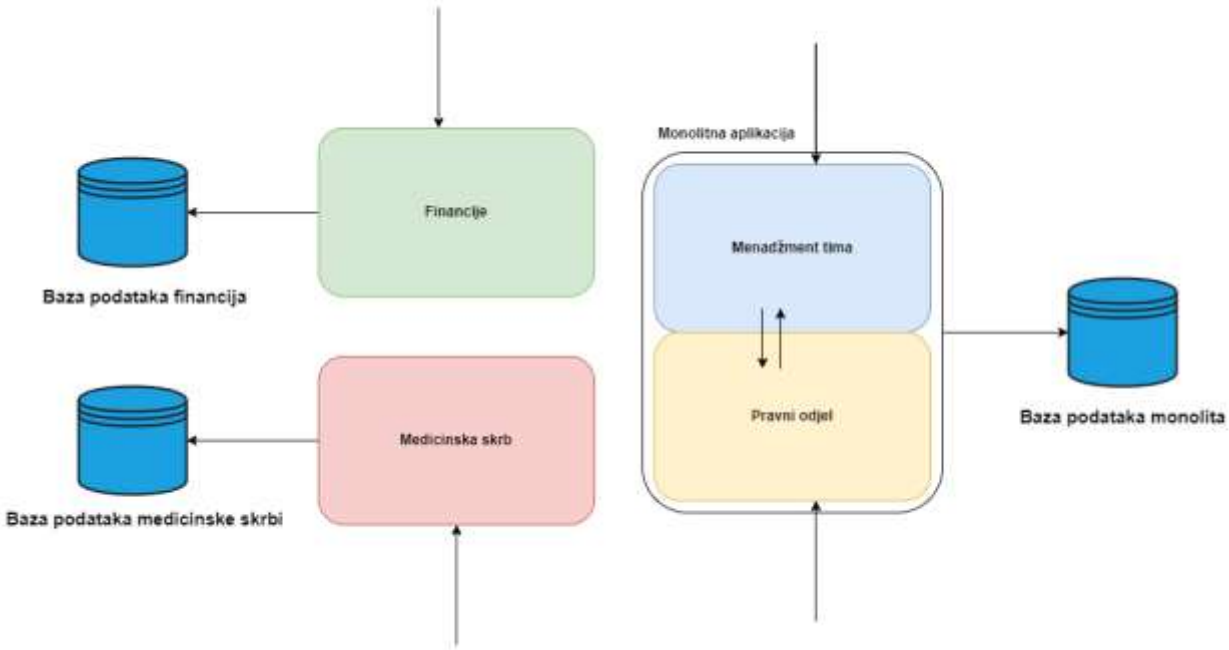
Za praćenje koraka obrasca *grananja* potrebno je zatim kreirati novu implementaciju sloja poslovne logike za upotrebu u usluzi. Nasljeđujemo postojeću apstrakciju te implementiramo metode definirane ugovorom (sučeljem). Kako bi smo imali dostupne podatke o igraču u novoj usluzi, potrebno je pri stvaranju igrača poslati poruku u red te u usluzi omogućiti osluškivanje tog reda te obradu zaprimljene poruke. Radi toga što i modul menadžmenta tima zahtijeva informacije o medicinskoj skrbi, šaljemo poruke i u obrnutom smjeru. Mikrousluga šalje poruke o nadolazećim pregledima i aktivnim ozljedama. Također po obradi pregleda i završetku ozljede šalju se poruke za brisanje tih entiteta iz modula menadžmenta tima, odnosno baze podataka koju modul koristi. Ovim načinom postižemo prikaz samo onih stavki koje su potrebne modulu menadžmenta, odnosno uokvirujemo potrebne podatke u granice konteksta menadžmenta tima. Po implementaciji navedenih stavki potrebno je kreirati i aplikacijski modul unutar projekta – nazvan *microservice-medical* koji u svoj objektni model projekta uključuje potrebne ovisnosti iz domene medicinske skrbi. Kreiramo novu bazu podataka na istom poslužitelju, te konfiguracijom postavljamo postavke veze s novom bazom te postižemo topologiju aplikacije kako je prikazano slikom 4.14 u nastavku.

Sa slike 4.14 vidljivo je postojanje dvije implementacije iste apstrakcije unutar projekta. Kako bi smo mogli odabrati koju implementaciju migrirane apstrakcije koristimo ponovno koristimo profile. Novoj implementaciji dodjeljujemo profil i označavamo ju primarnom koristeći anotaciju *@Primary*. Time se označava u Springu da treba koristiti ovu implementaciju za ubrizgavanje ovisnosti tamo gdje postoji više implementacija iste apstrakcije. Radi toga ne moramo prilikom pokretanja aplikacije u paralelnom radu modula u monolitu i mikrousluzi koristiti dva profila, nego je dovoljno upotrijebiti samo jedan profil.



Slika 4.14: Grananje putem apstrakcije za modul medicinske skrbi

Nadalje, za utvrđivanje valjanosti nove implementacije koristimo automatske testove, ali i pokrećemo obje implementacije paralelno i ručnim testiranjem utvrđujemo funkcionalnu jednakost usluge sa monolitnom izvedbom. Kada se otklone svi mogući nedostaci i utvrdi stvarna jednakost izvedbi obje implementacije, moguće je ukloniti prvotnu monolitnu implementaciju modula. Iz objektnog projektnog modela (POM) monolitne aplikacije uklanjamo biblioteke koje sadrže klase iz domene medicinske skrbi. Naposljetku možemo maknuti oznake profila s nove implementacije migrirane apstrakcije obzirom da ostaje jedina te vrste, a iz baze podataka uklanjamo tablice koje se tiču domene medicinske skrbi. Važno je napomenuti da ukoliko se migriranje odvija nakon što je aplikacija već puštena u rad, podatke iz tih tablica je potrebno migrirati u novu bazu prije brisanja tablica kako bi se spriječio gubitak postojećih podataka. Nakon poduzimanja ovih koraka, migrirana je još jedna usluga, te imamo aplikaciju s topologijom prikazanom na slici 4.15.



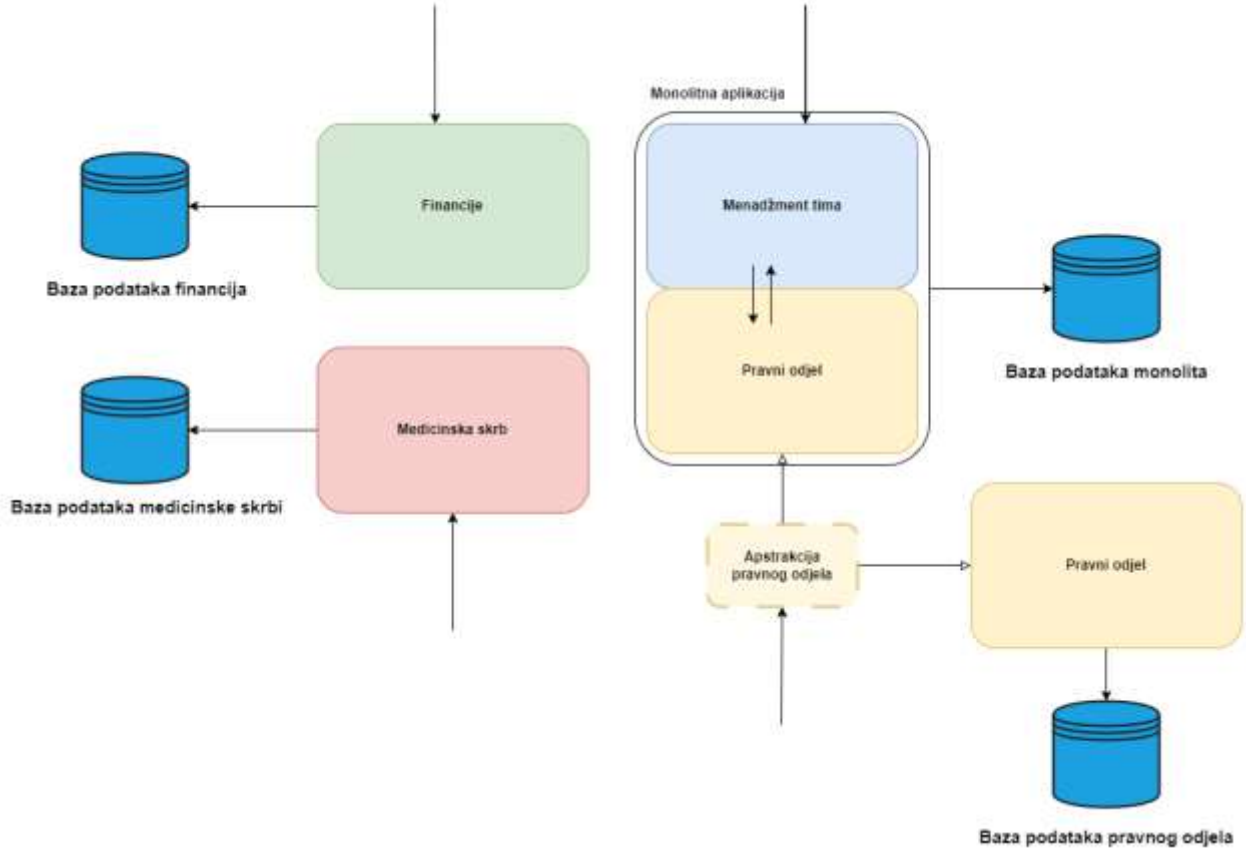
Slika 4.15: Topologija aplikacije nakon migriranja medicinske skrbi u zasebnu uslugu

Nakon migriranja medicinske skrbi u zasebni modul, unutar monolitne aplikacije ostali su moduli iz domena pravnog odjela i menadžmenta tima, kao što je prikazano slikom 4.16. Prema planu migriranja slijedi migriranje pravnog odjela. Također kao prethodni modul, sadrži pozive prema drugoj domeni za provjeru postojanja zaposlenika, a prima i pozive iz drugih domena za provjeru postojanja i trajanja ugovora igrača.

