Web aplikacija za iznajmljivanje stanova

Bašan, Andrija

Sveučilište Jurja Dobrile

Fakultet informatike u Puli

Andrija Bašan

Web aplikacija za iznajmljivanje stanova

Završni rad

Pula, rujan, 2023.

Sveučilište Jurja Dobrile

Fakultet informatike u Puli

Andrija Bašan

Web aplikacija za iznajmljivanje stanova

Završni rad

JMBAG: 0303088182

Redovni student

Studijski smjer: Informatika

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Kolegij: Programiranje

Mentor: izv. prof. dr. sc. Tihomir Orehovački

Pula, rujan, 2023.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Andrija Bašan, kandidat za prvostupnika informatike ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljeni način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

U Puli, 18.09.2023.

Andrija Bašan

I ZJAVA O KORIŠTENJU
AUTORSKOG DJELA

Ja, Andrija Bašan dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj Završni rad pod nazivom „Web aplikacija za iznajmljivanje stanova“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 18.09.2023.

Potpis

Sažetak

Oglasi se u današnje vrijeme skoro ekskluzivno izdaju online, to se također odnosni na oglase za izdavanje nekretnina. Današnje popularne aplikacije su često zastarjele i glavna namjena im nije za izdavanje nekretnina. “Najam stanova” je web aplikacija čiji je cilj korisnicima aplikacije omogućiti jednostavniji proces traženja i, izdavanja oglasa za nekretnine. Ovo postiže uz pomoć mogućnosti filtriranja oglasa i trenutnog slanja poruka među korisnicima, također ima pregledno i jednostavno sučelje za korištenje. Cilj ovog završnog rada je napraviti dokumentaciju koja objašnjava kako funkcionira ova aplikacija te koje su se tehnologije koristile u izradi aplikacije.

Ključne riječi

Vue.js, MongoDB, Node.js, iznajmljivanje, oglasi, nekretnine

Abstract

Ads today are almost exclusively published online, this is also true for real estate ads. Today’s popular applications are usually old and they’re main use isn’t for real estate ads. “Najam stanova” is a web application, with its main goal being to enable the users to have an easier process of finding as well as publishing ads for real estate. It achieves this by allowing the user to filter the ads as well as instant messaging between users, it also has a clear and easy-to-use user interface. The aim of this thesis is to create documentation that explains how this application works as well as which technologies were used to create the application.

Keywords

Vue.js, MongoDB, Node.js, renting, ads, real estate

Sadržaj

1. Uvod

Kako se tehnologija razvija i omogućava nam, osim ostalog, nove i brže načine komunikacije. Jedna od posljedica toga je to što se u današnjici oglasi za nekretnine najčešće postavljaju i traže online. Podstanarima naći mjesto za život može biti dosta kompliciran i dugotrajan proces, ali im tehnologija ovaj proces može dosta ubrzati. Najmodavcima također ovo ide u korist kada izdaju stan i traže prikladnog podstanara.

Trenutačna programska rješenja su često zastarjela, te im nije svrha prvenstveno za izdavanje nekretnina. To dovodi do otežavanje korištenja korisnicima. Jedno od najpopularnijih danas su web aplikacije kao što su Facebook i Njuškalo. Često imaju sučelje koje je dosta teško za koristiti, zahtijevaju da budemo dosta specifični, i ne daju nam na znanje unaprijed što možemo očekivati. Također često dopuštaju ljudima koji izdajaju oglase da daju nepotpune informacije, kao što je oglas bez slika, bez navedene cijene i sl.

Cilj ove web aplikacije je bilo napraviti primjer web aplikacije namijenjene za izdavanje i pregledavanje oglasa nekretnina, te time stvoriti temelj koje bi buduće ili postojeće web aplikacije mogle koristiti kao primjer uz pomoć kojeg mogu poboljšati svoju aplikaciju. Korištenjem novih tehnologija, ova aplikacija pokušava poboljšati iskustvo traženja podstanara, kao i traženje stana za podstanare. Kako se tehnologije razvijaju, tako bi i aplikacije trebale koristiti nove tehnologije kako bi poboljšali iskustvo i korištenje aplikacije svim korisnicima. Aplikacija omogućava korisnicima izdavanje oglasa, izmjenu tih istih oglasa nakon oglašavanja, kao i brisanje oglasa. Omogućava korisnicima da filtriraju oglase po svojim kriterijima, te ako ih zanima neki oglas mogu kontaktirati korisnika i imati razgovore u stvarno vrijeme s njima ako su spojeni na web aplikaciju.

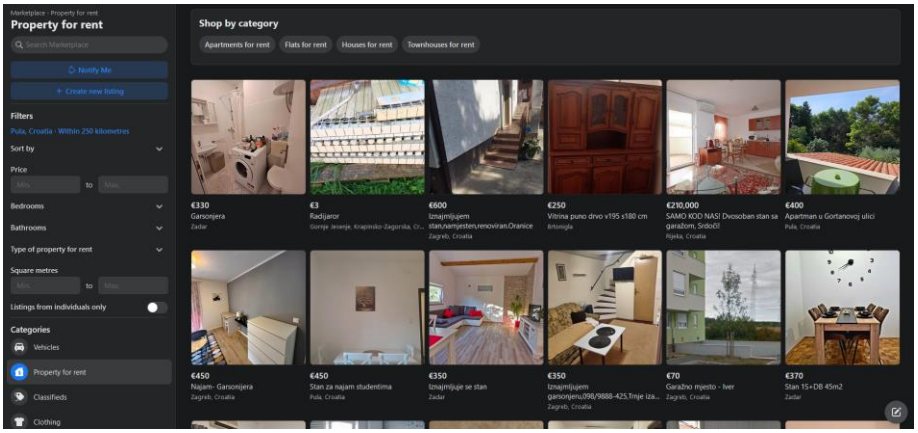
Ovaj završni rad je dokumentacija za ovu web aplikaciju. Tijekom završnog rada bit će razrađeno koje su tehnologije korištene i te same tehnologije će biti kratko pojašnjene. Nakon toga će biti pojašnjena sama aplikacija, njen backend i frontend. Frontend je dio aplikacije koje korisnik vidi i koristi, sastoji se od korisničkog sučelja i koda koji se izvodi na strani korisnika. Za izgradnju frontenda se koristio, Vue.js kao JavaScript framework za oblikovanje korisničkog sučelja, kao i brojne biblioteke dostupne za nadogradnju, neki od kojih su se koristili za komunikaciju s backendom, dok su se druge koristile za poboljšanje korisničkog sučelja. Backend je dio aplikacije

koji sadrži bazu podataka, kao i rute koje primaju HTTP zahtjeve koje šalju korisnici, te šalju odgovore na te zahtjeve. Backend se zasniva na Node.js-u kao radnom okruženju za pokretanje JavaScripta na strani servera, te je korištena biblioteka Express.js koja olakšava namještavanje portova, kao i primanje i slanje zahtjeva na točne rute. Kako bi se omogućilo slanje trenutačnih poruka između korisnika, korištena je biblioteke socket.io, koja omogućuje socket veze korisnika s backendom. Za bazu podatka je korišten MongoDB, dok je cijeli backend pokrenut u Dockeru kako bi omogućilo korištenje baze podataka lokalno, kao i lakše korištenje backenda na drugačijim računalima sa što manje problema. Cijeli projekt je pisan u uređivaču teksta Visual studio code, te je spremljen na GitHub kontrole verzioniranja.

2. Kritička analiza trenutačnih riješenja

Najpopularnija rješenja danas za pronalaženje stanova su online oglasi. Oglasi u sebi uglavnom sadrže sliku, lokaciju, cijenu i neki opis stana. Korisnici mogu pretraživati oglase po naslovu, kao i nekim atributima oglasa, uglavnom lokaciju i cijenu.

Jedno od najpopularnijih aplikacija za ovo je Facebook marketplace [1]. Korisničko sučelje je prikazano na slici 1. Jedno od najvećih prednosti Facebook marketplacea je veliki broj korisnika, kao i verifikacija korisnika koju pruža, jer možemo pregledati profil korisnika koji izdaje oglas, te vrlo lagano vidjeti ako je možda lažni korisnik, ili nešto drugo. Što se tiče samog sučelja, možemo vidjeti da ima dosta siromašne mogućnosti filtriranja, kao i mali broj informacija na pregledu oglasa. Razlog ovome je što Facebook marketplace nema glavni fokus na oglase nekretnina, već generalne oglase za prodaju.

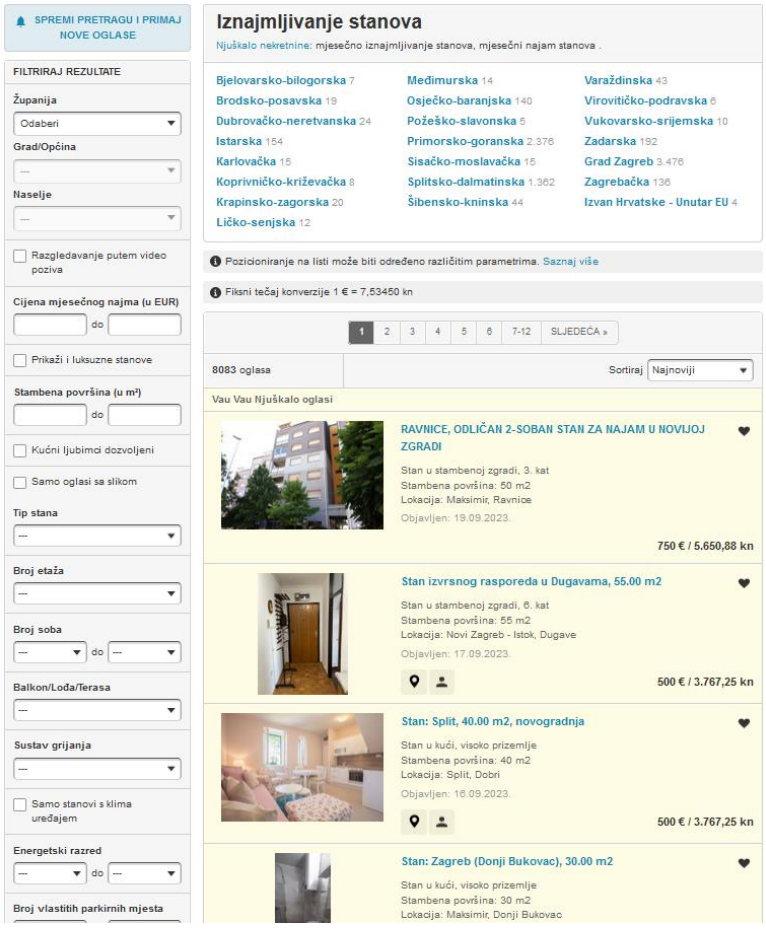


Slika 1. Korisničko sučelje Facebook marketplacea

Kada stisnemo na specifičan oglas možemo vidjeti na slici 2 da ne dobijemo previše dodatnih informacija, i sve dodatne informacije ovisi o tome što izdavač oglasa stavi u oglas, također korisnik ne može nikako filtrirati oglase po ovim pojedinostima. Ako korisnika zanimaju neke specifične informacije, kao što je dostupnost stana tijekom sezone, ako su dopušteni ljubimci i druge pojedinosti, korisnik bi morao kontaktirati izdavača oglasa, što značajno usporava proces pronalaženja stanova. Facebook marketplace bi ubrzao proces pronalaženja stanova dodavanjem više pojedinosti o stanovima, te omogućio filtriranje po tim pojedinostima.

Slika 2. Detalji oglasa na Facebook marketplaceu

Drugo najpopularnije rješenje je njuškalo [2]. Korisničko sučelje je prikazano na slici 2, možemo vidjeti da za razliku od Facebook marketplacea ima značajno više opcija za filtriranje oglasa, te daje izdavačima brojne opcije za dodavanja informacija o oglasima. Međutim korisničko sučelje je vrlo zastarjelo, i dosta nepregledno, kako bi uopće vidjeli sve moguće filtere moramo se spustiti na sredinu stranice. Drugi veliki nedostatak je dopisivanje između korisnika, koje nije trenutačno, već se vodi preko maila, te zahtjeva da korisnici osvježe stranicu kako bi vidjeli ako su primili novu poruku. Stoga dosta korisnika preferira komunikaciju preko telefona, međutim neki izdavači oglasa ne žele izdavati svoj broj javno, i nekad nisu u mogućnosti se javiti, ali u isto vrijeme žele naći podstanar za svoj stan. Njuškalo korisnici bi imali dosta koristi od novog i bolje korisničkog sučelja, kao i boljeg sučelja za komunikaciju koji dopušta trenutačnu komunikaciju.



Slika 3. Korisničko sučelje njuškala

3. Korištene tehnologije

Prilikom izrade web aplikacije korištene su razne tehnologije. Za pohranjivanje i praćenje razvoja koda korišten je GitHub, dok je za samo pisanje koda korišten Visual Studio Code. Frontend aplikacije je izgrađen uz pomoć Vue.js-a, dok se za backend koristio Node.js i Express.js. MongoDB je u ovoj web aplikaciji korišten kao baza podatka, te su backend i baza podataka spremljeni i pokrenuti uz pomoć Dockera.

3.1 Vue.js

Vue.js je JavaScript framework namijenjen za izradu korisničkih sučelja i aplikacija na jednoj stranici. Temelji se na HTML-u, CSS-u i JavaScriptu. Omogućava brzo i lako korištenje te integraciju s drugim projektima i programskim bibliotekama. Jedno od najbitnijih elemenata Vue.js-a je reaktivnost i deklarativno iscrtavanje elementa. Često se opisuje kao progresivni framework, jer omogućava korištenje malog dijela, kao i sve funkcionalnosti koje pruža. Dopušta početnicima kao i iskusnim programerima da naprave interaktivne i responzivne web aplikacije, od malih prototip, do velikih aplikacija [3]. Na slici 4 je prikazan primjer Vue.js koda.

```
<template>
  <div class="container">
    <div class="row">
      <!-- Sidebar -->
      <div class="col-3">
        <sidebar :props="minMax" />
      </div>

      <!-- Oglasi -->
      <div class="col-9 row">
        <ad
          class="col-span-4 m-2"
          v-for="ad of this.adsData"
          :key="ad._id"
          :props="ad"
        ></ad>
      </div>
    </div>
  </div>
</template>

<script>
/* Imports */
import sidebar from "@/components/Sidebar.vue";
import ad from "@/components/Ad.vue";
import { ads } from "@/services";

export default {
  data() {
    return {
      adsData: null,
      minMax: null,
    };
  },
  async mounted() {
```

Slika 4. Primjer Vue.js koda

3.2 Node.js

Node.js je radno okruženje za izvođenje JavaScripta na serveru, radi na svim internet preglednicima i više operativnih sustava, kao što su Windows, Linux i MacOS. Jedan je od najpopularnijih tehnologija koja se koristi za razvoj web aplikacija, te se često za dinamičke web stranice. Omogućava dostup velikom broju biblioteka uz pomoć npm-a koje pomažu u razvoju web aplikacije. Pruža neblokirajuću arhitekturu vođenu događajima. Neblokirajuća znači da se mogu izvoditi druge naredbe, iako nije neka druga završena, što je jako bitno za server koji može primati veliki broj zahtjeva u isto vrijeme. Vođenu događajima znači da je aplikacija zasnovana na događajima i "callback" funkcijama. Ta arhitektura pruža mogućnosti razvoja skalabilnih i brzih mrežnih aplikacija. Svi ti atributi su razlozi zašto je Node.js zato jedna od najpopularnijih odabira za izgradnju backenda [4]. Na slici 5 je prikazan primjer Node.js koda.

```
/* Importovi */
import express from "express"
import cors from "cors"
import Chat from "../Models/Chat.js"
import Ad from "../Models/Ad.js"
import User from "../Models/User.js"
import auth from "./auth.js"
import {
    Server
}
from "socket.io"
import connect from '../initDB.js';

/* Konstante za spajanje */
const app = express()
const port = 3000

app.use(cors())
app.use(express.json())

await connect()
```

Slika 5. Primjer Node.js koda

3.4 Express.js

Express.js je popularan framework za node.js koji se koristi za izgradnju API-ova i web stranica. Olakšava izradu backenda, time što već u sebi ima riješene stvari kao što su određivanje ruta i namještanje portova, što omogućava puno bržu i jednostavniju izradu web aplikacije. Omogućava jednostavne metode za slanje HTTP zahtjeva na specifičnu rutu, koja onda izvodi ono što je određeno da ta ruta radi. Express.js također dopušta korištenje middleware-a, to su neke operacije koje zadajemo da se izvrše prije nego što se izvrše naredbe na samoj rutu, što olakšava izvođenje i preglednost koda [5]. Na slici 6 je prikazan primjer koda jedne rute napisan uz pomoć Express.js-a.

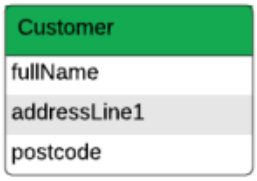
```
/* GET za specifični oglas */
app.get("/ads/:id", async (req, res) => {
    try {
        const id = req.params.id
        const ad = await Ad.findById(id).populate("createdBy").select("-pass");
        res.json(ad)
    } catch (e) {
        console.error(e)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 6. Primjer Express.js koda

3.5 MongoDB

MongoDB je nerelacijska baza podataka, što znači da ne sprema svoje podatke u tablici, već u nekoj drugoj strukturi podataka, u slučaju MongoDB to su kolekcije i dokumenti u JSON formatu. U relacijskim bazama tablice imaju međusobne odnose preko ključeva, dok MongoDB ima kolekcije koje su nestrukturirane, ovu razliku možemo vidjeti na slikama 7 i 8, koje prikazuju primjer podataka unutar relacijske, kao i unutar nerelacijske baze podataka. Kao i druge baze podataka ima mogućnost indeksiranja baze podataka kako bi se upiti što prije izvodili. Prednost nerelacijskih baza podataka je to što se puno brže čitaju, te se i također puno brže izmjenjuju pošto

nemaju nikakvih odnosa međusobno [6]. Korištena je biblioteka mongoose unutar Node.js-a za spajanje i kreiranje shema za MongoDB, primjer koda za spajanje s MongoDB je prikazan na slici 9.



Slika 7. Relacijska baza podataka

Slika 8. Nerelacijska baza podataka

