UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Momčilo Knežević

PROJEKTOVANJE I IMPLEMENTACIJA VEB APLIKACIJE ZA UPRAVLJANJE NEKRETNINAMA I PROCESOM IZDAVANJA

master rad

Beograd, 2026.

Mentor:

dr Mirjana Maljković Ružičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Vesna Marinković, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Staša Vujičić Stanković, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Projektovanje i implementacija veb aplikacije za upravljanje nekretninama i procesom izdavanja

Rezime: Ovaj master rad bavi se proučavanjem kontrole pristupa i segmentacije podataka u višekorisničkim veb sistemima. Posebna pažnja posvećena je situacijama u kojima prava korisnika ne zavise samo od njihove uloge, već i od poslovnog odnosa prema konkretnom resursu i eksplicitno dodeljenih prava. Kao praktični okvir za razmatranje ovog problema projektovana je i implementirana veb aplikacija za upravljanje nekretninama i procesom izdavanja, koja podržava direktno upravljanje nekretninama od strane vlasnika, kao i upravljanje preko agencije, uz definisane odnose između vlasnika, agencija i stanara. Za perzistenciju podataka kombinovani su relacioni i dokumentni model, uz korišćenje sistema za upravljanje bazama podataka (PostgreSQL-a i MongoDB-a) i GridFS-a u skladu sa karakteristikama podataka koji se čuvaju. Razvijena aplikacija obuhvata upravljanje nekretninama, procesom izdavanja, finansijskim evidencijama, zahtevima i dokumentacijom, uključujući generisanje ugovora i izveštaja. U radu su prikazani arhitektura sistema, organizacija serverske aplikacije, model podataka, najvažnije implementirane funkcionalnosti, kao i način testiranja i ograničenja razvijenog rešenja.

Ključne reči: veb aplikacija, upravljanje nekretninama, višekorisnički sistem, autorizacija, segmentacija podataka, PostgreSQL, MongoDB, GridFS

Sadržaj

Glava 1

Uvod

Razvoj informacionih tehnologija značajno je uticao na digitalizaciju velikog broja poslovnih procesa, omogućavajući efikasnije upravljanje podacima, bolju organizaciju rada i jednostavniju komunikaciju između različitih učesnika u sistemu. Jedna od oblasti u kojoj postoji izražena potreba za primenom savremenih veb tehnologija jeste upravljanje nekretninama i procesom njihovog izdavanja. Taj proces obuhvata veliki broj međusobno povezanih aktivnosti, kao što su evidencija nekretnina, upravljanje ugovorima, praćenje troškova, komunikacija između učesnika, podnošenje i obrada zahteva, kao i generisanje različite dokumentacije. Objedinjavanje ovih funkcionalnosti u okviru jedinstvenog informacionog sistema doprinosi većoj preglednosti, smanjenju mogućnosti greške i efikasnijem upravljanju celokupnim procesom.

Poseban izazov prilikom projektovanja ovakvih sistema predstavlja činjenica da u procesu izdavanja nekretnina učestvuje više različitih tipova korisnika, pre svega vlasnici nekretnina, agencije za izdavanje i stanari. Iako svi oni koriste iste resurse sistema, njihova prava pristupa, obim vidljivosti podataka i mogućnosti izmene značajno se razlikuju. Zbog toga je neophodno obezbediti pouzdan mehanizam kontrole pristupa koji će omogućiti deljenje podataka između korisnika samo u obimu koji odgovara njihovoj ulozi i konkretnom odnosu prema resursu, uz istovremeno očuvanje privatnosti i izolacije podataka.

Na tržištu postoje različita softverska rešenja namenjena upravljanju nekretninama, koja se razlikuju prema ciljnoj grupi i skupu funkcionalnosti. Rešenja kao što su RELPER, ESTATE Sistem i Dimedia nekretnine prvenstveno su usmerena na poslovne procese povezane sa evidencijom nekretnina i klijenata, dok Vivio obuhvata komunikaciju i usluge namenjene vlasnicima nekretnina, stanarima i upravnicima zgrada[7, 10, 48, 49]. Javno dostupne informacije o navedenim rešenjima ne pruža-

ju dovoljno detalja za pouzdano poređenje njihovih internih modela autorizacije. U ovom radu domen upravljanja nekretninama koristi se kao praktični okvir za proučavanje kontrole pristupa u kojoj odluka zavisi od korisničke uloge, odnosa prema konkretnom resursu i eksplicitno dodeljenih prava.

Predmet ovog master rada jeste kontrola pristupa i segmentacija podataka u višekorisničkim veb sistemima. Cilj rada je proučavanje načina na koji se odluka o pristupu može zasnovati ne samo na ulozi korisnika, već i na njegovom konkretnom odnosu prema resursu, kao i načina na koji se na osnovu takve odluke može odrediti obim podataka dostupan korisniku. Kao praktični okvir za proučavanje ovog problema projektovana je i implementirana veb aplikacija **Rentivo** namenjena centralizovanom upravljanju nekretninama i procesom njihovog izdavanja. Razvijeni sistem podržava različite modele izdavanja nekretnina, omogućava upravljanje korisničkim nalozima i njihovim ulogama, evidenciju nekretnina, ugovora, troškova i zahteva, kao i generisanje ugovora i izveštaja u PDF formatu. Posebna pažnja posvećena je projektovanju mehanizama autorizacije i segmentacije podataka, kako bi se obezbedilo da svaki korisnik pristupa isključivo onim resursima za koje ima odgovarajuća ovlašćenja.

Sistem je realizovan kao klijent-server veb aplikacija sa jasno razdvojenim klijentskim i serverskim delom. Klijentska strana realizovana je korišćenjem biblioteke React, dok je serverska strana implementirana kao REST servis u programskom jeziku Python, korišćenjem razvojnog okvira FastAPI. Za skladištenje podataka primenjena je kombinacija relacione i nerelacione baze podataka, pri čemu se relaciona baza koristi za čuvanje strukturiranih podataka, dok se nerelaciona baza koristi za podatke promenljive strukture i dokumentno orijentisane sadržaje.

U okviru rada analizirani su i primenjeni modeli kontrole pristupa pogodni za višekorisničko okruženje. Posebno su razmotreni mehanizmi koji omogućavaju deljenje resursa između različitih korisnika uz različite nivoe vidljivosti informacija i prava izmene. U razvijenom sistemu autorizaciona odluka zasniva se na korisničkoj ulozi, odnosu prema konkretnom resursu i eksplicitno dodeljenim pravima, dok se utvrđeni obim pristupa koristi kao osnov za segmentaciju podataka na serverskoj strani. Na taj način razvijena aplikacija predstavlja praktični okvir za primenu i proveru razmatranog pristupa kontroli pristupa u višekorisničkom okruženju.

Ostatak rada organizovan je tako da su u drugoj glavi predstavljene tehnologije i alati korišćeni prilikom razvoja sistema. U trećoj glavi opisana je arhitektura aplikacije i implementacija njenih najvažnijih funkcionalnosti, sa posebnim osvrtom na

model podataka, mehanizme autentifikacije i autorizacije, kao i način realizacije poslovne logike. Na kraju rada dat je zaključak, u kome su sumirani ostvareni rezultati i navedeni mogući pravci daljeg razvoja sistema.

Glava 2

Tehnologije korišćene u razvoju sistema

2.1 Programski jezik Python

Python je programski jezik visokog nivoa opšte namene, čiji je razvoj započeo Guido van Rossum početkom devedesetih godina XX veka. Njegova popularnost zasniva se na jednostavnoj sintaksi, velikoj čitljivosti izvornog koda i bogatom ekosistemu biblioteka koje omogućavaju razvoj složenih informacionih sistema uz relativno malu količinu koda [34].

Programski jezik Python predstavlja pogodan izbor za implementaciju aplikacija koje zahtevaju modularnu organizaciju, dobru održivost i mogućnost daljeg proširenja funkcionalnosti. Upravo iz tih razloga odabran je kao osnovni programski jezik za implementaciju serverskog dela sistema koji je predmet ovog master rada, dok će njegove konkretno primenjene mogućnosti biti detaljnije opisane u nastavku poglavlja.

Objektno orijentisano i asinhrono programiranje

Objektno orijentisano programiranje (engl. Object-Oriented Programming – OOP) predstavlja jednu od najznačajnijih programskih paradigmi savremenog razvoja softvera. Osnovna ideja ovog pristupa jeste modelovanje problema kroz objekte koji opisuju entitete iz realnog sveta ili apstraktne koncepte sistema. Svaki objekat objedinjuje podatke i operacije koje nad njima mogu biti izvršene, čime se postiže veća modularnost, preglednost i mogućnost ponovne upotrebe programskog koda [22].

Primena objektno orijentisanog pristupa posebno dolazi do izražaja kod razvoja složenijih informacionih sistema, gde veliki broj međusobno povezanih komponenti zahteva jasnu organizaciju i razdvajanje odgovornosti. Pravilnim modelovanjem domena problema postiže se veća održivost sistema, jednostavnije testiranje i lakša nadogradnja postojećih funkcionalnosti. Iz tog razloga se ovaj pristup često primenjuje u razvoju savremenih poslovnih aplikacija, nezavisno od programskog jezika ili razvojnog okvira koji se koristi [22].

Pored objektno orijentisanog pristupa, savremene serverske aplikacije sve češće koriste asinhrono programiranje kao način efikasnijeg izvršavanja operacija koje podrazumevaju čekanje spoljašnjih resursa, kao što su baze podataka, mrežni servisi ili sistemi datoteka. Za razliku od sinhronog izvršavanja, kod koga jedna operacija blokira izvršavanje narednih dok se ne završi, asinhroni model omogućava da se tokom čekanja na završetak ulazno-izlazne (I/O) operacije paralelno obrađuju drugi zadaci [1, 43].

Python od verzije 3.5 pruža ugrađenu podršku za asinhrono programiranje uvođenjem ključnih reči *async* i *await*, kao i biblioteke asyncio, koja omogućava organizaciju konkurentnog izvršavanja velikog broja zadataka unutar jednog procesa. Ovakav model je posebno pogodan za razvoj mrežnih aplikacija i REST servisa, gde značajan deo vremena izvršavanja predstavlja čekanje odgovora spoljašnjih servisa [13, 16, 43]. Asinhrono programiranje ne predstavlja zamenu za višenitno ili višestruko procesiranje, već drugačiji model organizacije izvršavanja programa. Njegova osnovna prednost ogleda se u efikasnijem rukovanju I/O operacijama, dok se za izrazito procesorski zahtevne zadatke i dalje koriste drugi mehanizmi paralelizacije. Pravilnim izborom između sinhronog i asinhronog pristupa moguće je ostvariti optimalan odnos između jednostavnosti implementacije i performansi sistema [1].

Python kao platforma za razvoj serverskih veb aplikacija

Razvoj serverskih veb aplikacija podrazumeva obradu korisničkih zahteva, izvršavanje poslovne logike, komunikaciju sa sistemima za upravljanje bazama podataka i razmenu podataka sa drugim informacionim sistemima, zbog čega programski jezik namenjen ovoj nameni treba da omogući dobru organizaciju koda, jednostavno održavanje i podršku za razvoj savremenih veb sistema. Programski jezik Python se zbog svoje fleksibilnosti široko koristi u ovoj oblasti i programerima nudi veliki izbor razvojnih okvira različite namene i složenosti. Neki okviri pružaju kompletno razvojno okruženje sa velikim brojem ugrađenih funkcionalnosti, dok drugi prate mi-

nimalistički pristup i ostavljaju veću slobodu pri organizaciji arhitekture aplikacije [34].

Kombinacija objektno orijentisanog i asinhronog programiranja posebno dolazi do izražaja upravo u razvoju serverskih veb aplikacija: savremene serverske aplikacije obrađuju veliki broj istovremenih korisničkih zahteva i oslanjaju se na intenzivnu komunikaciju sa sistemima za upravljanje bazama podataka i drugim mrežnim servisima, pa način na koji aplikacija obrađuje I/O operacije značajno utiče na njene performanse. Za razvoj asinhronih Python veb aplikacija koristi se ASGI (engl. Asynchronous Server Gateway Interface), standard koji definiše komunikaciju između aplikacije i serverskog okruženja [1]. Za razliku od starijeg WSGI standarda, prvenstveno namenjenog sinhronim aplikacijama, ASGI podržava asinhrono izvršavanje, dugotrajne konekcije i savremene komunikacione protokole, što ga čini pogodnim za razvoj modernih veb servisa [1].

Razvojni okviri zasnovani na ASGI standardu omogućavaju implementaciju REST servisa uz korišćenje asinhronog izvršavanja, automatske validacije podataka i generisanja dokumentacije aplikacije. Jedan od najpoznatijih predstavnika ove grupe jeste FastAPI, koji je zbog svoje jednostavnosti, performansi i dobre integracije sa Python ekosistemom odabran za implementaciju serverskog dela sistema razvijenog u okviru ovog rada. Njegove karakteristike i mogućnosti biće detaljnije opisane u narednom poglavlju.

2.2 Razvojni okvir FastAPI

Izbor odgovarajućeg okvira za izradu serverskog dela veb aplikacija zavisi od zahteva projekta, organizacije aplikacije i načina implementacije poslovne logike. U okviru ovog rada za razvoj serverskog dela sistema odabran je FastAPI, savremeni razvojni okvir namenjen implementaciji REST servisa u programskom jeziku Python [11, 13].

Okvir FastAPI je objavljen 2018. godine sa ciljem da objedini jednostavnost razvoja, visoke performanse i aktuelne mogućnosti Python ekosistema. Za razliku od pojedinih razvojnih okvira koji objedinjuju veliki broj funkcionalnosti u jedinstvenom okruženju, FastAPI prati minimalistički pristup. Njegova uloga prvenstveno je organizacija transportnog sloja aplikacije, dok su poslovna logika, modeli domena i pristup podacima smešteni u zasebne slojeve sistema. Takav način organizacije olakšava održavanje aplikacije, pojednostavljuje testiranje i doprinosi jasnom razdvaja-

nju odgovornosti između pojedinih delova sistema [21]. U nastavku će biti prikazane osnovne karakteristike okvira FastAPI koje su od značaja za razvoj serverskih veb aplikacija, nakon čega će biti opisani način implementacije REST servisa, kao i mehanizmi validacije podataka i automatskog generisanja dokumentacije API-ja.

Okvir FastAPI

Kao razvojni okvir otvorenog koda, FastAPI se arhitekturno zasniva na ASGI standardu, što omogućava asinhronu obradu zahteva i jednostavnu integraciju sa serverskim okruženjima. Za pokretanje aplikacije najčešće se koristi Uvicorn, ASGI server koji prihvata dolazne HTTP zahteve, prosleđuje ih aplikaciji i vraća odgovore klijentima [47].

Jedna od karakteristika okvira FastAPI jeste oslanjanje na sistem tipova programskog jezika Python. Tipovi podataka koji se navode prilikom definisanja funkcija ne služe isključivo boljoj čitljivosti izvornog koda, već predstavljaju osnov za validaciju ulaznih podataka, serijalizaciju odgovora i automatsko generisanje OpenAPI specifikacije. Na taj način jedan izvor informacija istovremeno opisuje strukturu podataka i ponašanje API-ja, čime se smanjuje mogućnost nastanka nekonzistentnosti između implementacije i dokumentacije [11].

Okvir FastAPI dodatno pruža jednostavan mehanizam organizacije aplikacije kroz rutere, modele podataka i mehanizam ubrizgavanja zavisnosti (engl. dependency injection), pri čemu svaka od ovih celina ima jasno definisanu odgovornost. Zahvaljujući tome, razvojni okvir ne nameće način implementacije poslovne logike niti organizaciju domena aplikacije, što je posebno pogodno za složenije poslovne aplikacije kod kojih je važno očuvati jasnu granicu između slojeva arhitekture [21].

Još jedna važna karakteristika okvira FastAPI odnosi se na upravljanje životnim ciklusom aplikacije. Tokom pokretanja aplikacije često je potrebno uspostaviti konekcije prema bazama podataka, inicijalizovati zajedničke resurse ili izvršiti određene pripremne operacije pre prihvatanja prvih korisničkih zahteva. Sa druge strane, prilikom gašenja aplikacije neophodno je pravilno osloboditi korišćene resurse i zatvoriti otvorene konekcije. FastAPI za ovu namenu koristi *lifespan* mehanizam, kojim se na jedinstvenom mestu definišu aktivnosti koje se izvršavaju pri pokretanju i gašenju aplikacije. Time se upravljanje životnim ciklusom sistema čini preglednijim i jednostavnijim za održavanje [11].

U okviru razvijenog sistema FastAPI je korišćen kao transportni sloj između klijentske aplikacije realizovane korišćenjem biblioteke React i poslovne logike im-

plementirane u programskom jeziku Python, pri čemu ruteri prihvataju HTTP zahteve, prosleđuju ih odgovarajućim servisima i vraćaju rezultate klijentu. Način na koji je ovo razdvajanje sprovedeno detaljnije je opisan u narednom potpoglavlju, u kontekstu implementacije REST servisa [21].

FastAPI se dodatno izdvaja automatskim generisanjem interaktivne dokumentacije API-ja, čime se pojednostavljuje razvoj i testiranje aplikacije, naročito u situacijama kada isti API koristi više klijentskih aplikacija ili kada na projektu učestvuje veći broj programera. Mehanizam generisanja dokumentacije, kao i validacija podataka na kojoj se on zasniva, detaljnije će biti opisani u jednom od narednih potpoglavlja.

Razvoj REST servisa

Razvoj savremenih veb aplikacija u najvećoj meri zasniva se na arhitekturi u kojoj su klijentski i serverski deo sistema međusobno razdvojeni i komuniciraju putem mrežnih protokola. U takvom okruženju REST (engl. Representational State Transfer) predstavlja najčešće korišćen arhitektonski stil za implementaciju programskih interfejsa namenjenih razmeni podataka između različitih sistema. REST nije protokol, već skup principa koji definišu način organizacije resursa, komunikacije i obrade zahteva između klijenta i servera [13].

Osnovna ideja REST arhitekture jeste da se funkcionalnosti sistema predstave kroz resurse kojima se pristupa jedinstvenim URL adresama. Operacije nad resursima realizuju se korišćenjem standardnih HTTP metoda. Metoda GET koristi se za preuzimanje podataka, POST za kreiranje novih resursa, PUT i PATCH za njihovu izmenu, dok je metoda DELETE namenjena uklanjanju resursa. Dodatno, server za svaki zahtev vraća odgovarajući HTTP statusni kod koji klijentu daje informaciju o ishodu izvršene operacije. Na taj način komunikacija između klijenta i servera ostaje jednostavna, jednoznačna i nezavisna od tehnologije korišćene na obe strane sistema [39].

FastAPI prati principe REST arhitekture kroz jasno definisanu organizaciju ruta. Svaka grupa funkcionalnosti može biti izdvojena u poseban ruter (APIRouter), čime se API deli na logičke celine, kao što su upravljanje korisnicima, nekretninama, ugovorima ili zahtevima. Ovakav pristup doprinosi preglednosti projekta i olakšava održavanje aplikacije, naročito kada broj dostupnih funkcionalnosti postane veći [11].

FastAPI takođe pruža ugrađenu podršku za mehanizam ubrizgavanja zavisnosti. Umesto da ruteri neposredno prave objekte neophodne za obradu zahteva, potrebne zavisnosti se prosleđuju putem posebnih funkcija koje FastAPI automatski poziva tokom obrade zahteva. Ovakav način rada doprinosi manjoj povezanosti između pojedinih delova sistema i olakšava zamenu implementacija, kao i pisanje automatizovanih testova [11]. U razvijenoj aplikaciji ovaj mehanizam koristi se za prosleđivanje identiteta prijavljenog korisnika i odgovarajućih servisnih klasa koje izvršavaju poslovnu logiku.

Način razmene podataka između klijenta i servera neposredno je povezan sa modelima koji opisuju strukturu zahteva i odgovora. FastAPI se u tom pogledu oslanja na biblioteku Pydantic [30], koja omogućava validaciju podataka i automatsko generisanje OpenAPI specifikacije. Ove mogućnosti predstavljaju jednu od najznačajnijih prednosti FastAPI okvira i biće detaljnije opisane u narednom potpoglavlju.

Validacija podataka i automatska dokumentacija API-ja

Pouzdanost jednog informacionog sistema u velikoj meri zavisi od kvaliteta podataka koji ulaze u aplikaciju. Neispravni ili nepotpuni podaci mogu dovesti do grešaka u izvršavanju poslovne logike, narušavanja integriteta baze podataka ili neočekivanog ponašanja sistema. Zbog toga validacija predstavlja sastavni deo razvoja REST servisa i obavlja se pre nego što zahtev bude prosleđen poslovnoj logici aplikacije.

FastAPI se za validaciju oslanja na biblioteku Pydantic, koja omogućava definisanje modela podataka pomoću tipova programskog jezika Python i dodatnih ograničenja. Na osnovu tako definisanih modela automatski se proverava da li ulazni podaci odgovaraju očekivanoj strukturi, da li su prisutna sva obavezna polja i da li su vrednosti odgovarajućeg tipa. Ukoliko zahtev ne zadovoljava definisana pravila, obrada se prekida, a klijentu se vraća odgovor sa odgovarajućim HTTP statusnim kodom i opisom greške. Na ovaj način validacija se obavlja na jedinstvenom mestu, bez potrebe za dodatnim proverama unutar poslovne logike [11, 30].

Pored osnovnih tipova podataka, Pydantic podržava definisanje dodatnih ograničenja, kao što su minimalna i maksimalna dužina tekstualnih vrednosti, opsezi numeričkih podataka, validacija adresa elektronske pošte i korišćenje enumeracija za unapred definisane skupove vrednosti. U okviru razvijenog sistema modeli zahteva i odgovora opisani su upravo na ovaj način, čime je postignuta jasna specifikacija strukture podataka koji se razmenjuju između klijentske i serverske strane aplikacije [30].

Značajna prednost FastAPI okvira jeste to što se na osnovu definisanih ruta i Pydantic modela automatski generiše OpenAPI specifikacija, koja predstavlja standardizovan opis REST API-ja. Na osnovu ove specifikacije FastAPI pravi interaktivne dokumentacije Swagger UI i ReDoc, dostupne odmah nakon pokretanja aplikacije. Programeri tako mogu pregledati sve dostupne rute, njihove ulazne i izlazne modele, kao i izvršavati testne zahteve direktno iz pregledača, bez korišćenja dodatnih alata [11, 38, 46].

2.3 Biblioteka React i programski jezik TypeScript

Savremene veb aplikacije sve češće se razvijaju kao sistemi sa jasno razdvojenim klijentskim i serverskim delom. Dok je serverska strana zadužena za obradu poslovne logike, upravljanje podacima i implementaciju bezbednosnih mehanizama, klijentska aplikacija odgovorna je za interakciju sa korisnikom i prikaz informacija.

Među bibliotekama otvorenog koda namenjenim razvoju korisničkog interfejsa posebno mesto zauzima React, jedan od najrasprostranjenijih alata za izradu jednostraničnih veb aplikacija (engl. Single Page Application – SPA), razvijen u kompaniji Meta (tada Facebook). Njegov komponentni model razvoja, deklarativni način opisivanja korisničkog interfejsa i bogat ekosistem biblioteka doprineli su širokoj primeni kako u industriji, tako i u akademskim projektima [23].

Kako aplikacije postaju složenije, raste i potreba za većom pouzdanošću izvornog koda. Iz tog razloga React se često kombinuje sa programskim jezikom TypeScript, koji uvodi statičku proveru tipova i omogućava ranije otkrivanje velikog broja grešaka tokom razvoja. Takva kombinacija pruža dobar odnos između jednostavnosti razvoja i kvaliteta implementacije, zbog čega je primenjena i u okviru sistema razvijenog u ovom radu.

U nastavku će biti prikazani osnovni principi na kojima se zasniva biblioteka React, način razvoja jednostraničnih veb aplikacija i organizacije klijentskog rutiranja, kao i pristupi upravljanju stanjem, validaciji podataka i primeni programskog jezika TypeScript.

Komponentni model biblioteke React

Osnovni cilj biblioteke React jeste pojednostavljivanje izgradnje složenih korisničkih interfejsa kroz podelu aplikacije na manje, međusobno nezavisne komponen-

te. Umesto posmatranja stranice kao jedinstvene celine, React podstiče razvoj kroz skup ponovo upotrebljivih komponenti od kojih svaka ima jasno definisanu odgovornost, što doprinosi boljoj organizaciji izvornog koda, jednostavnijem održavanju i lakšem proširivanju funkcionalnosti [23].

Osnovu ovog pristupa čini deklarativni model razvoja korisničkog interfejsa: programer opisuje kako interfejs treba da izgleda za određeno stanje aplikacije, dok React automatski određuje koje delove stranice je potrebno ažurirati nakon promene podataka. Na taj način izbegava se neposredna manipulacija DOM stablom (engl. Document Object Model) [50], što razvoj čini preglednijim i smanjuje mogućnost grešaka izazvanih ručnim upravljanjem elementima interfejsa [3].

Kako bi optimizovao prikaz promena, biblioteka React koristi koncept virtuelnog DOM-a (engl. virtual DOM): pri svakoj promeni stanja biblioteka najpre pravi njegovu virtuelnu reprezentaciju u memoriji, poredi je sa prethodnim stanjem i identifikuje samo elemente koji su zaista izmenjeni, nakon čega stvarni DOM ažurira isključivo na tim mestima. Ovakav mehanizam smanjuje broj operacija nad pregledníkom i doprinosi efikasnijem prikazu, naročito kod aplikacija sa većim brojem dinamičkih elemenata [23].

Savremene verzije biblioteke React zasnivaju se gotovo isključivo na funkcionalnim komponentama i konceptu *Hooks*. Funkcionalna komponenta je JavaScript ili TypeScript funkcija koja na osnovu ulaznih podataka (engl. props) vraća opis korisničkog interfejsa. Za razliku od ranije korišćenih klasnih komponenti, ovakav pristup omogućava jednostavniju organizaciju izvornog koda i prirodnije ponovno korišćenje logike između delova aplikacije [51].

Hooks su skup ugrađenih funkcija kojima funkcionalne komponente dobijaju mogućnost upravljanja stanjem i izvršavanja različitih efekata tokom životnog ciklusa komponente. Najčešće korišćeni su *useState*, namenjen čuvanju lokalnog stanja, i *useEffect*, koji se koristi za izvršavanje operacija nakon prikaza komponente ili promene podataka. Biblioteka React nudi i druge funkcije tipa Hook, za optimizaciju rada aplikacije, pristup kontekstu ili rad sa referencama na DOM elemente, koji se koriste samo kada za to postoji konkretna potreba [51].