Za migriranje ponovno stvaramo novi modul unutar projekta koji predstavlja novu mikrouslugu, nazvanu *microservice-contract*, konfiguriramo postavke aplikacije, dodjeljujemo novi priključak na kojem će biti izložena aplikacija, kreiramo novu bazu podataka (*contract*) na istom poslužitelju putem PostgreSQL poslužitelja. U objektni model projekta dodajemo potrebne ovisnosti biblioteka iz domene pravnog odjela. Nakon ovih koraka stvoreni su uvjeti za početak migriranja putem obrasca *grananja*.

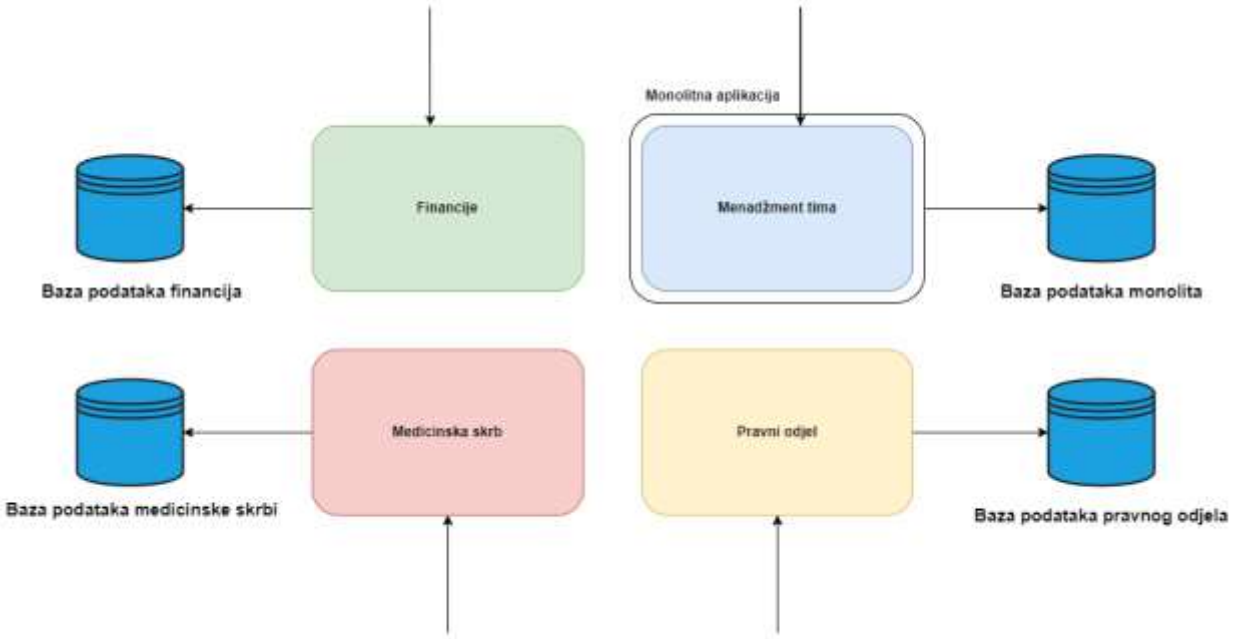
Oba navedena slučaja komunikacije domene pravnog odjela s drugim dijelovima unutar aplikacije su s domenom menadžmenta tima te će biti potrebno ponovno stvoriti dodatne entitete za ograničavanje granica konteksta poslovne logike svakog modula. Za pravni odjel to znači stvaranje novog entiteta zaposlenika, koji sadrži jedinstveni identifikator zaposlenika (UUID), ime, prezime i vrstu zaposlenika, je li netko igrač ili član stožera. Na drugoj strani, unutar domene menadžmenta tima stvaramo entitet koji predstavlja aktivne ugovore, s atributima jedinstvenog identifikatora ugovora, granicama trajanja – početnim i završnim danom valjanosti ugovora i jedinstvenog identifikatora zaposlenika kojem pripada.

Postojeću apstrakciju poslovne logike pravnog odjela implementiramo ponovno za mikrouslugu. Implementiramo zadane funkcionalnosti definirane sučeljem. Potrebno je dodati i slušatelje za poruke koje se šalju prema usluzi pravnog odjela – za kreiranje novog zaposlenika iz domene menadžmenta tima. Nakon implementacije sučelja za pokretanje u usluzi, pokrećemo obje implementacije paralelno te provjeravamo valjanost implementacije u sustavu prikazanom slikom 4.16.



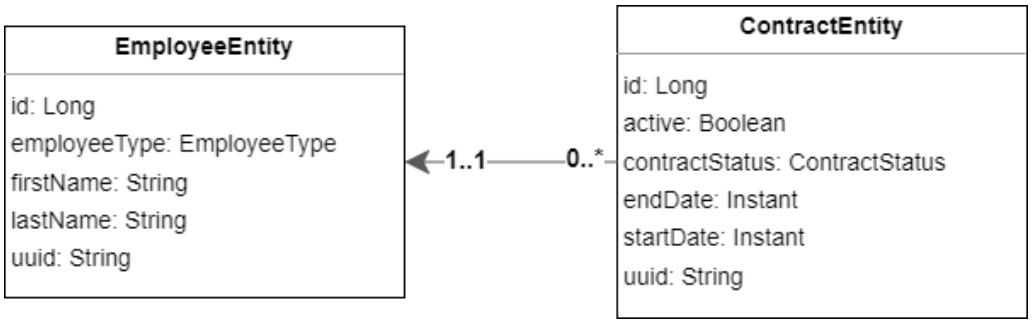
Slika 4.16: primjena obrasca *grananja* na domeni pravnog odjela

Ponovno se testiranjem provjerava valjanost implementacije nove usluge. Nakon što je potvrđena valjanost, uklanjamo migriranu domenu iz monolita, ostavljamo mikrouslugu aktivnom, a postojeće podatke ukoliko su prisutni prebacujemo iz baze podataka monolita u novu bazu podataka i tako završavamo migriranje. Nakon tog postupka ostaje struktura aplikacije kako je prikazano na slici 4.17.



Slika 4.17: topologija aplikacije nakon migriranja usluge pravnog odjela

Kako su sada u granicama konteksta pravnog odjela i igrači i članovi osoblja predstavljeni novim zajedničkim entitetom zaposlenika, nisu više potrebni asocijativni entiteti između ugovora i zaposlenika kao što je to bio slučaj s monolitnim rješenjem. Sada možemo ukloniti te entitete i izravno povezati ugovor sa zaposlenikom na kojeg se odnosi ostvarujući tako značajno jednostavniju strukturu tablica unutar baze podataka pravnog odjela, čiji su entiteti prikazani dijagramom na slici 4.18. Obzirom da ugovor mora sadržavati zaposlenika na kojega se odnosi, a zaposlenik može biti spremljen bez pripadajućeg ugovora, entitet ugovora sadrži referencu na zaposlenika.



Slika 4.18: ER dijagram pravnog odjela nakon uklanjanja asocijativnih entiteta

Kako bi ostavili pristupne točke na sučelju jednakima kao i prije promjene, u metodama za stvaranje ugovora igrača i člana osoblja provjeravamo atribut tipa zaposlenika je li odgovara traženom u toj metodi i stvaramo ugovor za igrača na isti način, što je prikazano na slici 4.19. U budućim inačicama aplikacije moguće je promijeniti pristupne točke i ukloniti nepotrebno, no za

