IZRADA WEB APLIKACIJE EHEALTH

Bašić, Tea

Undergraduate thesis / Završni rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Split / Sveučilište u Splitu**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:228:895949

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-10-21**

Repository / Repozitorij:

Repository of University Department of Professional Studies

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Računarstvo

TEA BAŠIĆ

Z A V R Š N I   R A D

IZRADA WEB APLIKACIJE EHEALTH

Split, rujan 2023.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE
Preddiplomski stručni studij Računarstvo

Predmet: Izrada web aplikacija

Z A V R Š N I   R A D

Kandidat: Tea Bašić

Naslov rada: Izrada web aplikacije eHealth

Mentor: Marina Rodić, viši predavač

Split, rujan 2023.

Sadržaj

Sažetak 1
Summary 1
eHealth web application development 1
1. Uvod 2
2. Korištene tehnologije 3
 2.1. NodeJS 3
 2.2. ExpressJS 3
 2.3. ReactJS 4
 2.4. MongoDB 4
 2.5. Postman 4
3. Oblikovanje završnog rada 5
 3.1. Postavljanje razvojnog okruženja 5
  3.1.1. Postavljanje Express aplikacije 5
  3.1.2. Postavljanje React aplikacije 6
 3.2. Struktura baze podataka 7
 3.3. Spajanje na bazu podataka 8
  3.3.1. Mongoose 8
  3.3.2. Diskriminatori modela 9
 3.4. Validacija 10
  3.4.1. Paketi Joi & joi-password-complexity 11
 3.5. Autentikacija 12
  3.5.1. Postavljanje lozinke putem elektroničke pošte 13
  3.5.2. Hashiranje lozinke 15
  3.5.3. Upotreba JWT tokena 16
  3.5.4. Autentikacijski kontekst 17
 3.6. Autorizacija 20
  3.6.1. Zaštita ruta na poslužiteljskoj strani 20
  3.6.2. Zaštita ruta na klijentskoj strani 22
 3.7. Izrada kalendara termina 23

3.8. Obavijesti o terminima 26
3.9. Izrada pacijentovog kartona 29
3.9.1. Spajanje na API s dijagnozama 29
3.10. Ocjenjivanje obavljenog pregleda 31
3.11. Implementiranje razgovora u stvarnom vremenu 34
3.12. Uvid u statističke podatke 37
4. Zaključak 38
Literatura 39
Dodatci 41

Sažetak

Cilj ovog završnog rada je razviti web aplikaciju koja bi se koristila u općim zdravstvenim ustanovama. Aplikaciju bi koristili liječnici i pacijenti kako bi lakše zakazali i dogovorili termine te pratili podatke pregleda. Zakazivanje termina se vrši putem kalendara u kojem su prikazani dostupni, kao i zakazani termini. Pregledi se mogu detaljnije dogovoriti putem razgovora (engl. *live chat*) unutar aplikacije. Uz grafički prikaz termina, lakšu međusobnu komunikaciju i dostupne podsjetnike, povećao bi se broj preventivnih pregleda i osigurali bolji zdravstveni uvjeti**.** Aplikacija pruža različita sučelja za gosta, pacijenta, liječnika i administratora u skladu s dodijeljenim dopuštenjima. Atraktivan dizajn i funkcionalnosti jednostavne za korištenje privukli bi liječnike i pacijente da se prijave i počnu koristiti aplikaciju. Općenito, korištenje ove aplikacije bio bi lakši način za zakazivanje sastanaka u ordinacijama i online.

Ključne riječi: ExpressJS, MongoDB, NodeJS, ReactJS

Summary

eHealth web application development

The goal of this final thesis is to develop a web application that would be used in general health care institutions. The application would be used by doctors and patients to make it easier to schedule and arrange appointments and track visit data. Appointments are arranged through a calendar that shows available and scheduled appointments. Visits can be arranged detailedly via live chat within the app. With a graphic display of appointments, easier mutual communication and available reminders, the number of preventive visits would increase and better health conditions would be ensured. The application provides different interfaces for guest, patient, doctor and administrator according to the assigned permissions. An attractive design and easy-to-use features would attract doctors to sign up and start using the app. Overall, using this app would be an easier way to make appointments both onsite and online.

Keywords: ExpressJS, MongoDB, NodeJS, ReactJS

1. Uvod

U suvremenom svijetu današnjice dolazi do porasta izloženosti različitim faktorima koji utječu na ljudsko zdravlje. Neki od njih su užurbani način života, rast i razvoj tehnologije i najnoviji izazov pandemijskih kriza. Problem nastaje kada svi ti faktori sprječavaju jedno od temeljnih ljudskih prava, a to je ostvarenje osnovnih zdravstvenih potreba.

Polazna ideja za razvoj ove aplikacije je omogućiti korisnicima jednostavan i zanimljiv način zakazivanja i održavanja termina pregleda. Pregledi se mogu jednostavno zakazati putem mreže (engl. *online*) kroz ponuđeni kalendar rasporeda. Jednom kada je pregled zakazan, prihvaćen ili odbijen, korisnici dobivaju obavijesti (engl. *notification*) o njegovom statusu kako bi lakše pratili zdravstvene termine uz ostale životne obveze. Osim toga, podsjetnici (engl. *notification reminder*) su zakazani i vremenski, jedan dan i sat vremena prije pregleda, kako bi se osiguralo obavljanje pregleda i korisnicima smanjila potreba za brigom o terminima. Obavijesti se šalju elektroničkom poštom (engl. *email*), a dostupne su za praćenje i unutar aplikacije. Nakon obavljenog pregleda, unose se podatci u pacijentov karton koji je vidljiv kroz sučelje i doktoru i pacijentu, a pacijent unosi recenziju pregleda na temelju vlastitih dojmova.

Uzevši u obzir sve navedene prednosti koje bi aplikacija omogućila korisnicima u svakodnevnom životu, nastaje nada za većom osviještenošću i brigom o vlastitom zdravlju te boljim zdravstvenim uvjetima pojedinca što bi bila i motivacija utkana u pozadinu njene izrade. Jedan od temeljnih motiva od kojih je nastala cijela ideja je omogućiti svakom pojedincu jednostavno upravljanje aplikacijom, bez ikakve obuke, osiguravajući mogućnost korištenja u raznim životnim uvjetima, bila to pandemijska kriza, ubrzani životni ritam ili uobičajena svakodnevnica, zdravlje pojedinca bi uvijek trebalo biti na prvom mjestu.

U sljedećim poglavljima sadržan je opis tehologija korištenih pri izradi aplikacije. Implementacija praktičnog dijela rada kroz razvoj aplikacije, opisana je u trećem poglavlju, odnosno oblikovanju završnog rada. Kratki osvrt o izradi ovog završnog rada, odnosno zaključak, sadržan je u zadnjem poglavlju nakon kojeg slijedi popis literature te slike korisničkog sučelja.

Za razvoj aplikacije sa poslužiteljske (engl. *server*) strane upotrebljeno je razvojno okruženje NodeJS-a, ExpressJS, dok je za klijentsku stranu korišten ReactJS. Razvoj aplikacije, kao i tehnologije koje su korištene, bit će detaljnije opisani u sljedećim poglavljima.

2. Korištene tehnologije

Za izradu aplikacije korišten je MERN stog (engl. *stack*) odnosno MongoDB, ExpressJS, ReactJS i NodeJS. MERN je kolekcija nabrojanih tehnologija, temeljenih na JavaScriptu koje se zajedno koriste za razvoj dinamičkih web aplikacija. Pisanje JavaScript kôda s poslužiteljske i klijentske strane omogućuje lakši prelazak s jednog na drugi kontekst (engl. *context shift*). Tehnologije su više opisane u sljedećim potpoglavljima.

2.1. NodeJS

NodeJS je izvršno okruženje (engl. *runtime environment*) koje se koristi na poslužiteljskoj strani. Pogodno je za izvođenje na više platformi za pokretanje web aplikacija izvan preglednika klijenta te je utemeljeno na otvorenom kôdu (engl. *open source*). Prednosti njegovog korištenja su odlične performanse, prenosivost te korištenje NPM (engl. *Node Package Manager*) upravljača paketima.

Kao asinkrono JavaScript okruženje vođeno događajima, NodeJS je dizajniran za izgradnju skalabilnih mrežnih aplikacija. Skup asinkronih I/O primitiva, dostupnih u standardnoj biblioteci, sprječavaju blokiranje JavaScript kôda. Niti procesa se ne blokiraju prilikom izvođenja I/O (engl. *Input/Output*) operacija, što omogućuje upravljanje mnoštvom istodobnih veza s jednim poslužiteljem.

2.2. ExpressJS

ExpressJS je minimalno i prilagodljivo razvojno okruženje izgrađeno na NodeJS-u koje pruža snažan skup funkcionalnosti za web okruženje. Snažnim ga čine biblioteke (engl. *library*) koje održava Express tim koje se između ostalog koriste za rad s kolačićima, sesijama, korisničkim prijavama, parametrima URL-a (engl. *Uniform Resource Locator*), HTTP (engl. *Hypertext Transfer Protocol*) zahtjevima i sigurnosnim zaglavljima.

Mogućnost umetanja bilo kojeg kompatibilnog *middlewarea* u lanac obrađivanja zahtjeva olakšava korištenje najprikladnijih alata za dovršenje određenog zadatka. ExpressJS je jednostavan za korištenje, lako se integrira s bazama podataka i sadrži proširivu i prilagodljivu arhitekturu što ga čini privlačnim početnicima, ali i iskusnijim programerima.

2.3. ReactJS

ReactJS nije razvojno okruženje ograničeno strukturiranjem i skupom pravila, već JavaScript biblioteka koja programerima omogućuje razvoj UI/UX (engl. *user interface design / user experience design*) komponenti za projekte. Neke od prednosti korištenja ReactJS-a su fleksibilna struktura kôda, mogućnost ponovnog korištenja komponenti (engl. *reusability*) te jednostavno skaliranje projekta.

Izradom jednostavnih enkapsuliranih komponenti napisanih u JavaScript kôdu omogućuje ponovnu upotrebu kôda uz prosljeđivanje različitih podataka. React upravlja osnovnim operacijama nad podatcima, upotrebom deklarativnog pristupa, u kombinaciji s kukama (engl. *hooks*). Podatci se ažuriraju bez ručnog manipuliranja DOM-om (engl. *Document Object Model*) ili rukovanja složenom logikom ažuriranja.

2.4. MongoDB

MongoDB je nerelacijska (engl. *NoSQL: not only SQL*) baza podataka koja se koristi za spremanje nestrukturiranih indeksiranih podataka, odnosno fleksibilnih dokumenata u obliku JSON (engl. *JavaScript Object Notation*) objekata. Može se koristiti putem usluge oblaka (engl. *Cloud*) ili lokalno kao poslužitelj. Za potrebe ove aplikacije korištena je usluga MongoDB Atlas oblaka.

Upotreba JSON objekata, asinkroni I/O model, horizontalna skalabilnost i fleksibilnost podataka čine ga popularnim za upotrebu u kombinaciji s NodeJS-om. Upotrebom besplatne usluge Atlas oblaka iskorištena je visoka skalabilnost pri prihvaćanju prometa i podataka bez potrebe za stvaranjem udaljenih kontejnera, što je omogućilo uštedu vremena i resursa.

2.5. Postman

Kako bi se testirali API pozivi i otklonile moguće pogreške na poslužiteljskoj strani, instalirana je Postman API platforma. Postman omogućuje jednostavnost testiranja ruta uz mogućnost organiziranja API zahtjeva upotrebom kolekcija. Izrađeni su zahtjevi s različitim metodama (GET, POST, PATCH, PUT, DELETE), zaglavljima, parametrima i tijelima zahtjeva za interakciju s API-jem i ispitivanje odgovora.

3. Oblikovanje završnog rada

3.1. Postavljanje razvojnog okruženja

Postavljanje razvojnog okruženja počinje konfiguracijom poslužiteljske strane odnosno instalacijom Express aplikacije na NodeJS izvršnom okruženju. Express aplikacija predstavlja API (engl. *Application Programming Interface*) koji će služiti za komunikaciju između ReactJS klijentske strane i baze podataka MongoDB.

Na samom početku potrebno je preuzeti i instalirati NodeJS izvršno okruženje s njegove službene web stranice kako bi se mogle koristiti usluge NPM upravljača paketima. NPM je zadani upravljač paketa za JavaScript NodeJS izvršno okruženje. Sastoji se od npm klijenta naredbenog retka i npm registra odnosno online baze podataka u kojoj su sadržani dostupni paketi. Počeo se koristiti kao način preuzimanja i upravljanja ovisnostima (engl. *dependencies*) paketa NodeJS-a, ali je postao alat koji se također koristi i s klijentske strane u JavaScript sučeljima.

3.1.1. Postavljanje Express aplikacije

Za potrebe ovog projekta stvorena je mapa naziva `backend` koja će predstavljati njegovu poslužiteljsku stranu. Dva početna dokumenta koja predstavljaju okosnicu i glavnu točku postavljanja Express aplikacije su `index.js` i `package.json`.

Datoteka `index.js` služi za registriranje Express aplikacije i sadrži konfiguracijske postavke poslužiteljske strane dok `package.json` datoteka služi za praćenje ovisnosti i registriranje *custom* skripti. Kako bi se izradila `package.json` datoteka, potrebno je unijeti naredbu "`npm init -y`" u naredbeni redak. Da bi se koristilo Express razvojno okruženje, potrebno je instalirati paket Express koristeći naredbu "`npm install express`". Nakon toga moguće je registrirati Express u `index.js` datoteci i pokrenuti poslužiteljsku stranu na dodijeljenom portu.