Ovaj model posebno dolazi do izražaja kod većih aplikacija, gde svaka komponenta može da se razvija, testira i održava nezavisno, dok se složeniji prikazi formiraju njihovim kombinovanjem. U okviru sistema razvijenog u ovom radu korišćene su isključivo funkcionalne komponente, dok su klasne komponente u potpunosti izostavljene, a korisnički interfejs organizovan je kroz veći broj međusobno nezavisnih

komponenti koje komuniciraju putem ulaznih parametara i lokalnog stanja.

Jednostranične veb aplikacije i klijentsko rutiranje

Nakon organizacije korisničkog interfejsa u komponente, sledeće pitanje jeste kako se korisnik kreće kroz aplikaciju. Tradicionalne veb aplikacije zasnivaju se na modelu u kome server za svaki korisnički zahtev generiše novu HTML stranicu i šalje je klijentu. Iako je ovakav pristup i dalje zastupljen u određenim oblastima, razvoj informacionih sistema sve češće se oslanja na SPA arhitekturu, kod koje se početna stranica učitava samo jednom, dok se sve naredne interakcije odvijaju dinamički, bez ponovnog učitavanja cele stranice. Podaci se preuzimaju putem REST API-ja, a korisnički interfejs se ažurira u skladu sa pristiglim odgovorima servera [23].

Jedan od ključnih elemenata SPA aplikacija jeste klijentsko rutiranje. Za razliku od tradicionalnog pristupa, gde server određuje koja će stranica biti prikazana, kod jednostraničnih aplikacija izbor odgovarajućeg prikaza obavlja se na klijentskoj strani. Promena URL adrese ne izaziva ponovno učitavanje cele aplikacije, već predstavlja signal ruteru da prikaže odgovarajuću komponentu. Na taj način zadržava se kontinuitet rada aplikacije, dok se korisnički interfejs menja samo u delu koji je neophodan [23].

U React ekosistemu za implementaciju klijentskog rutiranja najčešće se koristi biblioteka React Router, koja omogućava definisanje ruta, povezivanje URL adresa sa odgovarajućim komponentama i organizaciju navigacije unutar aplikacije. Osim osnovnog rutiranja, biblioteka pruža podršku za ugnježdene rute, programsko preusmeravanje korisnika i zaštitu pojedinih delova aplikacije u skladu sa definisanim pravilima pristupa [37].

U okviru razvijenog sistema React Router korišćen je za organizaciju svih korisničkih stranica i navigacije između njih. Pojedine rute dostupne su isključivo prijavljenim korisnicima, dok se prikaz određenih funkcionalnosti dodatno prilagođava ulozi korisnika. Ovakva ograničenja imaju isključivo ulogu unapređenja korisničkog iskustva — konačna provera prava pristupa izvršava se na serverskoj strani, tako da klijentska aplikacija ne donosi odluke o autorizaciji. Takav pristup odgovara principu razdvajanja odgovornosti između klijentskog i serverskog dela sistema i doprinosi bezbednosti aplikacije.

Pored organizacije komponenti i rutiranja, poseban značaj zato ima i način upravljanja stanjem aplikacije, kao i validacija korisničkog unosa.

Upravljanje stanjem, formama i validacijom

Jedna od osnovnih karakteristika interaktivnih veb aplikacija jeste mogućnost da se korisnički interfejs menja u skladu sa akcijama korisnika i podacima koji se razmenjuju sa serverskom stranom sistema. Da bi takvo ponašanje bilo moguće, neophodno je upravljati stanjem aplikacije (engl. state), odnosno podacima koji određuju trenutni izgled i ponašanje pojedinih komponenti. U biblioteci React svaka promena stanja automatski izaziva ponovno renderovanje odgovarajućih delova korisničkog interfejsa, čime se održava usklađenost između prikazanih podataka i stvarnog stanja aplikacije [23].

U manjim komponentama stanje se najčešće čuva lokalno, dok složenije aplikacije često zahtevaju deljenje podataka između većeg broja međusobno povezanih komponenti. U takvim situacijama React pruža više različitih pristupa upravljanju zajedničkim stanjem, pri čemu izbor odgovarajućeg rešenja zavisi od složenosti sistema i vrste podataka koji se dele. Kada je potrebno obezbediti pristup istim informacijama većem broju komponenti, često se koristi Context API, koji omogućava distribuciju zajedničkih podataka bez njihovog prosleđivanja kroz svaki nivo hijerarhije komponenti [3, 23].

Osim upravljanja stanjem, značajan deo razvoja klijentskih aplikacija odnosi se na obradu korisničkog unosa. Forme su osnovni način komunikacije između korisnika i informacionog sistema, zbog čega je važno obezbediti jednostavan unos podataka i pravovremenu povratnu informaciju o eventualnim greškama. Biblioteka React omogućava implementaciju kontrolisanih komponenti (engl. controlled components), kod kojih se vrednosti svih polja nalaze u stanju aplikacije, pa je svaka promena unosa odmah dostupna programskoj logici što pojednostavljuje proveru ispravnosti podataka i obradu događaja [3].

Kod složenijih formi često se koriste specijalizovane biblioteke koje pojednostavljuju upravljanje većim brojem polja i pravila validacije. Jedno od najrasprostranjenijih rešenja u React ekosistemu je biblioteka React Hook Form, koja smanjuje količinu potrebnog izvornog koda i omogućava efikasno upravljanje obrascima korišćenjem React Hooks mehanizma. Za definisanje pravila validacije često se koristi biblioteka Zod [52], koja omogućava deklarativan opis strukture podataka i proveru njihove ispravnosti pre slanja zahteva serverskoj strani [36, 52].

Iako validacija na klijentskoj strani doprinosi boljem korisničkom iskustvu, ona ne može biti jedini mehanizam zaštite aplikacije. Korisnik uvek može zaobići ograničenja implementirana u veb pregledaču ili direktno poslati HTTP zahtev serverskoj

aplikaciji. Iz tog razloga autoritativna validacija mora biti izvršena na serverskoj strani, dok validacija u korisničkom interfejsu ima prvenstveno zadatak da korisniku omogući brže otkrivanje grešaka prilikom unosa. Takav pristup predstavlja standardnu praksu u razvoju veb aplikacija i doprinosi većoj pouzdanosti informacionog sistema [11, 36].

Primena programskog jezika TypeScript

Pouzdanost korisničkog interfejsa ne zavisi samo od validacije unosa, već i od grešaka koje nastaju u samom izvornom kodu tokom razvoja. Budući da je JavaScript dinamički tipiziran programski jezik, veliki broj grešaka vezanih za tipove podataka nije moguće otkriti pre pokretanja aplikacije, a sa porastom broja modula, komponenti i programera koji učestvuju u razvoju sistema, održavanje izvornog koda postaje zahtevnije. Iz tog razloga sve češće se koristi TypeScript, programski jezik koji proširuje JavaScript uvođenjem statičke provere tipova i dodatnih mehanizama namenjenih razvoju većih softverskih sistema [24].

TypeScript je razvila kompanija Microsoft sa ciljem da zadrži potpunu kompatibilnost sa JavaScript ekosistemom, uz unapređenje pouzdanosti i preglednosti izvornog koda. Program napisan u programskom jeziku TypeScript prevodi se u standardni kod programskog jezika JavaScript, zbog čega se može izvršavati u svim veb pregledačima i okruženjima koja podržavaju JavaScript. Na taj način programeri mogu koristiti postojeće biblioteke i alate, bez potrebe za promenom načina izvršavanja aplikacije [24].

Jedna od najznačajnijih prednosti programskog jezika TypeScript jeste mogućnost eksplicitnog definisanja tipova podataka. Promenljive, parametri funkcija, povratne vrednosti i složeniji modeli mogu biti opisani odgovarajućim tipovima, čime prevodilac proverava njihovu međusobnu usklađenost pre pokretanja aplikacije. Zahvaljujući tome, veliki broj grešaka otkriva se već tokom razvoja, što doprinosi pouzdanijem radu aplikacije i pojednostavljuje proces održavanja [5].

Poseban značaj TypeScript ima u razvoju React aplikacija, gde se tipovi koriste za opis ulaznih parametara komponenti, modela podataka koji se razmenjuju sa serverskom stranom i strukture stanja aplikacije. Na taj način komunikacija između delova sistema postaje jednoznačnija, dok se mogućnost nastanka grešaka usled pogrešne upotrebe podataka značajno smanjuje [3, 24].

2.4 Sistemi za upravljanje bazama podataka

Tokom razvoja informacionih sistema razvijeni su različiti modeli organizacije podataka, od kojih su danas najzastupljeniji relacioni i nerelacioni pristup. Relacione baze podataka zasnivaju se na relacionom modelu, u kojem su struktura podataka i veze između njih definisane unapred utvrđenom šemom. Ovakva statička šema posebno odgovara sistemima kod kojih su važni konzistentnost podataka i jasno definisani odnosi između entiteta. Sa druge strane, nerelacione baze podataka nude veću fleksibilnost u organizaciji podataka i često se koriste kada je potrebno skladištiti informacije promenljive strukture ili velike količine nestrukturiranog sadržaja [6, 20].

Savremene aplikacije sve češće kombinuju više različitih tehnologija za skladištenje podataka, pri čemu se za svaki tip informacija bira rešenje koje najbolje odgovara njegovim karakteristikama. Ovakav pristup poznat je pod nazivom *polyglot persistence* i omogućava da se prednosti različitih modela baza podataka iskoriste u okviru jedne aplikacije, umesto da se svi podaci skladište na isti način [40].

U nastavku će najpre biti opisane osnovne karakteristike relacionih baza podataka i sistema PostgreSQL, zatim koncept nerelacionih baza i MongoDB sa GridFS mehanizmom za skladištenje datoteka, dok će na kraju biti objašnjeni razlozi za kombinovanje relacionog i dokumentnog pristupa u okviru jednog informacionog sistema.

Relacione baze podataka

Relacioni model podataka, koji podatke organizuje u obliku relacija koje se u sistemima za upravljanje bazama podataka predstavljaju tabelama, predložio je 1970. godine Edgar F. Codd. Svaka tabela opisuje skup podataka o jednoj vrsti entiteta, dok svaki red odgovara jednoj instanci te vrste entiteta, što omogućava jasno modelovanje poslovnih podataka i njihovih međusobnih odnosa [6, 44].

Osnovna prednost relacionog modela ogleda se u preciznom definisanju strukture podataka. Svaki zapis mora biti usklađen sa definisanom šemom tabele. Zahvaljujući tome postiže se visok stepen konzistentnosti podataka i smanjuje mogućnost nastanka logičkih grešaka izazvanih unosom neispravnih ili nepotpunih informacija [6].

Jedan od ključnih elemenata relacionih baza podataka jesu ključevi. Primarni ključ (engl. primary key) jedinstveno identifikuje svaki red u tabeli, dok strani ključ

(engl. foreign key) uspostavlja vezu sa redom u drugoj tabeli. Na ovaj način moguće je modelovati različite poslovne odnose, poput veza između korisnika i njihovih nekretnina, ugovora i zakupaca ili zahteva i odgovarajućih nekretnina. Sistem za upravljanje bazom podataka (SUBP) automatski proverava ispravnost definisanih veza i sprečava narušavanje referencijalnog integriteta [44].

Osim organizacije podataka, relacioni sistem za upravljanje bazama podataka (RSUBP) obezbeđuje i pouzdano izvršavanje transakcija. Transakcija je logička celina sastavljena od jedne ili više operacija koje se izvršavaju kao nedeljiva celina, a njihovo ponašanje opisano je ACID principima (engl. Atomicity, Consistency, Isolation, Durability) [20, 44].

Savremeni relacioni sistemi za upravljanje bazama podataka raspolažu i brojnim mehanizmima za optimizaciju rada. Jedan od najvažnijih su indeksi, koji omogućavaju brže pronalaženje podataka bez potrebe za sekvencijalnim pregledom cele tabele, čime je moguće značajno smanjiti vreme izvršavanja upita, naročito kod tabela sa velikim brojem zapisa. Međutim, budući da svaki indeks zauzima dodatni prostor i zahteva ažuriranje prilikom izmene podataka, njihova primena zahteva pažljivo projektovanje i izbor atributa koji se najčešće koriste prilikom pretrage [44].

Zahvaljujući jasno definisanoj strukturi podataka, podršci za transakcije i očuvanju referencijalnog integriteta, relacione baze podataka ostaju standardno rešenje za implementaciju poslovnih informacionih sistema u kojima su pouzdanost i konzistentnost podataka od presudnog značaja. RSUBP korišćen u ovom radu, PostgreSQL, opisan je u nastavku.

PostgreSQL predstavlja jedan od najpoznatijih i najrasprostranjenijih sistema za upravljanje relacionim bazama podataka otvorenog koda. Stekao je reputaciju pouzdanog i stabilnog sistema koji se koristi u velikom broju poslovnih, naučnih i industrijskih aplikacija, zahvaljujući usklađenosti sa SQL standardom, bogatom skupu funkcionalnosti i aktivnoj zajednici korisnika i programera [29].

Osim standardnih SQL mogućnosti, PostgreSQL podržava veliki broj dodatnih tipova podataka i mehanizama koji olakšavaju razvoj složenijih aplikacija. Među njima se izdvajaju numerički tipovi sa visokom preciznošću, različiti vremenski tipovi, univerzalni identifikatori (UUID), kao i podrška za rad sa JSON dokumentima. Ovakva fleksibilnost omogućava da se u okviru iste baze podataka efikasno skladište različite vrste strukturiranih informacija, uz zadržavanje prednosti relacionog modela [29].

Osim standardnog B-stabla indeksa, sistem podržava i druge tipove indeksa na-

menjene specifičnim vrstama podataka i upita, čime je moguće prilagoditi način pretrage konkretnim zahtevima aplikacije i smanjiti vreme izvršavanja složenijih SQL upita, posebno kod baza sa velikim brojem zapisa [29].

Važna prednost PostgreSQL sistema ogleda se i u njegovoj prenosivosti i širokoj podršci unutar softverskog ekosistema. Dostupan je za različite operativne sisteme, integriše se sa velikim brojem programskih jezika i razvojnih okvira, a zbog otvorenog koda i aktivnog razvoja često je prvi izbor prilikom implementacije poslovnih informacionih sistema [29].

U okviru sistema razvijenog u ovom radu PostgreSQL koristi se kao primaran SUBP za skladištenje strukturiranih poslovnih informacija, uključujući podatke o korisnicima, nekretninama, ugovorima i zakupima, čija međusobna povezanost zahteva očuvanje referencijalnog integriteta i transakcionu obradu. Takav izbor odgovara prirodi ovih podataka i omogućava da poslovna logika sistema počiva na pouzdanoj i konzistentnoj osnovi.

Nerelacione baze podataka

Iako relacione baze pokrivaju veliki deo potreba poslovnih informacionih sistema, savremene aplikacije sve češće obrađuju i podatke drugačije prirode: dokumente, multimedijalni sadržaj, informacije sa različitih uređaja i podatke čija se struktura menja tokom vremena. U takvim situacijama tradicionalni relacioni model nije uvek najpogodnije rešenje, zbog čega su razvijeni nerelacioni sistemi za upravljanje bazama podataka, poznati pod zajedničkim nazivom NoSQL (engl. Not only SQL) [20, 40].

Za razliku od relacionih sistema za upravljanje bazama podataka, NoSQL sistemi za upravljanje bazom podataka ne zasnivaju organizaciju podataka isključivo na tabelama i unapred definisanim šemama. Njihova osnovna ideja jeste prilagođavanje modela skladištenja konkretnom tipu podataka i zahtevima aplikacije. Zbog toga različiti NoSQL sistemi koriste različite modele organizacije podataka, pri čemu ne postoji jedinstvena struktura koja odgovara svim vrstama primene [20].

U literaturi se najčešće razlikuju četiri osnovne grupe NoSQL baza podataka: baze zasnovane na parovima ključ–vrednost (engl. key–value databases), dokumentne baze (engl. document databases), baze sa proširivim slogovima (engl. wide-column databases) i grafovske baze (engl. graph databases). Svaki od ovih pristupa razvijen je sa ciljem rešavanja specifičnih problema i optimizacije rada za određene vrste

podataka i upita. Izbor odgovarajućeg modela zavisi od prirode aplikacije i načina na koji se podaci koriste [20, 40].

Jedna od najrasprostranjenijih grupa NoSQL sistema jesu dokumentne baze podataka. Umesto organizacije podataka u međusobno povezane tabele, informacije se skladište u obliku dokumenata koji mogu sadržati jednostavne vrednosti, ugnježdene objekte i liste, čime se podaci koji logički pripadaju istoj celini čuvaju zajedno i lakše preuzimaju i obrađuju [20].

Fleksibilnost dokumentnog modela posebno dolazi do izražaja kod podataka promenljive strukture, gde različiti zapisi ne moraju sadržati potpuno isti skup atributa. Zbog toga se dokumentne baze često koriste za skladištenje logova, korisničkih dokumenata i drugih informacija koje nije praktično modelovati kroz veliki broj međusobno povezanih tabela [40].

Iako NoSQL sistemi donose veću fleksibilnost u organizaciji podataka, oni nisu zamišljeni kao zamena za relacione baze podataka u svim scenarijima. Naprotiv, u savremenim informacionim sistemima veoma često se koriste zajedno, pri čemu se za svaku grupu podataka bira model koji najbolje odgovara njenim karakteristikama. Jedan od najpoznatijih predstavnika dokumentnih sistema za upravljanje bazama podataka jeste MongoDB.

MongoDB podatke skladišti u obliku dokumenata grupisanih u kolekcije. Dokumenti su zapisani u BSON (engl. Binary JSON) formatu, binarnoj reprezentaciji JSON strukture koja omogućava efikasnije skladištenje i obradu različitih tipova podataka [25].

Dokumenti koji pripadaju istoj kolekciji ne moraju imati potpuno identičnu strukturu, što omogućava jednostavnije prilagođavanje promenama zahteva aplikacije bez potrebe za izmenom šeme cele baze podataka [20, 25]. SUBP MongoDB pruža i podršku za indeksiranje, rad sa ugnježdenim dokumentima i različitim tipovima upita, čime omogućava efikasno pretraživanje i obradu većih količina podataka. Reč je o zrelom rešenju koje obezbeđuje visok nivo pouzdanosti i primenjuje se u velikom broju informacionih sistema [25].

Osim skladištenja dokumenata, u okviru MongoDB ekosistema dostupan je i mehanizam GridFS, namenjen čuvanju datoteka koje nije praktično skladištiti kao deo standardnih dokumenata. Umesto da se kompletan sadržaj datoteke čuva u jednom dokumentu, GridFS je deli na manje segmente (engl. chunks), koji se skladište u posebnoj kolekciji, dok se metapodaci o datoteci čuvaju odvojeno. Na ovaj način omogućeno je efikasno skladištenje i preuzimanje većih binarnih sadržaja, bez

ograničenja koja postoje kod pojedinačnih BSON dokumenata [25].

GridFS nije zaseban sistem za skladištenje datoteka, već sastavni deo MongoDB platforme, zbog čega dokumenti i njihovi metapodaci ostaju u okviru istog sistema — što pojednostavljuje upravljanje datotekama i njihovo povezivanje sa ostalim podacima aplikacije. Ovakav pristup često se primenjuje u sistemima koji čuvaju različite vrste priloga, izveštaja, slika ili drugih dokumenata koje je potrebno kasnije preuzeti ili prikazati korisniku [25].

U okviru sistema razvijenog u ovom radu MongoDB i GridFS koriste se upravo na ovaj način, za podatke čija struktura nije pogodna za relacioni model i za korisničke dokumente koji se prilažu uz poslovne procese.

Kombinovanje relacionog i dokumentnog pristupa

Kao što je pokazano u prethodna dva potpoglavlja, PostgreSQL i MongoDB u razvijenom sistemu ne konkurišu jedan drugom, već pokrivaju različite vrste podataka što ilustruje širu zakonitost da nijedan model skladištenja nije podjednako dobro rešenje za sve vrste informacija koje aplikacija obrađuje. Dok relacione baze pružaju visok nivo konzistentnosti i pouzdanu transakcionu obradu strukturiranih poslovnih podataka, dokumentne baze omogućavaju veću fleksibilnost prilikom skladištenja informacija promenljive strukture. Iz tog razloga u praksi se sve češće primenjuje pristup poznat pod nazivom *polyglot persistence*, koji podrazumeva korišćenje više različitih sistema za upravljanje bazama podataka u okviru iste aplikacije, pri čemu se za svaku grupu podataka bira tehnologija koja najbolje odgovara njenim karakteristikama [20, 40].

Primena više različitih sistema za skladištenje podataka donosi i određene izazove. Budući da se informacije nalaze u različitim bazama podataka, ne postoji mogućnost izvršavanja zajedničkih SQL upita niti automatskog očuvanja referencijalnog integriteta između njih. Iz tog razloga projektovanje ovakvih sistema zahteva jasno razdvajanje odgovornosti pojedinih baza podataka, pažljivo definisanje granica između različitih tipova podataka i dosledno održavanje njihovih međusobnih veza na nivou aplikacije. Kada je ovakva podela pravilno projektovana, prednosti kombinovanog pristupa u velikoj meri prevazilaze njegovu dodatnu složenost [20, 40].

U okviru sistema razvijenog u ovom radu relaciona i dokumentna baza podataka imaju jasno razdvojene odgovornosti. PostgreSQL koristi se za skladištenje strukturiranih poslovnih podataka, uključujući korisničke naloge, nekretnine, ugovore i zakup, čiji odnosi zahtevaju očuvanje referencijalnog integriteta i transakcionu ob-

radu. MongoDB je, sa druge strane, namenjen skladištenju podataka promenljive strukture, kao što su evidencije aktivnosti i bezbednosni zapisi, dok se GridFS koristi za čuvanje korisnički otpremljenih dokumenata.

Ovakav pristup ne znači dupliranje istih podataka u različitim bazama, niti paralelno održavanje dve nezavisne verzije poslovnog modela. Svaka baza podataka čuva isključivo informacije koje pripadaju njenoj oblasti odgovornosti, dok se međusobna povezanost ostvaruje korišćenjem identifikatora koji omogućavaju aplikaciji da poveže podatke iz različitih izvora kada je to potrebno. Time se izbegava nepotrebno ponavljanje, a istovremeno zadržavaju prednosti oba modela skladištenja podataka.

2.5 Pristup podacima i validacija

Razvoj veb aplikacija ne podrazumeva samo izbor odgovarajućeg sistema za upravljanje bazom podataka, već i način na koji aplikacija pristupa podacima i proverava njihovu ispravnost. Direktna komunikacija sa SUBP i obrada korisničkog unosa dve su oblasti koje imaju značajan uticaj na pouzdanost, održavanje i bezbednost informacionog sistema. Zbog toga se u praksi često koriste specijalizovane biblioteke koje pojednostavljuju rad sa bazama podataka i omogućavaju doslednu validaciju podataka pre njihove dalje obrade.

U Python ekosistemu među najrasprostranjenijim rešenjima za ove namene izdvajaju se biblioteke SQLAlchemy i Pydantic [30, 45]. SQLAlchemy omogućava rad sa relacionim bazama podataka korišćenjem objektno-orijentisanog pristupa, dok Pydantic obezbeđuje proveru ispravnosti podataka i njihovu konverziju između različitih formata.

Osim organizacije pristupa podacima i njihove validacije, značajnu ulogu u razvoju veb aplikacija ima i asinhroni način rada. Budući da je komunikacija sa SUBP I/O operacija, asinhroni pristup omogućava efikasnije korišćenje sistemskih resursa i bolju sposobnost aplikacije da istovremeno obrađuje veći broj korisničkih zahteva. Zbog toga će u nastavku biti prikazane osnovne karakteristike biblioteka SQLAlchemy i Pydantic, kao i koncept asinhronog pristupa podacima koji je primenjen u okviru razvijenog sistema.

Biblioteka SQLAlchemy

SQLAlchemy predstavlja jednu od najpoznatijih biblioteka programskog jezika Python namenjenih radu sa relacionim bazama podataka. Jedna od osnovnih mogućnosti ove biblioteke jeste objektno-relaciono mapiranje (engl. Object-Relational Mapping – ORM). ORM omogućava da se podaci iz tabela predstave kroz objekte programskog jezika, čime se smanjuje razlika između relacionog modela baze podataka i objektno-orijentisanog modela aplikacije. Na taj način programer radi sa objektima i njihovim svojstvima, dok SQLAlchemy preuzima odgovornost za prevođenje tih operacija u odgovarajuće SQL upite i naredbe [45].

Uz mapiranje objekata, SQLAlchemy obezbeđuje mehanizme za upravljanje sesijama i transakcijama. Sesija je veza između aplikacije i SUBP tokom izvršavanja poslovnih operacija, dok transakcije omogućavaju da se više međusobno povezanih izmena izvrši kao jedinstvena logička celina [44, 45].

Novije verzije biblioteke SQLAlchemy pružaju podršku i za asinhroni rad, što omogućava efikasnu integraciju sa Python razvojnim okvirima koji koriste asinhroni model izvršavanja [45].

U okviru sistema razvijenog u ovom radu korišćen je SQLAlchemy ORM sa deklarativnim mapiranjem modela i asinhronim sesijama za komunikaciju sa RSUBP. Biblioteka je zadužena za mapiranje poslovnih entiteta na odgovarajuće tabele, upravljanje sesijama i izvršavanje transakcija, dok su arhitektonski obrasci koji organizuju pristup podacima, poput repozitorijuma i obrasca *Unit of Work*, deo same arhitekture sistema i biće detaljnije opisani u narednoj glavi rada.

Biblioteka Pydantic

Razmena podataka između klijentske i serverske strane aplikacije zahteva poseban, pouzdan mehanizam za proveru ispravnosti korisničkog unosa i konverziju podataka između različitih formata. Bez odgovarajuće validacije postoji mogućnost da aplikacija obradi nepotpune ili neispravne podatke, što može dovesti do grešaka u poslovnoj logici ili narušavanja integriteta sistema. Iz tog razloga Python razvojni okviri često koriste specijalizovane biblioteke koje omogućavaju automatsku proveru podataka pre njihove dalje obrade.

Jedna od najčešće korišćenih biblioteka za ovu namenu jeste Pydantic. Njena osnovna uloga jeste definisanje modela podataka i njihova validacija na osnovu Python tipova. Umesto ručne provere svakog pojedinačnog polja, programer opisuje

očekivanu strukturu podataka, dok Pydantic automatski proverava da li pristigli podaci odgovaraju definisanom modelu. Time se smanjuje količina pomoćnog koda i povećava pouzdanost aplikacije [30].

Pored validacije, Pydantic omogućava jednostavnu serijalizaciju i deserijalizaciju podataka. Prilikom prijema HTTP zahteva podaci iz JSON formata konvertuju se u odgovarajuće Python objekte, dok se prilikom slanja odgovora obavlja obrnuti postupak [30].