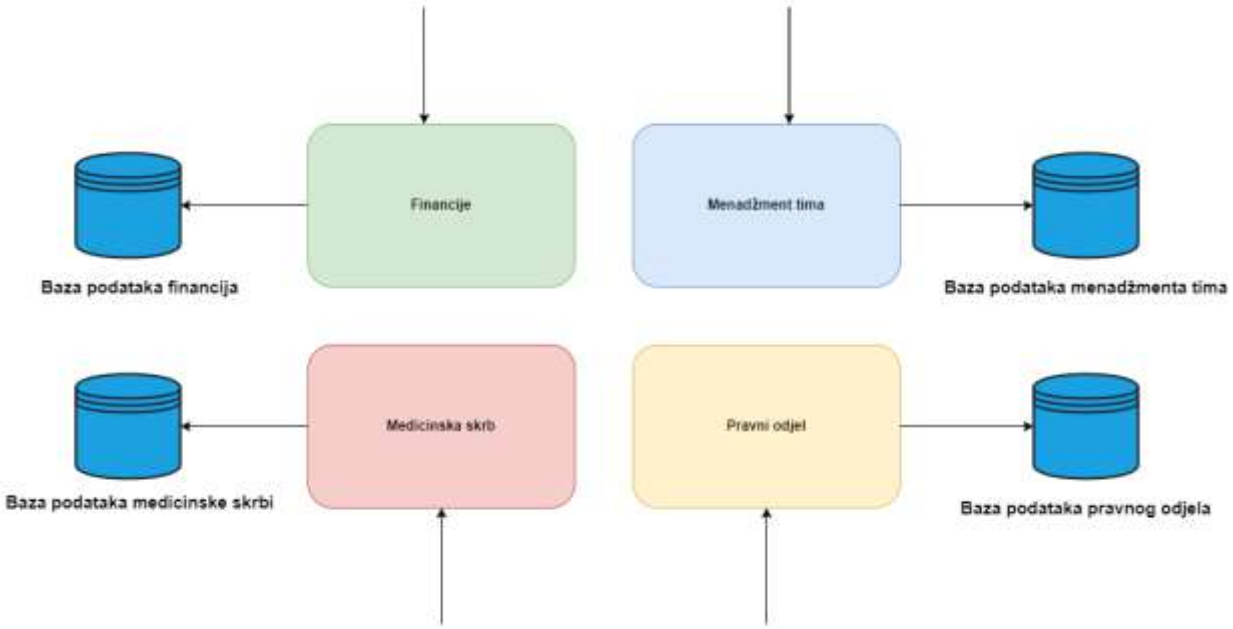
potrebe migriranja nužno je ostaviti ih jednakima radi provjere ispravne implementacije postojeće apstrakcije.

```
@Transactional
@Override
public ContractDTO createPlayerContract(String playerUuid, String request, MultipartFile pdfFile) {
    if (pdfFile.isEmpty()) {
        throw new RuntimeException("Must provide contract pdf document!");
    }
    EmployeeEntity employee = findPlayer(playerUuid);
    try {
        ContractEntity activeContract = getEmployeeActiveContract(employee).setActive(false);
        activeContract = contractRepository.save(activeContract);
        LOGGER.info("Changed contract status to inactive. {}", activeContract);
    }
    catch (Exception ignored) {
    }

    CreateContractRequest ccr = getContractRequestFromString(request);
    ContractEntity contract = contractMapper.toEntity(ccr)
            .setUuid(idGenerator.generateId().toString())
            .setActive(true)
            .setContractStatus(assignContractStatus(ccr.getStartDate(), ccr.getEndDate()))
            .setEmployee(employee);
    contract = contractRepository.save(contract);

    saveContractDocument(pdfFile, contract);

    ActiveContractMessage message = new ActiveContractMessage()
            .setUuid(contract.getUuid()).setEmployeeUuid(playerUuid)
            .setStartDate(contract.getStartDate()).setEndDate(contract.getEndDate());
    rabbitPublisher.sendPlayerActiveContractMessage(message);

    return contractMapper.toDTO(contract);
}
```

Slika 4.19: Implementacija stvaranja ugovora igrača u mikrousluzi

Nakon što je migriran i treći po redu modul, unutar izvornog monolita preostaje još jedino modul menadžmenta tima kako je prikazano i na slici 4.17. Migriranjem prethodnih modula provjerene su valjanosti svih dotadašnjih funkcionalnosti i komunikacije preostalog modula s novonastalim uslugama. Kako su već zbog tih migriranja uklonjene preostale domene koje čine aplikaciju, preostala domena sadrži potrebne modele za uokvirivanje funkcionalnosti unutar vlastitih granica konteksta prilagođene potrebama poslovne logike te domene, ima definiranu komunikaciju porukama s drugim uslugama aplikacije i u bazi podataka s kojom komunicira preostale su samo tablice preostale domene. Iz toga je vidljivo kako monolitna aplikacija u trenutnom stanju efektivno djeluje kao zasebna mikrousluga menadžmenta tima.

Tada možemo direktno premjestiti preostalu domenu u zasebnu uslugu uklanjanjem omotača monolitne aplikacije samo za potvrdu migriranja i preostalog modula. Kreiramo novu aplikaciju unutar projekta, u objektni model projekta unesemo sve jednake ovisnosti kao što su preostale u monolitnoj aplikaciji. Konfiguriramo uslugu da bude izložena na zasebnom priključku kako bi potvrdili migriranje preostale domene iz monolita u mikrouslugu dok bazu podataka možemo ostaviti istu ili kreirati novu. Ponovno radi potvrde izvršavanja migriranja kreiramo novu bazu podataka za uslugu menadžmenta tima i tamo primijenimo tablice iste kao u bazi podataka monolitne aplikacije. Nakon ovoga možemo u potpunosti ukloniti monolitnu aplikaciju iz projekta, obzirom da nakon migriranja posljednjeg modula ne sadrži više niti jednu funkcionalnost, a aplikacija je sada uspješno migrirana na arhitekturu mikrousluga i ima topologiju prikazanu slikom 4.20 ispod.



Slika 4.20: Topologija aplikacije s arhitekturom mikrousluga

5. KORIŠTENJE I ISPITIVANJE APLIKACIJE S ANALIZOM REZULTATA

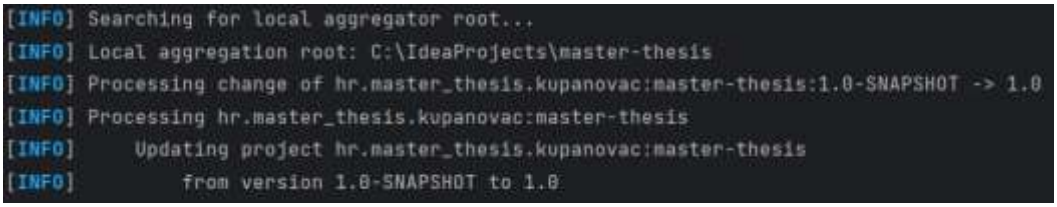
U ovom poglavlju bit će prikazan rad migrirane aplikacije uz osvrt i usporedbu s prethodnim monolitnim rješenjem. Objašnjen je način pokretanja aplikacije mikrousluga te provjera ispravnog rada funkcionalnosti aplikacije.

5.1. NAČIN KORIŠTENJA APLIKACIJE

U ovom dijelu prikazan je postupak izgradnje aplikacije i pokretanja iste od gotovog koda aplikacije migrirane na arhitekturu mikrousluga.

5.1.1. Izgradnja Maven projekta

Dovršetkom migriranja aplikacije, potrebno je poduzeti još neke korake kako bi aplikacija mogla biti izvršavana na računalima. Potrebno je izgraditi projekt, odnosno kreirati arhivske datoteke aplikacija ili biblioteka za daljnje korištenje lokalno unutar projekta. Maven omogućava praćenje inačica projekta kako bi mogli razlikovati stvorene izvršne datoteke prilikom upotrebe, koje po kreaciji dobivaju naziv u formatu <*artifactId*>-< inačica >*.jar*, gdje je *artifactId* svojstvo Mavena koje predstavlja naziv modula. Stoga je potrebno voditi brigu o pravilnom označavanju inačica tijekom rada na projektu. Uobičajeno je da inačice koje su u procesu promjene budu označene sufiksom -*SNAPSHOT*, dok se po dovršetku uklanja označavajući tako dovršenu inačicu projekta. Tako je prije izgradnje projekta promijenjena inačica projekta u svim modulima koji su dio projekta na način da uklonimo sufiks. Obzirom da projekti Mavena mogu sadržavati desetke, čak i stotine modula unutar svoje strukture, ručno mijenjanje bilo bi, u nekim situacijama, dugotrajan proces. Stoga Maven nudi opciju automatske promjene inačice naredbom u naredbenom retku (ili terminalu, ovisno o operacijskom sustavu). Naredba za to je *mvn versions:set -DnewVersion='*naziv_nove_inačice*' -DprocessAllModules*. Naredba je izvršena iz direktorija u kojemu se nalazi roditeljski dokument objektnog modela projekta (*pom.xml*), nakon čega se mijenja inačica modula i svih elemenata koji ga nasljeđuju unutar strukture projekta. Primjer rezultata izvršenja postavljanja projekta na inačicu 1.0 prikazan je slikom 5.1.



Slika 5.1: Zapis izvršenja promjene inačice maven projekta u naredbenom retku

Nakon promjene inačice moguće je izgraditi projekt. Izgradnja Maven projekta postignuta je pokretanjem naredbe *mvn install* koja izgrađuje aplikaciju stvarajući arhivske datoteke prema

ranije opisanom formatu. Izgrađene datoteke su JAR formata, što predstavlja Java Archive datoteke, odnosno arhivske Java datoteke. U arhivi su sadržane datoteke Java klasa i povezani metapodaci i resursi unutar jedne arhivske datoteke namijenjene za distribuciju Java koda. Kako bi smo mogli pokrenuti Java aplikaciju, potrebno je da JAR datoteka sadrži ulaznu točku aplikacije, odnosno klasu s metodom definicije *public static void main(String[] args)* [32]. Ukoliko arhiva sadrži takvu metodu, Java virtualni stroj prepoznaje ulaznu točku i može pokrenuti aplikaciju sadržanu u arhivi. Naredba kojom se pokreće Java aplikacija sadržana u arhivi je *java -jar <naziv_arhivske_datoteke>*.

5.1.2. Pokretanje aplikacije mikrousluga

Nakon izgradnje aplikacije, potrebno je konfiguriranje okoline u kojoj se aplikacija pokreće, što u ovom slučaju podrazumijeva pokretanje brokera poruka i baze podataka. Za potrebe prikaza funkcionalnosti aplikacije broker RabbitMQ i poslužitelj baze podataka PostgreSQL pokrenuti su unutar Docker kontejnera na pristupnim priključcima 5672 i 5432, koji su standardni priključci ovih usluga.

Aplikacija se sastoji od četiri usluge, iz čega je izgradnjom proizašlo 4 arhivske datoteke za pokretanje svake usluge zasebno. Za potrebe prikaza rada aplikacije, usluge se pokreću na istom računalu, no moguće je aplikaciju pokrenuti i na više računala raspoređivanjem usluga na različita računala. Kako bi usluge mogle istovremeno biti pokrenute na istom računalu, odnosno mrežnom čvoru, potrebno je da svaka usluga bude izložena na zasebnom priključku. Informacije o imenu arhivske datoteke i priključku dane su tablicom 5.1.

