Web aplikacija za organiziranje i povezivanje ljubitelja društvenih igara

Kuveždić, Branimir

Sveučilište Jurje Dobrile u Puli
Fakultet informatike u Puli

Web aplikacija za organiziranje i povezivanje ljubitelja društvenih igara

Završni rad

Pula, lipanj 2024.godine

Sveučilište Jurja Dobrile u Puli Fakultet Informatike

Branimir Kuveždić

Web aplikacija za organiziranje i povezivanje ljubitelja društvenih igara

Završni rad

JMBAG: 0303103111, redovni student

Studijski smjer: Sveučilišni preddiplomski studij informatika

Kolegij: Web aplikacije

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Nikola Tanković

Komentor: mag. inf. Robert Šajina

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Branimir Kuveždić, 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 nedozvoljen 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

Branimir Kuveždić

U Puli, lipanj, 2024. godine

IZJAVA

o korištenju autorskog djela

Ja, dolje potpisani Branimir Kuveždić dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom „Web aplikacija za organiziranje i povezivanje ljubitelja društvenih igara“ 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, lipanj 2024. godine

Student

Branimir Kuveždić

Sažetak

Cilj završnog rada je napraviti kompletnu dokumentaciju Frontend-a (client) i Backend-a (api). Web aplikacija omogućuje korisnicima jednostavno dodavanje i pronalaženje događaja čiji je cilj okupiti ljubitelje društvenih igara. Praćenjem standarda sigurnosti za prijavu i registraciju, korisnici imaju svoj jedinstveni profil pomoću kojega pretražuju ili dodaju događaje. Nakon dodavanja događaja ili pretraživanja istog, događaji zatim postaju vidljivi svim korisnicima, te je omogućeno i naknadno uređivanje svojega događaja. Ovo intuitivno i jednostavno rješenje je glavni razlog korištenja aplikacije. Api i client zaduženi su da nesmetano surađuju i na taj način omoguće brzo i sigurno korištenje svakog dijela web aplikacije

Ključne riječi: MERN, JavaScript. MongoDB, React, Express, Node, društvene igre, Frontend, Backend

---

Abstract

The goal of this bachelor's thesis is to create complete documentation for both the Frontend (client) and the Backend (api). The web application allows korisniks to easily add and find events aimed at bringing together board game enthusiasts. By adhering to security standards for login and registration, korisniks have their unique profiles, which they use to search for or add events. Once an event is added or searched, it becomes visible to all korisniks, and korisniks can also edit their events afterward. This intuitive and straightforward solution is the main reason for using the application. The API and client are responsible for working seamlessly together, thereby enabling fast and secure use of every part of the web application.

Keywords: MERN, JavaScript. MongoDB, React, Express, Node, board games, Frontend, Backend

Sadržaj

1. Uvod

Nedvojbeno je da internet u današnjem svijetu služi kao jedna od glavnih potreba. Ako se većina komunikacije odvija preko interneta, to je dovoljan pokazatelj njegove važnosti. Osnovne stvari, pa sve do određenih aktivnosti su u potpunoj ovisnosti o internetu. Činjenica je da je internet nezamjenjivi dio našega života. U ovom slučaju, usredotočit ćemo se na web stranicu, koja je primarni način prikazivanja informacija. Od interaktivnih, statičkih i dinamičkih web stranica, od izuzetne su važnosti. Popularne su web aplikacije, koje pružaju određenu radnju isključivo putem web aplikacije.

Zadatak ovog završnog rada je razvoj web aplikacije koja će pružiti uslugu koja je isključivo bazirana preko interneta. Cijela funkcionalnost ovisi o internetu, što govori i o njegovoj važnosti, ali i prikazuje kako iskoristiti njegovu moć uz pomoć raznih alata. Aplikacija koristi popularnu MERN arhitekturu. Glavni alati za izradu su MongoDB, ExpressJS, ReactJS i NodeJS. Možemo primjetiti i uzorak, odnosno JavaScript koji je glavni jezik za cjelokupnu izradu aplikacije.

Aplikacija pruža korisnicima jednostavno i intuitivno iskustvo stvaranja društvenih događaja, te okuplja ljubitelje društvenih igara na jednom mjestu. Korisniku se omogućuje registriranje i prijava. Sigurnost je ovdje ključna, zato se prate standardi i dobre prakse kako omogućiti sigurnu prijavu. Na početnom zaslonu aplikacije, korisnici mogu započeti pretragu i filtriranje događaja, poput žanra društvene igre, grada i mnogi drugi. Osim toga, omogućeno je korisnicima stvaranje vlastitog događaja, kako bi oni imali ulogu stvaranja događaja. Naknadno uređivanje događaja je također dostupno, što dodatno omogućava proširenje ili prepravke svakog događaja. Ključno je napomenuti da je sigurnost bitna, zato svaka osoba može uređivati svoj događaj, a gledati sve ostale. Isto tako, promjena osobnih podataka je moguća.

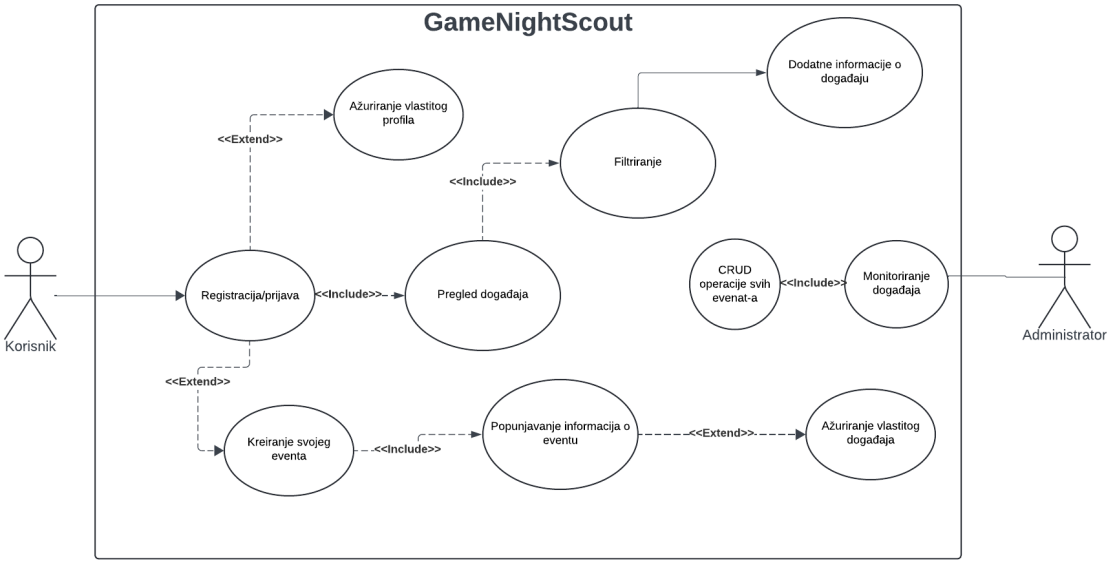
2. Motivacija

Ciljano tržište su osobe koje vole socijaliziranje i društvene igre. Puno je osoba koje spadaju pod ovu kategoriju, s obzirom na to da je način komunikacije pretežno putem interneta, to ne sprečava ljude da se okupljaju i dijele svoje vrijeme za zabavu. Kao glavna motivacija, ljudima je potreban alat koji na brz i jednostavan način omogućuje baš ono što im treba, u ovom slučaju pronalazak ili stvaranje društvenog događaja. Osim toga, pružanje ad-free iskustva, brzine i pouzdanosti su ključni motivi za korištenje ove aplikacije.

3. Korisnički zahtjevi

Korisnički zahtjevi su nam upute kojem nam daje korisnik. Te upute su stvar dogovora na koji način možemo stvoriti ono što je korisnik zamislio. Od detaljnijih, tehnički razrađenih i dobro promišljenih uputa, korisnici često znaju zadati nerealistična očekivanja, obično su to kratki rokovi u kojima trebamo nešto dostaviti. U ovom slučaju, razradit ćemo jedan teoretski korisnčki zahtjev na koji način nam je korisnik pristupio s željom za aplikaciju.

Use case dijagram nam prikazuje interakciju između korisnika i aplikacije. Kao dva početna aktera, korisnik i administrator, svaki sa svojim pravima koristi aplikaciju. Administrator ima potpunu ovlast nad aplikacijom, te ju monitorira i provjerava događaje. Korisnik kao početnu točku ima registraciju ili prijavu. Zatim ima nekoliko opcija: Ažuriranje profila, pregled događaja ili ipak stvaranje vlastitog događaja. Dijagram prikazuje grafički jasan put mogućeg kretanja korisnika po aplikaciji. Također, na taj način možemo i provjeriti smislenost tog kretanja. Greška bi bila ukoliko korisnik može doći do stranice za stvaranje događaja, bez da se prethodno registrirao.



Slika 1 Use case diagram

Registracija i Prijava Korisnika

- **Zahtjev**: Aplikacija mora omogućiti korisnicima registraciju i prijavu putem email adrese i lozinke.
- **Detalji**:
    - Korisnik unosi osnovne podatke (korisnikname, email, lozinka).
    - Lozinka mora biti hashirana i sigurno pohranjena u bazi podataka.
    - Prijava uz provjeru točnosti unesenih podataka.

Korisnički Profil

- **Zahtjev**: Svaki korisnik mora imati svoj jedinstveni profil.
- **Detalji**:
    - Mogućnost uređivanja osobnih podataka (korisnikname, email, lozinka).
    - Pregled svih stvorenih i prijavljenih događaja.
    - Mogućnost postavljanja profilne slike.

Kreiranje i Uređivanje Događaja

- **Zahtjev**: Korisnici moraju imati mogućnost kreiranja i uređivanja događaja.
- **Detalji**:
    - Unos detalja o događaju (naziv, opis, datum, vrijeme, lokacija, kategorija društvene igre).
    - Postavljanje slike za događaj.
    - Mogućnost naknadnog uređivanja detalja događaja samo od strane kreatora događaja.

Pretraživanje i Filtriranje Događaja

- **Zahtjev**: Aplikacija mora omogućiti pretraživanje i filtriranje događaja.
- **Detalji**:
    - Pretraživanje po nazivu događaja, kategoriji igre, gradu i datumu.
    - Filtriranje događaja po različitim kriterijima (npr. žanr igre, lokacija...).

Pregled

- **Zahtjev**: Korisnici moraju moći pregledavati
- **Detalji**:
    - Prikaz detalja o događaju (naziv, opis, datum, vrijeme, lokacija, kategorija igre).

Sigurnost

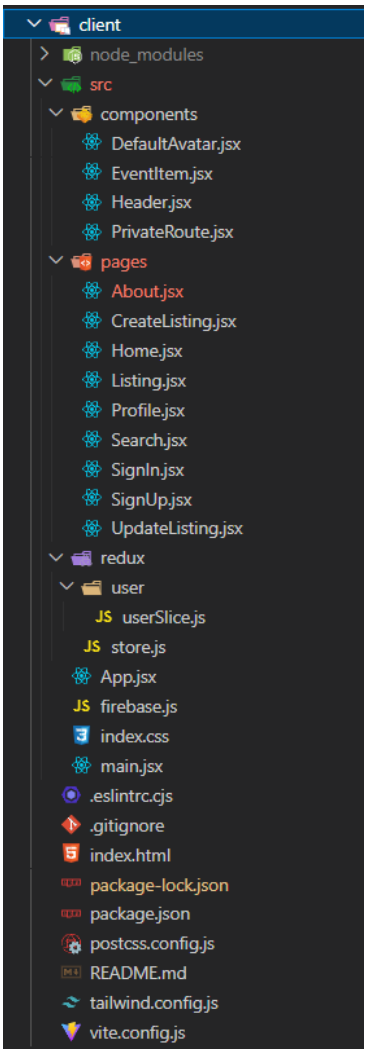
- **Zahtjev**: Aplikacija mora biti sigurna i zaštititi podatke korisnika.
- **Detalji**:
    - Hashiranje lozinki koristeći BcryptJS.
    - Sigurna pohrana osjetljivih podataka.
    - Koristiti standarde sigurnosti

3.1 Struktura direktorija

Projekt je podijeljen u 2 dijela: Frontend (client) i Backend (api). Krećemo prvo s opisom Frontenda. React je JavaScript library za izradu korisničkih sučelja. Omogućava izradu aplikacije koristeći komponente. JSX je JavaScript sintaksa koja omogućava pisanje HTML-a unutar JavaScript koda. TailwindCSS je Utility-first CSS framework za brzo i jednostavno stiliziranje aplikacije. To su glavni alati s kojima izrađujemo klijentski dio. Glavni cilj je da ta dva spomenuta dijela funkcioniraju zajedno. Zato ih i odvajamo u 2 posebna direktorija kako bi dobili smisao i vidljivu podjelu između ta dva dijela [2].

3.1.1 Struktura direktorija za frontend

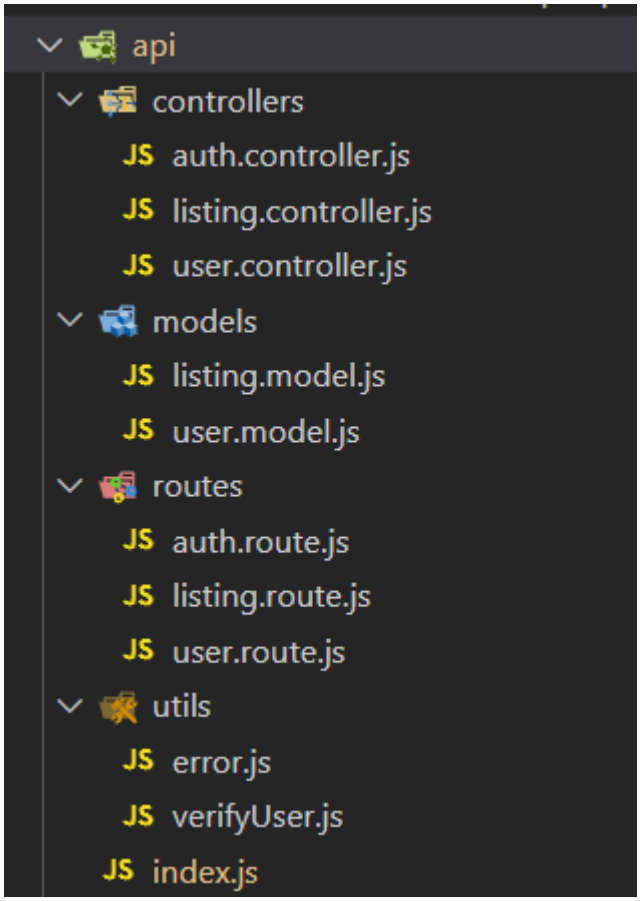
Funkcionalnost frontenda postavljamo na smisleni način, krenuvši od strukture direktorija. U client folderu node_modules sadrži sve potrebne npm pakete. Src folder je glavni folder koji će sadržavati izvorni kod. Components folder sadrži reusable komponente. Header.jsx je glavni primjer toga, jer tu komponentu koristimo preko cijele aplikacije kako bi smo imali dostupan header i na taj način iskorištavamo jednu komponentu više puta. Pages sadržava sve glavne stranice naše aplikacije. Svaka od njih odrađuje svoju funkciju kako i samo ime kaže. Slijedi zatim redux folder koji upravlja stanjem na frontend dijelu aplikacije. Redux se koristi za centralizirano upravljanje stanjem aplikacije te omogućava lako dijeljenje podataka između različitih dijelova aplikacije [3].