Važnu ulogu imaju i ograničenja definisana nad pojedinim poljima modela. Uz osnovnu proveru tipova, moguće je zadati dodatna pravila, poput minimalne dužine tekstualnih vrednosti, dozvoljenih opsega numeričkih podataka ili provere formata određenih vrednosti. Zahvaljujući tome, veliki broj nepravilnosti može biti otkriven odmah po prijemu zahteva, pre nego što podaci dospeju do poslovne logike aplikacije [30].

U okviru sistema razvijenog u ovom radu Pydantic modeli koriste se za definisanje strukture ulaznih i izlaznih podataka REST servisa. Njihova primena obuhvata validaciju korisničkog unosa, opis formata odgovora i konverziju podataka između JSON zapisa i Python objekata. Za učitavanje i validaciju konfiguracionih parametara aplikacije koristi se paket pydantic-settings, koji omogućava definisanje konfiguracije i primenu odgovarajućih pravila validacije [31].

Asinhroni pristup bazama podataka

Osim pristupa podacima i njihove validacije, jednako je važan i način na koji aplikacija komunicira sa SUBP tokom izvršavanja. Veb aplikacije često istovremeno obrađuju veliki broj korisničkih zahteva, a tokom izvršavanja pojedinih operacija provode značajan deo vremena čekajući odgovor spoljašnjih sistema, kao što su sistemi za upravljanje bazama podataka, sistemi za skladištenje datoteka ili mrežni servisi. Budući da se u tom periodu ne izvršavaju procesorski zahtevni zadaci, blokiranje izvršavanja programa može dovesti do nepotrebnog zauzimanja resursa i smanjenja ukupnih performansi sistema. Zbog toga se u razvoju serverskih aplikacija sve češće primenjuje asinhroni model rada [33, 35].

Programski jezik Python podržava asinhrono programiranje putem konstrukcija *async* i *await*, koje omogućavaju pisanje preglednog i čitljivog izvornog koda uz zadržavanje asinhronog načina izvršavanja. Savremene biblioteke za pristup bazama podataka uglavnom pružaju podršku za ovakav model rada, što omogućava da

pristup podacima bude usklađen sa ostatkom aplikacije i njenim načinom obrade korisničkih zahteva [33].

Asinhroni pristup ne menja poslovnu logiku aplikacije niti način organizacije podataka u bazi podataka. Njegova osnovna uloga jeste efikasnije korišćenje raspoloživih sistemskih resursa tokom izvršavanja operacija koje podrazumevaju čekanje na odgovor spoljnog sistema [35].

U okviru sistema razvijenog u ovom radu asinhroni model primenjen je u komunikaciji sa relacionim i dokumentnim SUBP. SQLAlchemy koristi asinhrone sesije za rad sa RSUBP PostgreSQL, dok se za rad sa SUBP MongoDB koristi asinhroni drajver.

2.6 Pomoćne tehnologije i biblioteke

Osim osnovnih tehnologija koje čine okosnicu razvijenog informacionog sistema, tokom implementacije korišćene su i pomoćne biblioteke namenjene rešavanju pojedinačnih zadataka. Njihova uloga nije da određuju arhitekturu aplikacije, već da pojednostave implementaciju funkcionalnosti kao što su komunikacija između klijentskog i serverskog dela sistema, validacija korisničkog unosa, autentifikacija korisnika, izvršavanje komponenti aplikacije u kontejnerima i automatizovano testiranje.

Izbor ovih biblioteka i alata zasnovan je na njihovoj širokoj zastupljenosti u razvoju Python i React aplikacija, dobroj dokumentovanosti i aktivnoj podršci zajednice. Njihova primena doprinosi preglednijoj organizaciji izvornog koda i pojednostavljuje razvoj, održavanje i testiranje aplikacije, bez uvođenja dodatne složenosti u arhitekturu sistema.

U nastavku će biti prikazane osnovne karakteristike biblioteka i alata koje su korišćene tokom razvoja aplikacije, sa naglaskom na njihovu ulogu u okviru realizovanog rešenja.

Biblioteka Axios

Komunikacija između klijentske i serverske strane jedan je od osnovnih aspekata rada svake veb aplikacije. Na klijentskoj strani često se koriste biblioteke koje pojednostavljuju slanje HTTP zahteva i obradu odgovora servera. Jedna od najrasprostranjenijih biblioteka za ovu namenu u React ekosistemu jeste Axios, koja omo-

gućava jednostavno izvršavanje HTTP zahteva korišćenjem intuitivnog programskog interfejsa [2].

Osim osnovnog slanja zahteva, Axios pruža mogućnost centralizovanog definisanja zajedničkih podešavanja za komunikaciju sa serverskom aplikacijom. Na taj način moguće je na jednom mestu definisati adresu REST servisa, obradu grešaka ili zajednička HTTP zaglavlja, čime se smanjuje ponavljanje izvornog koda i olakšava njegovo održavanje [2].

Značajna mogućnost ove biblioteke je mehanizam presretača (engl. interceptors), koji omogućava obradu zahteva pre njihovog slanja, kao i obradu odgovora nakon prijema sa servera. Ovakav pristup posebno je pogodan za implementaciju mehanizama autentifikacije, jer omogućava automatsko dodavanje tokena svakom zahtevu, kao i jedinstveno reagovanje u slučaju isteka korisničke sesije [2].

U okviru sistema razvijenog u ovom radu Axios je centralna biblioteka za komunikaciju sa REST servisima. Korišćenjem presretača obezbeđeno je automatsko dodavanje JWT pristupnog tokena u svaki zahtev, kao i transparentno osvežavanje sesije kada je to potrebno. Na taj način komunikacija između klijentskog i serverskog dela sistema ostaje jedinstvena i pregledna, dok poslovna logika ostaje izdvojena od tehničkih detalja razmene podataka.

Biblioteke React Hook Form i Zod

Biblioteka React Hook Form omogućava organizaciju stanja forme, obradu korisničkog unosa i jednostavnije rukovanje procesom slanja podataka. Biblioteka je zasnovana na React Hooks mehanizmu, zbog čega omogućava preglednu organizaciju izvornog koda i smanjuje potrebu za ručnim upravljanjem velikim brojem pojedinačnih polja forme [36].

Biblioteka Zod koristi se za opis strukture podataka i definisanje pravila njihove validacije. Na osnovu unapred definisane šeme moguće je proveriti da li uneti podaci odgovaraju očekivanom formatu, čime se korisniku odmah ukazuje na eventualne greške prilikom unosa. Na ovaj način poboljšava se korisničko iskustvo, dok odgovornost za konačnu proveru podataka i dalje ostaje na serverskoj strani aplikacije [52].

U okviru sistema razvijenog u ovom radu React Hook Form i Zod primenjeni su u autentifikacionom delu aplikacije, gde omogućavaju jednostavniju obradu korisničkog unosa i osnovnu validaciju podataka pre slanja zahteva serveru. Ostale forme implementirane su korišćenjem standardnih React mehanizama za upravlja-

nje stanjem, dok se konačna validacija svih zahteva izvršava na serverskoj strani sistema.

Standard JWT i algoritam za zaštitu lozinki Argon2

Bezbedna autentifikacija korisnika jedan je od osnovnih zahteva veb aplikacija. Osim provere identiteta korisnika prilikom prijave na sistem, potrebno je obezbediti da se identitet korisnika pouzdano potvrđuje i tokom narednih zahteva prema serverskoj aplikaciji. U tu svrhu često se koriste tokeni za autentifikaciju i pouzdani algoritmi za zaštitu korisničkih lozinki [4, 18, 19].

JSON Web Token (JWT) [18] je standard za bezbedan prenos informacija između dve strane u obliku digitalno potpisanog tokena. Nakon uspešne prijave korisnika, serverska aplikacija izdaje pristupni token koji klijent prilaže uz svaki naredni HTTP zahtev. Na taj način server može da utvrdi identitet korisnika na osnovu pristupnog tokena, bez provere serverskog stanja pri svakom zahtevu, što ovakav pristup čini pogodnim za razvoj REST servisa [18, 19].

U praksi se, uz pristupni token, često koristi i osvežavajući token, koji omogućava obnavljanje tokena bez potrebe za ponovnom prijavom korisnika. Pristupni tokeni imaju kraći period važenja, dok se osvežavajući token koristi za izdavanje novog para tokena nakon isteka pristupnog tokena [18, 19].

Uz autentifikaciju, poseban značaj ima i zaštita korisničkih lozinki. Uobičajena praksa podrazumeva da se lozinke nikada ne čuvaju u izvornom obliku, već isključivo kao rezultat procesa heširanja. Jedan od algoritama koji se danas preporučuje za ovu namenu jeste Argon2 [4, 27].

U okviru sistema razvijenog u ovom radu autentifikacija korisnika zasnovana je na JWT pristupnim i osvežavajućim tokenima, dok se korisničke lozinke čuvaju primenom Argon2 algoritma za heširanje. Na ovaj način ostvarena je bezbedna identifikacija korisnika i zaštita poverljivih podataka, uz primenu mehanizama koji predstavljaju prihvaćenu praksu u razvoju savremenih veb aplikacija.

Platforma Docker i Docker Compose

Razvoj složenijih softverskih sistema često podrazumeva korišćenje većeg broja međusobno povezanih komponenti, kao što su serverska aplikacija, SUBP i drugi prateći servisi. Kako bi se obezbedilo dosledno izvršavanje aplikacije u različitim

razvojnim okruženjima, sve češće se primenjuje kontejnerizacija, koja omogućava da aplikacija i njene zavisnosti budu objedinjene u jedinstvenu celinu [9].

Docker [9] je platforma za razvoj, distribuciju i izvršavanje aplikacija u kontejnerima. Kontejner sadrži aplikaciju zajedno sa svim bibliotekama, konfiguracijama i ostalim komponentama neophodnim za njen rad. Na taj način obezbeđuje se da se aplikacija ponaša na isti način bez obzira na operativni sistem ili okruženje u kome se pokreće, čime se smanjuje mogućnost problema izazvanih razlikama u konfiguraciji računara [9].

Kada je potrebno pokrenuti više međusobno povezanih servisa, koristi se Docker Compose. Ovaj alat omogućava da se kompletno razvojno okruženje opiše kroz jedinstvenu konfiguraciju, nakon čega se svi potrebni kontejneri mogu pokrenuti jednom komandom. Takav pristup pojednostavljuje razvoj i testiranje aplikacije, jer obezbeđuje da svi članovi razvojnog tima koriste isto izvršno okruženje [8].

U okviru sistema razvijenog u ovom radu Docker je korišćen za kontejnerizaciju serverske aplikacije i pratećih servisa, dok je Docker Compose omogućio njihovo zajedničko pokretanje i međusobnu komunikaciju u okviru jedinstvenog razvojnog okruženja.

Biblioteka Pytest

Provera ispravnosti softverskog sistema sastavni je deo procesa razvoja aplikacija. Jedna od najčešće korišćenih biblioteka za automatizovano testiranje u Python okruženju jeste Pytest. Biblioteka pruža jednostavan način organizacije i izvršavanja testova, pri čemu podržava različite nivoe testiranja, od provere pojedinačnih funkcija do testiranja međusobne saradnje više komponenti sistema. Zbog pregledne sintakse i velike fleksibilnosti Pytest je postao standardno rešenje za testiranje Python aplikacija [32].

U okviru sistema razvijenog u ovom radu korišćena je biblioteka Pytest za implementaciju automatizovanih testova kojima su proverene ključne funkcionalnosti serverske aplikacije.

2.7 Kontrola pristupa i autorizacija

Uz tehnologije opisane u prethodnim poglavljima, poseban teorijski osnov zahteva mehanizam koji određuje šta pojedini korisnik sme da uradi nad podacima

informacionog sistema. Ovo pitanje dobija poseban značaj u aplikacijama koje istovremeno koristi veći broj korisnika sa različitim poslovnim ulogama, a koji pristupaju istim ili sličnim resursima. U nastavku su zato izložene osnovne teorijske postavke kontrole pristupa koje čine osnov za model realizovan u okviru sistema opisanog u ovom radu, dok će njegova konkretna implementacija biti detaljno predstavljena u glavi 3.

Autentifikacija i autorizacija

Iako se pojmovi autentifikacije i autorizacije u svakodnevnoj upotrebi često poistovećuju, reč je o dva različita, iako povezana koncepta kontrole pristupa informacionom sistemu. Autentifikacija predstavlja postupak provere identiteta korisnika, odnosno utvrđivanja da je korisnik zaista onaj za koga se predstavlja. Autorizacija se, sa druge strane, odnosi na odluku o tome koje aktivnosti već identifikovan korisnik sme da izvrši i kojim podacima sme da pristupi [26].

Razlika između ova dva koncepta može se ilustrovati jednostavnim primerom: uspešna prijava na sistem korišćenjem korisničkog imena i lozinke potvrđuje identitet korisnika, ali sama po sebi ne govori ništa o tome kojim resursima taj korisnik sme da pristupi niti koje operacije sme da izvrši nad njima. U nastavku ovog poglavlja pažnja je zato usmerena isključivo na autorizaciju, odnosno na modele i principe na osnovu kojih se donosi odluka o dozvoljenom pristupu.

Kontrola pristupa zasnovana na ulogama i njena ograničenja

Za razvoj poslovnih informacionih sistema, uključujući i sistem razvijen u okviru ovog rada, polazni model autorizacije predstavlja kontrola pristupa zasnovana na ulogama (engl. *Role-Based Access Control* – RBAC) [12, 41]. Osnovna ideja ovog modela jeste da se prava pristupa ne dodeljuju direktno pojedinačnim korisnicima, već ulogama koje odgovaraju njihovoj funkciji u organizaciji ili poslovnom procesu, dok se svakom korisniku zatim dodeljuje jedna ili više takvih uloga. Ovakav pristup značajno pojednostavljuje administraciju prava pristupa, jer je dovoljno definisati skup dozvola za svaku ulogu, umesto da se prava posebno određuju za svakog korisnika pojedinačno [41].

Iako RBAC dobro opisuje osnovne mogućnosti korisnika na osnovu njegove poslovne uloge, oslanjanje isključivo na mali skup globalnih uloga nije dovoljno kada korisnici sa istom ulogom imaju različite odnose prema pojedinačnim resursima. U

takvim sistemima autorizaciona odluka mora uzeti u obzir i kontekst konkretnog resursa, jer sama činjenica da dva korisnika imaju istu ulogu ne znači da treba da imaju pristup istim instancama tog resursa [41].

Ograničenje oslanjanja na mali broj globalnih uloga neposredno se ispoljava i u sistemu razvijenom u okviru ovog rada. Ukoliko bi se pristup isključivo zasnivao na korisničkoj ulozi, svaki korisnik sa ulogom vlasnika imao bi jednaka prava nad svakom nekretninom evidentiranom u sistemu, umesto isključivo nad nekretninama koje su u njegovom vlasništvu. Isti problem javlja se i kod korisnika sa ulogom agencije ili stanara, čiji bi pristup, u odsustvu dodatnih informacija, obuhvatao sve nekretnine, ugovore ili zakupe u sistemu, umesto samo onih sa kojima su konkretno povezani. Budući da razvijeni sistem istovremeno koristi veći broj korisnika sa istom ulogom, od kojih svaki upravlja sopstvenim, nezavisnim skupom poslovnih resursa, sama korisnička uloga ne može biti jedini kriterijum na osnovu kojeg se donosi autorizaciona odluka.

Kontrola pristupa zasnovana na atributima

Drugi pristup koji omogućava donošenje odluka na osnovu šireg konteksta jeste kontrola pristupa zasnovana na atributima (engl. *Attribute-Based Access Control* – ABAC). U ovom modelu odluka o pristupu može zavisiti od atributa subjekta, resursa, zahtevane operacije i okruženja u kojem se zahtev izvršava [17]. Iako ABAC omogućava izražavanje složenih pravila pristupa, u razvijenom sistemu ključni dodatni kontekst predstavlja konkretan poslovni odnos korisnika prema resursu. Budući da su ti odnosi unapred definisani i eksplicitno modelovani, ABAC nije izabran kao osnovni model autorizacije, već je odluka o pristupu neposredno zasnovana na odnosu korisnika prema konkretnom resursu.

Kontrola pristupa zasnovana na odnosima

U modelu zasnovanom isključivo na ulogama, dozvole korisnika proizlaze iz njegove dodeljene uloge, ali kada sistem koristi mali broj globalnih poslovnih uloga, sama informacija o ulozi ne opisuje odnos korisnika prema konkretnoj instanci resursa. Zbog toga je, za potrebe razvijenog sistema, autorizacionu odluku potrebno dopuniti informacijom o tom odnosu.

Potreba da se prilikom autorizacije razmatra odnos između korisnika i konkretnog resursa srodna je pristupu poznatom u literaturi kao kontrola pristupa zasno-

vana na odnosima (engl. *Relationship-Based Access Control* – ReBAC) [14], kod koga postojanje odgovarajuće relacije između subjekta i resursa predstavlja jedan od elemenata na osnovu kojih se određuje dozvoljeni pristup. Za potrebe ovog rada odnos korisnika prema resursu modelovan je kroz unapred definisan, ograničen skup poslovnih odnosa; u razvijenom sistemu takav kontekst čine, između ostalog, vlasništvo, ugovorni odnos, zakup i eksplicitno dodeljeno pravo pristupa.

Elementi odluke o pristupu

Za potrebe ovog rada, osnovni elementi zahteva za pristup mogu se predstaviti kroz subjekta koji zahteva pristup, resurs nad kojim se pristup ostvaruje i operaciju koju subjekat pokušava da izvrši nad tim resursom, kao što su čitanje, unos, ažuriranje ili brisanje podataka. Ovakva odluka može se formalno zapisati na sledeći način:

$$Pristup(S, R, Op) \rightarrow \{dozvoli, zabrani\}$$

gde S označava subjekta, R resurs, a Op operaciju koja se nad njim pokušava izvršiti [26].

U razvijenom sistemu ova tri elementa nisu dovoljna za donošenje autorizacione odluke, jer dozvoljeni pristup zavisi i od konkretnog odnosa korisnika prema resursu. Na primer, dva korisnika sa ulogom vlasnika mogu zahtevati istu operaciju nad istim tipom resursa, ali pravo pristupa konkretnoj nekretnini ima samo korisnik koji je sa njom povezan odgovarajućim poslovnim odnosom. Zbog toga se u okviru ovog rada osnovna formulacija proširuje dodatnim elementom koji predstavlja odnos subjekta prema resursu.

Odluka o pristupu tako se donosi na osnovu četiri elementa:

$$Pristup(Subjekat, Resurs, Operacija, Odnos) \rightarrow \{dozvoli, zabrani\}$$

Ovakav model može se posmatrati kroz dva konceptualna nivoa odlučivanja. Prvi se odnosi na korisničku ulogu i osnovne mogućnosti koje iz nje proizlaze, dok drugi uzima u obzir konkretan odnos korisnika prema resursu nad kojim se operacija izvršava. Ova dva nivoa ne predstavljaju nužno dve odvojene sekvencijalne provere, već zajedno čine kontekst na osnovu kojeg se donosi konačna autorizaciona odluka. Način na koji su ovi elementi objedinjeni u jedinstven mehanizam odlučivanja, kao i konkretni poslovni odnosi na kojima se on zasniva, detaljno su opisani i implementirani u glavi 3 ovog rada.

Glava 3

Projektovanje i implementacija veb aplikacije

Nakon razmatranja tehnologija korišćenih tokom razvoja, u ovoj glavi biće prikazano projektovanje i implementacija razvijene veb aplikacije za upravljanje nekretninama i procesom njihovog izdavanja. Naglasak je stavljen na organizaciju sistema, način realizacije ključnih poslovnih procesa i mehanizme koji omogućavaju različitim korisnicima pristup zajedničkim resursima u skladu sa njihovim ulogama i pravima.

Najpre će biti predstavljeni funkcionalni zahtevi i pregled implementiranog sistema, nakon čega će biti opisana njegova arhitektura, model podataka i način organizacije serverske i klijentske aplikacije. Posebna pažnja biće posvećena autentifikaciji korisnika, modelu autorizacije i segmentaciji podataka, budući da upravo ovi aspekti predstavljaju centralni deo istraživačkog doprinosa rada. Na kraju će biti prikazane najvažnije implementirane funkcionalnosti, način testiranja sistema i ograničenja razvijenog rešenja.

Implementacija sistema opisana u ovoj glavi dostupna je u javnom Git repozitorijumu projekta *Rentivo*: `https://github.com/momciloknezevic7/Rentivo`.

3.1 Zahtevi i pregled sistema

Projektovanje informacionog sistema započinje analizom poslovnog domena i definisanjem zahteva koje sistem treba da ispuni. U slučaju aplikacije za upravljanje nekretninama nije dovoljno omogućiti samo evidenciju podataka o nekretninama i korisnicima. Potrebno je podržati različite načine izdavanja nekretnina, upravljanje ugovorima, praćenje finansijskih obaveza, razmenu dokumenata i obradu za-

hteva korisnika, pri čemu više korisnika može istovremeno učestvovati u radu nad istim resursima. Takvo okruženje zahteva odgovarajuću kontrolu pristupa podacima i funkcionalnostima sistema.

Funkcionalni zahtevi

Razvijeni informacioni sistem namenjen je centralizovanom upravljanju nekretninama i procesom njihovog izdavanja. Cilj sistema jeste objedinjavanje ključnih poslovnih aktivnosti vezanih za upravljanje nekretninama, ugovorima, finansijskim obavezama i pratećom dokumentacijom u okviru jedinstvenog rešenja.

Sistem podržava tri osnovne korisničke uloge: vlasnika nekretnine, agenciju za izdavanje i stanara. Svaka od navedenih uloga raspolaže skupom funkcionalnosti koje odgovaraju njenoj ulozi u procesu izdavanja, dok je pristup podacima ograničen u skladu sa poslovnim odnosom korisnika prema konkretnoj nekretnini. Na taj način omogućeno je da više korisnika koristi isti informacioni sistem i pristupa zajedničkim poslovnim resursima u skladu sa svojom ulogom, poslovnim odnosima i dodeljenim pravima, uz odgovarajuću izolaciju podataka.

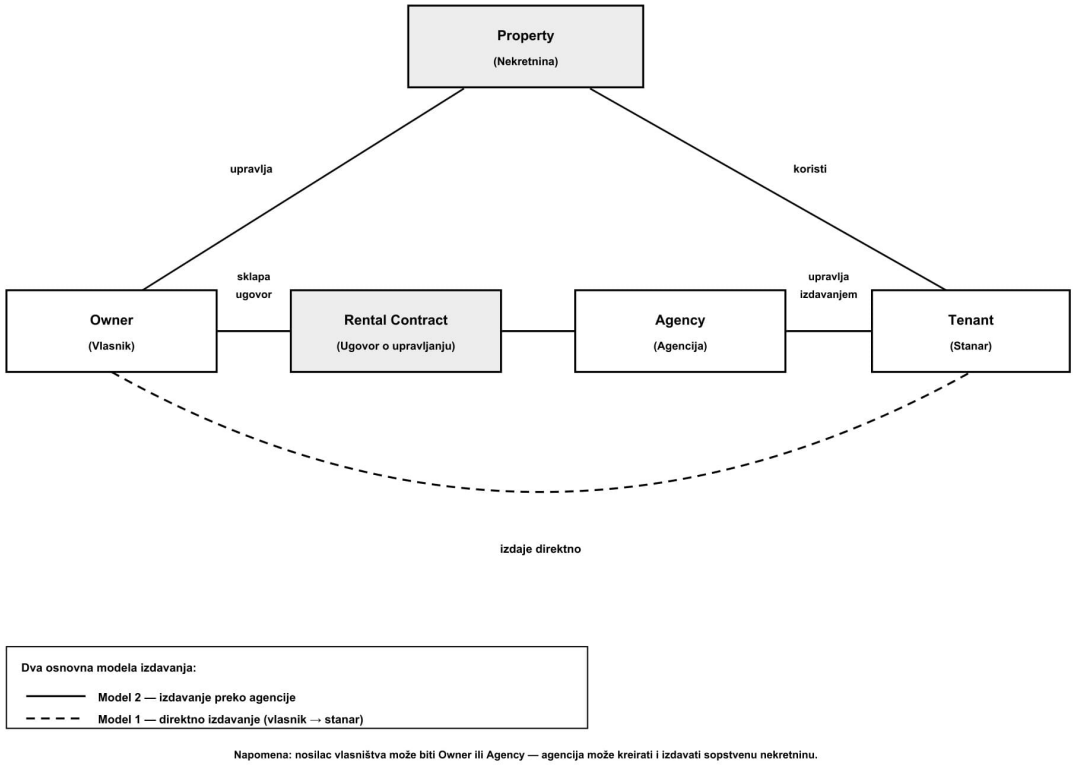
Polazni zahtev prilikom projektovanja bio je da aplikacija podrži različite modele upravljanja nekretninama. Pored mogućnosti evidencije i upravljanja nekretninama koje trenutno nisu izdate, sistem podržava i dva osnovna modela izdavanja. Prvi model podrazumeva direktan odnos između vlasnika i stanara, dok drugi uključuje agenciju koja u ime vlasnika učestvuje u upravljanju procesom izdavanja i komunikaciji sa stanarom. Na taj način aplikacija može biti prilagođena različitim načinima organizacije poslovanja bez potrebe za promenom osnovne arhitekture sistema.

Pored osnovnog upravljanja korisnicima i nekretninama, definisani su i zahtevi koji obuhvataju vođenje ugovora, praćenje istorije zakupa i upravljanje finansijskim obavezama. Sistem omogućava generisanje i čuvanje dokumenata, razmenu priloga između korisnika, kao i obradu različitih zahteva koji nastaju tokom trajanja zakupa. Na ovaj način objedinjeni su najvažniji poslovni procesi koji prate životni ciklus izdavanja jedne nekretnine.

Jedan od ključnih funkcionalnih zahteva odnosi se na omogućavanje različitih nivoa pristupa zajedničkim resursima. Korisnici mogu imati različita prava nad istom nekretninom ili dokumentom, u zavisnosti od poslovnog odnosa koji imaju prema konkretnom resursu.

Na slici 3.1 dat je konceptualni prikaz osnovnih aktera i njihovih poslovnih odnosa u procesu izdavanja nekretnine. Prikazana su dva podržana modela izdavanja:

direktan odnos između vlasnika i stanara i model u kojem agencija učestvuje u upravljanju nekretninom i procesu izdavanja. Dijagram istovremeno pokazuje da su korisnici sa nekretninom povezani kroz različite poslovne odnose, koji su relevantni prilikom određivanja njihovih prava pristupa.