Usluga	Naziv arhivske datoteke	Izloženi priključak	Baza podataka
Menadžment tima	microservice-team-1.0	8081	localhost:5432/team
Medicinska skrb	microservice-medical-1.0	8082	localhost:5432/medical
Pravni odjel	microservice-contract-1.0	8083	localhost:5432/contract
Financije	microservice-finance-1.0	8084	localhost:5432/finance

Tablica 5.1: Informacije pokretanja usluga

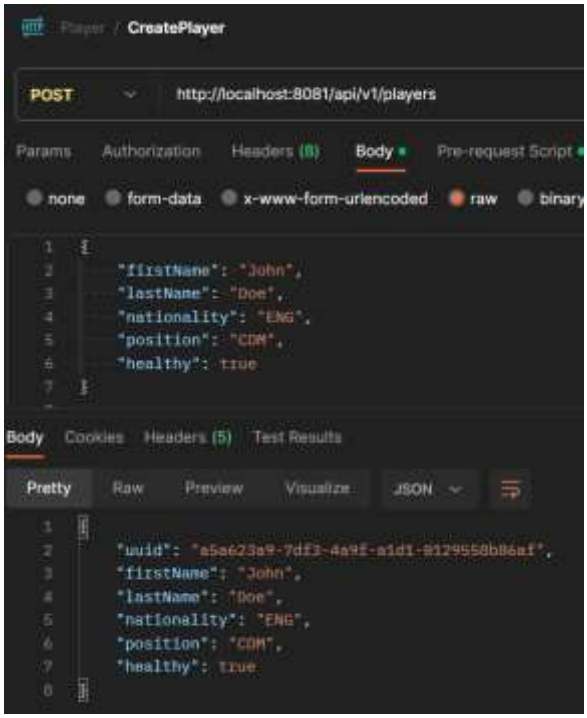
Usluge su pokrenute naredbom opisanom u odjeljku 5.1.1, a za pokretanje četiri usluge potrebna su 4 procesa. Stoga kako bi usluge bile pokrenute potrebno je otvoriti 4 naredbena retka (terminala) i za svaki zasebno otići unutar direktorija na kojemu se nalazi arhivska datoteka i pokrenuti ju. Tada je pokrenuta cijela aplikacija s arhitekturom mikrousluga koja se na računalu izvršava kao 4 različita procesa koji komuniciraju međusobno kako bi radili kao cjelina ranije objašnjenim načinima komunikacije.

5.2. ISPITIVANJE FUNKCIONALNOSTI I MIGRIRANJA APLIKACIJE

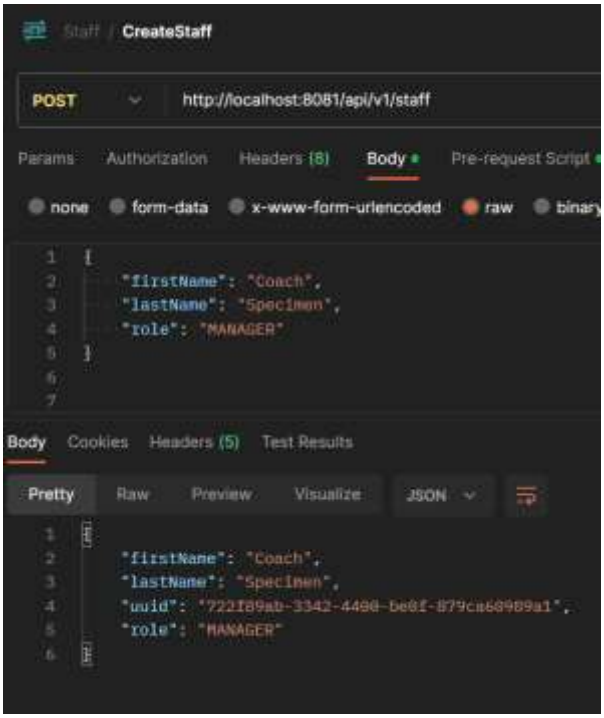
Nakon pokretanja migrirane aplikaciju kroz četiri zasebne usluge, mogu se koristiti sve njezine funkcionalnosti. Obično prilikom upotrebe aplikacija, korisnici pristupaju korisničkom sučelju putem određene klijentske aplikacije kao što su frontend aplikacija ili mobilna aplikacija. Tada se, skriveno od korisnika, izmijenjene pristupne točke prema poslužitelju samo preusmjere iz klijentskih aplikacija da pozivaju novu implementaciju na poziv korisnika. Kako je za potrebe ovog rada napravljena samo poslužiteljska aplikacija, prikaz implementiranih funkcionalnosti bit će prikazan izravnim REST API pozivima prema poslužitelju.

5.2.1. Ispitivanje usluge menadžmenta tima

Za ispitivanje valjanosti funkcionalnosti usluge menadžmenta tima pokrenuta je usluga i provjereni odzivi na postojeće pristupne točke. Za početak rada stvoreni su entiteti igrača i člana osoblja korištenjem metoda za stvaranje svakog entiteta zasebno, prikazane slikama 5.2 i 5.3.

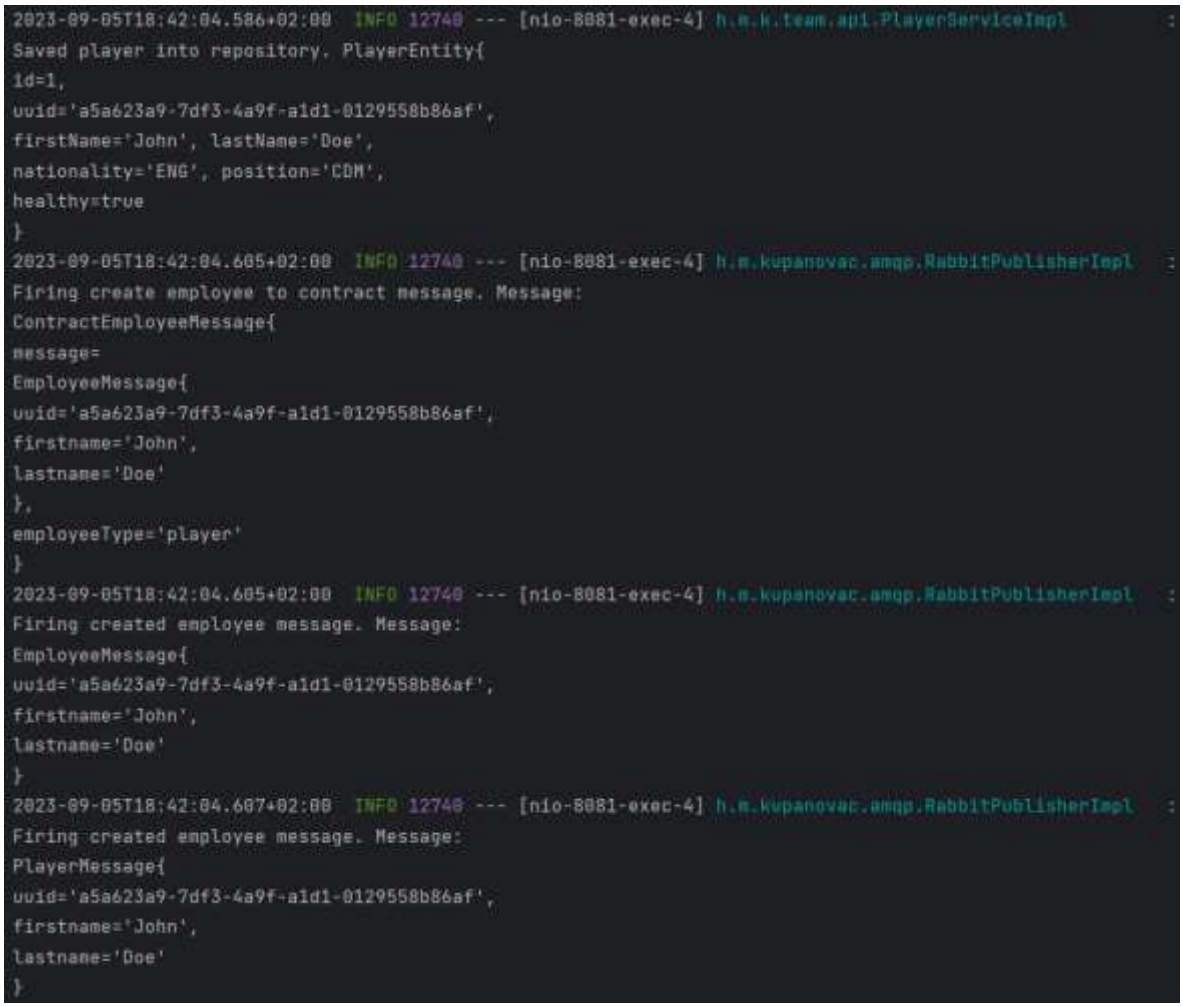


Slika 5.3: Stvaranje igrača



Slika 5.4: Stvaranje osoblja

Po stvaranju ovih entiteta, aplikacija u konzoli ispisuje implementirane zapise za praćenje događaja koji se izvršavaju. Implementirani su zapisi prilikom stvaranja novog entiteta i slanja te zaprimanja poruke. Na slici 5.5 prikazani su zapisi aplikacije nastali tijekom operacije stvaranja novog igrača sa slike 5.3.



