Automatiziranje administrativnih aktivnosti na primjeru mikroservisne web platforme za upravljanje ugostiteljskim jedinicama

Pavković, Nikola

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:712611

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-21**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Sveučilišni diplomski studij računarstva

Automatiziranje administrativnih aktivnosti na primjeru mikroservisne web platforme za upravljanje ugostiteljskim jedinicama

Diplomski rad

Nikola Pavković

Osijek, 2023. godina.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 20.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Nikola Pavković
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1235R, 08.10.2021.
OIB studenta:	41043811283
Mentor:	izv. prof. dr. sc. Josip Job
Sumentor:	,
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	prof. dr. sc. Marijan Herceg
Član Povjerenstva 1:	izv. prof. dr. sc. Josip Job
Član Povjerenstva 2:	doc. dr. sc. Hrvoje Leventić
Naslov diplomskog rada:	Automatiziranje administrativnih aktivnosti na primjeru mikroservisne web platforme za upravljanje ugostiteljskim jedinicama
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Prijava gostjiu u jedinice za noćenje višeslojan je proces koji uključuje naplatu noćenja, skeniranje identifikacijskih dokumenata, izradu predračuna i naknadni unos podataka o gostima u e-turist sustav. Navedeni proces oduzima veliku količinu vremena ugostiteljima i sklon je greškama uzrokovanim ljudskim faktorom prilikom unošenja velike količine podataka u više različitih sustava. Cilj izrade ovog diplomskog rada jest opisati proces dizajna jedinstvene platforme za automatiziranje administrativnih aktivnosti te oprimjeriti navedeni proces izradom platforme za upravljanje ugostiteljskim jedinicama. Tema rezervirana za: Nikola Pavković
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 2 razina
Datum prijedloga ocjene od strane mentora:	20.09.2023.

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

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

IZJAVA O ORIGINALNOSTI RADA

Osijek, 04.10.2023.

Ime i prezime studenta:	Nikola Pavković
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1235R, 08.10.2021.
Turnitin podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Automatiziranje administrativnih aktivnosti na primjeru mikroservisne web platforme za upravljanje ugostiteljskim jedinicama**

izrađen pod vodstvom mentora izv. prof. dr. sc. Josip Job

i sumentora ,

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

Potpis studenta:

SADRŽAJ

1. UVOD 1

2. PREGLED PODRUČJA RADA I ANALIZA POSTOJEĆIH SUSTAVA ZA PRIJAVU GOSTIJU 2

2.1. Analiza tržišta i postojeća rješenja za automatizaciju prijave gostiju 2

2.2. E-visitor sustav za administraciju i prijavu gostiju 3

2.3. Ograničenja *e-visitor* sustava i analiza korisničkog sučelja 4

3. KORIŠTENE TEHNOLOGIJE I ARHITEKTURE PRILIKOM RAZVOJA PROGRAMSKOG RJEŠENJA 7

3.1. Tehnički zahtjevi programskog rješenja 7

3.2. Pregled dostupnih tehnologija za razvoj višeplatformskih aplikacija 7

3.2.1. Pregled dostupnih tehnologija za razvoj pozadinskog dijela programskog rješenja 7

3.2.2. Pregled dostupnih tehnologija za razvoj korisničkog sučelja programskog rješenja 9

3.2.3. Programske arhitekture za razvoj pozadinskog dijela aplikacije 9

3.3. Kontejnerizacija elemenata sustava u mikroservisnoj arhitekturi 11

3.3.1. Karakteristike kontejnerskih aplikacija i orkestracija rada kontejnera 12

4. Razvoj *Properbooker* aplikacije i analiza performansi razvijenog sustava 13

4.1. Opis korištenih tehnologija za razvoj *Properbooker* aplikacije 13

4.1.1. *Spring Boot* radni okvir 13

4.1.2. *React Native* i programski jezik *Typescript* 15

4.2. Programska arhitektura *Properbooker* aplikacije 16

4.3. Razvoj pojedinih usluga unutar *Properbooker* mikroservisne arhitekture 18

4.3.1. Način rada usluge za očitavanje *MRZ* koda osobnih dokumenata 18

4.3.2. Način rada usluge za generiranje *PDF* predračuna 20

4.3.3. Usluga za *JWT* token autentikaciju korisnika i autorizacija *HTTP* zahtjeva 23

4.3.4. Usluga za upravljanje ugostiteljskim jedinicama 25

4.3.5. *Netflix Eureka* usluga za pronalaženje aktivnih instanci mikroservisa 26

4.3.6. *API Gateway* usluga, raspodjela računalnih resursa i preusmjeravanje *API* poziva 28

4.3.7. Način rada usluge za emulaciju rada *e-visitor* sustava 29

4.4. Klijentska *React Native* aplikacija za automatsku prijavu gostiju i generiranje predračuna 30
4.4.1. Izgled aplikacije i iskustvo korištenja 30
4.5. Implementacija dizajna i značajki *React Native* aplikacije 36
4.5.1. Projektna struktura klijentske *React Native* aplikacije 36
4.5.2. Autentikacija i autorizacija korisnika unutar aplikacije 37
4.5.3. Implementacija glavnog zaslona aplikacije i prikaz podataka o jedinicama za noćenje 39
4.5.4. Implementacija zaslona za prijavu gostiju 40
4.5.5. Implementacija zaslona za generiranje *PDF* predračuna 41
4.6. Analiza performansi implementiranog sustava 42
4.6.1. Metodologija testiranja performansi sustava i *Apache JMeter* alat za automatizaciju testiranja 42
4.6.2. Rezultati testiranja i analiza vrijednosti parametara 44
4.6.3. Zaključak analize rada sustava 46
4.7. Ograničenja i moguća proširenja razvijenog programskog rješenja 46
ZAKLJUČAK 48
SAŽETAK 49
ABSTRACT 50
LITERATURA 51
ŽIVOTOPIS 52
PRILOZI 53

1. UVOD

Tijekom prethodnog desetljeća Republika Hrvatska postala je jedno od najposjećenijih turističkih odredišta na Mediteranu. Razvojem turističke infrastrukture i povećanjem broja gostiju drastično se povećao broj malih i srednjih iznajmljivača čime je nastala potreba za rješenjem za praćenje broja dolazaka, administraciju turističkih jedinica i prikupljanje podataka o gostima iz zakonskih, marketinških i strateških razloga. Kao odgovor na navedenu potrebu, Hrvatska turistička zajednica, u suradnji s vanjskom informatičkom tvrtkom, 2015. godine razvija sustav za obradu i analizu podataka te izvještavanje u statističke svrhe – *e-visitor* [1]. Nastavno tomu, iznajmljivači su dužni prijavljivati i odjavljivati svako noćenje putem *e-visitor* sustava. *E-visitor* sustav pokazao se kao izvrsno rješenje za potrebe administracije i praćenja gostiju, no iz perspektive korisničkog iskustva postoji određen prostor za poboljšanje.

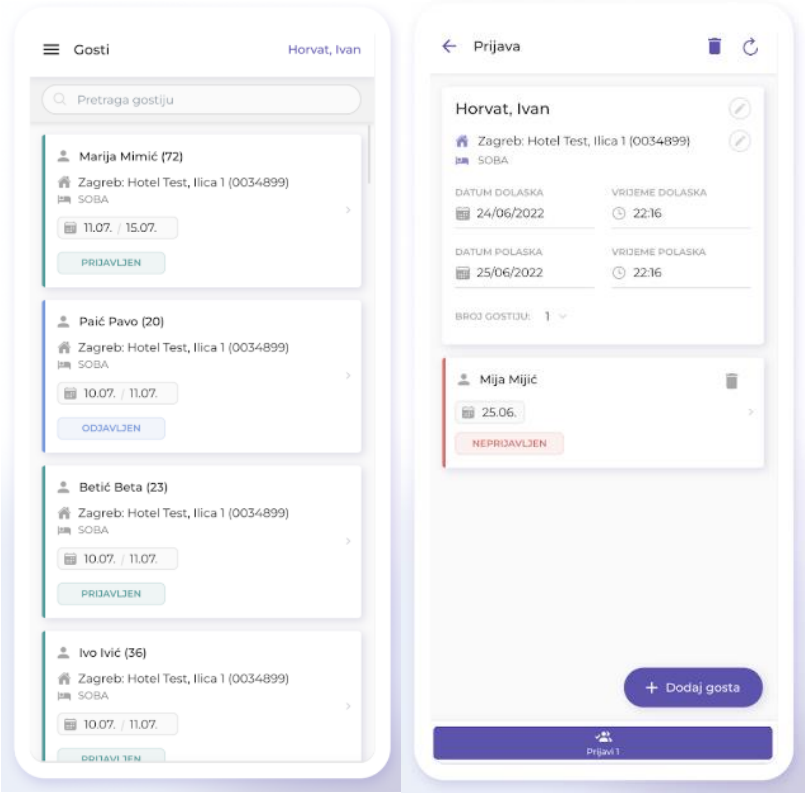
Smještaj gostiju u jedinice za noćenje u Republici Hrvatskoj višeslojan je proces koji uključuje naplatu noćenja, skeniranje identifikacijskih dokumenata, izradu predračuna i naknadni ručni unos podataka o gostima u državni *e-visitor* sustav za prijavu i praćenje posjetitelja. Navedeni proces oduzima veliku količinu vremena ugostiteljima i sklon je greškama uzrokovanim ljudskim faktorom koje nastaju prilikom unošenja velike količine podataka u više različitih informatičkih sustava. Cilj izrade ovog diplomskog rada jest opisati proces dizajna jedinstvenog sustava za automatizaciju poslovne administracije te oprimjeriti navedeni proces izradom platforme za upravljanje noćenjima unutar ugostiteljskih jedinica. U sklopu ovog diplomskog rada opisat će se proces prepoznavanja problema s postojećim sustavom za prijavu gostiju i administraciju jedinica za noćenje. Također, objasnit će se mikroservisna arhitektura web aplikacija i tehnologije za izradu korisničkog sučelja aplikacije. Objašnjene tehnologije, metodologije i arhitekture primijenit će se prilikom razvoja mikroservisne web aplikacije koja, korištenjem platformi za računalni vid i strojno učenje te sustava za analizu standardiziranog identifikacijskog koda osobnih dokumenata, poboljšava, ubrzava i pojednostavljuje prijavu gostiju u *e-visitor* sustav putem aplikacijsko-programskog sučelja. Također, pružit će se analiza programskoga rješenja iz perspektive skalabilnosti, proširivosti koda i održavanja mikroservisne arhitekture.

2. PREGLED PODRUČJA RADA I ANALIZA POSTOJEĆIH SUSTAVA ZA PRIJAVU GOSTIJU

U ovom poglavlju opisat će se postojeća rješenja za automatsku prijavu gostiju, moguća poboljšanja postojećih rješenja i detaljan opis državnog *e-visitor* sustava za prijavu gostiju u jedinice za noćenje.

2.1. Analiza tržišta i postojeća rješenja za automatizaciju prijave gostiju

Unatoč tomu što već postoje aplikacije sa sličnom namjenom poput aplikacije koja će se razvijati u sklopu ovog diplomskog rada, iskustvo korištenja postojećih aplikacija često je manjkavo iz perspektive preglednosti, potrebne količine korisničke interakcije za korištenje implementiranih značajki ili podržanosti različitih platformi. Na hrvatskom tržištu trenutno postoje tri aplikacije koje nude slične značajke poput one koja će se razvijati u sklopu ovog rada od kojih se najviše ističe aplikacija *Prijavi turiste* [2] razvijena za *Android* i *iOS* operacijske sustave. Ova je aplikacija prvo programsko rješenje ovakve vrste i nudi mnoštvo mogućnosti uz naplatu tokena za prijavu gostiju, no ostavlja prostor za napredak u pogledu iskustva korištenja i brzine pristupa značajkama aplikacije. Osim ove aplikacije, u proteklih se dvije godine na tržištu pojavljuju aplikacije *mVisitor* [3] i *eCheckin* [4] od kojih *eCheckin* donosi velik razvoj u pogledu iskustva korištenja i dizajna grafičkog sučelja što se može vidjeti na slici Slika 2.1. Porast broja aplikacija ovakve namjene dobar je pokazatelj razvitka sektora i postojanja potrebe na tržištu za opisanim rješenjima. Razvojem modularne aplikacije temeljene na mikroservisnoj arhitekturi omogućit će se dinamičko dodavanje značajki i praćenje razvoja konkurentnih aplikacija.



Slika 2.1 Prikaz korisničkog sučelja *eCheckin* aplikacije

2.2. E-visitor sustav za administraciju i prijavu gostiju

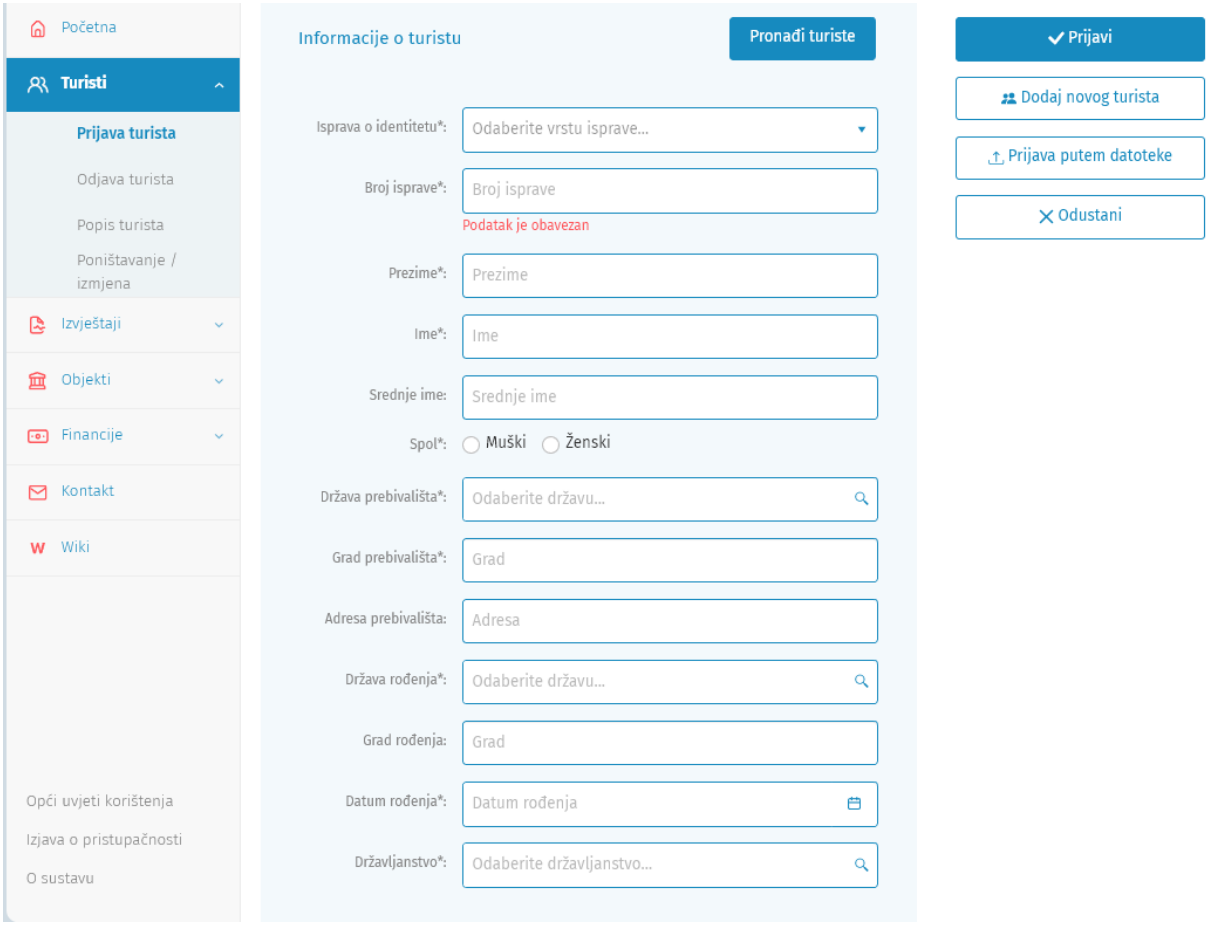
Kao što je spomenuto u uvodu, *e-visitor* predstavlja programsko rješenje za prijavu i odjavu gostiju u ugostiteljskim jedinicama. Sustav nastaje 2015. godine sa svrhom prikupljanja statističkih podataka o posjetiteljima zbog novonastale potrebe za preciznijim praćenjem broja gostiju u zemlji. Nakon puštanja u rad 2015. godine, sustav obavezno moraju koristiti svi iznajmljivači jedinica za noćenje u Republici Hrvatskoj. *E-visitor* sustav svakodnevno koristi oko 100 000 iznajmljivača iz čega se može zaključiti da postoji veliko opterećenje na sustav zbog čega se redovito održava i ažurira. Kako bi goste prijavili u *e-visitor* sustav, ugostitelji moraju otvoriti web aplikaciju posluženu na domeni *www.evisitor.hr*, unijeti korisničke podatke i *tan* kod te u korisničkom sučelju odabrati opciju *Prijava gostiju*. Prijava gostiju zahtijeva pristup podacima o imenu, prezimenu, datumu rođenja, broju identifikacijskog dokumenta, mjestu prebivališta, kontaktnim podacima, datumu dolaska i odlaska, vrsti identifikacijskog dokumenta, spolu i drugim. Navedeni podaci unose se ručno uz priloženi osobni identifikacijski dokument posjetitelja. Osim prijave gostiju, *e-visitor* sustav omogućuje administraciju ugostiteljskih jedinica i praćenje

iznosa boravišne pristojbe. Važno je napomenuti kako se svim podacima unesenim u *e-visitor* sustav prilikom prijave posjetitelja provjerava valjanost iz zakonskih razloga čime se onemogućuje maliciozno korištenje sustava. Također, podaci uneseni u sustav dostupni su nadležnim državnim institucijama poput Ministarstva unutrašnjih poslova Republike Hrvatske ponajviše zbog slučajeva kršenja imigracijske politike.

