Razvoj korisničkog sučelja pomoću React i Redux programskih biblioteka

Filip, Žmuk

Undergraduate thesis / Završni rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:389575

Rights / Prava: Attribution-NonCommercial 3.0 Unported/Imenovanje-Nekomercijalno 3.0

Download date / Datum preuzimanja: **2024-05-06**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Žmuk

RAZVOJ KORISNIČKOG SUČELJA POMOĆU REACT I REDUX PROGRAMSKIH BIBLIOTEKA

ZAVRŠNI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Žmuk

Matični broj: 43978/15–R

Studij: Informacijski sustavi

RAZVOJ KORISNIČKOG SUČELJA POMOĆU REACT I REDUX PROGRAMSKIH BIBLIOTEKA

ZAVRŠNI RAD

Mentor:

prof. dr. sc. Dragutin Kermek

Varaždin, kolovoz 2018.

Filip Žmuk

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada, te da se u izradi istoga nisam koristio drugim izvorima, osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne te prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Web je danas dostupan širokom broju ljudi, te su se aplikacije počele sve više izrađivati na webu. Potreba za razvojem web aplikacija potaknula je izradu okvira i biblioteka koje omogućuju njihov lakši i brži razvoj. Jedna od najpopularnijih takvih biblioteka je i React. React omogućuje podjelu korisničkog sučelja na manje komponente, koje se lakše izrađuju testiraju i održavaju. Komponente omogućuju izradu i najsloženijih korisničkih sučelja, ali ponekad je rad s podacima izrazito kompliciran. U tim slučajevima može se koristiti biblioteka Redux za lakše upravljanje podacima. U radu su detaljno opisane te biblioteke kao i rad s njima. Dotaknuto je i područje testiranja, te usporedba s drugim bibliotekama. Na kraju je izrađena je aplikacija pomoću tih biblioteke.

Ključne riječi: React; Redux; JavaScript; Web; korisničko sučelj

Sadržaj

1. Uvod 1
2. Web aplikacije 2
2.1. HTML 3
2.2. CSS 5
2.3. JavaScript 6
2.3.1. Osnovni elementi jezika JavaScript 7
2.3.2. Objekti i polja 8
2.3.3. Funkcije i klase 9
2.3.4. Moduli 11
2.3.5. Asinkrono izvođenje koda 12
2.3.6. Node.js 14
3. React 16
3.1. JSX 19
3.2. Komponente 23
3.3. Pomoćne biblioteke 29
3.3.1. webpack i Babel 29
3.3.2. Create React App 29
3.3.3. React Router 30
3.4. Testiranje 31
3.5. Usporedba s drugim bibliotekama 32
3.5.1. Angular 33
3.5.2. Vue.js 34
4. Redux 35
4.1. Akcije 35
4.2. Reduktor 36
4.3. Skladište 37
4.4. Asinkrone akcije 38

4.5. Upotreba s Reactom 39
4.6. Testiranje 41
4.7. Alternativne biblioteke 42
4.7.1. Flux 42
4.7.2. MobX 42
5. Projekt 43
5.1. Modeli podataka i baza podataka 44
5.2. Poslužiteljska strana aplikacije 46
5.3. Korisnička strana aplikacije 52
6. Zaključak 61
Popis literature 62
Popis slika 64
Popis tablica 65

1. Uvod

Razvojem tehnologije internet je svakim danom sve dostupniji. Zbog velikog broja korisnika i dostupnosti, web je postao dio svakodnevnog života. Web aplikacije su danas dostupne na gotovo svim pametnim uređajima, računalima, tabletima, mobilnim telefonima pa i pametnim satovima. Razvojem neke aplikacije na webu osiguravamo dostupnost izrazito velikom skupu korisnika. Međutim, velik broj različitih uređaja nosi i svoje nedostatke. Web aplikacije je potrebno prilagoditi širokom rasponu uređaja koji imaju različite veličine zaslona, brzinu interneta i snagu procesora.

Zbog sve veće potrebe za izradom web aplikacija, nastali su mnogi okviri i biblioteke, koji omogućuju njihov lakši i brži razvoj. Facebook je za potrebe svoje društvene mreže, između ostalih proizvoda, razvio programsku biblioteku React. React pojednostavljuje izradu interaktivnih korisničkih sučelja deklarativnim načinom pisanja programa, što znači da programer vodi računa što aplikacija treba raditi, a ne kako će se to ostvariti. Pomoću Reacta, web aplikaciju je moguće podijeliti u manje komponente. Komponente su napisane u JavaScriptu te mogu koristiti sve mogućnosti tog jezika, kao što su funkcije, grananja i petlje. Kako bi komponente zajedno tvorile aplikaciju, one mogu međusobno komunicirati slanjem podataka i događaja. Komponente mogu imati i svoje unutarnje podatke, ako im je to potrebno. Kako je React popularna biblioteka, za njega je napravljeno mnogo dodataka, koji uvelike olakšavaju razvoj programa [1].

Redux je biblioteka namijenjena za čuvanje stanja aplikacije. Koristi se u većim i složenijim aplikacijama, gdje je tok podataka složeniji i teško ga je implementirati samo pomoću komponenata. Kao i React, Redux je napisan u JavaScriptu. Treba napomenuti da Redux nije povezan s Reactom, pa ga je moguće koristiti s bilo kojom drugom bibliotekom [2].

U ovom završnom radu opisane su biblioteke React i Redux, te njihovo korištenje pri izradi korisničkih sučelja. Uz njih, opisane su još neke vrlo korisne biblioteke koje se često primjenjuju, kao dodaci za React i Redux ili samo kao pomoćne biblioteke. Na kraju je napravljen primjer aplikacije koja koristi React i Redux na korisničkoj strani te Node.js i PostgreSQL na poslužiteljskoj strani.

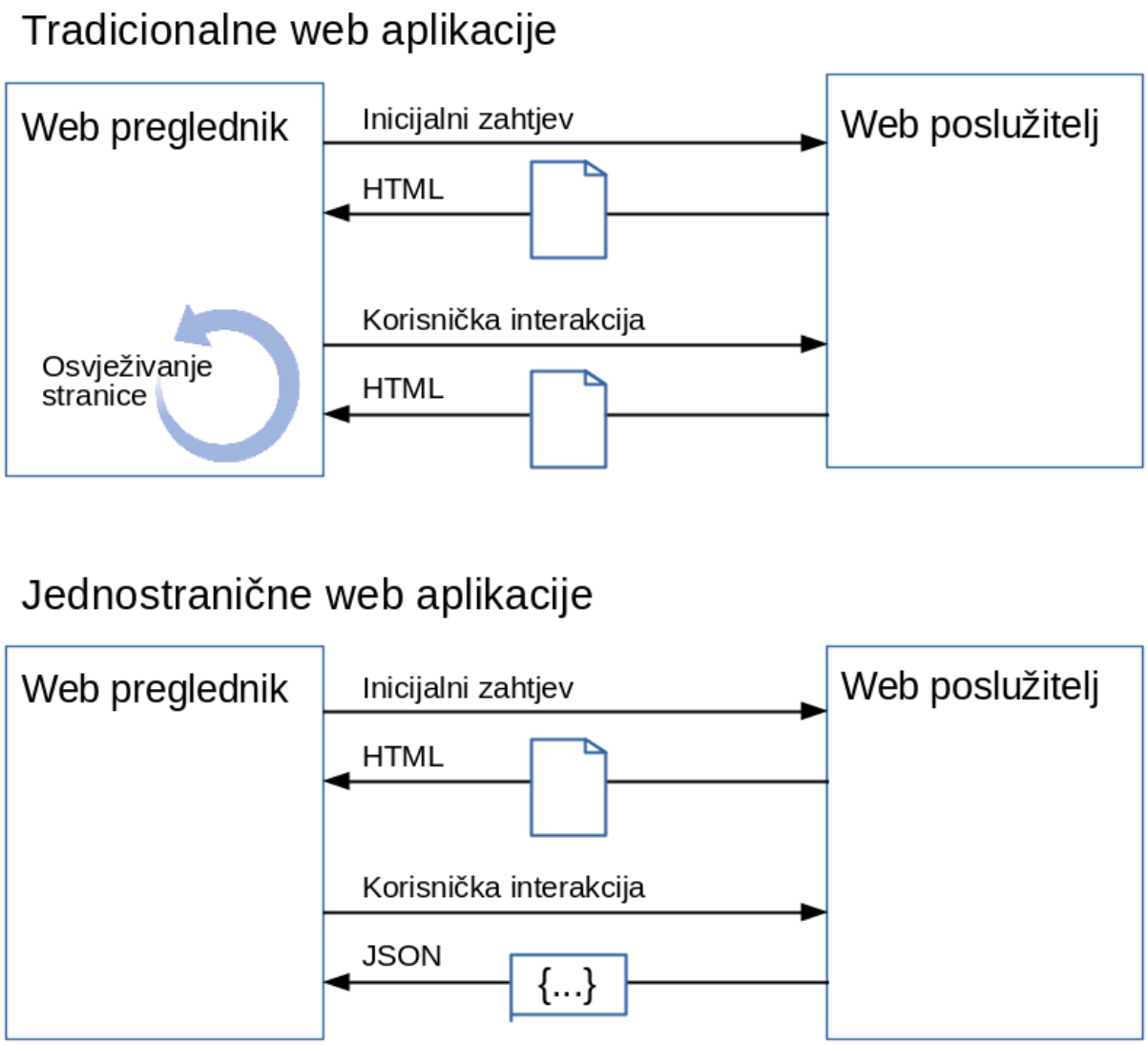
2. Web aplikacije

Web aplikacije izvršavaju se u web pregledniku. Pisane su u različitim jezicima, a u pregledniku se koriste HTML, CSS i JavaScript. Na poslužiteljskoj strani dostupan je širi izbor tehnologija, a među popularnijima su PHP, ASP.NET, Python i Ruby. Tradicionalne web stranice s poslužiteljem komuniciraju sinkrono. U sinkronoj komunikaciji klijent (web preglednik) prvo šalje zahtjev poslužitelju. Poslužitelj nakon obrade podataka, vraća odgovor klijentu, a odgovor je u ovom slučaju nova web stranica. Kada web preglednik dobije odgovor, dolazi do osvježavanja web stranice. Kako bi se spriječilo osvježavanje stranice, nastala je asinkrona komunikacija. Asinkrona komunikacija, poznata pod imenom Asinkron JavaScript i XML (eng. *Asynchronous JavaScript And XML,* kraće AJAX), omogućuje slanje zahtjeva i primanje odgovora u pozadini aplikacije. Kada klijent primi odgovor od poslužitelja, podaci koje je primio se prikazuju korisniku. Pri tome ne dolazi do osvježavanja cijele web stranice. Glavne prednosti ove tehnologije su bolja korisnička interakcija s aplikacijom, manja potrošnja podataka i brži rad aplikacije [3].

Jednostranične web aplikacije (*eng. Single Page Application*, kraće SPA) su izgrađene na način da nikada ne dolazi do potpunog osvježivanja web stranice. Nastale su kako bi se na mobilnim uređajima postiglo isto iskustvo, kakvo nastaje i pri korištenju nativne aplikacije, kao što je korištenja AJAX-a postiglo za desktop aplikacije. Aplikacija je podijeljena na komponente koje se izmjenjuju ovisno o podacima i korisnikovim akcijama. Kod inicijalnog učitavanja dohvaćaju se resursi potrebni za rad aplikacije, prvenstveno JavaScript, HTML i CSS datoteke te slike. Nakon inicijalnog učitavanja aplikacije, sva komunikacija s poslužiteljem koristi AJAX. Jednostranične web aplikacije zahtijevaju kompleksni JavaScript kod, pa su nastali okviri i biblioteke koje olakšavaju njihovu izradu [4].

Na Slici 1 je prikazana komunikacija između web preglednika i web poslužitelja. Na gornjem dijelu slike je prikazana komunikacija u tradicionalnim web stranicama. U inicijalnom zahtjevu klijent dohvaća prvu web stranicu. Iako nije prikazano na slici, tu se dohvaćaju i ostale datoteke koje su referencirane u HTML datoteci, kao što su slike, CSS stilske upute i JavaScript datoteke. Kod tradicionalnih web stranica, prilikom interakcije s korisnikom, dolazi do novog zahtjeva na poslužitelj. Poslužitelj zatim vraća novu HTML datoteku i u pregledniku se osvježava cijela stranica. Može se primijetiti kako nema većih razlika između inicijalnog učitavanja i ostalih zahtjeva, osim što su neke datoteke poput slika i CSS stilskih uputa već učitane pa se ne trebaju ponovno učitavati. Kod jednostraničnih web aplikacija, inicijalni zahtjev ne razlikuje se od zahtjeva kod tradicionalnih web stranica. Međutim, svi idući zahtjevi izvode se asinkrono. Poslužitelj ne vraća HTML datoteku, već samo podatke, najčešće u

JavaScript objektnoj notaciji (*eng. JavaScript Object Notation*, kraće JSON). Web aplikacija dalje upravlja s primljenim podacima. Prilikom toga ne dolazi do osvježavanja cijele stranice, već samo onih dijelova kod kojih je to potrebno [4].



Slika 1: Komunikacija klijenta i poslužitelja (Prema: Jadhav, Sawant i Deshmukh 2015)

2.1. HTML

HTML (*eng. HyperText Markup Language*) je jezik kojim se opisuje struktura i semantika web stranice. Prošao je kroz nekoliko iteracija, a danas je aktivna specifikacija HTML5. Na HTML-u rade dvije grupe, WHATWG (eng. *Web Hypertext Application Technology Working Group*) i W3C (*eng. World Wide Web Consortium*). WHATWG je grupa u kojoj sudjeluju proizvođači web preglednika, i u njoj je dogovoreno kako je HTML5 zadnja inačica

HTML-a, u koju se mogu postepeno dodavati nove mogućnosti. W3C standardizira HTML tako da uzima presjeke specifikacija, koje je izradila grupa WHATWG. Među najbitnijim novim mogućnostima iteracije HTML5 su novi multimedijski elementi, koji omogućavaju nativan prikaz video i audio sadržaja, kao i podršku za vektorsku grafiku korištenjem formata SVG (*eng. Scalable Vector Graphics*) [5].

HTML dijeli dokument na elemente. Elementi mogu imati neki sadržaj ili biti prazni. Prema elementima web preglednici izrađuju objektni model dokumenta (*eng. Document Object Model,* kraće DOM). DOM je zapravo stablo svih elemenata neke stranice, koje je kasnije moguće mijenjati pomoću JavaScripta. Element započinje i završava oznakom elementa. U specifikaciji HTML5, neki elementi ne moraju imati završnu oznaku, pa tada preglednik sam određuje gdje element završava. Oznake elemenata sadrže naziv elementa te dodatne atribute. Postoje generički atributi dostupni na svim elementima, ali i specifični, koji se koriste samo na nekim elementima. U modernim web aplikacijama, posebno su važni WEB-ARIA (*eng. Web - Accessible Rich Internet Applications*) atributi. Oni omogućavaju korisnicima s poteškoćama bolju pristupačnost web aplikacija. Posebno su važni u interaktivnim aplikacijama, gdje čitač zaslona teško može odrediti kako će se aplikacija ponašati i čemu služe pojedini elementi. U primjeru je prikazan osnovni kostur HTML dokumenta, zajedno s jednostavnim sadržajem. Element za odlomke `p` sadrži atribut `role`, koji je iz skupine WEB-ARIA i označava ulogu elementa [5].

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Naslov stranice</title>
</head>
<body>
    <p role="main"> Pozdrav svijetu. </p>
</body>
</html>
```

Dobra semantika znači da elementi dobro opisuju sadržaj. Iako većini korisnika to nije direktno vidljivo, semantika pomaže korisnicima s poteškoćama, koji koriste čitače zaslona, i tražilicama. Bolja semantika se postiže korištenjem elemenata namijenjenih toj svrsi. Primjer toga je korištenjem elementa `nav`, umjesto generičkog elementa `div` za odjeljak, ako taj odjeljak služi za navigaciju, ili korištenje elemenata za naslove primjerene razine. Dodatna poboljšanja se postižu korištenjem WEB-ARIA atributa [5].

2.2. CSS

CSS (*eng. Cascading Style Sheets*) je jezik pomoću kojeg se opisuje izgled HTML dokumenta u pregledniku. Važno je napomenuti kako CSS ne može nadomjestiti semantiku, pa iako stranica može izgledati isto korištenjem CSS-a, važno je upotrijebiti odgovarajuće HTML oznake za opis sadržaja. Trenutna inačica CSS-a nosi oznaku CSS3. Kao i kod HTML-a, u CSS se postepeno dodaju nove mogućnosti. Međutim, kako je CSS3 vrlo kompliciran, standard se ne nalazi u jednom dokumentu, već je podijeljen na module koji imaju svoje iteracije. Za izradu standarda zadužena je radna skupina koju je utemeljila organizacija W3C. U CSS3 dodane su mnoge nove mogućnosti, kao što su animacije i novi načini za razmještaj elemenata, te upiti o mediju (eng. media query). Upiti o mediju su omogućili responzivnost web stranice, tj. prilagođavanje web stranice web pregledniku. U primjeru je prikazan upit o mediju, koji provjerava je li širina zaslona veća od 320 piksela. Ako je tako, prikazuju se stilovi u vitičastim zagradama [5].

```
@media screen and (min-width: 320px) {
      /* stilovi koji se prikazuju */
}
```

CSS se može pisati unutar HTML datoteke korištenjem `style` elementa ili `style` atributa, međutim najčešće se CSS piše unutar zasebne datoteke, koja se povezuje s HTML datotekom korištenjem link elementa, kao što je prikazano u sljedećem primjeru.

```
<link rel="stylesheet" href=""/>
```

Stilske upute se pišu tako da se najprije napiše selektor, kojim se odabire na koje se to elemente žele primijeniti stilske upute, a zatim se u vitičaste zagrade pišu stilske upute. Upiti imaju mnogo mogućnosti pa mogu biti veoma složeni, a u primjeru je prikazan najjednostavniji, koji dohvaća element p. Stilske upute mijenjaju boju pozadine elementa i boju teksta [5].

```
p {
      color: #EEEEEE;
      background-color: #333333;
}
```

Kako je specifikacija podijeljena u module koji imaju svoje iteracije, web preglednici često zaostaju u implementaciji. Web preglednici pokušavaju unaprijed implementirati neke mogućnosti, prije nego je standard gotov. Kako bi se izbjeglo različito ponašanje u različitim inačicama preglednika, proizvođači koriste prefikse za mogućnosti koje još nisu potpuno

implementirane. Najčešće korišteni prefiksi su: `-moz-` (Mazila Firefox), `-webkit-` (Google Chrome i Safari), `-ms-` (Internet Explorer i Edge). U primjeru je prikazana upotreba prefiksa. Kako su stilovi kaskadni, preglednik zanemaruje stilove koje ne podržava, a upotrebljava se zadnji stil koji podržava. Prefiksi se mogu odnositi na vrijednost i stilsku osobinu [5].

