Web trgovina izrađena pomoću React-a

Jakšić, Matej

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni diplomski studij računarstva

WEB TRGOVINA IZRAĐENA POMOĆU REACT-A

Diplomski rad

Matej Jakšić

Osijek, 2020.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 17.09.2020.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	Matej Jakšić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-986R, 19.09.2019.
OIB studenta:	83450410856
Mentor:	Izv. prof. dr. sc. Ivica Lukić
Sumentor:	
Sumentor iz tvrtke:	Vlatko Vlahek
Predsjednik Povjerenstva:	Doc.dr.sc. Mirko Köhler
Član Povjerenstva 1:	Izv. prof. dr. sc. Ivica Lukić
Član Povjerenstva 2:	Doc.dr.sc. Zdravko Krpić
Naslov diplomskog rada:	Web trgovina izrađena pomoću React-a
Znanstvena grana rada:	**Informacijski sustavi (zn. polje računarstvo)**
Zadatak diplomskog rada:	Izraditi Web trgovinu pomoću React-a koja će omogućiti administratorsko sućelje za upravljanje trgovinom, sučelje za registraciju korisnika i dodavanje proizvoda u košaricu. Na kraju izraditi izvještaj o svim kupovinama u Web trgovini i obaviti testiranje. Tema rezervirana za studenta: Matej Jakšić
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	17.09.2020.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 30.09.2020.

Ime i prezime studenta:	Matej Jakšić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-986R, 19.09.2019.
Turnitin podudaranje [%]:	6

Ovom izjavom izjavljujem da je rad pod nazivom**: Web trgovina izrađena pomoću React-a**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Ivica Lukić

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA

o odobrenju za pohranu i objavu ocjenskog rada

kojom ja __Matej Jakšić_____, OIB: __83450410856_____, student/ica Fakulteta elektrotehnike, računarstva i informacijskih tehnologija Osijek na studiju _Diplomski sveučilišni studij Računarstvo_, kao autor/ica ocjenskog rada pod naslovom: _Web trgovina izrađena pomoću React-a**,**

dajem odobrenje da se, bez naknade, trajno pohrani moj ocjenski rad u javno dostupnom digitalnom repozitoriju ustanove Fakulteta elektrotehnike, računarstva i informacijskih tehnologija Osijek i Sveučilišta te u javnoj internetskoj bazi radova Nacionalne i sveučilišne knjižnice u Zagrebu, sukladno obvezi iz odredbe članka 83. stavka 11. *Zakona o znanstvenoj djelatnosti i visokom obrazovanju* (NN 123/03, 198/03, 105/04, 174/04, 02/07, 46/07, 45/09, 63/11, 94/13, 139/13, 101/14, 60/15).

Potvrđujem da je za pohranu dostavljena završna verzija obranjenog i dovršenog ocjenskog rada. Ovom izjavom, kao autor/ica ocjenskog rada dajem odobrenje i da se moj ocjenski rad, bez naknade, trajno javno objavi i besplatno učini dostupnim:

a) široj javnosti

b) studentima/icama i djelatnicima/ama ustanove

c) široj javnosti, ali nakon proteka 6 / 12 / 24 mjeseci (zaokružite odgovarajući broj mjeseci).

**U slučaju potrebe dodatnog ograničavanja pristupa Vašem ocjenskom radu, podnosi se obrazloženi zahtjev nadležnom tijelu Ustanove.*

__Osijek, 30.09.2020._____ _____

(mjesto i datum) (vlastoručni potpis studenta/ice)

SADRŽAJ

1. UVOD

Tema ovog diplomskog rada je izrada web trgovine pomoću React-a. React, također poznat i kao React.js ili ReactJS je JavaScript biblioteka otvorenog koda održavana od strane Facebook-a i zajednice individualnih developera. Može biti korišten kao osnova za izradu single-page aplikacija kao i mobilnih aplikacija (React Native). React nam omogućuje da složeni UI rastavimo na manje dijelove koji se zovu komponente, koje zatim možemo ponovno iskoristiti na više mjesta u aplikaciji. Korištenje React-a uobičajeno zahtijeva i rad s dodatnim bibliotekama, primjerice React Router koji omogućuje navigaciju između stranica u aplikaciji i npr. Redux za upravljanje sa stanjima. Za backend u aplikaciji koristi se Firebase, poznata Google-ova platforma koja radi na principu BaaS (*Backend-as-a-Service).* Za pisanje koda koristit će se uređivač koda Visual Studio Code, a glavni dio aplikacije bit će pisan u programskom jeziku Typescript, oboje kreirani od strane tvrtke Microsoft. Sama aplikacija bit će smještena na Firebase poslužitelju i bit će dostupna i kao PWA (*Progressive Web App).* Također bit će izrađen vlastiti CSS framework za potrebe aplikacije koji će biti optimiziran pomoću PostCSS-a.

1.1. Zadatak završnog rada

Potrebno je izraditi web trgovinu pomoću React-a koja će omogućiti administratorsko upravljanje trgovinom, sučelje za registraciju korisnika i dodavanje proizvoda u košaricu. Na kraju je potrebno izraditi izvještaj o svim kupovinama u web trgovini i obaviti testiranje.

1.2. Postojeća rješenja

Danas postoji raznolik izbor tehnologija koje se mogu implementirati s React-om. Neke od najpoznatijih su Shopify, Moltin i WooCommerce. Kako bi se web trgovina izradila od nule potrebno je puno vremena pa je poprilično teško naći web trgovinu implementiranu pomoću čistog React-a. Postojeće trgovine koje koriste React su petsmart.com, shopping.com, međutim one koriste Redux za upravljanje stanjima i Node.js za backend, dok će u ovome radu aplikacija biti kreirana pomoću React Context-a za upravljanje stanjima te Firebase funkcijama za izvršavanje plaćanja.

Zbog jednostavnosti i brze izrade, velik broj rješenja koristi neki od CSS okvira poput Material-UI, Bootstrap-a i Semantic UI, dok će se za potrebe diplomskog rada izraditi vlastiti CSS okvir.

2. KORIŠTENE TEHNOLOGIJE I ALATI

Moderna izrada web aplikacija zahtijeva poznavanje mnoštva tehnologija i alata. Glavni alati i tehnologije korištene u izradi ovog rada su :

- React
- Typescript
- HTML
- CSS
- PostCSS
- Firebase
- Stripe API
- Visual Studio Code

2.1. React

React je JavaScript biblioteka otvorenog koda za kreiranje korisničkih sučelja [1]. Održavana je od strane Facebook-a i zajednice individualnih developera. Može biti korišten kao osnova za izradu single-page aplikacija kao i mobilnih aplikacija (React Native). Omogućuje nam da korisničko sučelje rastavimo na manje dijelove koji se zovu komponente, koje zatim možemo ponovno iskoristiti na više mjesta u aplikaciji. Korištenje React-a uobičajeno zahtijeva i rad s dodatnim bibliotekama, primjerice React Router koji omogućuje navigaciju između stranica u aplikaciji i npr. Redux za upravljanje sa stanjima. React v16.8 donio je revoluciju u načinu implementacije React značajki poput stanja i životnih ciklusa u funkcijskim komponentama. Dodan je Hooks API koji omogućuje da se izvadi logika koja upravlja stanjima i nezavisno testira i ponovno iskorištava. U React-u v16.3.0 dodana je alternativa Redux-u, Context API. On nam omogućuje da dijelimo podatke kroz stablo komponenata bez potrebe za ručnim prosljeđivanjem kroz svaku komponentu. Ova praksa gdje se ručno prosljeđuju podaci kroz više komponenti zove se

prop-drilling i potrebno ju je izbjegavati jer dovodi do dijeljenja podataka kroz više komponenti, a koje te podatke ne koriste što dovodi do viška programskog koda. Za enterprise level aplikacije i dalje se preferira Redux zato što Context API nije optimiziran za česta ažuriranja stanja.

2.2. Typescript

Typescript je programski jezik otvorenog koda kreiran od strane tvrtke Microsoft. To je strogi sintaktički superset JavaScript-a koji opcionalno dodaje statičko tipkanje [2]. Dizajniran je za razvoj velikih aplikacija. Kod pisan u Typescript-u prvo se prevodi u JavaScript kod koji se zatim izvršava u browseru. U postavkama kompajlera možemo odrediti koji JavaScript standard želimo. Pri odabiru standarda potrebno je prethodno odlučiti koje sve verzije preglednika želimo podržavati. Statičko tipkanje unutar Typescript-a pridonosi smanjenju grešaka za vrijeme pisanja koda (podržava Intellisense), te samoj čitljivosti koda. Tip varijable određujemo tako da nakon imena varijable dodamo dvotočku te iza nje tip. Tako osiguravamo da jedna varijabla ne može biti bilo kojeg tipa kao što je to slučaj u JavaScriptu. Osim varijabli, možemo i funkcijama odrediti koji tip podatka one vraćaju. Još neke od značajki Typescript-a su preopterećivanje, što znači da možemo imati više funkcija s istim imenom, a različitim parametrima, sučelja (*interface),* klase i apstraktne klase, generičke funkcije, enumi, asinkrono izvršavanje pomoću async/await.

2.3. HTML

HTML je prezentacijski jezik za izradu web stranica. Glavna zadaća HTML-a je uputiti web preglednik kako da prikaže hipertekstualni dokument. On služi samo za opis hipertekstualnih dokumenata i nije sposoban izvršavati nikakvu drugu zadaću poput zbrajanja ili oduzimanja. Hipertekstualni dokumenti su HTML datoteke čiji je osnovni građevni element znak (tag) koji opisuje kako se nešto treba prikazati u web pregledniku [3]. U izradi aplikacije koristi se HTML5, novija revizija HTML standarda. Neka od najvažnijih novosti u odnosu na HTML su poboljšanja u baratanju s multimedijom, podrška za SVG sadržaj, semantički elementi za definiranje različitih dijelova web stranice koje koriste roboti/crawleri poput Google-a i Bing-a kako bi definirali koji sadržaj je važan, koji je pomoćni sadržaj, koji je za navigaciju i slično.