Kako bi se smanjila potreba za ponovnim pokretanjem aplikacije pri mijenjanju kôda i osigurala veća brzina razvojnog okruženja instaliran je paket nodemon unošenjem naredbe "`npm install -g nodemon`". Nodemon je alat koji nadzire direktorij projekta i automatski ponovno pokreće aplikaciju kada otkrije bilo kakve promjene što znači da nije potrebno zaustavljati i ponovno

pokretati aplikaciju kako bi promjene stupile na snagu.

3.1.2. Postavljanje React aplikacije

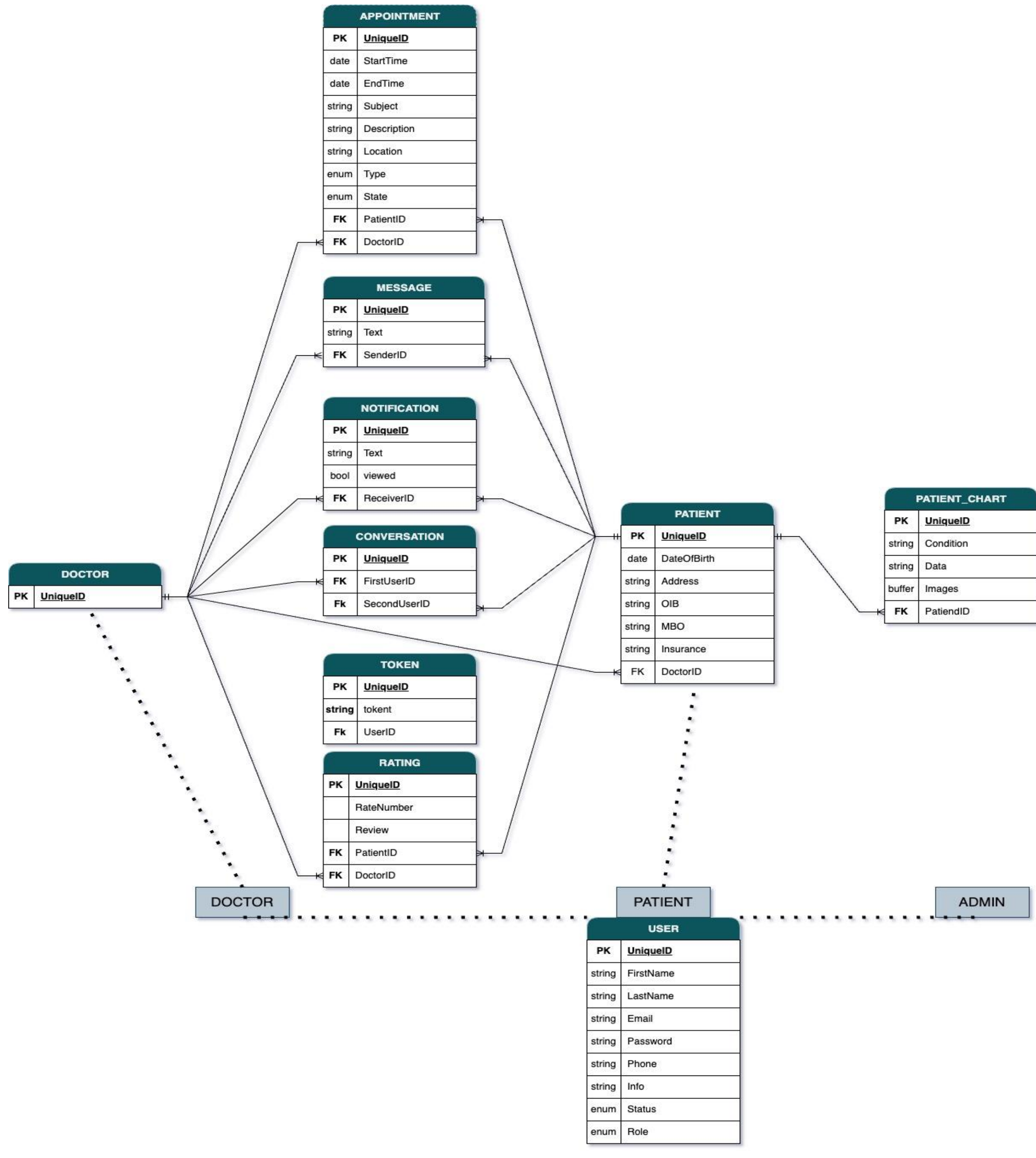
Nakon postavljanja poslužiteljske strane, stvorena je mapa naziva `frontend` koja predstavlja klijentsku stranu projekta. Kako bi se inicijalizirala React aplikacija potrebno je izvršiti naredbu “`npx create-react-app`”. NPX (engl. *Node Package Execute*) je npm paketni izvršitelj koji se koristi za direktno izvršavanje JavaScript paketa, bez njihove instalacije.

Usmjeravanje zahtjeva (engl. *routing*) nije sadržano u inicijaliziranom React projektu, već je potrebno instalirati *React Router* koji omogućuje navigaciju bez osvježavanja stranice, a njegova instalacija se izvršava naredbom “`npm install react-router-dom`”. Osnovne postavke usmjeravanja zahtjeva ovog projekta sadržane su u datoteci `App.js`.

Osim osnovnih funkcionalnosti, instaliran je i Tailwind CSS (engl. *Cascading Style Sheets*) kako bi se stilizirale komponente projekta. Tailwind CSS je uslužni CSS okvir koji omogućuje bržu i lakšu izradu responzivnih aplikacija koristeći pomoćne klase. Kako bi se koristio u ovom projektu izvršene su naredbe “`npm install -D tailwindcss`” i “`npx tailwindcss init`” te je uvezen (engl. *import*) u glavnu CSS datoteku `styles.css`.

3.2. Struktura baze podataka

Na slici 1 prikazan je model baze podataka upotrebljen za strukturiranje podataka i relacija aplikacije. Entiteti na slici predstavljaju sheme dokumenata korištenih za pohranu podataka u obliku JSON objekta.



Slika 1: Dijagram entiteta i veza baze podataka

3.3. Spajanje na bazu podataka

Spajanje na bazu podataka MongoDB izvršeno je preko usluge MongoDB Atlas oblaka. Kako bi koristili usluge Atlas oblaka potrebno je napraviti korisnički račun na njegovoj mrežnoj stranici nakon čije je izrade moguće napraviti vlastitu bazu podataka, odnosno besplatni dijeljeni klaster (engl. *cluster*) unutar usluge oblaka.

Nakon kreacije baze podataka stvara se poveznica (engl. *connection string*) preko koje je moguće pristupiti bazi iz NodeJS aplikacije uz upotrebu šifre i korisničkog imena. Zbog sigurnosti osjetljivih podataka i sprječavanja zloupotrebe istih korištena je `.env` datoteka u koju su spremljeni podatci poput poveznice na Atlas klaster.

Za potrebe projekta stvorene su odvojene `.env` datoteke u root direktorijima poslužitelja i klijenta. Ideja korištenja `.env` datoteke je skrivanje osjetljivih podataka, odnosno varijabli okruženja (engl. *environment variable*) pri objavljivanju projekta na različitim mrežnim platformama kao što je GitHub. Kako bi se koristile varijable spremljene u `.env` datoteci instaliran je paket dotenv izvršavanjem naredbe “`npm i dotenv`”. Paket dotenv učitava varijable iz `.env` datoteke u `process.env` objekt dostupan globalno u NodeJS okruženju.

3.3.1. Mongoose

S obzirom da je MongoDB nerelacijska baza podataka čija je karakteristika fleksibilni podatkovni model, instaliran je paket Mongoose naredbom “`npm install mongoose`” kako bi se osigurala strukturiranost podataka.

Mongoose ODM (engl. *Object Data Modeling*) je biblioteka za modeliranje objektnih podataka temeljena na NodeJS-u za MongoDB. Sličan je ORM-u (engl. *Object Relational Mapper*) za tradicionalne SQL baze podataka. Mongoose u suštini služi kao dodatni sloj na MongoDB-u koji omogućuje metode za čitanje i pisanje dokumenata i interakciju s bazom podataka, osigurava način deklariranja shema (engl. *schema*) modela koji osiguravaju strožu strukturu podataka. Mongoose također nudi razne funkcionalnosti poput kuka i provjere valjanosti modela koje olakšavaju rad s MongoDB-om.

3.3.2. Diskriminatori modela

Kao što je vidljivo na slici strukture modela baze podataka, model korisnika (na slici 1 user) predstavlja osnovni, odnosno bazni model s obzirom da sadržava polja koja su zajednička liječniku, pacijentu i administratoru. Kako bi se implementirale najbolje prakse kodiranja odnosno koristio *DRY* (engl. *Don't repeat yourself*) princip i poboljšala izvedba uvedeni su Mongooseovi model diskriminatori (engl. *model discriminators*).

Diskriminatori su mehanizmi nasljeđivanja sheme koji omogućuju ostvarivanje više modela s preklapajućim shemama na vrhu iste osnovne MongoDB kolekcije. Način na koji Mongoose razlikuje modele diskriminatora je ključ diskriminatora (engl. *discriminator key*) kojeg je moguće definirati u osnovnom modelu i koji se koristi kao jedinstveni ključ za razlikovanje dviju shema, u ovom slučaju korisnika i pacijenta.

Korištenjem diskriminatora smanjeno je dupliciranje kôda i ostvaren je *DRY* princip. Ukoliko dođe do porasta potreba aplikacije, na primjer dodavanje novih svojstava koje su zajedničke shemama korisnika i pacijenta, potrebno je napraviti promjene samo na jednom od modela. Ukoliko se otvori potreba za dodavanjem novih modela (na primjer medicinskog osoblja), moguće je lako proširiti postojeća svojstva.

```
const baseOption = {
  discriminatorKey: "type",
  collection: "user",
};
```

Ispis 1: Definiranje ključa diskriminatora (model korisnik)

U ispisu 1 nalazi se primjer spremanja osnovnih postavki sheme korisnik u varijablu `baseOption`. `discriminatorKey` predstavlja jedinstveni identifikator koji služi za razlikovanje kolekcija dok se `collection` koristi za premošćivanje (engl. *override*) imena dodanog modela. Varijabla `baseOption` se zatim koristi kao jedno od polja modela korisnik što je vidljivo na ispisu 3.

Primjer upotrebe funkcije diskriminator na modelu pacijenta prikazan je u ispisu 2.

```
export const Patient = User.discriminator("Patient", PatientSchema);
```

Ispis 2: Upotreba diskriminator funkcije (model pacijent)

3.4. Validacija

Validacija modela projekta ostvarena je koristeći Mongooseovu ugrađenu (engl. *built-in*) validaciju. Ona je zapravo *middleware* definiran na Mongoseovom SchemaTypeu odnosno konfiguracijskom objektu svakog polja modela.

U ispisu 3 je prikazana upotreba nekoliko ugrađenih validatora. Svi SchemaTypes imaju ugrađeni validator `required`, brojevi imaju `min` i `max` validatore, nizovi imaju validatore enum.

```
const UserSchema = new mongoose.Schema(
  {
    firstName: {
      type: String,
      required: true,
      min: 3,
      max: 255,
    },
    lastName: {
      type: String,
      required: true,
      min: 3,
      max: 255,
    },
    email: {
      type: String,
      required: true,
      unique: true,
      min: 6,
      max: 255,
    },
    password: {
      type: String,
      max: 1024,
      min: 6,
    },
    phone: {
      type: String,
      required: true,
      unique: true,
    },
    info: {
      type: String,
    },
    role: {
      type: String,
      enum: Object.values(userRole),
      default: userRole.Patient,
    },
  },
  baseOption
);
```

Ispis 3: Primjer upotrebe Mongooseove validacije (model korisnik)

3.4.1. Paketi Joi & joi-password-complexity

Osim upotrebe ugrađene validacije, instalirani su i dodatni paketi `joi` izvršavanjem naredbe "`npm install joi`" te `joi-password-complexity` izvršavanjem naredbe "`npm install password-complexity`". Upotreba `joi` modula za provjeru podataka koristeći funkcije `optional`, `required`, `min`, `email` i `validate` prikazana je u ispisu 4 u funkciji koja se koristi prilikom registracije korisnika.

Koristeći shemu definirana su pravila na poljima modela kao što su minimalna duljina funkcijom `min`, obavezan unos funkcijom `required` i provjera stringa kao valjane email adrese funkcijom `email`. Zatim se na definiranoj shemi koristi funkcija `validate` kojoj se prosljeđuje objekt (u ovom primjeru naziva `data`) kako bi se provjerila njegova valjanost.

```
import joi from "joi";
import JoiPasswordComplexity from "joi-password-complexity";

export const registerValidation = (data) => {
  const schema = joi.object({
    firstName: joi.string().min(3).required(),
    lastName: joi.string().min(3).required(),
    email: joi.string().min(6).required().email(),
    phone: joi.number().min(3).required(),
    role: joi.string().required(),
  });

  return schema.validate(data);
};
```

Ispis 4: Primjer upotrebe `joi` validacije za potrebe registracije

U ispisu 5 prikazana je potreba kombinacije paketa `password-complexity` i `joi` na primjeru postavljanja korisničke lozinke koji zajedno tvore snažnu validacijsku funkcionalnost. Validacija snage lozinke može se prilagoditi potrebama aplikacije definiranjem željenih postavki potrebnih vrijednosti.

