

Izrada progresivnih web aplikacija

Jezernik, Anastasija

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: <https://um.nsk.hr/um:nbn:hr:217:788207>

Rights / Prava: [In copyright](#)/[Zaštićeno autorskim pravom.](#)

Download date / Datum preuzimanja: **2025-03-29**



Repository / Repozitorij:

[Repository of the Faculty of Science - University of Zagreb](#)



SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO–MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Anastasija Jezernik

IZRADA PROGRESIVNIH WEB
APLIKACIJA

Diplomski rad

Voditelj rada:
Izv. prof. dr. sc., Ivica Nakić

Zagreb, veljača, 2020.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

*Ovaj diplomski rad posvećujem svojoj obitelji i prijateljima, bez kojih ne bih bila ovdje.
Hvala vam*

Sadržaj

Sadržaj	iv
Uvod	1
1 Uvod u PWA	2
1.1 Što je PWA?	2
1.2 Značajke PWA	3
1.3 Prednosti (i mane) PWA	3
2 Pregled tehnologija za kreiranje PWA	5
3 Aplikacija	8
3.1 Ideja i opis aplikacije	8
3.2 Paketi i biblioteke	9
3.3 React	12
3.4 Firebase	18
3.5 Korisnički računi	20
3.6 Firestore - baza podataka	22
3.7 Manifest datoteka	25
3.8 App shell	27
3.9 Service worker	27
3.10 React-Bootstrap	29
3.11 Generiranje izvještaja	29
3.12 Slanje mailova - Cloud Functions	30
3.13 Hosting	33
3.14 Konačni izgled aplikacije	34
4 Rezultati	40
4.1 Lighthouse rezultati	40
4.2 Zaključak	43

SADRŽAJ

v

Bibliografija

44

Uvod

U današnje vrijeme, većina ljudi posjeduje mobilni uređaj na kojem provode veliki dio slobodnog vremena. Stoga je tvrtkama u interesu da svoj proizvod mogu predstaviti u obliku aplikacije koju korisnik može instalirati. No, različiti operacijski sustavi često koriste različite tehnologije i programske jezike, što znači da za svaku mobilnu platformu treba kreirati odvojenu aplikaciju, a to zahtjeva puno vremena i sredstava. S druge strane, web aplikacije su dostupne u internet preglednicima, te se mogu pregledavati i na osobnim računalima i na mobilnim uređajima.

Progresivne web aplikacije (PWA) se javljaju kao logična kombinacija mobilnih (*native*) aplikacija i internet (*web*) aplikacija. Mogu se otvarati u internet preglednicima, ali se mogu i dodati na početni zaslon mobitela te koristiti kao mobilne aplikacije.

U prvom dijelu ovog diplomskog rada objasnit ćemo pojam PWA i njihova svojstva. U drugom dijelu ukratko spominjemo neke tehnologije koje se mogu koristiti pri kreiranju PWA. U trećem dijelu ćemo na konkretnom primjeru pokazati kako možemo iskoristiti React biblioteku za izradu složenije progresivne web aplikacije. U zadnjem dijelu komentiramo rezultat i proces razvijanja aplikacije.

Poglavlje 1

Uvod u PWA

1.1 Što je PWA?

Pojam “progresivne web aplikacije” prvi je put spomenut 2015. godine, kako bi se opisale web aplikacije koje koriste *service workere* (koji omogućavaju offline rad aplikacije), manifest dokumenta (koji omogućava spremanje web aplikacije na mobitel) i druge funkcionalnosti web preglednika [16].



Slika 1.1: PWA logo

S godinama se ta neformalna definicija mijenjala zbog napredovanja tehnologija koje omogućavaju “progresivnost” tih aplikacija. Službena definicija ne postoji, ali je Google sastavio popis zahtjeva koji moraju biti zadovoljeni da bi se neka aplikacija smatrala kao PWA [10]. Neki od tih uvjeta su:

- aplikacija se može instalirati na mobitel,
- aplikacija mora koristiti HTTPS protokol,
- svi URL-ovi aplikacije se moraju moći učitati kad je korisnik offline,
- aplikacija se može koristiti na sporijim mrežama,

- aplikacija se može koristiti na različitim internet preglednicima.

Google također nudi alat Lighthouse (detaljnije u poglavlju 4.1) koji između ostaloga provjerava zadovoljava li konkretna aplikacija sve te zahtjeve te može li se klasificirati kao PWA.

Napomenimo još da riječ “progresivan” u ovom kontekstu zapravo znači da kako napreduju korištene tehnologije, tako napreduje i sama aplikacija [17]. Naravno, možemo ići i u drugom smjeru: samo zato jer netko koristi stariju verziju mobitela ili web preglednika, ne znači da potpuno neće moći pristupiti aplikaciji, već samo znači da neće moći iskoristiti apsolutno svaku mogućnost koju ona pruža. Primjerice, na mobitelu starom pet godina aplikacija može biti dostupna samo u obliku statičkog HTML-a i CSS-a: puno bolje od alternative u kojoj se aplikacija ne može ni pokrenuti.

1.2 Značajke PWA

Objasnilo detaljnije uvjete Google-a za progresivne web aplikacije. Ako želimo da korisnik može instalirati aplikaciju na mobitel, želimo i da ima osjećaj da je to zapravo mobilna aplikacija, a ne web stranica. Dakle, mora biti prilagođena različitim veličinama ekrana. Također, želimo da nakon prvog “skidanja” i instaliranja aplikacije (koje može trajati duže), svako sljedeće otvaranje traje puno kraće. Jedan način da se to ostvari, je da se pri prvom otvaranju na uređaj skinu sve potrebne datoteke, te se pri idućim otvaranjima one preuzimaju iz memorije uređaja, umjesto da se ponovno skidaju (naravno, pri ažuriranju aplikacije se promijenjene datoteke ponovno spremaju, a stare se brišu). To se ostvaruje *cache*-om. Posljedica toga je da korisnik može koristiti aplikaciju i kad je offline ili na sporijoj mreži (jer datoteke preuzima iz priručne memorije).

Još jedna značajka progresivnih web aplikacija su *push* notifikacije: poruke koje se šalju preko internet preglednika ili mobilnih uređaja iz aplikacije koje u tom trenutku nisu bile otvorene. Zbog toga je potreban dio koda koji će stalno biti aktivan u pozadini i biti zadužen za prisluškivanje i objavljivanje tih poruka. Takve skripte se nazivaju *service worker*-i [15]. Budući da su *push* notifikacije vezane uz *service worker*-e, aplikacija koja koristi te notifikacije mora pratiti HTTPS protokol.

1.3 Prednosti (i mane) PWA

Jasno je da su neki uvjeti koji definiraju PWA te prethodno spomenute značajke ujedno i prednosti: aplikacija se može koristiti kad je slab internet signal, pa čak i kad uređaj uopće nije spojen na internet. Dakle, aplikacija ne ovisi o brzini interneta. Aplikacija je

brza i zauzima malo memorije. Može se instalirati kao mobilna aplikacija (te izgleda dobro na različitim uređajima s drukčijim veličinama ekrana) ili koristiti kao web aplikacija (također, na različitim web preglednicima).

Uz toliko prednosti, prirodno se postavlja pitanje: imaju li PWA neke mane? Ili barem nedostatke u odnosu na mobilne aplikacije? Naravno da da:

- web aplikacije imaju ograničen pristup hardverskim komponentama samih uređaja,
- Apple ograničava neke PWA mogućnosti,
- offline rad može biti značajno sporiji (u odnosu na online rad),
- aplikacija se ne može skinuti preko Google Play ili App Store platformi.

Možda se ti nedostaci i ne čine kao veliki problem, ali prije kreiranja progresivne web aplikacije trebamo provjeriti sve zahtjeve i vidjeti hoće li neki od nedostataka ograničavati rad aplikacije. U tom slučaju, PWA nije najbolji odabir.

Poglavlje 2

Pregled tehnologija za kreiranje PWA

U kasnijem poglavlju ćemo detaljnije i uz primjere opisati biblioteku React koju ćemo koristiti pri kreiranju naše aplikacije, ali ćemo prvo ukratko spomenuti i neke druge popularne tehnologije koje se također koriste pri kreiranju PWA.

Naravno, baza svih tehnologija za kreiranje web aplikacija su jezici HTML, CSS i JavaScript. HTML (HyperText Markup Language) je jezik koji se koristi za određivanje oblika i sadržaja web stranice, dok je CSS (Cascading Style Sheet) jezik kojim određujemo izgled stranice (čiji je oblik definiran pomoću HTML-a). JavaScript je skriptni programski jezik koji služi za stvaranje dinamičkih elemenata pomoću skripti koje se izvršavaju na klijentskoj strani (u web pregledniku).

Preact

Preact je biblioteka koju često spominju kao manju i bržu verziju Reacta. Zbog tih dobrih performansi, Preact nema toliko funkcionalnosti i mogućnosti kao React. Preact CLI je službeni build alat za Preact stranice koji kreira optimiziranu PWA [9].



Slika 2.1: Preact logo

Vue.js

Vue.js je okruženje za kreiranje korisničkog sučelja i *single page* aplikacija. Brz je i zauzima malo memorije. Na svojoj github stranici imaju nekoliko predložaka koji omogućuju brzo postavljanje progresivne web aplikacije [13].



Slika 2.2: Vue.js logo

Angular

Angular je okruženje za kreiranje web aplikacija. Trenutno nema toliko dobru podršku za kreiranje PWA kao što imaju Preact i Vue.js, ali zato rade na razvijanju *service worker*-a [1].



Slika 2.3: Angular logo

Ionic

Ionic je SDK (Software Development Kit) za hibride mobilnih aplikacija [4]. Može se koristiti kao omotač za kreiranje PWA koristeći Angular, React i Vue.js.



Slika 2.4: Ionic logo

Polymer

Polymer je JavaScript biblioteka koja sadrži komponente za izradu web aplikacija. Kreirali su predložak *PWA Starter Kit* [8] koji se može iskoristiti za razvijanje složenije PWA.