```
import mongoose from "mongoose";
import "dotenv/config";

export default () => {
    mongoose
        .connect(String(process.env.MONGODB_URI), {
            dbName: process.env.DB_NAME,
            user: process.env.DB_USER,
            pass: process.env.DB_PASS,
        })
        .then(() => {
            console.log('Mongodb spojen....');
        })
        .catch(err => console.log(err.message));

    mongoose.connection.on('connected', () => {
        console.log('Mongoose spojen na db...');
    });

    mongoose.connection.on('error', err => {
        console.log(err.message);
    });

    mongoose.connection.on('disconnected', () => {
        console.log('Mongoose je odspojen...');
    });

    process.on('SIGINT', () => {
        // @ts-ignore
        mongoose.connection.close(() => {
            console.log(
                'Mongoose je ugašen...'
            );
            process.exit(0);
        });
    })
};
```

Slika 9. Primjer koda napisanog uz pomoć mongoose biblioteke

3.6 Socket.io

Je biblioteka za Node.js koja omogućava uspostavljanje komunikacije između dva klijenta, uglavnom internet preglednika i servera. Omogućava socket veze između njih, koje omogućavaju konstantnu komunikaciju između dva klijenta nakon uspostavljanja veze. Prednost socket veze je to što nakon što je uspostavljena, klijenti čekaju događaj, i šalju događaje, te će se ovo odvijati dok se jedan od klijenta ne odspoji. Socket veze se često koriste u online video igricama, aplikacija za trenutačno dopisivanje, kolaboracijski alati i sl. Socket.io biblioteka jako olakšava stvaranje logike za uspostavljanje i održavanje socket veza, kao i slanje i čekanje događaja između klijenta [7]. Slika 10 prikazuje uspostavljanje socket veze, kao i definiranje događaja.

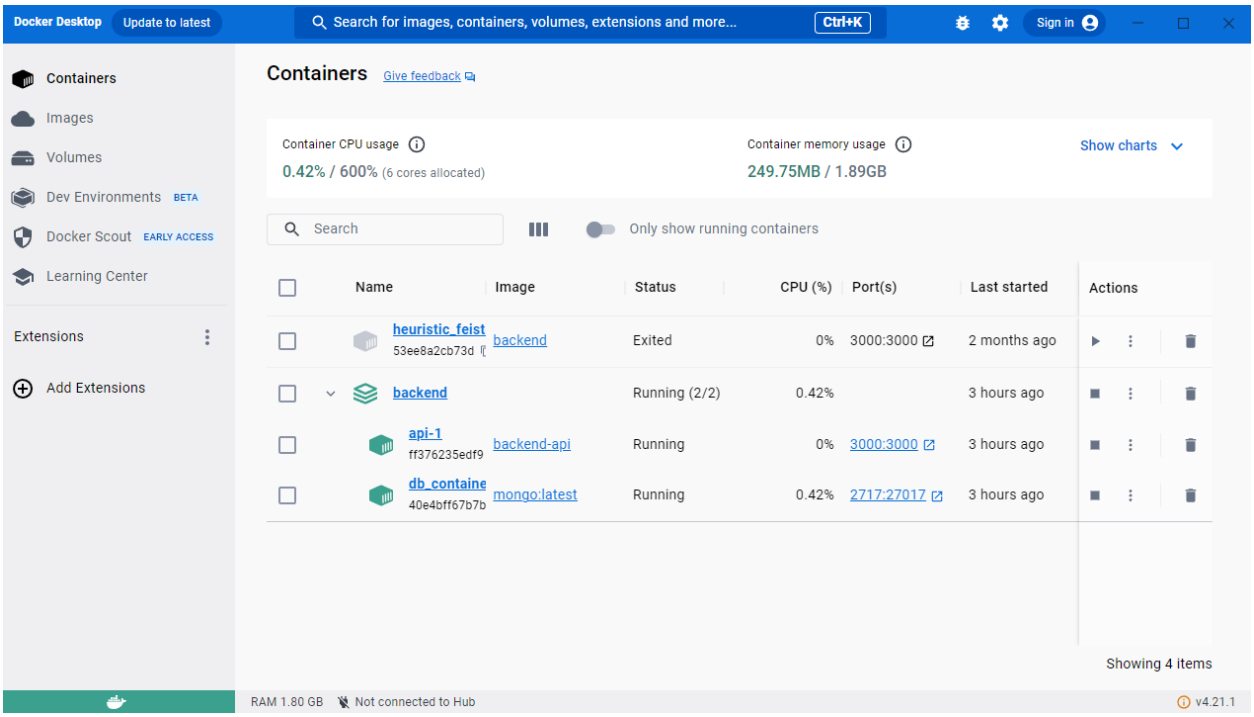
```
const httpServer = app.listen(port, () => {
    console.log(`Server listening on port ${port}`)
});

const io = new Server(httpServer, {
    cors: {
        origin: "http://localhost:8080",
        credentials: true,
    },
});

global.onlineUsers = new Map();

io.on("connection", (socket) => {
    console.log('Korisnik spojen');
    socket.on("add-user", (userId) => {
        global.onlineUsers.set(userId, socket.id);
    });

    socket.on("send-msg", (data) => {
        const sendUserSocket = global.onlineUsers.get(data.to);
        if (sendUserSocket) {
            io.to(sendUserSocket).emit("msg-recieve", data);
        }
    });

    socket.on('disconnect', () => {
        socket.disconnect();
        console.log('Korisnik odspojen');
    });
});
```

Slika 10. Primjer Socket.io koda

3.7 Docker

Docker je platforma za razvoj, isporuku i pokretanje aplikacija. Slično je u konceptu virtualnim mašinama, samo što umjesto da ima cijeli operacijski sustav u sebi, koristi “containere”, koji koriste isti operacijski sustav, i samo virtualiziraju na razini softwarea. Oni u sebi sadrže sve potrebno za pokretanje aplikacije kao što su konfiguracija i druge stvari o čemu ovisi. Prednost dockera je to što izbjegava probleme toga da ne radi na drugim sustavima pošto je virtualni prostor koji je konzistentan, potrebno je samo pokrenuti “container” i server može biti dostupan za korištenje vrlo brzo. U ovoj web aplikaciji je MongoDB stavljen u Docker te je i sam backend kod stavljen u Docker. Razlog zašto je MongoDB stavljen u Docker je kako bi imali puno bržu

komunikaciju s bazom podataka, te bi se mogla aplikacija pokretati offline [8]. Na slici 11 je prikazano sučelje dockera, na kojoj se mogu vidjeti containeri za backend (api-1) i za bazu podataka (db_container).



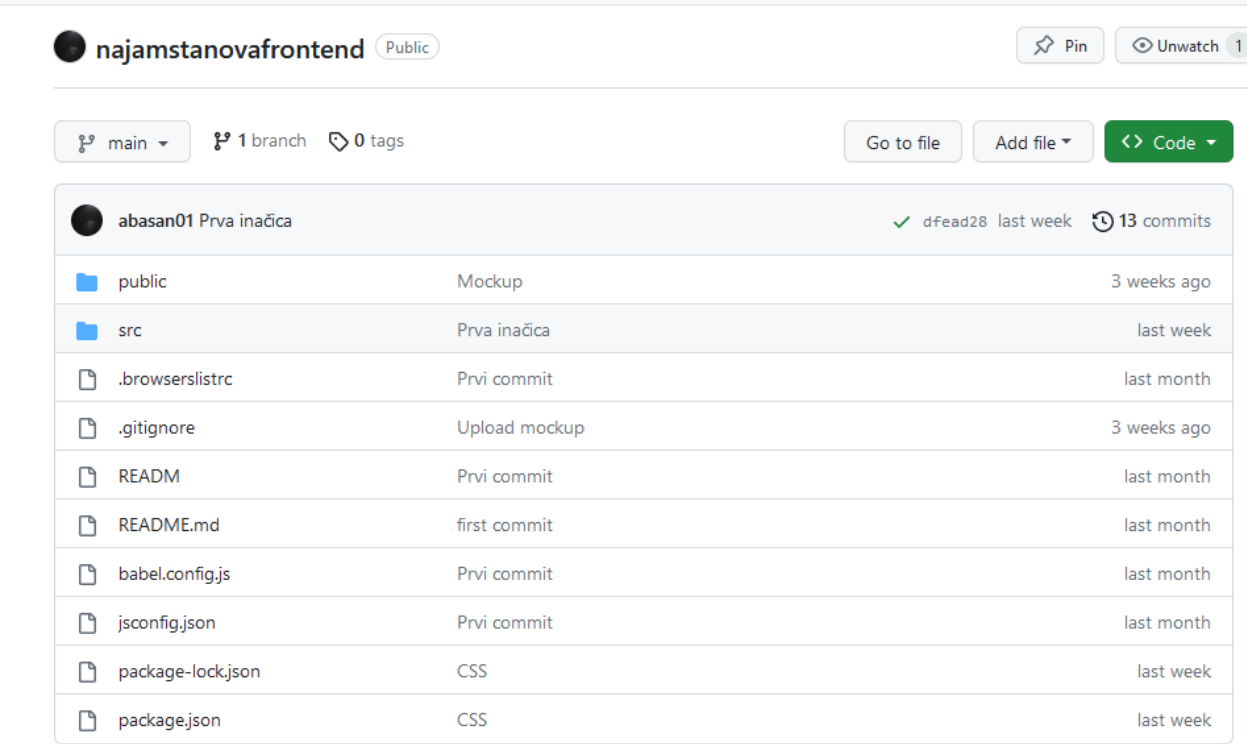
Slika 11. Docker sučelje

3.8 GitHub

GitHub je web platforma za kontroliranje verzija i kolaboraciju na razvoju softvera. Kontroliranje verzija omogućava korisnicima da prate i mijenjaju promijene na programskom projektu. Uz pomoću grana i spajanja olakšava rad na velikim i kompleksnim projektima, te ako nešto prođe krivu možemo se vratiti na stariju verziju, ili vidjeti točno koja promjena je izazvala probleme. Također omogućava više ljudi da usporedno rade na istom projektu [9]. Ovaj projekt se sastoji od dva GitHub projekta, jedan za frontend i jedan za backend:

- https://github.com/abasan01/najamstanovafrontend
- https://github.com/abasan01/najamstanovabackend.

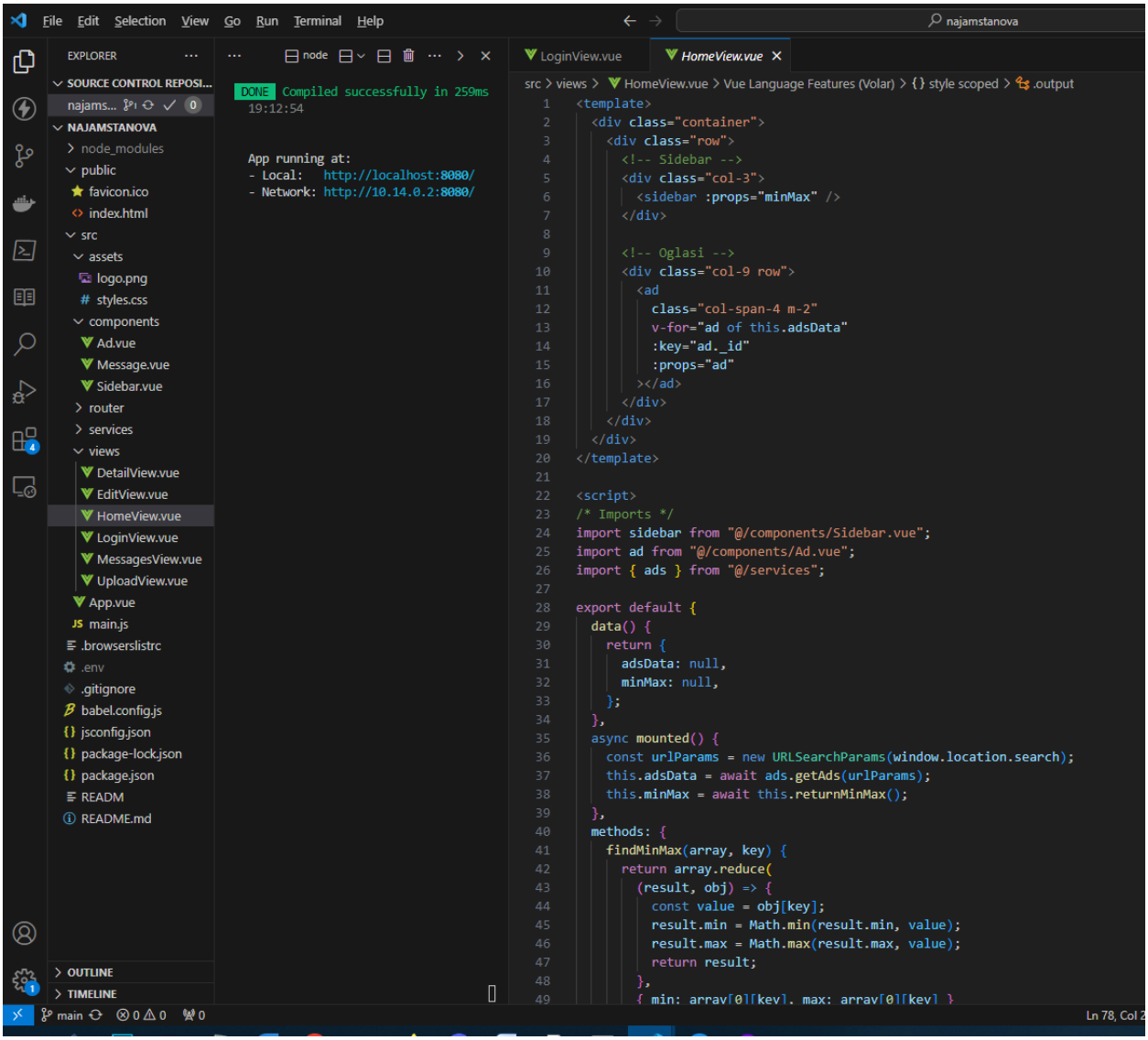
Primjer repozitorija možemo vidjeti na slici 12.



Slika 12. Repozitorij za frontend aplikacije

3.9 Visual Studio Code

Visual Studio Code je uređivač koda za razne operativne sustave, koji dolazi s potporom za više programskih jezika, kao i mogućnosti za pronalaženje greška u kodu kao i rješavanje istih greška. Omogućava visoku razinu prilagodbe korisničkog sučelja, također omogućava instalaciju raznih ekstenzija koje omogućavaju lakše i učinkovitije korištenje. Omogućava i integraciju s GitHubom kroz ekstenzije koje nam još više olakšava kontroliranje verzija i samo spremanje koda na GitHub [10]. Primjer korisničkog sučelja za Visual Studio Code je prikazan na slici 13.



Slika 13. Primjer Visual Studio Code sučelja

4. Funkcionalnosti

4.1 Dijagram slučajeva korištenja

Aplikacija ima dvije vrste korisnika, to su korisnici i gosti. Korisnik se smatra korisnikom koji ima svoj korisnički račun i prijavljen je, dok je gost korisnik bez korisničkog računa. Gost može samo gledati oglase i filtrirati ih, dok ih korisnik može objavljivati, izmjenjivati i brisati svoje oglase, te se dopisivati s drugim korisnicima. Za svaku funkcionalnost za koju je potreban korisnik

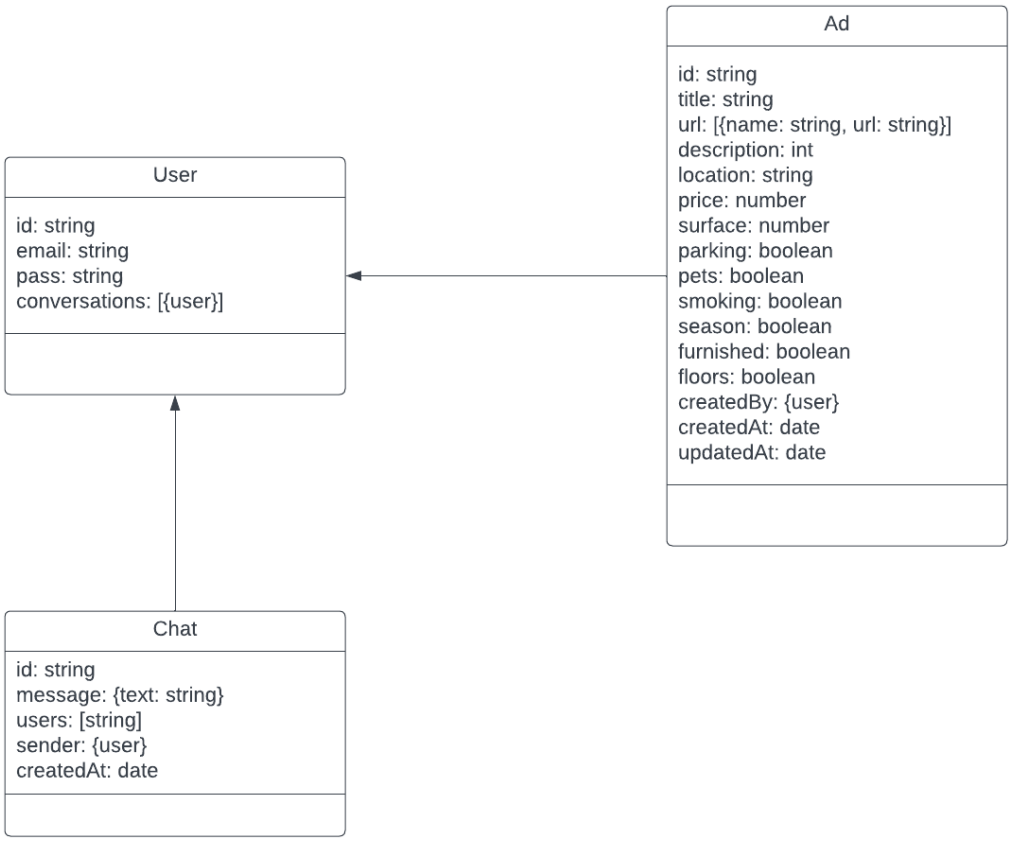
na backendu se provjerava ako je korisnik prijavljen. Use case dijagram je grafički prikaz mogućih interakcija korisnika s aplikacijom. Na slici 14 možemo vidjeti dijagram korištenja ove aplikacije. Jedna od opcija koju korisnik ima je pregled oglasa, te ako želi filtrirati prikaz oglasa. Nakon toga može odabrati neki oglas i vidjeti više o oglasu, te nakon toga može kontaktirati izdavača oglasa. Druga opcija je da korisnik napravi i objavi oglas koji će se prikazivati drugim korisnicima. Jedna od opcija je da korisnik pogleda svoje poruke, i šalje poruke. Zadnja opcija je da korisnik ažurira svoje trenutačne oglase, tako da ih ili izbriše ili izmijeni podatke u njima.

Slika 14. Dijagram slučajeva korištenja

4.2 Klasni dijagram

Klasni dijagram je strukturni dijagram koji pokazuje objekata unutar aplikacije te njihove atribute i odnose. Slika 15 prikazuje klasni dijagram unutar ove web aplikacije. Aplikacija u sebi ima tri glavne klase koje su oglas, korisnik i poruku. Oglas u sebi sadrži brojne atribute koji detaljno opisuju nekretninu koja se iznajmljuje. U sebi oglas sadrži potklasu za sliku, koja u sebi sadrži ime slike, te url slike, također pod atributom “createdBy” sadrži id korisnika koji se referira na korisnika koji je stvorio taj oglas, što omogućava lako spajanje korisnika s oglasom kako bi lako mogli potvrditi kojem korisniku pripada oglas u svrhe izmjene oglasa i kontaktiranje korisnika.

Korisnik sadrži malo podataka, samo email, te lozinku koja je u šifriranoj formi u bazi podataka. U sebi sadrži array objekta korisnika s kojima ima razgovore, kako bi lagano mogli spojiti korisnike s drugim korisnicima. Zadnji objekt unutar aplikacije je poruka, koji se sastoji od same poruke, liste korisnika između koja je ta poruka, te pošiljatelj poruke. Uz pomoću liste korisnika lagano odrediti kojem razgovoru pripada poruka, a uz pomoć pošiljatelja lako možemo odrediti tko je poslao poruku.



Slika 15. Klasni Dijagram

5. Backend

Backend se sastoji od strane aplikacije koja prima HTTP zahtjeve s frontenda, te vraća nazad odgovor na te zahtjeve. U ovoj aplikaciji se sastoji od baze podataka, koja je MongoDB, u

koju pohranjujemo podatke koje se šalju s frontenda. Za određivanje ruta i naredbi, te komunikacijom s bazom podataka, koje će se izvoditi na strani backenda su korišteni Index.js i Express.js. Cijeli backend se izvodi u Docker "containeru".

5.1 Docker

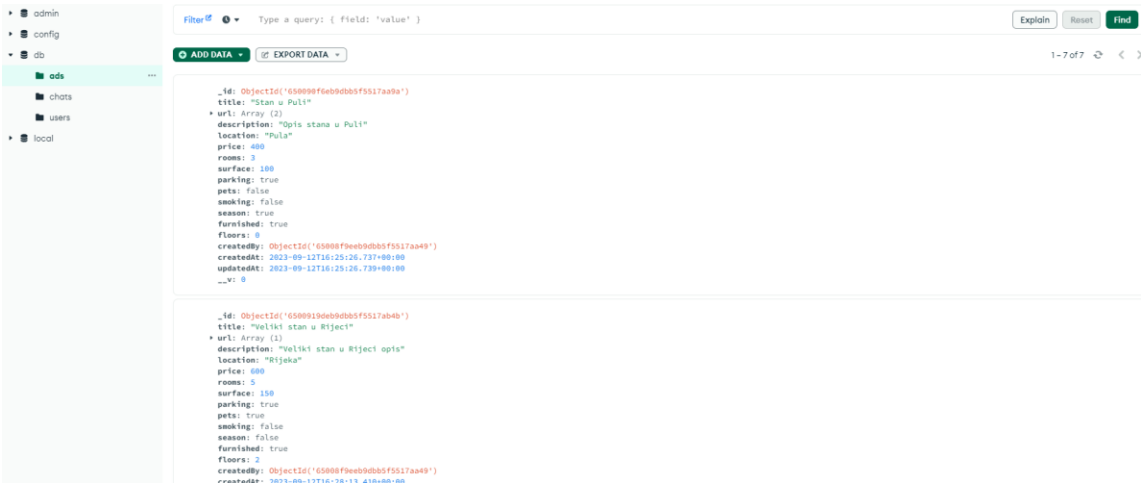
Korištenje Dockera za backend nam omogućava jednostavnije upravljanje, dijeljenje i korištenje našeg backenda. Osim toga, uz MongoDB u Dockeru, možemo lokalno imati našu bazu podataka koju možemo i offline koristiti, te je komunikacija puno brža. Na slici 16 je vidljivo kako je definiran servis za MongoDB, te smo također definirali da ćemo imati "volumes", što znači da će se svi podaci u bazi zadržati i nakon što smo ugasili i upalili naš backend. Također smo definirali komunikaciju između našeg backenda i baze podataka u dijelu za node API servis.

