Analiza i primjena progresivnih web tehnologija u razvoju programske podrške za trenutno slanje poruka

Ćurčija, Ivica

SVEUČILIŠTE U DUBROVNIKU

ODJEL ZA ELEKTROTEHNIKU I RAČUNARSTVO

IVICA ĆURČIJA

ANALIZA I PRIMJENA PROGRESIVNIH WEB TEHNOLOGIJA U RAZVOJU PROGRAMSKE PODRŠKE ZA TRENUTNO SLANJE PORUKA

DIPLOMSKI RAD

Dubrovnik, rujan, 2022.

SVEUČILIŠTE U DUBROVNIKU

ODJEL ZA ELEKTROTEHNIKU I RAČUNARSTVO

ANALIZA I PRIMJENA PROGRESIVNIH WEB TEHNOLOGIJA U RAZVOJU PROGRAMSKE PODRŠKE ZA TRENUTNO SLANJE PORUKA

DIPLOMSKI RAD

Studij: Primijenjeno/poslovno računarstvo

Kolegij: Elektroničko poslovanje

Mentor: prof.dr.sc. Vedran Batoš

Komentor: Ivan Grbavac, dipl.ing.

Student: Ivica Ćurčija

Dubrovnik, rujan, 2022.

SADRŽAJ

SAŽETAK

Razvoj mobilnih aplikacija je zahtjevan proces zbog različitih arhitektura mobilnih sustava i potrebe za pisanjem različitih baza koda za svaku vrstu uređaja (Android, iOS, Windows). Progresivne web aplikacije su hibridne web aplikacije koje izgledaju i ponašaju se kao nativne mobilne aplikacije. Cilj progresivnih web aplikacije je smanjiti razlike između nativnih i web aplikacija koristeći moderne metode, alate i tehnologije za stvaranje najboljeg korisničkog iskustva. Progresivne web tehnologije mogu biti veoma korisne u zemljama u razvoju gdje su nerijetko prisutne spore internet veze, a zbog načina predmemoriranja podataka i same veličine potpunih progresivnih web aplikacija idealni su za takva područja. Još jedna prednost ovakvog pristupa je jednostavnost izrade (jedna programska baza) te jednostavna prilagodba na različite uređaje. U ovom radu analizirane su metode koje se koriste tijekom stvaranja i projektiranja progresivnih web aplikacija, te će se na primjeru programske podrške za trenutno slanje poruka prikazati korištenje tih istih metoda. Aplikacija će se razvijati koristeći React, Node.js, Express, PostGreSQL, Redis i druge manje biblioteke.

Ključne riječi: progresivne web aplikacije, progresivne web tehnologije, web razvoj, React, Node.js, mobilni razvoj, PWA

ABSTRACT

Developement of mobile applications is a demanding process because of the different architectures of mobile systems and the need to write different code bases for each type of device (Android, iOS, Windows). Progressive web applications are hybrid web applications that look and feel just like native mobile applications. The goal of progressive web applications is to reduce the differences between native and web applications using modern methods, tools and technologies to create the best user experience. Progressive web technologies can be very useful in developing countries where low internet speeds are not uncommon, because of the way caching data and the actual size of full progressive web applications they are perfect for those areas. Another advantage of this approach is ease of development (1 code base) and simple adjustment to different devices. This paper will analyze methods used to create and design progressive web applications, and show the use of those methods on the example of an instant messaging app. The Application will be developed using React, Node.js, Express, PostGreSQL, Redis and some smaller packages.

Keywords: progressive web applications,progressive web technologies, web development, React,Node.js, mobile development, PWA

1. UVOD

Tehnologija se učestalo mijenja, broj mobilnih uređaja i korisnika raste iz dana u dan. S takvim okruženjem raste i potreba za softverom za mobilne uređaje. Do sada su se za razvoj mobilnih aplikacija koristile dvije metode: razvoj nativnih aplikacija i razvoj mobilnih web aplikacija. Kroz vrijeme mobilne web aplikacije postajale su stvar prošlosti tako da je danas teško pronaći uspješnu web aplikaciju koja nije razvijana u nativnim tehnologijama. Zbog modernijeg hardvera i softvera mobilni uređaji imaju veće procesorske sposobnosti te mogu pokretati napredne aplikacije koje pružaju puno bolje iskustvo od korištenja tradicionalnih mobilnih web aplikacija. Zbog toga je stvorena ideja progresivnih web aplikacija, one su hibrid između nativnih i web aplikacija, bazirane su na standardnim web tehnologijama ali se u rukama korisnika ponašaju slično nativnim aplikacijama.

Cilj rada je upoznati se s progresivnim web tehnologijama, analizirati njihove karakteristike, usporediti ih s nativnim tehnologijama te na primjeru programskog rješenja za trenutno slanje poruka pokušati primijeniti iste.

U drugom poglavlju rada pojašnjavaju se pojmovi progresivnih web aplikacija te metoda korištenih za izgradnju tih aplikacija, daju se usporedbe s nativnim aplikacijama i primjeri postojećih progresivnih web aplikacija.

U trećem poglavlju opisuju se alati i okruženja koji će se koristiti za izradu programskog rješenja.

Četvrto poglavlje sadrži detaljan opis izrade programskog rješenja, sadrži opise podataka, klijentske aplikacije kao i poslužiteljske aplikacije. Prikazani su ekrani aplikacije i programski kod koji stoji iza njih.

Peto poglavlje se odnosi na analizu aplikacije koristeći Google Lighthouse alat za analizu i ocjenjivanje osnovnih metrika PWA aplikacija.

2. Progresivne web tehnologije

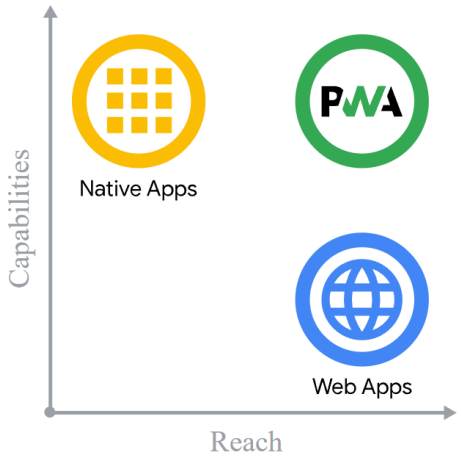
2.1 Progresivne web aplikacije

Progresivne web aplikacije (PWA) su hibrid nativnih i web aplikacija. PWA postaju sve popularnije zbog jednostavnog načina razvoja te sve većom potrebom za dostupnost aplikacija na različitim sustavima. PWA se mogu pokretati kao web stranica ili aplikacija, na samostojećim računalima ili mobilnim uređajima uz korištenje jedne programske baze. Imaju mogućnost slanja web *push* obavijesti, rada bez mreže, pristup hardverskim funkcionalnostima kao i veće brzine učitavanja stranice u usporedbi s uobičajenim web aplikacijama.

PWA imaju jednake mogućnosti kao i web stranice:

- Mogu biti indeksirane od strane tražilica
- Može im se pristupiti putem hiperlinka
- Može raditi na svim uređajima

PWA omogućuju jednostavniji i brži razvoj za višeplatformske aplikacije u usporedbi s nativnim aplikacijama koje zahtijevaju specifičnu programsku bazu za svaku platformu (Android, iOS, Windows i sl.)



Slika 1. Prikaz sposobnosti i dosega nativnih aplikacija, web aplikacija i PWA

2.2 Karakteristike PWA

U ovom poglavlju dan je pregled karakteristika progresivnih web aplikacija. Karakteristike PWA su :

- Progresivne – bez obzira na preglednik ili operacijski sustav koji se koristi i bez obzira na kvalitetu internet veze, PWA rade za sve korisnike,

- Responzivne – PWA mogu raditi na bilo kakvoj vrsti uređaja. Omogućuju rad na desktop računalima, mobilnim uređajima, tabletima i uređajima koji tek dolaze,
- Način korištenja – zbog načina na koji su PWA dizajnirane kroz sustav navigacije i interakcije s aplikacijom, korisnik ne može razlikovati PWA od nativnih aplikacija,
- Neovisne o povezanosti – *service workeri* pomažu progresivnim web aplikacijama tako što omogućuju rad bez mreže ili na mrežama s lošom kvalitetom,
- Instalacija – korisnici mogu spremati PWA na početni zaslon bez korištenja *App Storeova,*
- Uočljive – Progresivne web aplikacije su prepoznatljive kao aplikacije. *Service workeri* i manifest datoteka omogućuju tražilicama da lako otkriju PWA. Principi koji se koriste kod razvijanja ovakvih aplikacija ujedno odgovaraju SEO zahtjevima,
- Privlačne – Funkcionalnosti kao što su *push* notifikacije zadržavaju korisnike zainteresirane za aplikaciju. Svojom prezentacijom se ne razlikuju puno od nativnih aplikacija te imaju bolje metrike zadržavanja korisnika. [1]

2.3 Povijest

Tijekom predstavljanja prvog iPhonea 2007. godine, Steve Jobs je prvi puta predstavio ideju da će web aplikacije (razvijene u HTML-u , koristeći AJAX arhitekturu) biti standardni format iPhoneovih aplikacija. Aplikacije bi bile potpuno integrirane u uređaj koristeći Safari (web preglednik za Apple uređaje). Ovakav model je ubrzo zamijenjen s AppStoreom zbog nezadovoljstva programera i povećavanja sigurnosti uređaja te je ubrzo najavljeno prvo razvojno okruženje za razvoj nativnih aplikacija.

Ideja „univerzalnih aplikacija“ je nakon toga nestala iz javnog prostora, te su u to vrijeme nativne aplikacije postale primarni način korištenja interneta na mobilnim uređajima. Korisnici mobilnih uređaja za otkrivanje i instaliranje novih aplikacije koriste razne trgovine aplikacijama koje dolaze već instalirane na uređaj. App Store (Apple) i Google Play Store pokrenuti su 2008. godine. To su dvije najveće platforme za mobilnu trgovinu aplikacijama, filmovima, glazbom i sl.

Uz porast nativnih aplikacija, pojavila se nova paradigma, „Responzivni web dizajn“. Ovaj pristup omogućuje da se ista web stranica jednako dobro prikazuje na različitim uređajima. Nažalost aplikacije dizajnirane na ovaj način nisu imale jednak izgled i funkcionalnosti te su u tom aspektu bile lošije od nativnih aplikacija.

Responzivni web dizajn nikad nije u potpunosti uspio ispuniti zahtjeve korisnika, s napretkom tehnologije tj. boljim hardverom i softverom za mobilne uređaje, web aplikacije na mobilnim uređajima su se sve manje koristile.
2015. godine dizajner Frances Berriman i Google Chrome inženjer Alex Russell osmislili su izraz „Progresivne web aplikacije“ kojim opisuju aplikacije koje koriste funkcionalnosti modernih web preglednika kao što su *service workeri* i manifesti web aplikacija, te omogućuju korisnicima da pretvore web aplikacije u progresivne web aplikacije u svojim nativnim operacijskim sustavima. Firefox je omogućio korištenje *service workera* 2016. godine, kao i Microsoft Edge i Apple Safari 2018. Danas web aplikacije se mogu distribuirati putem Google Play, Microsoft Storea, Apple AppStorea i Samsung Galaxy Storea [2]

2.4 Zahtjevi progresivnih web aplikacija

U ovom poglavlju daje se pregled glavnih zahtjeva potrebnih za rad progresivnih web aplikacija. Ti zahtjevi su: Sigurna veza, *service workeri i web manifest datoteka.*

2.4.1 Sigurna veza

Za svrhe PWA potrebno je pružiti siguran poslužitelj sa HTTPS vezom. HTTPS je sigurna verzija HTTP-a, glavnog protokola za slanje podataka između preglednika i web stranice. HTTPS je kriptiran kako bi povećao sigurnost tijekom razmjene podataka. Na ovaj način se štite korisnički podatci kao i gradi dodatni sloj sigurnosti unutar web aplikacije.

2.4.2 Service workeri

Jedan je od ključnih elemenata za rad PWA, koji služi za upravljanje mrežnim zahtjevima. *Service worker* je klijentski JavaScript dokument koji radi u pozadini, čak i kada je aplikacija zatvorena, kako bi reagirao na događaje te služi kao spremnik za statične elemente unutar aplikacije. Pokreće se u kontekstu pružatelja usluge, nema DOM (eng. Document Object Model) pristup, i radi na odvojenoj dretvi od glavne JavaScript dretve koja pokreće aplikaciju.

2.4.3 Web app manifest datoteka

Web app manifest je JSON tekst datoteka koja sadrži podatke nužne kako bi se PWA mogla preuzeti i prikazati korisniku na sličan način kao i nativna aplikacija (npr. instalirati aplikaciju

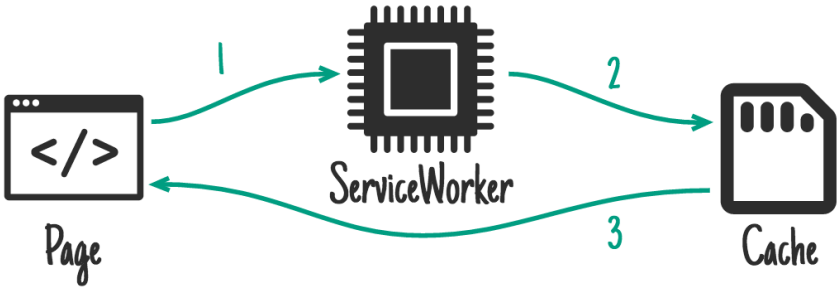
na radnu površinu uređaja). Manifest datoteka sadrži ime aplikacije, autora, ikone, verziju, opis, popis nužnih resursa i sl.

2.5 Strategije predmemoriranja

Strategije predmemoriranja (eng. *cache*) su različiti načini na koje se može izvesti funkcija *service workera*. Ova strategija se odnosi na interakciju između *service workerovog* "*fetch*" događaja i sučelja za predmemoriranje. Ovisno o potrebi, strategija se može napisati tako da se dokumenti drukčije dobavljaju u odnosu na statične elemente.

2.5.1 Isključivo predmemoriranje

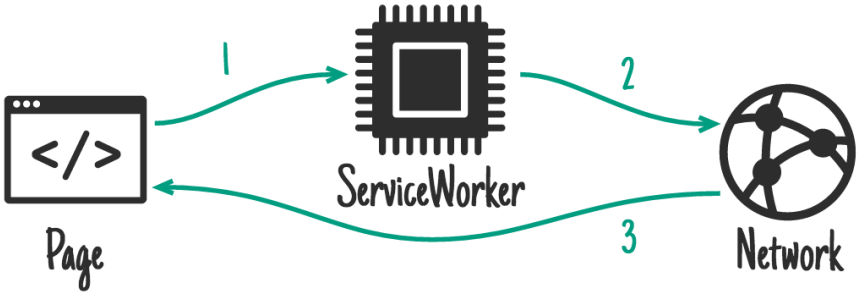
Ova strategija dohvaća odgovore samo iz predmemorije (eng. *Cache only*), ne oslanja se na mrežu. Strategija se koristi kod elemenata koji se nikad neće mijenjati nakon što su prvi put spremljeni u predmemoriju. Ti elementi se spremaju jednom i poslužuju samo iz memorije.



Slika 2. Isključivo predmemoriranje (eng. Cache only)

2.5.2 Isključivo mrežno

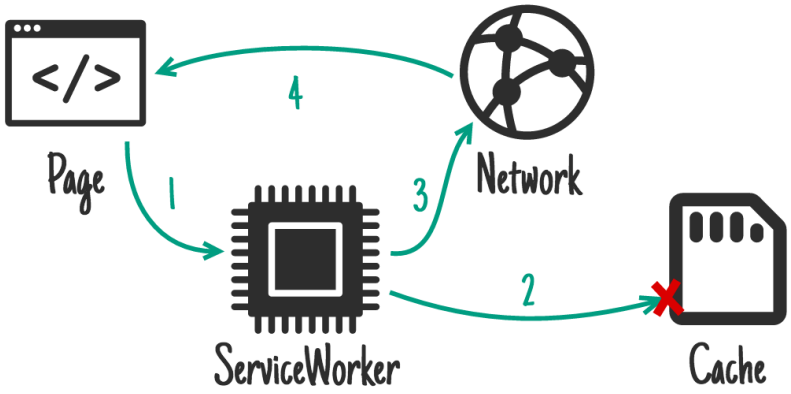
Suprotna strategiji isključivog predmemoriranja, ova strategija funkcionira tako da se zahtjevi prosljeđuju kroz *service worker* bez korištenja predmemorije. Ovakav pristup omogućuje da se korisniku prikazuje najnoviji sadržaj, ali loša karakteristika ovog pristupa je da ne radi kada je korisnik bez mreže. Ovakav pristup se koristi kod podataka koji se učestalo mijenjaju.