Slika 2.5: Polymer logo

Ostale tehnologije

Sve prethodno navedene biblioteke i okruženja se temelje na JavaScriptu jer je to najpopularniji jezik za *frontend* dio aplikacije. No postoje i neke druge tehnologije pomoću kojih možemo izraditi PWA ili barem njen dio:

- *django-pwa* je aplikacija koju instaliranjem u postojećem Django (web okruženje temeljenom na Pythonu) projektu pretvara taj projekt u PWA,
- *Laravel eCommerce PWA* je ekstenzija pomoću koje se aplikacija kreirana u Laravelu (PHP okruženju) može transformirati u PWA,
- Vaadin je Java okruženje koje dozvoljava pretvorbu aplikacije u PWA koristeći notaciju `@PWA` u svojim alatima,
- Ruby je objektno orijentirani programski jezik koji se koristi za razvijanje web aplikacija, te nudi kreiranje PWA pomoću *gem*-a.

Poglavlje 3

Aplikacija

3.1 Ideja i opis aplikacije

Glavni cilj ovog diplomskog rada je izraditi progresivnu web aplikaciju. Matematički odsjek Prirodoslovno matematičkog fakulteta ima nekoliko stranica za seminare gdje se objavljuju nadolazeća predavanja. Ideja je kreirati PWA koja će zamijeniti te stranice te omogućiti neke dodatne funkcionalnosti. Ono što želimo ostvariti je:

- unos novih predavanja,
- brisanje i mijenjanje postojećih predavanja,
- pregled nadolazećih i prošlih predavanja,
- dodavanje, brisanje, mijenjanje informacija o članovima seminara,
- generiranje izvještaja o svim predavanjima iz akademske godine,
- slanje emaila svim članovima seminara.

Također, trebamo dozvoliti unos i promjene podataka samo nekim članovima seminara.

Jasno je da nam treba baza u koju ćemo spremati podatke o predavanjima i članovima (poglavlje 3.6). Budući da moramo na neki način razlikovati koji korisnik ima pravo unositi promjene, a koji ne, moramo kreirati formu za prijavljivanje i napraviti korisničke račune (poglavlje 3.5). U poglavlju 3.11 ćemo objasniti implementaciju generiranja izvještaja, a u poglavlju 3.12 implementaciju slanja emailova.

3.2 Paketi i biblioteke

Glavna biblioteka koju ćemo koristiti za kreiranje aplikacije je React.js (detaljnije u poglavlju 3.3). No, prije toga ćemo spomenuti još nekoliko alata koje su nam potrebni u projektu.

Node.js i Yarn

Node.js je JavaScript *runtime* okruženje koje izvršava JavaScript izvan preglednika, te ga možemo preuzeti na službenoj stranici [6]. Nećemo ga koristiti direktno, nego ćemo koristiti njegov menadžer paketa, tj. **npm** (Node package manager), koji olakšava korištenje Node biblioteka i modula. Primjerice, potreban nam je paket **react-csv-reader** za učitavanje i baratanje csv dokumentima (koristimo ga kod unošenja informacija više članova seminara odjednom, vidljivo na slici 3.16).

No, **npm** ima karakteristike koje su nepoželjne za PWA: sporo skidanje i instaliranje paketa, nije deterministički (može davati različite rezultate za isti proces instalacije) te ne radi offline. Zbog tih razloga, koristimo **Yarn**: wrapper za **npm** kreiran od strane Facebook-a, sa istim funkcionalnostima, ali ne i navedenim manama. **Yarn** možemo preuzeti sa njihove službene stranice [14].

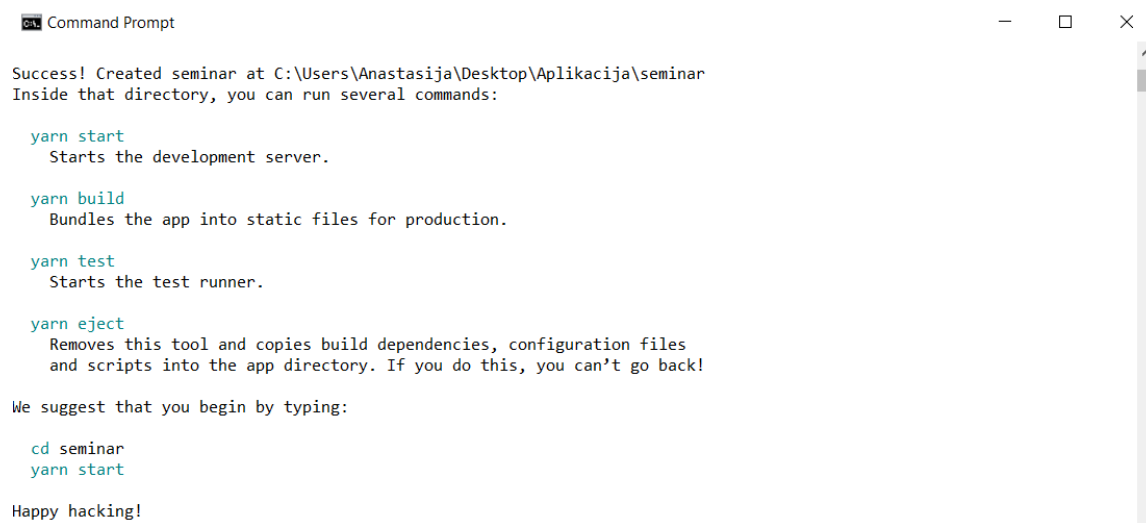
Create React App

Postoje dva glavna načina kreiranja PWA koristeći React. Prvi način je da sve radimo otpočetak, što uključuje instaliranje svih potrebnih biblioteka i konfiguriranje alata. Drugi način (koji ćemo koristiti pri kreiranju naše aplikacije) je koristeći **Create React App**, tj. **create-react-app**: naredbe koja kreira osnovnu web aplikaciju [2].

Koristeći **npm** (alat za izvršavanje Node paketa) instaliramo prvo **create-react-app**, a nakon toga pomoću **create-react-app** kreiramo projekt “seminar”:

- `$npm i create-react-app -g`
- `$npm create-react-app seminar`

Nas slici 3.2 možemo vidjeti strukturu direktorija unutar aplikacije nakon pokretanja naredbe **create-react-app**. U prvom su planu datoteke koje čine glavni dio aplikacije i koje ćemo sami koristiti i mijenjati.



```
Command Prompt

Success! Created seminar at C:\Users\Anastasija\Desktop\Aplikacija\seminar
Inside that directory, you can run several commands:

yarn start
  Starts the development server.

yarn build
  Bundles the app into static files for production.

yarn test
  Starts the test runner.

yarn eject
  Removes this tool and copies build dependencies, configuration files
  and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

cd seminar
yarn start

Happy hacking!
```

Slika 3.1: Poruke nakon create-react-app

Aplikaciju na lokalnom računalu pokrećemo na sljedeći način (na slici 3.1 možemo vidjeti da nam Create React App to i savjetuje za početak):

- `$cd seminar`
- `$yarn start`

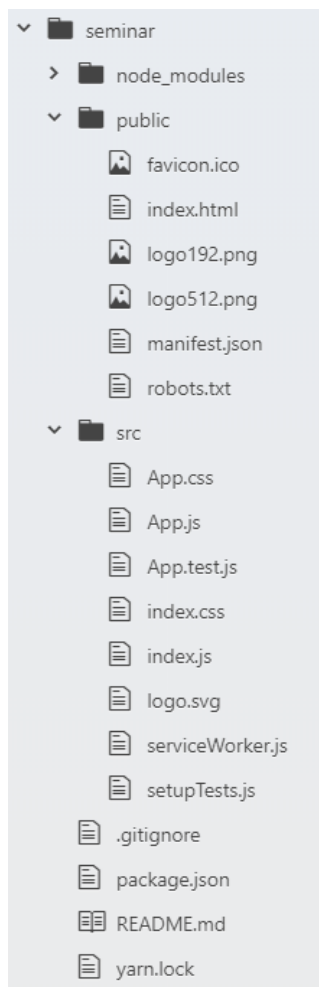
Dakle, pomoću Create React App već nakon par minuta možemo imati funkcionalnu aplikaciju (možemo vidjeti kako izgleda na slici 3.3). Doduše, ta aplikacija trenutno još ne radi ništa zanimljivo, ali smo zato već puno koraka ispred nekog tko instalira sve korak po korak. Prednosti ovog načina rada su očite:

- brzo i jednostavno postavljanje aplikacije,
- ne moramo se zamarati “pozadinskim konfiguracijama”,
- fokus je na samoj funkcionalnosti aplikacije.

Neki nedostaci Create React App načina rada su:

- nije dovoljno fleksibilan za aplikacije koje nisu SPA ¹,
- fokusiran je uglavnom na frontend,

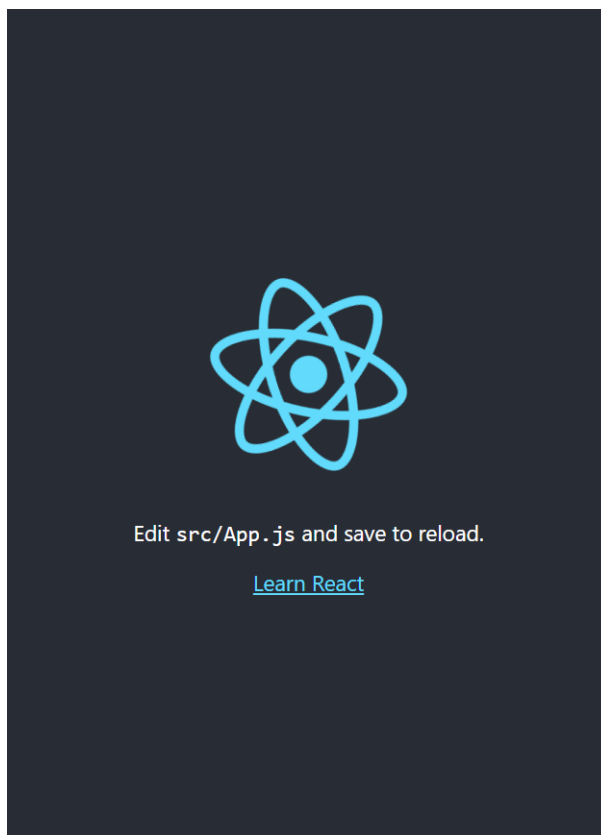
¹SPA (Single Page Application) je web aplikacija koja se sastoji od samo jedne stranice



Slika 3.2: Struktura aplikacije nakon create-react-app

- nema prostora za veće prilagodbe u konfiguracijama.