Zbog određenih ograničenja sustava, postoje i druga programska rješenja, opisana u prethodnom poglavlju, koja se koriste *e-visitor* aplikacijsko-programskim sučeljem koje omogućuje pristup svim funkcijama sustava putem *REST* zahtjeva uz odgovarajući ključ i korisničke podatke. Također, osim produkcijske inačice sustava, postoji i testno aplikacijsko-programsko sučelje pomoću kojega se olakšava i ubrzava razvoj novih programskih rješenja bez naplate korištenja produkcijskog sustava. Pristup navedenim sustavima ostvaruje se direktnim zahtjevom *e-visitor* podršci nakon čega se na temelju identifikacijskih podataka privatne osobe ili razvojne tvrtke generiraju testni i produkcijski *API* ključevi i pristupni podaci. *E-visitor API* detaljno je dokumentiran zajedno s mogućim problemima prilikom korištenja sustava na službenim web mjestima Hrvatske turističke zajednice. Zbog velike potražnje, ponekad je nemoguće pristupiti testnoj okolini zbog čega se prilikom razvoja ove aplikacije razvija i aplikacijsko-programsko testno sučelje koje emulira rad *e-visitor* sustava.

2.3. Ograničenja *e-visitor* sustava i analiza korisničkog sučelja

Unatoč uspješnom višegodišnjem radu *e-visitor* sustava i posluživanju više desetaka tisuća ugostitelja na dnevnoj bazi, postoje određena ograničenja sustava koja se ponajviše odnose na korisničko sučelje i iskustvo prilikom korištenja programskog rješenja. Kao što je spomenuto u prethodnom poglavlju, korištenje *e-visitor* sustava započinje prijavom administratora u sustav putem pristupnih podataka nakon čega se korisniku prikazuje početna stranica *e-visitor* web aplikacije. U sklopu ovog rada analizirat će se dio korisničkog sučelja koji se odnosi na prijavu posjetitelja u ugostiteljske jedinice što predstavlja najčešći scenarij korištenja sustava. Pritiskom na tipku *Prijava turista* otvara se mnoštvo polja za unos podataka o turistima navedenim u prethodnom poglavlju. Na slici Slika 2.2 prikazano je sučelje *e-visitor* sustava za prijavu gostiju.



Slika 2.2 Sučelje *e-visitor* sustava za prijavu gostiju

Sveukupno, od korisnika se traži ručni unos podataka u 21 podatkovno polje što iziskuje veliku količinu vremena i učestale naknadne ispravke pogrešnih unosa. Također, kao rezultat dugotrajnog procesa prijave gostiju, ugostitelji često fotografiraju osobne dokumente posjetitelja i naknadno unose potrebne podatke u sustav. Fotografiranje osobnih dokumenata često uzrokuje sumnju u legitimitet ugostiteljskog objekta kod posjetitelja što negativno utječe na percepciju kvalitete noćenja. Osim što uzrokuje negativnu percepciju usluge, fotografiranje osobnih dokumenata može se koristiti za krivotvorenje identifikacijskih dokumenata u sve češćim slučajevima malicioznog i nezakonitog djelovanja pojedinaca i poduzeća. Kod ugostiteljskih objekata koji koriste mrežne platforme za oglašavanje, fotografiranje osobnih dokumenata često negativno utječe na krajnju ocjenu objekta koju posjetitelj definira nakon boravka. Niske ocjene ugostiteljskog objekta utječu na konkurentnost objekta na određenoj platformi i nerijetko rezultiraju smanjenjem prihoda i broja prijavljenih gostiju. Kao alternativa fotografiranju dokumenata i naknadnom unosu podataka, podaci se mogu unositi tijekom primitka posjetitelja. Ovakav pristup smanjuje sumnju u legitimitet objekta kod posjetitelja, no zahtijeva veliku količinu vremena s naglaskom na veće skupine

posjetitelja što također rezultira niskim ocjenama i dojmom neprofesionalnosti u poslovanju. Posljedično navedenim problemima, postoji potražnja za programskim rješenjem koje se može koristiti na mobilnim uređajima i ubrzati postupak prijave gostiju. U sklopu ovog rada opisat će se intuitivno i jednostavno rješenje za prijavu gostiju tijekom primitka bez potrebe za ručnim unosom velike količine složenih podataka. Opisano programsko rješenje koristit će tehnologije računalnog vida i strojnog učenja za očitanje *MRZ* koda (eng. *Machine readable zone*) [5] koji, uz pravilno dekodiranje, pruža većinu informacija potrebnih za prijavu gostiju u *e-visitor* sustav putem aplikacijsko-programskog sučelja. Programsko rješenje pružit će intuitivno korisničko iskustvo i spriječiti potrebu za pohranom fotografija identifikacijskih dokumenata na osobnim uređajima ugostitelja.

3. KORIŠTENE TEHNOLOGIJE I ARHITEKTURE PRILIKOM RAZVOJA PROGRAMSKOG RJEŠENJA

3.1. Tehnički zahtjevi programskog rješenja

Aplikacija koja će se razvijati u sklopu ovog diplomskog rada, *Properbooker*, podrazumijeva i određene tehničke zahtjeve s obzirom na primjenu. Kao što je spomenuto u prethodnim poglavljima, aplikacija mora biti dostupna za korištenje na mobilnim uređajima, mora se moći jednostavno skalirati u slučaju naglog porasta broja korisnika, mora pružiti intuitivno korisničko iskustvo koje se vremenom može prilagođavati potrebama budućih korisnika na brz i učinkovit način te ju je potrebno moći koristiti na više različitih platformi kao što su *Android* i *IOS*. Nastavno na navedene zahtjeve, u tekstu niže analizirat će se trenutno dostupne tehnologije i argumentirati korištenje odabranih tehnologija za razvoj programskog rješenja.

3.2. Pregled dostupnih tehnologija za razvoj višeplatformskih aplikacija

3.2.1. Pregled dostupnih tehnologija za razvoj pozadinskog dijela programskog rješenja

Dizajn modernih skalabilnih web aplikacija iznimno je slojevit proces koji podrazumijeva temeljito poznavanje *frontend* i *backend* tehnologija te odabir prikladnih programskih arhitektura koje mogu podnijeti velik broj zahtjeva na razini sekunde ili milisekunde. Zbog spomenutih opterećenja sustava, odabir tehnologija za razvoj programskih rješenja ovisi o brojnim parametrima poput stabilnosti, ažurnosti, popularnosti, količine dokumentacije, cijeni i međusobnoj povezanosti odabranih tehnologija s drugim korištenim tehnologijama. Na današnjem tržištu postoji izrazito velik broj konkurentnih tehnologija, no razvijači programskih rješenja najčešće posežu za tehnologijama koje su dugotrajne i testirane u praksi zbog ažurnosti i sigurnosti istih.

S obzirom na činjenicu da predviđeno programsko rješenje koje će se razvijati u sklopu izrade ovog diplomskog rada zahtijeva mogućnost rada na više različitih platformi, skalabilnost, stabilnost u radu i obradu podataka na strani poslužitelja, opseg tehnologija postaje manji i samim time izbor postaje uži i jednostavniji. Za obradu zahtjeva na poslužiteljskoj strani ističu se tri tehnologije i tri programska jezika koji predstavljaju industrijske standarde za navedene zahtjeve. Spomenute tehnologije i jezici su *Node.js* i programski jezik *Javascript, ASP.NET* radni okvir i jezik *C#* te *Spring Boot* radni okvir i programski jezik *Java*. *Node.js* i *Javascript* predstavljaju dobro rješenje za razvoj pozadinskog rada aplikacije, no s obzirom na kratko vrijeme postojanja, postoji manje dokumentacije te manjak robusnosti i značajki unutar ekosustava. Unatoč određenim nedostacima, navedena kombinacija tehnologija često se koristi u slučaju kada se za izradu korisničkog sučelja i poslužiteljskog dijela aplikacije želi koristiti isti programski jezik s ciljem smanjenja složenosti razvoja. S druge strane, tehnologije i radni okviri temeljeni na *Javascript* programskom jeziku prethodnih godina doživljavaju ubrzan razvoj što rezultira povećanjem broja značajki i alata no, s obzirom na kratko vrijeme postojanja, takvi alati često nisu temeljito testirani u praksi zbog čega ih se izbjegava koristiti u sustavima s velikim brojem složenih zahtjeva. *ASP.NET* radni okvir i *C#* već neko vrijeme pokazuju primjetan razvoj te povećanje broja značajki i performansi zbog čega se sve češće koriste u velikim i robusnim programskim rješenjima [6] Iako *ASP.NET* i *C#* primjećuju sve veći porast broja korisnika, vrlo kratko postoje u domeni otvorenog koda što je rezultiralo malim brojem tehnološki zrelih i testiranih značajki, radnih okvira i alata. *Spring Boot* i *Java* programski jezik jedna su od najzrelijih kombinacija za razvoj robusnih, skalabilnih i dugotrajnih web rješenja [7]. S više od 20 godina aktivnosti u domeni otvorenog koda, broj značajki, alata i radnih okvira koji su više godina testirani u praksi, redovito ažurirani i detaljno dokumentirani, iznimno je velik što pogoduje razvoju ozbiljnih i stabilnih programskih rješenja [8]. Također, bitno je istaknuti kompatibilnost *Java* programskog jezika s njegovim prethodnim inačicama, što često nije slučaj s novijim tehnologijama. Unatoč ponekad slabijim performansama u usporedbi s novijim tehnologijama, spomenuta kombinacija postala je industrijski standard zbog uspješnog razvoja vodećih industrijskih programskih rješenja u periodu od više od dva desetljeća. Zbog navedenih karakteristika ovog radnog okvira, *Spring Boot*, zajedno s *Java* programskim jezikom, koristit će se za izradu pozadinskog, mikroservisnog, dijela *Properbooker* aplikacije.

3.2.2. Pregled dostupnih tehnologija za razvoj korisničkog sučelja programskog rješenja

S obzirom na definirane zahtjeve, tehnologije za razvoj korisničkog sučelja spomenutog programskog rješenja mogu se ograničiti na višeplatformske radne okvire. Višeplatformski radni okviri predstavljaju novije tehnologije koje najčešće omogućuju pisanje jedinstvenog izvornog koda izvršivog na više od jedne platforme, poput *Android, IOS* i mrežnih platformi. Takve tehnologije često posebno prevode izvorni kod za specifične platforme, transformiraju elemente korisničkog sučelja u uobičajene elemente platformi ili omogućuju pisanje posebnog sloja arhitekture koji osigurava interoperabilnost. U domeni višeplatformskih radnih okvira koji odgovaraju navedenim zahtjevima ponajviše se ističu *React Native* i *Flutter* tehnologije. *React Native* je višeplatformski radni okvir otvorenog koda koji se temelji na *ReactJS* biblioteci koju je razvio *Facebook* te *Javascript* programskom jeziku [9]. *React Native* nastaje 2015. godine s ciljem pružanja mogućnosti razvoja programskih rješenja za *Android, IOS* i mrežne platforme koristeći pristup utemeljen na komponentama, što je karakteristično za popularnu *ReactJS* biblioteku. Upravo zbog popularnosti ovog radnog okvira, razvijene su i brojne popratne biblioteke koje olakšavaju i ubrzavaju razvoj programskih rješenja, a zbog sličnosti s *ReactJS* bibliotekom, prijenos znanja s mrežnih radnih okvira prirodan je uz blagu krivulju učenja. Drugi istaknuti radni okvir za višeplatformski razvoj čine *Flutter* i programski jezik *Dart* [10]. Navedeni radni okvir radi na sličan način kao *React Native* i također nudi visoke performanse zbog čega predstavlja izvrsno rješenje za višeplatformski razvoj. Odabir između spomenutih radnih okvira najviše ovisi o prethodnom radnom iskustvu zbog čega će razvijači s iskustvom u razvoju web aplikacija najčešće posegnuti za *React Native* radnim okvirom dok će razvijači s iskustvom izrade mobilnih aplikacija češće odabrati *Flutter* i *Dart*. Osim navedenih radnih okvira postoje i druge tehnologije kao što su *Xamarin, Ionic* i *Kotlin Multiplatform mobile* koje nude slične mogućnosti, no s nedostatkom razvojnih alata i proširenja. Za izradu *Properbooker* aplikacije, koristit će se *React Native* radni okvir zbog prethodno spomenutih karakteristika.

3.2.3. Programske arhitekture za razvoj pozadinskog dijela aplikacije

Programska arhitektura predstavlja organizaciju, raspored i međudjelovanje određenih programskih cjelina unutar neke aplikacije ili rješenja. Različitom organizacijom i grupiranjem

programskih cjelina mogu se postići određene karakteristike programskog rješenja zbog čega odabir arhitekture ponajviše ovisi o specifičnoj primjeni rješenja, ali i o ograničenjima korištenih tehnologija, brojnosti stručnog osoblja i količini financijskih sredstava. Programske arhitekture, ovisno o željenim karakteristikama, najčešće se dijele na mikroservisne [11] i monolitne arhitekture. Monolitna arhitektura predstavlja uobičajeni pristup razvoju softvera prilikom čega se sav izvorni kod programskog rješenja nalazi unutar jedne izvršive aplikacije. Ovakav pristup, uz pravilnu organizaciju izvornog koda, podrazumijeva ubrzano podizanje rješenja na željene platforme i smanjenje mrežnih problema, no s druge strane uvodi ograničenja vezana uz skaliranje, preglednost i ažuriranje programskih rješenja. Također, bitno je spomenuti kako monolitne arhitekture često imaju iznimno dobre performanse jer se izvorni kod izvršava na jednoj hardverskoj konfiguraciji zbog čega je jednostavno prilagoditi programsko rješenje specifičnoj konfiguraciji i operacijskom sustavu. Glavni problem monolitnih arhitektura predstavljaju nefleksibilnost, složenost, usporen razvoj i visoka međupovezanost programskih komponenti unutar rješenja koja rezultira visokom složenošću promjena pojedinih dijelova sustava. Kao odgovor na spomenute probleme, početkom 2000-ih godina pojavljuje se koncept mikroservisne arhitekture. Ovakva arhitektura podrazumijeva razdvajanje programskih cjelina u manje samostalne izvršne jedinice, točnije, mikroservise koji se međusobno povezuju odabranim mrežnim tehnologijama. Prednosti ovog pristupa su mogućnost izmjene pojedinih programskih cjelina bez potrebe za izmjenom ostatka sustava, mogućnost podizanja i ažuriranja samo željenih dijelova sustava, jednostavno skaliranje povećanjem broja instanci mikroservisa te agilnost i brzina razvoja koju pruža raspodijeljena programska struktura. Unatoč brojnim prednostima koje karakteriziraju mikroservisnu arhitekturu, postoji i određen broj nedostataka u usporedbi s monolitnom arhitekturom [12]. Nedostaci mikroservisa najviše se očituju u pogledu performansi, mrežnih problema i složenosti prilikom orkestracije rada povezanih mikroservisa.

S obzirom na činjenicu da mikroservisi za obradu jednog korisničkog zahtjeva uz njega često moraju izvršiti i određen broj unutarnjih *API* poziva između povezanih mikroservisa, postoji veća latencija prilikom obrade zahtjeva u usporedbi s monolitnim arhitekturama koje moraju obraditi isključivo jedan *API* poziv. Također, mikroservisna programska rješenja najčešće zahtijevaju više računalnih resursa i složeniju mrežnu infrastrukturu koja se najčešće očituje u obliku tehnologije računalnog oblaka. Zbog navedenih prednosti i nedostataka opisanih arhitektura, odabir prikladne arhitekture ovisi o brojnim faktorima te funkcionalnim i nefunkcionalnim zahtjevima programskog rješenja. Na tržištu koje se redovito mijenja i vremenom postaje sve konkurentnije, sposobnost prilagodbe programskih rješenja i agilnog dodavanja novih značajki, od velike je

poslovne važnosti tvrtkama koje ih razvijaju. Osim toga, jednostavnost i brzina održavanja rješenja te skaliranje sustava s ciljem sposobnosti obrade velikog broja zahtjeva, također predstavljaju bitan faktor prilikom odabira programske arhitekture. Upravo zbog navedenih zahtjeva većina će proizvođača softvera odabrati mikroservisnu arhitekturu zbog pritiska tržišta, jednostavne organizacije timova prema specifičnim uslugama koje se razvijaju i brzine razvoja. S druge strane, razvijači sustava koji nužno moraju brzo i stabilno funkcionirati u bilo kojem trenutku, najčešće će odabrati monolitnu arhitekturu koja, zbog manjeg broja mogućih mrežnih problema i jednostavnije prilagodbe računalnim konfiguracijama i operacijskom sustavu, smanjuje broj programskih grešaka i povećava performanse sustava. Bitno je spomenuti kako je razvoj monolitnih programskih rješenja puno jeftiniji i zahtijeva manje računalnih resursa što također može predstavljati bitan faktor prilikom odabira arhitekture. S obzirom da broj korisnika *Properbooker* aplikacije može ubrzano narasti u svakom trenutku, za njen će se razvoj koristiti mikroservisna arhitektura. Ovakav će pristup omogućiti jednostavno skaliranje i ažuriranje što je ključno prilikom razvoja aplikacije ovakve namjene.