Slika 2. Prikaz strukture frontend direktorija

3.1.2 Struktura direktorija za backend

Backend, odnosno api, nam je „mozak“ cijelog projekta. U njemu se nalazi logika, koja povezuje i obrađuje informacije s frontenda. Controllers folder sadrži logiku za rukovanje HTTP zahtjevima. Models folder jednostvano predstavljaja strukturu podataka u bazi podataka. Routes povezuju URL putanje s odgovarajućim kontroler metodama, te na kraju imamo i utils (skraćeno od utilities). To je folder za pomoćne funkcije i module, poput provjere token za našeg korisnika, kako bi potvrdili je li on autoriziran za određene dijelove aplikacije [3].



Slika 3 Prikaz strukture backend direktorija

3.2. Implementacija backend-a

U slijedećem poglavlju detaljno će biti objašnjen kod za backend. Objasnit ćemo glavne dijelove koji tvore aplikaciju. Backend aplikacije služi kao posrednik između korisničkog sučelja (frontenda) i baze podataka. On upravlja podacima, provjerava valjanost podataka, autentificira korisnike te omogućava komunikaciju s bazom podataka[5].

3.2.1 Models

Modeli definiraju strukturu podataka u bazi podataka. Prvo uvozimo biblioteku „mongoose“, odnosno poveznicu za rad s MongoDB bazom. Definiramo shemu „listingSchema“ i u njoj se strukturirano nalaze slijedeća polja: name, description, address, city, genre, slot, ageOver18, time, imageUrls i korisnikRef. Nakon definiranja sheme, koristimo *mongoose.model* za stvaranje modela "Listing" koji će predstavljati dokumente u MongoDB bazi podataka prema definiranoj shemi. Na kraju izvozimo taj model kako bismo ga mogli koristiti u drugim dijelovima naše aplikacije.

```
import mongoose from "mongoose";

const listingSchema = new mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
    },
    description: {
      type: String,
      required: true,
    },
    address: {
      type: String,
      required: true,
    },
    city: {
      type: String,
      required: true,
    },
    genre: {
      type: String,
      required: true,
    },
```

Slika 4 Prikaz modela za događaje

```
slot: {
      type: Number,
      required: true,
    },
    ageOver18: {
      type: Boolean,
      required: true,
    },
    time: {
      type: Date,
      required: true,
    },
    imageUrls: {
      type: Array,
      required: true,
    },
    korisnikRef: {
      type: String,
      required: true,
    },
  },
  { timestamps: true }
);

const Listing = mongoose.model("Listing", listingSchema);
export default Listing;
```

Slika 5 Prikaz modela za događaje

Sljedeći model je model za korisnika. Ovaj kod koristi Mongoose za rad s MongoDB bazom podataka, isto kao i za prethodni model. Prvo se uvozi Mongoose biblioteka. Zatim se definira shema korisnika koja sadrži polja za korisničko ime, email, lozinku i avatar. Polja za korisničko ime i email su obavezna i moraju biti jedinstvena, dok je lozinka također obavezna. Polje za avatar je opcionalno. U shemu je uključeno i automatsko dodavanje vremenskih oznaka. Na temelju ove sheme kreira se model pod nazivom "Korisnik". Na kraju, model se izvozi kako bi se mogao koristiti u drugim dijelovima aplikacije.

4

```
import mongoose from "mongoose";

const korisnikSchema = new mongoose.Schema(
  {
    korisnikname: { type: String, required: true, unique: true },
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    avatar: {
      type: String,
    },
  },
  { timestamps: true }
);

const Korisnik = mongoose.model("Korisnik", korisnikSchema);

export default Korisnik;
```

Slika 6 Prikaz modela za korisnika

3.2.2. Controllers

Kontroleri su ključni dio backenda aplikacije i služe kao spona između modela (podataka) i ruta (kako se podaci pristupaju). Oni obrađuju dolazne zahtjeve, vrše potrebne operacije na podacima i vraćaju odgovore klijentima. Započinjemo s kontrolerom za korisnika. Ovaj isječak koda prikazuje osnovnu strukturu kontrolera u Node.js aplikaciji. Uvoze se potrebni moduli i modeli, te se definira jednostavna funkcija `test` koja vraća JSON odgovor klijentu. Funkcija `test` može se koristiti za provjeru radi li određena API ruta ispravno, što je korisno za inicijalno testiranje postavki API-a.

```
import bcryptjs from "bcryptjs";
import Korisnik from "../models/korisnik.model.js";
import { errorHandler } from "../utils/error.js";
import Listing from "../models/listing.model.js";

export const test = (req, res) => {
  res.json({
    message: "Api route is working!",
  });
};
```

Slika 7 Priprema kontrolera za korisnika

Ovaj kod definira asinkronu funkciju `updateKorisnik` koja ažurira korisničke podatke. Provjerava da li korisnik koji šalje zahtjev ima isti ID kao onaj u URL parametrima. Ako korisnik pokušava ažurirati lozinku, hashira novu lozinku koristeći `bcryptjs`. Koristi `Korisnik.findByIdAndUpdate` za ažuriranje korisničkih podataka u bazi podataka. Ažurirane korisničke podatke, osim lozinke, vraća kao odgovor klijentu. Ako dođe do greške, prosljeđuje grešku idućem middleware-u.

```
export const updateKorisnik = async (req, res, next) => {
  if (req.korisnik.id !== req.params.id)
    return next(errorHandler(401, "You can only update your own account!"));
  try {
    if (req.body.password) {
      req.body.password = bcryptjs.hashSync(req.body.password, 10);
    }

    const updatedKorisnik = await Korisnik.findByIdAndUpdate(
      req.params.id,
      {
        $set: {
          korisnikname: req.body.korisnikname,
          email: req.body.email,
          password: req.body.password,
          avatar: req.body.avatar,
        },
      },
      { new: true }
    );

    const { password, ...rest } = updatedKorisnik._doc;

    res.status(200).json(rest);
  } catch (error) {
    next(error);
  }
};
```

Slika 8 Kontroler za ažuriranje korisnika

Ovaj kod definira dvije asinkrone funkcije za rukovanje korisničkim zahtjevima u API-ju.

Funkcija `getKorisnikListings` dohvaća sve oglase koje je kreirao korisnik. Provjerava da li korisnik koji šalje zahtjev ima isti ID kao onaj u URL parametrima. Ako je ID isti, pretražuje

oglase u bazi podataka pomoću `Listing.find` i vraća ih kao odgovor. Ako dođe do greške, prosljeđuje grešku idućem middleware-u.

Funkcija `deleteKorisnik` briše korisnički račun. Provjerava da li korisnik koji šalje zahtjev ima isti ID kao onaj u URL parametrima. Ako ID nije isti, vraća grešku. Ako je ID isti, koristi `Korisnik.findByIdAndDelete` za brisanje korisničkog računa iz baze podataka, briše kolačić "access_token" i vraća odgovor o uspješnom brisanju računa. Ako dođe do greške, prosljeđuje grešku idućem middleware-u.

```
export const getKorisnikListings = async (req, res, next) => {
  if (req.korisnik.id === req.params.id) {
    try {
      const listings = await Listing.find({ korisnikRef: req.params.id });
      res.status(200).json(listings);
    } catch (error) {
      next(error);
    }
  } else {
    return next(errorHandler(401, "You can only view your own listings!"));
  }
};

export const deleteKorisnik = async (req, res, next) => {
  if (req.korisnik.id !== req.params.id)
    return next(errorHandler(401, "You can delete only your own account!"));

  try {
    await Korisnik.findByIdAndDelete(req.params.id);
    res.clearCookie("access_token");
    res
      .status(200)
      .json("Account has been deleted...")
      .clearCookie("access_token");
  } catch (error) {
    next(error);
  }
};
```

Slika 9 Kontroler za dohvaćanje korisnikovih događaja i brisanje korisnika

Sljedeći isječak, prikazuje asinkroni pristup za dohvaćanje korisnika iz baze podataka koristeći ID koji je dostupan u zahtjevu. Ako korisnik nije pronađen, šalje se odgovor s greškom statusa 404, nama dobro poznata error greška i uz to prikladna poruka: "Korisnik not found!". Ako je korisnik pronađen, izvlače se podaci iz korisničkog objekta. Password se izdvaja u zasebnu varijablu (pass), a ostali podaci se šalju kao odgovor s statusom 200 u obliku JSON objekta. U slučaju bilo kakve greške prilikom izvršavanja, greška se prosljeđuje dalje za obradu. Potrebno je odgovoriti na pitanje zašto smo koristili asinkroni pristup? Asinkroni pristup se koristi jer funkcija getKorisnik očekuje da se korisnik dohvati iz baze podataka. Dohvaćanje podataka iz baze podataka obično zahtijeva vremenski zahtjevne operacije, poput slanja upita prema bazi i čekanja na odgovor. Korištenjem asinkronog pristupa, aplikacija može nastaviti s izvršavanjem drugih operacija dok se podaci ne dohvate iz baze. To omogućuje bolju iskorištenost resursa i smanjuje blokiranje aplikacije, posebno u situacijama kada je potrebno čekati na vanjske resurse poput baze podataka.

```
export const getKorisnik = async (req, res, next) => {
  try {
    const korisnik = await Korisnik.findById(req.params.id);

    if (!korisnik) return next(errorHandler(404, "Korisnik not found!"));

    const { password: pass, ...rest } = korisnik._doc;

    res.status(200).json(rest);
  } catch (error) {
    next(error);
  }
};
```

Slika 10 Kontroler za dohvaćanje korisnika

Prelazimo sada na kontrolera za naše događaje. Česti naziv će biti „listing“. Naziv „listing“ se često koristi kao općeniti termin za bilo koji popis stavki, uključujući i događaje ili unose o događajima. Iako naziv "listing" može sugerirati da se radi samo o prodaji proizvoda ili usluga, u stvarnosti se može koristiti za različite vrste popisa, uključujući i popise događaja.

Funkcija createListing koristi asinkroni pristup za stvaranje novog unosa oglasa. Pokušava stvoriti novi unos oglasa koristeći podatke dobivene iz zahtjeva. Nakon što se unos oglasa stvori uspješno, vraća se odgovor s statusom 201 i JSON objektom koji predstavlja novi unos oglasa. U slučaju greške, greška se prosljeđuje dalje za obradu.

Funkcija deleteListing koristi asinkroni pristup za brisanje unosa oglasa. Prvo dohvaća unos oglasa iz baze podataka koristeći ID dobiven iz zahtjeva. Provjerava se je li korisnik koji šalje zahtjev vlasnik tog unosa oglasa. Ako nije, šalje se odgovor s greškom statusa 403 "Forbidden - not your event". U slučaju da je korisnik vlasnik, nastavlja se s brisanjem unosa oglasa. Nakon brisanja, šalje se odgovor s statusom 200 i porukom "Event deleted". U slučaju greške, greška se prosljeđuje dalje za obradu.

```
export const createListing = async (req, res, next) => {
  try {
    const listing = await Listing.create(req.body);
    res.status(201).json(listing);
  } catch (error) {
    next(error);
  }
};

export const deleteListing = async (req, res, next) => {
  const listing = await Listing.findById(req.params.id);

  if (req.korisnik.id !== listing.korisnikRef) {
    return next(errorHandler(403, "Forbidden - not your event"));
  }

  try {
    await Listing.findByIdAndDelete(req.params.id);
    res.status(200).json("Event deleted");
  } catch (error) {}

  next(error);
};
```

Slika 11 Kontroler za stvaranje događaja

Funkcija `updateListing` provjerava postoji li unos događaja s ID-om koji se nalazi u zahtjevu. Ako ne postoji, vraća se odgovor s greškom statusa 404 "Event not found!". Zatim provjerava je li trenutni korisnik vlasnik tog događaja. Ako nije, vraća se odgovor s greškom statusa 401 "That is not your event!!". Ako je korisnik vlasnik događaja, ažurira se taj događaj s podacima dobivenim iz zahtjeva i šalje se odgovor s ažuriranim događajem.

Funkcija `getListing` jednostavno dohvaća događaj s ID-om koji se nalazi u zahtjevu. Ako događaj ne postoji, vraća se odgovor s greškom statusa 404 "Event not found!". Ako događaj postoji, šalje se odgovor s podacima o tom događaju.

```
export const updateListing = async (req, res, next) => {
  const listing = await Listing.findById(req.params.id);
  if (!listing) {
    return next(errorHandler(404, "Event not found!"));
  }
  if (req.korisnik.id !== listing.korisnikRef) {
    return next(errorHandler(401, "That is not your event!!"));
  }

  try {
    const updatedListing = await Listing.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true }
    );
    res.status(200).json(updatedListing);
  } catch (error) {
    next(error);
  }
};

export const getListing = async (req, res, next) => {
  try {
    const listing = await Listing.findById(req.params.id);
    if (!listing) {
      return next(errorHandler(404, "Event not found!"));
    }
    res.status(200).json(listing);
  } catch (error) {
    next(error);
  }
};
```

Slika 12 Kontroler za ažuriranje i dohvaćanje događaja

Funkcija `getListing` se koristi za dohvaćanje pojedinačnog događaja (eventa) koristeći ID događaja koji se nalazi u zahtjevu. Vraća se samo taj jedan događaj ako postoji. Tu dolazi slična funkcija koja će sada biti prikazana.