2.4. CSS

CSS je stilski jezik koji se koristi za opis prezentacije dokumenta napisanog pomoću HTML jezika [4]. Dizajniran je kako bi se odvojila prezentacija i sadržaj. Ovakav način odvajanja može poboljšati dostupnost sadržaja, pružiti veću fleksibilnost i kontrolu u prezentaciji dokumenta te smanjiti ponavljanje strukturnog sadržaja. Također omogućuje različite načine prikazivanja poput prikaza na zaslonu, prikaza za ispis, putem glasa (pomoću čitača zaslona) i na taktilnim uređajima baziranim na Braille-u. Podržava i pravila za oblikovanje ako se sadržaju pristupa putem mobilnih uređaja.

2.5. PostCSS

PostCss je alat za kreiranje dodataka baziranih na JavaScript-u kako bi se automatizirale neke CSS rutine [5]. U izradi ove aplikacije koriste se sljedeći dodaci :

- Autoprefixer – postprocesor koji parsira CSS i olakšava developerima posao tako da automatski prepoznaje i dodaje CSS naredbe koje su specifične za neki web preglednik
- Postcss-preset-env – pretvara CSS funkcionalnosti koje su u razvoju te još nisu službeno implementirane u tip kojeg razumije većina web preglednika
- Postcss-import – dodatak za transformaciju *@import* pravila dodavanjem sadržaja
- Postcss-pxtorem – dodatak koji pretvara *px* jedinice u *rem* jedinice. Pikseli su najjednostavniji za korištenje, ali mana im je to što ne dopuštaju pregledniku da promijene zadanu veličinu fonta (16px). Ovaj dodatak omogućava pregledniku da postavi veličinu fonta.
- Postcss-flexbugs-fixes – dodatak koji automatski rješava probleme koji se događaju korištenjem flexbox-a (fleksibilni okvir za responzivnost) tako da prilagođava flexbox naredbe u način prilagođen specifičnom pregledniku
- Postcss-purgecss – dodatak koji optimizira veličinu CSS datoteka tako da briše nekorišteni CSS. Naime često se događa da se koriste različiti okviri u izradi web stranica što dovodi do velikih CSS datoteka. Za potrebe ovog rada izrađen je vlastiti okvir što je dovelo do toga da ima viška koda kojeg je potrebno ukloniti.

- Postcss-cssnano – dodatak koji optimizira veličinu CSS datoteka tako da briše nepotrebne razmake, prazne redove, kompresira identifikatore i briše nepotrebne definicije. Kod se piše na ovaj način kako bi developerima kod bio što čitljiviji, ali to dovodi do većih datoteka.

2.6. Firebase

Firebase je web i mobilna developerska platforma kreirana od tvrtke Firebase Inc. koja je 2014. godine kupljena od strane Google-a [6]. To je BaaS (*Backend-as-a-service)* platforma koja kombinira različite funkcije. U ovome radu korištene su sljedeće funkcionalnosti Firebase-a :

- Firebase Authentication – servis za autentifikaciju korisnika
- Cloud Firestore - skalabilna baza podatka za web i mobilne uređaje
- Firebase Hosting – web servis za smještanje web stranica na web server
- Firebase Functions – programski okvir koji pruža izvršavanje backend koda kao odgovor na događaje pokrenute Firebase značajkama i HTTPS zahtjevima

2.7. Stripe API

Stripe API je aplikacijsko programsko sučelje kreirano od tvrtke Stripe, Inc. Njegova svrha je procesiranje plaćanja u mobilnim i web aplikacijama. Organiziran je oko rest-a, prihvaća formom enkodirane zahtjeve, a vraća JSON enkodirane odgovore i koristi standardne HTTP kodove [7]. Podržava testni način rada koji se koristi u ovome radu kojime možemo testirati plaćanja bez da se zapravo naplati za proizvod. Sva plaćanja administrator može vidjeti unutar stripe dashboarda.

2.8. Visual Studio Code

Visual Studio Code je uređivač koda kreiran od tvrtke Microsoft. Podržava debugiranje, označavanje sintakse, podržava više programskih jezika, ima inteligentno samodovršavanje koda, podržava snippete i refaktoriranje koda i ima ugrađen Git [8]. Korisnik može mijenjati temu, prečace na tipkovnici, različite postavke. Jedna od najmoćnijih stavki koje Visual Studio Code ima su ekstenzije. Naime, podržava dodavanje ekstenzija kreiranih od strane zajednice i Microsofta.

3. Realizacija zadatka

U ovom poglavlju bit će opisan tijek izrade web trgovine, konfiguracija svih korištenih alata i implementacija Stripe API-ja za plaćanje. Sav kod pisan je u uređivaču koda Visual Studio Code.

3.1. Opis aplikacije

Potrebno je izraditi web aplikaciju korištenjem React-a i implementirati sve funkcionalnosti klasične web trgovine poput registracije korisnika, dodavanja/brisanja proizvoda iz košarice, kupnje pomoću API-ja poput Stripe-a, Braintree-a ili PayPal-a. Osim toga potrebno je omogućiti korisniku da ima uvid u prošle kupovine i kreirati administratorsko sučelje putem kojega će se upravljati trgovinom. Kako bi se smanjio broj grešaka u programskom kodu, odlučeno je da će se koristiti programski jezik Typescript. Typescript se koristi za enterprise aplikacije ponajviše zbog strogog načina tipkanja koje pridonosi lakšem debugiranju i boljoj kontroli izvršavanja programskog koda. Korištene su najnovije mogućnosti React-a, React Hooks, React Lazy i React Suspense, ali one će biti objašnjene nešto kasnije. Aplikacija je responzivna i ima mogućnost instalacije kao PWA (Progressive Web App). Kreirana je navigacijska traka koja je prikazana na svim stranicama te osim toga sadrži i switch gumb koji omogućuje mijenjanje između svijetle i tamne teme. Na početnoj stranici prikazane su sve glavne kategorije web trgovine. Klikom na neku od kategorija navigira se prema stranici za prikaz svih potkategorija. Moguće je kliknuti na „shop“ gumb na navigacijskoj traci koji vodi na stranicu koja prikazuje sve glavne kategorije i potkategorije. Klikom na neku od potkategorija prikazuje se stranica koja sadrži popis svih proizvoda određene potkategorije koji su klikabilni te vode na stranicu za prikaz detalja o proizvodu i njegovo dodavanje u košaricu. Izrađena je stranica za prijavu/odjavu koja sadrži dvije komponente u kojima su implementirane forme za prijavu i odjavu. Putem navigacijske trake također je moguće navigirati na stranicu za upravljanje računom i pregled svih kupovina, kao i na stranicu za administratorsko upravljanje trgovinom ako je prijavljeni korisnik administrator.

3.2. Postavljanje projekta

Projekt je kreiran pomoću create-react-app (u daljnjem tekstu CRA), predloška koji izrađuje osnovnu verziju React aplikacije, spremnu za daljnji rad te se pokazao kao vrlo kvalitetna polazna točka za kreiranje React aplikacija. Ovako kreiran projekt sadrži neka ograničenja poput nemogućnosti upravljanja Webpack-om, paketom čija je svrha zapakiravanje JavaScript datoteka za korištenje u web pregledniku [9]. S obzirom na to da se za stiliziranje aplikacije odlučilo za kreiranje vlastitog CSS okvira te je isti bilo potrebno optimizirati za produkciju, korišten je PostCSS, alat za automatizaciju određenih CSS rutina. Kako je za njegovu konfiguraciju, kao i konfiguraciju njegovih dodataka, potrebno modificirati Webpack, a CRA brani modifikaciju istog, bilo je potrebno instalirati react-app-rewired, paket koji omogućuje modifikaciju Webpacka bez potrebe za raspakiravanjem create-react-appa, nakon kojega je potrebno ručno konfigurirati Webpack. Raspakiravanje CRA je ireverzibilan proces stoga je preporučljivo „izvagati“ koje su to prednosti i mane raspakiravanja istog, odnosno odlučiti da li je to isplativo raditi. Za vrijeme kreiranja projekta moguće je naznačiti da se aplikacija želi pisati u Typescriptu pomoću zastavice – template typescript.

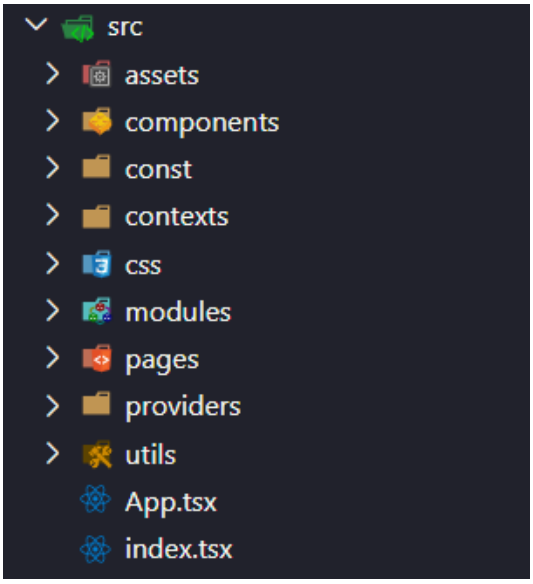
Nakon što je projekt kreiran, potrebno je kreirati Firebase projekt unutar Firebase konzole putem web preglednika nakon kojega se dobije konfiguracijska datoteka koja sadrži API ključeve za pristup. Zatim je potrebno instalirati Firebase paket kojega konfiguriramo pomoću naredbe *firebase init*. Potrebno je označiti koji će se sve Firebase servisi koristiti unutar aplikacije. Za potrebe ovog rada koristit će se Firebase Firestore, Firebase functions i Firebase hosting. Nakon inicijalizacije dobije se firestore.rules datoteka u koju je moguće zapisati pravila za pristup bazi podataka. Ovo je potrebno kako neovlašteni korisnik ne bi imao sposobnost čitanja, izmjene i brisanja podataka iz baze. Za potrebe projekta odlučeno je da će se kreirati dvije kolekcije u bazi, users i products. Za users kolekciju postavljeno je pravilo zapisivanja, čitanja i ažuriranja ako je trenutni korisnik autenticiran. Products kolekcija podešena je tako da svi mogu čitati iz nje, ali samo administrator može zapisivani podatke u nju. Pravila pristupa prikazana su na idućoj slici.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId}/{document=**} {
      allow read, create, update: if request.auth != null && request.auth.uid == userId;
    }