Glavni kompromis Create React App-a je da su svi alati konfigurirani tako da rade na točno određen način, što onemogućava moguće željene promjene. Ali, ako u bilo kojem trenutku odlučimo da nam trenutni način rada ne odgovara, možemo iskoristiti naredbu `eject`: na taj način “izlazimo” iz okvira koji Create React App zadaje. No, nakon tog izbacivanja povratka više nema te moramo nastaviti sa ručnim konfiguriranjem.

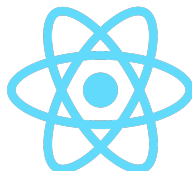


Slika 3.3: Osnovna Create React App aplikacija

3.3 React

React je JavaScript biblioteka koja služi za kreiranje korisničkog sučelja. Prva verzija je nastala u 2013. godini, a razvija ju Facebook [12].

React se često koristi u razvoju PWA zbog nekoliko razloga: brzine, jednostavnosti i mogućnosti podjele UI-a na manje dijelove (komponente) što pridonosi lakšem skidanju i bržem učitavanju stranice PWA. Pomoću sljedećih nekoliko isječaka koda naše aplikacije ćemo predstaviti osnovne koncepte korištenja React-a.



Slika 3.4: React logo

Renderiranje - ReactDOM

Pogledajmo dijelove koda koje nam je generirao Create React App. Prvo imamo `public/index.html` koji sadrži:

```
1 <body>
2   <div id="root"></div>
3 </body>
```

Ono što će se zapravo ispisati se nalazi u `src/App.js`:

```
1 import React from 'react';
2
3 function App() {
4   return (
5     <div className="App">
6       <p>Hello world</p>
7     </div>
8   );
9 }
10 export default App;
```

I na kraju, potreban nam je `src/index.js`:

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import App from './App';
4
5 ReactDOM.render(<App />, document.getElementById('root'));
```

Ovdje možemo vidjeti nekoliko stvari. Prvo, je da se u `<body>` tagu `index.html` datoteke nalazi samo `<div>` element s id-jem “root”, tj. korijen. Sve što želimo prikazati će se nalaziti u tom elementu. Drugo, funkcija `App` (iz `src/App.js`) je zapravo funkcionalna komponenta React-a koja vraća React element. U ovom slučaju, to je paragraf koji sadrži string “Hello world.”

Dakle, imamo ono što želimo prikazati (“Hello world”) i znamo gdje to želimo prikazati (“root”). Kako ta dva dijela spojiti? Spomenuli smo da je React paket za *kreiranje* dijelova korisničkog sučelja. Da bismo te dijelove zapravo i prikazali (tj. renderirali), koristimo **ReactDOM**: paket za renderiranje UI komponenti kreiranih pomoću React-a. Upravo to i činimo u `index.js` dokumentu: renderiramo element `<App>` u elementu “root”.

Spomenimo odmah još jednu stvar. Rekli smo da je `App` funkcionalna React komponenta. No, mogli smo ju zapisati i kao klasnu komponentu. U tom slučaju, `App` postaje klasa koja mora naslijediti React klasu `Component` i definirati funkciju `render`. U tom slučaju, `App.js` bi izgledao:

```
1 import React, { Component } from 'react';
2
3 class App extends
4 {
5     render ()
6     {
7         return (
8             <div className="App">
9                 <p> Hello world </p>
10            </div>
11        );
12    }
13 }
14 export default App;
```

Trenutno nema razlike između ta dva načina, ali ako želimo u nekoj komponenti koristiti stanja (states), moramo ju definirati kao klasnu komponentu.

Routing

Kako smo već spomenuli, koristeći Create React App, možemo kreirati samo SPA (Single Page Application). To znači da se koristi jedna `index.html` stranica na kojoj se renderiraju svi elementi, za razliku od ostalih aplikacija (MPA - Multiple Page Application) koje se sastoje od više stranica (npr. poseban html dokument za ispisivanje članova seminara, poseban html dokument za unos novog predavanja itd.). Ne možemo koristiti više html stranica, ali ih možemo simulirati koristeći biblioteku **React Router**. I dalje će nam glavni element biti `App`, ali ćemo unutar njega renderirati samo one stvari koje se nalaze na trenutnoj “ruti”. Ruta je u ovom slučaju React komponenta koja se renderira ovisno o URL-u.

Objasnimo na aplikaciji. Prvo moramo promijeniti `index.js`, na sljedeći način:

```
1 import { BrowserRouter } from 'react-router-dom';
2
```

```
3 ReactDOM.render (
4   <BrowserRouter> <App /> </BrowserRouter>,
5   document.getElementById('app')
6 );
```

BrowserRouter je React komponenta koja koristi HTML5 API koji sinkronizira korisničko sučelje u skladu s URL-om.

Dalje, recimo da želimo simulirati dvije stranice, tj. rute: home (na kojoj ćemo prikazivati popis sažetaka predavanja) i newLecture (na kojoj će se nalaziti forma za dodavanje novog predavanja). Neka su te dvije rute definirane React komponentama Home i NewLecture. Te dvije rute definiramo u render funkciji komponente App, koja sada izgleda ovako:

```
1 import Home from './Home'
2 import NewLecture from './NewLecture'
3 ...
4 render ()
5 {
6   return (
7     <div className="App">
8       <Route exact path="/" component = { Home } />
9       <Route path="/newLecture" component = {NewLecture }/>
10    </div>
11  );
12 }
13 withRouter(App);
```

U liniji 8 smo definirali da će nam glavna ruta (ona koja ima glavni URL) prikazivati komponentu Home, a ruta čiji URL ima nastavak “/newLecture” prikazivati komponentu NewLecture.

Kad imamo definirane rute, možemo ih koristiti u Link komponentama, koje će nas usmjeriti na željene rute (npr. u aplikaciji postoji komponenta Navigation koja sadrži linkove za sve rute). Recimo da želimo uvijek pokazivati linkove na Home i NewLecture. Tada u render funkciju komponente App dodajemo sljedeće:

```
1 import { Link } from 'react-router-dom';
2 ...
3 render ()
4 {
5   return (
6     <div className="App">
7       <p>
8         <Link to="/">Pocetna</Link>
9         <Link to="/newLecture">Novo predavanje</Link>
```

```
10     </p>
11     <Route exact path="/" component = { Home } />
12     <Route path="/newLecture" component = { NewLecture } />
13 </div>
14 );
15 }
```

State i props

Spomenuli smo da će nam `App` biti glavna komponenta. U njoj ćemo spremati polje članova (`users`) i polje predavanja (`lectures`). Ideja je da ta polja mogu koristiti i druge komponente (npr. želimo da komponenta `Home` može pristupiti polju `lectures`, kako bi u njoj mogli ispisati popis predavanja).

Kod klasnih komponenti smo spomenuli `state` (stanje). Na `state` možemo gledati kao lokalne varijable komponente, pa naša polja `users` i `lectures` možemo čuvati u stanju. U sljedećem isječku koda možemo vidjeti kako dohvaćati i mijenjati stanja. Funkcija `addUser` prima ime i prezime novog korisnika, zatim preuzme trenutno polje članova pomoću `this.state.users`, doda novog korisnika, i na kraju spremi novo polje u `state` pomoću `this.setState()`.

```
1 class App extends Component
2 {
3     state = { users: [], lectures: [] };
4     addUser = (name) =>
5     {
6         var user;
7         user.name = "Pero Peric";
8
9         var newUsers = this.state.users;
10        newUsers.push(user);
11        this.setState({ users: newUsers });
12    }
13    ...
14    render() {...}
15 }
```

Sada znamo kako ćemo spremati podatke u `App`, ali moramo nekako proslijediti te podatke komponenti `Home`. Zbog toga ćemo promijeniti definiciju rute `Home` na sljedeći način:

```
1 <Route exact path="/"
2     render = { (props) =>
```

```
3           <Home lectures = { this.state.lectures } />{  
4 />
```

Ovom sintaksom želimo reći App da proslijedi polje `this.state.lectures` komponenti Home u polju `lectures` kroz objekt `props` (properties - svojstva). Na taj način će Home moći pristupiti polju predavanja pomoću `this.props.lectures`.

Ovaj način rada ima veliku prednost: ako se u pozadini dogodi neka promjena u predavanjima, te se promijeni polje `state.lectures` u App komponenti, te iste promjene će se dogoditi i u polju `props.lectures`, jer je to zapravo jedno te isto polje.

Input i event handle

Obradit ćemo još jednu stvar specifičnu za React. Spomenuli smo da želimo omogućiti voditelju seminara da unese nove članove na našoj aplikaciji. Dakle, želimo mu ponuditi formu u kojoj će upisati ime i prezime, te klikom na gumb spremiti te podatke u polje `users` (možemo iskoristiti prethodno definiranu funkciju `addUser`). Taj proces unošenja će se odvijati u komponenti Home, ali želimo podatke spremiti u polje `users` komponente App, što znači da ćemo na neki način morati prenijeti podatke uzlazno po hijerarhiji komponenta.

Prvo moramo imati varijablu u koju ćemo spremiti uneseno ime i prezime (`name`). Ima smisla da se to nalazi u `state` objektu komponente Home. Trebaju nam i dvije funkcije: jedna za registriranje promjene imena i prezimena (`handleNameChange`) i druga koja će se pozvati kod klika na gumb (`onClick`). `Home.js` sada izgleda ovako:

```
1  class Home extends Component  
2  {  
3      state = { name: '' };  
4  
5      handleNameChange = (event) =>  
6      {  
7          this.setState({ name: event.target.value });  
8      };  
9      handleSubmit = () =>  
10     {  
11         this.props.addUser(this.state.name);  
12     }  
13  
14     render()  
15     {  
16         return(  

```