Funkcija `getListings`, s druge strane, koristi se za dohvaćanje liste događaja (events), uz mogućnost filtriranja, sortiranja, paginacije i pretraživanja. Omogućuje više fleksibilnosti u dohvatu podataka, jer korisnik može specificirati različite parametre za filtriranje i sortiranje, kao što su limit, startIndex, searchTerm, sort, order, ageOver18, i slot. Ova funkcija vraća popis događaja koji odgovaraju zadanim kriterijima pretrage.

```
export const getListings = async (req, res, next) => {
  try {
    const limit = parseInt(req.query.limit) || 0;
    const startIndex = parseInt(req.query.startIndex) || 0;
    const searchTerm = req.query.searchTerm || "";
    const sort = req.query.sort || "createdAt";
    const order = req.query.order || "desc";

    const filter = {
      $or: [
        { name: { $regex: searchTerm, $options: "i" } },
        { description: { $regex: searchTerm, $options: "i" } },
        { city: { $regex: searchTerm, $options: "i" } },
        { genre: { $regex: searchTerm, $options: "i" } },
      ],
    };

    if (req.query.ageOver18) {
      filter.ageOver18 = req.query.ageOver18 === "true";
    }

    if (req.query.slot) {
      filter.slot = parseInt(req.query.slot);
    }

    const listings = await Listing.find(filter)
      .sort({ [sort]: order })
      .limit(limit)
      .skip(startIndex);

    return res.status(200).json(listings);
  } catch (error) {
    next(error);
  }
};
```

Slika 13 Kontroler za dohvaćanje događaja iz filtera

Slijedi zatim kontroler za autentifikaciju svakog korisnika. Prvo moramo pripremiti kontroler s potrebnim importima. Ovaj dio koda implementira funkciju za registraciju novog korisnika (signup). Iz zahtjeva se izvlače korisničko ime (korisnikname), email i lozinka (password). Lozinka se hashira korištenjem bcryptjs biblioteke. Stvara se novi korisnik koristeći Korisnik model i hashiranu lozinku. Novi korisnik se sprema u bazu podataka pozivom funkcije save(). Ako registracija prođe uspješno, vraća se odgovor sa statusom 201 i porukom o uspješnoj registraciji. U slučaju greške, greška se prosljeđuje dalje za obradu.

```
import Korisnik from "../models/korisnik.model.js";
import bcrypt from "bcryptjs";
import { errorHandler } from "../utils/error.js";
import jwt from "jsonwebtoken";

export const signup = async (req, res, next) => {
  const { korisnikname, email, password } = req.body;
  const hashedPassword = bcrypt.hashSync(password, 10);
  const newKorisnik = new Korisnik({ korisnikname, email, password:
hashedPassword });
  try {
    await newKorisnik.save();
    res.status(201).json({ message: "Korisnik uspjesno stvoren" });
  } catch (error) {
    next(error);
  }
};
```

Slika 14 Priprema kontrolera i prijava korisnika

Funkcija `signin` prima zahtjev s korisničkim emailom i lozinkom. Provjerava postoji li korisnik s tim emailom u bazi podataka. Ako korisnik ne postoji, vraća grešku s porukom "Korisnik ne postoji". Ako korisnik postoji, uspoređuje unesenu lozinku s hashiranom lozinkom korisnika. Ako lozinke nisu jednake, vraća grešku s porukom "Wrong credentials". Ako su email i lozinka ispravni, generira JSON Web Token (JWT) s ID-om korisnika. Postavlja JWT kao cookie u odgovoru, čime korisnik dobiva autorizaciju. Na kraju, šalje odgovor s podacima korisnika bez lozinke.

Funkcija `signout` briše postavljeni cookie za pristupni token. Ovime se označava korisnika kao odjavljenog. Šalje odgovor sa statusom 200 i porukom "Korisnik is signed out".

Na ovaj način prikazujemo kako se radi s osjetljivim podacima, i na koji način ih sigurno obrisati iz trenutne sesije

```
export const signin = async (req, res, next) => {
  const { email, password } = req.body;
  try {
    const validKorisnik = await Korisnik.findOne({ email });
    if (!validKorisnik) return next(errorHandler(404, "Korisnik ne postoji"));
    const validPassword = bcrypt.compareSync(password, validKorisnik.password);

    if (!validPassword) return next(errorHandler(401, "Wrong credentials"));
    const token = jwt.sign({ id: validKorisnik._id }, process.env.JWT_SECRET,
{});
    const { password: pass, ...rest } = validKorisnik._doc;
    res
      .cookie("access_token", token, {
        httpOnly: true,
      })
      .status(200)
      .json(rest);
  } catch (error) {
    next(error);
  }
};

export const signout = (req, res) => {
  try {
    res.clearCookie("access_token");
    res.status(200).json("Korisnik is signed out");
  } catch (error) {
    next(error);
  }
};
```

Slika 15 Kontroler za prijavu i odjavu korisnika

3.2.3. Routes

Slijedi važan korak, a to su rute za autentikaciju korisnika koristeći Express. Rute u backendu definiraju putanje kojima klijenti (npr. web preglednici, mobilne aplikacije) pristupaju određenim resursima ili funkcionalnostima u aplikaciji putem HTTP zahtjeva. Svaka ruta definira odgovarajuću akciju koja se izvršava kada se taj zahtjev primi. Uvoze se potrebne funkcije za prijavu, odjavu i registraciju iz odgovarajuće kontrolerske datoteke. Stvara se novi

Router objekt. Definiraju se rute za registraciju, prijavu i odjavu korisnika. Na kraju se izvozi router objekt za korištenje u drugim dijelovima aplikacije.

```
import express from "express";
import { signin, signout, signup } from "../controllers/auth.controller.js";

const router = express.Router();

router.post("/signup", signup);
router.post("/signin", signin);
router.get("/signout", signout);

export default router;
```

Slika 16 Rute za registraciju, prijavu i odjavu

Ovaj dio koda definira rute za manipulaciju događajima (listings) u aplikaciji koristeći Express Router. Uvoze se funkcije i module za stvaranje, brisanje, ažuriranje, dohvaćanje pojedinačnog i dohvaćanje svih događaja iz odgovarajućih kontrolerskih datoteka. Također se uvozi funkcija za provjeru valjanosti JWT tokena iz utils datoteke. Definirane su rute za svaku operaciju: stvaranje, brisanje, ažuriranje, dohvaćanje pojedinačnog i dohvaćanje svih događaja. Rute koje mijenjaju stanje podataka (stvaranje, brisanje, ažuriranje) zahtijevaju provjeru valjanosti JWT tokena kroz `verifyToken` funkciju. Na kraju, router objekt je izvezen za upotrebu u drugim dijelovima aplikacije.

```
import express from "express";
import {
  createListing,
  deleteListing,
  updateListing,
  getListing,
  getListings,
} from "../controllers/listing.controller.js";
import { verifyToken } from "../utils/verifyKorisnik.js";

const router = express.Router();
router.post("/create", verifyToken, createListing);
router.delete("/delete/:id", verifyToken, deleteListing);
```

```
router.post("/update/:id", verifyToken, updateListing);
router.get("/get/:id", getListing);
router.get("/get", getListings);

export default router;
```

Slika 17 Rute za manipulaciju događajima

Prelazimo sada na rute koje su vezane za naše korisnike. Uvoze se potrebne funkcije i module za brisanje korisnika, testiranje rute, ažuriranje korisnika, dohvaćanje popisa događaja korisnika i dohvaćanje pojedinačnog korisnika iz odgovarajućih kontrolerskih datoteka. Također se uvozi funkcija za provjeru valjanosti JWT tokena iz utils datoteke. Definirane su rute za svaku operaciju: testiranje rute, ažuriranje korisnika, brisanje korisnika, dohvaćanje popisa događaja korisnika i dohvaćanje pojedinačnog korisnika. Rute koje mijenjaju stanje podataka (ažuriranje, brisanje, dohvaćanje popisa događaja, dohvaćanje pojedinačnog korisnika) zahtijevaju provjeru valjanosti JWT tokena kroz `verifyToken` funkciju. Na kraju, router objekt je izvezen za upotrebu u drugim dijelovima aplikacije.

```
import express from "express";
import {
  deleteKorisnik,
  test,
  updateKorisnik,
  getKorisnikListings,
  getKorisnik,
} from "../controllers/korisnik.controller.js";
import { verifyToken } from "../utils/verifyKorisnik.js";

const router = express.Router();

router.get("/test", test);
router.post("/update/:id", verifyToken, updateKorisnik);
router.delete("/delete/:id", verifyToken, deleteKorisnik);
router.get("/listings/:id", verifyToken, getKorisnikListings);
router.get("/:id", verifyToken, getKorisnik);

export default router;
```

Slika 18 Rute za korisnika

3.2.4. Utils

Slijedi posljednji folder u nasoj strukturi backenda, a to je utils. Započinjemo prvo s handlerom za errore. Ova funkcija `errorHandler` stvara i vraća novi objekt greške s definiranim

statusom i porukom. Prima dva parametra: `statusCode`, koji označava statusni kod HTTP odgovora, i `message`, koji opisuje grešku. Nakon što se stvori objekt greške, postavljaju se statusni kod i poruka, te se objekt greške vraća kao rezultat funkcije.

```
export const errorHandler = (statusCode, message) => {
  const error = new Error(message);
  error.statusCode = statusCode;
  error.message = message;
  return error;
};
```

Slika 19 Error handler

Ova funkcija `verifyToken` provjerava valjanost JWT tokena koji se nalazi u kolačićima zahtjeva. Prvo se dohvaća token iz kolačića `access_token` u zahtjevu. Ako token ne postoji, funkcija vraća grešku sa statusom 401 "Unauthorized" pomoću funkcije `errorHandler`. Zatim se koristi `jwt.verify` funkcija kako bi se provjerila valjanost tokena korištenjem tajnog ključa definiranog u varijabli okoline `process.env.JWT_SECRET`. Ako se pojavi greška prilikom provjere tokena, funkcija vraća grešku sa statusom 403 "Forbidden" pomoću funkcije `errorHandler`. Ako je token valjan, podaci o korisniku iz tokena se dodaju u zahtjev (`req.korisnik`) i poziva se sljedeća funkcija u lancu middleware-a.

```
import jwt from "jsonwebtoken";
import { errorHandler } from "./error.js";

export const verifyToken = (req, res, next) => {
  const token = req.cookies.access_token;

  if (!token) return next(errorHandler(401, "Unauthorized"));

  jwt.verify(token, process.env.JWT_SECRET, (err, korisnik) => {
    if (err) return next(errorHandler(403, "Forbidden"));

    req.korisnik = korisnik;
    next();
  });
};
```

Slika 20 Provjera tokena

Posljednja stavka je index.js. Datoteka index.js obično služi kao glavna datoteka ili "ulazna točka" aplikacije. Ovaj blok koda uvozi potrebne module i konfigurira Express aplikaciju za rad s bazom podataka MongoDB, kolačićima i rutama za upravljanje korisnicima, autentikacijom i događajima. Na kraju, definira globalni error handler za obradu grešaka.

```
import express from "express";
import mongoose from "mongoose";
import listingRouter from "./routes/listing.route.js";
import dotenv from "dotenv";
import korisnikRouter from "./routes/korisnik.route.js";
import authRouter from "./routes/auth.route.js";
import cookieParser from "cookie-parser";

dotenv.config();
mongoose
  .connect(process.env.MONGO)
  .then(() => {
    console.log("Uspješno spojeno na MongoDB!");
  })
  .catch((err) => {
    console.log("Error spajanja na MongoDB " + err);
  });

const app = express();

app.listen(8888, () => {
  console.log("Server radi na portu 8888!");
});

app.use(express.json());

app.use(cookieParser());
app.use("/api/korisnik", korisnikRouter);

app.use("/api/auth", authRouter);

app.use("/api/listing", listingRouter);

app.use((err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  const message = err.message || "Server error";
  return res.status(statusCode).json({
    success: false,
    statusCode,
```

```
    message,
  });
});
```

Slika 21 Index.js

3.3 Implementacija frontend-a

Frontend u aplikacijama služi za prikaz korisničkog sučelja i omogućava interakciju korisnika s aplikacijom. Frontend je odgovoran za izgled i dojam aplikacije, uključujući raspored elemenata, stilove, animacije i interaktivne funkcionalnosti. Omogućava korisnicima unos podataka, navigaciju kroz različite dijelove aplikacije i izvođenje akcija kao što su slanje obrazaca ili klikanje na gumbe. Frontend također komunicira s backendom kako bi dohvaćao i prikazivao podatke te slanje korisničkih zahtjeva za obradu i pohranu podataka. Prvo ćemo započeti s app.jsx, te ćemo prijeći na glavne foldere koji su dio frontenda. Sve je strukturirano tako da kroz ime datoteke, možemo prepoznati njezinu funkciju. Glavni alati su React i TailwindCSS.

3.3.1. Root datoteke client foldera

Ovaj kod definira glavnu aplikaciju koristeći React Router za upravljanje rutama. Unutar BrowserRouter komponenta omotava cijelu aplikaciju, omogućavajući navigaciju između različitih stranica. Na vrhu svake stranice prikazuje se komponenta Header. Unutar Routes komponenta definira različite rute za aplikaciju.

Rute uključuju Home, SignIn, SignUp, Search, Listing, i About, koje su dostupne svim korisnicima. Postoje i rute unutar PrivateRoute komponente koje su dostupne samo autentificiranim korisnicima, kao što su Profile, CreateListing i UpdateListing.

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Home from "./pages/Home";
import SignIn from "./pages/SignIn";
import SignUp from "./pages/SignUp";
import About from "./pages/About";
import Profile from "./pages/Profile";
import Header from "./components/Header";
import PrivateRoute from "./components/PrivateRoute";
import CreateListing from "./pages/CreateListing";
import UpdateListing from "./pages/UpdateListing";
import Listing from "./pages/Listing";
```

```
import Search from "./pages/Search";

export default function App() {
  return (
    <BrowserRouter>
      <Header />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/sign-in" element={<SignIn />} />
        <Route path="/sign-up" element={<SignUp />} />
        <Route path="/search" element={<Search />} />
        <Route path="/listing/:listingId" element={<Listing />} />
        <Route path="/about" element={<About />} />
        <Route element={<PrivateRoute />}>
          <Route path="/profile" element={<Profile />} />
          <Route path="/create-listing" element={<CreateListing />} />
          <Route
            path="/update-listing/:listingId"
            element={<UpdateListing />}
          />
        </Route>
      </Routes>
```

Slika 22 App.jsx

3.3.2. Components

Ovaj kod definira React komponentu `DefaultAvatar` koja prikazuje korisničku ikonu centriranu u div elementu i stiliziranu s Tailwind CSS klasama. Komponenta koristi ikonu iz `react-icons` paketa i eksportira se kao zadana komponenta.