Slika 5.5: Zapisi usluge menadžmenta tima prilikom stvaranja novog igrača

Sa slike je vidljivo kako stvaranje igrača prouzrokuje slanje 3 poruke drugim uslugama. Isto tako stvaranje osoblja šalje poruke, ali samo modulima financija i pravnog odjela, obzirom da domena medicinske skrbi brine samo o igračima. Stoga nakon dvije operacije u redovima poruka treba biti pet poruka na čekanju – tri za igrača, dvije za člana osoblja. Prikaz poruka na čekanju vidljiv je na slici 5.6.

Overview				Messages			Message rates		
Name	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**
createAppointmentQueue	classic	D	idle	0	0	0			
createContractQueue	classic	D	idle	2	0	2	0.00/s	0.00/s	0.00/s
createInjuryQueue	classic	D	idle	0	0	0			
medicalQueue	classic	D	idle	1	0	1	0.00/s	0.00/s	0.00/s
playerContractQueue	classic	D	idle	0	0	0			
resolveAppointmentQueue	classic	D	idle	0	0	0			
staffContractQueue	classic	D	idle	0	0	0			
transactionQueue	classic	D	idle	2	0	2	0.00/s	0.00/s	0.00/s

Slika 5.6: Prikaz poruka na čekanju u redovima brokera

5.2.2. Ispitivanje usluge medicinske skrbi

Nadalje za prikaz funkcionalnosti uključena je usluga medicinske skrbi. Kako usluga osluškuje red *medicalQueue*, po uspješnoj inicijalizaciji zaprimljena je poruka od brokera koju usluga obrađuje i zatim sprema igrača unutar vlastitih granica konteksta, prikazano slikom 5.7.

```
2023-09-05T18:44:55.275+02:00  INFO 16612 --- [ntContainer#0-1] h.m.k.m.api.AppointmentServiceImpl       :
Received message of type PlayerMessage:
PlayerMessage{
uuid='a5a623a9-7df3-4a9f-a1d1-0129558b86af',
firstname='John',
lastname='Doe'
}
2023-09-05T18:44:55.328+02:00  INFO 16612 --- [ntContainer#0-1] h.m.k.m.api.AppointmentServiceImpl       :
Saved player.
PlayerMedicalEntity{
id=1,
uuid='a5a623a9-7df3-4a9f-a1d1-0129558b86af',
firstName='John', lastName='Doe'
}
```

Slika 5.7: Zaprimanje poruke iz reda čekanja

Kada je spremljen igrač unutar baze podataka usluge medicinske skrbi, moguće je izvršavati funkcionalnosti za evidenciju igračevih ozljeda i pregleda.

Stvaranjem zapisa o ozljedi, sprema se u bazi podataka entitet ozljede s referencom na igrača i šalje poruka usluzi menadžmenta tima (zapis prikazan slikom 5.8) kako bi tamo bila pohranjena informacija o postojećoj ozljedi.

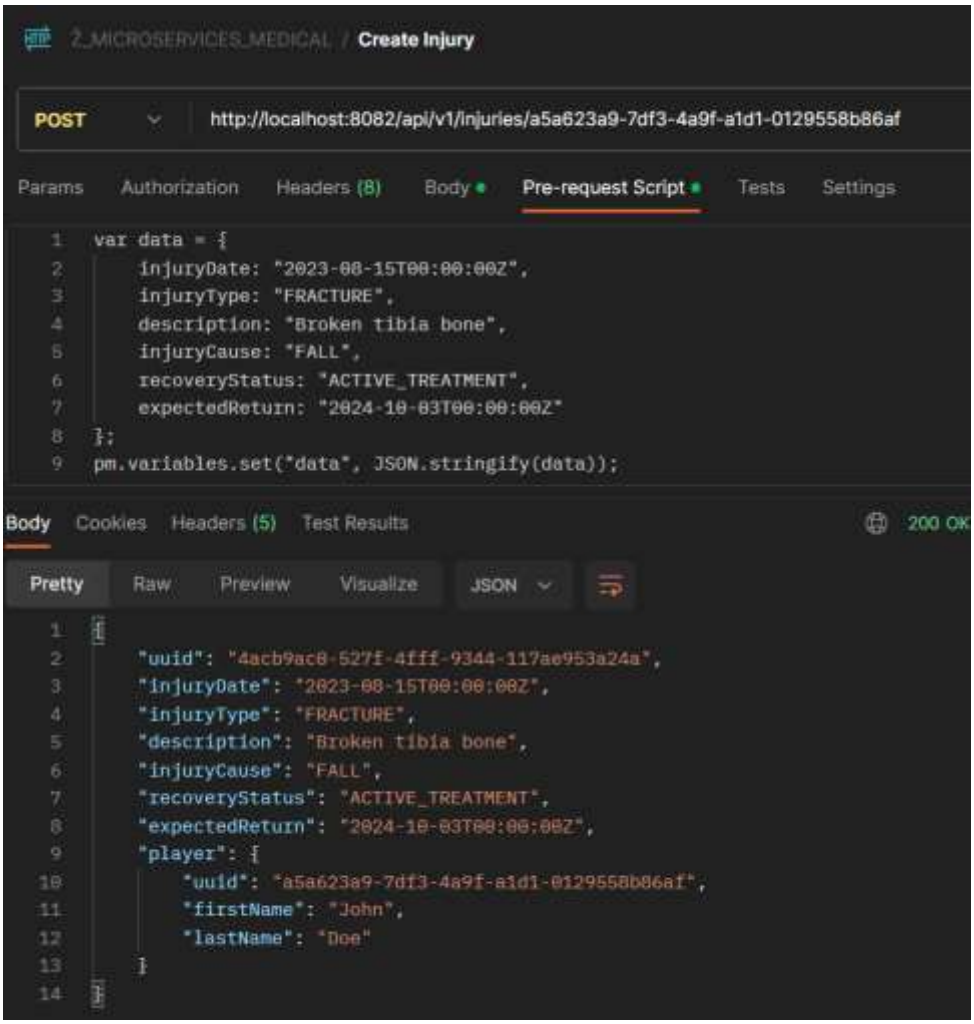
```
2023-09-05T18:58:18.454+02:00  INFO 16612 --- [nio-8082-exec-5] h.m.k.medical.api.InjuryServiceImpl      :
Created new injury.
InjuryEntity{
id=1,
uuid='4acb9ac0-527f-4fff-9344-117ae953a24a',
playerUuid='a5a623a9-7df3-4a9f-a1d1-0129558b86af',
injuryDate=2023-08-15T00:00:00Z,
injuryType=FRACTURE,
description='Broken tibia bone',
injuryCause=FALL,
recoveryStatus=ACTIVE_TREATMENT,
expectedReturn=2024-10-03T00:00:00Z
}
2023-09-05T18:58:18.463+02:00  INFO 16612 --- [nio-8082-exec-5] h.m.kupanovac.amqp.RabbitPublisherImpl    :
Firing create injury message. Message:
CreateInjuryMessage{
uuid='4acb9ac0-527f-4fff-9344-117ae953a24a',
playerUuid='a5a623a9-7df3-4a9f-a1d1-0129558b86af',
expectedReturn=2024-10-03T00:00:00Z
}
```

Slika 5.8: Zapis aplikacije na poziv funkcije stvaranja ozljede

Poslana poruka prema usluzi menadžmenta osim spremanja informacije o ozljedi mijenja i zdravstveno stanje ozlijeđenog igrača, što je vidljivo u prikazu baze podataka na slici 5.9.

Slika 5.9: promjena zdravstvenog stanja igrača nakon ozljede

U odzivu metode koji se vraća pozivatelju vidljivo je kako prikaz ozljede sadrži i podatke o igraču koji je ozlijeđen za razumljiviji prikaz svih podataka ozljede, kao što je prikazano slikom 5.10.



Slika 5.10: Tijelo zahtjeva i odziv funkcije stvaranja ozljede

Za evidenciju tretiranja ozljede korišteni su entiteta liječničkog pregleda. Pomoću njih moguće je pratiti stanje ozljede i tijek liječenja. Tako je za tretiranje nastale ozljede stvoren novi termin pregleda, koji se također šalje i usluzi menadžmenta tima za evidenciju nadolazećih pregleda igrača, čije zapise prikazuje slika 5.11.

Slika 5.11: Zapis aplikacije po stvaranju liječničkog pregleda

Kod pregleda važnija operacija je razrješenje pregleda, odnosno unos nalaza nakon liječničkog pregleda, čiji je rezultat prikazan slikom 5.12. Tu se unosi novi dokument liječničkog nalaza zajedno s podacima o ozljedi.

Slika 5.12: Zapis razrješenja pregleda

Razrješenjem ozljede ovisno o predanim parametrima u tijelu zahtjeva moguće je izmijeniti stanje ozljede. Postavlja se očekivani povratak igrača od ozljede kako bi se osvježio termin povratka nakon novog pregleda i postavlja se stanje oporavka, kao što je vidljivo u prikazanim atributima ozljede na slici 5.12, pod nazivima *expectedReturn* i *recoveryStatus* gdje je stanje promijenjeno u rehabilitaciju a termin postavljen na raniji od prethodno očekivanog.

Za evidenciju konačnog izlječenja od ozljede potrebno je kreirati novi pregled i razriješiti ga. Ovog puta prilikom razrješenja ozljede šalje se stanje oporavka kao potpuno oporavljen i očekivani povratak na trenutni datum. U zapisu aplikacije o promjenama, prikazanom slikom 5.13, vidljivo je kako je ozljeda osvježena i sada je stanje ozljede *potpuno izliječeno*.