Slika 3.Isključivo mrežno (eng. Network only)

2.5.3 Prvo predmemoriranje

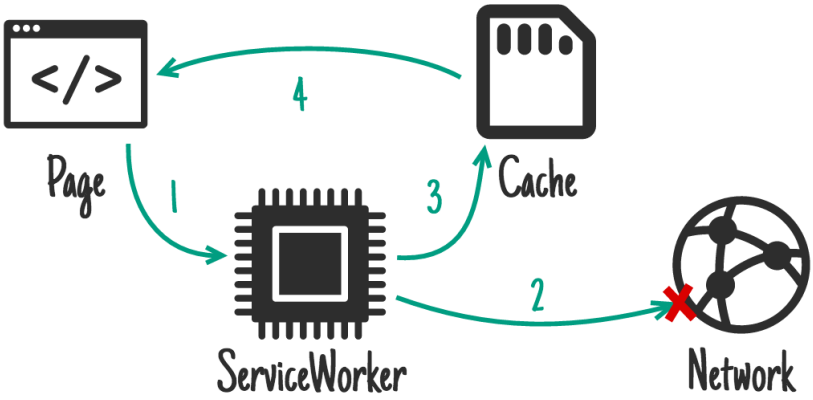
Za odgovaranje na zahtjeve, elementi se prvo pretražuju u predmemoriji ako su spremljeni poslužuju se iz predmemorije. Ako elementi iz zahtjeva nisu dostupni, zahtjev se šalje na mrežu, te se nakon toga pridodaju u predmemoriju i vraća se odgovor sa mreže.



Slika 4. Prvo predmemoriranje (eng. Cache first)

2.5.4 Prvo mrežno

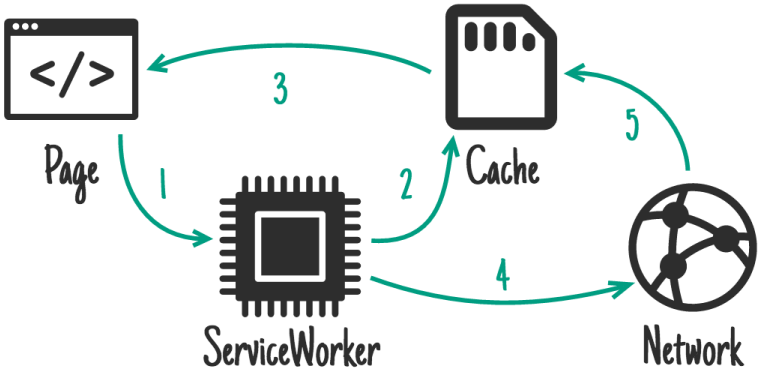
Prvo se pristupa mreži sa zahtjevom, odgovor se sprema u predmemoriju. Ako korisnik nema pristup mreži i elementi su dostupni u predmemoriji, onda se dobavljaju iz predmemorije. Ovakav pristup je dobar za HTML ili API zahtjeve kada tijekom rada na mreži, je potrebno dobavljati najnoviju verziju resursa, a tijekom rada bez mreže imati mogućnost dobaviti najnoviju dostupnu verziju.



Slika 5. Prvo mrežno (eng. Network first)

2.5.5 Strategija „*Stale-while-revalidate*“

Strategija koja prioritet stavlja na brzinu pristupa resursu, dok istovremeno pokušava održavati najnoviju verziju istog resursa u pozadini. Tijekom prvog zahtjeva za resursom, on se dobavlja s mreže, sprema u predmemoriju, te se vraća odgovor s mreže. Kod narednih zahtjeva resurs se dobavlja iz predmemorije, a u pozadini se ponovno šalje zahtjev te se resurs ažurira u predmemoriji. Resursi su ažurirani u predmemoriji ali promjene mogu dovesti do nesinkroniziranih podataka između predmemorije i onoga što korisnik vidi.

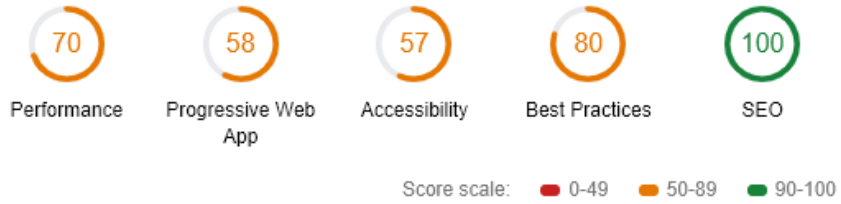


Slika 6. Stale while validate

Ovakav pristup se koristi kada je potrebno elemente održavati relativno novim, ali nije nužno. Dobar primjer su slike profila ili avatari koji ne moraju nužno biti aktualni. [3]

2.6 Google Lighthouse

Google Lighthouse je besplatan alat otvorenog koda koji služi za analizu web stranica. Dostupan je kao proširenje za Google Chrome preglednik, kao Node.js modul, web korisničko sučelje, te mu se može pristupiti putem DevTools menija. Lighthouse kroz samo nekoliko klikova evaluira web stranicu na temelju pet kategorija: performanse, pristupačnost, najbolje prakse, SEO i progresivna web aplikacija. Ova područja se ocjenjuju od 1-100 te Lighthouse dodatno komentira svaku kategoriju pojedinačno i upozorava na probleme unutar web stranice.



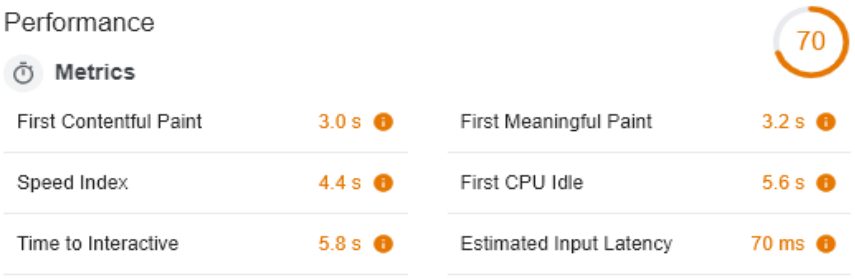
Slika 7. Lighthouse metrike

Rezultati 0-49 se smatraju nedovoljnima, 50-89 dovoljno dobrima te 90-100 odličnim.

2.6.1 Performanse

U ovoj kategoriji Lighthouse analizira koliko brzo se web stranica ili aplikacija učitava i koliko brzo korisnik može pregledati ili pristupiti sadržaju. Analizira se 6 glavnih metrika brzine:

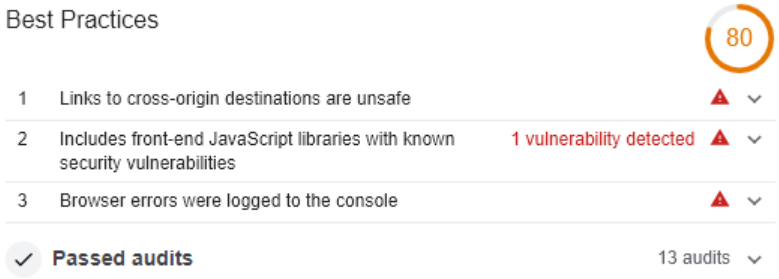
- „*First Contentful Paint*“ - vrijeme prije prikazivanja prve slike ili teksta korisniku,
- „*First Meaningful Paint*“- vrijeme u kojemu glavni sadržaj stranice postaje vidljiv korisniku,
- „*Speed Index*“- ujednačenametrika brzine učitavanja sadržaja stranice,
- „*Time to Interactive*“ – vrijeme u kojemu korisnik ne može u potpunosti koristiti stranicu,
- „*First CPU Idle*“ – vrijeme u kojemu je aktivnost glavne dretve stranice dovoljno niska da se ulazi mogu obraditi,
- „*Estimated Input Latency*“ – procjenakoliko aplikaciji treba do odgovori na ulaz tijekom najprometnijih 5 sekundi učitavanja stranice.



Slika 8. Metrike performansi Lighthouse analize

2.6.2 Najbolje prakse

Lighthouse testira 16 najboljih praksa koje se koncentriraju na sigurnosne aspekte web stranice i moderne standarde web razvoja. Lighthouse analizira koristi li se HTTPS i HTTP/2, te da li resursi dolaze iz sigurnih izvora te određuje slabosti Javascript biblioteka. Druge najbolje prakse uključuju korištenje sigurnih veza na bazu podataka, izbjegavanje korištenja nesigurnih naredbi kao što su *document.write()* te korištenje zastarjelih API-ja. Na slici 9 je prikazan rezultat Lighthouse analize koji upućuje na tri greške: nesigurnu vezu na drugo odredište, JavaScript biblioteku sa sigurnosnim problemima i ispis grešaka u konzolu preglednika.



Slika 9. Lighthouse analiza, Best Practices

2.6.3 SEO

SEO (eng. *Search Engine Optimization*) je skup praksi koje služe za poboljšavanje pozicije i izgleda web stranica u okviru organskih rezultata tražilica. Organski rezultati tražilica su neplaćeni rezultati koje je tražilica odredila kao najrelevantnije za korisnikov upit. Lighthouse trenutno pokreće 13 provjera unutar SEO kategorije. Ove provjere se fokusiraju na prilagođenost mobilnim uređajima ili PWA, točnu primjenu strukturiranih podataka i oznaka kao što su *hreflang*, *title* i *meta* opisi, te provjeru da li roboti za tražilice mogu pretraživati (eng. *crawl*) stranicu.

2.6.4 Pristupačnost

Provjere pristupačnosti (eng. *accessibility*) provjeravaju koliko dobro se web stranica ili aplikacija može koristiti od strane osobama s invaliditetom. To uključuje ispitivanje važnih elemenata kao što su gumbovi ili linkovi, da li su dovoljno dobro opisani, da li su slikama pridodani *alt-attribute* vrijednosti tako da se vizualni sadržaj može opisati čitačima ekrana za slabovidne ljude.

2.6.5 Progressive Web App

Provjerava glavnih funkcionalnosti web stranice ili aplikacije bez kojih ona ne može biti PWA. Neke od tih provjera su : da li stranica registrira *service workera*, da li radi bez mreže, da li vraća grešku s kodom 200, itd..

2.7 Usporedba PWA i nativnih aplikacija

Tablica 1. Usporedba PWA i nativnih aplikacija

	PWA	Nativne aplikacije
Instalacija	Jednostavno preuzimanje direktno putem preglednika (moguć rad i bez instalacije)	Preuzimanje s *AppStorea*
Pristup funkcionalnostima uređaja	Ograničen pristup	Mogu pristupiti svim hardverskim i softverskim funkcionalnostima
Rad bez mreže	Predmemoriranje omogućuje korištenje bez mreže, ali s ograničenim funkcionalnostima	Rad bez mreže je u potpunosti omogućen
Ažuriranja	Automatizirana i trenutna	Ručna ažuriranja pokrenuto od strane korisnika
Trajanje razvoja	Jednostavan i brz razvoj	Potrebno je duže vrijeme za razvoj

Sigurnost	HTTPS šifriranje	Dodatna sigurnost se može ugraditi ovisno o korištenom operacijskom sustavu
Push notifikacije	Nisu dostupne na iOS uređajima	Dostupne na svim uređajima

Prednost PWA i samog pristupa razvoju takvih aplikacija je što omogućuje i prilagođavanje postojećih web aplikacija u PWA format te povećavanje performansi i korisničkog iskustva za sve uređaje. PWA su manje veličine, što daje veliku prednost u novim tržištima koja nemaju pristup brzom internetu.

2.8 Primjeri postojećih PWA

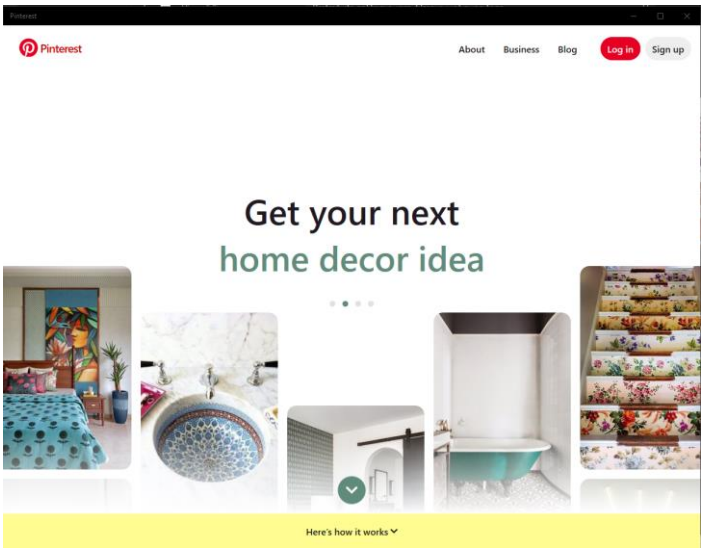
Mnoge velike kompanije s postojećim web aplikacijama su pokrenule svoje PWA kao alternativa za korištenje njihovih web aplikacija. Veliki broj ovih aplikacija se predstavljaju kao "*Lite*" verzije postojećih aplikacija, tako Google ima cijelu skupinu "Go" aplikacija kao što su Google Go, Gmail Go, YouTube Go i slično. Osim Googla postoje mnoge druge poznate kompanije koje su odlučile svoje aplikacije pretvoriti u PWA ili stvoriti posebne PWA za svrhe probijanja na nova tržišta. Neke od takvih aplikacija su :

- Pinterest,
- Twitter,
- Starbucks,
- Uber,
- George.com,
- Spotify,
- BMW,
- MakeMyTrip.

2.8.1 Pinterest

Pinterest je svoje mobilno web iskustvo iznova izgradio koristeći React, Redux i Webpack, što je dovelo do porasta samog vremena korištenja aplikacija, kao i osnovnih poslovnih metrika. Stara web aplikacija na mobilnim uređajima se sporo učitavala što prikazuju metrike „*First Paint*" i „*Time to Interactive*", koje označavaju vrijeme potrebno da preglednik počne

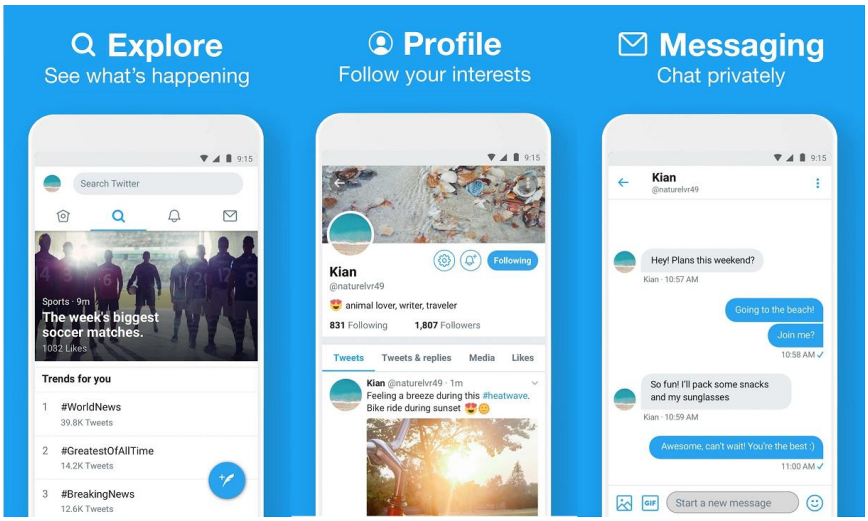
prikazivati sadržaj i vrijeme koje je potrebno da korisnik može koristiti stranicu. „*First Paint*“ je tada iznosio 4.2 sekunde, dok je „*Time to Interactive*“ iznosio 23 sekunde. S novom web aplikacijom veličina JavaScript paketa je smanjena sa 650kB na 150kB te se brzina učitavanja stranice znatno smanjila. „*First Paint*“ je iznosio 1.8 sekundi i „*Time to Interactive*“ 5.6 sekundi. Ovo su rezultati koristeći sporu 3G mrežu. Performanse web aplikacije su se dalje poboljšale zbog rada *service workera* koji nakon svakog sljedećeg pokretanja aplikacije smanjuje vrijeme učitavanja stranice. [4]



Slika 10. Pinterest PWA

2.8.2 Twitter

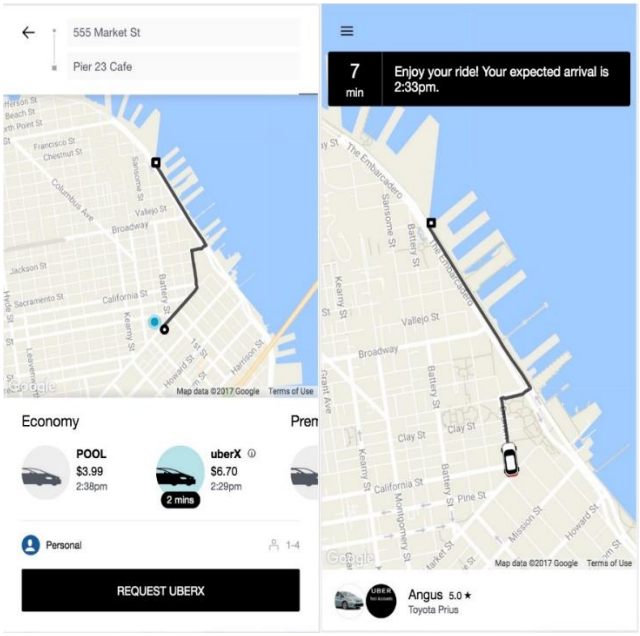
Twitter je 2017. godine pokrenuo Twitter Lite, svoj prvi PWA. Kroz razvoj svog web API-ja (eng. *Application Programming Interface*) omogućili su stvaranje minimalne aplikacije koja omogućuje brzo i responzivno sučelje, manju veličinu same aplikacije, *push* obavijesti i korištenje bez mreže. Arhitektura se sastoji od klijentske JavaScript aplikacije i jednostavnog Node.js poslužitelja. Poslužitelj osigurava osnovne radnje unutar aplikacije kao što su autentikacija korisnika, inicijalizira aplikaciju, i prikazuje HTML ljusku aplikacije. Kada se aplikacija učita u preglednik, dohvaća podatke direktno od Twitter API-ja. [5]



Slika 11. Twitter Lite

2.8.3 Uber

Kao alternativa nativnoj Uber aplikaciji, stvorena je m.uber aplikacija koja je PWA. M.uber zadovoljava potrebe korisnika da koriste Uber aplikaciju na jeftinijim uređajima, kao i onim koji nisu podržani od strane nativne aplikacije. Aplikacija je iznimno male veličine, sa samo 50kB moguće je koristiti i na 2G mrežama.