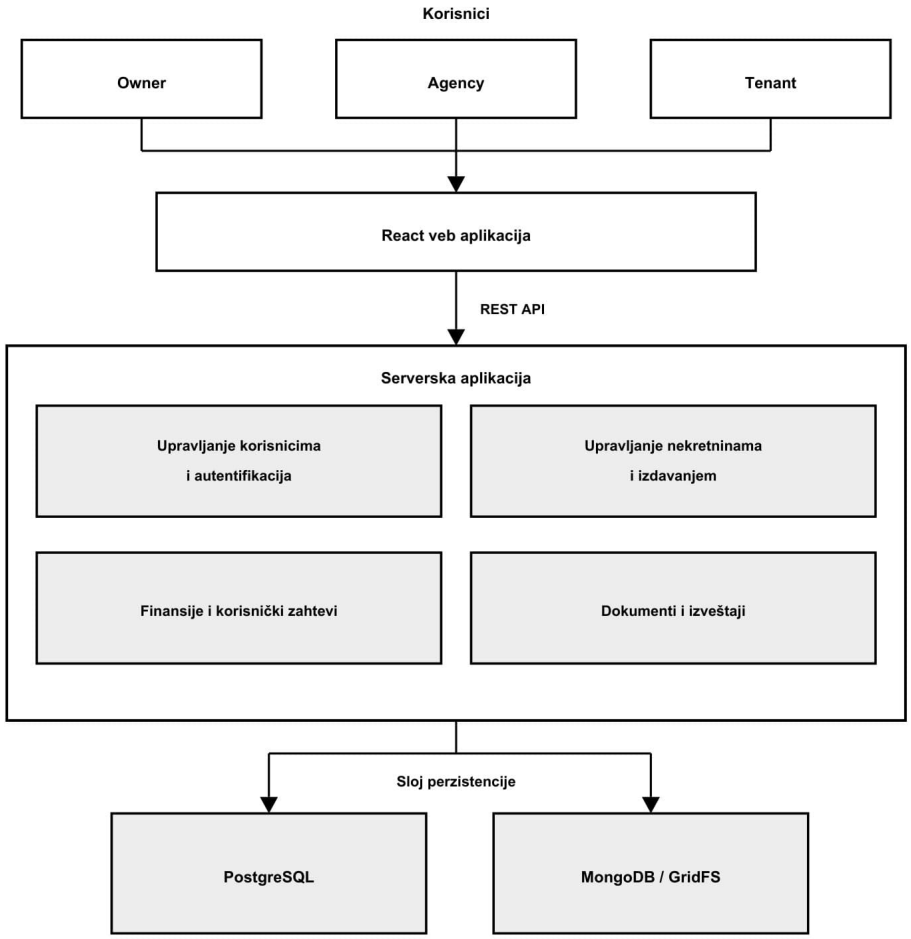


Slika 3.1: Osnovni akteri i poslovni odnosi u sistemu

3.2 Arhitektura sistema

Na slici 3.2 prikazan je sistem i njegove osnovne komponente. Osnovne komponente čine klijentska React aplikacija, REST API preko kojeg se ostvaruje komunikacija sa serverskom aplikacijom, funkcionalne celine serverske strane i sloj perzistencije. Serverska aplikacija obuhvata funkcionalnosti vezane za upravljanje korisnicima i autentifikaciju, nekretninama i izdavanjem, finansijama i korisničkim zahtevima, kao i dokumentima i izveštajima. Sloj perzistencije čine RSUBP PostgreSQL i SUBP MongoDB sa mehanizmom GridFS. U narednim poglavljima biće detaljno opisana arhitektura aplikacije, organizacija serverskog i klijentskog dela si-

stema, model podataka, kao i način realizacije najvažnijih poslovnih funkcionalnosti.



Slika 3.2: Pregled implementiranog sistema

Klijent–server arhitektura

Razvijena aplikacija zasnovana je na klijent–server arhitekturi, u kojoj su klijentski i serverski deo sistema logički razdvojeni i međusobno komuniciraju putem HTTP zahteva.

Korisnici sistemu pristupaju putem veb pregledača, koristeći klijentsku aplikaciju razvijenu u biblioteci React. Korisnički interfejs omogućava prikaz podataka, prikupljanje korisničkih unosa i pokretanje operacija koje se dalje obrađuju na serverskoj strani sistema.

Komunikacija između klijentske i serverske aplikacije ostvaruje se korišćenjem

REST programskog interfejsa. Nakon prijema zahteva, serverska aplikacija obrađuje odgovarajuću poslovnu operaciju, po potrebi komunicira sa RSUBP i SUBP MongoDB, a zatim formira odgovor koji se vraća klijentskoj aplikaciji. Na ovaj način korisnički interfejs ostaje nezavisan od načina na koji su podaci organizovani i obrađeni unutar serverskog dela sistema.

Tok obrade korisničkog zahteva

Obrada korisničkog zahteva u razvijenom sistemu odvija se kroz sled koraka koji obuhvata komunikaciju klijentske aplikacije sa serverskim delom, izvršavanje odgovarajuće poslovne operacije i formiranje odgovora. Konkretni način obrade zavisi od vrste korisničkog zahteva, dok osnovni tok komunikacije između komponenti ostaje isti.

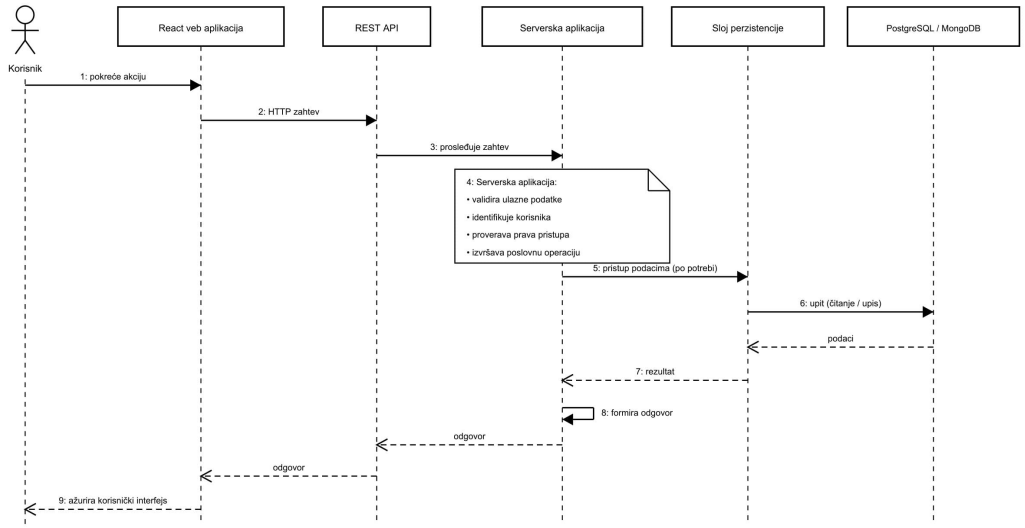
Proces započinje korisničkom akcijom u veb aplikaciji, kao što je prijavljivanje na sistem, pregled podataka ili izvršavanje neke poslovne operacije. Nakon korisničke interakcije klijentska aplikacija formira odgovarajući HTTP zahtev i prosleđuje ga serverskoj aplikaciji putem REST interfejsa. Na taj način klijentski deo ostaje zadužen isključivo za komunikaciju sa korisnikom i razmenu podataka sa serverskim delom sistema.

Po prijemu zahteva serverska aplikacija određuje odgovarajuću poslovnu operaciju koja treba da bude izvršena. U okviru njene obrade vrši se validacija pristiglih podataka, identifikacija korisnika, provera prava pristupa i izvršavanje poslovnih pravila koja odgovaraju konkretnoj funkcionalnosti. Kada je za realizaciju operacije potrebno pristupiti trajno sačuvanim podacima, serverska aplikacija komunicira sa slojem perzistencije, nakon čega se izvršava čitanje ili izmena sadržaja u jednoj ili obe baze podataka, u zavisnosti od prirode informacija koje se obrađuju.

Po završetku poslovne operacije formira se odgovor koji sadrži rezultat izvršavanja zahteva ili odgovarajuću poruku o grešci. Dobijeni odgovor vraća se klijentskoj aplikaciji, koja na osnovu njega ažurira korisnički interfejs i prikazuje odgovarajuće informacije korisniku. Na taj način kompletan tok obrade zahteva ostaje centralizovan u serverskoj aplikaciji, dok je klijentska aplikacija zadužena za prikaz rezultata i dalju interakciju sa korisnikom.

Opisani tok obrade predstavlja opšti obrazac komunikacije između komponenti sistema, nezavisno od toga da li se radi o upravljanju nekretninama, radu sa dokumentima, finansijskim podacima ili obradi korisničkih zahteva.

Na slici 3.3 prikazan je dijagram sekvence opisanog toka obrade korisničkog zahteva, od pokretanja akcije u klijentskoj aplikaciji do formiranja i prikaza odgovora korisniku.



Slika 3.3: Dijagram sekvence korisničkog zahteva

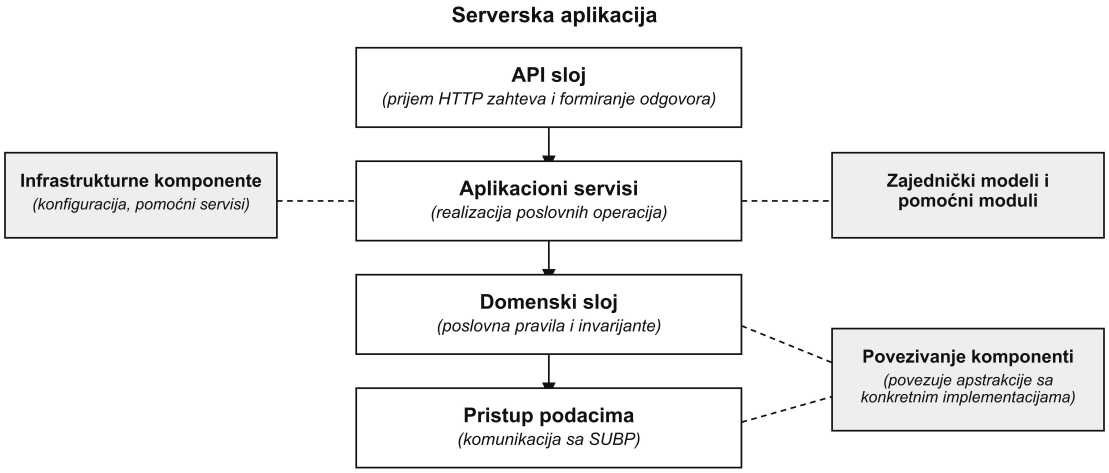
3.3 Arhitektura serverske aplikacije

Serverska aplikacija organizovana je kroz više celina sa jasno razdvojenim odgovornostima. Poslovna logika odvojena je od komunikacije sa spoljnim okruženjem i pristupa podacima. U nastavku su prikazani organizacija projekta, osnovni slojevi serverske aplikacije i arhitektonski obrasci primenjeni tokom razvoja sistema.

Organizacija projekta

Struktura projekta prati logičku podelu serverske aplikacije na API deo, aplikacione servise, domenski sloj i komponente za pristup podacima. Pored njih, izdvojeni su zajednički moduli za konfiguraciju i deljene modele, kao i deo zadužen za povezivanje konkretnih implementacija sa odgovarajućim apstrakcijama prilikom pokretanja aplikacije.

Na slici 3.4 prikazana je pojednostavljena organizacija serverske aplikacije i odnosi između njenih osnovnih celina.



Slika 3.4: Logička organizacija serverske aplikacije i povezivanje njenih osnovnih komponenti

Slojevi aplikacije

Prvi sloj je API sloj, odgovoran za komunikaciju sa klijentskom aplikacijom. Njegova uloga obuhvata prijem HTTP zahteva, prosleđivanje odgovarajućoj poslovnoj operaciji i formiranje odgovora koji se vraća korisniku. Na ovom nivou ne implementiraju se poslovna pravila, već se obavlja koordinacija između korisničkog interfejsa i ostatka sistema.

Iza API sloja nalazi se sloj aplikacionih servisa, koji koordinira izvršavanje poslovnih operacija, poziva odgovarajuću poslovnu funkcionalnost, proverava prava pristupa i po potrebi pokreće komunikaciju sa slojem za pristup podacima. Poslovni entiteti i pravila koja određuju njihovo dozvoljeno stanje i ponašanje definisani su u domenskom sloju. Domenski sloj ne zavisi neposredno od tehnologija korišćenih za komunikaciju i perzistenciju, čime se smanjuje uticaj infrastrukturnih izmena na poslovna pravila sistema.

Za rad sa trajno sačuvanim podacima zadužen je sloj pristupa podacima. Njegova odgovornost je komunikacija sa sistemima za upravljanje bazama podataka i izvršavanje operacija čitanja, upisa i ažuriranja podataka. Izdvajanjem ovih aktivnosti iz domenskog sloja i sloja aplikacionih servisa postiže se jasnije razdvajanje odgovornosti i smanjuje zavisnost ostatka aplikacije od konkretne implementacije perzistencije.

Pored navedenih slojeva, sistem sadrži i infrastrukturne komponente za konfi-

guraciju i deljene modele. Povezivanje slojeva u celinu, odnosno dodela konkretnih implementacija odgovarajućim apstrakcijama, odvija se u delu projekta zaduženom za sklapanje aplikacije pri pokretanju.

Primenjeni arhitektonski obrasci

Prilikom razvoja serverske aplikacije primenjeno je više arhitektonskih obrazaca čiji je cilj bio da se postigne jasna organizacija sistema i smanji međusobna zavisnost pojedinih komponenti.

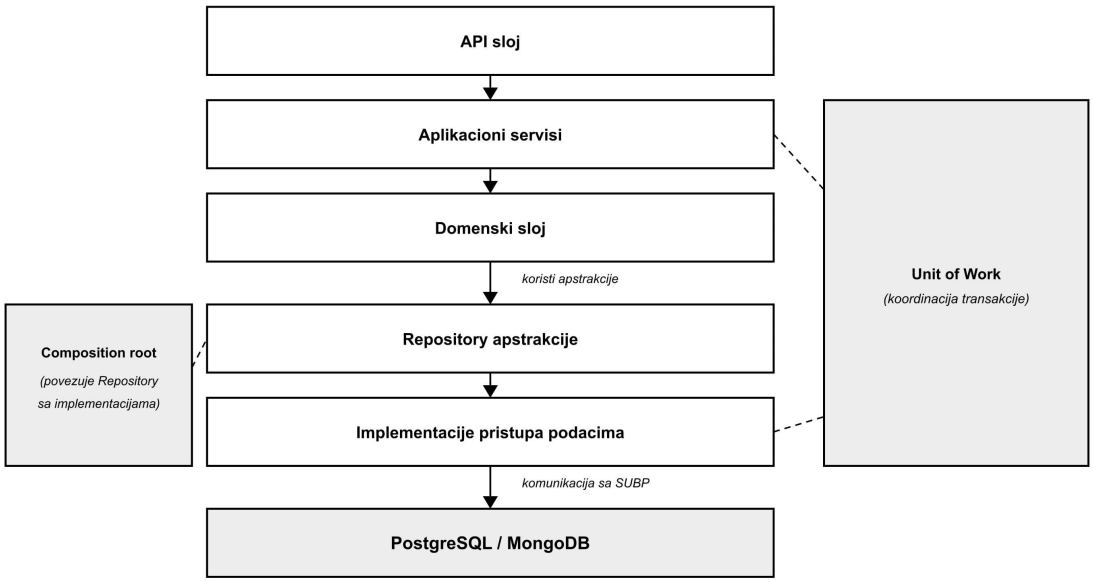
Jedan od osnovnih obrazaca primenjenih u sistemu jeste *Repository*, kojim je pristup trajno sačuvanim podacima izdvojen iz poslovne logike [15, 28]. Poslovne operacije ne komuniciraju neposredno sa SUBP, već koriste apstrakcije koje obezbeđuju jedinstven način rada sa poslovnim entitetima. Na taj način poslovna logika ostaje nezavisna od konkretne implementacije perzistencije, dok su detalji komunikacije sa SUBP koncentrisani u posebnom delu sistema.

Radi koordinacije operacija nad relacionim podacima koje pripadaju istoj poslovnoj celini primenjen je i obrazac *Unit of Work* [15, 28]. Njegova implementacija objedinjuje rad repozitorijuma koji dele istu sesiju u okviru jedne transakcije i omogućava kontrolisano potvrđivanje ili poništavanje svih izvršenih izmena kao celine.

U implementaciji je primenjen i princip zavisnosti od apstrakcija, pri čemu poslovna logika koristi definisane interfejse umesto konkretnih implementacija infrastrukturnih komponenti [22]. Na taj način pojedini delovi sistema ostaju međusobno slabo povezani, što omogućava jednostavnije testiranje i zamenu pojedinih implementacija bez izmene poslovnih pravila. Ovakav pristup posebno je značajan u aplikacijama koje kombinuju različite sisteme za perzistenciju podataka i više infrastrukturnih servisa.

Za povezivanje apstrakcija sa konkretnim implementacijama u projektu je izdvojen *composition root*, odnosno deo zadužen za povezivanje komponenti aplikacije prilikom njenog pokretanja [42]. Zajedno sa mehanizmom ubrizgavanja zavisnosti koji obezbeđuje razvojni okvir, on omogućava da komponente dobiju potrebne implementacije bez njihovog neposrednog kreiranja u poslovnoj logici.

Na slici 3.5 prikazani su primenjeni obrasci Repository i Unit of Work, kao i način povezivanja poslovnog sloja sa infrastrukturnim komponentama. Organizacija podataka i izbor sistema za njihovu perzistenciju predmet su narednog odeljka.



Slika 3.5: Primena obrazaca Repository i Unit of Work i povezivanje komponenti serverske aplikacije

3.4 Model podataka i perzistencija

Model podataka razvijene aplikacije obuhvata različite vrste informacija, od strukturisanih poslovnih entiteta i njihovih međusobnih odnosa do dokumenata i evidencija promenljive strukture. Zbog toga su u sistemu kombinovani relacioni i dokumentni pristup perzistenciji, pri čemu svaki od njih ima jasno određenu ulogu.

U nastavku će najpre biti prikazan relacioni model podataka, zatim organizacija dokumentnog modela, a na kraju razlozi zbog kojih je u sistemu primenjena kombinacija dve različite tehnologije za perzistenciju podataka.

Relacioni model podataka

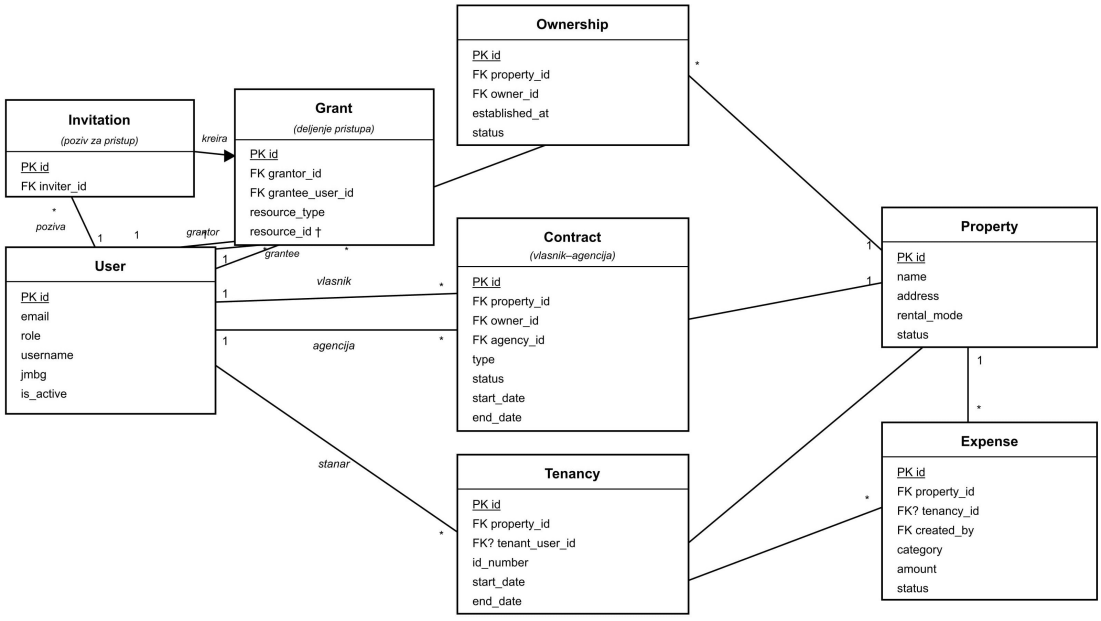
Relacioni model predstavlja osnovu za čuvanje poslovnih podataka koji opisuju korisnike, nekretnine, ugovorne odnose i ostale entitete neophodne za realizaciju procesa izdavanja.

Centralno mesto u modelu zauzima entitet nekretnine, oko koga su organizovani ostali poslovni podaci. Sa nekretninom su povezani podaci o vlasništvu, ugovornim odnosima, zakupima, finansijskim obavezama i ostalim poslovnim informacijama

koje nastaju tokom njenog životnog ciklusa. Na taj način svi podaci koji opisuju jednu nekretninu ostaju međusobno povezani kroz jasno definisane odnose.

Pored podataka o nekretninama, model obuhvata i korisnike sistema, njihove međusobne poslovne odnose, kao i informacije koje određuju način korišćenja aplikacije. Posebna pažnja posvećena je modelovanju odnosa između vlasnika, agencija i stanara, budući da isti korisnik može učestvovati u različitim poslovnim procesima, dok jedna nekretnina tokom vremena može prolaziti kroz više faza korišćenja i izdavanja. Relacioni model obuhvata i podatke koji omogućavaju deljenje pristupa nekretninama, uključujući pozivnice i eksplicitno dodeljena prava. Njihova uloga u modelu kontrole pristupa detaljnije je obrađena u odeljku 3.6. Organizacijom podataka kroz međusobno povezane entitete omogućeno je i očuvanje istorije značajnih poslovnih odnosa.

Na slici 3.6 prikazan je pojednostavljen logički model relacionog dela podataka razvijenog sistema. Radi preglednosti prikazani su najvažniji poslovni entiteti, njihovi osnovni atributi i međusobne veze, dok pomoćne tabele i tehnički detalji implementacije nisu uključeni. Centralni entitet predstavlja nekretnina, koja je sa ostalim entitetima povezana kroz odnose prikazane u tabeli 3.1.



*Legenda: PK – primarni ključ (podvučen), FK – strani ključ, FK? – opcioni strani ključ. † resource_id u Grant entitetu je polimorfna referenca — nema FK na nivou baze. Kardinalnosti: 1 (tačno jedan), * (jedan ili više), 0..1 (opciono). Zahtevi (Request) čuvaju se u MongoDB i nisu deo relacionog modela.*

Slika 3.6: Pojednostavljeni model relacionog dela podataka

Tabela 3.1: Opis relacija logičkog modela relacionog dela podataka

#	**Relacija**	**Kard.**	**Opis**
1	User → Ownership	1 : *	Korisnik može biti vlasnik više nekretnina.
2	Ownership → Property	* : 1	Svaki zapis o vlasništvu vezan je za jednu nekretninu.
3	User *(vlasnik)* → Contract	1 : *	Vlasnik može potpisati više ugovora kao vlasnička strana.
4	User *(agencija)* → Contract	1 : *	Agencija može biti strana u više ugovora.
5	Contract → Property	* : 1	Svaki ugovor odnosi se na jednu nekretninu.
6	User *(stanar)* → Tenancy	0..1 : *	Zakup može postojati i bez registrovanog korisnika — korisnički nalog stanara je opcion.
7	Tenancy → Property	* : 1	Svaki zakup odnosi se na jednu nekretninu.
8	Property → Expense	1 : *	Nekretnina može imati više evidentiranih troškova.
9	Tenancy → Expense	0..1 : *	Trošak opciono pripada zakupu — može biti evidentiran i bez tekućeg zakupa.
10	User → Invitation	1 : *	Korisnik može uputiti više pozivnica za pristup resursima.
11	Invitation → Grant	1 : 0..*	Prihvatanjem pozivnice kreira se po jedan zapis o pravima za svaku deljenu funkcionalnost.
12	User *(grantor)* → Grant	1 : *	Korisnik koji dodeljuje pristupna prava drugom korisniku.
13	User *(grantee)* → Grant	1 : *	Korisnik koji prima pristupna prava.

Dokumentni model podataka

Jednu grupu podataka čine dokumenti koji nastaju ili se razmenjuju tokom korišćenja sistema. Pored automatski generisanih PDF dokumenata, aplikacija omogućava korisnicima i čuvanje dodatnih priloga koji prate poslovne procese. Binarni sadržaj dokumenata i priloga čuva se korišćenjem GridFS mehanizma, čime je skladištenje datoteka izdvojeno iz relacionog modela podataka. Poslovni zapisi i sam binarni sadržaj dokumenta na taj način ostaju logički razdvojeni.

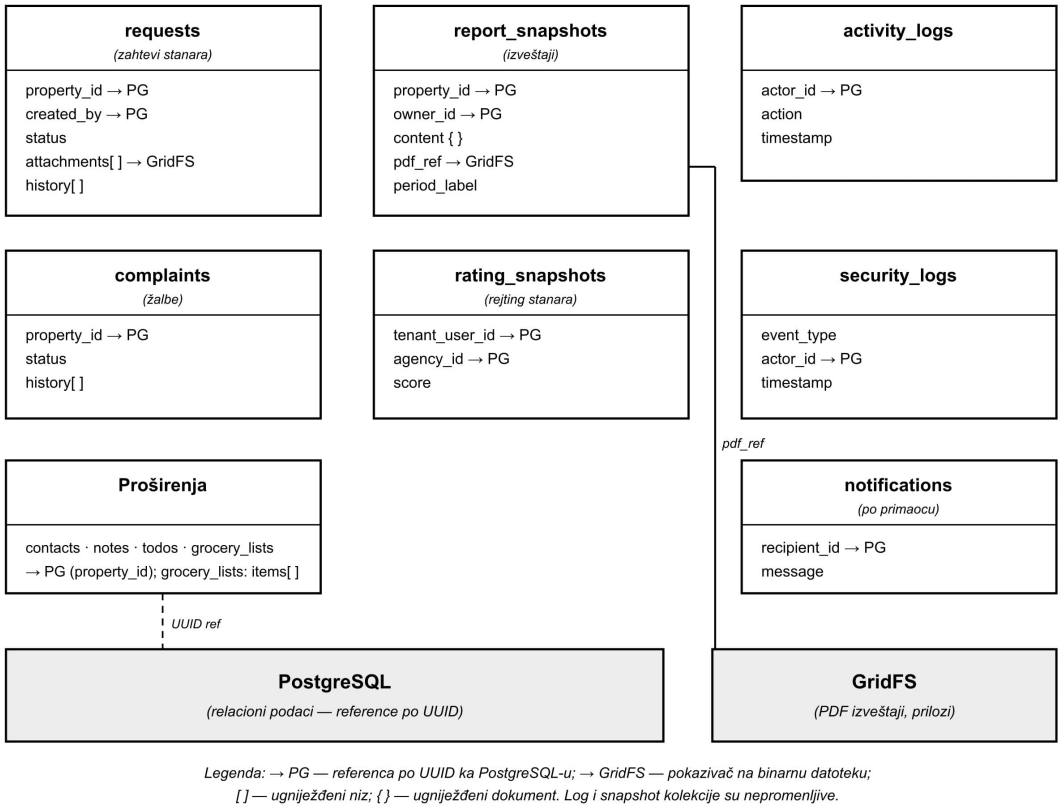
Dokumentni model koristi se i za čuvanje pojedinih evidencija koje nastaju kao rezultat rada korisnika u sistemu. Ovakvi zapisi predstavljaju hronološku evidenciju događaja i ne učestvuju direktno u realizaciji poslovnih transakcija. Njihova izdvojena organizacija omogućava čuvanje istorije aktivnosti bez opterećivanja relacionog

modela podataka.