3.3. Kontejnerizacija elemenata sustava u mikroservisnoj arhitekturi

Kao što je prethodno spomenuto, mikroservisne aplikacije se najčešće podižu na mrežne platforme temeljene na tehnologiji računalnog oblaka. S obzirom da postoji velik broj poslužitelja računalnih oblaka poput *Google Cloud*, *Amazon AWS* i *Microsoft Azure* platformi koje su često utemeljene na različitim operacijskim sustavima i tehnologijama, postoji potreba za tehnološki agnostičkom tehnologijom koja će omogućiti izvršavanje programskih rješenja neovisno o platformi na kojoj se nalaze. Rješenje ovog problema pronalazi se u vidu kontejnerizacije pojedinih usluga unutar sustava. Kontejnerizacija mikroservisa i aplikacija je proces izoliranja određenog samostalnog elementa aplikacije ili cijele aplikacije u posebnu izvršivu jedinicu zvanu kontejner. Kontejneri predstavljaju jedinice slične virtualnim strojevima, no za razliku od njih koriste *kernel* operativnog sustava na kojemu se izvršavaju. Unatoč ovisnosti kontejnera o *kernelu* operativnog sustava, postoje brojne tehnologije koje omogućuju izvršavanje kontejnera temeljenih na *Linux* operativnom sustavu na *Windows* i *MacOS* uređajima poput *WSL*, *HyperV* i *Hypervisor* virtualizacijskih tehnologija. Zbog navedenih alata moguće je izraditi kontejner na *Linux* operativnom sustavu i pokrenuti ga na bilo kojoj platformi što kontejnere čini tehnološki agnostičkom metodom dizajna arhitekture programskog proizvoda [13].

3.3.1. Karakteristike kontejnerskih aplikacija i orkestracija rada kontejnera

Za razliku od uobičajenih metoda posluživanja aplikacija, arhitekture temeljene na kontejnerizaciji pružaju niz prednosti iz perspektive sigurnosti, skalabilnosti i robusnosti koje su poželjne prilikom razvoja velikih programskih rješenja. Izolacija pojedinih dijelova aplikacije u kontejnere onemogućuje propagaciju malicioznog koda kroz cijelu aplikaciju te izolira problem na isključivo jedan kontejner kojega je jednostavno zamijeniti ažuriranom inačicom. Također kontejnerizacija omogućuje jednostavno ažuriranje koda bez utjecaja na korisničko iskustvo na način da se aktiviraju identični kontejneri prethodne inačice kontejnera zajedno s ažuriranom inačicom kontejnera. Prilikom ovakvog načina aktiviranja novog kontejnera, dio korisnika će koristiti staru inačicu programskog rješenja koju će postepeno zamjenjivati nova. Ovakav način puštanja aplikacija u rad naziva se *Canary deploy* [14]. Osim jednostavnog ažuriranja i poboljšanja iz perspektive sigurnosti, kontejnerizacija pruža i poboljšanje pouzdanosti programskog rješenja. U slučaju nepravilnog rada ili pada određene instance kontejnera, on se vrlo brzo može manualno ili automatski zamijeniti novom instancom, a uz redundantne instance baza podataka, korisnički podaci uvijek ostaju pohranjeni. S obzirom na činjenicu da manualna zamjena instanci kontejnera sa svrhom održavanja rada sustava ili poboljšanja njegovih performansi zahtijeva veliku količinu vremena i podložna je ljudskoj grešci, zamjena se najčešće radi koristeći posebne alate namijenjene orkestraciji rada većeg broja kontejnera. Alati za automatsku orkestraciju najčešće se koriste u sustavima koje koristi velik broj korisnika zbog čega je potrebno kontinuirano raspoređivati resurse skupinama kontejnera ovisno o njihovu broju. Također, navedeni alati omogućuju praćenje zdravlja instanci kontejnera, automatsko skaliranje, jednostavno ažuriranje i proširivanje programskih rješenja te zamjenu kontejnera kada je to potrebno [15]. Na tržištu trenutno postoji više alata za automatsku orkestraciju kontejnerskih aplikacija od kojih se ističu *RedHat Openshift, Kubernetes* i *Docker Swarm*. Navedeni alati su dobro dokumentirani i testirani u praksi i kritičnim uvjetima zbog čega predstavljaju izvrsno rješenje za automatsku orkestraciju. S obzirom na manji broj kontejnera koji čine aplikaciju koja se razvija u sklopu ovog diplomskog rada, koristit će se uobičajeni manualni način orkestracije rada kontejnera. Također, za kontejnerizaciju pojedinih komponenti aplikacije koristit će se *Docker* i *Docker compose* radni okviri.

4. Razvoj *Properbooker* aplikacije i analiza performansi razvijenog sustava

Nastavno na detaljnu analizu zahtjeva aplikacije i argumentaciju tehnologija korištenih za njen razvoj, u poglavljima u nastavku opisat će se pojedini dijelovi aplikacije, njihova međuovisnost, koraci razvoja i funkcija.

4.1. Opis korištenih tehnologija za razvoj *Properbooker* aplikacije

Kao što je spomenuto u prethodnom poglavlju, za razvoj pozadinskog dijela *Properbooker* aplikacije bit će korišteni *Java* programski jezik i *Spring Boot* radni okvir, dok će se za razvoj grafičkog sučelja i klijentske logike koristiti *React Native* zajedno s *Javascript* i *Typescript* programskim jezicima. U nastavku će se opisati pojedine tehnologije i njihova uloga prilikom razvoja programskog rješenja.

4.1.1. *Spring Boot* radni okvir

Spring Boot predstavlja radni okvir otvorenog koda koji je nastao kao proširenje *Spring MVC* platforme s ciljem ubrzavanja uobičajenog načina razvoja mikroservisnih aplikacija. Za razliku od *Spring MVC* platforme, *Spring Boot* smanjuje broj korisnički definiranih konfiguracijskih datoteka i automatizira razvojne procese poput integracije baze podataka, upravljanja programskim ovisnostima, programske zaštite aplikacije i testiranja. Također, *Spring Boot* dolazi s ugrađenim *Apache Tomcat* poslužiteljem za posluživanje aplikacije. Uobičajena *Spring Boot* mikroservisna aplikacija podijeljena je u *Service, Controller, Model* i *Repository* slojeve od kojih svaki implementira specifične funkcije unutar mikroservisa. *Repository* sloj mikroservisa upravlja implementacijom metoda za manipulaciju bazama podataka. Ovisno o tipu baze podataka, *JPA Repository* sučelje, koje većina implementacija ovog sloja koristi, pruža već ugrađene standardne metode za upravljanje bazom podataka i smanjuje vrijeme integracije istih prilikom razvoja aplikacije. *Model* sloj definira tranzitivne objekte koji će se koristiti prilikom izvršavanja *HTTP* zahtjeva prema mikroservisu, dok *Service* sloj implementira svu logiku aplikacije potrebnu za

uspješno izvršavanje *HTTP* zahtjeva. Kako bi *Spring Boot* uspješno razlučio kako se uistinu radi o *Service* sloju, potrebno je anotirati takvu klasu izrazom *@Service*. Na ovaj način *Spring Boot* automatski konfigurira mikroservisnu aplikaciju. *Controller* sloj definira *REST* točke za komunikaciju i, koristeći ubrizgavanje ovisnosti, poziva metode iz *Service* sloja. Pojedine metode, ovisno o namjeni, mogu se definirati kao *POST*, *GET*, *PUT*, *DELETE* i druge *HTTP* operacije korištenjem prikladnih anotacija. Također, kako bi *Spring Boot* uspješno prepoznao *Controller* sloj, potrebno je klasu anotirati izrazom *@RestController* i definirati *URL* putanju na kojoj će biti dostupne *REST* točke za komunikaciju. Nakon definiranja pojedinih slojeva i funkcija, dovoljno je pokrenuti aplikaciju i pomoću *HTTP* zahtjeva koristiti njene *REST* točke za komunikaciju. Primjer *Controller* sloja *Spring Boot* aplikacije prikazan je programskim kodom Programski kod 4.1 .

```
@RestController
@RequestMapping("/apartments")
@Api(tags = "apartments")
@RequiredArgsConstructor
public class ApartmentManagementController {
    private final TokenProvider tokenProvider;
    private final ApartmentManagementService apartmentManagementService;

    private final ApartmentRepository apartmentRepository;

    @GetMapping("/getall")
    @ApiOperation(value = "${ApartmentController.getAllApartments}")
    @ApiResponses(value = {//
            @ApiResponse(code = 400, message = "Something went wrong"), //
            @ApiResponse(code = 403, message = "Access denied")})
    public ResponseEntity<List<Apartment>> getAllApartments( HttpServletRequest
request) throws RuntimeException {
        String token = request.getHeader("Authorization");
        if (token == null || !token.startsWith("Bearer ")) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(new
ArrayList<>());
        }
        token = token.replace("Bearer ", "");
        String username = tokenProvider.getUsernameFromToken(token);
        List<Apartment> apartments =
apartmentRepository.findAllByUsername(username);
        return ResponseEntity.status(HttpStatus.OK).body(apartments);

    }
```

Programski kod 4.1 *Controller* sloj *ApartmentManagement* mikroservisa

4.1.2. *React Native* i programski jezik *Typescript*

React native je moderan radni okvir za izradu višeplatformskih aplikacija utemeljen na principima razvoja *ReactJS* web radnog okvira. *React Native* koristi komponentni pristup razvoju programskih rješenja koji podrazumijeva odvajanje pojedinih grafičkih i funkcionalnih cjelina u posebne datoteke. Ono što čini *React Native* prikladnim za razvoj modernih aplikacija jest broj dostupnih ugrađenih i vanjskih biblioteka koje pokrivaju većinu zahtjeva koje uobičajena moderna aplikacija može imati. Kako bi se inicijalizirala *React Native* aplikacija, prvobitno je potrebno odabrati vrstu aplikacije koja se želi razvijati. *React Native* omogućuje razvoj takozvanih *Bare React Native* i *Expo* aplikacija pri čemu *Bare React Native* aplikacije zahtijevaju ručno konfiguriranje specifičnih radnji i dopuštenja vezanih uz platformu na kojoj će se izvršavati, dok *Expo* aplikacije pružaju apstrahirani konfiguracijski sloj u kojemu je moguće jednostavno konfigurirati aplikaciju za izvršavanje na svim platformama unutar jedne datoteke. Za razvoj *Properbooker* aplikacije, koristit će se *Expo* način rada.

Expo pristup, omogućuje brzu inicijalizaciju i jednostavnu konfiguraciju projekta. Nakon izvršavanja naredbe *npx create-expo-app imeAplikacije*, stvara se osnovna struktura projekta. U glavnom direktoriju projekta mogu se pronaći različite datoteke kao što su *App.js* i *package.json*. Datoteka *App.js* sadrži početni kod aplikacije koji se može dodatno proširiti korisnički definiranim komponentama i logikom, dok *package.json* definira programske ovisnosti koje se mogu koristiti prilikom razvoja. Koristeći *JSX* (*JavaScript XML*) sintaksu, komponente se deklariraju na sličan način kao u *ReactJS*-u. Ovisno o potrebama, razvojni programer može uključiti vanjske komponente i biblioteke instalacijom istih putem *npm* ili *yarn* upravitelja paketima.

Kako bi se naposljetku aplikacija pokrenula, potrebno je izvršiti naredbu *npm start* ili *yarn start*, koja će pokrenuti *Expo* razvojni poslužitelj nakon čega se u terminalu prikazuje struktura s *QR* kodom koji se može skenirati pomoću *Expo Go* aplikacije na mobilnom uređaju. Nakon skeniranja, aplikacija će se automatski učitati na uređaju i omogućiti razvoj u stvarnom vremenu.

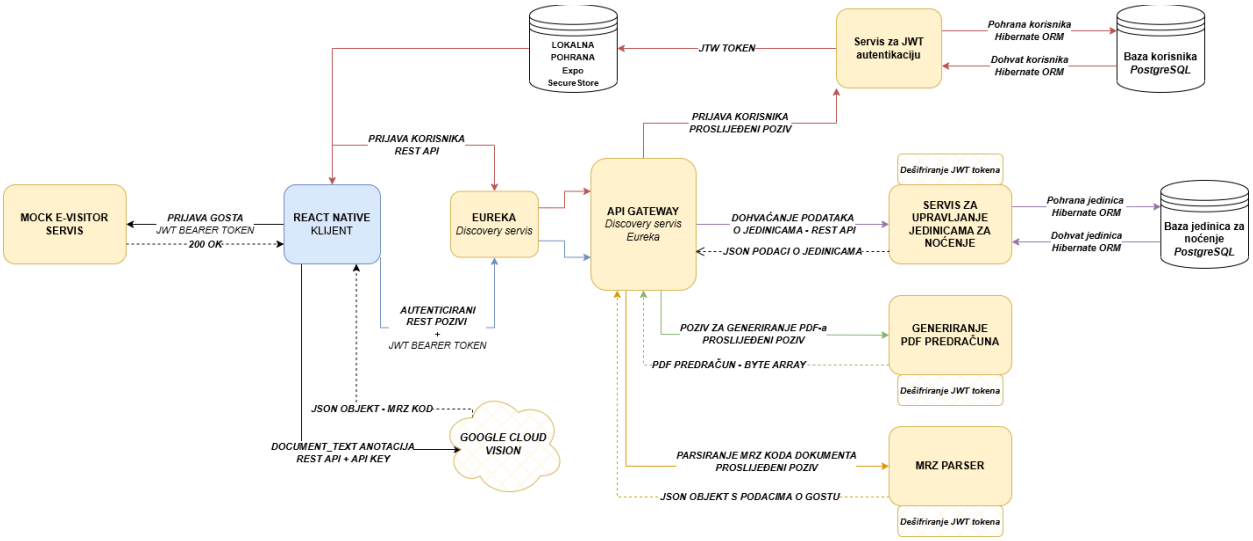
Osim *Javascript* programskog jezika, prilikom razvoja aplikacije, većinski će se koristiti *Typescript*. *Typescript* je moderan programski jezik koji nastaje sa svrhom proširenja funkcionalnosti *Javascript*-a dodavanjem nužnog definiranja tipova podataka prilikom definiranja

varijabli ili parametara i povratnih vrijednosti funkcija. Na ovaj se način smanjuju inače iznimno česte greške prilikom obrade podataka u *Javascript* ekosustavu.

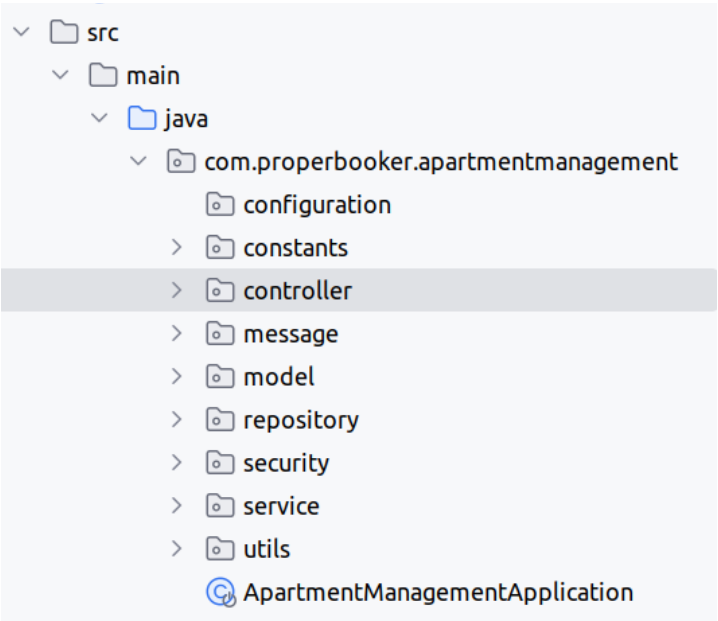
4.2. Programska arhitektura *Properbooker* aplikacije

S obzirom na funkcionalne i nefunkcionalne zahtjeve, za izradu *Properbooker* aplikacije koristit će se mikroservisna arhitektura u kojoj će pojedini elementi i baze podataka biti kontejnerizirani u obliku *Docker* kontejnera. Mikroservisna arhitektura odabrana je zbog velikog broja potencijalnih korisnika aplikacije, točnije broja izdavača jedinica za noćenje. Također, s obzirom na porast broja izdavača u prethodnim godinama, lako je pretpostaviti kako će biti potrebno modificirati sustav kako bi se mogao prilagoditi većem broju korisnika. Koristeći mikroservisnu arhitekturu, skaliranje će biti puno jednostavnije učiniti naspram monolitne aplikacije te neće nužno zahtijevati izmjenu izvornog koda pojedinih mikroservisa. Pojedini mikroservisi u sustavu pružat će zaštićena aplikacijsko-programska sučelja pomoću kojih će njihove metode pozivati klijentska *React Native* aplikacija koristeći prilagođene *REST* zahtjeve. Arhitektura ovog sustava izbjegava međusobnu komunikaciju mikroservisa kako bi se smanjila latencija prilikom pozivanja *HTTP* metoda. Također, kako bi se omogućila pravilna raspodjela *HTTP* zahtjeva pojedinim mikroservisima (*eng. load balacing*), točkama za komunikaciju svih mikroservisa bit će moguće pristupiti putem *API Gateway* mikroservisa koji će, koristeći *round-robin* algoritam, multipliciranim instancama pojedinih mikroservisa pridruživati ravnomjerno raspoređen broj klijentskih poziva. Korištenje *API Gateway* mikroservisa također omogućuje skrivanje postojanja više različitih mikroservisa od klijenta i objedinjuje klijentske *HTTP* zahtjeve pod jedan mikroservis. Kako bi *API Gateway* uspješno funkcionirao, potrebno je implementirati i takozvanu *discovery* uslugu. Navedena usluga pronalazi aktivne instance drugih mikroservisa u sustavu i omogućuje pravilno prosljeđivanje *HTTP* zahtjeva od *API Gateway*-a prema pretplaćenim mikroservisima. Zbog realistične emulacije rada *e-visitor* sustava, mikroservis koji predstavlja *e-visitor* sustav bit će izdvojen iz registra *API Gateway* mikroservisa s obzirom da se raspodjela računalnih resursa tog sustava inače izvršava na udaljenim državnim poslužiteljima. Na slici Slika 4.1. može se vidjeti cjelokupna arhitektura sustava koja uključuje implementirane mikroservise, baze podataka, sigurnosne mehanizme i pozive vanjskim aplikacijsko-programskim sučeljima. Svi mikroservisi unutar aplikacije izvedeni su prateći prilagođeni *MVC* obrazac prilikom čega su svi tranzitivni objekti organizirani u *Model* sloju mikroservisa, logika korištenih

metoda u *Service* sloju, metode za rad s bazom podataka u *Repository* sloju, dok su pozivi metoda i njihova pretvorba u *REST* točke za komunikaciju izdvojeni u *Controller* sloj. Na slici Slika 4.2 prikazana je opisana prilagođena *MVC* struktura mikroservisa. Načini rada i koraci razvoja pojedinih mikroservisa bit će opisani u nastavku.