Slika 12. m.Uber

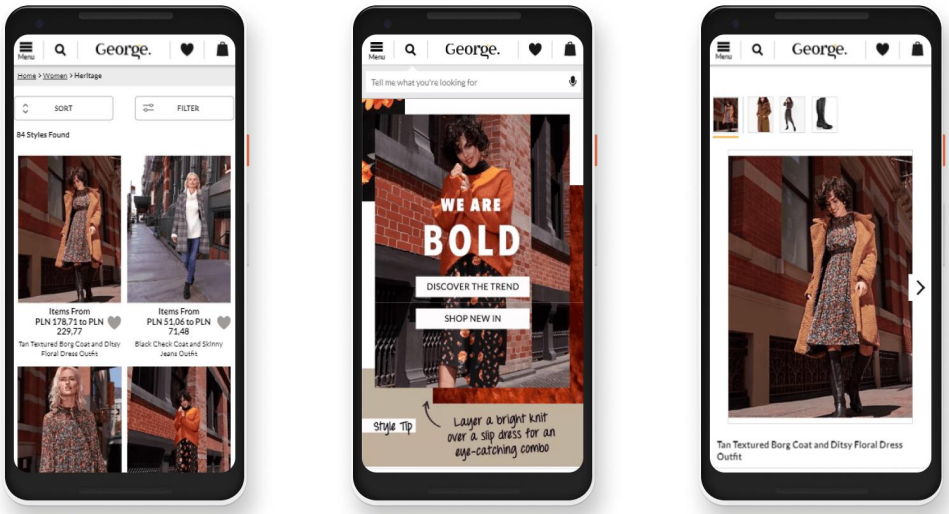
M.uber je napisan u ES2015+, koristeći Babel kao kompajler. Nativna Uber aplikacija je izgrađena koristeći React (uz Redux) i Browserify, koje su u m.uber aplikaciji zamijenjene sa Preactom i Webpackom. Razlog prijelaza s Reacta na Preact je sama veličina paketa, Preact

iznosi 3kB dok React zauzima 45kB. Preact je dobra alternativa za React ako je glavni fokus na performanse aplikacije.

2.8.4 George.com

George.com je jedna od vodećih robna marka u Ujedinjenom Kraljevstvu. Nakon nadogradnje njihove web stranice u PWA došlo je do porasta u mobilnoj konverziji od 31%. Kao i drugih mjerljivih promjena u radu aplikacije:

- 3.8 puta brže vrijeme učitavanja,
- 2 puta manja stopa odstupanja,
- 20% više pregleda stranica po posjeti,
- 28% duže prosječno vrijeme za posjete sa početnog ekrana. [6]



Slika 13. George.com PWA

3. Opis alata i okruženja korištenih za izradu programskog rješenja

Kako bi se realizirao praktični dio rada, potrebno je izraditi progresivnu web aplikaciju za trenutno slanje poruka. Programsko rješenje bit će podijeljeno u dva dijela a to su korisnički i pozadinski dio.

Za razvoj korisničke komponente programskog rješenja koristit će se React.js kao glavni JavaScript okvir, ChakraUI za komponente korisničkog sučelja i Formik za kontrolu unosa.

Za razvoj pozadinske komponente programskog rješenja koristit će se NodeJS, Express i Socket.io za razmjenu poruka. Neki manji razvojni okviri će se koristiti za manje zadatke kao sto su helmet, yup i sl.

Za spremanje podataka koristit će se dvije baze podataka PostGreSQL i Redis.

Kao upravitelj paketa koristit će se npm.

3.1 React

React.js ili React je JavaScript biblioteka otvorenog koda koja se koristi za stvaranje korisničkih sučelja za web aplikacije. Nastao je 2013. godine, a danas je jedna od najkorištenijih biblioteka za web razvoj sučelja. React olakšava gradnju korisničkih sučelja tako što stvara izolirane cjeline unutar same stranice, ove cjeline nazivamo komponente. Svaka komponenta je JavaScript funkcija koja vraća kod koji predstavlja dio web stranice. React koristi JSX, sintaksno proširenje za JavaScript. Umjesto razdvajanja tehnologija kao što su HTML i logičkih elemenata JavaScripta, JSX omogućuje sintaksu koja sadrži oboje.

React olakšava izgradnju dinamičnih web aplikacija jer zahtjeva manje koda i nudi više funkcionalnosti u usporedbi s JavaScriptom. Nudi poboljšane performanse jer koristi Virtualni DOM (eng. *Document Object Model*). Virtualni DOM uspoređuje stanje komponente s njenim prijašnjim stanjem i ažurira samo komponente koje su promijenjene bez ažuriranja svih komponenti, kao što to rade tradicionalne web aplikacije.

Prednost korištenja komponenti u razvoju React aplikacija je recikliranje koda za višekratnu upotrebu. Komponente sadrže logiku i kontrole i mogu se ponovno koristiti koliko je god potrebno puta unutar aplikacije, što drastično smanjuje vrijeme razvoja.

React prati jednosmjeran tok podataka, to znači prilikom razvoja web aplikacije, programeri često *child* komponente postavljaju unutar roditeljskih komponenti. S obzirom na to da je protok jednosmjeran, lakše je detektirati greške i znati gdje je nastao problem. [7]

3.2 ChakraUI

ChakraUI je moderna biblioteka komponenti za React. Prednost ove biblioteke je jednostavnost korištenja, modularnost i pristupačnost. Sama biblioteka se sastoji od svih osnovnih komponenata potrebnih za gradnju web aplikacije: gumbi, prostori za tekst, ikone, komponente za raspored, navigaciju i sl. Za instalaciju potrebno je u terminal upisati naredbu:

```
$ npm install --save @chakra-ui/react
```

3.3 Formik

Formik je besplatna biblioteka otvorenog koda za React. koja olakšava tri glavna problema kod stvaranja komponenti forme u Reactu:

- kako se manipulira stanjem,
- kako se vrši validacija i greške,
- kako se forme podnose.

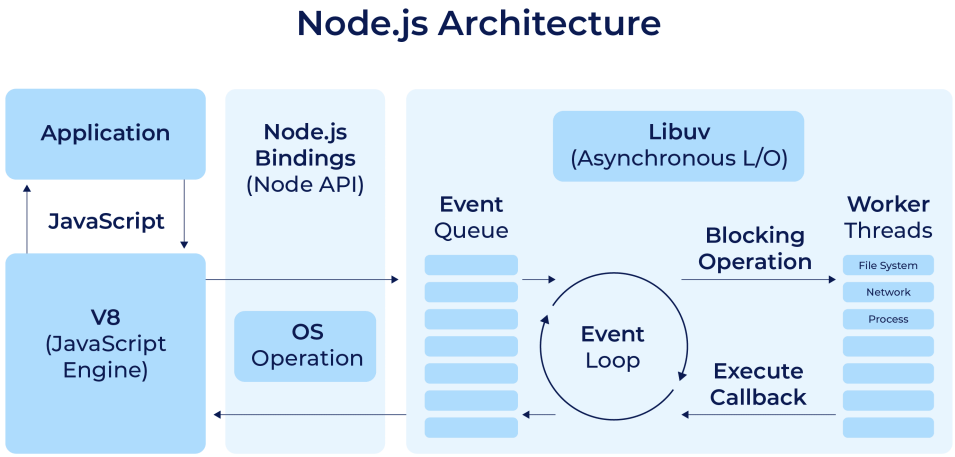
Formik je fleksibilna biblioteka, može se koristiti s HTML poljima za unos ili se mogu stvoriti prilagođena pravila validacije s Yup-om. Yup je JavaScript graditelj shema za raščlanjivanje vrijednosti i validaciju, dozvoljava definiranje sheme, pretvaranje vrijednosti u potrebni format, validaciju oblika postojeće vrijednosti ili oboje.

3.4 Node.js

Node.js je pozadinska platforma za izgradnju mrežnih aplikacija izgrađena na Google Chrome JavaScript Enginu (v8 Engine). Stvoren je 2009. godine i trenutna najnovija verzija je v0.10.36. Stvoren je za izgradnju brzih i skalabilnih mrežnih aplikacija. Node.js koristi, ne blokirajući I/O model baziran na događajima što ga čini laganim i efikasnim, savršenim za podatkovno intenzivne trenutne aplikacija koje rade preko distribuiranih uređaja. Aplikacije u node.js-u su napisane u JavaScriptu i mogu se pokretati koristeći Node.js runtime na OS X, Microsoft Windowsu i Linuxu. Njegove glavne funkcionalnosti su:

- Asinkron i baziran na događajima – sve API Node.js biblioteke su asinkrone , ne blokirajuće. To znači da poslužitelj temeljen na Node.js-u nikad ne čeka da mu API vrati podatke. Poslužitelj nastavlja na sljedeći API poziv, dok mehanizmi notifikacija događaja Node.js-a pomažu poslužitelju dobiti odgovor na prijašnji poziv,
- Brzina – s obzirom na to da je izgrađen na V8 JavaScript osnovi, Node.js je brz u izvođenju koda,
- Jednodretven ali visoko skalabilan – Node.js koristi model s jednom dretvom uz ponavljanje događaja. Mehanizmi događaja pomažu da poslužitelj odgovori na neblokirajući način i omogućuje visoku skalabilnost u usporedbi s tradicionalnim poslužiteljima koji imaju ograničen broj dretvi za rukovanje zahtjevima,
- Nema međuspremnika – aplikacije bazirane na Node.js-u ne spremaju podatke u međuspremnik, već samo šalju podatke u skupinama. [8]

Na slici 14. dan je prikaz Node.js arhitekture, „*Event Loop*“ omogućuje izvođenje neblokirajućih ulazno-izlaznih operacija tako što operacije prosljeđuje jezgri sustava. Jezgre su višedretvene što znači da mogu obavljati više operacija istovremeno u pozadini. Kada se operacija izvrši „*Worker Threads*“ sistem signalizira Node.js-u da odgovarajući povratni poziv može biti dodan u red za izvršavanje. Kada se Node.js pokreće, inicijalizira se „*Event Loop*“ , pokrene se ulazna skripta, te počine procesuiranje *Event Loopa*. Svaka faza ima FIFO (eng. *First in First Out*) red čekanja povratnog poziva (eng. *callback*) za izvršavanje. Kad je red čekanja istrošen ili je dosegnuta granica, *Event Loop* će nastaviti na sljedeću fazu.



Slika 14. Node.js Arhitektura [9]

3.5 Express

Express je jedan od najpopularnijih Node.js razvojnih okruženja, i služi kao temeljna biblioteka za neka druga popularna Node web razvojna okruženja.
Služi za definiranje načina rukovanja HTTP zahtjeva s različitih URL-ova (ruta). Integriran je s okruženjima za prikazivanje pogleda (eng. *view*) kako bi generirao odgovore umetanjem podataka u predloške. Također omogućuje postavljanje čestih postavki web aplikacija kao što su sučelja koja se koriste za spajanje i lokacija predloška koja se koristi za prikaz odgovora. Bitna karakteristika Expressa je dodavanje „*middlewarea*“ za dodatno procesuiranje zahtjeva u bilo kojem dijelu rukovanja zahtjevom.
Iako je minimalističan, postoje mnogi „*middleware*“ paketi kompatibilni s Expressom koji odgovaraju na razne probleme web razvoja. Postoje biblioteke za kolačiće, sesije, prijavu korisnika, sigurnosna zaglavlja i sl.

3.6 Socket.io

Socket.io je biblioteka koja omogućuje dvosmjernu komunikaciju između klijenta i poslužitelja. Za komunikaciju između klijenta i poslužitelja potrebno je imati Socket.IO poslužitelj i Socket.IO klijent aktivan na pregledniku. Veza se uspostavlja koristeći WebSocket API kad god je to moguće, a u protivnom se koristi HTTP *long polling* kao druga opcija.
Socket.io je razdvojen u dva dijela : Engine.IO i Socket.IO.
Engine.IO je odgovoran za uspostavljanje veze na nižoj razini. On detektira gubitak konekcije, transporte i mehanizme unaprjeđenja. Socket.IO je API više razine koji dodaje funkcionalnosti preko Engine.IO veze. Neke od tih funkcionalnosti su : automatsko ponovno uspostavljanje veze, međuspremnik paketa, odašiljanje svim klijentima ili podskupini klijenta, potvrde (komunikacija zahtjev-odgovor).

3.7 PostgreSQL

PostgreSQL je objektno relacijska baza podataka otvorenog koda koja koristi i proširuje SQL jezik zajedno sa mnogim funkcionalnostima koje sigurno spremaju i skaliraju podatkovne radna opterećenja. Nastao je 1986. godine kao dio POSTGRES projekta na Sveučilištu u Berkeleyu. PostgreSQL je prilagodljiv s funkcionalnostima kao što su definiranje vlastitih tipova podataka,

gradnja prilagođenih funkcija i pisanje koda u drugim programskim jezicima bez potrebe za ponovnim sastavljanjem (Perl,Python,Tcl).

3.8 Redis

Redis (eng. *Remote Dictionary Server*) je brzimemorijski ključ-vrijednost spremnik podatkovnih struktura te je ujedno i otvorenog koda. S obzirom na to da koristi *in-memory* pristup, tj. koristi radnu memoriju te periodično sprema podatke u trajnu memoriju, ovakvim principom rada postiže impresivne brzine dobavljanja i spremanja podataka. Koristi se kao predmemorija i trajni spremnik podataka. Zbog svoje brzine Redis se koristi za upravljanje sesijama, analitiku u stvarnom vremenu, geoprostorne podatke, chat/poruke, prijenos medija i sl.

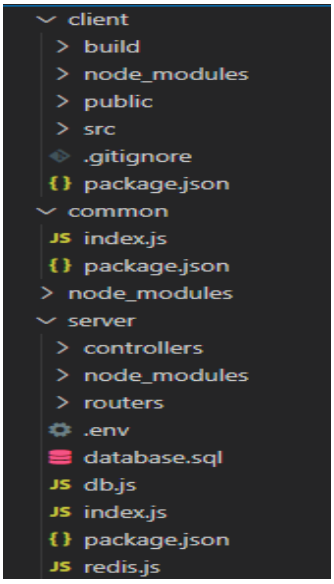
3.9 NPM

NPM (eng. *Node Project Manager*) je mrežni spremnik za objavljivanje Node.js projekta i naredbeni dodatak za interakciju sa spremnicima koji pomaže u instaliranju paketa, kontrolu verzija i upravljanje ovisnostima. NPM je najveći svjetski registar softvera. Sadrži preko 800,000 paketa koda.

Package.json je datoteka koja definira projekte nastale korištenjem NPM-a, ona sadrži sve potrebne informacije o projektu, neke od njih su: ime, verzija, opis, licenca i sl. Osim meta podataka package.json sadrži listu ovisnosti potrebnih za pokretanje same web aplikacije. Ove ovisnosti se instaliraju korištenjem naredbe "*npm install*" ili "*npm i*".

4. Izrada programskog rješenja

Aplikacija je zamišljena kao jednostavna PWA koja omogućuje trenutnu razmjenu poruka između korisnika. Pokretanjem aplikacije korisnik ima mogućnost prijaviti se ili stvoriti novi korisnički račun. Ako korisnik odabere stvaranje novog korisničkog računa otvara se nova forma u kojoj korisnik odabire svoje korisničko ime, zaporku i email adresu. Ako su svi ovi podatci točno uneseni zapisuje se novi korisnički račun u bazu podataka. Nakon prijave korisnik ima mogućnost dodavanja prijatelja na listu prijatelja te slanje poruka istima. Poruke i lista prijatelja također se spremaju u bazu podataka. Sučelje aplikacije treba biti responzivno i odgovarati veličini ekrana kako bi se aplikacija mogla koristiti na svim vrstama uređaja. Struktura aplikacije bit će podijeljena na dva dijela klijent i poslužitelj odnosno klijentski dio i pozadinski dio. Struktura datoteka unutar projekta je prikazana na slici 15. Projekt je podijeljen na tri glavne mape a to su *client, server i common*. *Client* mapa sadrži klijentsku stranu web aplikacije, u njoj je sadržano sve ono što vidi korisnik te logika koja je potrebna za komunikaciju s poslužiteljem. *Server* mapa sadrži pozadinsku logiku aplikacije: pristup bazi podataka, prihvat događaja od klijenta i povezivanje korisnika.
Node-modules mape služe za spremanje Node.js paketa koji su dodatni u projekt, neki od tih paketa su „ioredis“, „socket.io“,“pg“ (PostgreSQL),“express“ i sl.