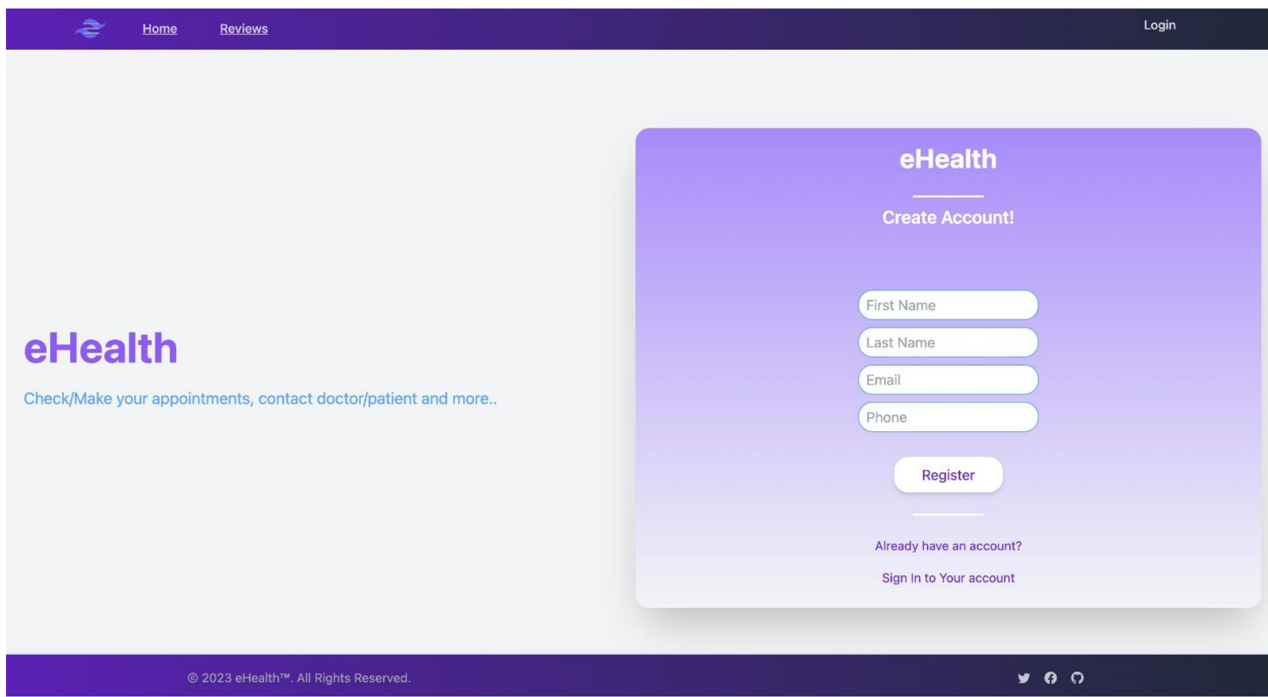
```
const complexityOptions = {
  min: 5,
  max: 1024,
  lowerCase: 1,
  upperCase: 1,
  numeric: 1,
  symbol: 1,
  requirementCount: 4,
};

export const passwordValidation = (data) => {
  const passwordSchema = joi.object({
    password: JoiPasswordComplexity(complexityOptions),
    confirmPassword: joi.string().required().valid(joi.ref("password")),
  });

  return passwordSchema.validate(data);
};
```

Ispis 5: Primjer upotrebe `joi-password-complexity` validacije

Kao što je prikazano u ispisu 5, definirana je varijabla `complexityOptions` kojoj su pridjeljene postavke koje predstavljaju uvjete potrebne kako bi lozinka bila valjana. U ovom slučaju to su minimalni i maksimalni broj slova, barem jedno veliko i malo slovo, barem jedan broj i simbol te minimalni broj uvjeta koji lozinka mora zadovoljiti kako bi bila valjana. Maksimalni broj slova postavljen je na veću vrijednost s obzirom da se lozinka treba hashirati (engl. *hashing*) pri spremanju u bazu podataka.

3.5. Autentikacija

Nakon što su implementirani modeli i njihova validacija, dodana je autentikacija kako bi se korisnici registrirali i prijavili u aplikaciju. Liječnici popunjavaju dostupnu registracijsku formu prikazanu na slici 2 te se uneseni podatci provjeravaju preko registracijske funkcije navedene u ispisu 4, ukoliko su uneseni podatci valjani korisnici primaju poruku u obliku elektroničke pošte koja sadrži link za postavljanje lozinke.



Slika 2: Registracijska forma

3.5.1. Postavljanje lozinke putem elektroničke pošte

Kako bi se mogle slati poruke putem elektroničke pošte instaliran je paket `nodemailer` izvršavanjem naredbe "`npm install nodemailer`". Za potrebe aplikacije korištena je usluga nodemailera u kombinaciji s Gmail računom.

Od 2022 godine onemogućena je podrška upotrebe korisničkog računa i lozine za pristup računu iz manje sigurnih aplikacija. Kako bi se Gmail koristio u kombinaciji s nodemailerom potrebno je konfigurirati lozinku aplikacije unutar postavki korisničkog računa.

Otvoren je novi Gmail korisnički račun naziva "ehealth.users@gmail.com" u kojem je eksterno, unutar Gmailovih postavki, omogućena potvrda u dva koraka (engl. *2-step verification*). Nako što se aktivira potvrda u dva koraka, moguće je generirati lozinku aplikacije, koja se zatim sprema u `.env` datoteku zajedno s adresom Gmail korisničkog računa. Email adresa i lozinka se zatim koriste u kôdu aplikacije kako bi se omogućilo slanje elektroničke pošte upotrebom paketa nodemailer.

Prikaz primjera upotrebe paketa `nodemailer` koristeći usluge Gmail elektroničke pošte vidljiv je u ispisu 6.

```
export const sendEmail = async (email, subject, text) => {
  try {
    const transporter = nodemailer.createTransport({
      host: process.env.NODEMAILER_HOST,
      service: process.env.NODEMAILER_SERVICE,
      port: 587,
      secure: true,
      auth: {
        user: process.env.NODEMAILER_USER,
        pass: process.env.NODEMAILER_PASSWORD,
      },
    });

    await transporter.sendMail({
      from: process.env.USER,
      to: email,
      subject: subject,
      html: text,
    });

    console.log("email sent sucessfully");
  } catch (error) {
    console.log(error, "email not sent");
  }
};
```

Ispis 6: Primjer upotrebe paketa `nodemailer`

U funkciji `sendEmail` prikazanoj u ispisu 6, definiran je objekt naziva `transporter` nad kojim se poziva funkcija za slanje elektroničke pošte. Definiranom objektu se pridjeljuju konfiguracijske postavke koje omogućuju slanje pošte korištenjem Gmail hosta, putem SMTP (engl. *Simple Mail Transfer Protocol*) servisa koristeći TLS (engl. *Transport Layer Security*) odnosno sigurnu vezu na portu 587. U postavkama je definiran i `auth` objekt u koji su spremljeni prethodno spomenuti Gmail račun i generirana lozinka.

Prikaz primjera implementacije funkcije `sendEmail` vidljiv je u ispisu 7.

Korisniku se šalje link koji ga vodi na stranicu aplikacije gdje može postaviti lozinku koju će koristiti prilikom prijave u aplikaciju. Kao dodatno osiguranje, link sadrži i verifikacijski token koji se sprema u bazu podataka. Kada korisnik šalje zahtjev za postavljanje lozinke, token se provjerava i mora biti valjan.

```
const user = new User(req.body);
    try {
      const savedUser = await user.save();
      let token = await Token.findOne({ userId: user._id });
      if (!token) {
        token = await new Token({
          userID: user._id,
          token: crypto.randomBytes(32).toString("hex"),
        }).save();
      }

      const link = `http://localhost:3000/set-password/${encodeURIComponent(
        user._id
      )}/${encodeURIComponent(token.token)}`;

      sendEmail(
        savedUser.email,
        "password setup",
        `
                   <h3>eHealth account created</h3>
         <p>A new account has been created for ${user.email}.
            Please follow the link to set up password</p>
                <a href="${link}">Create password</a>
        `
      );
```

Ispis 7: Slanje elektroničke pošte za postavljanje lozinke

3.5.2. Hashiranje lozinke

Za potrebe hashiranja lozinke instaliran je paket `bcrypt` koristeći naredbu “`npm install bcrypt`”. Bcrypt koristi varijantu Blowfish enkripcijskog algoritma za šifriranje i uvodi faktor rada, a njegova upotreba vidljiva je u ispisu 8.

Faktor rada omogućuje odrediti koliko će skupa biti hash funkcija zbog čega bcrypt može držati korak s Mooreovim zakonom. Kako računala postaju brža, moguće je povećati faktor rada i hash će biti sporiji.

```
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(req.body.password, salt);
```

Ispis 8: Hashiranje lozinke

U ispisu 8 faktor rada inicijaliziran je u funkciji `genSalt`, a njegova vrijednost je 10 što znači da će se lozinka hashirati u 10 rundi. Što je veći broj koji mu je dodijeljen, više je otporan na napade grube sile (engl. *brute-force*). Funkciji `hash` se prosljeđuje šifra i generirana sol (engl. *salt*) odnosno nasumični niz znakova kako bi se dodao dodatni sloj sigurnosti. Ukoliko dva korisnika koriste istu šifru, dodavanjem soli sprječeno je njihovo uspoređivanje u hashiranom obliku.

3.5.3. Upotreba JWT tokena

JWT token (engl. *JSON WEB token*) koristi se u aplikaciji kako bi poslužiteljska strana poslala informaciju o valjanoj autentikaciji prilikom prijave u aplikaciju klijentskoj strani. Instaliran je naredbom "`npm install jsonwebtoken`".

Sastoji se od tri dijela, odnosno zaglavlja (engl. *header*) koje sadrži korišteni algoritam i tip, sadržaja (engl. *payload*) u kojem su neosjetljivi podatci o korisniku te potpis (engl. *signature*) koji se koristi kako bi poslužitelj potvrdio autentičnost tokena.

Primjer upotrebe JWT tokena prikazan je u ispisu 9.

Upotrebom funkcije `sign` stvara se token sadržaja `id` kojem se dodaje automatsko zaglavlje s podacima o upotrebljenom algoritmu. Kako bi se onemogućila zloupotreba tokena njegovom dekripcijom i mijenjanjem podataka, koristi se `TOKEN_SECRET` varijabla kojoj je pridjeljen tajni string, a pohranjena je u `.env` datototeci. Tajni string se koristi prilikom hashiranja zaglavlja i podataka i mora biti poznat samo poslužiteljskoj strani, na taj način se onemogućuje mijenjanje sadržaja tokena s obzirom da se mijenjanjem sadržaja mijenja i tajni string.

```
const token = jwt.sign({ id: user._id }, process.env.TOKEN_SECRET, {
  expiresIn: "3d",
});

res.status(200).json({
  _id: user._id,
  email: req.body.email,
  token: token,
  role: user.role,
});
```

Ispis 9: Primjer upotrebe JWT tokena

3.5.4. Autentikacijski kontekst

Kako bi klijentska strana mogla pratiti stanja korisnika odnosno je li korisnik autenticiran ili ne, stvoren je autentikacijski kontekst koji sadrži neosjetljive korisničke podatke dostupne globalno, svim React komponentama. Primjer kôda sadržanog u datoteci `authContext.js` vidljiv je u ispisu 11.

Upotrebom funkcije `authReducer` prati se korisničko stanje odnosno je li korisnik prijavljen ili ne. Ukoliko je korisnik prijavljen, funkcija vraća njegove podatke poslane s poslužiteljske strane, ukoliko korisnik nije prijavljen, funkcija vraća `null`, zadana povratna vrijednost je početno stanje korisnika poslano funkciji.

Pozivom funkcije `createContext` stvoren je autentikacijski kontekst dostupan svim aplikacijskim komponentama upotebom varijable `AuthContextProvider`, odnosno komponente koja se uvozi u `index.js` datoteku i koristi kao omotač glavnoj (engl. *root*) aplikacijskoj komponenti.

Upotreba AuthContextProvidera kao omotača svih aplikacijskih komponenti vidljiva je na primjeru sadržanom u ispisu 10.

```
ReactDOM.createRoot(document.getElementById("root")).render(
  <AuthContextProvider>
    <App />
  </AuthContextProvider>
);
```

Ispis 10: Upotreba Autentikacijskog konteksta

`AuthContextProvider` komponenti se prosljeđuju dva parametra, odnosno varijabla `state` korištenjem *spread* operatora kako bi se omogućio pristup svim korisničkim podacima te `dispatch` funkcija.

Parametar `children` predstavlja sve komponente koje se nalaze unutar autentikacijskog konteksta dok varijabla `state` sadržava sve korisničke podatke. Na taj način čitava aplikacija ima pristup autentikacijskom kontekstu.

```
import { createContext, useReducer, useEffect } from "react";

export const AuthContext = createContext();

export const authReducer = (state, action) => {
  switch (action.type) {
    case "LOGIN":
      return { user: action.payload };
    case "LOGOUT":
      return { user: null };
    default:
      return state;
  }
};

export const AuthContextProvider = ({ children }) => {
  const [state, dispatch] = useReducer(authReducer, {
    user: null,
  });

  useEffect(() => {
    const user = JSON.parse(localStorage.getItem("user"));
    if (user) {
      dispatch({ type: "LOGIN", payload: user });
    }
  }, []);

  return (
    <AuthContext.Provider value={{ ...state, dispatch }}>
      {children}
    </AuthContext.Provider>
  );
};
```

Ispis 11: Autentikacijski kontekst

Svi korisnički podatci spremljeni su prilikom logiranja u aplikaciju unutar lokalne pohrane (engl. *local storage*) što je moguće vidjeti u ispisu 12.