```yaml
version: "3.9"

services:
  # MongoDB services
  mongo_db:
    container_name: db_container
    image: mongo:latest
    restart: always
    ports:
      - 2717:27017
    volumes:
      - mongo_db:/data/db

  # Node API service
  api:
    build: .
    ports:
      - 3000:3000
    volumes:
      - .:/src
    environment:
      PORT: 3000
      MONGODB_URI: mongodb://mongo_db:27017
      DB_NAME: db
      NAME: najamstanova
    depends_on:
      - mongo_db

volumes:
  mongo_db: {}
```

Slika 16. Definiranje dockera

5.2 MongoDB

Nakon što je Docker pokrenut, mi se možemo spojiti na bazu podataka. Na slici 17 je prikazana baza podataka web aplikacije, u sučelju MongoDB Compass koji nam omogućava jednostavan pregled dokumenta i kolekcija u našoj bazi podataka.



Slika 17. MongoDB Compass sučelje

5.3 MongoDB sheme

U kodu je definirana shema za svaku kolekciju, na slici 18 možemo vidjeti shemu za kolekciju korisnika. Definirani su svi atributi, te koje su vrste, također možemo vidjeti da su osim vrste atributa, dodane i validacije. To su provjere koje se odvijaju prije nego što se spremi dokument, npr. ne smijemo spremiti dokument koji ima prazan email, i već je prisutan u bazi podataka.

```
import mongoose from 'mongoose';

const userSchema = new mongoose.Schema({
    email: {
        type: String,
        required: true,
        unique: true,
    },
    pass: {
        type: String,
        required: true,
    },
    conversations: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: "User"
    }],
    createdAt: {
        type: Date,
        immutable: true,
        default: () => Date.now(),
    }
});

const User = mongoose.model('User', userSchema);

export default User;
```

Slika 18. Shema korisnika

5.4 Autorizacija

Kako bi potvrdili da je korisnik prijavljen, te da je dao ispravne podatke za korisnički nalog trebamo autorizirati te podatke. Uz pomoć biblioteka za kriptografiju i nekoliko funkcija uspijevamo ovo napraviti. Funkcije za autorizaciju su prisutne u odvojenoj .js skripti te se pozivaju u glavnoj skripti.

Funkciju za registraciju možemo vidjeti na slici 19, funkcija prima samo email i lozinku. Lozinku šifriramo u slučaju da netko pristupi podacima o korisniku, on svakako ne može ništa s njegovom lozinkom. Nakon što smo to napravili, spremamo u bazu podataka. Sve to stavljamo u “try catch” kako bi dobili i vidjeli grešku, i koja greška se dogodila.

```
export default {
    async registerUser(userData) {

        const doc = {
            email: userData.email,
            pass: await bcrypt.hash(userData.pass, 8)
        }

        try {
            return await User.create(doc)
        } catch (e) {
            throw new Error(e)
        }
    },
```

Slika 19. Registracija

Funkcija “authenticateUser” se koristi kako bi se korisnik prijavio u svoj nalog, prikazana je na slici 20. Prvo provjeravamo ako je funkcija uopće dobila korisnika, i da li taj korisnik ima šifru, te na kraju uspoređujemo lozinku u bazi podataka, s lozinkom koja je poslana u funkciju. Ako je sve to ispunjeno, uzimamo id korisnika, te uz pomoć JWT biblioteke pravimo šifrirani token s algoritmom H512 koji traje 1 dan. Pravimo token koji će korisnik moći koristiti za pristupanje aplikaciji kako se ne bi trebao prijaviti svaki put kada osvježi stranicu ili ugasi i upali preglednik. Također preko tokena možemo potvrditi je li to stvarno taj korisnik.

```
async authenticateUser(user, pass) {
    try {

        if (user && user.pass && (await bcrypt.compare(pass, user.pass))) {

            const payload = {
                _id: user._id,
            }

            const token = jwt.sign(payload, String(process.env.JWT_SECRET), {
                algorithm: "HS512",
                expiresIn: "1 day"
            })
            return {
                token,
                email: user.email
            }

        } else {
            throw new Error("Neispravni podaci")
        }
    } catch (e) {
        throw new Error(e)
    }
},
```

Slika 20. Izrada tokena

Na slici 21 vidimo funkciju za potvrdu tokena. Gleda se autorizacija koja je prisutna u headeru HTTP zahtjeva, te se dijeli na tip i token. Ako tip nije “Bearer”, šalje se greška 401, što znači da zahtjev nema prikladnu autentifikaciju za pristup. Ako je tip “Bearer”, provjeravamo ako je token ispravan i nije istekao, te spremamo rezultat provjere. Pošto je ovo middleware funkcija, vraćamo next() što znači da će se izvoditi sljedeća funkcija.

```
async verify(req, res, next) {
    try {
        if (req.headers.authorization) {

            const auth = req.headers.authorization.split(" ")
            const type = auth[0]
            const token = auth[1]

            if (type !== "Bearer") {
                return res.status(401).send()
            } else {
                req.jwt = jwt.verify(token, String(process.env.JWT_SECRET));
                return next();
            }

        } else {
            return res.status(401).send()
        }
    } catch (e) {
        return res.status(403).send(e.message);
    }
},
```

Slika 21. Potvrda tokena

Zadnja funkcija vezana uz autorizaciju prikazana na slici 22 analizira header, i dekodira token kako bi dobili nazad id korisnika.

```
async userInfo(req) {
    try {
        const header = req.headers.authorization
        const token = header.split(" ")
        const decoded = jwt.decode(token[1])
        return decoded
    } catch (e) {
        throw new Error(e)
    }
}
```

Slika 22. Dekodiranje tokena

5.5 Glavna skripta

Glavna skripta prima zahtjeve s frontenda, te šalje te zahtjeve na rutu gdje se dalje izvode naredbe. Na slici 23 možemo vidjeti sve što smo morali "importati" kako bi radile sve funkcije, te konstante koje koristimo za spajanje backenda na port. Prvo definiramo app kao funkciju express, te definiramo port. Također određujemo da aplikacija koristi cors kako bi izbjegli probleme koje se događaju kada su backend i frontend na istoj adresi. Definiramo da će aplikacija HTTP zahtjeve formatirati i čitati u obliku JSON objekta. Nakon što smo to odredili, pozivamo funkciju connect kako bi pokrenuli našu bazu podataka. Na kraju dokumenta prikazano na slici 24 prizivamo funkciju express, kojom pokrećemo express server, te određujemo na kojem portu će primati zahtjeve.

```
/* Importovi */
import express from "express"
import cors from "cors"
import Chat from "../Models/Chat.js"
import Ad from "../Models/Ad.js"
import User from "../Models/User.js"
import auth from "./auth.js"
import {
    Server
}
from "socket.io"
import connect from '../initDB.js';

/* Konstante za spajanje */
const app = express()
const port = 3000

app.use(cors())
app.use(express.json())

await connect()
```

Slika 23. Početak glavnog dokumenta

```
const httpServer = app.listen(port, () => {
    console.log(`Server listening on port ${port}`
});
```

Slika 24. Pokretanje servera

5.6 Socket.io

Socket.io je biblioteka koja pojednostavljuje stvaranje komunikacije između klijenta i servera, omogućava komunikaciju u pravo vrijeme uz pomoć socketa. Glavna razlika između ruta i socketa je to što nakon što je stvorena neprekidna veza između klijenta i servera ona je održana dok klijenta ili server odluči zaustaviti vezu, dok za rute nema neprekidna veza već nakon što je poslan zahtjev i odgovor je primljen veza se završava. Uz pomoć ovoga možemo ostvariti dopisivanje između korisnika koji neće morati slati zahtjeve konstantno za provjeru ako su primili poruku, već će čekati događaj na serveru, i server će čekati na događaje s klijenta.

Na kraju glavnog dokumenta, prikazano na slici 25 definiramo server preko kojeg ćemo komunicirati s klijentom kao express server koji smo pokrenuli. Zatim definiramo novi server koji će koristiti socket io, te definiramo odakle će dobivati zahtjeve, u ovom slučaju je to link na frontend.

Nakon toga stvaramo novu listu korisnika koji su trenutačno spojeni. Kada dobijemo zahtjev za spajanje na socket, u konzolu od servera zapisujemo da se spojio korisnik, te spremamo korisnika u varijablu svih spojenih korisnika, definiramo po id-u korisnika i id-u socketa koje korisnik koristi. Zatim ako klijent napravi događaj za slanje poruke, u njemu šalje podatke o poruci, među kojima je id korisnika kojem se šalje poruka. Tražimo socket vezu koja ima u sebi id korisnika kojem je poslana poruka, ako je veza uspostavljena, šaljemo događaj poslane poruke na taj klijent zajedno s podacima o poruci. Na kraju ako dobijemo događaj da se korisnik odspojio, odspajamo tu vezu, te ispisujemo da se korisnik odspojio u konzolu.

```
const httpServer = app.listen(port, () => {
    console.log(`Server listening on port ${port}`)
});

const io = new Server(httpServer, {
    cors: {
        origin: "http://localhost:8080",
        credentials: true,
    },
});

global.onlineUsers = new Map();

io.on("connection", (socket) => {
    console.log('Korisnik spojen');
    socket.on("add-user", (userId) => {
        global.onlineUsers.set(userId, socket.id);
    });

    socket.on("send-msg", (data) => {
        const sendUserSocket = global.onlineUsers.get(data.to);
        if (sendUserSocket) {
            io.to(sendUserSocket).emit("msg-recieve", data);
        }
    });

    socket.on('disconnect', () => {
        socket.disconnect();
        console.log('Korisnik odspojen');
    });
});
```

Slika 25. Socket.io povezivanje

5.7 Rute

Rute su određene naredbe koje aplikacija odrađuje kada dobijemo zahtjev na toj specifičnoj ruti, backend dobiva zahtjev i usmjeruje ga prema točnoj ruti. Backend prima URI i metodu zahtjeva, POST, GET, PATCH ili DELETE. U ovoj aplikaciji su korišteni principi REST-a. REST je arhitekturalni stil za software, koji omogućuje veliku fleksibilnost, sigurnost, interoperabilnost i skalabilnost.

5.7.1 Oglasi

Slika 26 prikazuje rutu za dobivanje oglasa. Prvo punimo jedan objekt sa svim podacima koje smo dobili iz zahtjeva s frontenda, pošto je GET zahtjev, koristimo podatke iz upita. To su podaci s kojima korisnik ograničuje koje sve oglase želi vidjeti. Ako korisnik nije dao neko ograničenje, podatak će biti postavljen kao null, jer bi u suprotnom bio nedefiniran. Nakon što smo ispunili objekt, započinjemo upit na kolekciji oglasa, te pravimo upit po ranije definiranom objektu. Provjeravamo za svaki atribut u objektu ako postoji, odnosno da li je nešto osim null, ujedno i ako je nula ili false, ako je nešto osim null, dodajemo to ograničenje u naš upit. Ovo radimo kako bi imali što kraći upit, zbog toga što duljina upita dodaje kompleksnost, i time se dulje izvršava upit. Za lokaciju koristimo regularni izraz, to je niz znakova koji opisuje druge nizove znakova. To nam omogućava pretragu koja će ignorirati mala i velika slova, i možemo imati samo dio naziva lokacije, npr. umjesto da moramo specifično tražiti “Pula” da bi pronašli sve za lokaciju “Pula”, možemo upisati i “pu” i dobit ćemo sve rezultate koji imaju lokaciju “Pula”. Za brojeve kao što su cijena, površina, i broj katova koristimo funkcije gte i lte. Gte određuje da mora biti veće od tog broja, dok lte određuje da mora biti manje od tog broja. Za filtriranje po dopuštenjima u oglasu kao što su pušenje, parking, lift, kućni ljubimci, namještaj i dostupnost tijekom sezone, oni će se samo filtrirati ako je korisnik naznačio da mu je bitno da ima to ograničenje, jer korisnik kojem oglas s parkingom ne mijenja odluku, neće značiti ništa ako oglas ima ili nema parking. Nakon što smo ispunili upit, pokrećemo ga izvršavati na kolekciji oglasa, stavljamo await prije kako bi pričekali da se izvrši upit do kraja prije nego što šaljemo na frontend. Također, upiti su poredani tako da su na početku najnoviji oglasi, dok su na kraju najstariji. Ako zahtjev nije poslao nikakva ograničenja u upitu, bit će prikazani svi oglasi, također poredani od najnovijeg do najstarijeg. Na kraju šaljemo sve dokumente u obliku JSON objekta koji su nađeni u upitu i šaljemo ih nazad u odgovoru na zahtjev.

```
app.get("/ads", async (req, res) => {
    try {
        const filters = {
            location: new RegExp(String(req.query.location), "i"),
            price: {
                min: req.query.minPrice || null,
                max: req.query.maxPrice || null,
            },
            rooms: {
                min: req.query.minRooms || null,
                max: req.query.maxRooms || null,
            },
            surface: {
                min: req.query.minSurface || null,
                max: req.query.maxSurface || null,
            },
            parking: req.query.parking || null,
            pets: req.query.pets || null,
            smoking: req.query.smoking || null,
            season: req.query.season || null,
            furnished: req.query.furnished || null,
            floors: {
                min: req.query.minFloors || null,
                max: req.query.maxFloors || null,
            },
            lift: req.query.lift || null,
        }

        /* query za filtriranje, ulančani ifovi kako se nepotrebno ne bi izvodio query za nešto što korisniku nije bitno */
        const query = Ad.find()

        if (req.query.location) query.where("location").regex(filters.location);
        if (filters.price.min) query.where("price").gte(Number(filters.price.min));
        if (filters.price.max) query.where("price").lte(Number(filters.price.max))
        if (filters.surface.min) query.where("surface").gte(Number(filters.surface.min))
        if (filters.surface.max) query.where("surface").lte(Number(filters.surface.max))
        if (filters.rooms.min) query.where("rooms").gte(Number(filters.rooms.min))
        if (filters.rooms.max) query.where("rooms").lte(Number(filters.rooms.max))
        if (filters.parking) query.where("parking").equals(true)
        if (filters.pets) query.where("pets").equals(true)
        if (filters.smoking) query.where("smoking").equals(true)
        if (filters.season) query.where("season").equals(true)
        if (filters.furnished) query.where("furnished").equals(true)
        if (filters.floors.min) query.where("floors").gte(Number(filters.floors.min))
        if (filters.floors.max) query.where("floors").lte(Number(filters.floors.max))
        if (filters.lift) query.where("lift").equals(true)

        const ads = await query.sort({
            updatedAt: -1
        }).populate("createdBy").select("-pass").exec();
        res.json(ads)
    } catch (e) {
        console.error(e.message)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 26. Dobivanje i filtriranje svih oglasa.

Na slici 27 prikazana je ruta za dobivanje informacija o specifičnom oglasu. S frontenda preko URL-a šaljemo parametre u zahtjevu. U parametrima šaljemo id oglasa koji nas zanima, nakon što

smo dobili zahtjev, spremamo taj id u konstantu, te preko tog id-a tražimo specifičan oglas u bazi podataka. Nakon što je nađen na njemu je izvršena funkcija populate po polju "createdBy", ta funkcija traži korisnika koji ima id isti kao u tom polju, te dobivamo tako korisne informacije o korisniku koji je napravio oglas zajedno s oglasom, kao što su korisnikov email i lista njegovih razgovora, kako ne bi odvojeno morali i tražiti korisnika nakon ovoga, i izbacujemo lozinku korisnika kako je ne bi slali drugima. Također koristimo select kako se ne bi prikazivala lozinka korisnika i slala na frontend. Na kraju šaljemo odgovor na zahtjev, koji sadrži JSON objekt tog oglasa kojeg je zahtjev tražio. Možemo vidjeti kako je u shemi za kolekciju oglasa određeno da se to polje odnosi na kolekciju korisnika na slici 28.

```
/* GET za specifični oglas */
app.get("/ads/:id", async (req, res) => {
    try {
        const id = req.params.id
        const ad = await Ad.findById(id).populate("createdBy").select("-pass");
        res.json(ad)
    } catch (e) {
        console.error(e)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 27. Dobivanje specifičnog oglasa

```
createdBy: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "User",
    required: true
},
```

Slika 28. Prikaz polja "createdBy"

Na slici 29 je prikazna ruta za dodavanje novog oglasa. Pošto dodajemo novi dokument u kolekciju koristimo POST metodu i šaljemo s frontenda objekt na backend. Prije nego što se izvrši ruta, izvršavamo funkciju iz autorizacije verify kako bi potvrdili da korisnik ima valjan token, pošto smo odredili da korisnik mora imati korisnički nalog kako bi mogao dodavati oglase, ovo vrijede i za sve druge funkcionalnosti koje su određene da samo smiju korisnici s korisničkim nalogom koristiti.

Ako je sve u redu s tokenom, kreću se izvršavati naredbe. Prvo dobivamo cijeli dokument korisnika iz baze podataka uz pomoć tokena. Kada dobijemo nazad korisnika, spremamo ga u polje objekta poslano s frontenda koji sadrži sve informacije o oglasu, nakon toga spremamo cijeli objekt u konstantu. Naposljetku se pravi novi dokument uz pomoć funkcije create, pošto je u shemi definirano da će polja "createdAt" i "updatedAt" biti trenutačni datum i vrijeme što možemo vidjeti na slici 30, korisnik ne treba slati ove podatke, već su oni automatski izračunati. Nakon što se dokument uspješno spremi, šaljemo odgovor, JSON objekt spremljenog oglasa.

```
/* POST za oglas */
app.post("/upload", [auth.verify], async (req, res) => {
    try {
        const user = await auth.userInfo(req)
        req.body.createdBy = user
        const body = req.body
        const ads = await Ad.create(
            body
        )
        res.json(ads)
    } catch (e) {
        console.error(e)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 29. Dodavanje novog oglasa

```
    createdAt: {
        type: Date,
        immutable: true,
        default: () => Date.now(),
    },
    updatedAt: {
        type: Date,
        default: () => Date.now(),
    }
```

Slika 30. Polja "createdAt" i "updatedAt"

Slika 31 prikazuje rutu za ažuriranje oglasa. Pošto mijenjamo postojeći dokument, koristimo PATCH metodu u zahtjevu. Prvo gledamo ako korisnik ima valjan token te tek onda izvršavamo naredbe na ruti. Iz parametra u URL-u dobivamo id, a u tijelu zahtjeva šaljemo objekt s podacima o

ažuriranom oglasu. Spremamo id u konstantu id, a tijelo zahtjeva u konstanti body, te zatim pokrećemo funkciju uz pomoć koje nađemo dokument oglasa po id-u oglasa, te ga ažuriramo s novim podacima iz tijela zahtjeva. Nakon što se to izvrši pokrećemo funkciju save kako bi ažurirali polje "updatedAt", koristimo save jer će se sve validacije iz sheme, te će se polje "updatedAt" staviti na trenutačno vrijeme i datum, zbog čega ponovno u zahtjevu nije potrebno da je to polje prisutno, a polje "createdAt" je definirano da se ne može mijenjati nakon što je spremljeno. Nakon što se spremi oglas, šaljemo odgovor, JSON objekt spremljenog oglasa.

```
/* PATCH za oglas */
app.patch("/upload/:id", [auth.verify], async (req, res) => {
    try {
        const id = req.params.id
        const body = req.body

        const ads = await Ad.findByIdAndUpdate(id, body)
        ads.save()

        res.json(ads)
    } catch (e) {
        console.error(e)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 31. Ažuriranje oglasa

Za brisanje oglasa se koristi ruta prikazana na slici 32, s tim da se prvo gleda ako zahtjev ima valjan token. U zahtjevu primamo id oglasa koji se briše preko parametara u zaglavlju zahtjeva, taj id oglasa se sprema u konstantu id. Nakon toga preko tog id-a tražimo dokument koji ima taj id u kolekciji oglasa, te brišemo cijeli dokument. Nakon što to uspješno prođe, šaljemo samo odgovor nazad koji govori da je oglas obrisan.