Dokumentni model koristi se i za pojedine poslovne evidencije čija struktura i način korišćenja odgovaraju dokumentnom pristupu, među kojima su zahtevi, žalbe i preseci stanja (engl. snapshot) koji nastaju prilikom generisanja izveštaja.

Prilikom generisanja pojedinih izveštaja čuva se i presek stanja podataka u trenutku nastanka dokumenta. Time se omogućava očuvanje istorijskog sadržaja izveštaja i u slučaju naknadnih promena poslovnih podataka.

Pojednostavljeni prikaz organizacije dokumentnog dela podataka i njegovih osnovnih elemenata dat je na slici 3.7.



Slika 3.7: Pojednostavljeni model dokumentnog dela podataka

Kombinovanje relacionog i dokumentnog pristupa

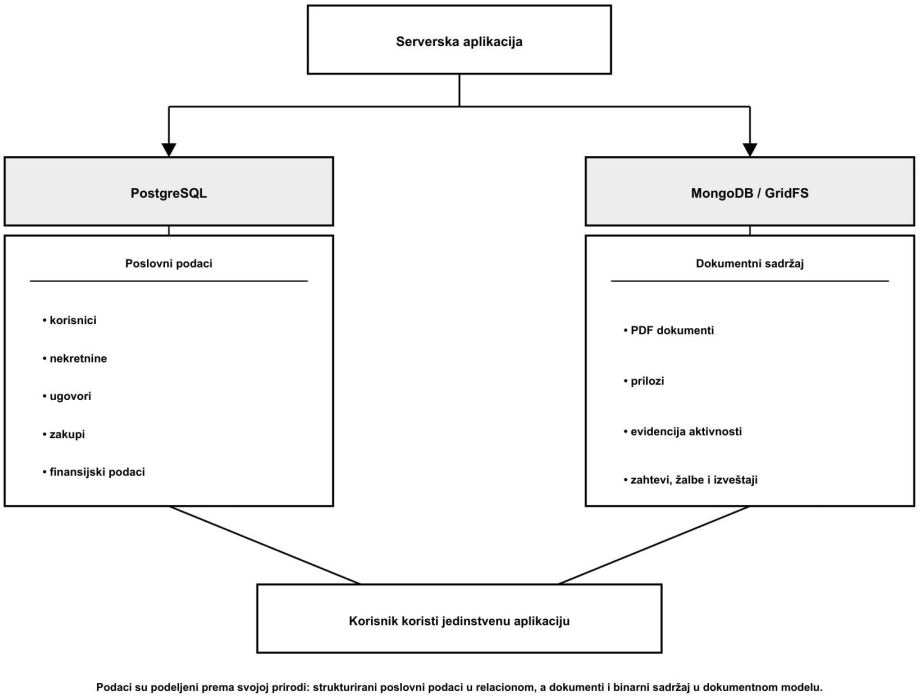
Za relacioni deo perzistencije koristi se PostgreSQL, dok se pristup podacima ostvaruje preko SQLAlchemy biblioteke. PostgreSQL predstavlja SUBP u kojem je

realizovan relacioni model.

Za dokumentni deo perzistencije koristi se MongoDB, kojem serverska aplikacija pristupa preko odgovarajućeg drajvera. MongoDB predstavlja SUBP u kojem je realizovan dokumentni model, dok se GridFS koristi kao mehanizam za čuvanje većih datoteka.

Korišćenje dve baze podataka ne znači postojanje dva nezavisna informaciona sistema. Naprotiv, sa stanovišta korisnika aplikacija predstavlja jedinstvenu celinu, dok serverska aplikacija koordinisano koristi oba sistema za perzistenciju u zavisnosti od vrste podataka koja se obrađuje.

Na slici 3.8 prikazana je podela podataka između relacionog i dokumentnog modela, kao i njihova povezanost u okviru jedinstvenog sistema perzistencije.



Slika 3.8: Podela podataka između relacionog i dokumentnog modela

3.5 Autentifikacija korisnika

Budući da aplikacija omogućava pristup podacima različitih korisnika i podržava više poslovnih uloga, bilo je neophodno obezbediti mehanizam kojim se pouzdano potvrđuje identitet korisnika pre izvršavanja bilo koje poslovne operacije. U razvije-

nom sistemu autentifikacija je organizovana tako da korisnik nakon uspešne prijave ostvaruje pristup zaštićenom delu sistema, dok se kontrola prava nad pojedinačnim funkcionalnostima i resursima obavlja kroz mehanizme autorizacije opisane u narednom poglavlju.

Proces autentifikacije obuhvata registraciju korisnika, prijavu na sistem i upravljanje korisničkom sesijom.

Registracija i prijava korisnika

Korišćenje sistema započinje kreiranjem korisničkog naloga. Nakon uspešne registracije korisnik se prijavljuje unosom svojih podataka za prijavu. Serverska aplikacija proverava ispravnost dostavljenih podataka i, u slučaju uspešne autentifikacije, omogućava pristup zaštićenom delu sistema.

U okviru procesa prijave posebna pažnja posvećena je zaštiti lozinki koje se ne čuvaju u izvornom obliku, već se pre skladištenja obrađuju odgovarajućom funkcijom za heširanje. U implementaciji je za tu namenu korišćen algoritam Argon2. Time se dodatno povećava bezbednost korisničkih naloga u slučaju neovlašćenog pristupa bazi podataka. JMBG se prikuplja isključivo od fizičkih lica (vlasnika i stanara), dok se za agenciju kao pravno lice koristi poreski identifikacioni broj (PIB), pa ovaj podatak nije obavezan za svakog korisnika. Za fizička lica JMBG služi kao jedinstveni identifikator i koristi se pri generisanju ugovora o zakupu, u kojima je uobičajen element. Vrednost se čuva u relacionoj bazi kao deo korisničkog profila, uz kontrolu pristupa kroz autorizacioni model, i ne prenosi se u autentifikacionim tokenima.

Upravljanje korisničkom sesijom

Prilikom uspešne prijave korisniku se izdaje par tokena, pristupni i osvežavajući token, koji imaju različite uloge u održavanju autentifikovanog pristupa. Pristupni token, koji ima kraći rok važenja, klijentska aplikacija prosleđuje uz svaki zahtev namenjen serverskoj aplikaciji, dok osvežavajući token, sa dužim rokom važenja, služi za izdavanje novog pristupnog tokena. Na osnovu dostavljenog pristupnog tokena serverska aplikacija identifikuje korisnika i omogućava obradu zahteva, dok se provera konkretnih prava pristupa nad traženim resursima obavlja u narednoj fazi obrade.

Kako bi se povećala bezbednost sistema i omogućilo jednostavnije upravljanje korisničkim sesijama, primenjen je mehanizam kojim se pristupni token može obnoviti pomoću osvežavajućeg tokena, bez potrebe za ponovnom prijavom korisnika. Prilikom svakog osvežavanja izdaje se novi osvežavajući, a prethodni token više ne može biti korišćen za regularno osvežavanje, čime se dodatno ograničava mogućnost njegove zloupotrebe u slučaju da bude kompromitovan.

Korisnička sesija prestaje da se obnavlja nakon odjave ili kada više ne postoje uslovi za važeće osvežavanje autentifikacionih tokena. Za nastavak korišćenja zaštićenih funkcionalnosti nakon isteka važećeg pristupa potrebna je nova uspešna autentifikacija korisnika.

3.6 Autorizacija i segmentacija podataka

Nakon uspešne autentifikacije korisnika potrebno je odrediti kojim podacima i operacijama on može pristupiti. Ovaj problem je posebno izražen u razvijenom sistemu, jer više korisnika može biti povezano sa istom nekretninom, ali po različitim osnovama i sa različitim pravima. Sama korisnička uloga zbog toga nije dovoljna za donošenje svih odluka o pristupu. Pored nje, potrebno je uzeti u obzir odnos korisnika prema konkretnom resursu i poslovni kontekst u kome se zahtev izvršava.

Kontrola pristupa u sistemu zato kombinuje korisničke uloge sa podacima koji proizlaze iz vlasništva, ugovornih odnosa, zakupa i eksplicitno dodeljenih prava. Na osnovu ovih informacija određuje se da li korisnik može izvršiti traženu operaciju, dok isti autorizacioni kontekst predstavlja osnov i za određivanje skupa podataka koji mu je dostupan. Autorizacija i segmentacija podataka na taj način čine povezane mehanizme: prvi kontroliše dozvoljene operacije nad konkretnim resursima, a drugi ograničava podatke koje korisnik može da vidi u skladu sa svojim pravima.

U nastavku je najpre opisan model kontrole pristupa, zatim postupak donošenja pojedinačne autorizacione odluke, a na kraju način na koji se isti kontekst koristi za segmentaciju podataka između korisnika sistema.

Model kontrole pristupa

Polaznu osnovu modela kontrole pristupa čine tri korisničke uloge definisane u sistemu: vlasnik, agencija i stanar. Uloga određuje osnovni skup mogućnosti korisnika, ali ne daje sama po sebi pravo pristupa svakom resursu koji pripada odgovarajućoj

kategoriji. Na primer, činjenica da korisnik ima ulogu vlasnika ne znači da može upravljati proizvoljnom nekretninom u sistemu. Za donošenje odluke potrebno je utvrditi njegov odnos prema konkretnoj nekretnini i povezanim poslovnim podacima.

U okviru modela opisanog u poglavlju 2.7, subjekat predstavlja autentifikovanog korisnika sistema, dok resurs predstavlja konkretan poslovni objekat nad kojim se pristup ostvaruje. Operacija označava akciju koju korisnik pokušava da izvrši nad tim resursom, dok odnos opisuje poslovnu vezu korisnika sa konkretnim resursom, kao što su vlasništvo, ugovorni odnos, zakup ili eksplicitno dodeljeno pravo.

Prvi takav odnos jeste vlasništvo. Ono povezuje korisnika sa nekretninom nad kojom ima vlasnička prava i predstavlja jedan od osnovnih elemenata konteksta na osnovu kog se određuje dozvoljeni pristup. Time se globalna korisnička uloga dopunjuje informacijom o konkretnom resursu: sistem ne proverava samo da li je korisnik vlasnik, već i da li je povezan sa nekretninom nad kojom pokušava da izvrši određenu operaciju.

U modelu izdavanja preko agencije značajan deo konteksta čini ugovorni odnos između vlasnika i agencije. Postojanje odgovarajućeg ugovora povezuje agenciju sa nekretninom i predstavlja osnov za izvršavanje operacija koje joj pripadaju u okviru tog poslovnog odnosa. Na taj način prava agencije nisu izvedena samo iz njene uloge, već i iz konkretne veze sa vlasnikom i nekretninom kojom upravlja.

Slično tome, zakup povezuje stanara sa nekretninom koju koristi. Informacija o postojanju odgovarajućeg odnosa zakupa omogućava da se pristup stanara ograniči na resurse koji se odnose na njegov konkretan zakup. Time se sprečava da korisnička uloga stanara sama po sebi predstavlja osnov za pristup podacima drugih nekretnina ili drugih korisnika.

Pored prava koja proizlaze iz osnovnih poslovnih odnosa, sistem podržava i eksplicitno deljenje pristupa. Ovaj mehanizam omogućava da korisniku budu dodeljena određena prava nad konkretnim resursom bez promene njegove osnovne korisničke uloge. Takav pristup koristi se i kada je potrebno uključiti korisnika koji nema samostalan vlasnički ili ugovorni odnos sa nekretninom, na primer člana porodice kojem se omogućava pristup određenom delu podataka. U sistemu se ovakva dodela predstavlja dozvolom, pri čemu se prava vezuju za konkretnog korisnika i resurs.

Ovakav model čini osnov za donošenje konkretne autorizacione odluke. Pre nego što serverska aplikacija dozvoli izvršavanje zaštićene operacije, relevantne informacije o korisniku i resursu objedinjuju se u kontekst na osnovu kojeg se proverava da

li tražena akcija treba da bude dozvoljena ili odbijena.

Donošenje autorizacione odluke

Model kontrole pristupa opisan u prethodnom odeljku definiše informacije koje utiču na prava korisnika, ali je za obradu svakog zaštićenog zahteva potrebno te informacije objediniti i na osnovu njih doneti konkretnu odluku. U razvijenom sistemu ovaj postupak razdvojen je na formiranje konteksta pristupa i samu evaluaciju prava. Time je prikupljanje podataka o odnosu korisnika prema resursu odvojeno od logike koja odlučuje da li je zahtev dozvoljen.

Za potrebe autorizacije formira se objekat AccessContext, koji opisuje relevantan odnos trenutno prijavljenog korisnika prema resursu nad kojim želi da izvrši operaciju. Kontekst, između ostalog, obuhvata informacije o vlasništvu, aktivnom ugovoru ili zakupu, odnosu agencije prema ugovoru, načinu upravljanja nekretninom i eventualno dodeljenim pravima. Prilikom njegovog formiranja uzimaju se u obzir aktivni poslovni odnosi i dozvole. Sam postupak izgradnje konteksta ne donosi odluku o pristupu, već priprema podatke potrebne za njeno donošenje.

Formiranje konteksta pristupa prikazano je u listingu 3.1. Metoda `resolve()` iz aktivnih poslovnih odnosa (vlasništvo, ugovor, zakup i dodeljena prava) sklapa objekat `AccessContext`, ali sama ne donosi odluku o pristupu.

```
async def resolve(self, identity: Identity, resource: Resource) -> AccessContext:
    property_id = self._target_property_id(resource)

    # grantovi su vezani za resurs
    grants = tuple(
        await self._grants.list_active_for_principal_and_resource(
            identity.user_id, resource
        )
    )
    # agencija na ugovoru moze da ga vidi i prihvati/odbije dok je jos u stanju cekanja
    is_contract_agency = await self._is_contract_agency(identity, resource)

    if property_id is None:
        # npr. kreiranje novog resursa: nema odnosa prema nekretnini
        return AccessContext(is_contract_agency=is_contract_agency, grants=grants)

    is_owner = await self._is_owner(identity, property_id)
```

```
18      has_contract = await self._has_active_contract(identity, property_id)
19      has_tenancy = await self._has_active_tenancy(identity, property_id)
20      is_agency_managed = await self._is_agency_managed(property_id)
21
22      return AccessContext(
23          is_owner=is_owner,
24          has_active_contract=has_contract,
25          has_active_tenancy=has_tenancy,
26          is_contract_agency=is_contract_agency,
27          property_is_agency_managed=is_agency_managed,
28          grants=grants,
29      )
```

Listing 3.1: Formiranje konteksta pristupa iz aktivnih poslovnih odnosa.

Formirani kontekst, zajedno sa identitetom korisnika, vrstom resursa i zahtevanom akcijom, prosleđuje se centralnoj autorizacionoj funkciji *authorize()*. Ova funkcija ne pristupa bazama podataka niti izvršava dodatne ulazno-izlazne operacije, već odluku donosi isključivo na osnovu prosleđenih informacija. Takvo razdvajanje doprinosi predvidljivosti autorizacione logike i olakšava njeno nezavisno testiranje.

Prilikom evaluacije razmatraju se različiti osnovi po kojima korisnik može ostvariti pristup konkretnom resursu: vlasništvo, odgovarajući ugovorni odnos, aktivni zakup i eksplicitno dodeljena prava. Uloga korisnika ima ograničenu funkciju i u trenutnoj implementaciji predstavlja neposredan osnov za kreiranje nekretnine kod vlasnika i agencije. Pristup postojećim poslovnim resursima, međutim, zavisi od konkretnog odnosa korisnika prema resursu, zbog čega korisnici sa istom ulogom ne moraju imati jednaka prava pristupa.

Rezultat evaluacije nije ograničen samo na binarnu odluku da li je operacija dozvoljena. Kada postoji odgovarajući osnov za pristup, određuje se i njegov obim. U implementaciji se razlikuju četiri nivoa obima pristupa koji prate dozvolu (peta vrednost označava izostanak pristupa): *FULL*, koji označava puni pristup resursu, *MANAGED*, koji obuhvata pristup resursima kojima korisnik upravlja, *SELF*, kojim se pristup ograničava na sopstvene podatke korisnika, i *LIMITED*, koji odgovara ograničenom pristupu zasnovanom na eksplicitno dodeljenim pravima. Ovi nivoi mogu se uočiti i u primerima iz tabele 3.2: vlasnik nekretnine ostvaruje puni pristup (*FULL*), agencija upravljački pristup (*MANAGED*), stanar pristup sopstvenim zapisima (*SELF*). Pravo pristupa dodeljeno putem pozivnice uvek nosi obim *LIMITED*. Pošto pozvani korisnik nad datom nekretninom u tom toku nema drugi

aktivan osnov za pristup, autorizaciona funkcija za njega vraća upravo ograničeni obim pristupa *LIMITED*).

Ovakvo ponašanje prati princip podrazumevane zabrane pristupa: odsustvo odgovarajućeg osnova ne dovodi do implicitne dozvole, već do odluke ZABRANI (*DENY*). U suprotnom se formira odluka DOZVOLI (*ALLOW*), zajedno sa obimom pristupa koji proizlazi iz utvrđenih odnosa. Centralizovanjem ovog postupka izbegava se da pojedinačne poslovne funkcionalnosti samostalno i na različite načine tumače pravila pristupa.

Način na koji servisni sloj sprovodi autorizacionu odluku prikazan je u listingu 3.2.

```
async def decide(self, identity: Identity, resource: Resource, action: Action) ->
     Decision:
    context = await self._resolver.resolve(identity, resource)
    return authorize(identity, resource.type, action, context)

async def enforce(
    self,
    identity: Identity,
    resource: Resource,
    action: Action,
    *,
    hide_existence: bool = False,
) -> Scope:
    decision = await self.decide(identity, resource, action)
    if not decision.allowed:
        if hide_existence:
            raise NotFoundError(decision.reason or "resource not found")
        raise ForbiddenError(decision.reason or "not authorized")
    assert decision.scope is not None
    return decision.scope
```

Listing 3.2: Sprovođenje autorizacione odluke.

Metoda `decide()` povezuje resolver i politiku: formira kontekst i poziva `authorize()`. Metoda `enforce()` zatim sprovodi ishod tako što na dozvolu vraća obim pristupa (`Scope`), koji se dalje koristi za segmentaciju podataka, a na zabranu podiže grešku. Za tokove u kojima bi i samo postojanje resursa predstavljalo curenje informacije, servisni sloj poziva proveru sa uključenim skrivanjem postojanja (`hide_existence`), pa se umesto odgovora 403 vraća 404 pri čemu korisnik ne može ni da zaključi da resurs postoji.

Razdvajanje formiranja AccessContext objekta od funkcije *authorize* ima značaj i za segmentaciju podataka. Autorizaciona odluka, pored odgovora na pitanje da li korisnik sme da pristupi određenom resursu, daje informaciju o obimu pristupa koji mu pripada. Upravo ta informacija predstavlja osnov za određivanje skupa podataka koji korisnik može da vidi, što će biti detaljnije razmotreno u narednom odeljku.

Sama odluka izračunava se u funkciji prikazanoj u listingu 3.3. Ona kao ulaz dobija identitet korisnika, tip resursa, zahtevanu akciju i formirani kontekst pristupa, a kao rezultat vraća odluku o dozvoli ili zabrani, zajedno sa odgovarajućim obimom pristupa.

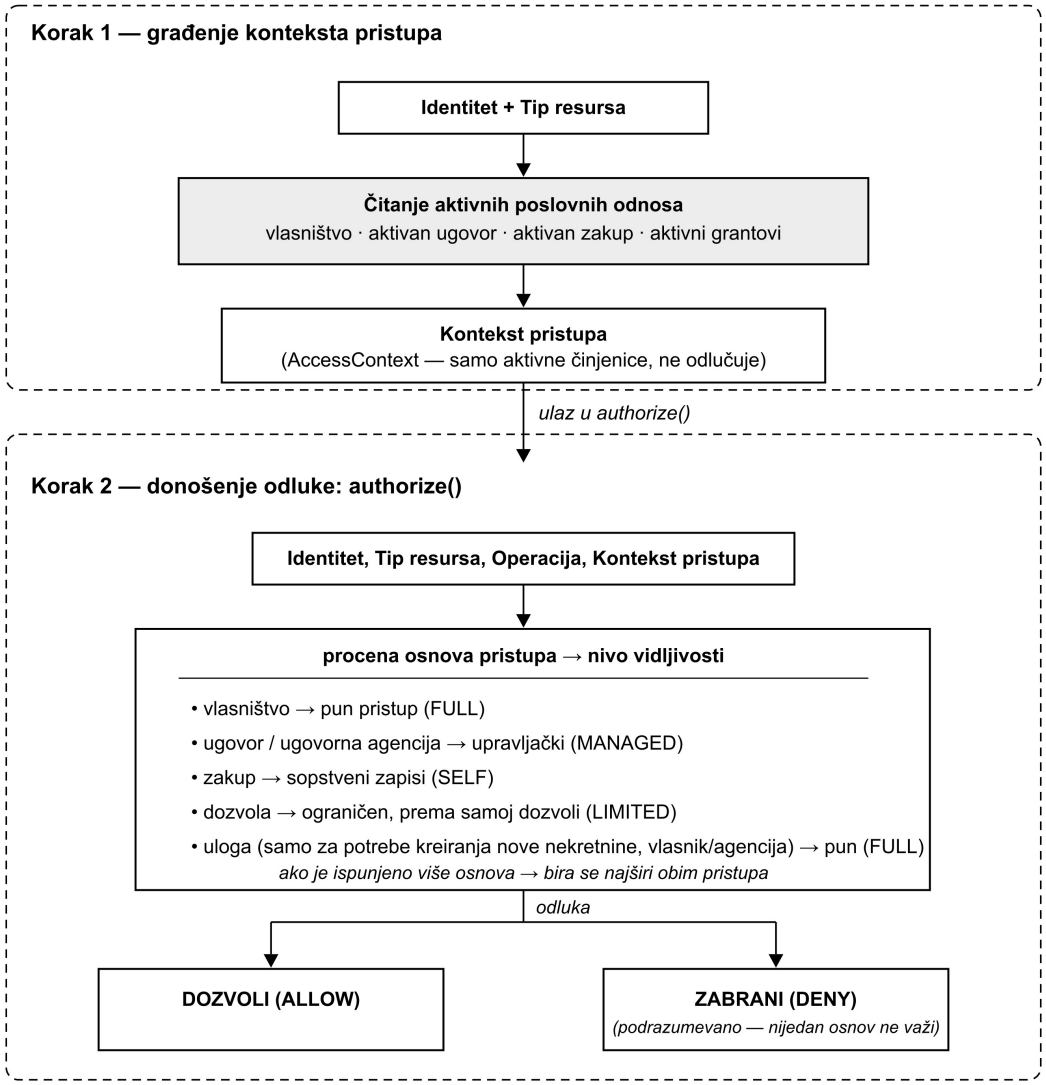
```
def authorize(
    identity: Identity,
    resource_type: ResourceType,
    action: Action,
    context: AccessContext,
) -> Decision:
    candidates: list[Scope] = []

    if context.is_owner and action in _OWNERSHIP_ACTIONS.get(resource_type,
    frozenset()):
        # u agencijskom rezimu vlasnistvo ne daje prava nad troskovima (EXPENSE)
        if not (resource_type is RT.EXPENSE and context.
    property_is_agency_managed):
            candidates.append(Scope(Visibility.FULL, Basis.OWNERSHIP))

    if context.has_active_contract and action in _CONTRACT_ACTIONS.get(
    resource_type, frozenset()):
        candidates.append(Scope(Visibility.MANAGED, Basis.CONTRACT))

    if context.is_contract_agency and action in _CONTRACT_AGENCY_ACTIONS.get(
        resource_type, frozenset()
    ):
        candidates.append(Scope(Visibility.MANAGED, Basis.CONTRACT))

    if context.has_active_tenancy and action in _TENANCY_ACTIONS.get(
    resource_type, frozenset()):
        candidates.append(Scope(Visibility.SELF, Basis.TENANCY))

    for grant in context.grants:
        if action in grant.allowed_actions:
            candidates.append(Scope(grant.visibility, Basis.GRANT))
```

```
28
29      if action in _ROLE_ACTIONS.get(identity.role, {}).get(resource_type,
        frozenset()):
30          candidates.append(Scope(Visibility.FULL, Basis.ROLE))
31
32      if candidates:
33          return Decision(allowed=True, scope=_broadest(candidates))
34      return Decision(
35          allowed=False,
36          reason=f"no active basis authorizes {action.value} on {resource_type.
        value}",
37      )
```

Listing 3.3: Centralna autorizaciona funkcija.

Za svaku osnovu po kojoj korisnik može ostvariti pristup dodaje se kandidat sa pripadajućim obimom vidljivosti. Ukoliko postoji više osnova, bira se najširi obim a ukoliko ne postoji nijedna, primenjuje se podrazumevana zabrana.

Na slici 3.9 prikazan je tok donošenja autorizacione odluke, od formiranja konteksta pristupa do evaluacije zahtevanog prava. U prvom koraku prikupljaju se podaci potrebni za formiranje objekta AccessContext, dok se zatim na osnovu identiteta korisnika, vrste resursa, zahtevane akcije i formiranog konteksta donosi odluka o dozvoli ili zabrani pristupa i određuje odgovarajući obim pristupa.



Slika 3.9: Donošenje autorizacione odluke

Naredna tabela prikazuje reprezentativne primere donošenja autorizacione odluke na osnovu subjekta, resursa, zahtevane operacije i odnosa prema konkretnom resursu. Primeri ilustruju kako različiti odnosi prema resursu utiču na dozvoljene operacije i obim pristupa.