Kako bi se samo jednom provjerilo postoje li korisnički podatci spremljeni unutar lokalne pohrane te prosljedilo podatke aplikacijskim komponentama upotrebom autentikacijskog konteksta, koristi se `useEffect` kuka koja se izvršava samo pri prvom renderiranju komponente. Ukoliko korisnički podatci postoje, prosljeđuju se svim komponentama upotrebom `dispatch`

funkcije.

Primjer upotrebe autentikacijskog konteksta prilikom logiranja u aplikaciju vidljiv je u ispisu 12.

Korisnički podatci dohvaćaju se s poslužiteljske strane, odnosno njezine odredišne točke locirane na ruti "/api/auth/login". Upotrebom HTTP POST zahtjeva, koristi se funkcija `fetch` uz upotrebu korisnikove lozinke i email adrese u obliku tijela zahtjeva. Ukoliko je zahtjev valjan na poslužiteljskoj strani, korisnički podatci se spremaju u lokalnu pohranu u obliku JSON objekta. Podatci se zatim prosljeđuju svim komponentama upotrebom funkcije `dispatch`.

```
  const { dispatch } = useAuthContext();

  const login = async (email, password) => {
    setIsLoading(true);
    setError(null);

    const response = await fetch("/api/auth/login", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        email,
        password,
      }),
    });

    const json = await response.json();

    if (!response.ok) {
      console.log("error", error);
      setIsLoading(false);
      setError(json.error);
    }
    if (response.ok) {
      localStorage.setItem("user", JSON.stringify(json));

      dispatch({ type: "LOGIN", payload: json });

      setIsLoading(false);
      navigate("/Home");
    }
  };
  return { login, isLoading, error };
}
```

Ispis 12: Upotreba Autentikacijskog konteksta prilikom logiranja

3.6. Autorizacija

Kako bi se određene rute zaštitile od neautoriziranog pristupa i bile dostupne samo prijavljenim korisnicima, dodane su *middleware* funkcije na poslužiteljskoj strani te autorizacijska zaglavlja na klijentskoj strani.

3.6.1. Zaštita ruta na poslužiteljskoj strani

Način na koji funkcija u ispisu 13 radi je provjera korisničkih podataka prilikom zahtjeva, odnosno korisnički podatci moraju sadržavati valjani JWT token kako bi pristupili zaštićenoj ruti. Iz zaglavlja zahtjeva potrebno je dobiti autorizacijska svojstva poslana s poslužiteljske strane koja se spremaju u varijablu `authorization`.

```
import jwt from "jsonwebtoken";
import { User } from "../models/UserModel.js";

export const requireAuth = async (req, res, next) => {
  const { authorization } = req.headers;

  if (!authorization) {
    return res.status(401).json({ error: "Authorization token required" });
  }

  const token = authorization.split(" ")[1];

  try {
    const { _id } = jwt.verify(token, process.env.TOKEN_SECRET);

    req.user = await User.findOne({ _id }).select("_id");
    next();
  } catch (error) {
    console.log(error);
    return res.status(401).json({ error: "Request is not authorized" });
  }
};
```

Ispis 13: Autorizacijska *middleware* funkcija

Funkcija prvo provjerava je li zaglavlje prazno te ukoliko je vraća status 401 koji označava neautorizirani zahtjev i poruku da je potreban autorizirajući token. S obzirom da bi primjer poslanog zaglavlja sadržavao string u kojem bi token bio na mjestu druge riječi i odvojen

razmakom, koristi se funkcija `split` koja prima parametar po kojem će razdvojiti string (u ovom slučaju razmak) i vraća niz od jednog elementa odnosno tokena. Primjer poslanog zaglavlja vidljiv je u ispisu 15.

Upotrebom funkcije `verify` provjerava se valjanost poslanog tokena, uz tajni string koji se koristio pri njegovoj kreaciji. Funkcija vraća podatke koji se nalaze u tokenu, odnosno korisnikov `id`. Zatim se provjerava postojanje korisnika u bazi podataka te upotrebom funkcije `next` prelazi na sljedeći *middleware*. Na taj način dohvaćeni korisnički podatci dostupni su svim funkcijama koje koriste `requireAuth`.

Primjer upotrebe `requireAuth` vidljiv je u ispisu 14 prilikom zaštite zahtjeva na rute koje pristupaju `appointment` kolekciji. Sve rute koje su definirane nakon upotrebe `use` funkcije s proslijeđenim `requireAuth` parametrom zahtijevaju autorizaciju.

```
import express from "express";

import {
  createAppointment,
  deleteAppointment,
  getAllAppointments,
  updateAppointment,
  getAppointmentsByMonth,
} from "../controllers/AppointmentController.js";

import { requireAuth } from "../middleware/requireAuth.js";

const router = express.Router();

router.get("/getByMonth", getAppointmentsByMonth);

router.use(requireAuth);

router.get("/getAll/:id", getAllAppointments);

router.post("/create/:email", createAppointment);
router.post("/update/:id", updateAppointment);

router.delete("/delete/:id", deleteAppointment);

export default router;
```

Ispis 14: Upotreba autorizacije na rutama zahtjeva

3.6.2. Zaštita ruta na klijentskoj strani

U ispisu 15 nalazi se primjer definiranja funkcije `fetchDoctors` u kojoj se dohvaćaju svi doktori s poslužiteljske strane. Kako bi se osigurao pristup ruti potrebno je prilikom zahtjeva unijeti zaglavlje koje sadržava korisnikov JWT token. Korisnički podatci na slici dohvaćeni su upotrebom spomenutog autentikacijskog konteksta. Upotrebom autorizacije putem zaglavlja zahtjeva, osiguran je pristup ruti samo autenticiranim korisnicima.

```
const fetchDoctors = async () => {
  const response = await fetch("/api/user/allDoctors", {
    headers: { Authorization: `Bearer ${user.token}` },
  });

  const json = await response.json();

  if (response.ok) {
    setDoctors(json);
  }
};
```

Ispis 15: Dodavanje autorizacijskog zaglavlja

Funkcije s definiranim zaglavljima koriste se u datotekama sa stranicama aplikacije upotrebom `useEffect` kuka. Primjer implementacije funkcije `fetchDoctors` vidljiv je u ispisu 16. U primjeru se prvo vrši provjera postojanja korisnika pri pristupanju ruti s klijentske strane, a zatim ukoliko korisnik postoji, poziva funkcija koja pristupa poslužiteljskoj strani i dohvaća podatke na temelju valjanosti JWT tokena.

```
useEffect(() => {
  if (user) {
    fetchDoctors();
  }
}, [user]);
```

Ispis 16: Upotreba funkcije s autorizacijskim zaglavljem

3.7. Izrada kalendara termina

Kako bi korisnici jednostavno upravljali pregledima i imali uvid u iste, instaliran je paket `react-schedule` izvršavanjem naredbe "`npm install @syncfusion/ej2-react-schedule`". Nakon instalacije dodane su i CSS reference u glavnu CSS datoteku, u ovom slučaju `styles.css` kao što je vidljivo u ispisu 17.

```
@import "../node_modules/@syncfusion/ej2-base/styles/material.css";
@import "../node_modules/@syncfusion/ej2-buttons/styles/material.css";
@import "../node_modules/@syncfusion/ej2-calendars/styles/material.css";
@import "../node_modules/@syncfusion/ej2-dropdowns/styles/material.css";
@import "../node_modules/@syncfusion/ej2-inputs/styles/material.css";
@import "../node_modules/@syncfusion/ej2-navigations/styles/material.css";
@import "../node_modules/@syncfusion/ej2-popups/styles/material.css";
@import "../node_modules/@syncfusion/ej2-react-schedule/styles/material.css";
```

Ispis 17: CSS reference potrebne za `Schedule` komponentu

U ispisu 18 prikazan je kôd korišten za izradu kalendara. Syncfusion omogućuje upotrebu modula koji služe za različiti prikaz vremena na kalendaru, za potrebe aplikacije uveden je prikaz dana i radnog tjedna upotrebom `Inject` komponente. Vrijeme dostupno za zakazivanje termina na kalendaru ograničeno je na radne sate ordinacije, upotrebom opcija `startHour` i `endHour` te postavljeno na vremenski period od 9:00 do 17:00 sati.

Objekt `scheduleObj` omogućuje upotrebu određenih metoda i utjecaj na izgled komponente. U kombinaciji s DOM manipulacijom unutar `onPopupObject` funkcije omogućio je preinake na zadanim postavkama kalendara, odnosno uklanjanje nepotrebnih polja u formi za odabir termina kao što su ponavljanje događaja i unos vremenske zone.

Objekt `eventSettings` služi da bi se se upravljalo postavkama kalendara i omogućuje njegovo populiranje, a njegova upotreba vidljiva je u ispisu 19. U ovom slučaju, njegovom upotrebom dodani su svi podataci o terminima dobiveni s poslužiteljske strane te je uređivanje i brisanje dopušteno na temelju rola.

Funkcijom `onActionBegin` implementirane su CRUD (engl. *Create, Read, Update and Delete*) operacije na terminima na temelju korisničkog zahtjeva na kalendaru.

Funkcijom `eventRender` stilizirane su boje dodanih termina na temelju statusa.

```
    return (
        <div className='schedule-control-section'>
          <div className='col-lg-9 control-section'>
            <div className='control-wrapper'>
              <ScheduleComponent
                className='schedule-full-width'
                height='650px'
                ref={scheduleObj}
                eventSettings={eventSettings}
                popupOpen={onPopupOpen}
                actionBegin={onActionBegin}
                eventRendered={onEventRendered}
              >
                <ViewsDirective>
               <ViewDirective option='Day' startHour='9:00'
                              endHour='17:00' />
                  <ViewDirective
                    option='WorkWeek'
                    startHour='9:00'
                    endHour='17:00'
                  />
                </ViewsDirective>
                <Inject services={[Day, WorkWeek]} />
              </ScheduleComponent>
            </div>
          </div>
          <div className='col-lg-3 property-section'>
            <PropertyPane>
              <table id='property' title='Properties'
                     style={{ width: "100%"     }}>
                <tbody>
                  <tr style={{ height: "50px" }}>
                    <td style={{ width: "100%" }}></td>
                  </tr>
                </tbody>
              </table>
            </PropertyPane>
          </div>
        </div>
      );
```

Ispis 18: Upotreba `Schedule` komponente

Uvid u termine različit je za pacijente, liječnike i administratore. Liječnici vide sve podatke o zakazanim terminima svojih pacijenata koje mogu mijenjati ili otkazati, pacijenti vide sve termine odabranog liječnika, ali uz restrikcije mijenjanja i otkazivanja dok administratori imaju uvid u termine svih liječnika uz mogućnost mijenjanja i brisanja. Podatci se filtriraju na poslužiteljskoj strani na temelju korisnikove role.

Upotrebom atributa kalendarskih postavki `allowEditing` i `allowDeleting` vidljivim u ispisu 19 dozvoljeno je uređivanje svim autenticiranim korisnicima osim pacijentima - dakle administratorima i liječnicima. Atributom `dataSource` populira se objekt koji sadržava sve termine dok je dodavanje termina dozvoljeno svim autenticiranim korisnicima.

```
useEffect(() => {
  const fetchAppointments = async () => {
    const response = await fetch(`/api/appointment/getAll/${user?._id}`, {
      headers: { Authorization: `Bearer ${user.token}` },
    });
    const json = await response.json();

    if (response.ok) {
      setAppointments(json);
      setEventSettings(() => ({
        allowAdding: true,
        allowEditing: user?.role !== userRole.Patient ? true : false,
        allowDeleting: user?.role !== userRole.Patient ? true : false,
        dataSource: [...json],
      }));
    }
  };
```

Ispis 19: Dohvaćanje podataka termina i primjena restrikcija na temelju rola

Kako bi osjetljivi korisnički podatci ostali zaštićeni, pacijenti ne mogu vidjeti podatke drugih korisnika, već im se prikazuju zasivljeni uz generički natpis o zauzetosti termina. Primjer izmjene podataka termina prikazan je u ispisu 20.

Termini se prikazuju u različitim bojama dodavanjem polja `categoryColor` na temelju njihovog statusa, odnosno narančasti termini su u zakazanom statusu dok su zeleni termini odobreni od strane liječnika. Pacijenti mogu vidjeti sve podatke koje su unijeli odnosno početak i kraj termina, naslov, lokaciju i kratki opis.

```
      let formattedAppointments = appointments.map((appointment) => {
        let categoryColor;
        if (
          user.role === userRole.Patient &&
          appointment.patientID.toString() !== user._id.toString()
        ) {
          appointment.Subject = "Scheduled appointment";
          categoryColor = Color.Gray;
        } else if (appointment.State === appointmentStatus.Accepted) {
          categoryColor = Color.Green;
        } else {
          categoryColor = Color.Orange;
        }

        const formattedAppointment = {
          ...appointment.toJSON(),
          CategoryColor: categoryColor,
        };

        if (user.role === userRole.Patient &&
          appointment.patientID.toString() !== user._id.toString()) {
          delete formattedAppointment.Location;
          delete formattedAppointment.Description;
        }
```

Ispis 20: Izmjena podataka termina

3.8. Obavijesti o terminima

Obavijesti o terminima dostupne su korisnicima putem elektroničke pošte i unutar same aplikacije, a primaju se prilikom kreacije, prihvaćanja i odbijanja termina.