Slika 4.1 Dijagram arhitekture *Properbooker* aplikacije



Slika 4.2 Prilagođena *MVC* struktura *Spring Boot* projekta

4.3. Razvoj pojedinih usluga unutar *Properbooker* mikroservisne arhitekture

4.3.1. Način rada usluge za očitavanje *MRZ* koda osobnih dokumenata

Kako bi se omogućila uspješna obrada podataka s osobnih dokumenata, mikroservis za dekodiranje *MRZ* koda s osobnih dokumenata koristi odgovor *Google Vision* modela strojnog učenja za računalni vid kao ulazni podatak. *Google Vision* model dostupan je i u obliku aplikacijsko-programskog sučelja koje će se koristiti prilikom razvoja aplikacije. Kako bi se *Google Vision API* mogao koristiti u programskom rješenju, potrebno je u *Credentials* izborniku *Google Cloud* konzole generirati *API* ključ pomoću kojega se korisnik autorizira za korištenje mikroservisa. Nakon izrade *API* ključa, šalje se slika pozadine osobnog dokumenta u *base64* formatu pomoću *HTTP POST* poziva *Vision API* usluzi. Kao odgovor navedenom pozivu, *Vision API* vraća *JSON* objekt s tekstualnim anotacijama fotografije ili prikladan odgovor u slučaju greške. Unutar tekstualnih anotacija pronalazi se posljednje pojavljivanje znaka „<“. Ovaj znak predstavlja kraj *MRZ* koda uz pomoć kojega je moguće jednostavno razlikovati *MRZ* kod od ostatka očitanog teksta poznavajući duljinu *MRZ* koda,. Nakon dohvaćanja *MRZ* koda, kod se prosljeđuje *MRZ Parser* mikroservisu. *MRZ Parser* je projekt otvorenog koda za dekodiranje *MRZ* kodova na osnovu pronalaska specifičnih uzoraka u tekstu koristeći regularne izraze. *MRZ Parser* trenutno podržava očitavanje većine identifikacijskih dokumenata na globalnoj razini te je izrađen u *Java* programskom jeziku što ga čini jednostavnim za pretvorbu u aplikacijsko-programsko sučelje. Za potrebe izrade *Properbooker* aplikacije, navedeni je projekt modificiran i, korištenjem *Spring Boot* radnog okvira, pretvoren u *REST* aplikacijsko-programsko sučelje s ciljem jednostavnog povezivanja mikroservisa i klijentske *React Native* aplikacije. Modificirani mikroservis pruža *REST* komunikacijsku točku kojoj se predaje *MRZ* kod, nakon čega vraća osobne podatke u obliku *JSON* objekta. Podaci se ne skladište u bazi podataka ni lokalnoj pohrani zbog zaštite privatnosti gostiju i praćenja *GDPR* [16] naputaka za rukovanje osjetljivim podacima. Pristup *MRZ Parser* mikroservisu ograničen je autoriziranim korisnicima putem provjere validnosti *JWT* tokena čiji će se način rada opisati u poglavljima niže. Također, način autorizacije i autentikacije korisnika detaljno će se opisati u nastavku. Programski kod Programski kod 4.2

prikazuje *HTTP* poziv prema *MRZ Parser* mikroservisu, dok je na slici Slika 4.3 prikazan *JSON* odgovor mikroservisa prilikom ispravnog dekoriranja *MRZ* koda.

Slika 4.3 *JSON* odgovor *MRZ Parser* mikroservisa

```
const getParsedMRZ = async () =>{
        let fullTextAnnotation = await getDocumentTextDetection();
        let lastMRZChar = fullTextAnnotation.lastIndexOf('<');
        let MRZ = fullTextAnnotation.substring(lastMRZChar-92, lastMRZChar+1);

        const requestBody = {
            mrz: MRZ.toString().replace(/^\n/, '')
        };

        try {
            const response = await axios.post("https://00cb-46-188-225-
44.ngrok.io/api/parse", requestBody, {
                headers: {
                    "Content-Type": "application/json"
                },
            });
            handleAutomaticFieldInput(response.data);
        }
        catch (error){
            ToastAndroid.show("Error reading ID card, please try again",
ToastAndroid.SHORT);
        }
    }
```

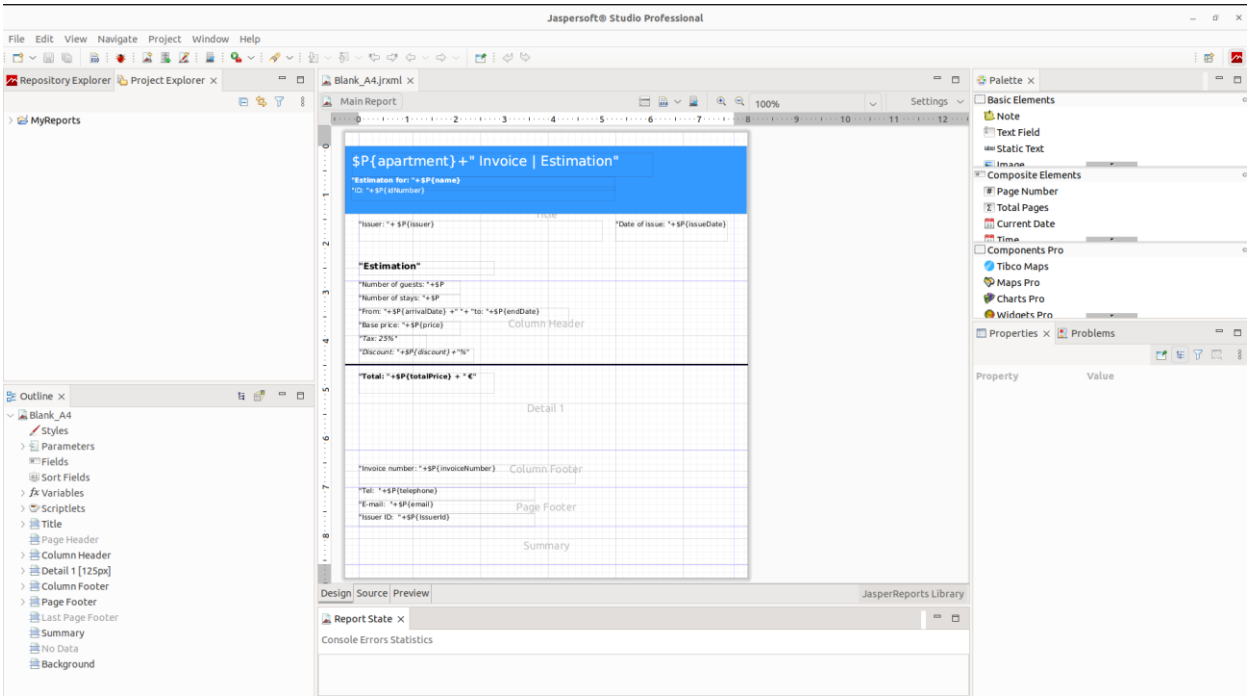
Programski kod 4.2 *HTTP* poziv *MRZ Parser* mikroservisu

4.3.2. Način rada usluge za generiranje *PDF* predračuna

Izrada predračuna i procjena troškova jedan je od najčešćih zahtjeva prilikom primanja većih skupina gostiju ili organiziranih poslovnih grupa zbog čega je bitno implementirati takvu značajku unutar aplikacije. Upravo zbog toga, implementiran je mikroservis za generiranje predračuna u *PDF* formatu koji, na osnovu unesenih podataka o noćenju, generira *PDF* dokument kojeg je moguće pohraniti ili proslijediti gostima. S obzirom da je generiranje *PDF* dokumenta programski zahtjevan proces, korištena je popularna *jasper-reports* biblioteka koja na osnovu *jrxml* datoteke generira *PDF* dokument i vraća ga kao odgovor na *HTTP* zahtjev. Kako bi se stvorila *jrxml* datoteka, potrebno je koristiti *Jaspersoft Studio* radni okvir u kojemu se mogu definirati varijabilna tekstualna polja koja će se moći programski popuniti unutar mikroservisa. Nakon definiranja dizajna dokumenta, *Jasper Reports Studio* generira *jrxml* datoteku iz koje se, koristeći metode *jasper-reports* biblioteke, generira konačni popunjeni *PDF* dokument. Opisani mikroservis u *Controller* sloju aplikacije pruža *REST* točku za komunikaciju koja kao argument prima *JSON* objekt s potrebnim podacima za generiranje predračuna. Na osnovu primljenih podataka stvara se *EstimationDetails* objekt čije se vrijednosti polja koriste za popunjavanje *jrxml* datoteke nakon čega se pomoću *JasperCompileManagement* klase i *compileReport()* statičke metode generira željeni dokument u *PDF* formatu. Točan način generiranja datoteke i ispune dokumenta prikazan je programskim kodom Programski kod 4.3. Na slici Slika 4.4 moguće je vidjeti sučelje *Jaspersoft Studio* radnog okvira, dok je na slici Slika 4.5 moguće vidjeti izgled generiranog *jrxml* dokumenta.

```java
@Service
@RequiredArgsConstructor
public class PdfGeneratorService {
    private final ObjectMapper objectMapper;

    public byte[] generatePdfFromJson(EstimationDetails details) throws Exception
{
        JasperReport jasperReport = compileReport();

        String json = objectMapper.writeValueAsString(details);

        JsonDataSource dataSource = new JsonDataSource(new
ByteArrayInputStream(json.getBytes()));

        Map<String, Object> parameters = new HashMap<>();
        parameters.put("name", details.getName());
        parameters.put("apartment", details.getApartment());
        parameters.put("idNumber", details.getIdNumber());
        parameters.put("numberOfGuests", details.getNumberOfGuests());

      //... Ostatak popunjavanja dokumenta
        parameters.put("telephone", details.getTelephone());
        parameters.put("email", details.getEmail());
        parameters.put("issuerId", details.getIssuerId());

        JasperPrint jasperPrint = JasperFillManager.fillReport(jasperReport,
parameters, dataSource);
        return JasperExportManager.exportReportToPdf(jasperPrint);
    }

    private JasperReport compileReport() throws JRException {
        String jrxmlPath = "src/main/resources/templates/pdfestimation.jrxml";
        return JasperCompileManager.compileReport(jrxmlPath);
    }
}
```

Programski kod 4.3 Generiranje *PDF* predračuna unutar *PDFGeneratorService* klase



Slika 4.4 Sučelje *Jaspersoft Studio* radnog okvira

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created with Jaspersoft Studio version 8.2.0.final using JasperReports Library version 6.20.3-415f9428cffdb6805c6f85bbb29ebaf18813a2ab  -->
<jasperReport xmlns="http://jasperreports.sourceforge.net/jasperreports" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="h
    <property name="com.jaspersoft.studio.data.defaultdataadapter" value="One Empty Record"/>
    <parameter name="name" class="java.lang.String"/>
    <parameter name="apartment" class="java.lang.String"/>
    <parameter name="idNumber" class="java.lang.String"/>
    <parameter name="numberOfGuests" class="java.lang.Integer"/>
    <parameter name="numberOfStays" class="java.lang.String"/>
    <parameter name="arrivalDate" class="java.lang.String"/>
    <parameter name="endDate" class="java.lang.String"/>
    <parameter name="discount" class="java.lang.Double"/>
    <parameter name="issuer" class="java.lang.String"/>
    <parameter name="issueDate" class="java.lang.String"/>
    <parameter name="price" class="java.lang.Double"/>
    <parameter name="totalPrice" class="java.lang.Double"/>
    <parameter name="invoiceNumber" class="java.lang.String"/>
    <parameter name="telephone" class="java.lang.String"/>
    <parameter name="email" class="java.lang.String"/>
    <parameter name="IssuerId" class="java.lang.String"/>
    <queryString>
        <![CDATA[]]>
    </queryString>
    <background>
        <band splitType="Stretch"/>
    </background>
    <title>
        <band height="199" splitType="Stretch">
            <frame>
                <reportElement mode="Opaque" x="-20" y="0" width="594" height="100" forecolor="#FFFFFF" backcolor="#3399FF" uuid="0fef841b-2b0d-4e20
                <textField>
                    <reportElement x="10" y="10" width="445" height="35" forecolor="#FFFFFF" uuid="ff7097fd-d73e-47bf-a8a2-57bb4ae3e96e"/>
                    <textElement>
                        <font size="20"/>
                    </textElement>
                    <textFieldExpression><![CDATA[$P{apartment}+" Invoice | Estimation"]]></textFieldExpression>
                </textField>
                <textField>
                    <reportElement x="9" y="45" width="390" height="20" forecolor="#FFFFFF" uuid="5f4960b3-07f7-4aa1-a915-780922d50d5b"/>
                    <textElement>
                        <font isBold="true"/>
                    </textElement>
                    <textFieldExpression><![CDATA["Estimaton for: "+$P{name}]]></textFieldExpression>
                </textField>
                <textField>
Design Source Preview
JasperReports Library
Report State
Console Errors (0) Statistics
Start Report Execution
Writable
Insert
1:1:0
```

Slika 4.5 Generirani *JRXML* kod

4.3.3. Usluga za *JWT* token autentikaciju korisnika i autorizacija *HTTP* zahtjeva

Kako bi se onemogućilo maliciozno korištenje mikroservisa i pristup osjetljivim podacima o gostima, implementiran je mikroservis za autentikaciju korisnika i generiranje *JWT* tokena za autorizaciju *HTTP* zahtjeva. Unatoč tomu što neki mikroservisi u sustavu ne skladište osjetljive podatke, omogućivanje korištenja mikroservisa isključivo registriranim korisnicima je preporučljivo zbog mogućih *denial-of-service* napada na sustav. *Denial-of-service* napadi predstavljaju skupinu hakerskih napada s ciljem preopterećivanja računalne konfiguracije na kojoj se pojedini mikroservis izvršava upućivanjem iznimno velikog broja zahtjeva. Ako mikroservis pozitivno odgovara na svaki zahtjev i izvršava određen proces, poput generiranja *PDF* datoteke ili dešifriranja *MRZ* koda, usluga može naglo prestati s radom zbog manjka računalnih resursa. Iako proces autorizacije zahtijeva određenu količinu resursa, takav proces je manje računalno intenzivan i manja je mogućnost pada mikroservisa. Osim toga, *Spring Security* radni okvir koji se koristi prilikom izrade ove aplikacije pruža mehanizme za ograničavanje spomenutih napada.

Mikroservis za *JWT* autorizaciju i autentikaciju korisnika uobičajen je *Spring Boot* mikroservis koji implementira *Spring Security* radni okvir za zaštitu *REST* točaka za komunikaciju drugih mikroservisa u sustavu. *JWT* token ili *JSON web token* predstavlja standard za izmjenu informacija među strankama koristeći prilagođeni *JSON* objekt koji se u svojem standardnom obliku sastoji od tri dijela. Prvi dio, ili zaglavlje, sadrži informacije o algoritmu pomoću kojega su šifrirani podaci unutar tokena i vrsti tokena. Drugi dio *JWT* tokena, *Payload*, sadrži podatke koji se žele izmijeniti među strankama. Navedeni dio tokena se sastoji od prethodno definiranih i korisnički definiranih polja. Predefinirana polja uključuju podatke o izdavaču tokena, vremenu isteka tokena, namijenjenoj publici i druge podatke, dok korisnički definirani podaci mogu biti bilo što što korisnik želi dodatno prenijeti unutar tokena. Posljednji dio tokena predstavlja digitalni potpis. Digitalni potpis je potreban kako bi se verificirala činjenica da podaci nisu izmijenjeni tijekom prijenosa. Digitalni potpis nastaje kriptiranjem svih navedenih dijelova tokena određenim algoritmom (najčešće *HMAC SHA256*) zajedno s privatnim ključem. Svaka stranka koja posjeduje privatni ključ može dešifrirati *JWT* token, verificirati njegovu validnost i pristupiti poslanim podacima. Opisani mehanizam ključan je za zaštitu *REST* točaka za komunikaciju u ovom sustavu. Mikroservis za autentikaciju prvobitno prima registracijske podatke od korisnika putem *REST* točke za komunikaciju, nakon čega skladišti podatke korisnika i kriptiranu lozinku u zaštićenoj bazi podataka. Nakon registracije, korisnik se mora prijaviti u sustav putem *HTTP* zahtjeva. Kao pozitivan odgovor na *HTTP* zahtjev, usluga vraća *JWT* autorizacijski token koji se koristi prilikom

korištenja metoda drugih mikroservisa. Token, zbog povećane sigurnosti, ostaje validan jedan sat, nakon čega se korisnik mora ponovno prijaviti u sustav kako bi se token osvježio i obnovila sesija. Ostali mikroservisi u sustavu skladište privatan ključ kojim je *JWT* token prvobitno kriptiran te, ako je token validan, pomoću njega uspješno dekriptiraju sadržaj tokena i pristupaju informacijama. U korištenoj implementaciji, u *subject* polje tokena upisuje se korisničko ime koje je unikatno u sustavu i služi za pristup podacima iz baza podataka mikroservisa. Prilikom svakog *HTTP* zahtjeva prema mikroservisima, zahtjev mora sadržavati autorizacijski *JWT* token u zaglavlju. Također, u svakom pojedinom mikroservisu u sustavu implementiran je mehanizam za dekriptiranje *JWT* tokena koristeći *jjwt* biblioteku koja pomoću algoritma definiranog u zaglavlju tokena i privatnog ključa dolazi do željenih podataka. Implementacija generatora tokena prikazana je programskim kodom Programski kod 4.4, dok je proces dekriptiranja tokena prikazan programskim kodom Programski kod 4.5.

