# Sustav zdravstva od kuće zasnovan na arhitekturi mikrousluga

**Aleksić, Davor**



**SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU**
**FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA**

**Sveučilišni studij**

# SUSTAV ZDRAVSTVA OD KUĆE ZASNOVAN NA ARHITEKTURI MIKROUSLUGA

**Diplomski rad**

**Davor Aleksić**

**Osijek, 2021.**

# 1. UVOD

Za svakog IT stručnjaka koji radi u zdravstvu, programska arhitektura trebala bi biti područje interesa, a svaki projekt trebao bi započeti arhitekturnim dizajnom. Izgradnja programske podrške je poput izgradnje kuće, moramo razmisliti gdje svaka komponenta odlazi, poznavati značajke, isprobati različite alternative, pronaći koja arhitektura odgovara našim potrebama i procijeniti troškove. Također, moramo stvoriti nacrte s vrlo detaljnim specifikacijama, osobito ako je izgradnja složena kao u nekih zdravstvenih sustava kao što su EHR, LIS, RIS, CDSS i drugi klinički sustavi. Zajedničko djelovanje spomenutih sustava vrlo je složeno što znači da dobar arhitekturni dizajn u zdravstvu nije opcija nego potreba.

Zadatak ovog diplomskog rada je izraditi sustav zdravstva od kuće koji je zasnovan na arhitekturi mikrousluga. U teorijskom dijelu objašnjena je poveznicu između zdravstva i arhitekture mikrousluga, a u praktičnom dijelu je ostvareno programsko rješenje za sustav zdravstva od kuće. Također, cilj ovog rada je analizirati i opisati izazove arhitekture mikrousluga u Spring Boot i Spring Cloud tehnologijama. Implementirani sustav zdravstva od kuće se sastoji od šest mikrousluga koje su međusobno povezane i čine jednu cjelinu.

Rad se sastoji od pet poglavlja. Drugo poglavlje prikazuje stanje u području (engl. state of the art), opisuje vezu između zdravstvenog sustava i mikrousluga. Treće poglavlje opisuje idejno rješenje sustava. Točnije, opisuje teorijsku stranu sustava zdravstva od kuće. U četvrtom poglavlju opisani su programski jezici, tehnologije i programsko rješenje za sustav zdravstva od kuće. U petom poglavlju način korištenja aplikacije, testiranje mikrousluga i analiza testova.

# 2. ARHITEKTURA MIKROUSLUGA I ZDRAVSTVENI SUSTAVI - PRIKAZ STANJA U PODRUČJU

U ovom poglavlju se obrađuje prikaz stanja u području gdje će se vidjeti kakvo je stanje zdravstvenih sustava u odnosu na informacijsku tehnologiju. Također, analizirat će se programska arhitektura mikrousluga te usporediti s monolitnom kako bi se vidjele prednosti i nedostaci obje arhitekture. U zadnjem dijelu poglavlja predstavljeni su zdravstveni sustavi koji imaju potencijal postati mikrousluge.

## 2.1. Prikaz stanja u području zdravstvene industrije

Zdravstvena industrija doživjela je velike poremećaje na više frontova. Zakonodavni pritisci stavljaju naglasak na smanjenju zdravstvenih troškova, uz poboljšanje skrbi. Tvrtke koje se bave zdravstvenim znanostima doživljavaju povećanje konkurencije koja želi proširiti svoju prisutnost u zdravstvu. Nadalje, brzo usvajanje EHR-a (engl. Electronic health record), kao i širenje prijenosnih i mobilnih zdravstvenih aplikacija, stvorilo je eksploziju zdravstvenih podataka. Otvarajući nove mogućnosti za zdravstvene tvrtke i njihove konkurente, dobitnici će biti oni koji mogu brzo potaknuti i stvoriti inovacije. Kao rezultat toga, ova industrijska transformacija stavila je ponovno usredotočenost na poslovnu agilnost [1].

U svrhu modernizacije zdravstva arhitektura mikrousluga je pokazala pravi put. Mikrousluge su evolucija najboljih praksi programskih rješenja koja isporučuju rješenja za poslovanje u obliku usluga. Tvrtke u svim industrijama pa tako i u zdravstvu moraju nastojati pružiti najbolje moguće iskustvo za pacijente, pružatelje usluga, partnere i druge ključne sudionike. IT mora donijeti rješenja koja mogu pružiti cjelovito i ujednačeno iskustvo na svim kanalima. Moderniziranje razvoja aplikacija i ubrzavanje pristupa tržištu privlačan je prijedlog za svakog kupca, menadžera i programera. Na primjer, pomoću mikrousluga portalom za pacijente može se upravljati s više usluga, a zatim se neovisno skalirati u različitim instancama tako da svi dijele jednu središnju bazu podataka. Ovakav način bi drastično smanjio troškove izgradnje i održavanja portala, kao što bi omogućilo daljnju skalabilnost. Jasno je da će mikrousluge otvoriti inovativne načine suočavanja s izazovima interoperabilnosti zdravstva [2].

## 2.2. Mikrousluge

Mikrousluge imaju više definicija te prema Martinu Fowleru [3] mikrousluge nemaju preciznu definiciju. Pojam "arhitekture mikrousluga" pojavio se tijekom posljednjih nekoliko

godina kako bi opisao određeni način dizajniranja softverskih aplikacija kao paketa neovisno raspoloživih usluga. Iako ne postoji precizna definicija ovog arhitekturnog stila, postoje određene zajedničke karakteristike oko organizacije, poslovne sposobnosti, automatizacije procesa i decentralizirane kontrole programskog jezika i podataka.

Arhitektura mikrousluga je način ili stil implementacije arhitekture orijentirane na usluge, gdje su sve komponente (mikrousluge) povezane s ostatkom, te nesmetano surađuju kako bi implementirale složenije poslovne procese. Tehnički je mikrousluga aplikacija sa svim arhitekturnim slojevima (postojanost, poslovna logika, prezentacija) i uključuje komunikacijski sloj koji se koristi za komunikaciju s drugim mikrouslugama.

Mikrousluge su distribuirani sustavi koji zajedno surađuju i u kojima svaki od sustava ima svoj životni ciklus. Budući da su mikrousluge prvenstveno modelirani na poslovnim domenama, oni izbjegavaju probleme tradicionalnih arhitektura. Mikrousluge također integriraju nove tehnologije i tehnike koje su se pojavile tijekom posljednjeg desetljeća. Takve su arhitekture tolerantnije prema promjenama zahtjeva korisnika.

Prema [4] da bi počeli koristiti mikrousluge prvo je potrebno identificirati potencijalne module mikrousluga u kojima se može pronaći jedna cjelina. Definirati širinu odgovornosti za identificirane mikrousluge i razmotrite vrstu informacija koje će se prenositi. Povezati svoje poslovne procese s prethodno definiranom tehničkom funkcionalnošću, a zatim povezati tehnološke procese s poslovnim procesima. Dok traje proces stvaranja preferira se istraživanje funkcionalnosti i tehnologija koje se ne nude danas, ali u budućnosti će svakako biti poželjne. Veličina jedne mikrousluge je tema mnogih debata, može imati jednu krajnju točku (engl. endpoint) ili više. Jedan od najboljih odgovora je dao Jeff Bezos s pravilom “Two pizza team rule” koje govori bi mikrouslugu trebalo održavati i razvijati onoliko ljudi koliko se može nahraniti s dvije pizze.

Definiranje API-ja (engl. Application Programming Interface) i razrada načina na koji će se usluga koristiti su početak kod dizajniranja mikrousluga. Kad dizajn bude dovršen, razvija se maketa (engl. mock-up) ili simulacija usluga. Nakon što se simulacija pokrene način na koji je osmišljena, radi se isporuka mikrousluge.

Mikrousluge aplikacijama daju skalabilnost, prilagodljivost i složenost koje inače ne bi bile dostupne. Troškovi prijelaza na mikrousluge brzo se nadoknađuju smanjenim troškovima održavanja i daljnjeg razvoja, nudeći uvjerljiv argument zašto bi programeri trebali posezati za mikrouslugama. Prijelaz s monolitne arhitekture na onu koja radi na arhitekturu mikrousluga zahtijeva posao, ali na kraju se isplati [44].

### 2.2.1 Karakteristike mikrousluga

Kod svake definicije koja ocrtava zajedničke karakteristike, nemaju sve arhitekture mikrousluga iste karakteristike, ali očekuje se da većina mikrousluga posjeduje većinu karakteristika [5]. U sljedećim točkama su navedene najbitnije karakteristike arhitekture mikrousluga.

- Sustav je izgrađen od mnogih mikrousluga
- Svaka mikrousluga može se razvijati, mijenjati, testirati i primjenjivati pojedinačno
- Svaka je mikrousluga usredotočena na određeni fragment cijele domene
- Svaka mikrousluga može se samostalno zamijeniti
- Mikrousluge trebaju komunicirati jedni s drugima i surađivati u provedbi zadanih funkcionalnosti
- Sloboda za pojedinačno skaliranje mikrousluga na potrebnu razinu

### 2.2.2. Prednosti arhitekture mikrousluga

Čini se da su prednosti mikrousluga dovoljno jake da su uvjerile neke velike poduzetnike, poput Amazona, Netflixa i eBaya da usvoje metodologiju. U usporedbi s monolitnim dizajnerskim strukturama, mikrousluge nude:

- Tolerancija na greške
- Jednostavnija implementacija visoke dostupnosti
- Mogućnost finog skaliranja
- Jednostavnije upravljanje razvojem

Dodatna prednost koja je nastala primjenom ove arhitekture je potreba za API-ima ili nekom vrstom sučelja na svakoj komponenti. Ako su API-ji dobro dokumentirani programerima se olakšava posao prilikom održavanja i dodavanja novih funkcionalnosti [43].

### 2.2.3. Nedostaci arhitekture mikrousluga

Mikrousluge su u svijetu trend, ali arhitektura ima nedostataka. Općenito, glavni nedostatak mikrousluga je složenost koju ima bilo koji distribuirani sustav. U sljedećim točkama su nabrojani neki od najuočljivijih nedostataka kod mikrousluga:

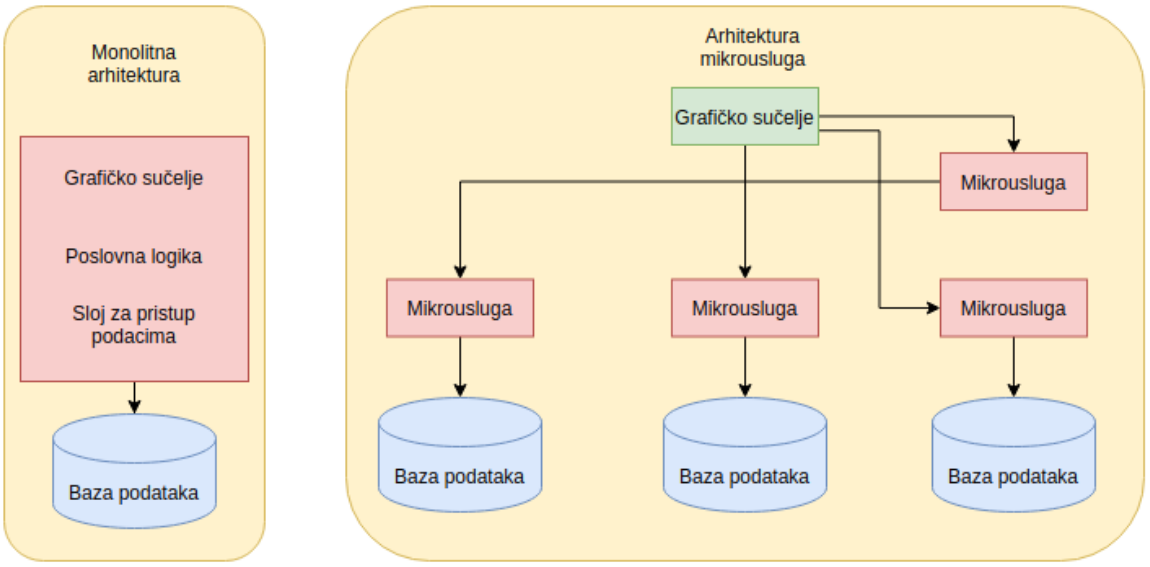
- Integracijski testovi su kompleksniji
- Isporuka (engl. deployment) je kompleksnija. Umjesto jedne aplikacije ima ih više
- Veća cijena održavanja, hardware resursa ima dosta

## 2.3. Usporedba mikrousluga s monolitnom arhitekturom

Na grčkom jeziku riječ monos znači pojedinac ili samac, a lithos znači kamen. Iz perspektive informacijskih tehnologija monolitna arhitektura je jedna aplikacija koja ima jedan izvorni kod i eventualno jednu bazu podataka. Monolitna arhitektura je tradicionalni način izrade programske podrške koji se koristi godinama u najvećim svjetskim kompanijama. U monolitnoj arhitekturi sve funkcionalnosti su ugniježđene u jednu aplikaciju te kako aplikacija postaje kompliciranija i veća puno je teže upravljati izvornim kodom i skalirati. S druge strane, monolitna aplikacija, ako nije kompliciranа, ima svoje prednosti kao što su na primjer: lakše implementiranje funkcionalnosti, jednostavnije testiranje i jednostavnija isporuka na produkcijske okoline [6].

Za usporedbu, monolitne arhitekture trebalo bi razvijati, modificirati, testirati i primjenjivati u cjelini, a skaliranje zahtijeva kopiranje cijelog monolita na različite poslužitelje i ne dopušta skaliranje pojedinih komponenata po potrebi. Nuspojava skaliranja pojedinih mikrousluga je bolja upotreba resursa poslužitelja, budući da skalirate samo one mikrousluge koje su potrebne za skaliranje, a ne cijeli monolit. To je izravno povezano s troškovima.

Za kraj ovog kratkog uvoda o monolitnoj arhitekturi ostaje pitanje jesu li monoliti loši. Nema konkretan odgovor na postavljeno pitanje, za male aplikacije nisu, nasuprot, prihvatljiviji su od arhitekture mikrousluga. Na slici 2.1. je prikazana je usporedba arhitekture mikrousluga (desno) i monolitne arhitekture (desno).




Slika 2.1. Usporedba monolitne i arhitekture mikrousluga [6]

### 2.3.1. Karakteristike monolitne arhitekture

Monolitna aplikacija izgrađena je na jednoj bazi koda s promjenjivim brojem modula. Broj modula ovisi o složenosti poslovanja i njegovim tehničkim značajkama. Cijela aplikacija i ovisnosti, izgrađene su na jednom sustavu s jednom izvršnom binarnom datotekom za implementaciju. Još neke od karakteristika su navedene u nastavku:

- Cijeli kod se nalazi u jednoj aplikaciji
- Sustav upravlja različitim nepovezanim područjima vaše domene (npr. EHR i Ljekarna)
- Podaci iz različitih nepovezanih područja pohranjuju se u istoj bazi podataka (npr. Pacijenti i lijekovi)
- Postoji samo jedna baza podataka
- Sva funkcionalnost upravljanja različitim područjima nalazi se na istom "izvršnom" programu
- Ne razmjenjuje informacije s drugim vanjskim sustavima ili je to izuzetan slučaj
- Sve protoke podataka kontrolira ista softverska komponenta
- Skalira se cijela aplikacija, u suprotnom je nemoguće skalirati

### 2.3.2. Prednosti monolitne arhitekture

Monolitna arhitektura je tradicionalni model za programsku podršku u kojem je struktura jedinstvena i nedjeljiva jedinica. Monolit ima jednu kodnu bazu s više modula. Ova je arhitektura dugogodišnji model funkcioniranja i veliki broj aplikacija uspješno je izgrađeno kao monolit. Osim što je veliki broj aplikacija izgrađen na monolitnoj arhitekturi, u sljedećim točkama je nabrojano nekoliko prednosti monolitne arhitekture:

- proces razvoja koda je lagan, sve u jednom projektu
- isporuka aplikacije na produkcijsku okolinu je jednostavna, jedna aplikacija za isporuku
- testiranje nije kompliciranо, jedna aplikacija za testirati

### 2.3.3. Nedostaci monolitne arhitekture

Kao i svako arhitekturno rješenje, monolitna arhitektura ima svoje nedostatke. U nastavku su prikazani nedostaci monolitne arhitekture:

- Kako poslovni zahtjevi za novim funkcionalnostima rastu, tako i monolitna aplikacija raste te je kompleksnost puno veća
- Može usmjeriti prema anti uzorcima - špageti kod, održavanje otežano
- Teška prilagodba - svaka najmanja promjena iziskuje cjelokupni deployment aplikacije
- Monolit je čvrsto vezan za par tehnologija, tehnologije se jako teško mijenjaju
- Teško je uvesti nove tehnologije
- CI/CD (kontinuirana integracija / kontinuirana implementacija) nije jednostavno implementirati

## 2.4. Aplikacije temeljene na arhitekturi mikrousluga

Većina velikih kompanija, kao i malih tvrtki, počinju izgradnju sustava koristeći monolitnu arhitekturu. Razlog tome je taj što u početnoj fazi projekta je brže razviti monolitnu aplikaciju i krenuti s poslovanjem. Nakon što početna faza završi, problemi su sve veći kako sustav raste. S rastom sustava, izvorni kod je kompliciraniji, arhitektura je kompleksnija te je potrebno puno više ljudi za održavanje sustava. U isto vrijeme gubi se na brzini razvoja, fleksibilnosti i mogućnosti prilagodbe onome što tržište zahtjeva.
Prije deset godina bilo je teško pronaći sustave koji su izgrađeni poštujući arhitekturu mikrousluga, ali u zadnje vrijeme broj kompanija koje su se okrenule mikrouslugama ubrzano raste. Sve poviše nabrojane probleme su imale velike svjetske kompanije poput Amazona i Netflix-a te nakon što su te dvije kompanije prošle proces prijelaza iz monolitne u arhitekturu mikrousluga, pojašnjen je koncept mikrousluga i utrt je put njihovim sljedbenicima. U nastavku su nabrojani najveći svjetski sustavi koji su zasnovani na arhitekturi mikrousluga.

### 2.4.1. Amazon

Tvrtka Amazon je jedna od prvih tvrtki gdje su mikrousluge preuzele glavnu ulogu u transformiranju cijelog poslovanja. Globalni div postigao je izvanredan uspjeh u vremenima kada je monolitna arhitektura bila "način" razvoja IT sustava. Iako Amazonova arhitektura nije bila tipični primjer monolita, sve su njegove usluge i komponente bile usko povezane te iz tog razloga Amazon više nije mogao brzo primijeniti promjene. Amazon je počeo koristiti mikrousluge za pojednostavljivanje i skraćivanje implementacije. Razbijanje struktura na pojedinačne aplikacije omogućilo je programerima da shvate gdje su uska grla, kakva je

priroda usporavanja. Također, uspostavili su male timove koji su bili posvećeni određenoj usluzi. Ono što je započelo poput čišćenja sustava, završilo je kao evolucija jednog od najvećih mrežnih igrača u modernoj arhitekturi. Zajedno s ovom promjenom, Amazon je utro put drugim tvrtkama i objavio niz rješenja otvorenog koda, poput AWS (Amazon Web Services) koja su sada sveprisutna [7].

### 2.4.2. Coca Cola

Kompanija Coca Cola koja ima podružnice u svim zemljama svijeta, suočila se s izazovom povezivanja entiteta na različitim kontinentima i potpore njihovom rastu. Globalna IT grupa Coca Cole odlučila je iskoristiti mikrousluge i API-je kako bi postigla taj cilj i postupno zamijenila njihov prijašnji sustav. U ovom slučaju brza promjena je bila nemoguća zbog više globalno implementiranih rješenja (ERP, pretvorbe, spremišta).

### 2.4.3. eBay

U 2011. godini, kada je eBay uvodio mikrousluge, tvrtka je imala 97 milijuna aktivnih korisnika i 62 milijarde bruto količine robe. Na tipični dan, eBay IT sustavi morali su se nositi s masovnim prometom, poput 75 milijardi poziva baze podataka, 4 milijarde pregleda stranica i 250 milijardi upita za pretraživanje. Promjena arhitekture mikrousluga nije bila prva velika promjena tehnologije na eBayu. Tvrtka se suočila sa sličnim prijelazima 1999. i 2005. godine.

eBay je znao da ostali konkurentni moraju ubrzavati brzu isporuku kvalitetnih značajki i inovacija. Podjela svega (baze podataka, razine aplikacija, pa čak i tražilica) i primjena arhitekture mikrousluga omogućili su eBay-u da odgovori na izazove rastuće složenosti izvornog koda, poboljšavajući produktivnost programera i omogućujući brže stavljanje na tržište zadržavajući stabilnost web mjesta. Slično drugim kompanijama koje su razvile mikrousluge, dok je rješavao vlastite probleme, eBay je objavio rješenja otvorenog koda za zajednicu programera.

### 2.4.4. Netflix

Netflix je jedan od prvih koji je usvojio mikrousluge i jedan od onih o kojima se najviše priča. Priča o okretanju Netflixa prema mikrouslugama započinje 2009. godine, kada taj pristup uopće nije bio poznat. Svoju su arhitekturu mikrousluga postavili na AWS. Njihov proces prijelaza napredovao je u koracima: prvo su premjestili kodiranje filmova i druge programe koji nisu suočeni s kupcima. Potom su razdvojili elemente koji se suočavaju s

kupcima, kao što su prijave na račun, odabir filmova, odabir uređaja i konfiguracija. Netflixu su trebale dvije godine da podijeli svoj monolit na mikrousluge, a 2011. najavio je kraj redizajniranja njihove strukture i organizacije pomoću arhitekture mikrousluga. Danas aplikacija Netflix koristi više od 500 mikrousluga i API pristupnika koji svakodnevno obrađuju preko 2 milijarde API zahtjeva.

### 2.4.5. Spotify

Do trenutka kada je Spotify imao 75+ milijuna aktivnih korisnika mjesečno, konstantno je tražio rješenje koje će se proširiti na milijune korisnika, podržati više platformi i baviti se složenim poslovnim pravilima. Željeli su biti konkurentni na tržištu koje se brzo kreće reagiranjem i htjeli su nadjačati konkurenciju. Njihovi tehnološki timovi pronašli su način da udovolje gore navedenim zahtjevima lansirajući mikrousluge kojima upravlja preko 90 autonomnih timova.

Trenutno Spotify koristi preko 810 usluga. Razdvajanjem njihovog starog sustava Spotify je omogućio izgradnju fleksibilnih struktura koje se lako mogu prilagoditi, riješiti stvarna uska grla u kratkom vremenu, lako testirati i zasebno verzionirati različita rješenja. Također su postali manje podložni velikim neuspjesima.

### 2.4.6. Uber

U svojim prvim danima, kada je Uber tek ulazio na tržište, izgradili su svoje rješenje za jednu ponudu u jednom gradu. No kako se tvrtka širila, njezin sustav, zasnovan na monolitnoj arhitekturi, počeo je stvarati probleme sa skalabilnošću i kontinuiranom integracijom. To je bio trenutak kada je Uber odlučio transformirati njihov globalni IT sustav u mikrousluge.

Unutar nove arhitekture Uber je predstavio API pristupnik i neovisne usluge koje imaju pojedinačne funkcije i koje se mogu zasebno implementirati i skalirati.

## 2.5. Najkorištenije i najpouzdanije tehnologije za razvoj arhitekture mikrousluga

Pomoću mikrousluga moguće je izgraditi pouzdanu platformu za proširenje poslovanja uz korištenje različitih programskih jezika i tehnologija. Naravno, mogućnost korištenja različitih tehnologija ne mora značiti učinkovitost. Arhitektura mikrousluga dolazi s mnogo operativnih troškova, stoga dodavanje različitih programskih jezika povrh toga može eksponencijalno povećati ulaganja. Da bi se smanjili troškovi, najbolji način je standardizirati

svoj tehnološki niz mikrousluga odabirom programskog jezika na temelju poslovnih potreba. Evo kriterija za procjenu programskog jezika za razvoj mikrousluga:

- Jezik mora podržavati jednostavnu automatizaciju
- Podrška za kontinuiranu integraciju
- Mogućnost decentralizacije komponenti
- Mogućnost jednostavnog skaliranja
- Mogućnost rastavljanja poslovnih procesa na module

Implementacija mikrousluga se može odraditi na puno različitih načina. Java, Python, C++, Node JS i .Net su samo neki od jezika preko kojih se mikrousluge kompletno mogu implementirati. Ovo su samo neki od najstandardiziranijih i najkorištenijih tehnologija za izradu mikrousluga [8]. Prema [9] u idućim potpoglavljima nabrojane su najkorištenije tehnologije u arhitekturi mikrousluga.

### 2.5.1. Docker i Kubernetes

Alat Docker je tehnologija kontejnerizacije koja pomaže u razvoju, testiranju i pokretanju softverskih sustava kao samostalnih paketa u spremniku. Kubernetes je sustav napravljen za automatizaciju ručnih zadataka angažiranih u implementaciji i rukovanju kontejnerskim aplikacijama. Korištenje kombinacije ovih tehnologija pomoći će u izgradnji responzivnog sustava mikrousluga.

### 2.5.2. REST (Representational State Transfer)

Programska arhitektura REST pomaže mikrouslugama u komunikaciji s drugim mikrouslugama. Omogućuje mikrouslugama izravnu komunikaciju putem HTTP-a. Isporučuje zahtjeve i odgovore u standardnim formatima kao što su JSON, HTML i XML. Odlična je programska arhitektura za stvaranje skalabilnih mikrousluga. U poglavlju 4.1.3. detaljnije je objašnjena programska arhitektura REST.

### 2.5.3. Prometheus

Alat Prometheus sustav za upozoravanje i za praćenje mikrousluga napravljen za više složenih topologija aplikacija koje sadrže mnoge čvorove. Ovaj alat koristi oznake "key-value" za izvršavanje višedimenzionalnih podataka i omogućuje pohranu podataka i bilješke. Prometheus je jednostavan i brz alat koji filtrira podatke ovisno o njihovim

oznakama te služi za opširnu vizualizaciju statistike i upozorenja. Također, omogućuje prikupljanje i vizualizaciju podataka te omogućuje vremenski temeljene izbore praćenja za učinkovito pronalaženje abnormalnih obrazaca. Nudi pojednostavljeno korisničko sučelje i različite bitne grafičke instrumente.

### 2.5.4. Consul

Alat Consul pomaže mikrouslugama u međusobnoj komunikaciji. Zbog svojih dodatnih značajki, izdvaja se od ostalih tehnologija za otkrivanje usluga. Zbog predloška Consul i DNS sučelja, možete koristiti Consul i s drugim tehnologijama.
Korištenje Consula za postavljanje arhitekture mikrousluga korisno je za sinkroni sustav. Uostalom, njegova infrastruktura ispunjava sve osnovne izazove sinkronih mikrousluga.

### 2.5.5. Java

Programski jezik Java je stabilan, jednostavan za čitanje i popularan programski jezik među programerima. Što se tiče izgradnje arhitekture mikrousluga njegova jednostavna sintaksa olakšava stvaranje arhitekture mikrousluga. Štoviše, Java je izvrsna opcija jer pruža korisničko sučelje, povezivanje s pozadinskim resursima i komponente modela.
Java donosi veću vrijednost u čitljivosti kada je u pitanju rad sa složenim sustavima. Glavni Java programski okviri koji podržavaju izgradnju arhitekture mikrousluga su Dropwizard, Spring Boot, Spark i Eclipse Microsprofile.

### 2.5.6. .NET

.NET je cross-platforma za mikrousluge. Omogućuje korištenje pouzdanog i ustaljenog jezika koji održava Microsoft. Ugrađeni Docker spremnici pomažu u razvoju mikrousluga. .NET mikrousluge se mogu jednostavno integrirati s aplikacijama napisanim na Node.JS, Javi ili bilo kojem drugom jeziku. To postupno pomaže pri prelasku na .NET core tehnologiju. .NET mikrousluge također mogu koristiti s uslugama u oblaku.
Microsoft održava stabilnu i sigurnu platformu pod nazivom Azure koja je također prikladna za .NET. Ovo pruža hibridnu metodu koja će pomoći da se neki moduli pokreću i u oblaku.

### 2.5.7. Python

Ovaj programski jezik na visokoj razini aktivno podržava integraciju s različitim tehnologijama. Python omogućuje brzo i jednostavno prototipiranje u usporedbi s drugim jezicima i okvirima.

Python je kompatibilan sa naslijeđenim jezicima, poput PHP-a i ASP -a. Razvojni programeri koriste RESTful API metodu pri izvođenju mikrousluga Python. Neki uspostavljeni Python okviri povoljni za razvoj web aplikacija uključuju Django, Bottle, Falcon, Flask, CherryPy i NameKo.

## 2.6. Zdravstveni informacijski sustavi

Zdravstveni informacijski sustav odnose se na sustav dizajniran za upravljanje podacima u zdravstvu. Uključuju sustave koji prikupljaju, pohranjuju, upravljaju i prenose pacijentovu elektroničku medicinsku dokumentaciju. Također, uključuju operativno upravljanje bolnicom ili sustav koji podržava odluke o zdravstvenoj politici.

Zdravstvene informacijske sustave mogu koristiti svi u zdravstvu, od pacijenata preko djelatnika u zdravstvu do službenika za javno zdravstvo. Oni prikupljaju podatke i sastavljaju ih na način koji se može koristiti za donošenje zdravstvenih odluka [10]. Jedni od najzastupljenijih zdravstvenih sustava su navedeni u idućim poglavljima. Preciznije rečeno naveden je set mikrousluga koje najviše koriste zdravstvene ustanove.