Slika 15. Struktura datoteka projekta

4.1 Stvaranje baze podataka

U ovom poglavlju opisat će se struktura podataka koji će se spremati unutar aplikacije i baze podataka koje će se koristiti.

Za spremanje podataka koristit će se PostgreSQL i Redis. Redis spremnik podataka radi u radnoj memoriji (RAM) i time ima brzo vrijeme spremanja i dohvata iz spremnika. Redis će se koristiti za spremanje poruka, liste prijatelja i spremanje podataka za uspostavljanje mreže kroz socket.io biblioteku. U bazu podataka PostgreSQL spremat će se podatci o korisničkim računima.

4.1.1 PostgreSQL

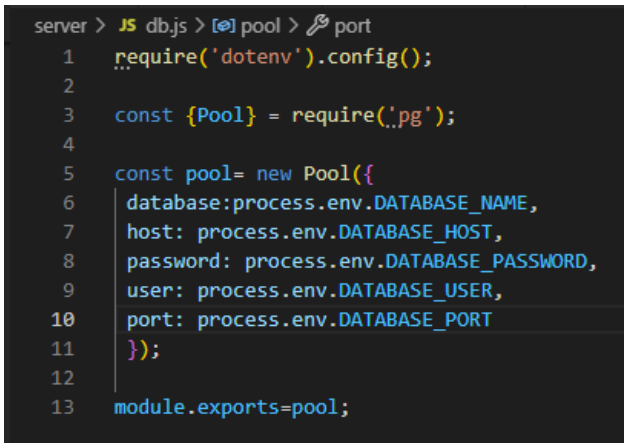
Za inicijalizaciju PostgreSQL baze podataka potrebno je instalirati "pg" paket kroz npm sustav na terminalu.

Nakon toga stvara se nova baza podataka naredbom "*createdb unichat*", te se njome pristupa naerdbom "*psql –d unichat*". Zatim je potrebno definirati samu tablicu unutar baze podataka. Na slici 16 je prikazana naredba "*CREATE TABLE*" za stvaranje tablice *users* s vrijednostima id,username, email, password i userid. Id je glavni ključ tablice a ostale vrijednosti su definirane kao jedinstvene (atribut. *unique*), te ne smiju biti ostavljene prazne (atribut *Not NULL*).

```
unlchat=# CREATE TABLE users (
unlchat(#     id SERIAL PRIMARY KEY,
unlchat(#     username VARCHAR(28) NOT NULL UNIQUE,
unlchat(#     email VARCHAR(28) NOT NULL UNIQUE,
unlchat(#     password VARCHAR NOT NULL,
unlchat(#     userid VARCHAR NOT NULL UNIQUE
unlchat(# );
CREATE TABLE
unlchat=#
```

Slika 16. Stvaranje tablice user

Unutar poslužitelja se definira datoteka db.js koja sadrži prijavu na bazu podataka od strane poslužitelja. Slika 17 pokazuje datoteku db.js, unutar koje se postavlja ime baze podataka, ime poslužitelja, zaporku baze, ime korisnika s kojim se pristupa bazi i sučelje. Ove varijable su definirane unutar .env datoteke radi sigurnosti, to je obična tekst datoteka koja sadrži osjetljive podatke. U .env datoteku se pohranjuju šifre, pristupni ključevi, podatci za prijavu koji ne bi trebali biti dostupni drugim korisnicima. Modul *pool* se izvozi te se koristi za spajanje na bazu podataka unutar poslužitelja.



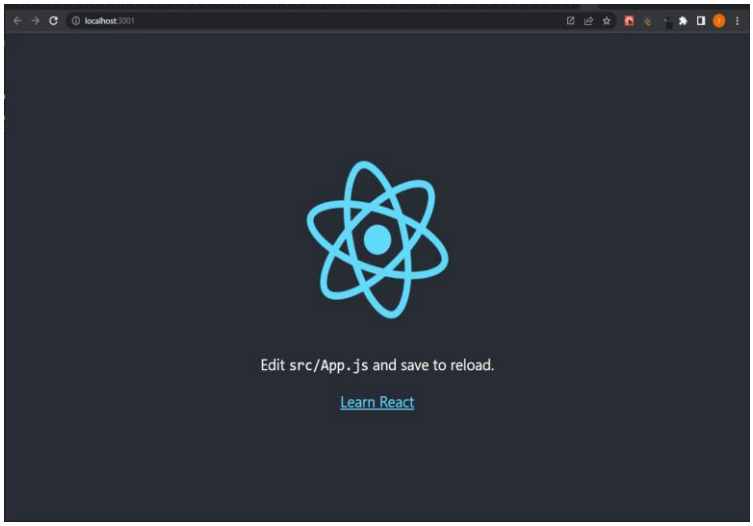
Slika 17. db.js datoteka

4.1.2 Redis

Redis je dostupan jedino na Linux operacijskim sustavima, ali ga je moguće pokretati na Windows sustavima uz WSL2 (eng. *Windows Subsystem for Linux*). WSL2 omogućuje rad Linux binarnih datoteka nativno na Windowsu. Za potrebe ovog projekta korišten je Ubuntu 20.04 LTS. Sljedeći korak je instaliranje Redisa na Linux podsustav, a ta operacija se izvodi pokretanjem naredbe "*sudo apt-get install redis*" nakon koje je Redis uspješno instaliran. Redis spremnik se pokreće koristeći naredbu "*sudo service redis-server start*", nakon koje se može razmjenjivati podatke sa spremnikom. Za spajanje na Redis s poslužitelja potrebno je instalirati "ioredis" paket preko npm-a. U programskom kodu potrebno je stvoriti novi object koji kao ulaznu vrijednost prima sučelje na kojemu je Redis aktivan, jer se uvijek nalazi na istoj IP adresi tj. 127.0.0.1 koja služi za povratnu adresu na računalu.

4.2 Stvaranje početnog predloška aplikacije

Početni predložak aplikacije stvara se korištenjem naredbe:"*npx create-react-app my-app*" koja će stvoriti osnovne datoteke potrebne za pokretanje React aplikacije. Pokretanjem aplikacije koristeći naredbu „*npm start*", pokreće se naredba definirana unutar package.json datoteke unutar „*scripts*" ključa. Naredba „*start*" odgovara pokretanju skripte „*react-scripts start*", koja stvara razvojno okruženje, pokreće poslužitelj za web aplikaciju i omogućuje učitavanje modula.



Slika 18. Početni zaslon create-react-app

Unutar *src* mape projekta nalaze se sve JavaScript i JSX datoteke koje sadrže korisničku stranu aplikacije. Datoteka App.js je glavna komponenta u Reactu i služi kao spremnik za sve ostale komponente. Slika 19 prikazuje sadržaj App.js datoteke, definirana je funkcija *App* koja sadrži poziv na spajanje koristeći uvezenu varijablu *socket* iz socket.js filea te u svojoj povratnoj funkciji vraća komponentu *Views* koja je sadržana u komponenti *UserContext*. Komponente *Views* i *Usercontext* su uvezene iz direktorija *components* koji sadrži sve komponente u klijentskom dijelu aplikacije.

```
client > src > JS App.js > ...
1   import Views from "./components/Views";
2   import UserContext from "./components/AccountContext";
3   import socket from "./socket";
4
5   function App() {
6     socket.connect();
7     return (
8       <UserContext>
9         <Views/>
10        </UserContext>
11        );
12  }
13  export default App;
14
```

Slika 19. App.js datoteka

Usmjeravanje unutar aplikacije je napravljeno koristeći React RouterDOM. React Router DOM je npm paket koji omogućuje implementaciju dinamičkog usmjeravanja u web aplikaciji. Koristi se za izgradnju aplikacija koje imaju više stranica ili komponenti, ali gdje ne dolazi do osvježavanja stranice već se sadržaj dinamično dobavlja temeljem URL-a. Ovakvo

usmjeravanje je definirano unutar Views.jsx datoteke koja sadrži sve moguće stranice koje korisnik može vidjeti unutar web aplikacije.

client > src > components > Views.jsx > ...

```
1  import React from 'react'
2  import { Route,Routes } from 'react-router-dom'
3  import  Login from './Login/Login'
4  import  SignUp from './Login/SignUp'
5  import PrivateRoutes from './PrivateRoutes'
6  import Home from './Home/Home'
7  const Views = () => {
8    return   (
9    <Routes>
10   <Route path="/" element={<Login/>}></Route>
11     <Route path="/register" element={<SignUp/>}></Route>
12     <Route element={<PrivateRoutes/>}>
13     <Route path="/home" element={<Home/>}></Route>
14     </Route>
15     <Route path="*" element={<Login/>}></Route>
16   </Routes>
17 );};
18 export default Views;
```

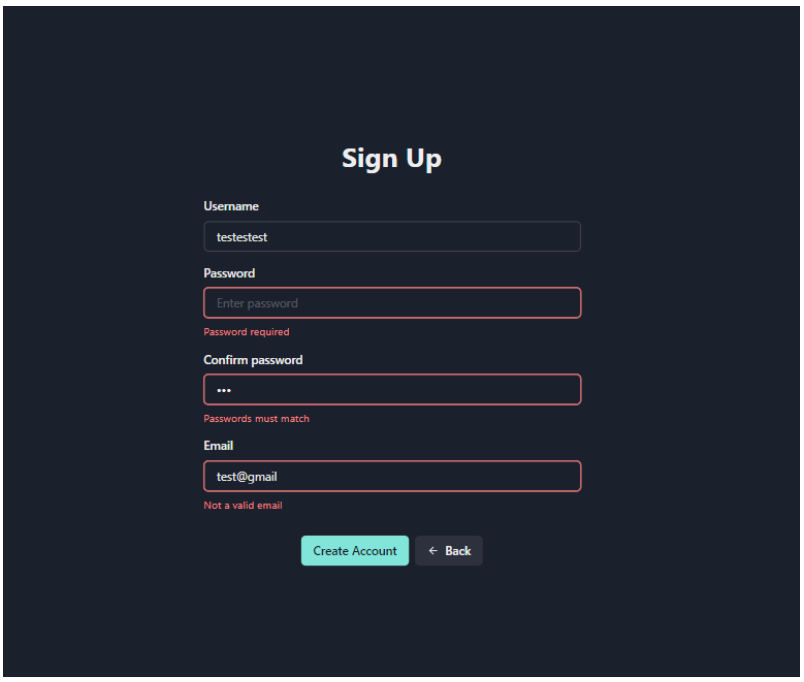
Slika 20. Views.jsx datoteka

Na slici 20. je prikazan sadržaj datoteke Views.jsx, u prvih 6 linija koda su uvezene sve potrebne datoteke koje predstavljaju različite „stranice“ unutar aplikacije: *Login*, *Signup* i *Home*. *Home* komponenta je postavljena unutar *PrivateRoutes* elementa koji služi za autentikaciju korisnika. U slučaju da korisnik koji nije prijavljen pokuša pristupiti *Home* komponenti putem poveznice „/home“ komponenta *PrivateRoutes* će ga preusmjeriti na Login komponentu. Osim njih uvoze se funkcije i komponente potrebne za logiku usmjeravanja i prijenos određenih podataka kroz komponente (*AccountContext*). Komponente unutar Reacta se definiraju putem funkcija (linija 7) te se izvoze na dnu datoteke kao i u datoteci Views.jsx koristeći „*export default View*“ (linija 18). Unutar povratne vrijednosti funkcije su definirani svi smjerovi tj. stranice te njihovi URL-ovi koje korisnik može posjetiti. Svaki smjer sastoji se od *path* varijable i element varijable. *Path* varijabla određuje URL same stranice, dok element varijable određuje koja komponenta će se učitati tijekom otvaranja tog URL-a. Ako se korisnik spoji na domena.com/register otvorit će mu se *SignUp* komponenta.

4.3 Forma za registraciju

Forma za registraciju, odnosno *SignUp* komponenta je realizirana koristeći Formik za kontrolu forme, te ChakraUI za pojedinačne elemente. *SignUp* komponenta je prikazana na slici broj 21. i sastoji se od 4 polja za unos i dva gumba. Polja za unos su *Username, Password, Confirm*

password i Email, dok gumbovi služe za povratak na prijašnju komponentu (*Back*) ili slanje obrasca (*Create Account*).



Slika 21 SignUp komponenta

Elementi unutar same forme su definirani unutar *VStack* elementa koji je deklariran kao forma koristeći ChakraUI „as“ varijablu kojom se elementi mogu definirati kao HTML elementi te zadržati karakteristike oba dvije vrste elementa. Primjer:

<Vstack as={Form}/>

Na slici 22. je prikazan primjer *TextField* elementa kojima su definirani pojedinačni ulazi te definicije dva gumba od kojih prvi služi za slanje obrasca, a drugi gumb sadrži *OnClick* funkciju koja pokreće funkciju *Navigate* te korisnika vraća na početnu stranicu aplikacije.

```
82          <TextField
83            name="email"
84            placeholder="Enter email"
85            autoComplete="off"
86            label="Email"
87          />
88
89          <ButtonGroup pt="1rem">
90            <Button colorScheme="teal" type="submit">
91              Create Account
92            </Button>
93            <Button onClick={() => navigate("/")} leftIcon={<ArrowBackIcon />}>
94              Back
95            </Button>
96          </ButtonGroup>
```

Slika 22. Primjer tekst polja i gumbova unutar SignUp.sjx datoteke

Na slici 23 je prikazana sama kontrola i obrada forme za registriranje novog korisnika koja se izvodi korištenjem Formik komponente. Unesene vrijednosti se spremaju u varijablu *Values* i prosljeđuju na poslužitelj koristeći API poziv „http://localhost:4000/auth/signup“, u slučaju da vrijednosti nisu ispravne prikazuje se greška na istom unosu. *ValidationSchema* varijabla označava skup pravila koja se koriste tijekom provjeravanja korisnikova unosa, kako bi se spriječilo unošenje krivih znakova, neupisivanje valjane adrese elektroničke pošte provjera duljine unosa i sl. Ova pravila su definirana unutar *common* mape projekta, te su jedina komponenta koju koriste i klijent i poslužitelj. Pravila su definirana koristeći Yup u datoteci index.js. Na slici 24 je prikazana datoteka common/index.js unutar koje su definirana pravila za unos u formu definirana unutar *formSchema* varijable. Definirano je da varijable *username* i *password* moraju biti upisane, moraju biti duže od 6 i kraći od 28 znakova. Adresa elektroničke pošte mora biti u pravilnom formatu i unosi za zaporke moraju biti jednaki. Uz svako pravilo dodane su poruke greške koje opisuju grešku npr. „*Username too short*, „*email required*“ i sl. Ako su svi ulazi točno popunjeni moguće je poslati API zahtjev prema poslužitelju. S obzirom na to da su podatci koji se šalju povjerljivi koristi se *POST* metoda. U zaglavlju zahtjeva definira se da je sadržaj zahtjeva u JSON obliku, te se samo tijelo zahtjeva nadopunjava s vrijednostima pretvorenim u JSON format. Unutar *fetch* zahtjeva definirana je varijabla „*credentials:include*“ koja označava da se kolačići, autorizacijska zaglavlja ili TLS klijent certifikati uključuju u zahtjev. U slučaju da nije došlo do greške od strane klijenta ili poslužitelja u odgovoru te je korisnik uspješno spremljen u bazu podataka, poslužitelj kao odgovor šalje status *loggedIn* i *username* korisnika.

```
const SignUp = () => {
  const { setUser } = useContext(AccountContext);
  const [error, setError] = useState(null);
  const navigate = useNavigate();
  return (
    <Formik
      initialValues={{ username: "", password: "" ,passwordConfirm:"",email:""}}
      validationSchema={formSchema}
      onSubmit={(values, actions) => {
        const vals = { ...values };
        actions.resetForm();
        fetch("http://localhost:4000/auth/signup", {
          method: "POST",
          credentials: "include",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify(vals),
        })
          .catch(err => {
            return;
          })
          .then(res => {
            if (!res || !res.ok || res.status >= 400) {
              return;
            }
            return res.json();
          })
          .then(data => {
            if (!data) return;
            setUser({ ...data });
            if (data.status) {
              setError(data.status);
            } else if (data.loggedIn) {
              navigate("/home");
            }
          });
      }}
```

Slika 23. Kontrola SignUp Forme

```
common > JS index.js > ...
1   const Yup = require("yup");
2
3   const formSchema = Yup.object({
4       username: Yup.string().required("Username required").min(6,"Username too short").max(28,"Username too long"),
5       password: Yup.string().required("Password required").min(6,"Password too short").max(28,"Password too long"),
6       email: Yup.string().email('Not a valid email'),
7       passwordConfirm: Yup.string().oneOf([Yup.ref('password'), null], 'Passwords must match'),
8
9   });
10  const friendSchema = Yup.object({
11      friendName:Yup.string().required("Username required").min(6,"Username too short").max(28,"Username too long"),
12  })
13  module.exports={ formSchema,friendSchema };
14
```

Slika 24. Pravila za unos Pravila za ulazne elemente

Na strani poslužitelja koristeći *express* usmjeravanje se sluša na *POST* zahtjev na „*http://localhost:4000/auth/signup*“ adresu te se prije pokušaja stvaranja novog zapisa unutar baze podataka izvršava još jedna validacija ista kao ona na klijent strani aplikacije koristeći *validateForm()* funkciju iz razloga što API zahtjev ne moraju nužno biti poslani iz sučelja aplikacije. Ako zahtjev nije poslan preko sučelja aplikacije podatci neće biti validirani. Također

je implementirano ograničenje na broj zahtjeva da bi se spriječili napadi preopterećivanjem poslužitelja. Slika 25 prikazuje authRouter.js datoteku koja se pokreće kao middleware unutar expressa te sadrži usmjernik koji ovisno o primljenom zahtjevu izvršava odgovarajuće funkcije. Linija broj 11 prikazuje odgovor usmjernika na „*/signup*“ poziv, nakon kojeg se izvršava *validateForm* funkcija, *rateLimiter* koji kao ulazne varijable prima vrijeme (u ovom slučaju 30 sekundi) i dozvoljeni broj pokušaja (u ovom slučaju 4) unutar tog vremena.