```
div {
      display: -webkit-flex;
      display: -moz-box;
      display: -ms-flexbox;
      display: flex;
}
```

Za CSS danas postoje kosturi i preprocesori koji omogućuju lakšu izradu stranica. CSS kostur je skup stilova, koji se vrlo često ponavljaju među stranicama, kao što su klase za raspored elemenata na stranici. Jedan od popularnijih je Twitterov Bootstrap koji nudi raspored elemenata, tipografiju, razne obrasce i komponente koje rade s JavaScriptom. CSS preprocesori su programi koji prevode neki svoj jezik u CSS. Primjer takvih jezika su LESS i Sass. Ti jezici nude napredne mogućnosti koje ne postoje u CSS-u, poput ugnježđivanja stilova, funkcija i varijabli. Zatim, moguće je automatsko dodavanje prefiksa spomenutih u prethodnom odjeljku kod stilova kod kojih je to potrebno. Nakon što je kod u preprocesorskom jeziku napisan, on se prevodi i šalje web pregledniku u obliku običnog CSS-a kojeg preglednik razumije [5].

2.3. JavaScript

Uz HTML i CSS, JavaScript je najpopularniji jezik u web aplikacijama. Kako HTML i CSS ne mogu sami mijenjati svoj sadržaj i oblik, JavaScript je potreban za dinamičko ponašanje aplikacije u pregledniku. Pomoću platforme Node.js, JavaScript se može koristiti i na poslužiteljskoj strani. JavaScript se temelji na specifikaciji ECMAScript, koja je objavljena u standardu ECMA-262. Standard se s vremenom mijenja i nadopunjuje. Šesta specifikacija poznatija pod nazivom ECMAScript 6 ili ECMAScript 2015, objavljena je šest godina nakon prijašnje specifikacije i donosi mnoge promjene u jeziku. Nakon ECMAScript 6, nova specifikacija objavljuje se svake godine, ali donosi znatno manje promjene u jeziku. Treba napomenuti kako ipak postoje razlike između specifikacije ECMAScript i implementacije JavaScript jezika, kojeg poznaju web preglednici i Node.js. Razlike nastaju jer preglednici ne mogu odmah biti ažurirani na zadnju specifikaciju. Ažuriranje se provodi postepeno, a neki ih korisnici kasnije nadograđuju. Zbog toga trebamo voditi računa o tome koje će preglednike koristiti korisnici web aplikacije. Ako se žele koristiti novije mogućnosti jezika, može se poslužiti programima za prevođenje u stariju specifikaciju [6].

2.3.1. Osnovni elementi jezika JavaScript

JavaScript je temeljen na programskom jeziku C, pa je većina osnovnih elemenata slično kao i u C-u. Neki vrlo slični elementi su matematičke operacije, blokovi naredbi, grananja i petlje. Varijable se mogu deklarirati pomoću ključnih riječi `var` i `let`, a konstante pomoću `const`. Razlika između `var` i `let` je ta da su varijable deklarirane pomoću `var`, dostupne i izvan bloka u kojem su deklarirane. Blokovi se označavaju pomoću vitičastih zagrada. Slijedi primjer deklaracije varijabli i bloka. Komentari su u JavaScriptu označeni s `//`.

```
{ // Početak bloka
      var a = 'Pozdrav svijetu!';
      let b = 10;
} // Kraj bloka
console.log(a); // Naredba ispisuje poruku "Pozdrav svijetu!"
console.log(b); // Naredba izbacuje pogrešku
```

Grananje se može ostvariti korištenjem `if-else` naredbi ili pomoću `switch-case`. Razlika je u tome što `if-else` može provjeriti samo jedan uvjet, dok ih `switch-case` može provjeriti više odjednom. Slijede primjeri `if-else` i `switch-case` grananja [6].

```
if (a > b) {
      console.log('a je veće od b');
} else {
      console.log('b je veće od a');
}

switch(rijec)
{
      case 'pozdrav':
            console.log('Riječ je "pozdrav".');
            break;
      case 'svijetu':
            console.log('Riječ je "svijetu".');
            break;
      default:
            console.log('Riječ nije ni "pozdrav" ni "svijetu".');
            break;
}
```

Petlje se koriste kada je neki blok koda potrebno ponoviti više puta. U JavaScriptu su dostupne petlje `for`, `while` i `do-while`, koje se ponašaju kao i u programskom jeziku C. Međutim, JavaScript ima specifične petlje koje prolaze kroz sve elemente nekog objekta ili polja. Primjer takve petlje je `for-of` petlja. U pozadina ta petlja koristi iteratore. Iteratori su objekti koji sadrže metodu `next` koja vraća vrijednost sljedećeg elementa. Sva polja, objekti, mape i skupovi imaju svoje iteratore, a ako je potrebno, mogu se izraditi pomoću generatorske funkcije. Slijedi primjer klasične `for` petlje, a zatim `for-of` petlje koja prolazi kroz sve elemente nekog polja i ispisuje ih. Vidljivo je da je `for-of` petlja preglednija te je potrebno napisati manje linija koda [6].

```
//klasična for petlja
for (let i = 0; i < polje.length; i++) {
      console.log(polje[i]);
}
// for-of petlja
for (let element of polje) {
      console.log(element);
}
```

2.3.2. Objekti i polja

Objekti i polja su složeni tipovi podataka podržani u JavaScriptu. U njih je moguće spremiti ostale tipove podataka. Rad s poljima i objektima je vrlo sličan. Iz oba tipa podataka se dohvaća vrijednost pomoću ključa. Kod polja, ključ je pozitivna cjelobrojna vrijednost, dok je kod objekta to niz znakova. U primjerima koji slijede prikazan je postupak stvaranja novih polja i objekata.

```
const polje1 = []; //prazno polje
const objekt1 = {}; //prazan objekt
const polje2 = ["pozdrav", "svijetu"]; // polje s dvije vrijednost
const objekt2 = {
      "ime": "Ariana",
      "fakultet": "FOI"
}; // objekt s ključevima "ime" i "fakultet"
```

Nakon što su napravljeni objekti i polja, u njih je moguće dodavati elemente, brisati ih ili mijenjati. Element se može dohvatiti korištenjem ključa na sljedeći način:

```
console.log(objekt2.ime);
console.log(objekt2["fakultet"]);
console.log(polje2[1]);
```

Na isti se način mogu izmijeniti postojeći elementi. Novi elementi se mogu dodati korištenjem novog ključa ili korištenjem metode `push` kod polja [6].

Objekti se mogu prikazati korištenjem JSON formata. Objekt, koji je tako prikazan, može sadržavati brojčane podatke, logičke vrijednosti, nizove znakova, polja i druge takve objekte. JSON se vrlo lako pretvara u niz znakova, te se niz znakova lako pretvara u JavaScript objekt. JSON zapis objekta je vrlo bitan jer se podaci kod web aplikacija najčešće šalju u tom formatu [4][6].

Kod polja i objekata je važno napomenuti kako se šalju preko reference. Dakle, ako funkciji proslijedimo objekt ili polje, funkcija može izmijeniti taj objekt ili polje. U ECMAScript 6 dodan je novi operator, koji omogućuje izradu novih objekata i polja uz zadržavanje starih vrijednosti. Radi se o operatoru širenja, koji se zapisuje pomoću tri točke. Operator širenja omogućuje da se polje ili objekt podijeli na vrijednosti koje sadrži. Zatim je moguće napraviti novi objekt ili polje iz tih vrijednosti. Važno je napomenuti kako tako nastaje novi objekt, samo od glavnog objekta. Ako polje i objekt sadrže druge objekte i polja, oni su spremljeni preko reference koja ostaje nepromijenjena [6].

```
const polje3 = [...polje2, "Pozdrav svijetu"]; // dodavanjeelementa na kraj novog polja
const polje4 = [...polje3]; // stvaranje novog polja od postojećeg
const objekt3 = {...objekt2,
"ime": "Paula"
}; //stvaranje novog objekta s različitom vrijednošću za ključ "ime"
```

Ponekad je korisno elemente iz polja i objekata spremiti u varijable. Taj postupak se naziva destrukturiranje. U primjeru je prikazana sintaksa destrukturiranja [6].

```
let {ime, fakultet} = objekt2;
console.log(ime); //ispisuje "Ariana"
console.log(fakultet); //ispisuje "FOI"
let [prvaRijec, drugaRijec]= polje2;
console.log(prvaRijec+" "+drugaRijec); //ispisuje "pozdrav svijetu"
```

2.3.3. Funkcije i klase

Funkcije su važan dio većine programskih jezika, pa tako i JavaScripta. U JavaScriptu, funkcije su objekti koji se dohvaćaju pomoću reference, pa se tako mogu spremiti u varijablu te se s njima može raditi sve što je moguće i s varijablama. Funkcija ima dvostruku namjenu. Može se ponašati kao funkcije u drugim jezicima. Tada može primati parametre i vraćati neku vrijednost. Međutim, ako se pozove uz ključnu riječ `new`, tada funkcija ima ulogu konstruktora.

Funkcija i dalje može primati parametre, ali sada vraća instancu funkcije, odnosno objekt. To ponašanje je slično klasama u drugim programskim jezicima. Funkciju je moguće deklarirati korištenjem ključne riječi `function` ili korištenjem strelice. Funkcija deklarirana pomoću strelice (*eng. arrow function*) ne stvara novi blok i nema svoj prototip. To znači da ne može biti pozvana s `new` i koristi blok varijabli u kojemu je deklarirana. [6] U primjeru su deklarirane tri funkcije. Funkciji `Student` napravljena je instanca i pozvana je njezina metoda. Kao što je pokazano, funkciji je moguće dodijeliti metode korištenjem prototipa.

```
const dodaj = (a, b) => a + b;
function oduzmi (a, b) {
	return a - b;
}
const Student = function(ime, prezime) {
	this.ime = ime;
	this.prezime = prezime;
}
Student.prototype.predstaviSe = function() {
	console.log("Bok, ja sam " + this.ime + ".");
}
let marko = new Student("Marko", "Markovic");
marko.predstaviSe(); // ispisuje "Bok, ja sam Marko. "
```

Klase su dodane tek u specifikaciji ES6. Klase u JavaScriptu ne donose nikakve nove funkcionalnosti naspram funkcija. Čak i ako se pozove operator `typeof` nad nekom klasom, dobit će se odgovor "`function`", odnosno funkcija. Glavna prednost korištenja klasa, umjesto konstruktorskih funkcija, je preglednost koda. Klase omogućuju i lakše otkrivanje nekih sintaktičkih pogreška jer će, za razliku od funkcija, izbaciti pogrešku ako se zaboravi ključna riječ `new`. Slijedi primjer korištenja klasa.

```
class Osoba {
	constructor(ime) {
		this.ime = ime;
	}
}

class Student extends Osoba {
	constructor (ime, fakultet) {
		super(ime);
		this.fakultet = fakultet;
	}
```

```
	ispisiPodatke() {
		console.log(this.ime + " studira na: " + this.fakultet);
	}
}

let marko = new Student("Marko", "FOI");
marko.ispisiPodatke();
```

U primjeru je korišteno nasljeđivanje klasa. Klasa `Student` specificira klasu `Osoba`, korištenjem ključne riječi `extends`. U klasi `Student` dostupna su sva svojstva i metode klase `Osoba`. Konstruktor klase `Osoba` dostupan je i u klasi `Student` pod nazivom `super` [6].

2.3.4. Moduli

Bez korištenja modula svi deklarirani elementi su dostupni svugdje. U složenijim aplikacijama to može uzrokovati probleme, jer tada svaka globalna varijabla mora imati različito ime. Korištenjem modula neku se varijablu može ograničiti samo na taj modul. Moduli se definiraju u različitim datotekama. Koristeći ključnu riječ `export`, definiraju se dijelovi koji se izvoze iz modula. Kasnije se mogu uvesti i koristiti u drugom modulu. Slijedi primjer modula iz kojeg se izvozi funkcija i konstanta.

```
export const e = 2.71;
function zbroj (a, b) {
	return a + b;
}
export zbroj;
```

Kasnije je moguće uvesti funkciju iz datoteke `primjer.js` koristeći naredbu:

```
import { zbroj } from './primjer.js';
```

Korištenjem ključne riječi `default`, može se postići podrazumijevan uvoz. Tada nije potrebno specificirati ime objekta koji se uvozi. Slijede primjeri takvog izvoza i uvoza [6].

```
// datoteka modul.js
export default function() {
	console.log('pozdrav svijetu');
}

// neka druga datoteka
import pozdrav from 'modul.js';
```

Node.js koristi drugačiju sintaksu uvoza i izvoza. Za razliku od web preglednika, svaka datoteka je tretirana kao posebni modul. Svaki modul sadrži objekt `exports`, koji služi za izvoz iz modula. Objekt `exports` se kasnije može uvesti u drugi modul korištenjem funkcije

`require`, koja se poziva s putanjom do tog modula. Funkcija `require` vraća objekt, koji je izvezen iz modula koji se uvozi.

```
// datoteka opseg.js
const PI = 3.14;
exports.opseg = (r) => 2 * PI * r ;

// druga datoteka
const opseg = require("./opseg.js");
```

ECMAScript moduli su zasad u eksperimentalnoj fazi, i vjerojatno će postati dostupni u nekom od budućih izdanje platforme Node.js [7].

2.3.5. Asinkrono izvođenje koda

JavaScript ne podržava višedretveno izvođenje programa. Ipak, kako mu je prva namjena bila primjena na webu, JavaScript podržava asinkronu interakciju s korisnikom. Asinkrono izvršavanje koda koristi petlju događaja (*eng. event loop*) i red poslova (*eng. job queue*). Kada se dogodi asinkroni događaj, on se stavlja na kraj red poslova. Petlja događaja je proces koji upravlja redoslijedom izvršavanja koda, tako da izvršava poslove od prvog do posljednjeg. Najjednostavniji naći pisanja asinkronih događaja je korištenjem funkcija povratnog poziva (*eng. callback function*). Tada se nekoj asinkronoj funkciji, kao parametar, prosljeđuje referenca na funkciju povratnog poziva. Kada asinkrona funkcija završi s radom, pozvat će proslijeđenu funkciju. Najjednostavnija takva funkcija je `setTimeout`, koja odgađa izvršavanje proslijeđene funkcije za neki broj milisekundi. U primjeru koji slijedi prikazana je njezina upotreba. Poruke koje se ispisuju u svom sadržaju imaju zapisan svoj redoslijed. Bitno je primijetiti kako se funkcija, koja je pozvana u sredini, izvršava na kraju, upravo zbog asinkronosti izvođenja koda. Vremenski pomak je postavljen na nulu, ali funkcija dolazi na kraj reda poslova.

```
console.log("Prva poruka.");
const ispisiPoruku = () => console.log("Treća poruka");
setTimeout(ispisiPoruku, 0);
console.log("Druga poruka.");
```

Ovakvo pisanje asinkronog koda je vrlo jednostavno. Međutim brzo može postati vrlo nepregledno. To je posebno slučaj kada se neimenovane funkcije pišu direktno u pozivne parametre asinkrone funkcije. U njima je moguće koristiti druge funkcije povratnog poziva, pa kod postaje ugniježđen i još nepregledaniji. Drugi nedostatak ovakvog korištenja asinkronog koda je nemogućnost korištenja try-catch bloka. Naime, ako se dogodi pogreška unutar funkcije povratnog poziva, ona se neće uhvatiti u try-catch u kojem je pozvana asinkrona funkcija. Kako bi se riješio taj problem, platforma Node.js predlaže da je prvi parametar funkcije

povratnog poziva objekt pogreške, te su tako napisane mnoge sistemske funkcije na toj platformi [6].

Sljedeća metoda za izradu asinkronog koda su obećanja (*eng. Promise*). Glavna razlika, naspram funkcija povratnog poziva, je ta da se ne prosljeđuje referenca na funkciju asinkronoj funkciji, već asinkrona funkcija vraća obećanje. Obećanja mogu biti u tri stanja: neriješeno, ispunjeno i odbijeno. Kada je obećanje tek napravljeno, nalazi se u stanju neriješeno. Nakon što se asinkrona radnja izvrši, obećanje prelazi u stanje ispunjeno, ako se asinkrona radnja uspješno izvršila, ili u odbijeno, ako je došlo do pogreške. U sljedećem isječku koda je prikazano stvaranje obećanja.

```
new Promise ( (prihvati, odbij) => {
      // ako je dobiven neki rezultat
      prihvati("Obećanje je izvršeno.");
      // ako je došlo do pogreške
      odbij(new Error("Pogreška. "));
});
```

Objekt obećanja se može izraditi pomoću funkcije, koja prima funkcije povratnog poziva. Korištenjem konstruktora `Promise` može se izraditi neriješeno obećanje. Za izradu riješenih obećanja dostupne su metode `Promise.resolve()` za ispunjeno obećanje i `Promise.reject()` za odbijeno obećanje. Njih se poziva s povratnom vrijednošću za ispunjeno obećanje, odnosno s objektom pogreške za odbijeno. Nakon što je obećanje riješeno, može se dobiti rezultat korištenjem metode `then`, odnosno pogreška korištenjem metode `catch`. U primjeru je prikazano slanje zahtjeva na poslužitelj i ispis koda statusa odgovora.

```
fetch("primjer.com")
      .then(odgovor => console.log(odgovor.status))
      .catch(pogreska => console.log(pogreska));
```

Metode `then` i `catch` vraćaju novo obećanje pa ih je moguće definirati ulančano. Ako dođe do bilo kakve pogreške u lancu obećanja, izvršit će se metoda `catch`. Ako nije definirana povratna vrijednost u metodi `then`, tada se vraća ispunjeno obećanje bez vrijednosti. Vrijednost se može postaviti tako da se vraća na neku vrijednost iz funkcije. Prava snaga obećanja dolazi do izražaja kada funkcija unutar `then` vrati novo obećanje. Takav slučaj je prikazan u sljedećem primjeru.

```
fetch("primjer.com")
      .then(odgovor => odgovor.json())
      .then(json => console.log(json))
      .catch(pogreska => console.log(pogreska));
```

U osmom izdanju standarda ECMAScript (ECMAScript 2017), dodan je još jedan način rada s obećanjima korištenjem `async` i `await`. S `async` se označava da je neka funkcija asinkrona prilikom deklaracije. `Await` se koristi kada se želi da se neko obećanje riješi i vraća se vrijednost tog obećanja, odnosno pogreška ako je obećanje odbijeno. Sljedeći primjer je funkcionalno identičan prethodnome, ali koristi noviju sintaksu [6].

```
const ispisiOdgovor = async () => {
      try {
            let odgovor = await fetch("primjer.com")
            let json = await odgovor.json();
            console.log(json);
      } catch (pogreska) {
            console.log("Došlo je do pogreške");
      }
}
ispisiOdgovor();
```

2.3.6. Node.js

Node.js je platforma koja omogućuje izvršavanje JavaScript programa izvan web preglednika. Vrlo često se koristi za izradu poslužiteljske strane aplikacije. Na ovoj platformi su napravljeni i mnogi programi koji omogućuju lakšu izradu aplikacija. Za izvršavanje JavaScripta koristi se Chromeov JavaScript stroj V8. Node.js je, pod upravom organizacije Node.js Foundation, osnovan 2015. godine, više od pet godina nakon izrade same platforme. U toj organizaciji sudjeluju neke od najvećih tehnoloških tvrtki danas, poput IBM-a, Microsofta, Googlea i Intela. U izradi Node.js-a je sudjelovalo više od 1 600 ljudi [7].

Kako koristi JavaScript, dostupno je isto asinkrono izvršavanje koda i upravljanje događajima kao i u web pregledniku. Node.js je dizajniran tako da može istovremeno obrađivati velik broj zahtjeva, te da se može skalirati ovisno o potrebama. Platforma potiče izradu neblokirajućih programa, što znači da ako postoji neka vremenski zahtjevna funkcija, ona ne smije blokirati ostatak izvođenja programa. Najčešće se to postiže upotrebom funkcija povratnog poziva. Pod vremenski zahtjevnim funkcijama, uglavnom se smatraju ulazno izlazne operacije, poput čitanja ili pisanja na disk, slanje upita na bazu, te slanje i dohvaćanje podataka s nekog stranog poslužitelja. U pozadini programa moguće je da proces ima više dretvi, no to nije vidljivo iz pozicije programera. Iako je platforma dizajnirana bez dretvi, moguće ih je napraviti unutar programa. U sljedećem primjeru je prikazan najjednostavniji program koji na sve zahtjeve šalje istu poruku.

```
const http = require("http"); // uvoz modula
```

```
const server = http.createServer((req, res) => {
      res.statusCode = 200;
      res.setHeader("Content-Type", "text/plain");
      res.end("Pozdrav svijetu \n");
});

server.listen(80, "127.0.0.1", () => {
      console.log("Poslužitelj je pokrenut");
});
```

Može se primijetiti kako je jezik gotovo jednak kao i u pregledniku. Glavna razlika u ovom primjeru je način na koji se uvoze moduli [7].

Node.js ne bi bio toliko uspješan da ne postoji veliki raspon dostupnih paketa. Paketi se uglavnom dohvaćaju upotrebom NPM-a (*eng. Node Package Manager*), iako postoje i druge opcije. Glavni repozitorij NPM-a danas ima više od 700 000 raznih paketa. Pomoću sljedeće naredbe u terminalu dohvaća se neki paket u projekt.