```
import React from "react";
import { FaKorisnik } from "react-icons/fa";

const DefaultAvatar = () => {
  return (
    <div className="flex justify-center items-center">
      <FaKorisnik className="text-orange-500 text-3xl" />
    </div>
  );
};

export default DefaultAvatar;
```

Slika 23 Default avatar komponenta

Slijedeći kod definira React komponentu `EventItem` koja prikazuje informacije o događaju u kartici. Komponenta prikazuje sliku događaja, naziv, adresu, opis, grad i starosnu dobnu

ograničenost. Svaki događaj je povezan linkom na svoju stranicu detalja koristeći React Router. Komponenta koristi ikone iz `react-icons` paketa za lokaciju, grad i starosnu dobnu ograničenost te je stilizirana s Tailwind CSS klasama. Tu dolazi moć Tailwind-a, s obzirom na to da smo jako brzo generirali css, i na taj način skratili trajanje razvoja naše aplikacije.

```
import { Link } from "react-router-dom";

import { MdLocationOn } from "react-icons/md";
import { FaCity } from "react-icons/fa";
import { HiOutlineIdentification } from "react-icons/hi";

export default function EventItem({ event }) {
  console.log("Event added and displayed:", event);
  return (
    <div className="bg-white shadow-md hover:shadow-lg transition-shadow
overflow-hidden rounded-lg w-full sm:w-[320px]">
      <Link to={`/listing/${event._id}`} className="block">
        <img
          src={event.imageUrls[0]}
          alt="Event cover image"
          className="h-[329px] w-full object-cover hover:scale-105 transition-
transform duration-300"
        />
        <div className="p-3">
          <p className="text-lg font-semibold text-gray-900 ">{event.name}</p>
          <div className="flex items-center gap-1 text-gray-600">
            <MdLocationOn className="h-4 w-4 text-green-700" />
            <p className="text-sm truncate">{event.address}</p>
          </div>
          <p className="text-sm text-gray-600 truncate">{event.description}</p>
          <div className="flex justify-between items-center mt-2">
            <div className="flex items-center text-sm text-gray-600">
              <FaCity className="h-4 w-4 mr-1" />
              <p>{event.city}</p>
            </div>
            <div className="flex items-center text-sm">
              <HiOutlineIdentification className="h-4 w-4 mr-1" />
              <p>{event.ageOver18 ? "18+" : "All ages"}</p>
            </div>
          </div>
        </div>
      </Link>
    </div>
  );
}
```

Slika 24 EventItem komponenta

Ovaj kod definira React komponentu `Header` koja prikazuje zaglavlje aplikacije s funkcionalnošću pretraživanja. Komponenta koristi ikone iz `react-icons` paketa, kao i React Router za navigaciju i Redux za dohvaćanje trenutnog korisnika.

Koristi `useState` za upravljanje stanjem pretraživačkog pojma i `useNavigate` za navigaciju na stranicu pretrage. Funkcija `handleSubmit` obrađuje podnošenje forme za pretraživanje i koristi `navigate` za preusmjeravanje na stranicu rezultata pretrage s odgovarajućim query parametrima.

`useEffect` se koristi za postavljanje početnog stanja pretraživačkog pojma iz URL-a kad se komponenta učita. Zatim, slijedi i stiliziranje komponente. Bitno je da je UX, odnosno korisničko iskustvo dobro. Zato koristimo jasne ikone, i dizajn koji nije previše upadan, kako bi se fokusirali na brzinu i glavnu stvar naše aplikacije, a to je funkcionalnost.

```
import { FaSearch, FaKorisnik } from "react-icons/fa";
import { GiTabletopPlayers } from "react-icons/gi";
import { Link, useNavigate } from "react-router-dom";
import { useSelector } from "react-redux";
import { useEffect, useState } from "react";
import DefaultAvatar from "../components/DefaultAvatar";

export default function Header() {
  const { currentKorisnik } = useSelector((state) => state.korisnik);
  const [searchTerm, setSearchTerm] = useState("");
  const navigate = useNavigate();

  const handleSubmit = (e) => {
    e.preventDefault();
    const urlParams = new URLSearchParams();
    urlParams.set("searchTerm", searchTerm);
    const searchQuery = urlParams.toString();
    navigate(`/search?${searchQuery}`);
  };

  useEffect(() => {
    const urlParams = new URLSearchParams(window.location.search);
    const searchTermFromUrl = urlParams.get("searchTerm");
    if (searchTermFromUrl) {
      setSearchTerm(searchTermFromUrl);
    }
  }, []);
```

Slika 25 Header komponenta

```
return (
    <header className="bg-blue-900 text-gray-200 shadow-md">
      <div className="flex p-4 justify-between items-center max-w-6xl mx-auto">
        <Link to="/" className="flex items-center gap-2">
          <GiTabletopPlayers className="text-3xl text-orange-500" />
          <h1 className="font-light text-2xl">GameNightScout</h1>
        </Link>

        <form
          onSubmit={handleSubmit}
          className="flex items-center bg-blue-800 p-2 rounded-lg"
        >
          <input
            type="text"
            placeholder="Search events..."
            className="bg-transparent focus:outline-none text-white placeholder-
gray-300 pl-4 pr-2 py-1 w-48"
            value={searchTerm}
            onChange={(e) => setSearchTerm(e.target.value)}
          />
          <button>
            <FaSearch className="text-orange-500" />
          </button>
        </form>
```

Slika 26 Stiliziranje Header komponente

Ovaj kod definira React komponentu `PrivateRoute` koja štiti rute koje zahtijevaju autentifikaciju. Komponenta koristi `useSelector` za dohvaćanje trenutnog korisnika (`currentKorisnik`) iz Redux stanja. Ako je korisnik prijavljen (`currentKorisnik` postoji), prikazuje se `Outlet`, što omogućava prikaz zaštićenih komponenti unutar ruta. Ako korisnik nije prijavljen, komponenta preusmjerava korisnika na stranicu za prijavu (`/sign-in`) koristeći `Navigate`. Opet demonstriramo kako smo pokrili slučajeve gdje korisnik može doći do dijela aplikacije koje ne bi smio vidjeti

```
import { useSelector } from "react-redux";
import { Outlet, Navigate } from "react-router-dom";
```

```
export default function PrivateRoute() {
  const { currentKorisnik } = useSelector((state) => state.korisnik);
  return currentKorisnik ? <Outlet /> : <Navigate to="/sign-in" />;
}
```

Slika 27 PrivateRoute komponenta

3.3.3. Pages

Stranice u web razvoju obično služe za organizaciju različitih dijelova aplikacije prema njihovim funkcionalnostima. Svaka stranica predstavlja jedan "rute" ili pristup korisničkom sadržaju, što olakšava navigaciju i upravljanje aplikacijom.

Struktura `pages` foldera je često hijerarhijska i odražava strukturu rute aplikacije. Na primjer, može sadržavati stranice poput "Home", "About", "Profile", "SignIn", "SignUp" i tako dalje, prema funkcionalnostima koje aplikacija pruža.

Osim organizacijske svrhe, stranice su također ključne za pravilno funkcioniranje React Router-a ili drugih sličnih biblioteka za upravljanje rutama. Svaka stranica obično ima svoj vlastiti URL put koji se povezuje s odgovarajućom komponentom, omogućavajući navigaciju kroz aplikaciju.

Ovaj kod definira React stranicu `About` koja prikazuje generalne informacije za našu aplikaciju. Komponenta sadrži naslov, opis i kontakt informacije. Stilizirana je pomoću Tailwind CSS klasa i uključuje ikonu iz `react-icons` paketa za vizualni element. Opisuje svrhu aplikacije i nudi korisnicima mogućnost sudjelovanja u tabletop događajima te kontaktiranja administratora u slučaju problema ili prijedloga.

```
import React from "react";
import { GiTabletopPlayers } from "react-icons/gi";

export default function About() {
  return (
    <div className="flex flex-col items-center justify-center h-screen bg-
gradient-to-r from-blue-400 to-purple-500 text-white">
      <div className="flex items-center gap-2 mb-8">
        <GiTabletopPlayers className="text-3xl text-orange-500" />
        <h1 className="text-4xl md:text-6xl font-bold text-center">
          About GameNightScout
        </h1>
        <GiTabletopPlayers className="text-3xl text-orange-500" />
      </div>
      <p className="text-lg md:text-xl mb-8 text-center max-w-lg">
        GameNightScout is your go-to platform for discovering and participating
        in tabletop events happening in your area. Whether you're a board game
        enthusiast, a role-playing game aficionado, or a fan of card games,
        GameNightScout connects you with like-minded individuals and exciting
        gaming opportunities.
      </p>
      <p className="text-lg md:text-xl mb-8 text-center max-w-lg">
        With GameNightScout, you can easily search for events based on your
        interests. Create your own event and invite others to join, or join
        events hosted by fellow tabletop enthusiasts. From casual game nights to
        competitive tournaments, GameNightScout helps you find the perfect
        gaming experience tailored to your preferences.
      </p>
      <p className="text-sm text-center">
```

```
        Having problems? Got any suggestions? Contact admin:{" "}
        <a href="mailto:brkuvezdic@student.unip.hr" className="underline">
          brkuvezdic@student.unip.hr
        </a>
      </p>
    </div>
  );
```

Slika 28 About stranica

Stranica `Home` prikazuje početnu stranicu aplikacije "GameNightScout" s naslovom, opisom i linkom za pretraživanje događaja. Naslov dočekuje korisnike, opis potiče na pretraživanje događaja, dok link omogućuje korisnicima da brzo pređu na stranicu za pretraživanje.

```
import React from "react";
import { Link } from "react-router-dom";

export default function Home() {
  return (
    <div className="flex flex-col items-center justify-center h-screen bg-
gradient-to-r from-blue-400 to-purple-500 text-white">
      <h1 className="text-4xl md:text-6xl font-bold mb-8 text-center">
        Welcome to GameNightScout
      </h1>
      <p className="text-lg md:text-xl mb-8 text-center">
        Find the best tabletop events in your area. Sign up to create or join an
        event.
      </p>
      <Link
        to="/search"
        className="bg-white text-blue-600 font-semibold px-6 py-3 rounded-lg
shadow-lg hover:bg-blue-600 hover:text-white transition duration-300"
      >
        Browse
      </Link>
    </div>
  );
}
```

Slika 29 Home stranica

Stranica `SignIn` omogućuje korisnicima prijavu na aplikaciju, prateći promjene u formi i šaljući zahtjev za prijavu na backend. Koristi `useSelector` za dohvaćanje stanja autentifikacije, a `useDispatch` za dispečiranje Redux akcija za upravljanje procesom prijave. Nakon uspješne prijave, korisnika preusmjerava na početnu stranicu, a u slučaju greške prikazuje odgovarajuću poruku.

```
import { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import { useDispatch, useSelector } from "react-redux";
import {
  signInStart,
  signInSuccess,
  signInFailure,
} from "../redux/korisnik/korisnikSlice";

export default function SignIn() {
  const [formData, setFormData] = useState({});
  const { loading, error } = useSelector((state) => state.korisnik);
  const navigate = useNavigate();
  const dispatch = useDispatch();
  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.id]: e.target.value,
    });
  };
  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      dispatch(signInStart());
      const res = await fetch("/api/auth/signin", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify(formData),
      });
      const data = await res.json();
      console.log(data);
      if (data.success === false) {
        dispatch(signInFailure(data.message));
        return;
      }
```

```
      dispatch(signInSuccess(data));
      navigate("/");
    } catch (error) {
      dispatch(signInFailure(error.message));
    }
  };
```

Slika 30 SignIn stranica

Stranica `SignUp` omogućuje registraciju korisnika na aplikaciju, prateći promjene u formi i šaljući zahtjev za registraciju na backend. Koristi `useState` hook-a za praćenje forme, greške i stanja učitavanja. Nakon uspješne registracije, korisnika preusmjerava na stranicu za prijavu (`/sign-in`), a u slučaju greške prikazuje odgovarajuću poruku.

```
import { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
export default function SignUp() {
  const [formData, setFormData] = useState({});
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(false);
  const navigate = useNavigate();
  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.id]: e.target.value,
    });
  };
  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      setLoading(true);
      const res = await fetch("/api/auth/signup", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify(formData),
      });
      const data = await res.json();
      console.log(data);
      if (data.success === false) {
        setLoading(false);
        setError(data.message);
        return;
```

```
      }
      setLoading(false);
      setError(null);
      navigate("/sign-in");
    } catch (error) {
      setLoading(false);
```

Slika 31 SignUp stranica

Stranica `CreateListing` omogućuje stvaranje novih događaja korisnicima.Koristi `useState` hook za praćenje stanja datoteka, forme i URL-ova slika.Firebase Storage se koristi za stvaranje, pohranu i dohvat slika za događaje. `useSelector` se koristi za dohvaćanje trenutnog korisnika iz Redux store-a. `useNavigate` omogućuje navigaciju korisnika nakon stvaranja događaja. Podaci forme se ažuriraju kako korisnik unosi nove informacije za događaj.

```
import { useState } from "react";
import {
  getStorage,
  ref,
  uploadBytesResumable,
  getDownloadURL,
} from "firebase/storage";
import { app } from "../firebase";
import { useSelector } from "react-redux";
import { useNavigate } from "react-router-dom";

export default function CreateListing() {
  const { currentKorisnik } = useSelector((state) => state.korisnik);
  const [files, setFiles] = useState([]);
  const navigate = useNavigate();
  const [formData, setFormData] = useState({
    imageUrls: [],
    name: "",
    description: "",
    address: "",
    city: "",
    genre: "",
    time: "",
    slot: 6,
    ageOver18: false,
  });
```

Slika 32 CreateListing stranica

Stranica koristi nekoliko hookova za upravljanje stanjima tijekom procesa stvaranja događaja. `error` i `loading` stanja prate općenite greške i stanje učitavanja. Specifično za postupak učitavanja slika, koriste se `imageUploadError` i `uploading` stanja. Funkcija `handleImageSubmit` provjerava broj slika koje korisnik želi učitati te upravlja postupkom učitavanja slika. Ako korisnik pokuša učitati više od 3 slike ili ako pojedina slika premašuje 2 MB, prikazuje se odgovarajuća poruka o grešci. U suprotnom, slike se učitavaju i dodaju u formu događaja.