### 2.6.1. Elektronički zdravstveni karton

Sustav za prikupljanje kliničkih podataka koji dolaze iz različitih sustava (aplikacije s medicinskim zapisima, LIS, RIS, farmacija, itd..) pa je ovo u osnovi centralizirani i jedinstveni EHR (Electronic Health Record) svakog pacijenta na platformi. Neke od osnovnih prednosti povezanih s EHR uključuju mogućnost lakog pristupa računalnim zapisima što je povijesno mučilo zdravstvo. Sustavi EHR-a mogu uključivati mnoge potencijalne sposobnosti, ali dvije posebne funkcionalnosti obećavaju poboljšanje kvalitete skrbi i smanjenje troškova na razini zdravstvenog sustava [11].

### 2.6.2. Ljekarnički informacijski sustav

Ljekarnički informacijski sustav (engl. Pharmacy Information System) je sustav koji ima mnogo različitih funkcija radi održavanja opskrbe i organizacije lijekova. Može biti zaseban sustav samo za upotrebu u ljekarnama ili se može koordinirati sa sustavom za unos naloga u bolnici CPOE (engl. computerized provider order entry). Ljekarnički informacijski sustav je uparen s CPOE omogućuje lakši prijenos informacija. Ovaj sustav upravlja zalihama, prima narudžbe, dostavlja lijekove pacijentima na različitim internim službama bolnice ili klinike [5].

### 2.6.3. Računalno omogućen unos naloga

Računalom omogućen unos naloga davatelja usluga (engl. computerized provider order entry, CPOE) odnosi se na postupak unosa i slanje upute za liječenje, uključujući lijekove, laboratorijske i radiološke narudžbe putem računalne aplikacije, a ne putem papira, faksa ili telefona.

Također, ponekad je potrebna komponenta za usmjeravanje narudžbi u odgovarajući sustav i primanje rezultata ili povratnih informacija od njih. CPOE se nalazi usred EHR-a (Upravljanje kliničkim informacijama) i sustava za punjenje narudžbi (LIS, RIS, farmacija itd.).

CPOE sustavi korisnicima omogućuju unos narudžbi (npr. za lijekove, laboratorijske testove, radiologiju, fizikalnu terapiju) u računalo, a ne na papiru. CPOE uklanja potencijalno opasne medicinske pogreške uzrokovane lošim znanjem ili lošim rukopisom liječnika. Također, čini postupak naručivanja učinkovitijim jer medicinsko osoblje i ljekarničko osoblje ne mora tražiti pojašnjenja niti tražiti informacije koje nedostaju iz nečitljivih ili nepotpunih narudžbi [12]. Prethodne studije pokazuju da se ozbiljne pogreške u lijekovima mogu smanjiti čak 55% korištenjem CPOE sustava, a za 83% u integraciji sa CDS sustavom koji stvara upozorenja na temelju onoga što liječnik odredi [13]. Korištenje sustava CPOE, osobito ako je povezan s CDS-om, može rezultirati poboljšanom učinkovitosti i djelotvornosti.

### 2.6.4. Sustav podrške kliničkih odluka

Sustav podrške kliničkih odluka (engl. clinical decision support system, CDS) omogućuje definiranje, upravljanje i izvršavanje pravila koja se odnose na CDS, a koja će rezultirati nekom vrstom radnje, poput prikazivanja upozorenja ili podsjetnika u stvarnom vremenu na aplikacijama Medicinski zapisi ili slanje obavijesti e-poštom u paketu, npr. kada se u bolnici otkrije više od 5 slučajeva neke vrste infekcije. Sustavi podrške kliničkim odlukama imaju za cilj pružiti adekvatne informacije liječniku potrebne za liječenje pacijenta. Bolnice i zdravstveni djelatnici se stalno tjeraju na povećanje prihoda dok se, pritom, moraju učinkovito brinuti o pacijentima. Sve više se smanjuje interakcija s pacijentima. Sustavi podrške kliničkim odlukama mogu olakšati opterećenje liječnika i pružiti skrb zasnovanu na protokolu i standardima za koju se pokazalo da poboljšava ishode liječenja pacijenata [14].

### 2.6.5. Sustav terminologije

Sustav terminologije (engl. terminology system) pruža usluge za pristup standardnim terminima koje se koriste za bilježenje i analizu kliničkih podataka pohranjenih u sustavu za upravljanje kliničkim informacijama.

Ovi standardi rješavaju temeljni zahtjev za učinkovitu komunikaciju. Predstavljaju koncepte na nedvosmislen način između pošiljatelja i primatelja informacija. Većina komunikacije između zdravstvenih informacijskih sustava oslanja se na strukturirane rječnike, terminologije, skupove kodova i klasifikacijske sustave koji predstavljaju zdravstvene koncepte [15]. Standardna terminologija pruža temelj za komunikaciju poboljšavajući učinkovitost razmjene informacija. Korištenje standardne terminologije trebalo bi biti jednostavan i logičan korak u informacijskim tehnologijama u zdravstvu. Međutim, s obzirom na složenost mnogih dijagnostičkih sustava kliničkog narativa prenesenog putem PDF formata, različitih sustava kodiranja i cijene digitalizacije svezaka referentnog materijala/rječnika, implementiranje sustava terminologije nije jednostavan proces. U nastavku su nabrojane neke od standardnih kodnih listi koje se koriste u zdravstvu.

- **LOINC (engl. Logical Observation Identifiers Names and Codes)** - univerzalni su kodni sustav za identifikaciju zdravstvenih mjerenja, opažanja i dokumenata. Ovi kodovi najviše služe za pomoć pri testiranjima ili mjerenjima. LOINC kodovi mogu se grupirati u laboratorijska i klinička testiranja, mjerenja i opažanja [16].
- **UCUM (engl. The Unified Code for Units of Measure) -** kodni sustav namijenjen uključivanju svih mjernih jedinica koje se koriste u međunarodnoj znanosti, inženjerstvu i poslovanju kako bi se olakšala nedvosmislena elektronička komunikacija [17].
- **SNOMED CT (engl. Systematized Nomenclature of Medicine-Clinical Terms)** - je sveobuhvatni klinički zdravstveni terminološki proizvod, u vlasništvu i distribuciji tvrtke SNOMED International. Omogućuje dosljedan, obradiv prikaz kliničkog sadržaja u elektroničkim zdravstvenim zapisima. Ovi kodovi često predstavljaju "odgovor" za test ili mjerenje na LOINC kod "pitanja" [18].
- **NDC (engl. National Drug Code)** - vodi Uprava za hranu i lijekove (engl. Food and Drug Administration, FDA) i sadrži popis svih lijekova proizvedenih, pripremljenih, složenih ili prerađenih za komercijalnu distribuciju [19].
- **RxNorm** - daje normalizirane nazive za kliničke lijekove i povezuje svoja imena s mnogim rječnicima lijekova koji se obično koriste u softveru za upravljanje

ljekarnama i interakcijama s lijekovima. Pružanjem veza između ovih rječnika, RxNorm može posredovati u porukama između sustava koji ne koriste isti softver i rječnik [20].

- **RedLex** - RadLex je jedinstveni jezik radioloških pojmova za standardizirano indeksiranje i pronalaženje radioloških izvora informacija. Objedinjuje i nadopunjuje druge leksikone i standarde, poput SNOMED-CT-a i DICOM-a [21].
- **MEDICIN -** je medicinska terminologija, koju održava Medicomp Systems, a obuhvaća simptome, povijest, fizički pregled, testove, dijagnoze i terapije [22].
- **CPT** - skup kodova, opisa i smjernica namijenjenih opisivanju postupaka i usluga koje izvode liječnici i drugi pružatelji zdravstvene zaštite. Svaki postupak ili usluga označeni su peteroznamenkastim kodom [23].

### 2.6.6. Radiološki informacijski sustav

Radiološki informacijski sustav (engl. Radiology Information System, RIS) koristi se za unos narudžbi, zakazivanje i upravljanje radnim popisom za svaki način imaginologije i slanje rezultata (radiološka izvješća) u EHR (Upravljanje kliničkim informacijama) [24]. RIS ima nekoliko funkcionalnost:

- **Upravljanje pacijentima** - RIS može pratiti cijeli tijek rada s pacijentom u odjelu radiologije. Radiolozi mogu dodati slike i izvješća u EHR, gdje ih može dohvatiti i pregledati ovlašteno radiološko osoblje.
- **Raspored** - RIS omogućuje osoblju zakazivanje termina i za stacionarne i za ambulantne bolesnike.
- **Praćenje pacijenata** - koristeći RIS sustav, pružatelji usluga mogu pratiti cijelu pacijentovu radiološku povijest od prijema do otpusta i uskladiti povijest s prošlim, sadašnjim i budućim terminima.
- **Ispisivanje izvještaja** - RIS može generirati statistička izvješća za jednog pacijenta, skupinu pacijenata ili određene postupke.
- **Praćenje kliničkih slika** - tradicionalno, radiologija koristi RIS za praćenje pojedinačnih snimki i njihovih povezanih podataka. No, kako su EHR-ovi postali standardni u zdravstvenoj industriji, a digitalizirane slike i PACS (engl. picture archiving and communication system) široko prihvaćeni, radiološki odjeli i njihovi RIS-PACS sustavi više su uključeni u klinički tijek rada.

- **Funkcija za naplaćivanje** - RIS sustavi pružaju detaljno vođenje financijskih evidencija i obradu elektroničkih plaćanja i automatiziranih potraživanja, iako se te funkcije uključuju u sveukupne sustave EHR-a.

### 2.6.7. Laboratorijski informacijski sustavi

Laboratorijski informacijski sustavi (engl. Laboratory Information Systems, LIS) koriste se za unos narudžbi, raspoređivanje i slanje rezultata natrag u EHR (Upravljanje kliničkim informacijama). LIS ima ključnu ulogu u laboratorijima koji zadovoljavaju standarde kvalitete. Smanjenju pogreške u transkripciji, skraćuju vrijeme obrade od primitka uzorka do dobivanja rezultata i poboljšavaju ishod liječenja pacijenata. U posljednjem desetljeću tehnološki napredak laboratorijske instrumentacije doveo je do veće količine uzoraka te veće potražnje i oslanjanja na laboratorijske podatke koji podržavaju kliničke i potrebe javnog zdravlja [25].

### 2.6.8. Sustav za arhiviranje slika i komunikacijski sustav

Sustav za arhiviranje slika i komunikacijski sustav (engl. picture archiving and communication system, PACS) pohranjuje i pruža usluge za pristup multimedijskim kliničkim podacima (npr. CT snimkama) i povezanim radiološkim izvješćima. PACS sustav nije samo sustav koji arhivira slike, jer može upravljati stvarima poput valnih oblika (EKG, ultrazvuk), a također i videozapisima (npr. Angiografija). PACS je najčešće integriran s:

- ulazom s digitalnih ili analognih uređaja, što može biti bilo koji radiološki modul, npr. rendgen, CT, MRI ili ultrazvuk
- uređajem za prikupljanje slike
- uređajem ili poslužiteljem za pohranu slika za kratkoročno ili dugoročno skladištenje podataka
- prijenosnom mrežom (lokalna ili web)
- zaslonskom postajom: radna stanica za obradu slika i korisničko sučelje
- kamerom
- radiološkim informacijskim sustavom (RIS) i bolničkim informacijskim sustavom (EHR)

### 2.6.9. Aplikacije za medicinske zapise

Aplikacije koje omogućuju bilježenje novih kliničkih podataka i pristup povijesnim zapisima iz sustava za upravljanje kliničkim podacima. Svaka medicinska specijalnost može imati vlastitu, specijaliziranu aplikaciju.

### 2.6.10. Autentifikacija i autorizacija korisnika

Komponenta infrastrukturne sigurnosti za upravljanje korisnicima i dozvolama za različite sustave i usluge na platformi.

- Autentifikacija se odnosi na verifikaciju korisnika, odnosno omogućuje korisniku prijavu u sustavu. Za autentifikaciju je potrebno korisničko ime i lozinka.
- Autorizacija se odnosi na pitanje što korisnik može raditi u sustavu. Na primjer uređivati i brisati pojedine dokumente ne mogu svi korisnici nego samo oni koji imaju dozvole, tj. određenu ulogu koja omogućuje navedene radnje.

Autentifikacijom i autorizacijom potrebno je rukovati u svakoj mikrousluzi, a ovu mikrouslugu potrebno je više puta implementirati u ostale usluge. Iako možemo koristiti bazu koda za ponovnu uporabu dijela koda, to će zauzvrat uzrokovati ovisnost svih mikrousluga o određenoj bazi koda i njezinoj verziji, što utječe na fleksibilnost izbora jezika/okvira mikrousluga.

Mikrousluge trebaju slijediti načelo jedinstvene odgovornosti te globalna logika autentifikacije i autorizacije ne bi se trebala stavljati u implementaciju ostalih mikrousluga [26].

### 2.6.11. Pomoćne usluge

Iznad su nabrojani najkorišteniji sustavi u zdravstvu. Iako je još mnoštvo sustava na globalnoj i lokalnoj razini ovo su još neki popularni sustavi koji služe kao pomoć ostalim sustavima:

- **DICOM Viewer** - jedna vrsta naprednog preglednika koji se koristi za dijagnoze. Ponekad je integrirano u aplikacije za medicinsko snimanje.
- **Kontrolni trag (Audit trail)** - zapisnik događaja koji su se dogodili u različitim aplikacijama i uslugama platforme i omogućuje otkrivanje neobičnih aktivnosti i kontrolu cijele platforme
- **Sustav za raspored (Scheduling)** - svaki resurs ili usluga u bolnici ili klinici treba biti rasporediv, tako da se dodjeljuje određenom pacijentu u određenom vremenskom

okviru. Budući da mnoge aplikacije i usluge na platformi zahtijevaju planiranje resursa, mogli bismo to odvojiti u određenu mikrouslugu na platformi.

### 2.6.12. Izazovi arhitekture mikrousluga u zdravstvenim sustavima

Postoje mnoge kulturne i psihološke zapreke za učinkovitu primjenu mikrousluga u području zdravstvene zaštite. Vrlo često se u zdravstvenim organizacijama s internim IT timovima i kod mnogih pružatelja softvera koji su prije pružali upravljački i računovodstveni softver za bolnice i klinike događa da koriste zastarjelu metodologiju dizajna arhitekture te sada ne znaju odgovoriti na zahtjeve modernih vremena.

Kao rezultat toga što se ljudi nisu obučili za dizajn moderne softverske arhitekture i pokušali primijeniti nove arhitekturne tehnike, koje bi mogle raditi na sustavima koji nisu povezani sa zdravstvom, na kraju imamo visoko povezane monolite, koje je teško modificirati i testirati, nisu tolerantni na kvarove i ne skaliraju se dobro [27].

U sljedećim točkama nabrojane su karakteristike koje blokiraju izgradnju kvalitetnog softvera:

- Arhitekture koje se ne temelje na standardima, obrascima dobrih praksi dizajniranja arhitekture.
- Nema stručnjaka za softversku arhitekturu, koji se stalno ažurira na tom području.
- Ne pregledavanje dizajna arhitekture kada se rade velike izmjene sustava.
- Ne ažuriranje dokumentacije o arhitekturi kada se sustav mijenja.

# 3. MODELIRANJE I IDEJNO RJEŠENJE SUSTAVA

U trećem poglavlju je opisano idejno rješenje sustava i razrada problematike mikrousluga u zdravstvu. Modelirano rješenje je napravljeno u ovisnosti od zahtjeva sustava koji su navedeni u idućem potpoglavlju. Također, ovo poglavlje obuhvaća opis svih mikrousluga i zahtjeva na pojedinu mikrouslugu te način komunikacije mikrousluga međusobno i s klijentom. Zadnje potpoglavlje donosi skicu i prijedlog kompletnog zdravstvenog sustava zasnovanog na arhitekturi mikrousluga.

## 3.1. Zahtjevi na sustav

Kako bi poslovnu ideju pretvorili u rješenje koje radi potrebno je definirati zahtjeve koji će pomoći da se ideja sprovede do kraja. Najčešći zahtjevi su:

- **Poslovni zahtjevi -** uključuju zahtjeve na višoj instanci, stanje ciljeva u pojedinim fazama projekta te potrebe projekta
- **Zahtjevi na kupca** - kupac pomaže kod savjetovanja i govori očekivanja koja se trebaju ispuniti
- **Zahtjevi na sustav** - dvije su vrste zahtjeva na sustav: funkcionalni i nefunkcionalni. Funkcionalni zahtjevi opisuju način ponašanja sustava. Nefunkcionalni osiguravaju kvalitetu sustava i opisuju općenite karakteristike.

Na slici 3.1. je prikazana piramida koja prikazuje zahtjeve po razini razrađenosti pa tako npr. poslovni zahtjevi su najmanje razrađeni, a zahtjevi na sustav imaju najviše traženih funkcionalnosti.


Slika 3.1. Razine zahtjeva (piramida) [28]

### 3.1.1. Funkcionalni zahtjevi na sustav

Funkcionalni zahtjevi na sustav opisuju ponašanje sustava i uključuju značajke i funkcije proizvoda koje razvojni programeri moraju dodati rješenju. Takvi zahtjevi bi morali biti precizni i za razvojni ti i za ostale sudionike u razvoju [28].
Ako tim koristi Agilnu (engl. agile) metodologiju, većinu zahtjeva će se oblikovati u pisanom obliku. Ipak, kako bi neke zahtjeve predstavili jasnije, tim ih može vizualizirati.
Funkcionalni zahtjevi na sustav zdravstva od kuće su podijeljeni po ulogama. Tri uloge su moguće: admin, liječnik i pacijent. Funkcionalni zahtjevi su prema ulogama navedeni u nastavku:

1. **Admin uloga**
    - Prijava u sustav
    - Odjava iz sustava
    - Kreiranje liječnika
    - Kreiranje pacijenta
    - Pregled svih liječnika
    - Pregled svih pacijenata
    - Brisanje liječnika
    - Brisanje pacijenta
    - Dodavanje lijekova
    - Pregled svih lijekova
    - Brisanje lijekova
    - Upravljanje narudžbama
    - Dodavanja termina dolaska pacijenta
    - Brisanje termina dolaska pacijenta
2. **Liječnik uloga**
    - Registracija liječnika
    - Prijava liječnika
    - Odjava liječnika
    - Pregled svih pacijenata
    - Pregled zakazanih termina od strane pacijenata
    - Izdavanje recepta
    - Pregled povijesti bolesti pacijenta
    - Narudžba lijeka

- Pregled svih lijekova

3. **Pacijent uloga**
    - Registracija pacijenta
    - Prijava pacijenta
    - Odjava pacijenta
    - Pregled svih liječnika
    - Prijava kod određenog liječnika
    - Zakazivanje termina kod liječnika
    - Otkazivanje termina
    - Ispisivanje recepta
    - Mogućnost narudžbe lijeka
    - Pregled svih lijekova
    - Ispis narudžbe
    - Otkrivanje dijabetesa

### 3.1.2. Nefunkcionalni zahtjevi na sustav

Iskustvo tima ili razvojnog programera je jako bitno kod definiranja nefunkcionalnih zahtjeva na sustav. Nefunkcionalni zahtjevi se nekada ne mogu odmah definirati, da bi se identificirali moraju se analizirati performanse proizvoda i nakon toga proizvod učiniti prikladnim i korisnim. Takvi se zahtjevi mogu pojaviti kada se proizvod redovito koristi[21]. Nefunkcionalni zahtjevi na sustav zdravstva od kuće su:

- Skalabilnost
- Jednostavnog održavanja koda
- Kvalitetno iskustvo korisnika (engl. user experience)
- Pouzdanost
- Sigurnost
- Mogućnost testiranja grafičkog sučelja
- Mogućnost testiranja koda
- Jednostavno uvođenje novih usluga
- Brzina odgovora sustava
- Mogućnost testiranja performansi

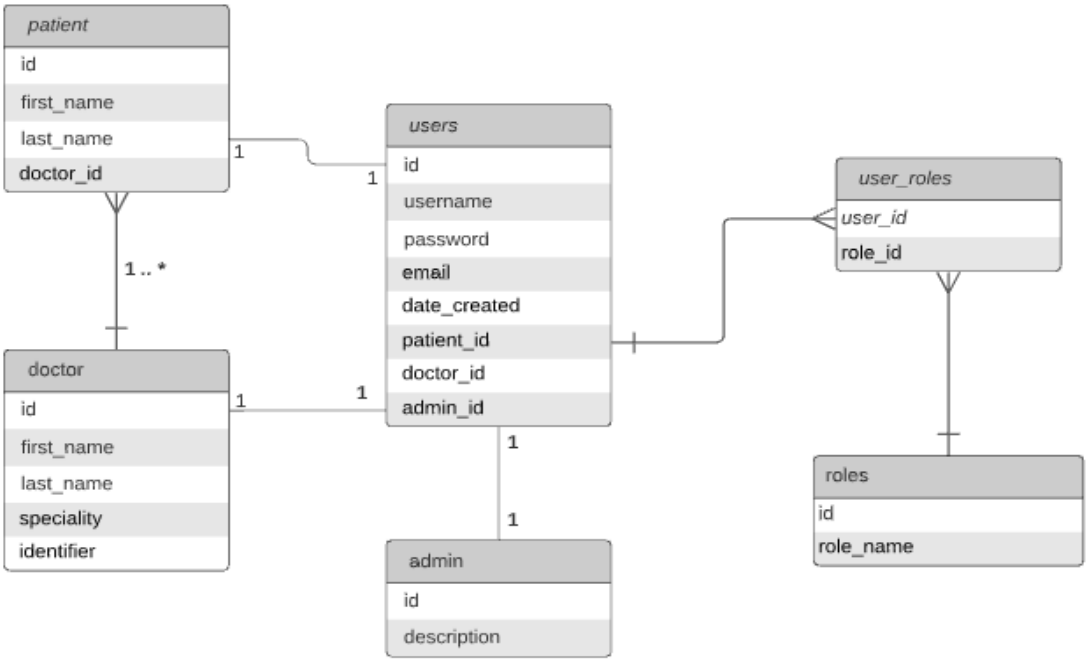
## 3.2. Idejno odgovaranje i opis mikrousluga

U ovom potpoglavlju je izrađeno idejno rješenje sustava i opisane su sve postojeće mikrousluge. U sustavu treba omogućiti registriranje i prijavu pacijenta i liječnika, naručivanje na preglede, pristup medicinskim zapisima pacijenata, izdavanje recepata, preporuke terapije lijekovima (recepti), kao i procjenu rizika obolijevanja od dijabetesa.

### 3.2.1. Usluga za upravljanje korisnicima "Users service"

Usluga za upravljanje korisnicima jedna je od najvažnijih mikrousluga i komunicira sa svim mikrouslugama jer za sve ostale mikrousluge je potrebna interakcija s korisnicima. Usluga za upravljanje korisnicima se razvija prva kako bi ostale mikrousluge mogle dohvaćati korisnike.

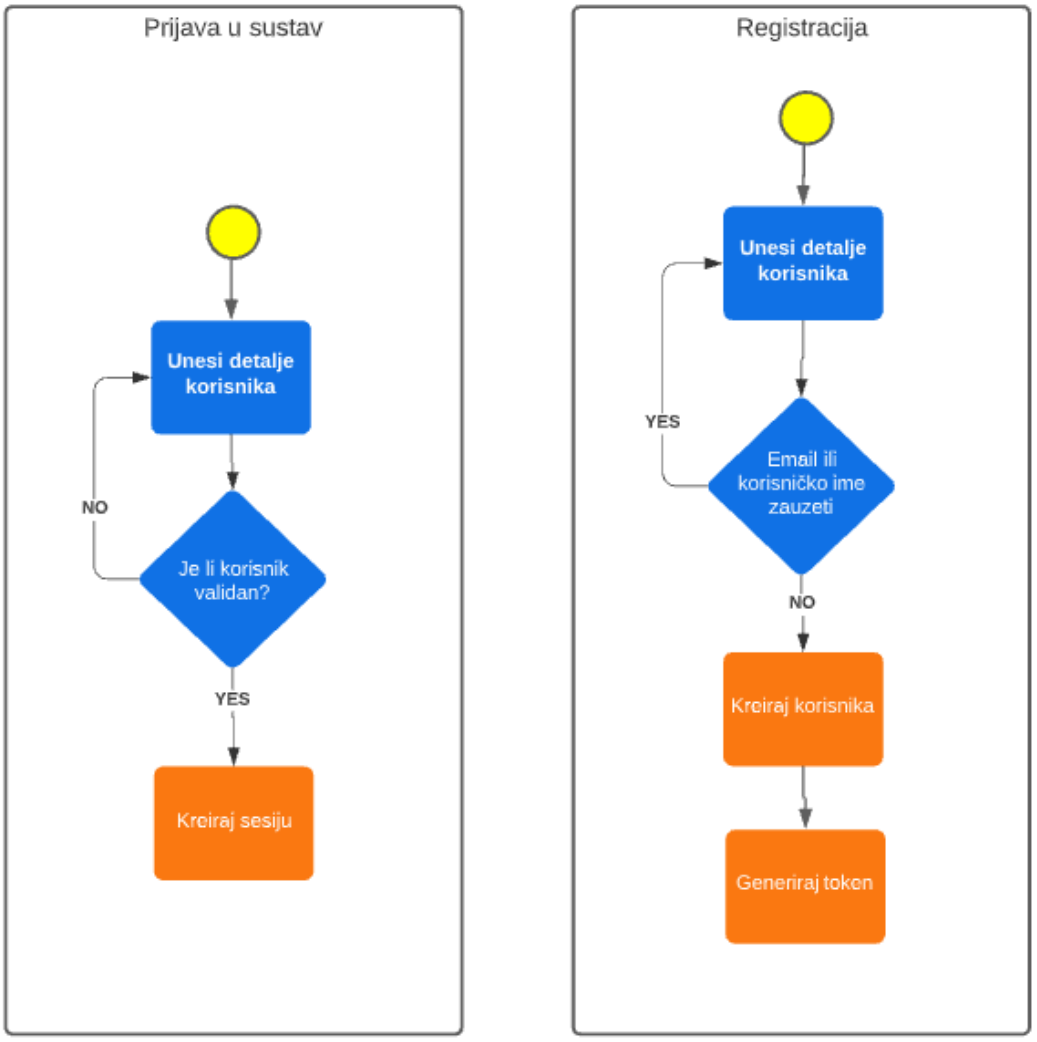
Usluga za upravljanje korisnicima se sastoji od pet entiteta: Korisnik (engl. user), Admin (engl. admin), Liječnik (engl. doctor), Pacijent (engl. patient), Uloga (engl. role). Entiteti Admin, Doctor i Pacijent su spojeni vezom 1 na 1 s entitetom Korisnik. Entitet Uloga je spojen vezom više na prema više jer jedan korisnik može imati više uloga, a jednu ulogu može imati više korisnika. Na slici 3.2. prikazan je ER dijagram baze podataka.




3.2. ER Dijagram baze podataka usluge za upravljanje korisnicima

Odgovornosti usluge za upravljanje korisnicima su:

- Kreiraj novog korisnika (pacijenta ili liječnika)
- Prijava u sustav kao admin, pacijent ili liječnik
- Dohvaćanje detalja o korisniku
- Brisanje korisnika
- Dohvaćanje svih pacijenata
- Dohvaćanje svih liječnika
- Ažuriranje pacijenta
- Ažuriranje liječnika




3.3 Dijagram toka koji prikazuje prijavu u sustav i registraciju

Nefunkcionalni zahtjevi na sustav su:

- Kada se korisnici prijave, za njih se generira JWT token koji vrijedi 24 sata. Pružanjem ovog tokena za sljedeće zahtjeve mogu izvesti operacije koje zahtijevaju provjeru autentičnosti
- Lozinke su šifrirane kriptografskim algoritmom (Bcrypt)
- Za interakciju s uslugom upravljanja korisnika predviđen je RESTful API s JSON formatom za razmjenu podataka

### 3.2.2. Usluga za upravljanje lijekovima "Pharmacy management service"

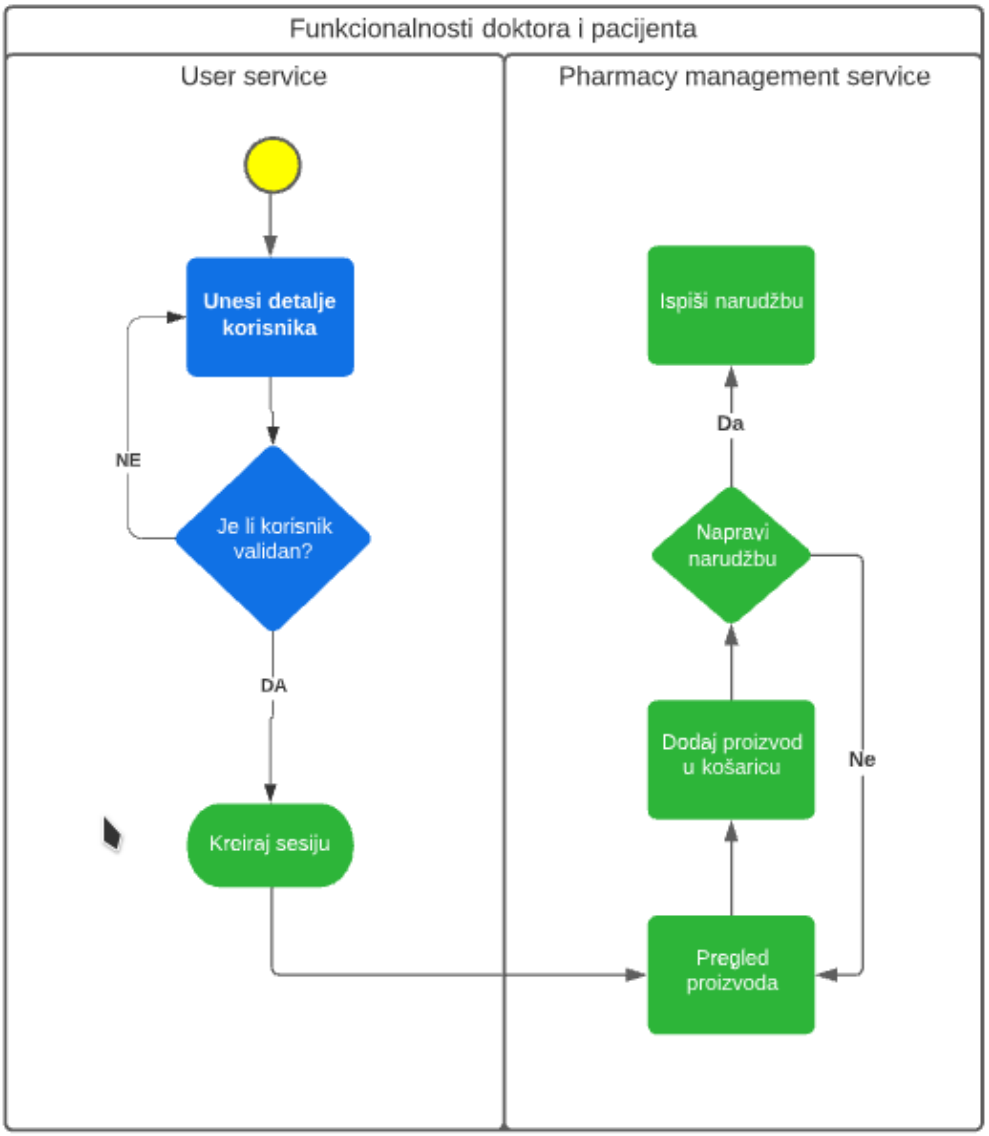
Usluga za upravljanje lijekovima usluga koja pohranjuje podatke i omogućuje funkcionalnost koja organizira i održava lijekove. U ovom modulu moguće je upravljanje lijekovima, kategorijama lijekova, tvrtkama koje proizvode lijekove i narudžbama lijekova.