```
/* DELETE za oglas */
app.delete("/upload/:id", [auth.verify], async (req, res) => {
    try {
        const id = req.params.id

        await Ad.findByIdAndDelete(id)

        res.json(`Oglas obrisan`)
    } catch (e) {
        console.error(e)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 32. Brisanje oglasa

5.7.2 Poruke

Za slanje poruka koristimo rutu prikazanu na slici 33, prvo provjeravamo token, zatim izvršavamo rutu. Uzimamo informacije o korisniku iz tokena i spremamo u konstantu user, tijelo zahtjeva spremamo u body koje sadrži tekst poruke i id korisnika koji šalje i koji prima poruku. Zatim se spremaju podaci u kolekciju poruka, tekst poruke, i lista korisnika između kojih je korisnika je dobivena iz tijela zahtjeva, dok za korisnika koji šalje informacije dobivamo iz tokena. Nakon što je spremljeno spremamo dokument u konstantu data, ako je uspješno spremljeno vraćamo u zahtjevu true, inače se tijelo if-a neće izvesti i vratiti će false.

```
/* POST za poruke */
app.post("/messages", [auth.verify], async (req, res) => {
    try {
        const user = await auth.userInfo(req)
        const body = req.body
        const data = await Chat.create({
            message: {
                text: body.message
            },
            users: [body.from, body.to],

            sender: user._id,
        });
        if (data) return res.json(true)
        return res.json(false)
    } catch (e) {
        console.error(e)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 33. Slanje poruke

Kako bi dobili poruke, koristimo rutu prikazanu na slici 34, pregledavamo token, ako je u redu izvršava se ruta. Iz upita iz zaglavlja zahtjeva dobivamo potrebne podatke kako bi našli sve poruke poslane između dva korisnika, te podatke spremamo u konstantu data. Spremamo sve poruke u konstantu messages kroz upit pokrenut na kolekciju poruka, gdje tražimo sve poruke koje sadrže dva id-a korisnika koji su poslani u upitu. Nije bitan redoslijed kojim šaljemo korisnike u upit, jer smo specificirali da želimo sve koje sadrže ta dva korisnika u polju users. Nakon što upit pronađe sve poruke, postavlja ih u redoslijed gdje su pri dnu najnovije poruke kako bi bile sortirane kad dođu na frontend. Nakon toga koristimo map funkciju kojom mijenjamo svaku poruku, ovo radimo kako bi dobili tip poruke, ako je id korisnika koji je pošiljatelj poruke, isti onog od pošiljatelja poruke iz liste korisnika, dobivamo true, što znači da je korisnik poslao ovu poruku, u suprotnom je

tip poruke false. Ovo koristimo na frontendu kako bi lakše mogli postaviti klase i raspoznavati koje su naše poruke, a koje su tuđe. Na kraju samo šaljemo odgovor s objektom svih mapiranih poruka.

```
/* GET za poruke */
app.get("/messages", [auth.verify], async (req, res) => {
    try {
        const data = req.query
        const messages = await Chat.find({
                users: {
                    $all: [data.from, data.to]
                }
            })
            .sort({
                updatedAt: 1
            })

        const mappedMessages = messages.map((item) => {
            return {
                id: item._id,
                type: item.sender.toString() === data.from,

                message: item.message.text,
                createdAt: item.createdAt
            }
        })
        return res.json(mappedMessages)
    } catch (e) {
        console.error(e)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 34. Dobivanje poruka.

Kako bi dobili sve potrebne informacije o korisnicima s kojima korisnik koji šalje zahtjev imaju razgovore, koristimo rutu prikazanu na slici 35, nakon što potvrdimo da je token valjan ruta se kreće izvršavati. Preko tokena dobivamo id od korisnika, te preko tog id-a tražimo korisnika u bazi podataka koji je zatražio zahtjev za dobivanje informacija o razgovorima koje ima. Koristimo

populate kako bi spojili dokumente korisnika preko id-eva koje su u listi razgovora. Time dobivamo informacije o korisnicima s kojima korisnik ima razgovore, specifično njihov email, stoga mičemo lozinku iz upita kako korisnik ne bi imao pristup njihovim lozinkama. Nakon što se izvrši taj upit, šalje se lista razgovora u JSON objektu kao odgovor na zahtjev.

```
/* GET za listu korisnika s kojim se dopisujemo */
app.get("/conversations", [auth.verify], async (req, res) => {
    try {
        const user = await auth.userInfo(req)
        const userConversations = await User.findById(user).populate("conversations").select("-pass")
        res.json(userConversations)
    } catch (e) {
        console.error(e)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 35. Dobivanje informacija o razgovorima

Za dodavanje novog razgovora između korisnika koristimo rutu na slici 36. Ako je token valjan izvršava se ruta, prvo se određuje id korisnika uz pomoć tokena iz zahtjeva, te preko tog id-a dobivamo sve razgovore od tog korisnika i informacije o korisnicima s kojima ima razgovore. Nakon toga spremamo id polje iz tijela zahtjeva u konstantu. Na kraju uz pomoć if-a gledamo ako korisnik već ima razgovor s tim korisnikom, koristimo funkciju some koja prolazi kroz svako polje liste i izvodi funkciju na svakom polju. Vraća true ako je jednom dobiven true u funkciji koja se izvodi, u ovom slučaju gledamo ako ima id razgovora koji je jednak novom razgovoru koji je poslan u zahtjevu. Ako ima već razgovor s tim korisnikom, šalje se odgovor true. U suprotnom slučaju, u polje razgovora trenutačnog korisnika se dodaje id novog razgovora, te preko tog id-a tražimo novog korisnika s kojim korisnik započinje razgovor. Nakon što dobijemo sve njegove razgovore, dodajemo novi razgovor tako da dodajemo u listu razgovora id korisnika koji šalje zahtjev. Na kraju koristimo save za oba dokumenta kako bi se ažurirali u bazi podataka, te šaljemo odgovor true nakon što se sve izvede.

```
/* PATCH za dodavanje novog korisnika za dopisivanje */
app.patch("/conversations", [auth.verify], async (req, res) => {
    try {
        const user = await auth.userInfo(req)
        const userConversations = await User.findById(user).populate("conversations").select("-pass")

        const newConversationId = req.body._id

        if (userConversations.conversations.some((conversation) => conversation._id.toString() === newConversationId)) {
            res.json(true)
        } else {
            userConversations.conversations.push(newConversationId)
            const newUser = await User.findById(newConversationId).populate("conversations").select("-pass")
            newUser.conversations.push(user)
            await userConversations.save()
            await newUser.save()
            res.json(true)
        }
    } catch (e) {
        console.error(e)
        res.status(500).json({
            error: "Internal Server Error"
        });
    }
})
```

Slika 36. Dodavanje novog razgovora

5.7.3 Korisnici

Za prijavljivanje koristimo rutu prikazanu na slici 37, iako ne stvaramo novi dokument, koristimo POST metodu jer pokrećemo neki proces na serveru, ali šaljemo podatke na server, pa ne možemo koristiti GET. U bazi podataka tražimo korisnika koji ima email kao onaj iz objekta u tijelu zahtjeva, zatim provjeravamo ako je korisnik pronađen. Ako nije dobit ćemo grešku da korisnik nije pronađen, ako ima korisnik nastavlja se izvoditi ruta. Ako je korisnik pronađen, ruta se nastavlja te se šalje korisnik, kao i lozinka poslana u tijelu zahtjeva. Ako je lozinka iz zahtjeva jednaka loznici iz baze podataka, funkcija će vratiti token koji spremamo u konstantu. Nakon što se izvede ta funkcija, u odgovoru šaljemo token koji korisnik može koristiti za korištenje funkcionalnosti aplikacije. Pošto se za odjavljivanje ništa ne mijenja na niti treba izvesti na serveru, nemamo rutu već se sve odvija na frontendu.

```
/* POST za login */
app.post("/login", async (req, res) => {
    try {
        /* Tražimo korisnika */
        const user = await User.findOne({
            email: req.body.email
        });
        /* Provjera u slučaju da ne nađemo korisnika */
        if (!user) {
            throw new Error("Korisnik nije pronađen!")
        }

        const authCheck = await auth.authenticateUser(user, req.body.pass) /* Provjerava ako je lozinka točna, dobivamo nazad token */
        res.json(authCheck)
    } catch (e) {
        console.error(e.message)
        res.status(500).send({
            error: e.message
        })
    }
})
```

Slika 37. Prijavljivanje

Kako bi se gost napravio svoj korisnički račun koristimo rutu na slici 38, također kao za prijavljivanje, koristimo POST metodu. Objekt iz tijela zahtjeva spremamo u konstantu, koji sadrži email i lozinku za novi korisnički račun. Šaljemo tu konstantu u funkciju za registraciju korisnika te čekamo da se stvori kriptirana lozinka i spreme podaci u bazu podataka. Nakon što se to izvrši šaljemo odgovor na zahtjev u obliku dokumenta novog korisnika koji je upravo spremljen.

```
/* POST za signup */
app.post("/signup", async (req, res) => {
    try {
        const newUser = req.body
        const savedDoc = await auth.registerUser(newUser)
        res.json(savedDoc)
    } catch (e) {
        console.error(e.message)
        res.status(500).send({
            error: e.message
        })
    }
})
```

Slika 38. Izrada novog korisničkog računa

Kako bi potvrdili da je korisnik prijavljen s valjanim tokenom, koristimo rutu prikazanu na slici 39. Ruta samo prije nego što se izvede provjerava ako je token valjan, ako je, ruta će se izvesti i dat će odgovor true, ako token ne valja neće se izvesti ruta te će verify funkcija vratiti error.

```
/* GET za provjeru autorizacije */
app.get("/auth", [auth.verify], async (req, res) => {
    try {
        res.json(true)
    } catch (e) {
        console.error(e.message)
        res.status(500).send({
            error: e.message
        })
    }
})
```

Slika 39. Autorizacija

6. Frontend

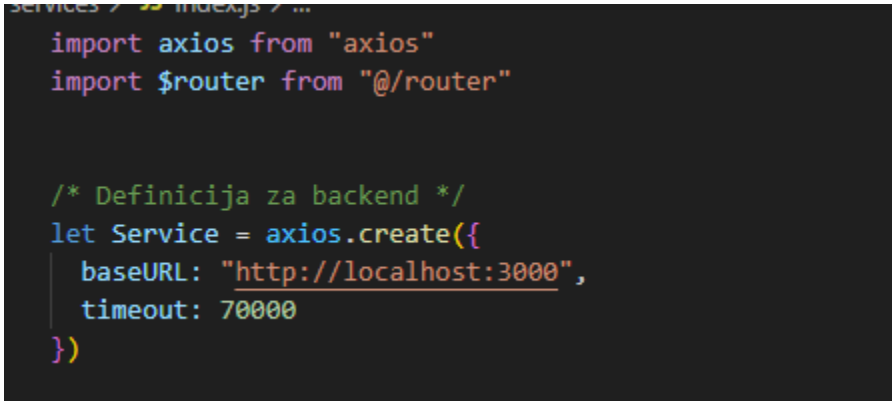
Frontend je termin koji se odnosi na dio aplikacije koji korisnici koriste. Sastoji se od korisničkog sučelja, izgled i dizajn vizualnih elementa koje korisnik koristi. Frontend ima svoj kod koji se koristi kako bi se oblikovao izgled stranice i stvorila komunikacija s korisnikom i aplikacijom preko funkcija. Također ima dio koji je zadužen za komunikaciju sa serverom, slanjem zahtjeva i primanjem odgovora.

6.1 Servisi

Kako bi uspostavili vezu s backendom koristimo dvije biblioteke, Axios i Socket.io. Axios je biblioteka koja se koristi za slanje i primanje zahtjeva sa servera. Dopušta jednostavan način za asinkronu komunikaciju. Na frontendu imamo odvojenu skriptu koja ima funkciju vezanu uz svaku rutu koja nam omogućava komunikaciju s backendom.

6.1.1 Axios

Na slici 40 vidimo definiciju Axios servisa. Prvo uvozimo sam Axios, te router koji koristimo za mijenjanje stranica. Zatim definiramo varijablu Service kao instancu Axiosa, koja se spaja na backend. Određujemo url backenda, te koliko dugo čekamo za odgovor s backenda u milisekundama.



Slika 40. Definicija Axios servisa

Slika 41 prikazuje funkcije koje se koriste kada šaljemo zahtjev na rutu koja zahtjeva token u zaglavlju zahtjeva. Prva funkcija provjerava da li zaglavlje zahtjeva token u headeru, ako zahtjeva, uzimamo token i spremamo uz pomoć funkcije prikazane na slici 42 koja samo uzima token koji je spremljen u pregledniku korisnika. Ako dobijemo token nazad, stavljamo ga u zaglavlje headera i definiramo ga kao bearer token. Ako je token prazan, dobivamo grešku da nedostaje token. Ako dobijemo grešku kao odgovor s backenda vezan uz token, provjeravamo ako je 401 ili 403 error. Ako je 401 znači da token fali li nije ispravan, te pokrećemo logout funkciju prikazanu na slici 43 koja samo briše token koji je spremljen lokalno u korisnikovom pregledniku i naređujemo routeru da ponovno ide na stranicu, koja će odraditi logiku u ruteru koja šalje korisnika na stranicu za prijavu ako je pokušavao ići na stranicu za koju je potrebno biti prijavljen. 403 error označuje da je token istekao, i odvijaju se iste naredbe kao i da je 401 error samo korisnik dobiva drugačiju poruku.

```
/* Token koji se šalje na backend */
Service.interceptors.request.use((request) => {
  try {
    if (request.headers.authRequired) /* Provjera jel uopće potrebno autorizirati prije nego što se šalje req */ {
      const token = users.getToken()
      if (token) {
        request.headers['Authorization'] = 'Bearer ' + token;
      } else {
        throw new Error("Nedostaje token"); /* Ukoliko fali token prekidamo sa requestom na backend */
      }
    }
  } catch (e) {
    console.error(e);
  }
  return request;
});

/* Ukoliko dobijemo error vezano uz token */
Service.interceptors.response.use((response) => response, async (err) => {
  console.log(err.response.status)
  /* Ukoliko ne valja token */
  if (err.response.status == 401) {
    await Alert("Neispravna prijava, molimo prijavite se opet")
    users.logoutUser();
    $router.go()
    return
  }
  /* Ukoliko je istekao token  */
  if (err.response.status == 403) {
    await Alert("Vaša prijava je istekla, molimo prijavite se opet")
    users.logoutUser();
    $router.go()
    return
  }
})
```

Slika 41. Funkcije za token

```
/* Za dobivanje tokena i emaila radi preglednosti */
getUser() {
  return localStorage.getItem("user")
},
/* Koristimo kada šaljemo header */
getToken() {
  const user = JSON.parse(this.getUser())
  if (user && user.token) {
    return user.token
  } else {
    return false
  }
},
```

Slika 42. Funkcija "getToken" i "getUser"

```
/* Logout za usera, patch pošto mijenjamo status korisnika */
logoutUser() {
  try {
    localStorage.removeItem("user")
    return
  } catch (e) {
    console.error(e.message)
    return false
  }
},
```

Slika 43. Funkcija za odjavu korisnika

Ostatak dokumenta se samo bavi sa slanjem zahtjeva na backend i primanjem odgovora s ruta koje su već prijašnje objašnjene u završnom radu, stoga ćemo prikazati samo par primjera. Na slici 44 možemo vidjeti 4 funkcije. Prva funkcija “getAds” prima parametar terms u sebi, te kreće izvršavati naredbe. Svi zahtjevi su unutar “try catch” kako bi mogli vidjeti greške ako dođe do njih tijekom izvršavanja zahtjeva. Kako se frontend treba spojiti na backend, backend treba odraditi naredbe unutar rute, te tek onda šalje odgovor na zahtjev, funkcije moraju biti asinkrone, što znači da kada je ispred funkcije await, neće se sljedeće naredbe izvršavati dok ne dobijemo nazad neki odgovor ili istekne vrijeme koje smo naznačili za čekanje odgovora. Šaljemo GET zahtjev na rutu “/ads” s parametrima iz funkcije, te spremamo odgovor tog zahtjeva u konstantu response, te šaljemo podatke iz odgovora nazad iz funkcije. Sljedeća funkcija “getAdsDetail” šalje id specifičnog oglasa koji želimo dobiti i šalje ga na backend, nakon što dobijemo odgovor na zahtjev, funkcija ga šalje nazad. Zadnja funkcija šalje POST zahtjev na backend, u tijelo zahtjeva šaljemo parametar body koji u sebi sadrži sve informacije potrebne za stvaranje novog oglasa u bazi podataka, te označujemo da je potreban valjan token u zaglavlju zahtjeva kako bi dobili odgovor nazad, stoga smo stavili da je “authRequired” true.

```
async getAds(terms) {
  try {
    const response = await Service.get("/ads", {
      params: terms
    })
    return response.data
  } catch (e) {
    console.error(e.message)
    return false
  }
},
/* Za dobivanje specifičnog oglasa */
async getAdsDetail(id) {
  try {
    const response = await Service.get(`/ads/${id}`)
    return response.data
  } catch (e) {
    console.error(e.message)
    return false
  }
},
/* Za dodavanje oglasa */
async postAds(body) {
  try {
    console.log("postAds")
    const response = await Service.post(`/upload`, body, {
      headers: {
        authRequired: "true" /* Pokazuje da je potrebna autorizacija za ovaj request */
      }
    });
    return response.data
  } catch (e) {
    console.error(e.message)
    $router.go()
    return false
  }
},
```

Slika 44. Funkcije za slanje i primanje zahtjeva

6.1.2 Socket.io

Kako bi mogli imati dopisivanje s korisnicima u pravom vremenu potrebno je definirati socket servis na frontendu. Na slici 45 možemo vidjeti definiciju tog servisa. Prvo uvozimo io iz socket biblioteke i eventBus koji je definiran u main.js datoteci. Io je objekt koji nam omogućava spajanje i komunikaciju preko socketa s backendom, dok nam eventBus omogućava komunikaciju

između našeg servisa i ostatka frontenda tako da druge komponente čekaju za događaje u event busu umjesto da pozivamo event bus. Prvo definiramo sam socket servis te zatim definiramo tri funkcije. Prva funkcija “setupSocketConnection” postavlja vezu s backendom, te šalje događaj na backend da se doda korisnika. Zatim čeka na događaj gdje prima poruku, kada se to dogodi šalje događaj u eventBus s podacima iz te poruke koje je servis dobio s backenda. “SendMessage” funkcija šalje događaj slanja poruke zajedno s podacima poruke na backend, ta zadnja funkcija disconnect samo provjerava ako postoji veza, ako postoji veza, ona se prekida.