```
public String createToken(String username, List<AppUserRole> appUserRoles) {

    Claims claims = Jwts.claims().setSubject(username);
    claims.put("auth", appUserRoles.stream().map(s -> new
SimpleGrantedAuthority(s.getAuthority())).filter(Objects::nonNull).collect(Collec
tors.toList()));

    Date now = new Date();
    Date validity = new Date(now.getTime() + validityInMilliseconds);

    return Jwts.builder()//
        .setClaims(claims)//
        .setIssuedAt(now)//
        .setExpiration(validity)//
        .signWith(SignatureAlgorithm.HS256, secretKey)//
        .compact();
  }
```

Programski kod 4.4 Generiranje *JWT* tokena

```
public Claims getAllClaimsFromToken(final String token) {
      System.out.println(token);
      return Jwts.parser()
              .setSigningKey("secret-key".getBytes())
              .parseClaimsJws(token)
              .getBody();
  }
```

Programski kod 4.5 Dekriptiranje *JWT* tokena koristeći *jjwt* biblioteku

4.3.4. Usluga za upravljanje ugostiteljskim jedinicama

S obzirom da je *e-visitor* sustav ponekad preopterećen, implementiran je vlastiti mikroservis za upravljanje ugostiteljskim jedinicama u obliku *Spring Boot CRUD* aplikacije. *CRUD* aplikacije predstavljaju sustave koji pružaju *REST* komunikacijske točke za stvaranje, brisanje, čitanje i ažuriranje korisnički definiranih objekata. Ovaj mikroservis stoga nudi potpunu kontrolu nad uskladištenim ugostiteljskim jedinicama i njihovom pohranom. Kako bi se uspješno pohranili podaci o ugostiteljskim jedinicama koristi se *Hibernate ORM* i *Spring PostgreSQL driver* biblioteka. *Hibernate ORM* (*Object-relational mapper*) služi kako bi se bez izravnih *SQL* upita uspješno skladištili podaci o ugostiteljskim jedinicama u *PostgreSQL* bazu podataka. Na ovaj se način izbjegavaju ljudske greške prilikom definiranja konkretnih *SQL* upita, a zbog korištenja *Spring Boot JPA Repository* radnog okvira, nije potrebno implementirati standardne metode za upravljanjem bazom podataka već je isključivo potrebno implementirati *JPARepository* sučelje u vlastitom *Repository* sloju i primijeniti već implementirane metode za manipulaciju podacima.

Mikroservis za upravljanje ugostiteljskim jedinicama, osim podataka o pojedinim jedinicama za noćenje, u bazi podataka skladišti i jedinstveno korisničko ime registriranih korisnika kako bi se jednostavno pristupilo ugostiteljskim jedinicama pojedinog korisnika. Navedena relacija je iznimno bitna zbog prikaza podataka u korisničkom sučelju aplikacije i osiguranja privatnosti osobnih informacija. Također, u opisanom je mikroservisu također implementiran mehanizam za dešifriranje autorizacijskih *JWT* tokena i mogu mu pristupiti isključivo registrirani korisnici. Programskim kodom Programski kod 4.6 prikazan je dio *Controller* sloja opisanog mikroservisa i pružene *REST* komunikacijske točke.

```
@RestController
@RequestMapping("/apartments")
@Api(tags = "apartments")
@RequiredArgsConstructor
public class ApartmentManagementController {
    private final TokenProvider tokenProvider;
    private final ApartmentManagementService apartmentManagementService;

    private final ApartmentRepository apartmentRepository;

    @GetMapping("/getall")
    @ApiOperation(value = "${ApartmentController.getAllApartments}")
    @ApiResponses(value = {//
            @ApiResponse(code = 400, message = "Something went wrong"), //
            @ApiResponse(code = 403, message = "Access denied")})
    public ResponseEntity<List<Apartment>> getAllApartments( HttpServletRequest
request) throws RuntimeException {
        String token = request.getHeader("Authorization");
        if (token == null || !token.startsWith("Bearer ")) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(new
ArrayList<>());
        }
        token = token.replace("Bearer ", "");
        String username = tokenProvider.getUsernameFromToken(token);
        List<Apartment> apartments =
apartmentRepository.findAllByUsername(username);
        return ResponseEntity.status(HttpStatus.OK).body(apartments);

    }
   // Ostatak REST točaka za komunikaciju
}
```

Programski kod 4.6 Prikaz dijela *Controller* sloja *ApartmentManagement* mikroservisa

4.3.5. *Netflix Eureka* usluga za pronalaženje aktivnih instanci mikroservisa

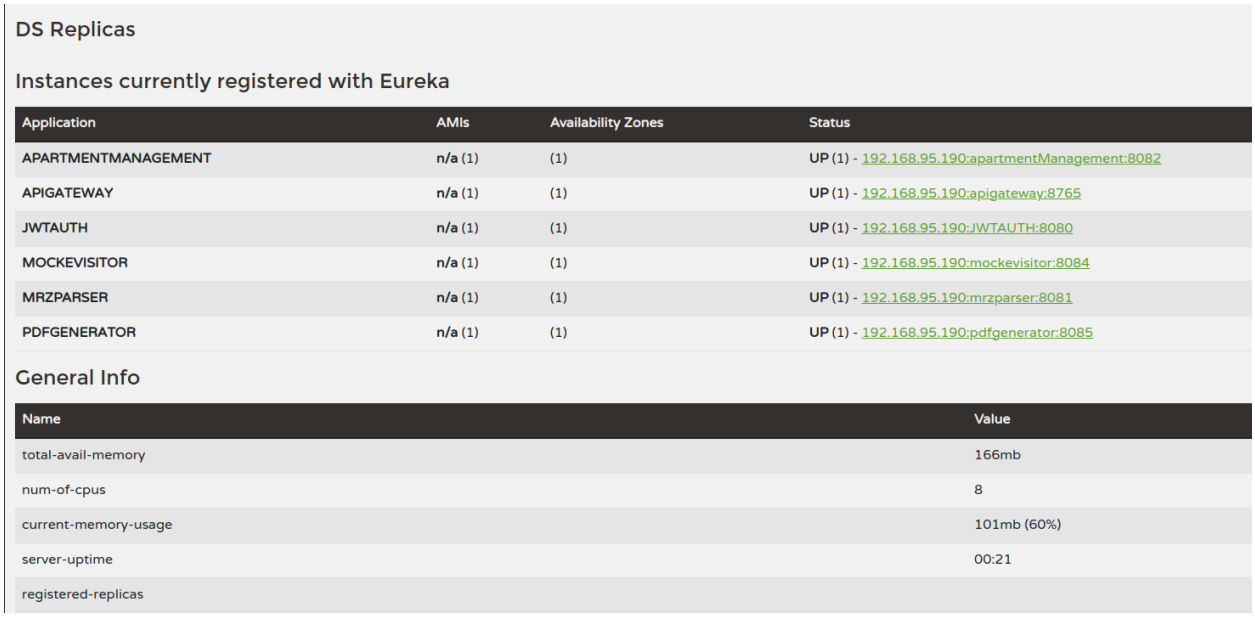
U distribuiranoj mikroservisnoj arhitekturi čest je slučaj da postoji iznimno velik broj mikroservisa, ali i njihovih multipliciranih instanci kojima se pristupa putem *REST* komunikacijskih točki izloženih na određenoj *IP* adresi i računalnom *port*-u . Prilikom povećanja broja instanci mikroservisa i složenosti arhitekture, pojavljuje se problem pronalaska aktivnih instanci pojedinih mikroservisa u sustavu. Kako bi se opisani problem izbjegao, u složene se

sustave u pravilu ugrađuje takozvani *discovery* mikroservis. Spomenuti mikroservis automatski pronalazi sve pretplaćene mikroservise u sustavu i njihove različite instance zbog lakšeg upravljanja i konfiguriranja istih. U sklopu izrade ove aplikacije koristit će se *Netflix Eureka Discovery Service*. Kako bi se uspješno implementirao *Eureka Discovery* mikroservis, potrebno je razumjeti različite pojmove koje *Eureka* definira. Prvi od tih pojmova jest *Eureka* poslužitelj. *Eureka* poslužitelj je običan mikroservis koji na osnovu korisnički definirane konfiguracije otkriva pojedine instance mikroservisa u sustavu. Kada se mikroservis pojavi u sustavu, dodjeljuje mu se jedinstveni identifikacijski kod i sprema se u registar pronađenih instanci. Sljedeći pojam jest *Eureka* klijent. Svaki mikroservis koji koristi *Spring Cloud Starter Netflix Eureka Client* biblioteku s verzijom prilagođenom verziji *Spring Boot* radnog okvira i određene konfiguracijske parametre naziva se *Eureka* klijentom. *Eureka* klijent je običan mikroservis koji je konfiguriran na način da ga *Eureka* poslužitelj može s lakoćom pronaći. Primjer konfiguracijskih parametara *Eureka* klijenta prikazan je programskim kodom Programski kod 4.7. Prikazani parametri preuzeti su iz *PDFGenerator* mikroservisa u kojemu je definirano ime mikroservisa koje će se prikazati u *Eureka* grafičkom sučelju i *port*, što je uz prikladnu programsku ovisnost dovoljno za prepoznavanje klijenta u *Eureka* poslužitelju. Posljednji pojam jest *Eureka* instanca. Instanca je jedan pokrenuti mikroservis kojeg *Eureka* poslužitelj prepoznaje. Može se zaključiti kako *Eureka* poslužitelj pronalazi *Eureka* klijente, dok se svaki multiplicirani *Eureka* klijent naziva instancom.

```
spring.application.name=pdfgenerator
server.port=8083
```

Programski kod 4.7 *YAML* konfiguracijska datoteka *Eureka* klijenta

Eureka poslužitelj također poslužuje i grafičko sučelje pomoću kojega se mogu pratiti pronađeni aktivni mikroservisi. Osim toga *Eureka* poslužitelj omogućuje i raspodjelu broja *HTTP* zahtjeva pojedinim instancama mikroservisa. Raspodjela se može korisnički definirati ili se može koristiti predefinirani *round-robin* algoritam. Na slici 4.6 moguće je vidjeti izgled grafičkog sučelja *Eureka* poslužitelja zajedno s pronađenim klijentima unutar arhitekture. Na primjeru prikazanom slikom Slika 4.6 vidljivi su isključivo klijenti jer postoji samo jedna instanca svakog pojedinog klijenta. U slučaju kada bi broj korisnika naglo porastao, broj instanci bi se mogao također dinamički povećati. Ovakav način skaliranja naziva se horizontalno skaliranje sustava.



DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
APARTMENTMANAGEMENT	n/a (1)	(1)	UP (1) - 192.168.95.190:apartmentManagement:8082
APIGATEWAY	n/a (1)	(1)	UP (1) - 192.168.95.190:apigateway:8765
JWTAUTH	n/a (1)	(1)	UP (1) - 192.168.95.190:JWTAUTH:8080
MOCKEVISITOR	n/a (1)	(1)	UP (1) - 192.168.95.190:mockevisitor:8084
MRZPARSER	n/a (1)	(1)	UP (1) - 192.168.95.190:mrzparser:8081
PDFGENERATOR	n/a (1)	(1)	UP (1) - 192.168.95.190:pdfgenerator:8085

General Info

Name	Value
total-avail-memory	166mb
num-of-cpus	8
current-memory-usage	101mb (60%)
server-uptime	00:21
registered-replicas	

Slika 4.6 Prikaz aktivnih instanci mikroservisa u *Eureka* sučelja

4.3.6. *API Gateway* usluga, raspodjela računalnih resursa i preusmjeravanje *API* poziva

Unatoč tomu što se implementacije mikroservisne arhitekture često razlikuju ovisno o namjeni, gotovo svaka distribuirana mikroservisna arhitektura uključuje i implementaciju *API Gateway* mikroservisa. *API Gateway* mikroservis omogućuje prosljeđivanje *HTTP* zahtjeva prikladnim instancama mikroservisa i sakriva implementaciju mikroservisne arhitekture od klijenta. Korištenjem API Gateway mikroservisa klijentu je poznata samo *IP* adresa i port *API Gateway* mikroservisa zbog čega ne mora tražiti konkretne mrežne lokacije instanci svakog pojedinog mikroservisa. Kako bi *API Gateway* mikroservis uspješno radio, potrebno je na prethodno opisani način konfigurirati *Eureka* poslužitelj za pronalaženje instanci ostalih mikroservisa. *API Gateway* zatim prosljeđuje *HTTP* pozive mikroservisima pronađenim u *Eureka* registru. U sklopu razvoja *Properbooker* aplikacije, *API Gateway* mikroservis definiran je kao zaseban *Spring Boot* mikroservis s posebnom konfiguracijom koja omogućuje pronalazak aktivnih instanci mikroservisa. Izgled konfiguracijske datoteke prikazan je programskim kodom Programski kod 4.8.

```
spring.application.name=apigateway
server.port=8765
spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.discovery.locator.lower-case-service-id=true
```

Programski kod 4.8 *YAML* konfikuracijska datotoeka *API Gateway* mikroservisa

Osim navedenih funkcija ovog mikroservisa, *API Gateway* ponekad uključuje i implementaciju logike za autentikaciju i generiranje *JWT* tokena. Unatoč tomu što je ovakav postupak česta praksa u industriji, prilikom izrade *Properbooker* aplikacije izbjegnut je ovakav pristup zbog očuvanja mogućnosti povećanja zalihosti mikroservisa za autentikaciju i generiranje *JWT* tokena. U slučaju da je spomenuta logika implementirana u *API Gateway* mikroservis, prilikom prestanka rada *API Gateway* mikroservisa, korisnik bi izgubio mogućnost prijave i registracije. Izdvajanje logike u poseban mikroservis izbjegava problem jedinstvene točke kvara sustava i osigurava stabilnost sustava. Opisani problem se također može riješiti stvaranjem više instanci *API Gateway* mikroservisa, no to nije uobičajena praksa prilikom izrade mikroservisne arhitekture.

4.3.7. Način rada usluge za emulaciju rada *e-visitor* sustava

Iako *e-visitor* sustav razvojnim programerima omogućuje korištenje testne okoline, zbog nekonzistentnosti u radu sustava, implementiran je poseban *Spring Boot* mikroservis koji emulira rad *e-visitor* sustava. U trenutnoj implementaciji, mikroservis za emulaciju rada *e-visitor* sustava pruža jednu *REST* komunikacijsku točku koja služi za validaciju primljenog zahtjeva za prijavu gosta. S obzirom da algoritam za validaciju podataka o prijavljenim gostima nije javno dostupan, mikroservis provjerava popunjenost svih polja u primljenom *JSON* objektu i vraća pozitivan odgovor u slučaju uspješne provjere. U produkcijskoj inačici *Properbooker* aplikacije ovaj će mikroservis, zajedno sa mikroservisom za upravljanje jedinicama za noćenje, biti moguće jednostavno zamijeniti aplikacijsko programskim sučeljem koje nudi produkcijski *e-visitor* sustav. Programskim kodom Programski kod 4.9 prikazan je *Controller* sloj opisanog mikroservisa u kojemu je definirana *REST* točka za komunikaciju. Zbog činjenice da produkcijski *e-visitor* sustav

koristi svoj mehanizam autentikacije korisnika i autorizacije *HTTP* zahtjeva, implementirani sustav neće koristiti *JWT* token za autorizaciju.