```
npm install <ime paketa> --save
```

Tako dohvaćeni paketi spremaju se u mapu node_modules, a podaci o njima u datoteku package.json. Pakete je moguće globalno dohvatiti te tada oni postaju dostupni u cijelom operacijskom sustavu. NPM je postao toliko važan i neophodan za razvoj Node.js aplikacija, da danas dolazi u instalaciji zajedno s tom platformom [8].

3. React

React je biblioteka namijenjena izradi pogleda kod korisničkih sučelja aplikacija. Nastala je za potrebe Facebooka, a i dalje najviše doprinosi razvoju ovog projekta. Kako je React biblioteka otvorenog koda, izvorni kod dostupan je na GitHubu i licenciran je pod MIT licencom, što znači da je slobodan za privatnu i komercijalnu upotrebu te su dozvoljene modifikacije. U trenutku pisanja ovog rada, aktivno je šesnaesto izdanje biblioteke. React je jedna od najpopularnijih biblioteka za razvoj korisničkih sučelja s preko 100 000 zvjezdica na GitHubu. Može se koristiti za poboljšanje interaktivnosti statičkih web stranica ili za izradu kompliciranijih jednostraničnih web aplikacija. Jednostranične web aplikacije mogu u cijelosti biti izrađene u Reactu, ali se često koriste biblioteke za upravljanje podacima aplikacije, pogotovo kod složenijih aplikacija [1].

Za izradu web aplikacija potrebno je uključiti dvije biblioteke: React i React DOM. To se najčešće postiže instalacijom paketa pomoću NPM-a. U rjeđim slučajevima, biblioteke se mogu preuzeti s mreže za dostavu sadržaja (*eng. content delivery network*, kraće CDN), ali tada nisu dostupne neke napredne mogućnosti. Za korisnike CDN-a, napravljene su odvojene biblioteke za razvoj i gotovu aplikaciju. Razlog odvojenosti biblioteka React i React DOM je mogućnost korištenja Reacta na platformama koje nisu web. U kombinaciji s bibliotekom React Native, React se koristi za izradu mobilnih aplikacija. U biblioteci React smještene su zajedničke funkcionalnosti, a u bibliotekama React DOM i React Native funkcionalnosti vezane uz platformu na kojoj će se aplikacija izvršavati. React je moguće koristiti s dijelovima izrađenim u drugim bibliotekama. Primjerice, ako postoji neka funkcionalnost napravljena pomoću biblioteke jQuery, može se upotrijebiti unutar React web aplikacije. Tako je omogućen postupan prijelaz na React kod postojećih web aplikacija [1].

React koristi deklarativan način pisanja programa. Kod deklarativnog programiranja, programer opisuje što program mora raditi, umjesto da piše korake kako će se to izvršavati, kao što je slučaj kod imperativnog programiranja. Kao primjer se može uzeti pretraživanje polja. U imperativnom programiranju koriste se petlja i varijabla, u koje se sprema je li element pronađen, dok se kod deklarativnog pristupa koristi metoda nad poljem, koja prolazi kroz elemente polja i pretražuje ih, prema uvjetu koji je proslijeđen kao parametar. Deklarativnim pristupom dobiva se kraći kod, koji je lakše razumjeti, a kao posljedica toga nastaje i kod s manje pogrešaka. Jedan od načina deklarativnog programiranja je funkcijsko programiranje. Kod funkcijskog programiranja, funkcije postaju slične matematičkim funkcijama. Funkcija ne može koristiti ni mijenjati varijable, koje su deklarirane izvan nje. Međutim, funkcija može primati parametre i vraćati rezultat, a kako koristi samo lokalne varijable za iste parametre,

uvijek vraća isti rezultat. Ovo svojstvo pojednostavljuje testiranje funkcije. Funkcija s ovakvim svojstvima, naziva se čista funkcija (eng. *pure function*). Kod izrade čistih funkcija u JavaScriptu, vrlo je bitno obratiti pažnju na polja i objekte, ako se koriste kao parametri. Oni se prenose preko reference, pa funkcija može izmijeniti polje ili objekt, koji je deklariran izvan nje te je potrebno pripaziti da ne dođe do toga. Kako bi se koristile sve pozitivne strane funkcijskog programiranja, potreban je jezik koji to podržava. U JavaScriptu je to omogućeno jer se funkcije ponašaju kao obične varijable, moguće ih je slati kao parametar ili vratiti kao rezultat funkcije. Funkcije koje uzimaju neku funkciju kao parametar te vraćaju funkciju s proširenom funkcionalnošću, nazivaju se funkcije višeg reda (eng. *Higher-order Functions*). Proširenje funkcionalnosti je moguće postići i kompozicijom funkcija. Nova funkcija se može napraviti i tako da se početnoj funkciji stave neke konstantne vrijednosti, kao parametri. Kod promatranja funkcionalnog programiranja i korisničkih sučelja, korisničko sučelje može se promatrati kao rezultat funkcije, koja prima unutarnje stanje aplikacije kao parametar [9].

Aplikacije izrađene u Reactu, za postizanje boljih performansi, koriste virtualni DOM. Za razliku od DOM-a, virtualni DOM se ne prikazuje korisniku. Kako se virtualni DOM ne prikazuje korisniku, preglednik kod njega ne mora stvoriti sliku za korisnika pa troši i znatno manje računalnih resursa. Aplikacija napisana u Reactu stvara virtualni DOM. Zatim se uspoređuju DOM i virtualni DOM, pa do promjena DOM-a dolazi, samo ako je to potrebno. Postupak pretvaranja virtualnog DOM-a u stvaran DOM, naziva se pomirenje (*eng. reconciliation*). Kako je React deklarativan, nije potrebno poznavati niti pisati algoritme za rad s virtualnim DOM-om. Međutim, iz njih proizlaze neke preporuke za pisanje aplikacija boljih performansi. Virtualni DOM nije naziv konkretne tehnologije, nego uzorak koji mogu koristiti aplikacije. Od šesnaeste inačice, biblioteke React svoj virtualni DOM ostvaruju pomoću stroja za pomirenje (*eng. reconciliation engine*) nazvanog React Fiber, koji ima u cilju poboljšanje performansi uvođenjem inkrementalnog nadograđivanja virtualnog DOM-a u DOM-u [1].

React je pisan u JavaScriptu, pa se i aplikacija može izraditi u JavaScriptu. Međutim, za korištenje Reacta je napravljen i jedan poseban jezik pod imenom JSX (*eng. JavaScript and XML*). Pomoću tog jezika se mogu lakše pisati aplikacije, jer omogućuje pisanje HTML-a direktno u komponentama. Detalji jezika JSX opisani su u sljedećem poglavlju, kao i prevođenje JSX-a u čist JavaScript. Prednost čistog JavaScripta je ta da nije potrebno prevoditi JSX u JavaScript. Najjednostavniji način pisanja manjih aplikacija je korištenje CDN-a i čistog JavaScripta. JSX se najčešće koristi u kombinaciji s webpackom. Webpack je program koji će uzeti sve datoteke koje postoje i spojiti ih u jednu. Pritom je moguće podesiti prevođenje JSX-a u JavaScript, koristiti CSS preprocesora i još mnogo toga, što je opisano kasnije u radu. Osnovni dio React aplikacije je element. Elemente je moguće jednostavno izraditi pomoću JSX-a.

```
const element = <p>Ovo je element</p>;
```

Već u ovom najjednostavnijem primjeru je vidljiva povezanost XML-a i JavaScripta u JSX-u. Važno je primijetiti kako ovaj Reactov element nije isto što i DOM element. Ako se ispiše u konzolu, dobit će se objekt sličan ovome:

```
{
      "type": "p",
      "key": null,
      "ref": null,
      "_owner": null,
      "_context": {},
      "props": {
            "children": "Ovo je element."
      }
}
```

Kao što je vidljivo, Reactov element je objekt. Objekt sadrži tip elementa, svojstva elementa i djecu elementa. U ovom jednostavnom primjeru to je samo niz znakova, ali u stvarnim aplikacijama to su vrlo često drugi elementi. Reactovi elementi predstavljaju stvarne DOM elemente, koji se trebaju prikazati korisniku, a kako se oni ne prikazuju, zapravo su virtualni DOM, koji je opisan u prethodnom odjeljku. Reactovi elementi su napisani u paradigmi funkcijskog programiranja, pa jednom kada se naprave, ne smiju se više mijenjati. Ako je potrebno promijeniti korisničko sučelje, izrađuju se novi elementi. Prethodno kreiran element, prikazuje se u web aplikaciji na sljedeći način:

```
<!--  DOM element u kojem će se element prikazati -->
<div id="app"></div>

// metoda kojom ga prikazujemo
ReactDOM.render(element, document.getElementById('app'));
```

Ako se u ovom primjeru želi doći do promjena u DOM-u, mora se ponovno pozvati metoda `render`. Kako bi se izbjegla potreba pozivanja metode `render`, i time imperativno programiralo, koriste se komponente. Komponente omogućuju izradu ponovo iskoristivih dijelova aplikacije i podjelu aplikacije na manje dijelove. Najjednostavnija komponenta je funkcija koja vraća Reactov element, a u složenijim slučajevima mogu se koristiti i klase. Slijedi primjer jednostavne funkcijske komponente.

```
const Pozdrav = () => {
      return <p>Pozdrav svijetu!</p>;
}
```

Komponenta se može prikazati korištenjem XML oznake. U primjeru ispod je pokazano prikazivanje ove komponente u web aplikaciji.

```
ReactDOM.render(<Pozdrav />, document.getElementById('app'));
```

Vidljivo je kako je prikazivanje elementa i komponente vrlo slično. Pozivanjem funkcije komponente nastaje virtualni DOM, koji se zatim može prikazati korisniku. Tip elementa može biti niz znakova, koji predstavlja oznaku elementa ili referenca na komponentu. Kako se element može sastojati od više komponenata i elemenata, mogu se i same komponente, što omogućuje izradu veoma moćnih i složenih aplikacija [1].

3.1. JSX

JSX (*eng. JavaScript and XML*) je programski jezik koji obogaćuje sintaksu JavaScripta, dodavanjem XML-a (*eng. Extensible Markup Language*). JSX nije namijenjen korištenju u web pregledniku, nego je namijenjen za razvoj aplikacija, koje kasnije preprocesori prevode u standardne JavaScript aplikacije. Najpopularniji preprocesor je Babel, za kojeg je napravljen i dodatak koji radi s Reactom. Kao i React, JSX je napravila tvrtka Facebook. JSX su kasnije preuzele i neke druge biblioteke, ali fokus ovog poglavlja je na Reactu, te kako se JSX prevodi u JavaScript namijenjen ovoj biblioteci [1].

Preprocesor prevodi XML elemente iz jezika JSX u Reactove elemente, korištenjem funkcije `React.createElement`. Zbog toga je važno u svaku komponentu uvesti biblioteku `React`, pa i onda kada njezino korištenje nije vidljivo u kodu. Ako se koristi JSX, ova se funkcija uglavnom neće direktno koristiti. Prvi parametar funkcije je tip elementa, koji će funkcija napraviti. To može biti niz znakova, ako se radi o HTML elementu ili komponenta. Preprocesori, ovisno o imenu, određuju radi li se o komponenti ili HTML elementu. Bitno je da je prvo slovo imena komponente veliko, jer će se u protivnom oznaka tretirati kao HTML element. Drugi argument funkcije `React.createElement` je objekt, koji sadrži svojstva komponente ili atribute HTML elementa, ovisno o prvom parametru. Ostali parametri funkcije su elementi djece tog elementa. Slijedi primjer prevođenja JSX elementa u JavaScript.

```
// originalni JSX
import React from 'react';
const element = <h1>Ovo je element</h1>;
// generirani JavaScript
import React from 'react';
const element = React.createElement(
  "h1",
  null,
```

```
  "Ovo je element"
);
```

Svojstva elementa postavljaju se kao atributi u XML-u. Kod HTML elemenata, to znači da je sintaksa JSX-a vrlo slična sintaksi HTML-a. Slijedi primjer poveznice, koja ima atribut `href`, te poziv funkcije `React.createElement`, nakon prevođenja [1].

```
const link = <a href="primjer.com">Poveznica</a>;
// nakon prevođenja
const link = React.createElement(
      "a",
      { href: "primjer.com" },
      "Poveznica"
);
```

Kako JSX koristi XML, postoje određene razlike između HTML-a i elemenata, koji se mogu pisati u tom jeziku. XML zahtjeva od svih elemenata da imaju završnu oznaku. Međutim, dozvoljeno je da jedna oznaka bude sama sebi početni i završni element. Kao primjer se može uzeti oznaka za novi red, koja u HTML-u nema završnu oznaku. U HTML-u bi se element napisao kao `<br>`, dok se u JSX-u koristi oznaka `<br />`. JavaScript ima svoje ključne riječi, poput `class` i `for`. Te ključne riječi mogu biti i atributi nekog HTML elementa, pa dolazi do kolizije između JavaScript ključnih riječi i HTML atributa. To je riješeno tako da su u JSX-u ti HTML atributi preimenovani. Umjesto atributa `class`, koristi se `className`, te se umjesto `for` koristi `htmlFor`. Atribut `style` bitno je različit od istog atributa kod HTML-a. Naime, u HTML-u se u njega pišu CSS upute za taj element. U JSX-u taj atribut prima objekt sa stilskim uputama. Upute u tom objektu se razlikuju po nazivima atributa. Umjesto korištenja crtice između riječi, početno slovo riječi je napisano velikim slovom. Slijedi primjer stiliziranog elementa.

```
const stilovi = {
      backgroundColor: '#555'
};
const element = <p style={stilovi}>Ovo je stilizirani element</p>;
```

Ostali HTML atributi koriste se jednako u JSX-u i HTML-u. Razlika ponekad postoji u imenu, pa ako se naziv atributa sastoji od više riječi, onda se sve riječi, osim prve, pišu velikim početnim slovom. Isti je slučaj i sa SVG atributima. JSX, za razliku od HTML-a, ignorira razmake između elemenata. Novi redovi i tabulatori su isto tako ignorirani. Naravno, unutar samog elementa razmaci se ne diraju. Ako je potrebno da razmak postoji, jedno od mogućih rješenja je ispis niza znakova, koji sadrži razmak [9].

Kako se elementi pretvaraju u poziv funkcije, važno je da postoji jedan korijenski element. Ako se element sastoji od drugih elemenata, mora ih se staviti u neki drugi element. U primjeru ispod je prikazana ispravna i neispravna sintaksa.

```
const neispravni = <span>prvi</span><span>drugi</ span >; // neispravno
const ispravni = (
      <div>
            <span>prvi</span>
            <span>drugi</span>
      <div/>
); // ispravno
```

U nekim slučajevima ovakvo rješenje, stvara nepravilan HTML ili preveliku dubinu DOM-a. Ako se ne želi stvoriti novi element u DOM-u, može se koristiti Reactov fragment. Fragment se stvara s oznakom `React.Fragment`, te se koristi isto kao i bilo koji drugi element. Predložena je i jednostavnija sintaksa za fragment s oznakama `<>` i `</>`, koja u trenutku pisanja rada nije podržana u svim alatima i preprocesorima [1].

JavaScript naredbe unutar XML oznaka se pišu unutar vitičastih zagrada. U njih se može napisati varijabla ili funkcija, ili bilo koji drugi JavaScript izraz. Prilikom izrade DOM-a, ignoriraju se logičke vrijednosti, nedefinirane i `null` vrijednosti. U primjeru je prikazano stvaranje elementa korištenjem funkcije i varijabli, te pisanje komentara u JSX-u.

```
const naslov = () => "Naslov";
const sadrzaj = <p>Sadržaj.</p>;
const element = (
      <div>
            <h1> { naslov() } </h1>
            { sadrzaj }
            {/* komentar */}
      </div>
);
```

Atributi se postavljaju jednako kao i sadržaj elemenata. Razlika postoji jedino u logičkim vrijednostima. Naime, svi atributi kojima nije zadana vrijednost imaju vrijednost `true`, slično kako se ponašaju i HTML atributi. Ako se želi kontrolirati vrijednost nekog atributa, može se postaviti na neki JavaScript izraz.

```
let onemogucenUnos = true;
const element = <input type="text" disabled={ onemogucenUnos } />;
```

U prethodnom primjeru je prikazano onemogućavanje tekstualnog unosa kontrolirano s logičkom varijablom. U stvarnim slučajevima to može biti funkcija ili izraz, koji provjerava druge elemente na obrascu [1].

U stvarnim aplikacijama česti su slučajevi kada se neki sadržaj prikazuje, ovisno o nekom logičkom izrazu, ili kada se neki element prikazuje više puta, ovisno o količini podataka koje je potrebno prikazati. Niti u jednom od ova dva slučaja, nije moguće unaprijed odrediti kako će izgledati konačni DOM. Najočitiji način uvjetnog prikazivanja elementa je korištenje `if-else` grananja. Unutar grananja se postavlja varijabla koja sadrži element, ovisno o logičkom uvjetu. Ponekad je preglednije grananje napisati direktno u JSX-u, bez korištenja varijabli. To se postiže korištenjem logičkog operatora i. Ako je prvi operand laž, drugi operand se ignorira i izraz se ne prikazuje korisniku. Ako je istina, cijeli izraz postaje jednak drugom operandu i kao takav se prikazuje korisniku. Za složenije slučajeve, gdje je potrebno zamijeniti `if-else` grananje, JavaScript podržava uvjetni izraz. U uvjetnom izrazu uvjet se piše prije upitnika, a zatim slijede izrazi, koji se prikazuju ovisno o uvjetu i odvojeni su dvotočkom. JSX prikazuje polja na način da prikaže svaki element polja posebno. Korištenjem tog svojstva, može se ostvariti ponavljanje elemenata. Polje podataka najlakše je pretvoriti u polje elemenata, korištenjem metode `map`. Metoda `map` stvara novo polje, tako da funkciju koja joj je proslijeđena pozove sa svakim elementom, a od povratnih vrijednosti funkcije stvori novo polje. Kod većih i složenijih prikazivanja polja preporučuje se postavljanje atributa `key`. Pomoću tog atributa, pomirenje DOM-a i virtualnog DOM-a postaje efikasnije i brže. Atribut `key` se postavlja na neku vrijednost podatka, koja se neće mijenjati, na primjer primarni ključ u bazi podataka. Slijedi primjer stvaranja elementa od polja podataka.