Tabela 3.2: Primeri autorizacione odluke Pristup(Subjekat, Resurs, Operacija, Odnos) → {dozvoli, zabrani}

Subjekat	**Resurs**	**Operacija**	**Odnos**	**Rezultat**
Vlasnik	Nekretnina	izmena	Ima pravo vlasništva nad nekretninom	**DOZVOLI** — pun (vlasnički) pristup
Vlasnik	Nekretnina *(tuđa)*	izmena	Ne postoji nijedan aktivan odnos prema toj nekretnini	**ZABRANI** — nema aktivne osnove
Vlasnik	Trošak	kreiranje	Vlasnik nekretnine koja se izdaje preko agencije	**ZABRANI** — u agencijskom režimu vlasnik nema prava nad troškovima
Agencija	Trošak	kreiranje	Upravlja nekretninom preko aktivnog ugovora	**DOZVOLI** — upravljački pristup
Stanar	Obaveza	plaćanje	Ima aktivan zakup nad tom nekretninom	**DOZVOLI** — pristup samo sopstvenim zapisima
Stanar	Obaveza	izmena	Ima aktivan zakup nad tom nekretninom	**ZABRANI** — zakup ne obuhvata izmenu obaveza
Pozvani korisnik	Deljena funkcionalnost	pregled	Ima pravo dodeljeno putem pozivnice (pozivnica → grant)	**DOZVOLI** — ograničen pristup, samo dodeljene operacije

Segmentacija podataka

Kontrola pristupa pojedinačnom resursu rešava pitanje da li korisnik sme da izvrši određenu operaciju, ali u višekorisničkom sistemu potrebno je odrediti i koji podaci treba da mu budu dostupni. Kod operacija čiji rezultat zavisi od prava konkretnog korisnika, autorizacioni kontekst koristi se i za ograničavanje skupa podataka koji mu se vraća.

Pod segmentacijom podataka u ovom radu podrazumeva se određivanje skupa podataka dostupnog konkretnom korisniku na osnovu njegovog identiteta, uloge, odnosa prema resursima i utvrđenog obima pristupa. Zbog toga dva korisnika koji pristupaju istoj funkcionalnosti ne moraju dobiti isti rezultat. Vlasniku su, na primer, dostupni podaci povezani sa nekretninama nad kojima ima odgovarajuća prava, dok agencija pristupa podacima koji pripadaju nekretninama i poslovnim odnosima u kojima učestvuje. Stanaru su dostupne informacije povezane sa njegovim zakupom

i resursima kojima mu je pristup dozvoljen.

Poseban slučaj predstavljaju korisnici sa ulogom stanara kojima je pristup eksplicitno dodeljen putem pozivnice. Dodeljena dozvola omogućava da takav korisnik dobije određena prava nad resursom, pri čemu se prilikom segmentacije uzima u obzir i obim dodeljenih prava. Korisniku zato nisu automatski dostupni svi podaci povezani sa nekretninom, već samo oni koji odgovaraju dozvoljenom obimu pristupa.

Kod funkcionalnosti čija vidljivost podataka zavisi od autorizacionog konteksta, segmentacija se sprovodi na serverskoj strani sistema. Kod pristupa pojedinačnom zaštićenom resursu proverava se da li korisnik ima odgovarajući osnov i pravo da nad tim resursom izvrši traženu operaciju. Kada funkcionalnost vraća kolekciju podataka, segmentacija se primenjuje tako što se rezultat ograničava na podatke koji su korisniku dostupni na osnovu njegovog identiteta, poslovnih odnosa i utvrđenog obima pristupa. Autorizacija i segmentacija tako koriste isti poslovni kontekst, ali imaju različite uloge: autorizacija određuje da li je određena operacija dozvoljena, dok segmentacija određuje koje podatke korisnik može da dobije kao njen rezultat. Klijentskoj aplikaciji se prosleđuje već odgovarajući skup podataka, dok prikazivanje ili skrivanje elemenata korisničkog interfejsa služi prilagođavanju korisničkog iskustva i ne predstavlja zamenu za kontrolu pristupa na serverskoj strani.

Primer segmentacije kolekcije prikazan je u listingu 3.4. Ista metoda vraća različit skup podataka u zavisnosti od obima pristupa koji je autorizacija utvrdila za pozivaoca.

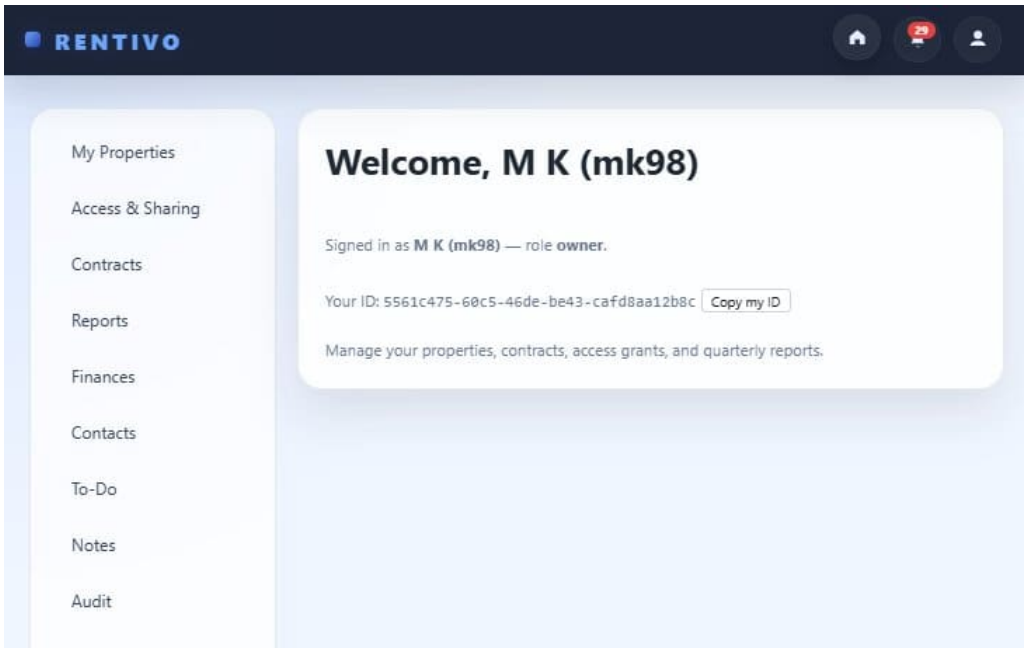
```
async def list_contacts(self, identity: Identity, property_id: UUID) -> list[
    Contact]:
    scope = await self._authorizer.enforce(identity, self._contact(property_id),
    Action.VIEW)
    contacts = await self._contacts.list_by_property(property_id)
    # stanar (agencijski zakup ILI pozvani korisnik) vidi samo kontakte vidljive
    stanaru;
    # vlasnik / agencija koja upravlja vide sve
    if is_resident(scope):
        return [c for c in contacts if c.visible_to_tenant]
    return contacts
```

Listing 3.4: Segmentacija rezultata prema obimu pristupa.

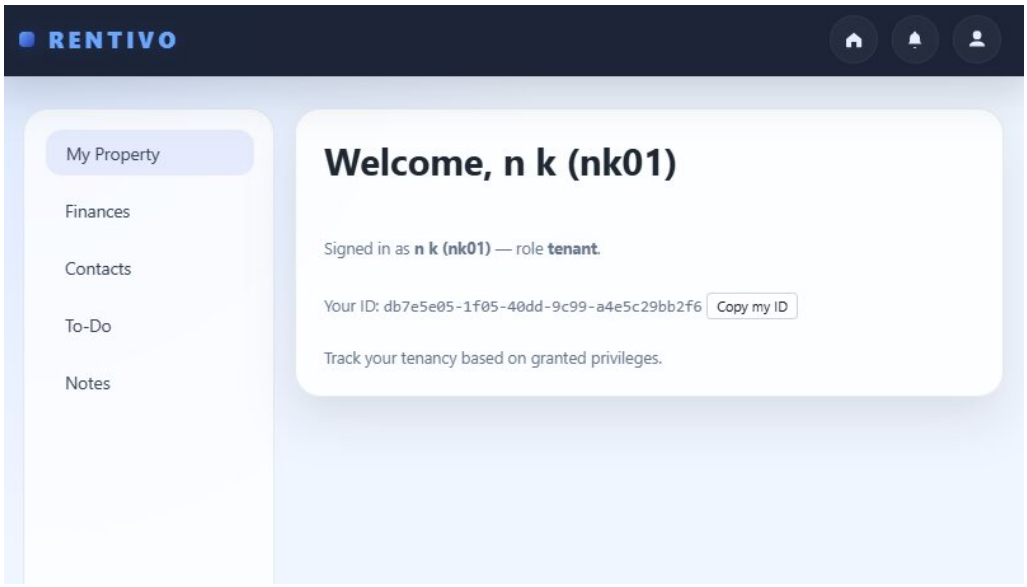
Metoda najpre poziva `enforce()`, čime istovremeno proverava pravo pristupa i dobija obim (`scope`). Zatim se učitani skup filtrira: predikat `is_resident(scope)` je tačan kada osnova pristupa potiče od zakupa ili eksplicitno dodeljenog prava

(`TENANCY` ili `GRANT`), pa takav korisnik dobija samo kontakte označene kao vidljive stanaru, dok vlasnik i agencija koja upravlja nekretninom dobijaju ceo skup. Na taj način dva korisnika koji pozivaju istu funkcionalnost nad istom nekretninom mogu dobiti različit rezultat, pri čemu se filtriranje u potpunosti sprovodi na serverskoj strani.

Praktičan primer primene opisanog modela prikazan je na slikama 3.10 i 3.11, na kojima se može uočiti razlika u dostupnim funkcionalnostima za korisnike sa različitim odnosom prema resursima sistema. Na slici 3.10 prikazan je interfejs iz perspektive vlasnika, kojem su dostupne funkcionalnosti povezane sa upravljanjem nekretninama, pravima pristupa, ugovorima, finansijama, izveštajima i revizijskim dnevnikom. Nasuprot tome, na slici 3.11 prikazan je interfejs stanara sa ograničenim pristupom, kojem je dostupan uži skup funkcionalnosti u skladu sa dodeljenim pravima. Ovaj primer pokazuje da sama pripadnost korisničkoj ulozi nije jedini kriterijum za određivanje dostupnih podataka i operacija, već se uzima u obzir i konkretan odnos korisnika prema resursu i utvrđeni obim pristupa.



Slika 3.10: Korisnički prikaz aplikacije iz perspektive vlasnika



Slika 3.11: Korisnički prikaz aplikacije stanara sa ograničenim pristupom

3.7 Implementirane funkcionalnosti sistema

Prethodna poglavlja prikazala su arhitekturu sistema, organizaciju podataka i mehanizme kojima se kontroliše pristup poslovnim resursima. Na toj osnovi razvijen je skup funkcionalnosti koje obuhvataju upravljanje korisnicima i nekretninama, različite modele izdavanja, finansijske obaveze, dokumentaciju i druge aktivnosti koje prate korišćenje nekretnine. Implementacija ovih funkcionalnosti povezuje prethodno opisane arhitektonske komponente sa konkretnim poslovnim procesima dostupnim korisnicima aplikacije.

Funkcionalnosti nisu jednake za sve korisnike. Njihova dostupnost zavisi od korisničke uloge i, kod pristupa postojećim resursima, od odnosa korisnika prema konkretnom resursu.

Upravljanje korisnicima i nekretninama

Upravljanje korisnicima predstavlja polaznu tačku za korišćenje poslovnih funkcionalnosti sistema. Registracijom se formira korisnički nalog sa jednom od tri podržane uloge: vlasnik, agencija ili stanar. Postupak registracije i autentifikacije detaljnije je opisan u poglavlju 3.5 (Autentifikacija korisnika), dok se u ovom odeljku razmatra korišćenje naloga u okviru poslovnih funkcionalnosti sistema. Korisnička uloga određuje osnovni položaj korisnika u sistemu, dok se prava nad konkretnim

nekretninama i povezanim podacima utvrđuju na osnovu modela kontrole pristupa predstavljenog u poglavlju 3.6 (Autorizacija i segmentacija podataka). Pozvani korisnik sa ograničenim pravima ne uvodi se kao posebna uloga, već zadržava ulogu stanara, dok dodeljena prava određuju kojim funkcionalnostima i podacima može da pristupi.

Nekretnina predstavlja centralni poslovni resurs oko kojeg je organizovan najveći deo funkcionalnosti aplikacije. Vlasnik može da kreira nekretninu kojom upravlja, dok je kreiranje nekretnine dostupno i korisniku sa ulogom agencije. Prilikom njenog evidentiranja uspostavlja se odnos vlasništva, kojim se konkretan korisnik povezuje sa nekretninom. Ovaj odnos ima dvostruku ulogu: čini deo poslovnog modela sistema i istovremeno predstavlja jedan od osnova za određivanje prava pristupa postojećoj nekretnini.

Način daljeg korišćenja nekretnine zavisi od izabranog režima upravljanja. Implementacija razlikuje direktno izdavanje od strane vlasnika i izdavanje preko agencije, dok postoji i režim u kome se nekretnina koristi bez izdavanja. Ova informacija nije samo svojstvo prikazano korisniku, već učestvuje u određivanju poslovnog toka koji je za konkretnu nekretninu primenljiv. Konkretni tokovi rada za oba modela izloženi su u narednom odeljku.

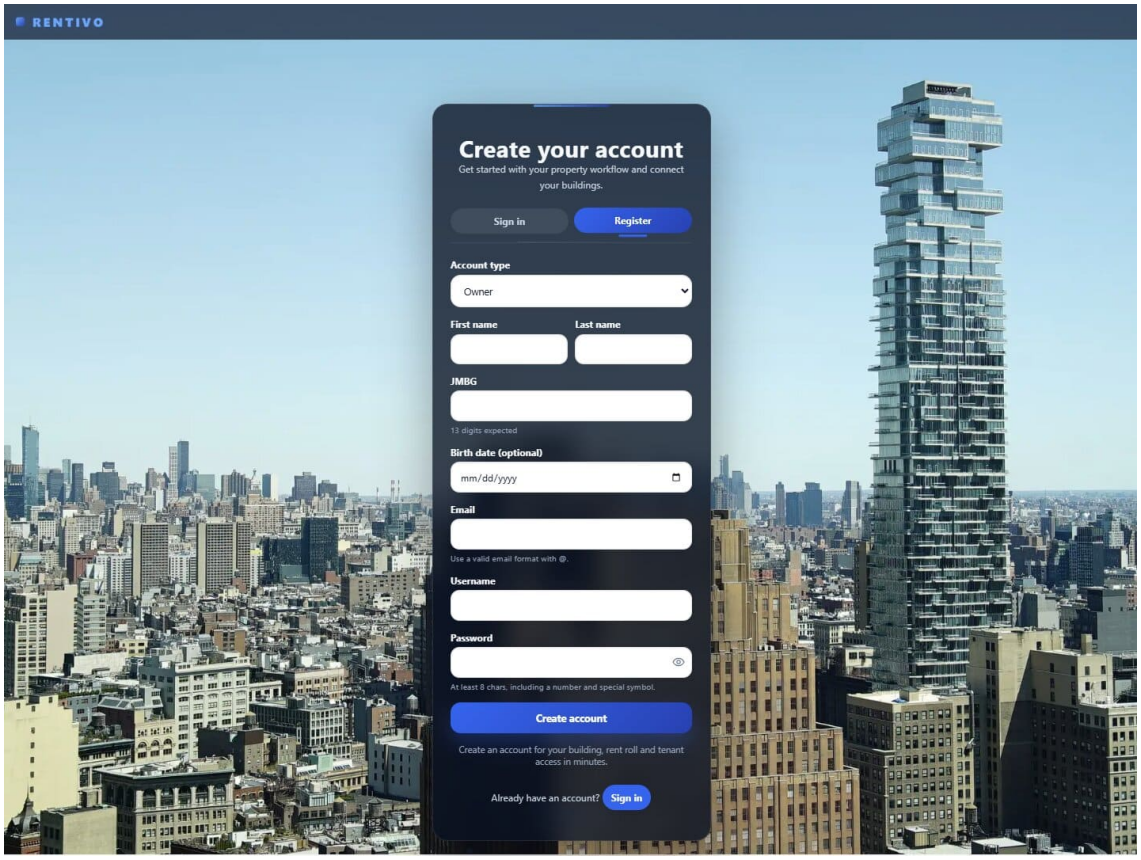
Agencija u sistemu može nastupati u dva različita konteksta. Može upravljati nekretninom na osnovu ugovornog odnosa sa vlasnikom, ali može biti i nosilac vlasništva nad sopstvenom nekretninom. Zbog toga se njene mogućnosti ne određuju samo na osnovu korisničke uloge, već i prema konkretnom odnosu koji postoji prema posmatranoj nekretnini. Ovakvo modelovanje omogućava da isti tip korisničkog naloga učestvuje u različitim poslovnim scenarijima bez uvođenja dodatnih uloga.

Stanari se sa nekretninama povezuju u skladu sa načinom njihovog korišćenja. Kada se nekretnina izdaje preko agencije, odnos sa stanarom evidentira se kroz zakup, čime se čuva veza između stanara i nekretnine tokom odgovarajućeg perioda. Kod direktnog izdavanja od strane vlasnika, pristup stanara može biti realizovan putem pozivnice i eksplicitno dodeljenih prava. Time je omogućeno i uključivanje drugih osoba kojima je potreban ograničen pristup pojedinim delovima aplikacije, bez izjednačavanja njihovih prava sa pravima vlasnika ili korisnika koji raspolaže širim poslovnim ovlašćenjima.

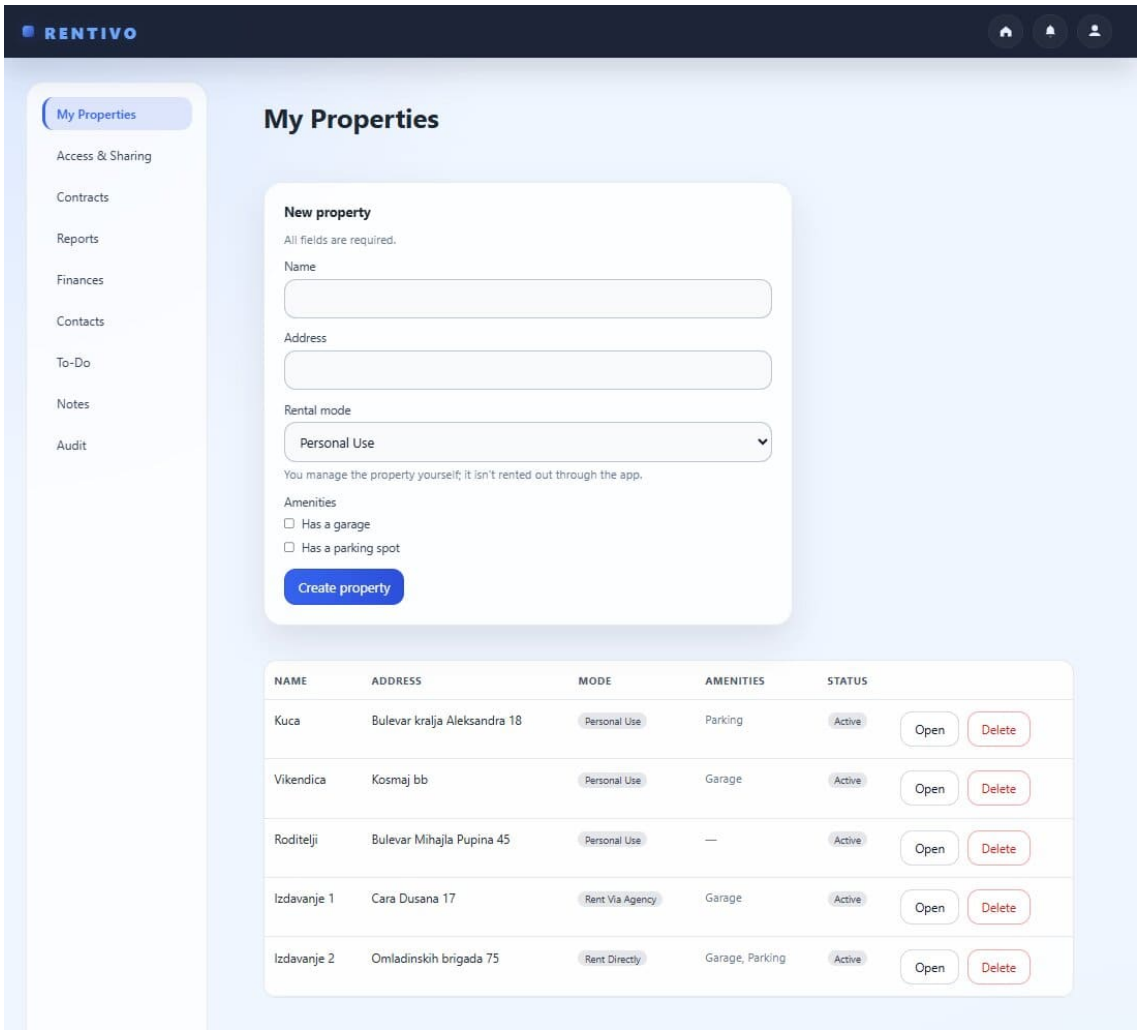
Upravljanje korisnicima i nekretninama na taj način postavlja osnovu za ostale funkcionalnosti sistema. Uspostavljeni poslovni odnosi omogućavaju da se za postojeću nekretninu odredi ko njome upravlja, na kom osnovu joj pristupa i koje operacije

može da izvršava. Ovi odnosi posebno dolaze do izražaja u procesu izdavanja, gde direktni i agencijski model formiraju različite tokove rada.

Na slici 3.12 prikazana je stranica za registraciju, na kojoj novi korisnik unosi podatke potrebne za kreiranje naloga i bira odgovarajuću korisničku ulogu. Nakon registracije i prijave, korisniku su dostupne funkcionalnosti u skladu sa njegovom ulogom i odnosom prema konkretnim resursima. Na slici 3.13 prikazan je deo aplikacije iz perspektive vlasnika, koji omogućava evidentiranje nove nekretnine izborom osnovnih podataka i režima njenog korišćenja, kao i pregled nekretnina koje su povezane sa njegovim nalogom.



Slika 3.12: Stranica za registraciju korisnika



Slika 3.13: Upravljanje nekretninama iz perspektive vlasnika

Proces izdavanja nekretnine

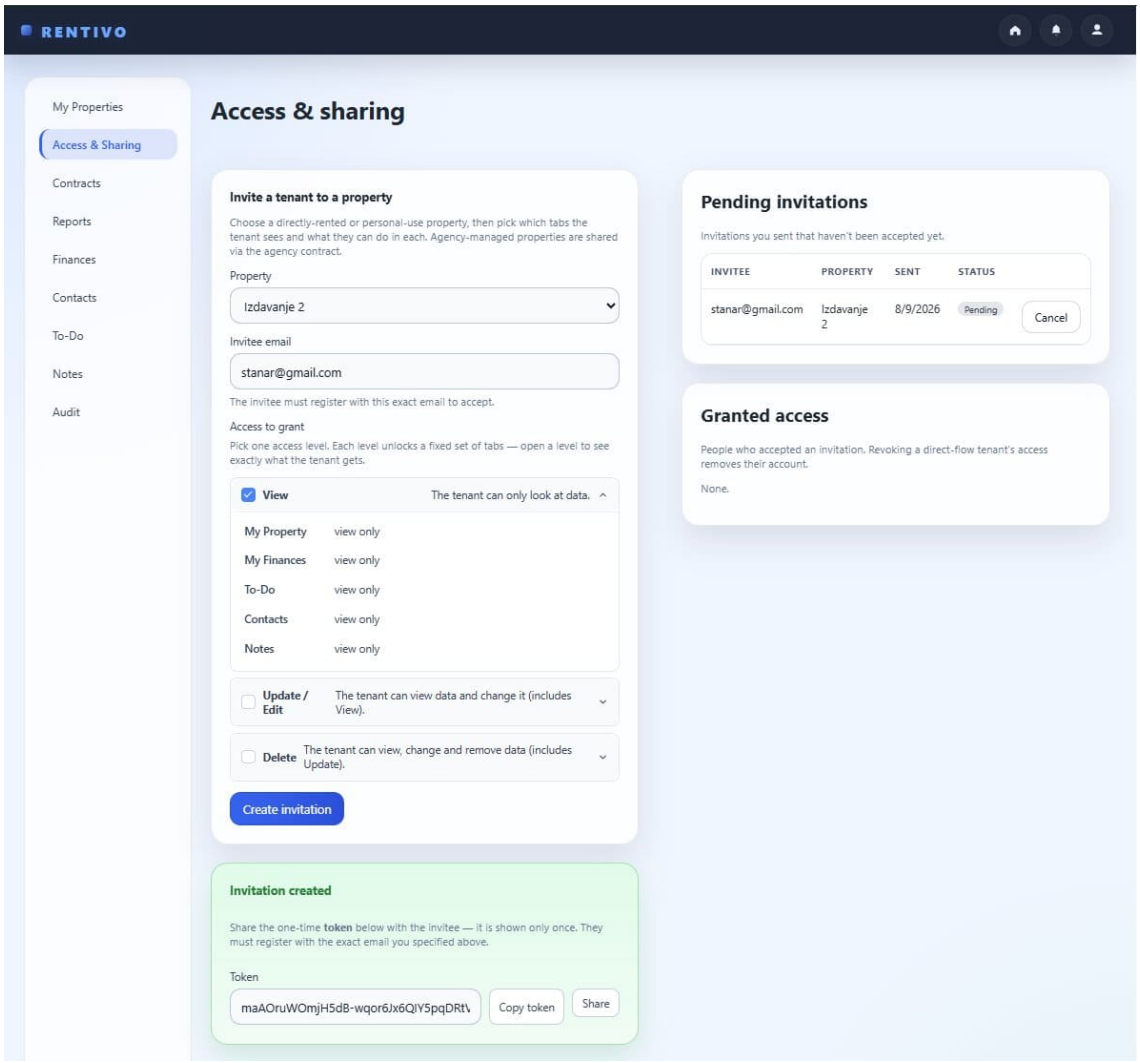
Proces izdavanja jedna je od centralnih poslovnih celina sistema, a njegova dva modela, direktno izdavanje od vlasnika i izdavanje preko agencije, razlikuju se prema poslovnim odnosima koji se uspostavljaju između vlasnika, agencije, stanara i nekretnine.

Kod direktnog izdavanja, vlasnik zadržava neposredno upravljanje nekretninom i može drugim korisnicima dodeliti pristup odabranim funkcionalnostima. Uključivanje korisnika započinje pozivnicom, nakon čijeg prihvatanja mogu biti formirana odgovarajuća prava pristupa. Na taj način pozvani korisnik dobija samo one mogućnosti koje su mu eksplicitno dodeljene, bez promene njegove osnovne korisničke uloge.