Slika 5.13: Zapis aplikacije nakon razrješenja pregleda i oporavka od ozljede

Zaprimljenu poruku o razrješenju ozljede mikrousluga menadžmenta tima zaprima i obrađuje, unoseći promjene u različitim tablicama nakon što je pregled razriješen, a ozljeda izliječena. Uklanja se termin nadolazećeg pregleda i aktivna ozljeda, a mijenja zdravstveno stanje igrača, što je prikazano slikom 5.14. Također na slici 5.15 je vidljiv zapis iz baze podataka koji prikazuje kako je igračevo zdravstveno stanje postavljeno na istinito, odnosno stanje *zdrav*.

Slika 5.14: Zapis mikrousluge menadžmenta tima po izlječenju ozljede igrača

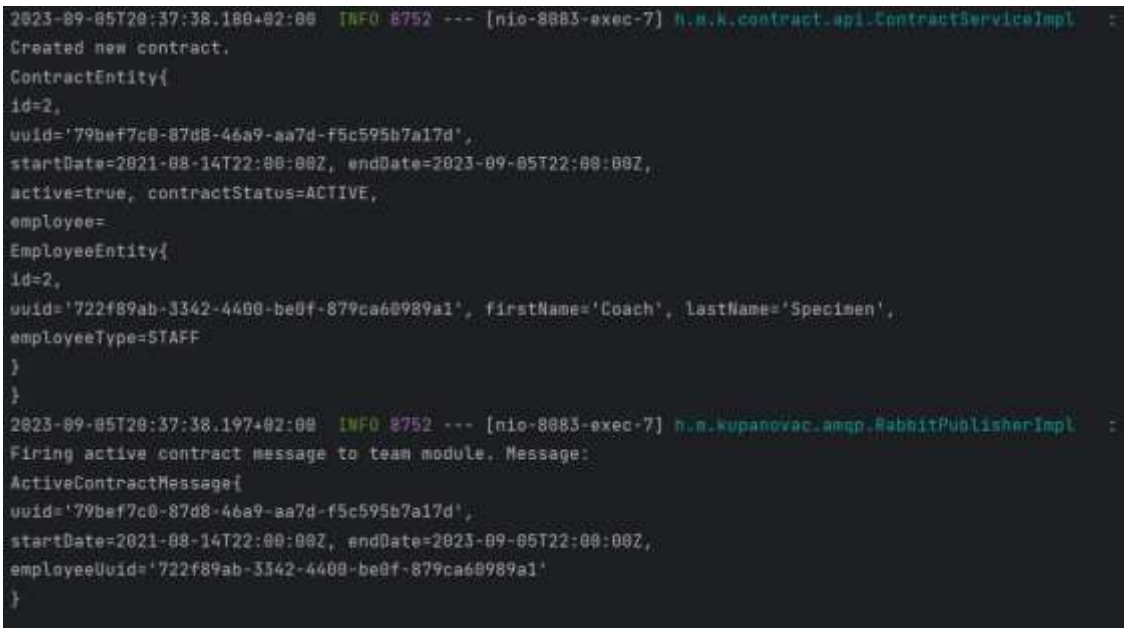
Slika 5.15: Osvježen zapis igrača u bazi nakon izlječenja ozljede

5.2.3. Ispitivanje usluge pravnog odjela

Pravni odjel služi za održavanje ugovora zaposlenih u klubu, igrača i osoblja. Ova usluga stoga ima ulogu voditi evidenciju o svim ugovorima koji su potpisani s klubom od strane zaposlenih. Po pokretanju usluge, zaprimljene su poruke o stvorenima igraču i članu osoblja čiji se podaci spremaju u tablicu zaposlenih, a zapis o zaprimljenoj poruci za stvorenog člana osoblja dan je slikom 5.16.

Slika 5.16: Zapis mikrousluge pravnog odjela o stvaranju entiteta zaposlenika

Kada se stvaraju ugovori, potrebno je referencirati zaposlenika na kojeg se odnosi ugovor. Također, kako bi se iz aplikacije kasnije mogli dohvatiti ugovori, potrebno je priložiti dokument ugovora u PDF formatu koji se sprema tijekom stvaranja ugovora u aplikaciji. Entitet ugovora sadrži atribute koji definiraju granice trajanja ugovora, informaciju o trenutnom stanju ugovora, je li na čekanju, aktivan ili istekao te referencu na zaposlenika kojemu pripada, dok su ostale informacije specifičnosti ugovora i sadržane su u dokumentu ugovora. Prilikom stvaranja novog ugovora potrebno je referencirati zaposlenika čiji se ugovor stvara i navesti granice trajanja ugovora, što je prikazano slikom 5.17.



Slika 5.17: Zapis stvaranja novog ugovora

Unutar mikrousluge implementirana je i ponavljajuća metoda koja svakog dana dohvaća ugovore koji nisu istekli i ovisno o njihovim vremenskim granicama mijenja stanje ugovora. Prvo dohvaća sve aktivne ugovore i provjerava je li im rok isteka raniji od trenutnog datuma. Ukoliko je, ugovor se označava isteknutim. Zatim se provjeravaju ugovori na u statusu čekanja i nad njima se vrši jednaka provjera s početnim datumom kako bi promijenili njihovo stanje iz čekanja u aktivno. Primjer zapisa aplikacije tijekom označavanja ugovora istaknutim u ponavljajućoj metodi dan je slikom 5.18.

Slika 5.18: Djelovanje ponavljajuće metode na izvršenje promjene statusa ugovora

Također, ponavljajuća metoda je implementirana i unutar domene menadžmenta tima te ona pronalazi ugovore koji nisu više aktivni kako bi ih obrisala, obzirom da mikrousluga menadžmenta tima treba samo informaciju o aktivnim ugovorima, ne i bilježiti sve ugovore kluba.

Slika 5.19: Brisanje neaktivnog ugovora iz mikrousluge menadžmenta tima

Nerijetko se u klubovima događa da se ugovori produljuju i prije isteka trenutnog aktivnog ugovora. Ukoliko novi ugovor ima početak raniji nego što završava trenutni ugovor, došlo bi do stvaranja dva aktivna ugovora u isto vrijeme. U tom slučaju potrebno je prethodno aktivni ugovor učiniti neaktivnim. Za prikaz dane funkcionalnosti stvoren je novi ugovor zaposleniku kako je prikazano slikom 5.20.

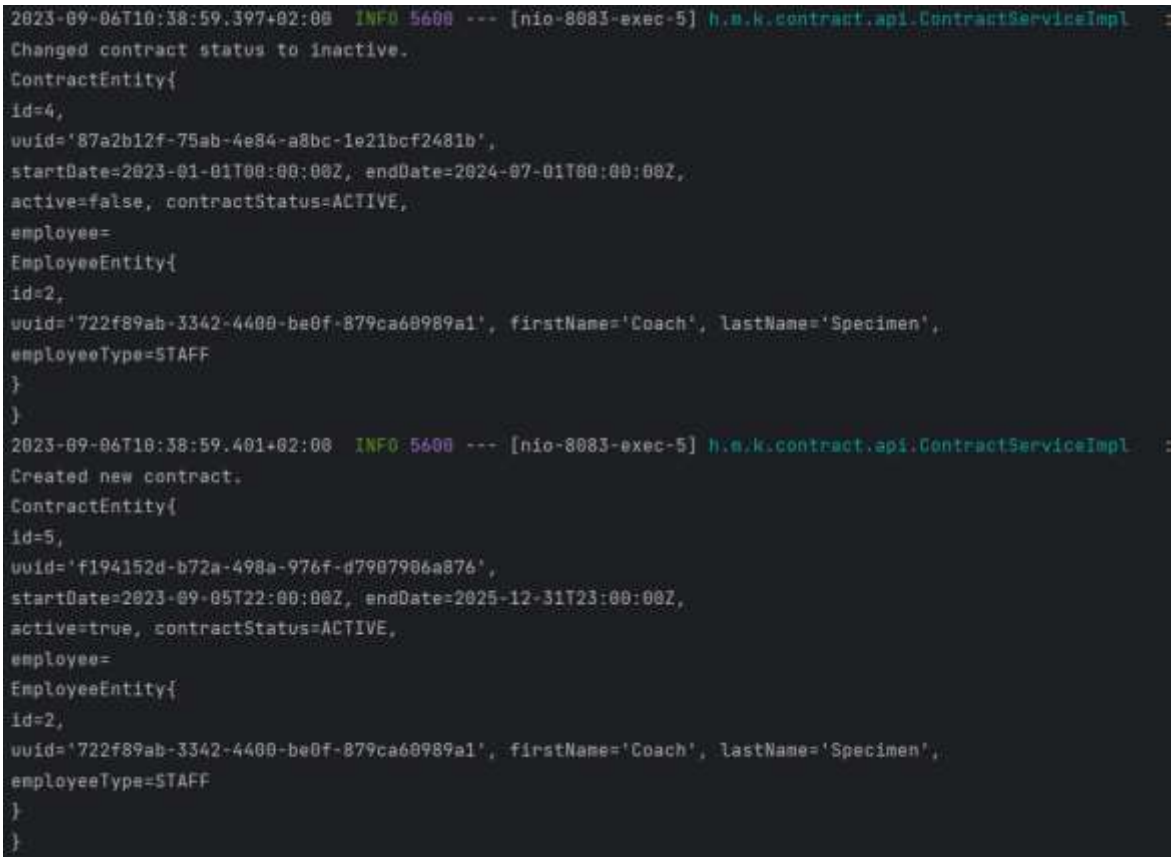
Slika 5.20: Zapis stvaranja novog ugovora

Nakon toga, prikazano je stvaranje novog ugovora koji ima početak raniji od dovršetka prethodnog ugovora sa slike 5.20. Tada aplikacija pronalazi aktivan ugovor danog zaposlenika i prvo postavlja neaktivnim postojeći ugovor, a zatim sprema novi ugovor koji je aktivan, prikazano slikom 5.21.