```
const elementiListe = polje.map((x) => <li key={x.id}>{x.text}</li>);
const elementiPostoje = polje.length > 0;
const ispisanoPolje = (
      <div>
            {elementiPostoje ? 'Polje sadrži elemente,' : 'Polje je prazno.'}
            {elementiPostoje && <ul>{elementiListe}</ul>}
      </div>
);
```

Deklaracija polja podataka nije prikazana u primjeru. Elementi se ispisuju u neuređenu listu, ako polje nije prazno. U ovom općenitom primjeru, za element se pretpostavlja da je neki objekt, koji ima svoj tekst i svoju identifikacijsku oznaku. Ovisno o broju elemenata, ispisuje se poruka pomoću uvjetnog izraza [1].

U ovom poglavlju je opisan JSX i kako se koristi React. Vidljivo je kako uvelike olakšava pisanje aplikacija i doprinosi preglednosti koda. Međutim, ni korištenje čistog JavaScripta nije previše kompliciranо. Čisti JavaScript posebno je primjeren za male aplikacije, na kojima se želi izbjeći korištenje preprocesora i želi se koristiti biblioteka Ract, preuzeta s CDN-a. Kako je JSX preporučen od strane autora Reacta i olakšava pisanje aplikacija, koristit ću ga u ostatku ovog rada.

3.2. Komponente

Osnovni dio aplikacije izrađene u Reactu, i razlog zašto se uopće ta biblioteka koristi, su komponente. Cijelo korisničko sučelje aplikacije može biti podijeljeno na manje komponente, koje su iskoristive u različitim dijelovima aplikacije. Podjela aplikacije na manje dijelove omogućuju lakši razvoj i testiranje aplikacija. U JSX-u, komponente se upotrebljavaju isto kao i HTML elementi. Najjednostavnije komponente su funkcije, koje vraćaju Reactov element. Funkcijske komponente primaju parametar sa svojstvima (*eng. properties ili props*). Vrlo je važno da su komponente čiste funkcije, drugim riječima, ne smiju se modificirati primljena svojstva. Svojstva se u JSX-u prosljeđuju pomoću atributa. Posebno svojstvo su djeca komponente, pod imenom `children`. Djeca su vrlo korisna kada komponenta ne zna koji sadržaj će se prikazivati u njoj, što je slučaj kod okvira za prikaz sadržaja. Za slučajeve kada je potrebno više djece, mogu se proslijediti pomoću drugih atributa. Slijedi primjer nekoliko funkcijskih komponenti.

```
const Naslov = ({ naslov }) => {
      return <h1>{naslov}</h1>;
};
const Sadrzaj = ({ children }) => {
      return <article>{children}</article>;
};
const GlavnaKomponenta = () => {
      return (
            <div>
                  <Naslov naslov="Ovo je naslov!" />
                  <Sadrzaj>Ovo je sadržaj.</Sadrzaj>
            </div>
      );
};
```

U stvarnim slučajevima, komponente se uglavnom pišu u odvojenim datotekama te ih je potrebno izvesti iz modula. U svaku komponentu treba uvesti komponente koje koristi. Glavna

komponenta se iscrtava na web stranici, korištenjem funkcije `ReactDOM.render`. U ovom primjeru, objekt sa svojstvima je odmah destrukturiran na jednostavnije varijable [1].

Složeniji način pisanja komponenata je pomoću klasa. Komponenta bazirana na klasi, mora imati metodu `render`. Metoda `render` vraća Reactov element, isto kao i funkcijska komponenta. Svaka komponenta bazirana na klasi, nasljeđuje klasu `React.Component`. Komponente bazirane na klasama su nešto složenije od funkcijskih komponenti, ali imaju neke prednosti. Prvi argument koji prima konstruktor su svojstva komponente, te ih je potrebno proslijediti konstruktoru klase `React.Component`. Ako konstruktor komponente nije napisan, to nije potrebno napraviti. U komponentama baziranim na klasi, svojstva su smještena u objektu `props`. Komponenta `Naslov` je pretvorena iz funkcijske na komponentu baziranu na klasi.

```
class Naslov extends React.Component {
  render() {
    return <h1>{this.props.naslov}</h1>;
  }
}
```

Na ovoj jednostavnoj komponenti nisu vidljive prednosti komponenta, baziranim na klasama. Glavni razlog za korištenje takvih komponenata je unutarnje stanje komponente (*eng. state*). Za razliku od svojstva, koje kontrolira komponenta roditelj, stanje kontrolira komponenta na koju se to stanje odnosi. Stanje se inicijalizira u konstruktoru pomoću varijable `state`. Stanje se može promijeniti korištenjem metode `setState`. Metoda prima objekt, koji sadrži promjene stanja. Ključevi koji nisu sadržani u tom objektu, neće se promijeniti u objektu stanja. Metoda `setState` ne garantira sinkrono izvršavanje, te se smatra da je asinkrona. Direktnim promjenama objekta stanja, neće doći do promjena u DOM-u, te to nije dozvoljeno. Slijedi relativno jednostavan primjer korištenja stanja. U primjeru je napravljen gumb, koji na sebi ima brojač klikova.

```
class Brojac extends React.Component {
      constructor(props) {
            super(props);
            this.state = { brojKlikova: 0 };
            this.dodajKlik = this.dodajKlik.bind(this);
      }

      dodajKlik() {
            this.setState({ brojKlikova: this.state.brojKlikova + 1 });
      }
```

```
	render() {
		return (
			<button onClick={this.dodajKlik}>
				Broj klikova: {this.state.brojKlikova}
			</button>
		);
	}
}
```

U primjeru su korišteni JavaScript događaji. Koriste se slično kao i u HTML-u, ali u JSX-u primaju referencu na funkciju, umjesto naziva funkcije. Nakon što se element pojavi u DOM-u, izgubit će referencu `this`. Kako bi se omogućila promjena stanja, `this` se povezuje s metodom u konstruktoru [1][9].

Posebne metode komponente su metode životnog ciklusa (*eng. lifecycle hooks*). Metode životnog ciklusa se pozivaju automatski, kada komponenta dođe u određenu fazi životnog ciklusa. Prilikom stvaranja komponente, prvo se izvršava konstruktor klase. Metoda `render` je također metoda životnog ciklusa, i izvršava se nakon konstruktora kod stvaranja komponente. Nakon što je komponenta umetnuta u DOM, poziva se metoda `componentDidMount`. U ovu metodu je preporučeno staviti AJAX pozive i dohvatiti podatke. Kada dođe do promjene stanja komponente ili promjena svojstva, komponenta prelazi u životni ciklus promjene. Nakon pozivanja metode `render` i promjena u DOM-u, poziva se metoda `componentDidUpdate`. Ova metoda u parametrima prima prijašnje stanje i prijašnja svojstva. Vrlo je važno provjeriti je li došlo do promjena u njima, jer je inače stvorena beskonačna petlja. Kod promjena postoji metoda koja se poziva prije metode `render`, a naziva se `shouldComponentUpdate`, te prima sljedeću vrijednost svojstava i stanja komponente. Ako ova metoda vrati logičku vrijednost laž, neće doći do poziva metode `render`. Korištenjem ove metode mogu se poboljšati performanse aplikacije. Kada komponenta više nije potrebna, prije izbacivanja iz DOM-a, poziva se metoda `componentWillUnmount`. U ovoj metodi potrebno je osloboditi zauzete resurse, poput mrežnih zahtjeva i vremenskih intervala. Metoda životnog ciklusa, koja se poziva ako dođe do pogreške, naziva se `componentDidCatch`. Metoda hvata sve pogreške, koje su se dogodile u djeci komponentama. Nakon poziva ove metode, pogreška se ne propagira dalje na roditeljske komponente. Ako ova metoda ne postoji, pogreškom se komponente i roditeljske komponente izbacuju iz DOM-a, te će korisnik na kraju dobiti prazan zaslon [1].

Komunikacija između komponenti je jedan od osnovnih preduvjeta, za podjelu složenije aplikacije na manje komponente. Ako više komponenti koristi iste podatke, ti podaci se trebaju

nalaziti na nekoj zajedničkoj roditeljskoj komponenti, te se zatim mogu poslati prema komponentama koje ih koriste. Slanje podataka od roditeljske komponente prema komponenti dijete je napravljeno pomoću svojstva, i već je pokazano. Komunikacija u suprotnom smjeru, odvija se slanjem funkcija povratnog poziva prema komponentama djeci. Funkcije se definiraju kao metode u klasi roditeljske komponente, te se šalju komponentama kao svojstva. Pozivom tih funkcija, komponenta dijete može mijenjati podatke na roditeljskoj komponenti. U primjeru je prikazan brojač klikova.

```
const Tipka = ({ dodaj, broj }) => {
  return <button onClick={dodaj}>Broj klikova: {broj}</button>;
};

class Brojac extends React.Component {
  constructor(props) {
    super(props);
    this.state = { brojKlikova: 0 };
    this.dodajKlik = this.dodajKlik.bind(this);
  }

  dodajKlik() {
    this.setState({ brojKlikova: this.state.brojKlikova + 1 });
  }

  render() {
    return (
      <Tipka dodaj={this.dodajKlik} broj={this.state.brojKlikova} />
    );
  }
}
```

Podaci se nalaze u komponenti `Brojac`, a mijenja ih komponenta `Tipka`. U ovom jednostavnom primjeru nije potrebno imati dvije komponente. Međutim, pomoću funkcije se mogu prenijeti bilo kakvi parametri. Ponekad je korisno na komponenti definirati metode, koje pozivaju funkciju s roditeljske komponente s određenim parametrima [1].

Jedan od načina razdvajanja funkcionalnosti na komponente su prezentacijske (*eng. presentational components*) i kontejnerske komponente (*eng. container components*). U kontejnerskim komponentama su smješteni podaci. To su komponente bazirane na klasi, koje prosljeđuju podatke prezentacijskim komponentama. Kontejnerske komponente sadrže svu logiku, dohvaćaju podatke s poslužitelja i prikazuju prezentacijsku komponentu. Prezentacijske

komponente je uglavnom moguće napisati kao funkcijske komponente, ali je moguće da i one imaju neko svoje stanje, koje se odnosi na prikaz te komponente. Prezentacijske komponente prikazuju primljene podatke korisniku, te brinu kako će se podaci prikazati. Odvajanje funkcionalnosti na prezentacijske i kontejnerske komponente ima brojne prednosti. Prezentacijske komponente se ponekad mogu iskoristiti u različitim kontejnerskim komponentama. Lakši je razvoj aplikacije, jer je moguće napraviti prezentacijsku komponentu s lažnim podacima, a kasnije mijenjati logiku i izgled aplikacije, odvojeno jedno od drugoga. Nedostatak ove podjele jest povećanje broja komponenati u projektu, i za neke funkcionalnosti ne postoji jasna podjela na ove vrste komponenti, dok za manje funkcionalnosti ne postoji potreba za odvajanjem [9].

DOM elementi za unos u obrascima, imaju vrijednost koju korisnik može mijenjati. Elementi za unos mogu biti kontrolirani i nekontrolirani. Kod kontroliranih komponenti, stanje komponente i vrijednost elementa za unos se uvijek podudaraju. Promjenom jednoga, mijenja se i drugo. Kod nekontroliranih komponenti, to nije uvijek slučaj. Moguće je, primjerice, postaviti samo inicijalnu vrijednost elementa, a da kasnije promjene stanja ne utječu na vrijednost elementa. Slijedi jednostavan primjer kontrolirane komponente.

```
class KontroliranaKomponenta extends React.Component {
      constructor(props) {
            super(props);
            this.state = { vrijednost: 'početna vrijednost'};
            this.handleChange = this.handleChange.bind(this);
      }

      handleChange({ target }) {
            this.setState(
                  { [target.name] : target.value}
            );
      }

      render() {
            return (
                  <input name="vrijednost"
                  onChange={this.handleChange}
                  value={this.state.vrijednost}
                  />
            );
      }
}
```

Metoda koja prati promjene, dobiva događaj. U tom događaju se nalaze podaci o novoj vrijednosti i ime elementa. Korištenjem imena elementa za ključ, pod kojim je spremljeno stanje, postiže se višestruka iskoristivost te metode. U protivnom bi se za svaki element, trebala napraviti posebna metoda. DOM elementu je moguće direktno pristupati putem reference. Reference se stvaraju u konstruktoru i dodjeljuju nekoj komponenti ili elementu. Za postavljanje početnih vrijednosti, React nudi atribut `defaultValue`. Ako se postavi `value`, neće biti moguće mijenjati sadržaj elementa. Slijedi primjer nekontrolirane komponente.

```
class NeKontroliranaKomponenta extends React.Component {
 constructor(props) {
   super(props);
   this.handleSubmit = this.handleSubmit.bind(this);
   this.inputRef = React.createRef();
 }

 handleSubmit(e) {
   e.preventDefault();
   console.log(this.inputRef.current.value);
 }

 render() {
   return (
     <form onSubmit={this.handleSubmit}>
       <input ref={this.inputRef} defaultValue={this.props.vrijednost} />
       <button>Pošalji!</button>
     </form>
   );
 }
}
```

Kako se obrazac ne bi poslao na poslužitelj, potrebno je pozvati metodu `preventDefault`. U ovom se primjeru vrijednost elementa samo ispisuje u konzolu. U primjerima su pokazane kontrolirane i potpuno nekontrolirane komponente. U praksi je moguć slučaj između ova dva ekstrema. Primjer može biti da se postavi početna vrijednost, i da promjene u elementu mijenjaju stanje, ali da promjene stanja ne mijenjaju sadržaj elementa. Kod izrade funkcionalnosti aplikacije, potrebno je odrediti kakav obrazac će se koristiti [1][9].

3.3. Pomoćne biblioteke

Jedna od velikih prednosti Reacta, prvenstveno zbog njegove popularnosti, su programi i dodaci koji rade s njim. Sintaksa JSX-a je podržana u mnogim uređivačima teksta, a napravljeni su i programi, koji pomaži pri otklanjanju pogreški u web pregledniku. Napravljeni su dodaci, koji omogućuju lako pisanje testova i testiranje aplikacija. Podržani su i razni načini pisanja CSS-a. Neki od njih niti ne koriste standardni CSS, već koriste JavaScript objekte, koji se pretvaraju u CSS. Zbog boljeg korisničkog iskustva, moguće je početni DOM pripremiti već na poslužitelju i tako poslati korisniku. Uz to, dostupne su i biblioteke koje sadrže gotove komponente, koje se mogu iskoristiti za brži razvoj aplikacija. U ovom poglavlju je opisano nekoliko takvih biblioteka i programa [9].

3.3.1. webpack i Babel

webpack je program koji uzima izvorni kod aplikacije te iz njega stvara statičke resurse, koji se pokreću u web pregledniku. Prvenstveno je namjene za spajanje različitih JavaScripta modula u jednu datoteku. Jako je fleksibilan, pa su nastali dodaci koji omogućuju prevođenje JSX-a i korištenje CSS preprocesora. Moguć je direktan uvoz stilskih datoteka i slika u JavaScript datoteci. webpack ima svojeg poslužitelja nazvanog DevServer. Već iz imena je vidljivo kako je to poslužitelj isključivo namijenjen izradi aplikacije, i ima mnoge mogućnosti koje to olakšavaju. Jedna od njih je zamjena modula, dok se njihov kod izvršava u pregledniku, ili osvježavanje stranice kod spremanja datoteke. Postavke se spremaju u datoteku `webpack.config.js` ili se upisuju u komandnu liniju pri pokretanju programa [10].

webpack može pozvati razne dodatke, od kojih je jedan od najpopularnijih Babel. Babel je program, kojem je prvenstvena namjena prevođenje JavaScripta na starije izdanje. Time se postiže veća podržanost napisanog koda u web preglednicima. Za gotovo svaku mogućnost JavaScripta, moguće je podesiti kako će se prevesti u datoteci `.babelrc`. Kao i za webpack, razvijeni su dodaci. Jedan od dodataka omogućuje prevođenje JSX-a u čisti JavaScript, što je neophodno za korištenje JSX-a u Reactu [11].

3.3.2. Create React App

Podešavanje Webpacka i Babela može biti izrazito mukotrpno i komplicirano, posebice kada se žele podesiti razne napredne opcije. Kako bi se olakšalo stvaranje projekta, napravljen je alat Create React App. Ovaj alat također koriste Webpack i Babel, ali ne zahtijeva od korisnika njihovo poznavanje. Napredni korisnici imaju mogućnost da u nekoj fazi projekta izbace konfiguracijske datoteke, i da ih koriste kao da Create React App nikada nije niti postojao. Program se preuzima s NPM-a, te se dalje koristi na sljedeći način.

```
create-react-app  ime-aplikacije
```

Program Create React App je podešen tako da postoji što manja potreba za promjenama postavki projekta. Napisane su skripte za izgradnju projekta, testiranje, kao i pokretanje Webpackovog DevServera, a omogućeno je lako dodavanje novih biblioteka u projekt. Kako ovaj program olakšava izradu aplikacija, koristit ću ga za izradu projekta na kraju ovog rada [12].

3.3.3. React Router

U tradicionalnim web stranicama, u adresnoj traci preglednika korisnik može provjeriti na kojoj se stranici nalazi. Kod jednostraničnih web aplikacija, moguće su potpune izmjene stranice bez da se promijeni sadržaj adresne trake, te korisnik ne može koristiti tipke naprijed i natrag za navigaciju. Kako je ovaj problem moguće riješiti u JavaScriptu, napravljena je biblioteka koja to rješava i nazvana je Reacr Router. Kao i React, sastoji se od dva dijela, React Router i React Router DOM. Biblioteka je deklarativna poput Reacta, i koristi se identično kao i komponente. Dio aplikacije koji koristi React Router, ili jednostavnije cijela aplikacija, mora biti u elementu `BrowserRouter`. Linkovi se ostvaruju pomoću elemenata `Link` ili `NavLink`. Kod oba treba podesiti atribut `to` na lokaciju linka. `NavLink` ima mogućnost postavljanje klase na element, kada je link aktivan. Kako bi se prikazala komponenta na nekoj lokaciji, koristi se element `Route`. Kod njega treba podesiti na koju se lokaciju odnosi, pomoću atributa `path`, i koja komponenta se prikazuje pomoću atributa `component`. Ako je potrebno pokazati samo jednu komponentu, a postoji mogućnost da se zadana lokacija poklopi u više njih, elementi `Route` se stavljaju u element `Switch`. Slijedi primjer uporabe React Routera.

```
const Aplikacija = () => (
      <div>
            <Link to="/">Početna</Link>
            <Link to="/prva ">Prva</Link>
            <Link to="/druga ">Druga</Link>

            <Switch>
                  <Route path="/prva" component={Prva} />
                  <Route path="/druga" component={Druga} />
                  <Route path="/" component={Pocetna} />
            </Switch>
      </div>
);
```

U primjeru nije prikazano da komponenta Aplikacija mora biti dijete elementa `BrowserRouter`. To je moguće napraviti i u njenoj roditeljskoj komponenti ili funkciji `ReactDom.render` [13].

3.4. Testiranje

Testiranje aplikacija ima mnogo pozitivnih učinaka. Tijekom razvoja jednog dijela, moguće je izazvati pogrešku u nekom drugom dijelu. Dobro napisani testovi olakšavaju otkrivanje ovakvih slučajeva. Testovi daju sigurnost programerima i omogućuju brži razvoj novih funkcionalnosti. Kada se otkrije pogreška u aplikaciji, moguće je napisati test koji ju pokriva, i osigurava da se ista pogreška opet ne pojavi. Kako bi se ostvarile sve prednosti testiranja, potrebno je napisati testove koji pokrivaju sve rubne uvjete. React preporučuje korištenje programa Jest za testiranje. Kao i React, razvijen je za Facebookove potrebe. Jest nije ni na koji način povezan s Reactom, i može se koristiti za testiranje bilo kakvih JavaScript aplikacija, te se React može testirati i s drugim programima poput Moche. [9].

Komponente primaju svojstva različitih imena i tipova. Uglavnom je potrebno osigurati da ime i tip ostanu nepromijenjeni. Isto tako, pogreške se mogu napraviti i kod poslužitelja, te će putem AJAX zahtjeva, komponenta dobiti objekt s krivo napisanim imenom ključa. To se može izbjeći zadavanjem tipova svojstava. Iako ovo zapravo nije dio testiranja, pomaže pri izbjegavanju pogreški. Ako se prekrši zadani tip svojstva, ispisat će se pogreška. Tipovi svojstava su specificirani u paketu PropTypes. Slijedi primjer komponente s definiranim tipovima svojstava [1].

```
const Okvir = ({ ime, children }) => {
      return (
            <div>
                  <h1> {ime}</h1>
                  {children}
            </div>
      );
}
Okvir.propTypes = {
      ime: PropTypes.string.isRequired
      children: PropTypes.element.isRequired
};
```

Jest je biblioteka za jedinično testiranje JavaScripta. Dolazi podešena, ako je aplikacija izgrađena s Create React App, ili se može preuzeti s NPM-a. Uz Jest se može koristiti biblioteka Enzyme. Enzyme je biblioteka razvijena u tvrtki AirBnB za testno iscrtavanje komponenti. Pomoću Enzyma je moguće iscrtati samo jednu komponentu, bez da se iscrtavaju komponente koje ta komponenta sadrži. Enzyme omogućuje pretraživanje elemenata iscrtanih komponenti i simulaciju događaja. Slijedi primjer jednostavne komponente koja će se testirati.