Ljekarnici većinu svog radnog vremena provode izdajući lijekove. Ovaj zadatak zahtjeva veliku koncentraciju, provjeru interakcije s lijekovima i mogućnost pogreške kada se npr. krivo pročita liječnikov rukopis.Uz uspostavljenu komunikaciju između usluga, sustavom se lako rukuje receptima, što ljekarnicima i liječnicima oslobađa više vremena za interakciju s pacijentima.

Funkcionalnosti ove usluge su:

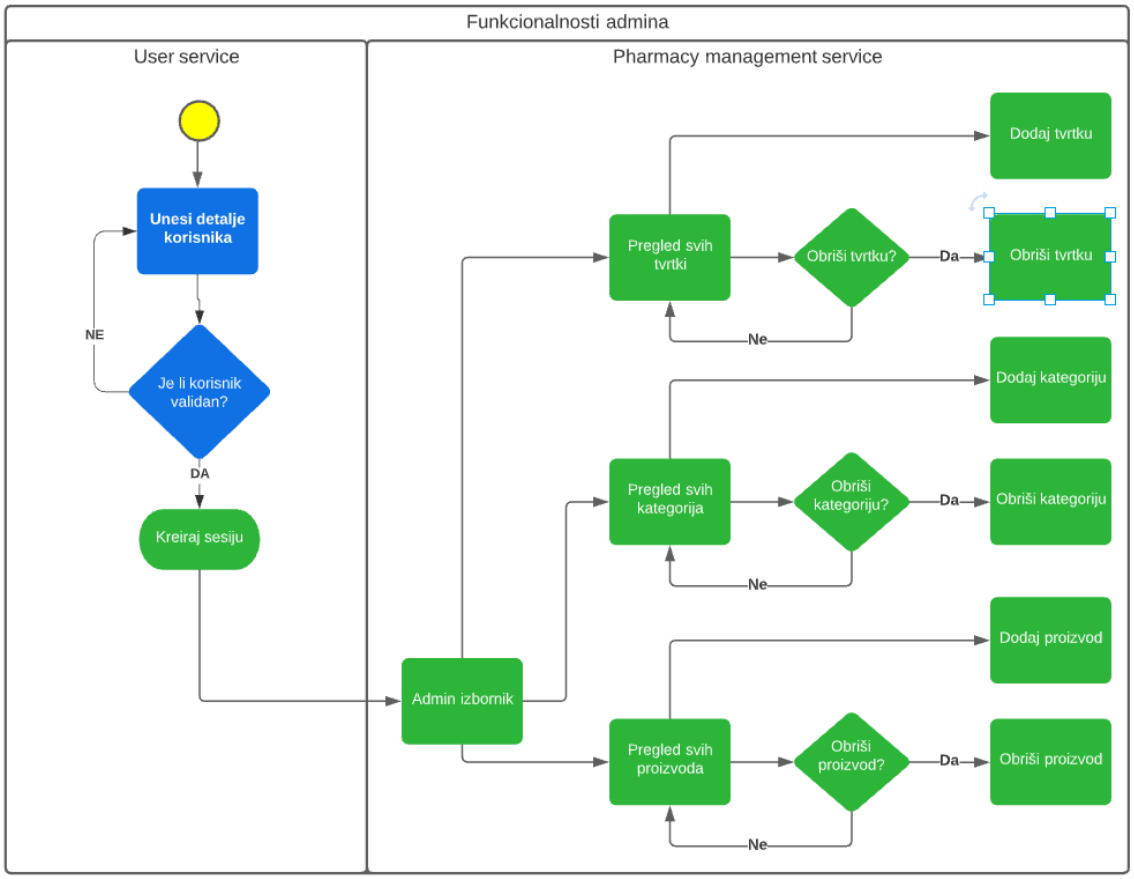
- Funkcionalnost naručivanja lijeka koji se može naručiti bez ili s receptom
- Lijek mogu naručiti liječnici i pacijenti
- Dodavanje, brisanje i ažuriranje lijekova
- Dodavanje brisanje i ažuriranje tvrtki koje proizvode lijekove
- Dodavanje brisanje i ažuriranje kategorija lijekova
- Operacije dodavanja, ažuriranja i brisanja odrađuje admin
- Ispisivanje računa
- Ispisivanje narudžbe u PDF
- Slanje e-pošte

Uslugom mogu upravljati tri vrste korisnika: admin, liječnik i pacijent. Uloge liječnik i pacijent ne mogu dodavati, brisati i ažurirati lijekove. Spomenute uloge mogu pregledavati proizvode, dodavati ih u košaricu i napraviti narudžbu. Također, osim s uslugom za upravljanje korisnicima usluga komunicira i s uslugom za upravljanje receptima. Nakon što se pacijentu prepiše recept isti može kliknuti na narudžbu lijeka. Na slici 3.4. prikazan je dijagram toka za ulogu pacijenta i liječnika.




Slika 3.4. Dijagram toka za pacijenta i liječnika

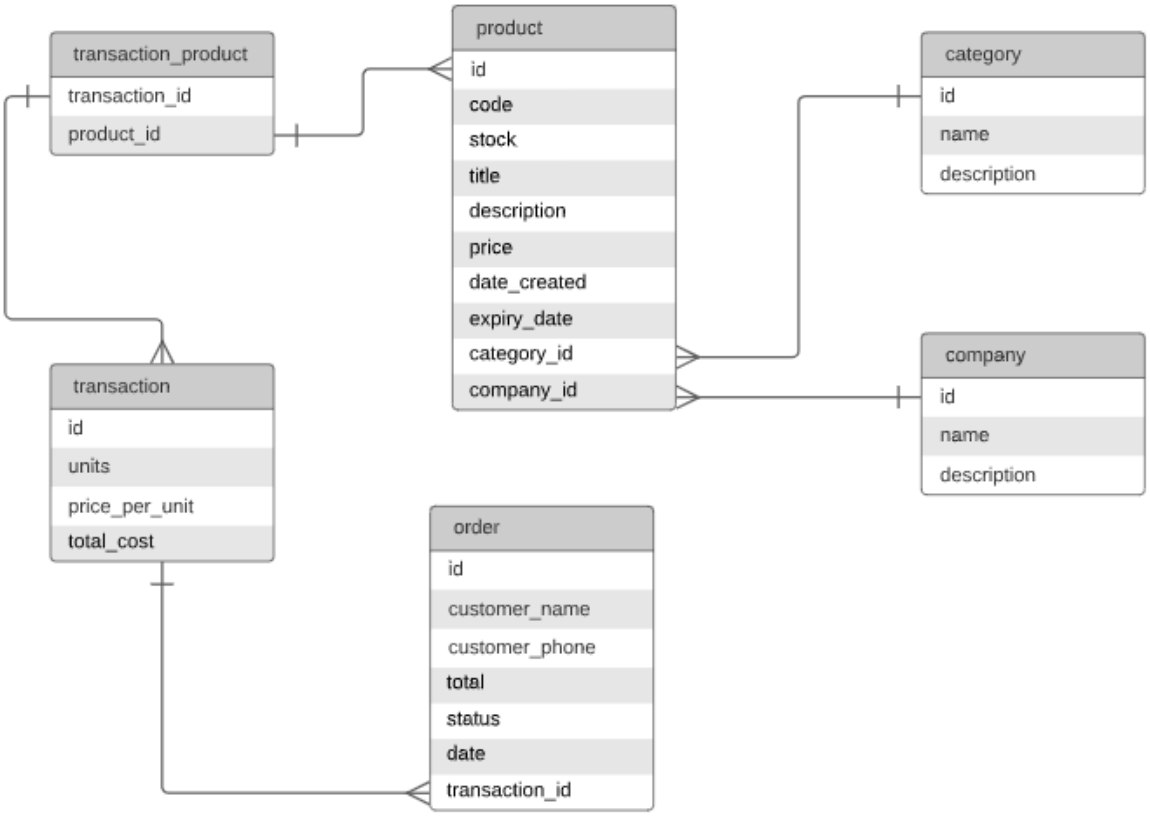
Osim liječnika i pacijenta postoji i admin uloga koja može dodavati, brisati i uređivati lijekove, kategorije lijekova i tvrtke koje proizvode lijekove. Dijagram toka za opisane funkcionalnosti admina je prikazan na idućoj slici. Također, usluga može poslati elektronsku poštu korisniku koji je naručio lijek i može ispisivati detalje narudžbe u PDF tekstualni format. Na slici 3.5. prikazan je dijagram toka za ulogu admin.




Slika 3.5. Dijagram toka admina

Baza podataka se sastoji od pet entiteta:

- Proizvod (engl. product) - glavni entitet u bazi na kojeg su spojena tri entiteta. Sadrži sve informacije o proizvodu.
- Transakcija (engl. transaction) - entitet koji je vezan za Proizvod s vezom više na više što znači da se u više transakcija može imati više proizvoda i obrnuto.
- Kategorija (engl. category) - spremaju se kategorije lijekova, vezana za entitet Proizvod vezom jedan na prema više što znači da jedna kategorija može imati više proizvoda
- Tvrtka (engl. company) - entitet u koji se spremaju tvrtke koje proizvode lijekove, vezane su za entitet Proizvod vezom jedan na prema više što znači da jedna tvrtka može imati više proizvoda.
- Narudžba (engl. order) - entitet koji je vezan za entitet Transakcija. Sadrži informacije o narudžbi i vezan je vezom jedan na prema više što znači da jedna transakcija može imati više narudžbi




Slika 3.6. ER dijagram usluge za upravljanje lijekovima

### 3.2.3. Usluga za naručivanje pacijenata “Appointment service”

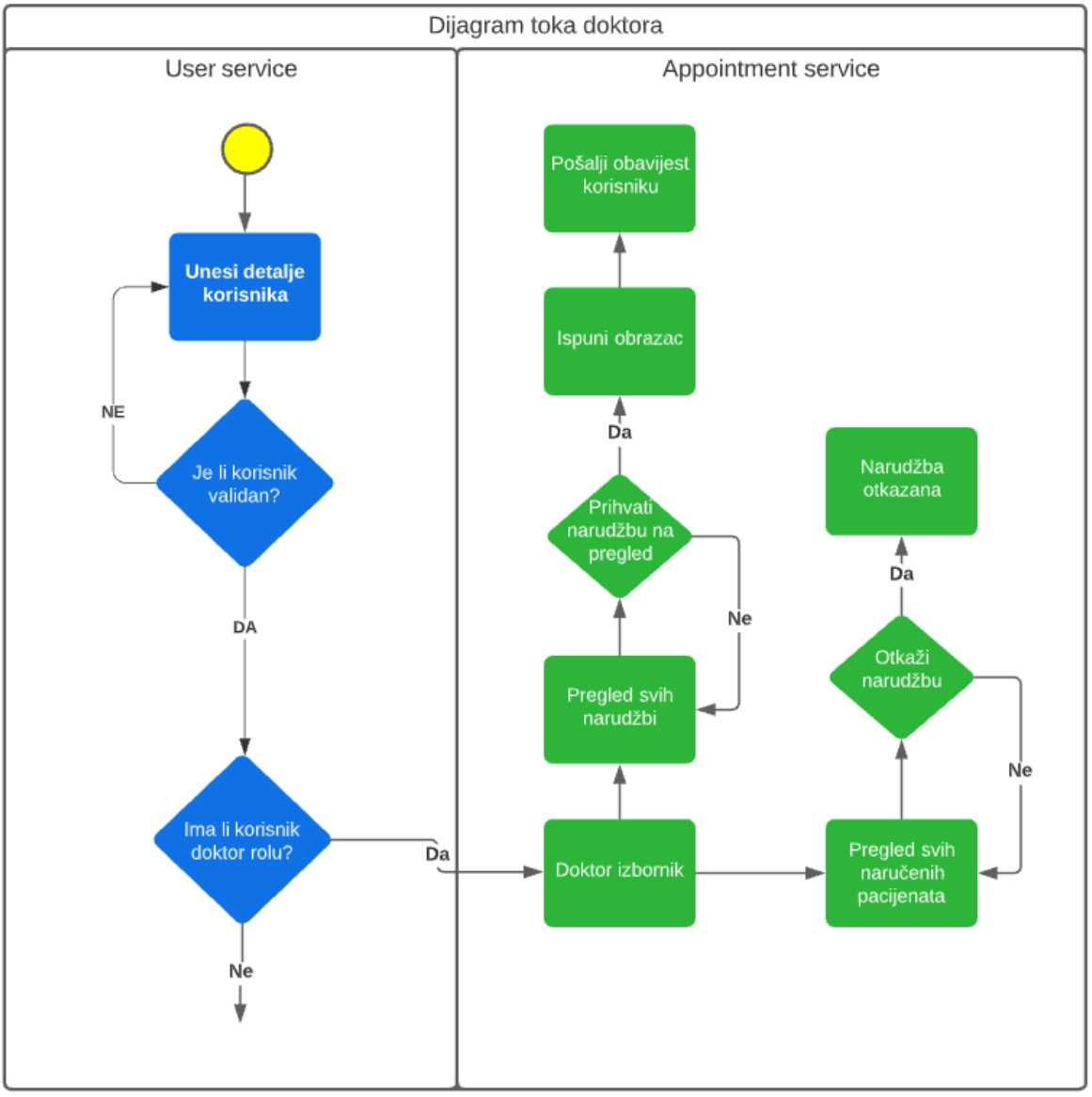
U ovoj usluzi su obrađeni najčešći problemi koji zahvaćaju bolnice i domove zdravlja, a to je duga čekanja na pregled kod liječnika. Usluga za naručivanje pacijenata omogućuje zakazivanje narudžbi kod pacijenata i prihvaćanje ili otkazivanje narudžbi kod liječnika. Duga čekanja i minimalne informacije opterećuju mnoge pacijente. Nepoznavanje kada treba biti poslužen stresan je faktor koji šteti pacijentovom iskustvu. Uz sustav zakazivanja, pacijent može bolje planirati svoj dan jer uklanja neizvjesnost u pogledu vremena čekanja. U ovom modulu moguće je upravljati posjetama pacijenata i ispisivati račune. Usluga za naručivanje pacijenata je usluga koja najviše komunicira s drugim uslugama. Osim s uslugom za upravljanje korisnicima ova usluga komunicira s uslugom za upravljanje receptima i uslugom za upravljanje kartonima.

Komunikacija s uslugom za upravljanje receptima se ostvaruje prilikom prepisivanja recepta za vrijeme posjete. Također, za vrijeme trajanja posjete pacijenta liječnik može poslati zahtjev

prema usluzi za upravljanje kartonima pacijenta te vidjeti zdravstvenu povijest pacijenta. Na slici 3.8. prikazan je dijagram toka za ulogu liječnik.

Funkcionalni zahtjevi na uslugu za ulogu liječnik su:

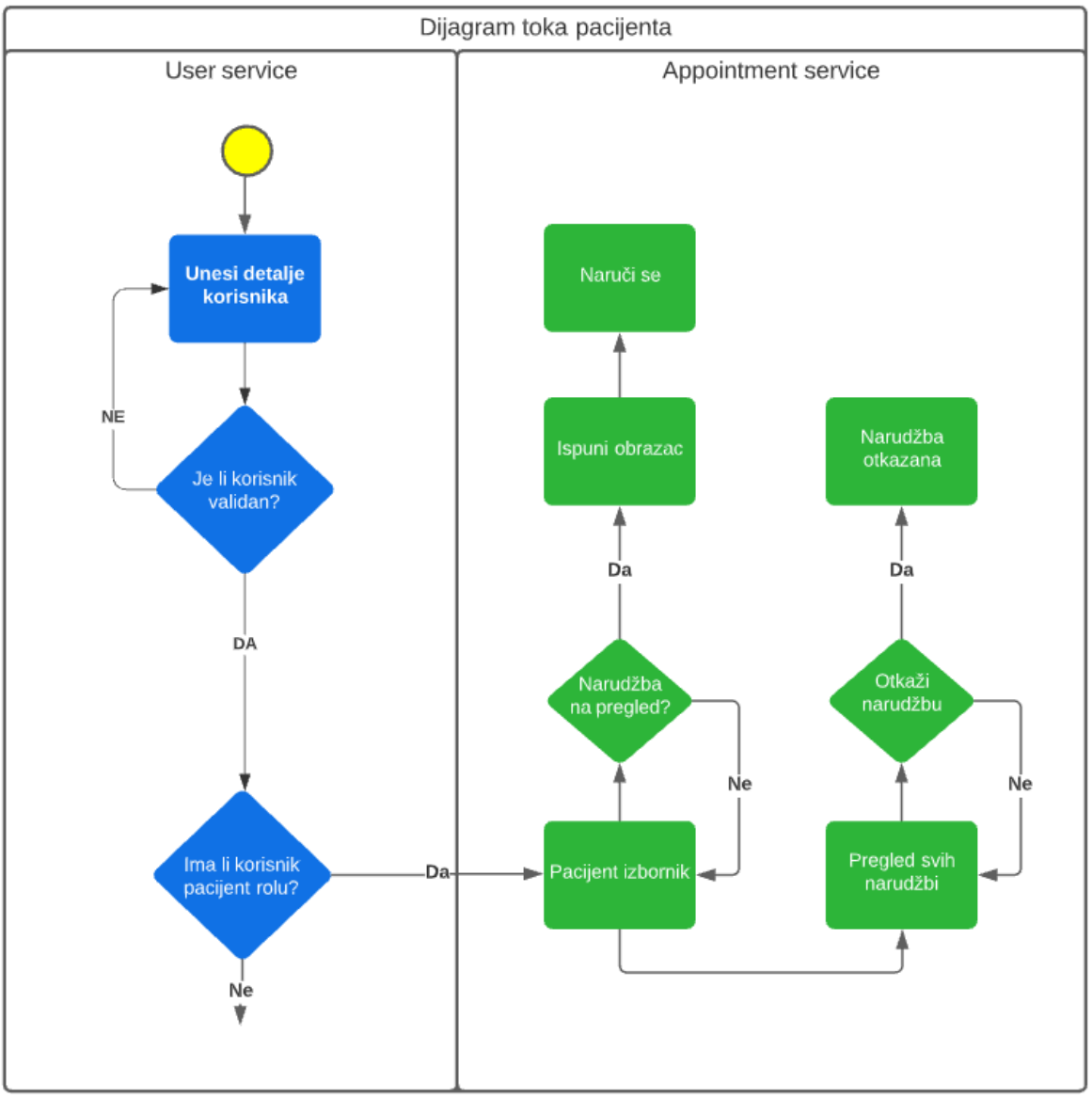
- Liječnik ima ispis svih naručenih pacijenata
- Liječnik ima ispis svih pacijenata koji su predali zahtjev za narudžbom
- Liječnik može prihvatiti narudžbu
- Liječnik može otkazati narudžbu
- Slanje e-pošte prema liječniku i pacijentu




Slika 3.7. Dijagram toka liječnika

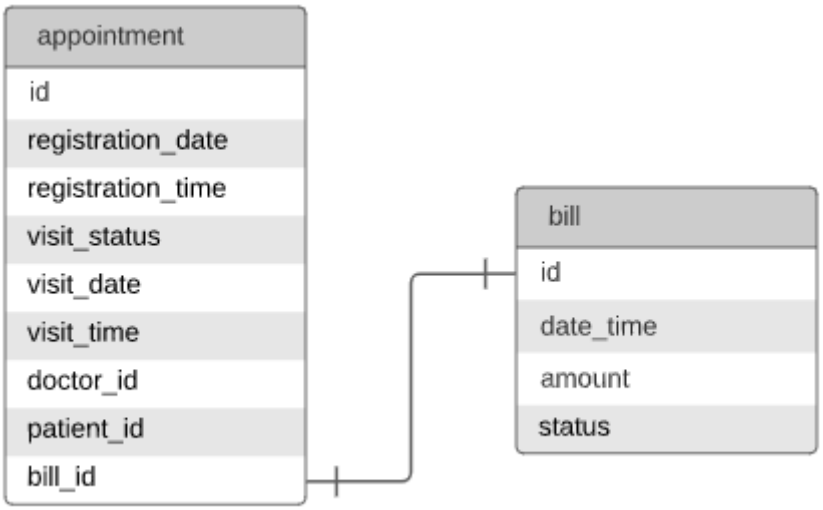
Funkcionalni zahtjevi na uslugu za ulogu pacijent:

- Pacijent se može naručiti na pregled kod liječnika
- Pacijent može otkazati pregled




Slika 3.8. Dijagram toka pacijenta

Baza podataka kod usluge za naručivanje pacijenata izgleda jednostavno no u bazu se upisuju podaci iz drugih sustava, id pacijenta i id liječnika. ER dijagram baze podataka se sastoji od dva entiteta, a to su: Pregled (engl. appointment), Račun (engl. bill). Entiteti su spojeni vezom jedan na jedan što znači da po jednom pregledu može biti samo jedan račun. Na slici 3.9. prikazan je ER dijagram usluge za naručivanje pacijenata.

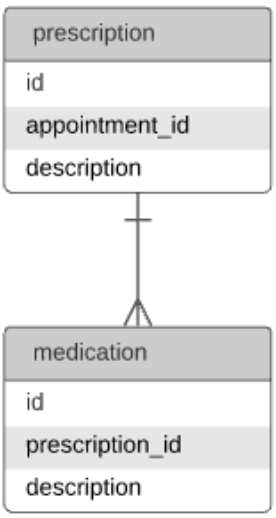



Slika 3.9. ER dijagram usluge za naručivanje pacijenata

### 3.2.4. Usluga za upravljanje receptima "Prescription service"

Usluga za upravljanje receptima je usluga koja može unijeti podatke o receptu i prenijeti recept prema usluzi za upravljanje lijekovima tako da pošalje zahtjev za lijekom nakon spremanja recepta. Usluga komunicira s uslugom za naručivanje pacijenata tako da usluga za naručivanje pozove uslugu za upravljanje receptima gdje onda liječnik može kreirati recept.

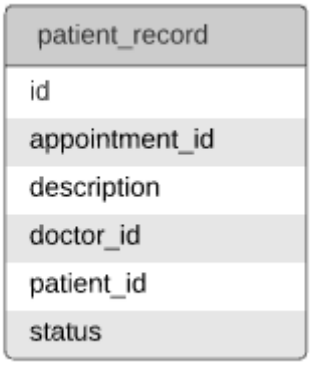
Kao kod sustava za naručivanje pacijenata baza podataka za upravljanje receptima je jednostavna i ima dva entiteta: Recept (engl. prescription), Lijek (engl. medication). Entiteti su povezani vezom jedan na prema više što znači da jedan recept može imati više lijekova.




Slika 3.10. ER dijagram usluge za upravljanje receptima

### 3.2.5. Usluga za upravljanje kartonima pacijenta “Patient record service”

Usluga za upravljanje kartonima pacijenata koristi se za pregled, dodavanje i brisanje zdravstvenog kartona pacijenta. Baza se sastoji od jednog entiteta Karton pacijenta (engl. Patient Record). U usluzi pacijent može samo pregledati svoj zdravstveni karton, a liječnik može raditi sve ostale operacije: dodavanje, brisanje, ažuriranje.




3.11. ER dijagram usluge za upravljanje kartonima pacijenata

### 3.2.6. Usluga za otkrivanje dijabetesa (Diabetic service)

Usluga za otkrivanje dijabetesa je potpuno različita od drugih mikrousluga i temelji se na strojnom učenju. Ova usluga nema bazu podataka i služi za otkrivanje dijabetesa na temelju unesenih parametara.

Proces otkrivanja dijabetesa će se odvijati pomoću logističke regresije. Logistička regresija je algoritam strojnog učenja koji se koristi za klasifikacijske probleme, algoritam je prediktivne analize i temelji se na konceptu vjerojatnosti. Logistička regresija je proces modeliranja vjerojatnosti diskretnog ishoda s obzirom na ulaznu varijablu. Najčešći modeli logističke regresije predstavljaju binarni ishod. Nešto što može poprimiti dvije vrijednosti, poput točno/netočno ili da/ne.

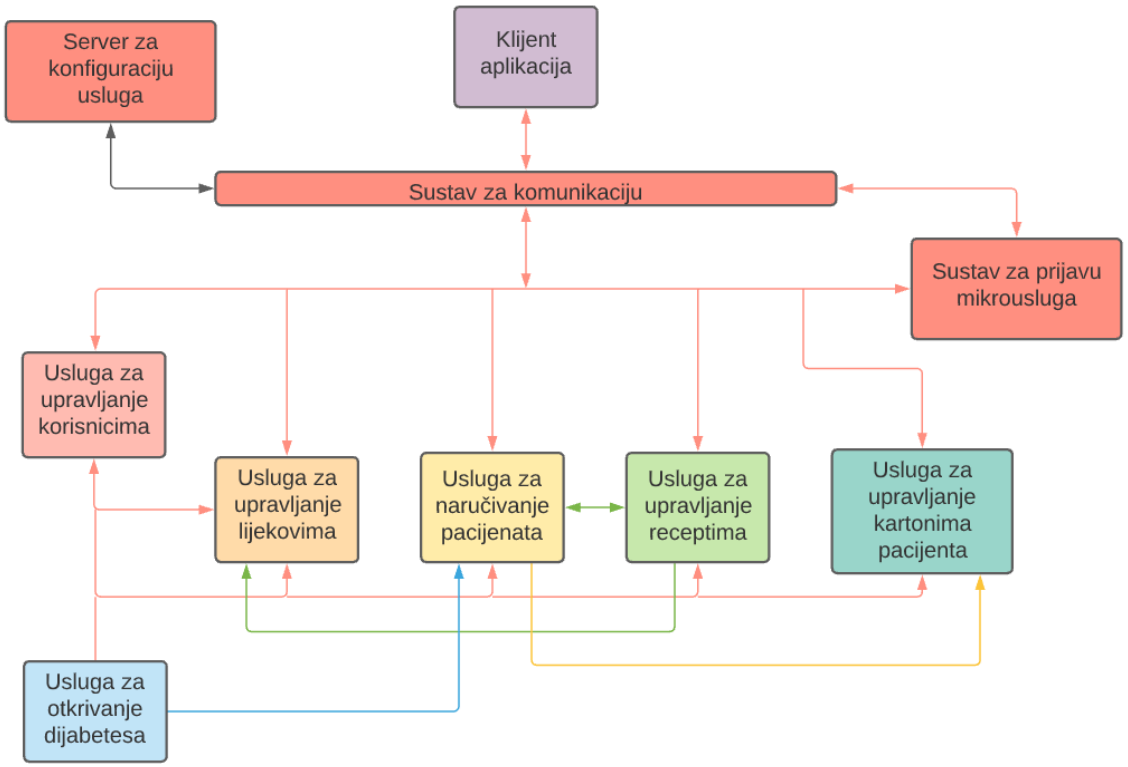
Logistička regresija će ponuditi skalu od 0 do 1 i ta skala će imati postavljen prag na vrijednost 0,5. Ako rezultat strojnog učenja bude iznad 0,5 velika je incidencija da osoba boluje od dijabetesa. Ovisno o rezultatu aplikacija će korisniku ponuditi naručivanje na pregled kod liječnika.

## 3.3. Komunikacija između mikrousluga

Kompletni sustav je složen od devet zasebnih sustava koji su povezani međusobno povezani. Komunikacija između svih mikrousluga je zasnovana na REST arhitekturnom stilu i HTTP metoda. Osim REST-a i HTTP metoda bitan faktor je JSON format koji je zaslužan za razmjenu podataka između mikrousluga

Od spomenutih devet sustava pet su mikrousluge koje odrađuju glavne funkcionalne zahtjeve. Tih pet mikrousluga i njihove funkcionalnosti su objašnjene u prethodnom poglavlju. Najbitniji faktor za funkcioniranje mikrousluga i stvaranje njihove sinergije je sustav za komunikaciju i sustav za prijavu mikrousluga. Također, osim sustava za komunikaciju i sustava za prijavu usluga postoji sustav za konfiguraciju mikrousluga. Taj sustav nam omogućuje centraliziranu konfiguraciju svih mikrousluga što je svakako prednost u odnosu da se svaka mikrousluga mora zasebno konfigurirati. Zadnji dio arhitekture je klijent aplikacija u kojoj se nalazi grafičko sučelje.