    match /products/{category}{
    allow read;
    allow create: if request.auth.uid == "8XGTrzETESfWEntD9EoPQb5QBou1";
    }
  }
}
```

Sl.3.1. Firebase pravila

Kako bi se pisao što kvalitetniji kod uz što manje potencijalnih grešaka, potrebno je instalirati Eslint, Prettier i Stylelint pakete u projekt i pripadajuće ekstenzije za Visual Studio Code. Za svaki od paketa potrebno je kreirati konfiguracijske datoteke unutar projekta u kojima se nalazi set pravila. Podržane ekstenzije konfiguracijske datoteke su .js, .json i .yml. Mogućnosti konfiguracije nalaze se na službenoj stranici dokumentacije svakog alata. Eslint je alat za prepoznavanje problematičnih uzoraka koji se nalaze u programskom kodu, Prettier je alat za formatiranje koda koji uljepšava kod tako da bude što čitljiviji, a Stylelint je alat za analizu problematičnih uzoraka u CSS kodu.

S obzirom na to da će projekt sadržavati velik broj različitih datoteka, bilo je potrebno implementirati neki od načina arhitekture. Arhitektura je strukturirana prema idućoj slici.



Sl.3.2. Arhitektura aplikacije

Kao što možemo vidjeti iz priložene slike, unutar src foldera izrađeno je više poddirektorija. U assets direktorij spremamo sve datoteke koje se koriste u aplikaciji poput slika i fontova. Components direktorij sadrži sve React komponente koje su nanovo iskoristive kroz cijelu aplikaciju, a modules direktorij sadržava komponente i ekrane koje se koriste samo za određene dijelove aplikacije te oni nisu nanovo iskoristivi. Const direktorij sadrži konstantne dijelove aplikacije poput ruta i config-a (u ovom slučaju Firebase config). Css direktorij sadrži sve stilove, a pages direktorij sadržava kreirane stranice koje pozivaju određene komponente. Utils direktorij sadrži regularne izraze za validaciju, a datoteke unutar contexts i providers direktorija služe za upravljanje stanjima aplikacije.

3.3. Izrada CSS okvira

Kako je rečeno u poglavlju 1.2., za potrebe aplikacije izradit će se vlastiti CSS okvir koji će biti optimiziran pomoću različitih PostCSS dodataka. PostCSS zahtijeva modifikaciju Webpack-a, a s obzirom na to da CRA daje dobru polazišnu točku za izradu React aplikacija, ali su mogućnosti njegove konfiguracije ograničene, koristio se paket react-app-rewired pomoću kojega je moguće izvesti potrebne modifikacije. Popis korištenih PostCSS dodataka kao i njihova uloga detaljno su

objašnjeni u poglavlju 2.5. U nastavku je prikazana config-overrides datoteka u kojoj pomoću react-app-rewired-a konfiguriramo PostCSS dodatke.

```
module.exports = config => {
    require('react-app-rewire-postcss')(config, {  1.2K (gzipped: 611)
        plugins: () => [
            require('postcss-import'),  130.1K (gzipped: 38K)
            require('postcss-preset-env')({
                features: {
                    'nesting-rules': true,
                    stage: 3,
                },
            }),
            require('postcss-flexbugs-fixes'),  96.8K (gzipped: 27.9K)
            require('postcss-pxtorem'),  98.8K (gzipped: 28.6K)
            require('@fullhuman/postcss-purgecss')({
                content: ['./**/*.tsx'],
            }),
            require('cssnano'),
        ],
    });

    return config;
};
```

Sl.3.3. config-overrides datoteka

S obzirom na to da je stiliziranje aplikacije vrlo različito te kako bismo dijelove poput boja, fontova i spacinga (margine, paddinzi) učinili nanovo iskoristivim kroz čitavu aplikaciju, za svaki dio napisane su CSS varijable koje će se zatim pozivati u različitim CSS klasama pomoću funkcije var(). Ovakav način zapisivanja vrlo je koristan zato što ovako imamo strogo definirane vrijednosti i ako se odlučimo na izmjenu fonta ili boje u aplikaciji, jednostavno izmijenimo vrijednost varijable na samo jednom mjestu te su izmjene trenutno vidljive u aplikaciji.

CSS direktorij sadrži tri tipa CSS datoteka. Prvi tip su datoteke nazvane vars.#.css gdje je # naziv dijela, odnosno svrha datoteke. Za globalni scope, sve varijable je potrebno pisati u root selektoru. Primjer vars datoteke možemo vidjeti na idućoj slici gdje je prikazana vars.colors.css datoteka koja sadrži boje za buttone i notifikacijsku komponentu.

```css
:root {
    --color-button-primary: #17a2b8;
    --color-button-primary--hover: #138496;
    --color-button-border-primary: #117a8b;

    --color-button-secondary: #28a745;
    --color-button-secondary--hover: #218838;
    --color-button-border-secondary: #1e7e34;

    --color-border--error: #f5c6cb;
    --color-border--success: #c3e6cb;

    --color--error: #721c24;
    --color--success: #155724;

    --color-background--error: #f8d7da;
    --color-background--success: #d4edda;
}
```

Sl.3.4. Arhitektura aplikacije

Drugi tip CSS datoteke imaju naziv components.#.css u kojima su zapisani svi stilovi za pojedinu React komponentu. Primjer ove datoteke prikazan je na idućoj slici koja sadrži stilove za gumbove u aplikaciji.

```css
.button {
    background-color: var(--color-button-primary);
    border: 1px solid transparent;
    border-radius: 4px;
    transition: 0.15s ease-in-out;

    &:hover {
        background-color: var(--color-button-primary--hover);
    }
}

.google {
    background-color: var(--color-button-secondary);

    &:hover {
        background-color: var(--color-button-secondary--hover);
    }
}
```

Sl.3.5. component.button.css datoteka

Treći tip datoteke su app.#.css datoteke koje sadrže klase koje utječu na izgled cjelokupne aplikacije, odnosno upravljaju layoutom, tipografijom, ikonama i slično. Glavna datoteka projekta je app.css pomoću koje uključujemo sve ostale CSS datoteke u projekt pravilom @import.

```
@import 'vars.media.css';
@import 'vars.colors.css';
@import 'vars.spacers.css';
@import 'vars.typography.css';

@import 'app.icons.css';
@import 'app.utils.css';
@import 'app.global.css';
@import 'app.layout.css';
@import 'app.animations.css';
@import 'app.typography.css';

@import 'component.list.css';
@import 'component.alert.css';
@import 'component.button.css';
@import 'component.header.css';
@import 'component.switch.css';
@import 'component.payment.css';
@import 'component.cartitem.css';
@import 'component.textinput.css';
```

Sl.3.6. App.css datoteka

Najbitnije CSS datoteke su app.utils. app.global i app.typography. Utils datoteka sadrži klase za upravljanje širinom HTML elemenata, kao i za upravljanje pozicioniranjem putem margina i paddinga. U app.global datoteci nalaze se selektori u kojima je postavljen stil za pojedine HTML elemente poput headera, anchora, listi i ostalo. Osim njih nalazi se body selektor koji definira defaultnu boju teksta kao i boju pozadine. Datoteka app.typography sadrži selektore u kojima je definirana defaultna veličina fonta, visina linije i naziv fonta. Ona također sadrži klase za manipulaciju veličinom fonta kao što je prikazano na idućoj slici.

```
.text-base {
    font-size: var(--font-size-base);
    line-height: var(--line-height-normal);
}

.text-lg {
    font-size: var(--font-size-lg);
    line-height: var(--line-height-relaxed);
}

.text-xl {
    font-size: var(--font-size-xl);
    line-height: var(--line-height-loose);
}
```

Sl.3.7. Isječak klasa za manipulaciju veličine fonta

Sve klase u izrađenom CSS okviru imenovane su tako da ih je vrlo jednostavno zapamtiti, iako postoji dodatak za Visual Studio Code koji sprema nazive klasa u cache. Veličine paddinga, margina i fontova imaju nekoliko različitih sufiksa u nazivu klasa gdje osim bazne klase postoje dvije manje te pet većih.

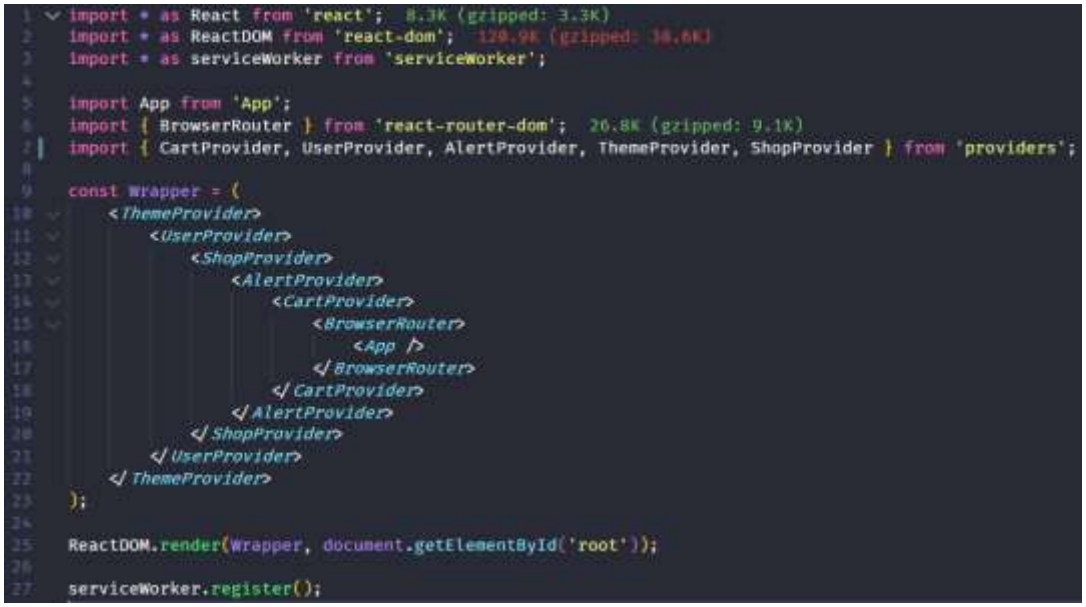
3.4. Upravljanje stanjima

Manipulacija globalnim stanjima kao i njihovo prosljeđivanje odvija se pomoću React Context API-ja koji je detaljnije objašnjen u poglavlju 2.1. U aplikaciji imamo nekoliko različitih stanja kojima je potrebno rukovati : podaci o trenutnom korisniku, podaci o proizvodima, podaci o trenutno odabranoj temi, podaci o košarici te podaci o trenutnoj obavijesti. Naredbom React.createContext kreiramo context unutar kojega specificiramo što će sve biti proslijeđeno putem Context providera koji dijeli podatke svim child komponentama unutar providera. Osim inicijalnog kostura što se sve prosljeđuje, potrebno je postaviti i početno stanje varijable. Na idućoj slici možemo vidjeti context za košaricu te sve što prosljeđujemo putem providera.