```
const [error, setError] = useState(false);
  const [loading, setLoading] = useState(false);
  const [imageUploadError, setImageUploadError] = useState(false);
  const [uploading, setUploading] = useState(false);
  console.log(formData);
  const [selectedGenre, setSelectedGenre] = useState("");
  const handleImageSubmit = () => {
    if (files.length > 0 && files.length + formData.imageUrls.length <= 3) {
      setUploading(true);
      setImageUploadError(false);
      const promises = [];

      for (let i = 0; i < files.length; i++) {
        promises.push(storeImage(files[i]));
      }
      Promise.all(promises)
        .then((urls) => {
          setFormData({
            ...formData,
            imageUrls: formData.imageUrls.concat(urls),
          });
          setImageUploadError(false);
          setUploading(false);
        })
        .catch((err) => {
          setImageUploadError("Image upload failed (2 MB max per image)");
          setUploading(false);
        });
    } else {
      setImageUploadError("You can only upload up to 3 images per event");
      setUploading(false);
    }
  };
```

Slika 33 Učitavanje slike

Funkcija `storeImage` koristi Firebase Storage za pohranu slika. Stvara se obećanje koje se izvršava nakon što se slika uspješno učita. Za svaku sliku dodaje se progresni indikator koji prati napredak učitavanja. Kada se učitavanje završi, dobavlja se URL slike i prosljeđuje kao rezultat obećanja.

`genreOptions` definira popis opcija žanrova događaja. `handleRemoveImage` funkcija omogućuje korisniku uklanjanje slika iz formulara događaja.

```
  const handleRemoveImage = (index) => {
    setFormData({
      ...formData,
      imageUrls: formData.imageUrls.filter((_, i) => i !== index),
    });
  };

  const handleGenreChange = (event) => {
    setSelectedGenre(event.target.value);
    setFormData({ ...formData, genre: event.target.value });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();

    try {
      if (formData.imageUrls.length === 0) {
        return setError("You need to upload at least one image");
      }

      setLoading(true);
      setError(false);
      const res = await fetch("/api/listing/create", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          ...formData,
          korisnikRef: currentKorisnik._id,
        }),
      });

      const data = await res.json();
      setLoading(false);
      if (data.success === false) {
        setError(data.message);
      }
```

```
      navigate(`/listing/${data._id}`);
    } catch (error) {
      setError(error.message);
      setLoading(false);
    }
  };
```

Slika 34 Promjene u formi

`Listing` stranica prikazuje detalje određenog događaja. Koristi `Swiper` komponentu za prikaz slika događaja. Podaci se dohvaćaju s API-ja kada se komponenta mounta ili promijeni ID događaja. Ako se pojavi greška, prikazuje se odgovarajuća poruka.

```
import { useEffect, useState } from "react";
import { useParams } from "react-router-dom";
import { Swiper, SwiperSlide } from "swiper/react";
import SwiperCore from "swiper";
import { Navigation } from "swiper/modules";
import "swiper/css/bundle";
import {
  FaMapMarkerAlt,
  FaCalendarAlt,
  FaKorisnikFriends,
  FaCity,
} from "react-icons/fa";
import { HiOutlineIdentification } from "react-icons/hi";
import { SiRiotgames } from "react-icons/si";

export default function Listing() {
  SwiperCore.use([Navigation]);
  const params = useParams();
  const [listing, setListing] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  const [creator, setCreator] = useState(null);

  useEffect(() => {
    async function fetchListing() {
      try {
        setLoading(true);
        const res = await fetch(`/api/listing/get/${params.listingId}`);
        const data = await res.json();
        if (data.success === false) {
          setError(true);
          setLoading(false);
          return;
        }
```

```
        if (data.time) {
          data.time = new Date(data.time).toLocaleString();
        }
        setListing(data);
        setLoading(false);
        setError(false);
      }
```

Slika 35 Promjene u formi

`useEffect` hook dohvaća detalje o događaju i informacije o kreatoru događaja s odgovarajućih API ruta kada se ID događaja promijeni. Nakon dohvaćanja podataka, formatira vremenske informacije i postavlja ih u stanje. Također postavlja stanje kreatora događaja. `handleGoogleMaps` funkcija otvara Google Maps s lokacijom događaja kada se klikne na odgovarajući gumb.

```
  useEffect(() => {
    async function fetchListing() {
      try {
        setLoading(true);
        const res = await fetch(`/api/listing/get/${params.listingId}`);
        const data = await res.json();
        if (data.success === false) {
          setError(true);
          setLoading(false);
          return;
        }

        if (data.time) {
          data.time = new Date(data.time).toLocaleString();
        }
        setListing(data);
        setLoading(false);
        setError(false);

        // Fetch creator's information
        const creatorRes = await fetch(`/api/korisnik/${data.korisnikRef}`);
        const creatorData = await creatorRes.json();
        setCreator(creatorData);
      } catch (error) {
        setError(true);
        setLoading(false);
      }
    }
    fetchListing();
  }, [params.listingId]);
  const handleGoogleMaps = () => {
```

```
    const googleMapsUrl =
`https://www.google.com/maps/search/?api=1&query=${encodeURIComponent(
      listing.city + " " + listing.address
    )}`;
    window.open(googleMapsUrl, "_blank");
  };
```

Slika 36 Dohvaćanje događaja

Stranica UpdateListing se koristi za uređivanje postojećeg događaja. Koristi se za prikupljanje informacija o događaju iz obrasca, uključujući slike, naziv, opis, lokaciju, žanr, broj mjesta i vremenski okvir. Podaci se šalju na odgovarajuću API rutu kako bi se stvorio novi događaj ili ažurirao postojeći. Učitavanje slika i njihovo prikazivanje u galeriji uključuje logiku za upravljanje uploadom slika i prikazivanjem pogrešaka ako se dogodi neuspjeh uploada. Ova stranica također dohvaća detalje postojećeg događaja ako je ID događaja dostupan u URL parametrima kako bi se mogli uređivati.

```
export default function CreateListing() {
  const { currentKorisnik } = useSelector((state) => state.korisnik);
  const [files, setFiles] = useState([]);
  const navigate = useNavigate();
  const params = useParams();
  const [formData, setFormData] = useState({
    imageUrls: [],
    name: "",
    description: "",
    address: "",
    city: "",
    genre: "",
    slot: 6,
    ageOver18: false,
    time: "",
  });
  const [error, setError] = useState(false);
  const [loading, setLoading] = useState(false);
  const [imageUploadError, setImageUploadError] = useState(false);
  const [uploading, setUploading] = useState(false);
  const [selectedGenre, setSelectedGenre] = useState("");
  useEffect(() => {
    const fetchListing = async () => {
      const listingId = params.listingId;
      const res = await fetch(`/api/listing/get/${listingId}`);
      const data = await res.json();
      if (data.success === false) {
        console.log(data.message);
```

```
      return;
    }
    if (data.time) {
      const formattedTime = new Date(data.time).toISOString().slice(0, -8);
      data.time = formattedTime;
    }
```

Slika 37 Stvaranje događaja

Sljedeća stranica „Profile“ se koristi za prikaz i uređivanje korisničkog profila. Korisnik može promijeniti svoju sliku profila klikom na gumb za upload slike. Osim toga, prikazuju se informacije o korisniku poput korisničkog imena i e-maila. Uz pomoć Reduxovog useSelector hooka, dobavljaju se podaci o trenutno prijavljenom korisniku. Kada korisnik odabere sliku za upload, koristi se Firebase Storage za pohranu slike, a zatim se dohvaća URL slike i ažurira se korisnički profil. Uz to, komponenta ima logiku za prikazivanje pogrešaka ako dođe do problema prilikom uploada slike.

```
export default function Profile() {
  const fileRef = korisnikef(null);
  const { currentKorisnik, loading, error } = useSelector((state) =>
state.korisnik);
  const [file, setFile] = useState(undefined);
  const [filePerc, setFilePerc] = useState(0);
  const [fileUploadError, setFileUploadError] = useState(false);
  const [formData, setFormData] = useState({});
  const dispatch = useDispatch();
  const [updateSuccess, setUpdateSuccess] = useState(false);
  const [showMyEventsError, setShowMyEventsError] = useState(false);
  const [korisnikEvents, setKorisnikEvents] = useState([]);
  useEffect(() => {
    if (file) {
      handleFileUpload(file);
    }
  }, [file]);
  const handleFileUpload = (file) => {
    const storage = getStorage(app);
    const fileName = new Date().getTime() + file.name;
    const storageRef = ref(storage, fileName);
    const uploadTask = uploadBytesResumable(storageRef, file);
    uploadTask.on(
      "state_changed",
      (snapshot) => {
        const progress =
          (snapshot.bytesTransferred / snapshot.totalBytes) * 100;
        setFilePerc(Math.round(progress));
```

```
    },
    (error) => {
      setFileUploadError(true);
    },
    () => {
      getDownloadURL(uploadTask.snapshot.ref).then((downloadURL) =>
        setFormData({ ...formData, avatar: downloadURL })
      );
    }
  );
```

Slika 38 Ažuriranje događaja

Sljedeće dvije funkcije služe za upravljanje promjenama i slanje podataka o korisniku na poslužitelj te za brisanje korisničkog računa.

`handleChange` funkcija se poziva svaki put kada se promijeni unos u formi za uređivanje profila. Ona ažurira `formData` stanje s novim vrijednostima unesenih polja.

`handleSubmit` funkcija se aktivira kada korisnik potvrdi promjene u formi. Ona šalje ažurirane podatke o korisniku na poslužitelj putem HTTP POST zahtjeva. Ako je odgovor sa servera neuspješan, prikazuje se poruka o grešci.

`handleDeleteKorisnik` funkcija služi za brisanje korisničkog računa. Kada se pozove, šalje HTTP DELETE zahtjev na poslužitelj koji briše korisnika iz baze podataka. Ako se brisanje uspješno izvrši, korisnik se odjavljuje iz aplikacije.

```
 const handleChange = (e) => {
  setFormData({ ...formData, [e.target.id]: e.target.value });
};

const handleSubmit = async (e) => {
  e.preventDefault();
  try {
    dispatch(updateKorisnikStart());
    const res = await fetch(`/api/korisnik/update/${currentKorisnik._id}`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(formData),
    });
    const data = await res.json();
```

```
      if (data.success === false) {
        dispatch(updateKorisnikFailure(data.message));
        return;
      }

      dispatch(updateKorisnikSuccess(data));
      setUpdateSuccess(true);
    } catch (error) {
      dispatch(updateKorisnikFailure(error.message));
    }
  };

  const handleDeleteKorisnik = async () => {
    try {
      dispatch(deleteKorisnikStart());
      const res = await fetch(`/api/korisnik/delete/${currentKorisnik._id}`, {
        method: "DELETE",
      });
      const data = await res.json();
      if (data.success === false) {
        dispatch(deleteKorisnikFailure(data.message));
        return;
      }
      dispatch(deleteKorisnikSuccess(data));
    } catch (error) {
      dispatch(deleteKorisnikFailure(error.message));
    }
  };
```

Slika 39 Profile stranica

`handleSignOut` funkcija se koristi za odjavu korisnika iz aplikacije. Kada se pozove, šalje HTTP zahtjev za odjavu korisnika na poslužitelj. Ako je odgovor s poslužitelja neuspješan, prikazuje se poruka o grešci.

`handleEventDelete` funkcija služi za brisanje događaja (listinga). Kada se pozove, šalje HTTP DELETE zahtjev na poslužitelj koji briše odabrani događaj iz baze podataka. Ako je odgovor sa servera neuspješan, prikazuje se poruka o grešci.

`handleShowMyEvents` funkcija se koristi za prikazivanje događaja (listinga) koje je korisnik stvorio. Kada se pozove, šalje HTTP zahtjev na poslužitelj kako bi dobio sve listinge koji pripadaju trenutno prijavljenom korisniku. Ako je odgovor sa servera neuspješan, postavlja se stanje `showMyEventsError` na `true`.

```
  const handleSignOut = async () => {
    try {
      dispatch(signOutKorisnikStart());
      const res = await fetch("/api/auth/signout");
      const data = await res.json();
      if (data.success === false) {
        dispatch(signOutKorisnikFailure(data.message));
        return;
      }
      dispatch(signOutKorisnikSuccess(data));
    } catch (error) {
      dispatch(signOutKorisnikFailure(error.message));
    }
  };

  const handleEventDelete = async (listingId) => {
    try {
      const res = await fetch(`/api/listing/delete/${listingId}`, {
        method: "DELETE",
      });
      const data = await res.json();
      if (data.success === false) {
        console.log(data.message);
        return;
      }
      setKorisnikEvents((prev) => prev.filter((event) => event._id !==
listingId));
    } catch (error) {
      console.log(error.message);
    }
  };

  const handleShowMyEvents = async () => {
    try {
      setShowMyEventsError(false);
      const res = await fetch(`/api/korisnik/listings/${currentKorisnik._id}`);
```

```
      const data = await res.json();
      if (data.success === false) {
        setShowMyEventsError(true);
        return;
      }
      setKorisnikEvents(data);
    } catch (error) {
      setShowMyEventsError(true);
    }
  };
```

Slika 40 Odjava i manipulacija eventima

`Search` stranica omogućuje korisnicima pretraživanje događaja na temelju različitih kriterija. Nakon što korisnik unese kriterije pretrage, komponenta dohvaća odgovarajuće događaje s poslužitelja i prikazuje ih u sučelju. Stanja `filterData` i `submitted` prate trenutne kriterije pretrage i jesu li već podneseni, dok se stanje `filteredEvents` koristi za prikaz rezultata pretrage. Nakon promjene kriterija pretrage, useEffect hook ponovno pokreće pretragu i ažurira prikazane događaje.

```
export default function Search() {
  const navigate = useNavigate();
  const location = useLocation();
  const initialFilterData = {
    searchTerm: "",
    city: "",
    genre: "",
    ageOver18: false,
    sort: "createdAt",
    order: "desc",
  };
  const [filterData, setFilterData] = useState(initialFilterData);
  const [loading, setLoading] = useState(false);
  const [submitted, setSubmitted] = useState(false);
  const [availableCities, setAvailableCities] = useState([]);
  const [filteredEvents, setFilteredEvents] = useState([]);

  useEffect(() => {
    if (!submitted) return;
    const urlParams = new URLSearchParams(location.search);
    const fetchEvents = async () => {
      setLoading(true);
      const searchQuery = urlParams.toString();
      const res = await fetch(`/api/listing/get?${searchQuery}`);
      const data = await res.json();
      setLoading(false);
```

```
const cities = [...new Set(data.map((event) => event.city))];
setAvailableCities(cities);
const filteredEvents = data.filter((event) => {
  let isFiltered = true;
  if (
    filterData.searchTerm &&
    !event.name
      .toLowerCase()
      .includes(filterData.searchTerm.toLowerCase())
  ) {
    isFiltered = false;
  }

  if (
    filterData.city &&
    event.city.toLowerCase() !== filterData.city.toLowerCase()
  ) {
    isFiltered = false;
  }
```

Slika 41 Search stranica

3.3.4. Redux

Ovaj kod konfigurira Redux store za upravljanje stanjem aplikacije pomoću Redux biblioteke. Kombinira više reduktora, u ovom slučaju samo `korisnikReducer`, koristeći `combineReducers`.

Nakon toga, konfigurira Redux-persist koji omogućuje pohranu stanja Redux store-a u lokalnoj pohrani preglednika. Konfiguracija Redux-persista uključuje određivanje ključa, tipa pohrane (u ovom slučaju, lokalna pohrana preglednika), i verzije.

Zatim, Redux store se stvara pozivom funkcije `configureStore`, kojoj se predaje kombinirani reduktor i konfiguracija. Također se koristi `getDefaultMiddleware` za postavljanje serijske provjere na `false` kako bi se izbjegli problemi s ne-serijskim podacima koji se pokušavaju pohraniti u Redux-persist.

Na kraju, izvoze se `store` i `persistor`, koji se mogu koristiti u aplikaciji za upravljanje stanjem i trajnom pohranom.