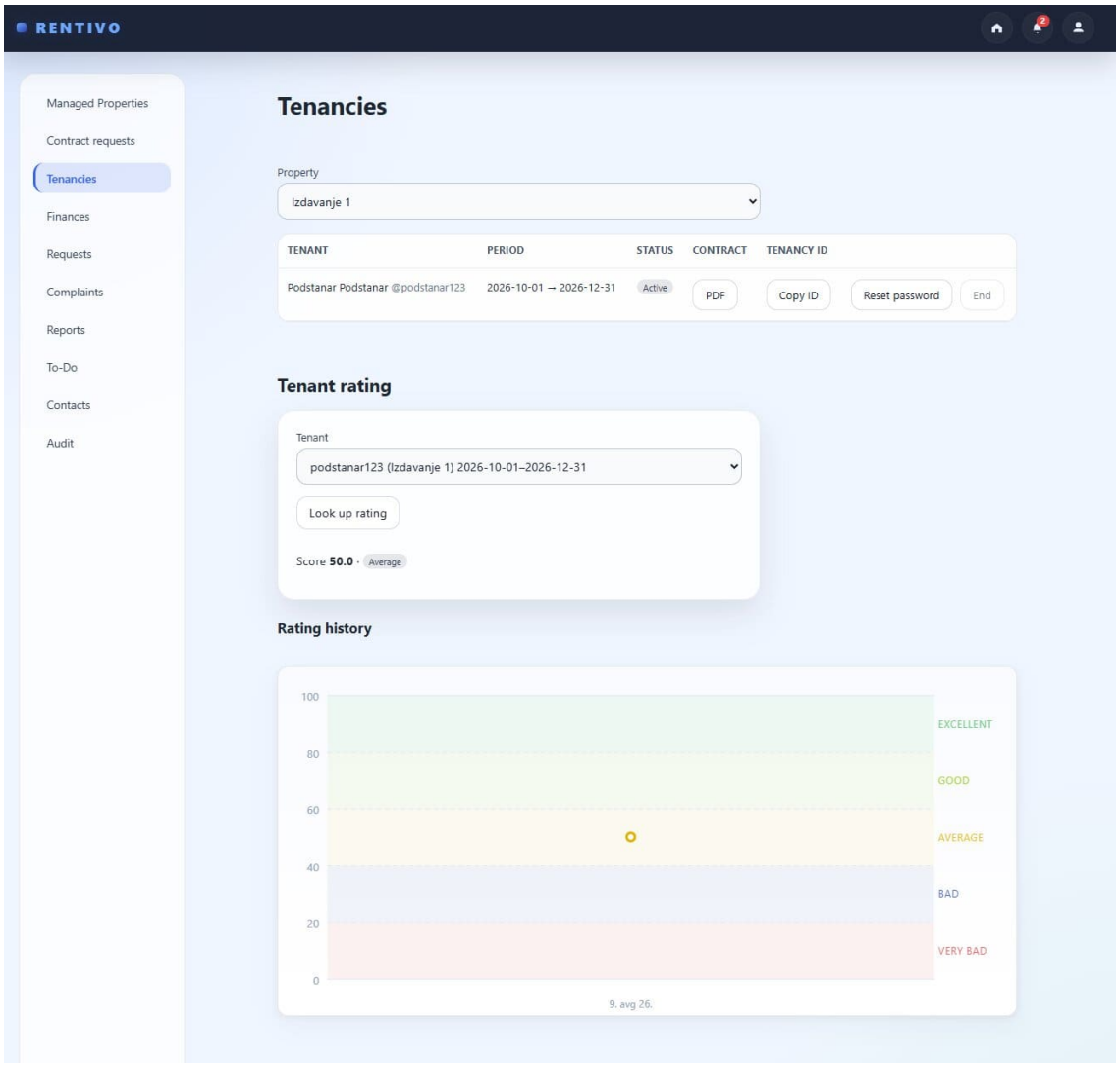
Kod izdavanja preko agencije, vlasnik i agencija povezuju se ugovornim odnosom, nakon čega agencija može da upravlja procesom izdavanja u okviru prava koja iz tog odnosa proizlaze. Stanar se sa nekretninom povezuje kroz zakup, čuvaju se podaci o periodu korišćenja i omogućeno je praćenje istorije stanara.

Tok direktnog izdavanja i dodeljivanja pristupa prikazan je na slici 3.14. Vlasnik iz pregleda konkretne nekretnine može da upravlja pristupom i pošalje pozivnicu korisniku kojem želi da omogući pristup. Prilikom dodeljivanja pristupa određuje se njegov dozvoljeni obim, tako da pozvani korisnik dobija samo prava koja su mu eksplicitno dodeljena. Na istom prikazu vlasnik može da vidi postojeća prava pristupa nad nekretninom i korisnike kojima su ona dodeljena.

Upravljanje nekretninom koja se izdaje preko agencije prikazano je na slici 3.15. Iz perspektive agencije dostupne su informacije o nekretnini i funkcionalnosti potrebne za vođenje procesa izdavanja u okviru uspostavljenog ugovornog odnosa. Na istom prikazu dostupna je i opcija resetovanja lozinke stanara, namenjena slučaju kada stanar izgubi pristup nalogu koji mu je agencija kreirala prilikom evidentiranja. Operacija podleže istom mehanizmu autorizacije kao i ostale akcije — zahteva pravo izmene nad zakupom, koje ima agencija (na osnovu ugovora), dok je sam stanar ne može izvršiti. Nova privremena lozinka čuva se isključivo u obliku Argon2 heša, a operacija se beleži u evidenciji aktivnosti. Agencija može da vidi i podatke povezane sa zakupom, kao i ocenu stanara. Ocena stanara je automatski izračunata vrednost od 0 do 100, namenjena agenciji. Polazi od podrazumevane vrednosti 50, koju umanjuju žalbe (najviše 30 poena) i zahtevi iz poslednjih dvanaest meseci (nerešeni najviše 20, rešeni najviše 8 poena, uz veću težinu višeg prioriteta), pri čemu se svaki od tih iznosa dostiže postepeno, srazmerno broju događaja. Za svaki ceo mesec bez žalbi i zahteva dodaje se bonus od 2 poena, a konačna vrednost ograničava se na opseg od 0 do 100. Rezultat se preslikava u pet kategorija: vrlo loše (*VERY BAD*, 0–20), loše (*BAD*, 20–40), prosečno (*AVERAGE*, 40–60), dobro (*GOOD*, 60–80) i odlično (*EXCELLENT*, 80–100). Agenciji je pristup ograničen na nekretnine za koje postoji odgovarajući poslovni odnos.



Slika 3.14: Direktno izdavanje i deljenje pristupa



Slika 3.15: Upravljanje izdavanjem preko agencije

Upravljanje finansijama i dokumentacijom

Finansijski deo sistema omogućava evidentiranje troškova i obaveza povezanih sa nekretninama i učesnicima u odgovarajućem poslovnom odnosu. Za obaveze koje se periodično ponavljaju predviđeni su šabloni, čime se smanjuje potreba za njihovim pojedinačnim unosom. Za evidentirane obaveze prati se njihovo stanje, a sistem beleži i njihovo izmirivanje. Na taj način korisnicima je omogućen pregled finansijskih podataka koji pripadaju nekretninama i poslovnim odnosima kojima imaju pravo pristupa.

Prilikom evidentiranja izmirenja obaveze moguće je priložiti i dokaz o uplati. Takav prilog povezuje finansijski događaj sa odgovarajućim dokumentom, dok se

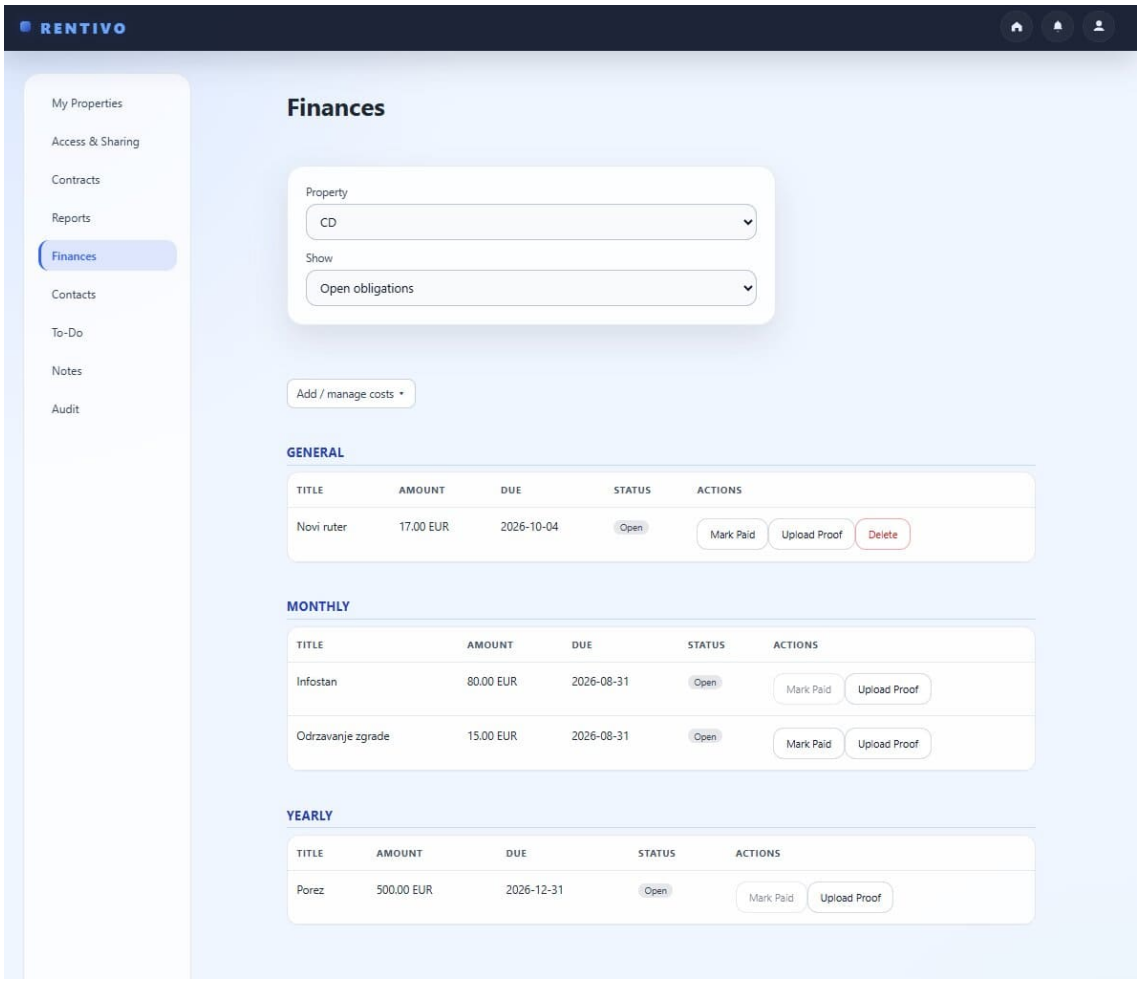
njegov binarni sadržaj čuva odvojeno od strukturisanih poslovnih podataka. Isti princip primenjuje se i na priloge koji nastaju u drugim delovima sistema. Binarni sadržaj datoteka skladišti se korišćenjem GridFS-a, dok se odgovarajuće reference i metapodaci koriste za povezivanje dokumenta sa poslovnim zapisom na koji se odnosi.

Dokumentacija u sistemu ne obuhvata samo sadržaje koje korisnici otpremaju. Na osnovu postojećih poslovnih podataka aplikacija generiše i PDF dokumente, među kojima se nalaze ugovori i izveštaji. Time se podaci evidentirani tokom rada sistema mogu predstaviti u obliku pogodnom za čuvanje, pregled i preuzimanje.

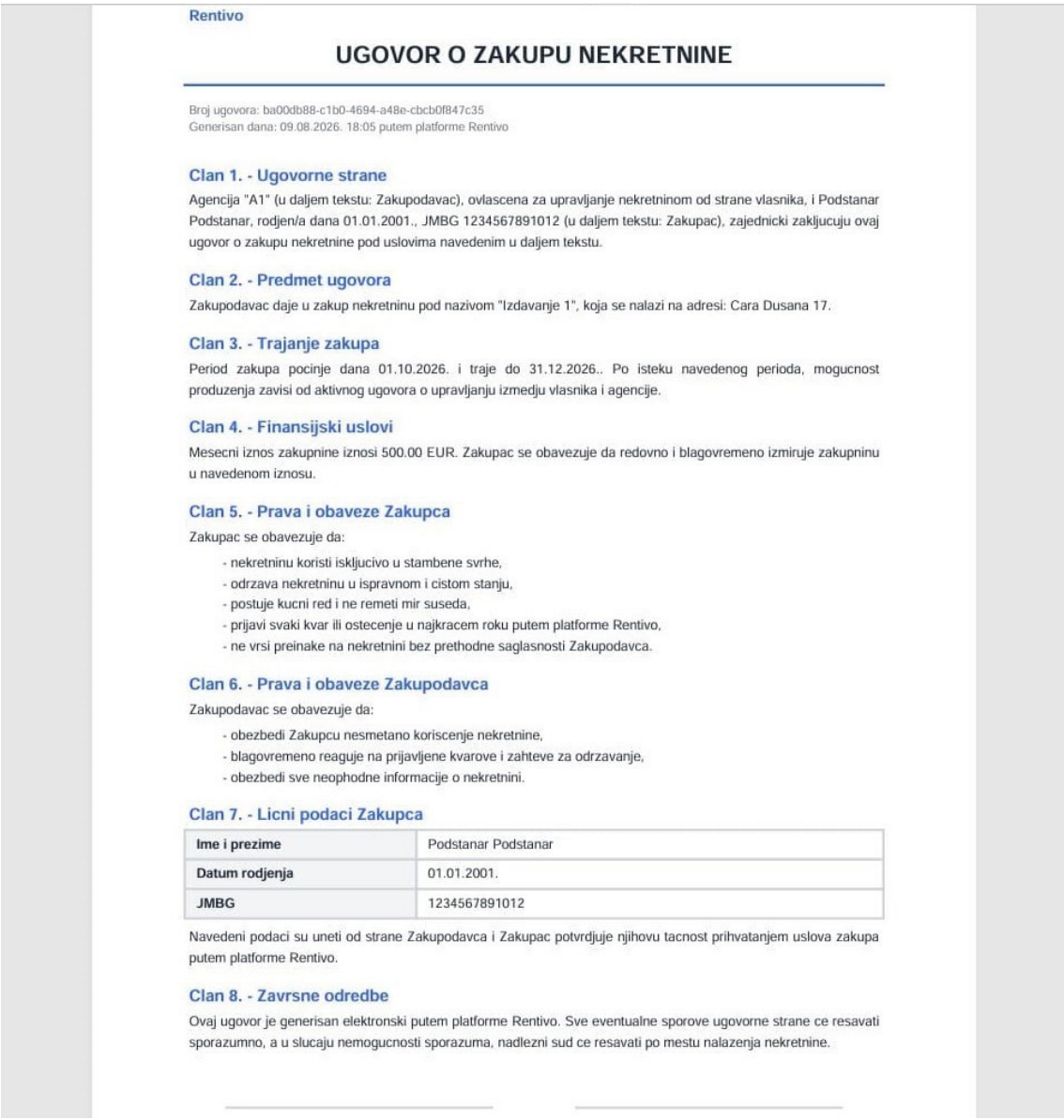
Poseban deo ove funkcionalnosti čine izveštaji. Implementacija podržava generisanje kvartalnog izveštaja za vlasnika i godišnjeg izveštaja o stanaru. Prilikom njihovog formiranja koriste se podaci dostupni u trenutku generisanja, nakon čega se čuva snapshot na osnovu kojeg nastaje PDF dokument. Time sadržaj već formiranog izveštaja nije neposredno zavisan od kasnijih promena izvornih poslovnih podataka. Snapshot se čuva u MongoDB bazi podataka, dok se generisani PDF skladišti putem GridFS-a. Na ovaj način model perzistencije opisan u poglavlju 3.4 primenjen je i na konkretne finansijske i dokumentacione funkcionalnosti sistema.

Na slici 3.16 prikazan je deo sistema namenjen praćenju finansijskih podataka povezanih sa nekretninom iz perspektive njenog vlasnika. Vlasnik može da pregleda opšte, mesečne i godišnje troškove, da ih označi kao plaćene i, kada je potrebno, priloži dokaz o uplati. Mesečni i godišnji troškovi mogu se definisati pomoću šablona na osnovu kojih se periodično kreiraju odgovarajuće obaveze, dok opšti troškovi nisu vezani za takav način ponavljanja. Za trošak se može definisati i rok za plaćanje, a ukoliko obaveza nije izmirena, približavanjem tog roka korisnik dobija odgovarajuće obaveštenje. Na taj način vlasnik na jednom mestu može da prati evidentirane finansijske obaveze i njihovo izmirenje.

Na slici 3.17 prikazan je primer PDF ugovora o zakupu generisanog na osnovu podataka evidentiranih u sistemu. Ugovor se zaključuje između agencije koja upravlja izdavanjem nekretnine i stanara, a sadrži podatke o nekretnini, periodu zakupa i ugovorenom iznosu zakupnine.



Slika 3.16: Pregled finansijskih obaveza nekretnine



Rentivo

UGOVOR O ZAKUPU NEKRETNINE

Broj ugovora: ba00db88-c1b0-4694-a48e-cbcb0f847c35
Generisan dana: 09.08.2026. 18:05 putem platforme Rentivo

Clan 1. - Ugovorne strane

Agencija "A1" (u daljem tekstu: Zakupodavac), ovlascena za upravljanje nekretninom od strane vlasnika, i Podstanar Podstanar, rodjen/a dana 01.01.2001., JMBG 1234567891012 (u daljem tekstu: Zakupac), zajednicki zakljucuju ovaj ugovor o zakupu nekretnine pod uslovima navedenim u daljem tekstu.

Clan 2. - Predmet ugovora

Zakupodavac daje u zakup nekretninu pod nazivom "Izdavanje 1", koja se nalazi na adresi: Cara Dusana 17.

Clan 3. - Trajanje zakupa

Period zakupa pocinje dana 01.10.2026. i traje do 31.12.2026.. Po isteku navedenog perioda, mogucnost produzenja zavisi od aktivnog ugovora o upravljanju izmedju vlasnika i agencije.

Clan 4. - Finansijski uslovi

Mesecni iznos zakupnine iznosi 500.00 EUR. Zakupac se obavezuje da redovno i blagovremeno izmiruje zakupninu u navedenom iznosu.

Clan 5. - Prava i obaveze Zakupca

Zakupac se obavezuje da:

- nekretninu koristi iskljucivo u stambene svrhe,
- odrzava nekretninu u ispravnom i cistom stanju,
- postuje kucni red i ne remeti mir suseda,
- prijavi svaki kvar ili ostecenje u najkracem roku putem platforme Rentivo,
- ne vrsi preinake na nekretnini bez prethodne saglasnosti Zakupodavca.

Clan 6. - Prava i obaveze Zakupodavca

Zakupodavac se obavezuje da:

- obezbedi Zakupcu nesmetano koriscenje nekretnine,
- blagovremeno reaguje na prijavljene kvarove i zahteve za odrzavanje,
- obezbedi sve neophodne informacije o nekretnini.

Clan 7. - Licni podaci Zakupca

Ime i prezime	Podstanar Podstanar
Datum rodjenja	01.01.2001.
JMBG	1234567891012

Navedeni podaci su uneti od strane Zakupodavca i Zakupac potvrdjuje njihovu tacnost prihvatanjem uslova zakupa putem platforme Rentivo.

Clan 8. - Zavrsne odredbe

Ovaj ugovor je generisan elektronski putem platforme Rentivo. Sve eventualne sporove ugovorne strane ce resavati sporazumno, a u slucaju nemogucnosti sporazuma, nadlezni sud ce resavati po mestu nalazenja nekretnine.

Slika 3.17: Primer generisanog PDF ugovora

Dodatne funkcionalnosti sistema

Pored osnovnih funkcionalnosti vezanih za upravljanje korisnicima, nekretninama, ugovorima, zakupima i finansijama, sistem sadrži i dodatne funkcionalnosti koje podržavaju svakodnevni rad korisnika i organizaciju aktivnosti povezanih sa nekretninama.

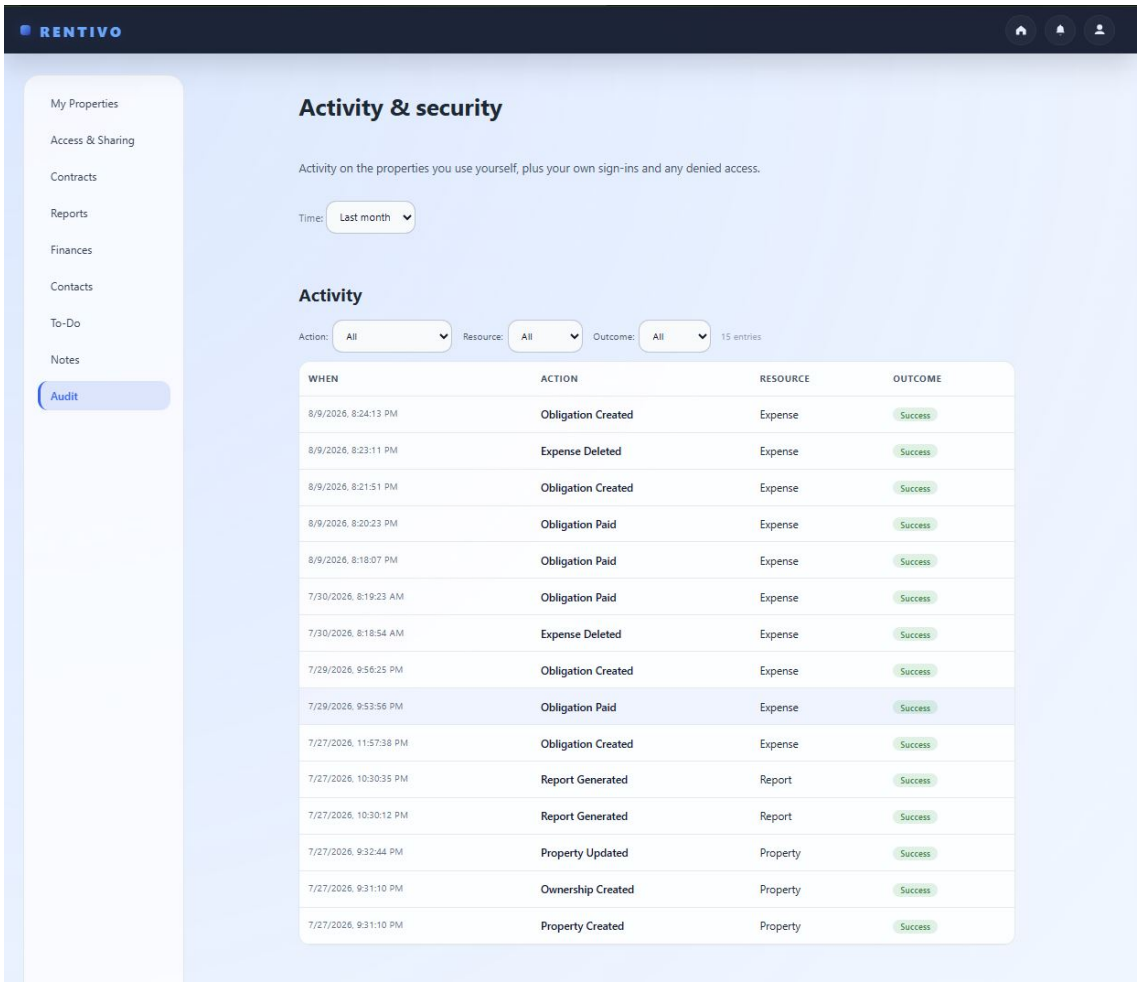
Jedna od značajnijih funkcionalnosti u ovoj grupi jeste upravljanje zahtevima. Korisnik sa odgovarajućim pravima može prijaviti zahtev vezan za nekretninu, nakon

čega se prati njegov status tokom obrade. U agencijskom toku podržan je prelazak zahteva od podnetog, preko obrade, do rešenog stanja. Uz zahtev mogu biti povezani prilozi, dok se evidentiraju i promene njegovog statusa. Podaci o zahtevima čuvaju se u dokumentnoj bazi podataka. Pristup zahtevima, kao i kod ostalih zaštićenih resursa, zavisi od odnosa korisnika prema nekretnini i odgovarajućeg autorizacionog osnova.

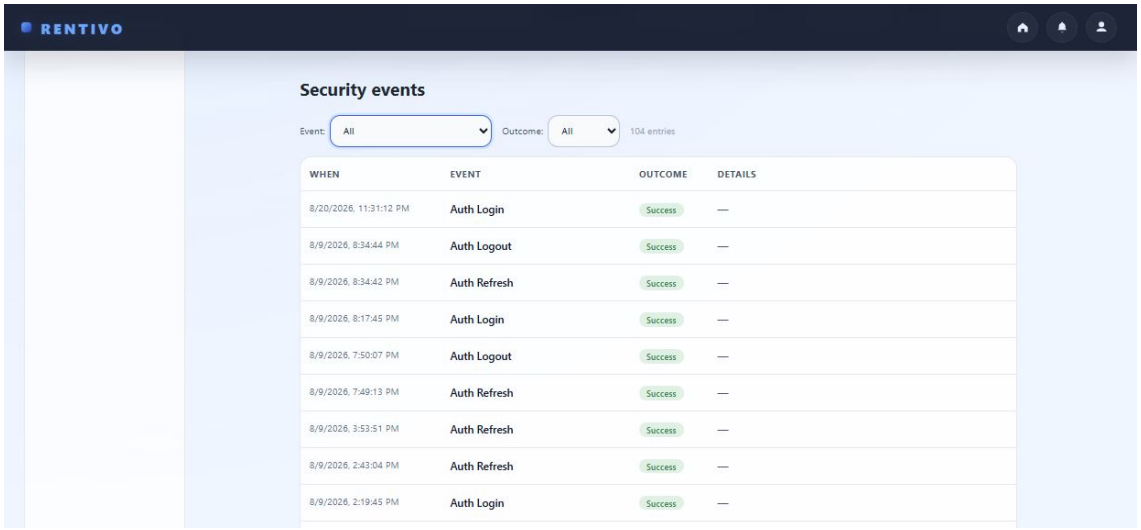
Sa procesom rešavanja problema povezane su i žalbe. One čine zasebnu poslovnu evidenciju i koriste se u okviru agencijskog modela, pri čemu je pristup zasnovan na ugovornom odnosu. Podaci o žalbama čuvaju se u dokumentnoj bazi podataka.

Sistem vodi i evidenciju aktivnosti i bezbednosnih događaja. Evidencija aktivnosti obuhvata događaje koji nastaju tokom izvršavanja pojedinih poslovnih operacija, dok je bezbednosna evidencija namenjena događajima povezanim sa autentifikacijom i kontrolom pristupa, uključujući prijavu, obnavljanje sesije, odjavu i odbijene pokušaje pristupa. Ove dve vrste zapisa čuvaju se odvojeno u MongoDB kolekcijama, čime se poslovni događaji razdvajaju od događaja bezbednosnog karaktera. Vlasnik i agencija mogu pregledati evidenciju aktivnosti kroz poseban prikaz u okviru aplikacije, čime ona nije namenjena isključivo internom vođenju zapisa.