```
1   import React from 'react';  8.3K (gzipped: 3.3K)
2
3   import { CartContextProps } from 'modules/cart';
4
5   export const cartContext = React.createContext<CartContextProps>({
6       dispatch: () => null,
7       shoppingCart: [],
8       cartCounter: 0,
9       cartTotal: 0,
10  });
```

Sl.3.8. Kreiranje contexta za košaricu

Kreirani context potrebno je proslijediti child komponentama. To se radi pomoću Context providera unutar kojega se nalaze komponente kojima prosljeđujemo podatke. Svaki provider je zapisan u odvojenu datoteku unutar kojega je implementirana sva logika. Kako bismo grupirali providere na jedno mjesto, svakome je proslijeđen property *children* koji prosljeđuje sve obavijeno parent komponentom. Napisane providere na ovaj način je vrlo jednostavno proslijediti App komponenti unutar Index.tsx datoteke kao što se može vidjeti na idućoj slici.



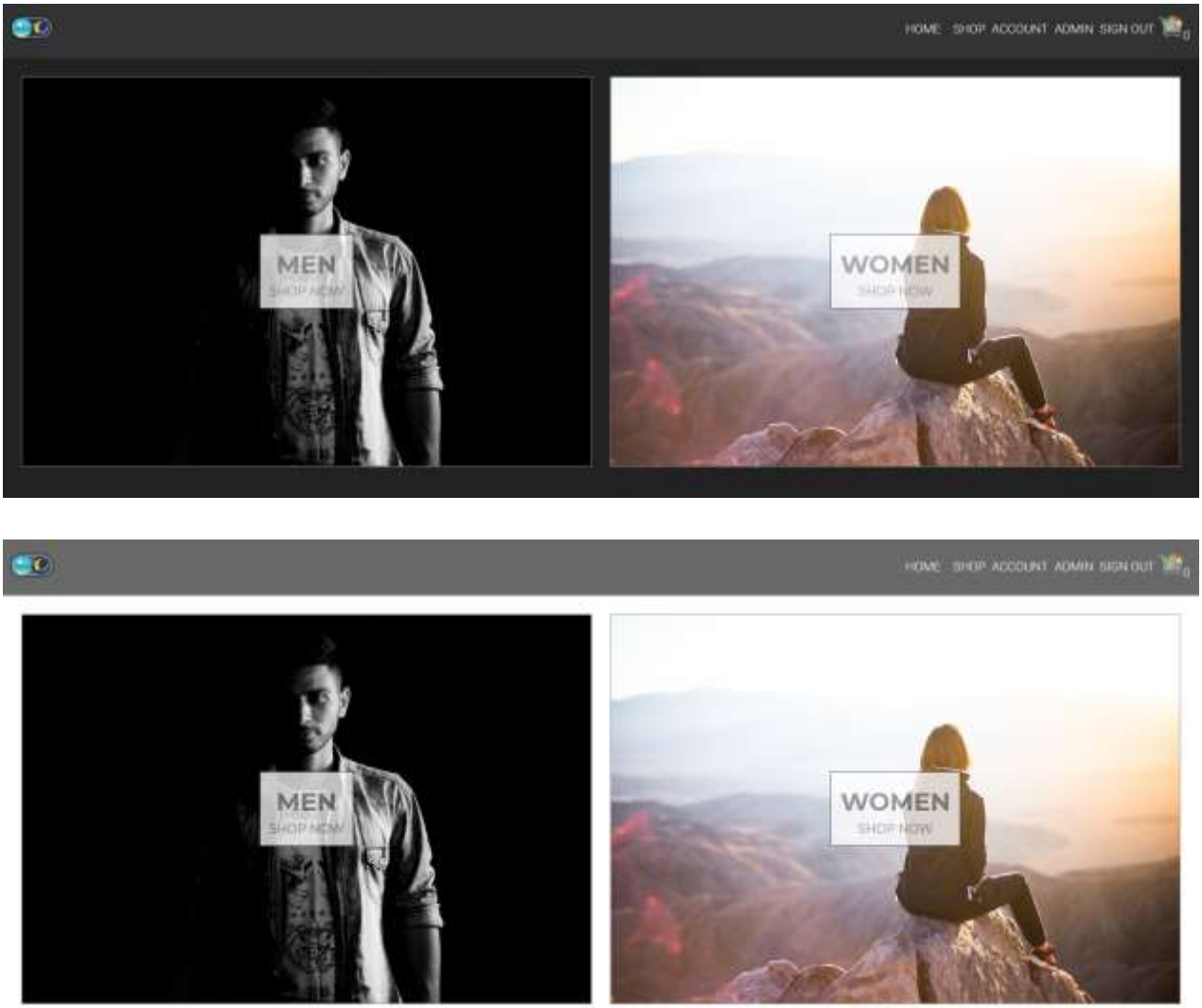
Sl.3.9. Index.tsx datoteka

ThemeProvider sadrži potrebnu logiku za mijenjanje teme aplikacije. Prvo iz modules foldera dohvaćamo polje stringova koje sadrži CSS varijable s podacima o boji koje zatim pri inicijalnom učitavanju aplikacije postavljamo ovisno ključu unutar localStorage-a, a ako on nije postavljen čitaju se postavke sustava te se postavlja preferirana tema. S obzirom na to da se React-ov hook useEffect izvršava asinkrono nakon crtanja i ažuriranja prikaza, sva logika za čitanje postavki sustava, localStorage-a i postavljanje teme je implementirana unutar useLayoutEffect hook-a koji se izvršava nakon DOM mutacija, odnosno nakon crtanja, ali sinkrono, prije ažuriranja prikaza.

```
useLayoutEffect(() => {
    const checkPreferenceAndSet = () => {
        if (window.matchMedia('(prefers-color-scheme: dark)').matches) {
            window.localStorage.setItem('theme', 'light');
            applyTheme(darkTheme);
        }

        if (window.matchMedia('(prefers-color-scheme: light)').matches) {
            window.localStorage.setItem('theme', 'dark');
            applyTheme(lightTheme);
        }
    };

    !window.localStorage.getItem('theme')
        ? checkPreferenceAndSet()
        : window.localStorage.getItem('theme') === 'dark'
        ? setLightWithKey()
        : setDarkWithKey();
}, [setDarkWithKey, setLightWithKey]);
```

Sl.3.10. UseLayoutEffect hook

Na slici je prikazana implementacija logike za promjenu teme aplikacije. Funkcija getItem() prima naziv ključa po kojemu se pretražuje localStorage. Ako on ne postoji poziva se funkcija checkPreferenceAndSet koja postavlja temu ovisno o postavkama sustava, a ako ključ postoji postavlja se tema ovisno o stringu kojeg ključ sadrži. Slika 3.11. prikazuje izgled aplikacije u svijetloj i tamnoj temi.



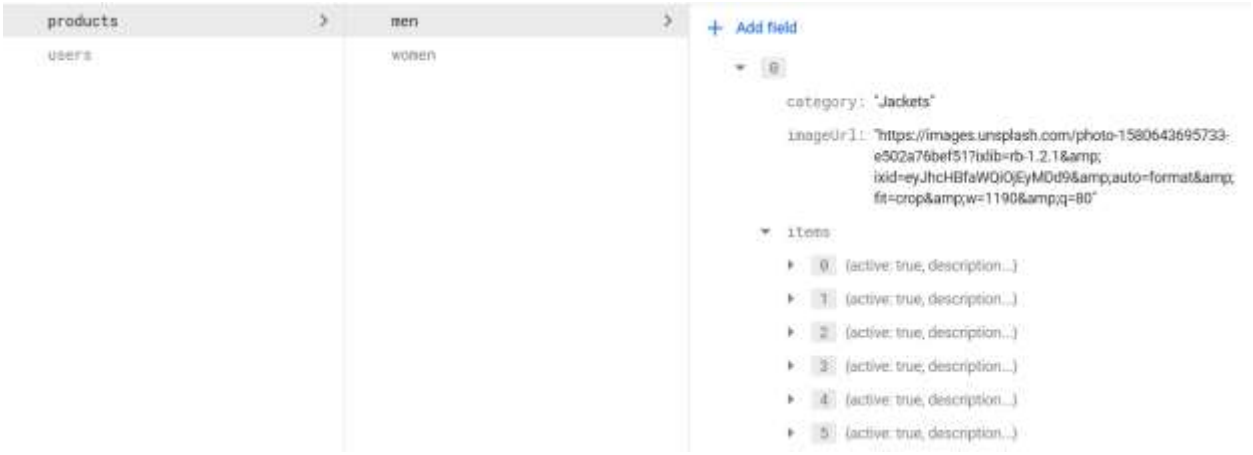
Sl.3.11. Tamna i svijetla tema

UserProvider koristi useEffect hook unutar kojega je implementiran Firebase listener koji prati i zapisuje stanje trenutnog korisnika. U prvi dio hook-a ide dio koji će se pozvati pri inicijalizaciji komponente ili ovisno o uvjetu unutar uglatih zagrada koje se nalaze na kraju hook-a, a unutar returna nalazi se dio koji se poziva kada se komponenta unmounta. Ako se ništa ne nalazi u zagradi to znači da se hook poziva pri učitavanju komponente.