```
import { combineReducers, configureStore } from "@reduxjs/toolkit";
import korisnikReducer from "./korisnik/korisnikSlice";
import { persistReducer, persistStore } from "redux-persist";
import storage from "redux-persist/lib/storage";

const rootReducer = combineReducers({ korisnik: korisnikReducer });

const persistConfig = {
  key: "root",
  storage,
  version: 1,
};

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = configureStore({
  reducer: persistedReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: false,
    }),
});

export const persistor = persistStore(store);
```

Slika 42 Redux

Sljedeći kod koji nam se nalazi u korisnikSlice.js se koristi za korisničko stanje aplikacije. Početno stanje sadrži `currentKorisnik` (trenutni korisnik), `error` (pogreška) i `loading` (stanje učitavanja).

Niz "reducers" definira akcije koje mogu promijeniti stanje. Na primjer, `signInStart`, `signInSuccess`, `signInFailure` obrađuju početak prijave, uspješnu prijavu i neuspješnu prijavu korisnika. Slično, postoje akcije za ažuriranje korisničkih podataka, brisanje korisnika i odjavu korisnika.

Svaka akcija mijenja odgovarajuće polje u stanju, postavljajući `loading` na `true` kada je prijava ili ažuriranje u tijeku, `currentKorisnik` na odgovarajuće vrijednosti i postavljanje `error` ako je došlo do pogreške.

Na kraju, izvoze se akcije kao izvozi `signInStart`, `signInSuccess`, itd., Kao i reduktor definiran u `korisnikSlice.reducer`.

```
import { createSlice } from "@reduxjs/toolkit";

const initialState = {
  currentKorisnik: null,
  error: null,
  loading: false,
};

const korisnikSlice = createSlice({
  name: "korisnik",
  initialState,
  reducers: {
    signInStart: (state) => {
      state.loading = true;
    },
    signInSuccess: (state, action) => {
      state.currentKorisnik = action.payload;
      state.loading = false;
      state.error = null;
    },
    signInFailure: (state, action) => {
      state.error = action.payload;
      state.loading = false;
    },
    updateKorisnikStart: (state) => {
      state.loading = true;
    },
    updateKorisnikSuccess: (state, action) => {
      state.currentKorisnik = action.payload;
      state.loading = false;
```