```
// datoteka tipka.js
export const Tipka = ({ naziv, akcija }) => (
      <button onClick={akcija}>
            {naziv}
      </button>
);
```

Testovi se uglavnom pišu u odvojenim datotekama. Datoteke s testovima se razlikuju od ostalih datoteka, tako da im ime završava s `test` ili `spec`. Slijedi primjer dva testa za napisanu komponentu.

```
// datoteka tipka.test.js
test("Tipka treba pokazivati tekst", () => {
      const testTekst = "Testni text";
      const tipka = shallow(<Tipka naziv={testTekst} />);
      expect(tipka.text()).toEqual(testTekst);
});

test("Tipka poziva akciju", () => {
      const akcija = jest.fn();
      const tipka = shallow(<Tipka akcija={akcija} />);
      tipka.simulate("click");
      expect(akcija).toBeCalled();
});
```

U prvom testu se provjerava ispisuje li komponenta tekst, koji joj je poslan. Drugi test provjerava hoće li se poslana funkcija pozvati, prilikom klika na tipku [9][14].

3.5. Usporedba s drugim bibliotekama

Uz React, najpopularnije biblioteke za razvoj jednostraničnih web aplikacija su Vue.js i Angular. Ove biblioteke imaju mnoge sličnosti. U ovim bibliotekama, aplikacija se dijeli na manje komponente. Za sve biblioteke su napravljeni alati, koji omogućuju lakši početak pisanja aplikacije, te postoji aktivna zajednica korisnika.

3.5.1. Angular

Angular je biblioteka nastala u Googleu, i trenutno je aktivna inačica Angular 6. Zbog svoje veličine i mogućnosti, nazivaju je platformom za razvoj web aplikacija. Napisana je u TypeScriptu, jeziku koji se prevodi u JavaScript, isto kao i JSX. Međutim, TypeScript sadrži statičku provjeru tipova podataka, što smanjuje vjerojatnost pisanja pogreški i nudi bolju podršku u editorima teksta. Angular je napravljen kao skup modula, od kojih se uključuju potrebni. Napravljeni su moduli za rješavanje gotovo svih bitnih segmenata web aplikacije poput AJAX zahtjeva, upravljanja obrascima i animacijama. Osim komponenti, podržava izradu servisa, modula i direktiva. Time je osigurana bolja podjela aplikacije na slojeve. Nedostatak Angulara je veličina biblioteke. To značajno povećava vrijeme učenja, a početna aplikacija napravljena pomoću programa Angular-CLI, je višestruko veća od biblioteke React. Uz to, namjena joj je isključivo za jednostranične web aplikacije, i nije ju moguće preuzeti s CDN-a. Slijedi primjer komponente napisane u Angularu.

```
import { Component } from '@angular/core';

@Component({
	selector: 'app-lista',
	template: `
		<h1>{{naslov}}</h1>
		<ul>My favorite hero is: {{myHero}}
			<li *ngFor="let rijec of lista">{{rijec}}</li>
		</ul>
	  `
})
export class Lista {
	@Input naslov: string;
	lista: string[];

	constructor() {
		this.lista = ['Prvi', 'Drugi', 'Treći'];
	}
}
```

Komponenta prima naslov te ga ispisuje, a ispod toga ispisuje listu. Ovo je komponenta u jednoj datoteci, ali moguće je razdvajanje HTML predloška i CSS u druge datoteke. Za razliku od Reacta, Angular ima vlastitu sintaksu predložaka pomoću atributa, kao što su `*ngFor` i `*ngIf`. Razlikuje se i sintaksa pisanja atributa na element. Ovisno o vrsti zagrada, određuje

se radi li se o događaju, uvjetno postavljenom atributu ili dvosmjernom povezivanju podataka [15].

3.5.2. Vue.js

Vue je mali okvir za izradu aplikacija od svega 30 kilobajta. Za razliku od Angulara, a poput Reacta, upravlja samo izgledom korisničkog sučelja. Vue je primjerno koristiti za izradu jednostraničnih web aplikacija, ali i za manje dijelove web stranica. Također, koristi virtualni DOM za poboljšavanje performansi aplikacija. Sintaksa HTML predloška slična je sintaksi koju koristi Angular. Vue koristi posebnu vrstu datoteka s nastavkom `.vue`, u koje su smještene komponente. U posebne HTML elemente su smješteni CSS stilovi, HTML predložak i JavaScript kod. Slijedi primjer Vue komponente.

```
<template>
      <div class="hello">
            <h1>{{ naslov }}</h1>
            <ul>
                  <li v-for="rijec in lista">{{ rijec }}</li>
            </ul>
      </div>
</template>

<script>
export default {
      name: "HelloWorld",
      data() {
            return {
                  lista: ['Prvi', 'Drugi', 'Treći'],
            };
      },
      props: ['naslov']
};
</script>
```

U primjeru je vidljiva sličnost HTML predloška s Angularom. Međutim podaci su odvojeni na svojstva i stanje, kao što je slučaj kod Reacta [16].

4. Redux

Redux je biblioteka namijenjena skladištenju stanja jednostranične web aplikacije. Redux je mala biblioteka od svega 2 kilobajta. Nastala ja kako bi se pojednostavilo upravljanje stanjem kod aplikacija, izrađenih u Reactu. React i Redux nisu nikako povezani i jedna biblioteka ne zahtijeva drugu. Primjerice, moguće je koristiti Redux s Angularom ili Vue.js-om. Facebook je napravio sličnu biblioteku nazvanu Flux, kojom je Redux inspiriran. Glavna motivacija za izradu Reduxa, bila je potreba za boljim i jednostavnijim načinom upravljanja stanjem aplikacije. U jednostraničnim web aplikacijama, može doći do promjena istog podatka u različitim dijelovima aplikacije, te jedna promjena može izazvati drugu. U nekom trenutku je moguće izgubiti kontrolu nad stanjem aplikacije, i promjene izazivaju nepredvidljivost stanja. Time je značajno otežan razvoj aplikacija, dodavanje novih funkcionalnosti i izmjena postojećih. Kako Redux rješava ove probleme, postao je popularan te su razvijeni mnogi dodaci [2].

Redux je zasnivan na tri principa: jedan izvor istine, stanje se može samo čitati i promjene se ostvaruju putem čistih funkcija. Svi podaci su spremljeni u JavaScript objektu, koji se naziva skladištem (*eng. store*). Skladište, kao jedini izvor istine, omogućuje lakšu izradu aplikacija. Moguće je razvijati komponente s istim podacima, bez da ih je potrebno unositi više puta u aplikaciju. Zatim, lako se sprema prošlo stanje aplikacije, ako je potrebno. Skladište nije moguće direktno mijenjati, ali se podaci mogu čitati. Promjene su moguće jedino slanjem akcija (*eng. action*). Time je osigurano da događaji nastali od korisnika ili poslužitelja, neće direktno promijeniti stanje. Akcije su JavaScript objekti koji imaju svoj tip. Uz tip, akcije mogu imati neke podatke. Kako su akcije obični JavaScript objekti, moguće ih je spremiti, i ponoviti kod razvoja aplikacija. Reduktori (*eng. reducer*) su funkcije koje primaju prijašnje stanje i akciju, a vraćaju novo stanje. Reduktori su čiste funkcije. Cijela biblioteka Redux je napravljena na principima funkcijskog programiranja, pa iskorištava prednosti te paradigme. Cijela aplikacija može koristiti jedan reduktor, ali kako kompleksnost raste, može se podijeliti na više njih. Moguće je izraditi generičke reduktore za neke česte funkcionalnosti, poput straničenja [2].

4.1. Akcije

Akcije su objekti koji sadrže informacije, koje se šalju u skladište. Ujedno su i jedini izvor informacija, koje skladište može primiti. Sve akcije moraju imati svoj tip. Tip akcije je niz znakova, po kojoj se akcija razlikuje od drugih. Dobra praksa je spremiti tipove akcija u odvojenu datoteku, čime je osigurano da ime akcije nije pogrešno napisano u kodu akcije ili u reduktoru. Ovisno o namjeni akcije, ona može imati i druge podatke. Akcije se uglavnom

izrađuju pomoću posebnih funkcija, koje se nazivaju kreatori akcija. Slijedi primjer s nekoliko kreatora akcija.

```
const DODAJ_RECENICU = 'DODAJ_RECENICU';
const OBRISI_SVE = ' OBRISI_SVE';

const dodajRecenicu = (recenica) => {
      return {
            type: DODAJ_RECENICU,
            recenica: recenica
      };
}
const obrisiRecenice = (recenica) => {
      return {
            type: OBRISI_SVE
      };
}
```

U primjeru su prikazane dvije funkcije koje stvaraju akcije. Akcije koje one stvore, moraju se poslati skladištu korištenjem metode `dispatch`, kako bi došlo do promjene stanja aplikacije. Ovi kreatori akcija su sinkroni. Kako postoje mnoge funkcionalnosti, poput slanja podataka poslužitelju, koje su asinkrone, moguće je napraviti i asinkrone kreatore akcija. Asinkroni kreatori akcija stvaraju više sinkronih akcija, ovisno o nekom asinkronom događaju [2].

4.2. Reduktor

Reduktor je čista funkcija u kojoj se implementira, kako će neka akcija promijeniti stanje aplikacije. Reduktor prima trenutno stanje i poslanu akciju, iz koje stvara i vraća novo stanje aplikacije. Slijedi primjer reduktora, koji prima akcije napisane u prethodnom poglavlju.

```
const recenice = (stanje = [], akcija) => {
      switch (akcija.type) {
            case DODAJ_RECENICU:
                  return […stanje, akcija.recenica];
            case OBRISI_SVE:
                  return [];
            default:
                  return stanje;
      }
}
```

Kako su reduktori čiste funkcije, pri dodavanju riječi u polje, bilo je potrebno napraviti novo polje. To je napravljeno korištenjem operatora širenja. U slučaju da stanje nije definirano, vrijednost je postavljena na prazno polje. Ako se niti jedna akcija ne poklopi, reduktor mora vratiti prijašnje stanje. Ovo je iznimno važno, jer će se u protivnom izgubiti stanje koje je prije postojalo. Ako je glavni reduktor stvoren od manjih reduktora, to će značiti da će pri svakoj akciji namijenjenoj nekom reduktoru, ostali izgubiti svoje stanje. Kako bi se više reduktora spojilo u jedan, koristi se Reduxova funkcija `combineReducers`. Funkcija prima reduktore, koje će spojiti u jedan. Svi reduktori se šalju u jednom objektu. Ključevi tog objekta su ključevi, pod kojim će njihovo stanje biti spremljeno u skladištu. Ova funkcija pokriva samo najjednostavnije slučajeve, kada je stanje spremljeno u čistim JavaScript objektima, a manji reduktori ne koriste podatke koje stvara drugi reduktor. Ako se radi o složenijim primjerima, potrebno je napisati funkciju koja spaja manje reduktore u jedan. To je vrlo jednostavno, jer je potrebno napraviti funkciju kojoj je jedina svrha pozvati sve reduktore s istom akcijom. Korištenjem koncepta funkcije višeg reda iz funkcijskog programiranja, mogu se napraviti i reduktori višeg reda. To je funkcija koja prima neki parametar, a vraća reduktor. Vrlo su korisni za funkcionalnosti koje se ponavljaju [2].

4.3. Skladište

Skladište je objekt u koji je smješteno stanje aplikacije. Skladište se stvara pozivanjem metode `createStore`. Ta metoda prima glavni reduktor, a vraća skladište. Redux je zamišljen tako da cijela aplikacija koristi jedno skladište. Preporučuje se normalizacija podataka, slično kao što je napravljeno u relacijskim bazama podataka. Time se izbjegava redundancija podataka, pa je i mijenjanje podataka jednostavnije, a reduktori ne moraju mijenjati duboku ugniježđene podatke. Akcije se šalju korištenjem metode `dispatch`. Metodu nije moguće pozvati unutar reduktora, već se za takve slučajeve trebaju koristiti asinkroni kreatori akcija. Stanje aplikacije vraća metoda `getState`. Međutim, ta metoda ne govori je li došlo do promjena stanja. Za praćenje promjena stanja, postoji metoda `subscribe`. Metoda prima funkciju koja će se pozvati prilikom promjene stanja, a vraća funkciju pomoću koje se prekida praćenje promjena. Slijedi primjer korištenja skladišta.

```
const pocetnoStanje = [];
const store = createStore(recenice, pocetnoStanje);
const prestaniPratiti = store.subscribe(() =>
  console.log(store.getState()); // ispiši stanje nakon promjene
);
prestaniPratiti(); // prekida praćenje promjena stanja
```

Posljednja metoda, koja se može pozivati na objektu skladišta, je `replaceReducer`. S tom metodom se zamjenjuje trenutni reduktor skladišta. Korisna je, ako se želi podijeliti aplikaciju na manje dijelove pa će se nakon dohvaćanja drugog dijela, novi reduktor dodati u skladište [2].

4.4. Asinkrone akcije

Stanje aplikacije u svakom trenutku mora biti poznato i određeno. Kako asinkronost donosi narušavanje ovog svojstvo, akcije moraju biti sinkrone. Asinkronost se postiže korištenjem više sinkronih akcija. Uglavnom je potrebno napraviti tri sinkrone akcije, jednu za početak asinkrone operacije, jednu ako je asinkrona operacija uspješno završila te jednu ako je došlo do pogreške. Asinkrone akcije nisu podržane u čistom Reduxu, ali je razvijen dodatak Redux Thunk. Redux Thunk nije jedini dodatak koji to omogućuje, ali je razvijen od autora Reduxa i vrlo je popularan, te ću ga koristiti u ovom radu. Svi takvi dodaci proširuju funkciju `dispach`, pa ona osim akcije, može primiti i funkciju ili obećanje. Funkcija kreator asinkronih akcija, ne mora biti čista funkcija te je dozvoljeno korištenje funkcija, koje rade AJAX pozive ili druge nepredvidljive radnje. U primjeru ispod, slijedi primjer asinkrone akcije.

```
const dohvatiKorisnika = (id)  => (dispatch) => {
      dispatch(zapocniDohvacanje())
      return fetch(`primjer.com/korisnik/${id}`)
            .then(
                  korisnik => korisnik.json(),
                  pogreska => dispatch(pogreskaDohvacanja())
            )
            .then(
                  korisnik => dispatch(korisnikDohvacen(korisnik))
            )
}
```

U primjeru, kreator akcije, umjesto akcije vraća funkciju. Ta funkcija u sebi sadrži tri funkcije koje kreiraju sinkrone akcije. U obećanju, nije se koristila metoda `catch` jer bi ona uhvatila pogrešku, neovisno o mjestu nastanka. Naime, funkcija `dispatch` poziva Reactovu metodu `setState`. Ako dođe do pogreške u komponenti, vrlo će je vjerojatno uhvatiti kreator asinkrone akcije. Time je otežano otkrivanje pogrešaka, jer se pogreška hvata na mjestu u kodu s kojim nije logički povezana. Na isto je potrebno obratiti pažnju, kod korištenja `try-catch` bloka u kombinaciji s `async-await`. Tada se funkcija `dispatch` ne smije pozvati unutar `try` bloka, jer će doći do istog problema[2].

4.5. Upotreba s Reactom

Redux je nastao u namjeri da se može koristiti s Reactom. Za to je razvijena i biblioteka React Redux. Kao i Redux, može se preuzeti s CDN-a ili NPM-a. Redux je moguće koristiti i bez te biblioteke, izradom skladišta te njegovim korištenjem u komponentama. Praćenje promjena se lako može napraviti u komponentama, korištenjem metode skladišta `subscribe` u metodi životnog ciklusa `componentDidMount`. React Redux može izraditi optimizirane kontejnerske komponente. U njima je definirana metoda `shouldComponentUpdate`, tako da se komponente ne iscrtavanju, ako nije došlo do promjena u podacima koje koriste. Jedino kontejnerske, direktno komuniciraju s Reduxom. U njima su definirane funkcije, koje se šalju prezentacijskim komponentama kao svojstva, tako da prezentacijske komponente ne šalju akcije direktno u skladište. Kako bi se kontejnerska komponenta napravila, potrebne su dvije funkcije, jedna koja vraća podatke koji će se poslati prezentacijskoj komponenti, te jedna koja će napraviti metode, koje će prezentacijska komponenta pozvati kako bi došlo do promjene stanja. Prva metoda prima stanje iz skladišta te svojstva, koja joj se šalju, kada je kontejnerska komponenta iskorištena u nekoj drugoj komponenti. Druga funkcija prima funkciju za slanje podataka, te ista poslana svojstva. Slijedi primjer izrade kontejnerske komponente korištenjem funkcije `connect` iz biblioteke React Redux.

```
const mapiranjeStanja = stanje => {
      return {
            recenice: stanje.recenice
      };
}

const mapiranjeFunkcija = dispatch => {
      return {
            dodajRecenicu: recenica => dispatch(dodajRecenicu (recenica)),
            obrisiSve: () => dispatch(obrisiRecenice())
      };
}

const Recenice = connect(mapiranjeStanja, mapiranjeFunkcija)(ListaRecenica);
```

U primjeru su korištene prethodno napisani kreatori akcija `dodajRecenice` i `obrisiSve`. Za njih su napravljene istoimene funkcije, koje su poslane prezentacijskoj komponenti. U ovom slučaju, komponenti su direktno poslane rečenice spremljene u skladištu, a u složenijim slučajevima, stanje se prilagođava za prikazivanje korisniku pomoću filtriranja, sortiranja i sličnih operacija. Metoda `connect` vraća funkciju, kojoj se prosljeđuje prezentacijska

komponenta, u ovom slučaju `ListaRecenica`. Preciznije rečeno, funkcija `connect` vraća komponentu višeg reda, koja se ponaša slično kao funkcija višeg reda. Funkcija `connect` može primiti još dva parametra. U trećem parametru je moguće specificirati funkciju, koja će napraviti svojstva, koja se prosljeđuju prezentacijskoj komponenti. Ovdje je moguće preimenovati svojstva. Zadnji parametar funkcije `connect` je objekt s postavkama. Ako se u postavkama postavi `pure` na istinitu vrijednost, komponenta će se optimizirati kao da se radi o čistoj funkciji. Ta komponenta će se ponovno iscrtavati, samo ako će doći do promjene u stanju skladišta [2].

Kako bi se promjene u skladištu mogle pratiti, sve kontejnerske komponente moraju imati pristup do tog skladišta. Kako bi se izbjegla potreba za slanjem skladišta do kontejnerskih komponenti putem atributa, napravljena je komponenta `Provider`. `Provider` ima jedan atribut koji mora biti postavljen, a to je `store` i postavlja se na skladište. Umotavanjem aplikacije u ovu komponentu, skladište postaje dostupno u cijeloj aplikaciji. Ako aplikacija koristi biblioteku React Router, prijedlog je da `Provider` bude roditeljski element komponente `BrowserRouter`, koja će biti roditeljska komponenta ostatku aplikacije. Problem ovakvog korištenja mogu biti dva izvora istine, adresa aplikacije i stanje u skladištu. Ovisno o aplikaciji, to može biti problem. Jedno od mogućih rješenja je korištenje parametara iz adrese u kontejnerskim komponentama. Ovo nije jedino rješenje, i svako rješenje ovog problema bit će specifično za slučaj korištenja [2].

Podaci aplikacije izrađene u Reactu mogu biti u stanju komponenata, u adresnoj traci preglednika, u vrijednosti HTML elemenata za unos, u JavaScript varijablama izvan Reacta te u skladištu stanja. Neki od tih podataka su i na poslužitelju pa treba postojati sinkronizacija. Prilikom odabira arhitekture aplikacije, potrebno je odlučiti koji podaci će se nalaziti u skladištu stanja poput Reduxa. Jedan od načina je da se svi podaci stave u skladište, ali za neke podatake to nije potrebno. Primjer takvih podataka su podaci o stanju korisničkog sučelja, poput otvorenosti ili zatvorenosti izbornika. Postoje i podaci koji se koriste u samo jednoj komponenti, i nije ih potrebno imati u skladištu. Pri izradi aplikacije, potrebno je odrediti postoji li uopće potreba za korištenjem Reduxa ili nekog drugog skladišta stanja. Autor, Michele Bertoli [9], smatra da se Redux i MobX koriste više nego je potrebno, te da mnogi programeri prerano počinju koristiti skladišta stanja, bez da nauče koristiti stanje unutar Reactovih komponenti. Autori Reacta [1] navode kako se korištenjem samog Reacta, mogu napraviti vrlo kompleksne aplikacije. Nadalje, autori Reduxa navode kako je Redux namijenjen za slučajeve, kada čuvanje stanja u korijenskim komponentama aplikacije postaje previše komplicirano. Rješava probleme koji nastaju kada više komponenata koriste iste podatke, i događaju se mnoge promjene u podacima kroz vrijeme, iako nekad nije najbrži način za pisanje aplikacija [2].

4.6. Testiranje

Funkcijsko programiranje omogućuje lako testiranje, jer se za iste parametre uvijek dobiva isti izlaz. Redux će pojednostaviti i pisanje testova za Reactove komponente. Razlog tome je micanje stanja iz tih komponenti. Testiranje aplikacije se može pojednostaviti korištenjem jednog programa za testiranje cijele aplikacije, pa će u primjeru biti prikazano korištenje biblioteke Jest. Jest nije ograničen na korištenje neke biblioteke, te kako se postavi za React, moći će se koristiti za Redux. Prvi dio aplikacije, koji se može testirati, su kreatori akcija. Slijedi primjer njihovog testiranja.