5.2.4. Ispitivanje usluge financija

Poput prethodno stvorenih usluga koje za izvršenje određenih funkcionalnosti trebaju referencu na igrača ili člana stožera, i ova mikrousluga zaprima poruke o stvorenim zaposlenicima kluba, kao što je prikazano na slici 5.22.

Transakcije kluba mogu biti ulazne i izlazne, odnosno da klub dobiva novac ili isplaćuje drugoj strani. Unutar aplikacije to je označeno predznakom iznosa, koji govori promjenu financijskog stanja kluba nakon transakcije. Uz ovaj podatak, entitet sadrži i vrijeme transakcije, opis, vrstu transakcije za razvrstavanje u kategorije poput plaća, održavanja, putovanja i slično. Za razlikovanje uplata zaposlenima u klubu i vanjskim strankama prisutna je i enumeracija uključene stranke te jedinstveni identifikator za transakcije koje uključuju zaposlene u klubu.

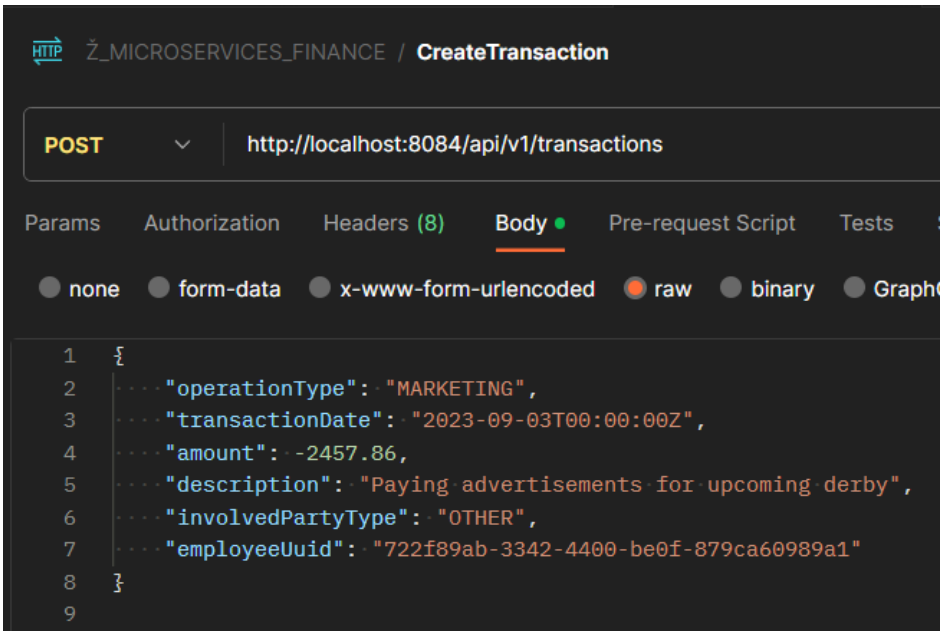


5.21: Postavljanje novog ugovora i deaktivacija postojećeg

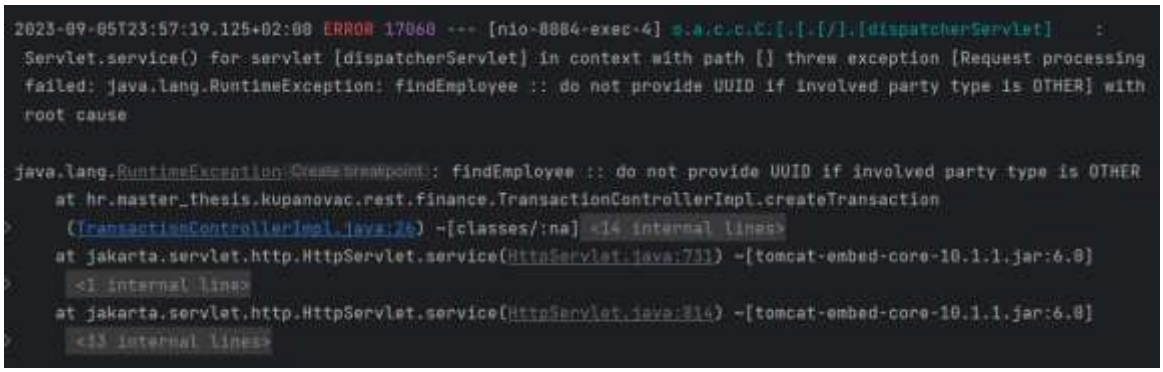
Slika 5.22: Zaprimljena poruka za novog zaposlenika

Prilikom stvaranja transakcija potrebno je unijeti navedene podatke. Ukoliko transakcija uključuje zaposlenika, nužno je unijeti i njegov identifikator inače nije moguće stvoriti transakcija. Također je implementirano ograničenje da u slučaju predavanja bilo kakvog identifikatora u zahtjevu, uključena strana ne smije biti vanjska stranka kao što je slučaj u zahtjevu sa slike 5.23, što je prikazano odgovorom na slici 5.24.

Ukoliko je stvaranje transakcije uspješno, podaci o transakciji iz entiteta se koriste za stvaranje zasebnog PDF dokumenta koji bilježi podatke o transakciji. Pomoću navedenih ograničenja stvaranja transakcija stvorena je mogućnost za detaljnije analize financijskih tokova i uspješnije upravljanje financijama kluba – transakcije mogu biti grupirane prema kategoriji, mogu se složiti razni financijski izvještaji i vršiti druge operacije koje bi pomogle klubu za što bolje financijsko poslovanje uz praćenje financijskih tokova.



Slika 5.23: Neispravan zahtjev prema trećoj strani uz identifikator zaposlenika



Slika 5.24: Odgovor aplikacije na nepravilan zahtjev za stvaranje transakcije

5.3. ANALIZA REZULTATA

Prikazanim pozivima prema migriranoj poslužiteljskoj aplikaciji dokazane su ispravno implementirane funkcionalnosti kako bi se zadržale sve postojeće funkcionalnosti koje su bile prisutne u aplikaciji s monolitnom arhitekturom.

Svaka usluga je zasebna samostalna jedinica čije funkcionalnosti se mogu izvršavati i dok druge usluge nisu aktivne, a zajedno kao cjelina čine kompletnu aplikaciju s arhitekturom mikrousluga. Ovim načinom poboljšali smo otpornost aplikacije na pogreške – ukoliko kod monolitne aplikacije dođe do pogreške koja ruši aplikaciju ili se dogodi kvar na računalnoj infrastrukturi koja pokreće aplikaciju, cijela aplikacija nije dostupna. Za razliku od monolita, aplikacija s arhitekturom mikrousluga ima raspodijeljene jedinice djelovanja aplikacije te pad, odnosno kvar ili neaktivnost jednog dijela ne uzrokuje pad cijele aplikacije nego druge usluge

mogu normalno raditi, što je prikazano pokretanjem usluga nakon što je druga usluga već izvršila određene operacije i inicirala komunikaciju prema toj usluzi.

Prikazanom komunikacijom između usluga osigurana je eventualna konzistentnost podataka – način održavanja ispravnosti i dosljednosti dijeljenih podataka između raspodijeljenih baza podataka gdje svaka baza sprema svoju kopiju dijeljenih podataka koji nisu nužno istog trenutka jednaki, no u jednom trenutku u budućnosti će sigurno biti konzistentni.

S novom arhitekturom prilikom velikog opterećenja na aplikaciju kako bi bile održane njezine performanse, moguće je zasebno prilagoditi resurse svake usluge kako bi ciljano uz manje troškove i manju potrebu za promjenom korištenih resursa aplikacija bila prilagođena da održi tražene performanse. Moguće je analizom identificirati koje usluge su više opterećene i prema tome izvršiti skaliranje, vertikalno – dodavanjem novih resursa u postojeće usluge, ili horizontalno dodavanjem većeg broja istovrsnih usluga. Ovim postupcima postignuta je veća skalabilnost aplikacije, što je jedna od vodećih prednosti migriranja na ovu arhitekturu.

6. ZAKLJUČAK

Ovaj diplomski rad prikazuje migriranje postojeće poslužiteljske aplikacije s monolitnom arhitekturom na arhitekturu mikrousluga. U radu su prvo teorijski razmotrene različite arhitekture – monolitne i raspodijeljene te prikazani prednosti i nedostaci migriranja s monolitne arhitekture na raspodijeljeni sustav kakav je aplikacija mikrousluga, a zatim i prikazani praktično, s naglaskom na prednosti koje je moguće ostvariti korištenjem arhitekture mikrousluga.

Na primjeru postojeće aplikacije za menadžment sportskog kluba prikazan je postupak migriranja monolitne arhitekture na arhitekturu mikrousluga. Aplikacija služi sportskim klubovima kako bi pomoću nje različiti akteri u klubu izvršavali obaveze i bilježili svoje djelovanje u svrhu postizanja boljih rezultata kluba. Navedena aplikacija sadrži četiri poslovne domene koje su migrirane na četiri samostalne mikrousluge. Tijekom procesa migriranja su prikazani različiti obrasci migriranja za usporedbu različitih mogućnosti migriranja. Objašnjene su ključne karakteristike postupka migriranja poput uspostavljanja granica konteksta, komunikacije između mikrousluga i uspostavljanja raspodijeljene baze podataka. Testiranjem po završetku migriranja utvrđeno je zadržavanje jednakih funkcionalnosti aplikacije i nakon promjene arhitekture aplikacije uz postizanje željenih prednosti ostvarenih migriranjem.