Slika 25. authRouter.js

AttemptSignup funkcija prikazana na slici 26 je asinkrona funkcija koja unutar zahtjeva prima unos iz forme za registraciju, a kao odgovor podatke o korisniku. Funkcija prvo šalje upit na bazu kojim se provjerava da li korisnik s tim korisničkim imenom već postoji, u slučaju da postoji funkcija vraća status „*loggedIn:false*“ i poruku „*Username taken*“. Kod slučaja kada *korisničko ime* nije zauzeto zaporka se kriptira koristeći bcrypt. Bcrypt je biblioteka za Node.JS koja služi za kriptiranje teksta baziran na Blowfish šifri. Zatim se šalje novi upit u bazu kojim se stvara novi zapis u bazi koristeći *INSERT* SQL naredbu. Na mjesto *usernamea* se postavlja korisničko ime iz zahtjeva, za *password* se postavlja šifrirana zaporka upisana od strane korisnika, *userid* se generira koristeći funkciju *uuidv4* koja je uvezena na vrhu datoteke i adresa elektroničke pošte je također upisan od strane korisnika i predan preko zahtjeva. Nakon stvaranja novog zapisa u bazi podataka funkcija šalje odgovor web aplikaciji (klijentu), postavlja sesiju i direktno ga preusmjerava na *Home* komponentu web stranice.

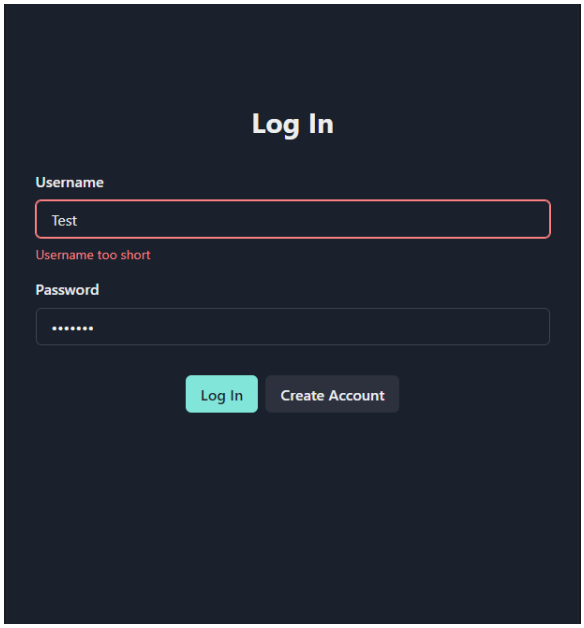
```
module.exports.attemptSignup = async (req,res) => {
    const existingUser=await pool.query("SELECT username from users WHERE username=$1",
    [req.body.username]
    );
    if (existingUser.rowCount === 0){
        const hashedPass=await bcrypt.hash(req.body.password,10);
;
        const newUserQuery= await pool.query("INSERT INTO users(username,password,userid,email) values($1,$2,$3,$4) RETURNING id,username,userid",
        [req.body.username,hashedPass,uuidv4(),req.body.email]);
        req.session.user = {
            username: req.body.username,
            id: newUserQuery.rows[0].id,
            userid: newUserQuery.rows[0].userid
        }
        res.json({loggedIn:true,username:req.body.username});
    }else{
        res.json({loggedIn:false, status:"Username taken"})
    }

}
```

Slika 26. attemptSignUp funkcija

4.4 Prijava u aplikaciju

Na slici 27 je prikazan izgled korisničkog sučelja za prijavljivanje u aplikaciju, ova komponenta je izvedena na sličan način *SignUp* komponenti. Sadrži istu validaciju unosa i raspored elemenata.



Slika 27. Login ekran

Slika 2. prikazuje logiku forme za prijavu od strane klijenta. Procedura je identična onoj kod *SignUp* komponente, upisane vrijednosti se provjeravaju koristeći prije definiranu shemu(slika 22), kod pritiska na gumb obrazac se šalje. Koristi se drukčiji API zahtjev, ovog puta sa „*/login*“ nastavkom, koristeći *POST* metodu. Podatci se pretvaraju u JSON format te se stavljaju u tijelo

zahtjeva. U slučaju grešaka tijekom API zahtjeva ili na strani klijenta izlazi se iz petlje. Nakon slanja obrasca unesene vrijednosti se brišu koristeći funkciju *actions.resetForm()*.

```
const Login = () => {
  const {error}=useState(null);
  const{setUser}=useContext(AccountContext);
  const navigate = useNavigate();
  return (
    <Formik
      initialValues={{ username: "", password: "" }}
     validationSchema={formSchema}
      onSubmit={(values, actions) => {
        const vals = {...values};
        actions.resetForm();
        fetch("http://localhost:4000/auth/login", {
          method: "POST",
          credentials:"include",
          headers:{
            "Content-Type":"application/json",
          },
          body:JSON.stringify(vals),

        }).catch(err => {
          return;
        }).then(res => {
          if (!res || !res.ok || res.status >= 400){
            return;
          }
          return res.json();
        }).then(data => {
          if (!data) return;
          console.log(data);
          setUser({...data});
          navigate("/home");
        });
        }}
    >
```

Slika 28. Login komponenta

API zahtjev se obrađuje na poslužitelju unutar authRouter.js datoteke(slika 25), koja prvo definira poziv *GET* koji se poziva s klijent strane aplikacije kada korisnik otvori početnu stranicu aplikacije. Ovaj poziv služi za preusmjeravanje korisnika koji imaju spremljenu sesiju tj. od prije su prijavljeni u aplikaciju. *POST* metoda se obrađuje tako da se radi validacije podataka forme, provjera učestalosti slanja zahtjeva koristeći *rateLimiter* funkciju koja je ograničava broj unosa na 5 puta unutar 30 sekundi. Na slici 29 je prikazana funkcija *attemptLogin*, asinkrona funkcija koja prima zahtjev i kao povratnu informaciju šalje odgovor. Potencijalna prijava se provjerava *SELECT* upitom kojim se traži zapis unutar baze podataka koji ima isto korisničko ime kao onaj poslan u tijelu zahtjeva. Ako postoji zapis s tim korisničkim imenom uspoređuju se uneseni password i kriptirani password iz baze podataka koristeći *bcrypt.compare()* funkciju. Stvara se nova sesija s varijablama *username,id* i *userid,* te se šalje odgovor sa statusom *loggedIn* i istim vrijednostima.

Slika 29. Obrada prijave na poslužitelju

4.5 Sesije

HTTP je konekcijski protokol bez stanja, ne može razlikovati između veza različitih korisnika. Kako bi HTTP mogao razlikovati korisnike te povezivati različite HTTP zahtjeve korisnički podatci se spremaju između zahtjeva. Zbog toga se koriste sesije, sesije predstavljaju vrijeme koje korisnik provodi koristeći web stranicu/aplikaciju. U praksi je nemoguće znati kada je korisnik prestao koristiti stranicu, pa se na sesije stavlja vrijeme isteka nakon kojeg sesija više nije valjana i potrebno je stvoriti novu. Sesije se mogu spremati unutar kolačića, u URL-u kao dio upita ili u bazi podataka. U ovoj web aplikaciji sesije će se spremati unutar kolačića pod ključem „*sid*“, *express-session* paket ima mogućnost povezivanja spremanja kolačića na poslužitelju unutar Redis spremnika. Sesije će unutar web aplikacije pratiti status korisnika te u slučaju da je već prijavljen na web aplikaciju, neće se morati ponovno prijavljivati.

Slika 30 prikazuje kako poslužitelj kontrolira kolačiće s podatcima o korisniku. Funkcija *session* sadrži nekoliko varijabli:

- *Secret* – označava tajnu zaporku pod kojom se sesija šifrira,
- *Credentials* – može li se zahtjev izvršiti koristeći kolačiće, TLS certifikate ili autorizacijska zaglavlja,
- *Store* – spremanje podataka sesije, inicijalna vrijednost je *MemoryStore*, koji namjerno nije dizajniran za produkcijsko okruženje. Ima problem s curenjem memorije i ne skalira

se dalje od jednog procesa, služi samo za otklanjanje grešaka i razvoj. Iz tih razloga za spremanje sesijskih podataka upotrebljava se Redis spremnik,

- *Resave* – Prisiljava sesiju da se sprema nazad u sesijski spremnik, onda i kada nije bila promijenjena tijekom zahtjeva,
- *saveUnintialized* – prosljeđuje sesiju koja nije inicijalizirana da se spremi u spremnik. Sesija nije inicijalizirana kada je nova ali nije modificirana,
- *Cookie: secure* – dozvoljavanje sigurnosnih kolačića, zahtjeva HTTPS vezu,
- *Cookie: httpOnly* – sprječava klijentskim skriptama pristup podatcima. Pristup kolačićima je dozvoljen samo poslužitelju,
- *Cookie: expire* – postavlja vrijeme u kojemu kolačić ističe ,
- *Cookie : sameSite* – ovisno o postavljenoj vrijednosti sprječava ili omogućuje slanje kolačića na druge domene. Moguće postavke su: *Lax*,*Strict* i *None*. Postavljen je *if* upit ako je okruženje postavljeno na „*production*“ *sameSite* će biti postavljen na „*none*“ u protivnom „*lax*“.

```
const sessionMiddleware = session({
    secret:process.env.COOKIE_SECRET,
    credentials:true,
    name:"sid",
    store: new RedisStore ({client: redisClient}),
    resave:true,
    saveUninitialized:false,
    cookie:{
        secure:process.env.NODE_ENVIRONMENT === "production",
        httpOnly: true,
        expires:1000*60*60*24*7,
        sameSite: process.env.NODE_ENVIRONMENT === "production" ? "none" : "lax",
    }
})
const corsConfig = {
    origin: 'http://localhost:4000',
        credentials:true,
        methods: ["GET", "POST"],
        transports: ['websocket', 'polling'],
}
const wrap = (expressMiddleware) => (socket,next) => expressMiddleware(socket.request, {}, next);
module.exports={sessionMiddleware,wrap,corsConfig};
```

Slika 30. Sesije na poslužitelju

AccountContext komponenta prikazana na slici 31 se koristi za provjeravanje statusa korisnika unutar cijele aplikacije. Unutar App.js datoteke *Views* komponenta je sadržana unutar komponente *UserContext* koja je izvedena iz ove datoteke. Unutar standardne React aplikacije, podatci se šalju od vrha prema dnu kroz *propse*, ali takav način dijeljenja podataka ne odgovara svim slučajevima korištenja. Kontekst nudi način za dijeljenje vrijednosti između komponenti bez potrebe da se šalje iz jednog nivoa na drugi. Kontekst se koristi kada je potrebno dijeliti

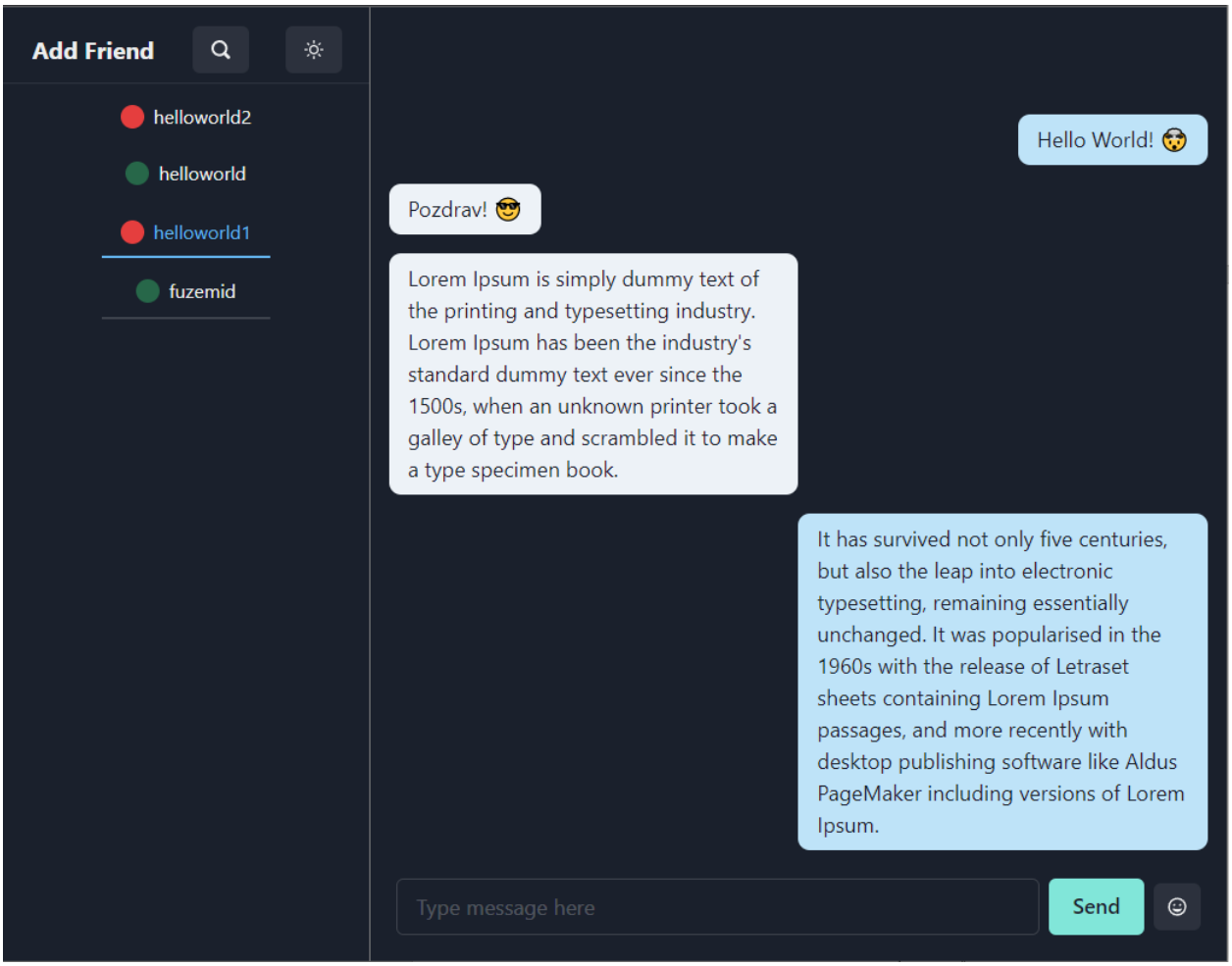
podatke koji su „globalni“ za neku aplikaciju kao što su autentificirani korisnik, teme ili jezik. *CreateContext* stvara *Context* objekt. Kada React prikaže komponentu koja je povezana na ovaj *context* objekt pročitat će trenutnu *context* vrijednost od najbližeg *Providera* na višem nivou unutar stabla.

Komponenta *UserContext* provjerava da li je korisnik prijavljen koristeći kolačiće i API poziv „*/auth/login*“. Razlika između ovog poziva i onog koji se koristi prilikom korištenja *Login* komponente je što *Login* komponenta korist *POST* metodu, dok ova komponenta koristi *GET*. Ovdje se koristi *GET* metoda jer nije potrebno slati nikakav sadržaj unutar tijela zahtjeva, već se autentikacija vrši preko kolačića i sesije. Ovaj API poziv se izvršava unutar *useEffect hooka*, on dozvoljava izvedbu popratnih efekata unutar komponente. Primjeri popratnih efekata su : dohvaćanje podataka, ažuriranje DOM-a i brojači. Ovisno o izvedbi *useEffect hooka* moguće je aktivirati ga tijekom svakog prikazivanja, samo na prvom prikazivanju ili tijekom mijenjanja ovisnosti. Ovisnosti se definiraju na dnu funkcije unutar uglatih zagrada, ako su zagrade prazne *hook* se aktivira samo na prvom prikazivanju. Ovakav pristup najviše odgovara potrebama ove komponente jer je potrebno provjeravati je li korisnik prijavljen samo kod inicijalnog učitavanja stranice. Postavljene su provjere koje u slučaju greške postavljaju korisnikov *loggedIn* status na *false*. U slučaju da je korisnik prijavljen i poslužitelj vrati odgovor korisnika se preusmjerava na *Home* komponentu.