```
@RestController
@RequestMapping("/evisitor")
public class GuestRegistrationController {

    @PostMapping("/register")
    public ResponseEntity<RegisterResponse> registerGuest(@RequestBody
Registration registration){
        return ResponseEntity.status(HttpStatus.OK).body(new
RegisterResponse("Guest successfully registered"));
    }
}
```

Programski kod 4.9 *Controller* sloj *Mock e-visitor* mikroservisa

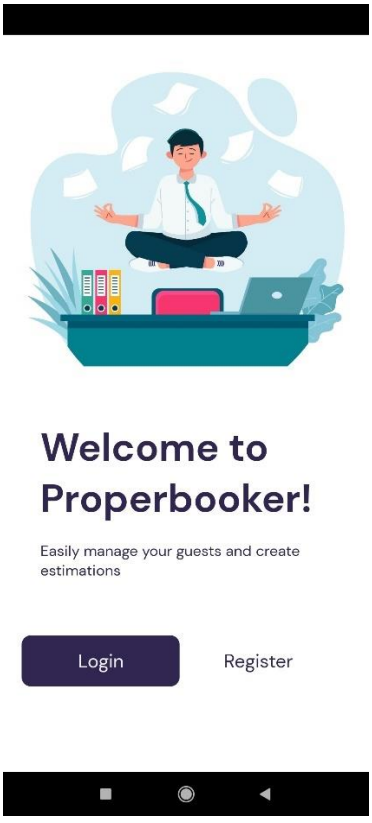
4.4. Klijentska *React Native* aplikacija za automatsku prijavu gostiju i generiranje predračuna

S obzirom da se korištenjem *Properbooker* aplikacije želi smanjiti potreba za lokalnom pohranom osobnih identifikacijskih dokumenata, aplikacija je dizajnirana za korištenje na mobilnim uređajima zbog jednostavnog pristupa kameri uređaja i jednostavnoj tranzitivnoj pohrani fotografija dokumenata unutar aplikacije bez trajnog skladištenja. Kao što je opisano u prethodnim poglavljima, *React Native* radni okvir odabran je zbog mogućnosti pokretanja aplikacije na većini mobilnih platformi, a zbog prenosivosti znanja iz domene razvoja web sučelja, zahtijeva relativno malo tehničke prilagodbe razvojnih inženjera u usporedbi s drugim radnim okvirima. U nastavku će biti objašnjena struktura klijentske aplikacije, rukovanje asinkronim *HTTP* pozivima, sigurno skladištenje osjetljivih informacija i dizajn iskustva korištenja *Properbooker* aplikacije.

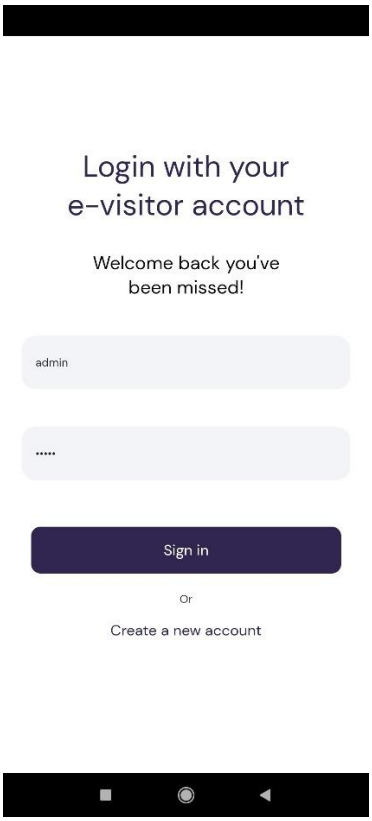
4.4.1. Izgled aplikacije i iskustvo korištenja

Properbooker aplikacija dizajnirana je na način da ubrza svaki korak prilikom prijave gostiju ili generiranja predračuna zbog čega je broj potrebnih korisničkih interakcija s aplikacijom smanjen

na minimum. Upravo zbog navedenog razloga, *Properbooker* aplikacija sastoji se od tri glavna zaslona i dva pomoćna funkcionalna zaslona za dijalog s korisnikom. Prvi zaslon jest zaslon za prijavu ili registraciju korisnika koji prikazuje poruku dobrodošlice i izbornik za prijavu u sustav ili registraciju novog računa. Pritiskom na tipku *Login*, korisnika se preusmjeruje na sljedeći zaslon u kojemu se traži unos korisničkih podataka za pristup glavnom dijelu aplikacije. U slučaju da korisnik ne posjeduje otvoren *e-visitor* račun, pritiskom na tekst *Create new account*, korisnika se preusmjeruje na zaslon s informacijama o otvaranju novog *e-visitor* računa. S obzirom da je otvaranje *e-visitor* računa složen postupak koji zahtijeva kontakt s *e-visitor* podrškom, registracija ne može biti implementirana unutar aplikacije. Izgled opisanih zaslona prikazan je slikama Slika 4.7 i Slika 4.8.

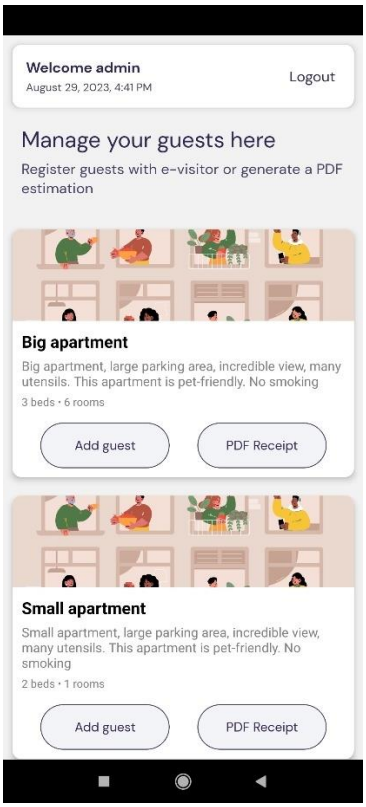


Slika 4.7 Zaslon dobrodošlice u *Properbooker* aplikaciju



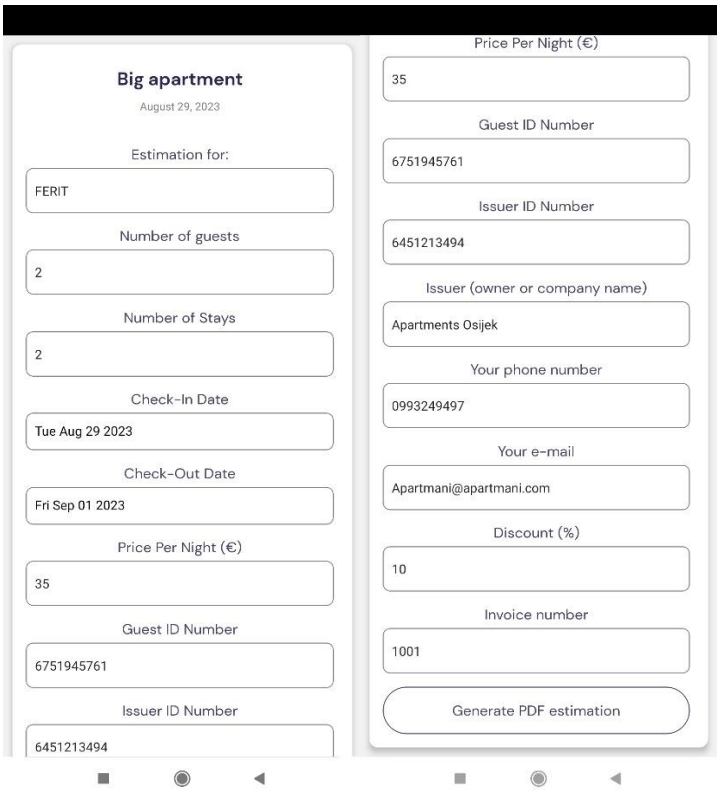
Slika 4.8 Zaslon za prijavu korisnika u aplikaciju

Nakon uspješne autentikacije, korisnika se preusmjerava na glavni zaslon za upravljanje gostima. U slučaju neuspješne prijave, pojavljuje se poruka o vrsti greške. U glavnom zaslonu aplikacije prikazuje se popis registriranih ugostiteljskih objekata u obliku liste kartica, tipka za odjavu, korisničko ime te trenutni datum i vrijeme. Svaka pojedina kartica sadrži ime jedinice za noćenje, detaljan opis jedinice, broj spavaonica i broj soba. Osim informacija o objektu, na svakoj pojedinačnoj kartici nalaze se tipke za prijavu gostiju i generiranje *PDF* predračuna kako bi se tim funkcijama moglo što brže pristupiti. Izgled glavnog zaslona zajedno s opisanim karticama, prikazan je slikom Slika 4.9.

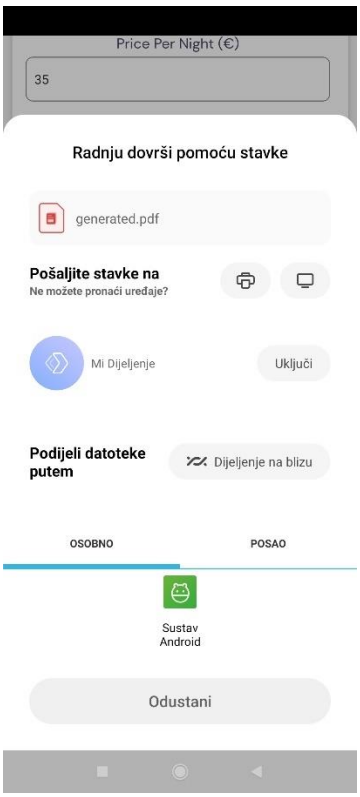


Slika 4.9 Izgled glavnog zaslona aplikacije

Pritiskom na tipku *PDF estimation* otvara se dijaloški prozor u kojemu se od korisnika traži unos potrebnih podataka za generiranje *PDF* dokumenta. Unatoč tomu što postoji velik broj potrebnih informacija koje korisnik mora unijeti, ovakav način unosa ubrzava stvaranje predračuna naspram standardnih alata zbog slijednog unosa podataka bez potrebe za unošenjem informacija na točno određenu lokaciju u dokumentu. U slučaju unosa nepravilne vrijednosti u određeno polje, korisniku se prikazuje prikladna poruka s uputama za pravilan unos podataka. Također, ako korisnik nije ispunio određeno polje, a pokušao je generirati predračun pritiskom na tipku *Generate PDF estimation*, prikazuje se poruka kako sva polja nisu ispunjena i onemogućuje se generiranje dokumenta. Nakon uspješnog unosa potrebnih informacija, pritiskom na tipku za generiranje predračuna otvara se novi dijaloški okvir u kojemu korisnik može odabrati aplikaciju pomoću koje će podijeliti generirani predračun. U slučaju da korisnik želi pohraniti predračun, u dijaloškom okviru može odabrati upravitelj datotekama te dokument spremiti na željenu lokaciju ili ga podići na pohranu u oblaku. Izgled zaslona za generiranje predračuna prikazan je slikom Slika 4.10, dok je slikom Slika 4.11 prikazan izgled dijaloškog okvira za pohranu ili dijeljenje generiranog predračuna.

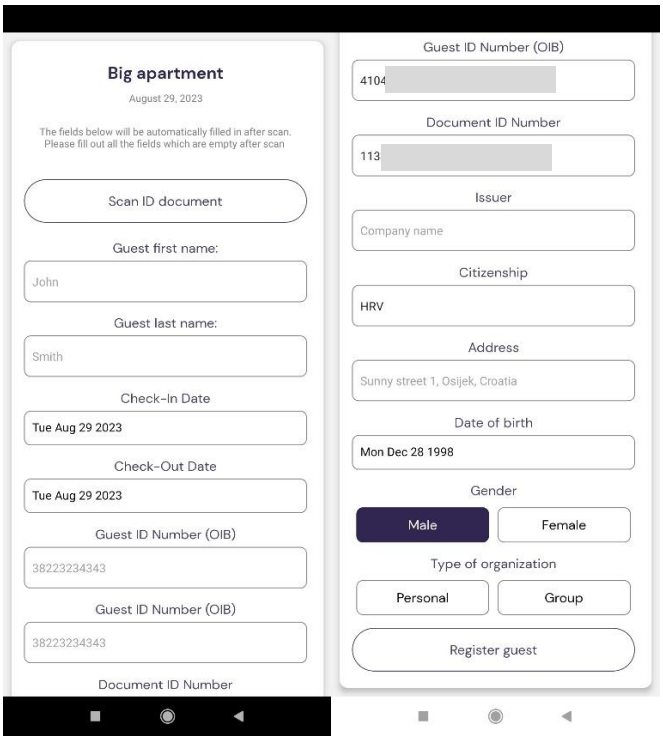


Slika 4.10 Dijaloški zaslon za unos podataka za generiranje *PDF* predračuna

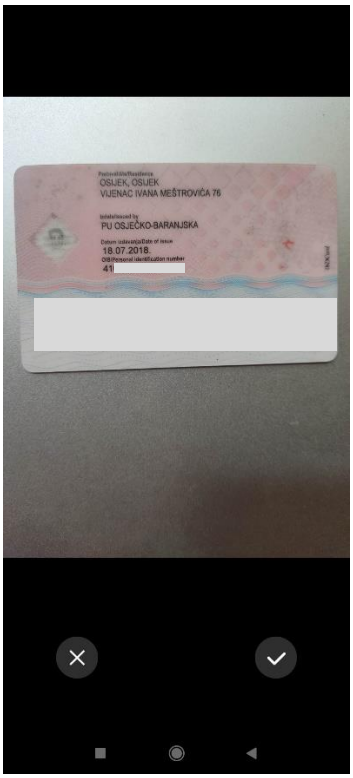


Slika 4.11 Dijaloški okvir za dijeljenje *PDF* predračuna

Posljednja funkcija i samim time posljednji zaslon unutar aplikacije jest zaslon za prijavu gostiju skeniranjem osobnog identifikacijskog dokumenta. Pritiskom na tipku *Add guest* na kartici za prikaz ugostiteljske jedinice, otvara se dijaloški zaslon za unos podataka o noćenju i osobnih podataka gosta te tipke za skeniranje osobnog dokumenta i prijavu gosta. Također, prikazano je ime ugostiteljskog objekta u koji se gost prijavljuje te upute za korištenje funkcije skeniranja osobnog dokumenta. Pritiskom na tipku *Scan ID document* otvara se kamera uređaja sa sučeljem za fotografiranje. Nakon fotografiranja *MRZ* koda identifikacijskog dokumenta, kod se dešifrira i automatski se popunjavaju sva polja pokrivena *MRZ* kodom. Na ovaj način, broj polja koje korisnik mora ispuniti, smanjuje se za više od 50%. Također, onemogućeno je skeniranje osobnih dokumenata iz lokalne pohrane kako bi se ograničilo skladištenje osjetljivih dokumenata na korisničkim uređajima. U slučaju neuspješnog skeniranja dokumenta, korisniku se prikazuje poruka s opisom greške koja se dogodila prilikom skeniranja. Podatke koji nisu pokriveni *MRZ* kodom, korisnik unosi ručno. S obzirom da podaci očitani iz *MRZ* koda ne mogu biti netočni zbog načina kriptiranja osobnih podataka, korisnik ne mora posebno pregledavati točnost automatski generiranih podataka. Izgled dijaloškog zaslona za dodavanje gostiju prikazan je slikom Slika 4.12, dok je sučelje za fotografiranje dokumenta prikazano slikom Slika 4.13.



Slika 4.12 Izgled sučelja za prijavu novih gostiju



Slika 4.13 Izgled sučelja za fotografiranje osobnih dokumenata

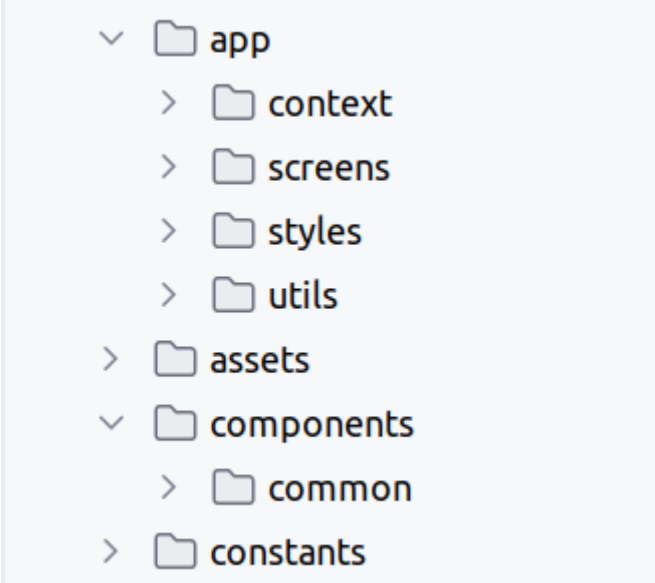
4.5. Implementacija dizajna i značajki *React Native* aplikacije

Implementacija prethodno opisanih zaslona i značajki složen je proces koji zahtijeva pravilno strukturiranje projektnih datoteka, praćenje programskih stanja i definiranje pravilne hijerarhijske strukture izrađenih *React Native* komponenti što će biti opisano u nastavku.

4.5.1. Projektna struktura klijentske *React Native* aplikacije

Prilikom razvoja *React Native* aplikacije, bitno je obratiti pozornost na strukturu pojedinih elemenata i komponenti aplikacije kako bi se osigurala visoka razina čitljivosti koda i olakšale moguće buduće izmjene. S obzirom da, poput *ReactJS* radnog okvira, *React Native* koristi funkcionalni i komponentni pristup razvoju. Svaka grafička cjelina koja se prikazuje na korisnikovom zaslonu organizirana je u poseban direktorij pod imenom *screens*. Manje komponente i cjeline koje pojedini zasloni koriste, nalaze se unutar direktorija *components*.

Pomoćne funkcije nalaze su unutar direktorija *utils*, dok se zajednički stilovi koji se koriste u većem broju zaslona nalaze u posebnom direktoriju *styles*. Na ovaj način omogućeno je jednostavno snalaženje unutar projekta i dodavanje komponenti te grafičkih i logičkih cjelina u primjerene direktorije. Također, ovakav pristup smanjuje broj linija koda u pojedinim datotekama i umanjuje potrebu za dupliciranjem programskog koda. Iako postoje brojni načini strukturiranja *React Native* projekata, opisana struktura prikladna je veličini *Properbooker* projekta. Na slici Slika 4.14 vidljiva je projektna struktura aplikacije zajedno s pripadajućim datotekama.



Slika 4.14 Projektna struktura *React Native* aplikacije

4.5.2. Autentikacija i autorizacija korisnika unutar aplikacije

Kako bi se korisnik uspješno mogao prijaviti u aplikaciju, potrebno je poslati zahtjev s korisničkim podacima mikroservisu za *JWT* autorizaciju i autentikaciju. Kao što je opisano u prethodnom poglavlju, ovaj će mikroservis kao odgovor vratiti važeći *JWT* token. *React Native* aplikacija zatim skladišti primljeni token u posebnu sigurnosnu strukturu *Secure Store*. Navedena struktura smanjuje mogućnost krađe generiranog tokena i pristupa osjetljivim podacima. Opisane radnje odvijaju se nakon pritiska tipke *Login* i izvršenja *login()* asinkrone funkcije. Funkcija, koristeći *Axios* biblioteku, izvršava *HTTP* poziv prema opisanom mikroservisu i u slučaju pozitivnog odgovora, skladišti dobiveni *JWT* token. S obzirom da predani korisnički podaci ne moraju nužno biti točni, potrebno je rukovati iznimkom u slučaju negativnog odgovora mikroservisa. Ako je

odgovor mikroservisa negativan, korisniku će se prikazati poruka o vrsti greške prilikom prijave. S obzirom na činjenicu da *HTTP* poziv i odgovor mikroservisa traju određeno vrijeme, potrebno je funkciju označiti ključnom riječju *async*. Na ovaj način omogućuje se *async-await* način rada te se označavanjem poziva funkcije ključnom riječju *await* prekida izvršavanje ostatka programskog koda do primitka odgovora mikroservisa i izvršavanja asinkrone funkcije. Naposljetku, u datoteci *App.ts*, sve se komponente definiraju kao djeca *AuthProvider* komponente kako bi mogle pristupiti stanju uskladištenog *JWT* tokena i pristupiti metodama definiranim u *AuthProvider* komponenti. Programski kod Programski kod 4.10 prikazuje način rada *login()* funkcije i rukovanje iznimkama, dok programski kod Programski kod 4.11 prikazuje način strukturiranja komponenti kako bi se moglo jednostavno rukovati autorizacijom i autentikacijom. Kako bi se moglo pristupiti ostalim zaslonima u aplikaciji, definiran je *Navigation Container*. Navedena struktura omogućuje slaganje zaslona prema načinu kretanja korisnika u aplikaciji. Ako je unutar *Navigation Container* strukture na prvo mjesto postavljen zaslon za prijavu i registraciju korisnika, taj će se zaslon prvi prikazati. Korištenje ove strukture također omogućuje i korištenje *useNavigation()* funkcije kojoj se može predati ime zaslona kojem se želi pristupiti i jednostavno otvoriti željeni zaslon.