```
    state.error = null;
  },
  updateKorisnikFailure: (state, action) => {
    state.error = action.payload;
    state.loading = false;
  },
  deleteKorisnikStart: (state) => {
    state.loading = true;
  },
  deleteKorisnikSuccess: (state) => {
    state.currentKorisnik = null;
    state.loading = false;
    state.error = null;
  },
```

Slika 43 Korisnik slice

4.Analiza sličnih rješenja

Trenutačno, na našim prostorima ne postoji središte koje bi objedinilo sve mogućnosti, poput osobnih računa, pregleda i stvaranja društvenih događaja s fokusom na društvene igre. Obično se to odvijalo preko Facebooka i njegovih događaja, ali taj način više nije prvi izbor za mlađe generacije. Takva situacija daje prostora za nova rješenja, koja će objedniti sve potrebne funkcionalnosti koju ova aplikacija pruža. No, i dalje postoje specifične grupe i web stranice koje se bave društvenim igrama na području hrvatske. Najrasprostranjenije su web stranice za sve događaje, ali one ne okupljaju specifičnu skupinu, nego se tamo nalaze koncerti, konferencije i slični događaji. Osim toga, na takvim mjestima se obično naplaćuje kreiranje događaja.

4.1. SWOT analiza

SWOT analiza nas postavlja u okruženje gdje razmišljamo o trenutnom i budućem stanju proizvoda. Svako slovo ima značenje. Strengths, Weaknesses, Opportunities i Threats su glavne komponente SWOT analize. U našoj analizi, primjećujemo da su već spomenute prednosti ključne za privlačenje mlađe skupine. Postoje poboljšanja koja se mogu napraviti s funkcionalne strane, poput više filtera, i filtera koji su specifični, npr filter za prikazivanje događaja u čiji je radijus od naše lokacije jednak 5 km. Pod prilike možemo navesti kako je zadovoljstvo korisnika najbitnije, zato su moguća unaprjeđenja glavna prilika za poboljšavanje aplikacije. Ono što prijeti aplikaciji su velike korporacije, koje imaju pristup sredstvima s kojima mogu napraviti velike i kvalitetne aplikacije [6].

Slika 44 SWOT analiza

4.2 Magic Omens

Magic Omens je specijalizirana trgovina za društvene igre, ali ona također pruža razne događaje s tematikom društvenih igara. Ovdje je bitno napomenuti kako Magic Omens ima rješenje za socijaliziranje i okupljanja s društvenim igrama, ali korisnici ovise isključivo o jednom izvoru događaja, a to je onda kada Magic Omens organizira događaj. Njihovo rješenje nije proširivo, s obzirom na to da su limitirane samo na svoje događaje. Magic Omens je naveden kao glavni konkurent, s obzirom na to da nam je najbliže kvalitetno rješenje koje pripada ovoj kategoriji.

5. Korišteni alati i tehnologije

Tijekom izrade web aplikacije, bitno je posvetiti vrijeme planiranju. U našem slučaju, postoji puno različitih pristupa izradi web aplikacije. Odabirom MERN arhitekture koja se bazira na JavaScript-u osiguravamo da koristimo isti jezik za sve dijelove aplikacije, što nam pruža konzistentnost. JavaScript, kao polazni alat, uz MongoDB, ExpressJS, NodeJS i ReactJS, pružaju nam temelje za stvaranje dobrog rješenja.

5.1. JavaScript

JavaScript [1] je popularan, dinamični skriptni jezik, koji je temelje našeg programskog rješenja. Rasprostranjen je, zove ga se „Language of the Web“ i prisutan je u skoro 99% client-side web stranica.

5.2. React.js i JSX

React.js [2] je JavaScript library za izradu korisničkih sučelja, razvijena od strane Facebooka. Omogućuje izradu interaktivnih i dinamičnih web aplikacija pomoću komponenata, koje su višekratno upotrebljivi dijelovi koda. Koristi virtualni DOM za optimizaciju ažuriranja sučelja, što omogućuje brže performanse. JSX sintaksa omogućava pisanje HTML-a unutar JavaScript koda, čime se olakšava stvaranje komponenti. Zbog svoje učinkovitosti i velike podrške zajednice, React je vrlo popularno rješenje za dinamičke web aplikacije. Valja napomenuti kako se u projektu koristi TailwindCSS, koji je jednostavni CSS framework koji omogućava brzu izgradnju css stilova.

5.3 Express.js

Express.js je minimalistički web framework za Node.js, dizajniran za izgradnju web aplikacija i API-ja. Omogućava jednostavno upravljanje rutama (routes) i middleware-ima, što olakšava obradu HTTP zahtjeva i odgovora. Zbog svoje fleksibilnosti i jednostavnosti, vrlo je popularan izbor među programerima za razvoj server-side aplikacija u JavaScriptu. Ključni su pojmovi API, rute i middlware. Svi oni će nam poslužiti kao alati kako bismo savladali prepreke

u stvaranju naše aplikacije. U kontekstu web aplikacija, POST i GET su HTTP metode koje koriste klijenti (npr. web preglednici) za komunikaciju sa serverom[4].

GET metoda se koristi za dohvaćanje podataka sa servera. Na primjer, kada u web pregledniku upišete URL i pritisnete Enter, preglednik šalje GET zahtjev serveru da bi dobio HTML stranicu.

POST metoda se koristi za slanje podataka na server. Na primjer, kada ispunite obrazac za prijavu na nekoj web stranici i pritisnete gumb "Pošalji", preglednik šalje POST zahtjev s podacima iz obrasca (kao što su korisničko ime i lozinka) serveru.

5.3.1 POST i GET primjer

S obzirom na to da su ovo ključni pojmovi, slijedi i jedan jednostavan primjer kako prikazati njihovu funkcionalnost.

GET:

Situacija: Traženje informacija o proizvodu na web shopu.

Primjer: Kada korisnik klikne na link proizvoda, preglednik šalje GET zahtjev serveru poput *GET/product/FIPU123*.

Rezultat: Server odgovara sa stranicom koja prikazuje detalje o proizvodu s ID-jem *FIPU123*.

POST:

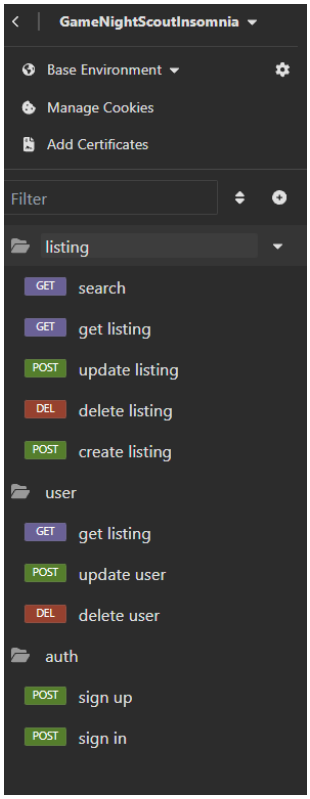
Situacija: Kupnja proizvoda na web shopu.

Primjer: Kada korisnik ispuni formu za kupnju i klikne "Kupi", preglednik šalje POST zahtjev serveru poput POST /checkout s podacima o kupnji (npr. adresa za dostavu, podaci o kreditnoj kartici).

Rezultat: Server obrađuje podatke, potvrđuje kupnju i šalje odgovor (npr. potvrda narudžbe).

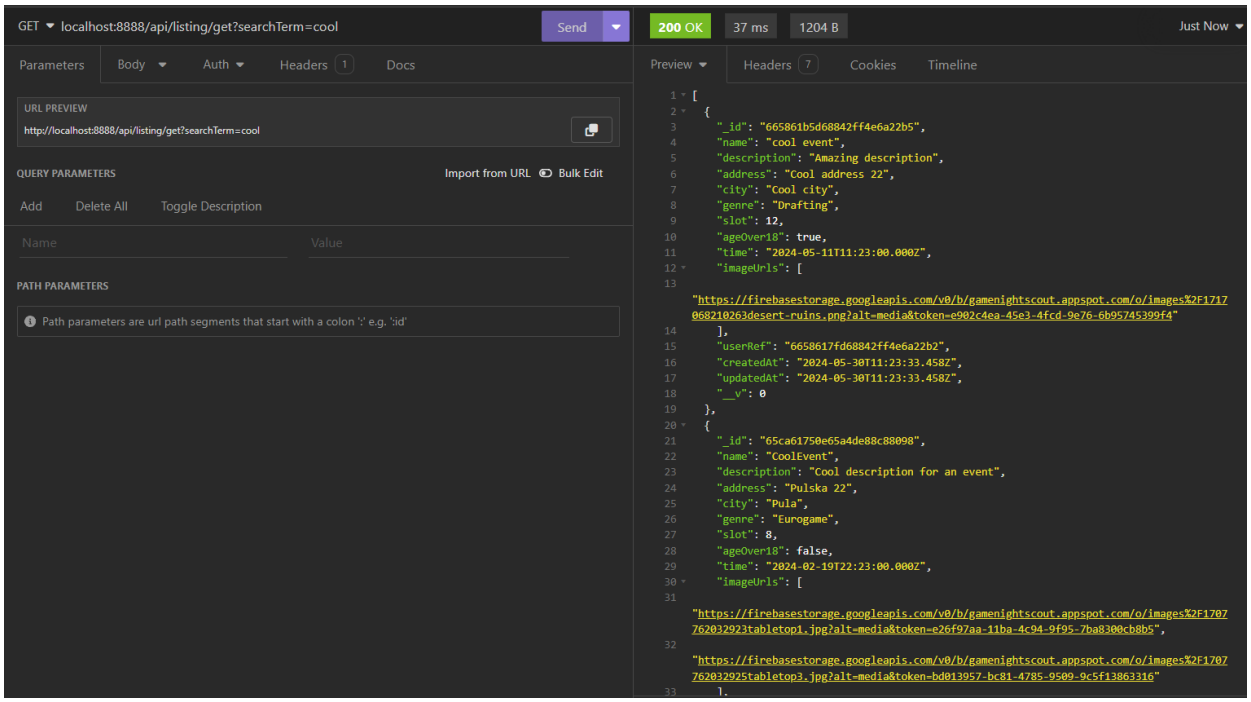
5.4 Insomnia

Insomnia je popularna aplikacija za razvoj, testiranje i dokumentiranje API-ja. Omogućuje programerima da lako šalju HTTP zahtjeve kao što su GET, POST, PUT, DELETE i druge metode te da pregledaju odgovore. Insomnia pruža intuitivan korisnički sučelje koje omogućuje kreiranje, organiziranje i testiranje raznih HTTP zahtjeva, uključujući parametre, zaglavlja i tijelo zahtjeva.



Slika 45 Prikaz glavnih funkcionalnosti aplikacije

Svaki od tih zahtjeva odrađuje određenu funkciju. Kombinacija zahtjeva i naziva govori što će se dogoditi ako napravimo taj zahtjev. Slijedi primjer filtriranja jednog od listing-a. U našem kontekstu, listing je jedna „objava“ odnosno, stvoreni događaj. A kako su događaji postavljeni u obliku liste, primjereno ih je nazivati „listing“



Slika 46 Dohvaćanje događaja koji ima naziv „cool“

Kada u url unesem pod searchTerm „cool“, primjećujemo 2 događaja koji u sebi imaju naziv „cool“. Vidimo njihove glavne podatke i sve što je spremljeno za njih. Poslali smo GET upit koji nam je vratio podatke koje smo tražili.

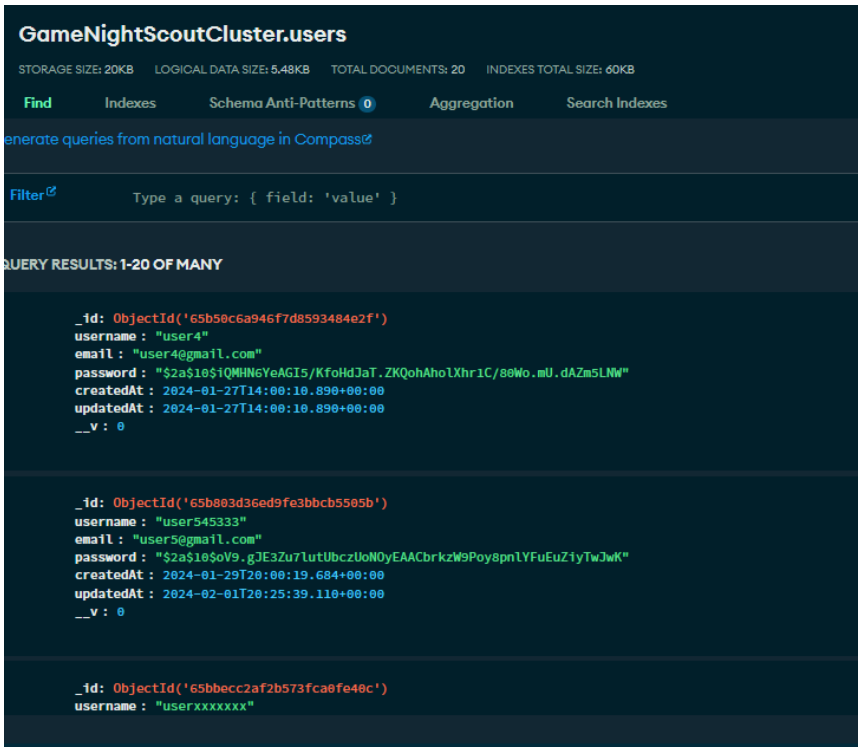
Svaki od Insomnia upita, vidljivim na 3. slici, korišteni su u svrhu testiranja aplikacije. Tijekom razvoja, prije nego što napravimo UI i Frontend, testirali smo naše rute u aplikaciji.

5.5. MongoDB