```
test("Kreator akcije vraća akciju", () => {
      const recenica = "Testna rečenica";
      const akcija = dodajRecenicu(recenica);
      const ocekivanaAkcija {
            type: DODAJ_RECENICU,
            recenica: recenica
      };
      expect(akcija).toEqual(ocekivanaAkcija);
});
```

Testiranje asinkronih akcija je složenije. Razlog tome su asinkrone funkcionalnosti koje koriste. Primjerice, koristi li asinkrona akcija AJAX poziv, potrebno je lažirati podatke, i cijelu AJAX funkcionalnost. Jest može testirati sve asinkrone mehanizme JavaScripta. Reduktori se testiraju postavljanjem na početno stanje te slanjem akcija. Prilikom testiranja, nije potrebno koristiti skladište stanja. Jedina funkcionalnost, koju programer aplikacije može mijenjati u skladištu stanja su reduktori, pa je njih bolje testirati direktno. Time se pojednostavljuje testiranje, jer nije potrebno svaki put stavljati novo stanje u skladište, te se testovi brže pokreću. Slijedi primjer testiranja reduktora.

```
test("Reduktor dodaje rečenicu", () => {
      const recenica = "Testna rečenica";
      const pocetnoStanje = [];
      const novoStanje = recenice(pocetnoStanje, dodajRecenicu(recenica));
      expect([recenica]).toEqual(novoStanje);
});
```

U primjeru se testira reduktor `recenice`. Test provjerava hoće li reduktor, uz definirano početno stanje i zadanu akciju vratiti novo očekivano stanje. Zadnji dio aplikacije koji se može testirati, su komponente nastale korištenjem React Reduxa. Njihovo testiranje je identično, kao i testiranje Reactovih komponenti. Kod njih je potrebno provjeriti prosljeđuju li ispravne podatke uz zadano stanje, te funkcije koje prosljeđuju [2][14].

4.7. Alternativne biblioteke

4.7.1. Flux

Flux je biblioteka za kontrolu stanja aplikacije, napravljena od strane tvrtke Facebook. Ima sličnu arhitekturu kao i Redux, jer ju je Redux upotrijebio kao inspiraciju. Flux se vrlo često smatra nazivom arhitekture aplikacija, umjesto konkretno ova biblioteka. Glavna razlika između Fluxa i Reduxa je da Redux ima jedno skladište podataka, dok ih Flux može imati više. Flux ne koristi funkcijsko programiranje te je dozvoljeno mijenjati podatke u skladištu. Više skladišta omogućuje praćenje promjena stanja samo u odabranim skladištima. Flux sadrži jednog pošiljatelja (*eng. dispatcher*). To je objekt koji upravlja promjenama stanja, a zapravo je skup funkcija povratnih poziva. Kreatori akcija napravljenih u Fluxu, pozivaju pošiljatelja akcija, umjesto da se akcije šalju u skladište kao kod Reduxa. Koncept pošiljatelja ne postoji u Reduxu. Fluxova skladišta su zapravo obični JavaScript objekti, koje funkcije definirane u pošiljatelju mijenjaju [17].

4.7.2. MobX

MobX koristi objektno orijentiranu paradigmu. Stanje aplikacije se sprema u klase, koje predstavlja skladište stanja. Poput Fluxa, moguće je imati više skladišta stanja. Akcije su napravljene kao metode klase. Asinkrone akcije je vrlo jednostavno napraviti, u jednoj metodi se pozove asinkrona funkcija, te se kao funkcija povratnog poziva pošalje druga metoda te iste klase. Ovisno o konfiguraciji podatke, moguće je mijenjati i izvan akcija, a MobX može i dalje pratiti promjene. Stanje je moguće direktno modificirati, a MobX prati promjene stanja. MobX uvodi koncepte reakcija i izračunatih vrijednosti. Oboje prate promjene podataka, te kada dođe do promjene, izvršava se reakcija ili se izračunava nova izračunata vrijednost. Pri korištenju Reacta, potrebno je deklarirati koje komponente prate promjene stanja i iz kojeg skladišta. Za razliku od Reduxa, ne preporuča se da podaci budu normalizirani, a garantira se da aplikacija neće dobiti nepotpuno izmijenjeno stanje [18].

5. Projekt

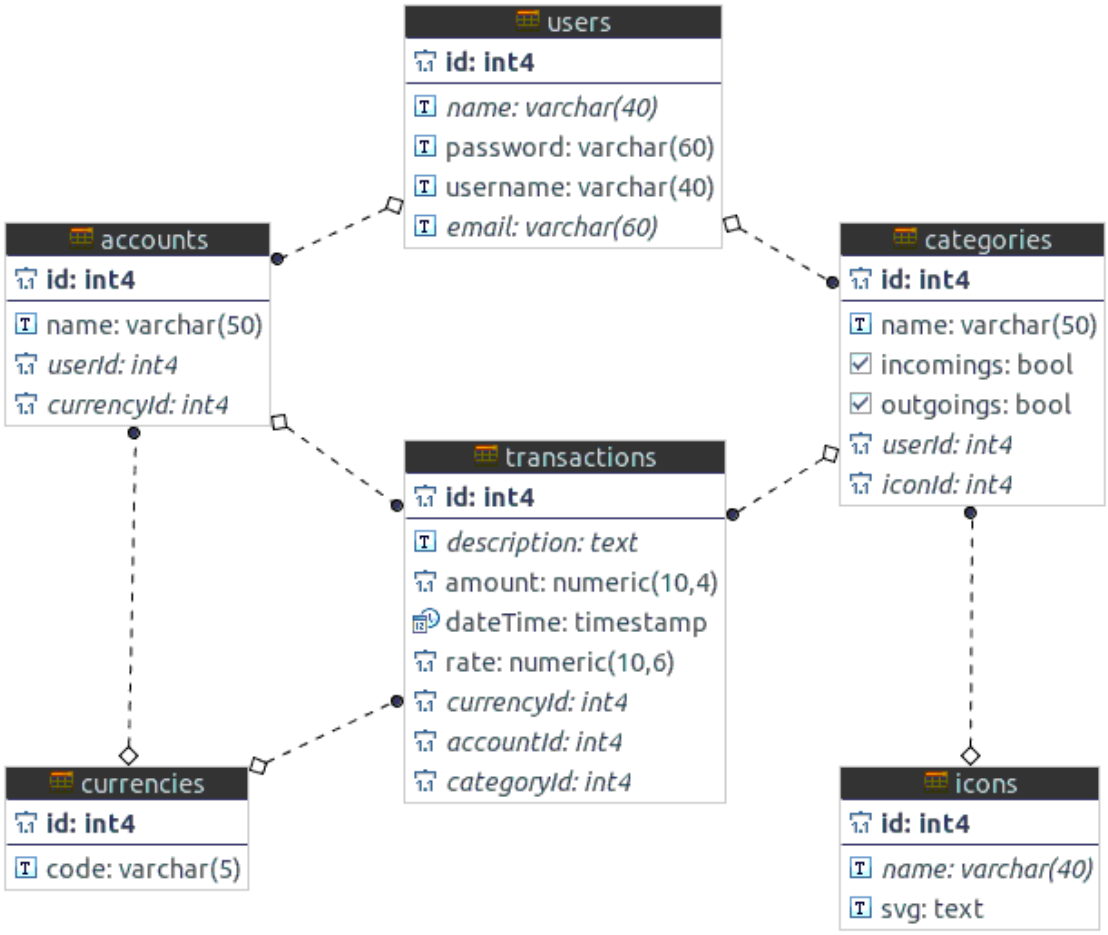
Za izradu projekta u ovom radu, odabrana je aplikacija koja će korisnicima pomoći pri praćenju njihovih financijskih transakcija. Aplikacija je zamišljena tako da korisnik u nju unosi sve svoje financijske transakcije, najbolje pri njihovom nastanku. To je omogućeno time da je aplikacija dostupna na webu te ju je moguće koristiti putem pametnih telefona, a zbog toga je neophodno da se aplikacija prilagođava uređaju na kojem se koristi. Prednost weba u ovoj aplikaciji je u tome da se svi podaci spremaju na poslužitelj te su dostupni na svim uređajima. Korisnik u aplikaciji može pratiti stanje svojih osobnih financija, prethodne transakcije te razne statističke podatke koji mu mogu pomoći pri donošenju financijskih odluka. Aplikacija je nazvana V-Alt.

Kako je ovo web aplikacija, korisnici se moraju prijaviti da bi je mogli koristiti. Neprijavljen korisnik ima mogućnost prijave, a ako nema svoj korisnički račun, može se registrirati. Prilikom registracije unosi svoje korisničko ime i lozinku, jer su to podaci potrebni za prijavu. Opcionalno može unijeti još svoje ime i email adresu. Nakon prijave, korisnik se preusmjerava na dio aplikacije u kojem se nalazi popis transakcija. Transakcije su poredane po vremenu nastanka. Moguće je dodavati nove te brisati i mijenjati postojeće. Svaka transakcija sadrži svoj iznos, valutu, vrijeme nastanka te račun na kojem je napravljena. Dodatno transakcije je moguće smjestiti u kategorije i dodati opis te transakcije. Početni računi i kategorije se stvaraju prilikom registracije, a korisnik ih kasnije može dodavati ili micati. Račun nije moguće izbrisati ako već sadrži transakcije, te mu nije moguće promijeniti zadanu valutu. Uz transakcije se sprema trenutni tečaj, za razmjenu valute transakcije u zadanu valutu računa. Svaka kategorija ima svoje ime i ikonu koju bira korisnik. Moguće je još ograničiti hoće li se u neku kategoriju spremati samo prihodi, samo rashodi ili oboje. Korisnik će se upozoriti, ako unosi pogrešan tip transakcije u kategoriju. Za razliku od kategorija, svaka transakcija mora biti izvršena na nekom računu. Računi se razlikuju po nazivu te zadanoj valuti, prilikom izrade računa, sprema se početno stanje, dodavanjem nove transakcije.

Aplikacija omogućuje korisniku pregled transakcija u nekom zadanom vremenu. Uz vrijeme, moguće je odabrati koje će se kategorije prikazivati te koji računi. Korisniku se prikazuje trenutno stanje računa, nakon izvršenja svih transakcija. Mogući su statistički prikazi ukupnih transakcija po nekoj kategoriji ili na nekom računu u zadanom vremenu. Radi bolje preglednosti podataka, aplikacija podržava izradu nekoliko vrsta grafova. Korištenjem tortnih grafova, prikazuje se omjer transakcija na nekom računu ili u nekim kategorijama. Stupčasti grafikoni prikazuju stanje na nekom računu kroz vrijeme.

5.1. Modeli podataka i baza podataka

Aplikacija koristi relativno jednostavan model podataka. Korisnici mogu mijenjati četiri modela podataka, a to su korisnički profil, transakcije, bankovni računi te kategorije. Ti modeli su direktno pretvoreni u tablice baze podataka. U bazi se nalaze i dvije dodatne tablice, u koje se spremaju ikone i valute. Između svih modela se nalazi veza jedan na više. Kako nema veze više na više, dodatne tablice za povezivanje podataka nisu potrebne. Na Slici 2 je prikazan ERA model baze podataka generiran pomoću programa DBeaver [19].



Slika 2: ERA model baze aplikacije V-Alt (vlastita izrada)

Baza podataka je izrađena u sustavu za upravljanje relacijskim bazama podataka PostgreSQL. Modeli podataka su automatski pretvoreni u tablice u bazi podataka, što je postignuto korištenjem alata TypeORM. TypeORM je alat za povezivanje objekata na relacije (eng. *Object-relational mapping,* kraće ORM). Nakon što su napisani modeli, TypeORM iz njih

stvara bazu podataka, koja će se koristiti u aplikaciji. Zatim, isti alat omogućuje lakši rad na poslužiteljskom dijelu aplikacije, pri izradi upita i manipuliranju podacima. Slijedi programski kod modela za račun, napisan u programskom jeziku TypeScript [20].

```
@Entity({ name: 'accounts' })
export class Account {
    @PrimaryGeneratedColumn()
    id: number;

    @Column({
        length: 50
    })
    name: string;

    @ManyToOne(
        type => User,
        user => user.accounts,
        { nullable: false }
    )
    @JoinColumn({ name: 'userId' })
    user: User;

    @Column({ nullable: false })
    userId: number;

    @ManyToOne(type => Currency, { nullable: false })
    @JoinColumn({ name: 'currencyId' })
    currency: Currency;

    @Column({ nullable: false })
    currencyId: number;

    @OneToMany(type => Transaction, transactions => transactions.account)
    transactions: Transaction[];
}
```

Na modelima su dostupna sva polja iz baze podataka. Dostupne su i veze s drugim modelima, pa tako ovdje prikazani model računa sadrži polje transakcija, te korisnika i valutu. U TypeORM-u modeli se prikazuju korištenjem TypeScript klasa i anotacija. Pomoću anotacija klasa i njenim atributima, specificiraju se dodatna svojstva.

Ovdje prikazani model računa, opisuje račun na kojem korisnik izvršava transakcije. U stvarno svijetu taj model predstavlja bankovne račune, novčanik ili slične koncepte. Račun ima svoje ime, valutu te korisnika, a sprema se u tablici `accounts`. Sljedeći model je profil korisnika. U profilu je sadržano ime korisnika, korisničko ime, email i kriptirana lozinka. Model profila se sprema u tablicu `users`, te još sadrži listu računa i listu kategorija. Kategorije spremljene u tablici `categories`, predstavljaju kategorije u koje korisnik može podijeliti svoje transakcije. Primjer kategorija mogu biti plaća ili stanarina. Kategorija sadrži naziv, ikonu, korisnika, te logičke vrijednosti koje opisuju spremaju li se u tu kategoriju prihodi, rashodi ili oboje. Kao i račun, kategorija sadrže listu transakcija. Transakcije su središnji dio ove aplikacije. Spremaju se u tablicu `transactions`. Svaka transakcija sadrži vrijeme kada se dogodila, iznos i valutu. Kako se valuta transakcije može razlikovati od valute na računu, sprema se i trenutni tečaj. Množenjem iznosa s trenutnim tečajem, dobiva se transakcija u valuti računa. Time je omogućena usporedba transakcija u zadanoj valuti. Transakcija može sadržavati opis i kategoriju, a mora sadržavati korisnika i račun. Baza sadrži dvije tablice, koje korisnici ne mogu mijenjati unutar aplikacije. Jedna od njih je tablica `currencies`, koja sadrži oznake valuta. Druga je tablica s ikonama nazvana `icons`. Ikone su predviđene za spremanje u SVG formatu, pa sadrže polje za unos SVG grafike i polje za naziv ikone.

5.2. Poslužiteljska strana aplikacije

Poslužiteljski dio aplikacije V-Alt je napravljen na platformi Node.js, te je napisan korištenjem programskog jezika TypeScript. Typescript za razliku od JavaScripta, nije direktno podržan na toj platformi. Međutim TypeScript se prevodi u JavaScript prije izvršavanja, što se postiže korištenjem TypeScript prevodioca. Prevođenje je moguće napraviti i u radnoj memoriji, bez da se nove datoteke stvore na tvrdom disku. Popularni NPM paket, koji to omogućuje, je TS Node. TS Node se pokreće naredbom `ts-node` u terminalu, a nakon pokretanja se ponaša isto kao i Node.js, osim što može izvršavati i TypeScript datoteke.

TypeScript je programski jezik koji proširuje JavaScript, pa je svaki JavaScript kod ujedno i validan TypeScript. Nastao je u tvrtki Microsoft i otvorenog je koda. Glavni razlog korištenja TypeScripta, a ujedno po čemu je i jezik dobio ime, su tipovi. Za razliku od JavaScript varijabli, u TypeScriptu varijablama je moguće definirati njihov tip. Time se sprječavaju moguće greške, nastale pri pisanju programa. Kako je tip varijable poznat prilikom pisanja koda, editor teksta može bolje ponuditi programeru sljedeću naredbu. Primjer toga su varijable i metode klase, koje je moguće predložiti, ako je poznat tip objekta. Uz to, editor teksta može upozoriti programera, ako utvrdi da je došlo do pogreške, poput pozivanja metode, koja ne postoji na nekom objektu ili pozivanju funkcije s krivim parametrima. Kod funkcija i metoda, može se

označiti tip podataka parametra i vrijednosti koje vraćaju, čime kod postaje čitljiviji, a te informacije editor teksta može prikazati programeru kod korištenja tih funkciji i metoda. TypeScript dodaje nove mogućnosti ECMAScripta, prije nego su dostupne u JavaScriptu, te ih se prevodi u neku stariju verziju JavaScripta, koju je moguće izvršiti u web pregledniku ili na platformi Node.js. Jedna od mogućnosti koja nije podržana u JavaScriptu, a koristi se u ovom radu, su anotacije koje su korištene kod modela podataka [8].

Aplikacija koristi okvir Express. Okvir Express omogućuje dodjeljivanje određene funkcije putanji, koju će web preglednik pozvati korištenjem neke HTTP metode. U ovom projektu to su isključivo AJAX pozivi, koji mogu imati podatke u tijelu zahtjeva u JSON formatu, te primaju odgovor također u tom formatu. Express omogućuje da zahtjevi prolazi kroz više funkcija, prije nego se odgovor proslijedi korisniku. Primjer toga u ovom projektu je funkcija koja identificira korisnika korištenjem JSON web tokena, a dostupna je u NPM paketu Express JWT. Pozivanje te funkcije je postavljeno na svim putanjama na kojima je potrebna identifikacija korisnika. Na putanjama poput prijave i registracije, koju mora moći dohvatiti neidentificirani korisnik, ta funkcija nije postavljena. Express svim funkcijama prosljeđuje objekt HTTP zahtjeva i HTTP odgovora, koji su zapravo prošireni Node.js objekti istog tipa. Bitno proširenje su funkcije, koje omogućuju lakše slanje odgovora na zahtjev korisniku. U primjeru je prikazana funkcija iz ovog projekta, koja vraća korisniku sve ikone za kategorije.