```
const login = async (username: string, password: string) => {
       try {
           const data = {
               username: username,
               password: password,
           };
           const response = await axios.post(`${API_URL}signin`, data, {
               headers: {
                   "Content-Type": "application/json",
               },
           });

           if (response.status === 200) {
               setAuthState({
                   token: response.data.token,
                   authenticated: true,
               });
               axios.defaults.headers.common['Authorization'] = `Bearer
               ${response.data.token}`;

               await SecureStore.setItemAsync(TOKEN_KEY, response.data.token);

               return response.data;
```

```
            }
        } catch (error) {
            if(error.response.status===422) {
                return {error: true, msg: "Invalid username or password"};
            }
        }
    };
```

Programski kod 4.10 Implementacija funkcije za prijavu gostiju

```
export default function App() {
  return (

   <AuthProvider>
      <Layout></Layout>
   </AuthProvider>
  );
}

export const Layout = () =>
 // ... definiranje hijerarhije zaslona unutar aplikacije
```

Programski kod 4.11 Definiranje hijerarhije sa svrhom korištenja autorizacijskog konteksta u svim zaslonima aplikacije

4.5.3. Implementacija glavnog zaslona aplikacije i prikaz podataka o jedinicama za noćenje

Nakon prijave, korisnika se koristeći *useNavigation()* funkciju upućuje na glavni zaslon za prikaz podataka o jedinicama za noćenje. Kako bi se dohvatili podaci o jedinicama za noćenje, koristi se *Axios* biblioteka pomoću koje se šalje zahtjev za dohvat svih jedinica. S obzirom da je mikroservis za upravljanje jedinicama za noćenje zaštićen, potrebno je prvo dohvatiti *JWT* token iz *Secure Store* pohrane. S obzirom da će se ovaj postupak često koristiti, logika dohvaćanja tokena je izdvojena u posebnu pomoćnu funkciju prikazanu programskim kodom Programski kod 4.12. Dohvaćeni token se zatim stavlja u zaglavlje *HTTP* zahtjeva s prefiksom *Bearer* i šalje mikroservisu kako bi se autoriziralo korištenje njegovih metoda. Kao odgovor, mikroservis vraća niz *JSON* objekata s informacijama o jedinicama za noćenje. Kako bi se dohvaćeni podaci prikazali u obliku kartice, implementirana je *ApartmentCard* komponenta koja predane informacije

organizira u grafičku cjelinu zajedno s tipkama za dodavanje gostiju i generiranje *PDF* predračuna. Koristeći *React Native Flatlist* komponentu, dobiveni niz *JSON* objekata raspoređen je u pojedinačne kartice na temelju jedinstvenog imena jedinice za noćenje.

```
import * as SecureStore from 'expo-secure-store';

export const retrieveToken = async () => {
    try {
        const token = await SecureStore.getItemAsync('my_jwt');
        return token;
    } catch (error) {
        console.error('Error retrieving token:', error);
        return null;
    }
};
```

Programski kod 4.12 Logika dohvaćanja *JWT* tokena iz sigurne pohrane

4.5.4. Implementacija zaslona za prijavu gostiju

Pritiskom na tipku *Add guest* otvara se novi zaslon za prijavu gostiju u sustav. Kako bi se prikazale određene informacije poput imena ugostiteljske jedinice u koju se gost prijavljuje, takvi se podaci prosljeđuju *Add guest* komponenti i prikazuju u grafičkom sučelju. Unutar ove komponente nalazi se veliki broj polja za unos od kojih svako polje ima posebnu funkciju za praćenje stanja i provjeru validnosti unesenog teksta. Koristeći *expo-camera* biblioteku, na pritisak tipke za skeniranje identifikacijskog dokumenta otvara se sučelje za fotografiranje u kojemu se pritiskom na prikladnu tipku, koristi ili odbacuje snimljena fotografija. Fotografija se zatim pomoću *HTTP* zahtjeva i *API* ključa šalje *Google Cloud Vision* usluzi koja, koristeći algoritme strojnog učenja i računalnog vida, anotira tekst s fotografije. Anotirani tekst se zatim obrađuje na način da se na osnovu pojave znaka „<“ pronalazi dio teksta koji sadrži *MRZ* kod. Naposlijetku se dobiveni *MRZ* kod šalje *MRZ Parser* mikroservisu koji kao odgovor vraća *JSON* objekt s podacima o gostu. Pomoću spomenutih pomoćnih funkcija za validaciju i praćenje stanja tekstualnih polja, vrijednosti polja za unos popunjavaju se dobivenim informacijama. Prikaz *HTTP* zahtjeva i obrada prepoznatog teksta prikazana je programskim kodom Programski kod 4.13. Nakon uspješnog popunjavanja polja za unos, korisnik pritišće tipku *Register guest* koja, posljedično, šalje *Axios HTTP* zahtjev

emuliranom *e-visitor* sustavu. Nakon uspješne ili neuspješne prijave gosta, korisniku se prikazuje prikladna poruka pomoću *Android Toast* komponente.

```
const getParsedMRZ = async () =>{
        let fullTextAnnotation = await getDocumentTextDetection();
        let lastMRZChar = fullTextAnnotation.lastIndexOf('<');
        let MRZ = fullTextAnnotation.substring(lastMRZChar-92, lastMRZChar+1);

        const requestBody = {
            mrz: MRZ.toString().replace(/^\n/, '')
        };