Slika 31. AccountContext komponenta

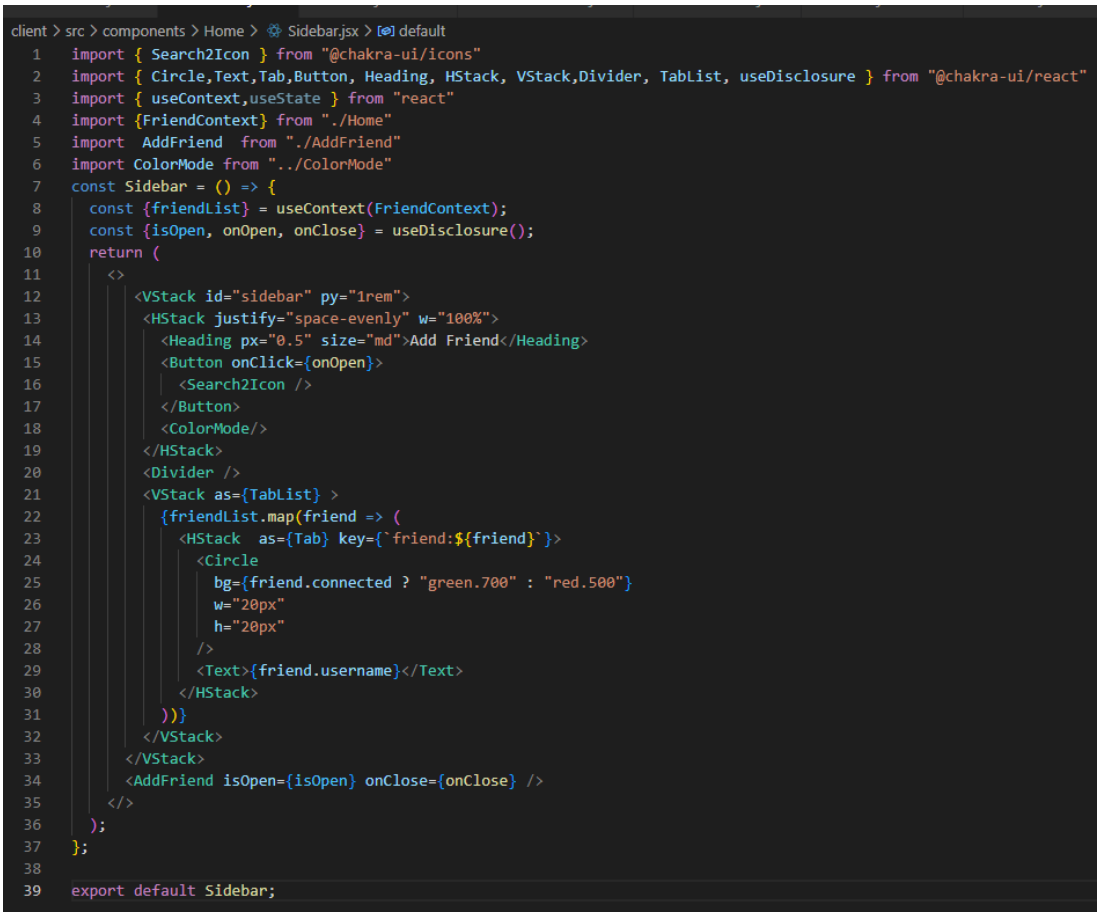
4.6 Glavni ekran aplikacije

Home komponenta (slika 32) služi kao glavni ekran aplikacije i sadrži listu prijatelja, gumb i modal za dodavanje prijatelja, tipku za način osvjetljenja, sučelje za trenutno slanje poruka i izbornik za dodavanje emotikona. Zbog modularnosti Reacta funkcije su podijeljene u zasebne komponente. *Home* komponenta u sebi sadrži *Drawer* komponentu koja služi kod prikaza aplikacije na manjim uređajima, *Sidebar* komponentu i *Chat* komponentu.



Slika 32. Home komponenta

Na slici 33 prikazana je *Sidebar* komponenta, ona sadrži sve elemente koji se nalaze s lijeve strane sučelja. *FriendContext* se uvozi iz *Home* komponente i služi za dobavljanje liste prijatelja sa socket.io paketom. *UseDisclosure* je prilagođen *hook* iz ChakraUI biblioteke, on olakšava kontroliranje modala za dodavanje prijatelja. Na vrhu *Sidebara* postavljen je gumb sa *onClick* događajem za otvaranje *AddFriend* modala i gumb za promjenu načina osvjetljenja. Lista prijatelja je postavljena kao *TabList* komponenta te se u nju *mapiraju* pojedini prijatelji. Pokraj imena prijatelja postavljen je krug koji je crvene boje u slučaju da je prijatelj na mreži i zelene u slučaju da nije. Kod implementacije modala postavljaju se vrijednosti



Slika 33. Sidebar.jsx datoteka

AddFriend komponenta definira modal koji se otvara pritiskom na gumb unutar *Sidebara*. Modal se sastoji od jednostavne forme s upisom imena prijatelja i tipkom za slanje. Forma je implementirana koristeći Formik komponentu kao i u prijašnjim formama. Korištena je validacijska shema koja provjerava je li ime prijatelja upisano, je li veće od 6 i manje od 28 znakova. Za zahtjev se koristi „*socket.emit(„add friend")*" funkcija koja radi tako da se s klijent strane emitira događaj („*add friend*") te na poslužiteljskoj strani aplikacije se postavlja *listener* za isti taj događaj, te nakon primanja događaja izvršava određeni kod. Funkcija prima *friendName* kao ulazni podataka, a vraća *errorMsg,done* i *newFriend* vrijednosti. Ako je uspješno dodan novi prijatelj modal se zatvara, a listi se dodaje novi prijatelj. Na slici 34 i 35 je prikazan kod *AddFriend* komponente i izgled modala unutar aplikacije.

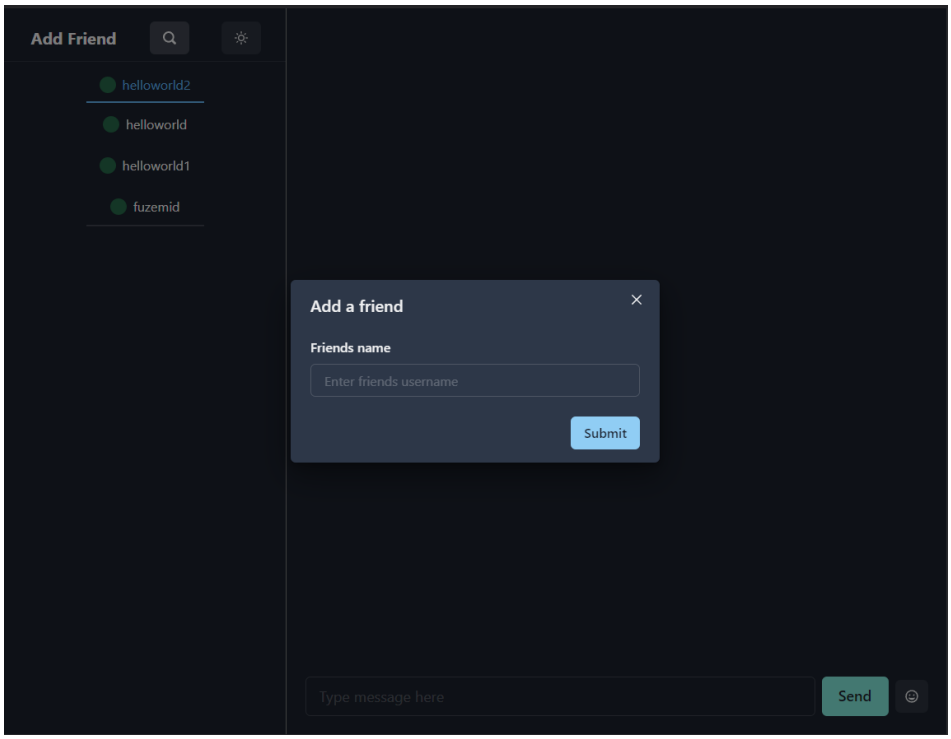
client > src > components > Home > AddFriend.jsx > ...

```jsx
import {useState,useCallback,useContext} from "react";
import { FriendContext } from "./Home";

const AddFriend = ({isOpen,onClose}) => {
  const [error,setError] = useState("");
  const closeModal = useCallback(
    () => {
        setError("");
        onClose();
    },
    [onClose],
  )
  const {setFriendList} = useContext(FriendContext);
  return (<Modal isOpen={isOpen} onClose={closeModal} isCentered>
        <ModalOverlay/>
        <ModalContent>
            <ModalHeader>Add a friend</ModalHeader>
            <ModalCloseButton></ModalCloseButton>
            <Formik initialValues={{friendName: ""}} onSubmit={(values,actions) => {
                socket.emit("add_friend", values.friendName,({errorMsg, done, newFriend}) => {
                    if (done){
                        setFriendList(c => [newFriend,...c])
                        closeModal();
                        return;
                    }
                    setError(errorMsg);
                });
            } }
            validationSchema={friendSchema}
            >
                <Form>
            <ModalBody>
                <Heading as="p"
                fontSize="xl"
                color="red.500"
                textAlign="center">{error}</Heading>
                <TextField label="Friends name" placeholder="Enter friends username" autoComplete="off" name="friendName"/>
            </ModalBody>
            <ModalFooter><Button colorScheme='blue'   type="submit">
              Submit
            </Button></ModalFooter>
            </Form>
            </Formik>
        </ModalContent>
    </Modal>
    );
}

export default AddFriend;
```

Slika 34. Datoteka AddFriend.jsx



Slika 35 AddFriend komponenta

Za povezivanje klijenta i poslužitelja koristeći socket.io potrebno je imati socket.io klijent na klijentskom dijelu aplikacije i socket.io poslužitelj na poslužiteljskom dijelu. Socket.io klijent je definiran u datoteci socket.js, a na njega se aplikacija spaja u datoteci App.js (slika 19). Na slici 35 je prikazana index.js datoteka unutar poslužitelja kojom se inicijalizira poslužitelj s definiranim *CORS* postavkama spremljenim u serverController.js datoteci(slika 30). CORS (emg. *Cross-Origin Resource Sharing*) je mehanizam baziran na HTTP zaglavlju koji omogućuje poslužitelju da postavi bilo koje izvore (domenu, shemu ili sučelje), osim njega samog, kojima bi preglednik trebao dopustiti učitavanje njegovih resursa. CORS se također oslanja na mehanizme po kojima preglednici šalju zahtjev kojim provjeravaju dopušta li poslužitelj stvarni zahtjev. CORS je bitan jer se unutar razvoja ne koristi HTTPS i CORS ne dozvoljava komunikaciju sa serverom ako IP adresa klijenta nije definirana unutar varijable *origins*. "*Middleware* na poslužitelju se pokreće koristeći *app.use()* funkciju, to su dodatne funkcionalnosti koje se koriste tijekom razmjene podataka između klijenta i poslužitelja. Korištene middleware komponente su: *Helmet* koji postavlja sigurnosna zaglavlja, cors, *express*, *sessionmiddleware* (slika 30) i *authRouter*(slika 25). Koristeći *wrap* funkciju ugrađuje se *sessionMiddleware* unutar socket.io poslužitelja, bez ove funkcije socket.io ne bi mogao dohvatiti podatke o sesiji korisnika.

server > JS index.js > ...

```
1  const cors = require("cors");
2  const helmet = require("helmet");
3  const { Server } = require("socket.io");
4  const express = require("express");
5  const app = express();
6  const server = require("http").createServer(app);
7  const authRouter = require ("./routers/authRouter");
8  const { sessionMiddleware,wrap,corsConfig } = require("./controllers/serverController");
9  const { authorizeUser, initializeUser, addFriend, onDisconnect, dm } = require("./controllers/socketController");
10 require("dotenv").config();
11 const io = new Server(server, {
12     cors: corsConfig,
13 });
14 app.use(helmet());
15 app.use(cors(corsConfig));
16 app.use(express.json());
17 app.use(sessionMiddleware)
18 app.use("/auth",authRouter);
19 io.use(wrap(sessionMiddleware));
20 io.use(authorizeUser);
21 io.on("connection",socket => {
22     initializeUser(socket);
23     socket.on("add_friend", (friendName,cb) =>{addFriend(socket,friendName, cb)});
24     socket.on("error",function(err){
25         console.log(err);
26       });
27       socket.on("dm", (message) => dm(socket,message))
28     socket.on("disconnecting", () => onDisconnect(socket));
29 });
30 server.listen(4000,() => {
31     console.log("Server listening on port 4000");
32 });
```

Slika 36 index.js poslužitelja

AuthorizeUser.js prikazan na slici 37 provjerava da li korisnik ima otvorenu sesiju na poslužitelju te da li je postavljen korisnik unutar sesije.

Slika 37 authorizeUser.js

Nakon autorizacije, korisnik se inicijalizira unutar datoteke initializeUser.js (Slika 38). Inicijaliziranje podrazumijeva zapisivanje korisnika u spremnik, ispisivanje liste prijatelja korisnika i ispisivanje poruka korisnika. Koristeći *redisclient.hset* naredbu u Redis se upisuje novi zapis ili ažurira stari. Prva vrijednost koju *hset* prima je ključ zapisa. Kao ključ se postavlja username dobiven iz sesije, ostale vrijednost upisuju se prema shemi ključ-vrijednost. Vrijednosti *userid* i status *connected se spremaju u Redis spremnik.*

Lista prijatelja se dobavlja *lrange* funkcijom, *lrange* funkcija vraća listu elemenata zapisanih pod zadanim ključem. *ParseFriendList* parsira danu listu prijatelja, ova funkcija je potrebna jer se podatci iz Redisa izvoze kao jedan tekstualni niz s pojedinim vrijednostima razdvojenim sa znakom točke. U slučaju da korisnik ima prijatelja, njima se prenosi „*connected*“ događaj koristeći *socket.to* funkciju. *Socket.to* prenosi događaj samo onim korisnicima koji su naznačeni u ulaznoj varijabli.

Poruke se učitavaju na sličan način kao i lista prijatelja. Pretražuju se po ključu (*userid*), te se parsiraju u objekt s 3 vrijednosti : *to*,*from* i *content*. Ove vrijednosti se na klijentskoj strani aplikacije koriste za ispisivanje poruka unutar Chat komponente. Poruke se prenose koristeći *socket.emit* s nazivom događaja „*messages*“.

Slika 38 initializeUser.js

Dodavanje novih prijatelja je implementirano kroz addFriend.js datoteku (slika 39). Dodavanje prijatelja započinje na klijent strani web aplikacije Nakon što korisnik uspješno pošalje obrazac, na poslužitelju se aktivira događaj *socket.on(„add_friend“)*(slika 35). Funkcija prima *socket*, *friendName* te kao odgovor na zahtjev šalje povratni poziv (eng. *callback*). Provjerava se postoji li prijatelj u spremniku i da li je već dodan na listu prijatelja korisnika. Radi se upit na spremnik koji se provjerava postoji li zapis s *userid*-em koji odgovara imenu prijatelja i upit koji vraća cijelu listu prijatelja korisnika. Prva *if* petlja provjerava je li prijatelj pronađen u spremniku tj. je li registriran kao korisnik, dok druga provjerava postoji li prijatelj unutar liste prijatelja korisnika. U slučaju da su svi uvjeti ispunjeni korištenjem *lpush* naredbe dodaje se novi zapis u spremnik, te se stvara objekt *newFriend* koji sadrži *username*, *userid* i status *connected* korisnika. *NewFriend* objekt se povratnim pozivom šalje na korisnički dio aplikacije zajedno

sa potvrdnom varijablom *done* koja je postavljena na *true*. Unutar *AddFriend* modala se nadopunjava lista prijatelja te se modal zatvara.

server > controllers > socketio > JS addFriend.js > addFriend

```
1   const redisClient = require("../../redis");
2
3   const addFriend = async (socket, friendName, cb) => {
4     if (friendName === socket.user.username) {
5       cb({ done: false, errorMsg: "Cannot add self!" });
6       return;
7     }
8     const friend = await redisClient.hgetall(`userid:${friendName}`);
9     const currentFriendList = await redisClient.lrange(
10      `friends:${socket.user.username}`,
11      0,
12      -1
13    );
14    if (!friend.userid) {
15      cb({ done: false, errorMsg: "User doesn't exist!" });
16      return;
17    }
18    if (currentFriendList && currentFriendList.indexOf(friendName) !== -1) {
19      cb({ done: false, errorMsg: "Friend already added!" });
20      return;
21    }
22
23    await redisClient.lpush(
24      `friends:${socket.user.username}`,
25      [friendName, friend.userid].join(".")
26    );
27
28    const newFriend = {
29      username: friendName,
30      userid: friend.userid,
31      connected: friend.connected,
32    };
33    cb({ done: true, newFriend });
34  };
35
36  module.exports = addFriend;
```

Slika 39 addFriend.js

Slika 39 prikazuje useSocketSetup.jsx, a to je komponenta unutar korisničkog dijela aplikacije koja je odgovorna za prihvaćanje emitiranih događaja s poslužitelja. Ova komponenta se uvozi u *Home* komponentu te kao ulazne vrijednosti prima dvije funkcije: *setFriendsList* i *setMessages*. Implementirana su 4 *listenera*:

- *Socket.on(„friends“)* - događaj služi za postavljanje liste prijatelja, on se poziva od strane poslužitelja tijekom inicijaliziranja korisnika,
- *Socket.on(„message“)* – događaj se poziva tijekom inicijaliziranja korisnika i služi za dohvaćanje poruka iz spremnika,

- *Socket.on(„dm“)* – događaj se emitira tijekom slanja poruka s jednog korisnika na drugog,
- *Socket.on(„connected“)* – događaj se emitira tijekom spajanja i odspajanja s poslužitelja, služi za postavljanje statusa korisnika unutar aplikacije.