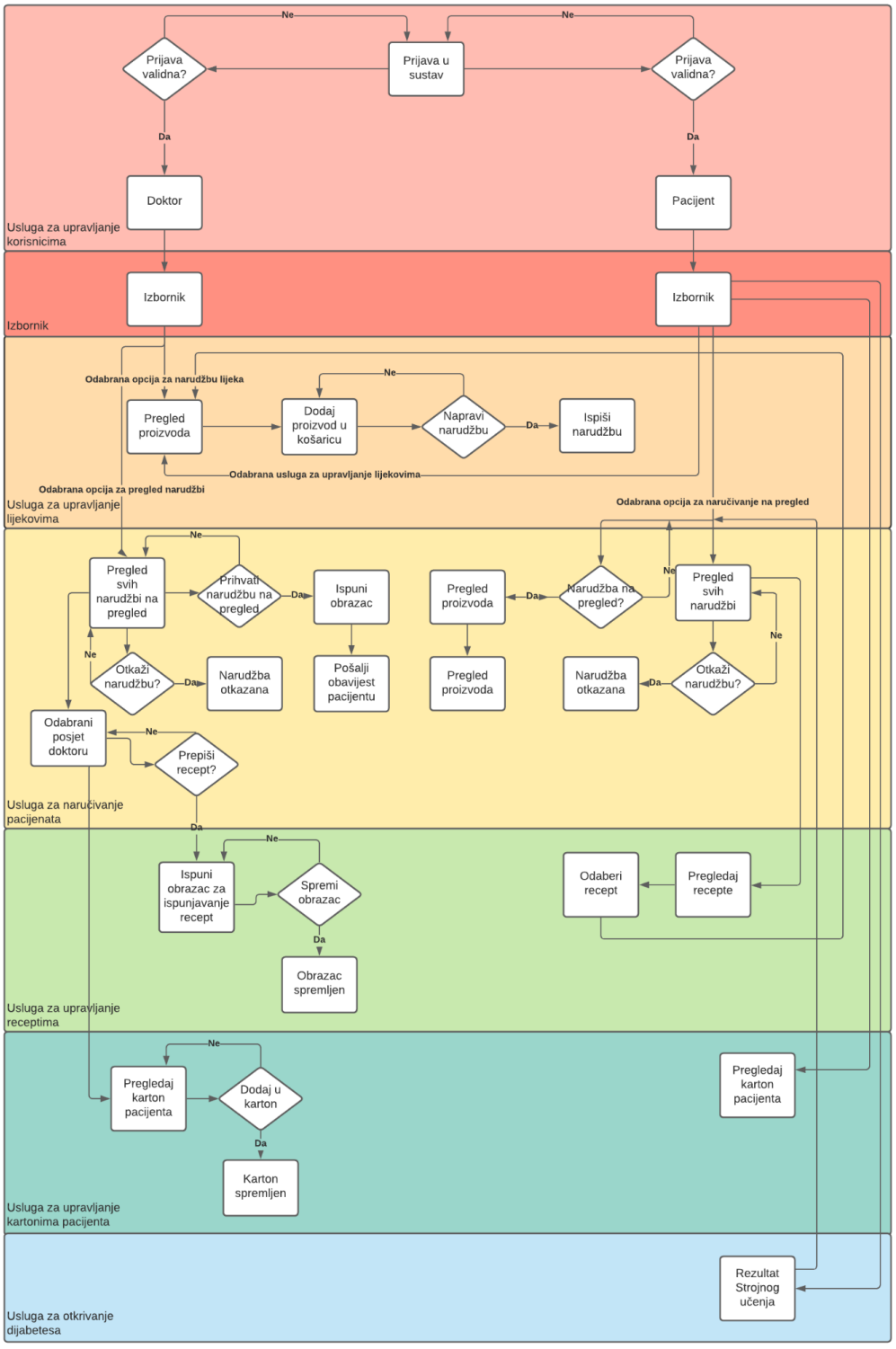
Preko klijent aplikacije korisnik šalje i prima zahtjeve. Prvi sustav koji obrađuje zahtjev je sustav za komunikaciju koji šalje zahtjev prema traženoj mikrousluzi. Kako bi mikrousluge bile vidljive sustavu za komunikaciju potrebno ih je prvo prijaviti. Nakon toga mikrousluge mogu komunicirati međusobno ili s klijent aplikacijom.




Slika 3.12. Prikaz komunikacije između mikrousluga

## 3.4. Prijedlog kompletnog koncepta rada sustava

U ovom potpoglavlju je na slici 3.11. prikazan dijagram toka koji obuhvaća sve mikrousluge i njihove radnje kroz sustav. Također, kroz dijagram toka se vidi komunikacija između mikrousluga.




Slika 3.13. Dijagram toka kompletnog sustava

# 4. PROGRAMSKO RJEŠENJE SUSTAVA

Ovo poglavlje obuhvaća opis korištenih tehnologija i postupak izrade sustava za medicinu od kuće temeljenog na arhitekturi mikrousluga. Na početku su prikazane i opisane tehnologije koje se koriste kroz cijeli sustav, a zatim su opisani postupci izrade sustava na strani poslužitelja i na klijentskoj strani. Kao što je navedeno u poglavlju 3. sustav se sastoji od šest mikrousluga, jednog klijenta i sustava za povezivanje svih mikrousluga. Za svaku mikrouslugu opisan je postupak izrade.

## 4.1. Korištene programske tehnologije

U ovom potglavlju predstavljene su i opisane tehnologije koje su korištene u stvaranju arhitekture mikrousluga. Iako je broj tehnologija veći, navedeno je devet glavnih tehnologija koje se protežu kroz cijeli sustav.

### 4.1.1. Programski jezik Java

Programski jezik Java je brz, siguran i pouzdan. Široko se koristi za razvoj Java aplikacija u računalima, podatkovnim centrima, igraćim konzolama, znanstvenim superračunalima, mobitelima itd.

U programskom jeziku Java programi se grade od klasa. Iz definicije klase možete stvoriti bilo koji broj objekata koji su poznati kao instance te klase. Klasa sadrži članove, a primarne vrste su polja i metode. Polja su varijable podataka koje pripadaju bilo na samu klasu ili na objekte klase, oni čine stanje objekta ili klase. Metode su zbirke izjava koje djeluju na poljima za manipuliranje stanjem. Izjave definiraju ponašanje klase: mogu dodijeliti vrijednosti poljima i drugim varijablama, ocijeniti aritmetičke izraze, dozvati metode i kontroliraju tijek izvođenja. Primjer koda u programskom jeziku Java je prikazan na slici ispod [29].

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world");
    }
}
```

Slika 4.1. Primjer koda u programskom jeziku Java

Tehnologije programskog jezika Java prikladne su za stvaranje mikrousluga, osobito Java API za RESTful Web usluge (JAX-RS) i konteksti (engl. context) te ubrizgavanje ovisnosti (engl. dependency injection) (CDI) za Java EE. Također, bitna stvar kod kreiranja

mikrousluga u Javi su anotacije koje se koriste s podrškom za ubrizgavanje ovisnosti (engl. dependency injection).
Pomoćni programi u Javi pružaju jednostavan mehanizam za korištenje reaktivnih biblioteka (engl. libraries) poput RxJava u prilagođen kontejnerima [30].

### 4.1.2. Programski okvir Spring Boot

Spring Boot je Java programski okvir otvorenog koda kojeg je razvio Pivotal Team. Koristi se za stvaranje i izgradnju samostalnih aplikacija. Osigurava veliku količinu biblioteka i pomoću njega ne moramo pisati „boilerplate“ logiku u kodu. Na taj način Spring nudi apstraktni sloj na J2EE.
Pri programiranju u čistom Java programskom jeziku programer je odgovoran za: učitavanje klasa, stvaranje veza, stvaranje objekata, rukovanje iznimkama, izradu upita, izvršavanje upita i zatvaranje veze. Sve nabrojano se tretira kao „boilerplate“ kod jer svaki programer piše isti kod za navedene stvari. Iz tog razloga Spring Boot brine o „boilerplate“ logici i olakšava programeru posao koji mora pisati samo poslovnu logiku i tako pomoću Spring programskog okvira možemo brzo razvijati projekte s minimalnim linijama koda, bez grešaka, troškova razvoja i vremena [31].
Spring Boot pruža veliki set mehanizma za stvaranje mikrousluga. Aplikacije izrađene u Spring Boot-u koriste programski model jedinstven za taj ekosustav.Spring Boot aplikacije mogu se pakirati za pokretanje kao aplikacija na poslužitelju ili pokretanje kao .jar datoteka koja sadrži sastav ovisnosti i neke vrsta ugrađenog poslužitelja (obično Tomcat). Spring Boot naglašava konvenciju bitnijom od konfiguracije i koristi kombinaciju anotacija kao funkcionalnost koja skraćuje vrijeme programiranja [32].

### 4.1.3. Usluge Rest Web

Rest (Representational State Transfer) Web usluge su okosnica korištenja arhitekture mikrousluga jer su najčešći komunikacijski okvir koji mikrousluge koriste. Zbog svoje jednostavnosti i svestranosti, REST Web usluge postale su standard za rad s web uslugama. REST pruža niz arhitekturnih pravila koja, kada se primijene u cjelini, naglašavaju skalabilnost interakcija između komponenti, općenitost sučelja, neovisno postavljanje komponenti i posredničke komponente za smanjenje kašnjenja interakcije, jačanje sigurnosti i inkapsuliranje naslijeđenih sustava [33].
Rest Web usluga pruža niz osnovnih tehnologija koje je potrebno znati kako bi se koristilo ovaj arhitekturni stil:

- **HTTP Metode** - GET, PUT, POST, DELETE i ostale. Metode koje služe za definiranje poslanog zahtjeva. HTTP metode su opisane u sljedećem poglavlju.
- **JSON** - popularan format razmjene podataka koji je ljudima je lako čitati i pisati. Strojevima je lako raščlaniti i generirati. Temelji se na podskupu JavaScript standarda za programski jezik ECMA-262, 3. izdanje, prosinac 1999. JSON je tekstualni format koji je potpuno neovisan o jeziku, ali koristi konvencije koje su poznate programerima C-obitelji jezika, uključujući C, C ++, C#, Java, JavaScript, Perl, Python i mnogi drugi. Ova svojstva čine JSON idealnim jezikom za razmjenu podataka [34].
- **XML** - drugi format razmjene podataka i oblikovanje dokumenata koji se koristi za stranice World Wide Weba. XML kôd, sličan je jeziku za označavanje hiperteksta (HTML)
- **URI (Uniform Resource Identifier)** - jedinstveni niz koji identificira resurs. Specifikacija URI sintakse ne implicira ništa o svojstvima imena i adresa u različitim prostorima imena koji su mapirani na skup URI nizova. Svojstva proizlaze iz specifikacija protokola i povezanih konvencija o upotrebi za svaku shemu [35].
- **URL (Uniform Resource Locator)** - kombinacija URI-aa i mrežne informacije. Primjer URL-a je: http://www.primjer.com.
- **Idempotencija (engl. Idempotence)** - svojstvo određenih operacija u matematici i računalnoj znanosti da se mogu primijeniti više puta bez promjene rezultata početnog stanja. Drugim riječima, radnju možete izvršiti više puta, a da pritom ne promijenite rezultat. Primjer idempotencije je osvježavanje WEB stranice HTTP GET metodom
- **Stateless** - usluga ne zadržava stanje klijenta. Nema spremanja podataka.
- **HATEOAS (Hypermedia As The Engine of application State )** - način dizajniranja REST API-ja. Točnije, to je specifično pravilo REST arhitekture koje u svojim odgovorima opisuje kako se može koristiti. Pružanjem URL-ova drugim dopuštenim radnjama

### 4.1.4. HTTP

HTTP (Hypertext Transfer Protocol) je trenutno prepoznat kao najčešće korišteni internetski protokol. Uglavnom se sastoji od HTTP zahtjeva i odgovora. Klijenti kao što su web preglednici, alati za indeksiranje i drugi, obično šalju HTTP zahtjeve na dobro poznate portove koje poslužitelji obrađuju dalje. Nakon što primi zahtjev od klijenta,
poslužitelj vraća odgovor klijentu [36].

Glavna značajka HTTP protokola su HTTP metode (engl. request methods) poznate kao glagoli i koriste se kako bi sustav prepoznao koja se radnja odvija. Prema [37] HTTP metode su:

- **GET** - dohvaćanje svih informacija koje je identificirao URI. Ova metoda se koristi za otvaranje web stranica
- **HEAD** - izvršava radnje poput GET metode, ali razlika je u tome što u HEAD traži samo meta podatke bez tijela (engl. body).
- **POST** - koristi se kako bi se poslali podaci s klijenta na poslužitelj. Tipični primjer je slanje, registracijske forme za registraciju korisnika, POST metodom koju poslužitelj prepoznaje kao zahtjev za spremanjem podataka u bazu. POST je samo metoda za kreiranje
- **PUT** - zahtijeva da se poslani entitet pohrani pod isporučenim URI-em. Ako se URI odnosi na već postojeći resurs, entitet se smatra izmijenjenom verzijom one koja se nalazi na izvornom poslužitelju. Ako URI ne upućuje na postojeći resurs, taj URI može definirati kao novi resurs te izvorni poslužitelj može stvoriti resurs s tim URI-jem. PUT je metoda koja može kreirati i ažurirati.
- **DELETE** - metoda koja na zahtjev briše traženi resurs iz URI-a
- **TRACE** - može se koristiti za provjeru je li zahtjev promijenjen nekim posredničkim poslužiteljima
- **OPTIONS** - vraća sve dostupne metode koje se koriste u poslužitelju na traženom URL-u
- **CONNECT** - pretvara HTTP zahtjev u transparentni TCP/IP tunel, koristi se za HTTPS (HyperText Transfer Protocol Secure).
- **PATCH** - Primjenjuje djelomične izmjene na navedeni resurs. Mijenja samo elemente koji su navedeni u zahtjevu.

U HTTP protokolu postoje sigurnosne metode (engl. safe methods) za koje je utvrđena konvencija da metode GET, HEAD, TRACE, OPTIONS ne smiju odrađivati nikakve druge radnje osim dohvaćanja. Spomenute metode se treba smatrati “sigurnim”. Ova konvencija daje do znanja klijentima da druge metode kao što su POST, PUT, DELETE su moguće rizične radnje.

### 4.1.5. Klijent za komuniciranje WebClient

Nastao je kao dio Spring Web Reactive modula i zamijenit će klasični RestTemplate. Osim toga, novi klijent reaktivno je rješenje koje ne blokira zahtjeve i funkcionira preko HTTP/1.1 protokola. Važno je napomenuti da, iako je u stvari klijent koji ne blokira i pripada biblioteci spring-webflux, rješenje nudi podršku i za sinkrone i za asinkrone operacije, što ga čini pogodnim i za aplikacije pokrenute koje su sa sinkronim načinom rada. WebClient sučelje ima jednu implementaciju, klasu DefaultWebClient koja se implementira u Config klasu projekta.

Prema službenoj Spring dokumentaciji [38] kako bi se koristio WebClient potrebno je dodati ovisnost sa slike:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Slika 4.2. Uključivanje Webflux biblioteke u projekt

Nakon dodavanja ovisnosti potrebno je kreirati instancu klijenta. Na sljedećoj slici je primjer kreiranja instance klijenta gdje se definira URL nakon kojeg se može poslati zahtjev na definirani URL preko WebClient-a.

```
private final WebClient webClient;