        try {
            const response = await axios.post("https://00cb-46-188-225-
44.ngrok.io/api/parse", requestBody, {
                headers: {
                    "Content-Type": "application/json"
                },
            });
            handleAutomaticFieldInput(response.data);
        }
        catch (error){
            ToastAndroid.show("Error reading ID card, please try again",
ToastAndroid.SHORT);
        }
    }
```

Programski kod 4.13 Obrada prepoznatog teksta i pripadajući *HTTP* poziv *MRZ Parser* mikroservisu

4.5.5. Implementacija zaslona za generiranje *PDF* predračuna

Pritiskom na tipku *PDF estimation*, otvara se novi zaslon i sučelje slično sučelju *Add guest* zaslona. Ova komponenta također implementira mnoštvo polja za unos s pripadajućim funkcijama te tipku za generiranje predračuna u *PDF* formatu. Nakon što korisnik ispuni sva tekstualna polja, pritiskom na tipku za generiranje predračuna, uneseni podaci pretvaraju se u *JSON* objekt i pomoću *HTTP* zahtjeva šalju mikroservisu za generiranje predračuna u *PDF* formatu. Nakon primitka *PDF* dokumenta, dokument se mora pretvoriti u *base64* format koristeći *expo-filesystem* biblioteku i tranzitivno spremiti na uređaj u direktorij *cache*. Pomoću *React Native Sharing* biblioteke, otvara

se dijaloški prozor koji korisniku omogućuje pohranu ili dijeljenje generirane datoteke. S obzirom da su operacije dohvaćanja, slanja i spremanja *PDF* datoteka asinkrone, koristi se *async-await* način rada unutar *try-catch* bloka kako bi se izbjeglo stvaranje iznimke kojom nije pravilno rukovano. U slučaju da nije moguće generirati ili pohraniti *PDF* dokument, korisniku se prikazuje prikladna poruka, a podaci unutar polja za unos ostaju nepromijenjeni kako bi korisnik jednostavno mogao ažurirati netočne ili necjelovite podatke.

4.6. Analiza performansi implementiranog sustava

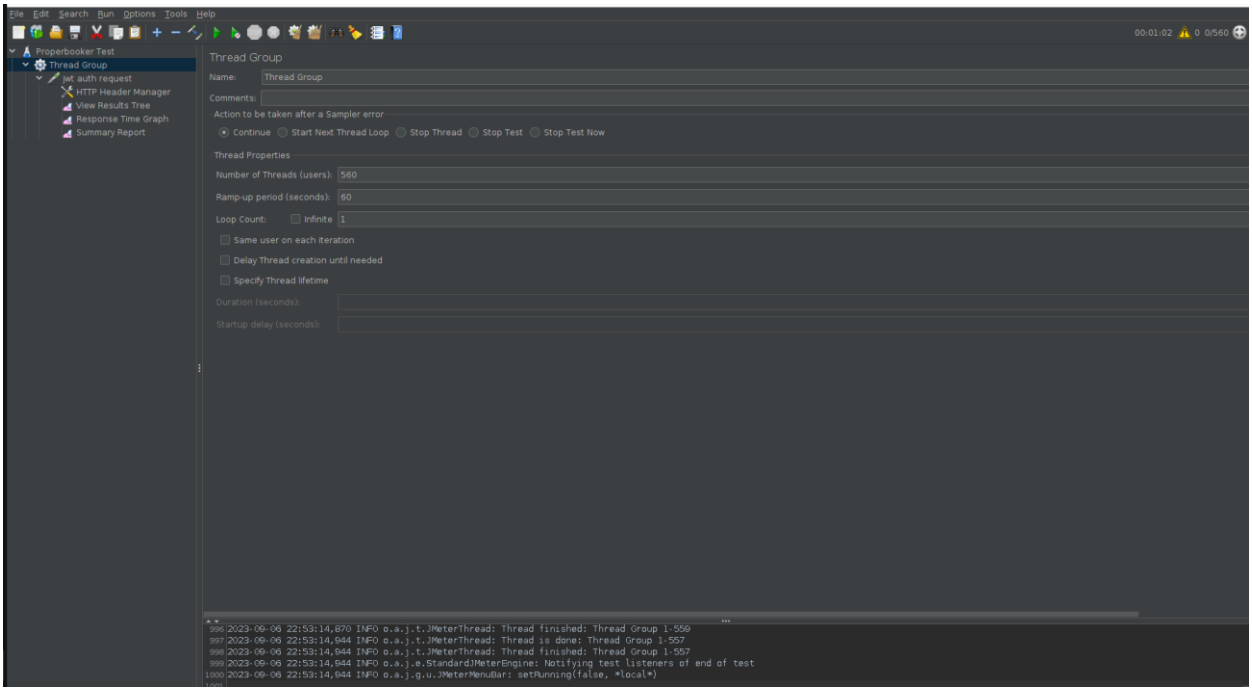
Ključan korak prilikom dizajna modularnog sustava s mikroservisnom *backend* arhitekturom jest testiranje performansi sustava s ciljem optimizacije programskog koda ili hardverske konfiguracije na kojoj se usluge programskog rješenja izvršavaju. Za produkcijsko testiranje sustava, potrebno je podići sustav na udaljeni poslužitelj i analizirati latenciju, propusnost *HTTP* zahtjeva i druge parametre iz perspektive korisnika. Prilikom analize rada *Properbooker* sustava, on će se testirati u razvojnoj okolinu na jednom poslužitelju prilikom čega će se očekivane vrijednosti parametara prilagoditi hardverskoj konfiguraciji i operacijskom sustavu.

4.6.1. Metodologija testiranja performansi sustava i *Apache JMeter* alat za automatizaciju testiranja

Kao što je prethodno spomenuto, testiranje performansi sustava izvršavat će se na jednom poslužitelju s hardverskom konfiguracijom prosječnog korisničkog računala. Specifikacije testnog poslužitelja uključuju *Intel Core i5-8250U* procesor, 16 GiB *DDR-4* radne memorije brzine 4800 MHz i *Ubuntu* operacijski sustav. S obzirom na hardversku konfiguraciju i potrošnju resursa operacijskog sustava, očekivani rezultati će u produkcijskoj okolini, koja će biti prilagođenija stvarnim uvjetima korištenja zbog distribuirane arhitekture i boljih specifikacija, biti puno bolji.

Testiranje performansi razvijenog sustava podrazumijeva slijedno ili paralelno izvršavanje velikog broja *HTTP* zahtjeva prema aplikacijsko-programskim sučeljima, zbog čega će se koristiti *Apache JMeter* alat za automatizaciju slanja *HTTP* zahtjeva. *Apache JMeter* je alat otvorenog koda koji omogućuje definiranje parametara *HTTP* zahtjeva, broja korisnika (niti) koji će izvršavati pozive,

broja *HTTP* poziva i mnoštva drugih parametara na osnovu kojih će se, pomoću integriranih modula za analizu performansi, generirati izvješće o radu sustava. Na slici Slika 4.15 može se vidjeti sučelje *Apache JMeter* alata zajedno s poljima za unos željenih parametara testiranja.



Slika 4.15 Sučelje *Apache JMeter* alata

Testiranje će se vršiti na način da će se postepeno povećavati broj paralelnih korisnika u određenom vremenskom okviru koristeći *Ramp-up* opciju unutar *Apache JMeter* alata i tražiti vrijednosti parametara na osnovu kojih će se donositi potrebni zaključci o potencijalnoj promjeni hardverske konfiguracije ili optimizacije programskog koda. S obzirom da se sustav sastoji od velikog broj mikroservisa, testirat će se usluga za autentikaciju korisnika koja predstavlja ulaznu točku u sustav. Bitno je napomenuti kako su tijekom testiranja ostale usluge također aktivne i koriste resurse poslužitelja. Također, prilikom testiranja će se promatrati i vrijednosti parametara prilikom paralelnog korištenja sustava u točno određenom trenutku te se neće koristiti *round-robin* algoritam za raspodjelu *HTTP* zahtjeva jer je sustav pokrenut na isključivo jednom poslužitelju zbog čega će *HTTP* zahtjevi uvijek biti obrađeni na istom računalu.

4.6.2. Rezultati testiranja i analiza vrijednosti parametara

Kako bi se pravilno analizirao rad sustava, bitno je definirati očekivane vrijednosti parametara testiranja. S obzirom da će se prilikom ovog testiranja mjeriti prosječna brzina odziva aplikacijsko-programskog sučelja i njegova propusnost, vrijednost brzine odziva bi trebala biti između 100 i 1000 milisekundi kako bi se korisniku omogućio nesmetan rad. U tablici na slici Slika 4.16 prikazani su rezultati testiranja sustava s pet paralelnih korisnika prilikom čega je prosječna brzina odziva sustava iznosila 436 milisekundi što je razmjerno hardverskoj konfiguraciji poslužitelja. Povećanjem broja paralelnih korisnika, vrijednost brzine odziva raste te se može zaključiti kako je trenutna hardverska konfiguracija u mogućnosti poslužiti 15 paralelnih korisnika bez osjetne razlike u brzini rada sustava što je vidljivo u tablici na slici Slika 4.17. Iako je prosječna vrijednost brzine odziva, vidljiva u stupcu *Average*, prihvatljiva, neki će korisnici ipak osjetiti veću latenciju od oko 1200 milisekundi (stupac *Max*) što je rezultat očekivanih odstupanja u radu sustava.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
jwt auth req...	5	436	0	487	42.37	0.00%	10.3/sec
TOTAL	5	436	0	487	42.37	0.00%	10.3/sec

Slika 4.16 Vrijednosti parametara testiranja na osnovu pet paralelnih korisnika

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
jwt auth req...	15	933	0	1242	197.28	0.00%	11.4/sec
TOTAL	15	933	0	1242	197.28	0.00%	11.4/sec

Slika 4.17 Vrijednosti parametara testiranja na osnovu 15 paralelnih korisnika

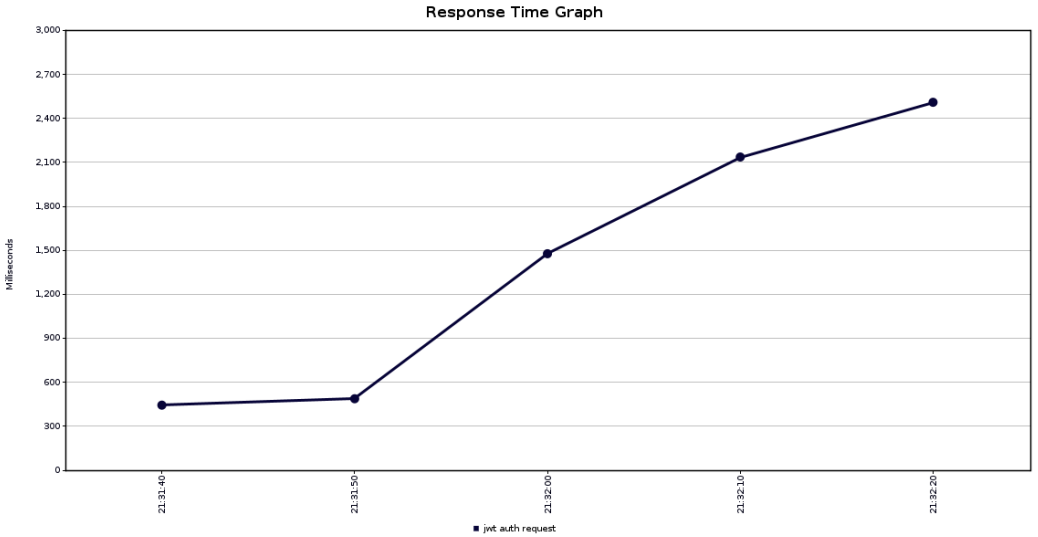
Kako bi se testirao rad sustava prilikom velikih opterećenja, pomoću *Ramp-up* značajke *Apache JMeter* alata, sustav je testiran prilikom postepenog povećanja broja paralelnih korisnika do 500 i 1000 u vremenskom okviru od 50 sekundi. U tablicama na slici Slika 4.18 i Slika 4.19 prikazani su rezultati testiranja iz kojih se mogu vidjeti zadovoljavajući rezultati i nizak postotak pogrešaka s obzirom na broj korisnika i hardverske specifikacije poslužitelja. Također, na slikama Slika 4.20 i Slika 4.21 prikazani su grafovi vremena odziva prilikom *Ramp-up* testiranja koji prikazuju rast latencije razmjeran broju paralelnih korisnika.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
jwt auth req...	500	1418	0	3533	876.18	0.00%	9.6/sec
TOTAL	500	1418	0	3533	876.18	0.00%	9.6/sec

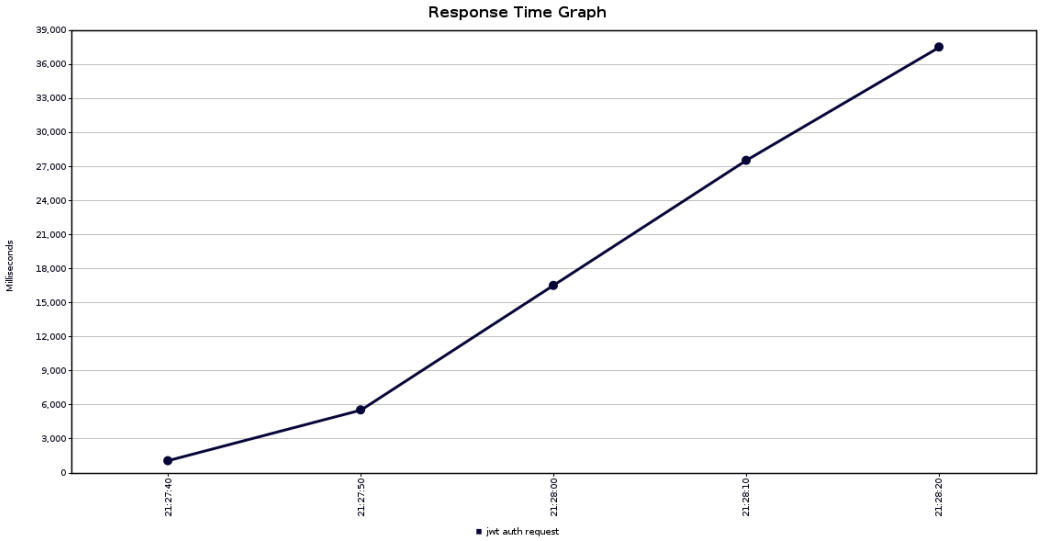
Slika 4.18 Vrijednosti parametara testiranja tijekom postupnog povećanja broja korisnika do 500

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
jwt auth req...	1000	26105	0	51749	15320.50	1.30%	9.8/sec
TOTAL	1000	26105	0	51749	15320.50	1.30%	9.8/sec

Slika 4.19 Vrijednosti parametara testiranja tijekom postupnog povećanja broja korisnika do 1000



Slika 4.20 Graf vremena odziva sustava tijekom postupnog povećanja broja korisnika do 500



Slika 4.21 Graf vremena odziva sustava tijekom postupnog povećanja broja korisnika do 1000

4.6.3. Zaključak analize rada sustava

S obzirom na dobivene rezultate tijekom testiranja, može se zaključiti kako sustav uspješno poslužuje do 15 paralelnih korisnika u razumnom vremenskom okviru dok je za uspješan rad s većim brojem korisnika potrebno povećati računalne resurse i iskoristiti implementiranu mogućnost raspodjele *HTTP* zahtjeva na više instanci mikroservisa. Iz razmjernog rasta latencije *HTTP* zahtjeva, moguće je zaključiti kako ne treba optimizirati programsko rješenje već pravilno iskoristiti mikroservisnu arhitekturu aplikacije i raspodijeliti instance mikroservisa na odvojene, hardverskom konfiguracijom prilagođenije, poslužitelje.

4.7. Ograničenja i moguća proširenja razvijenog programskog rješenja

Properbooker aplikacija na dostatnoj razini pokriva probleme iz domene sigurnosti, horizontalne skalabilnosti i stabilnosti za slučaj iznimno velikog povećanja broja korisnika. Unatoč robusnosti ove arhitekture, postoje dijelovi aplikacije i pratećeg *backend* sustava koji se mogu unaprijediti prije puštanja u produkciju. Iz sigurnosne perspektive, unaprjeđenje se može pronaći u obliku implementacije dodatnog modula za sprječavanje *denial-of-service* napada. Također, mogu se integrirati i usluge za statičku validaciju koda i sigurnosnih propusta poput *Sonarqube* i *SNYK* dodataka te omogućiti automatsko ažuriranje programskih ovisnosti i paketa koji sadrže

direktne ili tranzitivne sigurnosne propuste. Za poboljšanje korisničkog iskustva, moguće je dodati poseban zaslon koji, koristeći podatke dobivene s *Airbnb* i *Booking.com* aplikacijsko-programskih sučelja, sinkronizira kalendare zauzetosti jedinica za noćenje i skupno prikazuje rezervacije sa svih pokrivenih platformi. Zbog naplate korištenja aplikacijsko-programskih sučelja, opisana značajka nije implementirana u ovoj inačici sustava. Osim poboljšanja sigurnosti i povećanja broja značajki, *Properbooker* aplikacija mogla bi se poboljšati promjenom modela strojnog učenja za očitavanje osobnih dokumenata. Iako *Google Cloud Vision* model izrazito dobro anotira tekst neovisno o osvjetljenju i kvaliteti kamere uređaja, ova usluga nažalost ne pokriva *GDPR* naputke za rukovanje osjetljivim identifikacijskim podacima te koristi dobivene podatke za treniranje *Google Cloud Vision* i drugih modela strojnog učenja. Osim *Google Cloud Vision* modela, postoje drugi modeli koji prate *GDPR* i funkcioniraju na sličan način uz marginalno slabije performanse, no zahtijevaju dogovor plana korištenja i podrazumijevaju višu cijenu *HTTP* zahtjeva. Također, dodavanje dodatka za praćenje zdravlja mikroservisa i njihovih performansi zajedno s dodatkom za obavještavanje administratora o padu određenog mikroservisa ili velikog broja *HTTP* zahtjeva, olakšalo bi proces održavanja sustava u produkcijskoj okolini.

ZAKLJUČAK

Izrada mikroservisne aplikacije utemeljene na web tehnologijama koja je kompatibilna s različitim tipovima uređaja, složen je i višeslojan proces koji zahtijeva intenzivnu analizu funkcionalnih i nefunkcionalnih zahtjeva aplikacije prije samog početka razvoja. *Properbooker* aplikacija koristi činjenicu da postoji više od 100 000 privatnih iznajmljivača u Republici Hrvatskoj i jednak broj potencijalnih korisnika kao argument za korištenje modularnog mikroservisnog sustava s mogućnošću, jednostavnog i korisniku neprimjetnog, horizontalnog skaliranja. Korištenjem tehnologija temeljenim na modularnom ili komponentnom načinu razvoja kao što su *Spring Boot* i *React Native* te pravilnim strukturiranjem programskog koda, postignuta je jednostavna proširivost i omogućeno je dinamičko dodavanje novih mikroservisa, značajki i dodataka za praćenje performansi i sigurnosti sustava u cijelosti. Implementacijom sigurnosnog sustava koji koristi *JWT* autorizacijske tokene, osigurana je jednostavna zaštita *REST* komunikacijskih točki i ubrzan je proces zaštite svih mikroservisa i modula koji će potencijalno biti dio sustava u budućnosti. Opisana aplikacija nudi brojne značajke koje su potrebne za ubrzanje i pojednostavljenje procesa prijave gostiju i izrade predračuna te na taj način iznajmljivačima omogućuje uštedu vremena i očuvanje reputacije prilikom primitka gostiju u jedinice za noćenje. Korištenje ove aplikacije također može smanjiti probleme skladištenja osobnih identifikacijskih podataka gostiju u lokalnoj pohrani i povećati postotak iznajmljivača koji prate *GDPR* naputke za skladištenje i obradu istih. Daljnjim razvojem aplikacije uz suradnju nadležnih državnih institucija vezanih uz turizam i proširivanjem njene funkcionalnosti može se stvoriti slojeviti programski proizvod za automatizaciju gotovo cijelog administrativnog procesa iznajmljivanja jedinica za noćenje.

SAŽETAK

Ovaj diplomski rad opisuje proces razvoja moderne višeplatformske *React Native* aplikacije za upravljanje jedinicama za noćenje te pripadajuće pozadinske mikroservisne arhitekture. Rad također prikazuje analizu korisničkih, tehničkih i legalnih problema postojećih sustava za prijavu gostiju te, koristeći radne okvire temeljene na modularnim i komponentnim tehnologijama, nudi razvojnu inačicu programskog rješenja opisanih problema. Osim toga, prikazuju se koraci razvoja i implementacije pojedinih programskih i grafičkih cjelina unutar aplikacije, proces odabira prikladnih tehnologija i arhitekture te pravilno strukturiranje programskog koda. Rad detaljno opisuje korisničko iskustvo razvijene aplikacije i argumentira odabir opisanih tehnologija za njen razvoj. Naposlijetku, opisuju se moguća poboljšanja razvijene aplikacije iz perspektive sigurnosti, skalabilnosti i održavanja.

KLJUČNE RIJEČI: mikroservisna arhitektura, turizam, *React Native*, *Spring Boot*, višeplatformske aplikacije

ABSTRACT

AUTOMATION OF ADMINISTRATIVE ACTIVITIES EXEMPLIFIED BY THE DEVELOPMENT OF A MICROSERVICE-BASED WEB APPLICATION FOR ACCOMMODATION UNIT MANAGEMENT

This master's thesis describes the development process of a modern multiplatform microservice application for accommodation unit management using modular and component-based technologies. It also follows the steps needed to identify both user experience and technology related problems with existing accommodation unit management systems and exemplifies this process by offering a solution in the form of a *React Native* application with a *Spring Boot* microservice-based backend. The thesis also describes the chosen development technologies and explains the implementation process of logical and graphical elements in the application. Conclusively, the thesis offers several possible improvements to the developed product in the areas of security, scalability and system maintenance.

KEYWORDS: microservice architecture, tourism, *React Native*, *Spring Boot*, multiplatform applications

LITERATURA

[1] „E-visitor“ [online]. Dostupno na: https://www.htz.hr/hr-HR/projekti-i-potpore/evisitor. [Pristupljeno: 31.8.2023.].
[2] „Prijavi turiste“ [online]. Dostupno na: https://www.prijavituriste.com.hr/. [Pristupljeno: 8.8.2023.].
[3] „mVisitor“ [online]. Dostupno na: https://mvisitor.hr. [Pristupljeno: 14.8.2023.].
[4] „E-checkin“ [online]. Dostupno na: https://echeckin.hr. [Pristupljeno: 31.8.2023.].
[5] A., Hartl, C., Arth, S., Dieter, „Real-time Detection and Recognition of Machine-Readable Zones with Mobile Devices“, Graz, 2015.
[6] Microsoft, .„NET 8“ [online]. Dostupno na: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8. [Pristupljeno: 23.8.2023.].
[7] C., Walls, „walls“, u *Spring in Action*, Manning Publications, 2022, str. 7–20.
[8] B., Evans J., J., Clark, D., Fianagan, u *Java in a Nutshell*, O'Reilly Media Inc., 2023, str. 6–18.
[9] A., Kuttig Benedikt, u *Professional React Native*, Packt Publishing, 2022, str. 4–14.
[10] R., Rose, u *Flutter & Dart Cookbook*, O'Reilly Media Inc., 2022, str. 2–7.
[11] S., Newman, u *Building Microservices*, O'Reilly Media Inc., 2021, str. 2–8.
[12] S., Newman, u *Monolith to Microservices*, O'Reilly Media Inc., 2019, str. 3–12.
[13] S., Kane P., K., Matthias, u *Docker Up & Running*, O'Reilly Media Inc., 2023, str. 1–9.
[14] V., Servile, „servillr“, u *Canary Deployment*, O'Reilly Media Inc., 2023, str. 11–13.
[15] N., Kebbani, P., Tylenda, R., McKendrick, u *The Kubernetes Bible*, Packt Publishing, 2022, str. 4–8.
[16] „GDPR“ [online]. Dostupno na: https://gdpr-info.eu. [Pristupljeno: 12.8.2023.].

ŽIVOTOPIS

Autor ovog diplomskog rada, Nikola Pavković, rođen je 28. studenog 1998. u Osijeku. Upisuje Prirodoslovno-matematičku gimnaziju u Osijeku 2013. godine nakon čega 2017. upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija na Sveučilištu Josipa Jurja Strossmayera u Osijeku, smjer računarstvo. Uz izvršavanje studentskih obaveza, postaje predsjednik IEEE studentskog ogranka Osijek u sklopu kojega izvršava administrativne i organizacijske poslove. Tijekom studiranja obavlja posao predavača brojnih predmeta iz područja računarstva u privatnoj tvrtki. Nakon upisa na diplomski studij računarstva, smjer informacijske i podatkovne znanosti, odrađuje praksu u tvrtki Base58 d.o.o nakon čega nastavlja praksu i počinje raditi u tvrtki Infobip d.o.o na području razvoja mikroservisnih programskih rješenja u telekomunikacijskoj industriji.

_____

Potpis autora

PRILOZI

Programski kod aplikacije moguće je pronaći na sljedećim poveznicama na *Github* repozitorije:

Eureka Discovery mikroservis: nikanberko/discovery-service-properbooker (github.com)

Properbooker Apartment Management mikroservis: nikanberko/apartmentManagement: Apartment management application for properbooker react native app (github.com)

Properbooker JWT Auth mikroservis: nikanberko/properbookerJWTAuth: JWT authentication for the properbooker react native app and microservices (github.com)

API Gateway mikroservis: https://github.com/nikanberko/api-gateway-properbooker

Properbooker React Native aplikacija: nikanberko/properbookerFrontend (github.com)
Properbooker MOCK e-visitor mikroservis: https://github.com/nikanberko/MockEvisitor

Properbooker PDF Generator mikroservis: https://github.com/nikanberko/pdfgenerator

Properbooker MRZ Parser mikroservis: https://github.com/nikanberko/mrz-spring