Kada je aplikacija zatvorena potrebno je odspojiti *listenere* za ovaj događaj jer se oni izvode samo onda kad je korisnik na mreži, korištenjem *socket.off* funkcije na svim prije definiranim događajima oni su isključeni kada korisnik nije u aplikaciji.

client > src > components > Home > useSocketSetup.jsx > default

```jsx
import { useContext, useEffect } from "react";
import socket from "../../socket";
import { AccountContext } from "../AccountContext";

const useSocketSetup = (setFriendList, setMessages) => {
  const { setUser } = useContext(AccountContext);
  useEffect(() => {
    socket.connect();
    socket.on("friends", friendList => {
      setFriendList(friendList);
    });
    socket.on("messages", messages => {
      setMessages(messages);
    });
    socket.on("dm", message => {
      setMessages(prevMsgs => [message, ...prevMsgs]);
    });
    socket.on("connected", (status, username) => {
      setFriendList(prevFriends => {
        return [...prevFriends].map(friend => {
          if (friend.username === username) {
            friend.connected = status;
          }
          return friend;
        });
      });
    });
    socket.on("connect_error", () => {
      console.log("Error")
    });
    return () => {
      socket.off("connect_error");
      socket.off("connected");
      socket.off("friends");
      socket.off("messages");
      socket.off("dm");
    };
  }, [setUser, setFriendList, setMessages]);
};

export default useSocketSetup;
```

Slika 40 useSocketSetup.jsx

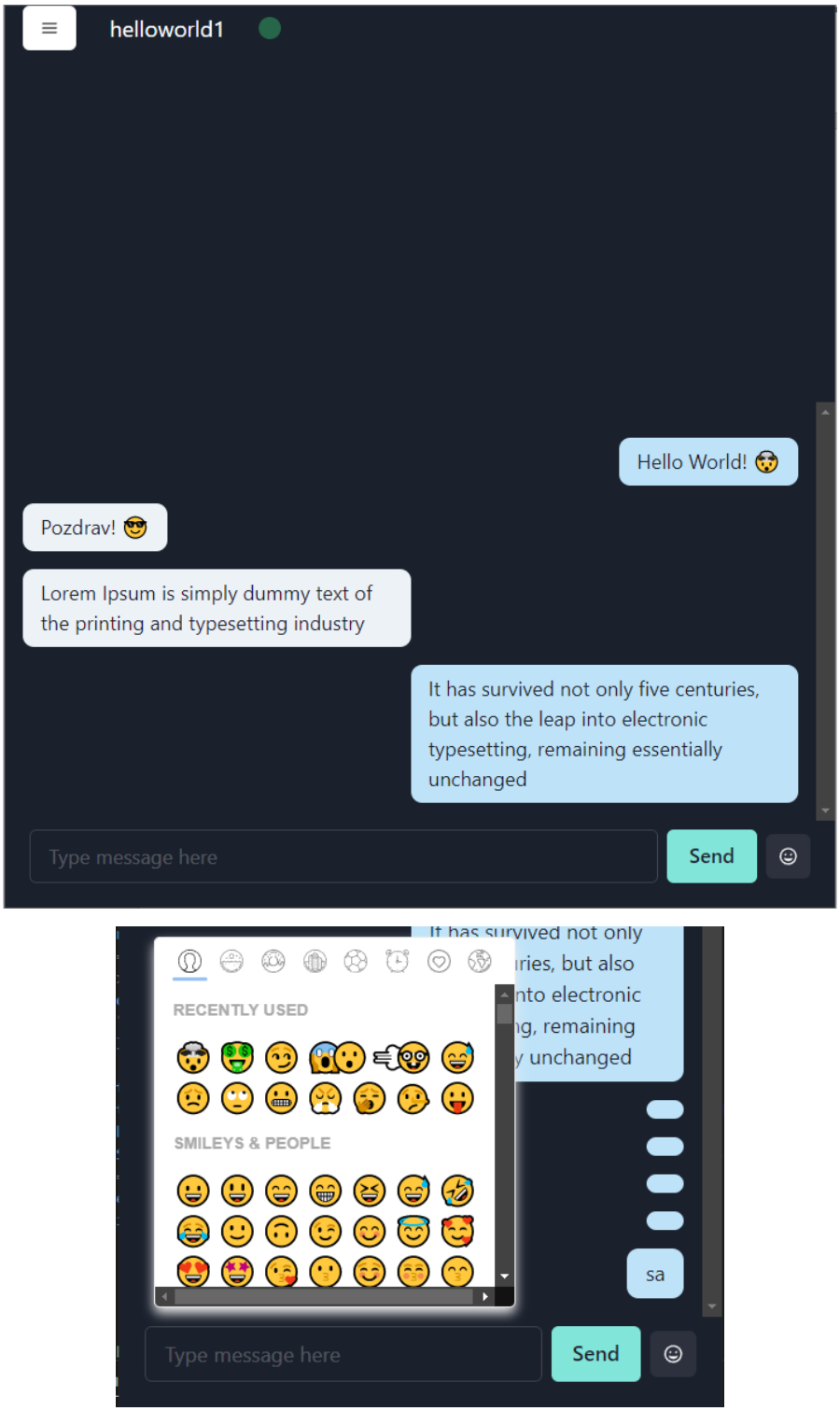
```jsx
<Grid
  templateColumns={{base:"repeat(10, 1fr)"}}
  h="100vh"
  as={Tabs}
  onChange={index => setFriendIndex(index)}
>
  <Show breakpoint='(min-width: 900px)'>
  <GridItem colSpan="3" borderRight="1px solid gray">
    <Sidebar />
  </GridItem>
  </Show>
  <GridItem  colSpan={{md:"10",sm:"10","xl":"7",lg:"7",}} maxH="100vh">
  <Show breakpoint='(max-width: 900px)'>
  <Container position={"fixed"} maxW='900px' bg='#1a202c' color='#262626'>
  <HStack spacing='30px'>

  <Button bg="white"  onClick={onOpen}>
      <HamburgerIcon />
    </Button>

    <Text fontSize='20px' color="white">{friendList[friendIndex]?.username}</Text> <Circle
    bg={friendList[friendIndex]?.connected ? "green.700" : "red.500"}
    w="20px"
    h="20px"
    />
    </HStack>
     </Container>
     </Show>
    <MessagesContext.Provider value={{ messages, setMessages }}>
      <Chat userid={friendList[friendIndex]?.userid} />
    </MessagesContext.Provider>
  </GridItem>
</Grid>
```

Slika 41. Raspored komponenti u Home.jsx

Sučelje *Home* komponente (slika 41) je definirano koristeći *Grid* komponentu. Ekran je podijeljen u omjeru *chat* prostor 70% i *sidebar* prostor 30%. Problem kod ovakvog rasporeda sučelja je što nije prigodno za mobilne uređaje, zbog manje veličine ekrana stvara loše vizualno iskustvo za korisnika. Zbog toga je komponenta *Sidebar* postavljena unutar Show komponente s *breakpoint* varijablom s vrijednošću (*min-width:900px*). *Breakpoint* varijabla postavlja u ovom slučaju minimalnu širinu na kojoj će se prikazivati *Sidebar* na 900 piksela. Jednako tako je postavljena komponenta koja se prikazuje ispod 900 piksela s gumbom za otvaranje *Drawer* komponente kroz *onClick listener*. *Drawer* komponenta će služiti kao zamjena za *Sidebar*. Radi definiranog *Grid* rasporeda potrebno je promijeniti širinu koju zauzima chat komponenta. Postavljanjem *colSpan* vrijednosti, kao objekt koji sadrži različite vrijednosti za različite veličine ekrana, ona će kada se sakrije *SideBar* sa show komponentom, zauzeti 100% širine ekrana.



Slika 42. Izgled Home komponente za mobilne uređaje

Chat komponenta je sastavljena od *ChatBox* i *Chat* komponente. *Chatbox* komponenta je donji dio ekrana koji sadrži ulaz za upisivanje poruka, gumb za slanje i meni za biranje emotikona. Izgled *Chatbox* komponente je opisan na slici 43. Elementi se postavljeni u *Hstack* komponentu koja je definirana kao Forma. Input polju se postavlja vrijednost jednaka varijabli „*input*“ te je

dodan *onChange* događaj koji dodaje tekst unutar samog elementa s funkcijom *setInput*.Varijabla „*input*“ se koristi za dodavanje emotikona kroz *onEmojiClick* funkciju.Bez value polja nije moguće izvan Input polja utjecati na vrijednost elementa. Izbornik za emotikone koji se koristi je „*emoji-picker-react*“ instaliran kroz npm. Za izbornik je definirana funkcija *onEmojiClick* koja u input unosi odabrani emotikon nakon što je on kliknut. Kada se forma podnosi odnosno kada korisnik pošalje poruku ona se postavlja u objekt u obliku koji je postavljen i na poslužitelju s 3 varijable(*to*, *from* i *content*). Nakon pripremanja poruke ona se emitira na socket.io preko događaja „*dm*“. Nakon slanja poruka se postavlja na vrh liste prijašnjih poruka i *Input* se ispražnjava. U slučaju da je *Input* prazan prilikom slanja, odmah se izlazi iz funkcije s *return* komandom.

```
const Chatbox = ({userid}) => {
  const [showPicker, setShowPicker] = useState(false);
  const [input, setInput] = useState("");
  const onEmojiClick = (event, emojiObject) => {
   setInput(input+emojiObject.emoji);
  };
const {setMessages} = useContext(MessagesContext)

return (<Formik  initialValues={{ message: "" }}
validationSchema={Yup.object({
  message: Yup.string().min(1).max(255),

  })}
  onSubmit={() => {
    if(input==""){return;};
    const message = {to: userid, from: null, content:input};
    socket.emit("dm", message);
    setMessages(prevMsgs => [message, ...prevMsgs ])
    console.log(JSON.stringify(message));
    setInput("");
  }}
    >
    <HStack as={Form} w="100%" pb="1.4rem" px="1.4rem">
        <Input as={Field} id="txt1" name="message"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        type="text"
        placeholder="Type message here"
        size="lg"
        autoComplete="off"
        />
        <Button type="submit" size="lg" colorScheme="teal">Send</Button>
        <IconButton aria-label="Emoji" icon={<FaRegGrin/>} onClick={() => setShowPicker(val => !val)} />
        {showPicker && <Picker
          pickerStyle={{ width: '60%' ,position:"absolute",height:"20rem","marginTop":"-25rem"}}
          native={true}
          disableSearchBar={true}
          onEmojiClick={onEmojiClick} />}
    </HStack>
    </Formik>
  );
};
```

Slika 43. ChatBox komponenta

Na slici 44 poslužitelja poruka se zaprima preko dm.js datoteke. Unutar datoteke se poruka priprema u oblik pogodan Redisu izvršavanjem *join* funkcije nad varijablama primljenim od klijenta. Kada se podatci pripreme, spremaju se u Redis sa *lpush* funkcijom. Poruke se spremaju

dva puta jednom za pošiljatelja, drugi put za primatelja. Poslužitelj emitira događaj „*dm*“ na primatelja te mu šalje sadržaj poruke. Na klijentu se emitirani događaj na *useSocketSetup* datoteci(slika 39) postavlja na vrh prijašnjih poruka s funkcijom *setMessages*.

server > controllers > socketio > JS dm.js > ...

```
1   const redisClient = require("../../redis");
2
3   const dm = async (socket, message) => {
4       console.log(message);
5     message.from = socket.user.userid;
6     // to.from.content
7     const messageString = [message.to, message.from, message.content].join(
8       "."
9     );
10
11    await redisClient.lpush(`chat:${message.to}`, messageString);
12    await redisClient.lpush(`chat:${message.from}`, messageString);
13
14    socket.to(message.to).emit("dm", message);
15  };
16
17  module.exports = dm;
```

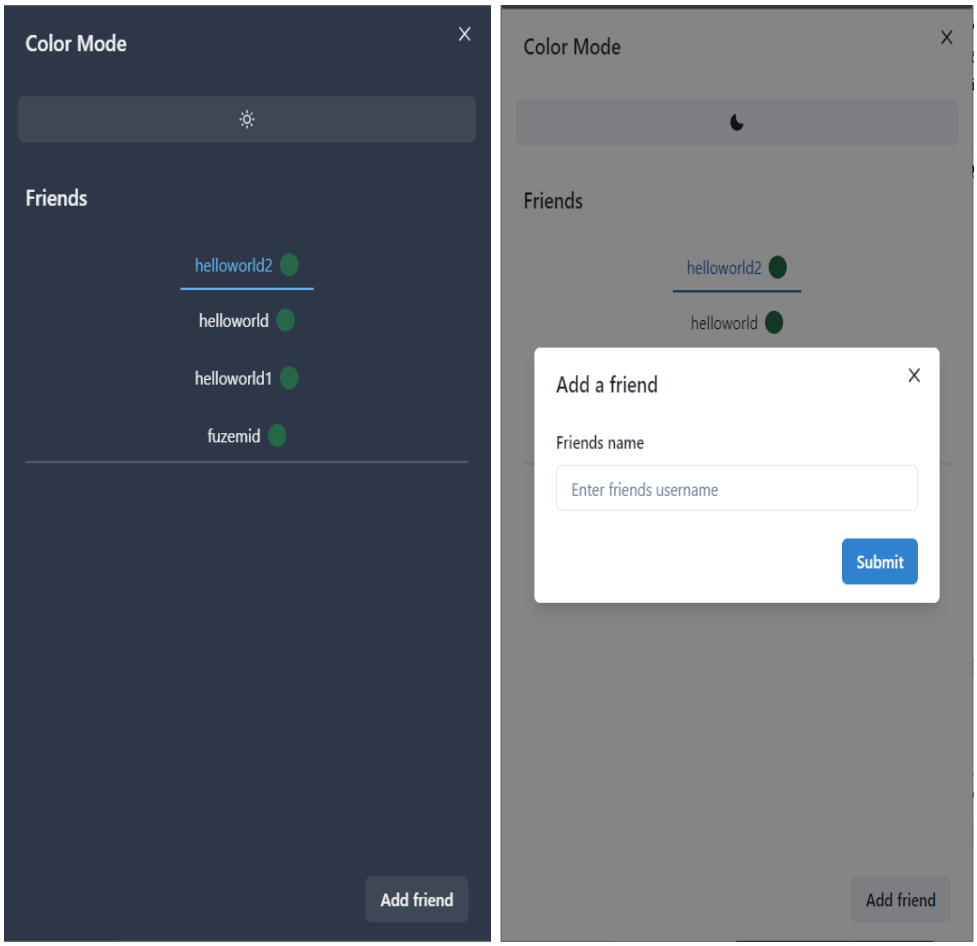
Slika 44. dm.js

Poruke se prikazuju unutar *Chat* komponente(Slika 45) koja je postavljena kao *VStack* koristeći *Tabpanele* za okvire poruka . Poruke se dodaju filtriranjem sveukupnih poruka spremljenih u *MessagesContext*, te se prikazuju samo poruke kojima je primatelj ili pošiljatelj prijatelj koji je trenutno odabran kroz listu prijatelja na *Sidebaru* ili *Draweru*. Ovisno o tome koje pošiljatelj poruke margine se postavljaju za lijevu ili desnu stranu (lijevo za poslane, desno za primljene poruke).

```jsx
const Chat = ({userid}) => {
const { friendList } = useContext(FriendContext);
const { messages } = useContext(MessagesContext);
const bottomDiv = useRef(null);
useEffect(() => {
  bottomDiv.current?.scrollIntoView();
}, );
  return friendList.length >0 ? (
    <VStack h="100%" justify="end">
        <TabPanels overflowY="scroll">
          {friendList.map( friend => (
            <VStack
            flexDir="column-reverse"
            as={TabPanel}
            key={`chat:${friend.userName}`}
            w="100%">
              <div ref={bottomDiv} />
              {messages.filter(msg => msg.to === friend.userid || msg.from === friend.userid)
              .map((message,idx) => (
                <Text m={ message.to === friend.userid ? "1rem 0 0 auto !important" : "1rem auto 0 0 !important" }
                maxW="50%"
                key={`msg:${friend.userName}.${idx}`}
                fontSize="lg"
                bg={ message.to === friend.userid
                  ? "blue.100" : "gray.100"}
                color="gray.800"
                borderRadius="10px"
                p="0.5rem 1rem"> {message.content} </Text>
              ))}
            </VStack>
          ))}
        </TabPanels>
        <Chatbox userid={userid}/>
    </VStack>
```

Slika 45. Chat.jsx komponenta

Slika 46 prikazuje *Drawer* komponentu korisničkog sučelja, ona služi kao zamjena za *SideBar* koja ne odgovara manjim ekranima (mobilnim uređajima) . Mobilne aplikacije često koriste dodatne menije kao što je ovaj kako bi se maksimalno iskoristio prostor na ekranu te kako bi se korisniku pružili bolje iskustvo korištenja aplikacije. Komponente korištene unutar *Drawera* su jednake komponentama unutar *SideBara*. Na vrhu *Drawera* je *Color Mode* opcija za biranje između svijetlog ili tamnog načina rada. Ispod nje se nalazi lista prijatelja a na dnu ekrana je gumb za dodavanje prijatelja koji otvara *AddFriend* modal jednako kao i kod *SideBar* komponente.