MongoDB je popularna, skalabilna i fleksibilna baza podataka koja spada u kategoriju NoSQL (ne-relacijskih) baza podataka. Umjesto tablica i relacija, MongoDB koristi dokumente u formatu BSON (Binary JSON) koji se pohranjuju u kolekcije. Ova baza podataka omogućuje brz i jednostavan razvoj aplikacija, pogotovo za aplikacije koje se brzo mijenjaju ili imaju veliku količinu nestrukturiranih podataka. MongoDB podržava razne funkcionalnosti kao što su indeksi,

agregacije, replikacija i sharding, te se može koristiti u različitim scenarijima, uključujući web aplikacije, mobilne aplikacije, analitiku podataka i mnoge druge.
U slijedećoj slici, možemo primjetiti kako su strukturirani podaci za svakog korisnika. Ono što je zaista bitno jest da je svaka šifra enkriptirana. S obzirom na to da radimo s osjetljivim podacima, bitno ih je spremiti na siguran način. Koristimo biblioteku za hashing, odnosno BcryptJS.

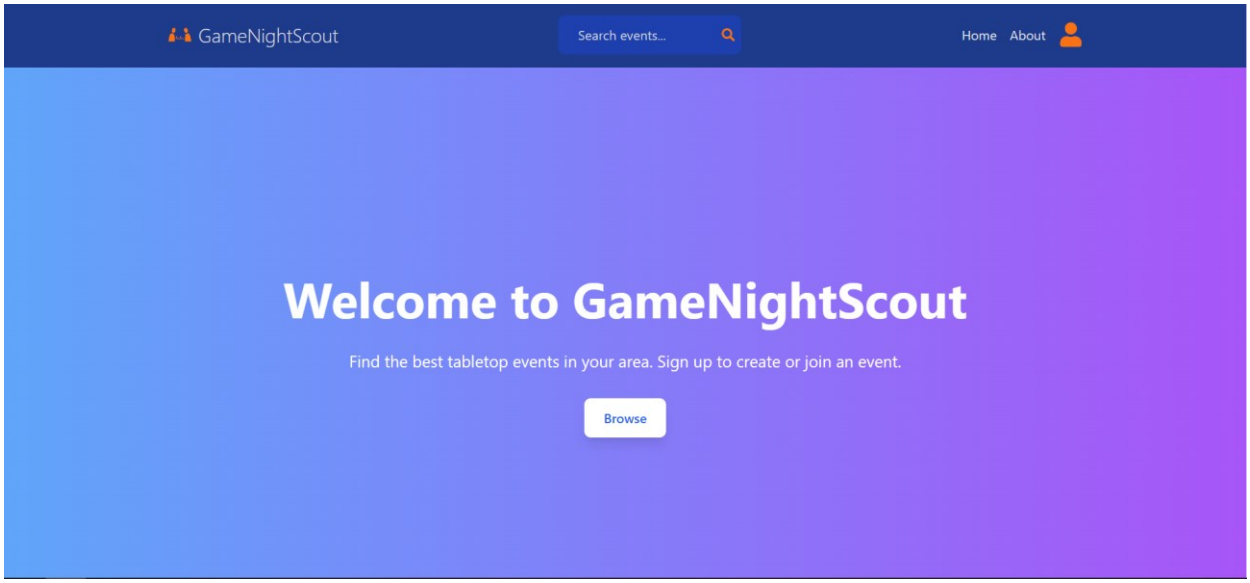
Bcrypt.js je biblioteka za kriptografsko heširanje lozinki u JavaScriptu. Omogućuje programerima da sigurno pohrane lozinke korisnika tako što ih hešira prije spremanja u bazu podataka. Bcrypt.js koristi algoritam bcrypt koji je otporan na različite oblike napada kao što su brute-force i rainbow table napadi. Ova biblioteka je česta opcija u web aplikacijama kako bi se osigurala sigurnost korisničkih lozinki.



Slika 47 Prikaz kolekcija za korisnik-a

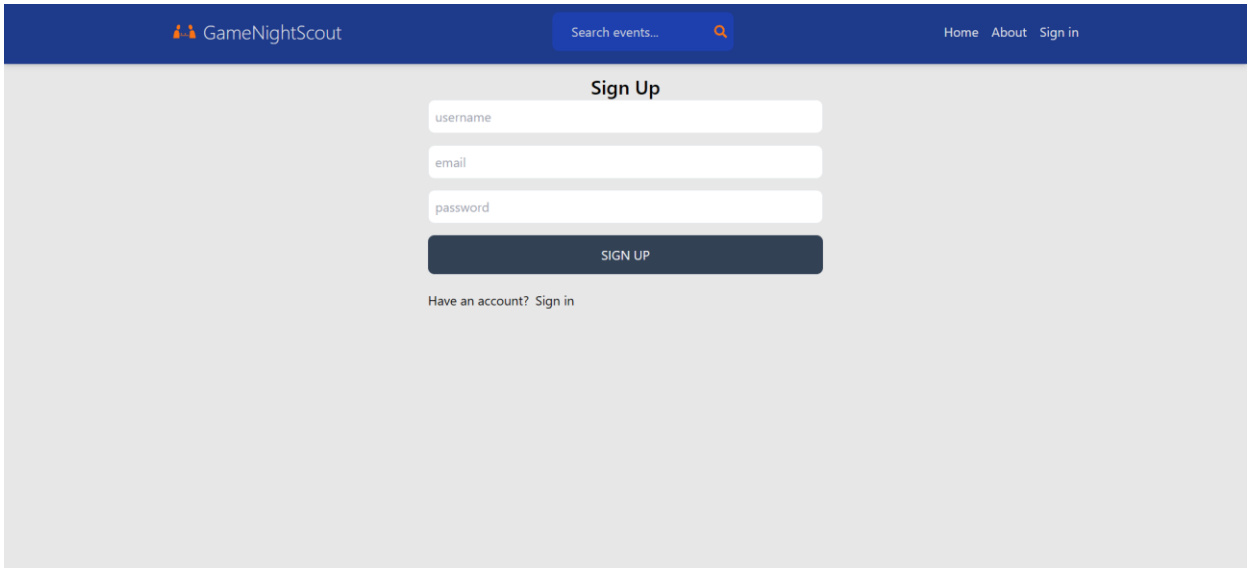
6. Izgled web aplikacije

Homepage nam služi kao početna točka svakog korisnika. U njoj se nalazi jasan tekst koji korisniku daje upute. Također, header komponenta je vidljiva, i na njoj je kao i obično account dio na desnoj strani. Svaki button i tekst vodi na svoj dio aplikacije



Slika 48 Homepage

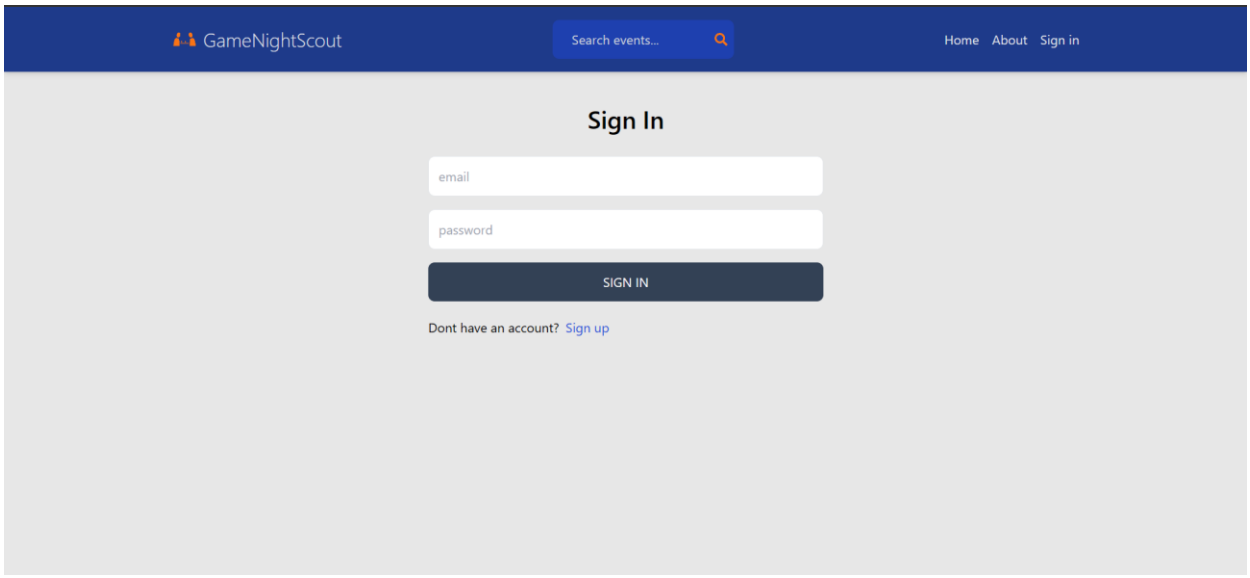
Sign up jednostavno registrira korisnika. Spremamo ga u bazu podataka i enkriptiramo njegovu šifru. Nakon što se korisnik ispravno registrira, vodimo ga na sign in stranicu



Slika 49 Sign Up

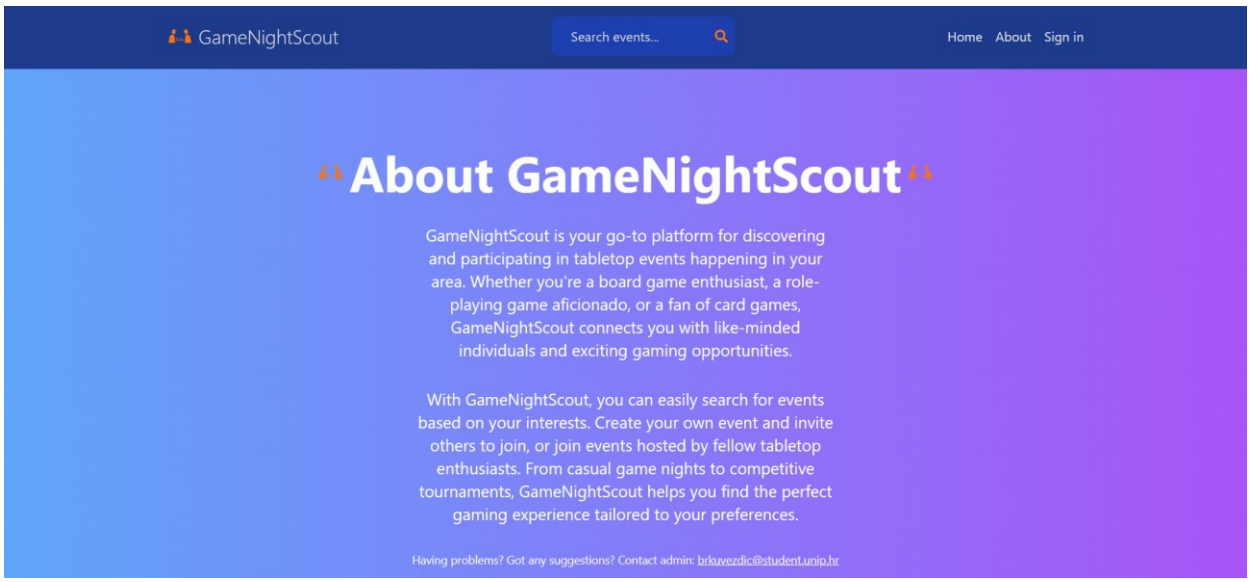
Sign In prijavljuje korisnika, ili ga odbacuje ako su pogrešni podaci. Možemo primjetiti kako nema default avatar ikone iz istoimene komponente. To je zato što korisnik nije prijavljen. Kada se prijavi pokazat će se ili defaultni avatar ili korisnikova ikona. Prijava zatim daje dodatne

mogućnosti koje samo autentificirani korisnici imaju.



Slika 50 Sign In

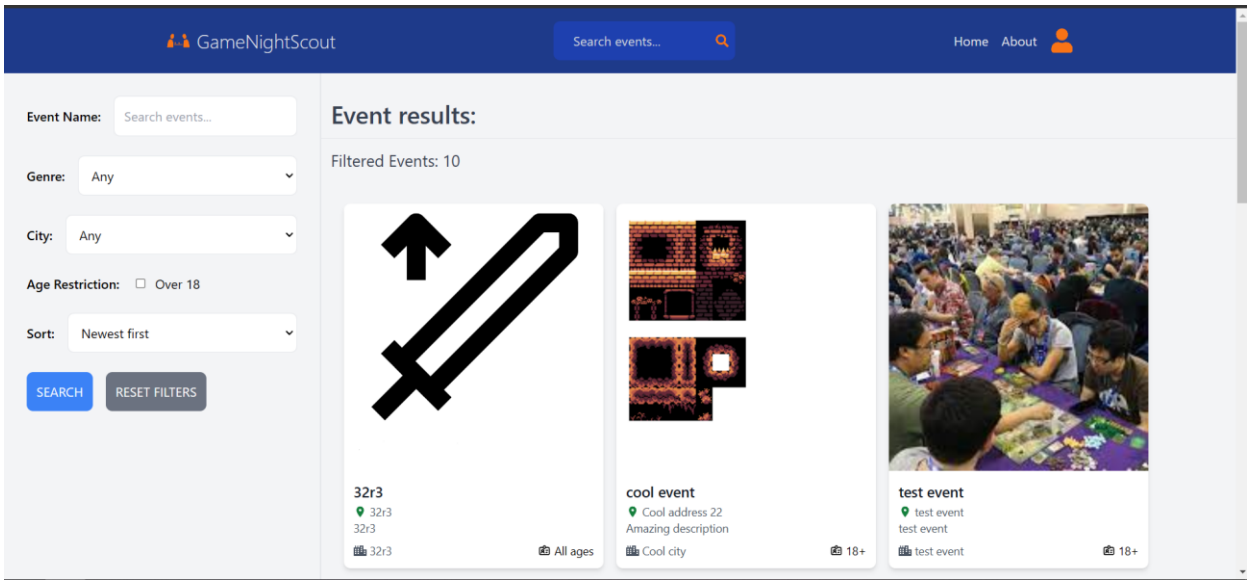
About stranica jednostavno govori o čemu se radi na aplikaciji. Ukratko daje informacije čemu služi aplikacija i njezine funkcionalnosti. Također se nalazi i kontakt informacije administratora ako je potrebno.



Slika 51 About

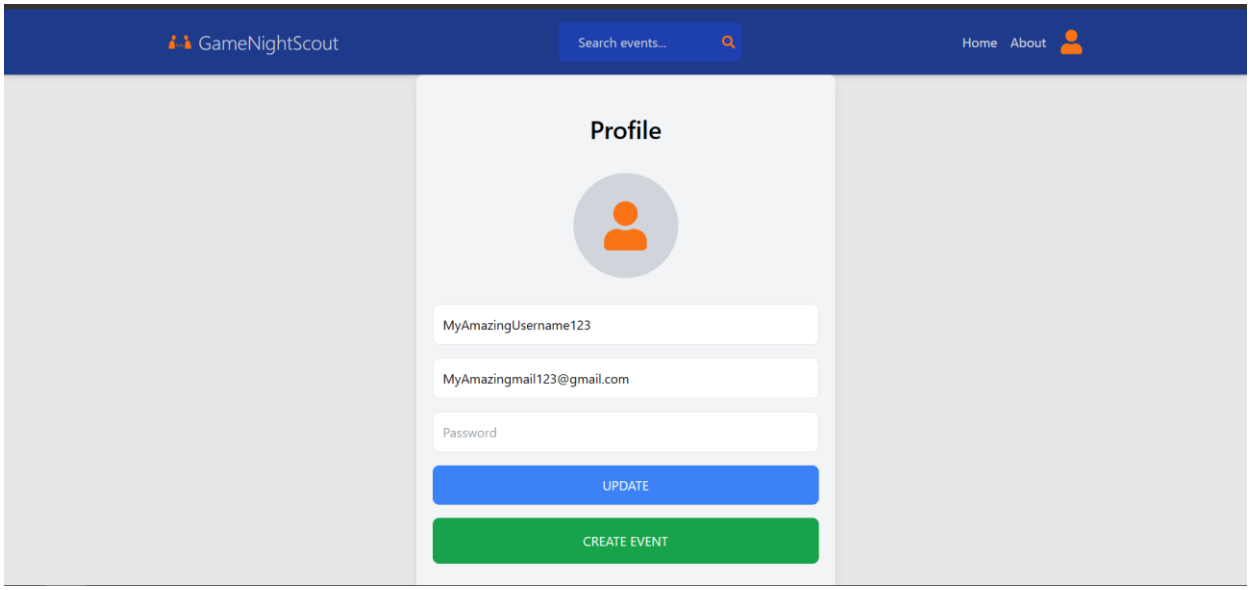
Glavni dio aplikacije je i stranica s filterima i događajima. Razni filteri, nakon primjene, prikazuju događaje. Postoji i search funkcionalnost, ili preko headera ili preko filtera. Zatim,

svaki od tih događaja ima i dodatne informacije kada im se pristupi



Slika 52 Filters

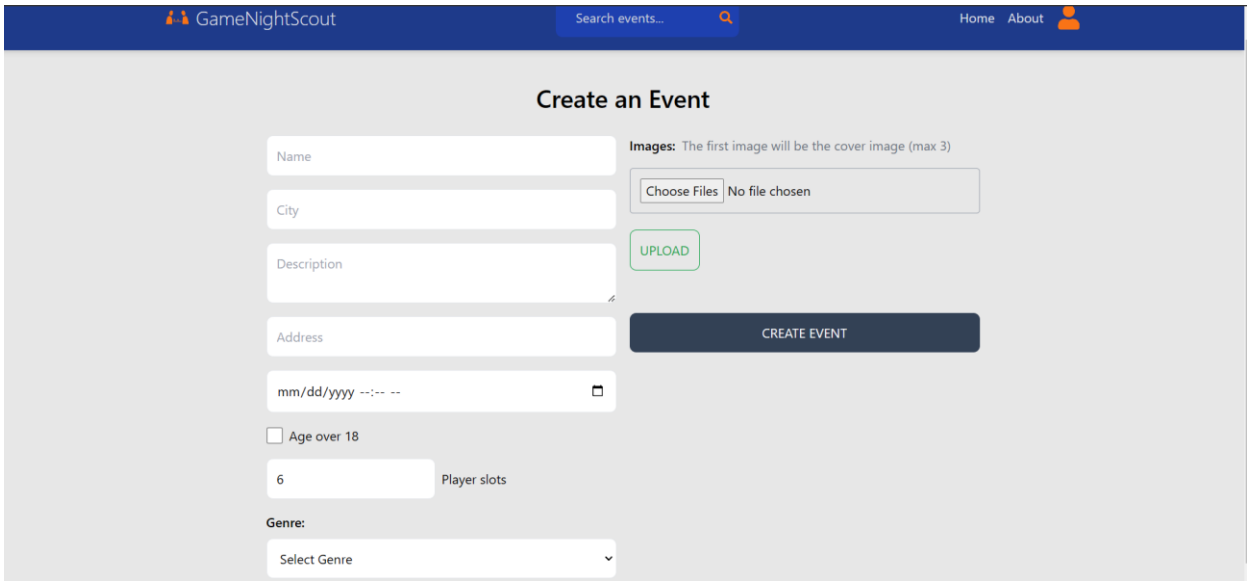
Profile stranica prikazuje osnovne informacije korisnika. Kada u ta polja upišemo druge podatke, možemo i ažurirati podatke, ako su ispravni i da nisu već zauzeti. Također imamo i button za stvaranje vlastitog događaja



Slika 53 Profile

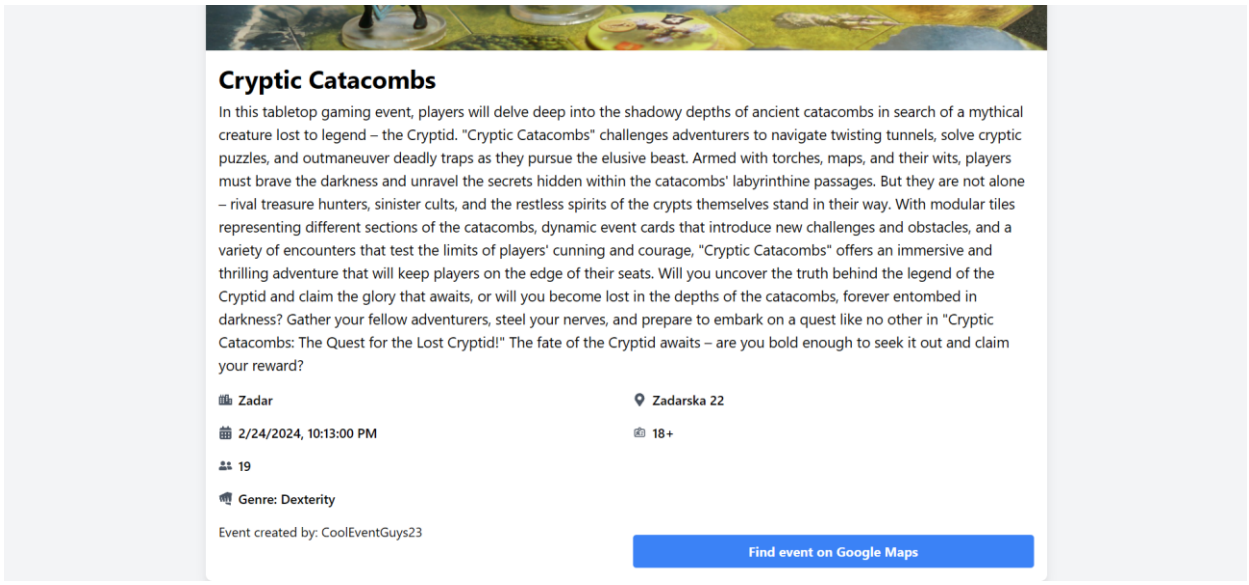
Stranica za stvaranje event-a jasno prikazuje koje informacije je potrebno ispuniti. Svi moraju proći test validacije i nekakvih pravila. Npr moguće je uploadati samo 3 slike, ili da su sva polja

obavezna itd. Također, slike ne smiju biti preko 2 MB.



Slika 54 Event creation

Ovo je detaljan prikaz svakog eventa. Prikazuju se sve informacije koje je korisnik napisao, ali isto tako i button koji nas vodi na Google maps i prikazuje lokaciju koju je korisnik napisao



Slika 55 Event page

Zaključak

Ljudi kao socijalna bića teže prema upoznavanju, druženju i socijalnoj interakciji. Gotovo svakodnevno se susrećemo na razne načine s ljudima oko sebe. U svrhu završnog rada, implementirali smo takvo rješenje, koje će povezati moć tehnologije s ljudskom potrebom, te smo ciljali određenu publiku za koju smo sigurni da traži ovakvo rješenje. Usporedbom tržišta, donesen je zaključak ako ne postoji nekakvo centralno rješenje koje će uzeti u obzir sve prepreke kako bi smo se povezali s istomišljenim ljudima, odnosno onima koji žele uživati u društvenim igrama.

MERN arhitektura kao temelj cijelog projekta, daje nam moć da koristimo Javascript kroz cijeli projekt. Mnogima razne tehnologije stvaraju problem, ali ako iskoristimo ono što znamo, a u isto vrijeme da s time možemo i raditi kroz cijeli projekt, to nam je glavni razlog zašto bismo trebali koristiti MERN arhitekturu. Rješenje je implementirano koristeći se sa stabilnom tehnologijom, i tehnologijom koja je rasprostranjena po cijelome svijetu.

Korisnicima je na ovaj način omogućeno da na jednome mjestu imaju sve što im treba. Prateći dobre standarde sigurnosti, dodatno ih podupiremo da nam se povjere u vezi svojih podataka. Naravno, tehnologije se svakodnevno mijenaju, zato je bitno ostati u koraku sa svim promjenama. Sigurno smo kako će MERN arhitektura ostati kao dobra opcija, zajedno sa svojim varijantama MEAN i MEVN arhitekturama, gdje koristimo Angular ili Vue.

Bitno je napomenuti kako rješenje nije idealno. Puno je prostora za poboljšanje. Od sigurnosti, brzine i još više mogućnosti u samoj aplikaciji. Ta poboljšanja nam mogu služiti kao motivacija. Feedback od korisnika je bitan, te ako im se aplikacija svidi, poboljšanja su i više nego dobrodošla, kako bismo proširili publiku.

Kvalitetna i cjelovata dokumentacija je izuzetno važna kod softverskih projekata. S obzirom na to kako je MERN arhitektura popularna, imamo dostupno dokumentacije koje objašnjavaju principe kako stvoriti aplikaciju poput ove. Insomnia je također bio bitan program, jer smo uz pomoću njega, testirali kako se ponašaju naši podaci. Kada smo bili sigurni u naše rute i informacije, prešli smo na povezivanje tih informacija i stvaranje dizajna i korisničkog sučelja. Valja napomenuti kako sva ta dokumentacija nas može generalno usmjeriti kako stvoriti aplikaciju, ali proces testiranja i popravaka bug-ova je zapravo bio glavni način učenja, poboljšavanja i formiranja cijelog koda u smisleni proizvod.

Literatura

[1] Mozilla Developer Network (MDN), 2024. JavaScript. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript [Pristupljeno 8. lipnja 2024.].

[2] Reactjs.org, 2024. Getting Started – React. Dostupno na: https://legacy.reactjs.org/docs/getting-started.html [Pristupljeno 8. lipnja 2024.].

[3] Banks, A., Porcello, E., 2017. Learning React: A Hands-On Guide to Building Web Applications Using React and Redux. 1. izd. O'Reilly Media. Dostupno na: https://books.google.hr/books?hl=en&lr=&id=NZCKCgAAQBAJ&oi=fnd&pg=PR6&dq=react+javascript&ots=KCswYoEy3i&sig=noe85MN0tfzCHxkPQr6rNMCS1lE&redir_esc=y#v=onepage&q=react%20javascript&f=false

[4] Mozilla Developer Network (MDN), 2024. Express/Node introduction. Dostupno na: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs [Pristupljeno 8. lipnja 2024.].

[5] Aryal, S., 2020. Full-Stack Web Development with MERN. Bachelor's Thesis. Metropolia University of Applied Sciences. Dostupno na: https://www.theseus.fi/bitstream/handle/10024/338237/Samikshya_Aryal_Final_Thesis.pdf?sequence=2

[6] Investopedia.com, 2023, Dostupno na: https://www.investopedia.com/terms/s/swot.asp

Popis slika