Primjer stvaranja obavijesti prilikom uspješne kreacije termina vidljiv je u ispisu 21, a nalazi se unutar dokumenta `useCreateAppointment.js`.

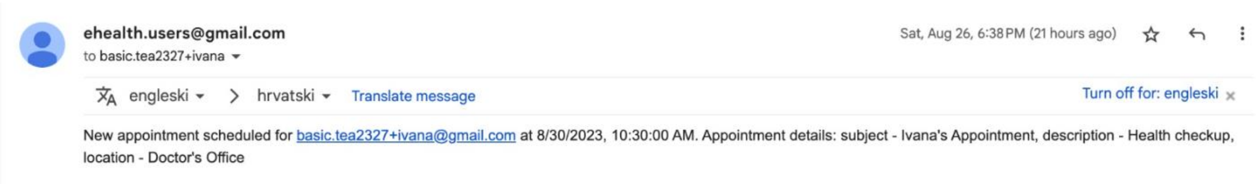
Prvo se provjerava uspješnost zahtjeva pri kreaciji termina pomoću statusa odgovora, u ovom slučaju varijable `response`. Kada je odgovor valjan, putem zahtjeva se pristupa ruti "`/api/notification/create/:id`" kojoj je kao parametar id proslijeđen jedinstveni identifikator termina.

Obavijest se kreira sadržavajući naslov, liječnikov i pacijentov identifikator, vrijeme početka termina te njegovu temu, opis i lokaciju.

```
          if (!response.ok) {
            setError(json.error);
          } else {
            await fetch(`/api/notification/create/${json.appointment._id}`, {
              method: "POST",
              headers: {
                "Content-Type": "application/json",
                Authorization: `Bearer ${user.token}`,
              },
              body: JSON.stringify({
                doctorID: json.appointment.doctorID,
                patientID: json.appointment.patientID,
                StartTime: data.StartTime,
                Subject: data.Subject,
                Description: data.Description,
                Location: data.Location,
                Text: "New appointment scheduled",
              }),
            });
          }
```

Ispis 21: Stvaranje obavijesti prilikom uspješne kreacije termina

Primjer primanja obavijesti putem elektroničke pošte pri kreaciji novog termina vidljiv je na slici 3. Obavijest sadržava sve podatke o terminu koji je zakazan kako bi korisnici mogli lakše pratiti liječničke preglede.



Slika 3: Primanje obavijesti putem elektroničke pošte

Osim spomenutih obavijesti, kreirani su podsjetnici zakazani za određeno vrijeme prije termina pregleda. Kako bi se podsjetnici slali na temelju vremena preostalog do korisnikovog termina instaliran je paket `node-cron` izvršavanjem naredbe “`npm install --save node-cron`”. Node Cron je paket koji omogućuje jednostavno zakazivanje zadataka u NodeJS-u, a utemeljen je na GNU (engl. *GNU's not Unix*) crontab-u.

Primjer upotrebe paketa node-cron nalazi se u ispisu 22.

```
const checkTimeToNotify = async () => {
  try {
    const now = new Date();
    const nowUTC = new Date(
      Date.UTC(
        now.getUTCFullYear(),
        now.getUTCMonth(),
        now.getUTCDate(),
        now.getUTCHours(),
        now.getUTCMinutes(),
        0,
        0
      )
    );
    const oneHourFromNow = new Date(nowUTC.getTime() + 60 * 60 * 1000);
    const oneDayFromNow = new Date(nowUTC.getTime() + 24 * 60 * 60 * 1000);

    const appointments = await Appointment.find({
      $and: [
        {
          $or: [{ StartTime: oneHourFromNow }, { StartTime: oneDayFromNow }],
        },
        { status: appointmentStatus.Accepted },
      ],
    }).sort({ StartTime: 1 });

    for (const appointment of appointments) {
      await createNotificationForAppointment(appointment);
    }

    console.log("Notifications created for appointments", appointments);
  } catch (error) {
    console.error("Error fetching appointments:", error);
  }
};

var task = cron.schedule("* 8-15,1 * * 1-5", checkTimeToNotify);
export default task;
```

Ispis 22: Upotreba paketa node-cron

U ispisu 22 koristi se funkcija `schedule` kojoj su kao parametri proslijeđeni vrijeme obavljanja zadatka i funkcija `checkTimeToNotify` koja predstavlja zadatak koji će se izvršiti.

Vrijeme izvršavanja zadatka mora biti definirano u formatu sekunda, minuta, sati, dana u mjesecu, mjeseci te dana u tjednu. U funkciji je definirano vrijeme u kojem će se zadatak izvršavati svake minute u vremenskom periodu od 8:00 do 15:00 sati u toku radnog tjedna zato što termini mogu biti zakazani jedino u okvirima radnog vremena liječničke ustanove, odnosno radnim danima od 9:00 do 16:00 sati.

Zadatak se u radnim danima prvi put izvrši u 8:00 sati uz provjeru vremena svih dostupnih termina. Upotrebom varijabli `oneHourFromNow` i `oneDayFromNow` provjerava se postojanje svh termina trenutnog i sljedećeg dana u vremenu od 9:00 sati koji se spremaju u varijablu `appointments`. S obzirom da varijabla predstavlja objekt u kojem su spremljeni svi termini, zakazani sat i dan vremena od vremena izvršavanja zadatka, iterativno se šalju pojedinačne obavijesti upotrebom funkcije `createNotificationForAppointment`.

Ponavljanje izvršavanja zadatka se obavlja svake minute u danu do 15:00 sati kada se vrši zadnja provjera odnosno postojanje termina u vremenu od 16:00 sati.

3.9. Izrada pacijentovog kartona

Za potrebe aplikacije implementirano je spajanje na ICD (engl. *International Classification of Diseases*) API s podacima o dijagnozama koje se koriste u međunarodnoj klasifikaciji bolesti i srodnih zdravstvenih problema. API podatci se dohvaćaju kako bi se unijelo pacijentovo stanje pri izvršenom pregledu u pacijentov karton. Osim toga omogućeno je dodavanje slika prikupljenih za vrijeme pregleda te kratkog opisa pregleda u kojem bi bio sadržan liječnikov uvid o izvršenom pregledu i propisivanje potrebne terapije.

3.9.1. Spajanje na API s dijagnozama

Kako bi se upotrebom API poziva povukli podatci o nazivima dijagnoza, upotrebljeni su korisnički podatci dobiveni prilikom registracije na ICD API stranici. Upotrebom korisničkih podataka stvara se autorizacijski token za pristup API-ju. Podatci se zatim parsiraju kako bi se povukli samo nazivi dijagnoza upotrebljeni u daljnjem razvoju aplikacije.

U ispisu 23 vidljiva je upotreba HTTP POST zahtjeva prilikom pristupanja podacima i

upotreba autorizacijskog tokena kako bi se dohvatili podatci u funkciji `fetchDataFromApi`.

```
async function fetchDataFromApi(url, accessToken) {
  try {
    const response = await fetch(url, {
      headers: {
        "API-Version": "v2",
        Accept: "application/json",
        Authorization: `Bearer ${accessToken}`,
        "Accept-Language": "en",
      },
    });
```

Ispis 23: Dohvaćanje podataka upotrebom tokena

Upotreba funkcije `fetchDataFromApi` vidljiva je u ispisu 24.

```
const apiResponse = await fetchDataFromApi(
      apiEndpoint,
      tokenData.access_token
    );
    const data = await apiResponse.json();

    if (apiResponse.ok) {
      if (data.child && Array.isArray(data.child)) {
        data.child = data.child.map((childEntityUrl) =>
          childEntityUrl.replace(/^http:/, "https:")
        );
        const childTitles = await Promise.all(
          data.child.map(async (childEntityUrl) => {
            try {
              const childApiResponse = await fetchDataFromApi(
                childEntityUrl,
                tokenData.access_token
              );
              const childEntityData = await childApiResponse.json();
              return getChildTitle(childEntityData);
            } catch (error) {
              console.error("Error fetching/parsing child entity:", error);
              return "Title not available";
            }
          })
        );
```

Ispis 24: Dohvaćanje i parsiranje podataka

Prilikom uspješnog zahtjeva podatci se dohvaćaju i parsiraju putem pristupa svim krajnjim točkama na API-ju. Upotrebom funkcije `all` na `Promise` objektu osigurano je čekanje prilikom istovremenog obrađivanja više URL-ova uz upotrebu asinkrone operacije.

Pristupom svim URL-ovima i parsiranjem podataka dohvaćeni su nazivi dijagnoza uz upotrebu funkcije `getChildTitle` čija je implementcija vidljiva u ispisu 25.

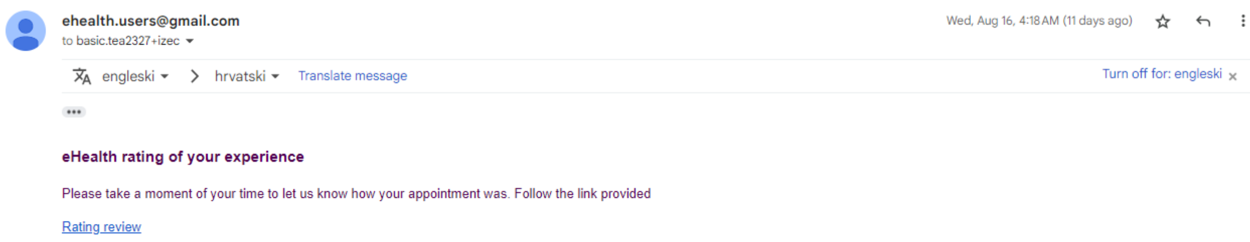
Upotrebom funkcije `fetchDataFromApi` dobiveni JSON objekt se šalje u funkciju `getChildTitle` u kojoj se upotrebom ključa `@value` dobiva vrijednost naziva dijagnoze te se na taj način dohvaćaju svi nazivi dijagnoza.

```
function getChildTitle(childEntityData) {
  if (childEntityData.title && childEntityData.title["@value"]) {
    return childEntityData.title["@value"];
  } else {
    return "Title not available";
  }
}
```

Ispis 25: Pomoćna funkcija za dohvaćanje naziva dijagnoza

3.10. Ocjenjivanje obavljenog pregleda

Nakon izvršenog pregleda i dodavanja podataka u pacijentov karton, pacijentu se putem elektroničke pošte šalje poveznica koja ga vodi na stranicu aplikacije gdje je moguće ocijeniti i dati uvid u obavljeni pregled. Sadržaj primljene elektroničke pošte moguće je vidjeti na primjeru sa slike 4.



Slika 4: Primljena elektronička pošta s linkom na formu za ocjenjivanje

Kako bi se izbjegla zloupotreba i onemogućilo dodavanje više ocjena, poveznica kojom se pristupa formi za ocjenjivanje sadrži generirani token i pacijentov jedinstveni identifikator na temelju kojih se procjenjuje može li korisnik ocijeniti obavljeni pregled.

U ispisu 26 je primjer generiranja tokena te umetanje istog u poveznicu koja se šalje pacijentu putem elektroničke pošte.

```
    let token = await Token.findOne({ userID: patientID });
        if (!token) {
          token = await new Token({
            userID: patientID,
            token: crypto.randomBytes(32).toString("hex"),
          }).save();
        }

        const patient = await Patient.findById(patientID);

        const link = `http://localhost:3000/Rating/${encodeURIComponent(
          patientID
        )}/${encodeURIComponent(token.token)}`;
        sendEmail(
          patient.email,
          "Rate your experience",
          `
            <h3>eHealth rating of your experience</h3>
           <p>Please take a moment of your time to let us know
         how your appointment was. Follow the link provided</p>
            <a href="${link}"> Rating review </a>
          `
        );
```

Ispis 26: Slanje elektroničke pošte s poveznicom za ocjenjivanje uz upotrebu tokena

Nakon što pacijent popuni dostupni obrazac za ocjenjivanje s klijentske strane šalje se zahtjev poslužitelju koji sadržava token i jedinstveni identifikator dohvaćene iz poveznice kojom je korisnik pristupio stranici. Dohvaćanje tokena i identifikatora iz URL-a vidljivo je u ispisu 27.

```
const { id, token } = useParams();
```

Ispis 27: Dohvaćanje identifikatora i tokena na klijentskoj strani

Upotrebom funkcije `createRating` koja je vidljiva u ispisu 28 vrši se provjera tokena i postojanja korisnika koji šalje zahtjev. Ukoliko je poslani token valjan i korisnik postoji, pacijentovi podatci odnosno ime i prezime, uvid u obavljeni pregled i ocjena se spremaju u bazu podataka.

Upotrebljeni token se pri uspješnom zahtjevu i upisu ocjene briše iz baze podataka kako bi se onemogućilo naknadno dodavanje ocjena na istoj poveznici od strane istog pacijenta.

```
export const createRating = async (req, res) => {
  const { id } = req.params;
  const { review, rateNumber } = req.body;

  try {
    const patient = await Patient.findById(id);
    const token = await Token.findOne({
      userID: patient._id,
      token: req.params.token,
    });

    if (!token)
      return res.status(400).json({ error: "Invalid link or expired" });

    const rating = await Rating.create({
      name: `${patient.firstName} ${patient.lastName}`,
      rateNumber,
      review,
      patientID: patient._id,
      userID: patient.doctorID,
    });

    await token.delete();

    res.status(200).json(rating);
  } catch (error) {
    console.log(error);
    res.status(404).json({ message: error.message });
  }
};
```

Ispis 28: Provjera korisnikovog identifikatora i tokena prilikom spremanja ocjene

3.11. Implementiranje razgovora u stvarnom vremenu

Kako bi se implementirala funkcionalnost razgovora između liječnika i pacijenta u stvarnom vremenu instaliran je paket `socket.io` izvršavanjem naredbi “`npm install socket.io`” s poslužiteljske strane i “`npm install socket.io-client`” s klijentske strane.