Na slikama 3.18 i 3.19 prikazan je revizijski dnevnik iz perspektive vlasnika. Na slici 3.18 prikazana je evidencija poslovnih aktivnosti, koja sadrži vreme izvršavanja, vrstu izvršene akcije, resurs nad kojim je akcija izvršena i njen ishod. Prikazane zapise moguće je filtrirati prema vremenskom periodu, akciji, resursu i ishodu. Na slici 3.19 prikazana je odvojena evidencija bezbednosnih događaja, koja obuhvata događaje povezane sa autentifikacijom i kontrolom pristupa, uz podatke o vremenu njihovog nastanka i ishodu. Na taj način korisniku je omogućen odvojen pregled poslovnih aktivnosti i događaja bezbednosnog karaktera.

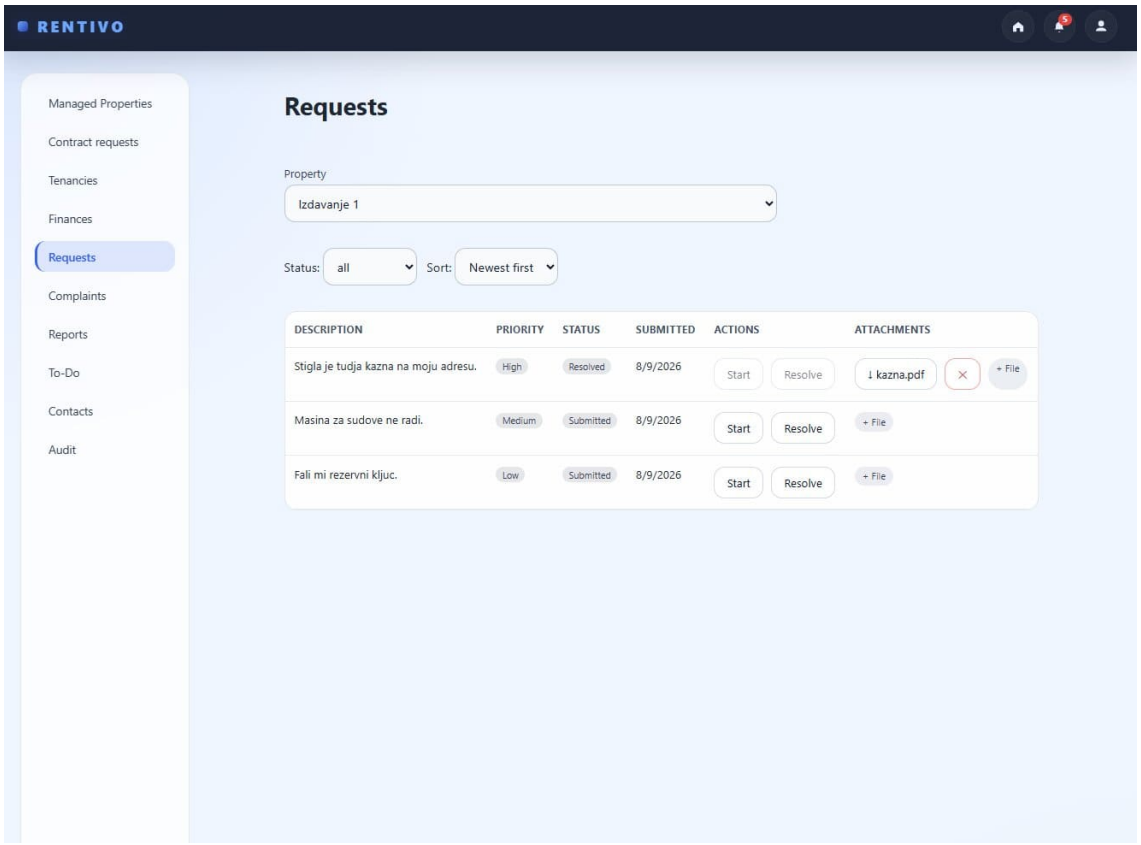


Slika 3.18: Pregled evidencije poslovnih aktivnosti iz perspektive vlasnika



Slika 3.19: Pregled evidencije bezbednosnih događaja iz perspektive vlasnika

Na slici 3.20 prikazan je pregled zahteva podnetih za konkretnu nekretninu iz perspektive agencije. Agencija može da vidi evidentirane zahteve stanara i podatke potrebne za njihovu obradu, uključujući opis i prioritet zahteva, kao i priložene dokumente kada postoje. Tokom obrade agencija može da menja stanje zahteva, čime se prati njegov tok od podnošenja, preko obrade, do rešavanja.



Slika 3.20: Pregled podnetih zahteva za jednu nekretninu iz ugla agencije

Pored već navedenih, još nekoliko dodatnih funkcionalnosti predstavljeno je u nastavku:

- **Kontakti** – omogućavaju vođenje imenika kontakata povezanih sa nekretninom. U okviru agencijskog modela moguće je odrediti da li je pojedinačni kontakt vidljiv stanaru.
- **Beleške** – omogućavaju evidentiranje i organizovanje beležaka povezanih sa nekretninom. U modelu direktnog upravljanja beleške mogu biti dostupne i pozvanom korisniku.

- **Zadaci** – omogućavaju vođenje zadataka povezanih sa nekretninom. U direktnom modelu izdavanja zadaci mogu biti deljeni između vlasnika i pozvanog korisnika, dok agencija može dodeljivati zadatke stanaru.
- **Lista za kupovinu** – omogućava vlasniku vođenje stavki za kupovinu i izvoz liste u PDF dokument.

3.8 Testiranje i ograničenja sistema

Nakon implementacije ključnih funkcionalnosti bilo je potrebno proveriti njihovo ponašanje na različitim nivoima sistema. Testiranje je zato obuhvatilo domensku logiku, aplikacione servise, mehanizme autorizacije, integraciono testiranje API sloja i proveru sistema u pokrenutom okruženju. Posebna pažnja posvećena je funkcionalnostima koje predstavljaju jezgro rada, pre svega kontroli pristupa i izolaciji podataka između korisnika.

Pored provere ispravnosti implementacije, važno je jasno odrediti i granice razvijenog rešenja. Sistem je razvijen u skladu sa funkcionalnim zahtevima definisanim u okviru rada i predviđen je za izvršavanje u lokalnom okruženju, pa pojedine karakteristike tipične za produkcione distribuirane sisteme nisu bile predmet razvoja. U nastavku su zato odvojeno predstavljeni pristup testiranju i najvažnija ograničenja trenutne verzije sistema.

Testiranje sistema

Strategija testiranja zasnovana je na kombinovanju više nivoa provere, uključujući testove domenske i aplikacione logike. Domenski testovi proveravaju pravila i invarijante koje mogu biti ispitane nezavisno od infrastrukture, dok se servisnim testovima proverava realizacija konkretnih slučajeva upotrebe i saradnja komponenti aplikacionog sloja. Za potrebe ovih testova koriste se memorijske zamene stvarnih repozitorijuma, čime se poslovna logika može proveravati bez povezivanja sa bazama podataka.

Posebnu grupu čine testovi autorizacije, što je značajno zbog centralne uloge kontrole pristupa u razvijenom sistemu. Njima se proveravaju različite kombinacije odnosa korisnika prema resursu i dozvoljenih akcija, kao i izolacija podataka između različitih korisnika. Dodatni testovi obuhvataju scenarije u kojima se pristup ostvaruje putem eksplicitno dodeljenih prava, čime se proverava da ograničene dozvole ne

omogućavaju pristup izvan predviđenog obima. Posebna pažnja posvećena je i negativnim scenarijima, u kojima se proverava da korisnik bez odgovarajućeg odnosa prema resursu ne može da mu pristupi, kao i da eksplicitno dodeljeno pravo ne omogućava operacije koje njime nisu obuhvaćene niti pristup drugim resursima. Na taj način testovima se proverava primena podrazumevane zabrane i izolacija podataka između korisnika.

Pored testova pojedinačnih komponenti, postoje i integracione provere API sloja. Njihov cilj je da se potvrdi saradnja ruta i ostalih delova serverske aplikacije kroz HTTP interfejs. Ovakvi testovi zahtevaju širi skup infrastrukturnih komponenti od domenskih i servisnih testova, zbog čega deo njih zavisi od dostupnosti odgovarajućeg okruženja.

Kao dopuna automatizovanim testovima koristi se i provera sistema u pokrenutom okruženju. Ona obuhvata izvršavanje reprezentativnih scenarija preko HTTP interfejsa sa različitim korisničkim nalozima. Na taj način proverava se da pojedinačno testirane komponente zajedno ostvaruju očekivano ponašanje, naročito u scenarijima koji uključuju različite korisničke uloge, autorizaciju i segmentaciju podataka.

U trenutnoj verziji sistema automatizovano testiranje obuhvata 369 testova, raspoređenih po nivoima kao što je prikazano u tabeli 3.3. Najveći broj čine servisni testovi slučajeva upotrebe i domenski testovi poslovnih pravila, dok poseban značaj imaju testovi autorizacije i izolacije podataka, kao jezgra istraživačkog dela rada.

Tabela 3.3: Broj automatizovanih testova po nivoima

Nivo testiranja	**Broj testova**
Domenski (jezgro i poslovna pravila)	101
Autorizacija i izolacija podataka	76
Servisni (slučajevi upotrebe)	172
Evidencija aktivnosti i bezbednosnih događaja	5
Infrastruktura i indeksi baze	15
Ukupno	**369**

Integracioni testovi API sloja izvršavaju se posebno, u punom okruženju sa pokrenutim bazama podataka, pa nisu obuhvaćeni navedenim brojem od 369.

Opisani pristup testiranju pruža proveru najvažnijih poslovnih pravila i kritičnih tokova razvijenog sistema, ali prolazak testova sam po sebi ne dokazuje potpunu

ispravnost softvera. U trenutnoj verziji nije uvedena metrika pokrivenosti koda testovima, a manuelna provera korisničkog interfejsa nije formalizovana kao zaseban automatizovani proces. Rezultate testiranja zato treba posmatrati kao potvrdu ponašanja u obuhvaćenim scenarijima, a ne kao tvrdnju da su proverene sve moguće kombinacije ulaza i stanja sistema.

Ograničenja sistema

Razvijena aplikacija realizuje funkcionalnosti predviđene obimom rada, ali trenutna implementacija ima ograničenja koja je potrebno uzeti u obzir prilikom razmatranja njene dalje primene. Ona se prvenstveno odnose na okruženje u kome se sistem izvršava, način raspoređivanja infrastrukturnih komponenti i pojedine tehničke odluke donete u okviru razvoja.

Sistem je predviđen za lokalno pokretanje korišćenjem Docker Compose okruženja i nije postavljen na javno dostupnu infrastrukturu u oblaku (engl. cloud). Sistemi za upravljanje bazama podataka, PostgreSQL i MongoDB, u trenutnoj konfiguraciji izvršavaju se kao pojedinačne instance, bez konfigurisane replikacije, horizontalnog skaliranja i drugih mehanizama visoke dostupnosti. Takva konfiguracija odgovara potrebama razvoja i demonstracije sistema, ali ne predstavlja infrastrukturu namenjenu produkcionom radu sa zahtevima za visoku dostupnost.

Korišćenje dve baze podataka uvodi i granicu u pogledu transakcione konzistentnosti. Sistemi za upravljanje bazama podataka, PostgreSQL i MongoDB, nisu obuhvaćeni zajedničkom distribuiranom transakcijom, pa se veze između podataka smeštenih u različitim bazama održavaju na aplikacionom nivou. Ovo je u skladu sa modelom perzistencije opisanim u poglavlju 3.4, ali bi kod složenijeg distribuiranog okruženja zahtevalo dodatne mehanizme za obradu parcijalnih neuspeha i očuvanje konzistentnosti između sistema za skladištenje podataka.

Budući da je JMBG jedinstveni identifikacioni podatak o ličnosti, njegova primena u produkcionom okruženju zahtevala bi dodatnu zaštitu pri čuvanju (na primer šifrovanje) i dosledno poštovanje načela minimizacije podataka, u skladu sa propisima o zaštiti podataka o ličnosti. U trenutnom prototipu ovi mehanizmi nisu implementirani i predstavljaju poznato ograničenje.

Pozivnice za deljenje pristupa u trenutnoj implementaciji ne šalju se putem integrisanog servisa elektronske pošte, već se koriste tokeni i odgovarajući linkovi koje aplikacija generiše. Slično tome, sistem nema mehanizam za slanje obaveštenja korisnicima putem spoljnog servisa za elektronsku poštu.

Automatsko ocenjivanje stanara otvara i pitanje zaštite podataka o ličnosti. U prototipu se ocena računa samo iz postojećih operativnih zapisa i vidljiva je isključivo agenciji koja upravlja nekretninom. Produkciona primena bi, međutim, zahtevala pravni osnov za obradu i mehanizme za ostvarivanje prava stanara na uvid i prigovor, što u trenutnoj verziji nije implementirano.

Konačno, razvojna konfiguracija baze podataka koristi automatsko kreiranje šeme, dok migracije šeme nisu deo trenutnog izvršnog procesa aplikacije. Ovakav pristup pojednostavljuje postavljanje lokalnog razvojnog okruženja, ali bi za dugoročno održavanje produkcione verzije bilo potrebno uvesti kontrolisani postupak verzionisanja i migracije baze podataka.

Navedena ograničenja ne odnose se na odsustvo osnovnih funkcionalnih zahteva sistema, već prvenstveno na način njegovog izvršavanja i nivo infrastrukturne pripremljenosti trenutne verzije. Njihovo jasno izdvajanje određuje granice u kojima su implementacija i rezultati ovog rada razmatrani, a istovremeno pruža osnov za definisanje mogućih pravaca daljeg razvoja.

Glava 4

Zaključak

Cilj ovog master rada bio je proučavanje kontrole pristupa i segmentacije podataka u višekorisničkim veb sistemima, sa posebnim osvrtom na situacije u kojima prava korisnika zavise ne samo od njihove uloge, već i od odnosa prema konkretnim resursima. Kao praktični okvir za razmatranje ovog problema projektovana je i implementirana veb aplikacija za upravljanje nekretninama i procesom izdavanja. Domen upravljanja nekretninama pogodan je za ovakvo razmatranje jer istim poslovnim resursima pristupaju vlasnici, agencije i stanari, pri čemu njihove mogućnosti i obim dostupnih podataka zavise od konkretnog odnosa prema posmatranoj nekretnini.

Aplikacija je realizovana primenom klijent–server arhitekture, sa jasno razdvojenim odgovornostima klijentskog i serverskog dela sistema. Serverska aplikacija organizovana je kroz više slojeva, pri čemu su poslovna pravila u domenskom sloju odvojena od orkestracije poslovnih operacija u sloju aplikacionih servisa i od infrastrukturnih detalja, uz primenu repozitorijuma, obrasca Unit of Work i povezivanja komponenti aplikacije sa mehanizmom ubrizgavanja zavisnosti. Ovakva organizacija omogućila je da se poslovna pravila i kontrola pristupa realizuju u serverskom delu, nezavisno od načina njihovog predstavljanja u korisničkom interfejsu.

Poseban značaj u radu ima model autorizacije i segmentacije podataka. Korisnička uloga određuje osnovne mogućnosti korisnika, dok se pristup postojećim poslovnim resursima određuje na osnovu konkretnog odnosa korisnika prema resursu. Vlasništvo, ugovorni odnos, zakup i eksplicitno dodeljena prava čine osnove na kojima se određuje dozvoljeni obim pristupa. Na ovaj način korisnici sa istom ulogom ne moraju imati pristup istim podacima, već skup dostupnih informacija zavisi od njihovog odnosa prema konkretnim resursima. Segmentacija podataka sprovodi

se na serverskoj strani, čime je omogućeno da više korisnika radi nad zajedničkim poslovnim resursima uz odgovarajuću izolaciju podataka.

Za čuvanje podataka kombinovani su relacioni i dokumentni model. PostgreSQL se koristi za strukturirane poslovne podatke i njihove međusobne veze, dok se MongoDB koristi za dokumentne i druge evidencije kojima više odgovara fleksibilniji model podataka. Za čuvanje binarnog sadržaja dokumenata primenjen je GridFS. Ovakva podela omogućila je da se različite vrste podataka čuvaju u modelu koji odgovara njihovoj strukturi i načinu korišćenja, uz njihovo povezivanje u okviru jedinstvene aplikacije.

Razvijeno rešenje obuhvata ključne procese upravljanja nekretninama i izdavanja, uključujući uspostavljanje odnosa između vlasnika, agencija i stanara, upravljanje ugovorima i zakupima, kao i dodeljivanje ograničenih prava pristupa. Pored toga, implementirane su funkcionalnosti za vođenje finansijskih evidencija, upravljanje dokumentacijom, generisanje PDF ugovora i izveštaja, obradu zahteva i žalbi, kao i ocenjivanje stanara u okviru agencijskog modela izdavanja. Dodatne funkcionalnosti, poput evidencije aktivnosti i bezbednosnih događaja, kontakata, beležaka, zadataka i lista za kupovinu, predstavljaju proširenja osnovne namene sistema.

Provera razvijenog rešenja obuhvatila je različite nivoe testiranja, sa posebnim značajem provere poslovnih pravila, autorizacije i izolacije podataka između korisnika. Rezultati testiranja potvrđuju ponašanje sistema u obuhvaćenim scenarijima, ali ne predstavljaju tvrdnju o potpunoj ispravnosti u svim mogućim uslovima. Pored funkcionalnih rezultata, razmotrena su i ograničenja trenutne implementacije. Sistem je prvenstveno razvijen i testiran u lokalnom okruženju zasnovanom na Docker Compose-u, bez mehanizama visoke dostupnosti i horizontalnog skaliranja. PostgreSQL i MongoDB su odvojeni sistemi perzistencije i nisu obuhvaćeni zajedničkom distribuiranom transakcijom, dok integracija sa spoljnim servisima za slanje elektronske pošte i drugih obaveštenja nije deo trenutnog rešenja.

Dalji razvoj sistema mogao bi biti usmeren na unapređenje produkcione infrastrukture, automatizaciju procesa isporuke i migracije podataka, kao i uvođenje mehanizama za skaliranje, visoku dostupnost i napredniji nadzor rada sistema. Moguće je i proširenje postojećih poslovnih funkcionalnosti i integracija sa spoljnim servisima za obaveštavanje korisnika. Ovakva unapređenja predstavljala bi nastavak postojećeg rešenja, bez potrebe za promenom njegovih osnovnih poslovnih modela.

Rezultat rada pokazuje da se u višekorisničkom veb sistemu kontrola pristupa ne mora zasnivati isključivo na korisničkoj ulozi, već se odluka može dopuniti in-

formacijama o konkretnom odnosu korisnika prema resursu i eksplicitno dodeljenim pravima. Na osnovu tako utvrđenog obima pristupa moguće je sprovesti segmentaciju podataka tako da različiti korisnici rade nad zajedničkim poslovnim resursima, ali dobijaju samo one operacije i podatke koji odgovaraju njihovom položaju u sistemu. Razvijena aplikacija za upravljanje nekretninama i procesom izdavanja poslužila je kao praktični okvir za primenu i proveru ovog pristupa, povezujući poslovne zahteve domena sa arhitektonskim i implementacionim principima razmatranim u radu.

Glava 5

Upotreba alata zasnovanih na veštačkoj inteligenciji

Tokom izrade ovog rada korišćeni su ChatGPT i Claude Code kao pomoćni alati zasnovani na veštačkoj inteligenciji. ChatGPT je korišćen kao podrška pri radu sa tekstom, uključujući preformulisanje i jezičko ujednačavanje pojedinih delova rada, kao i za pronalaženje relevantne literature i proveru pojedinih informacija. Claude Code je korišćen prvenstveno tokom razvoja programske implementacije, za implementiranje i testiranje pojedinih funkcionalnosti sistema, kao i za generisanje šablona za PDF dokumente i pojedinih dijagrama korišćenih u radu. Predlozi i sadržaji dobijeni korišćenjem ovih alata nisu preuzimani bez dodatne provere. Teorijske tvrdnje i informacije proveravane su na osnovu relevantne stručne literature, dok je ispravnost programske implementacije proveravana odgovarajućim testovima. Sadržaji uključeni u konačnu verziju rada dodatno su pregledani, korigovani i prilagođeni temi i ciljevima master rada.

Bibliografija

[1] ASGI. ASGI documentation: ASGI specification. (Pristupljeno: avgust 2026). Dostupno na: `https://asgi.readthedocs.io/`.

[2] Axios. Axios documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://axios-http.com/docs/intro`.

[3] Alex Banks and Eve Porcello. *Learning React: Modern Patterns for Developing React Apps*. O'Reilly Media, 2 edition, 2020.

[4] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. Argon2 memory-hard function for password hashing and proof-of-work applications. RFC 9106, Internet Engineering Task Force, 2021. (Pristupljeno: avgust 2026). Dostupno na: `https://www.rfc-editor.org/rfc/rfc9106`.

[5] Boris Cherny. *Programming TypeScript: Making Your JavaScript Applications Scale*. O'Reilly Media, 2019.

[6] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 8 edition, 2003.

[7] Dimedia nekretnine. Dimedia nekretnine – program za agencije za nekretnine, 2026. (Pristupljeno: avgust 2026). Dostupno na: `https://www.dimedianekretnine.com/o-programu`.

[8] Docker, Inc. Docker compose documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://docs.docker.com/compose/`.

[9] Docker, Inc. Docker documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://docs.docker.com/`.

[10] ESTATE Sistem. ESTATE sistem – softver za upravljanje nekretninama, 2026. (Pristupljeno: avgust 2026). Dostupno na: `https://estate.rs/`.

[11] FastAPI. FastAPI documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://fastapi.tiangolo.com/`.

[12] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[13] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. (Pristupljeno: avgust 2026). Dostupno na: `https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[14] Philip W. L. Fong. Relationship-based access control: Protection model and policy language. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 191–202, 2011.

[15] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[16] Caleb Hattingh. *Using Asyncio in Python: Understanding Python's Asynchronous Programming Features*. O'Reilly Media, 2020.

[17] Vincent C. Hu, David Ferraiolo, D. Richard Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. Guide to attribute based access control (ABAC) definition and considerations. Technical Report NIST Special Publication 800-162, National Institute of Standards and Technology, 2014. (Pristupljeno: avgust 2026). Dostupno na: `https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-162.pdf`.

[18] Michael Jones, John Bradley, and Nat Sakimura. JSON web token (JWT). RFC 7519, Internet Engineering Task Force, 2015. (Pristupljeno: avgust 2026). Dostupno na: `https://www.rfc-editor.org/rfc/rfc7519`.

[19] Michael Jones and Dick Hardt. The OAuth 2.0 authorization framework: Bearer token usage. RFC 6750, Internet Engineering Task Force, 2012. (Pristupljeno: avgust 2026). Dostupno na: `https://www.rfc-editor.org/rfc/rfc6750`.

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

[21] Bill Lubanovic. *FastAPI: Modern Python Web Development*. O'Reilly Media, 2023.

[22] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[23] Meta Platforms, Inc. React documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://react.dev/`.

[24] Microsoft. TypeScript documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://www.typescriptlang.org/docs/`.

[25] MongoDB, Inc. MongoDB manual. (Pristupljeno: avgust 2026). Dostupno na: `https://www.mongodb.com/docs/`.

[26] OWASP Foundation. Authorization cheat sheet. (Pristupljeno: avgust 2026). Dostupno na: `https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html`.

[27] OWASP Foundation. Password storage cheat sheet. (Pristupljeno: avgust 2026). Dostupno na: `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html`.

[28] Harry Percival and Bob Gregory. *Architecture Patterns with Python*. O'Reilly Media, 2020.

[29] PostgreSQL Global Development Group. PostgreSQL documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://www.postgresql.org/docs/`.

[30] Pydantic. Pydantic documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://docs.pydantic.dev/`.

[31] Pydantic. Pydantic settings documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://docs.pydantic.dev/latest/concepts/pydantic_settings/`.

[32] pytest Development Team. pytest documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://docs.pytest.org/`.

[33] Python Software Foundation. asyncio – asynchronous I/O. (Pristupljeno: avgust 2026). Dostupno na: `https://docs.python.org/3/library/asyncio.html`.

[34] Python Software Foundation. Python documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://docs.python.org/3/`.

[35] Luciano Ramalho. *Fluent Python*. O'Reilly Media, 2 edition, 2022.

[36] React Hook Form. React hook form documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://react-hook-form.com/`.

[37] React Router. React router documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://reactrouter.com/`.

[38] Redocly. Redoc CE: Open source API documentation tool. (Pristupljeno: avgust 2026). Dostupno na: `https://redocly.com/docs/redoc`.

[39] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. O'Reilly Media, 2013.

[40] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.

[41] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[42] Mark Seemann and Steven van Deursen. *Dependency Injection Principles, Practices, and Patterns*. Manning Publications, 2019.

[43] Yury Selivanov. PEP 492 – coroutines with async and await syntax, 2015. (Pristupljeno: avgust 2026). Dostupno na: `https://peps.python.org/pep-0492/`.

[44] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Education, 7 edition, 2019.

[45] SQLAlchemy. SQLAlchemy 2.0 documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://docs.sqlalchemy.org/`.

[46] Swagger. Swagger UI. (Pristupljeno: avgust 2026). Dostupno na: `https://swagger.io/tools/swagger-ui/`.

[47] Uvicorn. Uvicorn documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://uvicorn.dev/`.

[48] Vivio. Vivio – aplikacija za stanare i upravnike zgrada, 2026. (Pristupljeno: avgust 2026). Dostupno na: `https://vivio.pro/en/homepage/`.

[49] Volvox. RELPER – program za nekretnine, 2026. (Pristupljeno: avgust 2026). Dostupno na: `https://www.relper.com/`.

[50] WHATWG. DOM standard. (Pristupljeno: avgust 2026). Dostupno na: `https://dom.spec.whatwg.org/`.

[51] Robin Wieruch. *The Road to React*. Self-published, 2025. Samostalno izdata, kontinuirano ažurirana knjiga. (Pristupljeno: avgust 2026). Dostupno na: `https://www.roadtoreact.com/`.

[52] Zod. Zod documentation. (Pristupljeno: avgust 2026). Dostupno na: `https://zod.dev/`.

Biografija autora

Momčilo Knežević rođen je 17. novembra 1998. godine u Beogradu. Srednju školu završio je 2017. godine u Aleksandrovcu, nakon čega je upisao osnovne akademske studije na Matematičkom fakultetu Univerziteta u Beogradu, modul Informatika. Osnovne akademske studije završio je 2022. godine, nakon čega je nastavio školovanje na master akademskim studijama na Matematičkom fakultetu Univerziteta u Beogradu.