```
import React, { useState, useEffect } from 'react';  8.3K (gzipped: 3.3K)

import { auth } from 'const';
import { authContext } from 'contexts';

export const UserProvider: React.FC = ({ children }) => {
    const [currentUser, setcurrentUser] = useState(auth.currentUser);

    useEffect(() => {
        const unsubscribe = auth.onAuthStateChanged(currentUser => setcurrentUser(currentUser));

        return () => {
            unsubscribe();
        };
    }, []);

    return <authContext.Provider value={currentUser}>{children}</authContext.Provider>;
};
```

Sl.3.12. Implementacija UserProvider-a

ShopProvider dohvaća podatke iz baze podataka i sprema podatke u state koji se zatim prosljeđuju kroz aplikaciju. Koristi se onSnapshot() metoda koja prati stanje dokumenta u bazi te pomoću callbacka ažurira stanje aplikacije te se ažurira vrijednost unutar providera. Proslijeđene podatke koriste se pri prikazu kategorija i svih proizvoda, te u dijelu za administraciju trgovine. Baza podataka strukturirana je tako da postoji kolekcija *products* koja sadrži dokumente s nazivom glavnih kategorija proizvoda unutar kojih se nalaze podaci o slici kategorije te podaci o proizvodima unutar te kategorije. Podaci o proizvodima sadrže potkategorije unutar kojih se nalazi popis proizvoda koji pak sadrže informacije o slici, nazivu, opisu, cijeni, ID-ju i trenutnom statusu proizvoda. Statusom proizvoda upravlja se pomoću administratorskog sučelja putem kojega možemo upravljati ponudom proizvoda u našoj trgovini. Slika 3.13. prikazuje strukturu baze koja sadrži informacije o proizvodima.



Sl.3.13. Isječak iz Firestore baze

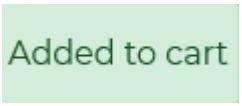
AlertProvider prosljeđuje informaciju o trenutnoj poruci, metodu za promjenu trenutne poruke te podatak o trenutnom prikazu. Pri pozivu metode addAlert() mijenja se trenutni sadržaj poruke i aktivira se flag koji je povezan s komponentom za prikaz obavijesti. Nakon 1.5 sekunde poziva se metoda koja skriva sadržaj poruke. Na ovaj način vrlo jednostavno je implementiran globalni sustav obavijesti.

```
import React, { useState } from 'react';  8.3K (gzipped: 3.3K)

import { AlertProps } from 'modules/alert';
import { alertContext } from 'contexts';

export const AlertProvider: React.FC = ({ children }) => {
    const [alertData, setAlertData] = useState<AlertProps | null>(null);
    const [isActive, setisActive] = useState(false);

    const addAlert = (item: AlertProps) => {
        setAlertData(item);
        setisActive(true);

        setTimeout(() => {
            setisActive(false);
        }, 1500);
    };

    return <alertContext.Provider value={{ addAlert, alertData, isActive }}>{children}</alertContext.Provider>;
};
```

Sl.3.14. Implementacija AlertProvidera

There is no user record corresponding to this identifier. The user may have been deleted.

Sl.3.15. Poruka greške



Sl.3.16. Poruka uspješno izvedene akcije

Slike 3.15. i 3.16. prikazuju implementiranu UI komponentu poruka, koja ispisuje poruku na ekranu i prilagođava boju pozadine ovisno o uspješnosti akcije.

CartProvider je najsloženiji provider te je u njemu implementirana logika za upravljanje košaricom. Prvo su napisane funkcije za dodavanje i uklanjanje proizvoda iz košarice koje su zatim implementirane unutar providera pomoću useReducer hook-a. UseReducer hook alternativa je useState hooku, ali se koristi kada imamo složenu logiku koja upravlja stanjima, gdje bitnu ulogu igra prethodno stanje. Ovaj hook idealan je za implementaciju košarice zbog toga što se broj proizvoda u košarici inkrementira za neki broj, a pritom je potrebno poznavati prethodno stanje. Koristi se *dispatch* funkcija koja garantira da se komponente neće rerenderati osim ako ona zahtijeva podatke o trenutnom stanju. Dispatch metoda proslijeđena je putem providera te se ona poziva kada korisnik klikne na gumb za dodavanje/brisanje proizvoda iz košarice. Na idućoj slici prikazan je dio implementiranih funkcija za upravljanje košaricom.

```
import { CartItemProps, ItemProps } from './';

export const addToCart = (shoppingCart: CartItemProps[], cartItem: ItemProps) => {
    if (shoppingCart.find(item => item.id === cartItem.id)) {
        return shoppingCart.map(item => (item.id === cartItem.id ? { ...item, quantity: item.quantity + 1 } : item));
    }
    return [...shoppingCart, { ...cartItem, quantity: 1 }];
};

export const removeFromCart = (shoppingCart: CartItemProps[], cartItem: ItemProps) => {
    const existingCartItem = shoppingCart.find(item => item.id === cartItem.id);

    if (existingCartItem?.quantity === 1) {
        return shoppingCart.filter(item => item.id !== cartItem.id);
    }

    return shoppingCart.map(item => (item.id === cartItem.id ? { ...item, quantity: item.quantity - 1 } : item));
};

export const getItemCount = (shoppingCart: CartItemProps[]) =>
    shoppingCart.reduce((accumulatedQuantity, cartItem) => accumulatedQuantity + cartItem.quantity, 0);

export const getCartTotal = (shoppingCart: CartItemProps[]) =>
    shoppingCart.reduce((acc, cartItem) => acc + cartItem.quantity * cartItem.price, 0);
```

Sl.3.17. Implementirane funkcije za upravljanje košaricom

3.5. Upravljanje rutama

React nema ugrađenu podršku za upravljanje rutama te se za to najčešće koristi Reach Router ili React Router. Odlučeno je da će se koristiti React Router s obzirom na to da je Reach Router nešto noviji pa postoje šanse da nije dovoljno dorađen što može dovesti do problema u radu aplikacije. U budućoj verziji React Routera plan je da se spoje obje biblioteke u jednu te će se tako preuzeti sve najbolje iz obje biblioteke te će tako React imati jedno vrlo kvalitetno rješenje za upravljanje rutama.

Routing u aplikaciji izveden na novi način zapisa ruta gdje se unutar children dijela Route komponente React Router-a ubacuje komponenta na koju ta ruta vodi. Osim toga potrebno je dodati property *path* koji sadrži putanju. Prije su se komponente dodavale kao property te su se podaci poput URL parametara, putanji i lokaciji automatski prosljeđivale putem propova u child komponentama. Noviji način zapisa je ljepši te se do navedenih podataka dolazi pomoću hookova useHistory, useParams, useLocation i useRouteMatch. Sve rute su okružene sa Switch i BrowserRouter komponentama. Switch komponenta prikazuje prvu komponentu koja odgovara putanji dok BrowserRouter koristi HTML5 history API kako bi UI i URL bili sinkronizirani. Sve rute su okružene s React.Suspense komponentom koja omogućuje da se obavlja neka radnja za vrijeme učitavanja koda. Trenutno je u eksperimentalnoj fazi i najčešće se koristi za prikaz spinnera za vrijeme učitavanja komponenti unutar ruti. Osim toga, implementiran je lazy loading koji dijeli stranice u manje komadiće (engl. *chunks)* i pokazao se kao odličan način za optimizaciju performansi aplikacije jer se prilikom inicijalnog učitavanja aplikacije ne treba učitavati kod koji se trenutno ne koristi. Na idućoj slici prikazan je isječak koda koji prikazuje implementaciju navedenih mogućnosti.

Sl.3.18. Isječak Routing komponente

Kao što možemo vidjeti na prethodnoj slici, u ovome dijelu je implementirana logika koja zabranjuje posjete određenim rutama ako korisnik nije prijavljen ili nema dovoljne razine ovlasti.

3.6. Izrada komponenti sučelja

Izrada korisničkog sučelja zahtijeva smišljenu strukturu kako bi bilo jednostavno održavati i nadograđivati postojeću aplikaciju.

S obzirom na to da je zaglavlje komponenta koja će se prikazivati u svim stranicama, kreirana je Layout komponenta u koju je smješteno zaglavlje, a unutar <main> oznake smještene su sve ostale komponente i notifikacijska komponenta. Ako aplikacija sadržava i komponentu podnožja najbolje ju je smjestiti odmah ispod <main> oznake. U nastavku je implementirana Layout komponenta s navedenom arhitekturom.