Upotrebom `Socket` klase stvoren je `socket` poslužitelj na portu 8900, što je moguće vidjeti u ispisu 29, na koji se upotrebom funkcije `io` spaja klijentska strana, kao što je vidljivo u ispisu 30. S obzirom da se vrši zahtjev na *socket* poslužitelj, URL koji se koristi pri spajanju sadrži ws (engl. *web socket*) umjesto HTTP protokola.

Funkcija `emit` prima naziv *eventa* koji *socket* sluša dok se upotrebom funkcije `on` na klijentskoj strani primaju podatci s poslužiteljskog *socketa*. Kako bi se omogućilo slanje privatnih poruka s poslužiteljske strane upotrebljena je funkcija `to`.

S obzirom da se jedinstveni identifikator *socketa* mijenja pri svakom spajanju na poslužitelj stvoren je niz korisnika na poslužitelju spremljen u varijablu `users`, u koji se pohranjuju podatci pri svakom spajanju korisnika. Upotrebom funkcije `emit` s klijentske strane, kojoj se prosljeđuje naziv *eventa* `addUser`, šalje se korisnikov jedinstveni identifikator. Identifikator se zatim dohvaća na poslužitelju upotrebom funkcije `on`, koja kao parametar prima naziv *eventa* poslanog s klijentske strane.

Kako bi se spriječilo višestruko dodavanje istih korisničkih podataka stvorena je funkcija `addUser` koja prije spremanja korisničkih podataka provjerava postoje li korisnički podatci u nizu te ukoliko ne postoje, u niz dodaje korisnikov i *socketov* identifikator.

Upotrebom funkcije `emit` na poslužitelju koja kao parametar prima naziv *eventa* `getUsers` podatci o spojenim korisnicima šalju se klijentskoj strani. Klijent zatim dohvaća poslane podatke upotrebom funkcije `on` kojoj prosljeđuje naziv *eventa* upotrebljenog na poslužitelju.

Pri slanju poruke s klijentske strane, šalju se identifikatori pošiljatelja, primatelja te poslana poruka poslužitelju. Parametri se šalju upotrebom funkcije `emit`, kojoj se prosljeđuje naziv *eventa* `sendMessage`. Poslužitelj zatim koristi isti naziv *eventa* za dohvaćanje podataka.

Identifikator *socketa* na koji je potrebno odaslati poruku s poslužitelja dobiven je

upotrebom funkcije `getUser` kojoj se kao parametar šalje primateljev identifikator poslan s klijentske strane.

Poslužitelj zatim odašilje privatnu poruku klijentu koja sadržava pošiljateljev identifikator i poruku koja se zatim prihvaća upotrebom istog naziva *eventa* odnosno `getMessage`.

```
  useEffect(() => {
    socket.current = io("ws://localhost:8900");
    socket.current.on("getMessage", (data) => {
      setArrivalMessage({
        sender: data.senderId,
        text: data.text,
        createdAt: Date.now(),
      });
    });
  }, []);

  useEffect(() => {
    socket.current.emit("addUser", user?._id);
    socket.current.on("getUsers", (users) => {});
  }, [user]);

const handleSubmit = async (e) => {
    if (!user) return;

    e.preventDefault();

    const res = await sendMessage(newMessage, user._id, currentChat._id);

    const receiverId =
      currentChat?.firstUserID !== user._id
        ? currentChat.firstUserID
        : currentChat.secondUserID;
    socket.current.emit("sendMessage", {
      senderId: user._id,
      receiverId,
      text: newMessage,
    });

    if (!error) {
      setMessages([...messages, res]);
      setNewMessage("");
    }
  };
```

Ispis 29: Upotreba *socketa* na klijentskoj strani

Upotrebom ugrađenog naziva *eventa* `disconnect` i funkcijom `removeUser` brišu se podatci korisnika koji više nije spojen na poslužitelja.

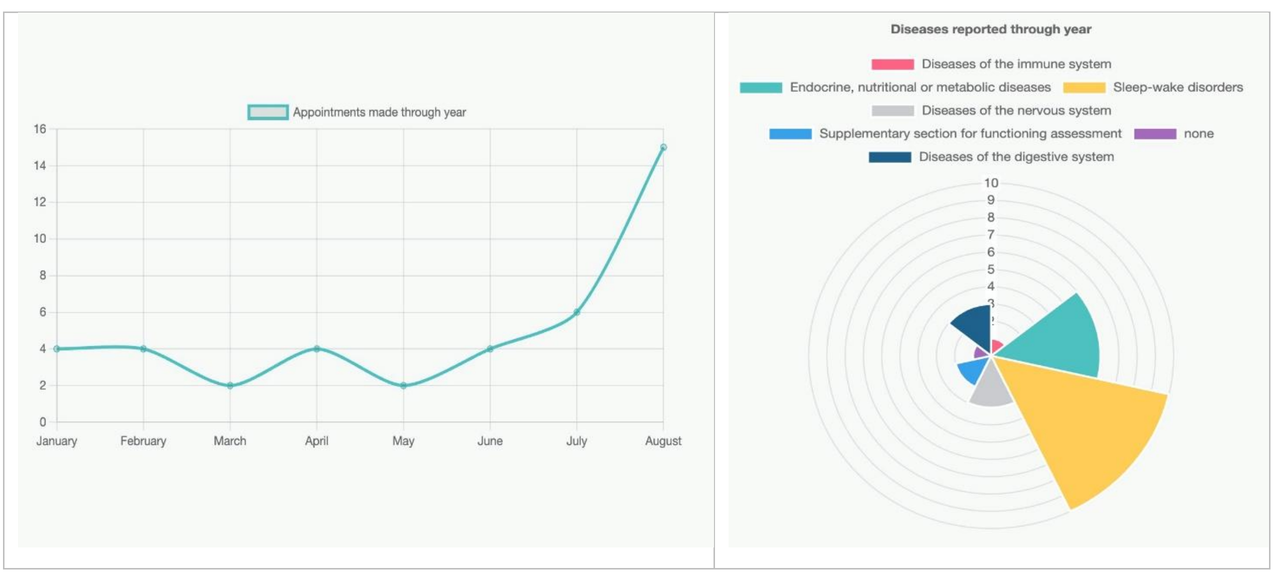
```
import { Server } from "socket.io";

const io = new Server(8900, {
  cors: {
    origin: "http://localhost:3000",
  },
});

let users = [];

const addUser = (userId, socketId) => {
  !users.some((user) => user.userId === userId) &&
    users.push({ userId, socketId });
};

const removeUser = (socketId) => {
  users = users.filter((user) => user.socketId !== socketId);
};

const getUser = (userId) => {
  return users.find((user) => user.userId === userId);
};

io.on("connection", (socket) => {
  socket.on("addUser", (userId) => {
    addUser(userId, socket.id);
    io.emit("getUsers", users);
  });

  socket.on("sendMessage", ({ senderId, receiverId, text }) => {
    const user = getUser(receiverId);
    io.to(user.socketId).emit("getMessage", {
      senderId,
      text,
    });
  });

  socket.on("disconnect", () => {
    removeUser(socket.id);
    io.emit("getUsers", users);
  });
});
```

Ispis 30: Upotreba *socketa* na poslužiteljskoj strani

3.12. Uvid u statističke podatke

Kako bi korisnici imali uvid u statističke podatke prikupljene unutar aplikacije vidljive u obliku grafova instaliran je paket `chart.js` izvršavanjem naredbe “`npm install chart.js`”. Dodane su interkativne komponente `LineChart` i `PolarAreaChart` odnosno linijski i kružni grafovi koji su populirani podacima o izvršenim pregledima.

Linijski graf sadržava podatke o broju pregleda obavljenih na mjesečnoj razini kroz protekli godišnji period kao što je vidljivo na slici 5. Kružni dijagram prikazuje broj različitih dijagnoza koje su ustanovljene na izvršenim pregledima što je moguće vidjeti na slici 6. Kako bi se osigurala povjerljivost i privatnost prilikom prikupljanja podataka, dijagrami ne sadrže osjetljive korisničke podatake.



Slika 5: Linijski graf

Slika 6: Kružni graf

4. Zaključak

Izradom ovog završnog rada predstavljena je mogućnost upotrebe hibridnog ili u cijelosti elektroničkog načina poslovanja općim zdravstvenim ustanovama koja bi osigurala lakši i pristupačniji način obavljanja svakodnevnih poslova, ali i praćenja zdravstvenih obaveza. Osim toga, upotrebom elektroničkog obrasca pregleda pojednostavnio bi se uvid u osobne podatke prikupljene prilikom pregleda.

Različite funkcionalnosti poput jednostavnosti interakcije između liječnika i pacijenta, vizualne privlačnosti pri zakazivanju termina te različitih vrsta obavijesti napravljene su u svrhu ugodnog korisničkog iskustva, koje bi doprinjelo povećanom broju korisnika aplikacije. Osim spomenutih, kroz aplikaciju su iskorištene i komponente koje bi omogućile sadržaj prilagođeniji korisniku kao što su upotreba paginacije, pretraživanja i *spinnera* prilikom učitavanja veće količine podataka.

Osiguravanjem dostupnosti iznošenja dojmova o obavljenom pregledu pridaje se veća sloboda pacijentima putem iznošenja vlastitog korisničkog iskustva na temelju kojeg bi liječnici mogli unaprijediti vlastiti razvoj i rad.

Mogućnost upotrebe najvećeg svjetskog registra paketa sadržanog u npm biblioteci olakšala je izradu aplikacije s obzirom da pruža mogućnost jednostavne instalacije paketa putem naredbenog retka uz opis funkcionalnosti i dokumentacijom istih. Osim upotrebe npm upravljača paketima lakšu izradu aplikacije omogućila je i upotreba unaprijed stiliziranih Tailwind CSS komponenti.

Ovaj završni rad također predstavlja i način za istraživanje i izučavanje novih tehnologija te njihovu primjenu. Kroz priloženi kôd i opise njegove implementacije omogućena je buduća referenca i polazna točka u izradi aplikacija u spomenutim tehnologijama, ali i mogućnost napretka u istim.