LITERATURA

[1] International Organization for Standardization, »ISO/IEC/IEEE 12207:2017,« 2017.

[2] R. C. Martin, Clean Architecture - A craftsman's guide to software structure and design, Prentice Hall, 2018.

[3] D. Garlan i M. Shaw, »An Introduction to Software Architecture,« School of Computer Science Carnegie Mellon University, 1994.

[4] B. Foote i J. Yoder, »Big Ball of Mud,« Department of Computer Science Universitiy of Illinois, 1999.

[5] M. Santana, A. Sampaio Jr., M. Andrade i N. Rosa, »Transparent Tracing of Microservice-based Applications,« 2019.

[6] A. S. Tanenbaum i M. Van Steen, Distributed Systems: Principles and Paradigms, Pearson Education, 2007.

[7] A. Rotem-Gal-Oz, »Fallacies of Distributed Computing Explained,« *Doctor Dobbs Journal,* 2008.

[8] M. Villamizar i H. Castro, »Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures,« lipanj 2017.

[9] N. Salaheddin i N. Ali Salem, »Microservices vs. Monolithic architectures [the differential structure between two architectures],« *International Journal of Applied Sciences and Technology,* svez. 4, br. 3, 2022.

[10] C. Harris i , »Microservices vs. monolithic architectures,« Attlasian, [Mrežno]. Available: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith#:~:text=A%20monolithic%20architecture%20is%20a,monolith%20architecture%20for%20software%20design. [Pokušaj pristupa 19 lipanj 2023].

[11] M. Fowler, »Conway's Law,« 2022. [Mrežno]. Available: https://martinfowler.com/bliki/ConwaysLaw.html. [Pokušaj pristupa 9 rujna 2023].

[12] M. Richards i N. Ford, Fundamentals of Software Architecture, O'Reilly Media, 2021.

[13] C. Bindu, »Distributed System: Microkernel Architecture Glimpse,« Medium, ožujak 2022. [Mrežno]. Available: https://medium.com/@bindubc/distributed-system-microkernel-architecture-db01f19062e2. [Pokušaj pristupa 21 lipnja 2023].

[14] M. Fowler i J. Lewis, »Microservices,« 25 ožujak 2014. [Mrežno]. Available: https://martinfowler.com/articles/microservices.html. [Pokušaj pristupa 21 lipanj 2023].

[15] V. Lähtevänoja, »Communication Methods and Protocols Between Microservices on a Public Cloud Platform,« Aalto University, 2021.

[16] C. S. Pachikkal, »Interservice Communication in Microservices,« *International Journal of Advanced Research in Science, Communication and Technology (IJARSCT),* svez. 5, br. 2, 2021.

[17] M. Kalske, N. Mäkitalo i T. Mikkonen, »Challenges When Moving from Monolith to Microservice Architecture,« Springer International Publishing, 2018.

[18] G. Blinowski, A. Ojdowska i A. PrzybyŁek, »Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation,« IEEE, 2022.

[19] D. Taibi, V. Lenarduzzi i C. Pahl, »Architectural Patterns for Microservices: A Systematic Mapping Study,« u *8th International Conference on Cloud Computing and Services Science,* 2018.

[20] M. Kalske, »Transforming monolithic architecture towards microservice architecture,« University of Helsinki, 2017.

[21] J. K. Baumgartner, »From Monolith to Microservices - Guidelines to establish an agile, scalable, and secure system architecture,« Universidade Nova de Lisboa, 2022.

[22] R. Steinegger, P. Giessler, B. Hippchen i S. Abeck, »Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications,« u *The Third International Conference on Advances and Trends in Software Engineering*, 2017.

[23] S. Newman i , Monolith to Microservices, O'Reilly, 2019.

[24] European Club Association, »ECA Club Management Guide,« 2018.

[25] L. Chung, »Non-Functional Requirements,« The University of Texas at Dalls, Department of Computer Science.

[26] »Geeks for Geeks,« 26. travnja 2023. [Mrežno]. Available: https://www.geeksforgeeks.org/non-functional-requirements-in-software-engineering/. [Pokušaj pristupa 9. kolovoza 2023].

[27] A. Smid, R. Wang i T. Černý, »Case study on dana communication in microservice architecture,« 2019.

[28] P. Bailis i A. Ghodsi, »Eventual Consistency Today: Limitations, Extensions, and Beyond,« 2013.

[29] »Major Features of Java Programming Language,« InterviewBit, ožujka 2023. [Mrežno]. Available: https://www.interviewbit.com/blog/features-of-java/. [Pokušaj pristupa 10. kolovoza 2023].

[30] »What is Java Spring Boot?,« Azure, 2023. [Mrežno]. Available: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-java-spring-boot/. [Pokušaj pristupa 10 kolovoza 2023].

[31] »What is IntelliJ IDEA?,« JetBrains, 2023. [Mrežno]. Available: https://www.jetbrains.com/idea/features/. [Pokušaj pristupa 10. kolovoza 2023].

[32] Oracle, »Java SE Documentation,« Oracle, 2018. [Mrežno]. Available: https://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html. [Pokušaj pristupa 2. rujna 2023].

ŽIVOTOPIS

Filip Kupanovac rođen je 18. lipnja 1999. godine u Osijeku. Nakon završetka svojeg osnovnoškolskog obrazovanja u Osnovnoj školi Ivana Kukuljevića u Belišću, upisuje Srednju školu Valpovo u Valpovu, smjer Elektrotehničar. Tijekom svojeg srednjoškolskog obrazovanja sudjelovao na višestrukim državnim i županijskim natjecanjima iz prirodoslovnih i tehničkih predmeta. Osim toga, u veljači 2018. godine imao je priliku sudjelovati u mobilnosti učenika unutar Erasmus projekta pod nazivom „Novim tehnologijama na tržište rada“ u španjolskoj Sevilli. Iste godine nakon završetka srednje škole sa odličnim uspjehom upisuje preddiplomski studij Računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Uz obaveze studiranja, veliku pažnju još od 2016. godine pridodaje svojoj karijeri košarkaškog suca u kojoj je 2021. godine došao do titule državnog suca, najvišeg ranga košarkaškog suca u Republici Hrvatskoj. Uz to, u slobodno vrijeme pridodaje i veliku pažnju tečajevima namijenjenim za usavršavanje programskih jezika koji su mu, uz redovno učenje na fakultetu, omogućili da savlada i dodatno usavrši programske jezike Java, C, C# te JavaScript. Unutar svojih jezičnih vještina, dosegao je zavidnu razinu znanja engleskog jezika kojim se bez poteškoća može koristiti u društvene i profesionalne svrhe, a uz njega poznaje španjolski i njemački.

_____

Potpis autora

SAŽETAK

Ovaj diplomski rad prikazuje migriranje postojeće poslužiteljske aplikacije s monolitnom arhitekturom na arhitekturu mikrousluga. Aplikacija za menadžment sportskog kluba na kojoj je prikazan postupak migriranja u svojoj strukturi sadrži četiri poslovne domene koje se migriraju na četiri samostalne mikrousluge. U radu su teorijski razmotrene prednosti i nedostaci prelaska na raspodijeljeni sustav kakav je aplikacija mikrousluga. Rad sadrži objašnjenja i korake postupka migriranja uz osvrt na ključne karakteristike provedbe postupka kao što su uspostavljanje granica konteksta, uspostavljanje komunikacije između različitih poslovnih domena i uspostavljanja raspodijeljene baze podataka. Programsko rješenje stvoreno je u razvojnom okruženju IntelliJ IDEA namijenjenom razvoju aplikacija koje se izvršavaju na Java virtualnom stroju. Aplikacija je pisana u programskom jeziku Java uz korištenje programskog okvira Spring Boot i alata za upravljanje projektima Maven. Aplikacija u konačnici sadrži četiri samostalne mikrousluge koje zajedno čine aplikaciju. Rezultati ispitivanja pokazuju kako nakon migriranja aplikacija nema promijenjenih funkcionalnosti i pruža jednake funkcionalnosti, a migriranjem su postignute prednosti koje se ostvaruju korištenjem arhitekture mikrousluga, poput veće neovisnosti komponenata sustava, bolje otpornosti na kvarove i povećane skalabilnosti aplikacije.

Ključne riječi: arhitektura programske podrške, menadžment sportskog kluba, mikrousluge, monolit, raspodijeljeno računarstvo.

ABSTRACT

This thesis demonstrates the migration of an existing server application from a monolithic architecture to a microservices architecture. The sports club management application, which illustrates the migration process, consists of four business domains that are migrated into four independent microservices within its structure. The paper theoretically discusses the advantages and disadvantages of transitioning to a distributed system such as the microservices application. The thesis includes explanations and steps of the migration process, with a focus on key implementation features, such as establishing boundaries of contexts, enabling communication between different business domains, and setting up a distributed database. The software solution was developed in the IntelliJ IDEA development environment, designed for applications running on the Java Virtual Machine. The application was written in the Java programming language, using the Spring Boot framework and Maven project management tools. Ultimately, the application comprises four independent microservices that collectively form the application. The research results indicate that after migration, the application retains its functionalities without any changes and provides the same functionality. The migration yields benefits associated with the use of microservices architecture, such as greater component independence, improved fault tolerance, and increased application scalability.

Keywords: software architecture, sports club management, microservices, monolith, distributed computing.

PRILOZI

Prilog 1. Diplomski rad u formatu .docx

Prilog 2. Diplomski rad u formatu .pdf

Prilog 3. Arhiva sa spremljenim Java arhivskim datotekama za pokretanje mikrousluga