public UserServiceImpl(WebClient.Builder webClientBuilder) {
    this.webClient = webClientBuilder.baseUrl("http://example.org").build();
}
```

Slika 4.3. Kreiranje instance WebClient-a

### 4.1.6. Baza podataka PostgreSQL

PostgreSQL je moćan, objektno-relacijski sustav baze podataka otvorenog koda koji koristi i proširuje SQL jezik u kombinaciji s mnogim značajkama koje sigurno pohranjuju i skaliraju najkompliciranija opterećenja podataka. Podrijetlo PostgreSQL -a datira od 1986. godine u sklopu projekta POSTGRES na Kalifornijskom sveučilištu u Berkeleyu i ima više od 30 godina aktivnog razvoja na jezgri platforme.

PostgreSQL zaslužio je snažnu reputaciju svojom dokazanom arhitekturom, pouzdanošću, integritetom podataka, robusnim skupom značajki, proširenošću i predanošću zajednice otvorenog koda koja stoji iza softvera kako bi dosljedno isporučivao učinkovita i inovativna rješenja. PostgreSQL radi na svim većim operativnim sustavima, od 2001. usklađen je s ACID-om i ima moćne dodatke, poput popularnog proširenja geoprostorne baze podataka PostGIS. Ne čudi što je PostgreSQL postao relacijska baza podataka otvorenog koda za mnoge ljude i organizacije.

PostgreSQL dolazi s mnogim značajkama čiji je cilj pomoći programerima u izgradnji aplikacija, administratorima u zaštiti integriteta podataka i izgradnji okruženja otpornih na greške te pomoći u upravljanju podacima bez obzira na to koliko veliki ili mali je skup podataka. Osim što je besplatan i otvorenog koda, PostgreSQL je vrlo proširiv. Na primjer, možete definirati vlastite tipove podataka, izgraditi prilagođene funkcije, čak i napisati kôd iz različitih programskih jezika bez ponovnog sastavljanja baze podataka.

PostgreSQL pokušava biti u skladu sa standardom SQL gdje takva usklađenost nije u suprotnosti s tradicionalnim značajkama ili bi mogla dovesti do loših arhitektonskih odluka. Mnoge značajke koje zahtijeva SQL standard su podržane, iako ponekad s malo drugačijom sintaksom ili funkcijom. S vremenom se mogu očekivati daljnji pomaci prema sukladnosti. Od izdanja verzije 13 u rujnu 2020., PostgreSQL je usklađen s najmanje 170 od 179 obaveznih značajki za usklađenost SQL: 2016 Core [39].

U ovom diplomskom radu PostgreSQL će se koristiti kao baza podataka za četiri mikrousluge, dok za jednu mikrouslugu će se koristiti MySql baza podataka kako bi se pokazala raznovrsnost arhitekture Mikrousluga.

### 4.1.7. Baza podataka MySql

MySQL je najpopularnija svjetska baza podataka otvorenog koda. Svojim dokazanim performansama, pouzdanošću i jednostavnošću korištenja, MySQL je postao vodeći izbor baze podataka za web-aplikacije koje koriste visokoprofilna web svojstva, uključujući Facebook, Twitter, YouTube, Yahoo! i još mnogo toga [40].

MySQL je sustav za upravljanje relacijskim bazama podataka (RDBMS) koji je razvio Oracle i temelji se na strukturiranom jeziku upita (SQL).

Baza podataka je strukturirana zbirka podataka. To može biti bilo što, od jednostavnog popisa za kupnju do galerije slika ili mjesta za pohranu ogromne količine informacija u

korporativnoj mreži. Konkretno, relacijska baza podataka je digitalna trgovina koja prikuplja podatke i organizira ih prema relacijskom modelu. U ovom modelu tablice se sastoje od redaka i stupaca, a odnosi između elemenata podataka slijede strogu logičku strukturu. RDBMS je jednostavno skup softverskih alata koji se koriste za zapravo implementaciju, upravljanje i postavljanje upita o takvoj bazi podataka. U ovom diplomskom radu MySQL baza podataka će se koristiti u jednoj mikrousluzi.

### 4.1.8. Programski okvir Angular

Prema [41] Angular je programski okvir za razvoj aplikacija i razvojna platforma za stvaranje učinkovitih i sofisticiranih aplikacija na jednoj stranici. Angular je razvojna platforma, izgrađena na TypeScript -u. Kao platforma, Angular uključuje:

- Okvir temeljen na komponentama za izradu skalabilnih web aplikacija
- Zbirka dobro integriranih biblioteka koje pokrivaju širok spektar značajki, uključujući usmjeravanje, upravljanje obrascima, komunikaciju klijent-poslužitelj i još mnogo toga
- Paket razvojnih alata koji će pomoći u razvoju, izgradnji, testiranju i ažuriranju koda

S Angular-om se iskorištava platforma koja se može skalirati od projekata za pojedinačne razvojne programere do aplikacija na razini poduzeća. Angular je osmišljen kako bi ažuriranje bilo što jednostavnije. Angular ekosustav sastoji se od raznolike skupine od preko 1,7 milijuna razvojnih programera, autora biblioteka i stvaratelja sadržaja.
U ovom diplomskom radu Angular programski okvir će se koristiti kako bi se prikazalo grafičko sučelje i kako bi se slali zahtjevi prema poslužitelju. Angular će zahtjeve slati na sustav za komunikaciju koji će poslani zahtjev prosljeđivati mikrouslugama.

### 4.1.9. Razvojno okruženje IntelliJ IDEA

Prema [41] IntelliJ IDEA inteligentan je IDE s kontekstom za rad s Javom i drugim JVM jezicima poput Kotlina, Scale i Groovyja na svim vrstama aplikacija. Osim toga, IntelliJ IDEA Ultimate pomaže u razvoju web aplikacija zahvaljujući moćnim integriranim alatima, podršci za JavaScript i srodnim tehnologijama te naprednoj podršci za popularne okvire poput Spring, Spring Boot, Jakarta EE, Micronaut, Quarkus, Helidon. Štoviše, IntelliJ IDEA-u je moguće proširiti besplatnim dodacima koje je razvio JetBrains, omogućujući rad s drugim programskim jezicima, uključujući Go, Python, SQL, Ruby i PHP. IntelliJ IDEA ima neke vrhunske produktivne značajke dovršavanja Java koda. Algoritam predviđanja može točno pretpostaviti što razvojni programer pokušava upisati i dovršiti ga umjesto njega, čak i

ako ne zna točan naziv određene klase, člana ili bilo kojeg drugog izvora. Kako bi se razvojnim programerima pomoglo u organizaciji njihovog tijeka rada, IntelliJ IDEA nudi skup alata, koji se sastoji od prevoditelja, Docker podrške, preglednika bajt kodova, FTP -a i mnogih drugih alata:Kako bi programerima pomogli u organizaciji njihovog tijeka rada, IntelliJ IDEA nudi im nevjerojatan skup alata, koji se sastoji od dekompilatora, Docker podrške, preglednika bajt kodova, FTP -a i mnogih drugih alata:

- **Kontrola verzija (engl. version control)** - IntelliJ podržava većinu popularnih sustava kontrole verzija kao što su Git, Subversion, Mercurial, CVS, Perforce i TFS.
- **Alati za izgradnju (engl. build tools)** - IntelliJ podržava Javu i druge alate za izgradnju kao što su Maven, Gradle, Ant, Gant, SBT, NPM, Webpack, Grunt i Gulp.
- **Pokretač testova i pokrivenost koda (engl. test runner and code coverage)** - IntelliJ IDEA izvršavanje jediničnog testiranja. IDE uključuje ispitne programe i alate za pokrivanje glavnih testnih okvira, uključujući JUnit, TestNG, Spock, krastavac, ScalaTest, spec2 i Karmu.
- **Prevoditelj** - ugrađenim prevoditelj za Java klase.
- **Konzola (engl. terminal)** - ugrađena konzola koja pomaže
- **Alati za upravljanje bazom podataka** - s ovim alatima se mogu slati SQL upiti
- **Aplikacijski poslužitelj** - IntelliJ podržava glavne poslužitelje aplikacija: Tomcat, JBoss, WebSphere, WebLogic, Glassfish i mnoge druge.

## 4.2. Spring Boot Cloud tehnologije za implementaciju arhitekture mikrousluga

Spring Boot Cloud nudi niz tehnologija za implementaciju arhitekture mikrousluga. U ovom poglavlju su predstavljene najkorištenije tehnologije i tehnologije koje su korištene u ovom diplomskom radu. Spring Cloud Eureka Discovery je mikrousluga u koju se prijavljuju sve mikrousluge. Drugi po redu opisan je Spring Cloud API gateway koji je komunikacijski poslužitelj za komunikaciju između mikrousluga. Zadnja opisana tehnologija je konfiguracijski poslužitelj Spring Cloud Config server.

### 4.2.1. Poslužitelj Spring Cloud Eureka Discovery

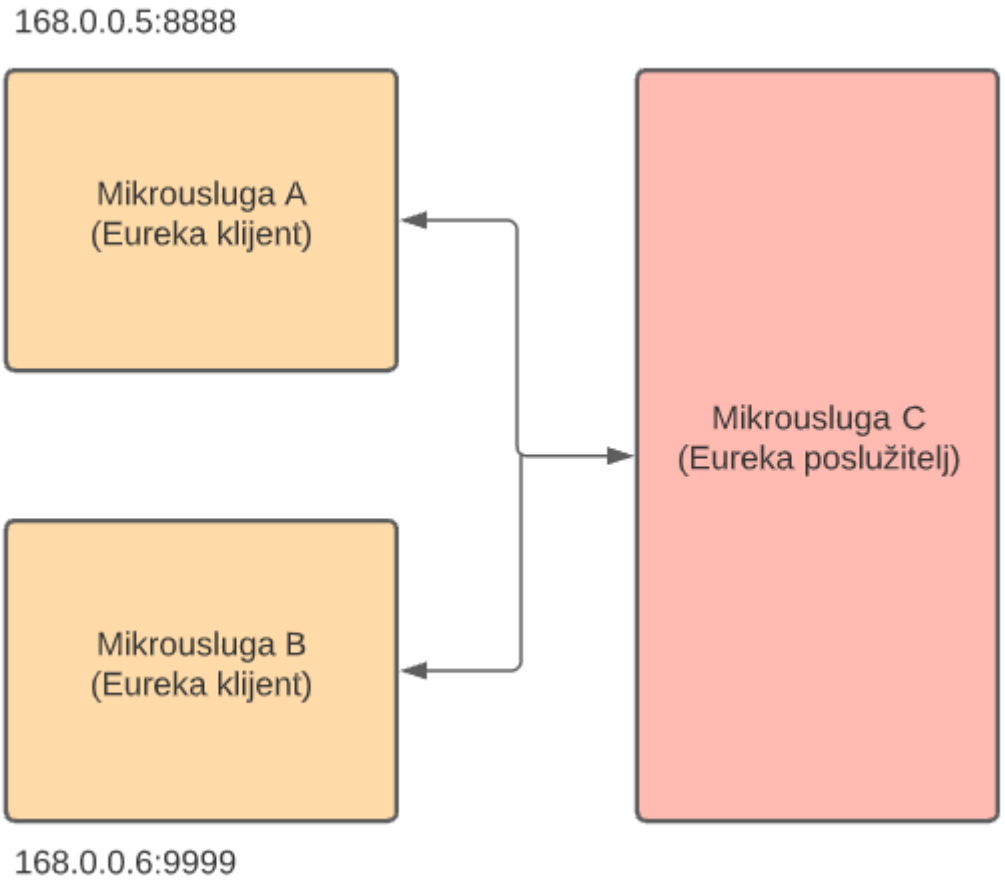
Jednostavnim riječima, mikrousluge su skupine malih aplikacija koje zajedno rade zajedno kako bi pružile cjelovito rješenje. Kad se kaže da mnogo malih aplikacija radi samostalno, tada će sve imati svoje URL -ove i pristupne točke. U tom bi scenariju bilo vrlo nezgodno

održavati sve te mikrousluge radi sinkronizacije, i što je još važnije, nadzor (engl. monitoring) bi bio gotovo nemoguć. Taj će se problem višestruko povećati kada se počnu primjenjivati uravnoteživači opterećenja (engl. load balancers).

Alat koji nudi rješenje gornjeg problema je Netflix Eureka. Sastoji se od Eureka poslužitelja i Eureka klijenta. Eureka poslužitelj sam je po sebi mikrousluga na koju se registriraju sve ostale mikrousluge. Eureka klijenti su neovisne mikrousluge.

Eureka je usluga temeljena na REST-u koja se prvenstveno koristi za prikupljanje informacija o mikrouslugama s kojima se želi komunicirati.

Ova REST usluga poznata je i pod imenom Eureka poslužitelj. Usluge koje se registriraju na Eureka poslužitelju radi prikupljanja informacija jednih o drugima nazivaju se Eureka klijenti. Sljedeći slika prikazuje kako se Eureka klijenti i Eureka poslužitelj uklapaju zajedno.



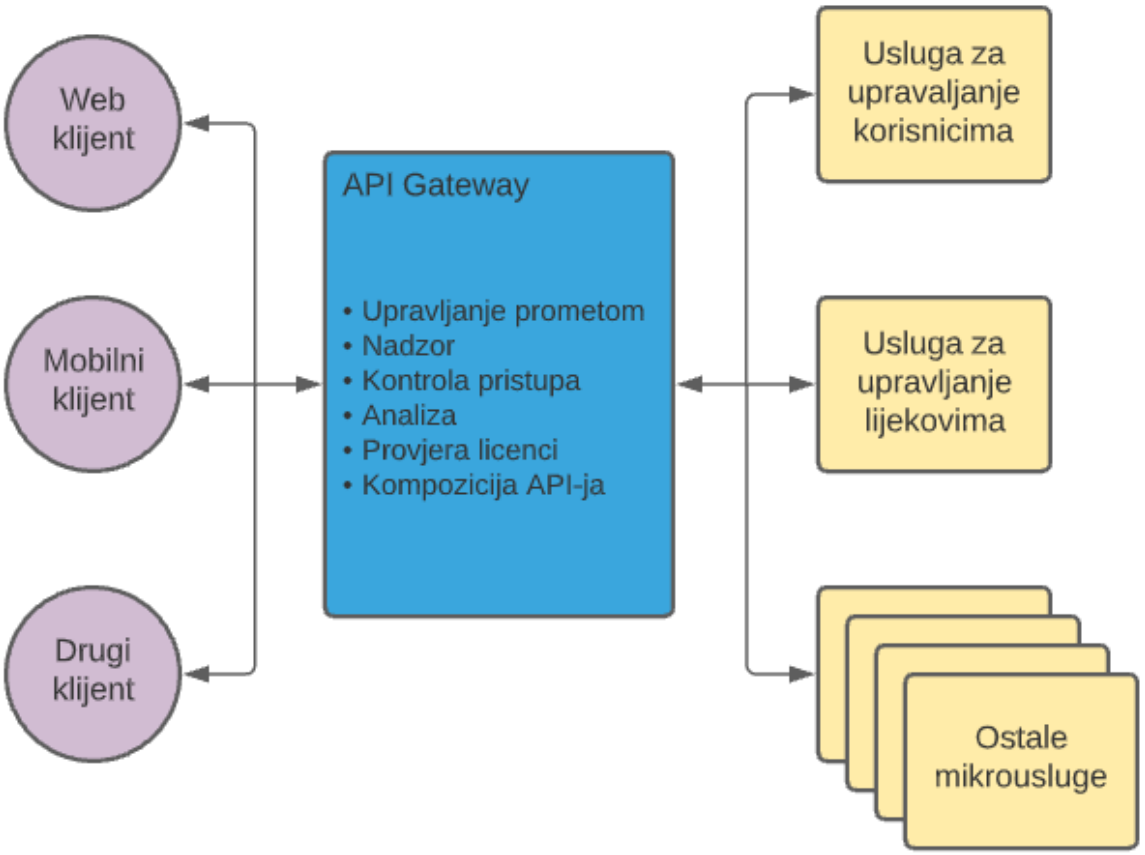

Slika 4.4. Prikaz komunikacije Eureka klijenta i Eureka poslužitelja

Kao što gornja slika pokazuje, usluga A i usluga B registrirane su na Eureka poslužitelju. Ako bilo koja od ove dvije usluge želi komunicirati s drugom uslugom, može dobiti ciljnu IP adresu i pristupnu točku putem Eureka poslužitelja. Podešavanje i implementacija Eureka

poslužitelja prikazano je u poglavlju 4.3.1, a prijava mikrousluga u Eureka poslužitelj objašnjena je u poglavlju 4.3.8.

### 4.2.2. Komunikacijski poslužitelj Spring Cloud API gateway

Spring Cloud API Gateway predstavlja jedan od najvažnijih uzoraka u arhitekturi mikrousluga. Ova tehnologija služi za komunikaciju između mikrousluga. API gateway prima sve API pozive od klijenata i usmjerava ih do odgovarajućih mikrousluga s usmjeravanjem zahtjeva, sastavom i prijevodom protokola. API Gateway je uveden kako bi se osigurali grubo zrnati API-ji koji mogu komunicirati s finim zrnatim API-jevima, objediniti odgovore i vratiti te odgovore krajnjem korisniku. U slici ispod je prikazana kompletna struktura i komunikacija s API-gateway-em od klijenta do poslužitelja. Također, navedeni su neki od poslova koje odrađuje API gateway.




Slika 4.5. Prikaz komunikacije API gateway-a s ostalim uslugama

Spring Cloud Gateway podržava mnoge međufunkcionalne značajke koje se vide na slici. Kako bi podržao širok raspon takvih značajki, okvir se temelji na tri osnovne komponente:

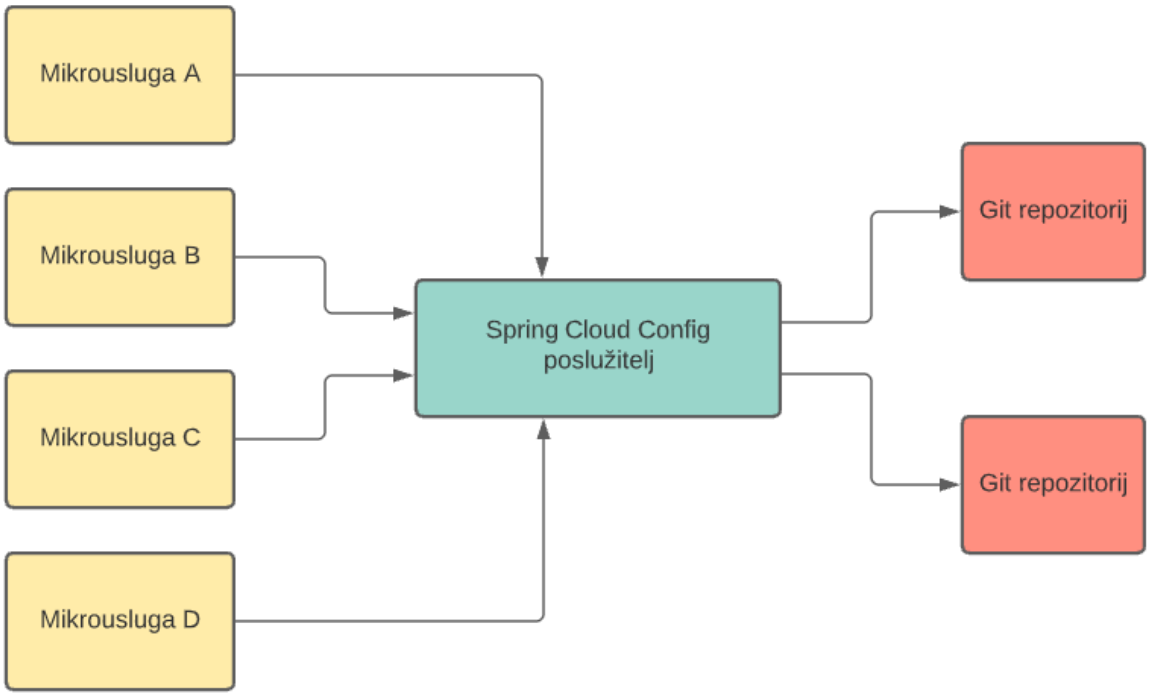
- **Rute** - Ovo je primarna komponenta koja se sastoji od ID-a, odredišnog Uri-ja, predikata i filtera.
- **Predikati** - postavlja se više kao uvjet koji odgovara ruti. Radi se na temelju HTTP zahtjeva, zaglavlja, parametara, putanje, kolačića i drugih kriterija zahtjeva.
- **Filtri** - Oni su dodaci za ažuriranje zahtjeva ili odgovora. Funkcioniraju slično filterima servleta. Filtre se može koristiti u više svrha, uključujući "dodavanje parametra zahtjeva", "dodavanje zaglavlja odgovora", "praćenje" i mnoge druge svrhe.

### 4.2.3. Konfiguracijski poslužitelj Spring Config

Najbolja praksa za razvoj aplikacije je da je labavo povežete. Trebali bismo moći dodavati nove značajke, svojstva bez utjecaja na njegove druge već postojeće funkcionalnosti. Kako sustav raste postaje teško upravljati svojstvima različitih mikrousluge i module sustava. Koje je svojstvo za razvojnu platformu, a koje za proizvodnu izgradnju. S promjenama svojstava, moramo ponovno izgraditi, ponovno testirati i ponovno napraviti isporuku aplikacije.
Navedeni problem je rješiv odvajanjem konfiguracije od logike mikrousluga i sve mikrousluge onda mogu čitati konfiguraciju u hodu bez ponovne isporuke. Postoje različiti pristupi poput Spring Cloud Config, Consul i Zookeeper. U ovom diplomskom radu koristi se Spring Cloud Config poslužitelj.
Prednosti Spring Cloud Config poslužitelja su:

- Dosljednost
- Ušteda vremena
- Konfiguracija temeljena na profilima

Sve konfiguracije mikrousluga Spring Cloud Config poslužitelj čita iz datoteka. Više je opcija kako spremiti datoteke. Prva opcija je da se spremaju lokalno u datotečnom sistemu, a druga i pouzdanija opcija je da se spremaju na vanjskom poslužitelju u oblaku. Najčešće se koristi Git Repo kao izvor konfiguracija. Spring Cloud Config poslužitelj dohvaća konfiguracije iz Git Repo-a te bilo koja mikrousluga može vidjeti promjene iz tih konfiguracijskih datoteka




Slika 4.6. Prikaz komunikacije mikrousluga sa Spring Cloud Config poslužiteljem

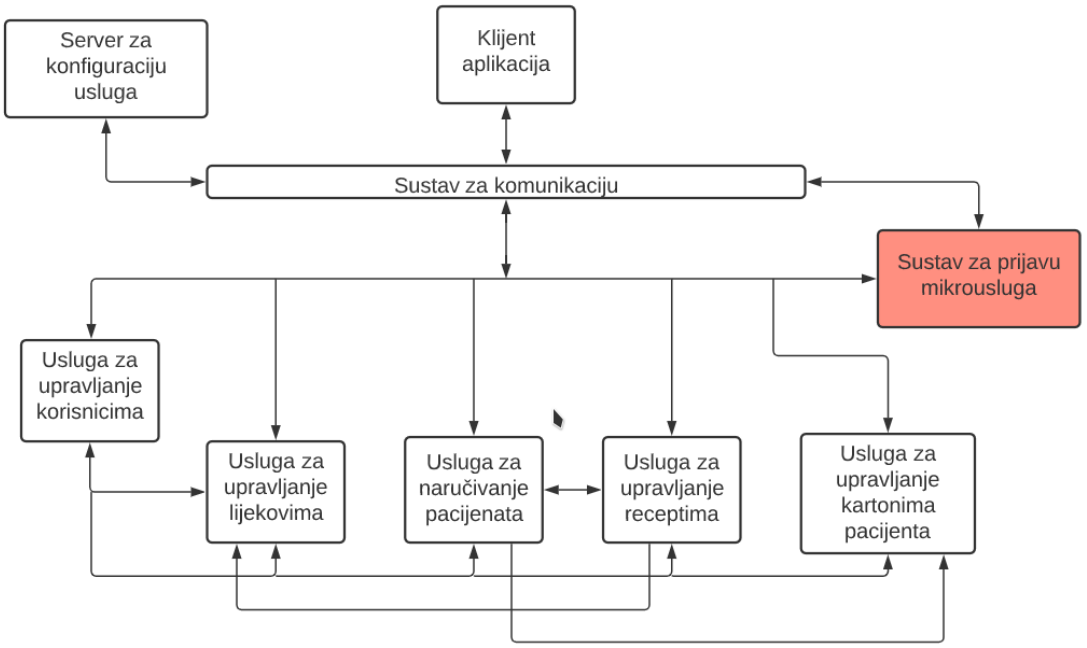
Ako se mikrousluga prijavi na Spring Cloud Config poslužitelj onda toj mikrousluzi lokalna konfiguracija prestaje biti primarna. Nakon prijave primarna postaje konfiguracija na Spring Cloud Config poslužitelju. Također, na poslužitelju se može postaviti više konfiguracija, najčešće su razvojna i produkcijska.

## 4.3. Programsko rješenje na strani poslužitelja

Rješenje cjelokupnog sustava je podijeljeno u manje sustave pa tako svaka veća funkcionalnost ima svoj sustav tj. svoju mikrouslugu. Sustav zdravstva od kuće bi se mogao podijeliti na tri dijela: klijent sustav, sustav za komunikaciju i sustav mikrousluga. Također, sustav za komunikaciju i sustav mikrousluga se dijele na manje podsustave. Svaka mikrousluga se pokreće s Javom 11 i Maven 3 alatom za izvršavanje. U sljedećim potpoglavljima je opisano kreiranje i implementacija sustava za komunikaciju i sustava mikrousluga.

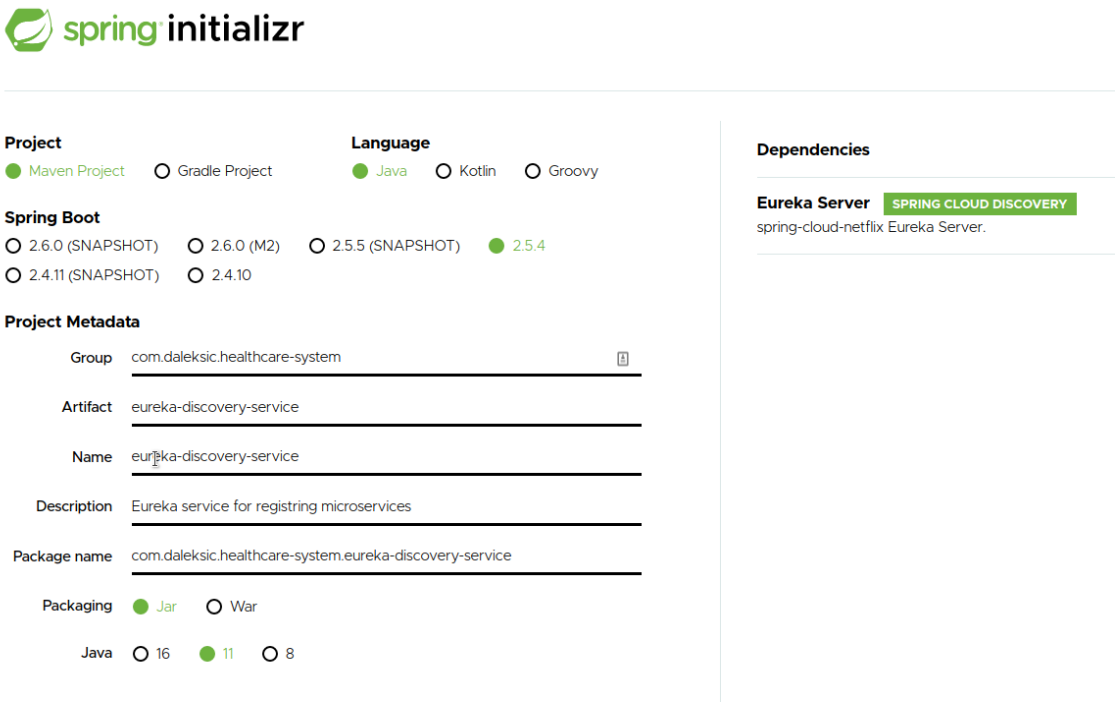
### 4.3.1. Kreiranje Eureka Discovery sustava

Kao što je napisano u poglavlju 4.2.2. Eureka je sustav za prijavu mikrousluga. Nakon kreiranja Eureke potrebno je svaku mikrouslugu prijaviti i uspostaviti vezu sa Spring Cloud-om. Prijava mikrousluga u Eureku i komuniciranje sa Spring Cloud-om je objašnjeno u potpoglavljima 4.3.8. i 4.3.9. Na slici ispod je prikazana pozicija Eureka poslužitelja u sustavu. Vidljivo je da sve mikrousluge komuniciraju s Eureka poslužiteljem.




Slika 4.7. Prikaz pozicije Eureke u sustavu

Eureka Discovery sustava može kreirati iz "Spring Initializr" alata za kreiranje Spring Boot projekata. Prilikom kreiranja Eureke u polje "Group" je postavljena vrijednost: com.daleksic.healtcare-system. Spomenuta vrijednost se postavlja kod svakog projekta kako bi sustav imao jednoznačna obilježja. Na sljedećoj slici prikazana početna faza kreiranja Eureka poslužitelja sa "Spring Initializr-om".




Slika 4.8. Kreiranje projekta Eureka Discovery

Pod polje "Artifact" je postavljena vrijednost eureka-discovery-service. Ova vrijednost je bitna zbog kreiranja paketa u kojemu su klase. Dalje, nakon popunjavanja obaveznih polja potrebno je dodati ovisnost (engl. dependency) koji će omogućiti kreiranoj aplikaciji da odrađuje funkcionalnosti Eureka Discovery poslužitelja

Nakon generiranja projekta u pom.xml datoteci će se pojaviti ovisnost na slici.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Slika 4.9.. Eureka server ovisnost

Ovisnost spring-cloud-starter-netflix-eureka-server je bitan jer će nam omogućiti funkcioniranje Eureka servera. Sljedeći korak je u Eureka aplikaciju dodati anotaciju "@EnableEurekaServer" kako bi prilikom pokretanja JVM (Java Virtual Machine) prepoznao da pokreće Eureka server.

Nakon dodavanja anotacije potrebno je konfigurirati aplikaciju na način da se postavi ime aplikacije i port na kojem se aplikacija pokreće.

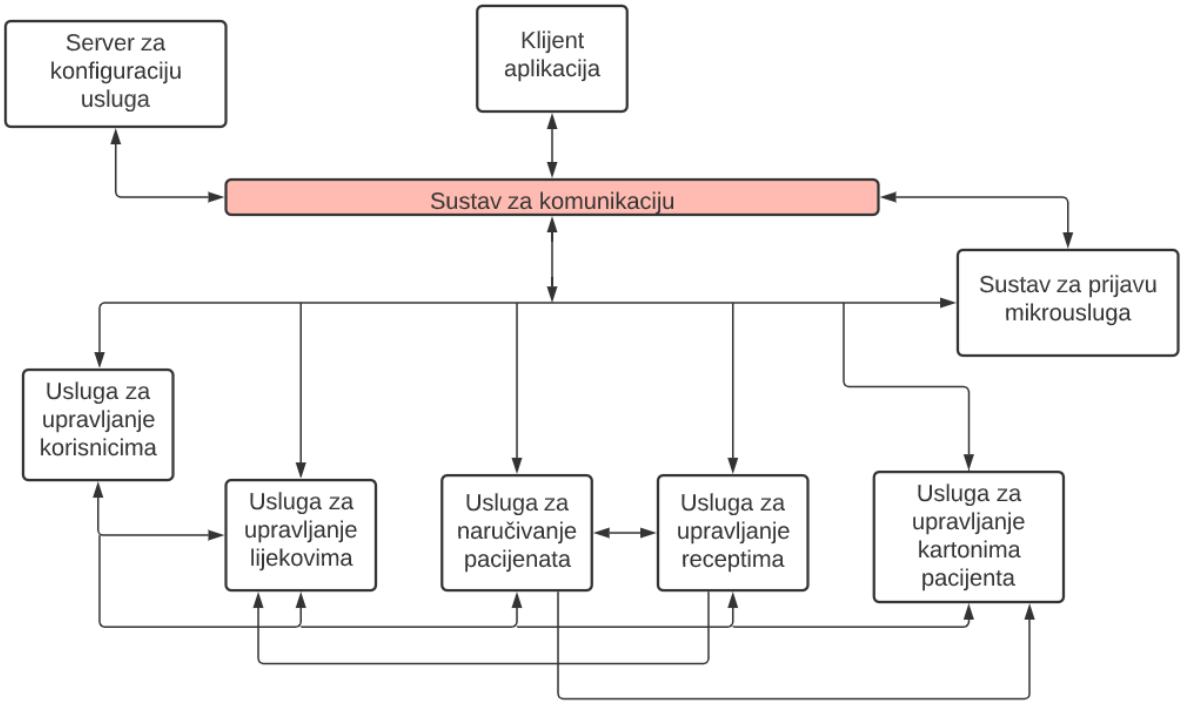
```
server.port=9001
spring.application.name=eureka-server
eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false
eureka.client.serviceUrl.defaultZone = http://localhost:9001/eureka
```

Slika 4.10.. Konfiguracija Eureka Servera

Pokretanjem se uspostavlja Eureka server koji u trenutku u ovom trenutku nema nijednu prijavljenu mikrouslugu. Eureka serveru se može pristupiti na portu 8761.

### 4.3.2. Kreiranje Spring Cloud API gateway sustava za komunikaciju

Spring Cloud API gateway sustav je slikovito rečeno "krvotok" cjelokupnog sustava. Ovaj sustav omogućava komuniciranje svih mikrousluga međusobno kao i komuniciranje klijenta s mikrouslugama. Više o Spring Cloud API gateway-u je rečeno u poglavlju 4.2.2.




Slika 4.11. Prikaz pozicije Spring Cloud API Gateway-a u sustavu

Kao kod eureke za kreiranje Spring Cloud API gateway-a koristi se "Spring Initializr" u kojem će se odabrati:

- Spring Cloud Routing Gateway - dodano kako bi dobili funkcionalnosti API gateway-a
- Spring Reactive Web - dodano kako bi dobili funkcionalnosti reaktivnog web programiranja koje će nam pomoći kod asinkronih slanja podataka
- Eureka Discovery Client - dodano kako bi API gateway imao mogućnost spajanja na Eureka Discovery Server

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Slika 4.12. Prikaz ovisnosti Spring Cloud API Gateway-a

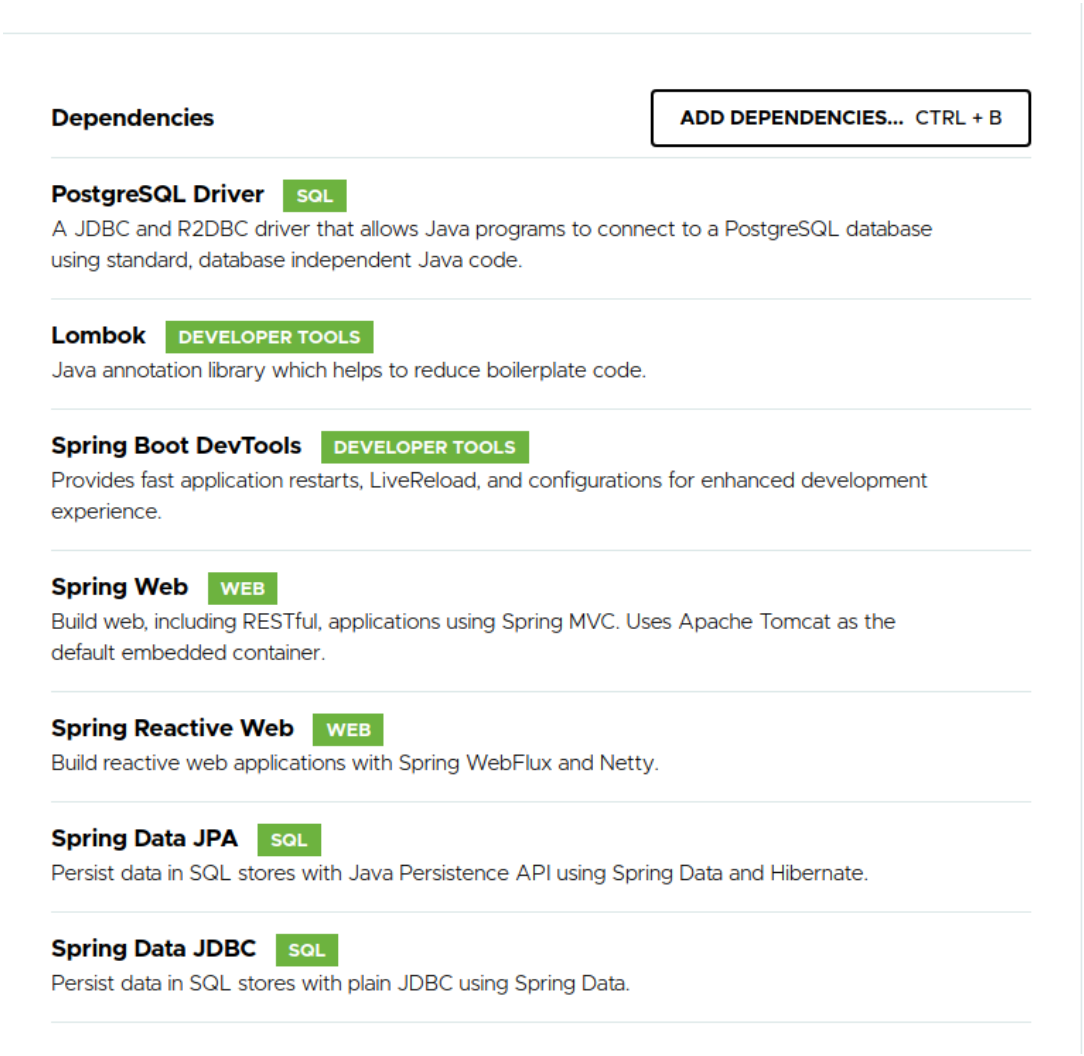
Nakon generiranja projekta i provjere ovisnosti potrebno je konfigurirati postavke aplikacije. Spring Cloud API gateway je postavljen na port 9100, dodijeljen je naziv api-gateway i uspostavljena je konekcija s Eureka Discovery Serverom

```
server.port=9100
spring.application.name=api-gateway
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
```

Slika 4.13. Prikaz postavki Spring Cloud API Gatewaya

### 4.3.3. Implementacija usluge za upravljanje korisnicima “Users service”

Usluga za upravljanje korisnicima je usluga koja komunicira sa svim mikrouslugama i jedna je od najbitnijih u cijelom sustavu. Preko ove usluge se može saznati preko koje uloge će se korisnik prijaviti u sustav. Implementacija počinje s alatom za kreiranje Spring Boot projekata Spring Initializr. Kod kreiranja je potrebno dodati ovisnosti (engl. dependency) sa slike.




Slika 4.14. Dodavanje ovisnosti (engl. dependency)

Na slici je vidljivo da se dodalo sedam ovisnosti koje će pomoći prilikom implementacije ove usluge:

- **PostgreSQL Driver** - omogućuje komuniciranje s PostgreSQL bazom podataka
- **Lombok** - čini kod preglednijim i uklanja "boilerplate" kod
- **Spring Boot DevTools** - sa svojim funkcionalnostima olakšava programiranje razvojnom programeru
- **Spring Web** - ovisnost (engl. dependency) koja omogućava RESTful arhitekturu
- **Spring Reactive Web** - potreban zbog WebClient sučelja i komunikacije između mikrousluga
- **Spring Data JPA** - ovisnost (engl. dependency) koja omogućava Java programski kod pretvara u SQL upite

Nakon dodavanja ovisnosti (engl. dependency) potrebno je podesiti postavke za bazu podataka. Na slici su vidljive postavke koje su potrebne da bi se uspostavila stabilna veza između aplikacije i PostgreSQL baze podataka.

```
spring.datasource.url=jdbc:postgresql://127.0.0.1:5432/users
spring.datasource.username=healthcare
spring.datasource.password=healthcare
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.database=POSTGRESQL
spring.jpa.database-platform = org.hibernate.dialect.PostgreSQL92Dialect
spring.jpa.open-in-view=true
spring.jpa.show_sql=false
spring.jpa.generate-ddl=false
spring.jpa.hibernate.use-new-id-generator-mappings=true
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.hibernate.ddl-auto=create
```

Slika 4.15. Podešavanje PostgreSQL baze podataka

Za kreiranje entiteta u bazi podataka nije potrebno raditi SQL upite. Spring Boot okvir ima funkcionalnost koja odrađuje kreiranje entiteta. Potrebno je samo definirati klasu za koju želite da bude entitet i označiti je s anotacijom "@Entity". Ova mikrousluga se sastoji od četiri entiteta: User, Patient, Doctor i Role.

U User entitetu se nalazi username, password i email koji će se kasnije koristiti za prijavu korisnika u sustav. User entitet ima vezu jedan na jedan s Doctor entitetom i Patient entitetom što u primjeru znači: jedan User može imati samo jednog liječnika. Također, Users entitet ima vezu više na više s Role entitetom što znači da više Role-a može imati više Usera i obrnuto. Role entitet služi za spremanje uloga u sustavu. Uz Role entitet kreiran je Enum u kojeg su spremljene tri vrijednosti: ADMIN, DOCTOR, PATIENT. Enum ograničava uloge u sustavu na samo ove tri navedene.

Patient entitet omogućava spremanje pacijentovih podataka u bazu. Vezan je u User entitet te se u tablicu Patient uz ostale vrijednosti sprema i "user_id". Također, uz "user_id" sadrži i "doctor_id" koji označava kojem liječniku pacijent pripada. Doctor entitet omogućava spremanje liječničkih podataka u bazu i kao Patient je vezan za User entitet te se u tablicu Doctor sprema i "user_id". Na sljedećoj slici prikazano je kreiranje User entiteta.


Slika 4.16. User entitet

Nakon kreiranja entiteta potrebno je kreirati kontrolere i implementaciju za kontrolere. Implementirana su tri kontrolera: UserController, PatientController i DoctorController. U UserControlleru je implementirano kreiranje Usera, dohvaćanje Usera, brisanje Usera i ažuriranje Usera. U Patient i Doctor kontrolerima je implementirano samo dohvaćanje liječnika  ili pacijenata, dok kreiranje pacijenata i liječnika odrađuje UserController. Na slici ispod prikazane su metode UserController-a.

```java
@RestController
@RequestMapping("/api/users")
@RequiredArgsConstructor
public class UserController {

    private final UserService userService;

    @PostMapping("/")
    public ResponseEntity<UserDto> create(@RequestBody UserRequestDto userRequestDto){
        return ResponseEntity.ok(userService.create(userRequestDto));
    }

    @GetMapping("/")
    public ResponseEntity<List<UserDto>> findAll() { return ResponseEntity.ok(userService.findAll()); }

    @GetMapping("/{id}")
    public ResponseEntity<UserDto> findById(@PathVariable Long id){
        return ResponseEntity.of(userService.findById(id));
    }
}
```

Slika 4.17. User kontroler

Nakon implementacije kontroler metoda, potrebno je implementirati sigurnosni dio sustava, a to je prijava u sustav. Sigurnost sustava je riješena s SpringSecurity bibliotekom koja ima funkcionalnost prijave u sustav. Na sljedećoj slici su prikazane ovisnosti (engl. dependency) koje je potrebno dodati kako bi SpringSecurity funkcionirao.

```xml
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

Slika 4.18. SpringSecurity ovisnosti (engl. dependency)

Prijava u sustav funkcionira tako da klijent šalje zahtjev za prijavom na prijavnu pristupnu točku, usluga za upravljanje korisnicima procesuira prijavu i generira Json Web Žeton (JWT). Nakon generiranja poslužitelj vraća žeton prema klijentu.