Literatura

[1] „NodeJS dokumentacija“

https://nodejs.org/en

- posjećeno 18.6.2023

[2] „ExpressJS dokumentacija“

https://expressjs.com/

- posjećeno 20.6.2023

[3] „NPM dokumentacija“

https://docs.npmjs.com/about-npm

- posjećeno 21.6.2023

[4] „Mongoose dokumentacija“

https://mongoosejs.com/docs/

- posjećeno 23.6.2023

[5] „ChartJS dokumentacija“

https://www.chartjs.org/docs/latest/

- posjećeno 25.6.2023

[6] „React-schedule dokumentacija“

https://www.npmjs.com/package/@syncfusion/ej2-react-schedule?activeTab=readme

- posjećeno 25.6.2023

[7] „Tailwind CSS dokumentacija“

https://tailwindcss.com/docs/guides/create-react-app

- posjećeno 27.6.2023

[8] „Socket.io dokumentacija“

https://socket.io/docs/v3/

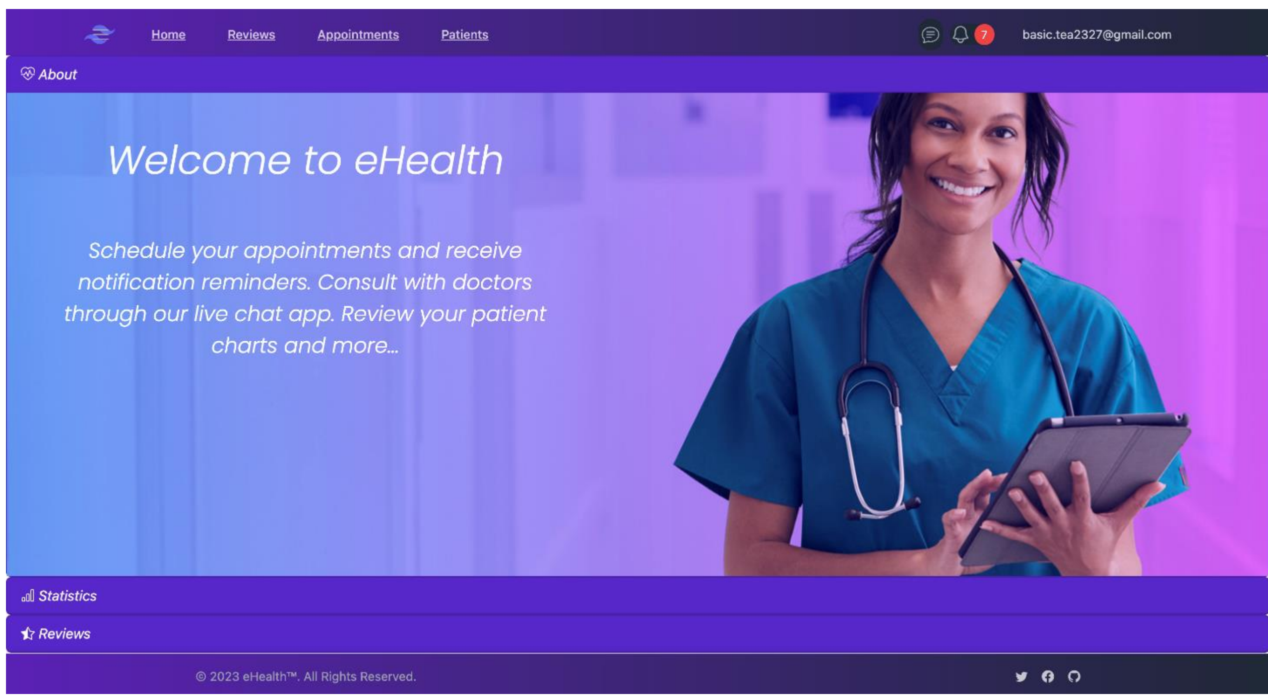
- posjećeno 29.6.2023

[9] „Cron“

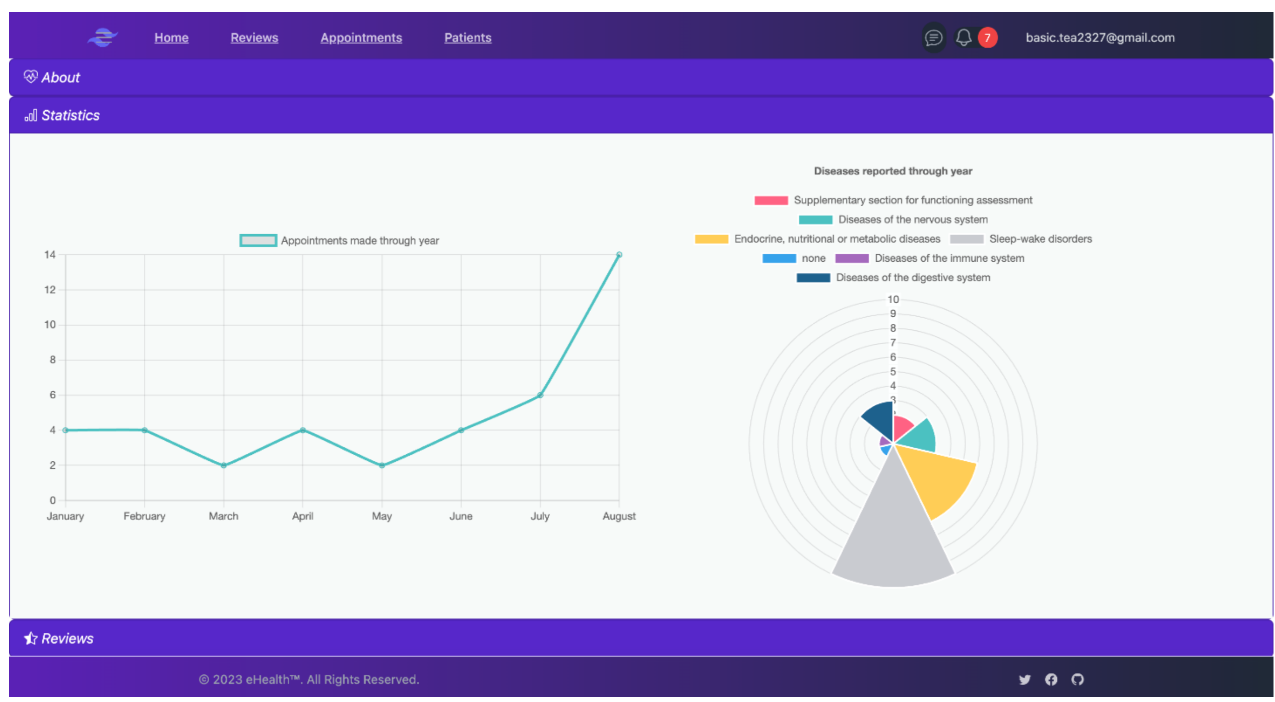
`https://crontab.guru/`

- posjećeno 30.6.2023

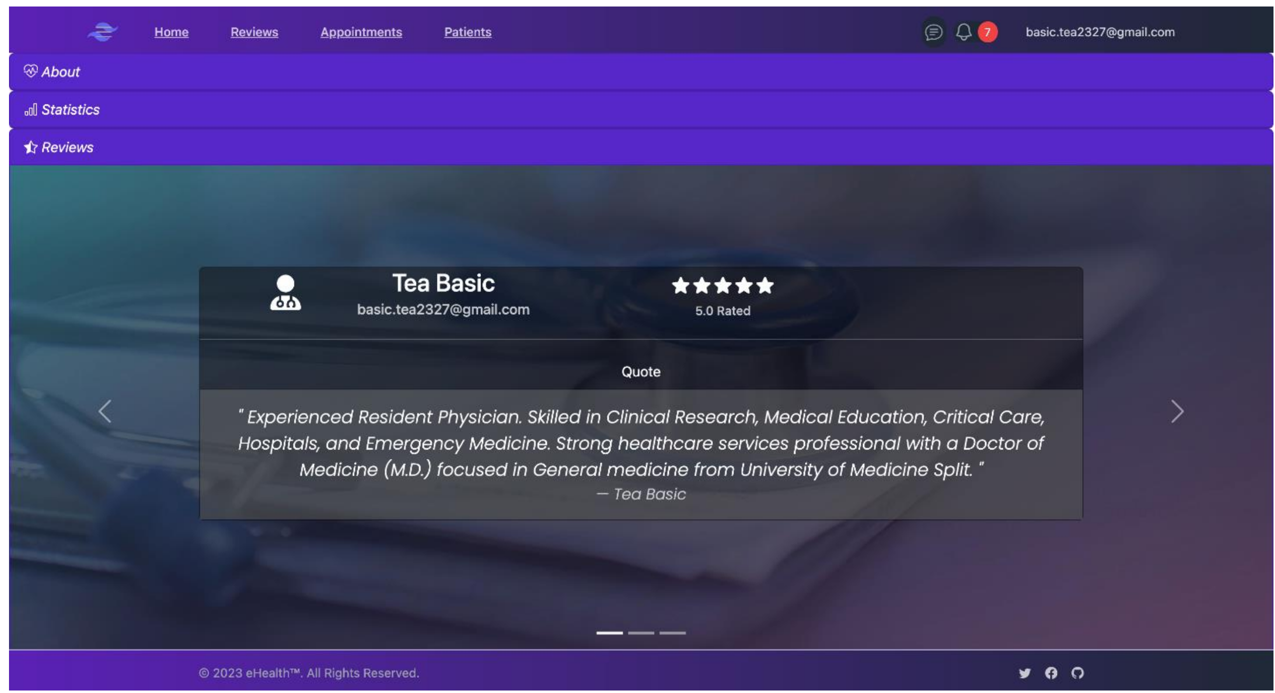
Dodatci



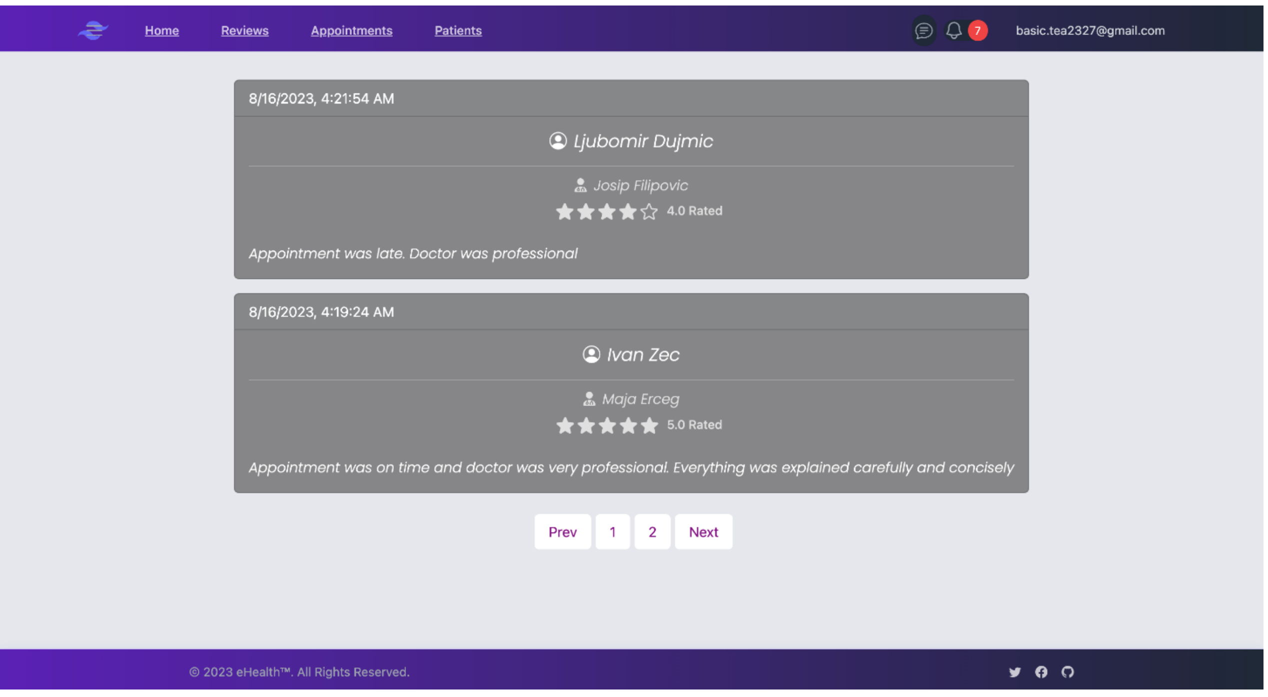
Slika 7: Naslovna stranica



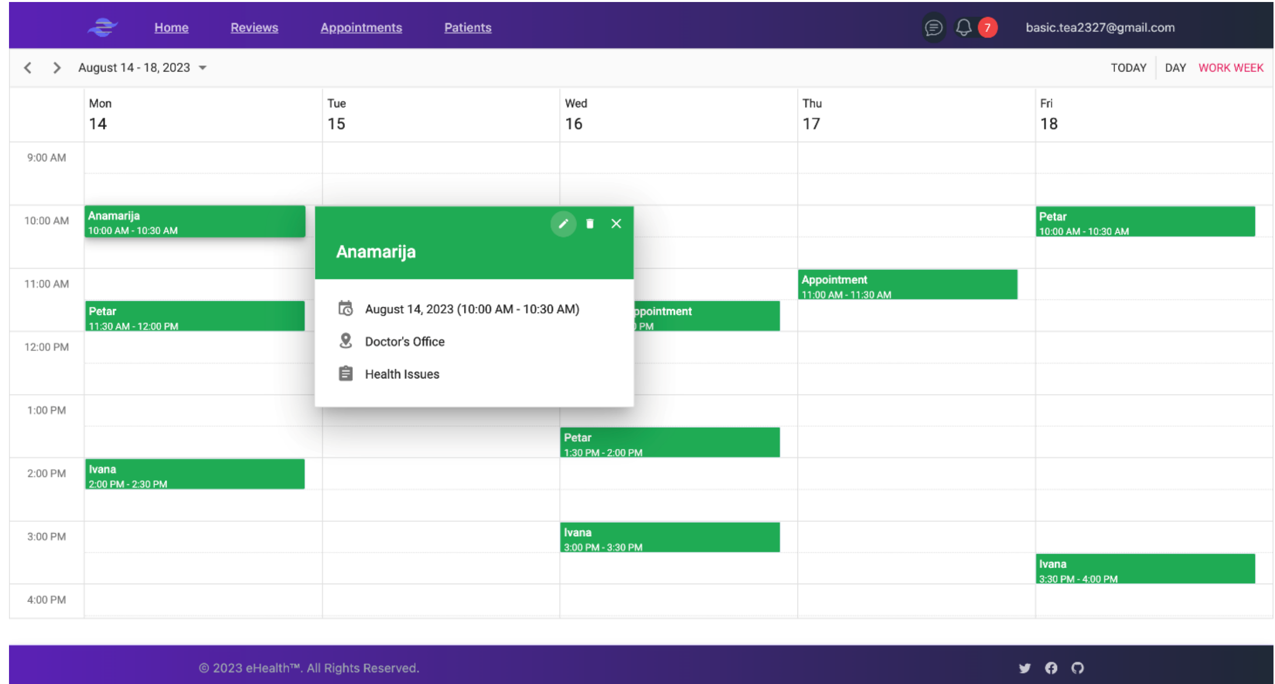
Slika 8: Statistički podatci



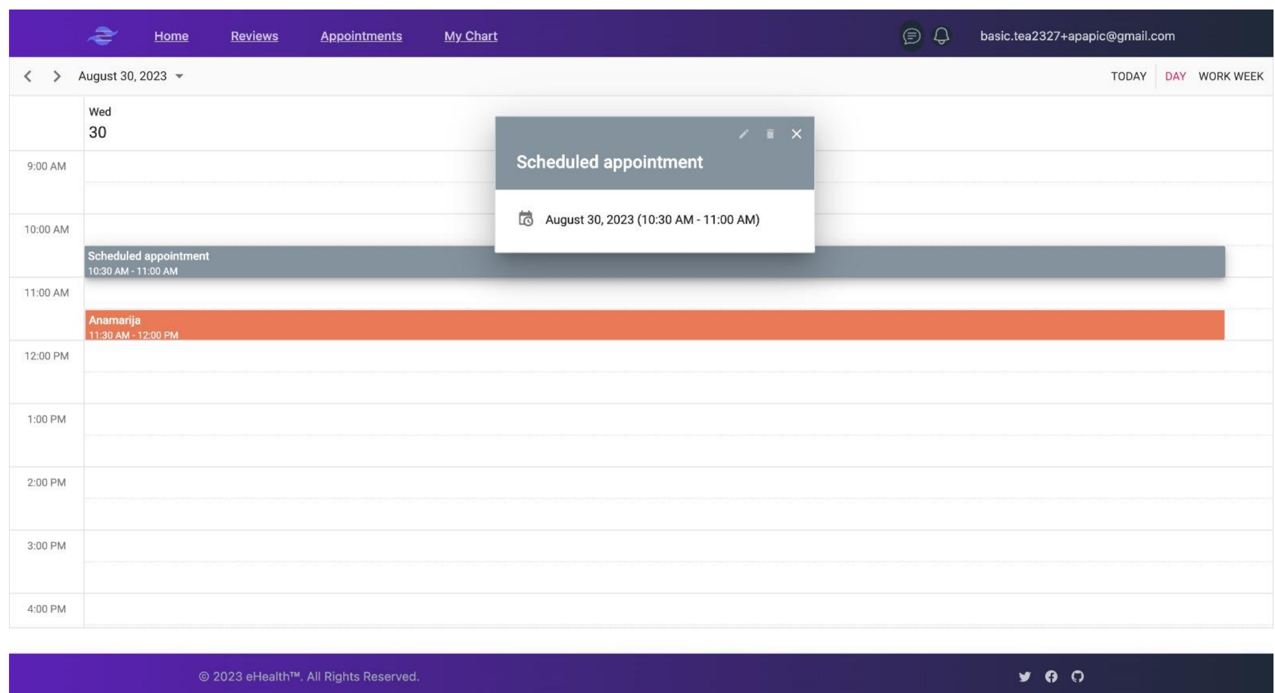
Slika 9: Prikaz prosjeka doktorovih ocjena i kratkog osobnog opisa