```
1   import React from 'react';  8.3K (gzipped: 3.3K)
2
3   import { Header } from 'components';
4   import { AlertNotification } from 'modules/alert';
5
6   export const Layout: React.FC = ({ children }) => (
7       <>
8           <Header />
9
10          <main>
11              {children}
12              <AlertNotification />
13          </main>
14      </>
15  );
```

Sl.3.19. Layout komponenta

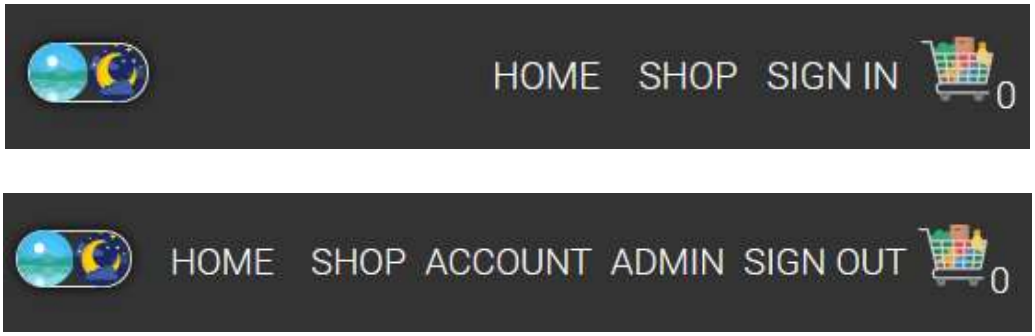
Routing i Layout implementirani su odvojeno kako bi App.tsx komponenta bila sa što manje koda. S obzirom na to da sadržava navedene dijelove jer je polazišna točka aplikacije, sve stavlja u App.tsx te ona postaje velika i teško je pratiti popratnu logiku što dovodi do težeg održavanja i nadogradnje aplikacije.

```tsx
import React from 'react';  8.3K (gzipped: 3.3K)

import { Layout, Routing } from './modules/app';

import 'normalize.css';
import 'css/app.css';

const App = () => {
    return (
        <Layout>
            <Routing />
        </Layout>
    );
};

export default App;
```

Sl.3.20. App.tsx

Kao što je moguće vidjeti iz slike, unutar komponente pozivaju se normalize.css (CSS reset koji omogućuje da preglednici prikazuju elemente konzistentnije i u skladu s modernim standardima), i app.css koji poziva sve napisane stilove kako bi se oni mogli koristiti u ostalim komponentama.

Iduće je implementirana komponenta zaglavlja koja sadrži gumb za promjenu teme koji se nalazi s lijeve strane, a s desne strane nalaze se linkovi za navigiranje po rutama. Link za pregled prošlih kupovina skriven je dok se korisnik ne prijavi u aplikaciju. Ako je korisnik admin, prikazuje mu se dodatan link koji vodi na stranicu za upravljanje trgovinom putem koje je moguće aktivirati/deaktivirati proizvode u trgovini kao i dodavati nove proizvode.

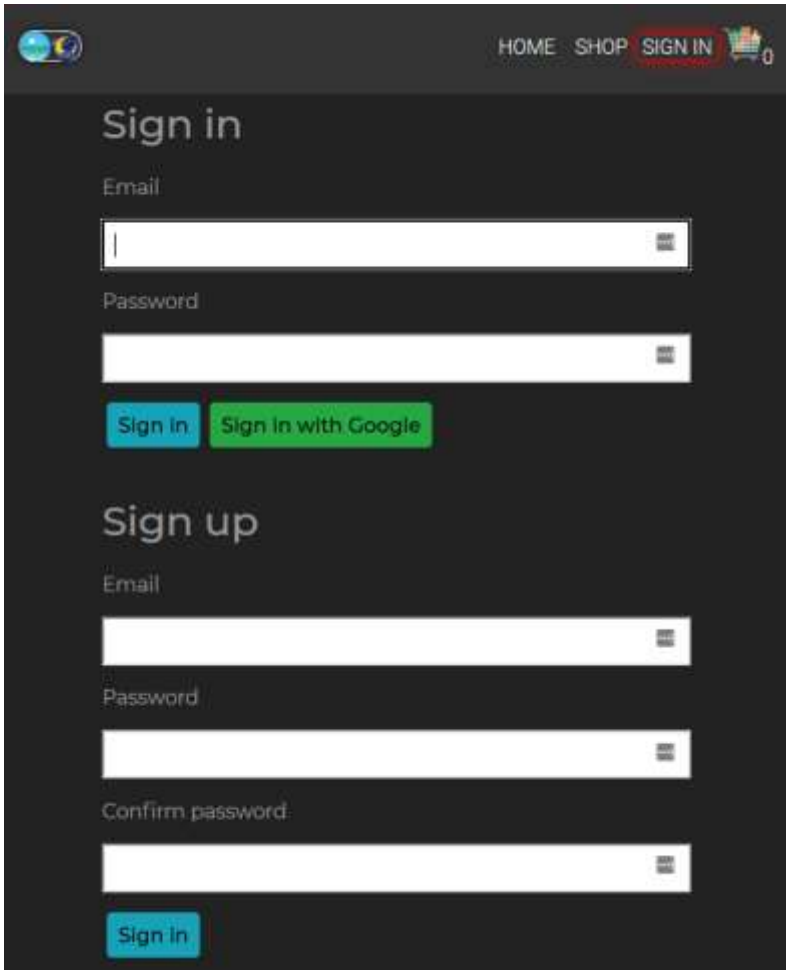


Sl.3.21. Različita stanja zaglavlja

Nadalje su napisane komponente koje će se koristiti više puta u aplikaciji poput gumba, polja za unos, komponenta za prikaz kategorije i proizvoda te kontejnerska komponenta ListOverview unutar koje se više puta poziva ListItem komponenta koja prikazuje podatak. Svaka komponenta prihvaća različite vrste podataka koji su potrebni za pravilan rad komponente. Primjer je TextInput komponenta koja prima naziv polja te referencu na input kojim upravlja react-hook-form paket za validaciju unosa. Za validaciju unosa napisana su dva regularna izraza. Prvi izraz provjerava da li je unesena e-mail adresa ispravnog formata, a drugi kontrolira pravilan unos šifre za vrijeme kreiranja korisničkog računa. Potrebno je imati šifru minimalne duljine 6 znakova, s barem jednim velikim slovom, malim slovom i brojem. Iduća slika prikazuje komponentu koja upravlja validacijom i prikazuje formu za prijavu korisnika,

```
export const SignIn = () => {
    const { addAlert } = useContext(alertContext);
    const { register, handleSubmit, getValues, errors } = useForm<SignInProps>();

    const onSubmit = () => {
        const { email, password } = getValues();
        auth.signInWithEmailAndPassword(email, password).catch(error =>
            addAlert({ message: error.message, status: 'error' }),
        );
    };

    const emailRef = register({
        required: true,
        pattern: {
            value: emailRegex,
            message: 'Invalid email address',
        },
    });

    const passwordRef = register({
        required: true,
    });

    return (
        <section className="flex flex-column justify-between mb-xl">
            <h1 className="text-xl mb-lg font-medium">Sign in</h1>

            <form onSubmit={handleSubmit(onSubmit)}>
                <TextInput name="email" type="email" label="Email" inputRef={emailRef} autoFocus />
                {errors.email && <p style={{ color: 'var(--color--error)' }}>{errors.email.message}</p>}

                <TextInput name="password" type="password" label="Password" inputRef={passwordRef} />
                {errors.password && <p style={{ color: 'var(--color--error)' }}>{errors.password.message}</p>}

                <Button type="submit">Sign in</Button>

                <Button onClick={signInWithGoogle} google>
                    Sign in with Google
                </Button>
            </form>
        </section>
    );
};
```

Sl.3.22. Komponenta za prikaz i validaciju forme za prijavu



Sl.3.23. Stranica za prijavu i registraciju

ListOverview je kontejnerska komponenta koja se prikazuje na tri stranice – stranice za prikaz svih glavnih kategorija, stranice za prikaz potkategorija te na stranici za prikaz svih proizvoda u odabranoj potkategoriji. Prima obrađene podatke iz page-a koji čita podatke iz contexta te ih prosljeđuje potkomponentama koje se zatim više puta prikazuju.

```
export const ListOverview: React.FC<ListOverviewProps> = ({ data, listName }) => {
    return (
        <section className="flex flex-wrap justify-between">
            {data.map(({ name, imageUrl, id }) => (
                <ListItem path={listName} name={name} imageUrl={imageUrl} key={imageUrl} id={id} />
            ))}
        </section>
    );
};
```

Sl.3.24. ListOverView komponenta

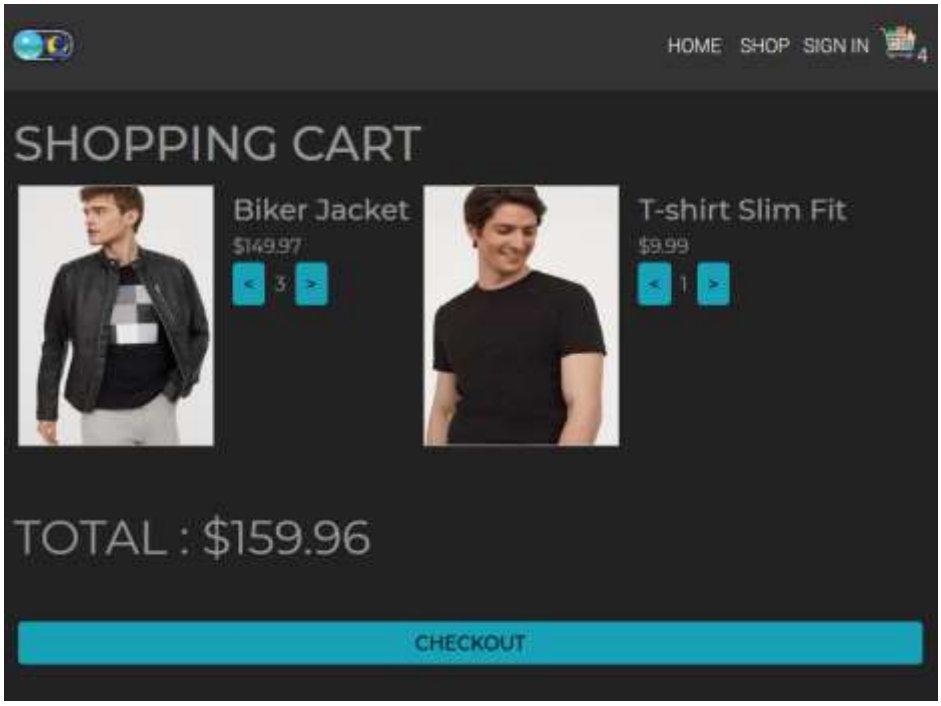
Kao što je vidljivo iz slike 3.24. ListItem komponenta prima naziv glavne kategorije na koju vodi kada se klikne na nju, naziv kategorije, URL slike, id i key. Ova komponenta prima više različitih podataka zato što se više puta koristi u aplikaciji te vodi na više različitih putanja ovisno o parametrima unutar URL-a. Ako se korisnik nalazi na shop stranici, ListItem vodi na potkategoriju gdje je prvi URL parametar glavna kategorija, a drugi odabrana potkategorija. Ako se korisnik nalazi na prikazu glavnih kategorija, ListItem vodi na potkategoriju. Ako postoji parametar potkategorije u URL-u onda ova komponenta vodi na prikaz proizvoda gdje je moguće pregledati detalje o proizvodu te isti dodati u košaricu kao je moguće vidjeti iz sljedeće slike.

Sl.3.25. Stranica detalja o proizvodu

Nakon dodavanja proizvoda u košaricu bilo je potrebno kreirati stranicu za prikaz svih proizvoda u košarici, njihovo brisanje, dodavanje kao i stranicu za obavljanje kupovine. Kreirana je CartList komponenta koja prima stanje košarice i ukupnu cijenu svih proizvoda. Pomoću map funkcije pozivamo CartItem komponentu koja prikazuje detalje o pojedinom proizvodu poput slike, naziva, cijene i broja istih u košarici. Ako je košarica prazna prikazuje se da je košarica prazna. Na idućim slikama prikazan je isječak koda CartList komponente, te stranica za prikaz proizvoda u košarici.