Slika 46. Drawer komponenta

4.7 PWA

Kako bi aplikacija zadovoljavala uvjete da postane progresivna web aplikacija mora sadržavati manifest.json ,datoteku ,*service workera* i druge mogućnosti web platforme u kombinaciji s progresivnim poboljšanjima koja korisnicima daju iskustvo slično nativnim aplikacijama. Progresivna poboljšanja su filozofija dizajna koja pruža osnovu nužnog sadržaja i funkcionalnosti što većem broju korisnika dok u isto vrijeme pružajući najbolje iskustvo korisnicima na modernim pretraživačima koji mogu koristiti potpuni kod. Posebna pažnja treba se obratiti na pristupačnost unutar aplikacija. Prihvatljive alternative trebale bi biti dostupne gdje je to moguće.

Na slici 47 prikazana je manifest.json datoteka, ona sadrži osnovne podatke o aplikaciji. *Web application manifest* je nužna datoteka za mogućnost dodavanja aplikacije na početni zaslon i prezentiranja iste na sličan način kao i nativnu aplikaciju. Manifesti progresivnih web aplikacija sadrže podatke o boji teme, pozadine, imenu aplikacije, opisu, ikone koje se koriste i sl.

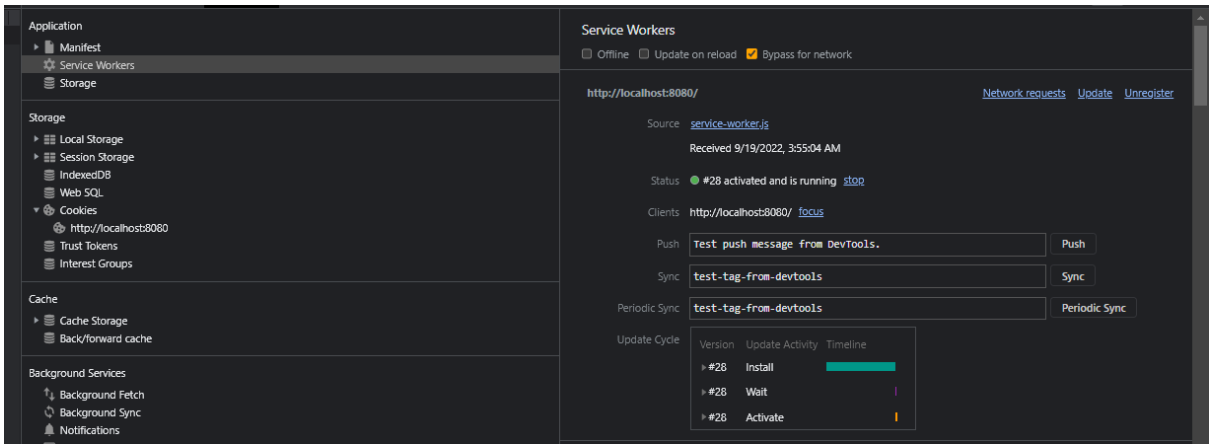
Slika 47. Manifest.json

Service workeri se definiraju korištenjem .js datoteka, potrebno ih je prijaviti unutar web aplikacije. *Service worker* se prijavljuje unutar index.js datoteke na klijentskoj strani aplikacije.(Slika 48). *Service worker* je pogonjen događajima što znači da tijekom rada prati zahtjeve na aplikaciji te može biti postavljen tako da na njih odgovara. *Service worker* u ovom projektu spremat će statične resurse kao što su slike i paketi.

```
client > src > JS index.js > ...
1   import React from 'react';
2   import ReactDOM from 'react-dom/client';
3   import {BrowserRouter} from 'react-router-dom';
4   import App from './App';
5   import * as serviceWorkerRegistration from './serviceWorkerRegistration';
6   import theme from "./theme";
7   import { ColorModeScript } from '@chakra-ui/react';
8   import { ChakraProvider } from '@chakra-ui/react';
9
10  const root = ReactDOM.createRoot(document.getElementById('root'));
11  root.render(
12    <React.StrictMode>
13      <BrowserRouter>
14      <ChakraProvider theme={theme}>
15        <ColorModeScript initialColorMode={theme.config.initialColorMode}/>
16        <App />
17      </ChakraProvider>
18      </BrowserRouter>
19
20    </React.StrictMode>
21  );
22  serviceWorkerRegistration.register("./service-worker.js");
23
```

Slika 48. index.js datoteka klijent strane

Ako je *service worker* prijavljen, on se pojavljuje unutar *Inspect* elementa, *Application* potkategorije(slika 49).



Slika 49. Service worker na pregledniku

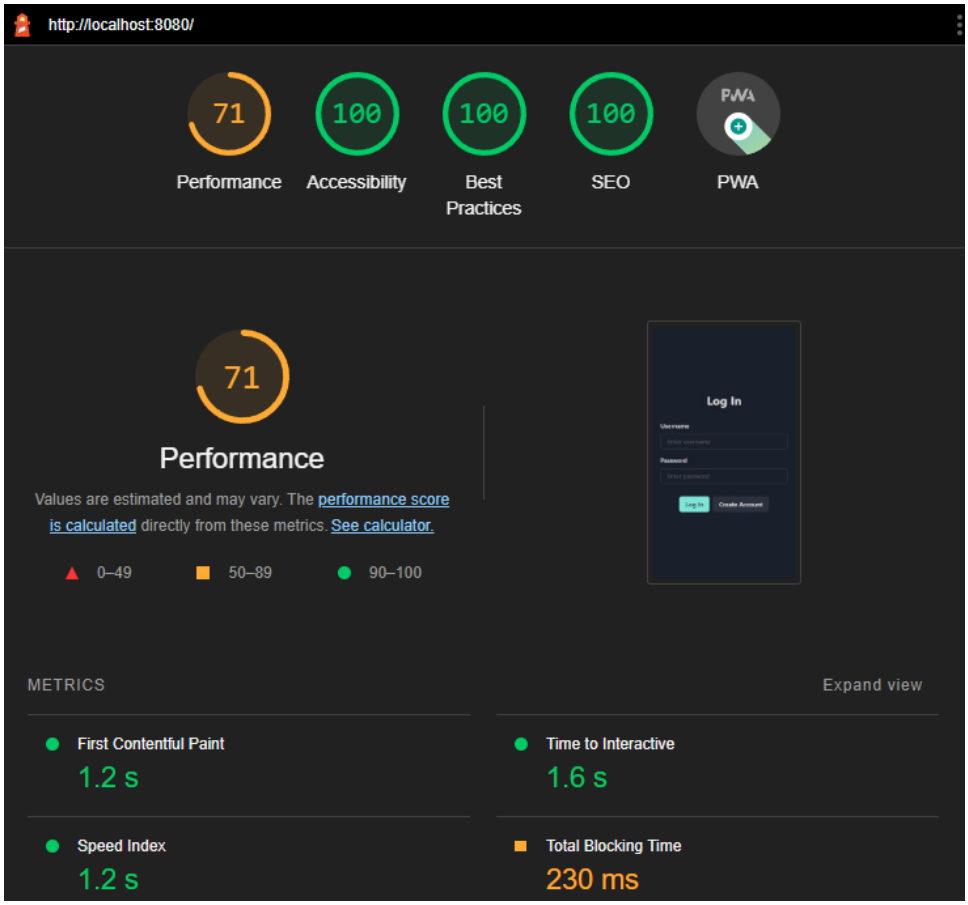
Unutar aplikacije *service worker* će spremati podatke u predmemoriju koristeći *Cache-First* metodu predmemoriranja. Na slici 50 je prikazan kod service-worker.js datoteke. *Listener* na „*Install*“ događaj predmemorira resurse spremljene u listu *PRECACHE_URLS*. „*Activate*“ *listener* briše stare podatke koji se zamjenjuju novima. *Fetch listener* pruža odgovore za resurse s istim podrijetlom. Ako nema odgovora, odgovor se predmemorira s mreže prije nego što se vraća na aplikaciju.

```
48  const PRECACHE = 'precache-v1';
49  const RUNTIME = 'runtime';
50
51
52  const PRECACHE_URLS = [
53    'index.js',
54    'App.js',
55    'public/icon-192x192.png',
56    'public/icon-256x256.png',
57    'public/icon-384x384.png',
58    'public/icon-512x512.png',
59    'demo.js'
60  ];
61  self.addEventListener('install', event => {
62    event.waitUntil(
63      caches.open(PRECACHE)
64        .then(cache => cache.addAll(PRECACHE_URLS))
65        .then(self.skipWaiting())
66    );
67  });
68  self.addEventListener('activate', event => {
69    const currentCaches = [PRECACHE, RUNTIME];
70    event.waitUntil(
71      caches.keys().then(cacheNames => {
72        return cacheNames.filter(cacheName => !currentCaches.includes(cacheName));
73      }).then(cachesToDelete => {
74        return Promise.all(cachesToDelete.map(cacheToDelete => {
75          return caches.delete(cacheToDelete);
76        }));
77      }).then(() => self.clients.claim())
78    );
79  });
80  self.addEventListener('fetch', event => {
81    if (event.request.url.startsWith(self.location.origin)) {
82      event.respondWith(
83        caches.match(event.request).then(cachedResponse => {
84          if (cachedResponse) {
85            return cachedResponse;
86          }
87          return caches.open(RUNTIME).then(cache => {
88            return fetch(event.request).then(response => {
89              return cache.put(event.request, response.clone()).then(() => {
90                return response;
91              });
92            });
93          });
94        })
95      );
```

Slika 50. service-worker.js

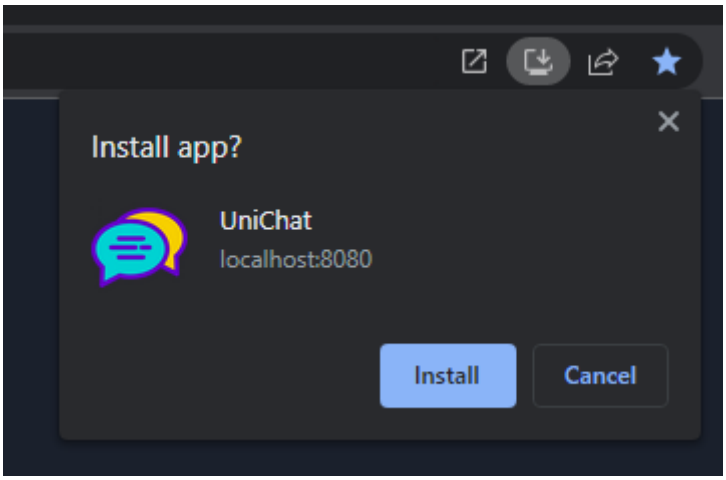
5. Lighthouse analiza

Za evaluaciju PWA karakteristika UniChat aplikacije koristit će se Lighthouse alat. Lighthouse ne analizira sve aspekte aplikacije i ne daje potpunu sliku o aplikaciji, ali je koristan za pregled osnovnih funkcionalnosti i karakteristika aplikacije koja se analizira. Aplikacija se ocjenjuje u 5 kategorija a one su: Performance ,Accessibility ,Best Practices, SEO I PWA. Rezultati Lighthouse analize prikazani su na slici 51. Aplikacija je ostvarila rezultat 71 u Performance kategoriji, zbog viška neiskorištenog JavaScript koda. Za rješavanje ovog problema potrebno je detaljnije pregledati instalirane module unutar samog npm-a. U svim drugim kategorijama aplikacija je dobila ocjenu 100, što pokazuje da je napravljena po standardima Googla i potrebnima za PWA.

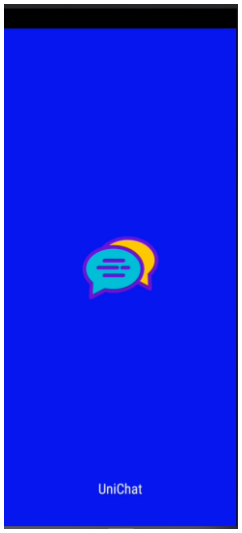


Slika 51. Lighthouse analiza aplikacije

Na slici 52 je prikazan iskočni ekran na pregledniku koji upućuje na instaliranje aplikacije kao PWA. Slika 53 prikazuje „početni ekran“ kod pokretanja UniChat aplikacije na mobilnom uređaju.



Slika 52. PWA instalacija



Slika 53. Splash screen

6. ZAKLJUČAK

Progresivne web tehnologije kao relativno nova tehnologija pokazale su se korisnima u stvaranju web aplikacija koje se svojim vizualnim identitetom sve više približavaju nativnim aplikacijama. Iako još nisu usporedive s nativnim aplikacijama, progresivne web aplikacije nude alternative koje omogućuju stvaranje kompaktnih i brzih web aplikacija korištenjem poznatih web tehnologija kao što su React ,Node.js ,Express i sl. Principi koji se koriste za izradu progresivnih web aplikacija su efikasni u svom cilju poboljšavanju performansi aplikacija kao i sveukupnog korisničkog iskustva. Progresivne web aplikacije mogu biti dobra alternative i za postojeće web aplikacije koje se žele probiti na mobilno tržište. Za sada web aplikacije nemaju jednake mogućnosti kao nativne aplikacije zbog razine pristupa hardveru i softveru mobilnih uređaja, ali s obzirom na to da postoji toliko interesa za ovakvu vrstu aplikacija od strane kompanija kao i korisnika, velika su šanse da će progresivne web aplikacije postati sve raširenije i u skoro vrijeme ostvariti veću integraciju s mobilnim uređajima.

Kroz razvoj aplikacije za trenutno slanje poruka u ovom radu su prikazane mogućnosti razvoja progresivnih web aplikacija koristeći moderne tehnologije za web razvoj. Razvoj aplikacije nije predstavljao veće probleme zbog mogućnosti i fleksibilnosti React i NodeJS biblioteka. Završna progresivna web aplikacija sadrži sve potrebne osnovne funkcionalnosti za rad aplikacije za trenutno slanje poruka. Moguće je u budućnosti aplikaciju proširiti s dodatnim funkcionalnostima koje bi upotpunile korisničko iskustvo, neke od takvih funkcionalnosti bi bile: korisnički profili, korisničke slike ili avatari, grupni razgovori, video pozivi, slanje datoteka i sl.

7. LITERATURA

[1] S. &. J. A. Tandel, »Impact of Progressive Web Apps on Web App Development,« *International Journal of Innovative Research in Science, Engineering and Technology,* svez. 7, br. 9, p. 9440, 2018.

[2] »Medium,« 2017. [Mrežno]. Dostupno: https://medium.com/level-up-web/a-quick-intro-into-progressive-web-app-7c9de2391a2d. [Pokušaj pristupa 24. kolovoz 2022].

[3] »Chrome Developers,« 24. rujan 2021.. [Mrežno]. Dostupno: https://developer.chrome.com/docs/workbox/caching-strategies-overview/. [Pokušaj pristupa 4. rujan 2022.].

[4] A. Osmani, »Medium,« 29. studeni 2017.. [Mrežno]. Dostupno: https://medium.com/dev-channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154. [Pokušaj pristupa 3. rujan 2022.].

[5] N. Gallagher, »Twitter Blog,« 6. travanj 2017.. [Mrežno]. Dostupno: https://blog.twitter.com/engineering/en_us/topics/open-source/2017/how-we-built-twitter-lite. [Pokušaj pristupa 3. rujan 2022.].

[6] S. Gobbo, »web.dev,« 23. veljača 2018.. [Mrežno]. Dostupno: https://web.dev/asda-george/. [Pokušaj pristupa 3. rujan 2022.].

[7] C. Deshpande, »Simplilearn,« 18. srpanj 2022.. [Mrežno]. Dostupno: https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs. [Pokušaj pristupa 5. rujan 2022.].

[8] »tutorialspoint,« [Mrežno]. Dostupno: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm. [Pokušaj pristupa 22. kolovoz 2022.].

[9] A. Rashevskaya, »Litslink,« 19. srpanj 2021.. [Mrežno]. Dostupno: https://litslink.com/blog/node-js-architecture-from-a-to-z. [Pokušaj pristupa 25. kolovoz 2022.].

[10] A. Karuppannan, »https://www.w2ssolutions.com/,« 14. srpanj 2021.. [Mrežno]. Dostupno: https://www.w2ssolutions.com/blog/pwa/. [Pokušaj pristupa 1. rujan 2022.].

8. PRILOZI

8.1 Popis tablica

8.2 Popis slika

IZJAVA

Izjavljujem pod punom moralnom odgovornošću da sam diplomski rad izradio samostalno, isključivo znanjem stečenim na studijima Sveučilišta u Dubrovniku, služeći se navedenim izvorima podataka i uz stručno vodstvo mentora prof.dr.sc Vedrana Batoša i komentora Ivana Grbavca, dipl. ing., kojima se još jednom srdačno zahvaljujem.

Ivica Ćurčija