```
import {
    io
}
from 'socket.io-client';
import {
    eventBus
} from "../main";


class SocketioService {
    socket;
    constructor() {}


    setupSocketConnection(user) {
        this.socket = io("http://localhost:3000");

        this.socket.emit("add-user", user)

        this.socket.on('msg-recieve', (data) => {
            eventBus.$emit('message-received', data);
        });
    }

    sendMessage(data) {
        this.socket.emit("send-msg", data)
    }

    disconnect() {
        if (this.socket) {
            this.socket.disconnect();
        }
    }


}

export default new SocketioService();
```

Slika 45. Socket.io servis

6.2 Ruter

Vue.js ima svoj ruter, koji nije povezan s ruterom na backendu. Koristi se za navigaciju među stranicama, Vue.js ustvari ima samo jednu stranicu, i elementi na stranici se dinamički mijenjanju. Same stranice, odnosno pogledi u Vue.js-u su komponente umjesto zasebne stranice. Prednost korištenja komponenti kao stranica je to što omogućava modularnost što znači da možemo koristiti iste dijelove na više stranica. Također puno se brže očitavaju stranice pošto se ne grade ispočetka, već se samo miču stari elementi i dodaju novi. Na slici 46 možemo vidjeti definiciju svih naših pogleda. Path je url tog pogleda, dok je name ime samog pogleda i nije bitno da su isti. Zatim definiramo koja je komponenta ustvari taj pogled. Na kraju neke stranice imaju meta, to su podatci koji smo stavili koji nam olakšava definicije nekih funkcija. Slika 47 prikazuje funkciju koja se poziva svaki put prije nego što se mijenja stranica. U parametrima sadrži informacije o stranici na koju se prebacujemo, te s koje stranice dolazimo, parametar next označuje na koju stranicu će nas router poslati. Prvo gledamo ako stranica na koju korisnik pokušava ići uopće postoji, !! pretvara vrijednost u boolean, te onda treći uskličnik negira. Ako je stranica na koju idemo nedefinirana dobit ćemo false vrijednost, i kada negiramo dobit ćemo true što je potrebno da se izvrši tijelo if-a, koje će poslati korisnika na glavnu stranicu. Ako stranica postoji, spremamo vrijednost “needsUser” od stranice na koju pokušavamo ići koja govori ako korisnik treba biti prijavljen kako bi pristupio stranici, ako nema postavljenu vrijednost sprema se kao false. Nakon toga tražimo token korisnika, što znači da je korisnik prijavljen. Nakon što su spremljene te konstante, funkcija gleda ako korisnik nije prijavljen, a stranica na koju pokušava ići zahtjeva da korisnik bude prijavljen, izvršava se tijelo if-a, i šalje ga na stranicu za prijavu. Ako je korisnik prijavljen, funkcija gleda ako korisnik pokušava ići na stranicu za prijavu ili dolazi sa stranice za prijavu. U oba slučaja šaljemo korisnika na glavnu stranicu s oglasima, ako je već prijavljen ne želimo mu dopustiti da se opet prijavi, a ako dolazi s login stranice a prijavljen je znači da se tek prijavio. Funkcija zadnje provjerava ako je korisnik prijavljen, i stranica na koju ide zahtjeva da je korisnik prijavljen. Ako je to slučaj, zovemo funkciju koja će provjeriti valjanost tokena, pošto ovo znači čekanje odgovor od backenda, moramo staviti await, ako token ne valja, funkcija za provjeru tokena će odjaviti korisnika i ponovno pokrenuti router. Ako nijedno od ovih slučajeva nije ispunjeno, šaljemo korisnika na stranicu na koju je htio ići.

```
const routes = [{
    path: '/oglasi',
    name: 'home',
    component: () => import('../views/HomeView.vue')
  }, {
    path: '/oglasi/:id',
    name: 'detail',
    component: () => import('../views/DetailView.vue')
  }, {
    path: '/upload',
    name: 'upload',
    meta: {
      needsUser: true
    },
    component: () => import('../views/UploadView.vue')
  },
  {
    path: '/edit/:id',
    name: 'edit',
    meta: {
      needsUser: true
    },
    component: () => import('../views/EditView.vue')
  },
  {
    path: '/login',
    name: 'login',
    component: () => import('../views/LoginView.vue')
  }, {
    path: '/messages',
    name: 'messages',
    meta: {
      needsUser: true
    },
    component: () => import('../views/MessagesView.vue')
  }
]
```

Slika 46. Definicija ruta

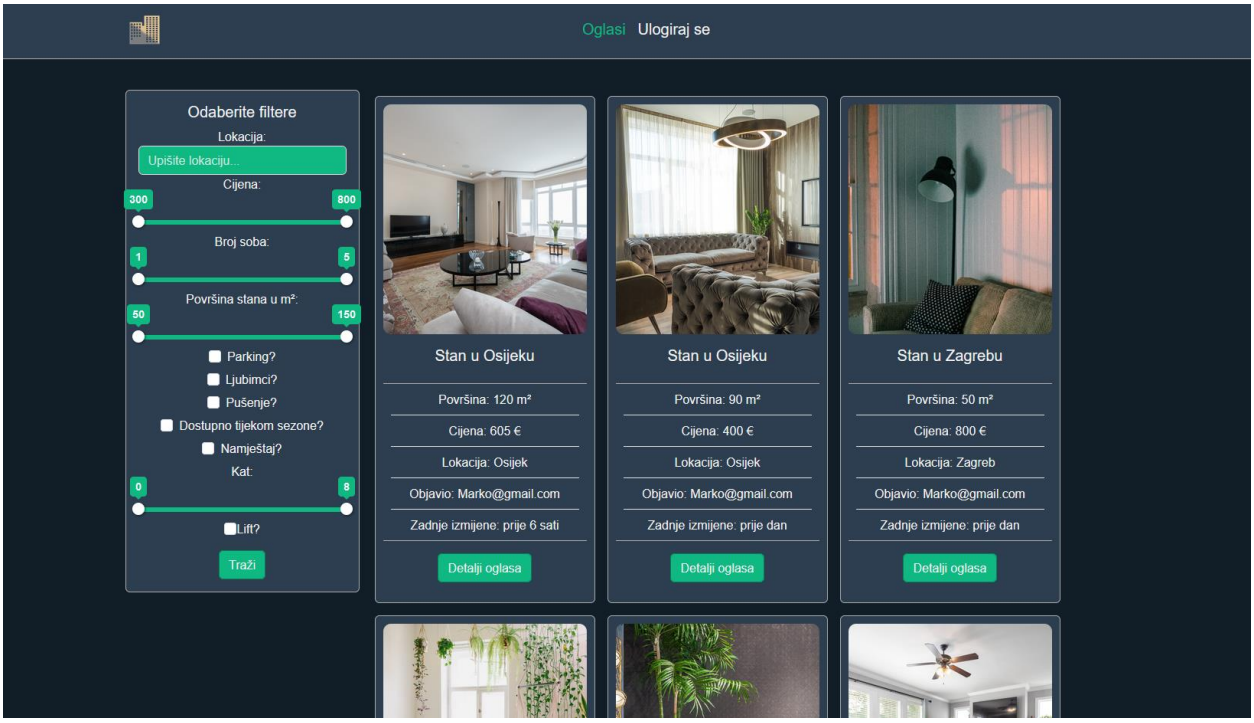
```
/* Provjera prije nego što se ide na slijedeću stranicu */
router.beforeEach(async (to, from, next) => {

  if (!!!to.name) {
    /* Ako korisnik ide na stranicu koja ne postoji, šaljemo ga na home */
    next("/oglasi")
    return
  }

  const needsUser = to.meta.needsUser || false; /* Boolean koji nam govori kao treba korisnik za stranicu na koju pokušavamo ići */
  const user = users.getUser();


  /* If ako nema korisnika */
  if (!user && needsUser) {
    /* Ako ide na stranicu za koju je potreban korisnik, šaljemo ga na login stranicu */
    next("/login")
    return
  }

  /* If ako ima korisnika */
  if (user) {
    if (to.path == "/login" || from.path == "/login") {
      /* Ako ide na stranicu za koju ne treba korisnik ili je prijašnja stranica login šaljemo ga na home*/
      next("/oglasi")
      return
    }
    if (needsUser) {
      await users.getAuth() /* Prije nego što ode na stranicu za koju treba korisnik provjerimo ako je istekao ili ne valja token */
    }
  }

  /* U slučaju da se ne ispune svi ifovi */
  next();
  return
})
```

Slika 47. Funkcija za provjeru prije svake promijene pogleda.

6.3 Početni pogled

Kada korisnik tek otvori web aplikaciju, prvo što vidi je glavni view koji u sebi ima sve oglase i filtar za te oglase, taj pogled je prikazan na slici 48. Sve slike za oglase su preuzete sa stranice pexels [11]. View se sastoji od komponente za filtriranje oglasa, te je svaki oglas zasebna komponenta. Na slici 49 vidimo podatke koji se koriste na ovoj komponenti, kao i funkcija mounted koja se izvodi svaki put kada se očita komponenta. “AdsData” su podaci o svim oglasima koje smo dobili i šaljemo u komponente za oglase, dok je “minMax” najveća i najmanje vrijednost za svaki broj koji je u oglasima, koji šaljemo u komponentu za filtriranje kako bi odredili najveću i najmanju moguću vrijednost cijene, broja soba, površine stana i kata. Prvo definiramo konstantu parametra izvučenih iz URL-a koje se zatim šalju u zahtjev koji te parametre šalje kao ograničenje za oglase koje ćemo dobiti u odgovoru iz backenda. Te na kraju punimo objekt s najmanjim i najvećim vrijednostima uz pomoć funkcije za to. Slanje objekata u komponente možemo vidjeti na slici 60, te

također možemo vidjeti kako se broj oglasa određuje s petljom koja će stvoriti jedan oglas za svako polje u "adsData" objektu. Na slici 61 su prikazane sve funkcije koje se koriste na ovoj. "FindMax" funkcija prolazi kroz svaki objekt u listi koju primi, te samo gleda ona polja u objektu koja smo tražili da gleda. Tu funkciju pozivamo u glavnoj funkciji za traženje najvećih i najmanjih vrijednosti, ona traži sve oglase tako što šalje zahtjev na backend bez ikakvih parametra, tako da će upit biti prazan, i dobit će kao odgovor sve oglase.



Slika 48. Početni pogled

```
<script>
/* Imports */
import sidebar from "@/components/Sidebar.vue";
import ad from "@/components/Ad.vue";
import { ads } from "@/services";

export default {
  data() {
    return {
      adsData: null,
      minMax: null,
    };
  },
  async mounted() {
    const urlParams = new URLSearchParams(window.location.search);
    this.adsData = await ads.getAds(urlParams);
    this.minMax = await this.returnMinMax();
  },
```

Slika 49. Definicija objekta i mounted funkcije na početnom pogledu

```
<template>
  <div class="container">
    <div class="row">
      <!-- Sidebar -->
      <div class="col-3">
        <sidebar :props="minMax" />
      </div>

      <!-- Oglasi -->
      <div class="col-9 row">
        <ad
          class="col-span-4 m-2"
          v-for="ad of this.adsData"
          :key="ad._id"
          :props="ad"
        ></ad>
      </div>
    </div>
  </div>
</template>
```

Slika 50. Komponente u početnom pogledu

```
methods: {
  findMinMax(array, key) {
    return array.reduce(
      (result, obj) => {
        const value = obj[key];
        result.min = Math.min(result.min, value);
        result.max = Math.max(result.max, value);
        return result;
      },
      { min: array[0][key], max: array[0][key] }
    );
  },
  async returnMinMax() {
    const array = await ads.getAds();

    return {
      price: this.findMinMax(array, "price"),
      rooms: this.findMinMax(array, "rooms"),
      surface: this.findMinMax(array, "surface"),
      floors: this.findMinMax(array, "floors"),
    };
  },
},
```

Slika 51. Funkcije početnog pogleda

6.3.1. Komponenta za filtriranje

Komponenta za filtriranje, u sebi ima samo velik broj formi koje pune jedan objekt, koji je zatim stavljen u parametre URL-a. Možemo vidjeti dio forma na slici 52 slideri su dio biblioteke vueform, omogućava nam lako ograničavanje po najvećoj i najmanjoj vrijednosti, najveću i najmanju moguću vrijednost prima iz props predanih iz prikaza. Slika 53 prikazuje kako radi forma za da/ne pitanja unutar filtera. Slika 54 prikazuje kako se opcija za filtriranje po liftu niti ne pokazuje ako je najveći kat koji je izabran 0, odnosno prizemlje, pošto je nemoguće imati kat na prizemlju. Slika 55 prikazuje sve objekte koji se koriste, kao i computed objekti, razlog tome je što su svi objekti lokalno spremljeni, kako bi korisniku ostali svi filteri nakon što osvježi stranicu ili ide na neku drugu stranicu dok koristi aplikaciju. Objekt se sprema svaki put kada se mijenja neka vrijednost u formi. Slika 56 prikazuje funkciju koja se poziva kada korisnik stisne gumb za filtriranje. Prvo se sva ograničenja spremaju u konstantu, te se izbacuju svi objekti iz konstante koji nemaju vrijednost. Na kraju se svi objekti stavljaju u parametre URL-a i ponovno se ide na rutu, kako bi se poslali parametri u zahtjev backendu, i prikazali oglasi prema novom filtriranju.

```
<template>
  <div class="sidebar card position-sticky" style="top: 20px">
    <div class="card-body" v-if="props">
      <h5 class="card-title">Odaberite filtere</h5>
      <!-- Pretraživanje lokacije -->
      <form @submit.prevent="search">
        <!-- Location Input -->
        <div class="form-group">
          <label for="location">Lokacija:</label>
          <input
            type="text"
            class="form-control"
            id="location"
            placeholder="Upišite lokaciju..."
            v-model="location"
          />
        </div>
        <!-- Slider-->
        <div class="form-group">
          <label for="price">Cijena: </label>
          <div class="form-group">
            <slider
              class="slider"
              v-model="price"
              id="price"
              :min="props.price.min"
              :max="props.price.max"
            />
          </div>
        </div>
        <!-- Broj soba -->
        <div class="form-group">
          <label for="rooms">Broj soba: </label>
          <div class="form-group">
            <slider
              class="slider"
              v-model="rooms"
              id="rooms"
              :min="props.rooms.min"
              :max="props.rooms.max"
            />
          </div>
```

Slika 52. Neke od forma korištene u komponenti za filtriranje

```
<!-- Pitanja -->
<div
  class="form-check form-group"
  v-for="(value, preference) in preferences"
  :key="preference"
>
  <label class="form-check-label clickable">
    {{ preference }}
    <input
      class="form-check-input clickable mb-2"
      type="checkbox"
      v-model="preferences[preference]"
    />
  </label>
</div>
```

Slika 53. Forma za da/ne upite

```
<div class="row align-items-center">
  <div v-if="ad.floors > 0" class="col-3 offset-2 text-start">
    <input class="form-check-input" type="checkbox" v-model="ad.lift" />
    <label class="form-check-label">Lift?</label>
  </div>
</div>
```

Slika 54. Logika za prikazivanje opcije lifta

```
export default {
  data() {
    return {
      price: [0, 1000],
      location: null,
      rooms: [1, 5],
      surface: [1, 200],
      preferences: {
        "Parking?": false,
        "Ljubimci?": false,
        "Pušenje?": false,
        "Dostupno tijekom sezone?": false,
        "Namještaj?": false,
      },
      floors: [0, 10],
      lift: false,
    };
  },
  props: {
    props: Object,
  },
  name: "Sidebar",
  computed: {
    sidebarData() {
      return {
        price: this.price,
        location: this.location,
        rooms: this.rooms,
        surface: this.surface,
        preferences: this.preferences,
        floors: this.floors,
        lift: this.lift,
      };
    },
  },
  mounted() {
    const storedData = JSON.parse(localStorage.getItem("sidebarData")) || {};
    Object.assign(this, storedData);
  },
  watch: {
    sidebarData: {
      deep: true,
      handler(newValue) {
        localStorage.setItem("sidebarData", JSON.stringify(newValue));
      },
    },
  },
```

Slika 55. Svi objekti korišteni u filtriranju, kao i spremanje tih filtera lokalno

```
methods: {
  search() {
    /* Parametri koji šaljemo backendu kako bi filtrirao rezultate iz baze podataka */
    const queryParams = {
      location: this.location,
      minPrice: this.price[0],
      maxPrice: this.price[1],
      minRooms: this.rooms[0],
      maxRooms: this.rooms[1],
      minSurface: this.surface[0],
      maxSurface: this.surface[1],
      minFloors: this.floors[0],
      maxFloors: this.floors[1],
      parking: this.preferences["Parking?"],
      pets: this.preferences["Ljubimci?"],
      smoking: this.preferences["Pušenje?"],
      season: this.preferences["Dostupno tijekom sezone?"],
      furnished: this.preferences["Namještaj?"],
      lift: this.lift,
    };
    /* Izbacujemo sve null, false i 0 vrijednosti kako ne bi nepotrebno imali dugačak URL */
    const filteredQueryParams = Object.fromEntries(
      Object.entries(queryParams).filter(
        ([key, value]) => value !== null && value !== false && value !== 0
      )
    );
    /* Dodajemo te parametre te idemo na tu stranicu */
    this.$router.push({ query: filteredQueryParams }).catch(() => {});
    this.$router.go();
  },
},
```

Slika 56. Slanje parametra filtera u URL.

6.3.2 Komponenta za oglase

Druga komponenta na početnom pogledu, je komponenta za oglase, prikazana na slici 57. Ona samo uzima podatke koji su proslijeđeni u objekt “props”, i pokazuje neke od njih, kao što su slika, cijena, lokacija itd. Ima gumb kao i sliku koje oboje pokreću funkciju “pushRoute”, koja u sebi ima parametar id-a tog oglasa. Funkcija prikazana na slici 58 jednostavno šalje korisnika na pogledu za detalje oglasa i u parametre stavlja id tog specifičnog oglasa.