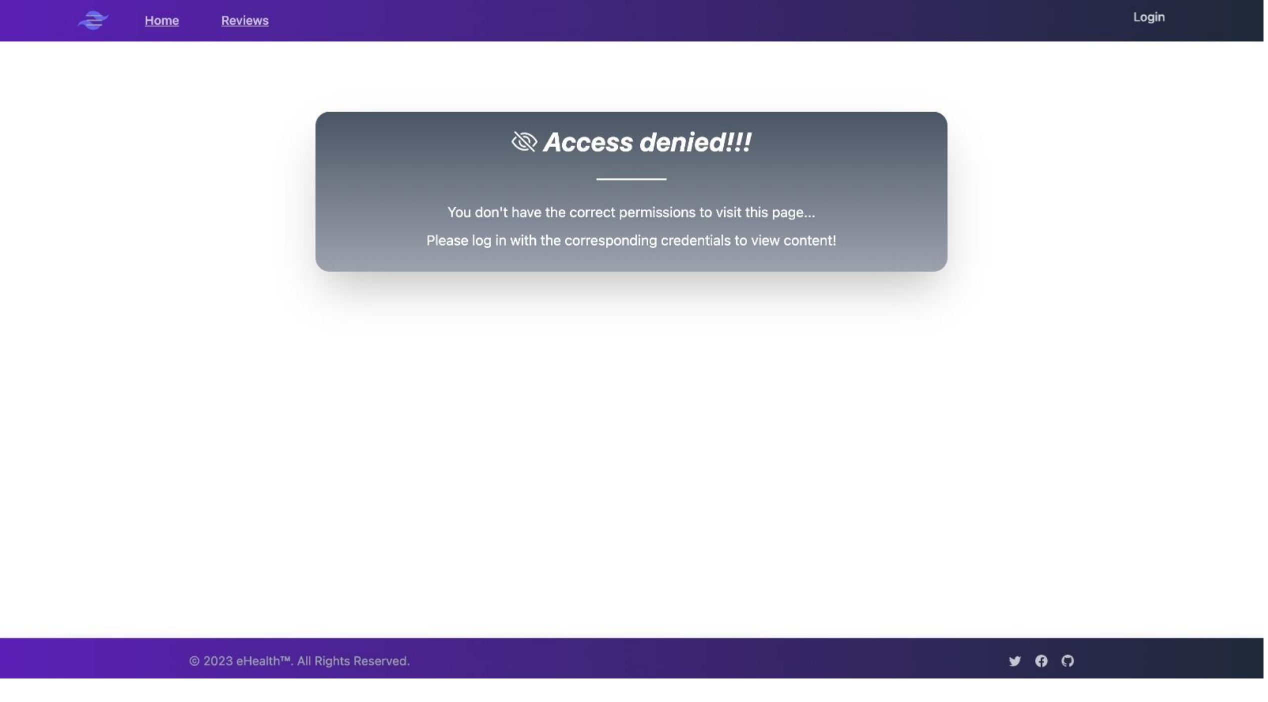
Slika 10: Stranica s korisničkim dojmovima



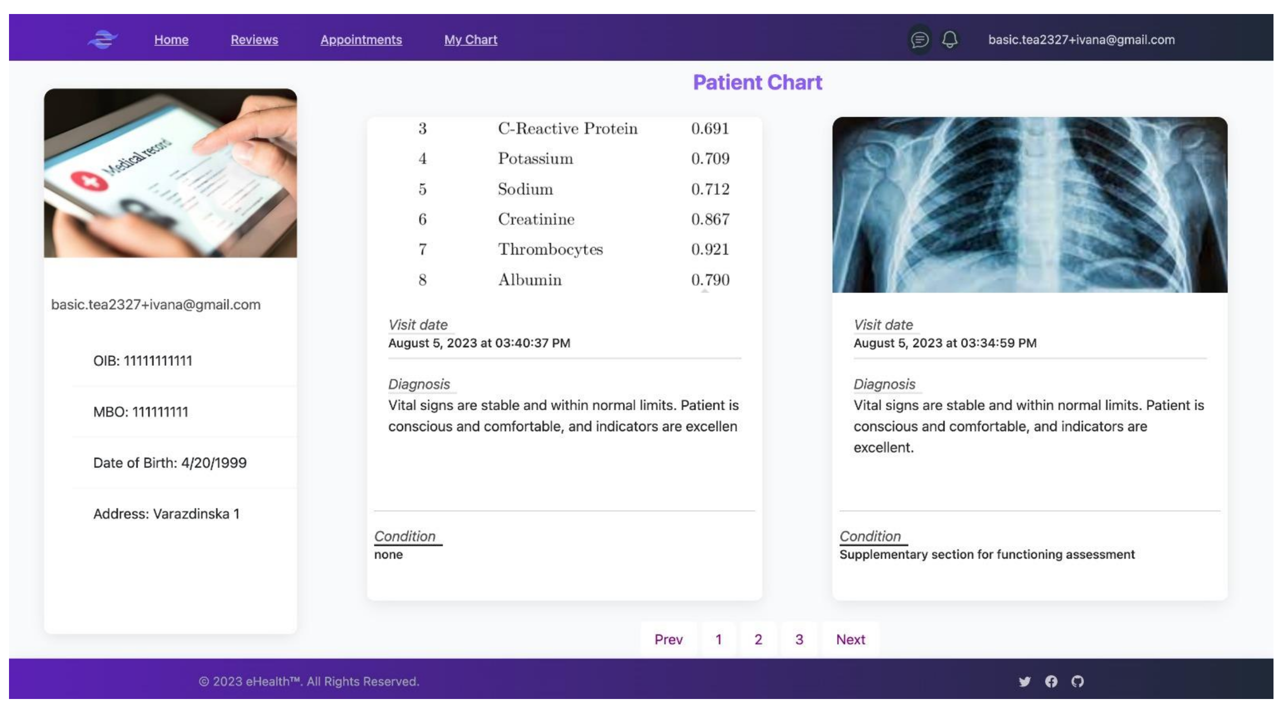
Slika 11: Kalendar termina u opciji prikaza radnog tjedna pregledavan od strane doktora



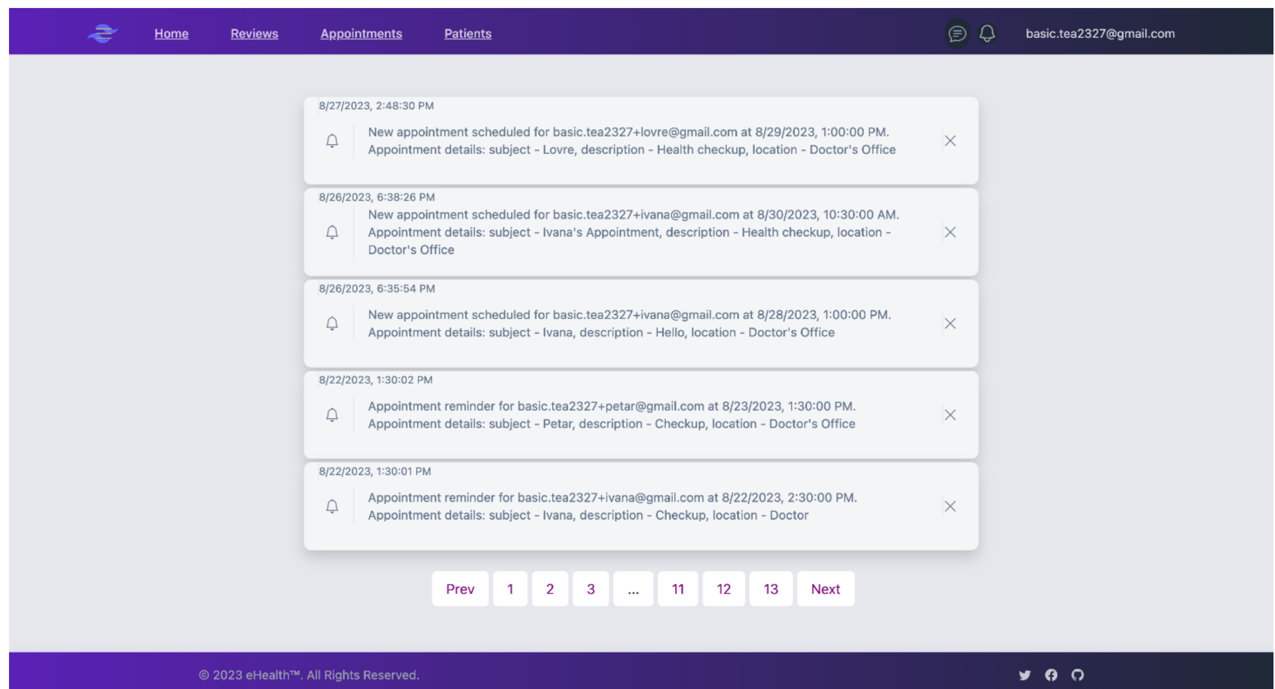
Slika 12: Kalendar termina u opciji dnevnog prikaza pregledavan od strane pacijenta



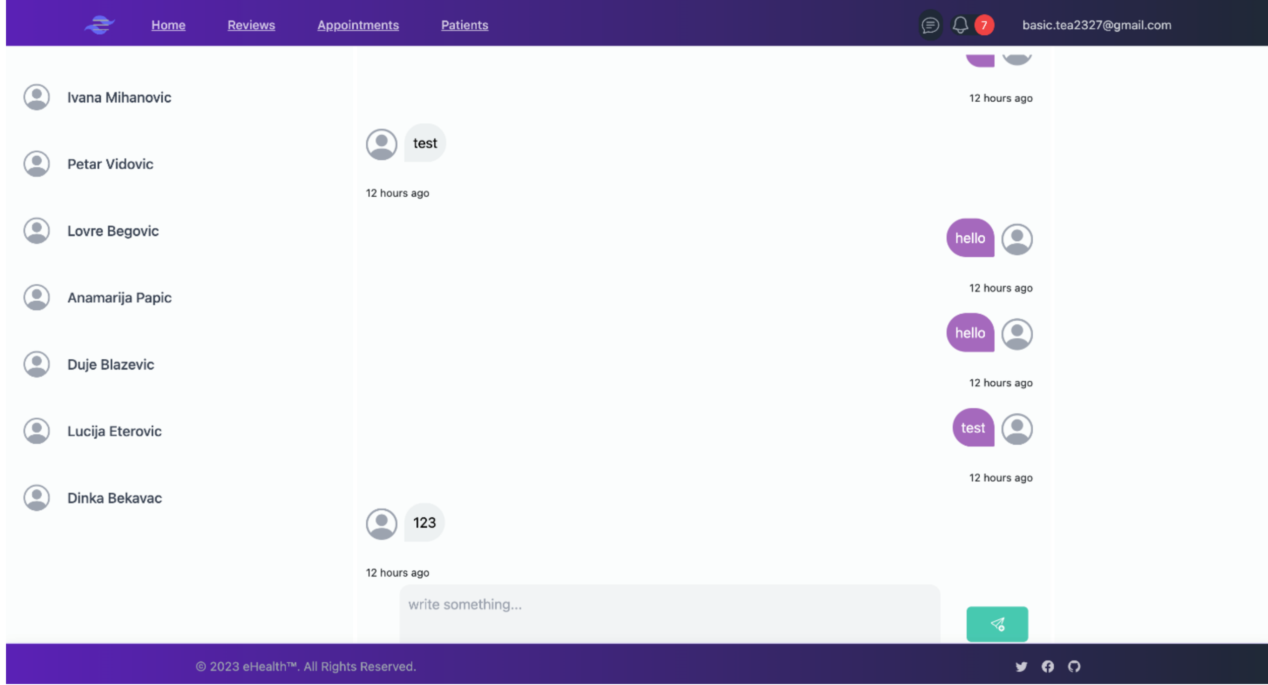
Slika 13: Prikaz aplikacije prilikom neautoriziranog zahtjeva



Slika 14: Pacijentovi podatci



Slika 15: Notifikacijski panel



Slika 16: Korisnički razgovori