U ovom diplomskom radu korisnik će se moći prijaviti s korisničkim imenom i lozinkom koji nam dopuštaju provjeru postojanja korisnika u Users tablici te ako postoji taj User će se koristiti u procesu prijave. Za provjeru postojanja User-a koristi se metoda "findByUsername" koja za uneseno korisničko ime vraća User-a kojeg će SpringSecurity dalje obrađivati.
Kako bi se odradila sigurnosna provjera User-a potrebno je konfigurirati SpringSecurity. Na sljedećoj slici je prikazana konfiguracijska klasa SpringSecurity-a koja se nalazi u paketu "security".

```java
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class WebSecurity extends WebSecurityConfigurerAdapter {

    private final Environment environment;
    private final UserService userService;
    private final BCryptPasswordEncoder bCryptPasswordEncoder;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable();
        http.authorizeRequests()
                .antMatchers( ...antPatterns: "/user-service/actuator/**").permitAll()
                .antMatchers( ...antPatterns: "/**").permitAll()
                .and()
                .addFilter(getAuthenticationFilter());
        http.headers().frameOptions().disable();
    }

    private AuthenticationFilter getAuthenticationFilter() throws Exception
    {
        AuthenticationFilter authenticationFilter = new AuthenticationFilter(userService, environment, authenticationManager());
        authenticationFilter.setFilterProcessesUrl(environment.getProperty("login.url.path"));
        return authenticationFilter;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userService).passwordEncoder(bCryptPasswordEncoder);
    }
}
```

Slika 4.19. Konfiguracijska klasa SpringSecurity

Prvo što se na slici može primjetiti je anotacija "@EnableWebSecurity". Ova anotacija aktivira web sigurnost integriranu u Spring Boot. Nakon anotacije važno je primijetiti configure metodu koja omogućuje korištenje prilagođene implementacije za dobivanje korisničkih podataka kada je autentifikacija validna. Metoda getAuthenticationFilter služi da bi promijenili URI od prijave u sustav.

Nakon što se konfigurira potrebno je dodati još jednu klasu pod nazivom AuthenticationFilter. Na slici ispod je prikazana klasa Authentication filter u kojoj se nalaze metode attemptAuthentication i successfulAuthentication.

```
@Override
public Authentication attemptAuthentication(HttpServletRequest request,
                                            HttpServletResponse response) throws AuthenticationException {
    try {

        LoginRequestDto creds = new ObjectMapper()
                .readValue(request.getInputStream(), LoginRequestDto.class);

        return getAuthenticationManager().authenticate(
                new UsernamePasswordAuthenticationToken(
                        creds.getEmail(),
                        creds.getPassword(),
                        new ArrayList<>())
        );

    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}


@Override
protected void successfulAuthentication(HttpServletRequest req,
                                        HttpServletResponse res,
                                        FilterChain chain,
                                        Authentication auth) throws IOException, ServletException {
    String userName = ((User) auth.getPrincipal()).getUsername();
    UserDto userDetails = userService.getUserDetailsByEmail(userName);

    String token = Jwts.builder()
            .setSubject(userDetails.getUserId())
            .setExpiration(new Date(System.currentTimeMillis() + Long.parseLong(environment.getProperty("token.expiration_time"))))
            .signWith(SignatureAlgorithm.HS512, environment.getProperty("token.secret") )
            .compact();

    res.addHeader( s: "token", token);
}
```

Slika 4.20. Klasa AuthenticationFilter

Metoda attemptAuthentication prima sesiju i starta proces autentifikacije korisnika. U slučaju da su korisnički podaci ispravni, generira se Json Web Žeton u metodi successfulAuthentication. Prvo se definira vrijeme isteka, a zatim generira ključ na koji će žeton biti potpisan. Na kraju za prijavu u sustav potrebno je upisati URI: /user-service/api/login kako bi se započeo proces prijave. Nakon prijave u sustav s određenom ulogom korisnik može dalje koristiti ostale mikrousluge.

### 4.3.4. Implementacija usluge za upravljanje lijekovima “Pharmacy management service”

Usluga za upravljanje lijekovima je usluga koja uz upravljanje lijekovima može izvršavati naručivanje lijekova, ispisivanje računa. Ova mikrousluga komunicira s:

- uslugom za upravljanje korisnicima preko koje dobije informacije o ulozi korisnika (admin, liječnik, pacijent)
- uslugom za upravljanje receptima preko koje dobije zahtjev za narudžbom lijeka preko recepta

Implementacija, kao i kod svake mikrousluge u ovom diplomskom radu, počinje s alatom za kreiranje Spring Boot projekata Spring Initializr-om. Dodat će se sve iste ovisnosti (engl. dependency) kao u usluzi za upravljanje korisnicima. Mikrousluga za upravljanje lijekovima koristi MySQL bazu podataka kako bi se pokazala raznovrsnost implementacije koju arhitektura mikrousluga nudi. Nakon kreiranja projekta potrebno je podesiti bazu podataka u konfiguracijskoj datoteci. Na slici ispod je prikazano podešavanje konfiguracije za MySQL bazu podataka u Spring Boot-u:

```
## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.url = jdbc:mysql://localhost:3306/pharmacy_shop_management_system?useSSL=false&useUnicode=true&
spring.datasource.username = root
spring.datasource.password =
## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.hibernate.naming.implicit-strategy=org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update
```

Slika 4.21. Konfiguracija MySQL baze podataka

Baza podataka se sastoji od pet entiteta: Product, Transaction, Category, Company, Order. Entitet na koji se svi ostali entiteti spajaju je Product. Podešavanjem baze podataka stekli su se uvjeti za komuniciranjem između aplikacije i baze podataka, a samim time i za kreiranjem entiteta iz Spring Boot programskog okvira. Primjer kreiranja jednog od entiteta je na slici ispod.


Slika 4.22. Product entitet

Nakon kreiranja svih entiteta potrebno je implementirati kontrolere i usluge za proizvode, transakcije, kategorije, tvrtke i narudžbe. Popis kontrolera i usluga:

- ProductController i ProductService
- TransactionController i TransactionService
- CategoryController i CategoryService
- CompanyController i CompanyService
- OrderController i OrderService

Svaki od kontrolera i usluga ima mogućnost pregledavanja, kreiranja, ažuriranja i brisanja podataka u bazi. Nakon implementacije kontrolera klijent do svakog od kontrolera može doći preko ovih URI-a:

1. ProductController URI-ji:
    - GET - /api/products/
    - POST - /api/products/
    - PUT - /api/products/{id}
    - DELETE /api/products/{id}
2. TransactionController URI-ji:
    - GET - /api/transactions/
    - POST - /api/transactions/
    - PUT - /api/transactions/{id}
    - DELETE /api/transactions/{id}

3. CategoryController URI-ji
    - GET - /api/categories/
    - POST - /api/categories/
    - PUT - /api/categories/{id}
    - DELETE /api/categories/{id}
4. CompanyController URI-ji
    - GET - /api/companies/
    - POST - /api/companies/
    - PUT - /api/companies/{id}
    - DELETE /api/companies/{id}
5. OrderController URI-ji
    - GET - /api/orders/
    - POST - /api/orders/
    - PUT - /api/orders/{id}
    - DELETE /api/orders/{id}

Na slici ispod je prikazana implementacija ProductController-a. Http metode GET, POST, PUT i DELETE su implementirane pomoću anotacije @<ImeMetode>Mapping, primjer: @GetMapping.

```
@GetMapping("/")
public List<Product> getAllProducts() { return productRepository.findAll(); }

@GetMapping("/products/{id}")
public ResponseEntity<Product> getProductById(@PathVariable(value = "id") Long productId)
        throws ResourceNotFoundException {
    Product product = productRepository.findById(productId)
            .orElseThrow(() -> new ResourceNotFoundException("Product not found for this id :: " + productId));
    return ResponseEntity.ok().body(product);
}

@PostMapping("/products")
public Product createProduct(@Valid @RequestBody Product product) { return productRepository.save(product); }

@PutMapping("/products/{id}")
public ResponseEntity<Product> updateProduct(@PathVariable(value = "id") Long productId,
        @Valid @RequestBody Product productDetails) throws ResourceNotFoundException {
    final Product updatedProduct= productRepository.save(productDetails);
    return ResponseEntity.ok(updatedProduct);
}

@DeleteMapping("/products/{id}")
public Map<String, Boolean> deleteProduct(@PathVariable(value = "id") Long productId)
        throws ResourceNotFoundException {
    Product product = productRepository.findById(productId)
            .orElseThrow(() -> new ResourceNotFoundException("Product not found for this id :: " + productId));

    productRepository.delete(product);
    Map<String, Boolean> response = new HashMap<>();
    response.put( k: "deleted", Boolean.TRUE);
    return response;
}
```

Slika 4.23. Prikaz implementacije Product Controllera

### 4.3.5. Implementacija usluge za naručivanje pacijenata "Appointment service"

Usluga za naručivanje pacijenata upravlja narudžbama pacijenata na pregled te prihvaćanjem ili otkazivanjem narudžbe od strane liječnika. Usluga ostvaruje komunikaciju s:

- uslugom za upravljanje kartonima pacijenata kojoj šalje zahtjev za pregledom zdravstvenog kartona pacijenta
- uslugom za upravljanje receptima kojoj šalje zahtjev kreiranje recepta
- uslugom za upravljanje korisnicima koja šalje informaciju koja je trenutna uloga korisnika (admin, liječnik ili pacijent)

Implementacija, kao i kod svake mikrousluge u ovom diplomskom radu, počinje s alatom za kreiranje Spring Boot projekata Spring Initializr-om. Dodat će se sve iste ovisnosti (engl. dependency) kao u usluzi za upravljanje korisnicima. Mikrousluga za upravljanje lijekovima koristi PostgreSQL bazu podataka. Nakon kreiranja projekta potrebno je podesiti bazu podataka u konfiguracijskoj datoteci. Na slici ispod je prikazano podešavanje konfiguracije za PostgreSQL bazu podataka u Spring Boot-u:

```
spring.datasource.url=jdbc:postgresql://127.0.0.1:5432/appointments
spring.datasource.username=healthcare
spring.datasource.password=healthcare
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.database=POSTGRESQL
spring.jpa.database-platform = org.hibernate.dialect.PostgreSQL92Dialect
spring.jpa.open-in-view=true
spring.jpa.show_sql=false
spring.jpa.generate-ddl=false
spring.jpa.hibernate.use-new-id-generator-mappings=true
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.hibernate.ddl-auto=create
```

Slika 4.24. Podešavanje PostgreSQL baze podataka

Baza podataka se sastoji od dva entiteta: Appointment i Bill. Entiteti su spojeni vezom jedan na prema jedan. U Appointment tablicu se spremaju podaci iz usluge za upravljanje korisnicima. Točnije, sprema se id pacijenta (patient id) i id liječnika (doctor id).

U usluzi za naručivanje pacijenata implementirani su dva kontrolera i dvije usluge za spomenute entitete te kontroleri za komunikaciju između usluge za upravljanje lijekovima, usluge za upravljanje receptima i usluge za upravljanje kartonima pacijenta.

Povezivanje i komunikacija s drugim uslugama se odvija preko Spring Boot WebFlux WebClient sučelja koje šalje zahtjeve prema drugim uslugama preko zadanih pristupnih točki sustava. Na slici je prikazana implementacija između usluge za naručivanje pacijenata i usluge za upravljanje receptima. Točnije, pozivanje metode za kreiranje recepta u usluzi za upravljanje receptima.

```
private final WebClient webClient;
public ResponseEntity<PrescriptionDto> create(PrescriptionDto prescriptionDto){
    return webClient
            .post() WebClient.RequestBodyUriSpec
            .uri( s: "http://localhost:9000/receipt-service/api/prescriptions") WebClient.RequestBodySpec
            .body(BodyInserters.fromValue(prescriptionDto)) WebClient.RequestHeadersSpec<...>
            .accept(MediaType.APPLICATION_JSON) capture of ?
            .retrieve() WebClient.ResponseSpec
            .toEntity(PrescriptionDto.class) Mono<ResponseEntity<PrescriptionDto>>
            .block();
}
```

Slika 4.25. Pozivanje metode za kreiranje recepta

U metodi “create” je vidljivo da vraća objekt tipa PrescriptionDto kojeg popunjava usluga za upravljanje receptima. Metode koje WebClient poziva su:

- **post** - metoda koja daje informaciju da se šalje Http POST metoda.
- **uri** - metoda koja daje informaciju na koju pristupnu točku treba slati zahtjev.
- **body** - metoda koja umeće tijelo zahtjeva koji se šalje.
- **accept** - metoda koja daje informaciju o kojem se formatu razmjene podataka radi
- **retreive** - metoda koja dohvaća podatke iz druge mikrousluge.
- **toEntity** - metoda koja stvara novu instancu objekta PrescriptionDto i popunjava tu instancu podacima koji su u odgovoru.
- **block** - metoda koja daje informaciju da je zahtjev koji se šalje blokirajući i da je primijenjeno sinkrono slanje podataka.

### 4.3.6. Implementacija usluge za upravljanje receptima “Prescription service”

Usluga za upravljanje receptima radi operacije pregledavanja, kreiranja, ažuriranja i brisanja recepata. Usluga komunicira s:

- uslugom za upravljanje lijekovima preko koje šalje zahtjev za narudžbom lijeka

- uslugom za naručivanje pacijenata koja šalje zahtjev za kreiranjem ili pregledavanjem recepta

Implementacija, kao i kod svake mikrousluge u ovom diplomskom radu, počinje s alatom za kreiranje Spring Boot projekata Spring Initializr-om. Dodat će se sve iste ovisnosti (engl. dependency) kao u usluzi za upravljanje korisnicima. Mikrousluga za upravljanje lijekovima koristi PostgreSQL bazu podataka.

Baza podataka se sastoji od dva entiteta: Prescription i Medication. Entiteti su spojeni vezom jedan na prema više što znači da jedan recept može imati više lijekova. U Prescription entitetu sprema se "appointment_id" koji daje informaciju u kojem pregledu se prepisao recept. Na slici ispod je prikazan poziv usluge za upravljanje receptima prema usluzi za upravljanje lijekovima.

```
private final WebClient webClient;
public ResponseEntity<OrderDto> create(OrderDto orderDto){
    return webClient
            .post() WebClient.RequestBodyUriSpec
            .uri( s: "http://localhost:9000/pharmacy-service/api/orders") WebClient.RequestBodySpec
            .body(BodyInserters.fromValue(orderDto)) WebClient.RequestHeadersSpec<...>
            .accept(MediaType.APPLICATION_JSON) capture of ?
            .retrieve() WebClient.ResponseSpec
            .toEntity(OrderDto.class) Mono<ResponseEntity<OrderDto>>
            .block();
}
```

Slika 4.26. Pozivanje metode za kreiranje narudžbe

### 4.3.7. Implementacija usluge za otkrivanje dijabetesa "Diabetic service"

Implementacija usluge za otkrivanje dijabetesa je odrađena s Flask programskim okvirom. Ova mikrousluga je pisana programskim jezikom Python. Za predikciju će se koristiti diabetic_analysis_logistic_regression.py model koji će dati rezultat boluje li osoba od dijabetesa ili ne. Mikrousluga koja je kreirana Flask programskim okvirom je spojena na Eureka Discovery poslužitelj i klijent aplikacija će standardno moći komunicirati s ovom mikrouslugom iako je implementirana u drugoj tehnologiji. Implementirane su dvije klase jedna je za dohvaćanje dohvaćanje modela strojnog učenja, a druga je za umetanje unosnih parametara i postavljanje u model. Na slici ispod prikazano je učitavanje modela u aplikaciju.

```
import pickle
import pandas as pd

class LoadDiabeticModel:
    #Loading the model
    def __init__(self,MODEL_PATH):
        self.loaded_model = pickle.load(open(MODEL_PATH, 'rb'))

    def predict_class(self, pregnant,insulin,bmi,age,glucose,bp,pedigree):
        # initialize list of lists
        data = [[pregnant,insulin,bmi,age,glucose,bp,pedigree]]

        # Create the pandas DataFrame
        df = pd.DataFrame(data, columns = ['pregnant','insulin','bmi','age','glucose','bp','pedigree'])
        new_pred = self.loaded_model.predict(df)
        return new_pred
```

Slika 4.27. Učitavanje modela

Nakon učitavanja modela u drugoj klasi definirana je krajnja točka "/diabetic-diagnose" i metoda "diagnosis". U ovoj metodi se unose parametri s kojima će model strojnog učenja raditi i dati krajnji rezultat. Na slici 4.20. je prikazana implementacija metode "diagnosis".

```
#Define diagnosis route
@app.route("/diabetic-diagnose", methods=['POST'])
def diagnosis():
    name = request.form['name']
    age = request.form['age']
    pregnant = request.form['pregnant']
    insulin = request.form['insulin']
    bmi = request.form['bmi']
    pedigree = request.form['pedigree']
    glucose = request.form['glucose']
    bp = request.form['bp']
    #Predict on the given parameters
    prediction = model.predict_class(pregnant,insulin,bmi,age,glucose,bp,pedigree)
    print(prediction)
    #Route for result
    if prediction[0] == '1':
        return 1
    elif prediction[0] == '0':
        return 0
```

Slika 4.28. Metoda za unošenje parametara u model strojnog učenja

### 4.3.8. Prijava mikrousluga u Eureka Discovery sustav

Kako bi sve usluge funkcionirale i komunicirale međusobno potrebno ih je prijaviti preko Spring Cloud Eureka poslužitelja. Više o Spring Cloud Eureka poslužitelju se može pročitati u poglavljima 4.2.1. i 4.3.1. Sve usluge prijavljene su na isti način, a primjer jedne prijave je prikazan na slici ispod gdje je u pokretačku klasu mikrousluge dodana anotacija "@EnableDiscoveryClient"

```java
@SpringBootApplication
@EnableDiscoveryClient
public class UsersServiceApplication {

    public static void main(String[] args) { SpringApplication.run(UsersServiceApplication.class, args); }

}
```

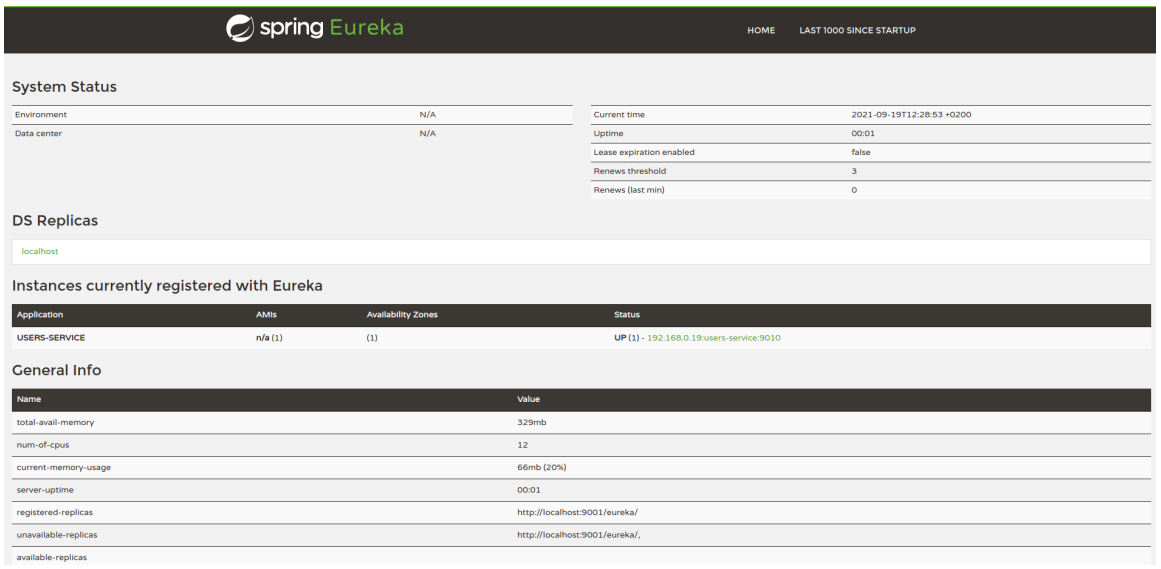
Slika 4.29. Omogućavanje Eureka Clienta

Nakon omogućavanja da se mikrousluga ponaša kao Eureka klijent, potrebno je podesiti postavke kako bi usluga bila vidljiva na Eureka poslužitelju. Na slici ispod su prikazane postavke koje služe za komuniciranje mikrousluge s Eureka poslužiteljem. Bitna postavka je eureka.client.serviceUrl.defaultZone koja daje informaciju na kojoj pristupnoj točki se nalazi Eureka poslužitelj.

```
server.port=9010
spring.application.name=users-service
spring.devtools.restart.enabled = true
eureka.client.serviceUrl.defaultZone = http://localhost:9001/eureka
```

Slika 4.30. Postavke mikrousluge

Nakon prijave u Eureka poslužitelj, mikrousluga bi trebala biti vidljiva na Eureka grafičkom sučelju. Eureka grafičkom sučelju se može pristupiti preko URL-a http://localhost:9001/. Na slici je prikazano grafičko sučelje Eureka poslužitelja i prijavljena mikrousluga unutar poslužitelja.




Slika 4.31. Grafičko sučelje Eureka poslužitelja

### 4.3.9. Komuniciranje mikrousluga pomoću Spring Cloud API Gateway-a

Nakon prijave svih mikrousluga u Eureka poslužitelj potrebno je implementirati Spring Cloud API Gateway kako bi sve mikrousluge imale isti pristupnu točku (engl. port). Kako bi usluge bile dostupne preko Gateway-a potrebno je podesiti postavke na slici ispod:

```
server.port=9000
spring.application.name=api-gateway
eureka.client.serviceUrl.defaultZone=http://localhost:9001/eureka

spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.discovery.locator.lower-case-service-id=true
```

Slika 4.32. Podešavanje Spring Cloud API Gateway-a

Za omogućavanje komunikacije između mikrousluga bitne su zadnje dvije postavke sa slike. Nakon pokretanja projekta API Gateway sve mikrousluge mogu komunicirati preko pristupne točke (engl. port) 9000. Svi pristupni sustavi su navedeni u poglavlju 4.3.11.

## 4.4. Programsko rješenje na strani klijenta

U ovom potpoglavlju je predstavljeno programsko rješenje na strani klijenta. Programsko rješenje je implementirano u Angular programskom okviru. Sustav trenutno ima jednog klijenta koji komunicira sa svim mikrouslugama. Komunikacija se odvija preko ulaznog URL-a: http://localhost:9000/ime-mikrousluge/api/*. Može se primjetiti da je za svaka mikrousluga prijavljena preko pristupne točke 9000 zbog Spring Boot Cloud API Gateway-a. Na početku je potrebno instalirati sve pakete koji su bitni za Angular programski okvir. Početni korak je instalacija Angular klijenta s naredbom sa slike.

```
npm install -g @angular/cli
```

Slika 4.33. Instalacija Angular klijenta

Nakon instalacije potrebno je izvršiti još jednu naredbu kako bi se generirao novi Angular projekt. Generiranje novog projekta se izvršava s naredbom sa slike ispod.

```
ng new home-healthcare-client
```

Slika 4.34. Generiranje novog projekta

Generiranjem novog projekta može započeti implementacija klijenta. Angular programska arhitektura se sastoji od komponenti te praktički svaka komponenta predstavlja jednu implementaciju funkcionalnosti. U idućim poglavljima opisane su neke od najvažnijih komponenti za sustav zdravstva od kuće.

### 4.4.1. Implementacija prijave i registracije korisnika

Za prijavu i registraciju korisnika potrebne su dvije implementacija. Jedna je implementacija usluge za autentifikaciju koja se nalazi u klijent projektu, a druga je implementacija komponente za prijavu (engl. login component). Usluga za autentifikaciju upravlja korisničkim informacijama i pristupa krajnjim pristupnim točkama za prijavu i za registraciju korisnika u mikrousluzi za upravljanje korisnicima. Na slici ispod su vidljive metode koje pozivaju pristupne točke za registraciju i prijavu korisnika.

```
export class AuthService {

  constructor(private http: HttpClient) { }

  login(credentials): Observable<any> {
    return this.http.post( url: AUTH_API + 'signin', body: {
      username: credentials.username,
      password: credentials.password
    }, httpOptions);
  }

  register(user): Observable<any> {
    return this.http.post( url: AUTH_API + 'signup', body: {
      username: user.username,
      email: user.email,
      password: user.password,
      role: user.role
    }, httpOptions);
  }
}
```

Slika 4.35. Metode za registraciju i prijavu korisnika

Nakon što se pozove registracija ili prijava korisnika klasa “AuthService” podatke prosljeđuje usluzi za spremanje Json Web Žetona “TokenStorageService” koji sprema dobiveni žeton dok traje sesija. Na slici je prikazana implementacija “TokenStorageService”.

```
export class TokenStorageService {

  constructor() { }

  signOut() {
    window.sessionStorage.clear();
  }

  public saveToken(token: string) {
    window.sessionStorage.removeItem(TOKEN_KEY);
    window.sessionStorage.setItem(TOKEN_KEY, token);
  }

  public getToken(): string {
    return sessionStorage.getItem(TOKEN_KEY);
  }

  public saveUser(user) {
    window.sessionStorage.removeItem(USER_KEY);
    window.sessionStorage.setItem(USER_KEY, JSON.stringify(user));
  }

  public getUser() {
    return JSON.parse(sessionStorage.getItem(USER_KEY));
  }
}
```

Slika 4.36. Implementacija klase TokenStorageService

Zadnji korak implementacije registriranja i prijave korisnika je implementacija komponenti za registraciju i prijavu. Komponenta za registraciju sadrži jednu metodu naziva “register” je li registracija uspješna ili nije. Ovisno o uspješnosti registracije metoda vraća poruku. Registracijska komponenta se sastoji od unosnog obrasca koji sadrži unosna polja za svako polje koje je potrebno unijeti prilikom registracije korisnika. Unosni obrazac je HTML datoteka. Također, registracijska komponenta sadrži i CSS datoteku koja služi za uređenje grafičkog sučelja. Na slici ispod prikazana je registracijska klasa naziva “RegisterComponent”.

```
export class RegisterComponent implements OnInit {
  form: any = {};
  isSuccessful = false;
  isSignUpFailed = false;
  errorMessage = '';

  constructor(private authService: AuthService) { }

  ngOnInit() {
  }

  onSubmit() {
    this.authService.register(this.form).subscribe(
      next: data => {
        console.log(data);
        this.isSuccessful = true;
        this.isSignUpFailed = false;
      },
      error: err => {
        this.errorMessage = err.error.message;
        this.isSignUpFailed = true;
      }
    );
  }
}
```

Slika 4.37. Registracijska komponenta

Komponenta za prijavu korisnika u sustav (engl. login component) je slična registracijskoj komponenti. Komponenta provjerava uspješnost prijava. Ako je korisnik autoriziran i autentificiran uspješno komponenta šalje poruku uspješnosti i prijavljuje korisnika u sustav. Nakon toga prosljeđuje se zahtjev na "user-service" koji po ulozi određuje koju stranicu otvara. U sljedećem potpoglavlju je opisana implementacija "user-service-a".

### 4.4.2. Implementacija uloga admin, pacijent i liječnik

Nakon uspješne prijave potrebno je utvrditi s kojom se ulogom korisnik prijavio. Postoje tri uloge: admin, liječnik i pacijent. Rukovanje ulogama odrađuje User usluga unutar Angular projekta koja prosljeđuje novu komponentu u ovisnosti o dobivenoj ulozi od strane

komponente za prijavu. Na slici ispod je prikazana implementacija User usluge "user-service".

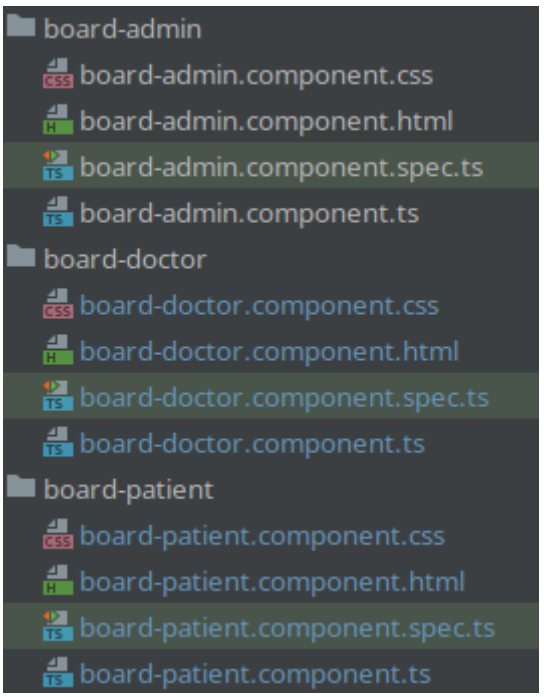
```
export class UserService {

  constructor(private http: HttpClient) { }

  getPublicContent(): Observable<any> {
    return this.http.get( url: API_URL + 'all',  options: { responseType: 'text' });
  }

  getPatientBoard(): Observable<any> {
    return this.http.get( url: API_URL + 'patient',  options: { responseType: 'text' });
  }

  getDoctorBoard(): Observable<any> {
    return this.http.get( url: API_URL + 'doctor',  options: { responseType: 'text' });
  }

  getAdminBoard(): Observable<any> {
    return this.http.get( url: API_URL + 'admin',  options: { responseType: 'text' });
  }
}
```

Slika 4.38. Usluga User

Na slici su vidljive četiri metode:

- **getPublicContent()** - dohvaća javni sadržaj dostupan svim korisnicima
- **getPatientBoard()** - dohvaća sadržaj stranice koji je dostupan samo ulozi pacijenta
- **getDoctorBoard()** - dohvaća sadržaj stranice koji je dostupan samo ulozi liječnika
- **getAdminBoard()** - dohvaća sadržaj stranice koji je dostupan samo ulozi admina

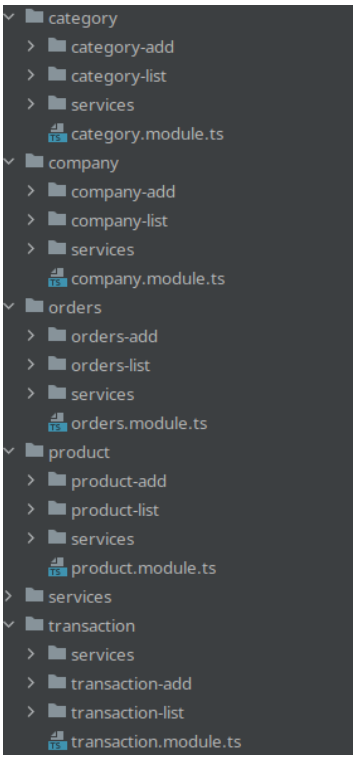
Svaka nabrojana uloga ima svoju komponentu u koju se implementira sadržaj i funkcionalnost za određenu ulogu. Na slici 4.29. su prikazane komponente za admina, pacijenta i liječnika.