```
17         <form>
18             Ime: <input type="text"
19                     onChange={ this.handleChange }
20                     value={ this.state.name }>
21             <br>
22             <button onClick={this.handleSubmit}>Spremi</button>
23         </form>
24     );
25 }
26 }
```

U elementu `input` imamo funkciju `onChange` koja će se izvršiti svaki put kad korisnik unese novo slovo ili obriše neko staro. Stavili smo da želimo da se za svaku promjenu u `input`-u pozove funkcija `handleChange`, koja prima objekt `event`. Vrijednost sadržaja `inputa` možemo preuzeti iz `event.target.value` (budući da se taj `event` šalje iz `input` elementa), te ju spremimo u `state`. Na taj način, uvijek imamo najnoviju verziju imena (tj. sadržaj `input-a`) u `state-u`. Također primijetimo da u `value` svojstvu `input` elementa prikazujemo aktualnu verziju imena (iz `state-a`).

Na kraju forme imamo `button` koji kod klika poziva funkciju `handleSubmit` koja samo poziva `addUser` (s argumentom `name`) iz `App` komponente kojoj pristupamo preko `props` objekta. No trenutno `Home` komponenta nema pristup toj funkciji što znači da moramo promijeniti `Route` element za `Home`:

```
1 <Route exact path="/"
2     render = { (props) =>
3         <Home lectures = { this.state.lectures }
4             addUser = this.addUser />
5 />
```

3.4 Firebase

Aplikacija zahtijeva online bazu podataka u koju ćemo spremati informacije o predavanjima i članovima seminara, te sustav autentifikacije. Te dvije stvari nam nudi **Firebase**: Google platforma za razvoj mobilnih i web aplikacija [3]. Koristit ćemo i opciju *hosting-a* aplikacije na `Firestore`-u, da nam olakša razvoj aplikacije.

Odmah ćemo napomenuti da postoje i alternative `Firestore-u`, od kojih je najpopularnija `Parse - open source backend` platforma, koja je fleksibilnija po pitanju naplaćivanja i prilagodbe `hosting-a`. Ali za potrebe naše aplikacije, `Firestore` će biti i više nego dovoljan.



Slika 3.5: Firebase logo

Da bismo mogli koristiti Firebase, treba nam samo Google račun. Nakon što se prijavimo na Firebase s tim računom, odaberemo *Add project*. Nakon što je projekt kreiran, na konzoli odaberemo *Get started by adding Firebase to your app* za web aplikaciju i pratimo sljedeće korake:

1. imenujemo aplikaciju (“seminar”) i odaberemo opciju *hosting*
2. dodamo Firebase SDK u našu aplikaciju tako da zalijepimo sljedeći kod u našu `index.html` datoteku (`firebaseConfig` objekt popunimo s vrijednostima koje možemo naći na Firebase konzoli):

```
1 <script src="https://www.gstatic.com/firebasejs/7.2.1/
  firebase-app.js"></script>
2 <script src="https://www.gstatic.com/firebasejs/7.2.1/
  firebase-auth.js"></script>
3 <script src="https://www.gstatic.com/firebasejs/7.2.1/
  firebase-firestore.js"></script>
4
5 <script>
6   var firebaseConfig = {
7     apiKey: "api-key",
8     authDomain: "project-id.firebaseio.com",
9     databaseURL: "https://project-id.firebaseio.com",
10    projectId: "project-id",
11    storageBucket: "project-id.appspot.com",
12    messagingSenderId: "sender-id",
13    appId: "app-id",
14    measurementId: "G-measurement-id"
15  };
16
17  window.firebase = firebase;
18  window.firebase.initializeApp(firebaseConfig);
19 </script>
```

Primijetimo da smo uključili linkove za usluge koje planiramo koristiti (autentifikacija i baza podataka - Firestore).

Nakon toga, budući da želimo koristiti Firebase *hosting*, instaliramo Firebase CLI u mapi našeg projekta pomoću naredbe `$npm install -g firebase-tools`. Zatim se prijavimo u Firebase koristeći `$firebase login` (sa Google računom koji smo koristili za prijavu na Firebase), te ga inicijaliziramo naredbom `$firebase init` (označimo da želimo koristiti “Hosting” i “Functions” - Firebase funkcije ćemo koristiti kod generiranja izvještaja).

Na kraju, budući da sa svakim deployem na Firebase želimo poslati najnoviju verziju aplikacije, u `package.json` dodajemo sljedeću `deploy` naredbu:

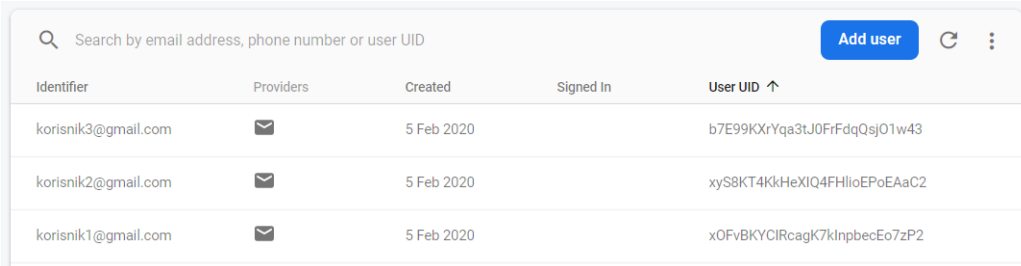
```
1 "scripts": { "deploy": "npm run build && firebase deploy" }
```

Kada želimo slati novu verziju, samo moramo upisati `$yarn deploy [15]`.

3.5 Korisnički računi

Želimo omogućiti unošenje predavanja na samoj aplikaciji, ali želimo tu mogućnost ograničiti na samo nekoliko korisnika. Najbolji način da to ostvarimo je da tim osobama dodijelimo korisničke račune pomoću kojih se mogu prijaviti i unositi predavanja, a svim drugim korisnicima dozvoliti samo pregled predavanja, bez mijenjanja.

Firebase nudi nekoliko vrsta autentifikacije korisnika, a u našoj aplikaciji ćemo koristiti najpopularniju: pomoću email adrese i lozinke. Prvo moramo na Firebase konzoli dozvoliti Email/Password autentifikaciju. Nakon toga dodamo željene račune s email adresama i nekim početnim lozinkama (na aplikaciji ćemo kasnije dozvoliti da svaki korisnik može sam promijeniti svoju lozinku), kao što vidimo na slici 3.6.



The screenshot shows the Firebase Authentication console interface. At the top, there is a search bar with the text "Search by email address, phone number or user UID". To the right of the search bar are three icons: a blue "Add user" button, a refresh icon, and a menu icon. Below the search bar is a table with the following columns: "Identifier", "Providers", "Created", "Signed In", and "User UID ↑". The table contains three rows of user data:

Identifier	Providers	Created	Signed In	User UID ↑
korisnik3@gmail.com	✉	5 Feb 2020		b7E99KXrYqa3tJ0FrFdqQsj01w43
korisnik2@gmail.com	✉	5 Feb 2020		xyS8KT4KkHeXIQ4FHlioEPoEAaC2
korisnik1@gmail.com	✉	5 Feb 2020		xOFvBKYCIRcagK7klnpbecEo7zP2

Slika 3.6: Primjer korisnika na autentifikacijskoj konzoli

U novu komponentu Login dodajemo formu gdje korisnik može unijeti svoju email adresu i lozinku, te gumb koji će nakon klika pozvati sljedeću funkciju login:

```
1 login = () =>
2 {
3   window.firebase.auth()
4   .signInWithEmailAndPassword(this.state.email,
5     this.state.password)
6   .then(res =>
7     {
8       console.log("Uspjesan login.");
9     })
10  .catch(error =>
11    {
12      if (error.code === 'auth/user-not-found')
13      {
14        console.log("Ne postoji korisnik s navedenim emailom.");
15      }
16      else if (error.code === 'auth/invalid-email')
17      {
18        console.log("Neispravni email.");
19      }
20    });
21 }
```

U liniji 3 prvo dohvaćamo sustav autentifikacije, i onda u liniji 4 pokušamo prijaviti korisnika Firebase funkcijom `signInWithEmailAndPassword` koja nam vraća promise. **Promise** je objekt koji se koristi za baratanje asinkronim operacijama koje se ne bi mogle riješiti običnim callback metodama (jer rezultati tih operacija nisu trenutni). Ako su email i lozinka bili ispravni, te se nije dogodila neka neočekivana pogreška, promise će vratiti pozitivan rezultat. U suprotnom, promise vraća error na temelju kojeg možemo otkriti što je pošlo po zlu.

Recimo da se korisnik uspješno prijavio. Kako aplikacija to može prepoznati te mu ponuditi opcije koje prije nije imao? Ovdje možemo iskoristiti Firebase funkciju `onAuthStateChanged` koja će se pozvati pri prijavljivanju i odjavljivanju. Tu funkciju možemo smjestiti u `App`, ali gdje točno? Odgovor se nalazi u lifecycle metodama React komponenti, konkretno u metodi `componentDidMount`: ta metoda se poziva svaki put kad se njena komponenta ubaci u hijerarhiju (stablo) komponenata aplikacije, i u nju stavljamo sve pozive i funkcije koje ovise o nekim vanjskim sistemima, kao što je u ovom slučaju Firebase autentifikacija. Dakle, u `App` ubacujemo funkciju `componentDidMount` koja sad izgleda ovako:

```
1 componentDidMount = () =>
2 {
```

```
3     window.firebase.auth().onAuthStateChanged(user =>
4     {
5         if (user)
6         {
7             this.setState({ thisUser : user });
8         }
9         else
10        {
11            this.setState ({ thisUser: null });
12        }
13    });
14 }
```

Sada je jasno kako ćemo prepoznati kada trebamo pokazivati dijelove aplikacije koji su dostupni samo korisnicima koji su prijavljeni: kada je `this.state.thisUser !== null`. Na slici 3.7 možemo vidjeti kako *Osobna stranica* izgleda prije prijavljivanja, a na slici 3.8 nakon prijavljivanja. Aplikacija zna koju verziju mora pokazati ovisno o vrijednosti varijable `thisUser`.

Spomenimo još neke korisne funkcije koje koristimo pri baratanju s autentifikacijom korisnika:

- `signOut()`
- `sendPasswordResetEmail()`

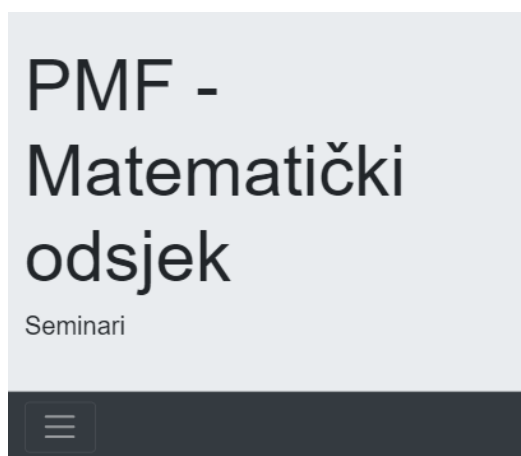
3.6 Firestore - baza podataka

Treba nam neka baza podataka u kojoj ćemo čuvati podatke o predavanjima i članovima seminara. Ta baza će biti na serveru (“u oblaku”), te će biti sinkronizirana kod svih korisnika: kada netko unese predavanje, želimo da drugi korisnik koji u tom trenutku pregledava aplikaciju automatski vidi promjenu, bez da mora ponovno učitati stranicu. Firebase nudi bazu podataka koja omogućava upravo to: **Firestore**.

Firestore je NoSQL baza: sastoji se od kolekcija, koje onda sadrže dokumente koji mogu sadržavati kompleksne objekte ili čak manje kolekcije. Na slici 3.9 možemo vidjeti kako to izgleda na Firebase konzoli.

U `index.html` (nakon inicijalizacije Firebasea) dodajemo sljedeću liniju:

```
1 window.firebase.firestore().enablePersistence();
```



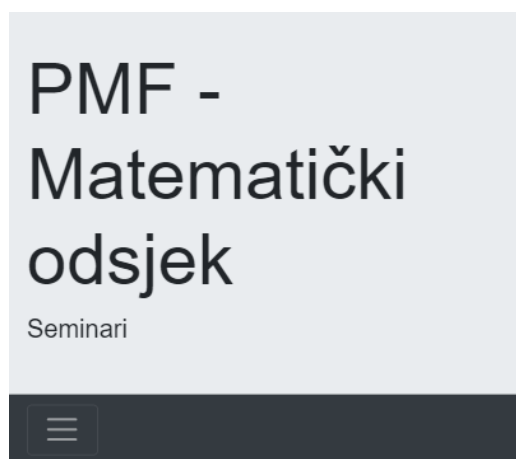
Uđi u svoj korisnički račun

Email

Lozinka

[Log in](#) [Zaboravio lozinku?](#)

Slika 3.7: Prije prijavljivanja



Korisnik: **Prof. dr. sc. Filip Filipic**

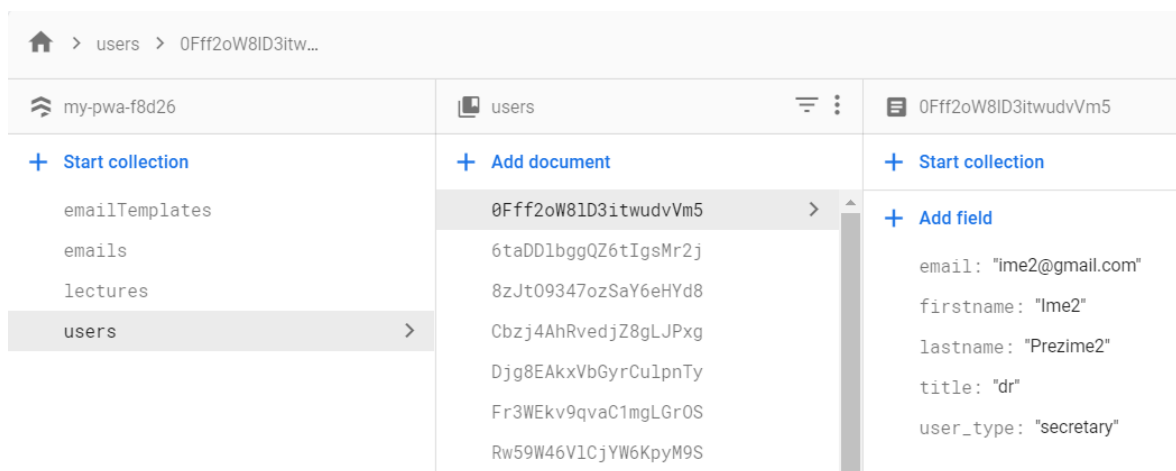
Email: **filip@gmail.com**

[Logout](#) [Promijeni lozinku](#) [Promijeni ime/prezime/titulu](#)

Slika 3.8: Nakon prijavljivanja

Ta linija će nam omogućiti offline rad: korisnik može unijeti novo predavanje kad je offline i zatvoriti aplikaciju. Kad ponovno dođe online, Firestore će u pozadini sinkronizirati lokalne podatke s bazom na serveru.

Sada kad smo objasnili koju bazu ćemo koristiti, možemo objasniti kako njome baratati. Komponenti App dodajemo funkciju koja će reagirati na promjene u određenoj kolekciji (za potrebe primjera neka to bude kolekcija users). U funkciju `componentDidMount` možemo dodati listenera (koji će pri prvom pokretanju aplikacije dohvatiti trenutnu verziju kolekcije).



Slika 3.9: Primjer sadržaja Firestore baze

```

1  componentDidMount()
2  {
3    window.firebase.firestore().collection("users")
4      .onSnapshot( snapshot =>
5        {
6          var users = [];
7          snapshot.forEach((doc) =>
8            {
9              const user = doc.data()
10             user.id = doc.id;
11             users.push(user);
12           });
13           this.setState({ users: users });
14         });
15  }

```

U liniji 3 dohvaćamo kolekciju `users` u objektu `snapshot` koji sadrži podatke svih dokumenata unutar te kolekcije. Prolazimo po tim dokumentima i spremamo ih u polje `users`, zajedno s njihovim id-jevima (koje Firebase automatski generira). Kako se ta funkcija poziva kod svake promjene u bazi, uvijek ćemo imati ažurirane podatke u polju.

Prethodno smo bili napisali funkciju `addUser` koja sprema novog člana u polje komponente `App`. Sada želimo promijeniti tu funkciju: umjesto spremanja u polje, želimo spremiti u bazu. Ponovno moramo dohvatiti kolekciju `users` te ubaciti člana koristeći (Fi-

restore) funkciju `add()`. Ta funkcija nam također vraća promise koji možemo iskoristiti da provjerimo je li ubacivanje prošlo u redu. U sljedećem isječku koda možemo vidjeti kako funkcija `addUser` trenutno izgleda.

```
1 class App extends Component
2 {
3   addUser = (name) =>
4   {
5     window.firebase.firestore().collection("users")
6     .add({ name: name})
7     .then( () =>
8     {
9       console.log("Dodan novi korisnik");
10    })
11    .catch(function(error)
12    {
13      console.error("Error adding document: " + error);
14    });
15 }
```

Postoje još neke često korištene funkcije za baratanje podacima Firestore baze:

- `update()` - mijenjanje sadržaja postojećeg dokumenta,
- `delete()` - brisanje postojećeg dokumenta iz baze,
- `get()` - dohvaćanje konkretnog dokumenta.

U sva tri slučaja radimo promjene na konkretnom dokumentu, pa moramo znati njegov id. Ako označimo taj id s `ID`, dokument možemo dohvatiti sa sljedećom naredbom:

```
1 window.firebase.firestore().collection("users").doc(`${ID}`)
```

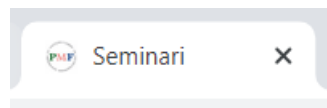
3.7 Manifest datoteka

Web app manifest je JSON datoteka koju web aplikacija mora sadržavati da bi ju korisnik mogao instalirati na mobilnom uređaju. Zbog te datoteke aplikacija ostavlja dojam *native* (mobilne) aplikacije. Mora sadržavati sljedeće informacije: `start_url` (URL koji se učitava pri pokretanju aplikacije), `name` (ime aplikacije) i `icons` (polje slika koje aplikacija koristi) [17].

Linija 2 sljedećeg koda datoteke `index.html` povezuje aplikaciju i manifest datoteku:

```
1 <head>
2   <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
3
4   <title>Seminari</title>
5   <link rel="icon" href="%PUBLIC_URL%/assets/favicon.ico" />
6 </head>
```

U linijama 4 i 5 definiramo naslov i *favicon* (slika koja će biti na tab-u u internet pregledniku) aplikacije. Na slici 3.10 možemo vidjeti kako to izgleda.



Slika 3.10: Tab aplikacije

Ispod se nalazi cijeli sadržaj `manifest.json` datoteke naše aplikacije:

```
1 {
2   "short_name": "PMF seminari",
3   "name": "PMF - Matematički odsjek - Znanstveni seminari",
4   "icons": [
5     {
6       "src": "favicon.ico",
7       "sizes": "64x64 32x32 24x24 16x16",
8       "type": "image/x-icon"
9     },
10    {
11      "src": "assets/android-icon-144x144.png",
12      "sizes": "144x144",
13      "type": "image/png"
14    },
15    {
16      "src": "assets/android-icon-192x192.png",
17      "sizes": "192x192",
18      "type": "image/png"
19    },
20    {
21      "src": "assets/android-icon-512x512.png",
22      "sizes": "512x512",
23      "type": "image/png"
24    }
25  ],
26  "start_url": ".",
```



```
27   "display": "standalone",
28   "theme_color": "#000000",
29   "background_color": "#ffffff"
30 }
```

Osim instaliranja na početni ekran mobilnih uređaja, `manifest.json` možemo iskoristiti i za definiciju vlastitog *splash screen*-a: ekrana koji se pojavi pri pokretanju aplikacije, prije nego što se renderira pravi sadržaj aplikacije (da korisnik ne mora gledati u potpuno prazan ekran). Osim `name`, i `icons`, moramo definirati i `background_color`.

Na slici 3.11 se nalazi *splash screen* koji je definiran s prethodnom `manifest.json` datotekom.

3.8 App shell

App shell (“ljuska aplikacije”) je minimalan HTML i CSS korisničkog sučelja. Ideja je da se taj dio (koji ne ovisi JavaScriptu i Reactu) prebaci u `index.html` datoteku, te se na taj način prikaže barem statički dio aplikacije, prije nego ReactDOM stigne renderirati ostatak aplikacije. U našem slučaju *app shell* je samo header (jer sve ostalo ovisi o Reactu), pa taj dio možemo kopirati u `index.html`:

```
1 <body>
2   <div id = "root">
3     <div class="jumbotron" height=200px margin-bottom=0px>
4       <h1 class="display-4">PMF - Matematički odsjek</h1>
5       <h2 class="lead">Seminar</h2>
6     </div>
7   </div>
8 </body>
```

Budući da smo header dodali u “root”, taj će se dio prikazati prije nego se renderira App komponenta, nakon čega će renderirani kod zamijeniti taj statički header [15].

3.9 Service worker

U uvodu u PWA smo spominjali service workere i cache, kojima omogućavamo offline rad aplikacije. Dobra stvar korištenja `create-react-app` je da je jedan takav service worker uključen u strukturu projekta: `src/serviceWorker.js`. U trenutku kreiranja aplikacije, taj service worker je isključen, ali ga možemo pokrenuti tako da u datoteci `src/index.js` promijenimo liniju `serviceWorker.unregister()`; u `serviceWorker.register()`; . Kad to promijenimo, nakon prvog loadanja će se datoteke cacheirati i ažurirati sa svakim



Slika 3.11: Splash screen

sljedećim depoly-em, što znači da ćemo aplikaciju moći koristiti na sporim i nepouzdanim mrežama, čak i potpuno offline.

3.10 React-Bootstrap

Za stiliziranje aplikacije ćemo koristiti **React-Bootstrap**: frontend okruženje za React [11]. Sadrži već definirane React komponente koje po potrebi *import*-amo. Instaliramo ga naredbom `$npm install react-bootstrap bootstrap`. Nakon instaliranja, u datoteku `src/index.js` dodajemo sljedeću liniju:

```
1 import 'bootstrap/dist/css/bootstrap.min.css';
```

A u `index.html` sljedeći link:

```
1 <link rel="stylesheet"
2     href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/
3     bootstrap.min.css"
4     integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/
5     iJTQOUhcWr7x9JvoRxT2MZw1T"
6     crossorigin="anonymous"
7     async>
```

Nakon što smo to dodali, možemo koristiti različite Bootstrap elemente. Pokažimo na primjeru Header komponente (rezultat možemo vidjeti na slici 3.7). Kao zaglavlje, koristimo komponentu Jumbotron, koju *import*-amo u liniji 2. Nakon toga ga koristimo kao običan HTML element, s tim da možemo definirati neke podvrste Jumbotrona (u ovom slučaju, `className="mb-0"` označava da želimo da gornja i donja margina budu 0).

```
1 import React from 'react';
2 import Jumbotron from "react-bootstrap/Jumbotron";
3 const Header = () =>
4 {
5   return (
6     <Jumbotron className="mb-0">
7       <h1 className="display-4" >PMF - Matematicki odsjek</h1>
8       <h2 className="lead">Seminari</h2>
9     </Jumbotron>
10  );
11 };
12 export default Header;
```

3.11 Generiranje izvještaja

Automatsko generiranje izvještaja o predavanjima je lako ostvariti, ali uvelike olakšava posao voditeljima seminara koji su dosad morali sami prolaziti po popisu predavanja iz cijele

akademske godine te ručno ispisivati i vaditi podatke. Generirani izvještaj će biti \LaTeX dokument s .txt ekstenzijom (jer trenutno ne postoji React modul koji bi kreirao dokument s ekstenzijom .tex).

Sam izvještaj popunimo prolazeći po poljima `users` i `lectures`, te filtriramo u posebna polja stringova, ovisno o tipu člana i predavanja. Na kraju spojimo sve te stringove u željenom formatu (dakle u obliku \LaTeX koda) u jedan zajednički string `reportText` koji prosljeđujemo funkciji `downloadReport` koja zapravo kreira i pokreće skidanje txt dokumenta. U toj funkciji prvo imenujemo txt file, kreiramo privremeni `<a>` HTML element, koristimo UTF-8 standard za kodiranje, definiramo `'download'` svojstvo jer želimo skinuti txt dokument lokalno na računalo, te nakon skidanja brišemo taj element.

Ispod se nalazi potpuni kod funkcije `downloadReport`.

```
1 downloadReport = (reportText) =>
2 {
3   var filename = "izvjestaj";
4   var element = document.createElement('a');
5
6   element.setAttribute('href', 'data:text/plain;charset=utf-8,' +
7     encodeURIComponent(reportText));
8   element.setAttribute('download', filename);
9   element.style.display = 'none';
10
11  document.body.appendChild(element);
12  element.click();
13  document.body.removeChild(element);
14 }
```

3.12 Slanje mailova - Cloud Functions

Jedan od zahtjeva koje aplikacija ima je mogućnost slanja emailova svim članovima seminara (čije email adrese imamo spremljene u Firestore bazi) pri kreiranju novog predavanja ili promjene nekog starog. No ovdje se javlja problem: ne možemo slati email koristeći samo React, jer se React aplikacije vrte na klijentskoj strani. Taj problem možemo zaobići koristeći Firebase **Cloud Functions** - “funkcije u oblaku”.

Cloud Functions nam omogućavaju automatsko pokretanje backend koda koji reagira na HTTPS zahtjeve i određena svojstva Firestore-a. Primjerice, možemo imati funkciju koja će se pozvati svaki put kada se u neku Firestore kolekciju doda novi dokument. U našem slučaju, želimo da se na događaj upisivanja novog dokumenta u kolekciju `emails`

EMAIL

Email
adrese:

Naslov
emaila:

Sadržaj
emaila:

Pošalji **Učitaj predložak** Dodaj novi predložak

- BEZ PREDLOŠKA
- predlozak 1

Slika 3.12: Forma za slanje emaila

pokrene funkcija koja će kreirati i poslati email s informacijama koje voditelj unese u formu na slici 3.12. Prednost ovog načina rada je da voditelj može “poslati” email čak i kada je offline, jer će se Cloud Functions pobrinuti za pravo slanje emaila kada se voditelj spoji natrag na mrežu.

Možemo primijetiti da na slici 3.12 imamo i gumbove za učitavanje i spremanje email predložka u bazu, da voditelj ne mora svaki put iznova pisati sadržaj emaila.

Prisjetimo se: pri inicijalizaciji Firebase-a u našoj aplikaciji, naveli smo da planiramo koristiti Firebase “Functions”, pa u našem projektu već imamo kreiranu mapu `functions` u kojem se nalazi dokument `index.js` gdje ćemo smjestiti našu funkciju slanja emaila. Za samo slanje mailova koristit ćemo Node.js modul **Nodemailer** [7]. Instaliramo ga pomoću naredbe `$npm install nodemailer --save`, (po potrebi ga moramo dodati u `functions/package.json` dokument pod `dependencies`).

Napomenimo još jednom: kada voditelj unese željene informacije o mailu, klikom na gumb “Pošalji” se te informacije (`emailAddresses`, `emailTitle` i `emailContent`)

spremaju u dokument, koji onda spremamo u kolekciju emails. Funkcija slanja emaila će se automatski pokrenuti u pozadini jer će reagirati na unos novog dokumenta u kolekciju emails.

Dolje je prikazan cijeli sadržaj dokumenta `index.js`, čiji ćemo kod objasniti dio po dio:

```
1  const functions = require('firebase-functions');
2  const admin = require("firebase-admin");
3  const nodemailer = require('nodemailer');
4
5  admin.initializeApp();
6
7  var transporter = nodemailer.createTransport({
8    service: 'gmail',
9    auth:
10   {
11     user: 'seminari.pmf@gmail.com',
12     pass: 'seminari.pmf2020'
13   }
14 });
15
16 exports.sendEmail = functions.firestore.document('emails/{emailID}')
17   .onCreate((snap, context) =>
18     {
19       admin.firestore().collection('emails')
20       .doc(`${context.params.emailID}`).get().then(doc => {
21         var emailAddresses = doc.data().emailAddresses;
22         var emailContent = doc.data().emailContent;
23         var emailTitle = doc.data().emailTitle;
24
25         const mailOptions =
26         {
27           from: 'PMF seminari <seminari.pmf@gmail.com>',
28           to: emailAddresses,
29           subject: emailTitle,
30           html: `

${emailContent} </p>`
31         };
32
33         return transporter.sendMail(mailOptions, (error, info) =>
34           {
35             if(error)
36             {
37               return res.send(error.toString());
38             }
39             else


```

```

40         {
41             admin.firestore()
42                 .collection("emails")
43                 .doc(`${context.params.emailID}`)
44                 .delete()
45                 .then(function() {console.log("Deleted!");})
46                 .catch(function() {console.log("Error!");});
47         }
48         return res.send('Sended');
49     });
50 });
51 });

```

Linije 1-5 predstavljaju inicijalizaciju: želimo koristiti Cloud Functions, imati pristup bazi podataka, te koristiti modul nodemailer. U linijama 7-14 definiramo transporter - objekt koji će obavljati slanje emaila. U našem slučaju, možemo iskoristiti Google račun koji smo iskoristili pri Firebase inicijalizaciji. Dakle, emailovi će biti slani s adrese "seminari.pmf@gmail.com". Linijama 16-17 definiramo naziv funkcije (sendEmail) i okidač: želimo da se funkcija pokrene svaki put kad se kreira (onCreate) novi email dokument (emails/{emailID}) koji će u bazi imati id emailID. U linijama 19-23 dohvaćamo novo uneseni dokument iz baze, te spremamo željene vrijednosti u istoimene varijable, a u linijama 25-31 definiramo email koji ćemo slati koristeći te vrijednosti. U liniji 33 šaljemo mail koristeći funkciju sendMail modula nodemailer, te ako je sve prošlo u redu, izbrišemo email dokument iz baze. Napomenimo da smo informacije o imenu i id-ju predavanja preuzeli iz objekta context koji nam vraća funkcija onCreate.

3.13 Hosting

Svaki seminar Matematičkog odsjeka Prirodoslovno matematičkog fakulteta koji će htjeti koristiti našu aplikaciju će imati potpuno odvojenu aplikaciju. Budući da se na raznim mjestima unutar projekta pojavljuju podaci specifični za konkretan seminar (naziv seminara, email adresa i lozinka za slanje emailova, te podaci iz firebaseConfig objekta), možemo olakšati samo postavljanje aplikacije na sljedeći način:

1. sva pojavljivanja specifičnih podataka zamijenimo s nekim jedinstvenim stringom (primjerice email zamijenimo s {XX_EMAIL_XX})
2. u poseban .txt file (keys.txt) stavimo popis podataka koje treba unijeti pri postavljanju aplikacije, u sljedećem formatu:

```

1 EMAIL: ""
2 NAZIV_SEMINARA: ""

```

```
3 API_KEY: ""
4 ...
```

3. kreiramo Python skriptu (`appSetup.py`) koja će u potrebnim dokumentima aplikacije zamijeniti lažne stringove s dobrim vrijednostima (isječak koda prikazuje primjer unošenja emaila)

```
1 import re
2
3 keys = open("keys.txt", "r")
4
5 # EMAIL
6 line = keys.readline()
7 email = re.search("(.+?)", line).group(1)
8
9 with open('index.js', 'r') as file :
10     filedata = file.read()
11     filedata = filedata.replace('{XX_EMAIL_XX}', email)
12 with open('index.js', 'w') as file:
13     file.write(filedata)
14
15 keys.close()
```

Dakle, pri prvom postavljanju aplikacije će voditelj seminara morati unijeti sve potrebne podatke između navodnika u dokumentu `keys.txt` te pokrenuti skriptu `appSetup.py`: skripta čita `.txt` datoteku redak po redak, izvadi pravu vrijednost pomoću regularnog izraza, te sve privremene stringove datoteke (u ovom primjeru `index.js`) zamijeni pravom vrijednosti.

3.14 Konačni izgled aplikacije

Budući da smo u prethodnim poglavljima obradili dijelove aplikacije pojedinačno, sada ćemo dati širu sliku aplikacije i opisati korisničko sučelje.

Na slici 3.13 možemo vidjeti početnu stranicu koja se sastoji od tri komponente: zaglavlja i navigacijske trake (koje su zajedničke svim stranicama), te popisa nadolazećih predavanja (ispod kojeg se nalazi i popis prethodnih predavanja). Klikom na gumb “Generiraj izvještaj” otvara se padajući izbornik s popisom akademskih godina, te klikom na željenu godinu se skida izvještaj. Vidljive su samo osnovne informacije o predavanjima (vrijeme, naziv predavača i naziv predavanja). Klikom na gumb “Više info” željenog predavanja otvara se stranica sa više informacija (slika 3.14). Ako je korisnik prijavljen, može vidjeti gumbove za promjenu podataka predavanja. Klikom na gumb “Generiraj email” se

prikazuje forma koju smo već vidjeli na slici 3.12. Klikom na “Obriši” se javlja prozorčić s porukom upozorenja (slika 3.15).

The screenshot shows the header of the PMF - Matematički odsjek website. The main title is "PMF - Matematički odsjek" with the subtitle "Seminari". Below the title is a navigation bar with links: "Predavanja", "Voditelji", "Tajnici", "Članovi", "Novo predavanje", and "Osobna stranica". A blue button labeled "Generiraj izvještaj" is positioned above the main content area. The main content area is titled "Nadolazeća predavanja" and displays three upcoming seminars in a grid format. Each seminar card includes the date and time, the name of the lecturer, the title of the seminar, and a "Više info" button.

sri. 05.02.2020. 17:00	pon. 10.02.2020. 16:00	čet. 20.02.2020. 13:00
Izv. prof. dr. sc. Sara Saric	Mr. sc. Pero Perić	Prof. dr. sc. Iva Ivić
Naziv trećeg predavanja	Naziv predavanja	Naziv drugog predavanja
Više info	Više info	Više info

Slika 3.13: Popis predavanja

Izgled “Osobne stranice” smo već pokazali na slikama 3.7 i 3.8. Na slici 3.16 vidimo popis članova seminara koji otvaramo klikom na pripadni gumb navigacijske trake (na isti način otvaramo i popise voditelja i tajnika seminara). Kada je korisnik prijavljen, tada ima opcije unosa novih članova: može unositi člana po član upisivanjem informacija u HTML formu ili učitavanjem .csv dokumenta za unos više članova odjednom, kao što vidimo na slici 3.17.

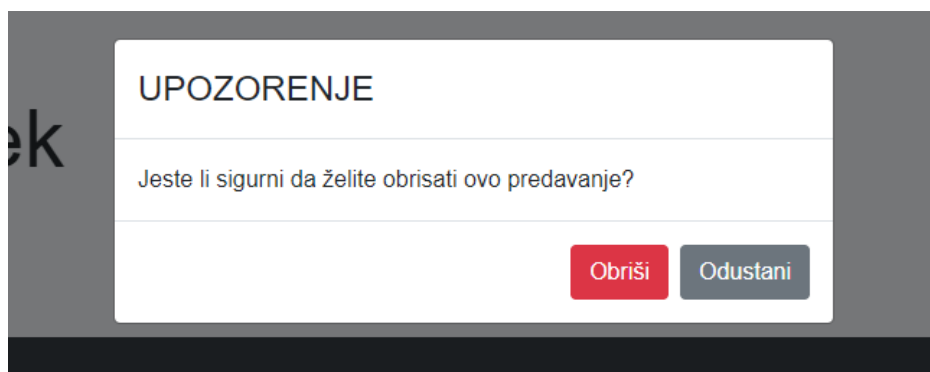
Klikom na “Novo predavanje” na navigacijskoj traci otvara se forma (slika 3.18) za unos informacija o novom predavanju (u slučaju kada korisnik nije prijavljen, prikazuje se samo poruka da mogućnost unosa imaju prijavljeni korisnici).



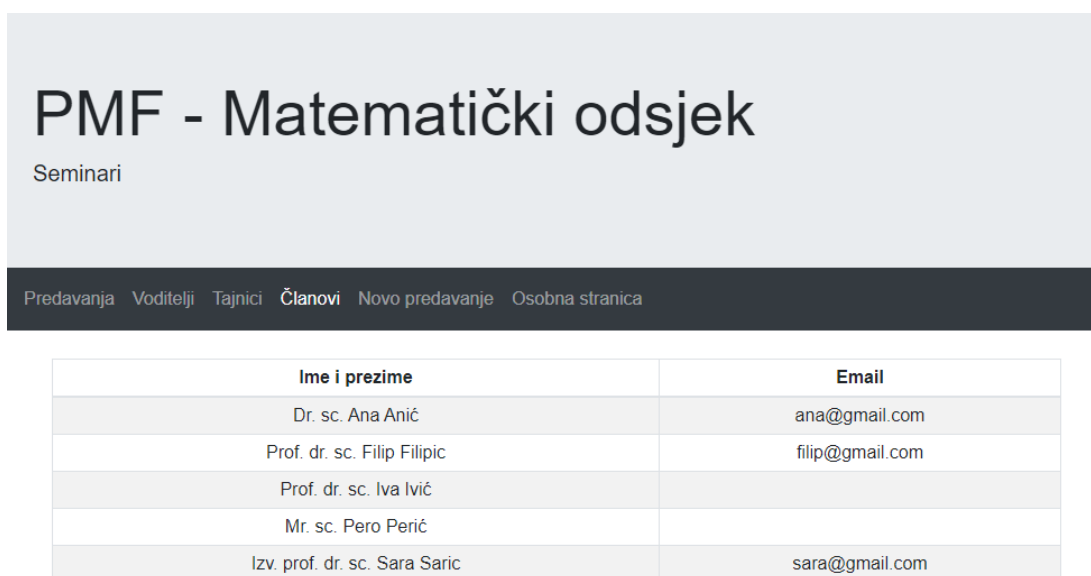
Naziv predavanja: Naziv trećeg predavanja
Predavač: Izv. prof. dr. sc. Sara Saric
Vrijeme: sri. 05.02.2020. 17:00
Tip: originalni rad
Opis: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

[Promijeni](#) [Obriši](#) [Generiraj email](#)

Slika 3.14: Pregled informacija o predavanju



Slika 3.15: Prozorčić s porukom



Slika 3.16: Popis članova

PMF - Matematički odsjek

Seminari

Predavanja **Voditelji** Tajnici Članovi Novo predavanje Osobna stranica

Odaberite .csv file s informacijama korisnika

Choose File No file chosen

Dodajte jednog korisnika

Unesite podatke novog korisnika:

titula ime prezime email

voditelj
 tajnik
 član

Dodaj člana

Ime i prezime	Email	
dr. Ivica Ivicic	ivica@ivica.com	Promijeni
dr. Ivo Ivić	ivo@ivo.com	Promijeni

Slika 3.17: Dodavanje korisnika

Unesite podatke o novom seminaru.

Predavanje

Predavač Odaberi ime ▼

U suradnji

Tip seminara

- originalni rad
- gost seminara
- gostovanje člana seminara (konferencija)
- gostovanje člana seminara (institucija)
- iz literature

Vrijeme

Datum završetka

Mjesto

Opis

Napomena

< siječanj 2020 >

po	ut	sr	če	pe	su	ne
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

00:00

Spremi

Slika 3.18: Forma za unos novog predavanja

Poglavlje 4

Rezultati

4.1 Lighthouse rezultati

Kako smo spomenuli u uvodu, za analiziranje “progresivnosti” naše web aplikacije koristit ćemo Googleov alat Lighthouse [5].



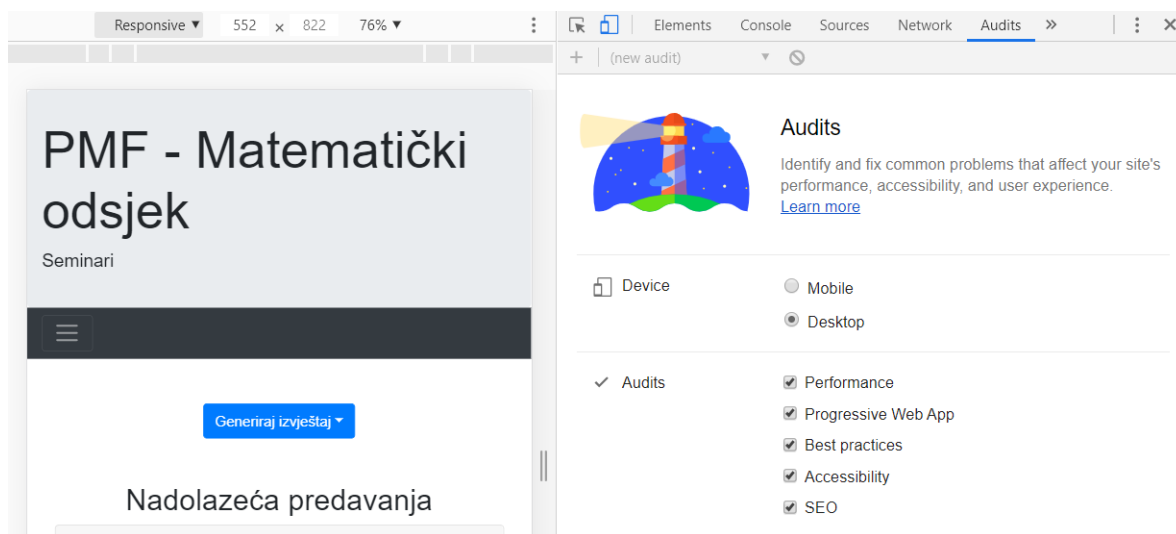
Slika 4.1: Lighthouse logo

Postoji nekoliko načina za korištenje:

- kao Node.js modul,
- unutar Chrome DevTools,
- kao Chrome ekstenziju.

Objasnimo kako ga koristiti unutar Chrome DevTools na sljedećem primjeru: otvorimo stranicu aplikacije (koristeći Chrome preglednik) i otvorimo DevTools (desni klik i odaberemo Inspect, ili `Ctrl+Shift+I`), te odaberemo tab *Audits* (slika 4.2). Nakon toga, kliknemo na *Run audits*.

Lighthouse analizira četiri vrijednosti (slika 4.3):

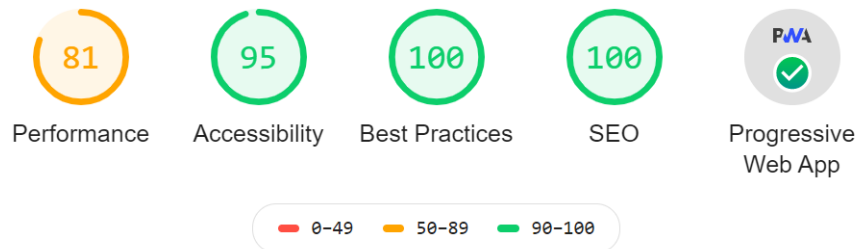


Slika 4.2: Generiranje izvještaja

- *Performance* - brzina renderiranja i vrijeme potrebno za prikaz aplikacije,
- *Accessibility* - je li aplikacija dovoljno jasna, ima li dobar kontrast boja, koliko je lako navigirati aplikacijom?
- *Best Practices* - prati li aplikacija HTTPS protokol, je li definiran *doctype*, izbjegava li korištenje zastarjelih API-jeva?
- *SEO (Search Engine Optimization)* - je li aplikacija optimizirana za rangiranje u internet pretraživačima?

Za svaku vrijednost daje pregled svih mjerenja i savjete kako popraviti loše rezultate na specifičnim mjerenjima.

Ono što nas zapravo zanima u Lighthouse izvještaju je može li se naša aplikacija smatrati progresivnom. Na slici 4.4 vidimo da zadovoljava sve uvjete za PWA koje je Google definirao (neke zahtjeve smo morali sami ispuniti, dok su se neki automatski ispunili jer smo koristili `create-react-app` i `Firebase`).