```
export const CartList = () => {
    const { shoppingCart, cartTotal } = useContext<CartContextProps>(cartContext);

    return (
        <>
            {shoppingCart.length > 0 ? (
                <>
                    <section className="flex flex-wrap width-100">
                        {shoppingCart.map(data => {
                            return (
                                <CartItem
                                    id={data.id}
                                    name={data.name}
                                    price={data.price}
                                    quantity={data.quantity}
                                    imageUrl={data.imageUrl}
                                    key={data.id}
                                />
                            );
                        })}
                    </section>
                    <p className="text-2xl">TOTAL : ${cartTotal.toFixed(2)}</p>
                </>
            ) : (
                <p>LIST EMPTY</p>
            )}
        </>
    );
```

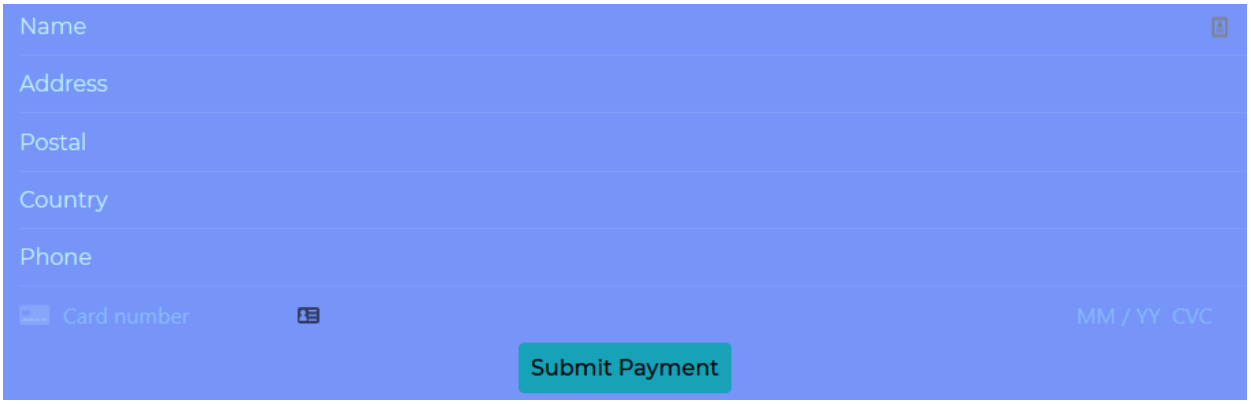
Sl.3.26. CartList komponenta



Sl.3.27. Prikaz sadržaja košarice i ukupne cijene

Za plaćanje korišten je Stripe API te se za njegovo korištenje potrebno registrirati kako bi se dobili API ključevi. Prvi ključ je javan i on služi za kreiranje tokena koji sadrži generirane podatke o kreditnoj kartici. Podaci sadrže vrstu kartice, generirani ključ koji se koristi umjesto broja kartice zbog zaštite podataka, datum isteka kartice i zadnje 4 znamenke. Drugi ključ je tajni ključ i potrebno ga je zaštititi jer se njime može obavljati plaćanje.

Za plaćanje se ne koristi vlastiti server već su alternativa Firebase funkcije koje omogućuju obavljanje backend dijela bez potrebe za kreiranjem vlastitog servera. Kreirana je forma za unos podataka s kartice i detaljima o korisniku poput imena, prezimena i adrese.



Sl.3.28. Forma za unos podataka s kartice i detalje korisnika

Nakon kreirane forme bilo je potrebno implementirati plaćanje. Za potrebe ovoga rada, koristi se testni API ključ uz kojega dolazi set kartica za testnu kupnju. Prvo je potrebno u konzolu upisati naredbu *firebase login:ci,* a zatim unijeti iduću naredbu :

```
firebase functions:config:set stripe.secret=<YOUR STRIPE SECRET KEY>
```

Nakon znaka jednakosti potrebno je unijeti vlastiti ključ koji se dobije nakon registracije.

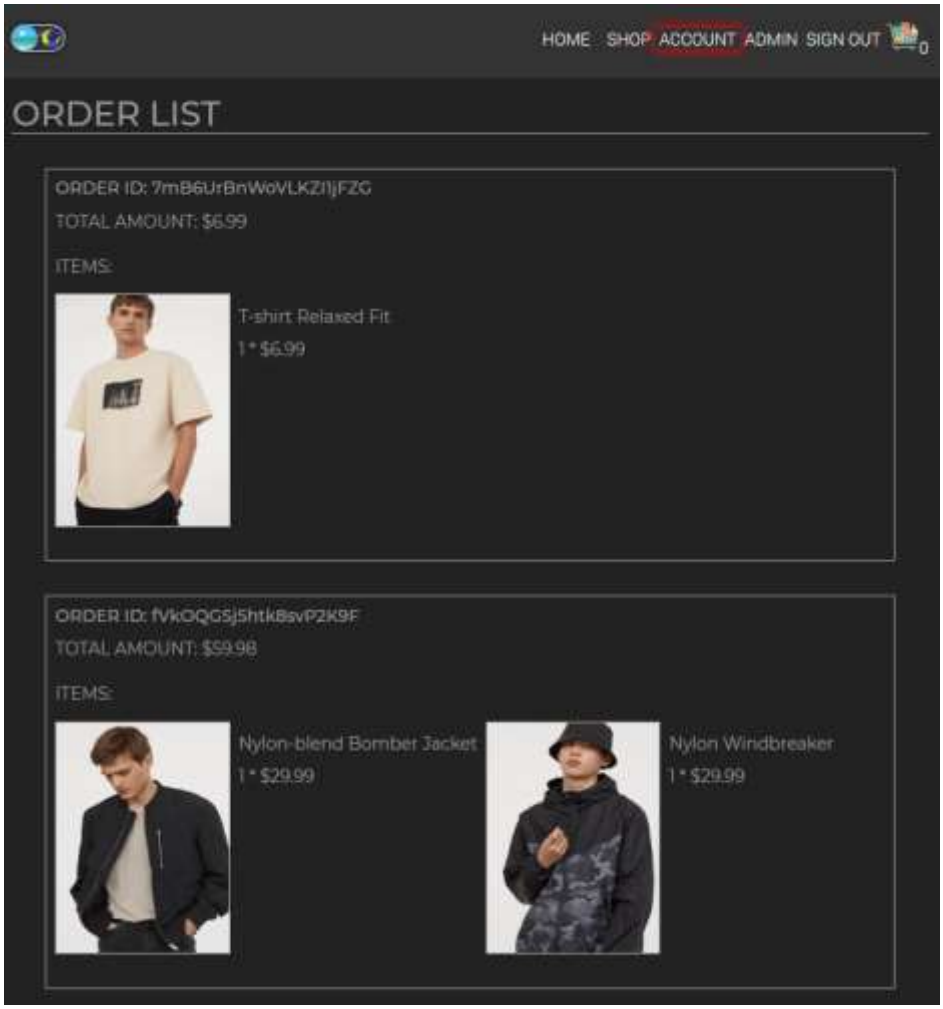
Za izvršavanje plaćanja potrebno je implementirati firebase funkciju te deployati naredbom *firebase deploy –only functions.*

U nastavku je prikazan isječak napisane funkcije kojoj su glavni dijelovi prosljeđivanje generiranog tokena kartice, iznos i valuta. Potrebno je napomenuti kako Stripe API prihvaća podatke u najmanjem mogućem iznosu plaćanja. Primjerice ako se koristi američki dolar koriste se centi, što znači da je potrebno iznos u dolarima pomnožiti sa 100. Ako se koriste valute koje nemaju decimalnu točku, iznos se ne množi. Aplikacija koristi američke dolare te se iznos množi sa 100 u liniji 25.