Slika 4.39. Komponente admin, liječnik i pacijent

### 4.4.3. Implementacija usluge za upravljanje lijekovima

Na klijent strani, usluga za upravljanje lijekovima, je implementirana tako da svaki entitet ima svoju komponentu. Implementirano je pet komponenti: Product, Order, Transaction, Category i Company. Na slici je prikazan pregled svih ranije spomenutih komponenti.




Slika 4.40. Pregled komponenti

Kao što je vidljivo na slici svaka komponenta se sastoji od usluge (engl. services) i dvije potkomponente: komponenta za dodavanje i komponenta za pregled. U uslugama se nalaze metode za dohvaćanje, dodavanje, ažuriranje i brisanje koje pozivaju API krajnje točke u od mikrousluge za upravljanje lijekovima. Temeljni URL za mikrouslugu za upravljanje lijekovima je http://localhost:9000/pharmacy-service/api/. Na sljedećoj slici je prikazan "ProductService" koji rukuje osnovnim operacijama nad proizvodima (engl. product). Na vrhu klase je definiran temeljni URL koji pozivaju sve metode i ako treba dodavaju se URI-ji kako bi metoda poslala zahtjev na pravilnu krajnju točku.

```
export class ProductService {

  private baseUrl = 'http://localhost:9000/pharmacy-service/api/products';

  constructor(private http: HttpClient) { }

  getProduct(id: number): Observable<Product> {
    return this.http.get<Product>( url: `${this.baseUrl}/${id}`);
  }

  createProduct(product: Object): Observable<Object> {
    return this.http.post( url: `${this.baseUrl}`, product);
  }

  updateProduct(id: number, value: any): Observable<Object> {
    return this.http.put( url: `${this.baseUrl}/${id}`, value);
  }

  deleteProduct(id: number): Observable<any> {
    return this.http.delete( url: `${this.baseUrl}/${id}`,  options: { responseType: 'text' });
  }

  getProductsList(): Observable<any> {
    return this.http.get( url: `${this.baseUrl}/all-products`);
  }
}
```

Slika 4.41. Implementacija "ProductService"

Nakon što se poslužitelj vrati traženi zahtjev odgovor se prosljeđuje na komponente koje postavljaju sadržaj zahtjeva u HTML datoteku. Komponente za dodavanje i pregled također imaju metode za pregled, dodavanje, ažuriranje i brisanje.

Jedna od funkcionalnosti koja ova mikrousluga ima, a ostale nemaju je ispis računa prilikom naručivanja lijeka. Na slici 4.32. je prikazano dodavanje "Ispiši" gumba na stranicu kako bi se mogao ispisati kreirani račun.

```
<div style="text-align: center;">
  <button type="button" class="btn btn-danger" onclick="window.print()">Print Receipt</button>
</div>
```

Slika 4.42. Dodavanje "Ispiši" gumba

### 4.4.4. Implementacija usluge za naručivanje pacijenata

Usluga za naručivanje pacijenata implementirana je u dvije komponente. Appointment i Bill. Appointment komponenta rukuje svim funkcionalnostima za naručivanje, a Bill komponenta rukuje svim funkcionalnostima za račune. Temeljni URL za mikrouslugu za naručivanje pacijenata je http://localhost:9000/appointment-service/api/. Funkcionalnost koja je drugačija u odnosu na druge komponente je dodavanje kalendara na grafičko sučelje aplikacije. Komponenta koja izvršava dodavanje datuma i vremena se zove "Scheduler". Na slici prikazana implementacija "Scheduler" komponente.

```
ngAfterViewInit(): void {
  this.ds.getResources().subscribe( next: result => this.config.resources = result);

  const from = this.scheduler.control.visibleStart();
  const to = this.scheduler.control.visibleEnd();
  this.ds.getEvents(from, to).subscribe( next: result => this.events = result);

}
```

4.43. Komponenta Scheduler

Za "Scheduler" komponentu se koristila "day-pilot-scheduler" biblioteka koja je omogućila pregled dana u tjednu, sati i mjeseci. Nakon implementaciju u TypeScript programskom jeziku potrebno je "Scheduler" pozvati u HTML datoteku koja će na stranici prikazati kalendar.

```
selector: 'scheduler-component',
template: `
<div class="body">
  <h1>Scheduler</h1>
  <daypilot-scheduler [config]="config" [events]="events" #scheduler></daypilot-scheduler>
</div>
`,
styles: [`.body { padding: 10px; }`]
```

4.44. HTML implementacija Scheduler komponente

### 4.4.5. Implementacija usluge za upravljanje receptima

Usluga za upravljanje receptima se sastoji od “Prescription” paketa koji sadrži više komponenti. Komponente od kojih se sastoji “Prescription” paket su:

- **prescription-add** - komponenta za dodavanje recepta
- **prescription-search** - komponenta za pretraživanje recepata
- **prescription-update** - komponenta za ažuriranje recepta

# 5. ANALIZA IMPLEMENTIRANOG RJEŠENJA I KORIŠTENJE APLIKACIJE

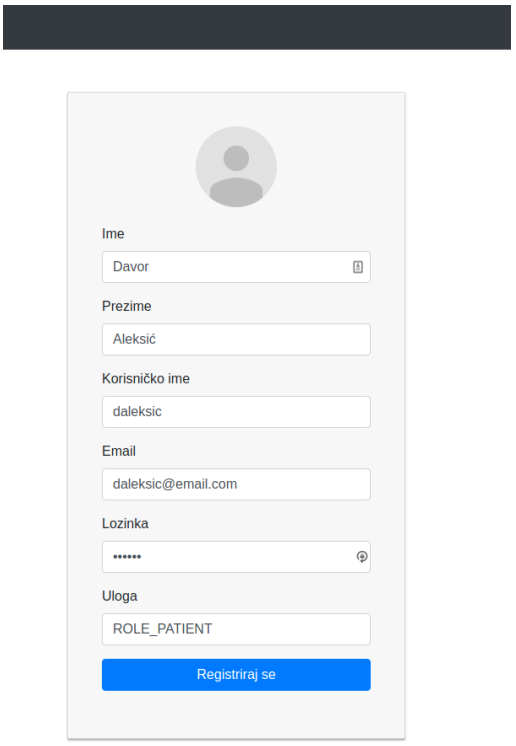
U ovom poglavlju opisane su upute za korištenje aplikacije te su pomoću testova opterećenosti prikazani rezultati performansi arhitekture mikrousluga.

## 5.1. Način korištenja aplikacije

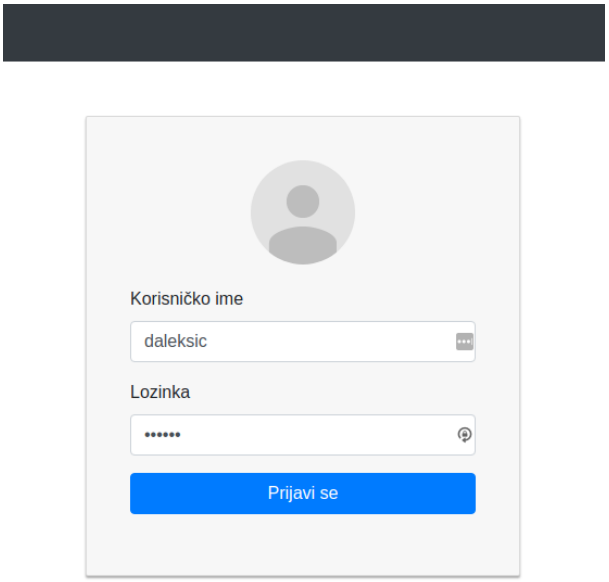
Kako bi se pokrenuo sustav potrebno je pokrenuti osam projekata i to redom: Eureka Discovery poslužitelj, API-Gateway, sve mikrousluge i Angular klijentsku aplikaciju. Nakon svih pokretanja sustavu se može pristupiti preko URL-a: http://localhost:9000/. Glavna podjela pogleda sustava je na pogled liječnika, pogled pacijenta i pogled admina.

### 5.1.1. Registracija i prijava korisnika

Na slici 5.1. prikazan je zaslon za registraciju korisnika. Kao što je ranije spomenuto kroz diplomski rad prilikom registracije se dodaje korisnik koji ima ulogu liječnika, pacijenta ili admina. Na zaslonu za registraciju korisnik mora unijeti: ime, prezime, korisničko ime, adresu e-pošte, lozinku i ulogu. Nakon uspješne registracije korisnik je preusmjeren na zaslon za prijavu koji je prikazan na slici 5.2. Prilikom prijave korisnik je dužan unijeti korisničko ime i lozinku.

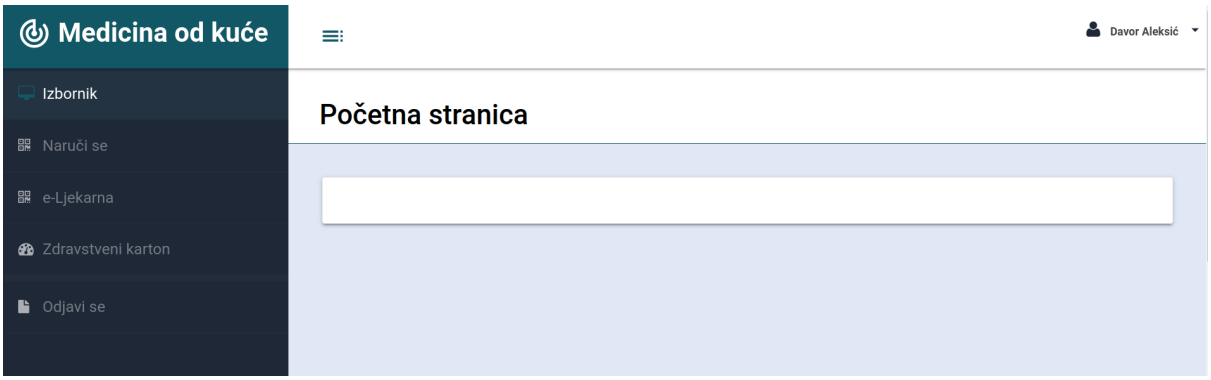



Slika 5.1. Zaslon za registraciju korisnika




Slika 5.2. Zaslon za prijavu korisnika

Nakon uspješne prijave korisnik je preusmjeren na početnu stranicu. Na slici 5.3. je prikazan izgled početne stranice. S lijeve strane je izbornik preko kojeg korisnik može odabrati opciju koju želi.

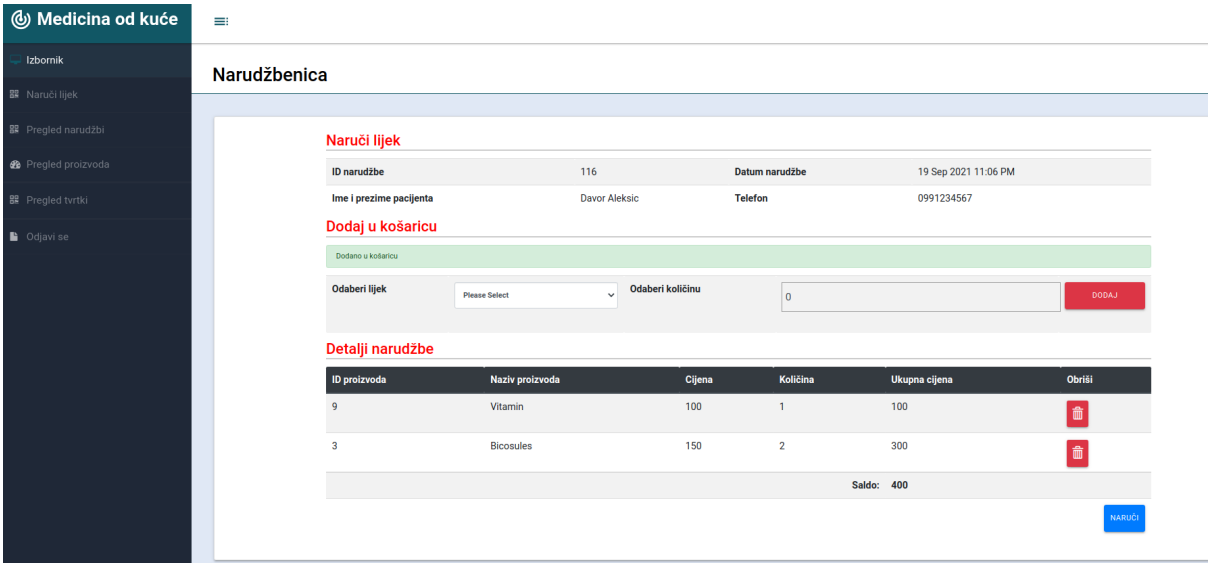



Slika 5.3. Početna stranica

### 5.1.1. Naručivanje i pregled lijekova

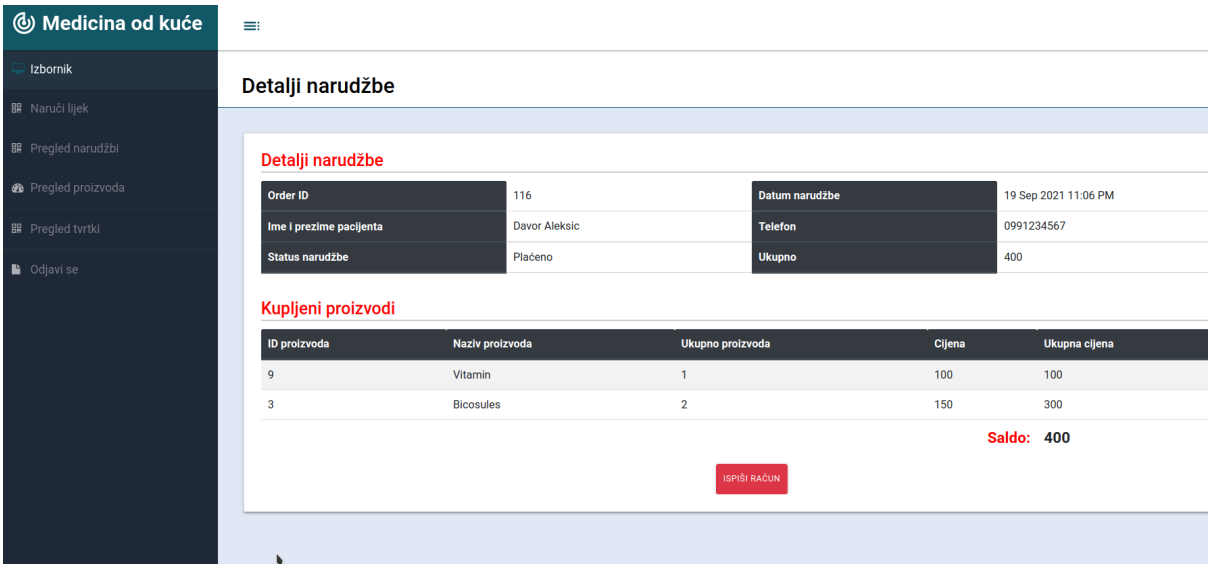
Kao što je vidljivo na slici 5.3. korisnik u izborniku s lijeve strane ima mogućnost izbora "e-Ljekarne". Nakon odabira "e-Ljekarne" korisnik može odabrati opciju "Naruči lijek". Odabirom te opcije otvara se obrazac za narudžbu lijeka prikazan na slici 5.4. Dovoljno je

samo da se odabere lijek i odredi količina te odabere opcija “Dodaj”. Klikom na gumb dodaj lijek se pojavljuje u detaljima narudžbe. Dodavati lijekove se može više puta.



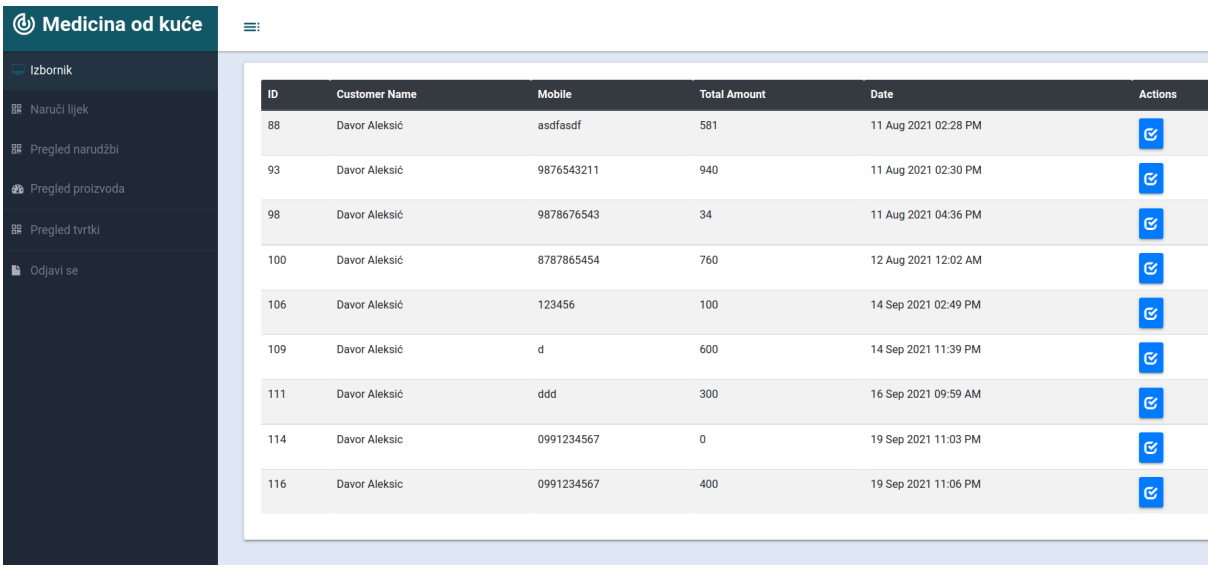

Slika 5.4. Unos narudžbe

Odabirom opcije “Naruči” korisniku se prikazuje zaslon sa slike 5.5 s detaljima narudžbe i mogućnošću ispisivanja računa.



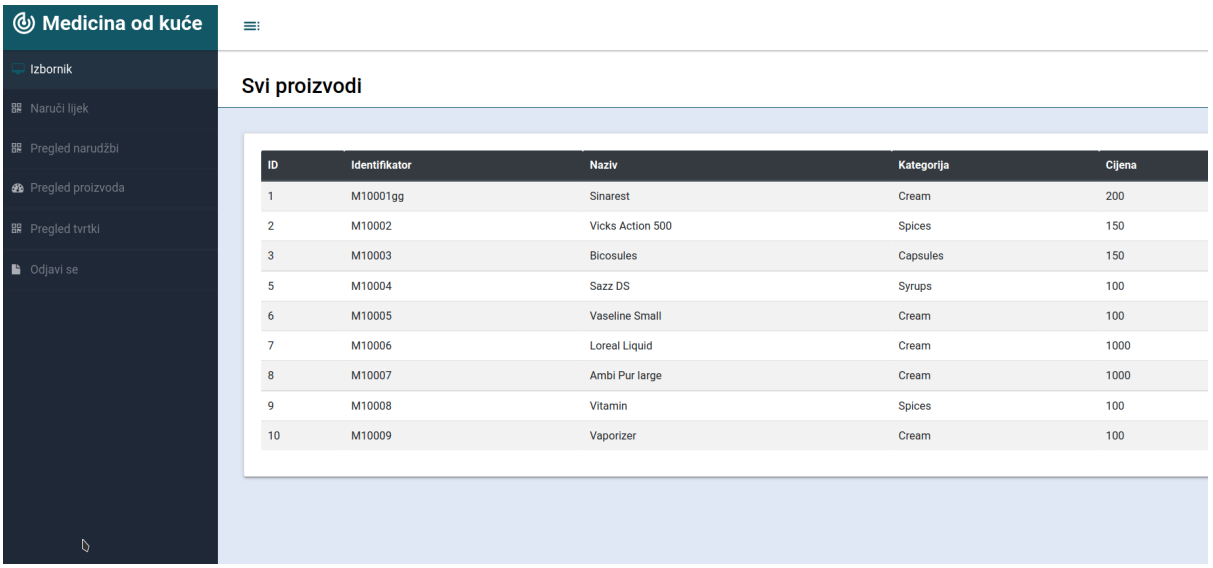

Slika 5.5. Detalji narudžbe

Uz naručivanje lijekova korisnik u “e-Ljekarni” može pregledati povijest svojih narudžbi. Na slici 5.6. prikazan je zaslon s pregledom svih narudžbi.




Slika 5.6. Pregled narudžbi

Također, s istog zaslona pritiskom na "Pregled proizvoda" korisniku se izlistavaju svi postojeći lijekov. Na slici 5.7. je prikazan zaslon s popisom svih lijekova u bazi.

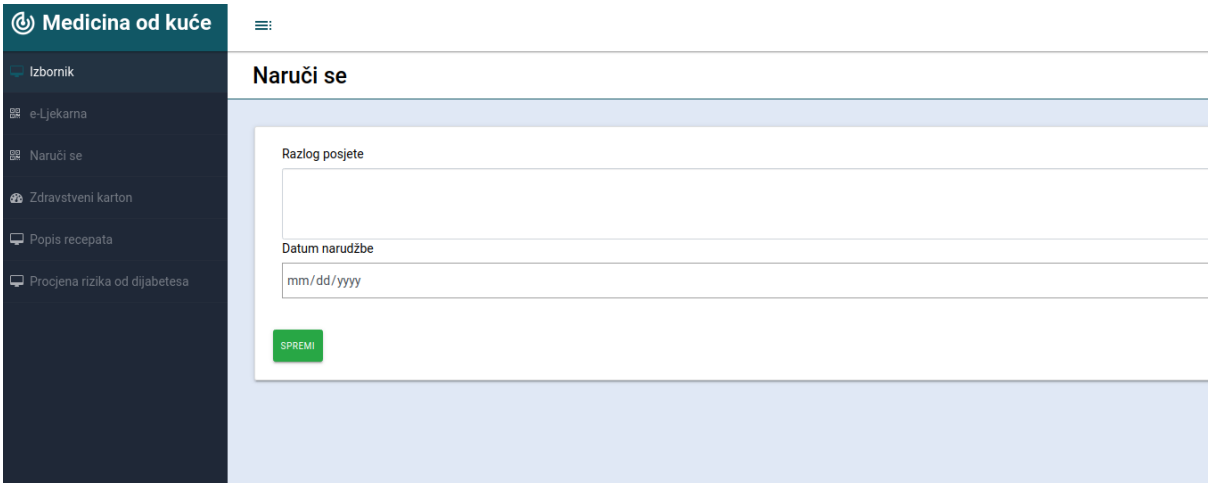



Slika 5.7. Popis svih lijekova

### 5.1.2. Interakcija između pacijenta i liječnika pomoću sustava za naručivanje

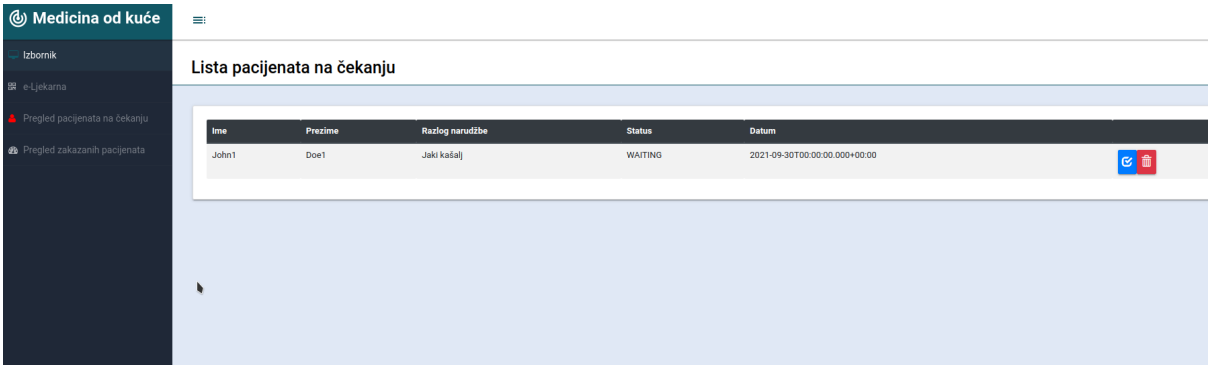
Kao što je obrađeno u poglavlju 3.2.3 korisnik s ulogom pacijenta ima mogućnost naručivanja kod liječnika. Pritiskom na "Naruči se" pacijentu se otvara unosni obrazac s tri unosna polja: odaberi liječnika, datum narudžbe i opis simptoma. Klikom na gumb "Naruči

se" liječniku dolazi obavijest da ima narudžbu na čekanju. Slika 5.8. prikazuje unosni obrazac za naručivanje.



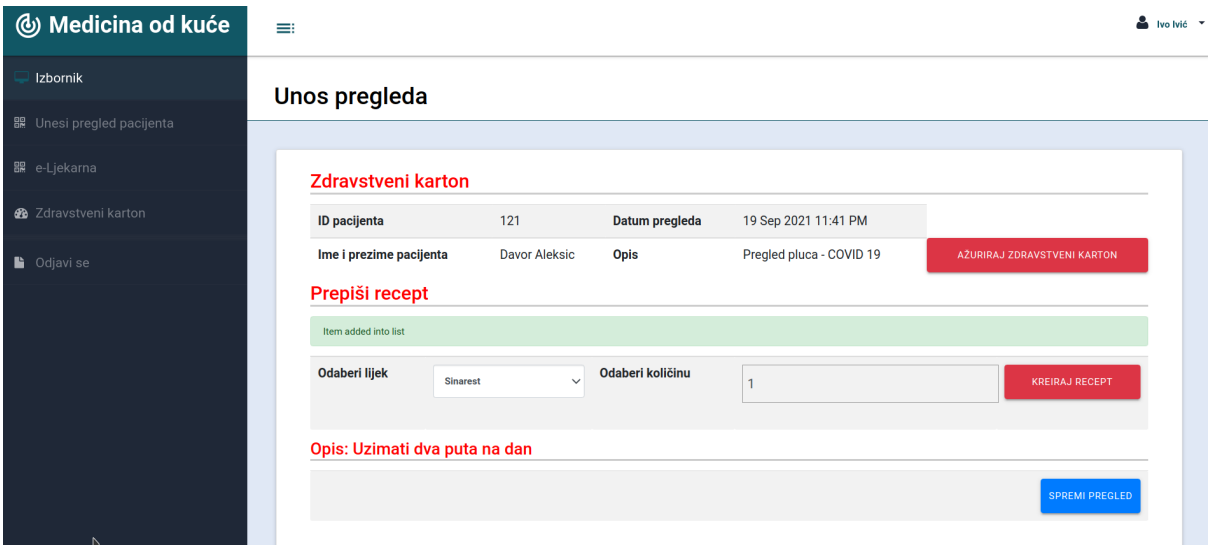

Slika 5.8. Obrazac za naručivanje

Nakon što se pacijent naručio prijaviti ćemo se s ulogom liječnika kako bi se vidjela obavijest koja označava da liječnik ima pacijenata na čekanju. Ako liječnik ima pacijenata na čekanju u lijevom dijelu ekrana mu se zacrveni ikona koja predstavlja korisnika. Klikom na ikonu otvara se "Pregled pacijenata na čekanju". Liječnik može prihvatiti prihvatiti i odbiti zahtjev za pregledom. Na slici 5.9. prikazan je zaslon koji daje listu svih pacijenata na čekanju.

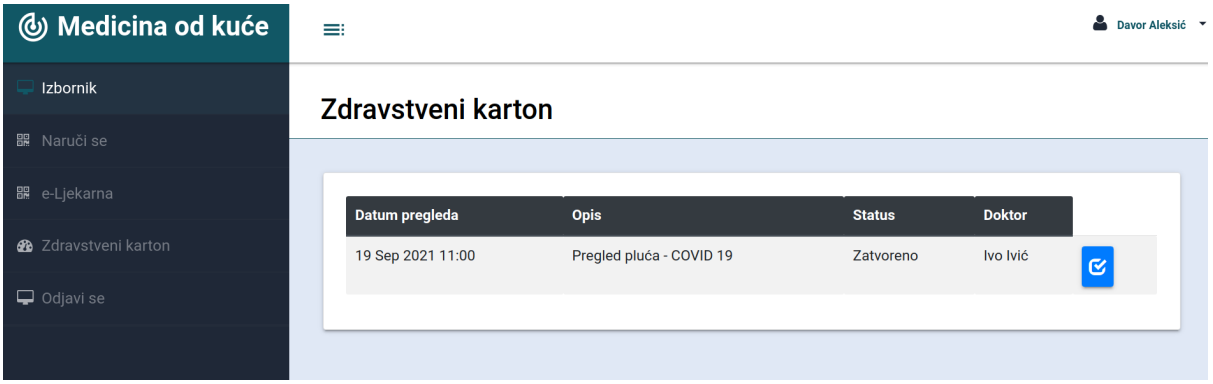



Slika 5.9. Pregled pacijenata na čekanju

Osim prihvaćanja pacijenta liječnik ima mogućnost unosa pregleda. Klikom na "Unesi pregled pacijenta" otvara se unosna forma u kojoj liječnik unosi opis pregleda i ažurira ga u zdravstveni karton pacijenta. Također, u unosnoj formi liječnik može prepisati recept te dati opis receptu. Klikom na "Spremi pregled", pregled se sprema u bazu podataka i pacijentu je vidljiv pregled u zdravstvenom kartonu. Slika 5.10. prikazuje unos pregleda od strane liječnika, a slika 5.11. prikazuje pregled zdravstvenog kartona od strane pacijenta.