```
<template>
  <div class="card" style="width: 18rem">
    <img
      @click="pushRoute(props._id)"
      :src="props.url[0].url"
      class="card-img-top clickable"
    />
    <div class="card-body">
      <h5 class="card-title">{{ props.title }}</h5>
    </div>
    <ul class="list-group list-group-flush">
      <li class="list-group-item">Površina: {{ props.surface }} m²</li>
      <li class="list-group-item">Cijena: {{ props.price }} €</li>
      <li class="list-group-item">Lokacija: {{ props.location }}</li>
      <li class="list-group-item">Objavio: {{ props.createdBy.email }}</li>
      <li class="list-group-item">Zadnje izmijene: {{ timeAgo }}</li>
    </ul>
    <div class="card-body">
      <button class="btn" @click="pushRoute(props._id)">Detalji oglasa</button>
    </div>
  </div>
</template>
```

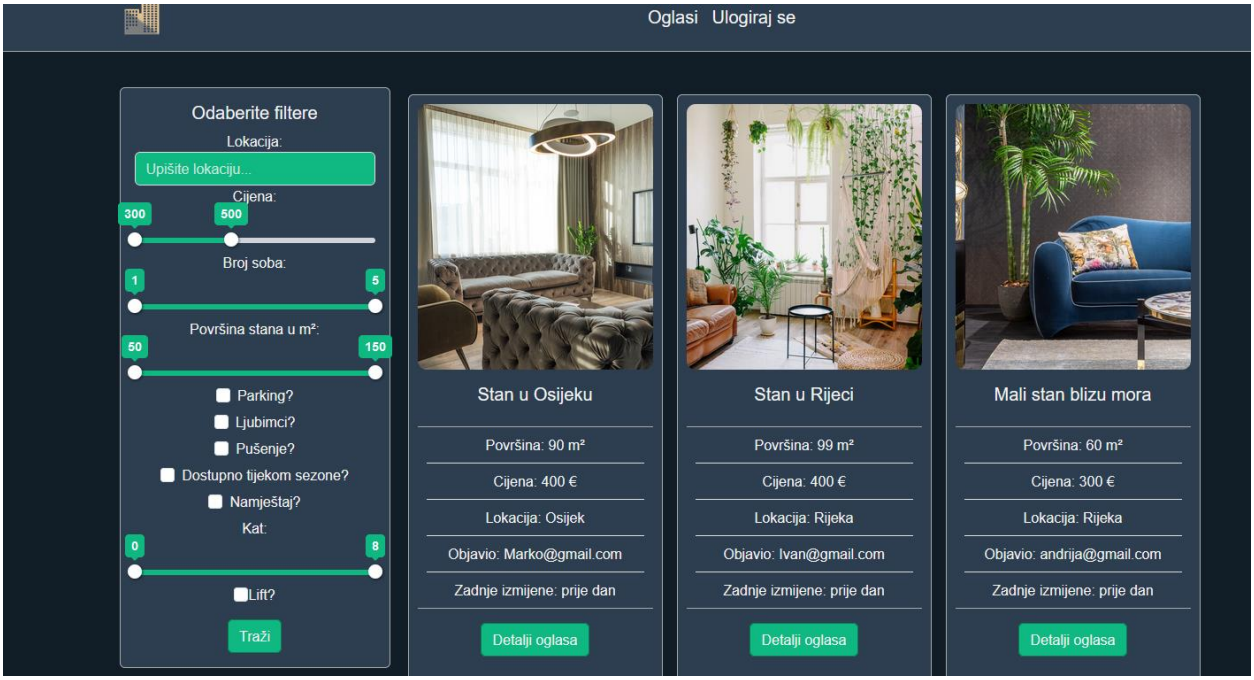
Slika 57. Komponenta oglasa

```
  methods: {
    pushRoute(paramsId) {
      this.$router.push({ name: "detail", params: { id: paramsId } });
    },
  },
```

Slika 58. Funkcija komponente oglasa

6.3.3. Filtriranje oglasa

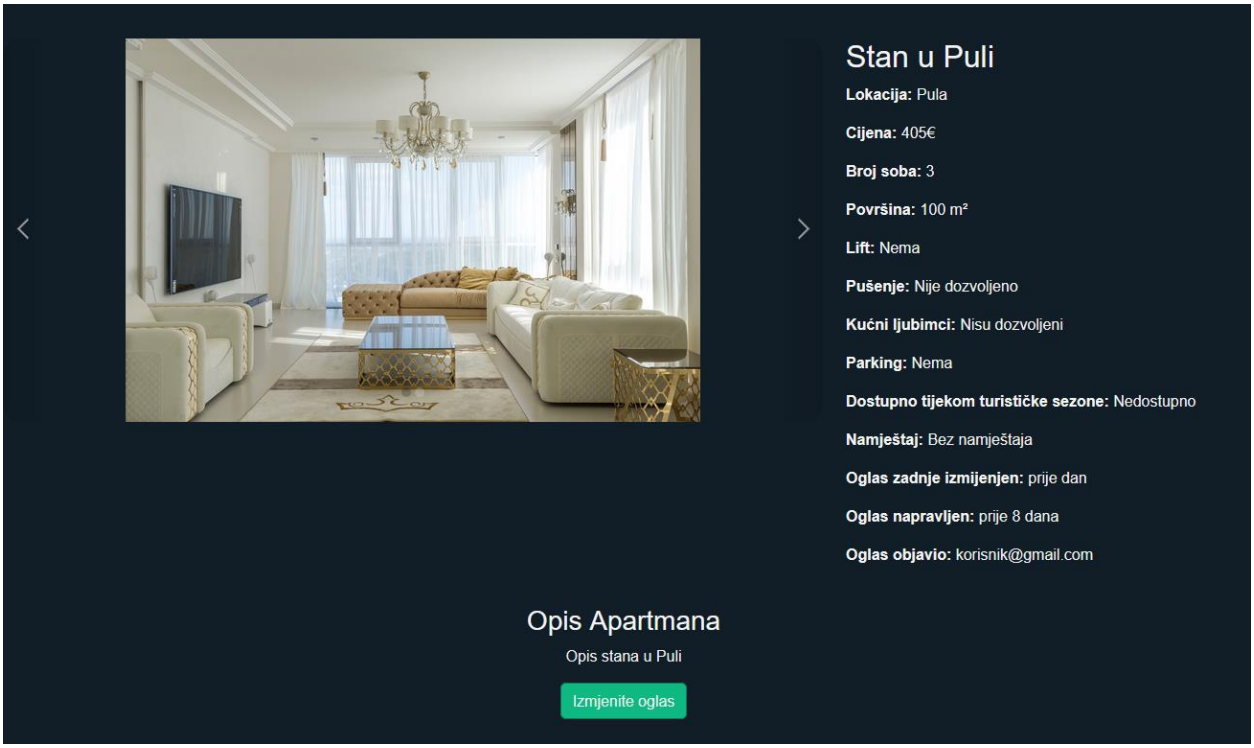
Nakon što korisnik postavi neke filtere, i stisne gumb za filtriranje, stranica se osvježi i dobiva nazad sve oglase koje idu po tom filteru, na primjeru na slici 59 možemo vidjeti da je korisnik ograničio da želi vidjeti samo stanove ispod 500€, te vidimo da svi oglasi skuplji od toga nisu prisutni.



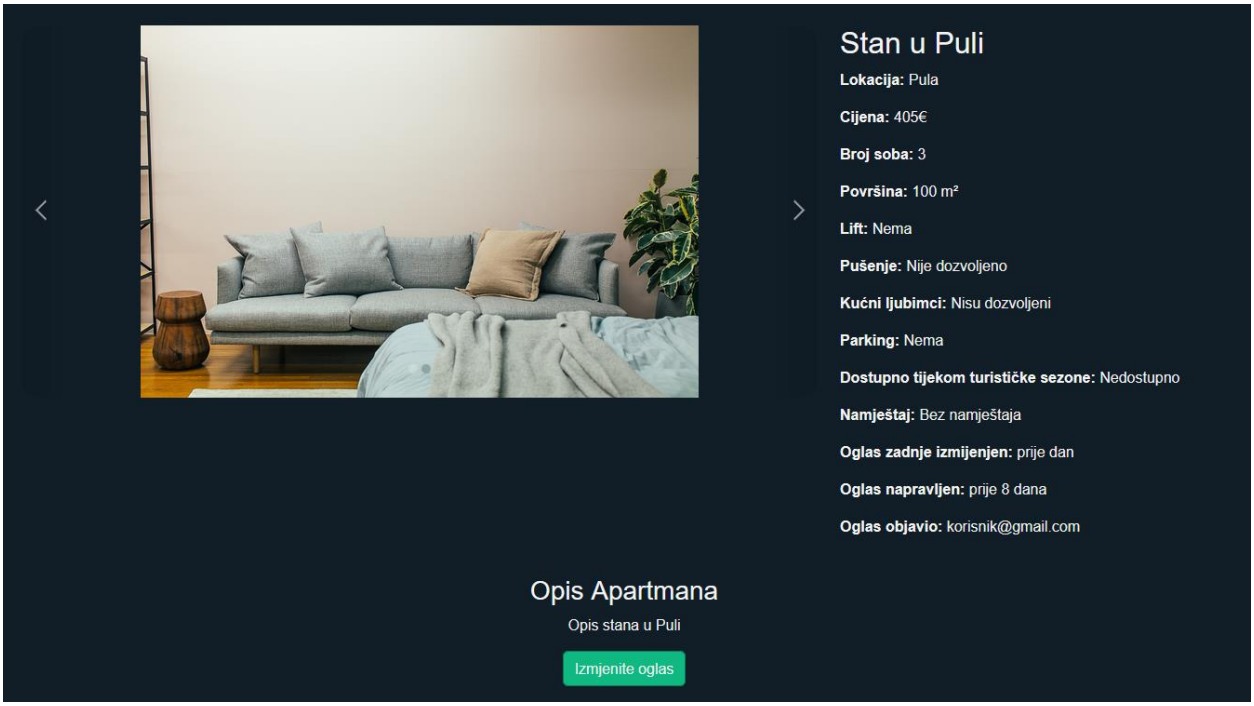
Slika 59. Filtriranje po cijeni

6.4 Detalji oglasa

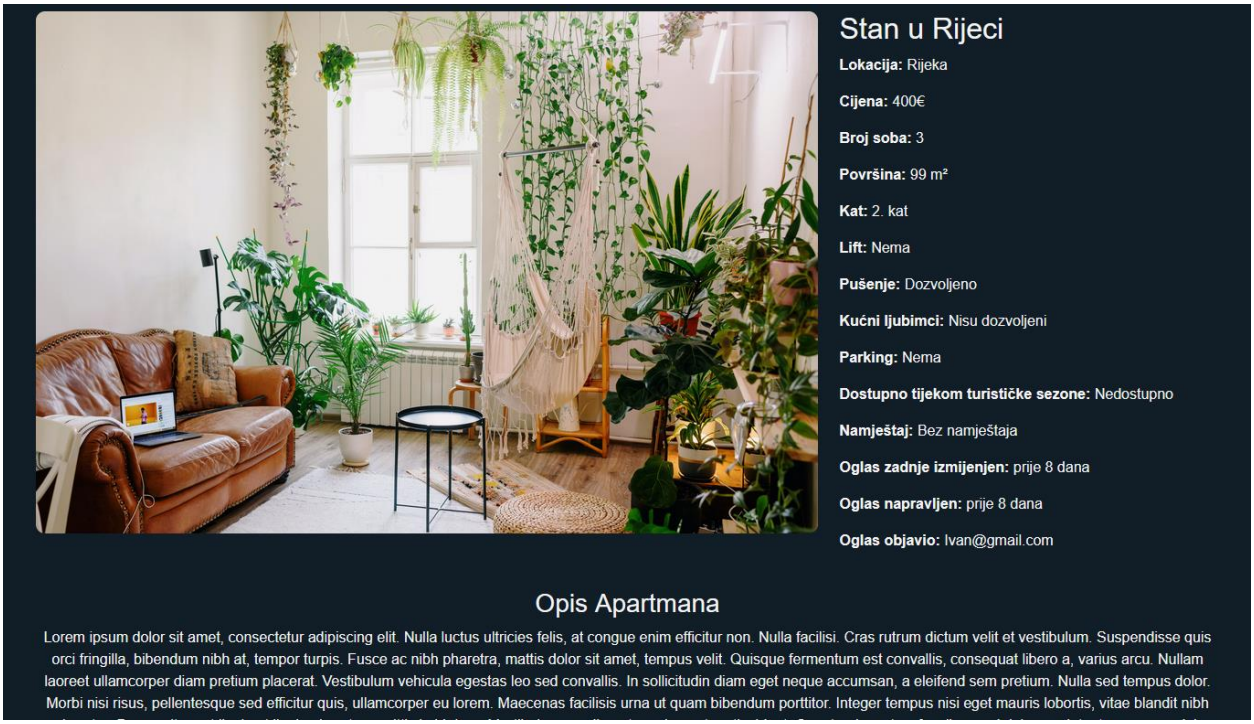
Pogled za detalje oglasa prikazuje sve podatke o oglasu, prikazan na slici 60. Ako oglas ima više slika koristit će carousel i samo će vrtjeti sve slike kao što je prikazano na slici 61, ili korisnik može stisnuti strelice i gledati sve slike. Ako je samo jedna slika, neće biti carousel već samo jedna slika prikazano na slici 62. Kod za tu logiku je prikazan na slici 63 samo gleda duljinu polja svih slika, ako je dulje od 1, znači da ima više slika te se stoga pokazuju sve slike u carouselu, dok je u suprotnom samo statična slika. Na dnu oglasa imamo dva gumba prikazana na slici 64 jedna je vidljiva samo korisniku koji je napravio taj oglas, drugi je vidljiv svim ostalim korisnicima, dok gosti neće vidjeti nijedan od tih gumbova. Slika 65 prikazuje obje funkcije, “pushRoute” funkcija šalje korisnika na stranicu za izmjenu podataka stranice, dok funkcija “newConversation”, šalje zahtjev za kreiranje novog razgovora, te nakon što dobiva odgovor šalje korisnika na stranicu za dopisivanje s parametrima odabranog korisnika kako bi mu bio odmah otvoren taj razgovor.



Slika 60. Detalji oglasa



Slika 61. Prikaz carousela



Slika 62. Statična slika

```
<div class="col-8">
  <slider animation="fade" v-if="adData.url.length > 1">
    <slider-item v-for="image in adData.url" :key="image._id">
      <img class="slider-image" :src="image.url" />
    </slider-item>
  </slider>
  <!-- Slika -->
  <img
    v-if="adData.url.length == 1"
    :src="adData.url[0].url"
    class="card-img-top"
  />
</div>
```

Slika 63. Logika za sliku ili carousel

```
    <button
      class="btn"
      v-if="currentUser != adData.createdBy.email && currentUser"
      @click="newConversation()"
    >
      Dopisujte se s korisnikom
    </button>
    <!-- Izmjenjivanje oglasa -->

    <button
      class="btn"
      v-if="currentUser == adData.createdBy.email"
      @click="pushRoute()"
    >
      Izmjenite oglas
    </button>
  </div>
```

Slika 64. Logika za prikaz gumbova

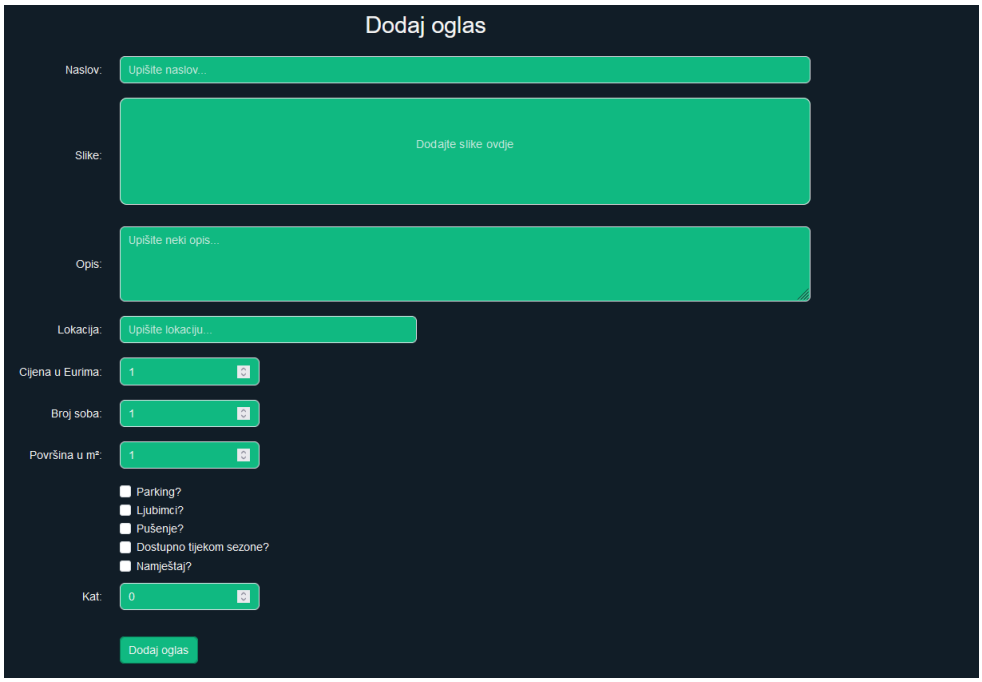
```
methods: {
  async newConversation() {
    await messages.addConversations(this.adData.createdBy);
    this.$router.push({
      name: "messages",
      params: { id: this.adData.createdBy },
    });
  },
  pushRoute() {
    this.$router.push({ name: "edit", params: { id: this.adData._id } });
  },
},
```

Slika 65. Funkcije detalja oglasa

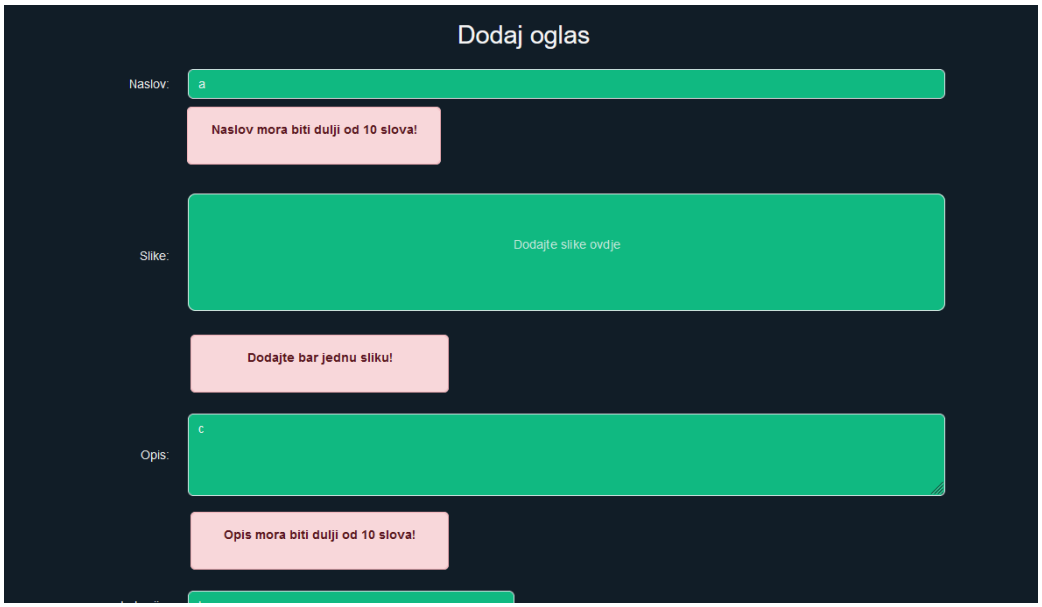
6.5 Dodavanje oglasa

Slika 66 prikazuje pogled za dodavanje novih oglasa. Sve elementi su obične forme jedina posebna forma je za slike, ako korisnik krivo ispuni forme dobit će greške koje će mu reći što treba ispraviti, što je prikazano na slici 67. Na slici 68 je prikazan ispunjen oglas, vidljive su crtice koje označuju napredak "uploadanja" slike. Za dodavanje slika koristi se biblioteka dropzone,

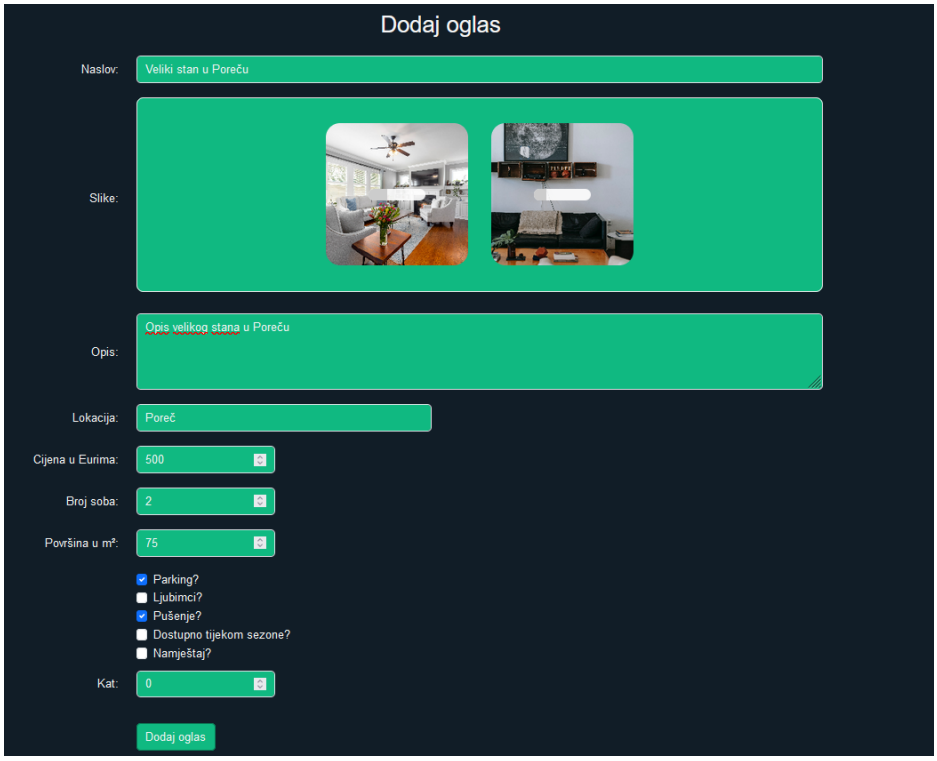
omogućava dodavanje slika sa “drag and dropom” ili možemo samo odabrati datoteke koje želimo poslati. Možemo vidjeti implementaciju na slici 69, slika 70 prikazuje funkcije koje su prisutne za dodavanje slika. Svaki put kad se doda nova slika pokreće se added file, te provjeravamo ako je već ta slika dodano, ako je mičemo je i dajemo upozorenje korisniku da ne može imati iste slike. Nakon što je slika uspješno dodana, kreće se slati na cloudinary di se ta slika sprema, te se nakon toga spremaju podaci u listu svih slika. Ako korisnik želi maknuti sliku, stisne na gumb za micanje slike prikazan na slici 71 te se poziva funkcija za micanje slike, koja miče sliku iz liste svih slika. Na slici 72 su prikazane postavke za dropzone, u sebi sadrži poruke za korisnika kako bi mogao razumjeti, te spremanje slike na cloudinary api. Kada je korisnik ispunio sva polja po pravilima aplikacije, može stisnuti gumb za dodavanje oglasa, zatim se oglas šalje na backend, sprema u bazu podataka, i korisnika se šalje na prikaz svih oglasa, što je prikazano na slici 73 gdje se može vidjeti oglas koji je ispunjen na slici 58. Također možemo vidjeti da je stavljen na prvo mjesto, jer je najnoviji oglas, a svi oglasi su sortirani tako da su na početku najnoviji oglasi.



Slika 66. Pogled dodavanja novog oglasa



Slika 67. Greške pri dodavanju novog oglasa



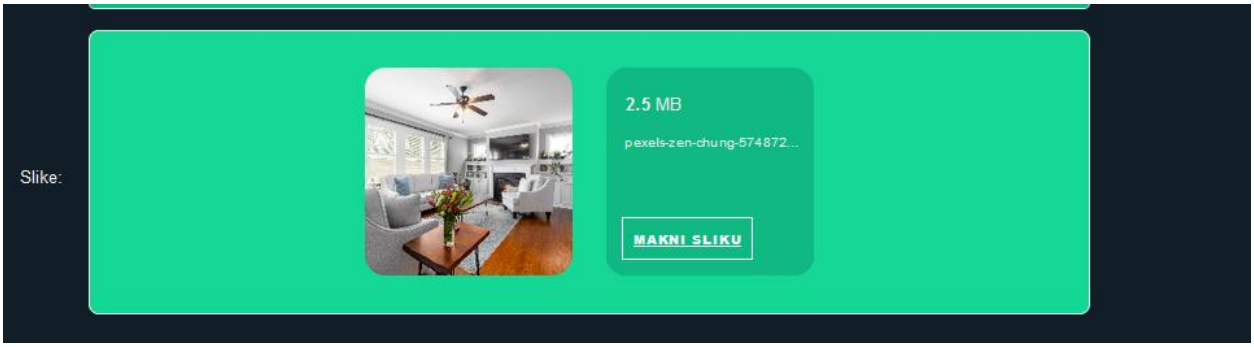
Slika 68. Ispunjen oglas