```
export const createStripeCharge = functions.firestore
    .document('/users/{userId}/payments/{paymentId}')
    .onCreate((snap, event) => {
        const payment = snap.data();
        const { userId, paymentId } = event.params;

        if (!payment || payment.charge) return null;

        return admin
            .firestore()
            .doc(`/users/${userId}`)
            .get()
            .then(snapshot => {
                snapshot;
            })
            .then(() => {
                const amount = Number((payment.amount * 100).toFixed(2));
                const idempotencyKey = paymentId;
                const source = payment.token.id;
                const customer = payment.customer_id;
                const charge = { amount, currency, customer, source };

                return stripe.charges.create(charge, { idempotencyKey });
            })
            .then(charge => {
                admin
                    .firestore()
                    .collection('/users')
                    .doc(userId)
                    .collection('payments')
                    .doc(paymentId)
                    .set(
                        {
                            charge: charge,
                        },
                        { merge: true },
                    )
                    .catch(error => {
                        console.log(error);
                        return error;
                    });
            });
    });
```

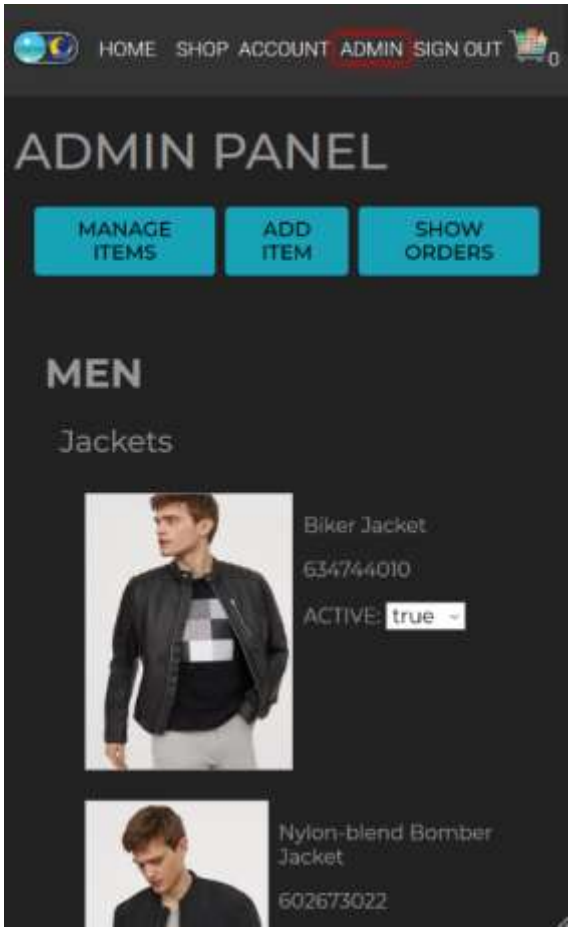
Sl.3.29. Firebase funkcija za izvršavanje plaćanja

Nakon implementacije kupnje potrebno je implementirati mogućnost prikaza svih kupovina. Izrađena je stranica i njezine potkomponente koje primaju i prikazuju potrebne podatke. Dovršena stranica prikazana je na idućoj slici.



Sl.3.30. Popis svih kupovina i njihov iznos

Nakon izvršenih kupnji potrebno je izraditi administratorsko sučelje za upravljanje trgovinom. Kreirane su tri komponente s tri gumba tipa toggle da se mogu prikazati i skrivati po želji. Prva komponenta prikazuje sve proizvode u trgovini i sadrži dropdown meni za odabir željenog stanja proizvoda, odnosno želi li se proizvod prikazati u trgovini ili ne.

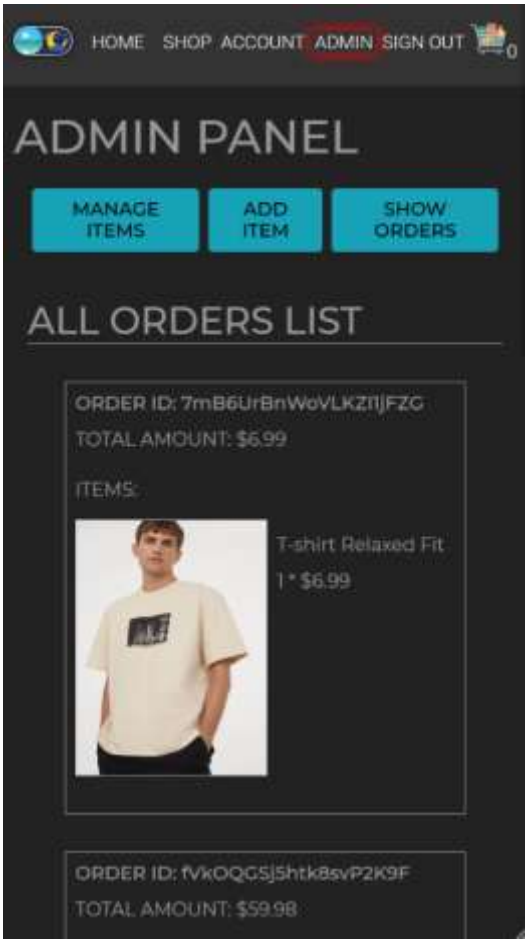


Sl.3.31. Sučelje za upravljanje proizvodima

Druga komponenta ima mogućnost dodavanja proizvoda u trgovinu. Unese se naziv glavne kategorije, potkategorije i ostali podaci o proizvodu poput naziva, cijene, slike i ostalo.

Sl.3.32. Sučelje za dodavanje proizvoda

Na kraju je bilo potrebno izraditi sučelje za prikaz svih kupovina. Iskoristilo se prethodno sučelje za prikaz svih kupovina uz sitne izmjene poput labele stranice.



Sl.3.33. Sučelje za prikaz svih kupovina

5. ZAKLJUČAK

U ovome radu detaljno je opisan tijek izrade React web aplikacije, konkretno web trgovine. Aplikacija je izgrađena od nule. Izrađen je vlastiti CSS okvir i optimiziran pomoću PostCSS-a i različitih dodataka. Aplikacija je pomno strukturirana s velikom pažnjom na arhitekturu kako bi istu bilo jednostavno nadograđivati i održavati. Izrađene su nanovo iskoristive komponente koje su se koristile na više stranica u aplikaciji. Kako bi se baza podataka zaštitila od neovlaštenog pristupa, napisana su Firebase pravila za pristup. Aplikacija je na kraju deployana te joj je moguće pristupiti putem web preglednika. Postavljen je GitHub workflow tako da se pri svakom mergeanju u master branch aplikacija automatski redeploya na Firebase hosting. Za izradu slične aplikacije potrebno je imati iskustva u React-u i Context API-ju, Stripe API-ju te Firebase-u i pripadajućim servisima.

LITERATURA

[1] React Documentation – A JavaScript library for building user interfaces, Facebook Inc., 2020., dostupno na: https://reactjs.org/ [21.4.2020.]

[2] What is Typescript? - Typescript Documentation, Microsoft Inc., 2020.,

dostupno na: https://www.typescriptlang.org/docs [24.4.2020.]

[3] HTML Mozilla MDN, Mozilla and individual contributors, 2020.,

dostupno na: https://developer.mozilla.org/en-US/docs/Web/HTML [15.5.2020.]

[4] CSS Docs - Mozilla MDN, Mozilla and individual contributors, 2020.,

dostupno na: https://developer.mozilla.org/en-US/docs/Web/CSS [15.5.2020.]

[5] PostCSS Documentation, Nikolay Latyshev, 2020.,

dostupno na: https://postcss.org/ [15.5.2020.]

[6] Firebase Documentation, Google Inc., 2020.,

dostupno na: https://firebase.google.com/docs [20.5.2020.]

[7] Stripe API Documentation, Stripe Inc., 2020.,

dostupno na: https://stripe.com/docs/api [25.5.2020.]

[8] Visual Studio Code Docs, Microsoft Inc., 2020.,

dostupno na https://code.visualstudio.com/docs [25.5.2020.]

[9] Create React App Docs, Facebook Inc., 2020.,

dostupno na: https://create-react-app.dev/ [30.5.2020.]

[9] React Hooks, Facebook Inc., 2020.,

dostupno na: https://reactjs.org/docs/hooks-reference.html [24.4.2020.]

SAŽETAK

U ovome radu detaljno je opisan tijek izrade React web aplikacije, konkretno web trgovine. Izrađen je vlastiti CSS okvir i optimiziran pomoću PostCSS-a i različitih dodataka. Aplikacija je pomno strukturirana s velikom pažnjom na arhitekturu kako bi istu bilo jednostavno nadograđivati i održavati. Kako bi se baza podataka zaštitila od neovlaštenog pristupa, napisana su Firebase pravila za pristup. Postavljen je GitHub workflow tako da se pri svakom mergeanju u master branch aplikacija automatski redeploya na Firebase hosting. Za izradu slične aplikacije potrebno je imati iskustva u React-u i Context API-ju, Stripe API-ju te Firebase-u i pripadajućim servisima.

Ključne riječi: Firebase, PostCSS, React, Stripe, Typescript, Visual Studio Code

ABSTRACT

Web store made with React

This paper describes in detail the process of creating a React web application, specifically a web store. Own CSS framework was created and optimized using PostCSS and its various plugins. The application is carefully structured with great attention to architecture to make it easy to upgrade and maintain. To protect the database from unauthorized access, Firebase access rules were written. The GitHub workflow is set up so that every time it merges into the master branch the application is automatically redeployed to Firebase hosting. To create a similar application, you need to have experience in the React and Context API, the Stripe API and Firebase and related services.

Key words: Firebase, PostCSS, React, Stripe, Typescript, Visual Studio Code

ŽIVOTOPIS

Matej Jakšić rođen je u Osijeku, 8. studenog 1995 godine. Pohađao je osnovnu školu „Ivan Filipović“ u Osijeku. Nakon osnovne škole, 2010. godine upisuje Prirodoslovno i matematičku gimnaziju u Osijeku koju završava 2014. godine. Obrazovanje nastavlja na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek gdje 2018. godine stječe zvanje prvostupnik inženjer računarstva te iste godine upisuje diplomski sveučilišni studij računarstva kojeg trenutno pohađa. Stručnu praksu odrađuje u tvrtki Prototyp, koja se bavi izradom korisničkih sučelja, web i mobilnih aplikacija za različite industrije poput sporta, financija, zdravstva, retail i gaminga, uz vodstvo iskusnih mentora. Za izradu ovih rješenja koriste se najmodernije tehnologije i metodologije koje prate industrijske standarde. Nakon stručne prakse ostaje raditi u navedenoj tvrtki kao student.

Vlastoručni potpis

_____

Matej Jakšić