Slika 5.10. Unos pregleda




5.11. Pregled zdravstvenog kartona

Klikom na gumb “Odjavi se” korisnik se odjavljuje iz sustava. Nakon odjave prikazuje se početna stranica s ponovnom mogućnošću prijave ili registracije korisnika.

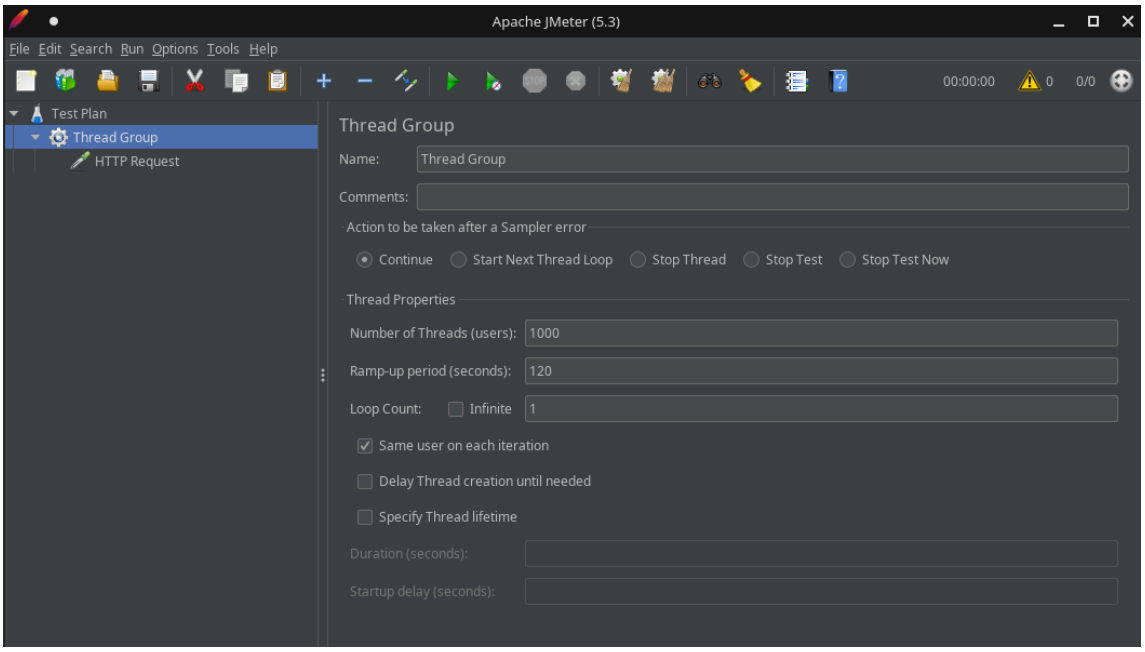
## 5.2. Testiranje i analiza rada sustava

U ovom diplomskom radu najveća pažnja je postavljena na mikrousluge. Kako je na početku diplomskog rada napisano da je jedna od najvećih prednosti arhitekture mikrousluga brzina sada će se testirati mikrousluge pod opterećenjem. Glavna funkcionalnost testa opterećenja je mjerenje performansi. Alat JMeter je korišten za mjerenje performansi. Testirat će se performanse usluge za upravljanje korisnicima. Točnije, http://localhost:9000/user-service/api/users krajnja točka koja služi za dohvaćanje svih korisnika iz baze. JMeter alat će se podesiti na 1000 usporednih korisnika svake sekunde. Sustav je testiran s tri testa:

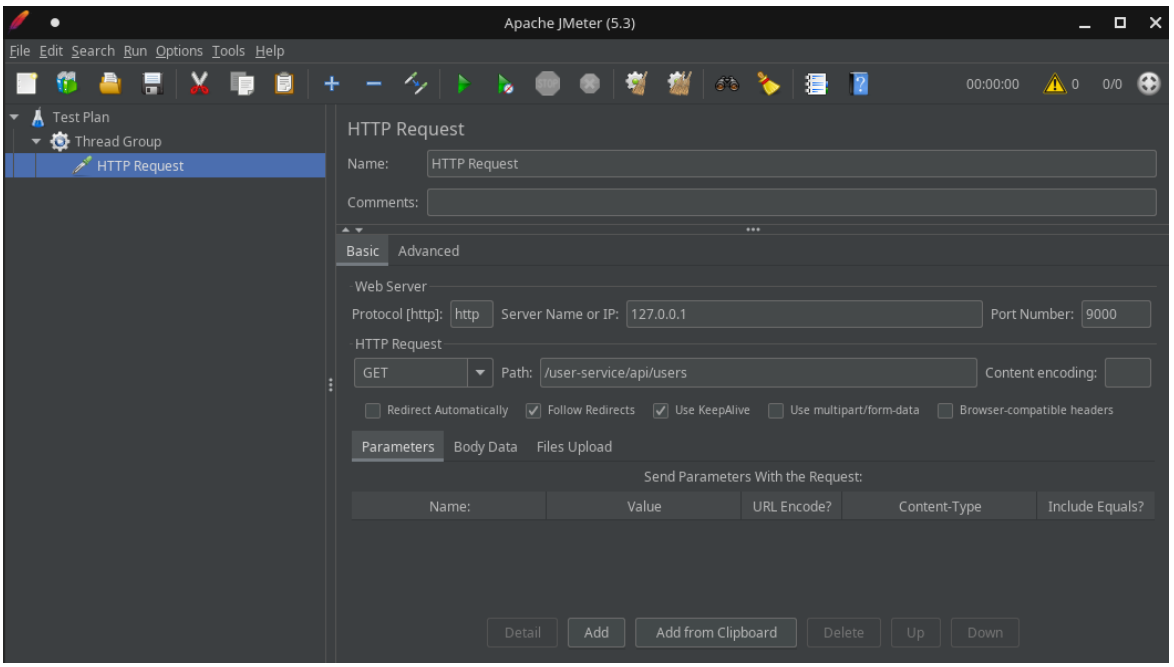
- 1000 usporednih korisnika
- 2000 usporednih korisnika
- 10000 usporednih korisnika

### 5.2.1. Prvi test (1000 usporednih korisnika, jedna instanca)

Prvi test s 1000 usporednih korisnika može početi. Slika 5.12. prikazuje podešavanje opterećenja na 1000 usporednih korisnika, a slika 5.13 prikazuje podešavanje Http zahtjeva.




Slika 5.12. Podešavanje opterećenja (1000 usporednih korisnika)




Slika 5.13. Podešavanje Http zahtjeva

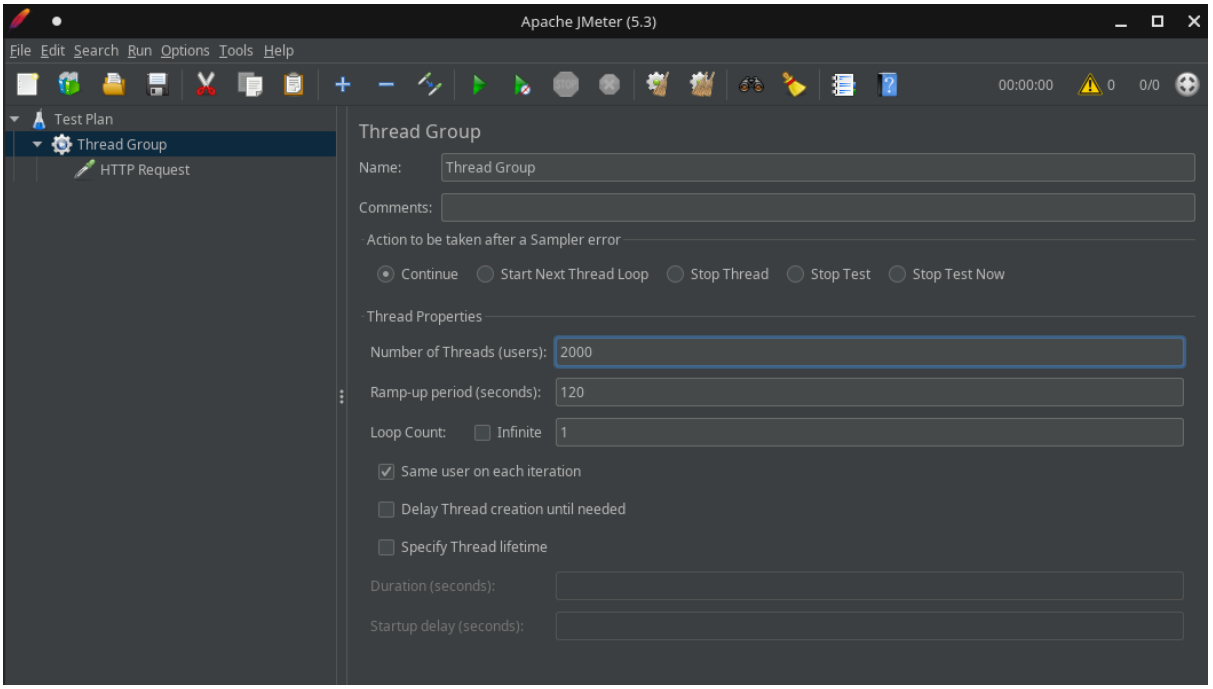
Prilikom testiranja nije bilo nikakvih usporavanja aplikacije. Usluga za upravljanje korisnicima je radila standardno. Rezultat testa je na prikazan na slici 5.14., a vidljivo je da nije bilo nikakvih grešaka i prosječno trajanje zahtjeva je bilo od jedne sekunde do tri sekunde što je prihvatljivo stanje

```
summary +    468 in 00:00:22 =   21.5/s Avg:   989 Min:    75 Max:  6760 Err:     0 (0.00%) Active: 181 Started: 181 Finished: 0
summary +   2060 in 00:00:30 =   68.7/s Avg:  2151 Min:    72 Max: 30404 Err:     0 (0.00%) Active: 430 Started: 430 Finished: 0
summary =   2528 in 00:00:52 =   48.8/s Avg:  1936 Min:    72 Max: 30404 Err:     0 (0.00%)
summary +   2173 in 00:00:30 =   72.4/s Avg:  3283 Min:    88 Max: 24316 Err:     0 (0.00%) Active: 679 Started: 679 Finished: 0
summary =   4701 in 00:01:22 =   57.5/s Avg:  2559 Min:    72 Max: 30404 Err:     0 (0.00%)
summary +   2719 in 00:00:30 =   90.4/s Avg:  4926 Min:   119 Max: 48715 Err:     0 (0.00%) Active: 929 Started: 929 Finished: 0
summary =   7420 in 00:01:52 =   66.3/s Avg:  3426 Min:    72 Max: 48715 Err:     0 (0.00%)
summary +   2328 in 00:00:30 =   77.7/s Avg:  7742 Min:  3007 Max: 70258 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =   9748 in 00:02:22 =   68.7/s Avg:  4457 Min:    72 Max: 70258 Err:     0 (0.00%)
summary +   3980 in 00:00:30 =  132.8/s Avg:  3152 Min:   115 Max: 98525 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =  13728 in 00:02:52 =   79.9/s Avg:  4079 Min:    72 Max: 98525 Err:     0 (0.00%)
summary +   2599 in 00:00:30 =   86.7/s Avg:  4288 Min:   374 Max: 121147 Err     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =  16327 in 00:03:22 =   80.9/s Avg:  4112 Min:    72 Max: 121147 Err     0 (0.00%)
summary +   2830 in 00:00:30 =   94.4/s Avg:  4465 Min:   322 Max: 140884 Err     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =  19157 in 00:03:52 =   82.7/s Avg:  4164 Min:    72 Max: 140884 Err     0 (0.00%)
summary +   4414 in 00:00:30 =  146.4/s Avg:  1719 Min:   124 Max: 76019 Err:     0 (0.00%) Active: 1000 Started: 1000 Finished: 0
summary =  23571 in 00:04:22 =   90.0/s Avg:  3706 Min:    72 Max: 140884 Err     0 (0.00%
summary +   4709 in 00:00:30 =  156.2/s Avg:  1493 Min:   253 Max: 93231 Err:     0 (0.00%  Active: 1000 Started: 1000 Finished: 0
summary =  28280 in 00:04:52 =   96.8/s Avg:  3338 Min:    72 Max: 140884 Err     0 (0.00%)
```

Slika 5.14. Rezultat prvog testa

### 5.2.2. Drugi test (2000 usporednih korisnika, jedna instanca)

Drugi test počinje s namještanjem JMeter alata na 2000 usporednih korisnika. U drugom testu cilj je vidjeti može li jedna instanca mikrousluge izdržati napad 2000 usporednih korisnika. Http zahtjev ostaje isti kao u prvom testu. Na slici 5.15. prikazano je podešavanje JMetera na 2000 korisnika.




Slika 5.15. Podešavanje opterećenja (2000 usporednih korisnika)

Nakon pokretanja testa čekalo se dosta dugo na izvršavanje pojedinih zahtjeva. Iako nije bilo nikakvih grešaka test je stao na 1191. izvršenom korisniku kao što je vidljivo na slici 5.16. što znači da sustav nije mogao podnijeti opterećenje od 2000 usporednih korisnika. Pošto već s 2000 usporednih korisnika sustav nije mogao izdržati. Idući test neće biti s 10000 usporednih korisnika nego će se opet slati 2000, ali pokrenuti će se dvije instance usluge za upravljanjem korisnicima. S ovim testom će se vidjeti hoće li nova instanca mikrousluge donijeti potrebnu propusnost.

```
summary +    588 in 00:00:14 =   43.5/s Avg:   127 Min:    73 Max:   753 Err:     0 (0.00%) Active: 223 Started: 223 Finished: 0
summary +   4152 in 00:00:30 =  138.3/s Avg:   940 Min:    79 Max: 10575 Err:     0 (0.00%) Active: 720 Started: 720 Finished: 0
summary =   4740 in 00:00:44 =  108.9/s Avg:   839 Min:    73 Max: 10575 Err:     0 (0.00%)
summary +   3369 in 00:00:30 =  111.4/s Avg:  4120 Min:   404 Max: 29905 Err:     0 (0.00%) Active: 1191 Started: 1191 Finished: 0
```

Slika 5.16. Rezultat drugog testa

### 5.2.3. Treći test (2000 usporednih korisnika, dvije instance)

Kao što je najavljeno u prošlom potpoglavlju treći test se izvršava s dvije instance usluge za upravljanje korisnicima. Generiranje više instanci omogućuje Spring Eureka poslužitelj i Spring Cloud API Gateway. Nakon pokretanja testa dobio se rezultat sa slike 5.17.

```
summary +   4764 in 00:00:30 =  158.7/s Avg:  9032 Min:   329 Max: 95943 Err:     1 (0.02%) Active: 2000 Started: 2000 Finished: 0
summary =  25714 in 00:02:44 =  156.6/s Avg:  4547 Min:    76 Max: 95943 Err:     1 (0.00%)
summary +   5036 in 00:00:30 =  167.8/s Avg:  9919 Min:   267 Max: 120786 Err:     0 (0.00%) Active: 2000 Started: 2000 Finished: 0
summary =  30750 in 00:03:14 =  158.3/s Avg:  5427 Min:    76 Max: 120786 Err:     0 (0.00%)
summary +   5068 in 00:00:30 =  169.1/s Avg:  8961 Min:   280 Max: 152058 Err:     0 (0.00%) Active: 2000 Started: 2000 Finished: 0
summary =  35818 in 00:03:44 =  159.8/s Avg:  5927 Min:    76 Max: 152058 Err:     0 (0.00%)
summary +   5121 in 00:00:30 =  170.6/s Avg:  9104 Min:   222 Max: 153682 Err:     0 (0.00%) Active: 2000 Started: 2000 Finished: 0
summary =  40939 in 00:04:14 =  161.0/s Avg:  6324 Min:    76 Max: 153682 Err:     0 (0.00%)
summary +   4828 in 00:00:30 =  161.0/s Avg:  9376 Min:   251 Max: 195133 Err:     0 (0.00%) Active: 2000 Started: 2000 Finished: 0
```

5.17. Rezultat trećeg testa

Iz rezultata je vidljivo da je treći test prošao i da su svih 2000 usporednih korisnika izvršeni što znači da je pomoglo povećanje broja instanci.

### 5.2.4. Analiza dobivenih rezultata

Testiranje arhitekture mikrousluga je proteklo u skladu s očekivanjima. Na manjem broju usporednih korisnika bez problema je radila jedna instanca mikrousluge. Već na 2000 usporednih korisnika stvarali su se problemi u performansama i zahtjevi prema sustavu nisu prolazili. Povećanjem broja instanci, usluge za upravljanje korisnicima, s jedne na dvije dobili su se brze performanse i svi izvršeni zahtjevi. Daljnjim testiranjem s dvije instance dobio se rezultat od oko 5500 usporednih korisnika. Dakle, na 5500 usporednih korisnika padaju performanse kada se testira s dvije instance. Ako želimo poboljšanje performansi i više usporednih korisnika u sustavu potrebno je opet povećavati broj instance.

# 6. ZAKLJUČAK

U ovom diplomskom radu je ostvareno programsko rješenje zdravstva od kuće s primjenom arhitekture mikrousluga. Analizom nekoliko najpopularnijih arhitekturnih dizajna u zdravstvu, mikrousluge su se pokazale kao idealan arhitekturni dizajn za zdravstveni sustav. Kako bi se dobila arhitektura mikrousluga prvo je potrebno osmisliti kvalitetan dizajn, a zatim krenuti u implementaciju. Implementacija je pratila definirane funkcionalnosti i zahtjeve. U praktičnom dijelu rada za razvoj zdravstvenog sustava od kuće koristi se programski okvir Spring Boot koji služi za kreiranje mikrousluga i implementaciju definiranih funkcionalnosti. Za komunikaciju između mikrousluga koristi se Spring Cloud tehnologija koja omogućava komunikaciju između mikrousluga. Izrađeni sustav s klijentom koji je implementiran u programskom okviru Angular ima mogućnost upravljanja korisnicima s ulogama pacijenta, liječnika i admina. Dalje, sustav daje mogućnost rukovanja lijekovima i naručivanja lijekova. Također, ima mogućnosti naručivanja pacijenta na pregled kod liječnika, pregled zdravstvenog kartona i izdavanje recepta od strane liječnika. U ovaj rad je uključena i mikrousluga koja odrađuje strojno učenje, tj. ulaskom u aplikaciju pacijent ima mogućnost otkrivanja boluje li od dijabetesa. Zadnja nabrojana funkcionalnost je napravljena u programskom okviru Flask i to potvrđuje raznovrsnost koju imaju mikrousluge te ističe činjenicu da mikrousluge ne ovise samo o jednoj tehnologiji ili programskom okviru. Kako bi se pokazala raznovrsnost korištene su i MySQL i PostgreSQL baza podataka što bi se u monolitnoj arhitekturi jako teško moglo implementirati tj. bilo bi kontraproduktivno.

U zadnjem dijelu rada testirale su se performanse mikrousluga na način da se opteretila jedna instanca mikrousluge s velikim brojem konkurentnih korisnika. Nakon što jedna instanca nije mogla rukovati velikim brojem korisnika definirana je druga instanca koja je riješila problem zagušenosti. Također, testiranja potvrđuju prednosti mikrousluga, a to su brzina odgovora i skalabilnost. Navedene značajke bi mogle promijeniti status “quo” mnogih tvrtki u području zdravstva i generirati tržište za manje tvrtke zadovoljavajući pritom potrebe zdravstvenog osoblja i pacijenata uz smanjene troškove održavanja. Mikrousluge su najprikladnije za velike primjene. Manje aplikacije obično su bolje s monolitnim arhitekturnim dizajnom. Iako je lakše razvijati i održavati neovisne mikrousluge, upravljanje mrežom zahtijeva dodatne napore te je to jedan od problema koji se treba riješiti u budućnosti. Kontejnerske platforme, DevOps prakse i računalstvo u oblaku mogu puno pomoći u usvajanju arhitekture mikrousluga.

Moguća unaprjeđenja sustava zdravstva od kuće se mogu tražiti u dodavanju novih mikrousluga kao što je usluga otkrivanja dijabetesa kako bi pacijenti preko sustava od kuće mogli ustanoviti boluju li od neke bolesti kao npr. prepoznavanje boluje li pacijent od upale pluća unesene slike s rendgena.

# LITERATURA


[1] Best practices for microservices in healthcare, MuleSoft, https://hosteddocs.ittoolbox.com/wp_Driving_healthcare_innovation_with_microservices.pdf, pristupljeno kolovoz 2021.

[2] The Five W's and an H of Microservices in Healthcare, Gabrielle Davis https://www.progress.com/blogs/the-five-ws-and-an-h-of-microservices-in-healthcare, pristupljeno kolovoz 2021.

[3] Microservices a definition of this new architectural term, Martin Fowler https://www.martinfowler.com/articles/microservices.html, pristupljeno kolovoz 2021.

[4] Microservice ecosystems for healthcare, Peter B. Nichol https://www.cio.com/article/3159071/microservice-ecosystems-for-healthcare.html, pristupljeno kolovoz 2021.

[5] Characteristics of a Microservice Architecture, Martin Fowler https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture

[6] From Monolithic Architecture to Microservices Architecture, Lorenzo De Lauretis Introduction, pristupljeno rujan 2021.

[7] Microservices on AWS https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html, pristupljeno rujan 2021.

[8] 5 Best Technologies To Build Microservices Architecture, Vinugayathri https://www.clariontech.com/blog/5-best-technologies-to-build-microservices-architecture, pristupljeno kolovoz 2021.

[9] Top Technologies and Languages to Pick for Building Microservices Architecture, Paresh Solanki, https://www.mindinventory.com/blog/technologies-for-mircroservices-architecture/

[10] What is a Health Information System, Chris Brook, https://digitalguardian.com/blog/what-health-information-system, pristupljeno kolovoz 2021.

[11] Benefits and drawbacks of electronic health record systems, Nir Menachemi and Taleah H Collum, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3270933/, pristupljeno kolovoz 2021.

[12] "What Is a Pharmacy Information System (PIS)? - Definition & Uses." Study.com, https://study.com/academy/lesson/what-is-a-pharmacy-information-systems-pis-definition-uses.html, pristupljeno kolovoz 2021.

[13] Effect of Computerized Physician Order Entry and a Team Intervention on Prevention of Serious Medication Errors, JAMA https://pubmed.ncbi.nlm.nih.gov/9794308/, pristupljeno kolovoz 2021.

[14] Creating Clinical Decision Support Systems for Respiratory Medicine, Carl G. Tams, Neil R. Euliano https://ieeexplore.ieee.org/document/7319596, pristupljeno kolovoz 2021.

[15] Terminology Standards, HIMSS https://www.himss.org/terminology-standards, pristupljeno kolovoz 2021.

[16] https://loinc.org/get-started/, pristupljeno kolovoz 2021.

[17] The Unified Code for Units of Measure, Gunther Schadow, Clement J. McDonald, https://ucum.org/trac, pristupljeno kolovoz 2021.

[18] Forty years of SNOMED: a literature review, Ronald Cornet & Nicolette de Keizer, https://bmcmedinformdecismak.biomedcentral.com/articles/10.1186/1472-6947-8-S1-S2, pristupljeno kolovoz 2021.

[19] National Drug Code Directory, fda.gov https://www.fda.gov/drugs/drug-approvals-and-databases/national-drug-code-directory, pristupljeno kolovoz 2021.

[20] Unified Medical Language System® (UMLS®) https://www.nlm.nih.gov/research/umls/rxnorm/index.html, pristupljeno kolovoz 2021.

[21] RedLex radiology lexicon, RSNA, https://www.rsna.org/practice-tools/data-tools-and-standards/radlex-radiology-lexicon pristupljeno kolovoz 2021.

[22] Unified Medical Language System® (UMLS®) https://www.nlm.nih.gov/research/umls/sourcereleasedocs/current/MEDCIN/index.html, pristupljeno kolovoz 2021.

[23] Current Procedural Terminology, Kim Pollock RN, MBA, CPC, https://www.sciencedirect.com/topics/medicine-and-dentistry/current-procedural-terminology, pristupljeno kolovoz 2021.

[24] Radiology Information System (RIS), Shaun Sutner https://searchhealthit.techtarget.com/definition/Radiology-Information-System-RIS,

pristupljeno kolovoz 2021.

[25] Laboratory Information Systems Project Management: A Guidebook for International Implementations, APHL, pristupljeno kolovoz 2021.

[26] Microservices Authentication and Authorization Solutions, Mina Ayoub, https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a, pristupljeno kolovoz 2021.

[27] Microservice architectures and open platforms for health care information systems, Pablo Pazos Gutierrez, https://www.cabolabs.com/blog/article/microservice_architectures_and_open_platforms_for_health_care_information_systems-5cab9ab60d88a.html, pristupljeno kolovoz 2021.

[28] The app solutions blog: Functional vs non-functional requirements https://theappsolutions.com/blog/development/functional-vs-non-functional-requirements/, pristupljeno rujan 2021.

[29] THE Java™ Programming Language, Fourth Edition, Ken Arnold, James Gosling, David Holmes, pristupljeno rujan 2021.

[30] Microservices Best Practices for Java, Michael Hofmann, Erin Schnabel, Katherine Stanley, pristupljeno rujan 2021.

[31] Spring Boot Overview, Pivotal, https://spring.io/projects/spring-boot, rujan 2021.

[32] Microservices Best Practices for Java, poglavlje 2.1.1., Michael Hofmann, Erin Schnabel, Katherine Stanley, pristupljeno rujan 2021.

[33] Architectural Styles and the Design of Network-based Software Architectures, poglavlje 5, Roy Thomas Fielding, doktorska disertacija

[34] ECMA-404 The JSON Data Interchange Standard, pristupljeno rujan 2021.

[35] Universal Resource identifiers in WWW, W3 https://www.w3.org/Addressing/URL/uri-spec.html, pristupljeno rujan 2021.

[36] Analysis of Web Traffic Based on HTTP Protocol, Jiajia Chen, Weiqing Cheng, https://ieeexplore.ieee.org/document/7772120, pristupljeno rujan 2021.

[37] Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616 Fielding, https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html, pristupljeno rujan 2021.

[38] Web on Reactive Stack, version 5.3.10. https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-client, pristupljeno rujan 2021.

[39] PostgreSQL, About
https://www.postgresql.org/about/, pristupljeno rujan 2021.
[40] MySQL, About
https://www.mysql.com/about/, pristupljeno rujan 2021.
[41] What is Angular?, Angular documentation
https://angular.io/guide/what-is-angular, pristupljeno rujan 2021.
[42] What is IntelliJ IDEA?, IntelliJ IDEA
https://www.jetbrains.com/idea/features/, pristupljeno rujan 2021.
[43] Performance Engineering for Microservices: Research Challenges and Directions
Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, Johannes Wettinger, pristupljeno rujan 2021.
[44] Microservices for Scalability, [Keynote Talk Abstract], Wilhelm Hasselbring
Software Engineering Group, Kiel University, D-24098 Kiel, Germany, prostupljeno rujan 2021.

# SAŽETAK

Sustav zdravstva od kuće bavi se olakšanjem komunikacije između liječnika i pacijenta te uspostavlja platformu koja se može jednostavno nadograditi novim funkcionalnostima. Arhitektura mikrousluga zaslužna je za jednostavnu nadogradnju sustava. U praktičnom dijelu rada izrađen je sustav zdravstva od kuće zasnovan na arhitekturi mikrousluga. Sustav je ostvaren korištenjem programskog okvira Spring Boot i tehnologije Spring Cloud te razvojnog okruženja IntelliJ IDEA. Za spremanje podataka su se koristile PostgreSQL i MySQL baze podataka. Na temelju predloženog teorijskog rješenja kroz rad je opisan razvoj sustava arhitekture mikrousluga i njegove funkcionalnosti. Analiziranjem rezultata potvrdile su se tvrdnje iz je arhitektura mikrousluga raznovrsna, brza i skalabilna.

**Ključne riječi:** brze performanse, komunikacija, mikrousluge, skalabilnost, Spring Boot, zdravstvo od kuće

## ABSTRACT

The home health care system deals with facilitating communication between doctors and patients and establishes a platform that can be easily upgraded with new functionalities. The microservice architecture is responsible for the simple upgrade of the system. In the practical part of the paper, a home health care system based on microservice architecture was developed. The system was realized using the Spring Boot software framework and Spring Cloud technology, as well as the IntelliJ IDEA development environment. PostgreSQL and MySQL databases are used to store data. Based on the proposed theoretical solution, the development system of microservice architecture and its functionality is described in the paper. By analyzing the results, the claims from the microservice architecture were confirmed, diverse, fast and scalable.

**Keywords**: communication, fast performances, home healthcare, microservices, scalability, Spring Boot

# ŽIVOTOPIS

Davor Aleksić rođen 24. svibnja 1994. godine u Wertheimu, Njemačka. Nakon završene srednje škole u Novoj Gradiški, 2013. godine ostvaruje upis na stručni studij Informatike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija, tadašnji Elektrotehnički fakultet u Osijeku i uspješno ga završava 2016. godine s titulom stručni prvostupnik. Nakon toga 2016. godine upisuje razlikovnu godinu na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek. 2017. upisuje sveučilišni diplomski studij, smjer Računarstvo, modul DRC - Programsko inženjerstvo na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek. Od 2019. godine radi u tvrtki Inovativni Trendovi d.o.o kao programer web aplikacija

Potpis:

_____

# PRILOZI

Prilog 1: Diplomski rad u .pdf formatu.

Prilog 2: Diplomski rad u .docx formatu

Prilog 3: Programski kod aplikacije i testova u .zip datoteci.