```
</div>
<!-- Upload za slike -->
<div class="center d-flex justify-content-center align-items-center">
  <dropzone
    ref="myDropzone"
    id="image-dropzone"
    :options="dropzoneOptions"
    @vdropzone-success="handleSuccess"
    @vdropzone-file-added="addedFile"
    @vdropzone-removed-file="removeFile"
  ></dropzone>
</div>
```

Slika 69. Dropzone komponenta

```
/* Funkcija koja se izvodi nakon što je slika uploadana, punimo ad.url array sa url-ovima */
handleSuccess(file, response) {
  console.log("Name:", file.name);
  this.ad.url.push({ name: file.name, url: response.url });
},
addedFile(file) {
  if (this.ad.url.some((obj) => obj["name"] === file.name)) {
    alert("Ne možete imati istu sliku dvaput!");
    this.$refs.myDropzone.removeFile(file);
  } else {
    console.log("OK");
  }
},
/* Funckija za micanje slika */
removeFile(file) {
  console.log("File removed! ", file.name);
  const index = this.ad.url.findIndex((obj) => obj["name"] === file.name);
  this.ad.url.splice(index, 1);
},
```

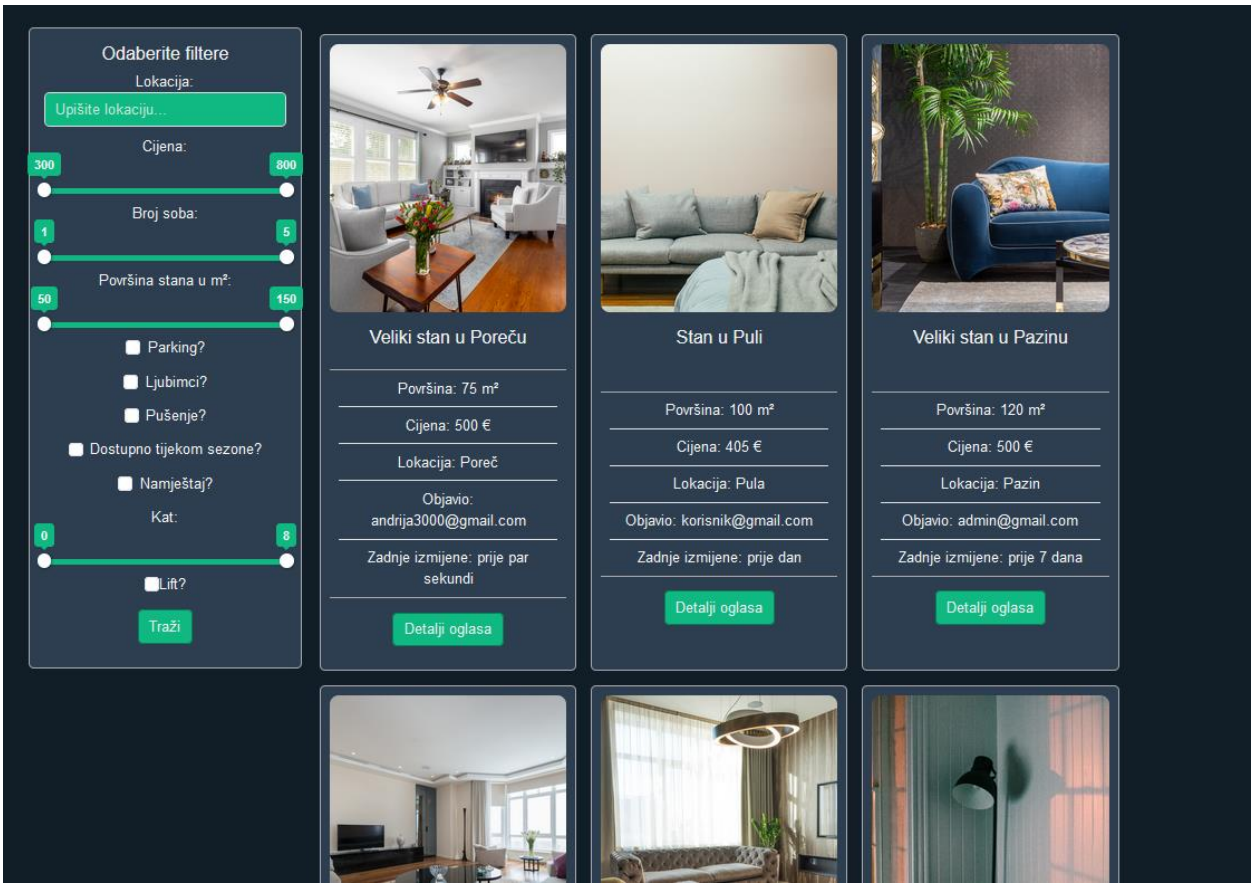
Slika 70. Funkcije dropzonea



Slika 71. Micanje slike

```
dropzoneOptions: {
  url: "https://api.cloudinary.com/v1_1/dymtk2sbc/upload",
  acceptedFiles: "image/*",
  dictRemoveFile: "Makni sliku",
  dictMessage: "Makni sliku",
  dictDefaultMessage: "Dodajte slike ovdje",
  dictFileTooBig:
    "Slika je prevelika ({{filesize}}MiB). Dopuštena veličina: {{maxFilesize}}MiB.",
  dictCancelUpload: "Makni sliku",
  dictUploadCanceled: "Slika maknuta.",
  dictInvalidFileType: "Nije slika.",
  params: {
    upload_preset: "ml_default",
  },
  addRemoveLinks: true,
},
```

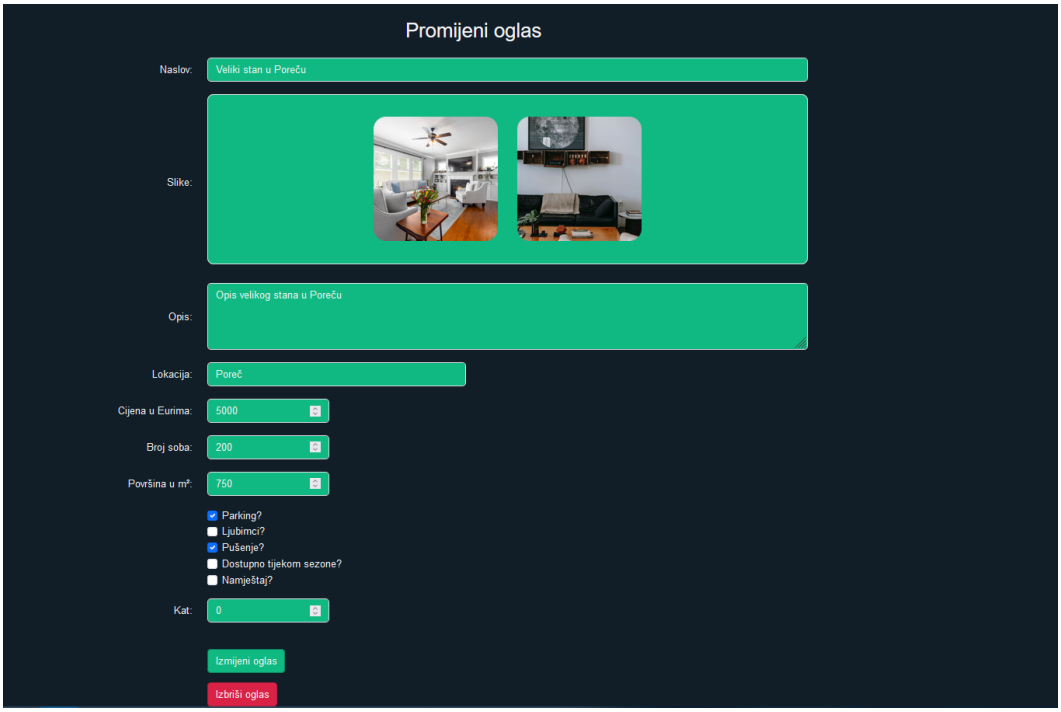
Slika 72. Opcije dropzonea



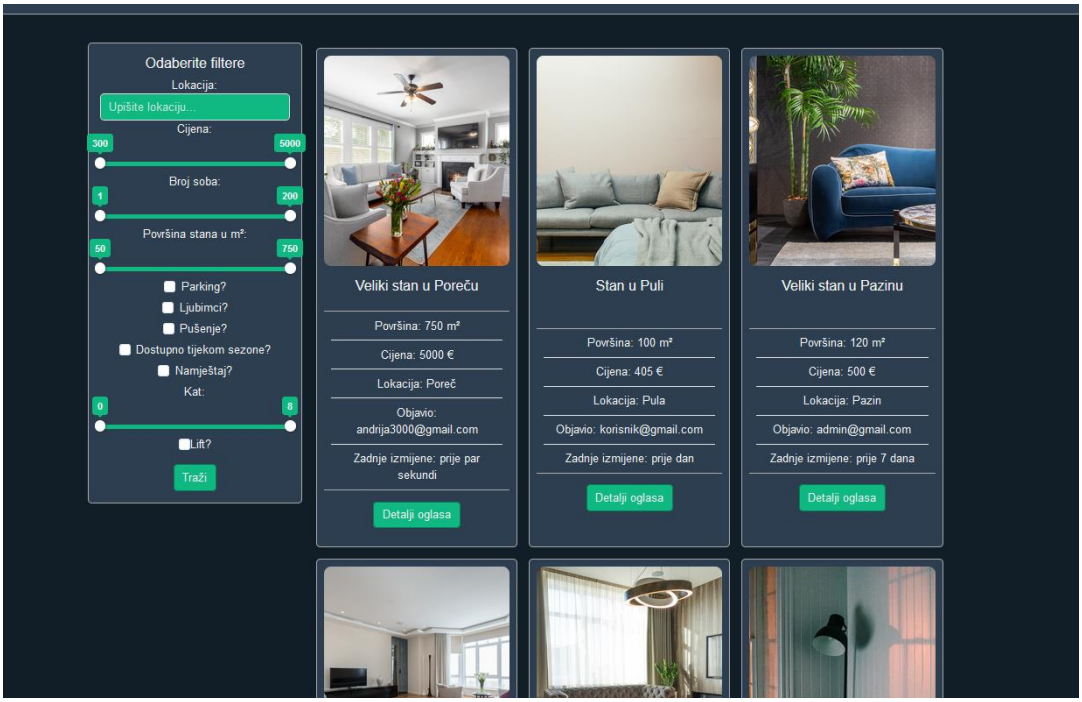
Slika 73. Novi oglas dodan

6.6 Izmjena oglasa

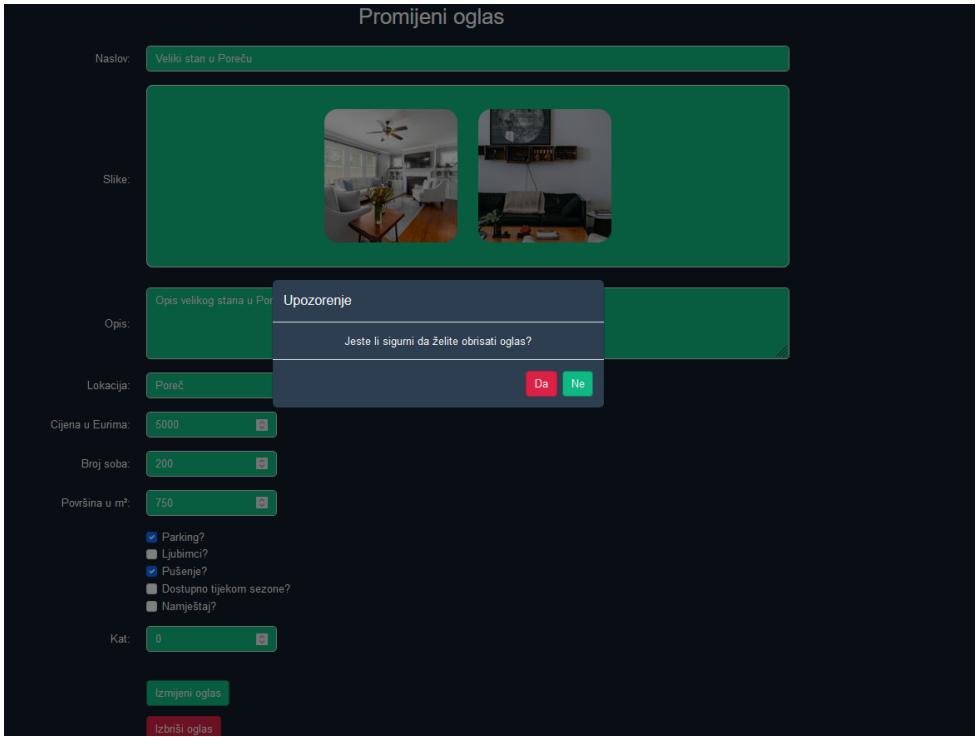
Pogled za izmjenu oglasa je identičan i radi identično kao pogled za dodavanje oglasa. Samo što uzima već sve podatke iz oglasa. Na primjeru na slici 74 je izmijenjena cijena, površina i broj soba oglasa, prikazanog na slici 73 nakon što korisnik izmjeni oglas, promijeniti će se podaci o oglasu u bazi podataka, te će se također promijeniti maksimalne i minimalne vrijednosti filtera, što je prikazano na slici 75. Ako korisnik odluči izbrisati oglas, pojavit će mu se upozorenje prikazano na slici 76 koje ga pita ako je siguran da želi maknuti oglas. Ako želi, stisne da i oglas će biti maknut te će korisnik biti poslan na početni pogled, gdje više neće biti taj oglas. Na slici 77 je prikazano brisanje izmjenjenog oglasa sa slike 74, oglas više nije u bazi podataka te se ne prikazuje više na prikazu svih oglasa, također su se promijenile maksimalne i minimalne vrijednosti filtera. Ako stisne ne, neće se ništa dogoditi, samo će se upozorenje maknuti s ekrana.



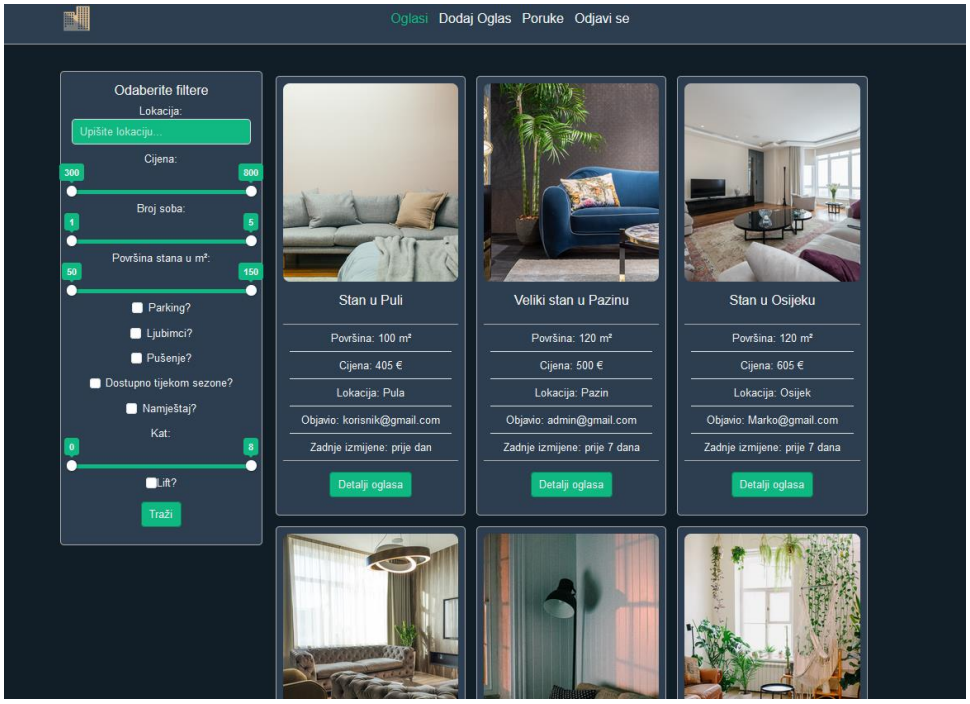
Slika 74. Izmjena oglasa



Slika 75. Izmijenjen oglas i maksimalne vrijednosti filtera



Slika 76. Brisanje oglasa

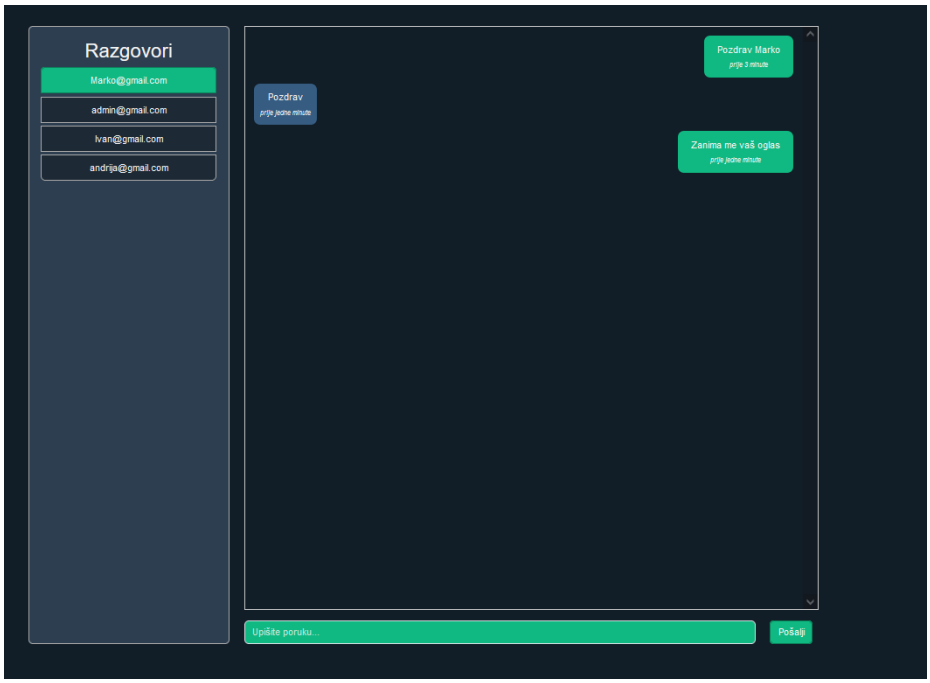


Slika 77. Izbrisan oglas

6.7 Poruke

Izgled poruka je prikazan na slici 78. S lijeve strane su prikazani svi razgovori, dok se u glavnom prozoru vide sve poruke u tom razgovoru. Slika 79 prikazuje kod stranice, razgovori su lista koja se stvara pokretanjem petlje na svim razgovorima koje korisnik ima. Poruke su odvojena komponenta koje se pune s podacima o svim porukama razgovora. Slika 80 prikazuje sve objekte na stranici, kao i mounted funkciju koja se poziva kada se očitava stranica, prvo se šalje zahtjev za sve razgovore korisnika na backend, nakon što se dobije taj odgovor, provjerava se ako uopće ima nekih razgovora. Ako nema daje se upozorenje korisniku da nema razgovora, i šalje korisnika na glavnu stranicu, ako ima razgovora, sprema id korisnika u objekt trenutačnog korisnika, te stvara vezu uz pomoć socket servisa koji je prijašnje objašnjen tako što šalje u funkciju za spajanje id korisnika. Nakon što je uspostavljena veza gleda se ako klijent ima neke parametre u svojoj putanji, što bi značilo da je korisnik došao na razgovor s oglasa te se šalje funkciji za odabir razgovora parametri iz putanje. Ako nema samo se odabire prvi razgovor u listi. Slika 81 prikazuje funkcije za stvaranje event busa koji sluša event koji će se dogoditi kada korisnik primi poruku preko socket veze, event bus se stvara čim se stvori stranica. Kada se dogodi događaj nove poruke, spremaju se podaci

poruke u listu svih poruka. Before destroy se izvršava prije nego što se mijenja pogled, ovdje se odspaja socket veza, te se gasi event bus pošto se ne koriste izvan stranice za poruke, te bi se svaki puta stvarali novi event busevi i socket veze kada se očita stranica, bez da se miču stari. Updated() se poziva svaki put kad se promijeni neka vrijednost, ovo se koristi kako bi se scroll pomaknuo na dno okvira svaki put kada korisnik upiše novu poruku, ili primi novu poruku. Slika 82 prikazuje funkcije korištene na stranici. Funkcija select conversation prima razgovor u parametre, te sprema to u objekt odabranog razgovora. Zatim sprema u objekt polje koje određuje između koja je dva korisnika razgovor, te šalje to polje na backend kako bi dobilo odgovor u kojem su sve poruke u tom razgovoru. Nakon što dobije odgovor, sprema sve poruke iz odgovora u listu svih poruka. Funkcija za slanje poruka će se izvesti samo za poruke koje nisu prazna polja, sprema se objekt koji se sastoji od same poruke kao i razgovora kojem ta poruka pripada, šalje event preko socket veze s tim objektom, te šalje isti objekt na backend kako bi se spremio u bazu podataka. Ako se dobije odgovor od baze podataka, poruka se lokalno sprema kako se ne bi nepotrebno tražilo opet sve poruke s backenda, u suprotnom funkcija baca error. Na kraju se objekt nove poruke briše kako korisnik ne bi morao sam brisati prošlu poruku.



Slika 78. Sučelje za poruke