```
router.get('/', async (req: Request, res: Response) => {
    try {
        const iconContext = getConnection().getRepository(Icon);
        const icons = await iconContext.find();
        res.send({ icons: classToPlain(icons) });
    } catch (err) {
        res.send({ error: 'DB error.' });
    }
});
```

Korištenje TypeScripta je vidljivo samo u prvoj liniji, gdje su specificirani tipovi koje funkcija koristi, a ostatak koda se ne razlikuje od čistog JavaScripta. Funkcija se poziva pri dolasku zahtjeva s HTTP metodom GET. Metoda `send` je jedna od metoda kojom Express proširuje Node.js objekt za zahtjev, a služi za slanje odgovora korisniku. Pri izradi odgovora, metoda postavlja potrebna zaglavlja te pretvara objekt u JSON, kako bi se mogao poslati klijentu [8].

Funkcije koje se pozivaju kod stvaranja odgovora na zahtjev korisnika, nalaze se u mapi `src/controller`. U datoteke su podijeljeni, tako da svaka tablica iz baze podataka ima svoju datoteku. Time je postignuta veća preglednost, te su funkcije podijeljene ovisno o putanji na kojoj se pozivaju. Dijeljenje funkcija na više datoteka, postignuto je korištenjem

instance klase `Router`. Klasa `Router` je također dostupan u Expressu, a omogućuje podjelu aplikacije na manje dijelove, koji se zatim koriste u glavnoj aplikaciji. Na svaku instancu klase Router, moguće je postaviti putanju koja se odnosi na sve funkcije, kao i funkcije koje se izvršavaju prije svakom zahtjevu. Slijedi isječak koda iz datoteke `UserController.ts`, koji služi za dohvaćanje svih računa koje korisnik posjeduje.

```
import { Router, Request, Response } from 'express';

const router = Router();
router.use(authorize);

router.get('/', async (req: Request, res: Response) => {
	const userId = req.user.id;
	try {
		const accountContext = getConnection().getRepository(Account);
		const accounts = await accountContext
			.find({ where: { userId } });

		res.send({ accounts: classToPlain(accounts) });
	} catch (err) {
		res.send({ error: 'DB error.' });
	}
});
export default router;
```

U toj datoteci je definirano još nekoliko funkcija, koje nisu ovdje prikazane. Prije poziva tih funkcija, poziva se funkcija `authorize`. Ta funkcija provjerava valjanost tokena, te ako je valjan, objekt zahtjeva `req` postavlja objekt korisnika `user`, koji se kasnije koristi u upitu baze podataka. Sve instance klase `Router` se izvoze iz datoteka, te se dodjeljuju putanjama u glavnoj aplikaciji. To je napravljeno u datoteci `indeks.ts` u mapi `controller`, korištenjem funkcije `applayRoutes`. Slijedi kod te funkcije.

```
export function applyRoutes(app) {
	app.use('/user', UserController);
	app.use('/icon', IconController);
	app.use('/currency', CurrencyController);
	app.use('/account', AccountController);
	app.use('/category', CategoryController);
	app.use('/transaction', TransactionController);
}
```

Express aplikacija se šalje kao parametar, a na određene putanje se dodaju instance klase `Router`, uvezeni iz datoteka u mapi `controller`. Početna datoteka, iz koje se aplikacija pokreće, je `index.ts` u mapi `src`. U njoj se kreira aplikacija, u koju se zatim dodaju funkcije. U Tablici 1 su opisane sve putanje, koje su definirane u aplikaciji V-Alt. Sve putanje koje dohvaćaju podatke nekog korisnika su zaštićene, a i osigurano je da prijavljeni korisnik ne može mijenjati podatke nekog drugog prijavljenog korisnika.

Tablica 1: Sve implementirane funkcije aplikacije (vlastita izrada)

Putanja	HTTP metoda	Opis funkcije postavljene na putanji
/user	GET	Ako korisnik proslijedi ispravan token, kao odgovor dobije objekt s podacima o korisniku (email, ime, korisničko ime).
/user	POST	Funkcija za stvaranje novog korisnika. Ako je stvoren, vraća objekt korisnika te token.
/user/usernametaken	GET	Provjerava je li korisničko ime poslano preko GET parametra zauzeto.
/user/emailtaken	GET	Provjerava je li email zauzet, radi poput prethodne funkcije.
/user/login	POST	Ako su proslijeđeni podaci točni, funkcija vraća objekt korisnika i token, služi za prijavu u aplikaciju.
/account	GET	Funkcija dohvaća sve račune korisnika.
/account	POST	Funkcija za stvaranje novog računa.
/account/{id}	PUT	Funkcija za izmjenu postojećeg računa, primarni ključ računa se šalje u putanji ili u tijelu zahtjeva.
/account/{id}	DELETE	Funkcija za brisanje računa.
/category	GET	Dohvaća sve kategorije prijavljenog korisnika.
/category	POST	Stvara novu kategoriju.
/category/{id}	PUT	Funkcija za izmjenu postojeće kategorije.
/category/{id}	DELETE	Funkcija briše kategoriju.
/transaction	GET	Kao GET parametri su šalju početni i završni datum. Funkcija vraća sve transakcije napravljene između ta dva datuma.
/transaction	POST	Funkcija za stvaranje nove transakcije.
/transaction/{id}	PUT	Izmjena transakcije.

/transaction/{id}	DELETE	Brisanje transakcije.
/icon	GET	Dohvaća sve ikone koje korisnik može koristiti.
/currency	GET	Dohvaća sve valute.

Osim mape `controller`, u mapi s kodom aplikacije `src`, postoje još i mapa u kojoj su smješteni modeli nazvanu `model`, mapa s pomoćnim funkcijama `utils`, te mapa s funkcijama koje se pozivaju, prije funkcija na nekim putanjama nazvana `middleware`. Sva konfiguracija aplikacije, koju je potrebno mijenjati, je postavljena u datoteku `.env`. U toj datoteci su podaci, potrebni za povezivanje na bazu podataka, port na kojem se aplikacija pokreće, kao i tajni niz znakova koji se koristi pri izradi i validaciji tokena.

Rad s bazom podataka, ostvaren je pomoću TypeORM-a. Definirane su klase za svaku tablicu baze podataka, koje se kasnije koriste za rad s bazom u aplikaciji. U tim klasama su dodane i dodatne anotacije iz paketa Class-transformer, koje označavaju kako će se podaci poslati kao odgovor klijentu. Podaci, poput enkriptirane lozinke i vanjskog ključa, prema tablici korisnik se ne šalju. Svi vanjski ključevi se šalju kao niz znakova. Korištenjem funkcije `classToPlain`, objekti klase se pretvaraju u objekte primjerene za slanje korisniku. Slijedi primjer korištenje anotacija za oblikovanje u klasi `User`.

```
import { Exclude, Transform } from 'class-transformer';

@Entity({ name: 'users' })
export class User {
	@PrimaryGeneratedColumn()
	@Transform(value => value.toString(), { toPlainOnly: true })
	id: number;

	@Column({
		length: 60
	})
	@Exclude()
	password: string;


}
```

Rad s bazom podatak vrlo je jednostavan. Već je u prethodnim primjerima bilo prikazano korištenje TypeORM-a, za dohvaćanje podataka iz baze. Slijedi primjer stvaranja novog podatka koji će se spremiti u bazu, pri stvaranju računa korisnika.

```
router.post('/', async (req: Request, res: Response) => {
    const userId = req.user.id;
    const currencyId = req.body.currency;
    const name = req.body.name;
    try {
        const accountContext = getConnection().getRepository(Account);

        const newAccount = new Account();
        newAccount.name = name;
        newAccount.currencyId = currencyId;
        newAccount.userId = userId;

        const account = await accountContext.save(newAccount);

        res.send({ accounts: [classToPlain(account)] });
    } catch (err) {
        res.send({ error: 'Account creation unsucesfull.' });
    }
});
```

Kod izmjene podataka, umjesto da se izradi nova instanca klase modela, model se dohvaća iz baze, a zatim mu se mijenjaju svojstva, nakon čega se sprema u bazu isto kao i kod izrade novog podatka. Prije spremanja u bazu, potrebno je usporediti identifikator korisnika s identifikatorom korisnika na modelu, kako zlonamjeran korisnik ne bi mogao mijenjati podatke drugih korisnika. Slijedi dio dijela funkcije za izmjenu imena računa, gdje se provjerava identifikator korisnika.

```
const account = await accountContext.findOne(accountId);

if (account.userId === userId) {
      account.name = name;
      const savedAccount = await accountContext.save(account);
      res.send({ account: [classToPlain(savedAccount)] });
} else {
      res.sendStatus(401);
}
```

Kod složenijih slučajeva, ovakav pristup rada s podacima nije dovoljan. U TypeORM-u je moguće izraditi složeniji upit korištenjem izrađivača upita, nazvanog `QueryBuilder`. Slijedi primjer izgradnje složenog upita. Upit dohvaća transakcije između dva datuma, koje se nalaze na nekom od računa zadanog korisnika [20].

```
const transactions = await getConnection()
            .getRepository(Transaction)
            .createQueryBuilder('transaction')
            .innerJoin('transaction.account', 'account')
            .where('transaction.dateTime >= :startDate', { startDate })
            .andWhere('transaction.dateTime <= :endDate', { endDate })
            .andWhere('account.userId = :userId', { userId })
            .getMany();
```

5.3. Korisnička strana aplikacije

Korisnička strana projekta, koja koristi prethodno opisan poslužitelj, započeta je korištenjem programa Create React App, kako bi se čim prije krenulo na izradu aplikacije, bez konfiguriranja webpacka ili sličnih programa. U projekt su dodane biblioteke Redux i React Router te Axios. Axios je biblioteka koja olakšava izradu AJAX poziva prema poslužitelju. Jedno od korištenih mogućnosti Axiosa je postavljanje zaglavlja, koje će se koristiti u svi AJAX pozivima. To je iskorišteno za spremanje identifikacijskog tokena, koji se zatim šalje kod svakog zahtjeva prema poslužitelju. Token se sprema u lokalno skladište u pregledniku tako da korisnik ostaje prijavljen, i nakon zatvaranja kartice ili prozora. Za potpunu odjavu potrebno je kliknuti na tipku „Odjavi se," te se tad briše token iz lokalnog skladišta preglednika. Izvorni kod aplikacije nalazi se u mapi `src`, u kojoj su smještene ostale datoteke i mape projekta. Aplikacija počinje s izvođenjem u datoteci indeks.js. U istoj se mapi nalazi i datoteka u kojoj se inicijalizira skladište stanja `store.js`, kao i datoteka s funkcionalnostima za identifikaciju korisnika `auth.js`. Ostali dijelovi aplikacije su razvrstani o mape, ovisno o funkciji koju ta datoteka obavlja. Datoteke s komponentama su podijeljene u dvije mape, `containers` i `components`. U mapi `containers` nalaze se kontejnerske komponente, koje su izrađene pomoću Reduxa. Svako kontejnerska ima i svoju istoimenu prezentacijsku komponentu, u mapi `components`. U toj mapi se nalaze sve komponente, koje nisu izrađene pomoću Reduxove funkcije `connect`. Kako se radi o prezentacijskim datotekama, tu su smještene i CSS datoteke. Komponente su grupirane prema funkcionalnosti u dodatne podmape. Kako ne bi došlo do kolizije stilova, sve klase koje se koriste u komponentama započinju s imenom komponente. U mapi `constants` nalaze se konstante koje aplikacija koristi, poput tipova

akcija. Kreatori akcija su smiješteni u mapi `actions`. Reduktori koji koriste navedene akcije, nalaze se u mapi `reducers` [8].

Korijenska komponenta aplikacije, i komponenta od koje počinje iscrtavanje aplikacije na ekran korisnika, nazvana je `Root`. Jedina zadaća te komponente je da omogući ostatku aplikacije korištenje usmjeravanja u web pregledniku korištenjem biblioteke React Router, te omogući dostupnost Reduxovog skladišta u ostatku aplikacije. Slijedi izvorni kod komponente `Root`.

```
const Root = () => (
      <BrowserRouter>
            <Provider store={store}>
                  <App />
            </Provider>
      </BrowserRouter>
);
```

Vidljivo je da je sljedeća komponenta koja se iscrtava komponenta `App`. To je kontejnerska komponenta, koja je napravljena korištenjem Reduxa. U njoj su smještene sve ostale komponente aplikacije, a sama obavlja funkcionalnost vezanu uz prijavu i registraciju korisnika. Slijedi izvorni kod kontejnerske komponente `App`.

```
import { connect } from 'react-redux';
import { logIn, logOut, signUp } from '../actions/user';
import App from '../components/App';

const mapStateToProps = state => ({
    user: state.user
});

const mapDispatchToProps = dispatch => ({
    logIn: (username, password) => dispatch(logIn(username, password)),
    logOut: () => dispatch(logOut()),
    signUp: (username, password, email, name) => dispatch(signUp(username,
password, email, name))
});

const AppContainer = connect(
    mapStateToProps,
    mapDispatchToProps,
    null,
```

```
    { pure: false }
)(App);

export default AppContainer;
```

Kako ova komponenta ne sadrži čiste funkcijske komponente, potrebno je postaviti vrijednost `pure` na laž, jer u suprotnom ostatak aplikacije neće raditi. Komponenta kojoj kontejnerska komponenta `App` šalje podatke o korisniku, kao i funkcije s kojima može pozivati akcije, ima isto ime. Slijedi kod te komponente.

```
class App extends Component {
      componentDidMount() {
            store.dispatch(fetchUser());
      }

      render() {
            const {
                  user,
                  logIn,
                  signUp,
                  logOut
            } = this.props;

      return (
            <Switch>
                  <Route
                        path="/login"
                        render={() => (
                        user
                        ? (<Redirect to="/" />)
                        : (<LogIn logIn={logIn} />))}
                  />
                  <Route
                        path="/signup"
                        render={() => (
                        user
                        ? (<Redirect to="/" />)
                        : (<SignUp signUp={signUp} />))}
                />
                  <Route
                        path="/"
```

```
                        render={() => (
                              user
                              ? (<Layout user={user} logOut={logOut} />)
                              : (<Redirect to="/login" />))}
                  />
            </Switch>
      );
      }
}
```

U ovoj komponenti napravljeno je početno usmjeravanje korisnika, ovisno o tome je li prijavljen. Ako nije, preusmjerava se na komponentu koja služi za prijavu, a ako je prijavljen onda može koristiti ostatak aplikacije. Ova komponenta pri svojem stavljanju u DOM, provjerava je li korisnik prijavljen korištenjem akcije `fetchUser`. Ako je korisnik prijavljen, ova akcija sprema podatke o korisniku u skladište. Slijedi izvorni kod ove akcije.

```
export const setUser = (user, token) => {
      if (token) {
            setToken(token);
      }
      return {
            user,
            type: SET_USER
      };
};

export const fetchUser = () => (dispatch) => {
      axios.get('/user')
            .then(({ data }) => {
                  dispatch(setUser(data.user));
                  dispatch(fetchIcons());
                  dispatch(fetchCurrencies());
                  dispatch(fetchAccounts());
                  dispatch(fetchCategories());
                  dispatch(fetchTransactions());
      });
};
```

Akcija nakon što dohvati korisnika, dohvaća ostale podatke potrebne za rad aplikacije, kao što su ikone, valute i transakcije. Kod reduktora, koji koristi ove akcije, relativno je jednostavan. Jedina funkcionalnost koju obavlja je postavljanje korisnika dohvaćenog s poslužitelja i brisanje korisnika iz skladišta. Slijedi kod tog reduktora.

```
const user = (state = null, action) => {
	switch (action.type) {
		case SET_USER:
			return action.user;
		case LOG_OUT:
			return null;
		default:
			return state;
	}
};
```

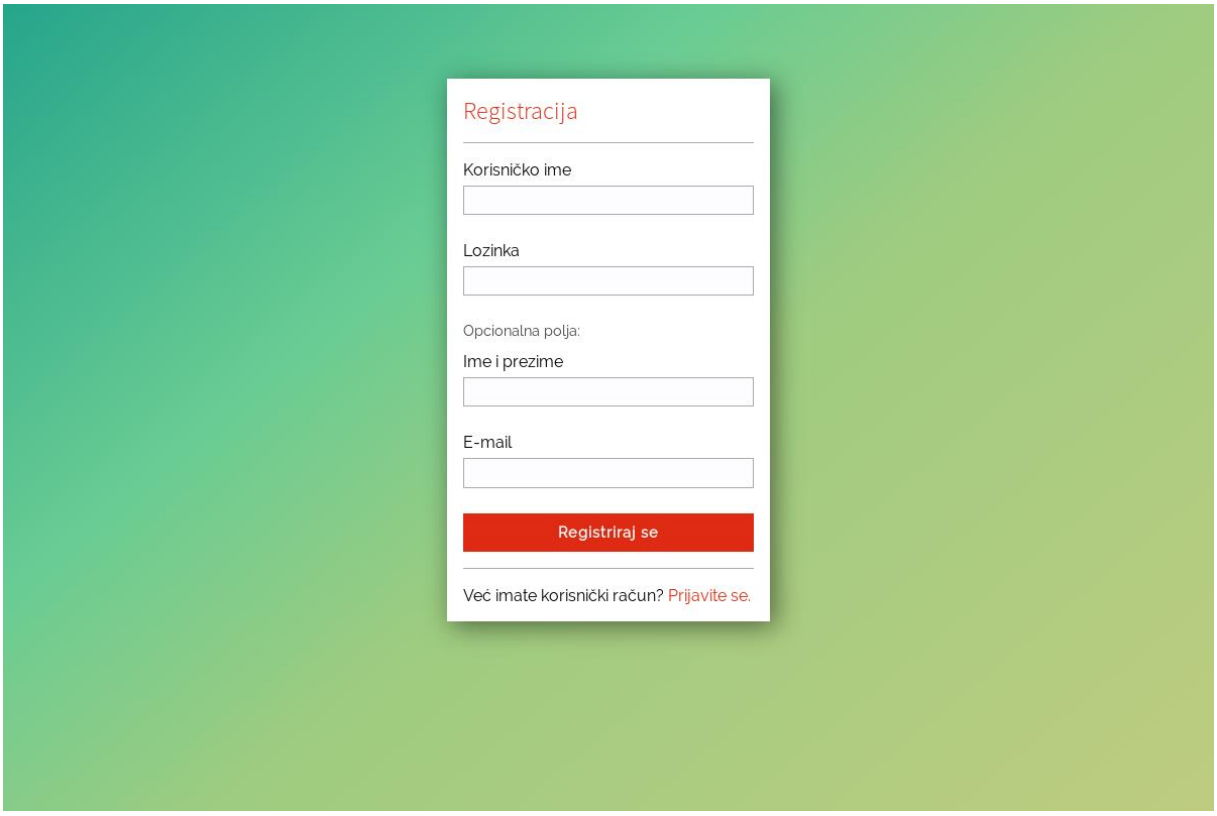
Za razliku od korisnika, svi ostali podaci iz baze podataka se dohvaćaju na isti način. Poslužitelj odgovara klijentu s poljem objekata, koji su nastali iz redova tablice u bazi podataka. Zatim se spremaju skladište kao parovi ključa i vrijednosti, gdje je ključ identifikator objekta, a ujedno i primarni ključ u bazi podataka. Kako se radi o sličnoj funkciji, bilo je moguće napisati funkciju koja stvara reduktore za određeni model. Funkcija prima vrijednost ključa pod kojom se objekti šalju na poslužitelju, a vraća reduktor. Slijedi kod navedene funkcije.