Slika 4.3: Sažetak izvještaja o našoj aplikaciji

Fast and reliable
Page load is fast enough on mobile networks
Current page responds with a 200 when offline
start_url responds with a 200 when offline
Installable
Uses HTTPS
Registers a service worker that controls page and start_url
Web app manifest meets the installability requirements
PWA Optimized
Redirects HTTP traffic to HTTPS
Configured for a custom splash screen
Sets a theme color for the address bar.
Content is sized correctly for the viewport
Has a <meta name="viewport"> tag with width or initial-scale
Contains some content when JavaScript is not available
Provides a valid apple-touch-icon

Slika 4.4: PWA uvjeti

4.2 Zaključak

Vidimo da nije teško kreirati progresivnu web aplikaciju. Tome najviše doprinose gotovi alati (dobro dokumentirani i potkrijepljeni primjerima) kao što je `create-react-app` koji nam je napravio veliki pozadinski dio pripreme, te smo se mi mogli posvetiti samim funkcionalnostima aplikacije. Postoji i zajednica programera koji kreiraju razne module i pakete koji također olakšavaju i ubrzavaju razvoj aplikacije.

Korištenje novijih tehnologija ima i svoje mane: alati koji se koriste za PWA su često u beta fazi i nisu stabilni kao neki drugi koji se razvijaju desecima godina. Kako se internet tehnologija brzo mijenja, te promjene moraju pratiti i alati, zbog čega se dokumentacija i sintaksa mijenja svakih par godina.

Firestore isto ima svoja ograničenja: za naše potrebe, svi resursi koje smo koristili bili su besplatni. Ali ako bi htjeli značajno povećati broj korisnika ili memorije koje baza zauzima, morali bi plaćati svaki resurs posebno (za usporedbu, naša aplikacija koristi manje od 5% besplatnih resursa).

Bibliografija

- [1] *Angular service worker*, <https://angular.io/guide/service-worker-intro>, siječanj 2020.
- [2] *Create React App*, <https://github.com/facebook/create-react-app>, siječanj 2020.
- [3] *Firebase*, <https://firebase.google.com/docs>, siječanj 2020.
- [4] *Ionic PWA*, <https://ionicframework.com/docs/publishing/progressive-web-app>, siječanj 2020.
- [5] *Lighthouse*, <https://developers.google.com/web/tools/lighthouse>, siječanj 2020.
- [6] *Node.js*, <https://nodejs.org/en/>, siječanj 2020.
- [7] *Nodemailer*, <https://nodemailer.com/about/>, siječanj 2020.
- [8] *Polymer - PWA Starter Kit*, <https://github.com/polymer/pwa-starter-kit>, siječanj 2020.
- [9] *Preact CLI*, <https://github.com/preactjs/preact-cli>, siječanj 2020.
- [10] *PWA zahtjevi*, <https://developers.google.com/web/progressive-web-apps/checklist>, siječanj 2020.
- [11] *React Bootstrap*, <https://react-bootstrap.github.io/getting-started/introduction>, siječanj 2020.
- [12] *React*, <https://reactjs.org/tutorial/tutorial.html>, siječanj 2020.
- [13] *Vue.js PWA*, <https://github.com/vuejs-templates/pwa>, siječanj 2020.
- [14] *Yarn*, <https://yarnpkg.com/en/docs/install>, siječanj 2020.
- [15] S. Domes, *Progressive Web Apps with React*, Packt, 2017.

- [16] A. Russell, *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*, <https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/>, 2015.
- [17] D. Sheppard, *Beginning Progressive Web App Development - Creating a Native App Experience on the Web*, Apress, 2017.

Sažetak

PWA (Progressive Web Application) je naziv za web aplikaciju koja je prilagođena za korištenje na internet preglednicima i na mobilnim uređajima, neovisno o platformi. PWA su brze i pouzdane, čak i na sporim mrežama ili potpuno offline. Mogućnosti koje aplikacija nudi ipak ovise o uređaju: najnoviji uređaj će moći koristiti sve funkcionalnosti, a stariji uređaj samo dio, odakle i dolazi naziv “progresivna” aplikacija. Neke od tehnologija koje se mogu koristiti za kreiranje PWA su: React, Preact, Angular, Vue, Ionic.

Koristeći React alat Create React App koji u samo nekoliko koraka kreira osnovnu aplikaciju i njenom nadogradnjom razvijamo aplikaciju za seminare na Matematičkom odsjeku Prirodoslovno matematičkog fakulteta. U aplikaciji je omogućena prijava korisnika, dodavanje novih predavanja i članova, generiranje izvještaja i slanje emaila. Prikazujemo osnovne koncepte korištenja React biblioteke na isječcima koda iz aplikacije (renderiranje, routing, state, props, event handling). Kao bazu podataka i sustav autentifikacije koristimo Firebase. Mogućnost instaliranja aplikacije na mobilni uređaj ostvarujemo manifest datotekom, a offline rad i spremanje u priručnu memoriju generiranim service workerom. Na kraju analiziramo aplikaciju alatom Lighthouse koji potvrđuje da je naša aplikacija zaista PWA.

Summary

PWA (Progressive Web Application) is a name for a web application that can be used on different web browsers and mobile devices, not depending on the platform or the device. PWAs are fast, reliable and work on slow networks and even offline. Functionalities that an app offers do depend on the device: the newest mobile would have an access to all functionalities, while an older device could only use its portion, hence the word “progressive” application. Some of the technologies used in creating a PWA are: React, Preact, Angular, Vue, Ionic.

Using a React tool Create React App we can create a basic application in just a few steps, which we then rework to create an application for seminars that are held at the Department of Mathematics of the Faculty of Science. We offer user login, ability to create new lectures, add new members, generate a report and send emails. Using snippets of the app code we explain some main React concepts (rendering, routing, state, props, event handling). We use Firebase tools to handle our database and authentication. App installation is enabled by a manifest file, while offline work and caching is enabled by a service worker. In the end, we use a tool called Lighthouse which confirms that our created app is in fact a PWA.

Životopis

Anastasija Jezernik rođena je 08.04.1995. u Čakovcu. Nakon Osnovne škole Podturen, pohađa Gimnaziju Josipa Slavenskog Čakovec. Godine 2014. upisuje preddiplomski studij matematike na Prirodoslovno matematičkom fakultetu u Zagrebu. Kao diplomski studij odabire Računarstvo i matematiku. Tijekom studija radi kao demonstrator na kolegiju Modeli geometrije. Već pola godine radi kao junior game developer na mobilnim aplikacijama za iOS i Android.