```
<!-- sidebar-messages -->
<div class="sidebar-messages col-3 rounded">
  <h2>Razgovori</h2>
  <ul class="list-group" v-if="this.selectedConversation">
    <li
      class="list-group-item clickable"
      v-for="conversation in conversations"
      :key="conversation._id"
      :class="{
        active: selectedConversation._id == conversation._id,
      }"
      @click="selectConversation(conversation)"
    >
      {{ conversation.email }}
    </li>
  </ul>
</div>

<!-- Glavni prozor -->
<div class="col-9">
  <div class="container chat-container rounded">
    <div
      class="chat-area"
      ref="scrollContainer"
      v-if="this.selectedConversation"
    >
      <message
        v-for="message in messages"
        :key="message.id"
        :message="message"
      />
    </div>

    <!-- input -->
    <div class="input-group mt-3 row">
      <div class="col-11">
        <input
          type="text"
          class="form-control"
          placeholder="Upišite poruku..."
          v-model="newMessage"
          @keyup.enter="sendMessage"
        />
      </div>
      <div class="input-group-append col-1">
        <button class="btn btn-primary" @click="sendMessage">
          Pošalji
        </button>
```

Slika 79. Kod za pogled poruka

```
export default {
  name: "MessagesView",
  data() {
    return {
      conversations: [],
      selectedConversation: null,
      messages: [],
      currentUser: "",
      newMessage: "",
    };
  },
  async mounted() {
    const response = await messages.getConversations();

    response.conversations.forEach((item) => {
      this.conversations.push(item);
    });

    if (!this.conversations) {
      await alert("Nemate otvorene razgovore!");
      this.$router.push({ name: "home" });
    } else {
      this.currentUser = response._id;
      SocketioService.setupSocketConnection(this.currentUser);

      if (this.$route.params.id) {
        this.selectConversation(this.$route.params.id);
      } else {
        this.selectConversation(this.conversations[0]);
      }
    }
  },
```

Slika 80. Objekti i mounted funkcija poruka

```
  created() {
    eventBus.$on("message-received", (data) => {
      this.messages.push({ type: false, message: data.message });
    });
  },
  beforeDestroy() {
    SocketioService.disconnect();
    eventBus.$off("message-received");
  },
  updated() {
    this.$refs.scrollContainer.scrollTop =
      this.$refs.scrollContainer.scrollHeight;
  },
```

Slika 81. Kreiranje i micanje eventa i socketa

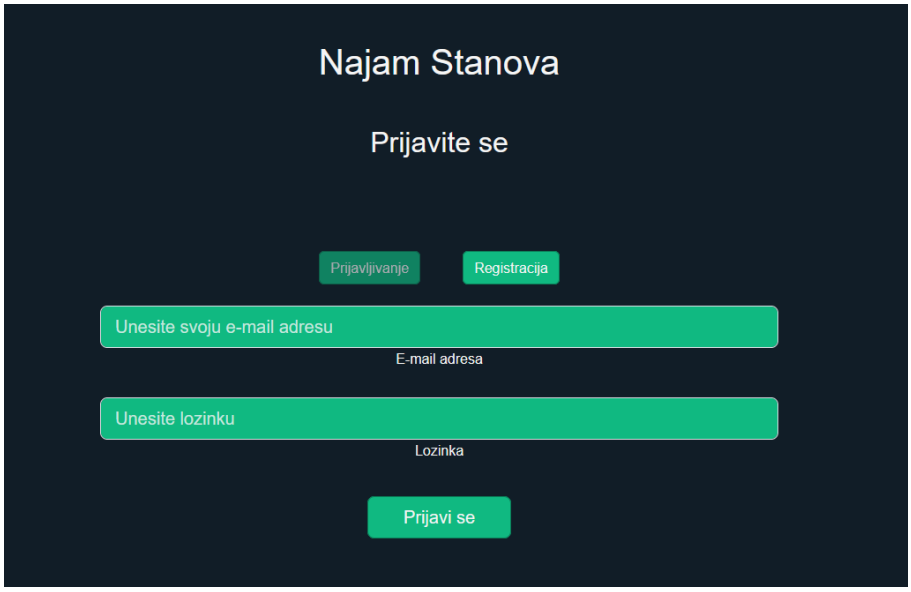
```
methods: {
  async selectConversation(conversation) {
    console.log(conversation);
    this.selectedConversation = conversation;
    const params = {
      from: this.currentUser,
      to: this.selectedConversation._id,
    };
    const messagesArray = await messages.getMessages(params);
    console.log(messagesArray);
    this.messages = messagesArray;
  },
  async sendMessage() {
    if (this.newMessage.trim() !== "") {
      const body = {
        message: this.newMessage,
        from: this.currentUser,
        to: this.selectedConversation._id,
      };
      SocketioService.sendMessage(body);
      const response = await messages.addMessage(body);
      if (response) {
        this.messages.push({ type: true, message: this.newMessage });
      } else {
        throw new Error(500);
      }

      this.newMessage = "";
    }
  },
},
```

Slika 82. Funkcije poruka

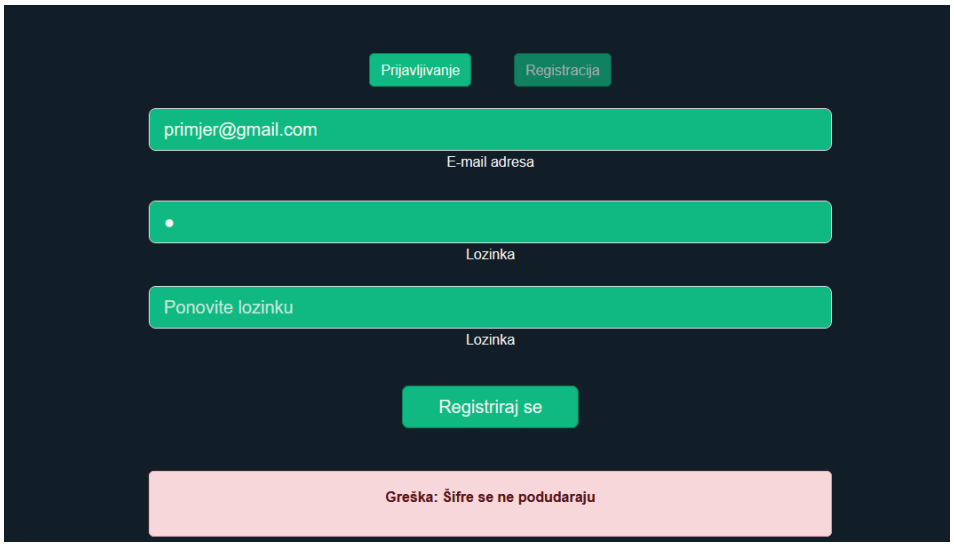
6.8 Prijava/Registracija

Za prijavu i registraciju koristi se ista stranica, korisnik samo stisne gumb kako bi odabrao koju funkciju želi izvršiti, prikazano na slici 83. Jedina razlika je što registracija ima dodatno polje za potvrđivanje lozinke, prikazano na slici 84. Registracija provjerava ako se lozinke podudaraju, ako ne daje do znanja korisnika s upozorenjem, prikazano na slici 85. Slika 86 prikazuje neuspješnu prijavu, događa se ako je šifra kriva, ili ne postoji korisnički nalog u bazi podataka. Slika 87 prikazuje funkciju za prijavu kao i funkciju za registraciju. Funkcija za prijavu prvo gleda ako su ispunjena sva polja, ako jesu šalje zahtjev na backend za prijavu korisnika, koja će spremiti token u preglednik korisnika ako je točna lozinka i korisnik. Ako funkcija dobije odgovor, osvježi stranicu,

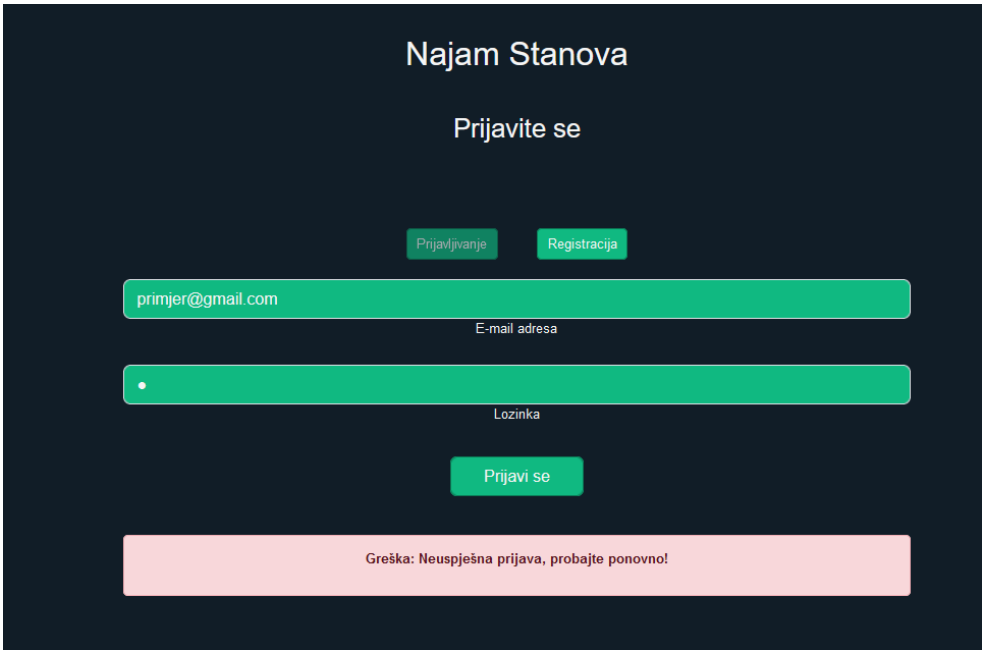
pošto je korisnik prijavljen sada poslat će ga na glavnu stranicu. Ako nije poslat će korisniku upozorenje da ne valja prijava. Funkcija za registraciju radi na isti način jedino je razlika na backendu, dobit će odgovor za registraciju korisnika, koji će biti kriv ako su podaci krivi ili je već zauzet email. Ako funkcija dobije uspješan odgovor, pokreće funkciju za prijavu s istim emailom i lozinkom.



Slika 83. Prijava



Slika 84. Upozorenje za registraciju



Slika 85. Neuspješna prijava

```
async login() {
  if (this.user.email && this.user.pass) {
    const success = await users.loginUser(this.user);
    console.log("success: ", success);
    if (success) {
      this.$router.go();
    } else {
      this.errorState = true;
      this.errorMessage = "Neuspiješna prijava, probajte ponovno!";
    }
  } else {
    this.errorState = true;
    this.errorMessage = "Ne smijete ostaviti prazna polja!";
  }
},
async signup() {
  if (this.user.email && this.user.pass) {
    const success = await users.setUser(this.user);
    console.log("success: ", success);
    if (success) {
      this.login();
    } else {
      this.errorState = true;
      this.errorMessage = "Krivi podaci, ili je zauzet email!";
    }
  } else {
    this.errorState = true;
    this.errorMessage = "Ne smijete ostaviti prazna polja!";
  }
},
```

Slika 86. Funkcije prijave i registracije

7. Zaključak

Cilj ovog završnog rada je bilo stvoriti web aplikaciju za izdavanje i gledanje oglasa za iznajmljivanje stanova. Osim oglasa ima i funkcionalnost dopisivanja između korisnika u stvarno vremenu. Web aplikacija je napravljena uz pomoć više tehnologija, za stvaranje korisničkog sučelja kao i koda koji se odvija na strani klijenta, je korišten framework Vue.js koja pruža jednostavan i fleksibilan razvoj web aplikacije. Ima veliki broj mogućnosti, te mnogobrojno nadogradnja mogućih kroz instalaciju biblioteka. Za izgradnju servera korišten je Node.js uz pomoć Express.js-a, ove dvije tehnologije zajedno pružaju temelj za jednostavan i brz razvoj servera, te pomažu u stavljanja fokus na logiku i kodiranje, umjesto na uspostavljanje veza i definiranja pravila za rute. Za bazu podataka web aplikacije je korišten MongoDB koji omogućava vrlo brzo i jednostavno postavljanje baze podataka. Osim toga pošto je nerelacijska baza podataka, dopušta puno brži rad od relacijskih baza podataka. Cijeli backend je pokrenut u Docker containeru koja omogućava pokretanje samog servera i baze podataka. Ovo omogućava izbjegavanje problema kompatibilnosti, kao i pokretanje baze podataka lokalno što ubrzava proces razvoja kao i komunikaciju sa samom bazom podataka.

U završnom radu je dokumentirana cijela aplikacija, kako se koristi i što se događa u kodu aplikacije. Objašnjena je implementacija ruta na serveru, socket veze koje se koriste za dopisivanje u stvarnom vremenu te korisničko sučelje koje korisnici koriste. Dokumentacija detaljno objašnjava cijelu aplikaciju što dovodi do toga da je vrlo lagano razumjeti aplikaciju i zašto su neke stvari implementirane.

Jedno od glavnih ograničenja aplikacije bi bilo to što poruke između korisnika nisu kriptirane već su u običnom tekstu. Ovo je rizik jer omogućava prisluškivanje razgovora između korisnika, što bi moglo dovesti do sigurnosnih problema ako korisnici razgovaraju o privatnim podacima. Ovo bi se moglo popraviti implementacijom neke vrste enkripcije kao što je “end-to-end” enkripcija, to bi osiguralo da poruke koje šaljemo sigurno idu korisniku kojem su namijenjene i može ih samo pročitati korisnik za kojeg su namijenjene poruke. Aplikacija također nema razinu prepoznatljivosti i količinu korisnika kao druga popularna rješenja.

Unatoč tome, web aplikacija za iznajmljivanje stanova ima dosta prednosti. Aplikacija ima moderno i pregledno korisničko sučelje, koje se vrlo lako koristi. Način na koji je aplikacija izgrađena, kao i tehnologije koje koristi omogućavaju vrlo laganu izmjenu kao i nadogradnju funkcionalnosti aplikacije. Nakon analize trenutačnih programskih rješenja, identificirali smo neke prednosti i nedostatke. Aplikacija je dizajnirana po tim nedostacima i prednostima, pokušala je riješiti nedostatke tih aplikacija, i ukomponirati dobre strane kako bi se stvorila optimalna aplikacija.

Aplikacije bi mogla koristiti od nekoliko proširenja. Jedno od njih bi mogla biti verifikacija korisnika preko njihovog broja mobitela, ili OIB-a. Moglo bi se dodati funkcionalnost ocjenjivanja korisnika, kao i prijava korisnika za zloupotrebu aplikacije. To bi se pokazivalo na profilu korisnika kao i na njihovim oglasima. Kako se aplikacija koristi, korisnici aplikacije bi mogli davati ocjene funkcionalnosti, i preporučiti što bi se trebalo promijeniti, dodati ili maknuti. Osim toga trebalo bi uvesti veću razinu sigurnosti na stranici.

Literatura

1. Facebook marketplace. (n.d.). Retrieved September 19, 2023, from https://www.facebook.com/marketplace/

2. Njuškalo. (n.d.). Reterieved September 19, 2023, from https://www.njuskalo.hr/

3. Introduction Vue.js. (n.d.). Retrieved September 20, 2023, https://vuejs.org/guide/introduction.html

4. Kopecky, C. (2020, September 30). What is Node.js? A beginner's introduction to JavaScript runtime, https://www.educative.io/blog/what-is-nodejs#what-is

5. Codecademy What is Express.js?. (n.d.). Retrieved September 20, 2023, https://www.codecademy.com/article/what-is-express-js

6. Get Started with MongoDB. (n.d.). Retrieved September 20, 2023, https://www.mongodb.com/basics/get-started

7. Introduction Socket.IO. (n.d.). Retrieved September 20, 2023, https://socket.io/docs/v4/

8. Docker overview. (n.d.). Retrieved September 20, 2023, https://docs.docker.com/get-started/overview/

9. Why Visual Studio Code. (2023, September 7). Retrieved September 20, 2023, https://code.visualstudio.com/Docs/editor/whyvscode

10. Get started with GitHub documentation. (n.d.). Retrieved September 20, 2023, https://docs.github.com/en/get-started

11. Pexels (n.d.). Retrieved September 20, 2023, https://www.pexels.com

Popis slika