```
import { ADD_ITEMS, REMOVE_ITEM } from '../constants/actionTypes';

const curreciesReducer = itemName => (state = {}, action) => {
	if (action.type === ADD_ITEMS && action.items[itemName]) {
		const items = action.items[itemName];
		return Object.assign(
			{},
			state,
			...items.map(item => ({ [item.id]: item }))
		);
	}
	if (action.type === REMOVE_ITEM && action.item[itemName]) {
		const itemId = action.item[itemName];
		const { [itemId]: _, ...newState } = state;
		return newState;
	}
	return state;
};
```

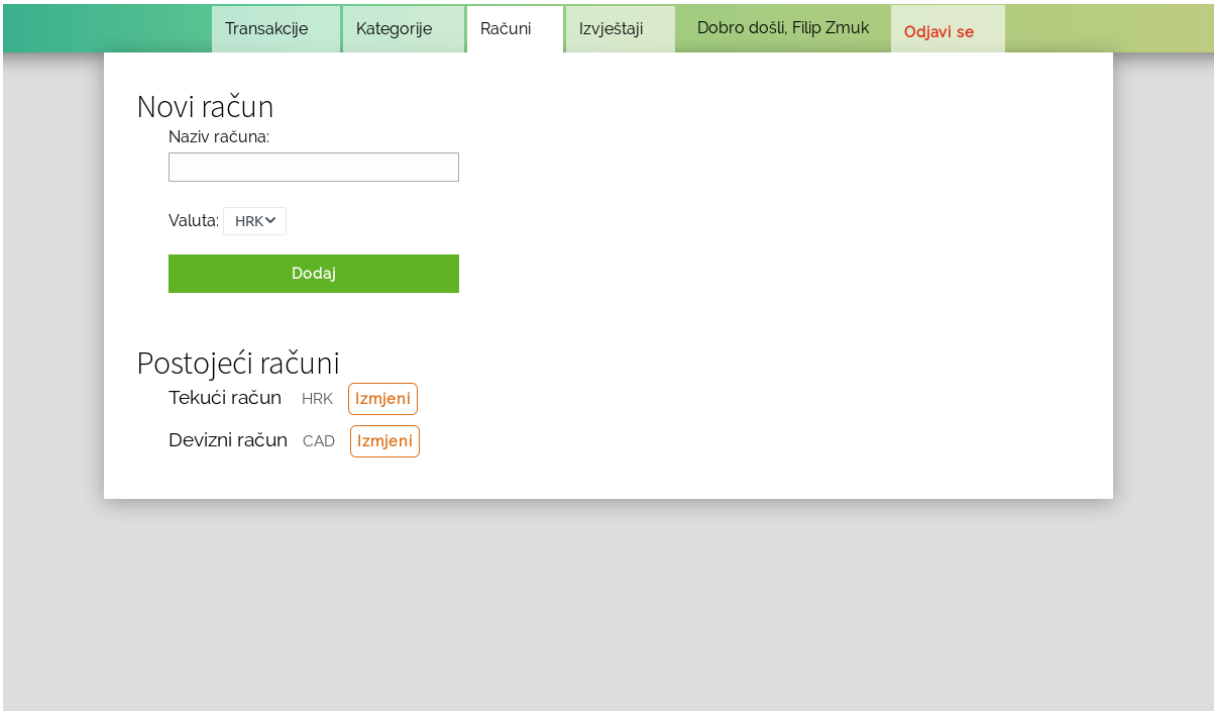
Ova funkcija se poziva nekoliko puta, kako bi se napravili reduktori za korisnikove račune, kategorije, transakcije, ikone te valute. Reduktor podržava dva tipa akcije, akciju dodavanja objekta te akciju brisanja objekta. Kod dodavanja objekta, stvara se objekt od polja vraćenog s poslužitelja. Funkcija proširuje dosadašnje stanje pa se može koristiti i za dodavanje novog

objekta. Zbog toga ju je moguće koristiti i kod izmjene postojećeg objekta. Akcija brisanja stvara novo stanje, u kojem se ne nalazi određeni objekt.



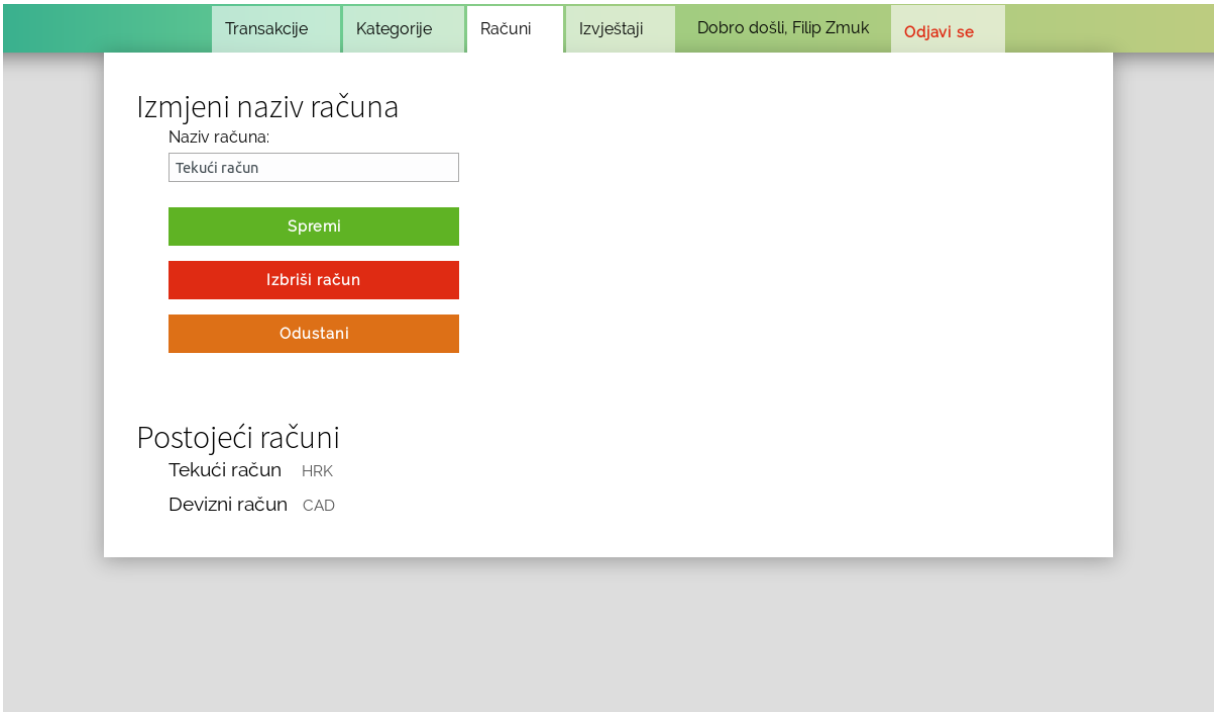
Slika 3: Obrazac za registraciju korisnika (vlastita izrada)

Na Slici 3 je prikazan obrazac za registraciju korisnika. Izrađen je u komponenti `SignUp`. Komponenta u svom stanju ima spremljene podatke, koji se nalaze na obrascu pa su polja kontrolirane komponente. To omogućuje da se prilikom pisanja korisničkih podataka, podaci provjeravaju, a korisniku daje povratna informacija. Jedna od informacija koje se šalju korisniku, su podaci o zauzetosti korisničkog imena i email adrese. To je ostvareno slanjem zahtjeva na poslužitelj, prema putanjama koje su opisane u Tablici 1. Ako se utvrdi da je neki podatak neispravan, korisniku se ispisuje poruka iznad polja za unos. Kod klika na tipku registrira, šalje se akcija u skladište, te ako je korisnik uspješno registriran, postaje i prijavljen. Slično je napravljena i komponenta `LogIn`, koja služi za prijavu korisnika u aplikaciju. Međutim, ta komponenta sprema status prijave u skladište, pa je iz skladišta vidljivo je li korisnik neprijavljen, prijavljen ili se je pokušao prijaviti s pogrešnim podacima. Kao i kod prethodne komponente, korisniku se ispisuje poruka, ako se pokušao prijaviti s pogrešnim podacima.



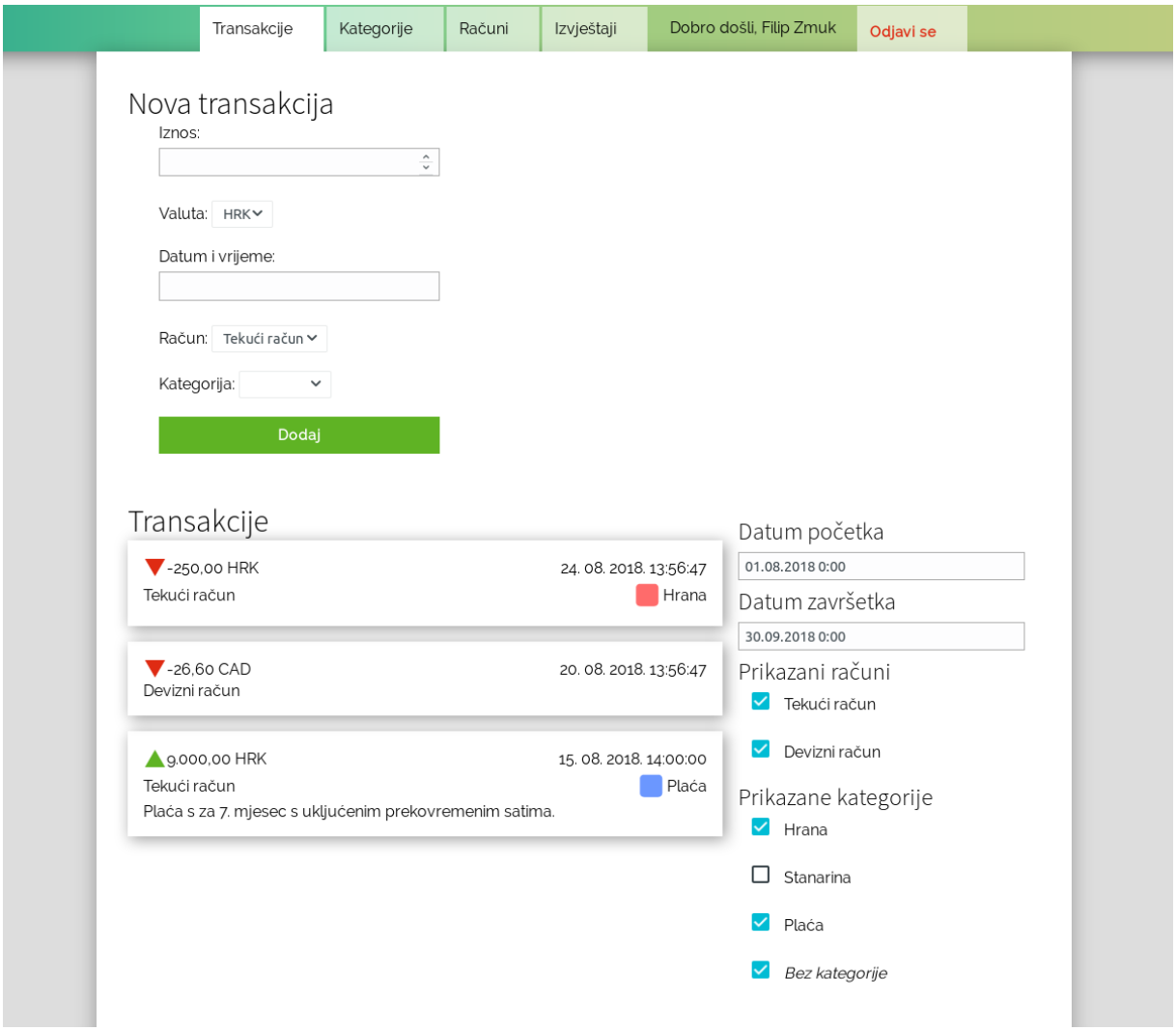
Slika 4: Prikaz računa korisnika (vlastita izrada)

Na Slici 4 je prikazana komponenta `Account`. Komponenta se sastoji od nekoliko manjih komponenti. Komponenta `AccountList` se sastoji od polja komponenti `AccountListItem`. Ove komponente prikazuju postojeće račune korisnika. Kada korisnik klikne na tipku izmijeni pokraj nekog računa, račun se sprema u skladište kao označeni račun. Tada ga je moguće urediti te spremiti ili izbrisati. Komponenta `NewAccount` nalazi se na gornjem dijelu komponente `Account` i služi za izradu novog računa. Račun se dodaje u skladište, nakon što se dobije odgovor od poslužitelja da je račun uspješno spremljen, a čim se pojavi u skladištu, pojavljuje se i na listi postojećih računa. Ovo može poslužiti kao primjer deklarativnog programiranja i pokazuje kako React olakšava izradu web aplikacija, jer nije bilo potrebno implementirati nikakve dodatne naredbe za dodavanja računa u listu. Ako je neki račun označen za izmjene, tada se ne pokazuje komponenta `NewAccount`. Umjesto nje, pokazuje se `EditAccoun`, kao što je prikazano na Slici 5. Tada se ne prikazuju tipke za izmjenu računa, sve dok korisnik odabrani račun ne izbriše ili spremi, a može i odustati od toga. Kako korisnik ne može promijeniti valutu računa, lista valuta se ne prikazuje. Nakon takve korisnikove akcije, više nije označen niti jedan račun pa je opet vidljiva komponenta za dodavanje računa. Komponenta za unos, prikaz, izmjenu i brisanje kategorija, radi gotovo identično kao i komponenta koja obavlja iste funkcije s računima.



Slika 5: Obrazac za izmjenu računa (vlastita izrada)

Transakcije su vidljive odmah nakon prijave u aplikaciju. Na Slici 6 je prikazana komponenta koja prikazuje transakcije. Komponenta ima mogućnost filtriranja transakcija po kategoriji, računu i vremenu kada su se dogodile. Prikaz transakcija implementiran je tako da se samo transakcije koje se trebaju prikazati dohvaćaju s poslužitelja. Posljednji dio aplikacije su izvještaji. Na toj komponenti je prikazan statistički podaci u nekom razdoblju koje korisnik određuje, a podaci se prikazuju i pomoću grafova.



Slika 6: Komponenta za prikazi i unos transakcija (vlastita izrada)

6. Zaključak

U ovom radu istražena je biblioteka React. Korištenje Reacta u izradi aplikacija odrađeno je izradom projektnog dijela ovog rada, za što se React pokazao kao dobra biblioteka. Kako potreba za izradom novih web aplikacijama i dalje postoji, postoji i potreba za bibliotekama koje olakšavaju njihov razvoj. React pojednostavljuje izradu aplikacije podijelom na manje i lakše održive dijelove. Također smanjuje količinu koda potrebnu za pisanje aplikacije korištenjem deklarativnog načina pisanja programa. React je i dalje u aktivnom razvoju, što se potvrđuje i izlaskom nekoliko manjih izdanja u vremenu pisanja i istraživanja za ovaj rad. Također postoji velik skup datoteka razvijenih za korištenje s Reactom.

Redux je mala biblioteka koja služi za skladištenje stanja aplikacija. Bazira se na paradigmi funkcijskog programiranja, a kao dodatnu prednost navodi jedan izvor podataka aplikacije. Za Redux je razvijena biblioteka React Redux, koja omogućuje lakše korištenje Reduxa s Reactom. Redux je namijenjen složenijim aplikacijama sa složenim podacima. U ovom radu je izrađena jednostavna aplikacija pa se prednosti Reduxa nisu osjetile. Neko komponente poput komponente za registraciju, svoje podatke su držale unutar sebe, jer je to bio jednostavniji način njihove izrade. Ovime je potvrđeno mišljenje nekolicine navedenih autora spomenutih u ovom radu, koji smatraju kako je React dovoljan za izradu mnogih web aplikacija. Kao i React, Redux ima nekoliko konkurentskih biblioteka, ali su one manje popularne, i ukratko su opisane u ovom radu.

Međutim, također React ima snažnu i popularnu konkurenciju. Za JavaScript se razvijaju mnoge biblioteke, a nove nastaju gotovo svakodnevno. U ovom radu React je ukratko uspoređen s dvije, uz React najpopularnije biblioteke, Angular i Vue.js. Niti jedna biblioteka se nije posebno istaknula od ostalih, po svojim mogućnostima. Zbog njihove popularnosti, za sve su razvijeni brojni alati i biblioteke, pa se također u ovom području niti jedna posebno ne ističe. Može se zaključiti da će izbor biblioteke ponajviše ovisiti o znanju i preferenciji programera, a manje o tome koja je biblioteka pogodnija za određeni projekt, jer sve imaju dovoljno mogućnosti i velik skup korisnika.

Popis literature

[1] Facebook Inc. (bez dat.) *React documentation* [Na internetu]. Dostupno: https://reactjs.org/docs/ [pristupano 09.07.2018.].

[2] „Redux documentation“ (bez dat.) Redux [Na internetu]. Dostupno: https://redux.js.org/ [pristupano 09.07.2018.].

[3] J. S. Zepeda, S. V. Chapa „From Desktop Applications Towards Ajax Web Applications,“ *2007 4th International Conference on Electrical and Electronics Engineering*, Mexico City, 2007

[4] M. A. Jadhav, B. R. Sawant, A. Deshmukh, „Single Page Application using AngularJS,“ *International Journal of Computer Science and Information Technologies*, sve. 6, izd. 3, 2015, [Na internetu] Dostupno: CiteSeerX http://citeseerx.ist.psu.edu/. [pristupano 12.07.2018.].

[5] P. Gasston, Moderni web. Zagreb: Dobar plan. 2013.

[6] N. C. Zakas, Naučite ECMAScript 6. Zagreb, Dobar plan. 2017.

[7] Node.js Foundation (bez dat.) *Node.js Documentation* [Na internetu]. Dostupno: https://nodejs.org/ [pristupano 09.07.2018.].

[8] npm, Inc. (bez dat.) *npm Documentation* [Na internetu]. Dostupno: https://docs.npmjs.com/ [pristupano 09.07.2018.].

[9] M. Bertoli, React Design Patterns and Best Practices. Birmingham, UK: Packt, 2017

[10] webpack (bez dat.) *webpack documentation* [Na internetu]. Dostupno: https://webpack.js.org/ [pristupano 14.08.2018.].

[11] Babel (bez dat.) *Babel documentation* [Na internetu]. Dostupno: https://babeljs.io/ [pristupano 14.08.2018.].

[12] Facebook Inc. (bez dat.) *Create React App* [Na internetu]. Dostupno: https://github.com/facebook/create-react-app [pristupano 14.08.2018.].

[13] React Training (bez dat.) *React Router Guides* [Na internetu]. Dostupno: https://reacttraining.com/react-router/web/guides/ [pristupano 14.08.2018.].

[14] Facebook Inc. (bez dat.) *Testing React Apps* [Na internetu]. Dostupno: https://jestjs.io/docs/en/tutorial-react.html [pristupano 15.08.2018.].

[15] Google (bez dat.) *Angular Documentation* [Na internetu]. Dostupno: https://angular.io/docs [pristupano 15.08.2018.].

[16] Evan You (bez dat.) *Vue.js Guide* [Na internetu]. Dostupno: https://vuejs.org/v2/guide/ [pristupano 15.08.2018.].

[17] Facebook Inc. (bez dat.) *Flux Application Architecture* [Na internetu]. Dostupno: https://facebook.github.io/flux/ [pristupano 16.08.2018.].

[18] MobX (bez dat.) *Introduction MobX* [Na internetu]. Dostupno: https://mobx.js.org/ [pristupano 16.08.2018.].

[19] Dbeaver.com *DBeaver Community* (Verzija 5.1.6) (2018) [Desktop aplikacija]. Dostupno: https://dbeaver.io/

[20] TypeORM (bez dat.) *TypeORM - Amazing ORM for TypeScript and JavaScript* [Na internetu]. Dostupno: http://typeorm.io/ [pristupano 26.08.2018.].

Popis slika

Slika 1: Komunikacija klijenta i poslužitelja (Prema: Jadhav, Sawant i Deshmukh 2015) 3

Slika 2: ERA model baze aplikacije V-Alt (vlastita izrada) 44

Slika 3: Obrazac za registraciju korisnika (vlastita izrada) 57

Slika 4: Prikaz računa korisnika (vlastita izrada) 58

Slika 5: Obrazac za izmjenu računa (vlastita izrada) 59

Slika 6: Komponenta za prikazi i unos transakcija (vlastita izrada) 60

Popis tablica

Tablica 1: Sve implementirane funkcije aplikacije (vlastita izrada) 49